**LEARN VERILOG/SYSTEM VERILOG**

Contents

[1. Data Types: 2](#_Toc58139959)

[2. Simulation Modeling: 2](#_Toc58139960)

[3. Building Blocks: 2](#_Toc58139961)

[a. Module: 2](#_Toc58139962)

[b. Ports: 3](#_Toc58139963)

[c. Initial block: 3](#_Toc58139964)

[d. Always block: 3](#_Toc58139965)

[Pitfalls: 3](#_Toc58139966)

[e. Always\_ff, always\_latch, always\_comb: 4](#_Toc58139967)

[f. Combinational/Sequential Circuits: 4](#_Toc58139968)

[g. Operators: 5](#_Toc58139969)

[h. Parameters: 5](#_Toc58139970)

[i. Generate: 5](#_Toc58139971)

[4. Control Flow: 6](#_Toc58139972)

[a. Blocking /Non-Blocking statements: 6](#_Toc58139973)

[b. Assign statement: 6](#_Toc58139974)

[c. If, For, while, do while statements: 6](#_Toc58139975)

[d. Case statements – case, casex, casez: 6](#_Toc58139976)

[e. Forever loop: 7](#_Toc58139977)

[f. Function, Task: 7](#_Toc58139978)

[g. Strings, Enumerations: 9](#_Toc58139979)

[h. Arrays: 9](#_Toc58139980)

[I. Packed Arrays: 9](#_Toc58139981)

[II. Unpacked Arrays: 10](#_Toc58139982)

[III. Static Arrays: 10](#_Toc58139983)

[IV. Dynamic Arrays: 10](#_Toc58139984)

[V. Associative Arrays: 10](#_Toc58139985)

[VI. Queues: 10](#_Toc58139986)

[i. Structures, typedefs: 11](#_Toc58139987)

[j. Events: 11](#_Toc58139988)

[5. System tasks/functions: 11](#_Toc58139989)

[a. Display tasks: 11](#_Toc58139990)

[b. Math functions: 11](#_Toc58139991)

[c. Timescale: 12](#_Toc58139992)

[6. Verilog scheduling region: 13](#_Toc58139993)

[7. Processes: 14](#_Toc58139994)

[8. Program Block: 15](#_Toc58139995)

[9. Package: 15](#_Toc58139996)

[10. Interface: 15](#_Toc58139997)

[a. Modports: 16](#_Toc58139998)

[11. Clocking Blocks: 16](#_Toc58139999)

[12. Class: 16](#_Toc58140000)

[a. Inheritance: 16](#_Toc58140001)

[b. Polymorphism: 16](#_Toc58140002)

[c. Abstract/Pure Virtual class: 17](#_Toc58140003)

[d. Shallow & Deep Copy: 18](#_Toc58140004)

[e. Parameterized class: 19](#_Toc58140005)

[f. Local access qualifier: 20](#_Toc58140006)

[13. Constraints: 20](#_Toc58140007)

[14. Misc. constructs: 20](#_Toc58140008)

[15. Functional coverage: 20](#_Toc58140009)

[16. Assertions: 20](#_Toc58140010)

# Data Types:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Type** | **Size** | **2-state/4-state** | **Signed/Unsigned** | **C-Type** | **SV/Verilog** |
| wire | >=1 | 4 | Unsigned |  | Verilog/SV |
| Reg | >=1 | 4 | Unsigned |  | Verilog/SV |
| Integer | 32-bit | 4 | Signed |  | Verilog/SV |
| Real | 64 |  |  | Double | Verilog/SV |
| Time | 64 | 4 | Unsigned | Long int | Verilog/SV |
| Real Time | 64 |  |  | Double | Verilog/SV |
| Logic | >=1 | 4 | Unsigned |  | SV |
| Bit | >=1 | 2 | Unsigned |  | SV |
| Byte | 8 | 2 | Signed | Char | SV |
| Shortint | 16 | 2 | Signed | Short int | SV |
| Int | 32 | 2 | Signed | Int | SV |
| Longint | 64 | 2 | Signed | Long int | SV |
| Shortreal | 32 |  |  | Float | SV |

# Simulation Modeling:

* Gate Level AND(out,a,b);
* Behavior level out = a & b;
* Switch level nmos(out,d,cntrl);

# Building Blocks:

## Module:

module Adder();

…………………..

endmodule

Adder A0();

Adder A1();

## Ports:

Input, output, inout

Module Adder(

input a,

input b,

output out

);

Wire a0, b0;

Reg out0;

Adder A0(.a(a0), .b(b0), .out(out0));

## Initial block:

This is used to set initial values for simulation in test bench. Executes sequentially and non-synthesizable. You can have multiple initial blocks and all of them will be executed in parallel.

Initial being

a = ‘hACED;

#10;

$finish;

end

## Always block:

Always @ (event) being

…………………..

end

event: posedge; negedge; input or output

Will trigger when there is change in event. Can have multiple always blocks. If events are same in multiple blocks, all of them will trigger at same time and order cannot be guaranteed.

* Always make sure sensitive list is complete with statements being assigned in block.
* Always make sure all assignments happen irrespective of path in the block. Else latch will be implemented instead of flip flop.

### Pitfalls:

Always @(A) begin (In-correct)

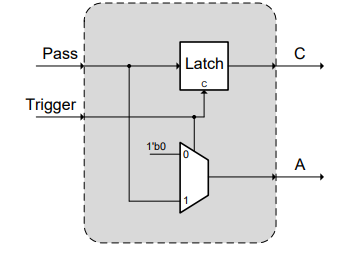
Out = A & B;

End

Always @(A, B) begin (Correct)

Out = A & B;

End

Always @ (\*) begin (In-correct)

A = 1’b0;

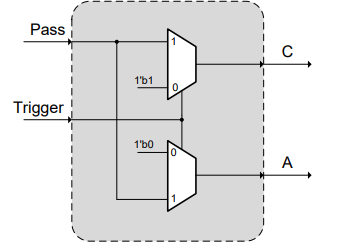
If (trigger) begin

A = pass;

C = pass;

end

end



Always @ (\*) begin (correct)

A = 1’b0;

C = 1’b0;

If (trigger) begin

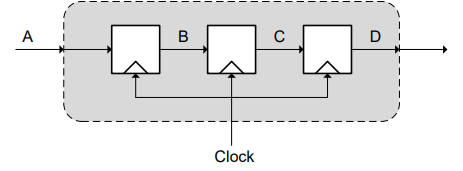
A = pass;

C = pass;

end

end

**Shift register:**

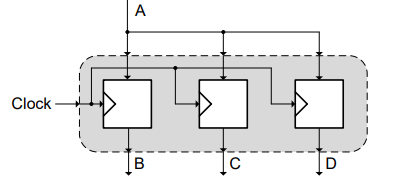
Always @ (\*) begin (correct)

B <= A;

C <= B;

D <= C;

End



Always @ (\*) begin (In-correct)

B = A;

C = B;

D = C;

end

## Always\_ff, always\_latch, always\_comb:

Always\_ff – uses only non-blocking statements and implements flip-flop.

Replaces always @(posedge).

Always\_latch – used to implement latch. Replaces always @ (\*)

Always\_comb - used to implement combinational circuits.

## Combinational/Sequential Circuits:

Combinational:

* Use Assign statements for combinational circuit. Cannot contain any storage elements like reg, clk etc.
* Always block can also be used for combinational.

Sequential:

* Should contain always blocks with clk as event.
* Storage elements must be present as reg/logic etc.

## Operators:

Most of the operators are same as C-style.

A === B; A should be equal to B including X, Z states

A!==B; A should not be equal to B including X, Z states

A == B; A should be equal to B, excluding X, Z states

## Parameters:

Global: parameter MSB = 7;

Module:

Module sum (a, b, out);

Parameter BUS\_WIDTH = 32, DATA\_WIDTH = 64;

Endmodule

(or)

Module sum #(parameter BUS\_WIDTH =32, parameter DATA\_WIDTH = 64) (a, b, out);

Endmodule

Module tb;

Sum s0 #(BUS\_WIDTH = 64, DATA\_WIDTH = 128) ([port\_list]);

Endmodule

## Generate:

Used to instantiate a module multiple times.

// Design for a half-adder

module ha ( input a, b,

output sum, cout);

assign sum = a ^ b;

assign cout = a & b;

endmodule

// A top level design that contains N instances of half adder

module my\_design

#(parameter N=4)

( input [N-1:0] a, b,

output [N-1:0] sum, cout);

// Declare a temporary loop variable to be used during

// generation and won't be available during simulation

genvar i;

// Generate for loop to instantiate N times

generate

for (i = 0; i < N; i = i + 1) begin

ha u0 (a[i], b[i], sum[i], cout[i]);

end

endgenerate

endmodule

module tb;

parameter N = 2;

// Instantiate top level design with N=2 so that it will have 2

// separate instances of half adders and both are given two separate inputs

my\_design #(.N(N)) md( .a(a), .b(b), .sum(sum), .cout(cout));

endmodule

# Control Flow:

## Blocking /Non-Blocking statements:

“=” Blocking statement and execute in sequence;

“<=” Non-blocking statement and execute in parallel.

a = #10 1’b1; a = 1 @ time 10

b = #20 1’b0; b = 0 @ time 30

c <= #10 ‘hACED; c = ACED @ time 10

d <= #5 ‘hDEAD; d = DEAD @ time 5

## Assign statement:

This can be used only for non-storage elements as wire, logic etc.

## If, For, while, do while statements:

Same as c-style.

## Case statements – case, casex, casez:

? – wildcard or don’t care

casex: X, Z are don’t cares

casez: Z is don’t care.

**Example:**

initial begin

#1 $display ("\n Driving 0");

sel = 0;

#1 $display ("\n Driving 1");

sel = 1;

#1 $display ("\n Driving x");

sel = 1'bx;

#1 $display ("\n Driving z");

sel = 1'bz;

#1 $finish;

end

always @ (sel)

case (sel)

1'b0 : $display("Normal : Logic 0 on sel");

1'b1 : $display("Normal : Logic 1 on sel");

1'bx : $display("Normal : Logic x on sel");

1'bz : $display("Normal : Logic z on sel");

endcase

always @ (sel)

casex (sel)

1'b0 : $display("CASEX : Logic 0 on sel");

1'b1 : $display("CASEX : Logic 1 on sel");

1'bx : $display("CASEX : Logic x on sel");

1'bz : $display("CASEX : Logic z on sel");

endcase

always @ (sel)

casez (sel)

1'b0 : $display("CASEZ : Logic 0 on sel");

1'b1 : $display("CASEZ : Logic 1 on sel");

1'bx : $display("CASEZ : Logic x on sel");

1'bz : $display("CASEZ : Logic z on sel");

endcase

**Result:**

Driving 0

Normal: Logic 0 on sel

CASEX: Logic 0 on sel

CASEZ: Logic 0 on sel

Driving 1

Normal: Logic 1 on sel

CASEX: Logic 1 on sel

CASEZ: Logic 1 on sel

Driving x

Normal : Logic x on sel

CASEX : Logic 0 on sel

CASEZ: Logic x on sel

Driving z

Normal: Logic z on sel

CASEX: Logic 0 on sel

CASEZ: Logic 0 on sel

## Forever loop:

Forever begin

@ (posedge clk) begin

………………

end

end

## Function, Task:

**Function [automatic] [return\_type] name ([port\_list]);**

**Endfunction**

Function [7:0] sum;

Input [7:0] a, b;

Begin

Sum = a +b;

end

endfunction

(or)

Function [7:0] sum (input [7:0] a, b);

Begin

Sum = a +b;

End

Endfunction

* Automatic is used for dynamically allocate rather than share between invocations.
* Functions cannot have time-controlled statements as #, @, wait, posedge, negedge
* Should have at least one input
* Cannot have non-blocking assignments, force-release, assign-deassign
* Cannot have output, inout, triggers
* Can call other functions but not task
* Can return a single value

Task [name];

Input [port\_list];

Output [port\_list];

Inout [port\_list];

Begin

…………

End

Endtask

Task [name] (input [port\_list], inout [port\_list], output [port\_list]]);

Begin

……….

End

Endtask

Task [name] ();

Begin

………..

End

Endtask

* Automatic is used for dynamically allocate rather than share between invocations.
* Can have time-controlled statements
* Can have/not have input, inout, output
* Cannot return a value, but can achieve using output
* Can call tasks or functions

Global tasks:

Task display ();

…………

Endtask

* Global task can be outside of module
* Disable can be used using “disable” keyword

## Strings, Enumerations:

String variable\_name = “Hello World”;

Enum {RED,YELLOW,GREEN} Traffic\_light;

Traffic\_light t1; // RED,YELLOW,GREEN={0,1,3}

Enum {RED,YELLOW=9,GREEN} Traffic\_light;

Traffic\_light t1; // RED,YELLOW,GREEN={0,9,10}

## Arrays:

***In a list of multi dimensions, the rightmost one varies most rapidly than the left most one. Packed dimension varies more rapidly than an unpacked.***

**Accessing:** Start from left and figure out number of elements.

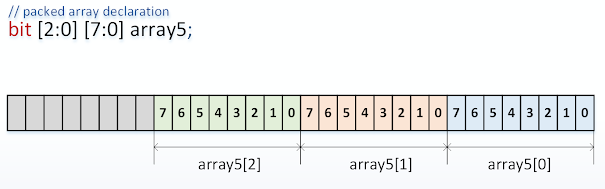
Bit [7:0] a1; //a1[i] = 1’b

Bit [2:0][7:0]a2; //a2[i] = 8’b; a2[i][j] = 1’b

Bit [1:0][2:0][7:0]; //a3[i] = 24’b; a3[i][j] = 8’b; a3[i][j][k] = 1’b1

### Packed Arrays:

Bit [2:0][7:0] packed\_array;



1 D packed array is also called vector.

***Example:***

foreach(a3[i,j]) begin

a3[i][j] = 'haa;

end

foreach(a3[i]) begin

$display("a3[%0d]=%b", i, a3[i]);

end

$display("-----------------------");

foreach(a3[i,j]) begin

$display("a3[%0d][%0d]=%b", i, j, a3[i][j]);

end

$display("-----------------------");

foreach(a3[i,j,k]) begin

$display("a3[%0d][%0d][%0d]=%b", i,j,k, a3[i][j][k]);

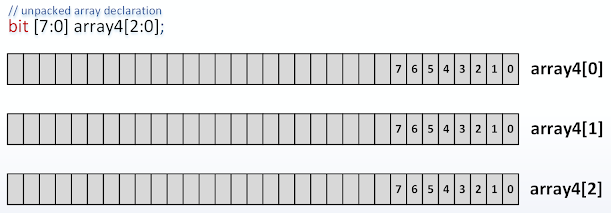
end

***Output:***

a3[1]=101010101010101010101010  
a3[0]=101010101010101010101010  
-----------------------  
a3[1][2]=10101010  
a3[1][1]=10101010  
a3[1][0]=10101010  
a3[0][2]=10101010  
a3[0][1]=10101010  
a3[0][0]=10101010  
-----------------------  
a3[1][2][7]=1  
a3[1][2][6]=0  
a3[1][2][5]=1  
a3[1][2][4]=0  
a3[1][2][3]=1  
a3[1][2][2]=0  
a3[1][2][1]=1  
a3[1][2][0]=0  
a3[1][1][7]=1  
a3[1][1][6]=0  
a3[1][1][5]=1  
a3[1][1][4]=0  
a3[1][1][3]=1  
a3[1][1][2]=0  
a3[1][1][1]=1  
a3[1][1][0]=0  
a3[1][0][7]=1  
a3[1][0][6]=0  
a3[1][0][5]=1  
a3[1][0][4]=0  
a3[1][0][3]=1  
a3[1][0][2]=0  
a3[1][0][1]=1  
a3[1][0][0]=0  
a3[0][2][7]=1  
a3[0][2][6]=0  
a3[0][2][5]=1  
a3[0][2][4]=0  
a3[0][2][3]=1  
a3[0][2][2]=0  
a3[0][2][1]=1  
a3[0][2][0]=0  
a3[0][1][7]=1  
a3[0][1][6]=0  
a3[0][1][5]=1  
a3[0][1][4]=0  
a3[0][1][3]=1  
a3[0][1][2]=0  
a3[0][1][1]=1  
a3[0][1][0]=0  
a3[0][0][7]=1  
a3[0][0][6]=0  
a3[0][0][5]=1  
a3[0][0][4]=0  
a3[0][0][3]=1  
a3[0][0][2]=0  
a3[0][0][1]=1  
a3[0][0][0]=0

### Unpacked Arrays:

Bit [7:0] unpacked\_array [2:0];



### Static Arrays:

Bit [7:0] Static\_Array;

Static\_Array = 8’hAE;

### Dynamic Arrays:

new(); delete() methods

Bit [7:0] Dynamic\_array[];

Dynamic\_array = new[10];

Growing size by 1: Dynamic\_array = new[Dynamic\_array.size()+1](Dynamic\_array);

### Associative Arrays:

Associative Arrays implements lookup table.

Int ar1[int]; ar1 = {1:22, 6:34};

Int ar2[string]; ar2={“Joe”:78, “Trump”:74};

String ar3[string]; ar3={“Joe”:”Win2020”, “Trump”:”Lost2020”};

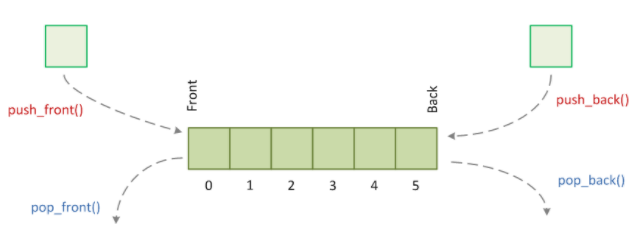
### Queues:

One-dimensional unpacked array that grow and shrinks automatically.

Int Queue[$];

Queue.push\_back(10);

Int data = Queue.pop\_front();



## Structures, typedefs:

Struct {

String name,

Int id

} Employee;

Typedef enum {RED, YELLOW, GREEN} Lights;

## Events:

Static object handle to synchronize between two concurrent active processes.

Trigger: -> or ->>

Wait: @ or wait(<even\_name>.triggered)

# System tasks/functions:

## Display tasks:

* $display: Appends new line
* $strobe: Final value of variable at end of current delta time-step
* $write: Doesn’t append new line
* $monitor: Prints whenever variable changes

|  |  |
| --- | --- |
| %h, %H | Display in hexadecimal format |
| %d, %D | Display in decimal format |
| %b, %B | Display in binary format |
| %m, %M | Display hierarchical name |
| %s, %S | Display as a string |
| %t, %T | Display in time format |
| %f, %F | Display 'real' in a decimal format |
| %e, %E | Display 'real' in an exponential format |

## Math functions:

|  |  |
| --- | --- |
| $ln(x) | Natural logarithm log(x) |
| $log10(x) | Decimal Logarithm log10(x) |
| exp(x) | Exponential of x (ex) where e=2.718281828... |
| sqrt(x) | Square root of x |
| $pow(x, y) | xy |
| $floor(x) | Floor x |
| $ceil(x) | Ceiling x |
| $sin(x) | Sine of x where x is in radians |
| $cos(x) | Cosine of x where x is in radians |
| $tan(x) | Tangent of x where x is in radians |
| $asin(x) | Arc-Sine of x |
| $acos(x) | Arc-Cosine of x |
| $atan(x) | Arc-tangent of x |
| $atan2(x, y) | Arc-tangent of x/y |
| $hypot(x, y) | Hypotenuse of x and y : sqrt(xx + yy) |
| $sinh(x) | Hyperbolic Sine of x |
| $cosh(x) | Hyperbolic-Cosine of x |
| $tanh(x) | Hyperbolic-Tangent of x |
| $asinh(x) | Arc-hyperbolic Sine of x |
| $acosh(x) | Arc-hyperbolic Cosine of x |
| $atanh(x) | Arc-hyperbolic tangent of x |

## Timescale:

**`timescale <time\_unit/time\_precision>**

initial begin

val <= 0; // Initialize the signal to 0 at time 0 units

#1 $display ("T=%0t at time #1", $realtime); // Advance by 1-time unit, disp msg and toggle val

val <= 1;

#0.49 $display ("T=%0t at time #0.49", $realtime); // Advance by 0.49-time unit and toggle val

val <= 0;

#0.50 $display ("T=%0t at time #0.50", $realtime); // Advance by 0.50-time unit and toggle val

val <= 1;

#0.51 $display ("T=%0t at time #0.51", $realtime); // Advance by 0.51-time unit and toggle val

val <= 0;

#5 $display ("T=%0t End of sim", $realtime); // Let sim run for another 5 time units and exit

End

**`timescale 1ns/1ns**

T=1 At time #1

T=1 At time #0.49

T=2 At time #0.50

T=3 At time #0.51

T=8 End of simulation

**`timescale 1ns/1ps**

T=1000 At time #1

T=1490 At time #0.49

T=1990 At time #0.50

T=2500 At time #0.51

T=7500 End of simulation

**`timescale 10ns/1ns**

T=10 At time #1

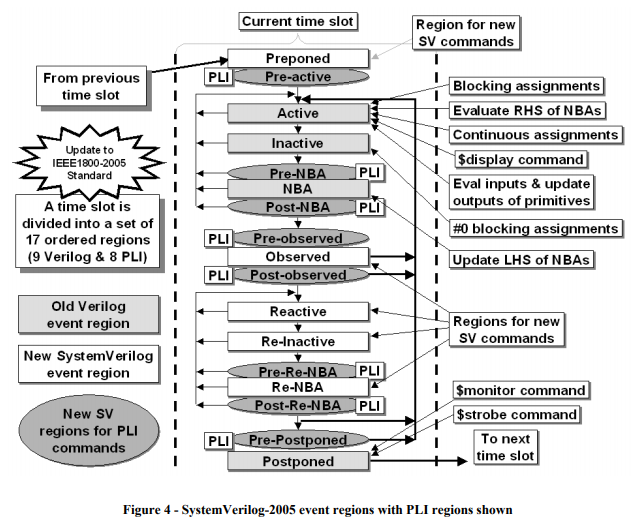
T=15 At time #0.49

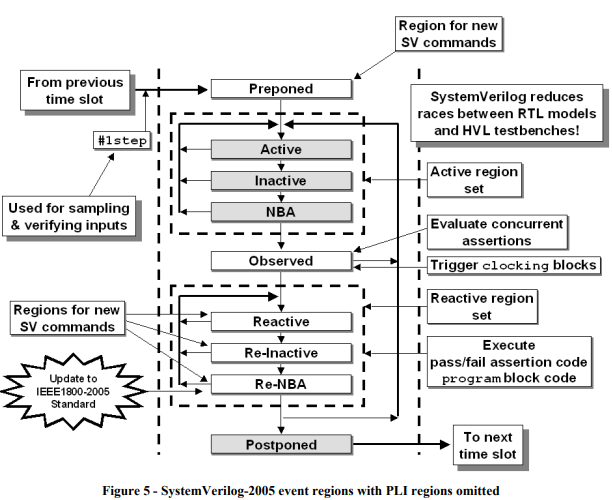
T=20 At time #0.50

T=25 At time #0.51

T=75 End of simulation

# Verilog scheduling region:





# Processes:

System Verilog can have 3 types of threads. Threads are spawned using fork block.

Join Join\_any join\_none

Fork

Initial begin

Thread1……………..

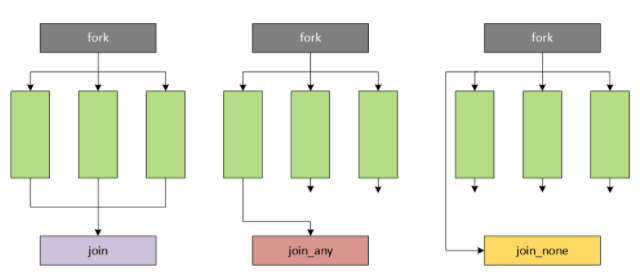
End

Initial begin

Thread2……………..

end

Join\_any



Fork can be disabled by ‘disable fork’.

# Program Block:

Program block are introduced to provide entry point for testbench execution, to have tasks & functions in one place and avoid race condition between dut and test bench.

* Program block can be inside a module.
* Program block cannot contain always block, modules, interfaces or other programs.
* A module cannot call tasks/functions inside a program block, but the other way is possible.
* Program block cannot have non-blocking assignments.

program testbench(input bit [7:0] addr);

initial begin

$display("\t Addr = %0d",addr);

end

endprogram

# Package:

Package can have collection of reusable parameters, structures, functions and tasks.

Package my\_pkg;

Enum

Structuts

Functions/tasks

Endpackage

// Importing package

Import my\_pkg::\*;

# Interface:

interface intf (input clk);

logic read, enable,

logic [7:0] addr,data;

modport dut (input read,enable,addr,output data);

modport tb (output read,enable,addr,input data);

endinterface :intf

module Dut (intf.dut dut\_if); // declaring the interface with modport

assign dut\_if.data = temp1 ? temp2 : temp3 ; // using the signal in interface

endmodule

module Testbench(intf.tb tb\_if);

.....

.....

Endmodule

module top();

logic clk;

intf bus\_if(clk); // interface instantiation

Dut d(bus\_if); // Pass the interface

Testbench TB (bus\_if); // Pass the interface

Endmodule

## Modports:

Specified direction of signals.

# Clocking Blocks:

Clocking blocks are used to sample inputs before and outputs after clocking event. This helps to avoid race conditions.

clocking cb @(posedge clk);

default input #1ps output #2;

input from\_Dut;

output to\_Dut;

endclocking

# Class:

Most of the concepts are similar to C++.

Create a new instance using new keyword. Access parent method using super keyword.

## Inheritance:

Class Base;

Bit [3:0] a;

Bit sel;

Endclass

Class Derived extends base;

function display();

$display(“a:%b, sel:%b”,a,sel);

endunction

endclass

module tb;

Derived d1;

Initial begin

d1.a = 7;

d1.sel = 1;

d1.display();

end

endmodule

## Polymorphism:

Class packet;

Bit [2:0] type;

Byte data;

Function new(bit[2:0]type=0, byte data=0);

this.type = type;

this.Data = data;

endfunction

virtual Function display();

$display(“Print packet type:%d, data=%b”, this.type, this.data);

endfunction

Endlclass

Class header extends packet;

Function new(bit[2:0]type=3, byte data);

This.type=type;

This.data=data;

Endfunction

Function display();

$display(“Print header type:%d, data=%b”, this.type, this.data);

endfunction

Endclass

Module tb;

Packet pkt;

Header hd;

Initial begin

hd = new(3,8’hAB);

hd.display(); // header display executes

pkt = hd;

pkt.display(); //As display func is virtual, header display executes

end

endmodule

## Abstract/Pure Virtual class:

virtual class BaseClass; // Abstract class

int data;

pure virtual function int getData(); // pure virtual method. Need to be

endclass // implemented by derived class

class ChildClass extends BaseClass;

virtual function int getData();

data = 32'hcafe\_cafe;

return data;

endfunction

endclass

module tb;

ChildClass child;

initial begin

child = new();

$display ("data = 0x%0h", child.getData());

end

endmodule

## Shallow & Deep Copy:

Shallow doesn’t copy nested objects, instead handles are copies and shares same data.

Whereas deep copy, copies all the data including nested objects (Note: you need to create your own copy method)

Example:

class Header;

int id;

function new (int id);

this.id = id;

endfunction

function showId();

$display ("id=0x%0d", id);

endfunction

endclass

class Packet;

int addr;

int data;

Header hdr;

function new (int addr, int data, int id);

hdr = new (id);

this.addr = addr;

this.data = data;

endfunction

function copy (Packet p);

this.addr = p.addr;

this.data = p.data;

this.hdr.id = p.hdr.id;

endfunction

function display (string name);

$display ("[%s] addr=0x%0h data=0x%0h id=%0d", name, addr, data, hdr.id);

endfunction

endclass

//shallow:

packet p1,p2;

p1 = new(32'hface\_cafe, 12345678, 26);

p2 = new p1;

//changing p1

p1.addr = 32'habcd\_ef12;

p1.data = 32'h5a5a\_5a5a;

p1.hdr.id = 17;

//prints:

[p1] addr=0xfacecafe data=0x12345678 id=26

[p2] addr=0xfacecafe data=0x12345678 id=26

[p1] addr=0xabcdef12 data=0x5a5a5a5a id=17

[p2] addr=0xfacecafe data=0x12345678 id=17 (Note: id has changed to p2 aswell)

//Deep:

packet p1,p2;

p1 = new(32'hface\_cafe, 12345678, 26);

p2 = new(1,2,3);

p2.copy(p1);

//changing p1

p1.addr = 32'habcd\_ef12;

p1.data = 32'h5a5a\_5a5a;

p1.hdr.id = 17;

//prints:

[p1] addr=0xfacecafe data=0x12345678 id=26

[p2] addr=0xfacecafe data=0x12345678 id=26

[p1] addr=0xabcdef12 data=0x5a5a5a5a id=17

[p2] addr=0xfacecafe data=0x12345678 id=26 (Note: id has not changed to p2)

## Parameterized class:

Similar to parameterized module

// Declaration

class <class\_name> #(parameters);

class Trans #(addr=16);

// override class parameter

<class\_name> #(prameters) <name\_of\_instance>

Trans #(.addr(32)) t1;

Templated class:

// Declaration

Class Trans #(type T = int);

T item;

endclass

// override class template

<class\_name> #(template) <name\_of\_instance>

Trans #(bit[3:0]) t1;

## Local access qualifier:

Local members are only accessible to its own class but not to child class.

# Constraints:

# Misc. constructs:

# Functional coverage:

# Assertions: