SYSTEM VERILOG

Topics

Data types
Arrays
Classes
Randomization
Program block

IPC
Interface
Coverage
Assertion
DPI

Data Types

Data types

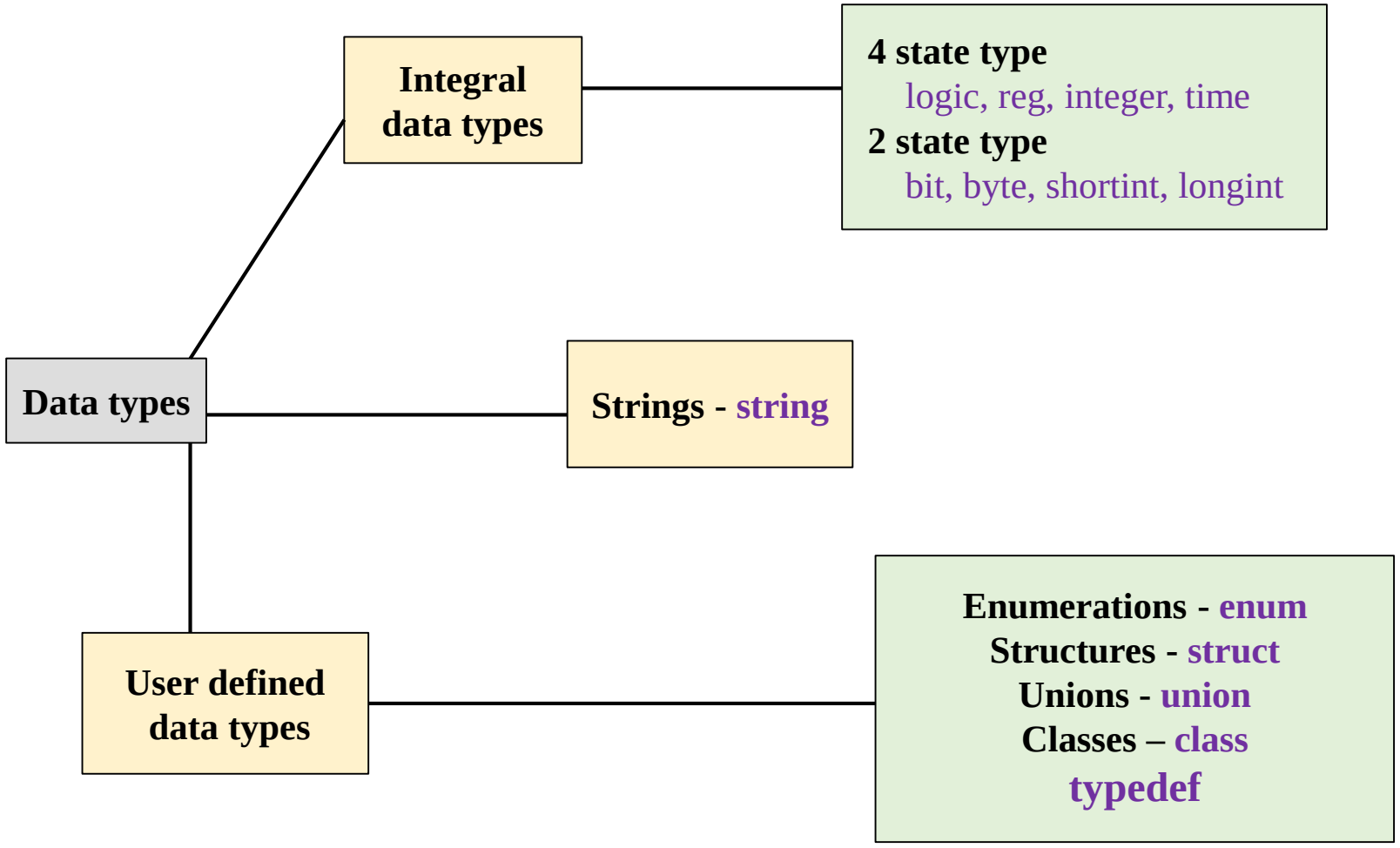
- 4-state & 2-state data types
- Signed, Unsigned
- Logic data type
- Strings
- User defined data type
- Enumeration
- Array, Structures and Union
- Casting
- Procedural Statements
- always blocks
- Operators
- Task and Function

Data types:

- An attribute which **specifies the type** of the value of the variable
- Has default values and rules to operate among different operators

VERILOG	SYSTEM VERILOG
• Four State Variable(0,1,X,Z) • Wire and Reg • No User Defined Data types	• Two State variable • Logic data type used instead of wire and reg • User Defined Data types • Casting (conversion of one data type to another)

Contd..


Data types
Integral data types
4 state type
logic, reg, integer, time
2 state type
bit, byte, shortint, longint
Strings - string
User defined data types
Enumerations - enum
Structures - struct
Unions - union
Classes – class
typedef

Integral data types

Data type	2-state/ 4-state	Signed/ unsigned	Default width	SV/ Verilog	Default Value
reg	4	unsigned	1	SV, Verilog	X
integer	4	signed	32	SV, Verilog	X
time	4	unsigned	64	SV, Verilog	X
bit	2	unsigned	1	SV	0
byte	2	signed	8	SV	0
logic	4	unsigned	1	SV	X
int	2	signed	32	SV	0
shortint	2	signed	16	SV	0
longint	2	Signed	64	SV	0

4-state

- 0(low),1(High),X(Unknown),Z(High Impedance)
- Note that reg can only be driven in procedural blocks like always and initial.
- while wire data types can only be driven in assign statements
- SystemVerilog introduces a new 4-state data type called logic that can be driven in both procedural blocks and continuous assign statements.
- But, a signal with more than one driver needs to be declared a net-type such as wire so that SystemVerilog can resolve the final value.

```
module tb;
  logic [3:0]  my_data;
  logic        en;
   initial begin
      $display ("my_data=%0h en=%0b", my_data, en);
              my_data = 4'hB;
 #1
$display ("my_data=%0h en=%0b", my_data, en);
         end
     //assign en = my_data[0];
endmodule
```

Without assign Statement

my_data=x en=x
my_data=b en=x

With assign statements

my_data=x en=x
my_data=b en=1

2-state

- SystemVerilog adds many new 2-state data types that can only store and have a value of either 0 or 1.
- This will aid in faster simulation, take less memory and are preferred in some design styles.
- When a 4-state value is converted to a 2-state value, any unknown or high-impedance bits shall be converted to zeros

```
module tb;
  bit       var_a;       // Declare a 1 bit variable of type "bit"
  bit [3:0] var_b;       // Declare a 4 bit variable of type "bit"
  initial begin
    $display ("Initial value var_a=%0b var_b=0x%0h", var_a, var_b);
    var_a = 1;
    var_b = 4'hF;
    $display ("New values    var_a=%0b var_b=0x%0h", var_a, var_b);
    var_b = 16'h481a;
    $display ("Truncated value: var_b=0x%0h", var_b);
    var_b = 4'b01zx;
    $display ("var_b = %b", var_b);
  end
endmodule
```

Result:

```
Initial value var_a=0 var_b=0x0
New values var_a=1 var_b=0xf
Truncated value: var_b=0xa
var_b = 0100
```

Signed:

- By default, integer variables are signed in nature and hence can hold both positive and negative

```
module tb;
 shortint      var_a;
 int           var_b;
 longint       var_c;
  initial begin
  $display ("Sizes var_a=%0d var_b=%0d var_c=%0d", $bits(var_a), $bits(var_b), $bits(var_c));
  #1 var_a = 'h7FFF;
     var_b = 'h7FFF_FFFF;
     var_c = 'h7FFF_FFFF_FFFF_FFFF;
  #1 var_a += 1;   // Value becomes 'h0
     var_b += 1;   // Value becomes 'h0
     var_c += 1;
 end
 initial
  $monitor ("var_a=%0d var_b=%0d var_c=%0d", var_a, var_b, var_c);
endmodule
```

Result:

Sizes var_a=16 var_b=32 var_c=64
var_a=0 var_b=0 var_c=0
var_a=32767 var_b=2147483647 var_c=9223372036854775807
var_a=-32768 var_b=-2147483648 var_c=-9223372036854775808

Unsigned

```
module tb;
  shortint unsigned     var_a;
  int      unsigned                    var_b;
  longint  unsigned      var_c;

  initial begin
    $display ("Sizes var_a=%0d var_b=%0d var_c=%0d", $bits(var_a), $bits(var_b), $bits(var_c));

     #1 var_a = 'hFFFF;
       var_b = 'hFFFF_FFFF;
       var_c = 'hFFFF_FFFF_FFFF_FFFF;


    #1 var_a += 1;   // Value becomes 'h0
       var_b += 1;   // Value becomes 'h0
       var_c += 1;
  end
  initial
    $monitor ("var_a=%0d var_b=%0d var_c=%0d", var_a, var_b, var_c);
endmodule
```

Result:

Sizes var_a=16 var_b=32 var_c=64
var_a=0 var_b=0 var_c=0
var_a=65535 var_b=4294967295 var_c=18446744073709551615
var_a=0 var_b=0 var_c=0

Logic data type

- Can be used instead of reg and wire
- So, logic can be used in continuous and procedural blocks
- Default value 'X’

```
module test;
logic y;

    assign y= 1;
    always@(*)
        y= 0;
endmodule
```

```
Error-[ICPSD] Illegal combination
of drivers
testbench.sv, 3
  Illegal combination of
structural and procedural drivers.
  Variable "y" is driven by an
invalid combination of structural

and procedural drivers.
```

//Continuous assignments

Procedural assignments

```
module Mod5_counter(input clk,
                    input rst,
                    output logic [2:0]cnt);
  logic [2:0] cnt_next;

  always@(posedge clk)
  begin
    if(rst)    cnt <= 0;
    else       cnt <= cnt_next;
  end
  assign  cnt_next = (cnt==4) ? 0 :cnt+1;
  endmodule
```

```
module tb;
  logic       clk,rst;
  logic [2:0] cnt;

  Mod5_counter(clk,
               rst,
               cnt);
  always #5 clk++;

  initial begin
   clk=0;rst=1;
    repeat(2)@(negedge clk);rst=0;
    repeat(5)@(posedge clk);$finish();
  end

endmodule
```

Strings

- Ordered collection of characters
- Varies size dynamically
- Strings are assigned to arrays also
- System Verilog provides special methods to work on strings

```
module tb;
 string name1="MOSCHIP";
string name2="SEMICONDUCTORS";
string name3;
int Num =3;
initial begin
  $display("\n%s \n%s",name1,name2);
 name2={name1,name2};
 name1="vLsI" ;
  $display("%s \n %s",name1,name2);
 name1={Num{name1}};
 name2 =name2.tolower();
 name3=name2.toupper();
  $display("%s \n %s \n %s",name1,name2,name3);
end
endmodule
```

Result

MOSCHIP
SEMICONDUCTORS
vLsI
MOSCHIPSEMICONDUCTORS
vLsIvLsIvLsI
moschipsemiconductors
MOSCHIPSEMICONDUCTORS

User defined data type

Typedef

- Allows to give a user defined name for existing data type
- Can be declared before the contents of data types have been declared

Ex:

```
typedef bit[31:0] utype;
        utype a , b, c; // a ,b and c are utype, utype is a datatype of bit[31:0]

initial begin
        a = 1;
        b = 2;
        c = a+b;
end
```

Enumeration

- An enum defines a set of named integral constants
- Default int type
- Other data types are explicitly defined
- If set names are not assigned a value explicitly then values are auto incremented
- Autoincrement is also done according to the values assigned to the names in a set

```
enum {a, b, c, d} alphabet;
        //then a=0,b=1,c=2,d=3

enum {a=0, b=6, c, d=8} alphabet;
// c=7 LEGAL
enum {a=0, b=7, c, d=8} alphabet;
// ILLEGAL ERROR
```

Contd..

- Any of the set name have 'X or 'Z with proper auto incrementing has syntax error

```
enum int {a, b='x, c='b01, d='b10,e} state, next; // error
enum int {a,b='x, c,d} state, next; // error  c=? d=? auto incrementing fails
```

- User Defined Data types with **enum**

- **last()** returns the value of last element in an enum
- **first**() returns the value of the first element in an enum
- **next**(i) returns the next value after the ith element in an enum
- **prev**(i) returns the next value after the ith element in an enum
- **num**() returns number of elements in an enum
- **name**() returns the corresponding name of the value in an enum

Example for enum methods

```
module enum_type;
        enum byte {Hyderabad, Bangalore, Kolkatta = 15, Mumbai, Chennai, Delhi = 100} city;
        initial
         begin
         city=city.last();
         $display("\n%s has the internal value of %d", city, city);
      if (city == city.last())
              begin
         $display("Done with the content");
        end
      city=city.next();
        $display("\n Total number if cities is %d\n", city.num());
        $display("%s ---- %s ----- %d \n", city, city.next(2), city.next(2));
        city = Delhi;
        $display("%s  ---- %s ---- %d\n",city, city.prev(1),city.prev(1));
        end
endmodule
```

Result

Delhi has the internal value of 100
Done with the content

Total number if cities is 6

Hyderabad ---- Kolkatta ----- 15

Delhi ---- Chennai ---- 17

enum methods

Example:

```
module typedef_enum;
      typedef enum byte { Hyderabad, Bangalore, Kolkatta = 15,Mumbai, Chennai, Delhi = 100} cities;
            initial
              begin
            cities city;
            city= city.first();
            forever
              begin
                $display("\n %s has the internal value of %d", city, city);
              if (city == city.last())
              break;
              city=city.next();
              end
          end
endmodule
```

Result

Hyderabad has the internal value of 0

Bangalore has the internal value of 1

Kolkatta has the internal value of 15

Mumbai has the internal value of 16

Chennai has the internal value of 17

Delhi has the internal value of 100

ARRAY

An *array* is a collection of variables, all of the same type, and accessed using the same name plus one or more indices. there are different types of arrays, few array declaration examples are given below.

```
reg[1:0] array1 [6];          //fixed size single dimension array
int array2 [5:0];             //fixed size single dimension array
bit[2:0] array3 [3:0][2:0];   //fixed size multi dimension array
```

```
module tb;
 typedef int uarr_4ele[4];
 uarr_4ele  inp1_array='{2,3,7,5};
 uarr_4ele inp2_array='{6,8,4,3};
 uarr_4ele Exp_sum_array;


 initial
 begin
       Exp_sum_array = Add_Array(inp1_array,inp2_array);
       $display("Exp_sum_array=%p",Exp_sum_array);
 end

 function uarr_4ele  Add_Array(input uarr_4ele array1 ,input uarr_4ele array2);
      foreach(array1[ele])
      begin
           Add_Array[ele] = array1[ele] + array2[ele];
      end
 endfunction:Add_Array


endmodule
```

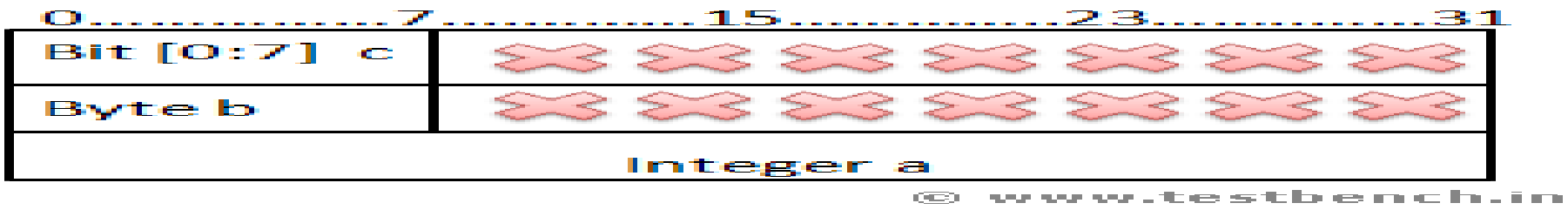
O/P

Exp_sum_array='{8, 11, 11, 8}

STRUCTURES

- The disadvantage of arrays is that all the elements stored in then are to be of the same data type.
- If we need to use a collection of different data types, it is not possible using an array.
- When we require using a collection of different data items of different data types we can use a structure.
- Structure is a method of packing data of different types.
- A structure is a convenient method of handling a group of related data items of different data types.

struct { int a; byte b; bit [7:0] c; } my_data_struct;



UNION

- *Unions like structure* contain members whose individual data types may differ from one another.
- However the members that compose a union all share the same storage area.
- A union allows us to treat the same space in memory as a number of different variables.
- That is a Union offers a way for a section of memory to be treated as a variable of one type on one occasion and as a different variable of a different type on another occasion.

union { int a; byte b; bit [7:0] c; } my_data;
memory allocation for the above defined struct "my_data_union

Class

- A class is a collection of data and a set of subroutines that operate on that data.
- The data in a class are referred to as class properties, and its subroutines are called methods.
- The properties and subroutines of class are called members.
- The object-oriented class extension allows objects to be created and destroyed dynamically.
- Class instances, or objects, can be passed around via object handles
- An Object can be declared as an argument with direction **input**, **output**, **inout**, or **ref**.
- In each case, the argument copied is the object handle, not the contents of the object.
- Any data type can be declared as a class member.
- A Class is declared using the **class** ... **endclass** keywords.

Ex:

```
class  Ethernet_Packet;
//Properties are address, data, and crc
    int address;
    bit [63:0] data;
    shortint crc;
// Methods are send and new

   function void Display();
        $display("data=%d
address=%d crc=%d",
data,address,crc);
   endfunction:Display
 endclass : Ethernet_Packet
```

Void type:

- The void data type represents non-existent data.
- It can be used as the return type of functions to indicate no return value.

Ex: **void'**(function_call());

```
module tb;
initial begin
  $display("Calling function with void");
    void'(sum(10,5));
    sum();
    current_time();
end



function int sum(int a=1,b=2);
  int c;
 c= a+b;
  $display("c=%d a=%0d b=%0d",c,a,b);
endfunction

  function void current_time();
    $display("current time is %0d",$time);
endfunction
endmodule
```

Output:

c= 15 a=10 b=5

c= 3 a=1 b=2

current time is 0

Casting

Casting are two types:
a)Static casting
b)Dynamic casting

Static Casting:

- A data type can be changed by using a cast (') operation.
- If the expression is assignment compatible with the casting type,

then the cast shall return the value that a variable of the casting type would hold after being assigned the expression.

Syntax: **casting_type ' (expression)**

Dynamic Casting :

$cast system task is to assign values to variables that might not ordinarily be valid because of differing data type
$cast can be called as either a task or a function.

Example: static casting and dynamic casting

```
typedef enum { Red, Green, Blue, Yellow, White, Black }Colors;
module tb;
Colors col,col1;
initial
                        begin
                          $cast( col, (2 + 2) );          //  Dynamic casting
                          col1 = Colors'(2+3);            //  Static casting
                          $display("\n\t%s\n",col);  //   White
                          $display("\t%s\n",col1);   //   Black

        if (!$cast(col,2))
                          $display("Casting does not happen as Source does not exists") ;
         else
        $display("casting done");  //displays
        end
endmodule
```

O/p:

White
Black
casting done

Procedural Statements

A *loop* is a piece of code that keeps executing over and over.
Different types of looping constructs in SystemVerilog are given in the table below.

Loops	Description
repeat	Repeats the given set of statements for a given number of times
while	Repeats the given set of statments as long as given condition is true
for	Similar to while loop, but more condense and popular form
do while	Repeats the given set of statements atleast once, and then loops as long as condition is true
foreach	Used mainly to iterate through all elements in an array
forever	Runs the given set of statements forever

for loop

for loop in **System Verilog has some added feature compared to the for loop in Verilog**

Inside for loop:

- Can declare a variable

```
for(int loop=0;loop<8;loop++)
```

- Can initialize more than one variable

```
for(int loop1=2,loop2=5; loop1<10;loop1++)
```

- More than one step assignment is possible

```
for(int loop_k=4,loopl2=9; loop_k>20;loop_k--,loop12++)
```

++,-- operator is allowed in SV

foreach loop

- Foreach is similar to for loop in terms of running the iteration based on the looping variable
- **But only the elements of arrays can be used in foreach loop**

```
int a[4];
foreach(a[i])
a[i]=i;
```

correct

```
int a=7,b[7];
foreach(a)
b[i]=i;
```

wrong

- Each loop variable corresponds to one of the dimensions of the array.

foreach(a[i])

One dimensional

foreach(a[i,j])

Two dimensional

i,j are automatically declared. The user no need to declare

Difference

for	foreach
int a,b[$]={4,5,6,7,8,9,10,11}; **for**(int i=0;i<8;i++) **begin** a=b.**pop_back**; $**display**("/n deleted elements=%d",a); **end** $**display**("/n remaining elements=%p",b); end Output: deleted elements=11 deleted elements=10 deleted elements=9 deleted elements=8 deleted elements=7 deleted elements=6 deleted elements=5 deleted elements=4 remaining elements={}	**int** a,b[$]={4,5,6,7,8,9,10,11}; **foreach**(b[i]) **begin** a=b.**pop_back**; $**display**("/n deleted elements=%d",a); **end** $**display**("/n remaining elements=%p",b); **end** Output: deleted elements=11 deleted elements=10 deleted elements=9 deleted elements=8 remaining elements={4,5,6,7}

pop_back and pop_front is not recommended inside foreach loop

Why?

forever

```
module tb;
  initial begin
    forever begin
      #5 $display("Moschip");
    end
  end

  initial
    #40 $finish;
endmodule
```

Result

```
Moschip
Moschip
Moschip
Moschip
Moschip
Moschip
Moschip
```

Repeat

```
module tb;
  initial begin
    repeat(2) begin
      #5 $display("Moschip Semiconductor");
    end
  end
endmodule
```

```
Moschip Semiconductor
Moschip Semiconductor
```

While

```
module tb;
bit clk;
  always #5 clk=~clk;

  initial begin

    bit[3:0] count;
    while(count<5) begin
      @(posedge clk);
    count++;
    $display("Moschip count = %0d",count);
    end
  end
endmodule
```

```
Moschip count = 1
Moschip count = 2
Moschip count = 3
Moschip count = 4
Moschip count = 5
```

do while

```
module tb;
bit clk;
  always #5 clk=~clk;
   initial begin
        bit[1:0] count;
    do begin
      @(posedge clk);
     $display("Moschip count = %0d",count);
      count++;
        end
    while(count<2);
  end
endmodule
```

```
Moschip count = 0
Moschip count = 1
```

Foreach

```
module tb_top;
   bit [7:0] array [8];
    initial begin
      foreach (array [index]) begin
        array[index] = index;
      end
      foreach (array [index]) begin
        $display ("array[%0d] = 0x%0d", index, array[index]);
      end
   end
endmodule
```

```
array[0] = 0x0
array[1] = 0x1
array[2] = 0x2
array[3] = 0x3
array[4] = 0x4
array[5] = 0x5
array[6] = 0x6
array[7] = 0x7
```

break

```
module tb;
    initial begin
       for (int i = 0 ; i < 4; i++) begin
        $display ("Iteration [%0d]", i);
         if (i == 2)
          break;
      end
    end
endmodule
```

```
Iteration [0]
Iteration [1]
Iteration [2]
```

Continue

```
module tb;
    initial begin
       for (int i = 0 ; i < 4; i++) begin
            if (i == 2)
          continue;
          $display ("Iteration [%0d]", i);
      end
    end
endmodule
```

```
Iteration [0]
Iteration [1]
Iteration [3]
```

Types of always blocks

- always @* is intended to infer a complete sensitivity list.
- However it does not infer a complete sensitivity list when the always @* block contains functions.
- To overcome that, there are different types of always block in SV:
  a)always_comb
  b)always_latch
  c)always_ff @( )

S.No	Verilog	SV
1	always@(*) b=5;	always_comb b=5;
	The always_comb in SV replace always@(*) in Verilog	
2	always@(*) if(en) q<=d	always_latch if(en) q<=d ;
	The always_latch in SV replace always@(*) in Verilog	
3	always@(posedge clk) b<=5	always_ff@(posedge clk) b<=5
	The always_ff @() in SV replace always@() in Verilog	

Contd..

- The main difference comes when **parameters or constant** used in always@(*), the block **doesn't trigger** at all.
- But in case of **always_comb** and **always_latch**, the block triggers at **0ns** irrespective of whatever is inside .

Example:

```
parameter g=7;
always@(*) begin
$display("g=%d",g);
end
```

Output:
Nothing is displayed

```
parameter g=7;
always_ff@(posdge clk)
begin
$display("g=%d",g);
end
```

Output: nothing at 0ns

```
parameter g=7;
always_latch begin // always_comb
$display( "g=%d",g);
end
```

Output:
g=7

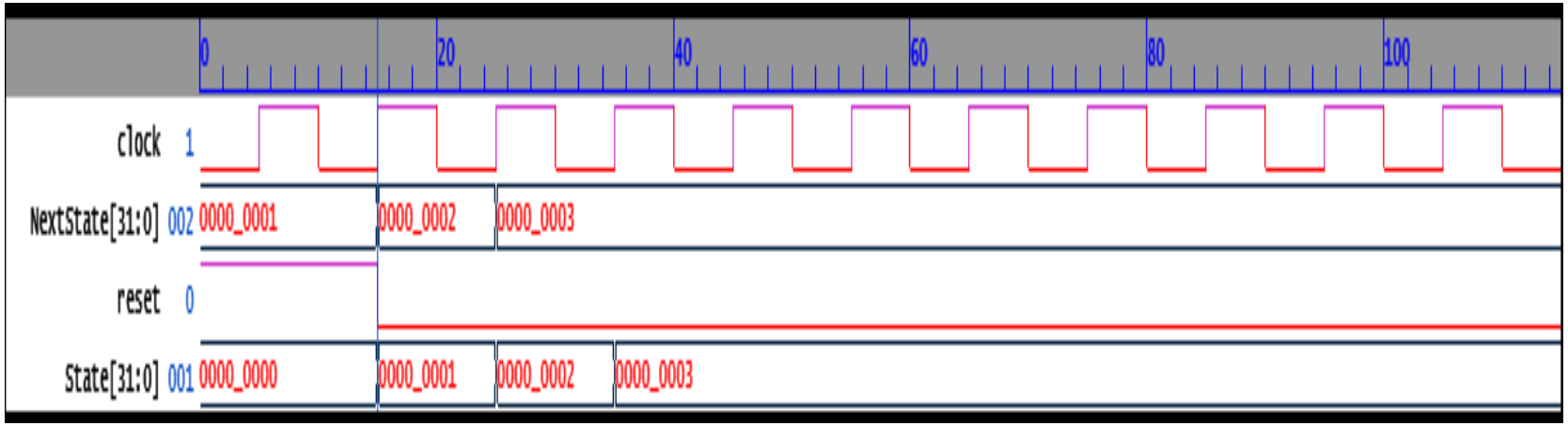
- The SV names always_ff, always_latch and always_comb have stricter criteria for when they are triggered, this means the chance for RTL to Gate level (post synthesis) **mismatch is reduced.**

```
module tb;
  reg clock;
  reg reset;
  enum{IDLE,LOAD, STORE, WAIT} State, NextState;
 // reg [1:0] State, NextState;
 // parameter IDLE=0,LOAD=1, STORE=2, WAIT=3;
always @(posedge clock,negedge reset)
  if (reset) State <= IDLE;
else
  State<=NextState;

  //always@(State)
  always_comb
  begin
    case(State)
      IDLE :NextState = LOAD;
      LOAD :NextState = STORE;
      STORE:NextState = WAIT;
    endcase
  end
```

```
always #5 clock = clock+1;

initial
  begin
      $dumpfile("dump.vcd"); $dumpvars;
      clock = 0;
      reset = 1;
      repeat(2)@(posedge clock);
      reset = 0;
    repeat(10)@(posedge clock);
    $finish;
  end
endmodule
```



Operators

- System Verilog supports all the operators used in Verilog
- In addition to Verilog operators, following operators are used in SV

Operator symbols	Example	Description
++	**int** i=3; i++;	i=4; (i=i+1)
--	**int** i=3; i--;	i=2; (i=i-1)
+=	**int** i=3; i+=5;	i=8; (i=i+ 5)
-=	**int** i=10; i-=7;	i=3; (i=i-7)
*=	**int** i=3; i*=8;	i=24; (i=i*8)
%=	**int** i=25; i%=2;	i=1; (i=i%2)
/=	**int** i=99; i/=3;	i=33; (i=i/3)
&=	**int** i=4'b1001,j=4'b0101; i&=j	Bit by bit and operation i=1001 j=0101(&) 0001 i=0001
|=	i |= j	Bit by bit or operation1101 is assigned to i
^=	i ^= j	Bit by bit Ex-Or operation1100 is assigned to i

Operator symbols	Example	Description
==?	a ==? b	a equals b, X and Z values in b act as wildcards
!=?	a !=? b	a does not equal b, X and Z values in b act as wildcards
>>=	**int** a=7'b0010111; a>>=1;	a=a>>1 (right shift by 1 and assigned to a) a=0001011
<<=	**int** a=7'b0010111; a<<=1;	a=a<<1(left shift by 1) a=0101110
>>>=	**int** a=5'b11000; a>>>=1; **int signed** a=5'b11000 a>>>=1;	'a' is right shifted by 1 a=01100(normal logical shifting) As it is signed, the vacated bit is filled with MSB. a=11100
<<<=	**int** a=5'b11000; a<<<=1; **int signed** a=5'b11000 a<<<=1;	'a' is left shifted by 1. a=10000(normal logical shifting) Even though it is signed, as the MSB is discarded, it performs normal logical left shift. a=10000

Task

A Task can contain a declaration of parameters, input arguments, output arguments, in-out arguments, registers, events and zero or more behavioral statements.

SystemVerilog allows,

- More capabilities for declaring task ports
- Multiple statements within task without requiring a begin...end or fork...join block
- Passing values by reference, value, names and position

```
module sv_task;
  int x;

  //task to add two integer numbers.
  task sum(input int a,b, output int c);
    c = a+b;
  endtask

  initial begin
    sum(1,5,x);
    $display("\tValue of x = %0d",x);
  end
endmodule
```

Value of x = 6

```
module sv_task;
  int x;

  //task to add two integer numbers.
  task sum;
    input int a,b;
    output int c;
    c = a+b;
  endtask

  initial begin
    sum(1,5,x);
    $display("\tValue of x = %0d",x);
  end
endmodule
```

Value of x = 6

Function

A *Function* can contain declarations of range, returned type, parameters, input arguments, registers and events.
A function without a range or return type declaration, returns a one-bit value.
Any expression can be used as a function call argument.
Functions cannot contain any time-controlled statements, and they cannot enable tasks.
Functions can return only one value.

SystemVerilog allows,

- More capabilities for declaring function ports
- Multiple statements within function without requiring a begin...end or fork...join block
- Returning from function before reaching the end of function
- Passing values by reference, value, names and position
- Default argument values
- Function output and inout ports
- Default arguments type is logic if no type has been specified.

```
module sv_function;
  int x;

  //function to add two integer numbers.
  function int sum;
    input int a,b;
    return a+b;
  endfunction

  initial begin
    x=sum(10,5);
    $display("\tValue of x = %0d",x);
  end
endmodule
```

```
Value of x = 15
```

```
module sv_function;
  int x;

  //void function to display current simulation time
  function void current_time;
    $display("\tCurrent simulation time is %0d",$time);
  endfunction

  initial begin
    #10;
    current_time();
    #20;
    current_time();
  end
endmodule
```

```
Current simulation time is 10
Current simulation time is 30
```

```
module argument_passing;
  int x,y,z;
  //function to add two integer numbers.
  function int sum(int x,y);
    x = x+y;
    return x+y;
  endfunction

  initial begin
    x = 20;
    y = 30;
    z = sum(x,y);
    $display("-----------------------------------------");
    $display("\tValue of x = %0d",x);
    $display("\tValue of y = %0d",y);
    $display("\tValue of z = %0d",z);
    $display("-----------------------------------------");
  end
endmodule
```

```
Value of x = 20
Value of y = 30
Value of z = 80
```

```
module argument_passing;
  int x,y,z;
     function int sum(ref int x,y);  //function to add two integer
numbers.
    x = x+y;
    return x+y;
  endfunction
   initial begin
    x = 20;
    y = 30;
    z = sum(x,y);
    $display("-----------------------------------------------------");
    $display("\tValue of x = %0d",x);
    $display("\tValue of y = %0d",y);
    $display("\tValue of z = %0d",z);
    $display("-----------------------------------------------------");
  end
endmodule
```

```
Value of x = 50
Value of y = 30
Value of z = 80
```

```
module argument_passing;
  int sig1,sig2;
  initial begin
    sig1 = 20;
    sig2 = 30;
    inc(sig1,sig2);
  end
  initial begin

        $display("\t$time=%0t Value of sig1 = %0d  \t sig2 = %0d",$time,sig1,sig2);
   #1;
        $display("\t$time=%0t Value of sig1 = %0d  \t sig2 = %0d",$time,sig1,sig2);
   #6;
        $display("\t$time=%0t Value of sig1 = %0d  \t sig2 = %0d",$time,sig1,sig2);
  end
  //task inc(inout int var1,var2);  //function to add two integer numbers.
  //task inc(ref int var1,var2);
  task inc(input int var1,var2);
     var1= var1+10;
     var2=var2+20;
      #2;
  endtask
endmodule
```

INPUT

$time=0 Value of sig1 = 20 sig2 = 30
$time=1 Value of sig1 = 20 sig2 = 30
$time=7 Value of sig1 = 20 sig2 = 30

REF

$time=0 Value of sig1 = 30 sig2 = 50
$time=1 Value of sig1 = 30 sig2 = 50
$time=7 Value of sig1 = 30 sig2 = 50

INOUT

$time=0 Value of sig1 = 20 sig2 = 30
$time=1 Value of sig1 = 20 sig2 = 30
$time=7 Value of sig1 = 30 sig2 = 50

Overview

System Verilog

Data types : 4-state & 2-state data types

- **Signed, Unsigned**
- **Logic data type**
- **Strings**
- **User defined data type**
- **Enumeration**
- **Structures & Union**
- **Casting**
- **Procedural Statements**
- **always blocks**
- **Operators**
- **Task and Function**

KNOWING IS NOT ENOUGH;
WE MUST APPLY.

WILLING IS NOT ENOUGH;
WE MUST DO

Bruce lee

THANK YOU

Date type Question

Justify 4 State and 2 state with program
What is the difference between reg, wire and Logic.
What is user define data type and write a program with one example.
What is Enum? Write the program using Enum typedef?
What is Array , Structure , Union and Class.
Explore the looping statements ?
Describe always block with examples.
Write the program using two variable to understand the operator