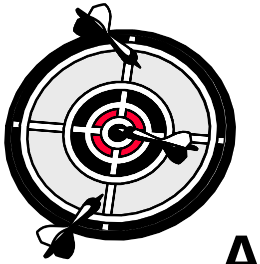
Agenda

DAY 2

5	Concurrency
6	Object Oriented Programming (OOP) – Encapsulation
7	Object Oriented Programming (OOP) – Randomization

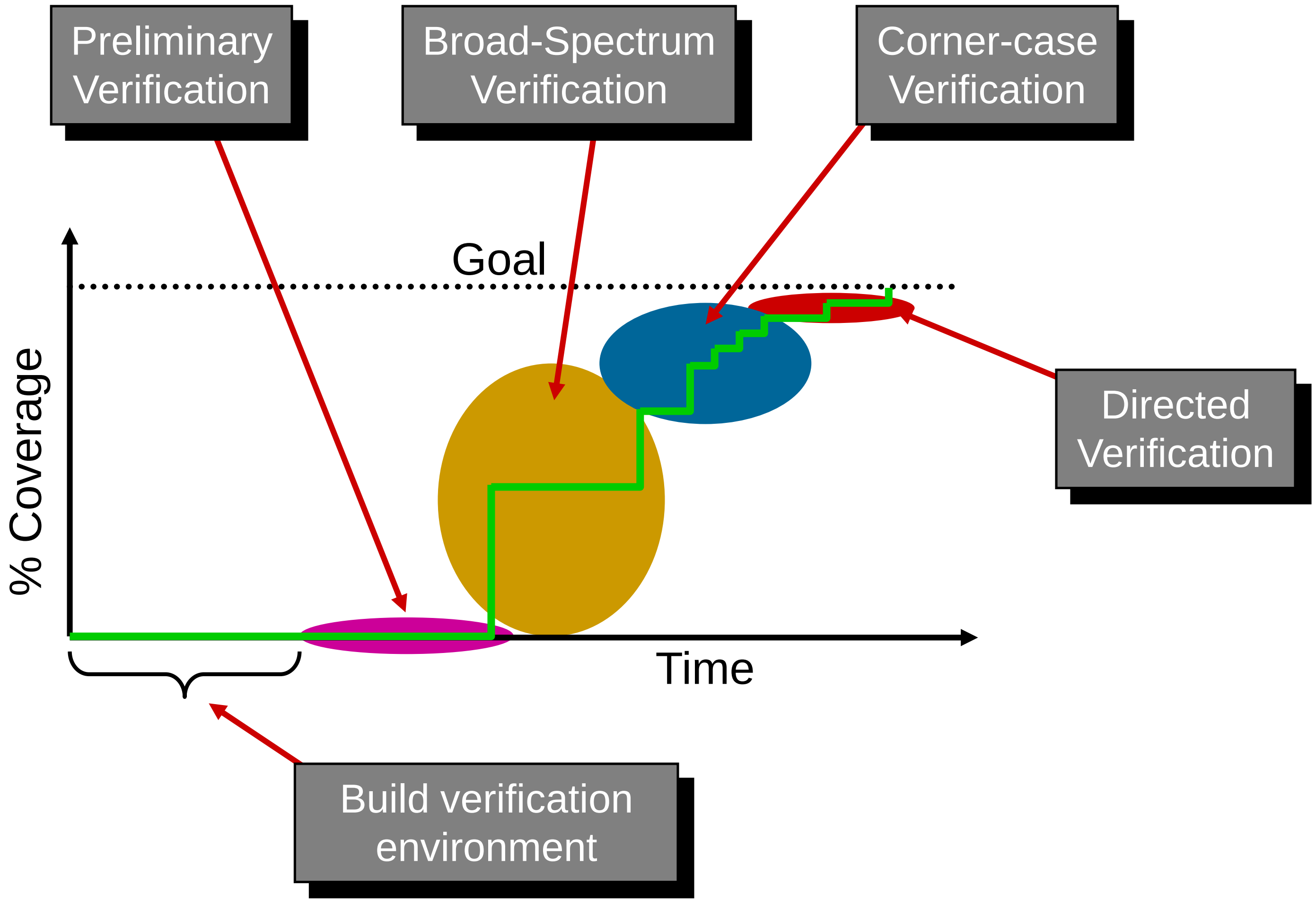
Unit Objectives



After completing this unit, you should be able to:

- **Divide a testbench into multiple current threads to execute parallel tasks**

Phases of verification



Day 1 Review (Building Testbench)

```
program automatic test(simple_bus.tb sb);
  initial
    run_test();
endprogram: test
```

Develop test program

```
module cpu(simple_bus sb);
  …
endmodule: cpu
```

```
interface simple_bus(input bit clk);
  logic req, gnt;
  logic [7:0] addr;
  wire  [7:0] data;
  clocking cb @(posedge clk)
    output req;
    input gnt;
    …
  endclocking: cb
  modport tb(clocking cb);
endinterface: simple_bus
```

Define interface

```
module top;
  logic clk = 0;
  always #10ns clk = !clk;
  simple_bus sb(clk);
  test t1(sb);
  cpu c1(sb);
endmodule: top
```

Encapsulate DUT, test and interface in harness module

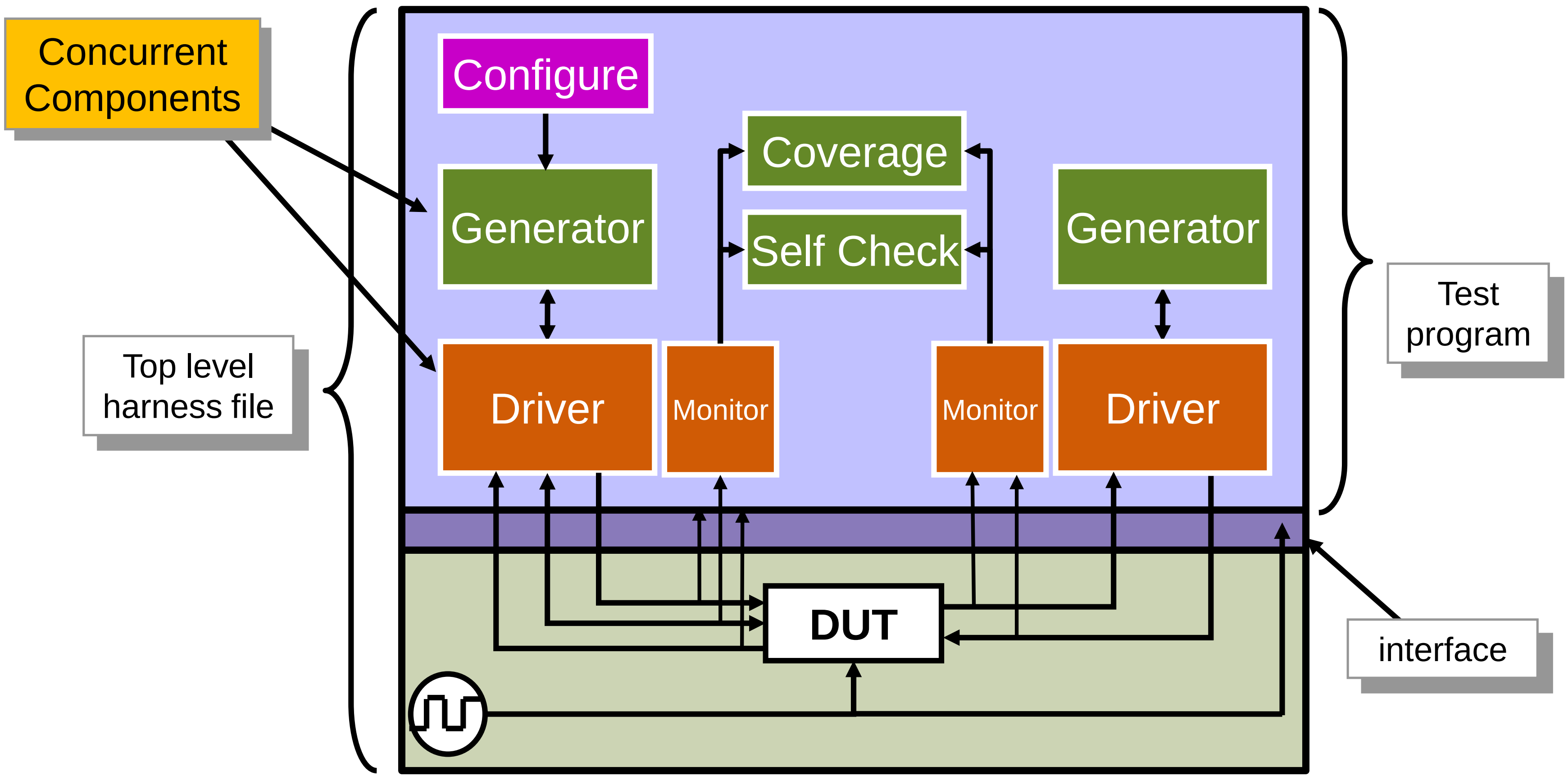
Connect DUT and program using interface

Compile and run with VCS

```
% vcs –sverilog cpu.v test.v interface.v top.v
% simv
```

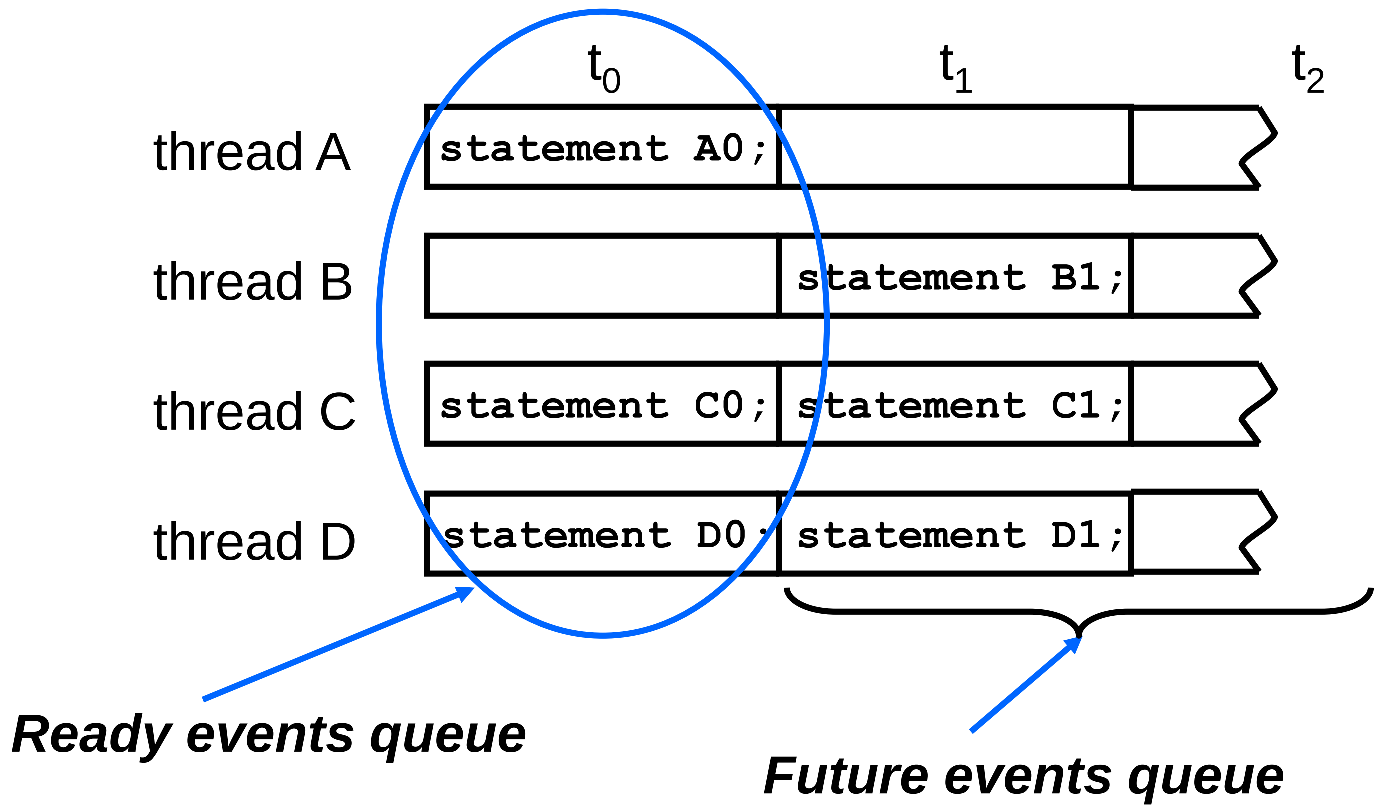
Testbench Requires Concurrency

- **Components of the testbench run concurrently**
  - Concurrent components run as separate threads



Concurrency in Simulators

- **A simulator can only execute one thread at a time in a single-core CPU.**
  - Multiple threads waiting to execute at one simulation time point have to be scheduled in queues to run one-at-a-time.



Creating Concurrent Threads

- **Concurrent threads are created in a `fork-join` block:**

```
int a, b, c;  //parent variables
fork
   [fork local declarations] // visible to all threads
   statement0;  //child thread 1
   begin        //child thread 2
     statement1;
     statement2;
   end
join | join_any | join_none
statement3;
```

  - Statements enclosed in **`begin-end`** in a **`fork-join`** block are executed sequentially as a single concurrent child thread
  - No predetermined execution order for concurrent threads
  - parent variables cannot be referred to in **`join_any`** or **`join_none`** except to initialize variables in fork local declarations

How Many Child Threads?

A:

```
fork
  begin
   recv();
  end
  begin
   send();
  end
join
```

B:

```
fork
  recv();
  send();
join
```

C:

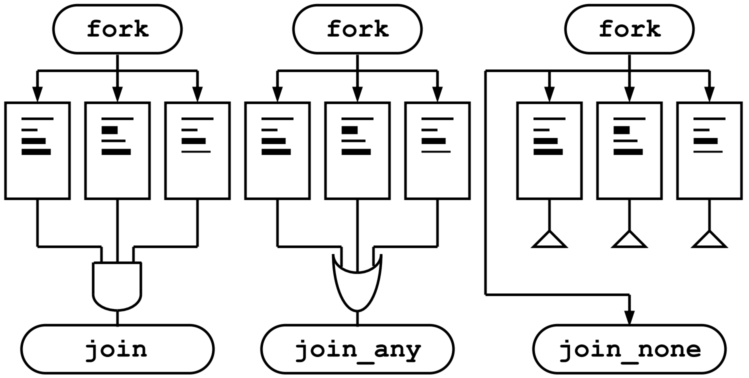
```
fork
  begin
    recv();
    send();
  end
join
```

D:

```
fork
  begin
    begin
      send();
      recv();
    end
    check();
  end
join
```

Join Options

```
fork
   statement1;
   statement2;
   statement3;
join | join_any | join_none
statement4;
```



join - child threads execute and all child threads must complete before statement4 is executed

join_any - child threads execute and one child thread must complete before statement4 is executed. Other child threads continue to run.

join_none - child threads are queued,statement4 executes. Child threads not executed until parent thread encounters a blocking statement or completes

Thread Execution

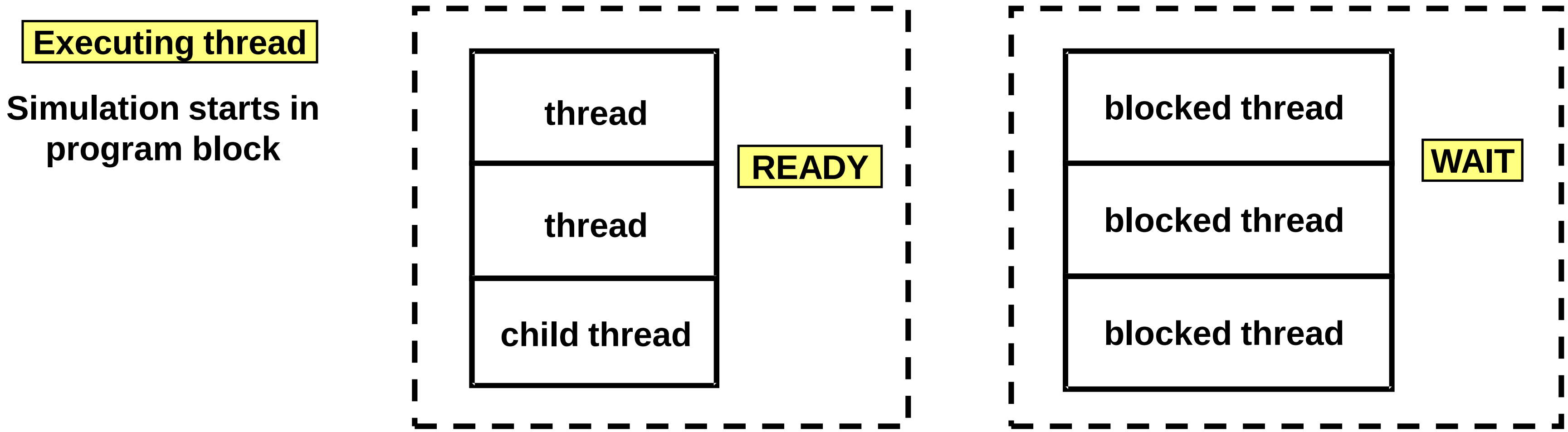
- **Once a thread executes, it continues to execute until it finishes or a blocking statement is encountered**
  - Child threads generated by it are queued
- **When executing thread encounters a blocking statement, it is queued and a queued ready thread executes**
- **Time advances when all threads are blocked**

Examples of blocking statements:

```
@(rtr_io.cb);
wait (var_a == 1);
#10;
join_any *
join *
```

Thread Execution Model

- **One executing thread, all other threads reside on queues**
  - READY - to be executed at current simulation time
  - WAIT - blocked from execution until wait condition is met
- **When the executing thread goes into a wait state, it moves to a WAIT queue, the next READY thread then executes**
- **Simulation time advances when all threads are in WAIT**



Schedule for execution at current simulation time

Moves to READY queue when wait condition is met

Thread Design (1/2)

```
a = 0;
fork
  begin: thread_1
    while ( a != 5 )
      if ( $time > MAX_TIME )
        $finish;
  end

  begin: thread_2
    repeat(5) @rtr_io.cb;
    bus.cb.reg <= 1'b1;
    a = 5;
  end
join
```

Will this work?

Thread Design (2/2)

In multi-threaded programs, all threads must be finite or advance the clock!

```
a = 0;
fork
  begin
    while ( a != 5 )
       if ( $time > MAX_TIME )
         $finish;
      else
         @(bus.cb);
  end
  begin
    repeat(5) @rtr_io.cb;
    bus.cb.reg <= 1'b1;
    a = 5;
  end
join
```

Sharing Variables Among Threads

```
program automatic fork_join1;
  initial begin
    int a = 0, b = 1;
    fork
      begin
        int d=3;
        a = b + d;
      end
      begin
        int e=4;
        b = a + e;
      end
    join
    $display("a = %0d", a);
    $display("b = %0d", b);
  end
endprogram: fork_join1
```

Child threads share the same parent variables

Can the child thread access a and b?

What are the final values of a and b?

Thread v/s Program Completion

```
program automatic test();
  initial begin
    for (int i = 0; i < 16; i++)
      send(i);
  end
  task send(int j);
    fork
      begin
        $display("Driving port %0d", j);
        #1ns;
      end
    join_none
  endtask: send
endprogram: test
```

Simulation ends at time 0. Why?

Waiting for Child Threads to Finish

- **To prevent improper early termination of simulation, one can use `wait fork`**
  - suspends thread until all children have completed execution

```
program automatic test();
  initial begin
    for (int i = 0; i < 16; i++)
      send(i);
    wait fork;
  end
  task send(int j);
    fork
      begin
        $display("Driving port %0d", j);
        #1ns;
      end
    join_none
  endtask: send
endprogram: test
```

Blocking statement to control proper termination of simulation (more in later units)

Thread Execution Issues

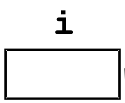
```
program automatic test;
  initial begin
    for (int i = 0; i < 16; i++)
      fork
        send(i);  // illegal – (OK in VCS)
      join_none
    wait fork;
  end
  task send(int j);
    $display("Driving port %0d", j);
    #1ns;
  endtask: send
endprogram: test
```

Produces output:

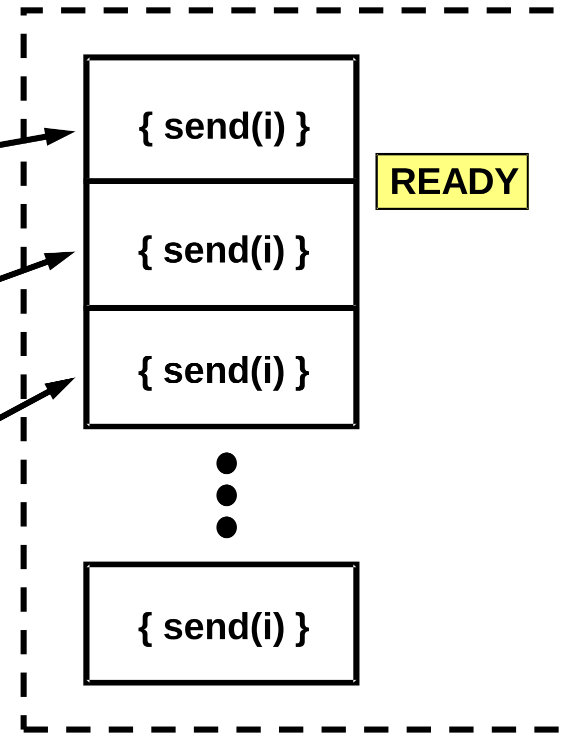
```
Driving port 16
Driving port 16
Driving port 16
Driving port 16
...
Driving port 16
Driving port 16
Driving port 16
Driving port 16
```

Why?

Thread Execution Issues: Unroll the for-loop




```
program automatic test;
  initial begin
    int i = 0;
    fork
      send(i);
    join_none
    i++; // i = 1;
    fork
      send(i);
    join_none
    i++; // i = 2;
    fork
      send(i);
    join_none
    i++; // i = 3;
    … (repeated 16 times total)
    wait fork;
  end
endprogram: test
```



Threads scheduled for execution at current simulation time, but i = 16 before they execute

Thread Execution Issues: Local Variable

- **Local variables:**
  - Once created, variables are local to the child context
  - Can copy parental value at creation

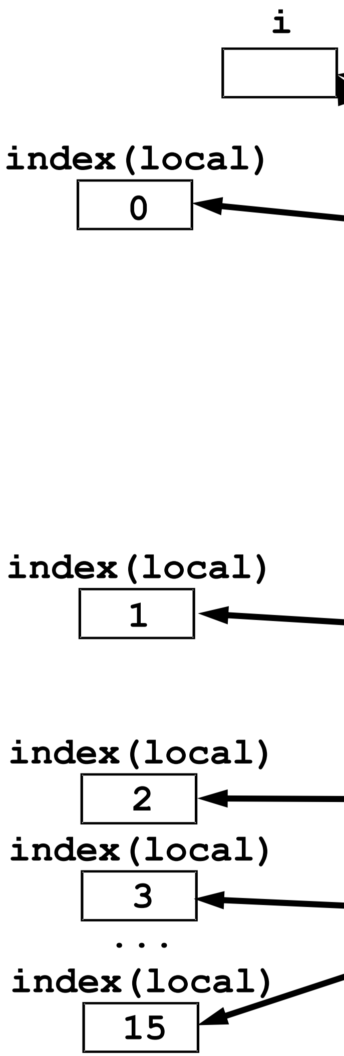
```
program automatic test;
  initial begin
    for (int i = 0; i < 16; i++) begin
      fork
         int index = i; // local fork variable
         send(index);
      join_none
    end
    wait fork;
  end
  task send(int j);
    $display("Driving port %0d", j);
    ...
  endtask: send
endprogram: test
```

Simulation terminates when all procedural code inside `program` block reaches `end`

Desired output:

```
Driving port 0
Driving port 1
Driving port 2
Driving port 3
Driving port 4
Driving port 5
Driving port 6
Driving port 7
...
```

Thread Execution Issues: Unroll the for-loop

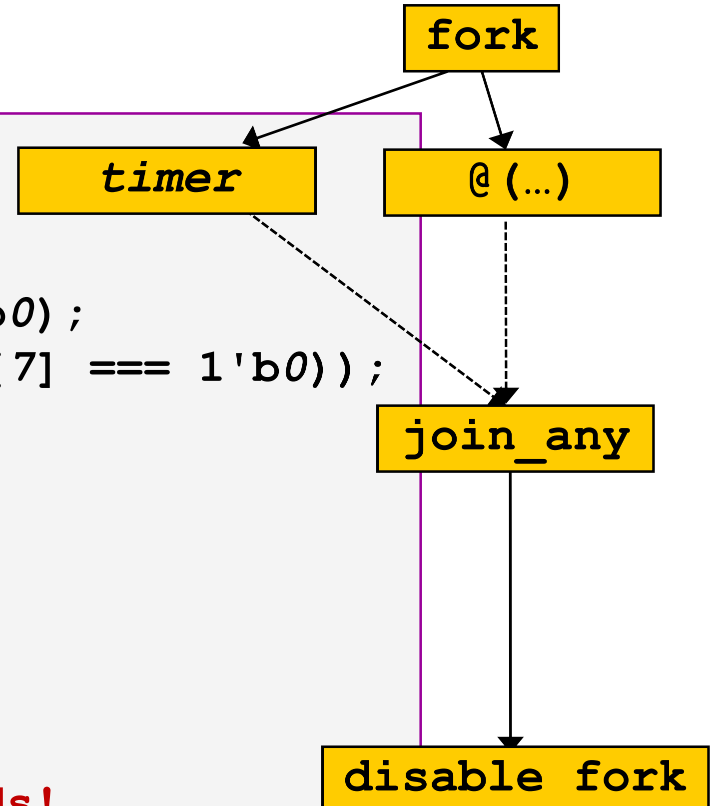



```
program automatic test;
  initial begin
    int i = 0;
    begin
      fork
        int index = i; // index = 0;
        send(index);
      join_none
      i++; // i = 1;
    end
    begin
      fork
        int index = i; // index = 1;
        send(index);
      join_none
      i++; // i = 2;
    end
    … (repeated 16 times total)
   wait fork
  end
endprogram: test
```

variables are created in the declarative space of each fork-join_none

Implement Watch-Dog Timer with `join_any`

- Typically used in conjunction with `disable fork`




```
task recv();
  fork
    begin: frameo_n_negedge
      wait (rtr_io.cb.frameo_n[7] !== 1'b0);
      @(rtr_io.cb iff(rtr_io.cb.frameo_n[7] === 1'b0));
    end
    begin: timer
      repeat(1000) @(rtr_io.cb);
      $display("Timed out!");
      $finish;
    end
  join_any
  disable fork; // kills all child threads!
  get_payload();
endtask
```

See Note

Avoiding disable fork Problems

- **Use enclosing fork join to localize disable fork**
  - Do NOT use disable <block_name> in classes. Kills same-named threads in other objects of same class!

```
task recv();
  fork begin // enclosing fork-join
    fork: recv_wd_timer
      begin: frameo_n_negedge
        wait (rtr_io.cb.frameo_n[7] !== 1'b0);
        @(rtr_io.cb iff(rtr_io.cb.frameo_n[7] === 1'b0));
      end
      begin: timer
        ...;
        $finish;
      end
    join_any
    disable fork; // kill all child threads
//  disable recv_wd_timer // LEGAL BUT DO NOT USE!
  end join
  get_payload();
endtask
```

See Note

Helpful Debugging Features

- **What to print for debugging?**
  - Use `%m` and `$display()` to print the simulation time and location of call

Indicate message type (ERROR, DEBUG, etc.)

Simulation time

```
function void check();
  static int cnt = 0;
  string message;
  if (!compare(message)) begin
    $display("%m\n[ERROR]%t: %s", $realtime, message);
    $finish;
  end
  $display("[NOTE]%t: %0d Packets passed\n",$realtime, ++cnt);
endfunction: check
```

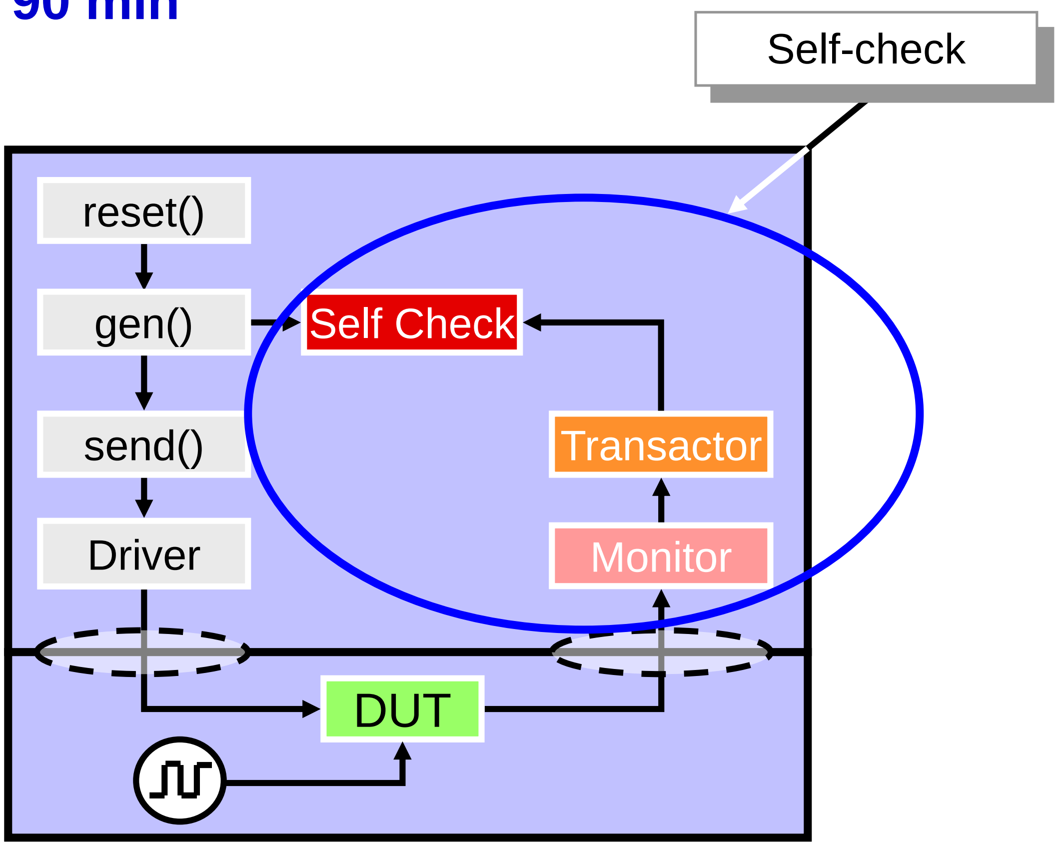
hierarchical path to check()

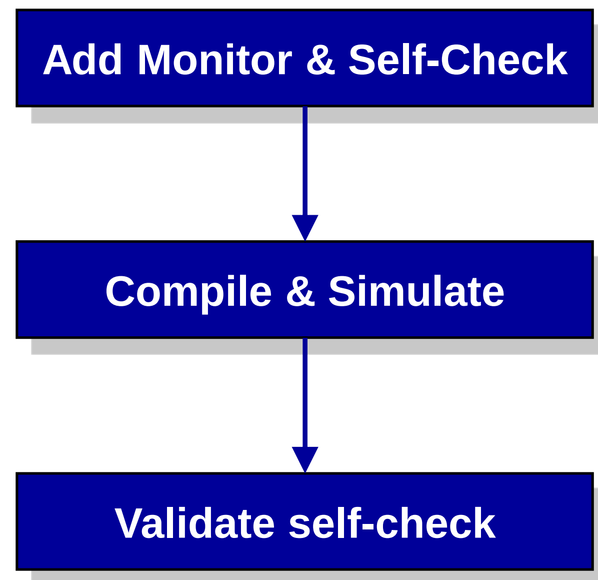
// $timeformat sets the format for %t
// $timeformat [(*units*, *precision*, *suffix_string* , *minimum_field_width*)] ;
// We are using $timeformat(*-9*, *1*, "*ns*", *10*) in the labs
// $time returns time as a 64-bit integer
// $realtime returns time as a real value

Lab 3 Introduction

90 min

Add Concurrent Monitor & Self-check





Unit Objectives Review

Having completed this unit, you should be able to:

- **Divide a testbench into multiple current threads to execute parallel tasks**

Appendix

Alternatives to `disable fork`

Alternatives to `disable fork`

Alternatives to `disable fork` – `kill()`

- SystemVerilog allows you to store handles to threads created by the `fork-join` structure
- The mechanism is a data type called `process`
- Use `process::self()` to retrieve and store the handle to the thread where the method is called
- The thread handle can then be used to kill the thread using the `kill()` method

Get and Save Thread Process Handle

- **Store thread process handles**

```
task recv();
  process thread_q[$];
  fork
    begin
      thread_q.push_back(process::self());
      wait (rtr_io.cb.frameo_n[7] !== 1'b0);
      @(rtr_io.cb iff(rtr_io.cb.frameo_n[7] === 1'b0));
    end
    begin
      thread_q.push_back(process::self());
      repeat(1000) @(rtr_io.cb);
      $display("Timed out!");
      $finish;
    end
  join_any
// see next slide for remaining code
```

Managing Time Consuming Threads

- **For threads which consume time**
  - Use process **`kill()`** method to terminate thread for all processes in queue

```
…continued from previous slide
  foreach (thread_q[i])
    if (thread_q[i].status != process::FINISHED)
      thread_q[i].kill();
  thread_q.delete(); // once killed, remove from queue
endtask
```

Clean up threads

Managing Non-Time-Consuming Threads

- **For threads which may terminate in 0 simulation time**
  - Use dynamic array to store thread process handles
  - Make sure all threads had a chance to start before using **kill()** method to terminate remaining threads

```
task recv();
  process threads[] = new[2];
  foreach threads[i] begin
    fork
      int thread_index = i;
      begin
        threads[thread] = process::self();
        case thread_index
          0: begin wait (...); ...; end
          1: begin ... end
        endcase
      end
    join_any
  end
  foreach (threads[i]) wait (threads[i] != null);
  // Clean up threads - see previous slide and note
endtask
```

Use dynamic array to store thread process handle

Threads which may terminate in 0 simulation time

Ensure thread process started.
Otherwise, **kill()** may not work properly.