Chapter 7
Threads and Interprocess Communication

In real hardware, the sequential logic is activated on clock edges, whereas combinational logic is constantly changing when any inputs change. All this parallel activity is simulated in Verilog RTL using `initial` and `always` blocks, plus the occasional gate and continuous assignment statement. To stimulate and check these blocks, your testbench uses many threads of execution, all running in parallel. Most blocks in your testbench environment are modeled with a transactor and run in their own thread.

The SystemVerilog scheduler is the traffic cop that chooses which thread runs next. You can use the techniques in this chapter to control the threads and thus your testbench.

Each of these threads communicates with its neighbors. In Fig. 7.1, the generator passes the stimulus to the agent. The environment class needs to know when the generator completes and then tell the rest of the testbench threads to terminate. This is done with interprocess communication (IPC) constructs such as the standard Verilog events, event control and `wait` statements, and the SystemVerilog mailboxes and semaphores.[1]

[1]The SystemVerilog LRM uses "thread" and "process" interchangeably. The term "process" is most commonly associated with Unix processes, in which each contains a program running in its own memory space. Threads are lightweight processes that may share common code and memory, and consume far fewer resources than a typical process. This book uses the term "thread." However, "interprocess communication" is such a common term that it is used in this book.

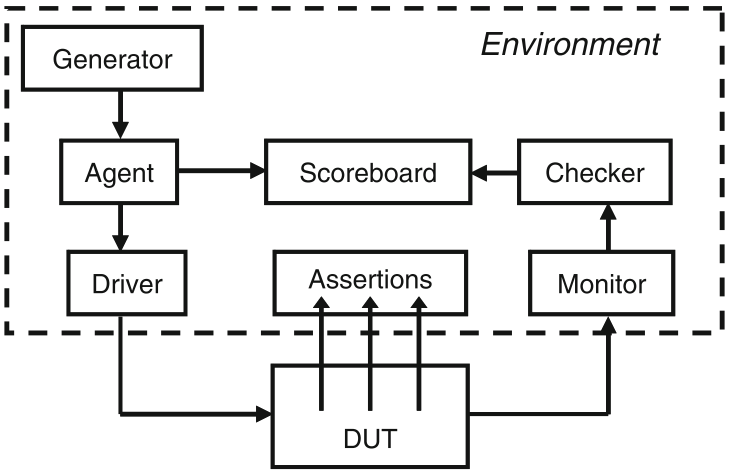

Fig. 7.1 Testbench environment blocks

7.1 Working with Threads

While all the thread constructs can be used in both modules and program blocks, your testbenches belong in program blocks. As a result, your code always starts with `initial` blocks that start executing at time 0. You cannot put an `always` block in a program. However, you can easily get around this by using a `forever` loop in an `initial` block.

Classic Verilog has two ways of grouping statements — with a `begin...end` or `fork...join`. Statements in a `begin...end` run sequentially, whereas those in a `fork...join` execute in parallel. The latter is very limited in that all statements inside the `fork...join` have to finish before the rest of the block can continue. As a result, it is rare for Verilog testbenches to use this feature.

SystemVerilog introduces two new ways to create threads — with the `fork...join_none` and `fork...join_any` statements, shown in Fig. 7.2.

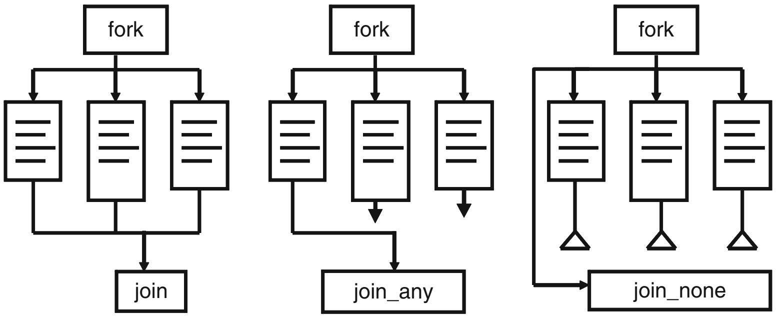

Fig. 7.2 `Fork...join` blocks

Your testbench communicates, synchronizes, and controls these threads with existing constructs such as events, @ event control, the `wait` and `disable` statements, plus new language elements such as semaphores and mailboxes.

7.1.1 *Using* `fork...join` *and* `begin...end`

Sample 7.1 has a `fork...join` parallel block with an enclosed `begin...end` sequential block, and shows the difference between the two.

Sample 7.1 Interaction of `begin...end` and `fork...join`

```
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
```

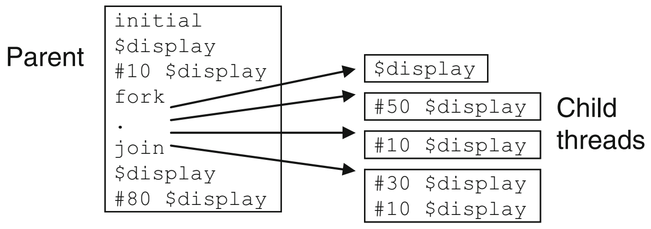

Fig. 7.3 `Fork...join` block

In the output below, the code in the `fork...join` executes in parallel, so statements with shorter delays execute before those with longer delays. As shown in Sample 7.2, the `fork...join` completes after the last statement, which starts with `#50`.

Sample 7.2 Output from `begin...end` and `fork...join`

```
@0: start fork...join example
@10: sequential after #10
@10: parallel start
@20: parallel after #10
@40: sequential after #30
@50: sequential after #10
@60: parallel after #50
@60: after join
@140: finish after #80
```

7.1.2 *Spawning Threads with* `fork...join_none`

A `fork...join_none` block schedules each statement in the block, but execution continues in the parent thread. Sample 7.3 is identical to Sample 7.1 except that the `join` has been converted to `join_none`.

Sample 7.3 `Fork...join_none` code

```
initial begin
  $display("@%0t: start fork...join_none 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
```

The diagram for this block is similar to Fig. 7.3. Note that the statement after the `join_none` block in Sample 7.4 executes before any statement inside the `fork… join_none`.

Sample 7.4 `Fork…join_none` output

```
@0: start fork...join_none example
@10: sequential after #10
@10: after join_none
@10: parallel start
@20: parallel after #10
@40: sequential after #30
@50: sequential after #10
@60: parallel after #50
@90: finish after #80
```

7.1.3 *Synchronizing Threads with* `fork…join_any`

A `fork…join_any` block schedules each statement in the block. Then, when the first statement completes, execution continues in the parent thread. All other remaining threads continue. Sample 7.5 is identical to the previous examples, except that the `join` has been converted to `join_any`.

Sample 7.5 `Fork…join_any` code

```
initial begin
  $display("@%0t: start fork...join_any 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
```

Note in Sample 7.6, the statement `$display("after join_any")` completes after the first statement in the parallel block.

Sample 7.6 Output from `fork…join_any`

```
@0: start fork...join_any example
@10: sequential after #10
@10: parallel start
@10: after join_any
@20: parallel after #10
@40: sequential after #30
@50: sequential after #10
@60: parallel after #50
@90: finish after #80
```

7.1.4 Creating Threads in a Class

You can use a `fork…join_none` to start a thread, such as the code for a random transactor generator. Sample 7.7 shows a generator / driver class with a `run` task that creates N packets. The full testbench has classes for the driver, monitor, checker, and more, all with transactors that need to run in parallel.

Sample 7.7 Generator / Driver class with a run task

```
class Gen_drive;

  // Transactor that creates N packets
  task run(input int n);
    Packet p;

    fork
      repeat (n) begin
        p = new();
        `SV_RAND_CHECK(p.randomize());
        transmit(p);
      end
    join_none      // Use fork-join_none so run() does not block
  endtask

  task transmit(input Packet p);
  ...
  endtask
endclass

Gen_drive gen;

initial begin
  gen = new();
  gen.run(10);
  // Start the checker, monitor, and other threads
  ...
end
```

There are several points to note with Sample 7.7. First, the transactor is not started in the `new()` function. The constructor should just initialize values, not start any threads. Separating the constructor from the code that does the real work allows you to change any variables before you start executing the code in the object. This allows you to inject errors, modify the defaults, and alter the behavior of the object. Next, the `run` task starts a thread in a `fork…join_none` block. The thread is a part of the transactor and should be spawned there, not in the parent class.

7.1.5 Dynamic Threads

Verilog's threads are very predictable. You can read the source code and count the `initial`, `always`, and `fork…join` blocks to know how many threads were in a module. On the other hand, SystemVerilog lets you create threads dynamically, and does not require you to wait for them to finish.

In Sample 7.8, the testbench generates random transactions and sends them to a DUT that stores them for some predetermined time, and then returns them. The testbench has to wait for the transaction to complete, but does not want to stop the generator.

Sample 7.8 Dynamic thread creation

```
program automatic test(bus_ifc.TB bus);
  // Code for interface not shown
  task check_trans(input Transaction tr);
    fork
      begin
      wait (bus.cb.data == tr.data);
      $display("@%0t: data match %d", $time, tr.data);
      end
    join_none    // Spawn thread, don't block
  endtask

  Transaction tr;

  initial begin
    repeat (10) begin
      tr = new();              // Create a random transaction
      `SV_RAND_CHECK(tr.randomize());
      transmit(tr);            // Send transaction into the DUT
      check_trans(tr);         // Wait for reply from DUT
    end
    #100; // Wait for final transaction to complete
  end
endprogram
```

When the check_trans task is called, it spawns off a thread to watch the bus for the matching transaction data. During a normal simulation, many of these threads run concurrently. In this simple example, the thread just prints a message, but you could add more elaborate controls.

7.1.6 Automatic Variables in Threads

A common but subtle bug occurs when you have a loop that spawns threads and you don't save variable values before the next iteration. Sample 7.8 only works in a program or module with automatic storage. If check_trans used static storage, each thread would share the same variable tr, so later calls would overwrite the value set by earlier ones. Likewise, if the example had the fork...join_none inside the repeat loop, it would try to match incoming transactions using tr, but its value would change the next time through the loop. Always use automatic variables to hold values in concurrent threads.

Sample 7.9 has a fork...join_none inside a for loop. SystemVerilog schedules the threads inside a fork...join_none but they are not executed until after the original code blocks, here because of the #0 delay. So Sample 7.9 prints "3 3 3" which are the values of the index variable j when the loop terminates.

Sample 7.9 Bad fork...join_none inside a loop

```
program no_auto;
  initial begin
    for (int j=0; j<3; j++)
      fork
        $write(j);  // Bug - prints final value of index
      join_none
    #0 $display;
  end
endprogram
```

Sample 7.10 Execution of bad `fork...join_none` inside a loop

```
j   Statement        Comment
0   for (j=0; ...
0   Spawn $write(j)  [thread 0]
1   j++              j=1
1   Spawn $write(j)  [thread 1]
2   j++              j=2
2   Spawn $write(j)  [thread 2]
3   j++              j=3
3   join_none
3   #0               Delay before $display
3   $write(j)        [thread 0: j=3]
3   $write(j)        [thread 1: j=3]
3   $write(j)        [thread 2: j=3]
3   $display;
```

The `#0` delay blocks the current thread and reschedules it to start later during the current time slot. In Sample 7.10, the delay makes the current thread run after the threads spawned in the `fork...join_none` statement. This delay is useful for blocking a thread, but you should be careful, as excessive use causes race conditions and unexpected results.

You should use `automatic` variables inside a `fork...join` statement to save a copy of a variable as shown in Sample 7.11.

Sample 7.11 Automatic variables in a `fork...join_none`

```
initial begin
  for (int j=0; j<3; j++)
    fork
      automatic int k = j;    // Make copy of index
      begin
        $write(k);            // Print copy
      end
    join_none
  #0 $display;
end
```

The `fork...join_none` block is split into two parts, declarations and procedural code. The automatic variable declaration with initialization runs in the thread inside the `for` loop. During each loop, a copy of `k` is created and set to the current value of `j`. Then the body of the `fork...join_none ($write)` is scheduled, including a copy of `k`. After the loop finishes, `#0` blocks the current thread, so the three threads run, printing the value of their copy of `k`. When the threads complete, and there is nothing else left during the current time-slot region, SystemVerilog advances to the next statement and the `$display` executes.

Sample 7.12 traces the code and variables from Sample 7.11. The three copies of the automatic variable `k` are called `k0`, `k1`, and `k2` for this sample.

Sample 7.12 Steps in executing automatic variable code

```
j k0 k1 k2  Statement
0           for (j=0; ...
0  0        Create k0, spawn $write(k) [thread 0]
1  0        j++
1  0  1     Create k1, spawn $write(k) [thread 1]
2  0  1     j++
2  0  1  2  Create k2, spawn $write(k) [thread 2]
3  0  1  2  j<3
3  0  1  2  join_none
3  0  1  2  #0
3  0  1  2  $write(k0)       [thread 0]
3  0  1  2  $write(k1)       [thread 1]
3  0  1  2  $write(k2)       [thread 2]
3  0  1  2  $display;
```

Another way to write Sample 7.11 is to declare the automatic variable outside of the `fork...join_none`. Sample 7.13 works inside a program with automatic storage.

Sample 7.13 Automatic variables in a `fork...join_none`

```
program automatic bug_free;
  initial begin
    for (int j=0; j<3; j++) begin
      automatic int k = j;       // Make copy of index
      fork
        begin
          $write(k);           // Print copy
        end
      join_none
    end
    #0 $display;               // New line after all threads end
  end
endprogram
```

7.1.7 *Waiting for all Spawned Threads*

In SystemVerilog, when all the `initial` blocks in the program are done, the simulator exits. Sample 7.14 shows how you can spawn many threads, which might still be running. Use the `wait fork` statement to wait for all child threads.

Sample 7.14 Using `wait fork` to wait for child threads

```
task run_threads();
  ...                   // Create some transactions
  fork
    check_trans(tr1);   // Spawn first thread
    check_trans(tr2);   // Spawn second thread
    check_trans(tr3);   // Spawn third thread
  join_none
  ...                   // Do some other work

  // Now wait for the above threads to complete
  wait fork;
endtask
```

7.1.8 Sharing Variables Across Threads

Inside a class's routines, you can use local variables, class variables, or variables defined in the program. If you forget to declare a variable, SystemVerilog looks up the higher scopes until it finds a match. This can cause subtle bugs if two parts of the code are unintentionally sharing the same variable, perhaps because you forgot to declare it in the innermost scope.

For example, if you like to use the index variable, `i`, be careful that two different threads of your testbench don't concurrently modify this variable by each using it in a `for` loop. Or you may forget to declare a local variable in a class, such as `Buggy`, shown below. If your program block declares a global `i`, the class just uses the global instead of the local that you intended. You might not even notice this unless two parts of the program try to modify the shared variable at the same time.

Sample 7.15 Bug using shared program variable

```
program automatic bug;

  class Buggy;
    int data[10];
    task transmit();
      fork
        for (i=0; i<10; i++)  // i is not declared here
          send(data[i]);
      join_none
    endtask
  endclass

  int i;                      // Program-level i, shared
  Buggy b;
  event receive;

  initial begin
    b = new();
    for (i=0; i<10; i++)      // i is not declared here
      b.data[i] = i;
    b.transmit();

    for (i=0; i<10; i++)      // i is not declared here
      @(receive) $display(b.data[i]);
    end
endprogram
```

The solution is to declare all your variables in the smallest scope that encloses all uses of the variable. In Sample 7.15, declare index variables inside the `for` loops, not at the program or class level. Better yet, use the `foreach` statement whenever possible.

7.2 Disabling Threads

Just as you need to create threads in the testbench, you also need to stop them. The Verilog `disable` statement works on SystemVerilog threads. The following sections show how you can asynchronously disable threads. This can cause unexpected behavior, so you should watch out for side effects when a thread is stopped midstream. You may, instead, want to design your algorithm to check for interrupts at stable points, then gracefully give up its resources.

7.2.1 *Disabling a Single Thread*

Here is the `check_trans` task, this time using a `fork…join_any` plus a `disable` to create a watch with a time-out. In this case, you are disabling a labelled block, to precisely specify what to stop.

The outermost `fork…join_none` is identical to Sample 7.8. This version implements a time-out with two threads inside a `fork…join_any` so that the simple `wait` statement is executed in parallel with a delayed `$display`. If the correct bus data comes back quickly enough, the `wait` construct completes, the `join_any` executes, and then the `disable` kills off the remaining thread. However, if the bus data does not get the right value before the `TIME_OUT` delay completes, the error message is printed, the `join_any` executes, and the `disable` kills the thread with the `wait`.

Sample 7.16 Disabling a thread

```
parameter TIME_OUT = 1000ns;

task check_trans(input Transaction tr);
  fork

    begin
      // Wait for response, or some maximum delay
      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

  join_none    // Spawn thread, don't block
endtask
```

Watch out, as you might unintentionally stop too many threads with `disable` label. This statement stops every process executing the labeled block, as might occur if you have multiple driver or monitor objects running in parallel. If your code only has one instance, `disable` label is a safe way to stop a thread.

7.2.2 *Disabling Multiple Threads*

Sample 7.16 used the classic Verilog `disable` statement to stop the threads in a named block. SystemVerilog introduces the `disable fork` statement so you can stop all child threads that have been spawned from the current thread.

Watch out, as you might unintentionally stop too many threads with `disable fork`, such as those created from surrounding task calls. You should always surround the target code with a `fork…join` to limit the scope of a `disable fork` statement.

The next few samples use the `check_trans` task from Sample 7.16. You can just think of this task as doing a `#TIME_OUT`. Sample 7.17 has an additional `begin…end` block inside the `fork…join` to make the statements sequential.

Sample 7.17 Limiting the scope of a disable fork

```
initial begin
  check_trans(tr0);          // thread 0

  // Create a thread to limit scope of disable
  fork                       // thread 1
    begin
      check_trans(tr1);      // thread 2
      fork                   // thread 3
        check_trans(tr2);    // thread 4
      join

      // Stop threads 2-4, but leave 0 alone
      #(TIME_OUT/10) disable fork;
    end
  join
end
```

Fig. 7.4 shows a diagram of the spawned threads.

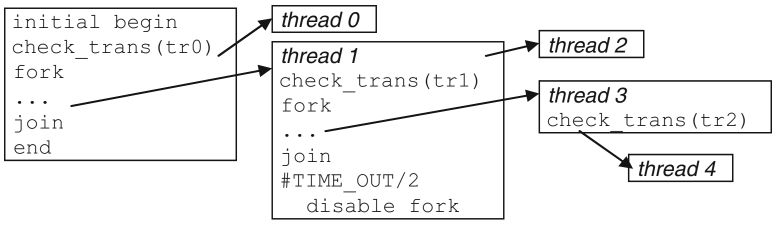

Fig. 7.4 `Fork…join` block diagram

The code calls `check_trans` that starts thread 0. Next a `fork…join` creates thread 1. Inside this thread, one is spawned by the `check_trans` task and one by the innermost `fork…join`, which spawns thread 4 by calling the task. After a delay, a `disable fork` stops and all the child threads, 2-4. Thread 0 is outside the `fork…join` block that has the `disable`, so it is unaffected.

Sample 7.18 is the more robust version of Sample 7.17, with `disable` with a label that explicitly names the threads that you want to stop.

Sample 7.18 Using `disable` label to stop threads

```
initial begin
  check_trans(tr0);      // thread 0
  fork                   // thread 1
    begin : threads_inner
      check_trans(tr1);  // thread 2
      check_trans(tr2);  // thread 3
    end

    // Stop threads 2 & 3, but leave 0 alone
    #(TIME_OUT/10) disable threads_inner;
  join
end
```

7.2.3 Disable a Task that was Called Multiple Times

Be careful when you disable a block from inside that block - you might end up stopping more than you expected. As expected, if you disable a task from inside the task, it is like a return statement, but it also kills all threads started by the task. Additionally, a single `disable` label terminates all threads using that code, not just the current one.

In Sample 7.19, the `wait_for_time_out` task is called three times, spawning three threads. Then, thread 0 also disables the task after #2ns. When you run this code, you will see the three threads starting, but none finishes, because of the `disable` in thread 0 stops all three threads, not just one. If this task was inside a driver class that was instantiated multiple times, a `disable` label in one could stop all the blocks.

Sample 7.19 Using `disable` label to stop a task

```
task wait_for_time_out(input int id);
  if (id == 0)
    fork
      begin
        #2ns;
        $display("@%0t: disable wait_for_time_out", $time);
        disable wait_for_time_out;
      end
    join_none

  fork : just_a_little
    begin
      $display("@%0t: %m: %0d entering thread", $time, id);
      #TIME_OUT;
      $display("@%0t: %m: %0d done", $time, id);
    end
  join_none
endtask
```

```
initial begin
  wait_for_time_out(0); // Spawn thread 0
  wait_for_time_out(1); // Spawn thread 1
  wait_for_time_out(2); // Spawn thread 2
  #(TIME_OUT*2) $display("@%0t: All done", $time);
end
```

7.3 Interprocess Communication

All these threads in your testbench need to synchronize and exchange data. At the most basic level, one thread waits for another, such as the environment object waiting for the generator to complete. Multiple threads might try to access a single resource such as bus in the DUT, so the testbench needs to ensure that one and only one thread is granted access. At the highest level, threads need to exchange data such as transaction objects that are passed from the generator to the agent. All of this data exchange and control synchronization is called interprocess communication (IPC), which is implemented in SystemVerilog with events, semaphores, and mailboxes. These are described in the remainder of this chapter.

There are generally three parts to IPC: a producer that creates the information, a consumer that accepts the information, and the channel that carries the information. The producer and consumer are in separate threads.

7.4 Events

A Verilog event synchronizes threads. It is similar to a phone, where one person waits for a call from another person. In Verilog a thread waits for an event with the `@` operator. This operator is edge sensitive, so it always blocks, waiting for the event to change. Another thread triggers the event with the `->` operator, unblocking the first thread.

System Verilog enhances the Verilog event in several ways. An event is now a handle to a synchronization object that can be passed around to routines. This feature allows you to share events across objects without having to make the events global. The most common way is to pass the event into the constructor for an object.

There is always the possibility of a race condition in Verilog where one thread blocks on an event at the same time another triggers it. If the triggering thread executes before the blocking thread, the trigger is missed. SystemVerilog introduces the `triggered` status that lets you check whether an event has been triggered, including during the current time-slot. A thread can wait on this function instead of blocking with the `@` operator.

7.4.1 Blocking on the Edge of an Event

When you run Sample 7.20, one initial block starts, triggers its event, and then blocks on the other event, as shown in the output in Sample 7.21. The second block starts, triggers its event (waking up the first), and then blocks on the first event. However, the second thread locks up because it missed the first event, as it is a zero-width pulse.

Sample 7.20 Blocking on an event in 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
```

Sample 7.21 Output from blocking on an event

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

7.4.2 Waiting for an Event Trigger

Instead of the edge-sensitive block `@e1`, use the level-sensitive `wait(e1.triggered)`. This does not block if the event has been triggered during this time step. Otherwise, it waits until the event is triggered.

Sample 7.22 Waiting for an event

```
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
```

When you run Sample 7.22, one initial block starts, triggers its event, and then blocks on the other event. The second block starts, triggers its event (waking up the first) and then blocks on the first event, producing the output in Sample 7.23.

Sample 7.23 Output from waiting for an event

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

Several of these samples have race conditions and may not execute exactly the same on every simulator. For example, the output in Sample 7.23 assumes that when the second block triggers `e2`, execution jumps back to the first block. It would also be legal for the second block to trigger `e2`, wait on `e1`, and display a message before control is returned back to the first block.

7.4.3 *Using Events in a Loop*

You can synchronize two threads with an event, but use caution.

If you use `wait (handshake.triggered)` in a loop, be sure to advance the time before waiting again. Otherwise your code will go into a zero delay loop as the `wait` continues over and over again on a single event trigger. Sample 7.24 incorrectly uses a level-sensitive blocking statement for notification that a transaction is ready.

Sample 7.24 Waiting on event causes a zero delay loop

```
forever begin
  // This is a zero delay loop!
  wait(handshake.triggered);
  $display("Received next event");
  process_in_zero_time();
end
```

Just as you learned to always put a delay inside an `always` block you need to put a delay in a transaction process loop. The edge-sensitive delay statement in Sample 7.25 continues once and only once per event trigger.

Sample 7.25 Waiting for an edge on an event

```
forever begin
  // This prevents a zero delay loop!
  @handshake;
  $display("Received next event");
  process_in_zero_time();
end
```

You should avoid events if you need to send multiple notifications in a single time slot, and look at other IPC methods with built-in queuing such as semaphores and mailboxes, discussed later in this chapter.

7.4.4 Passing Events

As described above, an event in SystemVerilog can be passed as an argument to a routine. In Sample 7.26, an event is used by a transactor to signal when it has completed.

Sample 7.26 Passing an event into a constructor

```
program automatic test;

class Generator;
  event done;
  function new (input event done); // Pass event from TB
    this.done = done;
  endfunction

  task run();
    fork
      begin
      ...                          // Create transactions
      -> done;                     // Tell the test we are done
      end
    join_none
  endtask
endclass

  event gen_done;
  Generator gen;

  initial begin
    gen = new(gen_done);           // Instantiate testbench
    gen.run();                     // Run transactor
    wait(gen_done.triggered);      // Wait for finish
  end
endprogram
```

7.4.5 Waiting for Multiple Events

In Sample 7.26, you had a single generator that fired a single event. What if your testbench environment class must wait for multiple child processes to finish, such as N generators? The easiest way is to use `wait fork`, that waits for all child processes to end. The problem is that this also waits for all the transactors, drivers, and any other threads that were spawned by the environment. You need to be more selective. You still want to use events to synchronize between the parent and child threads.

You could use a `for` loop in the parent to wait for each event, but that would only work if thread 0 finished before thread 1, which finished before thread 2, etc. If the threads finish out of order, you could be waiting for an event that triggered many cycles ago.

The solution is to make a new thread and then spawn children from there that each block on an event for each generator, as shown in Sample 7.27. Now you can do a `wait fork` because you are being more selective.

Sample 7.27 Waiting for multiple threads with wait fork

```
event done[N_GENERATORS];

initial begin
  foreach (gen[i]) begin
    gen[i] = new(done[i]); // Create N generators
    gen[i].run();          // Start them running
  end

  // Wait for all gen to finish by waiting for each event
  foreach (gen[i])
    fork
      automatic int k = i;
      wait (done[k].triggered);
    join_none

  wait fork;  // Wait for all those triggers to finish
end
```

Another way to solve this problem is to keep track of the number of events that have triggered, as shown in Sample 7.28.

Sample 7.28 Waiting for multiple threads by counting triggers

```
event done[N_GENERATORS];
int done_count;

initial begin
  foreach (gen[i]) begin
    gen[i] = new(done[i]); // Create N generators
    gen[i].run();          // Start them running
  end

  // Wait for all generators to finish
  foreach (gen[i])
    fork
      automatic int k = i;
      begin
        wait (done[k].triggered);
        done_count++;
      end
    join_none
  wait (done_count==N_GENERATORS); // Wait for triggers

end
```

That was slightly less complicated. Why not get rid of all the events and just wait on a count of the number of running generators? This count can be a static variable in the `Generator` class. Note that most of the thread manipulation code has been replaced with a single `wait` construct. The last block in Sample 7.29 waits for the count using the class scope resolution operator, `::`. You could have used any handle, such as `gen[0]`, but that would be less direct.

Sample 7.29 Waiting for multiple threads using a thread count

```
class Generator;
  static int thread_count = 0;

  task run();
    thread_count++;       // Start another thread
    fork
      begin
      // Do the real work in here
      // And when done, decrement the thread count
      thread_count--;
      end
    join_none
  endtask
endclass

Generator gen[N_GENERATORS];

initial begin
  // Create N generators
  foreach (gen[i])
    gen[i] = new();

  // Start them running
  foreach (gen[i])
    gen[i].run();

  // Wait for all the generators to complete
  wait (Generator::thread_count == 0);
end
```

7.5 Semaphores

A semaphore allows you to control access to a resource. Imagine that you and your spouse share a car. Obviously, only one person can drive it at a time. You can manage this situation by agreeing that whoever has the key can drive it. When you are done with the car, you give up the car so that the other person can use it. The key is the semaphore that makes sure only one person has access to the car. In operating system terminology, this is known as “mutually exclusive access,” so a semaphore is known as a “mutex” and is used to control access to a resource.

Semaphores can be used in a testbench when you have a resource, such as a bus, that may have multiple requestors from inside the testbench but, as part of the physical design, can only have one driver. In SystemVerilog, a thread that requests a key when one is not available always blocks. Multiple blocking threads are queued in FIFO order.

7.5.1 Semaphore Operations

There are three basic operations for a semaphore. You create a semaphore with one or more keys using the new method, get one or more keys with the blocking task `get()`, and return one or more keys with `put()`. If you want to try to get a semaphore, but not block, use the `try_get()` function. If keys are available, `try_get()` obtains them and returns 1. If there are not sufficient keys, it just returns a 0. Sample 7.30 shows how to control access to a resource with a semaphore.

Sample 7.30 Semaphores controlling access to hardware resource

```
program automatic test(bus_ifc.TB bus);
  semaphore sem;            // Create a semaphore
  initial begin
    sem = new(1);           // Allocate with 1 key
    fork
      sequencer();          // Spawn two threads that both
      sequencer();          //   do bus transactions
    join
  end

  task sequencer();
    repeat($urandom()%10)  // Random wait, 0-9 cycles
      @bus.cb;
    sendTrans();            // Execute the transaction
  endtask

  task sendTrans();
    sem.get(1);             // Get the key to the bus
    @bus.cb;                // Drive signals onto bus
    bus.cb.addr <= tr.addr;
    ...
    sem.put(1);             // Put it back when done
  endtask
endprogram
```

7.5.2 Semaphores with Multiple Keys

There are two things you should watch out for with semaphores. First, you can `put` more keys back than you took out. Suddenly you may have two keys but only one car! Secondly, be careful if your testbench needs to get and put multiple keys. Perhaps you have one key left, and a thread requests two, causing it to block. Now a second thread requests a single semaphore – what should happen? In SystemVerilog the second request, `get(1)`, sneaks ahead of the earlier `get(2)`, bypassing the FIFO ordering.

If you are mixing different sized requests, you can always write your own class. That way you can be very clear on who gets priority.

7.6 Mailboxes

How do you pass information between two threads? Perhaps your generator needs to create many transactions and pass them to a driver. You might be tempted to just have the generator thread call a task in the driver. If you do that, the generator needs to know the hierarchical path to the driver task, making your code less reusable. Additionally, this style forces the generator to run at the same speed as the driver, that can cause synchronization problems if one generator needs to control multiple drivers.

Think of your generator and driver as transactors that are autonomous objects that communicate through a channel. Each object gets a transaction from an upstream object (or creates it, as in the case of a generator), does some processing, and then passes it to a downstream object. The channel must allow its driver and receiver to operate asynchronously. You may be tempted to just use a shared array or queue, but it can be difficult to create threads that read, write, and blocks safely.

The solution is a SystemVerilog mailbox. From a hardware point of view, the easiest way to think about a mailbox is that it is just a FIFO, with a source and sink. The source puts data into the mailbox, and the sink gets values from the mailbox. Mailboxes can have a maximum size or can be unlimited. When the source thread tries to put a value into a sized mailbox that is full, that thread blocks until the value is removed. Likewise, if a sink threads tries to remove a value from a mailbox that is empty, that thread blocks until a value is put into the mailbox.

Figure 7.5 shows a mailbox connecting a generator and driver.

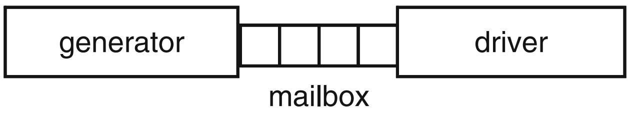

Fig. 7.5 A mailbox connecting two transactors

A mailbox is an object and thus has to be instantiated by calling the `new` function. This takes an optional `size` argument to limit the number of entries in the mailbox. If the size is 0 or not specified, the mailbox is unbounded and can hold an unlimited number of entries.

You put data into a mailbox with the `put()` task, and remove it with the blocking `get()` task. A `put()` blocks if the mailbox is full, and `get()` blocks if the mailbox is empty. Use `try_put()` if you want to see if the mailbox is full. and `try_get()` to see if it is empty. The `peek()` task gets a copy of the data in the mailbox but does not remove it.

The data is a single value, such as an integer, or logic of any size or a handle. A mailbox never contains objects, only references to them. By default, a mailbox does not have a type, so you could put any mix of data into it. Don't do it! Enforce one data type per mailbox by sticking with parameterized mailboxes as shown in Sample 7.31 to catch type mismatches at compile time.

Sample 7.31 Mailbox declarations

```
mailbox #(Transaction) mbx_tr;  // Parameterized: recommended
mailbox mbx_untyped;            // Unspecialized: avoid
```

A classic mailbox bug, shown in Sample 7.32, is a loop that randomizes objects and puts them in a mailbox, but the object is only constructed once, outside the loop. Since there is only one object, it is randomized over and over.

Sample 7.32 Bad generator creates only one object

```
task generator_bad(input int n,
                   input mailbox #(Transaction) mbx);
  Transaction tr;
  tr = new();                 // Create just one transaction
  repeat (n) begin
    `SV_RAND_CHECK(tr.randomize());
    $display("GEN: Sending addr=%h", tr.addr);
    mbx.put(tr);              // Send transaction to driver
  end
endtask
```

Figure 7.6 shows all the handles pointing to a single object. A mailbox only holds handles, not objects, so you end up with a mailbox containing multiple handles that all point to the single object. The code that gets the handles from the mailbox just sees the last set of random values.

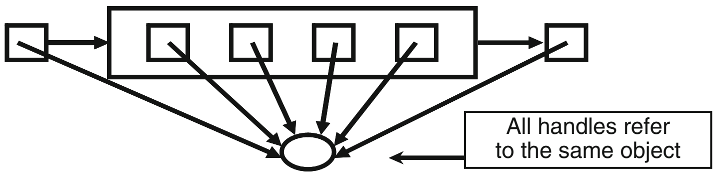

Fig. 7.6 A mailbox with multiple handles to one object

The solution, shown in Sample 7.33, is to make sure your loop has all three steps of constructing the object, randomizing it, and putting it in the mailbox. This bug is so common that it is also mentioned in Section 5.14.3.

Sample 7.33 Good generator creates many objects

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

The result, shown in Fig. 7.7, is that every handle points to a unique object. This type of generator is known as the Blueprint Pattern and described in Section 8.2.

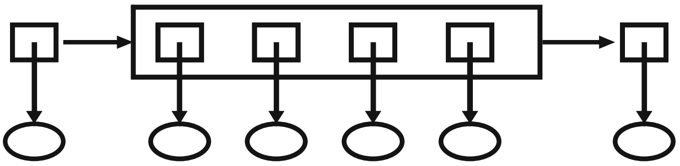

Fig. 7.7 A mailbox with multiple handles to multiple objects

Sample 7.34 shows the driver that waits for transactions from the generator.

Sample 7.34 Good driver receives transactions from mailbox

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

If you don't want your code to block when accessing the mailbox, use the `try_get()` and `try_peek()` functions. If they are successful, they return a nonzero value; otherwise, they return 0. These are more reliable than the `num()` function, as the number of entries can change between when you measure it and when you next access the mailbox.

7.6.1 Mailbox in a Testbench

Sample 7.35 shows a program with a Generator and Driver exchanging transactions using a mailbox.

Sample 7.35 Exchanging objects using a mailbox: the Generator class

```
program automatic mailbox_example(bus_ifc.TB bus);

class Generator;
  Transaction tr;
  mailbox #(Transaction) mbx;

  function new(input mailbox #(Transaction) mbx);
    this.mbx = mbx;
  endfunction

  task run(input int count);
    repeat (count) begin
      tr = new();
      `SV_RAND_CHECK(tr.randomize);
      mbx.put(tr);    // Send out transaction
    end
  endtask
endclass


class Driver;
  Transaction tr;
  mailbox #(Transaction) mbx;

  function new(input mailbox #(Transaction) mbx);
    this.mbx = mbx;
  endfunction

  task run(input int count);
    repeat (count) begin
      mbx.get(tr);      // Fetch next transaction
      // Drive transaction here
    end
  endtask
endclass
```

```
  mailbox #(Transaction) mbx;   // Mailbox connecting gen & drv
  Generator gen;
  Driver drv;
  int count;

  initial begin
    mbx = new();                // Construct the mailbox
    gen = new(mbx);             // Construct the generator
    drv = new(mbx);             // Construct the driver
    count = $urandom_range(50); // Run up to 50 transactions

    fork
      gen.run(count);           // Spawn the generator
      drv.run(count);           // Spawn the driver
    join                        // Wait for both to finish

  end
endprogram
```

7.6.2 Bounded Mailboxes

By default, mailboxes are similar to an unlimited FIFO — a producer can put any number of objects into a mailbox before the consumer gets the objects out. However, you may want the two threads to operate in lockstep so that the producer blocks until the consumer is done with the object.

You can specify a maximum size for the mailbox when you construct it. The default mailbox size is 0 which creates an unbounded mailbox. Any size greater than 0 creates a bounded mailbox. If you attempt to put more objects than this limit, `put()` blocks until you `get` an object from the mailbox, creating a vacancy.

Sample 7.36 Bounded mailbox

```
program automatic bounded;
  mailbox #(int) mbx;

  initial begin
    mbx = new(1);  // Mailbox size = 1
    fork

      // Producer thread
      for (int i=1; i<4; i++) begin
        $display("Producer: before put(%0d)", i);
        mbx.put(i);
        $display("Producer: after  put(%0d)", i);
      end
      // Consumer thread
      repeat(4) begin
        int j;
        #1ns mbx.get(j);
        $display("Consumer: after  get(%0d)", j);
      end
    join
  end
endprogram
```

Sample 7.36 creates the smallest possible mailbox, which can hold a single message. The Producer thread tries to put three messages (integers) in the mailbox, and the Consumer thread slowly gets messages every 1ns. As Sample 7.37 shows, the first `put()` succeeds, then the Producer tries `put(2)` which blocks. The Consumer wakes up, gets a message 1 from the mailbox, so now the Producer can finish putting the message 2.

Sample 7.37 Output from bounded mailbox

```
Producer: before put(1)
Producer: after  put(1)
Producer: before put(2)
Consumer: after  get(1)
Producer: after  put(2)
Producer: before put(3)
Consumer: after  get(2)
Producer: after  put(3)
Consumer: after  get(3)
```

The bounded mailbox acts as a buffer between the two processes. You can see how the Producer generates the next value before the Consumer reads the current value.

7.6.3 Unsynchronized Threads Communicating with a Mailbox

In many cases, two threads that are connected by a mailbox should run in lockstep, so that the producer does not get ahead of the consumer. The benefit of this approach is that your entire chain of stimulus generation now runs in lock step. The highest level generator only completes when the last low level transaction completes transmission. Now your testbench can tell precisely when all stimulus has been sent. In another example, if your generator gets ahead of the driver, and you are gathering functional coverage on the generator, you might record that some transactions were tested, even if the test stopped prematurely. So even though a mailbox allows you to decouple the two sides, you may still want to keep them synchronized.

If you want two threads to run in lockstep, you need a handshake in addition to the mailbox. In Sample 7.38 the Producer and Consumer are now classes that exchange integers using a mailbox, with no explicit synchronization between the two objects. As a result, as shown in Sample 7.39, the producer runs to completion before the consumer even starts.

Sample 7.38 Producer–consumer without synchronization

```
program automatic unsynchronized;

  mailbox #(int) mbx;

  class Producer;
    task run();
      for (int i=1; i<4; i++) begin
        $display("Producer: before put(%0d)", i);
        mbx.put(i);
      end
    endtask
  endclass

  class Consumer;
    task run();
      int i;
      repeat (3) begin
        mbx.get(i);      // Get integer from mbx
        $display("Consumer: after  get(%0d)", i);
      end
    endtask
  endclass

  Producer p;
  Consumer c;

  initial begin
    // Construct mailbox, producer, consumer
    mbx = new();     // Unbounded
    p = new();
    c = new();

    // Run the producer and consumer in parallel
    fork
      p.run();
      c.run();
    join
 end
endprogram
```

The above sample holds the mailbox in a global variable to make the code more compact. In real code, you should pass the mailbox into the class through the constructor and save a reference to it in a class-level variable.

Sample 7.38 has no synchronization so the Producer puts all three integers into the mailbox before the Consumer can get the first one. This is because a thread continues running until there is a blocking statement, and the Producer has none. The Consumer thread blocks on the first call to `mbx.get`.

Sample 7.39 Producer–consumer without synchronization output

```
Producer: before put(1)
Producer: before put(2)
Producer: before put(3)
Consumer: after  get(1)
Consumer: after  get(2)
Consumer: after  get(3)
```

This example has a race condition, so on some simulators the consumer could activate earlier. The result is still the same as the values are determined by the producer, not by how quickly the consumer sees them.

7.6.4 Synchronized Threads Using a Bounded Mailbox and a Peek

In a synchronized testbench, the Producer and Consumer operate in lock step. This way, you can tell when the input stimuli is complete by waiting for any of the threads. If the threads operate unsynchronized, you need to add extra code to detect when the last transaction is applied to the DUT.

To synchronize two threads, the Producer creates and puts a transaction into a mailbox, then blocks until the Consumer finishes with it. This is done by having the Consumer remove the transaction from the mailbox only when it is finally done with it, not when the transaction is first detected.

Sample 7.40 show the first attempt to synchronize two threads, this time with a bounded mailbox. The Consumer uses the built-in mailbox method `peek()` to look at the data in the mailbox without removing. When the Consumer is done processing the data, it removes the data with `get()`. This frees up the Producer to generate a new value. If the Consumer loop started with a `get()` instead of the `peek()`, the transaction would be immediately removed from the mailbox, so the Producer could wake up before the Consumer finished with the transaction. Sample 7.41 has the output from this code.

Sample 7.40 Producer–consumer synchronized with bounded mailbox

```
program automatic synch_peek;
// Uses Producer from Sample 7-38

  mailbox #(int) mbx;

  class Consumer;
    task run();
      int i;
      repeat (3) begin
        mbx.peek(i);      // Peek integer from mbx
        $display("Consumer: after peek(%0d)", i);
        mbx.get(i);       // Remove from mbx
      end
    endtask
  endclass : Consumer

  Producer p;
  Consumer c;

  initial begin
    // Construct mailbox, producer, consumer
    mbx = new(1);   // Bounded mailbox - limit 1!
    p = new();
    c = new();

    // Run the producer and consumer in parallel
    fork
      p.run();
      c.run();
    join
 end
endprogram
```

Sample 7.41 Output from producer–consumer with bounded mailbox

```
Producer: before put(1)
Producer: before put(2)
Consumer: after  peek(1)
Consumer: after  peek(2)
Producer: before put(3)
Consumer: after  peek(3)
```

You can see that the Producer and Consumer are in lockstep, but the Producer is still one transaction ahead of the Consumer. This is because a bounded mailbox with size=1 only blocks when you try to do a put of the second transaction.[2]

[2]This behavior is different from the VMM channel. If you set a channel's full level to 1, the very first call to put() places the transaction in the channel, but does not return until the transaction is removed.

7.6.5 *Synchronized Threads Using a Mailbox and Event*

You may want the two threads to use a handshake so that the Producer never gets ahead of the Consumer. The Consumer already blocks, waiting for the Producer using a mailbox. The Producer needs to block, waiting for the Consumer to finish the transaction. Do this by adding a blocking statement to the Producer such as an event, a semaphore, or a second mailbox. Sample 7.42 uses an event to block the Producer after it puts data in the mailbox. The Consumer triggers the event after it consumes the data.

If you use `wait (handshake.triggered)` in a loop, be sure to advance the time before waiting again, as previously shown in Section 7.4.3. This `wait` blocks only once in a given time slot, so you need move into another. Sample 7.42 uses the edge-sensitive blocking statement `@handshake` instead to ensure that the Producer stops after sending the transaction. The edge-sensitive statement works multiple times in a time slot but may have ordering problems if the trigger and block happen in the same time slot.

Sample 7.42 Producer–consumer synchronized with an event

```
program automatic mbx_evt;
  mailbox #(int) mbx;
  event handshake;

  class Producer;
    task run();
      for (int i=1; i<4; i++) begin
        $display("Producer: before put(%0d)", i);
        mbx.put(i);
        @handshake;
        $display("Producer: after  put(%0d)", i);
      end
    endtask
  endclass : Producer

  class Consumer;
    task run();
      int i;
      repeat (3) begin
        mbx.get(i);
        $display("Consumer: after  get(%0d)", i);
        ->handshake;
      end
    endtask
  endclass : Consumer
  Producer p;
  Consumer c;

  initial begin
    p = new();
    c = new();
    mbx = new();

    // Run the producer and consumer in parallel
    fork
      p.run();
      c.run();
    join
 end
endprogram
```

Now the Producer does not advance until the Consumer triggers the event, as shown in Sample 7.43.

Sample 7.43 Output from producer–consumer with event

```
Producer: before put(1)
Consumer: after  get(1)
Producer: after  put(1)
Producer: before put(2)
Consumer: after  get(2)
Producer: after  put(2)
Producer: before put(3)
Consumer: after  get(3)
Producer: after  put(3)
```

You can see that the Producer and Consumer are successfully running in lockstep by the fact that the Producer never produces a new value until after the old one is read.

7.6.6 Synchronized Threads Using Two Mailboxes

Another way to synchronize the two threads is to use a second mailbox that sends a completion message back to the Producer, as shown in Sample 7.44.

Sample 7.44 Producer–consumer synchronized with a mailbox

```
program automatic mbx_mbx2;
  mailbox #(int) mbx, rtn;
  class Producer;
    task run();
      int k;
      for (int i=1; i<4; i++) begin
        $display("Producer: before put(%0d)", i);
        mbx.put(i);
        rtn.get(k);
        $display("Producer: after  get(%0d)", k);
      end
    endtask
  endclass : Producer

  class Consumer;
    task run();
      int i;
      repeat (3) begin
        $display("Consumer: before get");
        mbx.get(i);
        $display("Consumer: after  get(%0d)", i);
        rtn.put(-i);
      end
    endtask
  endclass : Consumer

  Producer p;
  Consumer c;
  initial begin
    p = new();
    c = new();
    mbx = new();
    rtn = new();

    // Run the producer and consumer in parallel
    fork
      p.run();
      c.run();
    join
 end
endprogram
```

The return message in the `rtn` mailbox is just a negative version of the original integer. You could use any value, but this one can be checked against the original for debugging purposes.

Sample 7.45 Output from producer–consumer with mailbox

```
Producer: before put(1)
Consumer: before get
Consumer: after  get(1)
Consumer: before get
Producer: after  get(-1)
Producer: before put(2)
Consumer: after  get(2)
Consumer: before get
Producer: after  get(-2)
Producer: before put(3)
Consumer: after  get(3)
Producer: after  get(-3)
```

You can see from Sample 7.45 that the Producer and Consumer are successfully running in lockstep.

7.6.7 Other Synchronization Techniques

You can also complete the handshake by blocking on a variable or a semaphore. An event is the simplest construct, followed by blocking on a variable. A semaphore is comparable to using a second mailbox, but no information is exchanged. SystemVerilog's bounded mailbox just does not work as well as these other techniques as there is no way to block the producer when it puts the first transaction in. Sample 7.41 shows that the Producer is always one transaction ahead of the Consumer.

7.7 Building a Testbench with Threads and IPC

Way back in Section 1.10 you learned about layered testbenches. Figure 7.8 shows the relationship between the different parts. Now that you know how to use threads and IPC, you can construct a basic testbench with transactors.

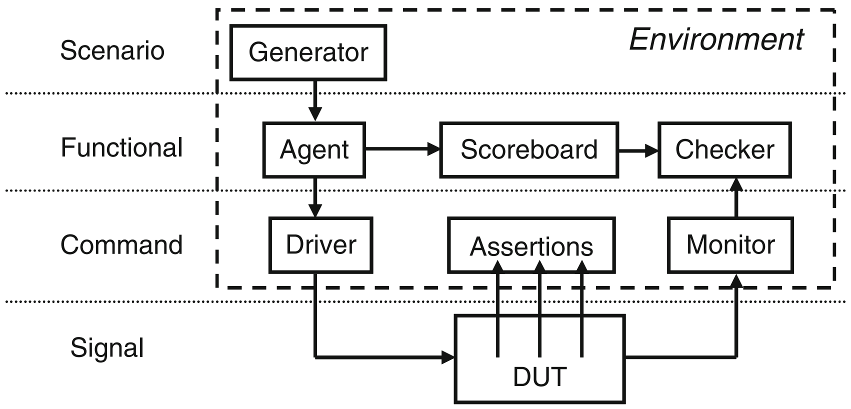

Fig. 7.8 Layered testbench with environment

7.7.1 *Basic Transactor*

Sample 7.46 is the Agent class that sits between the Generator and the Driver.

Sample 7.46 Basic Transactor

```
class Agent;

  mailbox #(Transaction) gen2agt, agt2drv;
  Transaction tr;

  function new(input mailbox #(Transaction) gen2agt, agt2drv);
    this.gen2agt = gen2agt;
    this.agt2drv = agt2drv;
  endfunction

  task run();
    forever begin
      gen2agt.get(tr);    // Get transaction from upstream block
      ...                 // Do some processing
      agt2drv.put(tr);    // Send it to downstream block
    end
  endtask

  task wrap_up();   // Empty for now
  endtask

endclass
```

7.7.2 Configuration Class

The configuration class allows you to randomize the configuration of your system for every simulation. Sample 7.47 has just one variable and a basic constraint.

Sample 7.47 Configuration class

```
class Config;
  rand bit [31:0] run_for_n_trans;
  constraint reasonable
    {
     run_for_n_trans inside {[1:1000]};
    }
endclass
```

7.7.3 Environment Class

The Environment class, shown as a dashed line in Fig. 7.8, holds the Generator, Agent, Driver, Monitor, Checker, Scoreboard, and Config objects, and the mailboxes between them. Sample 7.48 shows a basic Environment class.

Sample 7.48 Environment class

```
class Environment;

  Generator  gen;
  Agent      agt;
  Driver     drv;
  Monitor    mon;
  Checker    chk;
  Scoreboard scb;
  Config     cfg;
  mailbox #(Transaction) gen2agt, agt2drv, mon2chk;

  extern function new();
  extern function void gen_cfg();
  extern function void build();
  extern task run();
  extern task wrap_up();
endclass


function Environment::new();
  cfg = new();
endfunction

function void Environment::gen_cfg();
  `SV_RAND_CHECK(cfg.randomize);
endfunction
```

```
function void Environment::build();
  // Initialize mailboxes
  gen2agt = new();
  agt2drv = new();
  mon2chk = new();

  // Initialize transactors
  gen = new(gen2agt);
  agt = new(gen2agt, agt2drv);
  drv = new(agt2drv);
  mon = new(mon2chk);
  chk = new(mon2chk);
  scb = new(cfg);
endfunction

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

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
```

Chapter 8 shows more details on how to build these classes.

7.7.4 Test Program

Sample 7.49 shows the main test, which is in a program block. As discussed in Section 4.3.4, you can also put a test in a module, but at a slight increase in the chances of race conditions.

Sample 7.49 Basic test program

```
program automatic test;

  Environment env;

  initial begin
    env = new();
    env.gen_cfg();
    env.build();
    env.run();
    env.wrap_up();
  end

endprogram
```

7.8 Conclusion

Your design is modeled as many independent blocks running in parallel, so your testbench must also generate multiple stimulus streams and check the responses using parallel threads. These are organized into a layered testbench, orchestrated by the toplevel environment. SystemVerilog introduces powerful constructs such as `fork...join_none` and `fork...join_any` for dynamically creating new threads, in addition to the standard `fork...join`. These threads communicate and synchronize using events, semaphores, mailboxes, and the classic `@` event control and `wait` statements. Lastly, the `disable` command is used to terminate threads.

These threads and the related control constructs complement the dynamic nature of OOP. As objects are created and destroyed, they can run in independent threads, allowing you to build a powerful and flexible testbench environment.

7.9 Exercises

1. For the following code determine the order and time of execution for each statement if a `join` or `join_none` or `join_any` is used. Hint: the order and time of execution between the `fork` and `join/join_none/join_any` is the same, only the order and execution time of the statements after the `join` are different.

```
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);
    #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
```

2. For the following code what would the output be with and without a `wait fork` inserted in the indicated location?

```
initial begin

  fork
    transmit(1);
    transmit(2);
  join_none

  fork: receive_fork
    receive(1);
    receive(2);
  join_none

  // What is the output with/without a wait fork here?

  #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
```

3. What would be displayed with the following code? Assume that the events and task `trigger` is declared inside a program declared as automatic.

```
event e1, e2;
task trigger(event local_event, input time wait_time);
  #wait_time;
  ->local_event;
endtask

initial begin
  fork
    trigger(e1, 10ns);
    begin
      wait(e1.triggered());
      $display("%0t: e1 triggered", $time);
    end
  join
end

initial begin
  fork
    trigger(e2, 20ns);
    begin
      wait(e2.triggered());
      $display("%0t: e2 triggered", $time);
    end
  join
end
```

4. 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.

5. What would be displayed with the following code that calls the task from Exercise 4?

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

6. What would be displayed with the following code?

```
program automatic test;
  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
endprogram
```

7. Look at Fig. 7.8 "Layered testbench with environment" on page 265 and create the `Monitor` class. You can make the following assumptions.

   a. The `Monitor` class has knowledge of class `OutputTrans` with member variables `out1` and `out2`.
   b. The DUT and `Monitor` are connected with an interface called `my_bus`, with signals `out1` and `out2`.
   c. The interface `my_bus` has a clocking block, `cb`.
   d. On every active clock edge, the Monitor class will sample the DUT outputs, `out1` and `out2`, assign them to an object of type `OutputTrans`, and place the object in a mailbox.