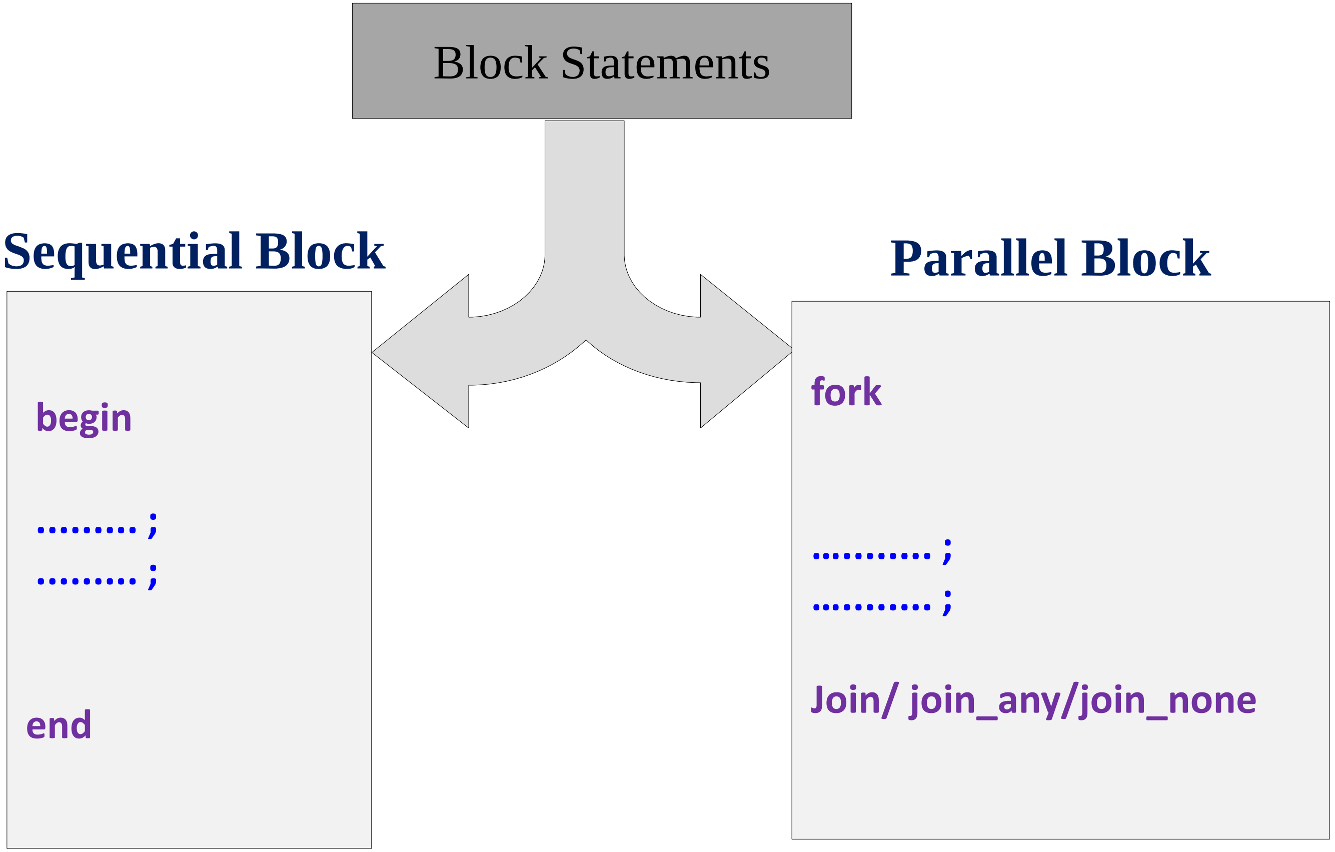
INTERPROCESS COMMUNICATION (IPC)

Topics

- Process execution threads
- Inter-process communication and Synchronization.
- Mailbox
- Semaphores.
- Events.

Process execution threads

Classic Verilog has two ways of grouping statements - with a **begin**...**end** or **fork**...**join**



Block Statements

Sequential Block

```
begin

......... ;
......... ;

end
```

Parallel Block

```
fork

........... ;
........... ;

Join/ join_any/join_none
```

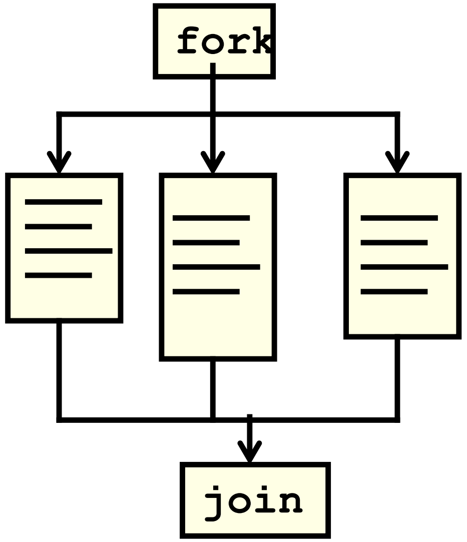
Parallel blocks

The fork-join *parallel block* construct enables the creation of concurrent processes from each of its parallel
statements.

A parallel block shall have the following characteristics:

- Statements shall execute concurrently.
- Delay values for each statement shall be considered relative to the simulation time of entering the block.
- Delay control can be used to provide time-ordering for assignments.
- Control shall pass out of the block when the last time-ordered statement executes based on the type of join keyword.

Fork-join

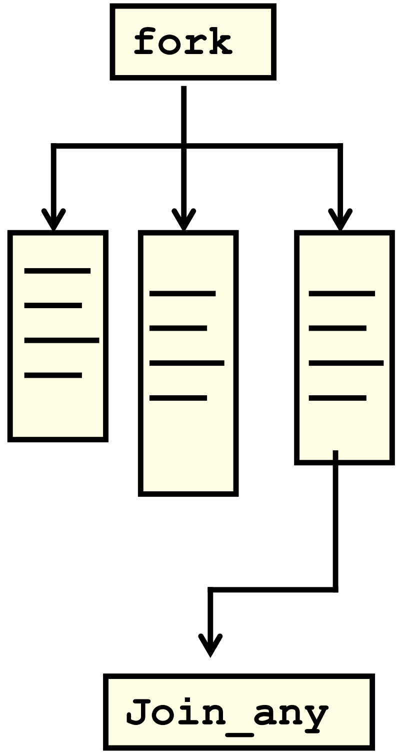


```
module fork_join;
 initial begin
  $display("-----------------------------------------------------------------");

  fork

   begin
          $display($time,"\tProcess-1 Started");
          #5;
          $display($time,"\tProcess-1 Finished");
   end

  begin
          $display($time,"\tProcess-2 Startedt");
          #20;
          $display($time,"\tProcess-2 Finished");
   end
  join

        $display($time,"\tOutside fork_join");
        $display("-----------------------------------------------------------");
 end
endmodule
```

Result:

```
-------------------------------------------
0 Process-1 Started
0 Process-2 Startedt
5 Process-1 Finished
20 Process-2 Finished
20 Outside fork_join
-------------------------------------------
```

Fork - join_any

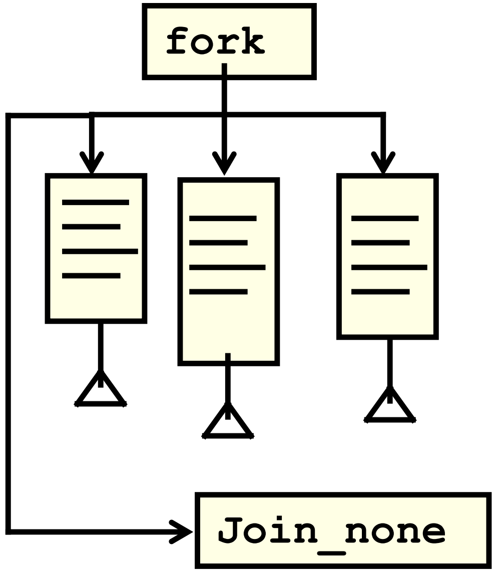


```
module fork_join_any;
 initial begin
  $display("-------------------------------------------------------");

  fork

   begin
          $display($time,"\tProcess-1 Started");
          #5;
          $display($time,"\tProcess-1 Finished");
   end

   begin
          $display($time,"\tProcess-2 Startedt");
          #20;
          $display($time,"\tProcess-2 Finished");
   end
  join_any


  $display($time,"\tOutside fork_join_any");
  $display("-------------------------------------------------------");
 end
endmodule
```

Result:

```
----------------------------------------------
0 Process-1 Started
0 Process-2 Startedt
5 Process-1 Finished
5 Outside fork_join_any
----------------------------------------------
20 Process-2 Finished
```

Fork - join_none



```
module fork_join_none;
 initial begin
  $display("--------------------------------------------------------");

  fork
   //Process-1
   begin
          $display($time,"\tProcess-1 Started");
          #5;
          $display($time,"\tProcess-1 Finished");
   end
   //Process-2
   begin
          $display($time,"\tProcess-2 Startedt");
          #20;
          $display($time,"\tProcess-2 Finished");
   end
  join_none

        $display($time,"\tOutside Fork-Join_none");
        $display("------------------------------------------------");
 end
endmodule
```

Result:

```
-------------------------------------------
0 Outside Fork-Join_none
-------------------------------------------
0 Process-1 Started
0 Process-2 Startedt
5 Process-1 Finished
20 Process-2 Finished
```

Process control

System Verilog provides the following constructs that allow one process to terminate another process or wait for its completion.

- wait ,wait_fork , wait_order
- disable, disable_fork

wait fork

- ➢ The wait fork statement blocks the execution flow until all processes forked by the current process complete.
- ➢ The descendants of forked processes (if any) are ignored and do not block execution.

Ex: Wait

```
module top();
 int cnt;
 bit clk;

 always #5 clk = ~clk;

 always @ (posedge clk)
   cnt <= cnt + 1;

 initial
   begin
     $display("***** wait to finish *****");
     wait(cnt == 8)
     $finish;
   end

 initial $monitor("cnt - %0d", cnt);
endmodule
```

Result:

```
***** wait to finish *****
cnt - 0
cnt - 1
cnt - 2
cnt - 3
cnt - 4
cnt - 5
cnt - 6
cnt - 7
```

Ex: join_any

```
module wait_fork;
 initial begin
  $display("-------------------------------------------------------------------");

  fork
   begin
            $display($time,"\tProcess-1 Started");
            #5;
            $display($time,"\tProcess-1 Finished");
   end
  begin
            $display($time,"\tProcess-2 Startedt");
            #20;
            $display($time,"\tProcess-2 Finished");
   end
  join_any

         $display($time,"\t End of  simulation");
               $display("------------------------------------------------------");
$finish;
 end
endmodule
```

```
-------------------------------------------------------------
-------
0 Process-1 Started
0 Process-2 Startedt
5 Process-1 Finished
5 End of simulation
-------------------------------------------------------
```

Ex: wait fork with join_none

```
module wait_fork;
  initial begin
    $display("-----------------------------------------------------------------");

    fork

      begin
              $display($time,"\tProcess-1 Started");
              #5;
              $display($time,"\tProcess-1 Finished");
      end

     begin
              $display($time,"\tProcess-2 Startedt");
              #20;
              $display($time,"\tProcess-2 Finished");
      end
    join_none
    wait fork;

                $display($time,"\tOutside wait_fork");
$finish;
                $display("-----------------------------------------------------");
  end
endmodule
```

Result:

```
---------------------------------------------------------------
-----
0 Process-1 Started
0 Process-2 Startedt
5 Process-1 Finished
20 Process-2 Finished
20 Outside wait_fork
```

Disable fork

The disable fork statement terminates recursively all processes forked-off by the current process.

```
module disable_fork;
 initial begin
  $display("-------------------------------------------------------------------");

  fork
   //Process-1
   begin
    $display($time,"\tProcess-1 Started");
    #5;
    $display($time,"\tProcess-1 Finished");
   end
     //Process-2
   begin
    $display($time,"\tProcess-2 Started");
    #5;
    $display($time,"\tProcess-2 Finished");
   end
  join
fork
   //Process-3
   begin
    $display($time,"\tProcess-3 Startedt");
    #20;
    $display($time,"\tProcess-3 Finished");
   end
   //Process-4
   begin
    #5
    $display($time,"\tProcess-4 Startedt");
    #20;
    $display($time,"\tProcess-4 Finished");
   end
  join_any
  disable fork;

   $display($time,"\t After disable fork");
$finish;
             $display("-------------------------------------");
 end
endmodule
```

Result:

```
0 Process-1 Started
0 Process-2 Started
5 Process-1 Finished
5 Process-2 Finished
5 Process-3 Startedt
10 Process-4 Startedt
25 Process-3 Finished
25 After disable fork
```

Disable fork

```
module disable_fork;
 initial begin
  $display("-----------------------------------------------");

  fork
   //Process-1
   begin
    $display($time,"\tProcess-1 Started");
    #5;
    $display($time,"\tProcess-1 Finished");
   end
     //Process-2
   begin
    $display($time,"\tProcess-2 Started");
    #5;
    $display($time,"\tProcess-2 Finished");
   end
  join_any
```

```
fork
   //Process-3
         begin
          $display($time,"\tProcess-3 Startedt");
          #20;
          $display($time,"\tProcess-3 Finished");
         end
   //Process-4
   begin
          #5
          $display($time,"\tProcess-4 Startedt");
          #20;
          $display($time,"\tProcess-4 Finished");
         end
  join_none
  disable fork;

             $display($time,"\t After disable fork");
             $finish;
             $display("--------------------------------------");
 end
endmodule
```

Result:

```
0 Process-1 Started
0 Process-2 Started
5 Process-1 Finished
5 After disable fork
```

Disable named block in other process

```
module wait_fork;
 initial begin
  $display("-----------------------------------------------------------------");

  fork
          //Process-1
          begin
                  $display($time,"\tProcess-1 Started");
                  #5;
                  $display($time,"\tProcess-1 Finished");
          end
          //Process-2
          begin : blk_ Process-2
                  $display($time,"\tProcess-2 Startedt");
                  #10;
                  $display($time,"\tProcess-2 Finished");
          end
          //Process-3

          begin
                  $display($time,"\tProcess-3 Startedt");
                  disable blk_ Process-2;

                  #15;

                  $display($time,"\tProcess-3 Finished");
          end
  join
```

```
              $display($time,"\t Outside disable ");
$finish;
              $display("--------------------------------");
 end
endmodule
```

Result:

```
----------------------------------------
0 Process-1 Started
0 Process-2 Startedt
0 Process-3 Startedt
5 Process-1 Finished
15 Process-3 Finished
15 Outside disable
```

Disable named block in same process

```
module wait_fork;
 initial begin
  $display("-----------------------------------------------------");

  fork
   //Process-1
   begin
    $display($time,"\tProcess-1 Started");
    #5;
    $display($time,"\tProcess-1 Finished");
   end
   //Process-2
   begin : blk_ Process-2
    $display($time,"\tProcess-2 Startedt");
    #10;
     disable blk_ Process-2;
    $display($time,"\tProcess-2 Finished");
   end
   //Process-3
   begin
    $display($time,"\tProcess-3 Startedt");


    #15;

    $display($time,"\tProcess-3 Finished");
   end
  join
           $display($time,"\tOutside disable");
           $finish;
           $display("----------------------------------------");
 end
endmodule
```

Result:

```
--------------------------------
0 Process-1 Started
0 Process-2 Startedt
0 Process-3 Startedt
5 Process-1 Finished
15 Process-3 Finished
15 Outside disable
```

Driver	task Driver::Send_to_dut	task Driver::Drive_Inputs

class Driver;
virtual Mem_intf vintf;
 	
function new(input virtual Mem_intf vintf);
this.vintf = vintf;
endfunction:new
 	
//-------List of Methods-------//
extern task Send_to_dut();
extern task Initilaization();
extern task Assert_Deassert_rst();
extern task Drive_Inputs();
extern task Write_mem(input integer num_of_writes);
extern task Read_mem(input integer num_of_reads);
endclass:Driver
	task Driver::Send_to_dut();
	Initilaization();
	Assert_Deassert_rst();
	Drive_Inputs();
	endtask:Send_to_dut
	
	task Driver::Initilaization();
	vintf.rst=1;vintf.cb_driver.wr<=0;vintf.cb_driver.rd<= 0;vintf.cb_driver.addr<=0;vintf.cb_driver.datain<=0;
	endtask:Initilaization
	
	task Driver::Assert_Deassert_rst();
	vintf.rst =1;
	repeat(2)@(posedge vintf.clk);
	vintf.rst = 0;
	endtask:Assert_Deassert_rst
		task Driver::Drive_Inputs();
		Write_mem(10);
		Read_mem(10);
		repeat(5)@(posedge vintf.clk);
		endtask:Drive_Inputs
		
		task Driver::Write_mem(input integer num_of_writes);
		for(int wr_loop=0;wr_loop<num_of_writes;wr_loop=wr_loop+1) begin
		@(posedge vintf.clk);
		vintf.cb_driver.wr <= 1;
		vintf.cb_driver.addr <= wr_loop;
		vintf.cb_driver.datain <= $urandom_range(200,600);
		end
		@(posedge vintf.clk);
		vintf.cb_driver.wr <= 0;
		endtask:Write_mem

Driver::Read_mem	Input_monitor	Output_monitor

task Driver::Read_mem(input integer num_of_reads);
for(int rd_loop=0;rd_loop<num_of_reads;rd_loop=rd_loop+1) begin
@(posedge vintf.clk);
vintf.cb_driver.rd   <= 1;
vintf.cb_driver.addr   <= rd_loop;
vintf.cb_driver.datain <= 0;
end
@(posedge vintf.clk);
vintf.cb_driver.rd   <= 0;
endtask:Read_mem
	class Input_monitor;
	virtual Mem_intf vintf;
	function new(input virtual Mem_intf vintf);
	this.vintf = vintf;
	endfunction:new
	//-------List of Methods-------//
	extern task Sample_Inputs(ref int arr_exp_data[int]);
	endclass:Input_monitor
	
	task Input_monitor::Sample_Inputs(ref int arr_exp_data[int]);
	forever begin
	@(negedge vintf.clk);
	if(vintf.cb_monitor.wr)
	begin
	arr_exp_data[vintf.cb_monitor.addr] = vintf.cb_monitor.datain;
	end
	end
	endtask:Sample_Inputs
		class Output_monitor;
		virtual Mem_intf vintf;
		function new(input virtual Mem_intf vintf);
		this.vintf = vintf;
		endfunction:new
		//-------List of Methods-------//
		extern task Sample_Outputs(ref int arr_actdata[int]);
		endclass:Output_monitor
		
		task Output_monitor::Sample_Outputs(ref int arr_actdata[int]);
		forever begin
		@(posedge vintf.clk);
		if(vintf.cb_monitor.rd) begin
		arr_actdata[vintf.cb_monitor.addr] = vintf.cb_monitor.dataout;
		end
		end
		endtask:Sample_Outputs

Compare_inp_out

```
class Compare_inp_out;
 //-------List of Methods-------//
 extern task Comp_Exp_Act_data(ref int
arr_datain[int],ref int arr_dataout[int],ref logic[7:0]
addr);
endclass:Compare_inp_out

   task Compare_inp_out::Comp_Exp_Act_data(ref
int arr_datain[int],ref int arr_dataout[int],ref
logic[7:0] addr);
    forever begin
      @(arr_dataout.size());
      if(arr_datain[addr] == arr_dataout[addr])
      begin
         $display($time,"\n******-------
PASS:Exp_datain[%0d]=%0d----
Act_dataout[%0d]=%0d",addr,arr_datain[addr],addr,a
rr_dataout[addr]);
       end
       else
       begin
          $display($time,"\n******-------
FAIL:Exp_datain[%0d]=%0d----
Act_dataout[%0d]=%0d",addr,arr_datain[addr],addr,a
rr_dataout[addr]);
       end
    end
endtask:Comp_Exp_Act_dataclass Compare_inp_out;
```

Input_monitor

```
//-------List of Methods-------//
 extern task Comp_Exp_Act_data(ref int
arr_datain[int],ref int arr_dataout[int],ref logic[7:0]
addr);

endclass:Compare_inp_out

   task Compare_inp_out::Comp_Exp_Act_data(ref int
arr_datain[int],ref int arr_dataout[int],ref logic[7:0]
addr);
    forever begin
      @(arr_dataout.size());
      if(arr_datain[addr] == arr_dataout[addr])
      begin
         $display($time,"\n******-------
PASS:Exp_datain[%0d]=%0d----
Act_dataout[%0d]=%0d",addr,arr_datain[addr],addr,arr
_dataout[addr]);
       end
       else
       begin
          $display($time,"\n******-------
FAIL:Exp_datain[%0d]=%0d----
Act_dataout[%0d]=%0d",addr,arr_datain[addr],addr,arr
_dataout[addr]);
       end
    end
endtask:Comp_Exp_Act_data
```

tb_Memory

```
module tb_Memory(Mem_intf tb_intf);
 int      Exp_datain[int];
 int      Act_dataout[int];

   //------List of classes-------//
   Driver h_drv;
   Input_monitor h_imon;
   Output_monitor h_omon;
   Compare_inp_out h_comp_inout;
   initial
   begin
      $dumpfile("dump.vcd"); $dumpvars;
      h_drv        =  new(tb_intf);
      h_imon       =  new(tb_intf);
      h_omon       =  new(tb_intf);
      h_comp_inout =  new();
      fork
         h_drv.Send_to_dut();
         h_imon.Sample_Inputs(Exp_datain);
         h_omon.Sample_Outputs(Act_dataout);
         h_comp_inout.Comp_Exp_Act_data(Exp_datain,
Act_dataout,tb_intf.addr);
      join_any
    $finish();
   end


endmodule
```

Results

```
PASS:Exp_datain[0]=569----Act_dataout[0]=569
PASS:Exp_datain[1]=522----Act_dataout[1]=522
PASS:Exp_datain[2]=598----Act_dataout[2]=598
PASS:Exp_datain[3]=316----Act_dataout[3]=316
PASS:Exp_datain[4]=429----Act_dataout[4]=429
PASS:Exp_datain[5]=284----Act_dataout[5]=284
PASS:Exp_datain[6]=563----Act_dataout[6]=563
PASS:Exp_datain[7]=375----Act_dataout[7]=375
PASS:Exp_datain[8]=323----Act_dataout[8]=323
PASS:Exp_datain[9]=355----Act_dataout[9]=355
```

Inter-process communication and Synchronization

- System Verilog offers some set of capabilities that allows us to create inter-process Communication and Synchronization dynamically.

  Ex:

  Mailbox

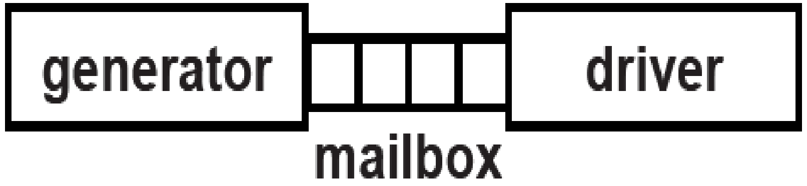
  Semaphores

- System Verilog also enhances Verilog's named **event** data type to satisfy many of the system-level synchronization requirements

MAILBOX

INTRODUCTION

- A mailbox is a communication mechanism that allows messages to be exchanged between processes.
- Data can be sent to mailbox by one process and retrieve by another process.
- It is a built-in class.
- Used as a communication channel between processes to pass messages between two threads.
- Can be used to represent FIFO implementations
- Default mailbox has no data type.



```
mailbox #(class_type) mbx;
mbx=new();
mbx.put(trans_h);
mbx.get(trans_h);
count=mbx.num();
```

- Mailboxes are created as having either a bounded or unbounded queue size.
- A bounded mailbox becomes full when it contains the bounded number of messages.
- A process that attempts to place a message into a full mailbox shall be suspended until enough space becomes available in the mailbox queue.
- Unbounded mailboxes are with unlimited size.

There are two types of mailboxes

1. **Generic Mailbox (type-less mailbox)** : The default mailbox is type-less, that is, a single mailbox can send and receive any type of data.

```
mailbox mailbox_name;
```

2. **Parameterized mailbox (mailbox with particular type)** : Mailbox is used to transfer a particular type of data.

```
mailbox#(type) mailbox_name;
```

Note:

-> mailbox mbx: For type mismatches it gives Runtime errors

-> mailbox #(type) mbx ; For type mismatches it gives the com[il time errors

Methods

new() : Create a mailbox

put() : Place a message in a mailbox

try_put() : Try to place a message in a mailbox without blocking

get() : Retrieve a message from a mailbox

peek() : Retrieve copy of the message from a mailbox

try_get() : Try to retrieve a message from a mailbox without blocking

try_peek() : Try to retrieve copy of the message from a mailbox without blocking

num() : Retrieve the number of messages in the mailbox

New()

- Mailboxes are created with the **new**() method.
- The prototype for mailbox **new**() is as follows:

```
function new(int bound = 0);
```

- The **new**() function returns the mailbox handle. If the bound argument is 0, then the mailbox is unbounded (the default) and a put() operation shall never block.
- If bound is nonzero, it represents the size of the mailbox queue.
- The bound shall be positive.
- Negative bounds are illegal and can result in indeterminate behavior, but implementations can issue a warning.

Num()

- The number of messages in a mailbox can be obtained via the num() method.
- The prototype for num() is as follows:

```
function int num();
```

- The num() method returns the number of messages currently in the mailbox.
- The returned value should be used with care because it is valid only until the next get() or put() is executed on the mailbox.
- These mailbox operations can be from different processes from the one executing the num() method.
- Therefore, the validity of the returned value depends on the time that the other methods start and finish.

Put()

- The put() method places a message in a mailbox.

- The prototype for put() is as follows:

```
task put( singular message);
```

- The message is any singular expression, including object handles.

- The put() method stores a message in the mailbox in strict FIFO order.

- If the mailbox was created with a bounded queue, the process shall be suspended until there is enough room in the queue.

Try_put()

- The try_put() method attempts to place a message in a mailbox.

- The prototype for try_put() is as follows:

```
function int try_put( singular message);
```

- The message is any singular expression, including object handles.

- The try_put() method stores a message in the mailbox in strict FIFO order.

- This method is meaningful only for bounded mailboxes. If the mailbox is not full, then the specified message is placed in the mailbox, and the function returns a positive integer.

- If the mailbox is full, the method returns 0.

Get()

- The get() method retrieves a message from a mailbox.
- The prototype for get() is as follows:

```
task get( ref singular message );
```

- The message can be any singular expression, and it shall be a valid left-hand expression.
- The get() method retrieves one message from the mailbox, that is, removes one message from the mailbox queue*.*
- If the mailbox is empty, then the current process blocks until a message is placed in the mailbox.

Try_get()

- The try_get() method attempts to retrieves a message from a mailbox without blocking.
- The prototype for try_get() is as follows:

```
function int try_get( ref singular message );
```

- The message can be any singular expression, and it shall be a valid left-hand expression.
- The try_get() method tries to retrieve one message from the mailbox. If the mailbox is empty, then the method returns 0.
- If a message is available and the message type is equivalent to the type of the message variable, the message is retrieved, and the method returns a positive integer.

Peek()

- The peek() method copies a message from a mailbox without removing the message from the queue.
- The prototype for peek() is as follows:

```
task peek( ref singular message );
```

- The message can be any singular expression, and it shall be a valid left-hand expression.
- The peek() method copies one message from the mailbox without removing the message from the mailbox queue.
- If the mailbox is empty, then the current process blocks until a message is placed in the mailbox.
- Calling the peek() method can also cause one message to unblock more than one process.
- As long as a message remains in the mailbox queue, any process blocked in either a peek() or get() operation shall become unblocked.

Try_peek()

- The try_peek() method attempts to copy a message from a mailbox without blocking.
- The prototype for try_peek() is as follows:

```
function int try_peek( ref singular message );
```

- The message can be any singular expression, and it shall be a valid left-hand expression.
- The try_peek() method tries to copy one message from the mailbox without removing the message from the mailbox queue.
- If the mailbox is empty, then the method returns 0.
- If a message is available and its type is equivalent to the type of the message variable, the message is copied, and the method returns a positive integer.

Mailbox size (unbounded)

```
module tb;
 int gen_data,drvr_data;
 mailbox mbx;

 initial
 begin
   mbx = new();

   fork
     Generator_put();
     Driver_get();
   join_any

  #10;
  $finish;
 end
```

```
task Generator_put();
    repeat(4) begin
      gen_data = $urandom_range(10,20);
      mbx.put(gen_data);
      $display("After putting:::::::into
mailbox gen_data=%0d",gen_data);
    end
  endtask
```

```
task Driver_get();
   forever begin
      mbx.get(drvr_data);
      $display("After getting:::::::: from mailbox
drvr_data=%0d", drvr_data);
    end
  endtask
endmodule
```

Result:

```
After putting:::::::into mailbox gen_data=10
 After putting:::::::into mailbox gen_data=20
 After putting:::::::into mailbox gen_data=16
 After putting:::::::into mailbox gen_data=30
 After getting:::::::: from mailbox drvr_data=10
 After getting:::::::: from mailbox drvr_data=20
 After getting:::::::: from mailbox drvr_data=16
 After getting:::::::: from mailbox drvr_data=30
```

Mailbox size (1)

```
module tb;
 int gen_data,drvr_data;
 mailbox mbx;

 initial
 begin
   mbx = new(1);

   fork
     Generator_put();
     Driver_get();
   join_any

  #10;
  $finish;
 end
```

```
task Generator_put();
    repeat(4) begin
      gen_data = $urandom_range(10,20);
      mbx.put(gen_data);
      $display("After putting:::::::into
mailbox gen_data=%0d",gen_data);
    end
  endtask
```

```
task Driver_get();
   forever begin
      mbx.get(drvr_data);
      $display("After getting:::::::: from mailbox
drvr_data=%0d", drvr_data);
    end
  endtask
endmodule
```

Result:

```
After putting:::::::into mailbox gen_data=10
After getting:::::::: from mailbox drvr_data=10
After putting:::::::into mailbox gen_data=20
After getting:::::::: from mailbox drvr_data=20
After putting:::::::into mailbox gen_data=16
After getting:::::::: from mailbox drvr_data=16
After putting:::::::into mailbox gen_data=30
After getting:::::::: from mailbox drvr_data=30
```

Main Program

```
module tb;
 integer gen_data,drvr_data;
 mailbox mbx;

 initial
 begin
  $dumpfile("dump.vcd"); $dumpvars;
  mbx = new(1);

  fork
   Generator_put();
   Driver_get();
  join_any

  #10;
  $finish;
 end
```

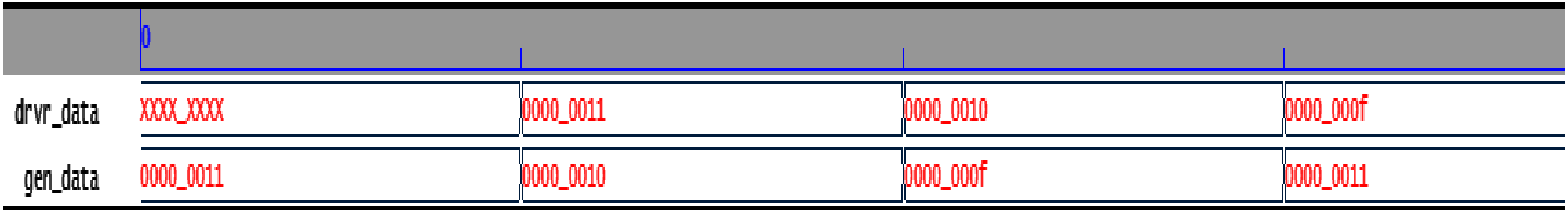
Put task

```
 task Generator_put();
   repeat(4) begin
    gen_data = $urandom_range(10,20);
    #1;
    mbx.put(gen_data);
    $display("time=%0t-After putting:::::::into mailbox gen_data=%0d",$time,gen_data);
   end
  endtask
```

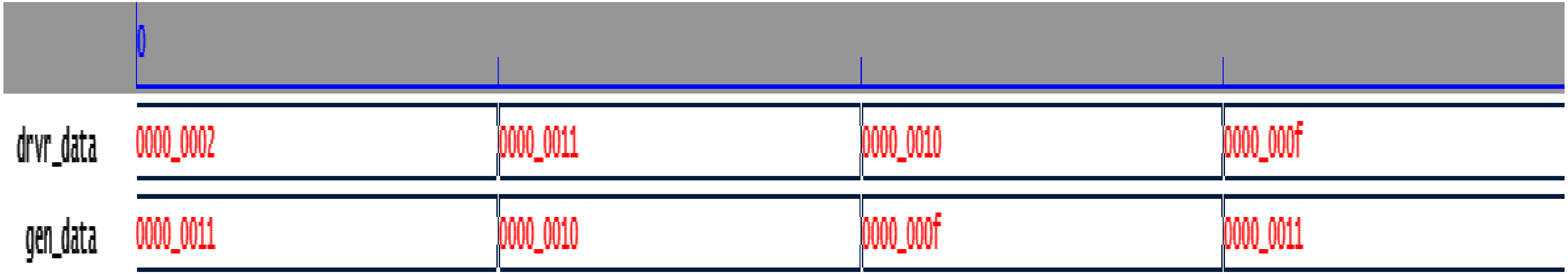
Get task

```
task Driver_get();
  forever begin
   // mbx.try_get(drvr_data);
    mbx.get(drvr_data);
    $display("time=%0t-After getting:::::::: from mailbox drvr_data=%0d",$time,drvr_data);
    if(drvr_data === 'bx)
    begin
         drvr_data = 2;
      $display("driving some data inmeanwhile of getting actual data drvr_data=%0d",drvr_data);
    end

    #1;
   end
  endtask
endmodule
```

Result with get



Result with try_get



Semaphores:-

- Built-in class
- Use semaphores to arbitrate between two or more threads.
- Semaphore are a key based synchronization mechanism
- Used for mutual exclusion
- controlling access to shared resources
- process synchronization
- It has predefined methods inside this semaphore.

Predefined Methods

Methods in the Semaphores :

new() : Create a semaphore with specified number of keys.

get() : Obtain one or more keys from a semaphore and block until keys are available.

try_get() : Obtain one or more keys from a semaphore without blocking**.**

put() : Return one or more keys to a semaphore.

New()

- Semaphores are created with the **new**() method.

- The prototype for **new**() is as follows:

```
function new(int keyCount = 0 );
```

- The keyCount specifies the number of keys initially allocated to the semaphore bucket.

- The number of keys in the bucket can increase beyond keyCount when more keys are put into the semaphore than are removed.

- The default value for keyCount is 0.

- The **new**() function returns the semaphore handle.

Put()

- The semaphore put() method is used to return keys to a semaphore.

- The prototype for put() is as follows:

```
function void put(int keyCount = 1);
```

- The keyCount specifies the number of keys being returned to the semaphore. The default is 1.

- When the semaphore.put() function is called, the specified number of keys is returned to the semaphore.

- If a process has been suspended waiting for a key, that process shall execute if enough keys have been returned.

Get()

- The semaphore get() method is used to procure a specified number of keys from a semaphore.

- The prototype for get() is as follows:

```
task get(int keyCount = 1);
```

- The keyCount specifies the required number of keys to obtain from the semaphore. The default is 1.

- If the specified number of keys is available, the method returns and execution continues. If the specified
  number of keys is not available, the process blocks until the keys become available.

- The semaphore waiting queue is first-in first-out (FIFO). This does not guarantee the order in which processes arrive at the queue, only that their arrival order shall be preserved by the semaphore.

Try_get()

- The semaphore try_get() method is used to procure a specified number of keys from a semaphore, but without blocking.
- The prototype for try_get() is as follows:

```
function int try_get(int keyCount = 1);
```

- The keyCount specifies the required number of keys to obtain from the semaphore. The default is 1.
- If the specified number of keys is available, the method returns a positive integer and execution continues.
- If the specified number of keys is not available, the method returns 0.

Semaphore

```
module sema_test;
 semaphore sema = new(2);
 initial
  begin
   $display("Before Fork- time %0d",$time);
   fork

    begin
     sema.get(2);
     $display("\n get(2) - time %0d",$time);
     #10;
     sema.put(2);
     $display("\n put(2) - time %0d",$time);
    end

    begin
     #1;
     sema.get(1);
     $display("\n get(1) - time %0d",$time);
     #10;
     sema.put(1);
     $display("\n put(1) - time %0d",$time);
    end
```

```
begin
     #1;
     sema.get(1);
     $display("\n get(1) - time %0d",$time);
     #10;
     sema.put(1);
     $display("\n put(1) - time %0d",$time);
    end

begin
     #1;
     sema.get(1);
     $display("\n get(1) - time %0d",$time);
     #10;
     sema.put(1);
     $display("\n put(1) - time %0d",$time);
    end

   join
  end
endmodule
```

Result:

```
Before Fork- time 0

get(2) - time 0

put(2) - time 10

get(1) - time 10

get(1) - time 10

put(1) - time 20

put(1) - time 20

get(1) - time 20

put(1) - time 30
```

Events

- An event trigger is a basic building block in any Hardware description Language.
- Verilog provides two such synchronization mechanisms are -> and **@**.
- The main drawback of these two operators is that can only work on and create **static events**.
- System Verilog provides additional two such synchronization mechanisms are **->>** and **triggered property**.

Triggering an event

-> operator

- Named events are triggered via the -> operator unblocks all processes currently waiting on that event.

->>operator

- Nonblocking events are triggered using the ->> operator.

Waiting for an event

@ operator

- Waiting for the triggered event can be realized using the @ operator.
- The calling process is blocked until the named event is triggered.
- The effect of the ->> operator is that the statement executes without blocking, and it creates a nonblocking assign update event in the time in which the delay control expires or the event control occurs.

Triggered Property

- The triggered property represents the state of the named event. The property evaluates to true if the given event has been triggered in the current time step and false otherwise.

```
wait(named_event.triggered);
```

//*************** Problem with the event -> , @ *************//

```
module tb;
 event bus;

  initial
  begin
     $display("time=%0t-----------Before bus came",$time);
     ->bus;
  end

  initial
  begin
     @(bus);
     $display("time=%0t-------------waited for bus , its came",$time);

  end
  endmodule
```

```
//Result:
//time=0-----------Before bus came
```

```
 //=========problem overcomed by .triggered==========//
module tb;
event bus;

  initial
  begin
     $display("time=%0t-----------Before bus came",$time);
     ->bus;
  end

  initial
  begin
     wait(bus.triggered);
     $display("time=%0t-------------waited for bus , its came",$time);
     #1;
     wait(bus.triggered);
     $display("time=%0t-------------After #1 waited for bus , its
came",$time);
  end
endmodule
```

```
//Result:
//time=0-----------Before bus came
//time=0-------------waited for bus , its came
```

Blocking event trigger:

```
module test();
  event e;
  integer i =0;
 always @e
 $display(“i is %d”,i);
 initial
  begin
     i<=1;
    -> e;
  end
endmodule
```

Result: i is 0

Non blocking event trigger:

```
module test();
    event e;
    integer i =0;
 always @e
 $display(“i is %d”,i);
  initial
    begin
      i<=1;
      ->> e;
    end
endmodule
```

Result : i is 1

Event sequencing

The wait_order construct suspends the calling process until all of the specified events are triggered in the given order (left to right) or any of the un-triggered events are triggered out of order and thus causes the operation to fail.

Ex:- wait_order(e1,e2,e3);

Result:

```
We are in initial time 10
Events are in order 20
Fatal Error: Got an out of order event in wait_order persistent triggered
property set on non-first event at time 20
```

```
module main;
event e_pkt1,e_pkt2,e_pkt3;
initial
begin
#10;
  $display("We are in initial time %0t",$time);
#10;
-> e_pkt1;
-> e_pkt2;
-> e_pkt3;
  #10;
-> e_pkt1;
-> e_pkt3;
-> e_pkt2;#1;
    #10;
-> e_pkt1;
-> e_pkt2;
-> e_pkt3;#1;
end
  initial begin
    forever
    begin
      wait_order(e_pkt1,e_pkt2,e_pkt3);
          $display(" Events are in order %0t",$time);

     end
  end
endmodule
```

THANK YOU