**System Verilog**

**Polymorphism: by using parent class handle, we can override baseclass method by child class method. There are 3 rules to satisfy polymorphism**

1. **Base class method should be virtual**
2. **Base class and derived class method should be same**
3. **Base class = derived class legal**
4. **Derived class = base class (illeg al) we can use $cast means check the type compatibles of both classes that derived class extended from base class or not**

class base\_class;

virtual function void mahesh();

$display("inside base class");

endfunction

endclass

class extend\_class1 extends base\_class;

function void mahesh();

$display("inside extend1 class");

endfunction

endclass

class extend\_class2 extends base\_class;

function void mahesh();

$display("inside extend2 class");

endfunction

endclass

module tb;

initial

begin

base\_class b;

extend\_class1 e1=new;

extend\_class2 e2=new;

b=e1;

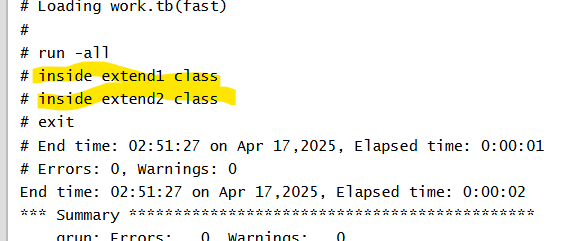
b.mahesh();

b=e2;

b.mahesh();

end

endmodule



**Inheritance: allows us extended class from base class. Derived class inherits new properties and methods and existing behavior of base class properties and methods by derived class handle**

class base\_class;

bit [2:0] a;

endclass

class extend\_class1 extends base\_class;

bit [3:0] b;

int c;

endclass

module tb;

initial

begin

extend\_class1 e1=new;

e1.a= 3'd4;

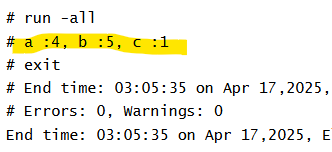
e1.b=4'd5;

e1.c= 1;

$display("randomized data :%p, randomized data :%p, randomized data :%p", e1.a,e1.b,e1.c);

end

endmodule

****

**THREADS**

**3 types of Threads: fork-join, fork-join\_any and fork-join\_none**

**fork join 🡪 it will execute parallel threads. In below example, we have 3 threads in fork join**

module tb;

initial begin

$display ("[%0t] Main Thread: Fork join going to start", $time);

fork

// Thread 1

#30 $display ("[%0t] Thread1 finished", $time);

// Thread 2

begin

#5 $display ("[%0t] Thread2 ...", $time);

#10 $display ("[%0t] Thread2 finished", $time);

end

// Thread 3

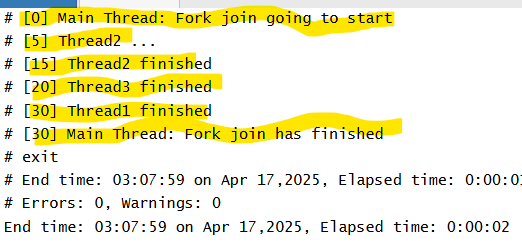
#20 $display ("[%0t] Thread3 finished", $time);

join

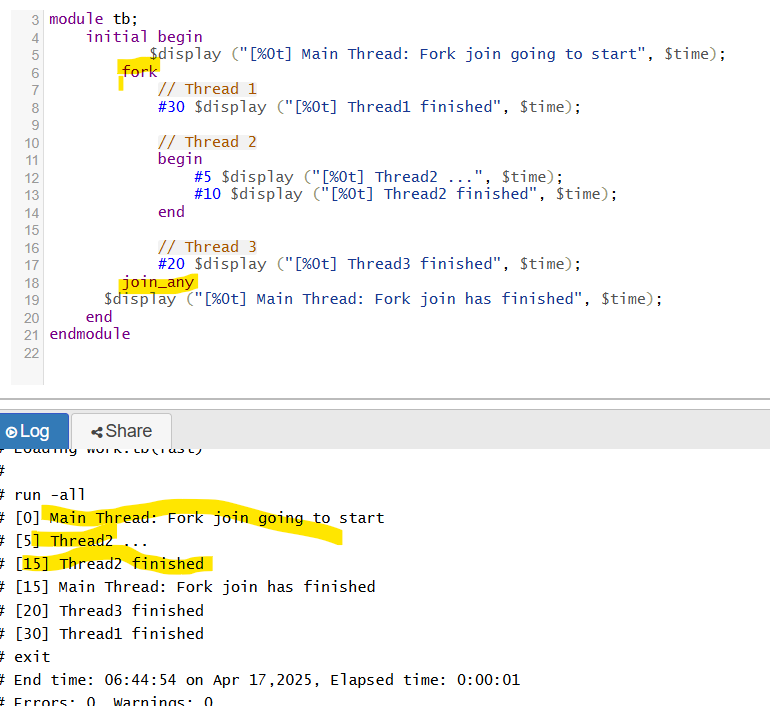
$display ("[%0t] Main Thread: Fork join has finished", $time);

end

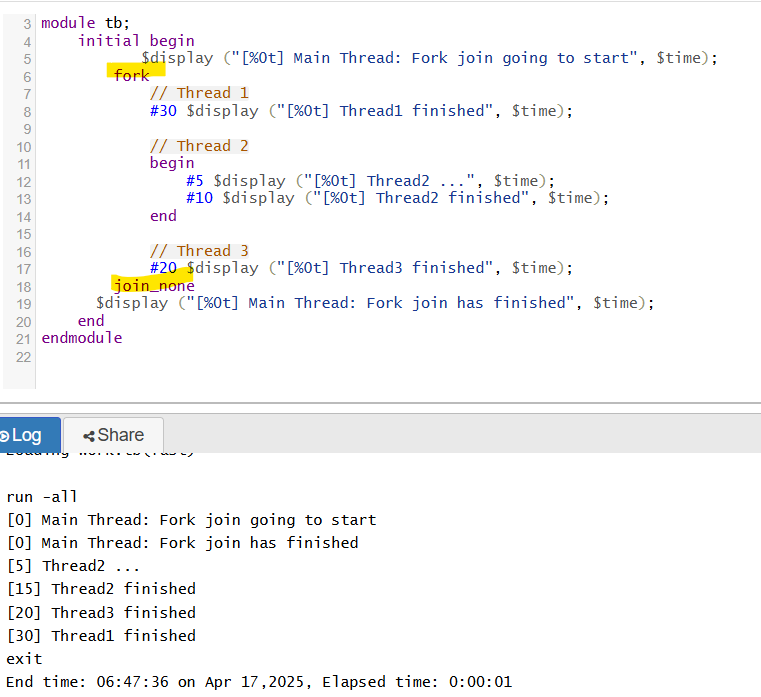
endmodule



**Fork-join\_any: any one process will execute and come out of fork join\_any block and will start parent thread.**

****

**Fork-join\_none : It skips all child processes inside fork and execute parent thread first**

****

**Semaphore**

**It is a built-in class and used for synchronous process. For semaphore we have new method and get, put methods. Its like bucket contains number of keys to access the process. So that it will execute synchronously.**

**In below code, we have three processes and 2 keys are present. Based on delays, process will execute by using get and put methods.**

module tb();

semaphore sem;

task display();

sem.get(1);

#5;

$display("process 1",$time);

sem.put(1);

endtask

task display1();

sem.get(1);

#4

$display("process 2",$time);

sem.put(1);

endtask

task display2();

sem.get(1);

#2

$display("process 3",$time);

sem.put(1);

endtask

initial

begin

sem = new(2);

fork

display();

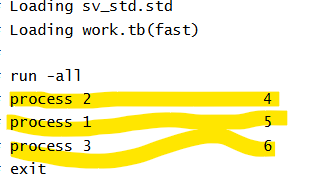
display1();

display2();

join

end

endmodule

****

**Shallow copy: nested class. it copies only parent properties not the sub class properties. Allocates same memory.**

**If we try to modify the sub class property using second handle, it will reflect previous handle also.**

class dummy;

int a;

endclass

class main;

dummy d\_h = new();

int d;

endclass

module tb;

initial

begin

main m1, m2;

m1 = new;

m1.d = 10;

m1.d\_h.a=11;

$display("m1 --> a = %0d, d = %0d", m1.d, m1.d\_h.a);

//shallow copy

m2 = new m1;

$display("before modification");

$display("m2 --> a = %0d, d = %0d", m2.d, m2.d\_h.a);

m2.d\_h.a =20;

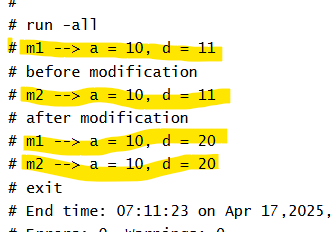
$display("after modification");

$display("m1 --> a = %0d, d = %0d", m1.d, m1.d\_h.a);

$display("m2 --> a = %0d, d = %0d", m2.d, m2.d\_h.a);

end

endmodule

****

**Deep copy: copies both classes of properties and allocates separate memory**

class dummy;

int a;

function dummy copy();

copy = new();

copy.a = this.a;

return copy;

endfunction

endclass

class main;

dummy d\_h = new();

int d;

function main copy();

copy = new;

copy.d = this.d;

copy.d\_h = this.d\_h.copy;

return copy;

endfunction

endclass

module tb;

initial

begin

main m1, m2;

m1 = new;

m1.d = 10;

m1.d\_h.a=11;

$display("a = %0d, d = %0d", m1.d, m1.d\_h.a);

//deep copy

m2 = m1.copy();

$display("a = %0d, d = %0d", m2.d, m2.d\_h.a);

$display("modification");

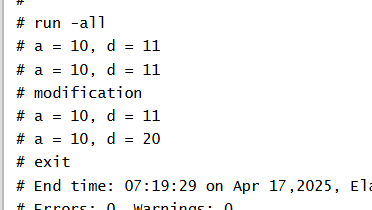
m2.d\_h.a =20;

$display("a = %0d, d = %0d", m1.d, m1.d\_h.a);

$display("a = %0d, d = %0d", m2.d, m2.d\_h.a);

end

endmodule



Assertions: to check trueness of the design signal.to validate the behavior of design according to specification. They will help to debug more errors

1. Immediate assertions-> inside procedural statements
2. Concurrent assertions -> depends on clock signal

module assertion();

logic clk;

bit req;

bit ack;

bit done;

initial

begin

clk = 0;

forever #5 clk = ~clk;

end

initial

begin

req =0;

repeat(1)

@(posedge clk);

req =1;

end

initial

begin

ack =0;

repeat(2)

@(posedge clk);

ack =1;

end

initial

begin

done =0;

repeat(3)

@(posedge clk);

done =1;

end

//assertion

**property p1;**

**@(posedge clk) $rose(req) |=> ack ##1 done;**

**endproperty**

**assert property(p1)**

**$display($time, "assertion passed");**

**else**

**$display($time, "assertion failed");**

initial

begin

$dumpfile("dump.vcd");

$dumpvars(1,assertion);

end

initial

begin

#100;

$finish;

end

endmodule

**constraints:**

constraint name implies to satisfy conditions and range of values for randomization purpose. We will give limit the variables how much range it should be randomize. We have several types of constraints.

**Soft constraint** -> by using soft keyword, it will override by another constraint.

**Inline constraint** -> inside initial begin after created object, keyword called **randomize with**

**Distribution constraint**-> applies to weightage of some values in the variable. [2:0]a , constraint c1 { dist [2,3,4] := 1}

**Functional coverage**: this will do by verification engineers. it defines how much design functionality has been covered in percentage.

It checks quality of design functionality (DV)

Covergroup cg;

// Variables

Coverpoint: also called as variables that are interested to cover.

Cross: cross variables are nothing but we will cover how many bins that multiply both coverpoints.

Coverpoint a

Coverpoint b

Cross aXb

Endcoverage

**Code coverage**: checks the quality of testcases. Types of coverage

1. Branch
2. Fsm
3. Toggle
4. Statement

We will cover all types of coverage to achieve 100% in code coverage. This is done in a RTL design.