Chapter 2: Basic Syntax and Data Types

Lexical Conventions

SystemVerilog follows specific lexical rules that define how the language is structured and interpreted.

Keywords and Reserved Words

SystemVerilog has reserved keywords that cannot be used as identifiers. These include: - `module`, `endmodule`, `class`, `endclass` - `function`, `task`, `if`, `else`, `case`, `default` - `for`, `while`, `repeat`, `forever` - `logic`, `bit`, `reg`, `wire`, `int`, `real`

Case Sensitivity

SystemVerilog is case-sensitive. `Signal` and `signal` are different identifiers.

Whitespace

Spaces, tabs, and newlines are considered whitespace and are generally ignored except when they separate tokens.

Numbers and Literals

```
// Decimal numbers
123
4'b1010     // 4-bit binary
8'h2F       // 8-bit hexadecimal
12'o377     // 12-bit octal

// Real numbers
3.14
2.5e-3      // Scientific notation
```

Comments and Identifiers

Single-line Comments

```
// This is a single-line comment
logic [7:0] data; // Comment at end of line
```

Multi-line Comments

```
/*
  This is a multi-line comment
  that spans multiple lines
*/
logic clock;
```

Identifiers

Identifiers are names used for variables, modules, functions, etc.

Rules for Identifiers: - Must start with a letter (a-z, A-Z) or underscore (_) - Can contain letters, digits (0-9), underscores, and dollar signs ($) - Cannot be a reserved keyword

```
// Valid identifiers
logic data_bus;
logic Clock_Signal;
logic $display_enable;
logic counter_8bit;

// Invalid identifiers
// logic 8bit_counter;  // Cannot start with digit
// logic class;         // Reserved keyword
```

Four-State vs. Two-State Data Types

SystemVerilog distinguishes between four-state and two-state data types based on the values they can represent.

Four-state types are essential for accurate hardware modeling because they can represent the realistic conditions found in actual digital circuits. The 'X' state is particularly valuable during simulation as it indicates uninitialized or conflicting values, helping designers identify potential issues early in the design process.

Two-state types offer several advantages in specific scenarios. They consume less memory during simulation since they only need to track two possible states instead of four. They also provide faster simulation performance and are often preferred for testbench code where the additional states (X and Z) are not necessary.

Four-State Data Types

Can represent four logic values: - `0` - Logic zero - `1` - Logic one - `X` - Unknown/don't care - `Z` - High impedance/tri-state

```
logic [3:0] four_state_signal;
reg [7:0] legacy_register;
wire enable_signal;
```

Two-State Data Types

Can only represent two logic values: - `0` - Logic zero - `1` - Logic one

```
bit [3:0] two_state_signal;
byte counter;
int address;
```

When to Use Each: - Use four-state types for hardware modeling where X and Z states are meaningful - Use two-state types for testbenches and high-level modeling for better performance

Integer Types

SystemVerilog provides several integer data types with different sizes and characteristics.

bit

- Two-state data type
- Can be single bit or vector
- Default value: 0

```
bit single_bit;           // 1-bit
bit [7:0] byte_vector;    // 8-bit vector
bit [31:0] word_data;     // 32-bit vector
```

byte

- Two-state, 8-bit signed integer
- Range: -128 to +127

```
byte signed_byte = -50;
byte unsigned_byte = 200;  // Will wrap around due to signed nature
```

shortint

- Two-state, 16-bit signed integer
- Range: -32,768 to +32,767

```
shortint temperature = -25;
shortint pressure = 1013;
```

int

- Two-state, 32-bit signed integer
- Range: -2,147,483,648 to +2,147,483,647

```
int address = 32'h1000_0000;
int counter = 0;
```

longint

- Two-state, 64-bit signed integer
- Range: -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807

```
longint timestamp = 64'h123456789ABCDEF0;
longint large_number = 1000000000000;
```

Comparison Table

Type	Size	States	Signed	Range
bit	1+	2	No	0 to 2^n-1
byte	8	2	Yes	-2^7 to +2^7-1 (-128 to +127)
shortint	16	2	Yes	-2^15 to +2^15-1 (-32,768 to +32,767)
int	32	2	Yes	-2^31 to +2^31-1 (-2,147,483,648 to +2,147,483,647)
longint	64	2	Yes	-2^63 to +2^63-1 (-9,223,372,036,854,775,8 to +9,223,372,036,854,775,8

Real and String Types

Real Types

SystemVerilog supports floating-point numbers with different precisions.

```
// Double-precision real (64-bit)
real temperature = 25.5;
real voltage = 3.3e-3;

// Single-precision real (32-bit)
shortreal frequency = 100.0e6;  // 100 MHz

// Real number operations
real result = 3.14159 * 2.5;
```

String Type

Dynamic string type that can hold variable-length strings.

```
string message = "Hello, SystemVerilog!";  // String initialization
string empty_string = "";                  // Empty string initialization
string formatted;                          // Uninitialized string

// String operations
formatted = $sformatf("Value: %d", 42);  // Formatted string
message = {message, " Welcome!"};        // Concatenation
```

```
// String methods
int len = message.len();  // Get length

// Case conversion
string upper = message.toupper();  // Convert to uppercase
string lower = message.tolower();  // Convert to lowercase

// More substring operations
string sub = message.substr(0, 5);          // Extract substring
string sub_from_index = message.substr(7);  // From index 7 to end
string sub_range = message.substr(7, 6);    // 6 chars starting from index 7

// Character access
byte first_char = message.getc(0);  // Get character at index 0
byte char_at_5 = message.getc(5);   // Get character at index 5
message.putc(5, ",");               // Set character at index 5 to ','

// String comparison
string test_str = "Hello";
int compare_result = message.compare(test_str);  // Compare strings
bit is_equal = (message == test_str);            // Direct equality check

// String to number conversion
string number_str = "456";               // String representing a number
string float_str = "78.9";               // String representing a float
int converted_int = number_str.atoi();   // String to integer
real converted_real = float_str.atof();  // String to real/float

// Additional formatting examples
string hex_format = $sformatf("Hex: %h", 255);                      // Hexadecimal format
string bin_format = $sformatf("Binary: %b", 15);                    // Binary format
string multi_format = $sformatf("Dec:%d Hex:%h Oct:%o", 42, 42, 42);  // Multiple formats
```

Arrays: Packed vs. Unpacked

Arrays in SystemVerilog can be either packed or unpacked, each serving different purposes in digital design and verification. Understanding the differences between these two array types is crucial for efficient coding and optimal hardware synthesis.

Packed Arrays

Packed arrays store all elements contiguously in memory as a single vector, making them ideal for hardware representation and bitwise operations.

Characteristics of Packed Arrays:

- Elements are stored as a continuous block of bits
- Support bitwise operations across the entire array
- More efficient for hardware synthesis
- Dimensions are specified **before** the variable name
- Can be used in assignments as a single entity
- Support bit-select and part-select operations

Syntax:

```
// General syntax: data_type [packed_dimension] variable_name
bit [7:0] packed_byte;           // 8-bit packed array
logic [3:0][7:0] packed_2d;      // 2D packed array: 4 elements of 8 bits each
reg [15:0][31:0] packed_mem;     // 16 words of 32 bits each
```

Example Usage:

```
bit [3:0][7:0] packed_array;     // 4 bytes packed together
packed_array = 32'hABCD_EF01;   // Assign entire array at once
packed_array[2] = 8'h55;        // Access individual byte
packed_array[1][4] = 1'b1;      // Access individual bit
```

Unpacked Arrays

Unpacked arrays store elements as separate entities in memory, with each element having its own memory location.

Characteristics of Unpacked Arrays:

- Each element occupies separate memory locations
- Cannot perform bitwise operations across the entire array
- More flexible for complex data structures
- Dimensions are specified **after** the variable name
- Elements must be accessed individually
- Better suited for representing memories and queues

Syntax:

```
// General syntax: data_type variable_name [unpacked_dimension]
bit unpacked_byte [8];           // 8 separate bits
logic unpacked_2d [4][8];        // 2D unpacked array: 4 rows, 8 columns
reg unpacked_mem [16][32];       // 16 words, each word has 32 bits
```

Example Usage:

```
bit [7:0] unpacked_array [4];    // 4 separate bytes
unpacked_array[0] = 8'hAB;       // Assign individual elements
unpacked_array[1] = 8'hCD;
// unpacked_array = 32'h12345678; // ERROR: Cannot assign entire array
```

Key Differences Summary

Aspect	Packed Arrays	Unpacked Arrays
Memory Layout	Contiguous bits	Separate memory locations
Dimension Syntax	Before variable name	After variable name
Bitwise Operations	Supported across entire array	Only on individual elements
Assignment	Can assign entire array	Must assign elements individually

Aspect	Packed Arrays	Unpacked Arrays
Hardware Synthesis	More efficient	Less efficient for simple operations
Use Cases	Buses, registers, bit vectors	Memories, lookup tables, complex structures

Mixed Arrays

SystemVerilog also supports mixed arrays that combine both packed and unpacked dimensions:

```
// Mixed array: packed and unpacked dimensions
logic [7:0] mixed_array [16];    // 16 separate bytes (each byte is packed)
bit [3:0][7:0] mixed_2d [8];     // 8 separate 32-bit packed words
```

Best Practices

1. **Use packed arrays** for:
   - Simple bit vectors and buses
   - When you need bitwise operations
   - Hardware registers and signals
   - When memory efficiency is important
2. **Use unpacked arrays** for:
   - Memory models
   - Lookup tables
   - When elements represent distinct entities
   - Complex data structures
3. **Consider mixed arrays** when you need both individual element access and bitwise operations within elements

Understanding these differences helps you choose the appropriate array type for your specific design and verification needs, leading to more efficient and maintainable SystemVerilog code.

Packed Arrays

Elements are stored contiguously in memory as a single vector.

```
// Packed array declaration
logic [7:0] packed_array;     // 8-bit packed array
bit [3:0][7:0] packed_2d;     // 2D packed array: 4 elements of 8 bits each

// Accessing packed arrays
packed_array[7] = 1'b1;       // Set MSB
packed_array[3:0] = 4'b1010;  // Set lower 4 bits

// Packed arrays can be treated as vectors
logic [31:0] word = packed_2d; // Entire array as 32-bit vector
```

Unpacked Arrays

Elements are stored as separate entities in memory.

```
// Unpacked array declaration
logic [7:0] unpacked_array [0:15];    // 16 elements of 8 bits each
```

```
int memory [0:1023];                  // 1024 integer elements
bit [3:0] lookup_table [0:255];       // 256 elements of 4 bits each

// Accessing unpacked arrays
unpacked_array[0] = 8'hAA;           // Set first element
memory[100] = 32'h12345678;          // Set element at index 100

// Multi-dimensional unpacked arrays
int matrix [0:7][0:7];               // 8x8 matrix
matrix[2][3] = 42;                   // Set element at row 2, column 3
```

Dynamic Arrays

Arrays whose size can be changed during runtime.

```
// Dynamic array declaration
int dynamic_array [];

// Allocate memory
dynamic_array = new[10];  // Create array with 10 elements

// Access elements
dynamic_array[0] = 100;
dynamic_array[9] = 200;

// Resize array
dynamic_array = new[20](dynamic_array);  // Resize to 20, preserve data
```

Comparison: Packed vs. Unpacked

Aspect	Packed	Unpacked
Storage	Contiguous bits	Separate elements
Vector operations	Supported	Not supported
Bit selection	Supported	Element-wise only
Memory efficiency	Higher	Lower
Flexibility	Limited	Higher

Structures and Unions

SystemVerilog supports user-defined composite data types through structures and unions, providing powerful data organization capabilities for complex hardware designs.

Packed vs Unpacked Structures

Packed structures store data contiguously in memory and can be treated as vectors, while unpacked structures may have gaps between members and are typically used for software-like data organization.

Union Data Types

Unions allow multiple data types to occupy the same memory space, enabling efficient memory usage and type conversion operations in hardware modeling.

Typedef Declarations

The `typedef` keyword enables creation of reusable custom data types, promoting code modularity and making complex structures easier to manage across large designs.

Structure Assignment and Comparison

SystemVerilog supports aggregate assignment operations for structures, allowing entire structures to be copied or compared in single operations rather than member-by-member manipulation.

Tagged Unions

Tagged unions provide type-safe variant data types with explicit type identification, enabling robust handling of data that can represent different types at runtime.

Structures (struct)

Group related data items together.

```
// Basic structure definition
typedef struct {
    logic [7:0] opcode;
    logic [15:0] operand1;
    logic [15:0] operand2;
    logic valid;
} instruction_t;

// Using the structure
instruction_t cpu_instruction;
cpu_instruction.opcode = 8'h01;
cpu_instruction.operand1 = 16'h1234;
cpu_instruction.operand2 = 16'h5678;
cpu_instruction.valid = 1'b1;

// Packed structures
typedef struct packed {
    logic [3:0] command;
    logic [7:0] address;
    logic [7:0] data;
    logic parity;
} packet_t;

packet_t network_packet;
// Can be treated as a 20-bit vector
logic [19:0] raw_data = network_packet;
```

Unions

Allow different data types to share the same memory space.

```
// Union definition
typedef union {
    logic [31:0] word;
    logic [15:0] half_word [0:1];
    logic [7:0] byte_data [0:3];
```

```
} data_union_t;

// Using the union
data_union_t data_reg;
data_reg.word = 32'h12345678;

// Access the same data in different formats
$display("Word: %h", data_reg.word);           // 12345678
$display("Half[0]: %h", data_reg.half_word[0]); // 5678
$display("Half[1]: %h", data_reg.half_word[1]); // 1234
$display("Byte[0]: %h", data_reg.byte_data[0]); // 78
```

Tagged Unions

Unions with type information to ensure safe access.

```
typedef union tagged {
    logic [7:0] byte_val;
    logic [15:0] word_val;
    string str_val;
} tagged_union_t;

tagged_union_t data;

// Setting values with tags
data = tagged byte_val 8'hAA;
data = tagged word_val 16'h1234;
data = tagged str_val "Hello";

// Safe access using case statement
case (data) matches
    tagged byte_val .b: $display("Byte: %h", b);
    tagged word_val .w: $display("Word: %h", w);
    tagged str_val .s: $display("String: %s", s);
endcase
```

Best Practices and Common Pitfalls

When working with SystemVerilog structures and unions, following established best practices helps avoid synthesis issues, simulation mismatches, and maintainability problems. Common pitfalls include mixing packed and unpacked types incorrectly, forgetting synthesis tool limitations, and creating overly complex nested structures that impact performance. Proper naming conventions, careful consideration of bit-width requirements, and understanding tool-specific constraints are essential for successful implementation in both simulation and hardware synthesis environments.

Best Practices

1. **Choose appropriate data types:**

   ```
   // Good: Use two-state types for testbenches
   int test_counter = 0;

   // Good: Use four-state types for hardware modeling
   ```

```
logic [7:0] data_bus;
```

2. **Use meaningful identifiers:**

```
// Good
logic clock_enable;
logic [7:0] instruction_opcode;

// Avoid
logic ce;
logic [7:0] data;
```

3. **Initialize variables:**

```
int counter = 0;
logic [3:0] state = 4'b0000;
```

Common Pitfalls

1. **Mixing four-state and two-state types:**

```
// Problematic: X/Z values will be converted to 0
logic [7:0] four_state = 8'bxxxx_xxxx;
int two_state = four_state;  // X becomes 0
```

2. **Array indexing confusion:**

```
// Packed array - bit selection
logic [7:0] packed_data;
packed_data[0] = 1'b1;  // Sets LSB

// Unpacked array - element selection
logic [7:0] unpacked_data [0:3];
unpacked_data[0] = 8'hFF;  // Sets first element
```

3. **Signed vs. unsigned operations:**

```
byte signed_val = -1;           // 8'hFF
bit [7:0] unsigned_val = 8'hFF; // 255

// Comparison might not work as expected
if (signed_val > unsigned_val)  // May not behave as intended
```

Example 1 - Hello World

Design under Test (DUT)

```
// design_under_test.sv
module design_module_name ();               // Design under test
  initial $display();                       // Display empty line
  initial $display("Hello from design!");   // Display message
endmodule
```

Design Unit Test (DUT) Testbench

```
// design_under_test_testbench.sv
module test_bench_module;  // Testbench module
  design_module_name DESIGN_INSTANCE_NAME();  // Instantiate design under test

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

  end

endmodule
```

Running the Testbench

To run the testbench, a Verilator envinronment is set up inside a Docker container.

The following python script can be used to run the testbench.

```
from verilator_runner import run_docker_compose

run_docker_compose("Chapter_2_examples/example_1__hello_world/")
```

```
Docker Compose Output:
================================================================================
make: Entering directory '/work/obj_dir'
ccache g++ -Os  -I.  -MMD -I/usr/local/share/verilator/include
-I/usr/local/share/verilator/include/vltstd -DVM_COVERAGE=0 -DVM_SC=0
-DVM_TIMING=1 -DVM_TRACE=1 -DVM_TRACE_FST=0 -DVM_TRACE_VCD=1 -DVM_TRACE_SAIF=0
-faligned-new -fcf-protection=none -Wno-bool-operation -Wno-shadow -Wno-sign-
compare -Wno-subobject-linkage -Wno-tautological-compare -Wno-uninitialized
-Wno-unused-but-set-parameter -Wno-unused-but-set-variable -Wno-unused-parameter
-Wno-unused-variable    -DVL_TIME_CONTEXT   -fcoroutines -c -o verilated.o
/usr/local/share/verilator/include/verilated.cpp
ccache g++ -Os  -I.  -MMD -I/usr/local/share/verilator/include
-I/usr/local/share/verilator/include/vltstd -DVM_COVERAGE=0 -DVM_SC=0
-DVM_TIMING=1 -DVM_TRACE=1 -DVM_TRACE_FST=0 -DVM_TRACE_VCD=1 -DVM_TRACE_SAIF=0
-faligned-new -fcf-protection=none -Wno-bool-operation -Wno-shadow -Wno-sign-
compare -Wno-subobject-linkage -Wno-tautological-compare -Wno-uninitialized
-Wno-unused-but-set-parameter -Wno-unused-but-set-variable -Wno-unused-parameter
-Wno-unused-variable    -DVL_TIME_CONTEXT   -fcoroutines -c -o verilated_vcd_c.o
/usr/local/share/verilator/include/verilated_vcd_c.cpp
ccache g++ -Os  -I.  -MMD -I/usr/local/share/verilator/include
-I/usr/local/share/verilator/include/vltstd -DVM_COVERAGE=0 -DVM_SC=0
-DVM_TIMING=1 -DVM_TRACE=1 -DVM_TRACE_FST=0 -DVM_TRACE_VCD=1 -DVM_TRACE_SAIF=0
-faligned-new -fcf-protection=none -Wno-bool-operation -Wno-shadow -Wno-sign-
compare -Wno-subobject-linkage -Wno-tautological-compare -Wno-uninitialized
-Wno-unused-but-set-parameter -Wno-unused-but-set-variable -Wno-unused-parameter
-Wno-unused-variable    -DVL_TIME_CONTEXT   -fcoroutines -c -o
verilated_timing.o /usr/local/share/verilator/include/verilated_timing.cpp
ccache g++ -Os  -I.  -MMD -I/usr/local/share/verilator/include
```

```
-I/usr/local/share/verilator/include/vltstd -DVM_COVERAGE=0 -DVM_SC=0
-DVM_TIMING=1 -DVM_TRACE=1 -DVM_TRACE_FST=0 -DVM_TRACE_VCD=1 -DVM_TRACE_SAIF=0
-faligned-new -fcf-protection=none -Wno-bool-operation -Wno-shadow -Wno-sign-
compare -Wno-subobject-linkage -Wno-tautological-compare -Wno-uninitialized
-Wno-unused-but-set-parameter -Wno-unused-but-set-variable -Wno-unused-parameter
-Wno-unused-variable    -DVL_TIME_CONTEXT   -fcoroutines -c -o
verilated_threads.o /usr/local/share/verilator/include/verilated_threads.cpp
python3 /usr/local/share/verilator/bin/verilator_includer
-DVL_INCLUDE_OPT=include Vtest_bench_module.cpp
Vtest_bench_module___024root__DepSet_h8ce72585__0.cpp
Vtest_bench_module___024root__DepSet_h7f22f951__0.cpp
Vtest_bench_module__main.cpp Vtest_bench_module__Trace__0.cpp
Vtest_bench_module___024root__Slow.cpp
Vtest_bench_module___024root__DepSet_h7f22f951__0__Slow.cpp
Vtest_bench_module__Syms.cpp Vtest_bench_module__Trace__0__Slow.cpp
Vtest_bench_module__TraceDecls__0__Slow.cpp > Vtest_bench_module__ALL.cpp
ccache g++ -Os  -I.  -MMD -I/usr/local/share/verilator/include
-I/usr/local/share/verilator/include/vltstd -DVM_COVERAGE=0 -DVM_SC=0
-DVM_TIMING=1 -DVM_TRACE=1 -DVM_TRACE_FST=0 -DVM_TRACE_VCD=1 -DVM_TRACE_SAIF=0
-faligned-new -fcf-protection=none -Wno-bool-operation -Wno-shadow -Wno-sign-
compare -Wno-subobject-linkage -Wno-tautological-compare -Wno-uninitialized
-Wno-unused-but-set-parameter -Wno-unused-but-set-variable -Wno-unused-parameter
-Wno-unused-variable    -DVL_TIME_CONTEXT   -fcoroutines -c -o
Vtest_bench_module__ALL.o Vtest_bench_module__ALL.cpp
echo "" > Vtest_bench_module__ALL.verilator_deplist.tmp
g++    verilated.o verilated_vcd_c.o verilated_timing.o verilated_threads.o
Vtest_bench_module__ALL.a    -pthread -lpthread -latomic   -o Vtest_bench_module
rm Vtest_bench_module__ALL.verilator_deplist.tmp
make: Leaving directory '/work/obj_dir'
- V e r i l a t i o n   R e p o r t: Verilator 5.036 2025-04-27 rev v5.036
- Verilator: Built from 0.035 MB sources in 3 modules, into 0.029 MB in 10 C++
files needing 0.000 MB
- Verilator: Walltime 26.956 s (elab=0.007, cvt=0.061, bld=26.666); cpu 0.048 s
on 1 threads; alloced 20.180 MB

Hello from design!
Hello from testbench!

- S i m u l a t i o n   R e p o r t: Verilator 5.036 2025-04-27
- Verilator: end at 1ps; walltime 0.004 s; speed 665.823 ps/s
- Verilator: cpu 0.002 s on 1 threads; alloced 25 MB
================================================================================
Process finished with return code: 0
Removing Chapter_2_examples/example_1__hello_world/obj_dir directory...
Chapter_2_examples/example_1__hello_world/obj_dir removed successfully.
0
```

Example 2 - Four State Simple

Design under Test (DUT)

```
// four_state_simple.sv
module four_state_simple();

  // Four-state data types - can store 0, 1, X, Z
```

```
  logic [3:0] signal_a;
  logic [3:0] signal_b;
  logic [3:0] result;

  initial begin
    $display("\n=== Four-State Data Types with Logic Operations (Verilator) ===\n");

    // Test 1: Normal binary values with AND operation
    $display("1. Normal Binary Values (AND operation):");
    signal_a = 4'b1010;             // 10 in decimal
    signal_b = 4'b0110;             // 6 in decimal
    #1;                             // Wait for assignments to complete
    result = signal_a & signal_b;   // AND operation
    #1;                             // Wait for result calculation
    $display("   signal_a = %b", signal_a);
    $display("   signal_b = %b", signal_b);
    $display("   result   = %b (a & b)", result);
    $display("   Expected: 1010 & 0110 = 0010");
    $display("   Correct?  %s", (result == 4'b0010) ? "YES" : "NO");

    #10;

    // Test 2: Unknown (X) values with OR operation
    $display("\n2. Unknown (X) Values (OR operation):");
    signal_a = 4'bXXXX;             // All unknown
    signal_b = 4'bX010;             // Mixed unknown and known
    #1;                             // Wait for assignments
    result = signal_a | signal_b;   // OR operation
    #1;                             // Wait for result
    $display("   signal_a = %b (Verilator converted XXXX)", signal_a);
    $display("   signal_b = %b (Verilator converted X010)", signal_b);
    $display("   result   = %b (a | b)", result);
    $display("   Note: Verilator converts X to deterministic 0/1 values");

    #10;

    // Test 3: High-impedance (Z) values with XOR operation
    $display("\n3. High-Impedance (Z) Values (XOR operation):");
    signal_a = 4'bZZZZ;             // All tri-state
    signal_b = 4'b1Z0Z;             // Mixed tri-state and known
    #1;                             // Wait for assignments
    result = signal_a ^ signal_b;   // XOR operation
    #1;                             // Wait for result
    $display("   signal_a = %b (Verilator converted ZZZZ)", signal_a);
    $display("   signal_b = %b (Verilator converted 1Z0Z)", signal_b);
    $display("   result   = %b (a ^ b)", result);
    $display("   Note: Verilator converts Z to deterministic 0/1 values");

    #10;

    // Test 4: Mixed states with NOT operation
    $display("\n4. All Four States (NOT operation):");
    signal_a = 4'b01XZ;             // One of each state
    #1;                             // Wait for assignment
    result = ~signal_a;             // NOT operation
```

```
    #1;                                 // Wait for result
    $display("   signal_a = %b (Verilator converted 01XZ)", signal_a);
    $display("   result   = %b (~a)", result);
    $display("   Manual check: ~%b%b%b%b = %b%b%b%b",
             signal_a[3], signal_a[2], signal_a[1], signal_a[0],
             result[3], result[2], result[1], result[0]);

    #10;

    // Test 5: Show what Verilator actually does with assignments
    $display("\n5. Verilator's X/Z Conversion Demonstration:");

    // Multiple assignments of same X/Z pattern
    signal_a = 4'bXXXX;
    #1;
    $display("   4'bXXXX â†’ %b (assignment 1)", signal_a);

    signal_a = 4'bXXXX;
    #1;
    $display("   4'bXXXX â†’ %b (assignment 2 - same pattern)", signal_a);

    signal_a = 4'bZZZZ;
    #1;
    $display("   4'bZZZZ â†’ %b (Z assignment)", signal_a);

    signal_a = 4'b01XZ;
    #1;
    $display("   4'b01XZ â†’ %b (mixed assignment)", signal_a);

    #10;

    $display("\n=== Demo Complete ===");
    $display("SUMMARY for Verilator (2-state simulator):");
    $display("- X and Z are converted to deterministic 0/1 values");
    $display("- Same X/Z pattern always converts to same 0/1 pattern");
    $display("- Logic operations work on the converted 0/1 values");
    $display("- Case equality (===) compares the converted values");
    $display("\nFor true 4-state simulation, use:");
    $display("- Icarus Verilog: iverilog + vvp");
    $display("- ModelSim/QuestaSim");
    $display("- Synopsys VCS");
  end

endmodule
```

Design Unit Test (DUT) Testbench

```
module four_state_simple_testbench;

  // Instantiate the design
  four_state_simple DUT();

  initial begin
    // Setup waveform dumping
```

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

    // Wait for design to complete
    #60;

    $display("\nTestbench finished - check waveforms!");
    $display();
    $finish;
  end

endmodule
```

Running the Testbench

To run the testbench, a Verilator envinronment is set up inside a Docker container.

The following python script can be used to run the testbench.

```
from verilator_runner import run_docker_compose

run_docker_compose("Chapter_2_examples/example_2__four_two_state/")
```

```
Docker Compose Output:
================================================================================
make: Entering directory '/work/obj_dir'
ccache g++ -Os  -I.  -MMD -I/usr/local/share/verilator/include
-I/usr/local/share/verilator/include/vltstd -DVM_COVERAGE=0 -DVM_SC=0
-DVM_TIMING=1 -DVM_TRACE=1 -DVM_TRACE_FST=0 -DVM_TRACE_VCD=1 -DVM_TRACE_SAIF=0
-faligned-new -fcf-protection=none -Wno-bool-operation -Wno-shadow -Wno-sign-
compare -Wno-subobject-linkage -Wno-tautological-compare -Wno-uninitialized
-Wno-unused-but-set-parameter -Wno-unused-but-set-variable -Wno-unused-parameter
-Wno-unused-variable    -DVL_TIME_CONTEXT   -fcoroutines -c -o verilated.o
/usr/local/share/verilator/include/verilated.cpp
ccache g++ -Os  -I.  -MMD -I/usr/local/share/verilator/include
-I/usr/local/share/verilator/include/vltstd -DVM_COVERAGE=0 -DVM_SC=0
-DVM_TIMING=1 -DVM_TRACE=1 -DVM_TRACE_FST=0 -DVM_TRACE_VCD=1 -DVM_TRACE_SAIF=0
-faligned-new -fcf-protection=none -Wno-bool-operation -Wno-shadow -Wno-sign-
compare -Wno-subobject-linkage -Wno-tautological-compare -Wno-uninitialized
-Wno-unused-but-set-parameter -Wno-unused-but-set-variable -Wno-unused-parameter
-Wno-unused-variable    -DVL_TIME_CONTEXT   -fcoroutines -c -o verilated_vcd_c.o
/usr/local/share/verilator/include/verilated_vcd_c.cpp
ccache g++ -Os  -I.  -MMD -I/usr/local/share/verilator/include
-I/usr/local/share/verilator/include/vltstd -DVM_COVERAGE=0 -DVM_SC=0
-DVM_TIMING=1 -DVM_TRACE=1 -DVM_TRACE_FST=0 -DVM_TRACE_VCD=1 -DVM_TRACE_SAIF=0
-faligned-new -fcf-protection=none -Wno-bool-operation -Wno-shadow -Wno-sign-
compare -Wno-subobject-linkage -Wno-tautological-compare -Wno-uninitialized
-Wno-unused-but-set-parameter -Wno-unused-but-set-variable -Wno-unused-parameter
-Wno-unused-variable    -DVL_TIME_CONTEXT   -fcoroutines -c -o
verilated_timing.o /usr/local/share/verilator/include/verilated_timing.cpp
ccache g++ -Os  -I.  -MMD -I/usr/local/share/verilator/include
-I/usr/local/share/verilator/include/vltstd -DVM_COVERAGE=0 -DVM_SC=0
-DVM_TIMING=1 -DVM_TRACE=1 -DVM_TRACE_FST=0 -DVM_TRACE_VCD=1 -DVM_TRACE_SAIF=0
-faligned-new -fcf-protection=none -Wno-bool-operation -Wno-shadow -Wno-sign-
compare -Wno-subobject-linkage -Wno-tautological-compare -Wno-uninitialized
-Wno-unused-but-set-parameter -Wno-unused-but-set-variable -Wno-unused-parameter
```

```
-Wno-unused-variable    -DVL_TIME_CONTEXT   -fcoroutines -c -o
verilated_threads.o /usr/local/share/verilator/include/verilated_threads.cpp
python3 /usr/local/share/verilator/bin/verilator_includer
-DVL_INCLUDE_OPT=include Vfour_state_simple_testbench.cpp
Vfour_state_simple_testbench___024root__DepSet_hf99045df__0.cpp
Vfour_state_simple_testbench___024root__DepSet_h48076a0c__0.cpp
Vfour_state_simple_testbench__main.cpp
Vfour_state_simple_testbench__Trace__0.cpp
Vfour_state_simple_testbench___024root__Slow.cpp
Vfour_state_simple_testbench___024root__DepSet_hf99045df__0__Slow.cpp
Vfour_state_simple_testbench___024root__DepSet_h48076a0c__0__Slow.cpp
Vfour_state_simple_testbench__Syms.cpp
Vfour_state_simple_testbench__Trace__0__Slow.cpp
Vfour_state_simple_testbench__TraceDecls__0__Slow.cpp >
Vfour_state_simple_testbench__ALL.cpp
ccache g++ -Os  -I.  -MMD -I/usr/local/share/verilator/include
-I/usr/local/share/verilator/include/vltstd -DVM_COVERAGE=0 -DVM_SC=0
-DVM_TIMING=1 -DVM_TRACE=1 -DVM_TRACE_FST=0 -DVM_TRACE_VCD=1 -DVM_TRACE_SAIF=0
-faligned-new -fcf-protection=none -Wno-bool-operation -Wno-shadow -Wno-sign-
compare -Wno-subobject-linkage -Wno-tautological-compare -Wno-uninitialized
-Wno-unused-but-set-parameter -Wno-unused-but-set-variable -Wno-unused-parameter
-Wno-unused-variable    -DVL_TIME_CONTEXT   -fcoroutines -c -o
Vfour_state_simple_testbench__ALL.o Vfour_state_simple_testbench__ALL.cpp
echo "" > Vfour_state_simple_testbench__ALL.verilator_deplist.tmp
g++    verilated.o verilated_vcd_c.o verilated_timing.o verilated_threads.o
Vfour_state_simple_testbench__ALL.a    -pthread -lpthread -latomic   -o
Vfour_state_simple_testbench
rm Vfour_state_simple_testbench__ALL.verilator_deplist.tmp
make: Leaving directory '/work/obj_dir'
- V e r i l a t i o n   R e p o r t: Verilator 5.036 2025-04-27 rev v5.036
- Verilator: Built from 0.041 MB sources in 3 modules, into 0.054 MB in 11 C++
files needing 0.000 MB
- Verilator: Walltime 26.622 s (elab=0.001, cvt=0.149, bld=26.191); cpu 0.036 s
on 1 threads; alloced 20.176 MB

=== Four-State Data Types with Logic Operations (Verilator) ===

1. Normal Binary Values (AND operation):
   signal_a = 1010
   signal_b = 0110
   result   = 0010 (a & b)
   Expected: 1010 & 0110 = 0010
   Correct?  YES

2. Unknown (X) Values (OR operation):
   signal_a = 1010 (Verilator converted XXXX)
   signal_b = 0110 (Verilator converted X010)
   result   = 1110 (a | b)
   Note: Verilator converts X to deterministic 0/1 values

3. High-Impedance (Z) Values (XOR operation):
   signal_a = 1010 (Verilator converted ZZZZ)
   signal_b = 1110 (Verilator converted 1Z0Z)
   result   = 0100 (a ^ b)
   Note: Verilator converts Z to deterministic 0/1 values
```

```
4. All Four States (NOT operation):
   signal_a = 1110 (Verilator converted 01XZ)
   result   = 0001 (~a)
   Manual check: ~1110 = 0001

5. Verilator's X/Z Conversion Demonstration:
   4'bXXXX â†’ 1110 (assignment 1)
   4'bXXXX â†’ 1110 (assignment 2 - same pattern)
   4'bZZZZ â†’ 1110 (Z assignment)
   4'b01XZ â†’ 1110 (mixed assignment)

Testbench finished - check waveforms!

- four_state_simple_testbench.sv:16: Verilog $finish
- S i m u l a t i o n   R e p o r t: Verilator 5.036 2025-04-27
- Verilator: $finish at 62ps; walltime 0.003 s; speed 41.977 ns/s
- Verilator: cpu 0.001 s on 1 threads; alloced 25 MB
================================================================================
Process finished with return code: 0
Removing Chapter_2_examples/example_2__four_two_state/obj_dir directory...
Chapter_2_examples/example_2__four_two_state/obj_dir removed successfully.

0
```

Summary

This chapter covered the fundamental building blocks of SystemVerilog:

- **Lexical conventions** provide the basic rules for writing SystemVerilog code
- **Comments and identifiers** help in code documentation and naming
- **Four-state vs. two-state** data types serve different modeling needs
- **Integer types** offer various sizes and characteristics for different applications
- **Real and string types** handle floating-point numbers and text data
- **Arrays** provide both packed and unpacked storage options
- **Structures and unions** enable creation of complex data types

Understanding these concepts is crucial for writing effective SystemVerilog code, whether for synthesis, simulation, or verification purposes. The next chapter will explore operators and expressions, building upon these fundamental data types.