INF5430

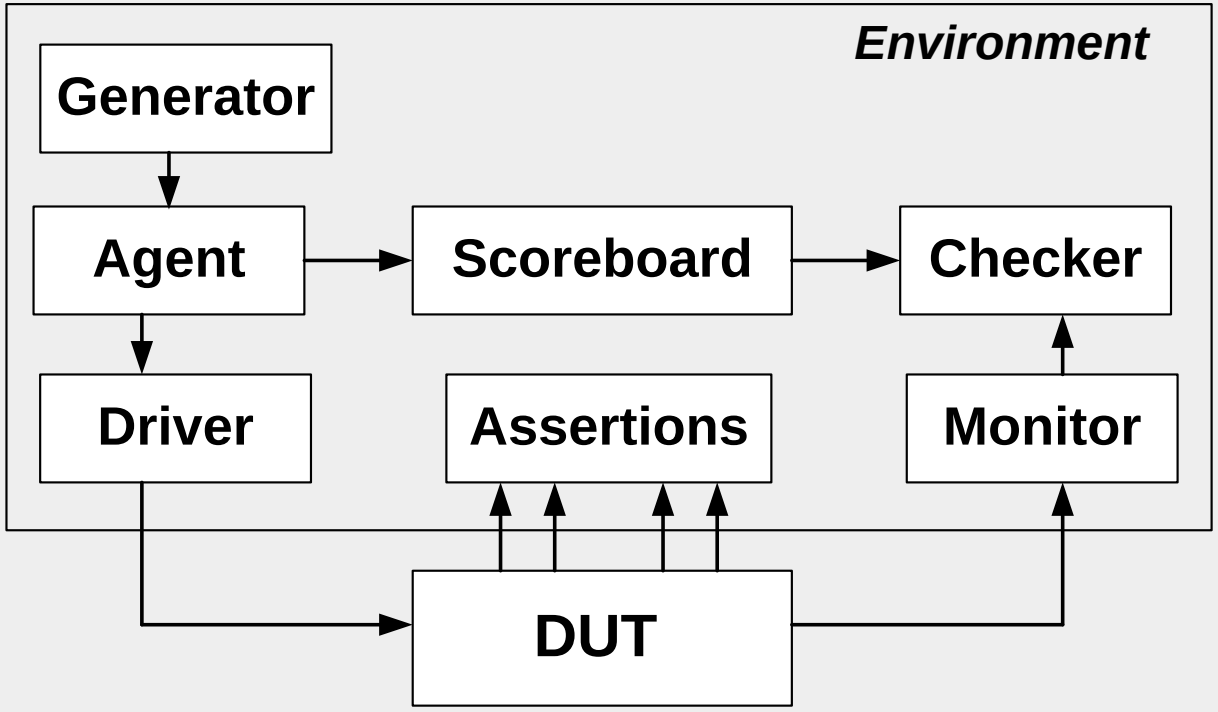
Spear & Tumbush

SystemVerilog for Verification

Chapter 7.1-4

Chapter 7 Threads and Interprocess Comm

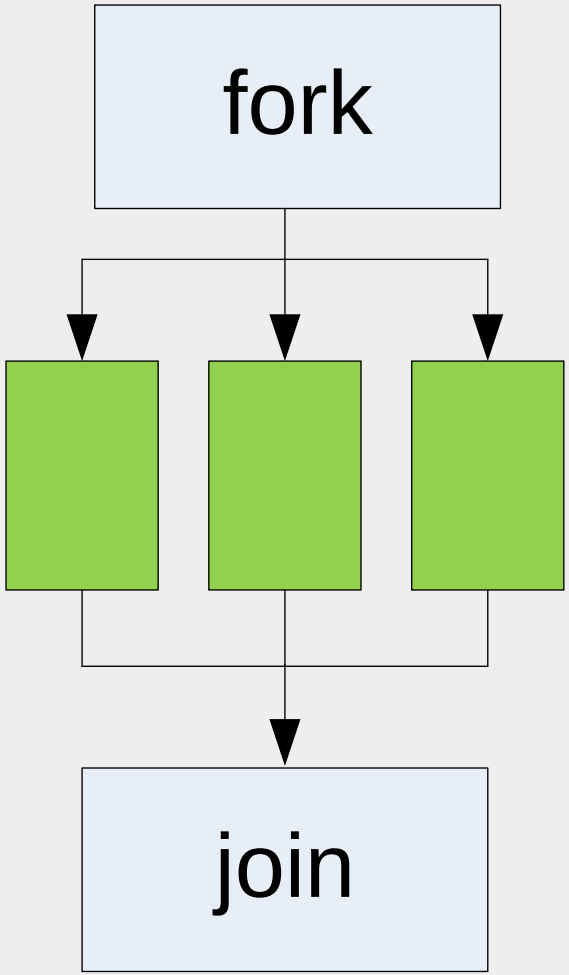
- The testbench has many threads running in parallel



- 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
```

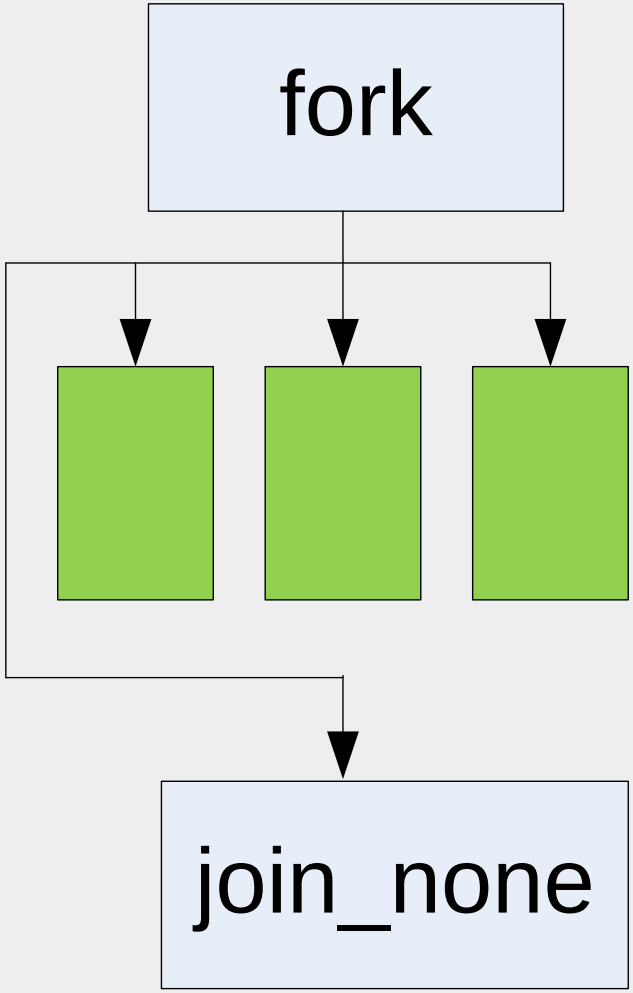
7.1.1 Using fork..join and begin..end

Order	Time
1	0
2	10
3	10
7	60
4	20
5	40
6	50
8	60
9	140

```
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
```

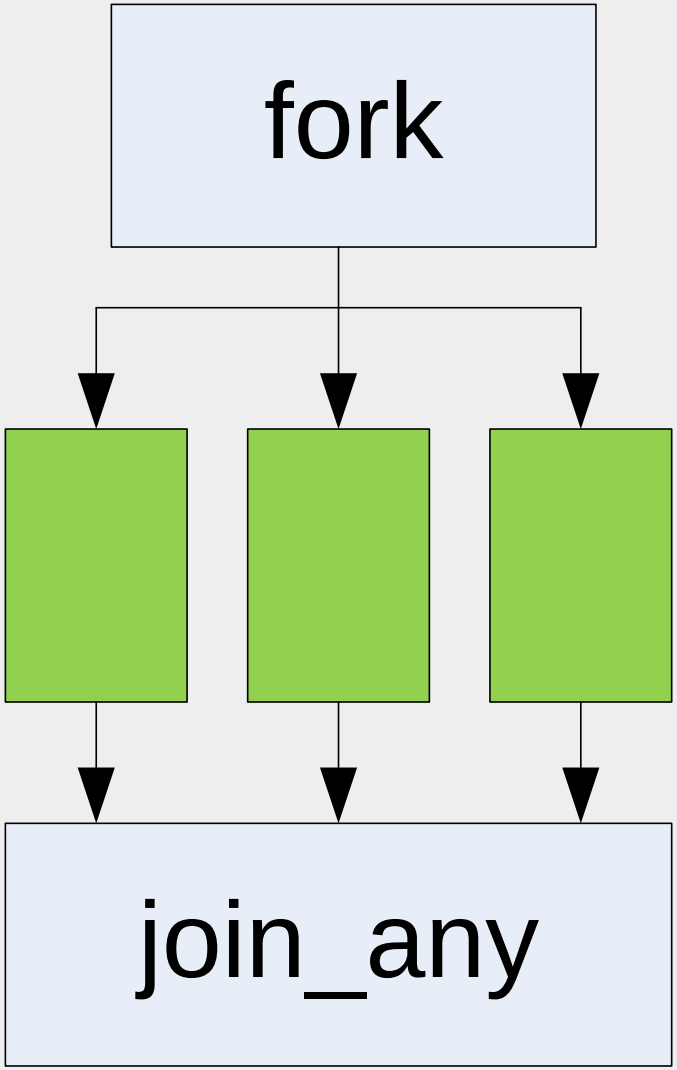
7.1.2 Spawning Threads w/ fork..join_none

Order	Time
1	0
2	10
3 or 4	10
8	60
5	20
6	40
7	50
3 or 4	10
9	90

```
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

Order	Time
1	0
2	10
3	10
8	60
5	20
6	40
7	50
4	10
9	90

```
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
```

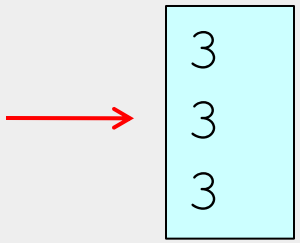
7.1.4 Creating Threads in a Class

```
class Gen_drive;
   task run(input int n);
      Packet p;
      fork
         repeat (n) begin
            p = new();
            `SV_RAND_CHECK(p.randomize());
            transmit(p); //Transmit the object. Defined elsewhere.
         end
      join_none  //Use fork … join_none so run() does not block.
   endtask
endclass
Gen_drive gen;
initial begin
   gen = new();
   gen.run(10); // Tell run to create 10 new transactions.
   // Start the checker, monitor, and other threads
end
```

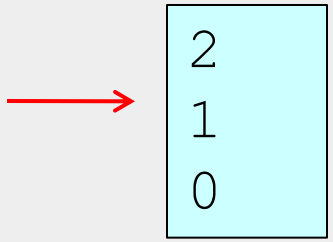
7.1.6 Automatic Variables in Threads

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

```
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
```

~~Done at time 0~~

index = 1
index = 2
index = 3
Done at time 10

- 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

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

To disable unneeded threads

```
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

```
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
  join_none
endtask
```

Thread 1

Thread 2

Thread 3

Thread 4

7.2.2 Disabling Multiple Threads

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

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

Thread 0

Thread 1

Thread 2

Thread 3

Thread 4

7.2.2 Disabling Multiple Threads (cont)

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

Thread 0

Thread 1

Thread 2

Thread 3

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
- 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

```
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
```

```
@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

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

initial begin
4     $display("@%0t: 2: before trigger", $time);
5     -> e2;
6     wait (e1.triggered());
7     $display("@%0t: 2: after trigger", $time);
end
```

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

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.

```
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;
end

task process_in_zero_time();
endtask
```

@handshake;

```
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
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.

```
package my_package;
   class Generator;
     event done;
     function new (event done);
        this.done = done;
     endfunction
     task run();
        fork
           begin
              #10ns;
              -> done;
            end
        join_none
     endtask
   endclass
endpackage
```

7.4.4 Passing Events (cont)

Usage of class Generator

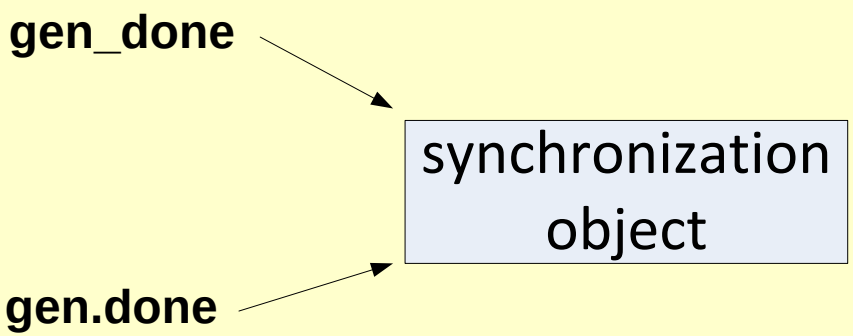
```
program automatic test;

    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()); OR: wait(gen.done.triggered());
        $display("%0t: Done waiting on gen_done", $time);
    end

endprogram
```

```
0: Waiting on gen_done
10: Done waiting on gen_done
```



7.4.5 Waiting for multiple Events

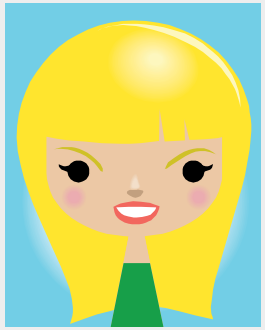
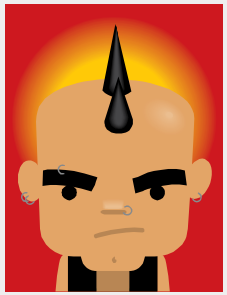
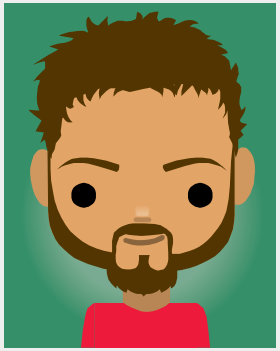
How to wait for multiple threads of class Generator to finish?

```
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
```

7.5 Semaphores

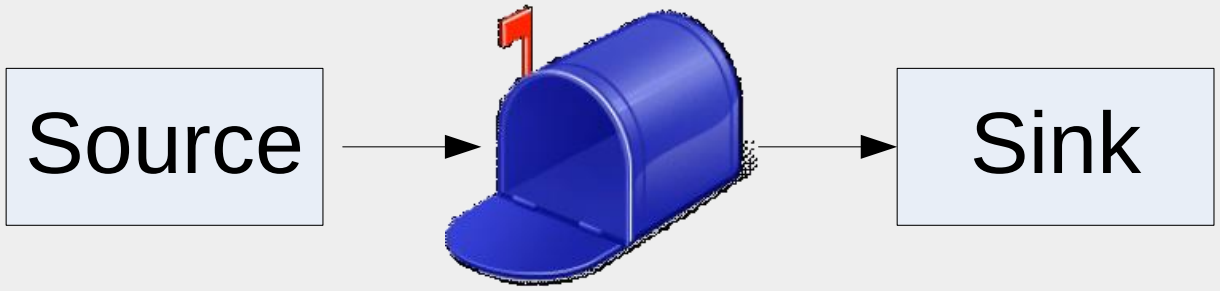
- Semaphores allow the testbench to control access to a resource
- Think of a semaphore as a bucket containing 1 or more keys
- Access to a resource requires a key

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

7.6 Mailboxes

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



- 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**