Advanced Concepts in Simulation Based Verification

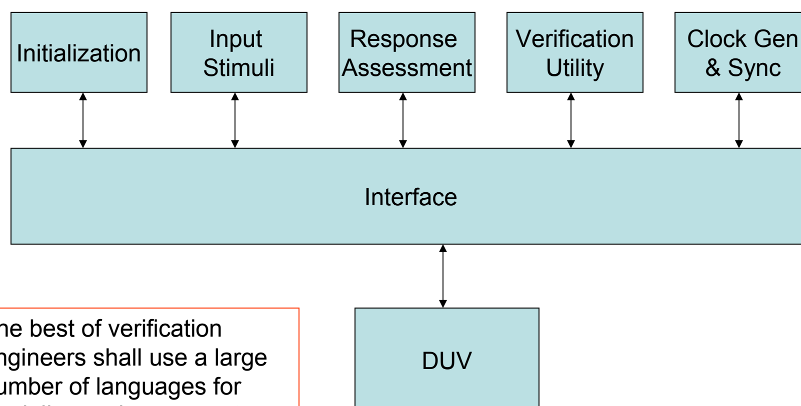
Topics planned to be covered

- Test Bench Organization and Design
- Test Scenarios, Assertions and Coverage
  - Checking and Coverage Analysis in relation to Specman

Test Bench Organization & Design

-- Simulation Based Verification is all about writing proper test-benches

Components of a test-bench



The best of verification engineers shall use a large number of languages for modeling various aspect…

We know e, verilog, C, C++…

How to invoke verilog from e

Using HDL Tasks and Functions

- In TCMs, it is possible to call HDL tasks and functions directly from e code
- Useful because some codes are better written in certain languages
- We wish to use the best of all…
- From industry's point of view, there are some legacy codes in verilog, C/C++ which are tested and hence are proven

Bus Functional Models
(One Such Case)

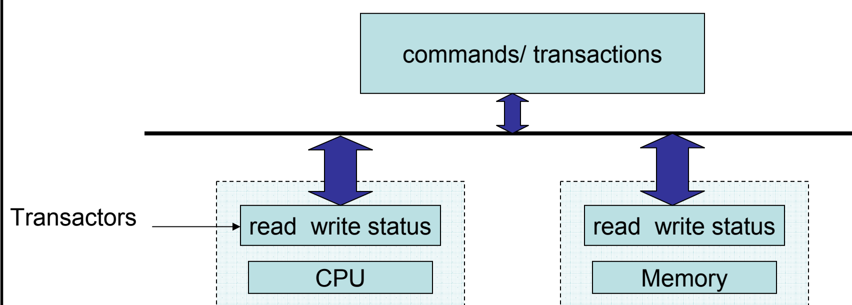
Bus Functional Models

- Encapsulates detailed operations among the test-bench and the device under verification as high level procedures
- High level bus instructions, instead of bit patterns, are issued
- Instructions are translated into lower level bit values and applied to the design
- Interactions between the test-bench and the DUT are at the transaction level

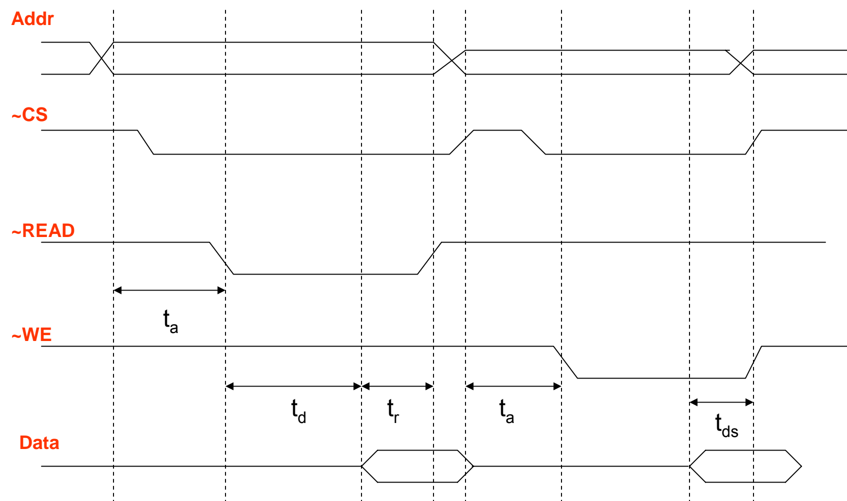
BFM (contd..)

- Create a wrapper that enables the device to receive and send bus commands, and the wrapper disassembles/assembles the commands to/from bits
- Wrapper is called an interpreter or transactor
- Verification environment becomes easy to maintain

Components and Structures



Memory read/write Timing Diagram



Transactors using Verilog Task

```
• task read_memory;
   input [31:0] in_address;
   output [31:0] out_data;
   begin
      addresstemp <= in_address;
      CS<=1'b0;
      #`ta READ <= 1'b0;
      #'td out_data <= data;
      #'tr READ <= 1'b1;
      CS <= 1'b1;
   end
 endtask
```

Transactors using Verilog Task

```
• task write_memory;
    input [31:0] in_address;
    input [31:0] in_data;
    begin
      addresstemp <= in_address;
      CS<=1'b0;
      #`ta WE <= 1'b0;
       dataread <= in_data;
      #'tds WE <= 1'b1;
      CS <= 1'b1;
    end
  endtask
```

The complete BFM

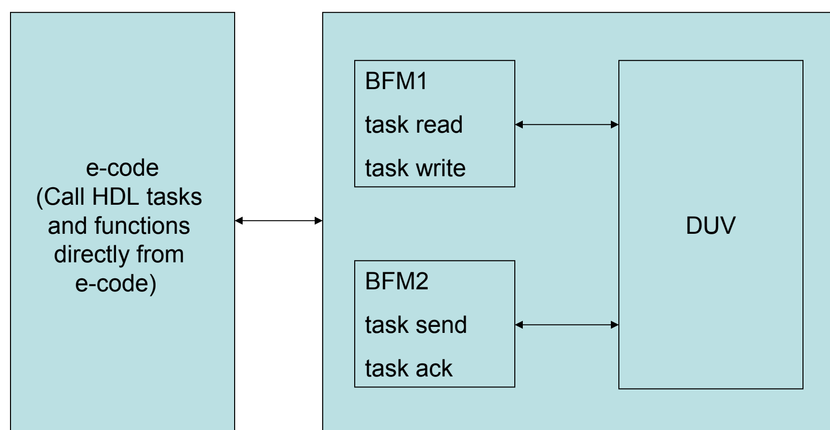
```
module memory_BFM(address,data);
input [31:0] address;
inout [31:0] data; reg[31:0] dataread, addresstemp;
memory
   mem(.CS(CS),.read(READ),.WE(WE),.address(addresstemp),.data(data));
assign data = (READ?) out_data:dataread;// as inout is a wire…
task read_memory;
…
end task
task write_memory;
…
end task
endmodule
```

Test-Bench in verilog

```
• module testbench;
   memory_BFM mem(.address(address),.data(data));
   always @(posedge clk)
   begin
      mem.write_memory(addr,data);
      @(posedge clk)
      mem.write_memory(addr,data);
      @(posedge clk)
      mem.read(addr,data);
    end
    endmodule
```

Note, now the BFM has only Transaction-level entities like address and data

Specman & Verilog Tasks



Verilog Task through e

```
<' struct mem_w{
      addr: int;
      data: int(bits: 32);
   };
   unit receiver{
      verilog task 'top.write_mem'(addr:32:in,data:32:out);
      put_mem(mw:mem_w) @mem_write_enable is{
         'top.write_mem'(mw.addr,mw.data);
   };
};
'>
```

See for "verilog function"…Page 162, Palnitkar's Book

Initialization

- All good circuits should initialize after power on
- However it is required to maintain separate initialization constructs in the test-bench, as:
  - Simulation starting from its legal state shall take a lot of time
  - Simulation emulates an exceptional condition that the normal sequence starting from the legal initial state will not reach.

Initialization

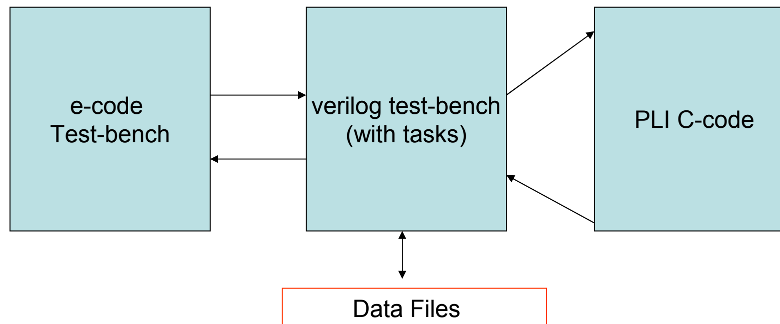
- Hard coding initialization blocks in the test-bench is not a good practice, instead describe methods
  - initialize(…);
- Do not embed the test-bench initialization block into the DUT

Sometimes using verilog and PLI test-benches s can be handy

- Reasons:
  - File Management ($readmemh)
  - PLI supports efficient searching for state elements and memory, and initalize them
  -

```
void initialize_flipflop( ){
 db=fopen("database","r");
 module=acc_fetch_by_name("my design");
 cell=NULL;
 while(cell=acc_next_cell(module,cell){
    if(cell is sequential){
       port=acc_next_port(cell,port);
       if(port is output){
          get_init_value(db,port,&value);
          acc_set_value(port,&value,&delay);
       }
    }
  }
}
```

Integrating the testbenches



Clock Generation Module

- In Specman we have seen examples of generating clock
- A circuit may have multiple clock domains
- Generate them independently, if they are so
- Write Clock Multipliers and dividers separately

Clock Dividers

- i=i%N; //Divide by N
  if(i==0) derived_clock=~derived_clock;
  i=i+1;
  (This is not the best way to divide in hardware, but for test-benches it is more compact and without any hardware components)

Clock Multipliers(verilog)

- always @(posedge base_clk)
  begin //if N is known before hand
  repeat (2N) clock = #(period/(2N)) ~clock;
  end
- forever clock = #(period/(2N)) ~clock;
  //if N is not known before hand

Lab exercise

- Write an e module to call a verilog task to generate three clocks: Clock 1, 2 and 3.
  We do it so that, Clock 2 is a divided by 2 and Clock 3 is a multiplied by 2 clock…

Modelling Jitter

- Jitter is a common phenomenon in digital design, we may need to verify in such an environment
- Verilog Modelling:
  – initial clock1=1’b0;
    always clock1 = #1 ~clock1;
    jitter=$random(seed) % RANGE;
    assign clock1_jittered= #(jitter) clock1;

How can we model jitter in ‘e’?

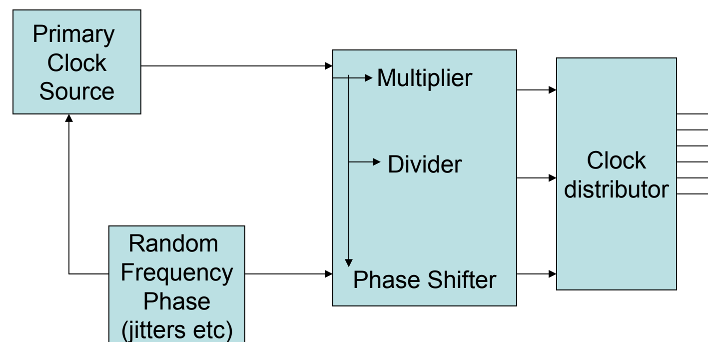
Clock Synchronization

- Independent waveforms should be first synchronized for various reasons
- always (fast_clock)

  clock_synchronized<=clock1

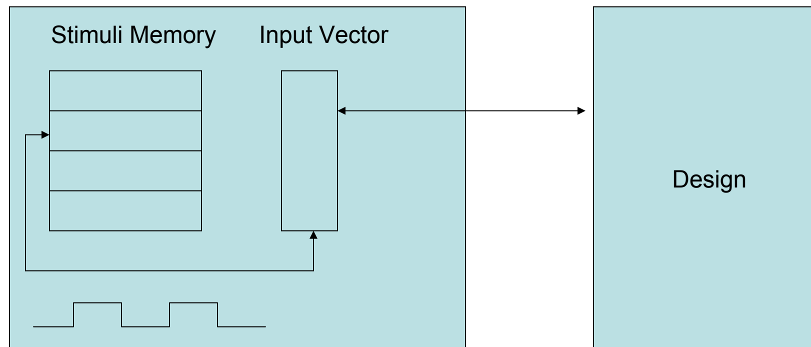
//fast clock is the fastest clock in the design

Clock Generator Network



Stimulus Generation

- Synchronous Stimuli

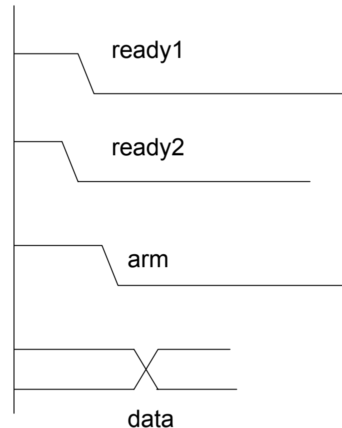


Corresponding Code in Verilog

```
• reg [M:0] input_vectors [N:0];
  reg [M:0] vector;
  initial begin
    $load_memory (input_vectors,"stim_file");
    i=0;
  end
  always @(posedge stimulus_clock)
   begin
      if(apply_input==TRUE) begin
        vector = input_vectors[i];
        design.address<=vector[31:0];
        …
        design.address<=vector[M:M-31];
      end
    end
```

Asynchronous Generation

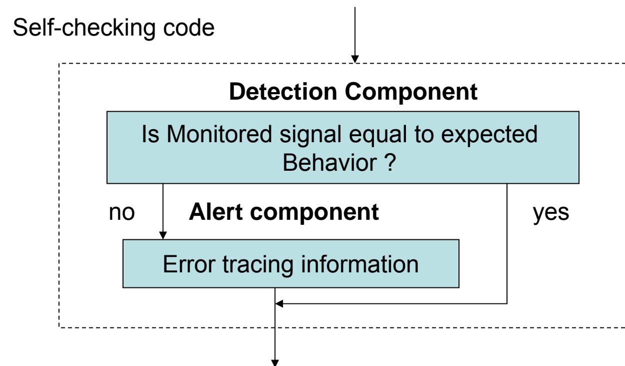
```
• always
  begin
    @(ready1 or ready2)
     arm = ready1 | ready2;
   end
   always @(negedge arm)
    begin
     transmit_data();
    end
```



Self Checking Codes

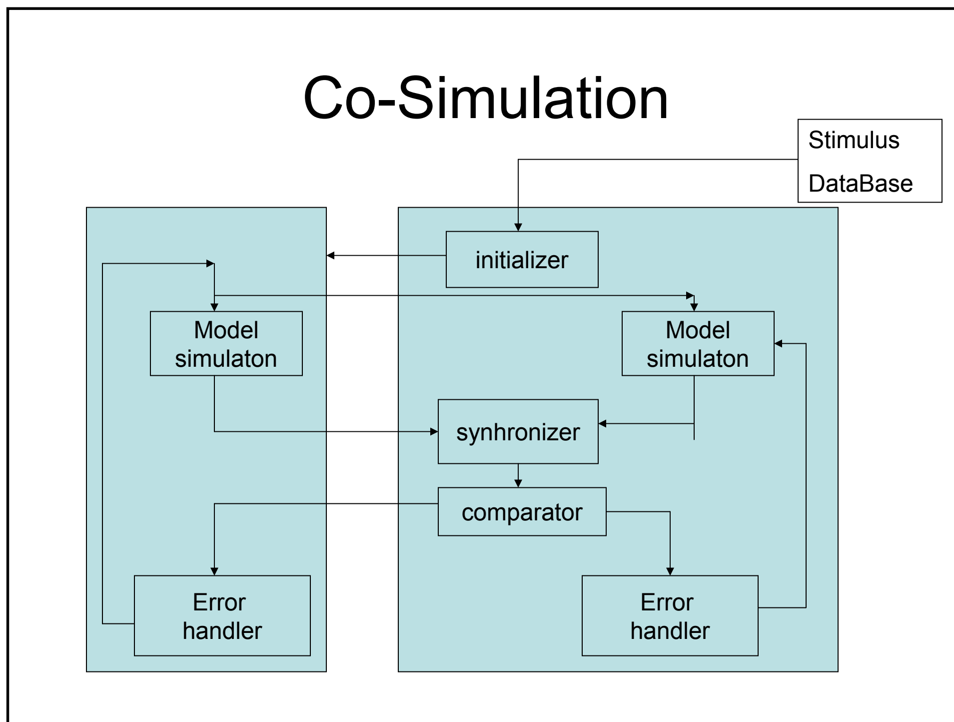
- Dumping signals and comparing with golden responses slow down the simulation process
- Checking is moved to the test-bench, so that signals are monitored and compared against, continuously
- Technique is called self-checking
- Consists of two parts: detection and alert

Self-checking test-bench structure



Example of a self-checking test-bench for a Multiplier

```
multiplier inst(.in1(mult1),.in2(mult2),.prod(prod));
--------------------------------------------
expected = mult1 * mult2;
--------------------------------------------
if(expected != prod)
begin
  $display("ERROR: incorrect product");
  print the exception values…
end
```



Co-simulation (contd.)

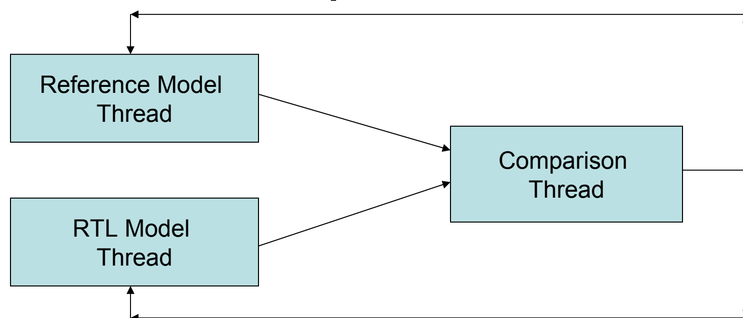
- Assume that DUV is a microprocessor and the reference model is instruction level accurate
- After both models are initialized, the RTL model is started with the reset signal
- Reference model also simulates off its memory
- Then they resynchronize after each instruction (instruction level accurate)
- Exchange signals like instruction_retired

Re-synchronization

- When the instruction_retired signal rises, the RTL model blocks and passes the register values and memory to the ref. model for comparison (call back in 'e')
- always @(instruction_retired)

```
begin
 if(instruction_retired)
   begin
     halt_simulation;
     $pass_states_for_comparison;
     resume_simulation;
   end
 end
```

An Implementation



- A Sample Implementation can be through semaphores

An Implementation

```
•    //reference model thread

•    void execute_instructions()
{
 done=0;
 while(!done){
   next_pc(&pc);
   execute_instr(pc);
   sem_incr(&ref_comp);
   sem_wait(&ref_resume);
    }
}
```

```
•    //RTL model thread

•    void pass_states_for_compare()
   {
    …
    instruction_thread = tf_get(1);
    gr1=tf_get(2);
    …
    sem_incr(&rtl_comp);
    sem_wait(&rtl_resume);
    }
•    // comparator thread

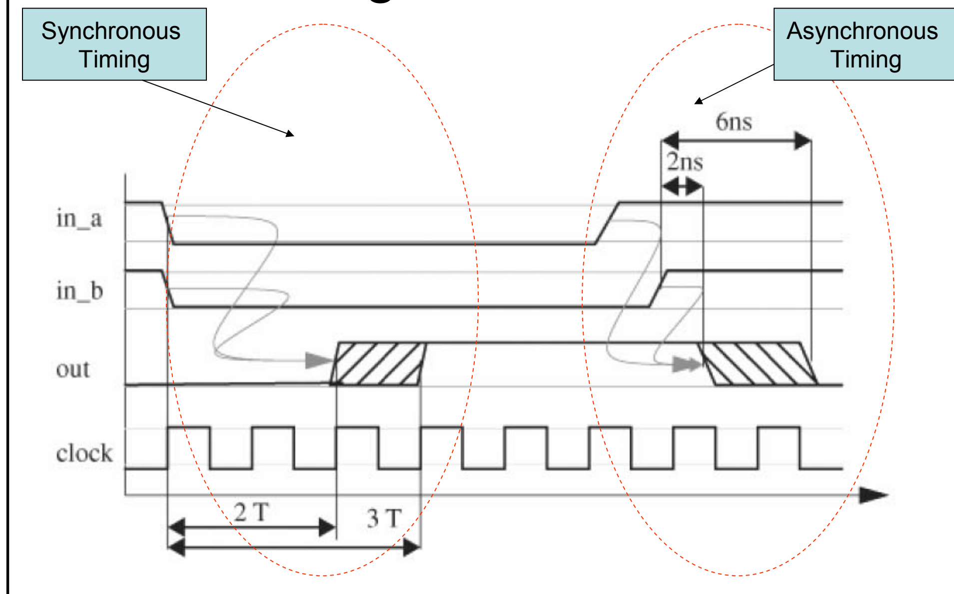
•    void compare_thread()
    {
      sem-wait(&rtl_comp);
      sem_wait(&ref_comp);

      if(!errors){
            sem_incr(&rtl_resume);
           sem_incr(&ref_resume);
       }
      else //handle errors
    }
```

Checking Temporal Expressions

- Timing verification requires to express timing requirements in terms of clock cycles (synchronous) and absolute time intervals (asynchronous)
- Ex:
  - Out must rise between second and third clock cycles after the lowering of both in_a and in_b
  - Out must rise between 2 and 3 ns after the lowering of both in_a and in_b

Timing to be verified



Lab Exercise

- Write the corresponding e-codes if the timing is synchronous (i.e 2 and 3 are clock cycles).

Asynchronous Timing : Fork and Join

- condition = in_a & in_b;

```
…
@(posedge condition)
 begin
    arrived=1'b0;
    fork: chk_lower_lmt;
       #2 disable chk_lower_lmt;
       @(negedge out) arrived=1'b1;
     join
   if(arrived==1'b1) error("lower lmt time is violated");
  end
```

Asynchronous Timing : Fork and Join

- condition = in_a & in_b;

```
…
@(posedge condition)
 begin
    arrived=1'b0;
    fork: chk_upper_lmt;
       #6 disable chk_upper_lmt;
       @(negedge out) begin
               arrived=1'b1;disable chk_upper_lmt; end
     join
   if(arrived !=1'b1) error("upper lmt time is violated");
  end
```

Checking for absence of Transitions

- @(negedge CS)

```
begin
  fork: stable_address;
    @(address[0] or address[1] or … address[31])
    $error("address changing while accessing");
    @(posedge CS)  disable stable_address;
  join
end
```

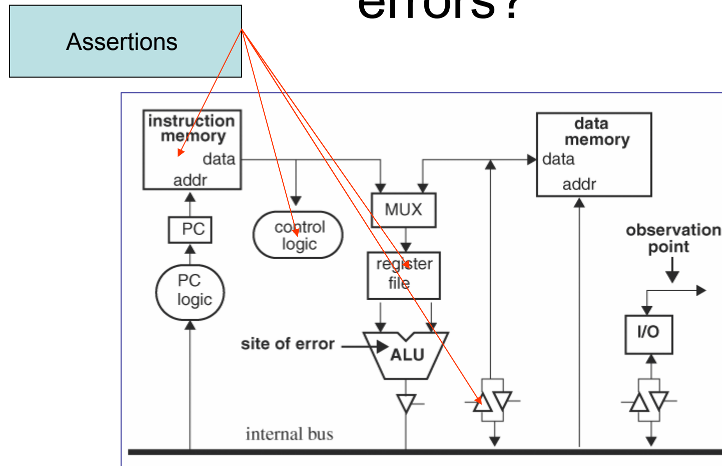
Test Scenarios, Assertions and Coverage

Verification Space

- When shall you say that a sequential circuit is completely verified?
  - If all its reachable states are visited from an initial state and all transitions from that state are verified.
- If there is a sequential circuit with S states, and R is the number of possible inputs, then the number of input patterns is SR

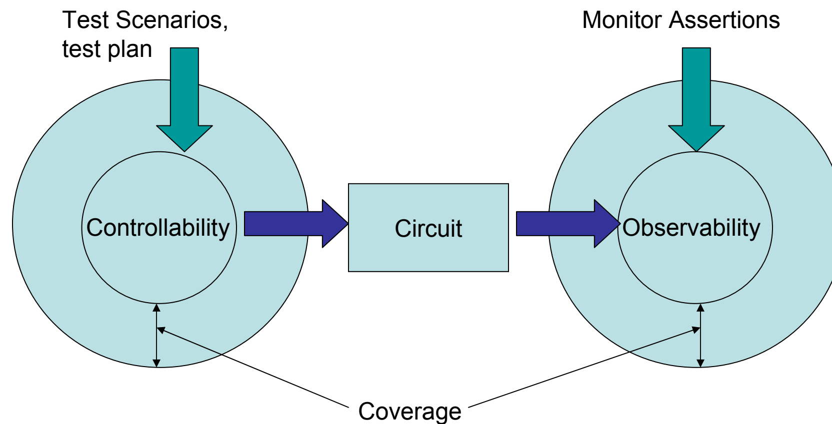
- So, that is not possible!
- So, our goal is to simulate the design over a well-selected subset of all inputs in a systematic manner, so that my level of confidence is also high
- So, we need metrics for coverage analysis

How do we know when we have errors?



Checking the output response is not the best idea, because of latency problems (Why?). It is also possible that the error is not detected.

Relationships among test scenarios, assertions and coverage



Test Plan: Extracting Functionality from Architectural Specifications

- Represent a Digital System as a finite state machine:
  - Q_0: Initial States
  - S: Valid Initial States
  - I: Valid Inputs
  - O: Valid Outputs
- $X\text{->}Y|_I$: Transition from X to Y under valid input I
- $X\text{=>}Y|_I$: Sequence of transitions from X to Y under application of consequent inputs
- Ω is a don't care state

Generate $\Omega \to \Omega|_\Omega$

$\Omega \to \Omega|_\Omega$

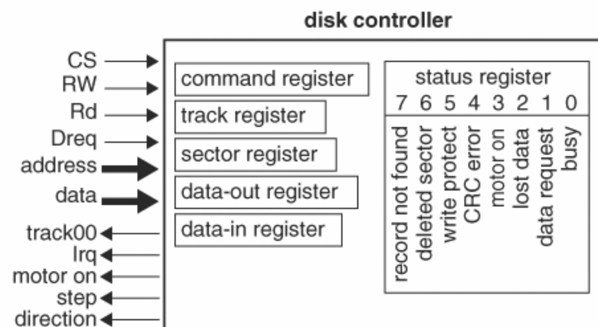
1. $\Omega \to \Omega|_{\bar{I}}$: Design receives an invalid input
2. $\bar{S} \to \Omega|_\Omega$: Design is in an invalid state, how does it recover?
3. $S \to \bar{S}|_I$: Design starts in a valid state, goes to an invalid state (bug)
4. $S \to S|_I$: Design behaves as per the specifications
5. $\Omega \Rightarrow Q_0$: Design on power on enters into a valid initial state from any state
6. $\Omega \Rightarrow \bar{Q}_0$: Design on power on enters into an invalid state (bug)

- Prove that this covers the entire input and state space…
- Combine 5 and 6
- Combine 3 and 4, then with 2 and then with 1.

Example of a Disk Controller

- Generate test scenarios for a disk controller, using the state space approach.
- Architectural Specifications:
  - A disk is partitioned into concentric circular strips called tracks and each track is further partitioned into sectors.
  - There are 6 types of registers: command, track, sector, data-out, data-in and status
  - Commands can be Restore, Seek, Step-in, Step-out, Read(track and sector), Write(track and sector), Address, Interrupt

Disk Controller



Possible Test Scenarios

- **Category 1:** Give illegal inputs, like illegal address, consecutive data requests without a grant and see how the design responds.
- **Category 2:** Set the controller in an illegal state (may be an illegal op-code) and observe whether the controller detects and recovers from it.

Possible Test Scenarios

- **Category 3:** Assume that the controller is in a valid state, and attempt to drive it to an invalid state with valid inputs. For example, are there track or sector values for which the controller produces illegal pulses?

- **Category 4:** Assume that the controller is in a valid state, apply valid inputs and see whether the controller goes to an invalid state. Apply the valid track and sector values and check whether the response signals are correct.

Possible Test Scenarios

- **Category 5:** What are the register values when the device is powered on? Are they correct?
- **Category 6:** Under what conditions will the controller enter an illegal state on power on? What input patterns could drive the design to abnormal conditions, like motor on is continuously on, on power-on?

Effective in creating the test-scenarios…

Writing Assertions

Combinational & Sequential Assertions

- An assertion is combinational if all its terms are so; otherwise its sequential
- To code combinational assertions, only combinational logic is required
- To code sequential assertions, a FSM is required
- The time interval between the most past and the most future is known as the window of the expression

Signal Range

```
• always @(posedge clk)
  begin
    if((rdy_to_chk==1'b1)&&('LOWER >
  S||S>'UPPER'))
      $display(…);//error
   end
```

Unknown Value

- Checking whether signal A is equal to x will not work, as none of the bits should be an unknown value.
- Trivial solution shall be to check each bit of A compared to 1’bx
- Better solution is to use a reduction xor operator
- By defn of xor, if one of the bits is x, so is the output.
- if(^A==1’bx) $display(“Unknown value”);

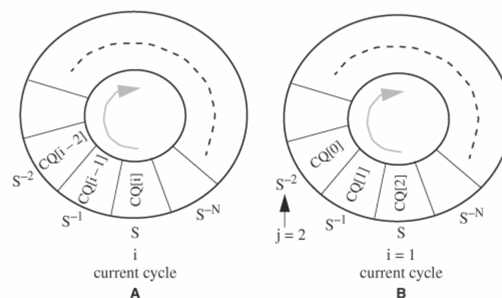
One hot signals

- A signal is one-hot if exactly one of the bits is 1 at any time.
- if(|(B & (B-’width’b1))!=1’b0)
  $display(“Bus B is not one hot);
- Example: B=8’b00001000 //decimal 8
  B-8’b1=8’b00000111 //decimal 7
  B & (B-8’b1)=8’b00000000
- Workout an example where the input is not one hot
- 8’b0 is not one hot, but will bypass this test, so keep an additional check for this…

Sequential Assertions

- Property: A gray coded signal changes exactly one bit at a time
- A = prev_S ^ S;
  if(~((A==0)||(A & (A-1)))
   $display(“….error…”);
- prev_S = curr_S; //at the pos. edge of clock
  curr_S=S;

Circular Queue




```
always @(posedge clk)
  begin
    i=i+1 % (N+1);
    CQ[i]=S;
    j=(i-k>=0) ? i-k : i-k+N+1;
    prev_k_S=CQ[j];
   end
```

Example

- If req is 1, ack goes high 3 cycles later, which causes req to go low seven cycles later
- Using circular queue:
  - Assertion:

```
if(prev_10_req == 1'b1)
  if(!(prev_7_ack==1'b1 && req == 1'b0))
   $display("assertion failed");
```

```
always @(posedge clk) begin
 i=(i+1)%11;
j1=(i-7>=0) ? i-7 : i-7+11;
j2=(i-10>=0)?i-10:i-10+11;
if(CQ_req[j2]==1'b1)
   if(!(CQ_req[j1]==1'b1) && (req == 1'b0)) $display("error");
end
```

Assignments for Lab

- Write the e-code for the above assertion
  - Using circular queue
  - Using temporal expressions

(I am still waiting for Tutorial-3 !!!)

Unclocked Timing Assertions

- Ensure that S remains steady throughout the interval marked by events E_1 and E_2. The assertion is active when start signal is high.

- Two named blocks are created : one waiting on E1 and the other waiting on E2. If the assertion fails before E2 arrives an error is displayed and block for E2 is disabled. If E2 arrives before any failure then block for E1 is disabled.

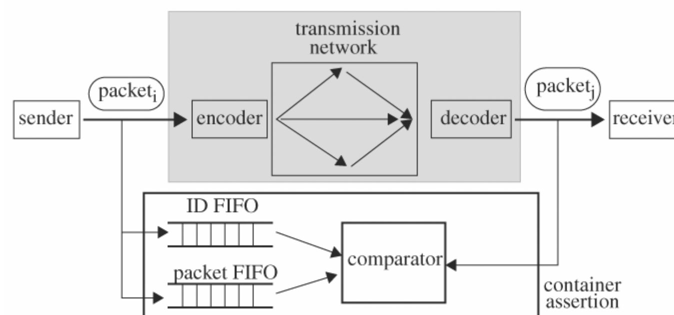
Verilog Assertion

- ```
  always @(posedge start)
  begin: check
     @(E1);
     @(S) $display("Error");
          disable stop;
   end
  always @(posedge start)
  begin: stop
     @(E2); disable check;
  end
  ```
  ```

Possible Questions??

- Write the above code in e. (How can you disable tcm's in e ?)
- A trivial solution can be to gate the trigger event of the tcm required to be disabled.

Container Assertions



- These assertions check the integrity of the data without knowing the exact values. Example, the cache data is unaltered from being filled to it being accessed, packet received is the same as that being sent…even though it has been processed. Can be quite efficiently implemented using 'e' because of the 'push' and 'pop' commands in 'lists'…

Coverage

3 Types of Coverage

1. **Code Coverage:** insight to how thoroughly the code is executed by simulation

2. **Parameter Coverage:** Reveals the extension that dimensions and parameters in functional units of a design are stressed

3. **Functional Coverage:** This accounts for the amount of operations features or functions in a design that are exercised

They are complementary

- Code and parameter coverage are with respect to an implementation and are thus are more easy to compute
- Functional Coverage is based on specification and hence are more objective but are difficult to compute

Example

- Consider a 64 bit adder wrongly designed as a 60 bit adder.
- We can obtain 100 % code coverage and the design can still have a bug.
- A parameter coverage shall be better.
- But what if the carry does not propagate correctly? We require functional coverage.

Code Coverage

- Statement Coverage: Collect Statistics about statements that are executed in a simulation.

```
➡ line 1: always @(posedge clock)
  line 2: begin
➡ line 3:   a = b + c;
➡ line 4:   x = (y << 4);
➡ line 5:   if (a > x)
  line 6:   begin
➡ line 7:      y = b & a;
➡ line 8:      x = (x >> 2);
  line 9:   end
  line 10:  else
  line 11:     b = b ^ c;
➡ line 12:  if ( x == y)
➡ line 13:     c = y;
  line 14:  else
  line 15:     y = a;
  line 16: end // end of always
```

Excluding the begin & end there are 10 lines
⇒Statement Coverage=80%

Block Coverage

- Coverage of some statements imply that of other statements.

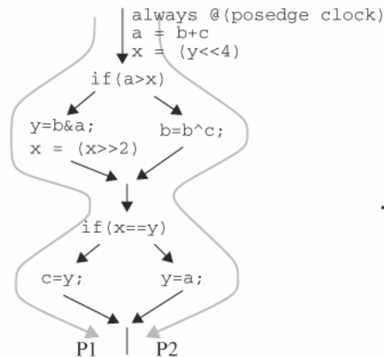
```
line 1: [always @(posedge clock)]   block 1
line 2: begin
line 3:   [a = b + c;    ]           block 2
line 4:   [x = (y << 4); ]
line 5:   if (a > x)
line 6:   begin
line 7:      [y = b & a;    ]        block 3
line 8:      [x = (x >> 2); ]
line 9:   end
line 10:  else
line 11:     [b = b ^ c;]            block 4
line 12:  if ( x == y)
line 13:     [c = y;]                block 5
line 14:  else
line 15:     [y = a;]                block 6
line 16: end // end of always
```

BLOCKS

Coverage: 66.67%

Path Coverage

- Measures the percentage of path exercised
- Path grows exponentially with the number of conditional statements
- If P1 is true, path coverage is 25%



Expression Coverage

- Sometimes the transition from a state to another depends on complicated expressions
- Question is how does the expression evaluate?
- Any complicated expression can be broken down in levels
- For example: (x1x2+x3x4) evaluating to 1, may be deeper investigated to see whether x1x2 or x3x4 or both evaluates to 1.
- Helps to guide test benches to exercise as many parts of the expression as possible

Expression Coverage(contd.)

- Layer 1: E=y1 + y2;
- Layer 2: y1=x1x2, y2=x3x4

f = x_1 & x2	x_1	x_2
0	0	
0		0
1	1	1

Minimum Input Table of an AND logic

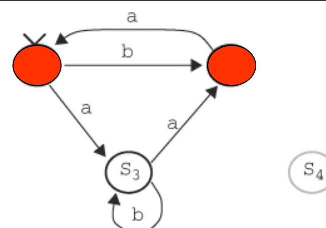
Expression Coverage is the ratio of the cases exercised to the total number of cases possible (rows in the minimum input table)

State Coverage

- Calculates the number of states visited over the total number of states in a FSM.
- initial pres_state=‘S1;

```
always @(posedge clk)
 case((pres_state,in))
  (`S1,a) : next_state=`S3;
  (`S2,a) : next_state=`S1;
  (`S3,a) : next_state=`S2;
  (`S1,b) : next_state=`S2;
  (`S3,b) : next_state=`S3;
 endcase
```

Simulation generates seq…a, b, a, b, …



Transition Coverage

- Records the percentage of transitions traversed, and also the frequency.
- Missing Transition: What happens if the input is b at S2?
- In the above FSM, with the sequence of a, b, a, b, … the transition coverage is 33.3%
- Total number of transitions should be the total number of states multiplied by the number of possible inputs.

Sequence Coverage

- A state sequence is the sequence of states that the FSM traverses under an input sequence.
- Each design (specification) shall have some sequences defined as legal and some defined as illegal transitions.
- Provides information about which legal state sequences are traversed and which illegal sequences are encountered.

Sequence Coverage

- For the previous run of a, b, a, b, …
  the sequences are S1->S2 and S2->S1
- Legal Sequences: S1-> S2; S2->S1; S1->S3; S3->S2; S3->S3
- Sequence coverage is 40%
- State Coverage is 66.67%

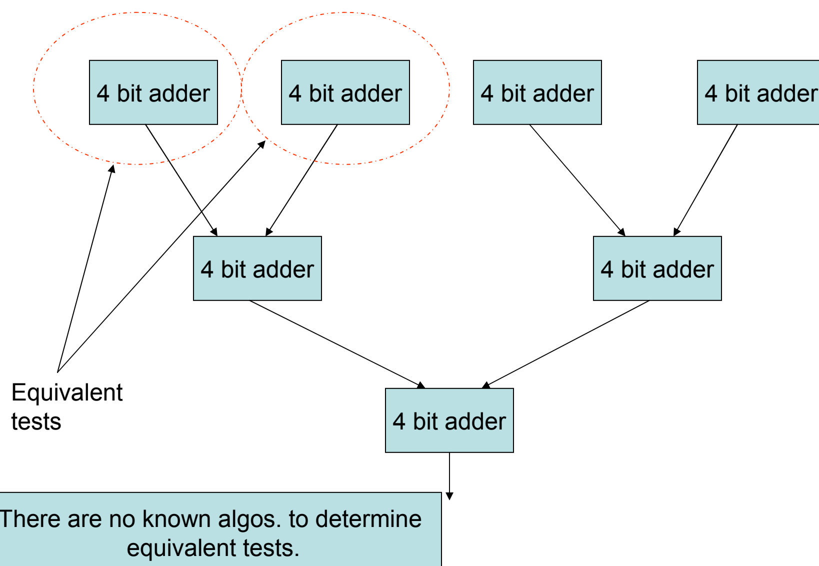
Parameter Coverage

- Code Coverage is only a starting point
- An empty verification suite has 100% code coverage
- To gain further insight into operational correctness, parameter coverage is used to measure functional units.
- Lets consider an example of *verifying a stack*

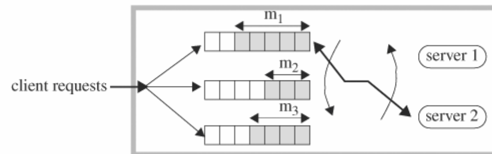
Stack

- Has depth…
- Verifying pop and push, verifies only a point in the operational space.
- Possible parameters: Stack depth
- Parameter coverage records the depths encountered throughout the simulation
- A good verification code should also check the full and empty conditions

Parameters & Equivalent checks



Three queue, 2 server system



- Spec: Clients can arrive from any source & wait in the queue. Server retrieves request from the 3 queues, if it is empty the server moves to the next. Can have a processing time varying from 3 to 6 units, and depth is 7.
- Parameters: Server status, Task processing time, maximum length of queue, m_i

Parameter Coverage Matrix

Parameter	Range
Server	$\{(S_1,S_2)$: S_1, S_2 is either on or off$\}$
Task processing time	$\{3,\ldots,6\}$
Queue length	$\{(m_1,m_2,m_3)$: $0 \le m_i \le 7$, $1 \le i \le 3\}$

- Set up monitors in the verification suite.
- 2 bits for server status
- Measure the time for processing, see if it was recorder before, if not record it.
- All possible lengths should be noted, for simplicity we store the maximum value.
- Compute the coverage.

Functional coverage

Its Different

- Code Coverage & Parameter Coverage measures how much of the design has been measured;
- *Functional Coverage measures how much of the design specification has been measured.*

Example

- Decoding of a CPU instruction may have separate case statements in the design
- Due to the combination of previously decoded values you may have a code coverage of 100%
- But the sequence of execution of a specific instruction might be incorrect: Should be trapped in Func. coverage

Points: its not a wonder drug!

- Relevant and Interesting cases must be manually defined
- Did I generate all the relevant cases (with respect to what I have said to be relevant)?
- Define what to sample: If you are interested say in the FIFO occupancy level what do you sample? Difference between pointers.

Contd.

- Where to sample? Opcode of an instruction can be sampled at several paces: decoding unit, execution pipeline, program memory interface
- When to sample? Over sampling will make performance reductions. Under sampling means you can miss bugs.

Cross & Transition Coverage

- Measures the presence or occurrence of combination of values.
- Need to be sampled at the same time.
- Only sample points within the same group can be crossed.
- Measures the n-dimensional space where each point is a vector with components being the covered items.
- Transition coverage measures the occurrence of sequence of values.

100 % Functional Coverage

- …means you have covered all of the coverage points you have included in the simulation.
- It makes no statement about the completeness of your functional coverage model.

One case for cache coherency protocol

- One interesting scenario: Two processors A and B, have two separate caches. When A initiates a read and registers a miss in the cache, it snoops Processor B and asks it to look in its cache. If it is not there, B confirms A and asks main memory to send the data to A. Data is stored in the memory and marked as exclusive.

Monitor

- Check whether its read request.
- If it so, check whether it is a hit in A's cache.
- If no, check whether it is a hit in B's cache.
- If no, emit an event that the scenario has occurred.
- Check whether data is marked exclusive.

Functional coverage 100 % will only ensure that the scenario I have told to be interesting has occurred. It will not check whether the scenario is *correct* or *complete.*

State Machine Coverage using Specman

Code Taken from SNUG'03

```
/*
 -----------------------------------------------------------------------
 -- Module Name         : onehot_moore_fsm5
 -- Encoding            : One-hot
 -- Implementation      : Moore
 -- States              : 5
 -- Outputs             : Registered
 -----------------------------------------------------------------------
*/


module  onehot_moore_fsm5
  (
  // Inputs
  clk_i, rst_i, in1, in2, in3, in4, in5, in6, in7, in8, in9,

  // Output
  out
  );

//--------------------- Global parameters Declarations ----------------

  // Binary states
  parameter       IDLE = 0,  // Idle state
```

```
                  S1   = 1,  // ......
                  S2   = 2,  // ....
                  S3   = 3,  // ...
                  S4   = 4;  // ..

  // One-hot states
  parameter [S4 : IDLE]     IDLE_S = 1 << IDLE,
                            S1_S   = 1 << S1,
                            S2_S   = 1 << S2,
                            S3_S   = 1 << S3,
                            S4_S   = 1 << S4;

//--------------------- Input Declarations ----------------------------

  input  clk_i, rst_i, in1, in2, in3, in4, in5, in6, in7, in8, in9;

//--------------------- Output Declarations ---------------------------

  output [S4 : IDLE] out;

//--------------------- Output Registers ------------------------------

  reg    [S4 : IDLE] out;

//--------------------- Internal Register Declarations ----------------

  reg    [S4 : IDLE] tmp_out;

//--------------------- State Registers -------------------------------

  reg    [S4 : IDLE] next_state, current_state;
```

```
//--------------------- Start of Code ---------------------------------

// Combinational part of FSM

  always @(in1 or in2 or in3 or in4 or in5 or in6 or
           in7 or in8 or in9 or current_state) begin

    case (1'b1)  // synopsys parallel_case

      current_state[IDLE] : begin  // State 1

        if (in1 && in2 && ~in3 && in4) begin
          next_state = S1_S;
        end
        else begin
          next_state = IDLE_S;
        end
      end

      current_state[S1] : begin  // State 2

        if (~in1 || ~in2 || in5) begin
          next_state = S4_S;
        end
        else begin
          if (in9) begin
            next_state = S2_S;
          end
          else begin
            next_state = S1_S;
          end
```

```
          end
        end

        current_state[S2] : begin // State 3

          if (in1 && in2) begin
            if (in6 && in7) begin
              next_state = S3_S;
            end
            else begin
              next_state = S2_S;
            end
          end
          else begin
            next_state   = S4_S;
          end
        end

        current_state[S3] : begin // State 4

          if (in1 && in2) begin
            next_state = S3_S;
          end
          else begin
            next_state = S4_S;
          end
        end

        current_state[S4] : begin // State 5

          if (in8) begin
            next_state = IDLE_S;
          end
          else begin
            next_state = S4_S;
          end
        end

        default : begin

          next_state = IDLE_S;

          // synopsys translate_off
          $display (" FSM is in invalid state, switching to IDLE ");
          // synopsys translate_on
        end
      endcase
    end
```

```
//  Sequential part of FSM - Registering the outputs & state
  always @(posedge clk_i or negedge rst_i) begin
    if (~rst_i) begin
      out           <= 5'b0_0000;
      current_state <= IDLE_S;
    end
    else begin
      out           <= tmp_out;
      current_state <= next_state;
    end
  end


 //  Output generation
   always @(current_state) begin
     case (1'b1)  // synopsys parallel_case
       current_state[S1] : tmp_out = 5'b0_0010;
       current_state[S2] : tmp_out = 5'b0_0110;
       current_state[S3] : tmp_out = 5'b0_1110;
       current_state[S4] : tmp_out = 5'b1_0000;
       default :           tmp_out = 5'b0_0001;
     endcase
   end

 endmodule
```

Observe:

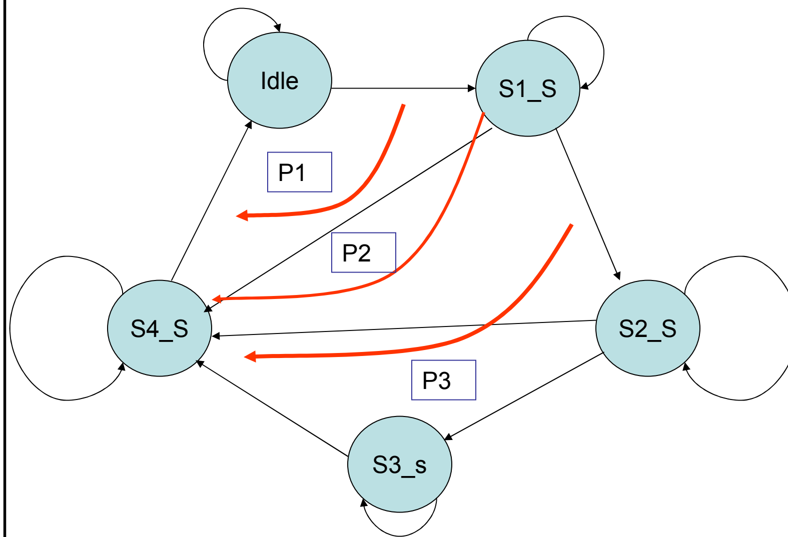
States: One-hot (inputs from RTL code)
States: IDLE_S, S1_S, S2_S, S3_S and S4_S

Legal two state transitions: 12 (inputs from Bubble diagram)
Legal two state transitions: (state == IDLE_S and prev_state == IDLE_S)
(state == IDLE_S and prev_state == S4_S)
(state == S1_S and prev_state == S1_S)
(state == S1_S and prev_state == IDLE_S)
(state == S2_S and prev_state == S2_S)
(state == S2_S and prev_state == S1_S)
(state == S3_S and prev_state == S3_S)
(state == S3_S and prev_state == S2_S)
(state == S4_S and prev_state == S4_S)
(state == S4_S and prev_state == S1_S)
(state == S4_S and prev_state == S2_S)
(state == S4_S and prev_state == S3_S)

Functional paths: 3 (inputs from Designer)
Functional paths: path1 - IDLE_S > S1_S > S4_S > IDLE_S
path2 – IDLE_S > S1_S > S2_S > S4_S > IDLE_S
path3 – IDLE_S > S1_S > S2_S > S3_S > S4_S > IDLE_S

Complex States & its active inputs: 3 (inputs from RTL code)
Complex States & its active inputs: IDLE_S; in1, in2, in3, in4
S1_S; in1, in2, in5, in9
S2_S; in1, in2, in6, in7

Bubble Diagram




```
------------------------------------------------------------------------
-- Module Name        : u_onehot_moore_fsm5_cov.e
-- Function           : Coverage definitions for u_onehot_moore_fsm
------------------------------------------------------------------------

<'

// Enumerated type definition for FSM states
type T_onehot_moore_fsm5 : [IDLE_S    = 5'b0_0001,
                            S1_S      = 5'b0_0010,
                            S2_S      = 5'b0_0100,
                            S3_S      = 5'b0_1000,
                            S4_S      = 5'b1_0000] (bits: 5);


// Instantiation of coverage unit in top, declare the coverage
// definition of each FSM in separate unit
extend u_e_top
{
  u_onehot_moore_fsm5_cover       : onehot_moore_fsm5_cov is instance;
  keep  u_onehot_moore_fsm5_cover.hdl_path() == "u_onehot_moore_fsm5";
};


// Coverage defining unit for u_onehot_moore_fsm5
unit onehot_moore_fsm5_cov
{

--------------------- Virtual Field Declaration -----------------------
// Functional paths
 !path1 : bool;
 !path2 : bool;
 !path3 : bool;
--------------------- Event Declaration -------------------------------

// Clock event
 event clk       is rise ('clk_i') @sim;

// State events
 event e_IDLE_S is
true('current_state'.as_a(T_onehot_moore_fsm5)==IDLE_S)@clk;
 event e_S1_S   is
true('current_state'.as_a(T_onehot_moore_fsm5)==S1_S)  @clk;
 event e_S2_S   is
true('current_state'.as_a(T_onehot_moore_fsm5)==S2_S)  @clk;
 event e_S3_S   is
true('current_state'.as_a(T_onehot_moore_fsm5)==S3_S)  @clk;
 event e_S4_S   is
true('current_state'.as_a(T_onehot_moore_fsm5)==S4_S)  @clk;

// Temporal expression for defining functional path with events
// More specific knowledge about the occurrence (number of clocks
// in which the FSM exists in a state) helps to write more
// specific expression.

 event e_path1 is {[..]*@e_IDLE_S;
                   @e_S1_S;    [..]*@e_S1_S;
                   @e_S4_S;    [..]*@e_S4_S;
                   @e_IDLE_S;} @clk;
```

```
event e_path2 is {[..]*@e_IDLE_S;
                  @e_S1_S;    [..]*@e_S1_S;
                  @e_S2_S;    [..]*@e_S2_S;
                  @e_S4_S;    [..]*@e_S4_S;
                  @e_IDLE_S} @clk;

event e_path3 is {[..]*@e_IDLE_S;
                  @e_S1_S;    [..]*@e_S1_S;
                  @e_S2_S;    [..]*@e_S2_S;
                  @e_S3_S;    [..]*@e_S3_S;
                  @e_S4_S;    [..]*@e_S4_S;
                  @e_IDLE_S} @clk;

// Sampling event for path coverage
event e_path  is {@e_path1 or @e_path2 or @e_path3} @clk;

-------------------- On Struct Member -------------------------------

// Setting/Resetting functional path flags
on e_path1
{
  path1 = TRUE;
  path2 = FALSE;
  path3 = FALSE;
  out (sys.time, " e_path1 occurred ");
};

on e_path2
{
  path1 = FALSE;
  path2 = TRUE;
  path3 = FALSE;
  out (sys.time, " e_path2 occurred ");
};

on e_path3
{
  path1 = FALSE;
  path2 = FALSE;
  path3 = TRUE;
  out (sys.time, " e_path3 occurred ");
};
```

```
-------------------- Coverage Groups ---------------------------------

cover clk  using text = "onehot_moore_fsm5 coverage" is {

   // State coverage
   item state :
T_onehot_moore_fsm5='current_state'.as_a(T_onehot_moore_fsm5);


   // 2 State Transition coverage
   transition state using text = "2 state transitions", illegal =
      not((state == IDLE_S  and  prev_state == IDLE_S)  or
          (state == IDLE_S  and  prev_state == S4_S)    or
          (state == S1_S    and  prev_state == S1_S)    or
          (state == S1_S    and  prev_state == IDLE_S)  or
          (state == S2_S    and  prev_state == S2_S)    or
          (state == S2_S    and  prev_state == S1_S)    or
          (state == S3_S    and  prev_state == S3_S)    or
          (state == S3_S    and  prev_state == S2_S)    or
          (state == S4_S    and  prev_state == S4_S)    or

          (state == S4_S    and  prev_state == S1_S)    or
          (state == S4_S    and  prev_state == S2_S)    or
          (state == S4_S    and  prev_state == S3_S));

  };
```

```
// Coverage for functional paths
cover e_path using text = "functional path coverage" is {

    item path1;
    item path2;
    item path3;
};

  // Expression coverage for complex conditions in
  // the next state determining combinational logic
  cover e_IDLE_S using text = "expression coverage at IDLE_S state" is
{
      item in1 : bit = 'in1';
      item in2 : bit = 'in2';
      item in3 : bit = 'in3';
      item in4 : bit = 'in4';

    cross in1, in2, in3, in4;
  };

  cover e_S1_S using text = "expression coverage at S1_S state" is {

      item in1 : bit = 'in1';
      item in2 : bit = 'in2';
      item in5 : bit = 'in5';
      item in9 : bit = 'in9';

    cross in1, in2, in5, in9;
  };

  cover e_S2_S using text = "expression coverage at S2_S state" is {

      item in1 : bit = 'in1';
      item in2 : bit = 'in2';
      item in6 : bit = 'in6';
      item in7 : bit = 'in7';

    cross in1, in2, in6, in7;
  };

}; -- onehot_moore_fsm5_cov
'>
```

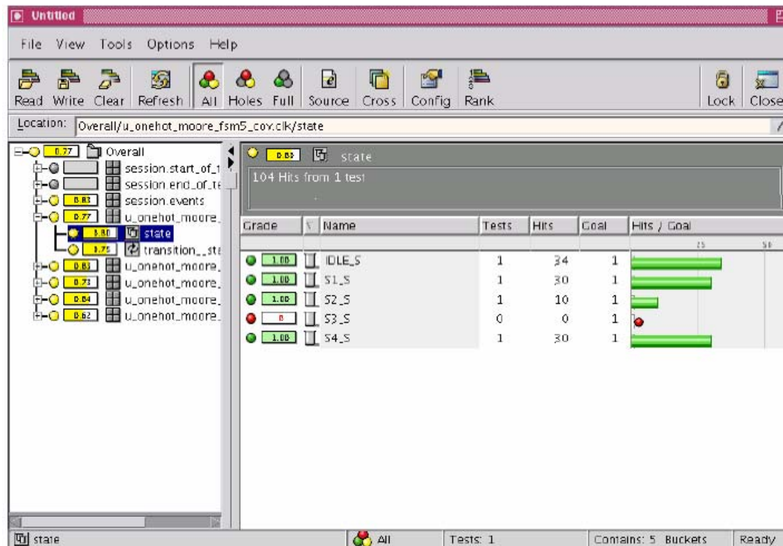


Fig 1: State Coverage

MISSING STATE: S3_S

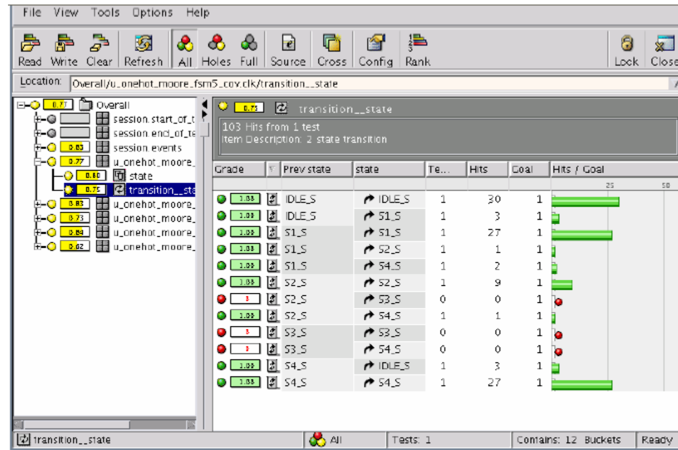


Fig 2: Two State Transition Coverage

Missing transitions are S2_S > S3_S,
S3_S > S3_S,
S3_S > S4_S.

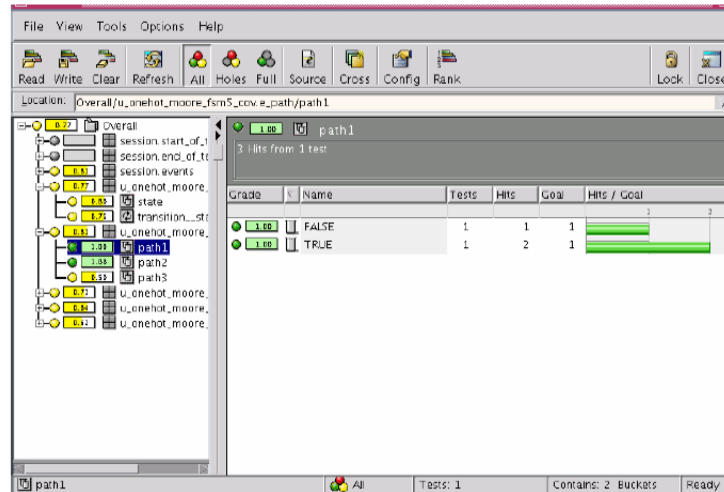


Fig 3: path1 Coverage

Total occurrence of functional paths: 1+ 2 = 3
Occurrence of functional path1: 2

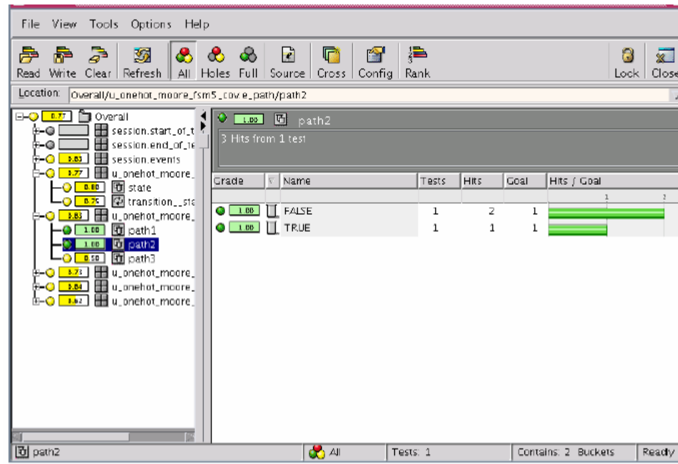


Fig 4: path2 Coverage

Total occurrence of functional paths: 2 +1 = 3
Occurrence of functional path2: 1

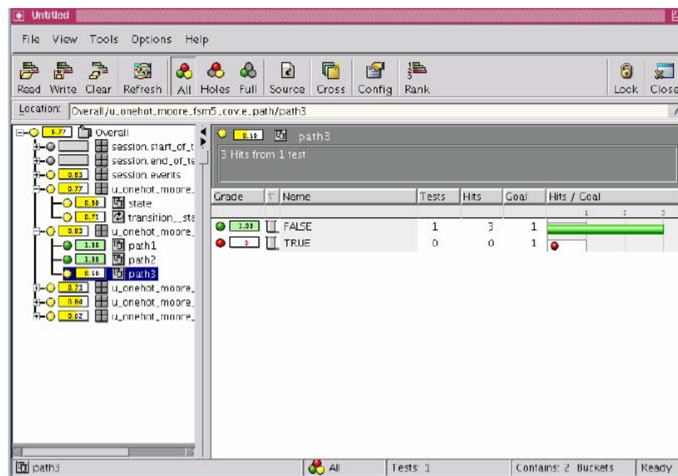


Fig 5: path3 Coverage

Total occurrence of functional paths: 3
Occurrence of functional path3: 0

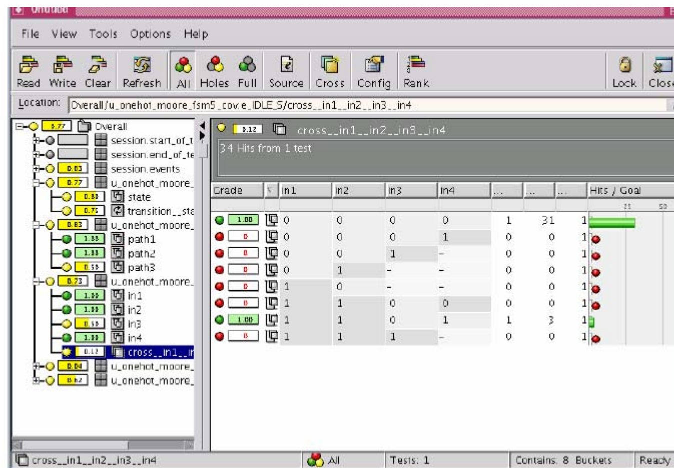


Fig 6: Cross Coverage for Inputs in IDLE_S State

Active inputs combinations that hit at IDLE_S (in1, in2, in3, in4): (0,0,0,0),
(1,1,0,1).

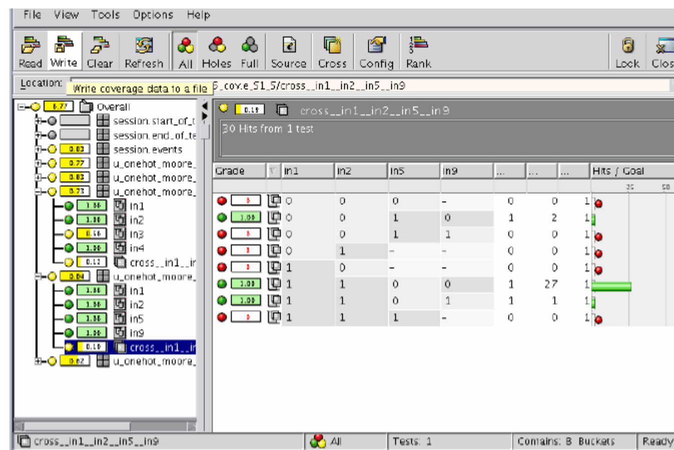


Fig 7: Cross Coverage for Inputs in S1_S State

Active inputs combinations that hit at S1_S (in1, in2, in5, in9): (0,0,1,0),
(1,1,0,0),
(1,1,0,1).

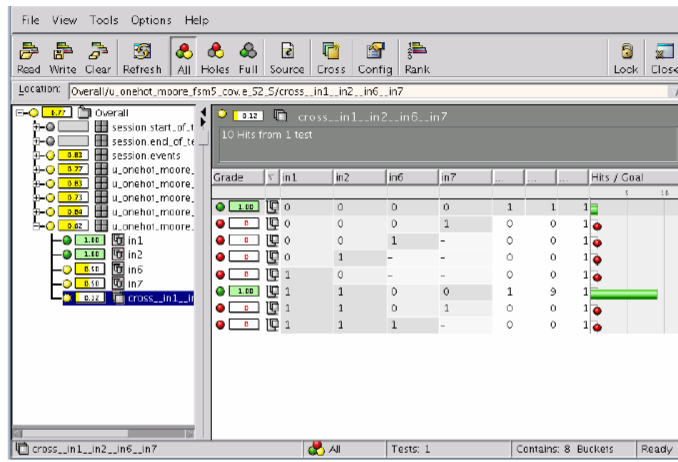


Fig 8: Cross Coverage for Inputs in S2_S State

Active inputs combinations that hit at S2_S (in1, in2, in6, in7): (0,0,0,0),
(1,1,0,0).