

A blue wavy background graphic with a gradient from light blue to dark blue, featuring a white wavy line at the top.

SystemVerilog Data Types

Data Types in Verilog

- **4-state** data types :
 - Values : 0, 1, X, Z
 - reg, integer, time (default value : X)
 - wire, wand, wor, uwire (default value : Z)
- **2-state** data types :
 - Values : 0, 1
 - real, realtime (default value : 0.0)

```
reg [3:0] v; //range 0 to 15
```

```
reg signed [3:0] signed_reg; //range -8 to 7
```

Value assignments

- Verilog allows a vector can be easily filled with
 - all zeros,
 - all Xs (unknown),
 - all Zs (high-impedance).
- Verilog does not provide a convenient mechanism to fill a vector with all ones.

```
module test;  
parameter SIZE = 64;  
reg [SIZE - 1 : 0] data;  
  
initial begin  
data = 0; // fills all bits of data with zero  
data = 'bz; // fills all bits of data with Z  
data = 'bx; // fills all bits of data with X  
  
data=64'hFFFFFFFFFFFFFFFF;  
end  
endmodule
```

Value assignments

- SystemVerilog allows to fill a vector with all ones.
- The syntax is to specify the value with which to fill each bit, preceded by an apostrophe ('):
 - '0 fills all bits on the left-hand side with 0
 - '1 fills all bits on the left-hand side with 1
 - 'z or 'Z fills all bits on the left-hand side with z

```
module test;  
parameter SIZE = 64;  
reg [SIZE-1:0] data;  
  
initial begin  
data = '1; // fills all bits of data with one  
data = '0; // fills all bits of data with zero  
data = 'z; // fills all bits of data with Z  
data = 'x; // fills all bits of data with X  
  
end  
endmodule
```

SV 2-state data types

Type	Sign	Size	Value Range
bit	unsigned	1-bit	0,1
byte	signed	8-bits	-128 to +127
short int	signed	16-bits	-32,768, +32,767
int	Signed	32-bits	-2,147,483,648 to +2,147,483,647
long int	Signed	64-bits	-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807
byte unsigned	unsigned	8-bits	0 to 255
short int unsigned	unsigned	16-bits	0 to 65,535
int unsigned	unsigned	32-bits	0 to 4,294,967,295
long int unsigned	unsigned	64-bits	0 to 18446744073709551615

```
module test;
```

```
bit b1;  
bit [7:0] b2;  
byte b3;
```

```
short int i1;  
int i2;  
long int i3;
```

```
byte unsigned u1;  
short int unsigned u2;  
int unsigned u3;  
long int unsigned u4;
```

```
initial begin  
b1=1; b2=4'b1010; b3 = -4;  
i1=16'habcd;  
i2=32'habcd_ffff;  
i3=64'haaaa_bbbb_cccc_dddd;  
u1=255; u2=65535;  
u3=4294967295;  
end  
endmodule
```

String Data type

- **string** : Ordered collection of characters.

str.compare return value:

if Return **value = 0** then it indicates **str1 is equal to str2**

Output:

```
str1=Lucid VLSI  
str2=Lucid VLSI  
str1 Matched str2
```

```
module test;  
string str1,str2;  
initial begin  
    str1 = "Lucid VLSI";  
    $display("str1=%s",str1);  
  
    str2=str1;//Copy  
    $display("str2=%s",str2);  
  
    if ( str1.compare(str2))  
        $display ("str1 Not Matched str2");  
    else  
        $display("str1 Matched str2");  
  
end  
  
endmodule
```


Continuous assignment to variables

Below one **not allowed** in verilog.

```
module test;
```

```
reg out;
```

```
assign out = sel ? Inp1 : inp2 ;
```

```
endmodule
```

```
vlog test.v      (verilog semantics)  
vlog -sv test.v (SV semantics)
```

Below one **allowed in SV**:

```
reg out; //logic out;
```

```
assign out = sel ? Inp1 : inp2 ; //OK
```

- A logic signal **can be used anywhere a net is used**, except that a **logic variable cannot be driven by multiple structural drivers**

A **logic** signal can be used anywhere a **net** is used

```
module logic_data_type(input rst,clk,d );  
  logic q1,q2,q3,q4,inp;  
  initial q1 = 0;           //Procedural assignment  
  
  assign q2 = inp;          //Continuous assignment  
  
  not not_inst (q3, inp); // q3 is driven by primitive  
  
  dff dff_inst (q4,clk,rst,d ); // q4 is driven by module  
endmodule
```

initial q1 = 0;
always@(inp)
q1 = inp;

initial q1 = 0;
assign q1 = 1;

assign q1 = 0;
assign q1 = 1;

initial q3 = 0;
not n1(q3,inp);

assign q3 = 0;
not n1(q3 ,inp);

not n1(q4 ,inp);
dff d1 (q4,d,clk,rst);

➤ logic variable cannot be driven by multiple structural drivers

User defined data types :: typedef

- Verilog does not allow users to define new data types.
- SystemVerilog provides a method to define new data types using **typedef**

```
typedef int unsigned uint;  
uint a, b;
```

```
typedef bit [31:0] bit32;  
bit32 data1, data2;
```

```
module dut (input uint inp1 .....
```

```
function uint add(input uint inp1, inp2)
```

```
function int unsigned add(input int unsigned i1, i2
```

```
function bit32 add(input bit32 inp1, inp2)
```

```
function bit[31:0] add(input bit[31:0] i1, i2
```

Enumerated types

```
enum {IDLE, WAIT, LOAD, DONE} states;  
initial begin  
    states=WAIT;  
    $display("name=%s value=%d ", states.name(), states);  
end
```

IDLE=0, WAIT = 1, LOAD=2, DONE=3

```
typedef enum {IDLE, WAIT, LOAD, DONE} states_type ;  
states_type curr_state,next_state;  
initial begin  
    curr_state=WAIT;  
    $display("value=%d name=%s ", curr_state, curr_state.name());  
end
```

module fsm (input states_type state)

module fsm (input enum {IDLE, WAIT, LOAD, DONE} states)

Packed and Unpacked Arrays

- ***Packed*** array of scalar bit types
- bit **[7:0]** *packed_arr* ;
- ***Unpacked*** array of bit types
- bit *unpacked_arr* **[7:0]** ;

Unpacked array storage

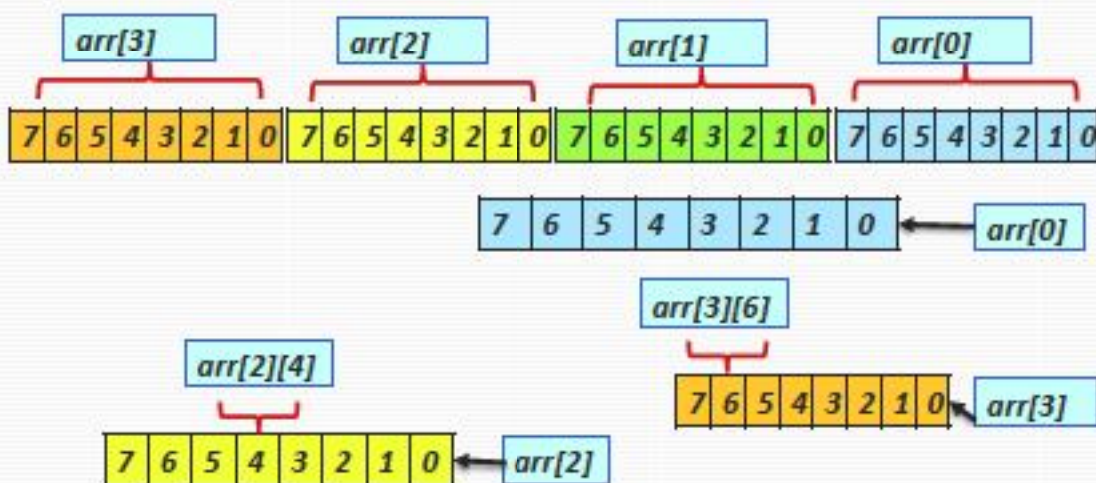
- bit [7:0] unpack_arr [4]; // Unpacked array

	4 th byte (31-24)	3 rd byte(23-16)	2 nd byte (15-8)	1 st byte(7-0)
unpack_arr[0]	Unused Space	Unused Space	Unused Space	7 6 5 4 3 2 1 0
unpack_arr[1]	Unused Space	Unused Space	Unused Space	7 6 5 4 3 2 1 0
unpack_arr[2]	Unused Space	Unused Space	Unused Space	7 6 5 4 3 2 1 0
unpack_arr[3]	Unused Space	Unused Space	Unused Space	7 6 5 4 3 2 1 0

Packed Arrays

- `bit [3:0] [7:0] arr; // 4 arr packed into 32-bits`
- It is stored as a contiguous set of bits with no unused space, unlike an unpacked array.

Packed array is accessed as array and as a single vector



Structures

- Structures allow multiple variables to be grouped together under a common name
- These variables can then be assigned independently, or the entire group can be assigned in a single statement.

```
module test;
struct {
  logic [31:0] data;
  bit [7:0] addr;
} packet;

  initial begin
    packet.data=32'hffff;
    packet.addr=8'd9;
    packet = '{data:20 , addr:13};
  end
endmodule
```

What is the output ?

```
struct {  
    logic [31:0] in1;  
    logic [31:0] in2;  
    opcode_t op_code;  
} packet;
```

```
typedef enum {NOP,ADD,SUB,MUL,DIV} opcode_t;
```

```
program test;  
    initial begin  
        packet.in1=10;  
        packet.in2=20;  
        packet.op_code=ADD;  
        $display("in1=%d in2=%d opcode=%s",packet.in1, packet.in2,packet.op_code.name());  
        packet = '{in1:23 , in2:44 , op_code:MUL};  
        $display(" packet = %p ",packet);  
    end  
endprogram
```

```
in1=10 in2=20 opcode=ADD
```

```
packet = '{in1:'h17, in2:'h2c, op_code:MUL}
```

Unpacked structure

```
typedef struct {  
    logic [7:0] sa; //MSB  
    logic [7:0] da;  
    logic [7:0] crc;  
    logic [7:0] payload; //LSB  
} unpacked_st ;  
  
program test;  
    unpacked_st pkt;  
    initial begin  
        pkt.sa=1; pkt.da=4; pkt.payload=8'hff;  
        pkt [7:0] = 40;  
    end  
endprogram
```

Slicing on unpacked struct is not allowed , it will result in compile error.
We should access only elements of struct , not individual fields.

Packed Structure

```
typedef struct packed {  
    logic [7:0] sa; //MSB  
    logic [7:0] da;  
    logic [7:0] crc;  
    logic [7:0] payload; //LSB  
} packed_st;  
  
program test;  
    packed_st pkt;  
    reg [31:0] din;  
  
    initial begin  
        pkt.sa=1;  
        pkt.da=4;  
        pkt.payload=8'hff;  
  
        pkt[7:0] = 40;  
        $display("pkt.payload=%0d",pkt.payload);  
  
        din = pkt;  
    end  
endprogram
```

All members of a packed structure must be integral values.
An integral value is a value that can be represented as a vector, such as *byte*, *int* and vectors created using *bit* or *logic* types.

Packed structure cannot contain real or shortreal variables, unpacked structures, unpacked unions, or unpacked arrays.

[31:24]sa	[23:16]da	[15:8]crc	[7:0]payload
-----------	-----------	-----------	--------------

pkt.payload=40

```
struct packed {  
    int arr [10];  
    real r;  
    short real k;  
    unpacked_st st;  
} packet;
```

Union

- A *union* is a data type that represents a single piece of storage that can be accessed using one of the named member data types.
- **Only one of the data types in the union can be used at a time.**
- Unions are useful when you frequently need to read and write a register in several different formats.
- If a value is stored using one union member, and read back from a different union member, then the value that is read is not defined, and may yield different results in different simulators.

Unpacked unions are not synthesizable.

```
module test;
  int k;

  union {
    int i; //32 bits
    real f; //64 bits
  } un;

  initial begin
    $display("un.f=%f un.i=%d",un.f,un.i);
    un.f = 1.234; //write
    k = un.i; //read
    $display("un.f=%f un.i=%d",un.f,un.i);
  end
endmodule
```

un.f=0.000000 un.i=0

un.f=1.234000 un.i=-927712936

tagged union

```
module test5;
```

```
union tagged {  
  int i;  
  real f;  
} un;
```

```
int int_v;  
real real_v;
```

```
initial begin  
  un = tagged f 1.234;  
  real_v = un.f;  
  $display("un.f=%0f ",un.f);  
  int_v = un.i;  
  un = tagged i 10;  
  int_v=un.i;  
  $display("un.i=%0d",un.i);  
end
```

A tagged union contains an implicit member that stores a tag, which represents the name of the last union member into which a value was stored. When a value is stored in a tagged union using a tagged expression, the implicit tag automatically stores information as to which member the value was written

```
un.f=1.234000
```

**** Error (suppressible): (vsim-8011) Union is tagged 'f', but is referenced as 'i'.**
Time: 0 ns Iteration: 0 Process: /test5/#INITIAL#19 File: tagged_union_ex.sv
Line: 25

packed union

- A union can be declared as packed.
- Only packed unions are synthesizable.
- In a packed union, the number of bits of each union member must be the same
- A packed union can only store integral values.
- A packed union cannot contain real or short real variables, unpacked structures, unpacked unions, or unpacked arrays.
- A packed union allows data to be written using one format and read back using a different format

```
union packed {  
    bit    [7:0] v1;  
    logic [7:0] v2;  
} un;
```

```
union packed {  
    bit    [31:0] v1;  
    logic [7:0] v2;  
} un;
```

**** Error: **** while parsing file testbench.sv(7)
**** at packed_union.sv(3): (vlog-13201)**
Packed union fields must all be the same width.

```
module test;  
  
    union packed {  
        bit    [7:0] v1;  
        logic [7:0] v2;  
    } un;  
  
    initial begin  
        un.v1=10;  
        $display("un.v1=%d un.v2=%d",un.v1,un.v2);  
  
        un.v2=22;  
        $display("un.v1=%d un.v2=%d",un.v1,un.v2);  
    end  
endmodule
```

Fixed-Size Arrays

- `int arr [3:0];` // 4 elements of ints
- `int c_style[4];` //4 elements of ints

Index	Value
0	data
1	data
2	data
3	data

```
reg [31:0] v;  
reg [31:0] arr[10];  
  
initial begin  
    v=arr[13]; //READ  
    arr[15]=99; //WRITE  
    addr=4'b10x1;  
    v=arr[addr];  
end
```

- What if your code accidentally tries to read from an **out-of-bounds address** ?
- SystemVerilog will return the default value for the array **element type**.
 - array of **4-state types**, such as logic, **will return X's**.
 - array of **2-state types**, such as int or bit, **will return 0**.
- This applies for all array types – fixed, dynamic, associative, and queue, also **if your address has an X or Z**

MDA: multi dimensional arrays

```
module test;  
  
int arr1 [4];  
int arr2 [4];  
int arr3 [4];  
int arr4 [4];  
int arr5 [4];  
int arr6 [4];  
  
int mda [6][4];  
  
initial begin  
  for (int i=0 ; i<4 ;i++) begin  
    arr1[i]=5;  
    $display("arr[%0d]=%0d ",i,arr1[i]);  
  end  
  
  for (int i=0 ; i<6 ;i++) //for each array  
    for (int j=0 ; j<4 ;j++) begin //for each element of array  
      mda [i] [j] = i+j;  
      $display("mda[%0d][%0d]=%0d ",i,j,mda[i][j]);  
    end  
  
end  
  
endmodule
```

Index	Value

Index	Value

Index	Value

Index	Value

Index	Value

Index	Value

Index	Value
0	data
1	data
2	data
3	data

Initializing an array

- Initialization at declaration

module test;

int arr1[5] = '{ 10,11,12,13,14 }'; // Initialize 5 elements

string str[4] = '{ "adi", "balu", "cnu", "sita" }';

int arr2 [5] = '{ 0:2,1:3, default:-1 }';

endmodule

arr2[0]=2;

arr2[1]=3;

arr2[2]= -1;

arr2[3]= -1;

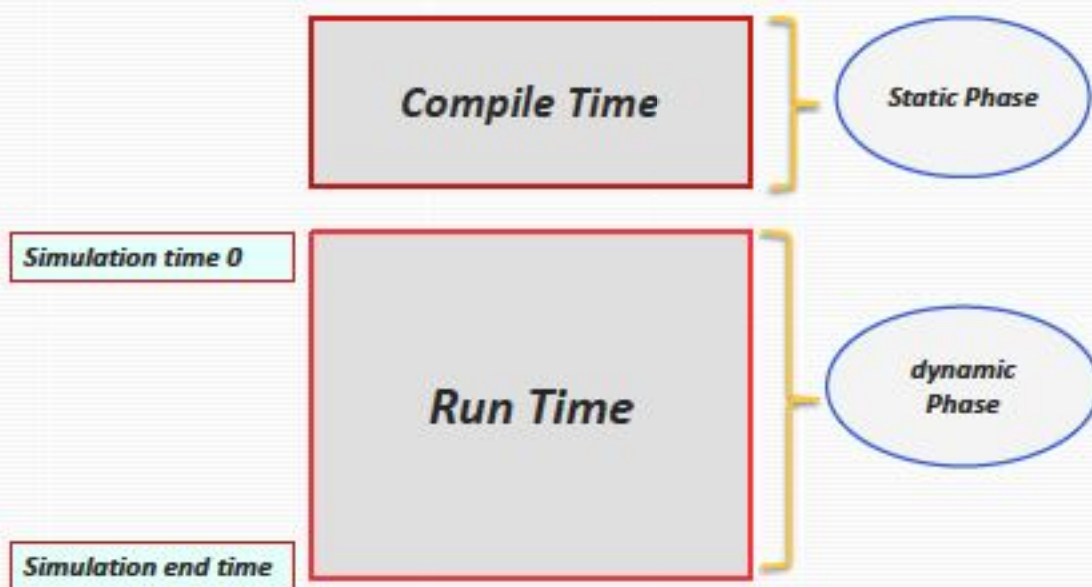
arr2[4]= -1;

```
int arr [5];
initial begin
  for (int i=0;i<5;i++)
    arr[i]=i; //initializing array
end
```

```
initial begin
  for(int i=0; i< arr1.size();i++)
    $display("arr1[%0d] = %0d ",i,arr1[i]);
end
```

OR

```
initial $display("arr1 = %p ",arr1);
```

Dynamic Arrays

```
module test;  
  int d1[], d2[];  
  initial begin  
    #100 d1 = new[4];  
    d1[0]=11;d1[1]=22;d1[2]=33;d1[3]=44;  
    d2 = d1;  
    #100 d1 = new[8] (d1);  
    #100 d1 = new[100];  
    $display("size=%0d",d1.size());  
    #100 d2.delete();  
  end  
endmodule
```

Associative arrays

- `data_type array_id [index_type];`
- `bit [7:0] array[int];` // integer index
- `int array_b[string];` // string indexing
 - `array.size();` //returns the number of entries
 - `array.num();` //returns the number of entries
 - `array.delete(5);` //deletes entry whose index 5
 - `array.exist(5);` // checks whether an element exists at the specified index
- `first(), last(), prev(), next()`
- `int record[string] = {"Peter":20, "Paul":22};`
- `record["Peter"] = 20;`
- `record["Paul"] = 22;`

Associative array example

```
module test;
bit [7:0] arr [int] //associative array

initial begin
arr[0]=10;
arr[4]=20;
arr[7]=30;
$display("size=%0d",arr.size());
arr.delete(4); //delete index 4
$display("size=%0d",arr.size());

if(arr.exists(7))//Is index 7 exists?
arr[8]=40;
else arr[7]=40;

$display("size=%0d",arr.size());
end
endmodule
```

Index	Element
0	10
4	20
7	30
8	40

Associative array example

```
module test;  
bit [7:0] arr[string]; //associative array  
  
initial begin  
    arr["ram"] = 10;  
    arr["Srinivas"] = 20;  
    arr["raja"] = 30;  
    $display("size=%0d", arr.size());  
    arr.delete("raja"); //delete index "raja"  
    $display("size=%0d", arr.size());  
  
    if(arr.exists("Srinivas"))  
        arr["ravi"] = 40;  
    else arr["Srinivas"] = 40;  
  
end  
endmodule
```

Index	Element
ram	10
srinivas	20
raja	30
ravi	40

Associative array methods

```
module test;  
int a[int];  
int k;  
initial begin  
a[4]=10;  
a[8]=20;  
a[12]=30;
```

Index	Element
4	10
8	20
12	30

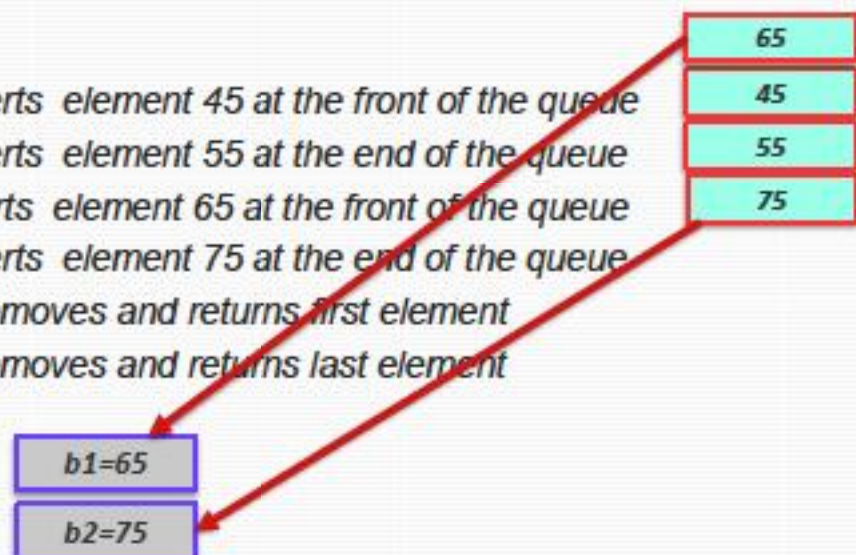
```
$display("Size of a=%0d",a.size());  
  
if(a.first(k))  
    $display("First index=%0d value=%0d",k,a[k]);  
  
if(a.first(k)) begin  
    do  
        $display("k=%0d value=%0d",k,a[k]);  
    while(a.next(k));  
end  
end  
endmodule
```

Queue

- Queue allows insertion and removal at the beginning or the end of the queue
- `int q1[$];` // A queue of integers
- `string names[$];` // A queue of strings
- `bit q2[$:255];` // A queue whose maximum size is 256 bits.
- `integer q[$] = { 3, 2, 7 };` // An initialized queue of integers
- The empty array literal `{}` is used to denote an empty queue.
`q={};` //Empty the queue.

Queue Methods

```
module test;  
bit [31:0] Q1[$]; //queue of elements each of 32-bits.  
bit [31:0] b1,b2;  
initial begin  
    Q1.push_front(45); //inserts element 45 at the front of the queue  
    Q1.push_back(55); //inserts element 55 at the end of the queue  
    Q1.push_front(65); //inserts element 65 at the front of the queue  
    Q1.push_back(75); //inserts element 75 at the end of the queue  
    b1 = Q1.pop_front(); //removes and returns first element  
    b2 = Q1.pop_back(); //removes and returns last element  
end  
endmodule
```



```
module tb;
```

```
int index= 1; int j;
```

```
int q[$] = {5,6,20,30,7};
```

```
initial begin
```

```
q.insert(index, 10);
```

```
q.delete(index);
```

```
q.push_front(9);
```

```
j = q.pop_back();
```

```
q.push_back(8);
```

```
j = q.pop_front();
```

```
q.delete();
```

```
end
```

```
endmodule
```

{5,10,6,20,30,7} insert 10 before 6
{5,6,20,30,7} Delete entry 1
{9,5,6,20,30,7} add 9 at front
{9,5,6,20,30} remove 7 at back : j=7
{9,5,6,20,30,8} add 8 at back
{5,6,20,30,8} remove 9 at front : j=9
{ } deletes entire queue

item represents a single *element of the array*.
find_index() returns the *indices of all the elements*
satisfying the given expression

```
module test;  
int d[] = '{9,1,12,3,4,4,32}';
```

```
int tq[$];
```

```
initial begin
```

```
tq = d.find_with (item > 3);
```



```
tq = d.find_index_with (item > 3);
```



```
tq = d.find_first_with (item > 42);
```



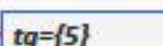
```
tq = d.find_first_index_with (item == 12);
```



```
tq = d.find_first_index_with (item == 4);
```



```
tq = d.find_last_index_with (item == 4);
```



```
end
```

```
endmodule
```