Chapter 7, Threads and Interprocess Communication Topics

• Working with Threads
• fork join/join_any/join_none
• Waiting for Threads
• Disabling Threads
• Interprocess communication
• Events
• Semaphores
• Mailboxes
Chapter 7  Threads and Interprocess Communication

• The testbench has many threads running in parallel

Communication between and control of these threads is through
  • Standard Verilog events, event control, and wait statements
  • SystemVerilog mailboxes and semaphores
7.1 Working with Threads

Verilog 2001 has only fork/join to control thread execution

```
initial begin
    fork
    begin
        ...thread 1...
    end
    begin
        ...thread 2...
    end
    begin
        ...thread 3...
    end
    join
end
```
7.1.1 Using fork..join and begin..end

<table>
<thead>
<tr>
<th>Order</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>40</td>
</tr>
<tr>
<td>6</td>
<td>50</td>
</tr>
<tr>
<td>7</td>
<td>60</td>
</tr>
<tr>
<td>8</td>
<td>60</td>
</tr>
<tr>
<td>9</td>
<td>140</td>
</tr>
</tbody>
</table>

```
initial begin
    $display("%@0t: start fork...join example", $time);
    #10 $display("%@0t: sequential after #10", $time);
    fork
        $display("%@0t: parallel start", $time);
        #50 $display("%@0t: parallel after #50", $time);
        #10 $display("%@0t: parallel after #10", $time);
        begin
            #30 $display("%@0t: sequential after #30", $time);
            #10 $display("%@0t: sequential after #10", $time);
        end
    join
        $display("%@0t: after join", $time);
        #80 $display("%@0t: finish after #80", $time);
end
```
7.1.2 Spawning Threads w/ fork..join_none

SystemVerlog has added fork....join_none to control thread execution

initial begin
  fork
  begin
    ...thread 1...
  end
  begin
    ...thread 2...
  end
  begin
    ...thread 3...
  end
  join_none
end
7.1.2 Spawning Threads with `fork..join_none`

<table>
<thead>
<tr>
<th>Order</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>3 or 4</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>40</td>
</tr>
<tr>
<td>7</td>
<td>50</td>
</tr>
<tr>
<td>8</td>
<td>60</td>
</tr>
<tr>
<td>9</td>
<td>90</td>
</tr>
</tbody>
</table>

```
initial begin
    $display("@%0t: start fork...join example", $time);
    #10 $display("@%0t: sequential after #10", $time);
    fork
        $display("@%0t: parallel start", $time);
        #50 $display("@%0t: parallel after #50", $time);
        #10 $display("@%0t: parallel after #10", $time);
        begin
            #30 $display("@%0t: sequential after #30", $time);
            #10 $display("@%0t: sequential after #10", $time);
        end
    join_none
        $display("@%0t: after join_none", $time);
        #80 $display("@%0t: finish after #80", $time);
end
```
7.1.3 Spawning Threads w/ fork..join_any

SystemVerlog has added fork....join_any to control thread execution

```
initial begin
  fork
  begin
    ...thread 1...
  end
  begin
    ...thread 2...
  end
  begin
    ...thread 3...
  end
  join_any
end
```
### 7.1.3 Spawning Threads w/ `fork..join_any`

<table>
<thead>
<tr>
<th>Order</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>40</td>
</tr>
<tr>
<td>7</td>
<td>50</td>
</tr>
<tr>
<td>8</td>
<td>60</td>
</tr>
<tr>
<td>9</td>
<td>90</td>
</tr>
</tbody>
</table>

```plaintext
initial begin

    $display("@%0t: start fork...join example", $time);
    #10 $display("@%0t: sequential after #10", $time);
    fork

    $display("@%0t: parallel start", $time);
    #50 $display("@%0t: parallel after #50", $time);
    #10 $display("@%0t: parallel after #10", $time);
    begin

        #30 $display("@%0t: sequential after #30", $time);
        #10 $display("@%0t: sequential after #10", $time);

    end

    join_any

    $display("@%0t: after join_any", $time);
    #80 $display("@%0t: finish after #80", $time);

end
```
fork..join/join_any/join_none exercise

initial begin

$display("@%0t: start fork...join_none example", $time);

fork
begin

#20 $display("@%0t: sequential A after #20", $time);
#20 $display("@%0t: sequential B after #20", $time);
end

$display("@%0t: parallel start", $time);

#50 $display("@%0t: parallel after #50", $time);

begin

#30 $display("@%0t: sequential after #30", $time);
#10 $display("@%0t: sequential after #10", $time);
end

join or join_any or join_none

$display("@%0t: after join", $time);

#80 $display("@%0t: finish after #80", $time);

end
fork..join/join_any/join_none exercise

<table>
<thead>
<tr>
<th>Order</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>30</td>
</tr>
<tr>
<td>5</td>
<td>40</td>
</tr>
<tr>
<td>6</td>
<td>50</td>
</tr>
<tr>
<td>7</td>
<td>50</td>
</tr>
<tr>
<td>8</td>
<td>130</td>
</tr>
</tbody>
</table>

```
initial begin
  $display("@%0t: start fork...join example", $time);
  fork
    begin
      #20 $display("@%0t: sequential A after #20", $time);
      #20 $display("@%0t: sequential B after #20", $time);
    end
  $display("@%0t: parallel start", $time);
  begin
    #50 $display("@%0t: parallel after #50", $time);
    begin
      #30 $display("@%0t: sequential after #30", $time);
      #10 $display("@%0t: sequential after #10", $time);
    end
  end $display("@%0t: after join", $time);
  join
  $display("@%0t: after join", $time);
  #80 $display("@%0t: finish after #80", $time);
end
```
fork..join/join_any/join_none exercise

<table>
<thead>
<tr>
<th>Order</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>40</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>50</td>
</tr>
<tr>
<td>5</td>
<td>30</td>
</tr>
<tr>
<td>7</td>
<td>40</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>80</td>
</tr>
</tbody>
</table>

initial begin
fork
begin
#20 $display("@%0t: sequential A after #20", $time);
#20 $display("@%0t: sequential B after #20", $time);
end
$display("@%0t: parallel start", $time);
#50 $display("@%0t: parallel after #50", $time);
begin
#30 $display("@%0t: sequential after #30", $time);
#10 $display("@%0t: sequential after #10", $time);
end
join_any
$display("@%0t: after join", $time);
#80 $display("@%0t: finish after #80", $time);
end
fork..join/join_any/join_none exercise

```
initial begin
  $display("@%0t: start fork...join_none example", $time);
  fork
    begin
      #20 $display("@%0t: sequential A after #20", $time);
      #20 $display("@%0t: sequential B after #20", $time);
    end
  $display("@%0t: parallel start", $time);
  #50 $display("@%0t: parallel after #50", $time);
  begin
    #30 $display("@%0t: sequential after #30", $time);
    #10 $display("@%0t: sequential after #10", $time);
  end
  join_none
  $display("@%0t: after join", $time);
  #80 $display("@%0t: finish after #80", $time);
end
```
class Gen_drive;
    task run(input int n);
        Packet p;
        fork
            repeat (n) begin
                p = new();
                `SV_RAND_CHECK(p.randomize());
                transmit(p);
            end
        join_none
    endtask
endclass
Gen_drive gen;
initial begin
    gen = new();
    gen.run(10);
    // Start the checker, monitor, and other threads
end
7.1.6 Automatic Variables in Threads

Variables shared by threads should be declared as automatic

```
program automatic test;
  initial begin
    for (int j=0; j<3; j++) begin
      fork $display(j); join_none
    end
  end
endprogram
```

```
program automatic test;
  initial begin
    for (int j=0; j<3; j++) begin
      automatic int k = j;
      fork $display(k); join_none
    end
  end
endprogram
```
7.1.7 Waiting for all Spawned Threads

• When all initial blocks are done, the simulator exits

```verilog
initial begin
    fork
        transmit(1);
        transmit(2);
        transmit(3);
    join_none
        // Spawn monitor, checker, etc.
        wait fork;
        $display("Done at time %t", $time);
end

task transmit(int index);
    #10ns;
    $display("index = %0d", index);
endtask
```

• Use the `wait fork` construct to wait for all threads to complete
7.2 Disabling Threads

• Why? To keep a wait from hanging your testbench

```verilog
initial begin
    wait_for_response(dead_port);
end
```

To disable unneeded threads

```verilog
task get_first(output int adr);
    fork
    wait_device(1, adr);
    wait_device(7, adr);
    wait_device(13, adr);
    join_any
endtask
```

• How?
  • disable
  • disable fork
7.2.1 Disabling a Single Thread

“Disable shall end all processes executing a particular block, whether the processes were forked by the calling thread or not”

- Disable uses labels to determine which fork to disable

```verilog
task check_trans(Transaction tr);
  fork
    begin
      fork : timeout_block
        begin
          wait (bus.cb.data == tr.data);
          $display("@%0t: data match %d", $time, tr.data);
        end
        #TIME_OUT $display("@%0t: Error: timeout", $time);
        join_any
        disable timeout_block;
      end
    end
  join_none
endtask
```
7.2.2 Disabling Multiple Threads

"disable fork shall end only the processes that were spawned by the calling thread"

```verilog
initial begin
    check_trans(tr0);
    // Create a thread to limit scope of disable
    fork : threads_inner
    begin
        check_trans(tr1);
        fork
        check_trans(tr2);
        join
        #(TIME_OUT/10) disable threads_inner;
    end
    join
end
```

Stops threads 2-4 only
7.2.2 Disabling Multiple Threads (cont.)

task get_first( output int addr );
    fork Thread 0
    wait_device(1, addr);
    wait_device(7, addr);
    wait_device(13, addr);
    join_any
    disable fork;
endtask
initial begin
    fork transmit(1); transmit(2); join_none
    fork: receive_fork
        receive(1);
        receive(2);
    join_none
    #15ns disable receive_fork;
    $display("%0t: Done", $time);
end

task transmit(int index);
    #10ns;
    $display("%0t: Transmit is done for index = %0d", $time, index);
endtask

task receive(int index);
    #(index * 10ns);
    $display("%0t: Receive is done for index = %0d", $time, index);
endtask

What is the output with/without wait fork;
Wait fork and fork disable exercise

Since wait fork only resumes execution when all child threads are complete:

With wait fork:

# 10: Receive is done for index = 1  // Transmit for index 1 and 2, and receive 1 are all
done at 10ns. Order is indeterminate
# 10: Transmit is done for index = 1
# 10: Transmit is done for index = 2
# 20: Receive is done for index = 2  // Due to wait fork we wait for all child processes to
complete so the receive(2) completes
# 35: Done  // All processes are complete so the #15ns disable receive_fork line is not
required. It waits for 15ns but there are no receive threads to disable

Without wait fork the transmit and receive processes are all spawned at time 0 and
execution continues to the disable #15ns receive_fork line.
# 10: Receive is done for index = 1
# 10: Transmit is done for index = 1
# 10: Transmit is done for index = 2  // receive(2) does not complete until 20ns and is
disabled at 15ns.
# 15: Done
7.3 Interprocess Communication (IPC)

• The testbench needs to control:
  • Threads waiting for each other
  • Threads competing for a resource
  • Exchange of data between objects

• Three parts to IPC:
  • producer – creates the information
  • consumer – accepts the information
  • channel – carries the information
  • producer and consumer are separate threads

• Control is accomplished through
  • Events
  • Semaphores
  • Mailboxes
7.4 Events

• Verilog 2001 had primitive event synchronization
  • Declare variables as type `event`
  • Trigger events with `->` operator
  • Wait for events with `@` operator
• SystemVerilog enhances Verilog 2001 events
  • An event is now a handle to a synchronization object
  • The `triggered()` method checked if an event has triggered
7.4.1 Blocking on the edge of an event

With Verilog 2001 a thread can miss an event and stall

```verilog
event e1, e2;
initial begin
    $display("@%0t: 1: before trigger", $time);
    -> e1;
    @e2;
    $display("@%0t: 1: after trigger", $time);
end

initial begin
    $display("@%0t: 2: before trigger", $time);
    -> e2;
    @e1;
    $display("@%0t: 2: after trigger", $time);
end
```

OR

```verilog
    @0: 1: before trigger
    @0: 2: before trigger
    @0: 1: after trigger

OR

    @0: 2: before trigger
    @0: 1: before trigger
    @0: 2: after trigger
```
7.4.2 Waiting for an event trigger

SystemVerilog introduces the `triggered()` method

```verilog
event e1, e2;
initial begin
  $display("@%0t: 1: before trigger", $time);
  -> e1;
  wait (e2.triggered());
  $display("@%0t: 1: after trigger", $time);
end

initial begin
  $display("@%0t: 2: before trigger", $time);
  -> e2;
  wait (e1.triggered());
  $display("@%0t: 2: after trigger", $time);
end
```
7.4.3 Using Events in a Loop

Since `triggered()` looks for events in the current time step if time does not advance `triggered()` is always satisfied.

```verilog
event handshake;
initial begin
  forever begin
    wait(handshake.triggered());
    $display("%t: Received next event", $time);
    process_in_zero_time();
  end
end

initial begin
  #10ns;
  -> handshake;
  #10ns;  task process_in_zero_time();
end  endtask
```

10: Received next event
10: Received next event
10: Received next event
10: Received next event
10: Received next event
10: Received next event
10: Received next event
10: Received next event
10: Received next event
10: Received next event
7.4.4 Passing Events
With SystemVerilog, an event is now a handle to a synchronization object so it can be passed to tasks and functions.

```verilog
package my_package;
    class Generator;
        event done;
        function new (event done);
            this.done = done;
        endfunction
    endclass
endpackage
```
### 7.4.4 Passing Events (cont.)

#### Usage of class Generator

```verilog
program automatic test;

import my_package::*;
event gen_done;
Generator gen;
initial begin
    gen = new(gen_done);
    gen.run();
    $display("%0t: Waiting on gen_done", $time);
    wait(gen_done.triggered());
    $display("%0t: Done waiting on gen_done", $time);
end

endprogram
```

- **gen_done**
- **synchronization**
- **object**

0: Waiting on gen_done
10: Done waiting on gen_done
7.4.5 Waiting for multiple Events

How to wait for multiple threads of class Generator to finish?

```plaintext
event done[N_GENERATORS];
initial begin
    foreach (gen[i]) begin
        gen[i] = new(done[i]);
        gen[i].run();
    end

    foreach (gen[i])
        fork
            automatic int k = i;
            wait (done[k].triggered());
        join_none
    wait fork;
end
```
event e1, e2;

task trigger(event local_event, input time wait_time);
    #wait_time;
    ->local_event;
endtask // void

initial begin
    fork
        trigger(e1, 10ns);
        begin
            wait(e1.triggered());
            $display("%t: e1 triggered", $time);
        end
    join
end

initial begin
    fork
        trigger(e2, 20ns);
        begin
            wait(e2.triggered());
            $display("%t: e2 triggered", $time);
            $display("%t: e1 triggered", $time);
        end
    join
end
7.5 Semaphores

- Semaphores allow the testbench to control access to a resource.
- Think of a semaphore as a bucket containing 1 or more keys.
- Access to a resource requires a key.

- A thread that requests an unavailable key blocks.
- Multiple blocking threads are queued in FIFO order.
7.5.1 Semaphore Operations

• Declare a semaphore with `semaphore <name>;`
• Create a semaphore with `<name> = new(<# of keys>);`
• Get 1 or more keys with `<name>.get(<# of keys>);`
• Return 1 or more more keys with `<name>.put(<# of keys>);`
• Get 1 or more keys but do not block with `<name>.try_get(<# of keys>);`
program automatic test(bus_ifc.TB bus);
    semaphore sem;
    initial begin
        sem = new(1);
        fork
            sequencer();
            sequencer();
        join
    end

    task sequencer;
        repeat($urandom()%10) @bus.cb;
        sendTrans();
    endtask

    task sendTrans;
        sem.get(1);
        @bus.cb;
        bus.cb.addr <= t.addr;
        ... sem.put(1);
    endtask
7.5.2 Semaphores with Multiple Keys

Put allows more keys to be returned than were allocated.

```verilog
semaphore sem;
initial begin
    sem = new(1);
    #10ns;
    sem.put(2);
    #10ns;
    sem.get(2);
    #10ns;
    $display("%t: Done", $time);
end
```

30: Done
A get for less keys can bypass a blocked get

```vhdl
semaphore sem;

initial begin
    sem = new(2);
    sem.get(1);
    $display("%0t: Done0", $time);
    sem.get(2);
    $display("%0t: Done1", $time);
end

initial begin
    #50ns;
    sem.get(1);
    $display("%0t: Done2", $time);
end
```
Semaphore Exercise

1. Create a task called `wait10` that for 10 tries will wait for 10ns and then check for 1 semaphore key to be available. When the key is available quit the loop and print out the time.

```verilog
task wait10;
    int i= 10;
    do begin
        #10ns;
        i--;
    end
    while (((i!= 0) && !sem.try_get())); // Didn’t get key, keep trying
    $display("%0t: Done at i=%0d", $time, i);
endtask // wait10
```
Semaphore Exercise

2. What is the output with the following code?

```verilog
initial begin
  fork
    begin
      sem = new(1);
      sem.get(1);
      #45ns;
      sem.put(2);
    end
  wait10();
  join
end
```

# 50: Done at i=5    Since key not available until after 45ns
7.6 Mailboxes

• Mailboxes are used to pass information between 2 threads.
• Allows threads to operate autonomously and asynchronously

- Mailbox is an object but never contains objects
  - Must be instantiated by calling the new() function
- Think of a mailbox as a fifo with a source and sink
  - Source puts data in the mailbox
  - Sink takes data out of the mailbox.
  - Now the generator does not have to know anything about the driver, and visa versa
- Mailboxes can be sized or unlimited
• If a source tries to put into a full mailbox it is blocked
• If a sink tries to get from an empty mailbox it is blocked
Mailbox Operations

• Declare a mailbox:
  • mailbox [ #(type) ] <name>;
    • [ #(type) ] is optional
• Create a mailbox:
  • <name> = new (<size>);
    • If size is 0 or not specified, then unbounded
• Put 1 message (blocks if mailbox is full):
  • <name>.put(<singular message>);
• Get 1 message (blocks if mailbox is empty):
  • <name>.get(<singular message>);
• Get 1 message but do not block:
  • <name>.try_get(<singular message>);
• Put 1 message but do not block:
  • <name>.try_put(<singular message>);
• Find the number of messages in a mailbox:
  • <name>.num();
• Copy a message from the mailbox (does not remove from mailbox):
  • <name>.peek(<singular message>);
• Copy a message from the mailbox but does not block:
  • <name>.try_peek(<singular message>);
module mailboxes();
    mailbox #(int) m = new;

task generator();
    for(int i = 0; i<4; i++) begin
        m.put(i);
        $display("Generator put: %0d", i);
        #5;
    end
    endtask // generator

task receiver();
    int temp;

forever begin // or while(1) begin
    m.get(temp);
    $display("Receiver get: %0d", temp);
end
    endtask // receiver

initial begin
    fork
        generator();
        receiver();
    join_none
end
endmodule // mailboxes
Declaring the type of a mailbox

- The default mailbox is typeless
- Can result in run-time errors between the mailbox and the type used to retrieve the message.
- Recommended to specify the type of your mailbox

```vhdl
MemTrans MyMemTrans;
Transaction MyTransaction;
mailbox mbx;

initial begin
  mbx=new;
  MyMemTrans = new();
  MyTransaction = new();
  mbx.put(MyMemTrans);
  mbx.get(MyTransaction);
end
```

** Warning: mailboxType.sv(13): (vlog-2181) Use of a parameterized class mailbox as a type creates a default specialization.

Error: put is one data type and get is another.
Declaring the type of a mailbox

```verilog
package types;
    typedef struct {
        int pid;
    } packet;
endpackage // types

module mbox;
import types::*;

mailbox #(packet) exp_mbox = new;
mailbox #(packet) act_mbox = new;
int error_cnt = 0;

task stimulus();
    packet stim_pkt;
    for (int i = 0; i < 256; i++) begin
        stim_pkt.pid = i;
        $display("Sending pkt: ", i);
        act_mbox.put(stim_pkt);
        exp_mbox.put(stim_pkt);
    end
endtask // for

task chkr();
    packet exp_pkt, act_pkt;
    while(1) begin
        exp_mbox.get(exp_pkt);
        act_mbox.get(act_pkt);
        if(exp_pkt != act_pkt )
            error_cnt++;
        if(act_pkt.pid == 255)
            break;
        $display("Sending pkt: ", i);
    end
endtask // while

task initial();
    fork
        stimulus();
        chkr();
    join_none
    end
endmodule // mbox
```

# Sending pkt: 0
# Sending pkt: 1
# Sending pkt: 2
# Sending pkt: 3
# Sending pkt: 4
# Sending pkt: 5
# Sending pkt: 6
# Sending pkt: 7
# Sending pkt: 8
# Sending pkt: 9
...
# Sending pkt: 251
# Sending pkt: 252
# Sending pkt: 253
# Sending pkt: 254
# Sending pkt: 255
# Finished: 0 errors
task generator_bad(input int n, 
    mailbox #(Transaction) mbx);

Transaction tr;
tr = new();
repeat (n) begin
    `SV_RAND_CHECK(tr.randomize());
    $display("GEN: Sending addr=%h", tr.addr);
    mbx.put(tr);
end
endtask
Mailbox Example 2

task generator_good(input int n, mailbox #(Transaction) mbx);
Transaction tr;
repeat (n) begin
    tr = new();
    `SV_RAND_CHECK(tr.randomize());
    $display("GEN: Sending addr=%h", tr.addr);
    mbx.put(tr);
end
endtask
Mailbox driver example

task driver(input mailbox #(Transaction) mbx);
    Transaction tr;
    forever begin
        mbx.get(tr);
        $display("DRV: Received addr=%h", tr.addr);
        // Drive transaction into DUT
    end
endtask
class Generator;

    // Handle to a transaction
    Transaction tr;

    // Declare a handle to a mailbox that holds objects of type transaction.
    mailbox #(Transaction) mbx;

    // Custom constructor for the Generator. Argument to the constructor is variable mbx of mailbox type.
    function new(input mailbox #(Transaction) mbx);
    // Assign passed in handle mbx to handle mbx in the class.
    this.mbx = mbx; // End function

    // Declare task run that has one argument, an integer count
    task run(input int count);

    // Create count objects, randomize them, and put them in the mailbox. Note we have no idea what size the mailbox is.
    repeat (count) begin
        tr = new();
        `SV_RAND_CHECK(tr.randomize);
        mbx.put(tr);
    end
    endtask

endclass
7.6.1 Mailbox in a Testbench - driver

```verbatim
// Driver gets transactions from the generator and drives the DUT's interface
class Driver;
  Transaction tr;
  mailbox #(Transaction) mbx;

  // Custom constructor for the mailbox. Argument to the constructor is variable mbx of mailbox type.
  function new(input mailbox #(Transaction) mbx);
    // Assign passed in handle mbx to handle mbx in the class.
    this.mbx = mbx;
  endfunction

  // Declare task run that has one argument, an integer count
  task run(input int count);
  // Fetch count transaction objects from the mailbox and drive the interface with a value. This is the pin level driver.
  // Note we still have no idea what size the mailbox is.
    repeat (count) begin
      mbx.get(tr); // Drive transaction here
    end
  endtask
endclass
```
7.6.1 Mailbox in a Testbench - program

```verilog
// Declare a program and pass in an interface with modport TB called bus
program automatic mailbox_example(bus_if.TB bus, ...);

// Include the transaction, generator, and driver class definitions. Need to do this here so the generator and driver class know //about the bus interface. When we learn about virtual interfaces in Chapter 10 we will see how to eliminate these includes.
'include "transaction.sv"
'include "generator.sv"
'include "driver.sv"

// Declare a variable mbx of type mailbox connecting gen & drv
mailbox #(Transaction) mbx;

Generator gen; // Declare variable gen of type Generator
Driver drv; // Declare variable drv of type Driver
int count;
initial begin
    // Construct the mailbox. Since new has no arguments the mailbox is of unbounded size.
    mbx = new();
    // Construct the generator and pass in the handle to the mailbox to the custom constructor
    gen = new(mbx);
    // Construct the driver and pass in the handle to the mailbox to the custom constructor
    drv = new(mbx);

    count = $urandom_range(50); // Randomize count to be in the range from 1 to 50
    fork
        gen.run(count); // Spawn the generator
        drv.run(count); // Spawn the driver
    Join // Wait for both the generator and driver to finish
end
endprogram
```

initial begin
  mbox = new(1);
  fork
    for (int i=1; i<4; i++) begin
      $display("Producer: before put(%0d)", i);
      mbox.put(i);
      $display("Producer: after put(%0d)", i);
    end
  repeat(4) begin
    mbox.get(j);
    $display("%0t: Consumer: Working on (%0d)", $time, j);
    #10ns;
    $display("%0t: Consumer: Done Working on (%0d)", $time, j);
  end
  join
end
Output:

```
0: Producer: before put(1)
0: Producer: after put(1)
0: Producer: before put(2)
0: Producer: after put(2)
0: Producer: before put(3)
0: Consumer: Working on (1)
10: Consumer: Done Working on (1)
10: Consumer: Working on (2)
10: Producer: after put(3)
20: Consumer: Done Working on (2)
20: Consumer: Working on (3)
30: Consumer: Done Working on (3)
```
mailbox #(int) mbx;
int value;

initial begin
    mbx = new(1);
    $display("mbx.num()=%0d", mbx.num());
    $display("mbx.try_get= %0d", mbx.try_get(value));
    mbx.put(2);
    $display("mbx.try_put= %0d", mbx.try_put(value));
    $display("mbx.num()=%0d", mbx.num());
    mbx.peek(value);
    $display("value=%0d", value);
end
7.6.3 Unsynchronized Threads
Communicating with a Mailbox

• In many cases threads should run in lockstep
• The highest level producer completes only when the lowest level consumer completes
• Possible solutions:
  a) Use a bounded mailbox, `peek()` to copy a message if available, and `get()` only after consumer is done processing.
  b) Use an unbounded mailbox and an event triggered by the consumer to cause the producer to block.
  c) Use a 2\textsuperscript{nd} mailbox for the consumer to send a completion message to the producer.
• Will discuss first approach in class. Other two are covered in book.
program automatic synch_peek;

// Declare a global mailbox called mbx of type int. This means we don't need a custom constructor that creates another handle to the shared mailbox but it means the class declarations need to occur within the program so the classes knows about mbox.
mailbox #(int) mbx;

class Consumer; // Consumer class definition
    task run();

    int i; // Declare integer i that will be the message passed from producer to consumer
    repeat (3) begin
        // Peek integer from mbx. Peek blocks so execution will stop here if no message // is available. Copy the message into integer i but do not remove it from the mailbox
        mbx.peek(i);
        $display("%0t: Consumer: Working on (%0d)", $time, i);
        #10ns; // Wait for 10ns to cause consumer to take time for processing
        $display("%0t: Consumer: Done Working on (%0d)", $time, i);
        mbx.get(i); // Remove from mbx
    end
endtask
endclass
.....
class Producer;
    task run();
        for (int i=1; i<4; i++) begin  // Produce 3 transactions
            $display("%0t: Producer: before put(%0d)", $time, i);
            mbx.put(i);  // As is blocking, “put” in the loop will only advance if
                // there is room in the mailbox
            $display("%0t: Producer: after put(%0d)", $time, i);
        end
    endtask
endclass
7.6.4 Synchronized Threads Using a Bounded Mailbox and Peek (cont.)

Producer p;
Consumer c;
initial begin
    mbx = new(1);
p = new();
c = new();
fork
    p.run();
c.run();
join
end
endprogram

Bounded mailbox of size 1

0: Producer: before put(1)
0: Producer: after put(1)
0: Producer: before put(2)
0: Consumer: Working on (1)
10: Consumer: Done Working on (1)
10: Producer: after put(2)
10: Producer: before put(3)
10: Consumer: Working on (2)
20: Consumer: Done Working on (2)
20: Producer: after put(3)
20: Consumer: Working on (3)
30: Consumer: Done Working on (3)
7.7 Building a Testbench w/ Threads & IPC

[Diagram showing the components of a testbench with threads and IPC:
- Generator
  - gen2agt mailbox
- Agent
  - agt2drv mailbox
- Driver
- Scoreboard
- Checker
  - mon2chk mailbox
- Monitor
- DUT
  - Environment]
class Agent;
    mailbox #(Transaction) gen2agt, agt2drv; // Declare 2 mailboxes handles
    Transaction tr; // Declare a handle to a transaction class

    // Custom constructor that sets the local mailbox handles to the passed in mailbox
    // handles. We’ve seen this before as a method to sharing a mailbox.
    function new(mailbox #(Transaction) gen2agt, agt2drv);
        this.gen2agt = gen2agt;
        this.agt2drv = agt2drv;
    endfunction

task run();
    forever begin // As the agent is a block in the middle of generator and driver its run task never stops.
        gen2agt.get(tr); // Get transaction from upstream generator block
        ... // Do some processing
        agt2drv.put(tr); // Send it to downstream driver block
    end
endtask

    task wrap_up(); // empty wrap up task
endtask
endclass
class Config;
    // Unsigned integer indicating the # of transactions to generate
    rand bit [31:0] run_for_n_trans;
    constraint reasonable
        // Constrain the # of transactions from 1 to 1000
        {run_for_n_trans inside {[1:1000]};};
endclass
class Environment; // Declare handles to each class
  Generator gen;
  Agent agt;
  Driver drv;
  Monitor mon;
  Checker chk;
  Scoreboard scb;
  Config cfg; // Configuration class handle. Not in the block diagram.
  // Handles to all 3 mailboxes
  mailbox #(Transaction) gen2agt, agt2drv, mon2chk;
  // These are all functions that the Environment class will use. They are declared as extern so they can be defined outside the class. We’ll look at what each of these does in following slides.
  extern function new();
  extern function void gen_cfg();
  extern function void build();
  extern task run();
  extern task wrap_up();
endclass
// Custom constructor that creates a new configuration object
function Environment::new();
    cfg = new();
Endfunction

// Using the cfg object call randomize. Important to build the configuration object and call randomize first before build in case the values randomized impact the build function.
function void Environment::gen_cfg;
    `SV_RAND_CHECK(cfg.randomize);
endfunction

function void Environment::build(); // Build creates objects from all the classes
    gen2agt = new(); // Create mailboxes
    agt2drv = new(); // Create mailboxes
    mon2chk = new(); // Create mailboxes
    // Initialize transactors by passing in a mailbox to each of their custom constructors.
    gen = new(gen2agt);
    agt = new(gen2agt, agt2drv);
    drv = new(agt2drv);
    mon = new(mon2chk);
    chk = new(mon2chk);
    scb = new(); // This is the scoreboard.
endfunction
// In this task call run for each object in parallel. Note that only the generator and the scoreboard needs to know how many transactions will be created. The reason the scoreboard needs to know is so it can determine if it received all the transactions.

```cpp
task Environment::run();
    fork
        gen.run(cfg.run_for_n_trans);
        agt.run();
        drv.run();
        mon.run();
        chk.run();
        scb.run(cfg.run_for_n_trans);
    join
endtask
```

// Run each objects wrap up function in parallel
```cpp
task Environment::wrap_up();
    fork
        gen.wrap_up();
        agt.wrap_up();
        drv.wrap_up();
        mon wrap_up();
        chk.wrap_up();
        scb.wrap_up();
    join
endtask
```
7.7.4 Test Program

program automatic test;
    Environment env;
    initial begin
        env = new(); // Call the environment’s custom constructor which creates the configuration object.
        env.gen_cfg(); // Now that the configuration object is constructed, randomize the configuration
        env.build(); // Create all the mailboxes and objects of the classes
        env.run(); // Start each object by calling each object’s run task
        env.wrap_up(); // Wrap up each object by calling each object’s wrap_up task.
    end
endprogram
Create the Monitor class in the diagram assuming DUT binary outputs \textit{out1} and \textit{out2} and that the class is aware of bus interface \textit{my\_bus}.
// Class OutputTrans may not be required if the monitor uses the Transaction object from the Generator to store results into.
class OutputTrans;
  bit out1;
  bit out2;
endclass

class Monitor;
  OutputTrans tr; // or Transaction tr
  mailbox #(OutputTrans) mbx;
  function new(mailbox #(OutputTrans) mbx);
    this.mbx = mbx; // handle to shared mailbox mon2chk
  endfunction
  task run();
    forever begin
      tr = new();
      @my_bus.cb;
      tr.out1 = my_bus.out1;
      tr.out2 = my_bus.out2;
      mbx.put(tr);
    end
  endtask
  task wrap_up();
  endtask
endclass