SystemVerilog Language Constructs

Introduction

Before studying SystemVerilog data types in detail, it is important to understand **why SystemVerilog syntax exists in the first place**.

SystemVerilog was not created to replace Verilog arbitrarily.
It was designed to **extend Verilog**, addressing limitations that became evident as designs and verification environments grew in complexity.

Understanding the **syntactic evolution from Verilog to SystemVerilog** helps avoid:

- writing Verilog-style code in SystemVerilog
- misusing language constructs
- missing the advantages SystemVerilog provides for verification

The below highlights **key syntax-level differences** between Verilog and SystemVerilog, with emphasis on **verification relevance**.

1. Language Purpose and Philosophy

Verilog

- Originally designed as a **hardware description language**
- Focused on:
  - modeling RTL
  - describing hardware structure
  - synthesizable logic
- Limited support for large testbenches

SystemVerilog

- Designed as a **unified language for design and verification**
- Extends Verilog with:
  - higher abstraction
  - verification-friendly constructs
  - software-like features
- Suitable for:
  - RTL
  - testbenches
  - verification frameworks

Key shift: From *hardware description only* ***to*** *hardware + verification language*.

2. Variable Declarations

Verilog Syntax
In Verilog, variables are declared using reg and wire:

- wire for continuous assignments
- reg for procedural assignments

This creates confusion:

- reg does not mean hardware register
- Semantic intent is unclear

SystemVerilog Syntax
SystemVerilog introduces logic:

- Can be used in procedural blocks
- Can replace most reg usage
- Clearer and safer semantics

Impact: SystemVerilog simplifies variable declaration and reduces ambiguity in verification code.

3. Module Port Declarations

Verilog Style
Verilog requires separate port and signal declarations, increasing verbosity:

- Ports declared first
- Signals declared later
- Easy to mismatch types

SystemVerilog Style
SystemVerilog allows **data types to be declared directly in the port list**:

- Less boilerplate
- Improved readability
- Reduced errors

Impact: Cleaner module interfaces and more maintainable code.

4. Data Structures

Verilog Limitations
Verilog supports:

- basic vectors
- limited arrays

Lacks:

- structures
- enumerations
- dynamic collections

As a result:

- Complex data must be flattened

- Code becomes difficult to read and maintain

SystemVerilog Enhancements
SystemVerilog introduces:

- struct
- enum
- typedef
- dynamic arrays
- queues
- associative arrays

Impact: Verification code can model **transactions**, not just signals.

5. Procedural Blocks

Verilog
Uses:

- always
- initial

Ambiguity exists:

- Same block used for combinational and sequential logic
- Easy to introduce simulation mismatches

SystemVerilog
Adds:

- always_comb
- always_ff
- always_latch

These blocks:

- Express intent clearly
- Help tools catch coding errors
- Improve code quality

Impact: Better correctness and readability in both design and verification.

6. Control and Looping Constructs

Verilog
Supports basic constructs:

- if
- case
- for
- while

Limited expressiveness for complex test scenarios.

SystemVerilog
Extends control constructs with:

- enhanced foreach
- better looping semantics
- improved case handling

Impact: Testbench code becomes more concise and expressive.

7. Strings and Debugging

Verilog

- No native string type
- Debug messages rely on fixed-size vectors
- Poor readability in logs

SystemVerilog

- Native string type
- Rich formatting support
- Easier debug and logging

Impact: Significantly improved debug productivity

8. Randomization and Constraints (High-Level View)

Verilog

- No built-in randomization
- Test stimulus must be manually coded

SystemVerilog

- Built-in randomization
- Constraint support at the language level

(Note: Detailed coverage of this comes later.)

Impact: SystemVerilog enables **constrained-random verification** natively.

Understanding these differences ensures that:

- SystemVerilog is used **as intended**
- Verification code is written at the **right abstraction level**
- Language features are leveraged effectively

System Verilog Datatypes:

Day 2 focuses on **SystemVerilog data modeling and execution semantics**, covering:

- Data representation (2-state vs 4-state)
- Variable and net types
- All array categories
- User-defined data structures
- Procedural execution and assignments

1. Data Representation: 2-State vs 4-State

1.1 What Are Logic States?

Digital hardware signals can represent more than just 0 and 1. SystemVerilog supports **four logic states** to accurately model real hardware behavior.

State	Meaning
0	Logic Low
1	Logic High
X	Unknown (Unutilized)
Z	High Impedence (Tri-State)

1.2 2-State Data Types

2-state types can represent only 0 and 1.

Common 2-state types:

- bit
- byte
- int
- shortint
- longint

Key Properties:

- Faster simulation
- Lower memory usage
- Cannot represent unknown (X) or high-impedance (Z)

Usage:

- Testbench variables
- Counters, loop indices
- Algorithmic modeling

1.3 4-State Data Types

4-state types can represent 0, 1, X, and Z.

Common 4-state types:

- logic
- reg (legacy)
- wire
- integer

Key Properties:

- Accurately models hardware
- Required for RTL design
- Detects uninitialized signals

Usage:

- RTL signals
- Buses and tri-state logic

Summary:

Feature	2 - State	4 - State
Supports X / Z	No	Yes
Simulation Speed	Faster	Slower
Hardware Accuracy	Low	High
RTL Suitability	No	Yes

2. Variable and Net Types

2.1 Variables

Variables store values procedurally and retain their value until they are explicitly reassigned. They model **storage elements** in hardware or temporary storage in testbenches. Variables can be **2-state or 4-state**, depending on the data type used.

Common variable types are:

Type	**State**	**Description**
bit	2 state	Fast, memory efficient
int	2 state	Arithmetic, counters
logic	4 state	RTL storage signals
integer	4 state	Arithmetic type

Below are in detailed variable datatypes:

Integral Data Types:

Data Type	**State**	**Size**	**Signed / Unsigned**	**Description / Usage**
bit	2-state	1 bit	Unsigned	Fast, memory-efficient single bit
bit [N:0]	2-state	User-defined	Unsigned	Multi-bit 2-state vector
logic	4-state	1 bit	Unsigned	RTL storage signal (recommended)
logic [N:0]	4-state	User-defined	Unsigned	RTL buses, registers
byte	2-state	8 bit	**Signed**	Small integers, data bytes
shortint	2-state	16 bit	**Signed**	Compact arithmetic
int	2-state	32 bit	**Signed**	Counters, loop variables
int unsigned	2-state	32 bit	Unsigned	Addresses, sizes
longint	2-state	64 bit	**Signed**	Large counters
integer	4-state	32 bit	**Signed**	Legacy arithmetic type
time	4-state	64 bit	Unsigned	Simulation time storage

Floating-Point & Real Types:

Data Type	**State**	**Size**	**Description**
real	2-state	64 bit	Double precision floating point
shortreal	2-state	32 bit	Single precision floating point
realtime	4-state	64 bit	Time with real precision

Where Variables Are Assigned ?

Variables are assigned **inside procedural blocks**:

- initial
- always
- always_ff
- always_comb

Key Points About Variables

- Variables **store values**
- Assigned procedurally
- Retain last assigned value
- Used for registers, counters, FSM states

Key Rule: Variables represent **storage elements**.

2.2 Nets

Nets represent **physical connections** between hardware components.

Common net type:

- Wire

Key Characteristics:

- Do not store values
- Driven continuously
- Can have multiple drivers

Assigned using:

- assign statements

Net Types:

Data Type	**State**	**Storage**	**Description**
wire	4-state	No	Physical connection
tri	4-state	No	Tri-state wire
wand	4-state	No	Wired-AND logic
wor	4-state	No	Wired-OR logic

Variable vs Net:

Feature	**Variables**	**Nets**
Stores value	Yes	No
Assigned in	Procedural blocks	Continuous assign
Can be 2-state	Yes	No
Can be 4-state	Yes	Yes

Feature	Variables	Nets
Multiple drivers	No	Yes
Models	Storage	Connectivity

Recommended Usage:

Purpose	Recommended Type
RTL registers	logic
RTL buses	logic [N:0]
Counters	int / int unsigned
Testbench variables	bit, int, real
Physical connections	wire
FSM states	enum logic

System Verilog Example codes:

- ***Behavior of 2 state vs 4 state variable***

```
module two_vs_four_state;

  bit   a;      // 2-state
  logic b;      // 4-state

  initial begin
    a = 1'bx;   // x converted to 0
    b = 1'bx;   // Retains x

    $display("2-state bit a   = %b", a);
    $display("4-state logic b = %b", b);
  end

endmodule
```

Observation : bit silently converts X → 0, and logic preserves X, exposing uninitialized bugs. **This is why verification prefers 4-state modeling.**

- ***Variable Storage Behavior***

```
module variable_storage;
```

```
  logic reg_like;

  initial begin
    reg_like = 1'b1;
    $display("Time = %0t | Stored value = %b", $time, reg_like);
    #5;
    $display("Time = %0t | Stored value = %b", $time, reg_like);
  end

endmodule
```

Output:

```
# Time = 0 | Stored value = 1
# Time = 5 | Stored value = 1
```

Observation: The variable retains its value across time. Models registers, counters, FSM states

- ***Net (wire) Connectivity Behavior***

```
module net_example;

  wire w;
  logic driver;

  assign w = driver;

  initial begin
    driver = 1'b1;
    #1 $display("Time = %0t | wire w = %b",$time, w);

    driver = 1'b0;
    #1 $display("Time = %0t | wire w = %b",$time, w);
  end

endmodule
```

Output:

```
# Time = 1 | wire w = 1
# Time = 2 | wire w = 0
```

Observation : wire reflects the driver value instantly. No storage involved. **Nets model physical**

connections, not memory.

3. Arrays

SystemVerilog supports multiple array categories for flexible data modeling.

Unlike Verilog, which had limited array support, SystemVerilog allows engineers to represent:
- Bit-level data
- Collections of elements
- Variable-length data
- Dynamic and sparse data structures

Choosing the correct array type is critical for **readability, correctness, and performance** in verification environments.

3.1 Packed Arrays

Packed arrays represent a **group of bits packed together** into a single vector. They are treated as **one composite value**, similar to a register or bus.

Example: logic [7:0] data;

Key Characteristics
- Bit-level representation
- Stored as a single value
- Supports arithmetic and bitwise operations
- Can be sliced and indexed at the bit level

Use Cases
- Buses
- Registers
- Protocol fields
- Data words

Important Note: Packed arrays are indexed **from left to right** and behave like a single signal.

3.2 Unpacked Arrays

Unpacked arrays represent a **collection of individual elements**, where each element is a separate variable.
logic data [7:0];
Key Characteristics
- Element-level storage

- Each index represents a separate variable
- Accessed using array indexing
- Cannot be treated as a single arithmetic value

Use Cases
- Memories
- FIFOs
- Lookup tables
- Storage arrays

Key Difference from Packed Arrays: Packed arrays group bits → Unpacked arrays group elements.

3.3 Fixed-Size Arrays

Fixed-size arrays have a size that is **known at compile time**.

Example: int arr[4];

Characteristics
- Static memory allocation
- Size cannot change during simulation
- Simple and efficient

Use Cases
- Small buffers
- Fixed lookup tables
- Register files with known depth

3.4 Dynamic Arrays

Dynamic arrays allow the array size to be **decided at runtime**.
Example : int arr[];
arr = new[10];

Key Characteristics
- Allocated using new[]
- Size can change dynamically
- Memory efficient for variable-length data

Advantages
- Flexible sizing
- Efficient memory usage
- Ideal for verification stimulus

Use Cases
- Variable-length packets

- Protocol payloads
- Dynamic stimulus generation

Note: Dynamic arrays must be explicitly allocated before use.

3.5 Queues

Queues are **ordered collections** with automatic resizing and built-in methods.
Example: int q[$];
q.push_back(5);
q.pop_front();

Key Characteristics
- Auto-growing and shrinking
- Supports FIFO and LIFO behavior
- Provides built-in methods

Common Methods
- push_back()
- push_front()
- pop_back()
- pop_front()
- size()

Use Cases
- Transaction buffering
- Stimulus-response matching
- Modeling pipelines and FIFOs

Queues are heavily used in **verification environments**.

3.6 Associative Arrays

Associative arrays are indexed using **keys instead of numeric indices**.
Example: int mem[string];
mem["addr1"] = 10;

Key Characteristics
- Sparse storage
- Index can be int, string, or other types
- No fixed size

Use Cases
- Scoreboards
- Lookup tables
- Sparse memories
- Reference models

Note: Associative arrays store only the entries that are used, saving memory.

4. User-Defined Data Structures

SystemVerilog allows engineers to create **custom data types** that express intent clearly and reduce code complexity.

4.1 typedef

typedef creates an alias for an existing data type.
Example: typedef logic [7:0] byte_t;

Benefits
- Improves readability
- Simplifies refactoring
- Encapsulates type meaning

Note: Widely used in verification and RTL code.

4.2 struct

Structures group **related fields into a single data type**.

```
Example: typedef struct {
  int   id;
  logic valid;
} packet_t;
```

Benefits
- Clean data organization
- Groups logically related data
- Improves readability and maintainability

Use Cases
- Transactions
- Packets
- Configuration objects

Note:Structs allow verification code to model **transactions, not just signals**.

4.3 enum

Enumerations define a **fixed set of named values**.
Example: typedef enum {IDLE, BUSY, DONE} state_t;

Advantages
- Avoids magic numbers
- Improves FSM clarity
- Easier debugging and logging

Use Cases
- FSM states
- Protocol states
- Mode selection

5. Procedural Execution and Assignments

SystemVerilog defines **how and when code executes** through procedural blocks and assignment semantics.

5.1 Procedural Blocks

Block	**Purpose**
initial	Executes once at time 0
always	Repeats forever
always_comb	Combinational logic
always_ff	Sequential logic (flip-flops)

Advantages of Specialized Blocks
- Clear intent
- Automatic sensitivity handling
- Tool-assisted error detection

always_comb and always_ff are preferred over plain always.

5.2 Blocking Assignments (=)

Blocking assignments execute **immediately and sequentially**.
Example:
```
a = b;
b = c;
```

Characteristics
- Executes in order
- Used in combinational logic
- Common in testbenches

5.3 Non-Blocking Assignments (<=)

Non-blocking assignments execute **concurrently at the end of the timestep**.
Example:
```
a <= b;
```

```
b <= c;
```

Characteristics

- Models hardware parallelism
- Used in sequential logic
- Prevents race conditions

5.4 Blocking vs Non-Blocking Summary

Feature	**Blocking (=)**	**Non-Blocking (<=)**
Execution	Immediate	End of timestep
Used in	Combinational	Sequential
Order matters	Yes	No

Golden Rule

- Combinational → Blocking (=)
- Sequential → Non-Blocking (<=)

- ## System Verilog Code examples:

1. **Packed Arrays**

```
module packed_array_example;
logic [7:0] data;

initial begin
data = 8'hA5;
$display("data = %h", data);
end
endmodule
```

```
Output:  data = a5
```

2. **Unpacked Arrays**

```
module unpacked_array_example;
logic data [7:0];

initial begin
data[0] = 1'b1;
```

```
data[1] = 1'b0;
$display("data[0] = %b", data[0]);
end
endmodule
```

Output : data[0] = 1

3. **Fixed Size Arrays**

```
module fixed_array_example;
int arr[4];


initial begin
arr[0] = 10;
arr[1] = 20;
$display("arr[1] = %0d", arr[1]);
end
endmodule
```

Output : arr[1] = 20

4. **Dynamic Arrays**

```
module dynamic_array_example;
int arr[];

initial begin
arr = new[3];
arr[0] = 5;
arr[1] = 15;
arr[2] = 25;
$display("arr size = %0d", arr.size());
end
endmodule
```

Output : arr size = 3

5. **Queues**

```
module queue_example;
int q[$];
```

```
initial begin
q.push_back(10);
q.push_back(20);
$display("pop = %0d", q.pop_front());
end
endmodule
```

Output : pop = 10

6. **Associative Arrays**

```
module associative_array_example;
int mem[string];

initial begin
mem["addr1"] = 100;
mem["addr2"] = 200;
$display("mem[addr1] = %0d", mem["addr1"]);
end
endmodule
```

Output : mem[addr1] = 100

7. **Typedef**

```
module typedef_example;
typedef logic [7:0] byte_t;
byte_t data;


initial begin
data = 8'hFF;
$display("data = %h", data);
end
endmodule
```

Output : ff

8. **Struct**

```
module struct_example;
typedef struct {
int id;
logic valid;
} packet_t;
```

```
packet_t pkt;


initial begin
pkt.id = 1;
pkt.valid = 1'b1;
$display("id=%0d valid=%b", pkt.id, pkt.valid);
end
endmodule
```

Output : id=1 valid=1

9. **Enum**

```
module enum_example;
typedef enum {IDLE, BUSY, DONE} state_t;
state_t state;


initial begin
state = BUSY;
$display("state = %0d", state);
end
endmodule
```

Output : state = 1

Key Takeaways – Day 2

- SystemVerilog provides **multiple array types** for different modeling needs
- Packed arrays model **bit-level data**
- Unpacked, dynamic, queue, and associative arrays model **collections**
- typedef, struct, and enum improve **clarity and intent**
- Correct use of procedural blocks and assignments is **critical for correctness**
- Strong verification begins with **strong data modeling**

What Comes Next ?

Procedural Blocks, Tasks & Functions