**SYSTEMVERILOG**

SystemVerilog is a superset of Verilog that adds design, modeling, and verification enhancements:

In short: **Verilog = Design** whereas **SystemVerilog = Design + Verification + Scalability**

🔹 **RTL Clarity**: always\_comb, logic, interface — fewer bugs, more intent.  
🔹 **Verification Power**: Classes, Constraints, Coverage, Assertions, Random coverage Verification.  
🔹 **Scalability**: Reusable, modular, layered testbenches.  
🔹 **OOP:** Support object-oriented programming (OOP) with its properties(Classes, Inheritance, Polymorphism)  
🔹 **IPC:** Offers better IPC tools (mailbox, semaphore)   
🔹 **Enhanced datatype**: dynamic allocation, Stronger type system (logic, bit, struct, enum)

If Verilog is the alphabet, SystemVerilog is the poetry

DATA TYPE

**HARDWARE (SYTHESIZABLE)**

**USER DEFINE(SYTHESIZABLE)**

**VARIABLE(SIMULATION)**

**wire**

* Used to **connect components**.
* Cannot hold values; must be driven continuously by assigned or module outputs.
* Default value: ’**z**’(impedance)
* **Can’t** be used in **procedural assignment**

**Struct**

Group of different data type into single variable.

struct {

data\_type1 member1;

data\_type2 member2; …

} struct\_name;

**Floating number**

|  |  |  |
| --- | --- | --- |
| Type | Bit-Width | Precision |
| shortreal | 32-bit | Single-precision |
| real | 64-bit | Double-precision |

**Enum**

Define name constants

typedef enum {IDLE, START, STOP} state\_t;

**Fixed number**

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Signed? | Range (N-bit) | USE |
| logic | Unsigned | 0 to 2^N - 1 | General-purpose  RTL |
| reg | Unsigned | 0 to 2^N - 1 | Legacy Verilog (use logic in SV) |
| bit | Unsigned | 0 to 2^N - 1 | 2-state (0,1),  no 'X'/'Z' |
| int | Signed | -2^31 to 2^31-1 | 32-bit |
| integer | Signed | -2^31 to 2^31-1 | 32 Bit |
| byte | Signed | -128 to 127 | 8-bit signed |
| shortint | Signed | -32768 to 32767 | 16-bit signed |
| longint | Signed | -2^63 to 2^63-1 | 64-bit signed |

* Used for loop iteration as it can’t have x / z
* Improve performance
* Manage usages
* Bit- faster

**reg**

* Used in **procedural assignment**(always or initial).
* Holds its value between assignments.
* Default value: '**x**‘ (unknown).
* Can’t be used in **concurrent (continuous) assignments** (like assign).

**Union**

Shared storage with grouping

typedef union {

int i;

shortreal f;

} num\_u;

**Array**

Fixed-Size Arrays

int arr[4]; // 4-element

arr = '{10, 20, 30, 40};

Dynamic Arrays

int dyn\_arr[]; // Declare

dyn\_arr = new[5]; // Allocate 5 elements

**logic**

* **4 state- 1,0,x,z**
* Used in both procedural and continuous assignments.
* Can **replace reg and wire** (not for bidirectional inout nets).
* Holds values between assignments (like reg).
* Default value: 'x’ (unknown).
* Cannot be used in multiple conflicting drivers (unlike wire)

**Queue**

int q[$]; // Declare queue

q.push\_back(10); // at end

q.push\_front(20); // at front

int y = q.pop\_back; //remove get

www.linkedin.com/in/harshita-harshi-9377891bb

Array

**Fixed-size (Packed and Unpacked):**

* Packed array: Contiguous data set of the same data type, like a wide bus, we can’t declare a packed array using byte, int, shortint, long int, because they are already packed and we can’t do bit select.

logic [7:0] byte\_data; // 8 bits

* Unpacked array: Dimension declared after identifier name, may or may not represent a continuous set of bits, like memory

logic [7:0] mem\_array [0:15]; // 16 locations, 8-bit each

Used for the bus interface and control signal.

**👉 Packed = bit-wise storage, used for arithmetic 👉 Unpacked = element-wise storage (memory-style)**

**Dynamic Arrays (Runtime-sized):**

Size can be decided during simulation**.**

* **Contiguous memory block** with runtime-resizable size
* Default: **Empty** (unallocated) → Requires explicit new[]
* Support all array operations (slicing, methods) B

int dyn\_array[];

* dyn\_arr = new[10]; // after this, only memory is

allocated, and the previously stored data is cleared

* arr new[30](array); // will copy previous value
* arr1 = arr; // copying
* arr.delete( ); //delete all the elements
* $size(arr); //size of array
* arr[index]=value; // saving value to particular index

**Use Cases**

* RTL: Configurable memory buffers
* Testbenches: Flexible data sets
* Modeling: Unknown-size datasets

**Associative Arrays (Key-Value):**

Indexed by key (not just integers). Good for sparse data.

* Non-contiguous, dynamically allocated.
* Indexed by any datatype (e.g., string, int, class).
* Efficient for sparse data (only stored keys use memory)
* implements lookup table of its declared type.
* int aarr[string]; // Key: string, Value: int
* aarr["id1"] = 42; // Assign
* $display(aarr["id1"]); // Read (42)
* aarr.delete("key1"); // Remove entry
* aarr.delete(); // Clear entire array
* first( ); next( ); prev( ); last( );
* num( ); size( )

**Use Cases**

* + Sparse memory modeling (e.g., cache/MMU designs)
  + Configuration registers (string-based addressing)
  + Testbench databases (storing stimulus patterns)

**Limitations**

* Not synthesizable (simulation only)
* No guaranteed ordering of elements

**Special Features in Array Declaration**

**🔁 Default Value Initialization**

You can initialize an array with default values:

bit [7:0] arr [0:3] = '{default: 8'hFF};

This sets all 4 elements to 8'hFF.

**♻️ Repetition in Initialization**

Use '{'n{value}} to repeat:

bit [7:0] arr [0:3] = '{4{8'h0A}};

Same as setting all 4 values to 8'h0A.

**✅ Unique Initialization per Index**

bit [7:0] arr [0:3] = '{8'h01, 8'h02, 8'h03, 8'h04};

Each element is individually specified.

These syntaxes are powerful in testbenches for setting up register maps, memory images, etc.

**Array Manipulation**

Location:

find( ); find\_index( ); find\_first( ); find\_first\_index( );  
min( ); max( ); unique( ); unique\_index( );

Ordering:  
reverse( ); sort( ); rsort( ); shuffle( ):

Reduction:  
sum( ); product( ); and( ); or( ); xor( );

www.linkedin.com/in/harshita-harshi-9377891bb

**Queues**

Dynamic, ordered collection (FIFO/LIFO behavior)

Use .push\_back(), .pop\_front() for insertion/removal

Perfect for modeling transaction flows

* Unbounded Queue (int q[$])

Dynamic size: Grows/shrinks infinitely (limited only by memory).

No overflow risk: Always accepts new elements.

* Bounded Queue (int q[$:N])

Fixed maximum size: Limited to N elements (index 0:N).

Overflow protection: Rejects new pushes when full (no auto-resize).

int q[$];

q.push\_back(10); // Enqueue

q.push\_front(5); // Insert at front

int val = q.pop\_front(); // Dequeue (FIFO)

✨ Great for TB stimulus, scoreboard, packet queues

**⏲️ Blocks in SystemVerilog**

* **initial :** work only onces to initialize the global signals or get data signals value,monitor the response

initial begin

$display("Simulation started!");

#10; // Delay for 10 time units

$finish; // End simulation

end

* **always\_comb :** Automatic sensitivity list for combinational logic

always\_comb begin

y = a & b;

end

No risk of missing signals like in Verilog's always @(\*)

* **always\_ff :** For sequential logic (flip-flops)

always\_ff @(posedge clk or negedge rst\_n) begin

if (!rst\_n)

q <= 0;

else

q <= d;

end

Encourages proper reset + edge-triggered coding

Why this matters? Separating ff and comb logic avoids race conditions and unintended latches.

* **always\_latch :** For inferred latches (use rarely, usually a red flag!)

always\_latch begin

if (en)

q = d;

end

Use only when latch behavior is intentional

❗ Use **cautiously** — latches are hard to synthesize, and can cause timing issues.

|  |  |  |  |
| --- | --- | --- | --- |
| BLOCK | WHEN IT RUNS? | USE IN ? | SYNTHESIZABLE OR NOT? |
| initial block | Once at the 0th time | Testbench (initialization) | Non- synthesizable |
| always\_comb | On any input change | Combinational logic with avoiding latch Inference | Synthesizable |
| always\_ff | On clock change | Sequential logic | Synthesizable |
| always\_latch | On enable/ no else/ no default | Latch (Avoid it) | synthesizable |

www.linkedin.com/in/harshita-harshi-9377891bb

foreach loop

* **Auto-iterates over arrays**
* No manual index management

int arr[4] = '{1, 2, 3, 4};

foreach (arr[i]) $display("arr[%d] = %d", i, arr[i]);

repeat

* **Execute a block N times** (no counter)
* Ideal for **fixed repetitions** (e.g., clock cycles)

repeat (5) begin

$display("Repeating 5 times");

end

for loop

* Standard counter-based iteration
* Use for fixed-size arrays or known ranges

for (int i = 0; i < 10; i++) begin

$display("Iteration %d", i);

end

**Repetitive Operation (must be in procedural block)- Sequential execution**

\

**fork/join (must be in procedural block)- Concurrent execution**

SystemVerilog supports concurrent execution using fork and join. This is critical for writing verification code that models parallel activities like:

* Driving stimulus and checking output at the same time
* Running multiple threads of activity

fork- join\_any

Exit When Any Thread Finishes

Continues execution when any one thread completes.

fork

#10 a=1; // Thread 1

#20 b= 2; // Thread 2

#30 b= 2; // Thread 3

join

// joint execution starts after 10 when 1st execution completes

fork- join\_any

Starts threads but does not wait.

Continues execution without waiting for any thread to complete.

fork

#10 a=1; // Thread 1

#20 b= 2; // Thread 2

#30 b= 2; // Thread 3

join

// joint execution starts after 0

fork- join  
fork: Starts a parallel block (all statements inside run **concurrently**).

join: Waits for **all** parallel threads to finish before proceeding.

fork

#10 a=1; // Thread 1

#20 b= 2; // Thread 2

#30 b= 2; // Thread 3

join

// joint execution starts after 30

fork

fork

fork

T1

T3

T2

Thread 3

Thread 2

Thread 1

Thread 3

Thread 2

Thread 1

join

join

join

Tasks

Purpose: Execute procedural code that may include timing controls and multiple outputs

* Can contain delays (#10), event waits (@posedge), and blocking/non-blocking assignments
* Can call other tasks and functions
* Do not return values directly (but can modify output arguments)
* Can consume simulation time

task task\_name ([input/output args]);

// Procedural statements

endtask

Functions

Purpose: Compute and return a single value (like mathematical operations)

* Must return a value using return (SystemVerilog) or by assigning to the function name (Verilog)
* Cannot contain time delays (#, @, wait)
* Can call other functions but not tasks
* Execute in zero simulation time

function [return\_type] function\_name ([inputs]);

// Statements

return value; // SystemVerilog style

endfunction

www.linkedin.com/in/harshita-harshi-9377891bb