

Introduction to High Level Simulation using VCS

Ronald Valenzuela
Corporate Application Engineer

Legal Reminder

CONFIDENTIAL INFORMATION

- Information contained in this presentation reflects Synopsys plans as of the date of this presentation. Such plans are subject to completion and are subject to change. Products may be offered and purchased only pursuant to an authorized quote and purchase order. Synopsys is not obligated to develop the software with the features and functionality discussed in the materials.

SYNOPSYS CONFIDENTIAL

- Copyright ©2009 Synopsys Inc. All Rights Reserved. Forwarding or copying of this document, in any medium, in whole or in part, or disclosure of its contents, to other than the authorized recipient, is strictly prohibited.

Agenda

- High level Simulation
 - Introduction
 - Basic Verilog Simulation
 - SystemVerilog for Verification
 - A SystemVerilog Example
- Simulating with VCS
 - VCS Introduction
 - Debugging

Agenda

- High level Simulation
 - Introduction
 - Basic Verilog Simulation
 - SystemVerilog for Verification
 - A SystemVerilog Example
- Simulating with VCS
 - VCS Introduction
 - Debugging

Why High Level Simulation

Challenges:

- Design Size/Complexity with Many module and design interdependencies.

Solution Requirements

- Achieve the most reliable, smart, efficient and expeditious to deliver first-time-working silicon on time.
- Find and fix all bugs in design early before tapeout. Cost of fixing them increase exponentially with time as design evolve.

Proposed Solution:

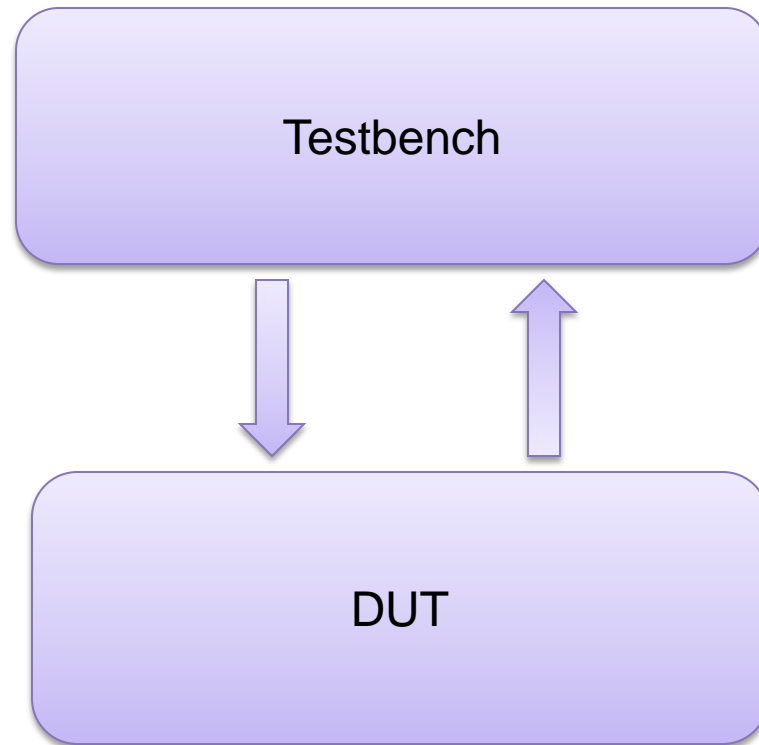
- SystemVerilog provides key technologies. Complex synchronization and timing mechanism, concurrent process allowing to simulate real and dynamic tests.
- SystemVerilog support OO methodology allowing development of reliable and reusable test environments.

Test environment

Testbench —

A complete verification environment applying *stimulus and checking the response of a design to implement one or more testcases*. A testcase can be verified using a *directed testbench or constrained-random testbench with functional coverage*.

*Verification Methodology Manual for SystemVerilog



Agenda

- High level Simulation
 - Introduction
 - Basic Verilog Simulation
 - SystemVerilog for Verification
 - A SystemVerilog Example
- Simulating with VCS
 - VCS Introduction
 - Debugging

The Simplest Verilog Testbench

```
// Timescale: #1=1ns, resolution 10ps
`timescale 1ns/10ps

module FSM_Testbench;
// Inputs
reg Clock;
reg Reset;

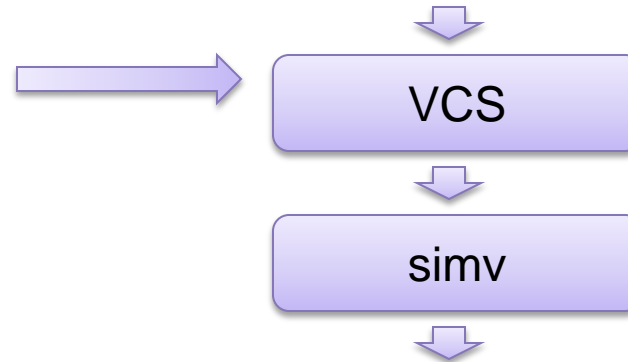
// Outputs
wire [1:0] state;
wire out;

// Device Under Test
FSM_module DUT(
    // Inputs
    .Clock(Clock),
    .Reset(Reset),
    // Outputs
    .state(state),
    .out(out)
);

// Clock element
always #5 Clock = ~Clock;

// Input stimulus
initial begin
    Clock = 0;
    // Reset
    #10 Reset = 1;
    #20 Reset = 0;
    #10
    // Monitor
    $dumpvars;
    $monitor ( "State: %d Output: %d!", state, out);
    #40;
    $finish;
end
endmodule
```

```
module FSM_module (
    input Clock,
    input Reset,
    output reg [1:0] state,
    output reg out
);
...
endmodule
```



```
Chronologic VCS simulator copyright 1991-2008
Contains Synopsys proprietary information.
Compiler version B-2008.12; Runtime version B-2008.12; Dec 20 05:08
2010
```

```
State: 1 Output: 0
State: 2 Output: 1
State: 0 Output: 0
State: 1 Output: 0
State: 2 Output: 1
$finish called from file "../rtl/simple_verilog.v", line 37.
$finish at simulation time      8000
      VCS Simulation Report
Time: 80000 ps
CPU Time: 0.020 seconds;   Data structure size: 0.0Mb
Mon Dec 20 05:08:19 2010
```


The Simplest Verilog Testbench

```
// Timescale: #1=1ns, resolution 10ps
`timescale 1ns/10ps
```

```
module FSM_Testbench;
// Inputs
reg Clock;
reg Reset;
```

```
// Outputs
wire [1:0] state;
wire out;
```

```
// Device Under Test
FSM_module DUT(
```

```
// Inputs
.Clock(Clock),
.Reset(Reset),
// Outputs
.state(state),
.out(out)
```

```
);
```

```
// Clock element
always #5 Clock = ~Clock;
```

```
// Input stimulus
initial begin
```

```
Clock = 0;
// Reset
#10 Reset = 1;
#20 Reset = 0;
#10
// Monitor
$dumpvars;
$monitor ("State: %d Output: %d!", state, out);
#40;
$finish;
```

Aux vars. to interface stimulus

DUT Instantiation

Clocking device

*-VCD File Out
-Display Changes*

```
endmodule
```

```
module FSM_module (
input Clock,
input Reset,
output reg [1:0] state,
output reg out
);
...
endmodule
```

VCS

simv

Simulation Executable

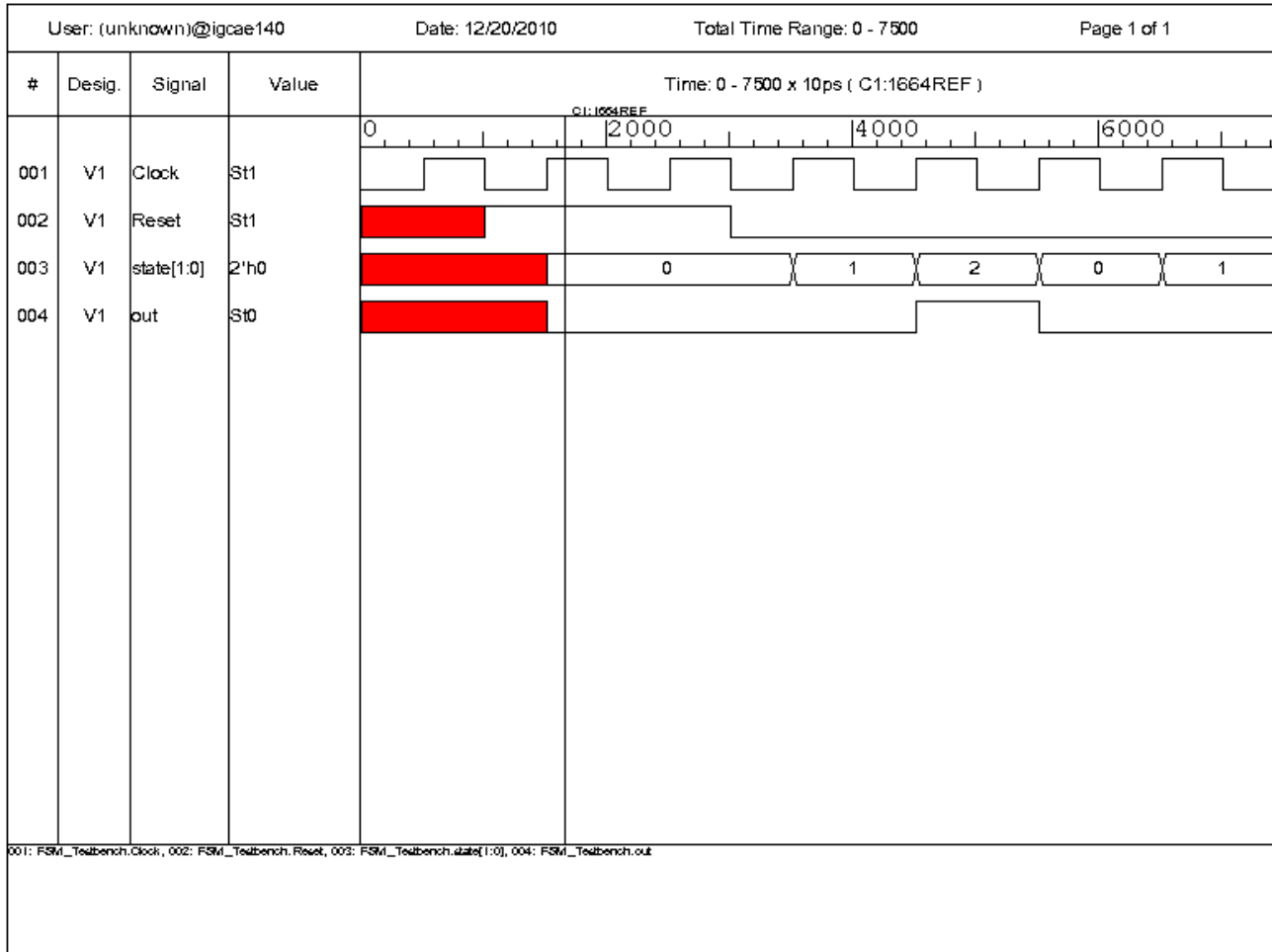
```
Chronologic VCS simulator copyright 1991-2008
Contains Synopsys proprietary information.
Compiler version B-2008.12; Runtime version B-2008.12: Dec 20 05:08 2010
```

```
State: 1 Output: 0
State: 2 Output: 1
State: 0 Output: 0
State: 1 Output: 0
State: 2 Output: 1
$finish called from file ".../rtl/sim_ple
$finish at simulation time
VCS Simulation Report
```

```
Time: 80000 ps
CPU Time: 0.020 seconds; Data structure size: 0.0Mb
Mon Dec 20 05:08:19 2010
```

Monitor Output

Value Change Dump



*File:
verilog.dump
Can be loaded into
Discovery
Visualization
Environment (DVE).*

Other Features

- Tasks:

```
task reset_test;  
  $display("Task reset_test: asserting and checking reset\n");  
  reset_p <= 1;  
  #40 reset_p <= 0;  
  if( fsm.state != 0)  
    $strobe( "\n***Time: %04d State: %02d. Failed\n", $time, fsm.state );  
endtask
```

- Functions

```
function even_parity;  
  input [7:0] data;  
  integer i;  
  for (i=0; i<8; i= i+1) begin  
    even_parity = even_parity ^ data[i];  
  end  
endfunction
```

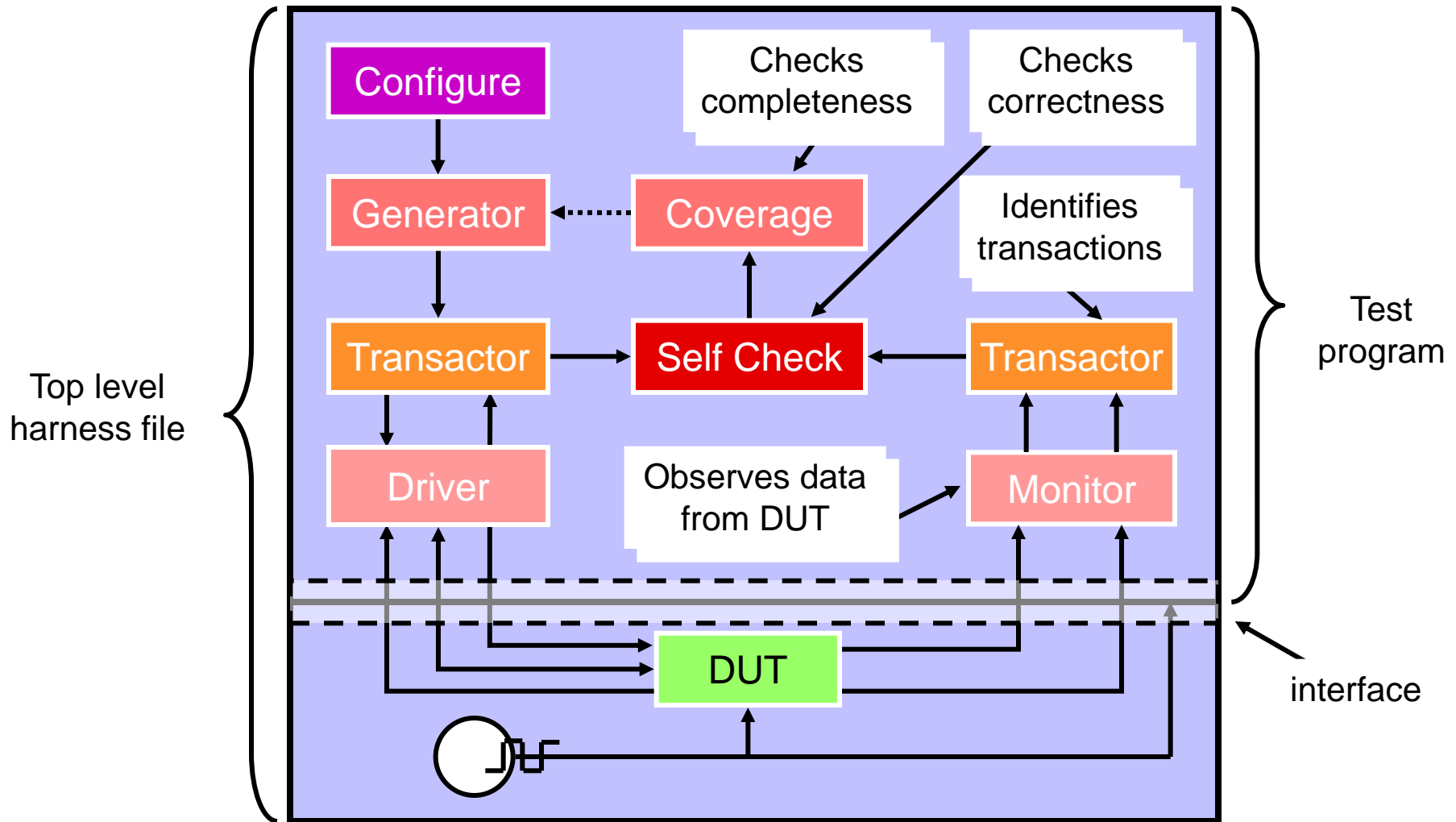
Agenda

- High level Simulation
 - Introduction
 - Basic Verilog Simulation
 - **SystemVerilog for Verification**
 - A SystemVerilog Example
- Simulating with VCS
 - VCS Introduction
 - Debugging

SystemVerilog for Verification

- Clocking Blocks
- Classes Verification Expect Checks
- Methods, Properties (OOM) Concurrency and Control
- Functional Coverage

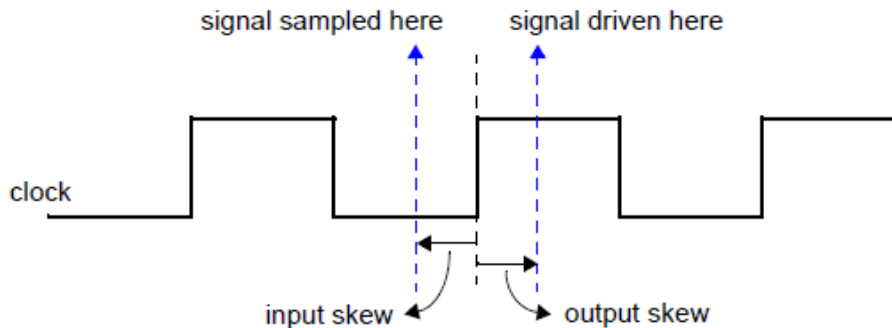
The SystemVerilog Test Environment



Clocking Block

“The clocking block construct identifies clock signals and captures the timing and synchronization requirements of the blocks being modeled”

```
clocking bus @(posedge clock1);  
  default input #10ns output #2ns;           // Clocking Skew  
  input data, ready, enable = top.mem1.enable;  
  output negedge ack;                       // Defaults Overriden  
  input #1step addr;                       // Defaults Overriden  
endclocking
```



* Source IEEE Std. 18000

Interface Block

```
interface #(WIDTH=8) simple_bus; // Definition
    logic req, gnt;
    logic [WIDTH-1:0] addr, data;
    logic [1:0] mode;
    logic ready, data;
endinterface
```

```
module memMod(simple_bus a, input logic clk);
// Access the simple_bus interface
```

```
module cpuMod(interface a, input logic clk);
// Access the simple_bus interface
```

```
module top; // Interface Instantiation
    logic clock;
    simple_bus sb_intf;
    memMod mem1 (.a(sb_intf), .clk(clock) );
    cpuMod cpu1 (.a(sb_intf), .clk(clock) );
endinterface
```

- Variables & Nets are *inout* and *ref* by default.
- An interface can have parameters, constants, variables, functions, and tasks.
- The types of elements in an interface can be declared, or the types can be passed in as parameters.

Clocking & Interfaces directions

```
interface bus_A (input clk);  
  logic [15:0] data;  
  logic write;  
  modport test (input data, output write);  
  modport dut (output data, input write);  
endinterface
```

```
program test( bus_A.test a, bus_B.test b );  
  
  clocking cd1 @(posedge a.clk);  
    input data = a.data;  
    output write = a.write;  
    inout state = top.cpu.state;  
  endclocking
```

- **modport** allows to declare direction explicitly.
- Clock block sees direction as seeing from outside the block.

Program Block

```
program fifo_test (  
    output logic rst_n;  
    input clk;  
    output logic [15:0] data_in;  
    input [15:0] data_out;  
    output logic push_req_n, pop_req_n, diag_n;  
    input empty, full, almost_empty,  
           almost_full, half_full, error;  
    // declarations  
    // instantiations  
    // tasks...  
    // initial block  
endprogram
```

- *Simplify the creation and maintenance of testbenches.*
- *Can be instantiated and individually connected.*

**Since the program block drives data to the DUT input and reads data from the DUT output, the port directions in the programblock are opposite to those of the DUT.*

Constrained Random Stimulus

```
////////////////////////////////////  
/// Definition: Random Write Data Class  
////////////////////////////////////
```

```
class random_write_data;  
  rand logic [`WIDTH-1:0] data [`DEPTH];  
  rand logic [15:0] data_rate;  
  constraint reasonable {  
    data > 0;  
    data_rate dist { [1:8] := 10 , [128:135] := 10,  
                    [512:519] :=1 };  
  }  
endclass
```

```
int result;  
random_write_data r = new;  
result = r.randomize();
```

- *rand keyword identifies signals as randomizable data field*
- *dist construct specify ranges and weights*
- *Can be instantiated in a program block*

Verification Expect Checks

- Assertions:
 - Immediate assertions
 - Follows simulation semantics, same as procedural block. False if evaluates to X,Z or 0.
 - Concurrent assertions
 - Based on clock semantics, use sampled values of variables. Expression is tied to clock definition.

** While an immediate assertion describes a logic behavior at an instant of time, a concurrent assertion detects a behavior over a period of time.*

Immediate assertions

```
label : assert (expression) statement1; else statement2;
```

```
task req_test;  
  time t;  
  ...  
  if (fsm.state == REQ)  
    assert (req1 || req2) ;  
    // Do Nothing  
  else begin  
    t = $time;  
    #5 $error("assert failed at time %0t", t)  
  end  
endtask
```

- *Action blocks or statement can be null*
- *Failure block can have one of the following: \$fatal, \$error, \$warning, \$info.*

Concurrent assertions

```
label : assert property ( property statement) statement1; else statement2;
```

```
module fsm_module ()  
...  
always @ (posedge clk) begin  
...  
  if (state == REQ)  
    assert property ( @(posedge clk)  
                      Req |-> ##[1:2] Ack );  
    // Do Nothing  
  else begin  
    t = $time;  
    #5 $error("assert failed at time %0t", t)  
  end  
end  
endmodule
```

- *If variable used is clocking block input then is sampled by it (the actual value is produced by the block).*
- *Characterized by property keyword, specify design behavior*
 - *properties can be defined as sequences*

Expect statement

```
label : expect property ( property statement) statement1; else statement2;
```

```
task mytask;  
...  
if (expr1) begin  
    expect ( @(posedge clk) a ##1 b ##1 c )  
        pass_block();  
    else  
        fail_block();  
end  
endtask
```

- *Similar syntax to concurrent assertion but must occur within procedural block.*
- *It blocks until evaluation.*

Classes, Properties & Methods

- A class is a type that includes data and subroutines (functions and tasks) that operate on those data.

```
class Packet;
  bit [7:0] command;
  bit [40:0] address;
  bit [4:0] packet_id;
  integer status;
  function new();
    command = IDLE;
    address = 41'b0;
    packet_id = 5'bx;
  endfunction
...
  function integer current_status();
    current_status = status;
  endfunction
endclass
```


Classes, Properties & Methods

```
virtual class device;
  task driveRamData(input logic [7:0] data);
    vintf.CBcntrlr.ramData <= data;
  endtask
  function logic [7:0] getBusData();
    return vintf.CBcntrlr.busData;
  endfunction
  function logic [5:0] getRamAddr();
    return vintf.CBcntrlr.ramAddr;
  endfunction
  function logic [7:0] getRamData();
    return vintf.CBcntrlr.ramData;
  endfunction
  function logic getRdWr_N();
    return vintf.CBcntrlr.rdWr_N;
  endfunction
  extern virtual function logic getCe_N();
  extern virtual task waitCe_N();
endclass
```

- We can define virtual methods that will be defined in derived classes

```
class device0 extends device;
  function logic getCe_N();
    return vintf.CBcntrlr.ce0_N;
  endfunction
  task waitCe_N();
    @vintf.CBcntrlr.ce0_N;
  endtask
endclass
```

Concurrent Processes

```
fork
//fork CPU 0
  repeat(256) begin
    cpu0.randomize();
    cpu0.request_bus();
    cpu0.writeOp();
    cpu0.release_bus();
    cpu0.request_bus();
    cpu0.readOp();
    cpu0.release_bus();
    cpu0.delay_cycle();
  end
//fork CPU 1
  repeat(256) begin
    cpu1.randomize();
    cpu1.request_bus();
    cpu1.writeOp();
    cpu1.release_bus();
    cpu1.request_bus();
    cpu1.readOp();
    cpu1.release_bus();
    cpu1.delay_cycle();
  end
join
```

- Fork/join blocks are the primary mechanism for creating concurrent processes.

```
fork
  statement1;
  statement2;
  ...
  statementN;
join
```

Functional Coverage

- Is a user-defined metric that measures how much of the design specification has been exercised.
 - It is user-specified and is not automatically inferred from the design.
 - It is based on the design specification, independent of the actual design.

The SystemVerilog functional coverage constructs

- Coverage of variables and expressions, as well as cross coverage between them.
- Automatic as well as user-defined coverage bins
- Associate bins with sets of values, transitions, or cross products
- Filtering conditions at multiple levels
- Events and sequences to automatically trigger coverage sampling
- Procedural activation and query of coverage
- Optional directives to control and regulate coverage

covergroups & coverpoints

```
covergroup range @(negeedge, memsys_test_top.dut.Umem.adxStrb);  
  a: coverpoint memsys_test_top.dut.Umem.busAddr {  
    bins m_state[] = {[0:255]};  
  }  
endgroup
```

```
enum logic [1:0] {IDLE, START, WRITE0, WRITE1} st;  
covergroup cntlr_cov @vintf.CBmemsys;  
  b: coverpoint memsys_test_top.dut.Umem.state {  
    bins t0 = (IDLE => IDLE);  
    bins t1 = (IDLE => START);  
    bins t2 = (START => IDLE);  
    bins t3 = (START => WRITE0);  
    bins t4 = (WRITE0 => WRITE1);  
    bins t5 = (WRITE1 => IDLE);  
    bins bad_trans = default sequence;  
  }  
endgroup
```

- For example we can check that an entire address space is tested.
- Or we can also check valid transitions for certain state machine.

HTML Report

Testbench Group List

[dashboard](#) | [hierarchy](#) | [modlist](#) | **[groups](#)** | [tests](#) | [asserts](#)

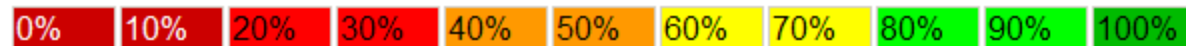
Total Groups Coverage Summary

SCORE	WEIGHT
100.00	1

Total groups in report: 2

SCORE	WEIGHT	GOAL	NAME
100.00	1	100	memsys_test_top.testbench::range
100.00	1	100	memsys_test_top.testbench::cntrl_cov

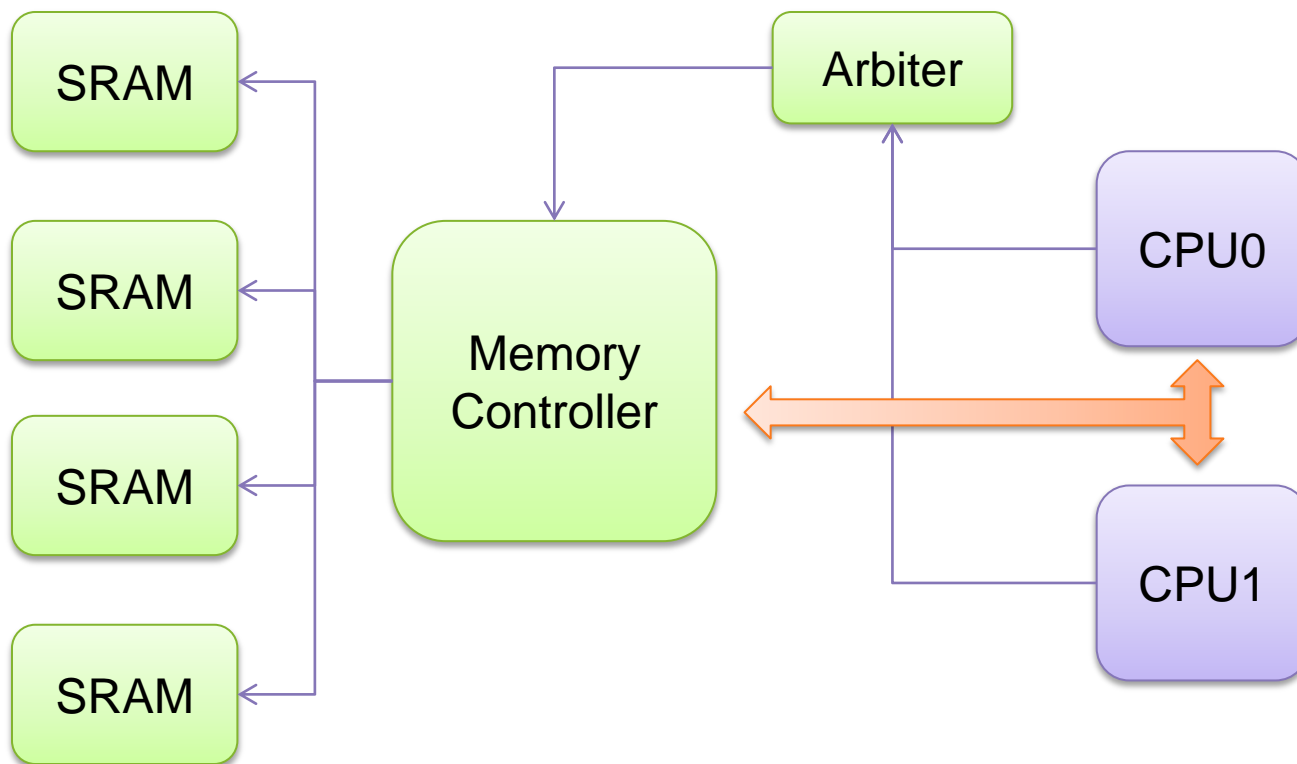
[dashboard](#) | [hierarchy](#) | [modlist](#) | **[groups](#)** | [tests](#) | [asserts](#)



Agenda

- High level Simulation
 - Introduction
 - Basic Verilog Simulation
 - SystemVerilog for Verification
 - A SystemVerilog Example
- Simulating with VCS
 - VCS Introduction
 - Debugging

Example: Typical system scenario



* Example taken from VCS[®]/VCSi[™] Testbench Tutorial Suite

Arbiter

- Implements round-robin arbitration algorithm between two CPUs.
- CPUs drive a request through request[n].
- Arbiter queues the request and determine which CPU gains access (grant[n]).

Reset



CPU0 Places a Request

Arbiter grant access to CPU0

Testing: Defining a program

```
program arb_test( input clk, input [1:0] grant_p, output logic [1:0] request_p, output logic reset_p);
```

```
task reset_test;
```

```
    $write("Task reset_test: asserting and checking reset\n" );
```

```
    reset_p <= 1;
```

```
    repeat (2) @(posedge clk);
```

```
    reset_p <= 0;
```

```
    request_p <= 2'b00;
```

```
    reset_check: expect(@(negedge clk) grant_p == 2'b00) $display($time, "Passed")  
                    else $display($time, " Failed");
```

```
endtask
```

```
task drive_test1;
```

```
    $write("Task drive_test1: driving request and checking grant for CPU0\n");
```

```
    @(posedge clk) request_p <= 2'b01;
```

```
    @(posedge clk);
```

```
    grant_h: expect(@(negedge clk) grant_p == 2'b01) $display($time, "Passed")  
            else $display($time, " Failed");
```

```
    @(posedge clk) request_p <= 2'b00;
```

```
    @(posedge clk);
```

```
    grant_l: expect(@(negedge clk) grant_p == 2'b00) $display($time, "Passed")  
            else $display($time, " Failed");
```

```
endtask
```

```
...
```

Testing: Defining a program (contd.)

```
...
task drive_test2;
  $write("Task drive_test2: driving request and checking grant for CPU1\n");
  @(posedge clk) request_p <= 2'b10;
  @(posedge clk);
  grant_h: expect(@(negedge clk) grant_p == 2'b10) $display($time, "Passed")
              else $display($time, " Failed");

  @(posedge clk) request_p <= 2'b00;
  @(posedge clk);
  grant_l: expect(@(negedge clk) grant_p == 2'b00) $display($time, "Passed")
              else $display($time, " Failed");
endtask
```

```
...
```

Testing: Defining a program (contd.)

```
task drive_test3;
  $write("Task drive_test3: driving request and checking grant for both CPU0 and CPU1 \n");
  @(posedge clk) request_p <= 2'b11;
  @(posedge clk);
  expect(@(negedge clk) grant_p == 2'b01) $display($time, "Passed")
                                     else $display($time, " Failed");

  @(posedge clk) request_p <= 2'b10;
  @(posedge clk);
  expect(@(negedge clk) ##[0:2] grant_p == 2'b10) $display($time, "Passed")
                                     else $display($time, " Failed");

  @(posedge clk) request_p <= 2'b00;
  @(posedge clk);
  expect(@(negedge clk) grant_p == 2'b00) $display($time, "Passed")
                                     else $display($time, " Failed");
endtask

...
initial begin
  reset_test();
  drive_test1();
  drive_test2();
  drive_test3();
  $finish;
end
endprogram
```

Testing: Top level module (contd.)

```
module arb_test_top;
  parameter clock_cycle = 100 ;
  reg clk ;
  wire reset ;
  wire [1:0] request ;
  wire [1:0] grant ;
  arb dut(
    .clk ( clk ),
    .reset ( reset ),
    .request ( request ),
    .grant ( grant )
  );
  arb_test testbench(
    .clk( clk ),
    .reset_p( reset ),
    .request_p( request ),
    .grant_p( grant )
  );
  initial begin
    clk = 1'b0;
    forever begin
      #(clock_cycle/2) clk = ~clk ;
    end
  end
endmodule
```

- Top level module instantiate program and dut.
- Stand alone regs and wire approach for sampling
 - Reg based clocking element doesn't relate any design elements.

Verification Results

Chronologic VCS simulator copyright 1991-2008

Contains Synopsys proprietary information.

Compiler version B-2008.12; Runtime version B-2008.12; Dec 21 09:52 2010

Task reset_test: asserting and checking reset

200 Passed

Task drive_test1: driving request and checking grant for CPU0

400 Passed

600 Passed

Task drive_test2: driving request and checking grant for CPU1

800 Passed

1000 Passed

Task drive_test3: driving request and checking grant for both CPU0 and CPU1

1200 Passed

1500 Passed

1700 Passed

\$finish called from file "arb_test.v", line 61.

\$finish at simulation time 1700

V C S S i m u l a t i o n R e p o r t

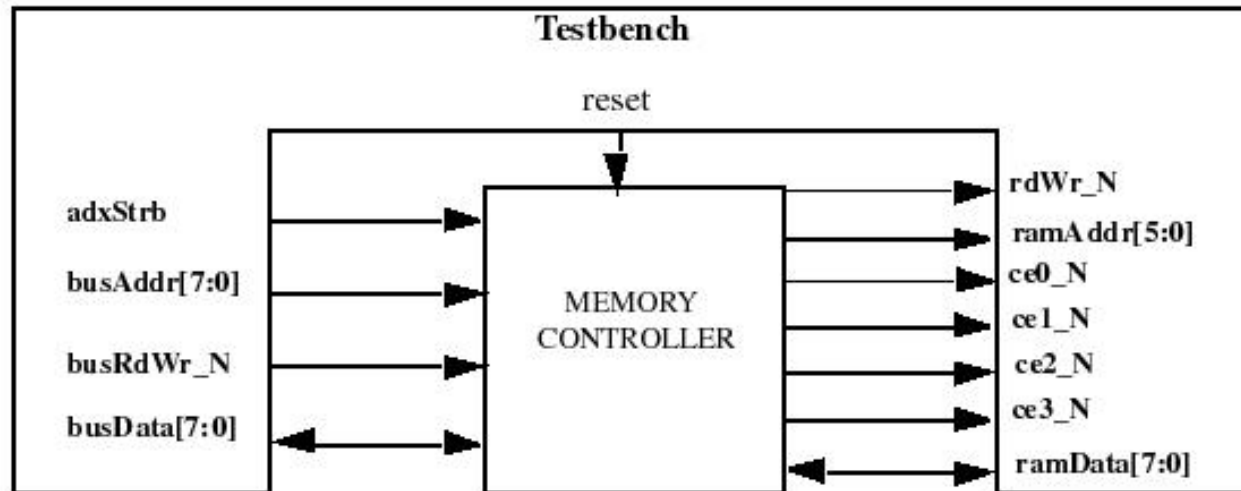
Time: 1700

CPU Time: 0.010 seconds; Data structure size: 0.0Mb

Tue Dec 21 09:52:11 2010

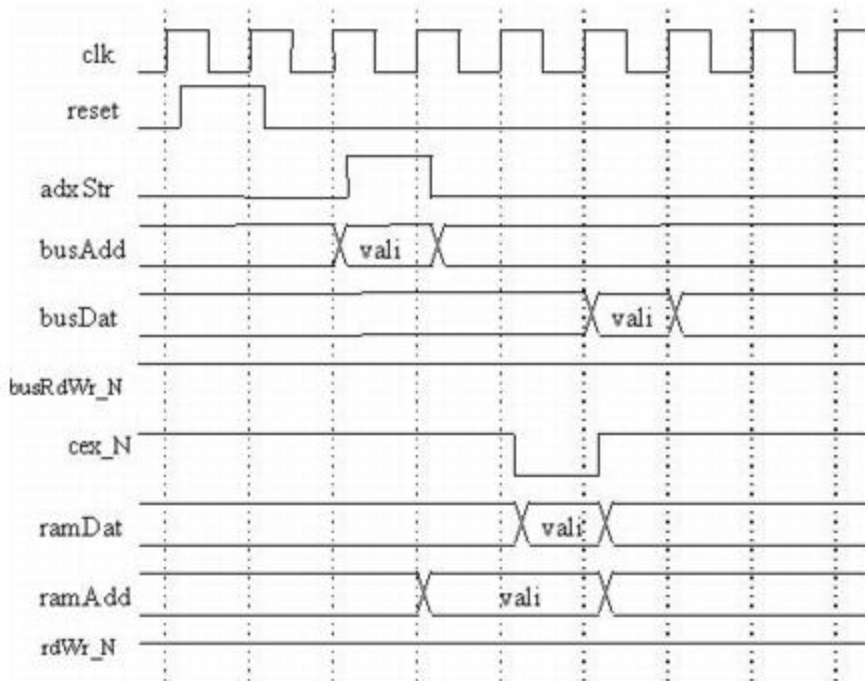
Memory Controller block

- Reads requests from the system bus and generates control signals for the SRAM devices
- Reads data and transfers it back to the bus

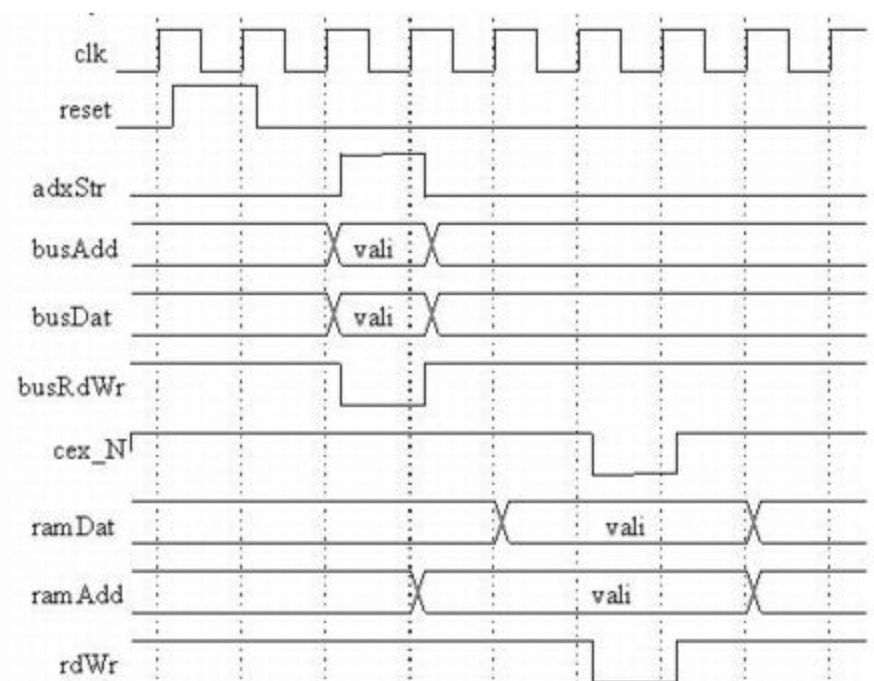


Timing Diagrams

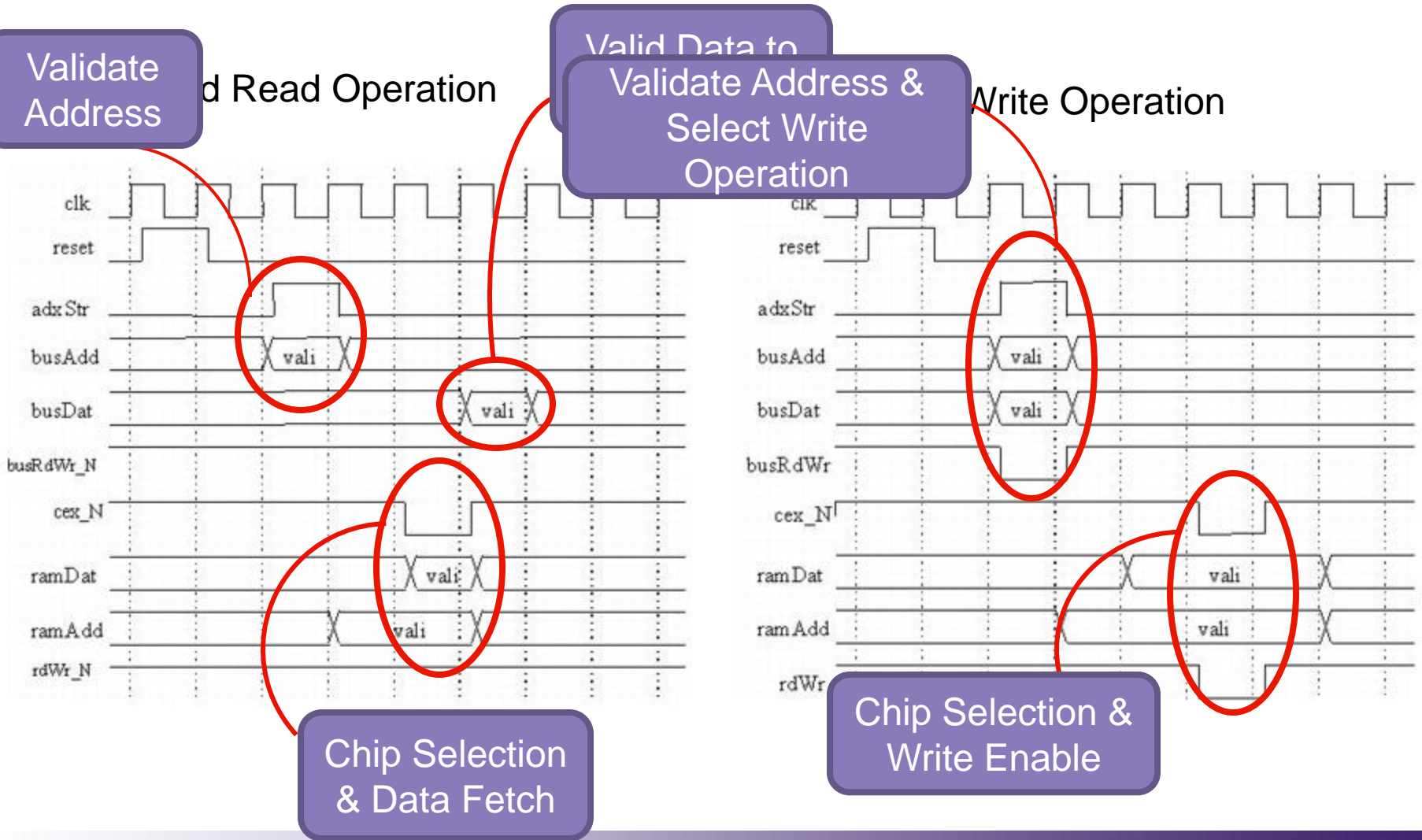
Valid Read Operation



Valid Write Operation

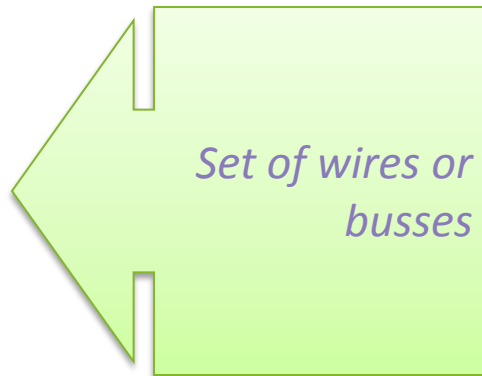


Timing Diagrams



Testing: the Interface

```
interface cntrlr_intf(input clk);  
  wire reset ;  
  wire [7:0] busAddr ;  
  wire [7:0] busData ;  
  wire busRdWr_N ;  
  wire adxStrb ;  
  wire rdWr_N ;  
  wire ce0_N ;  
  wire ce1_N ;  
  wire ce2_N ;  
  wire ce3_N ;  
  wire [5:0] ramAddr ;  
  wire [7:0] ramData ;  
  clocking CBcntrlr @(posedge clk);  
    output reset,busAddr,busRdWr_N,adxStrb;  
    input rdWr_N,ce0_N,ce1_N,ce2_N,ce3_N,ramData;  
    inout busData,ramData;  
  endclocking  
endinterface
```



- We create an interface to manage the interconnection of our devices.
- We define the sampling nature of our signals by creating clocking block

Testing: the Top level module

```
module cntrlr_test_top;
  parameter clock_cycle = 100 ;
  bit clk ;
  cntrlr_intf intf(clk);
  cntrlr_test test_program(intf);
  cntrlr dut(
    .clk ( clk ),
    .reset ( intf.reset ),
    .busAddr ( intf.busAddr ),
    .busData ( intf.busData ),
    .busRdWr_N ( intf.busRdWr_N ),
    .adxStrb ( intf.adxStrb ),
    .rdWr_N ( intf.rdWr_N ),
    .ce0_N ( intf.ce0_N ),
    ...
  );
  initial begin
    clk = 1'b0;
    $monitor("copi %d : %d", intf.ramAddr,$time);
    forever begin
      #(clock_cycle/2) clk = ~clk;
    end
  end
endmodule
```

- We use the interface to create connections between the program and DUT.
- Clock inputs the clockblocking module.

Testing: Simulating SRAMs

```
virtual class device;
  task driveRamData(input logic [7:0] data);
    vintf.CBcntrlr.ramData <= data;
  endtask
  function logic [7:0] getBusData();
    return vintf.CBcntrlr.busData;
  endfunction
  function logic [5:0] getRamAddr();
    return vintf.CBcntrlr.ramAddr;
  endfunction
  function logic [7:0] getRamData();
    return vintf.CBcntrlr.ramData;
  endfunction
  function logic getRdWr_N();
    return vintf.CBcntrlr.rdWr_N;
  endfunction
  extern virtual function logic getCe_N();
  extern virtual task waitCe_N();
endclass
```

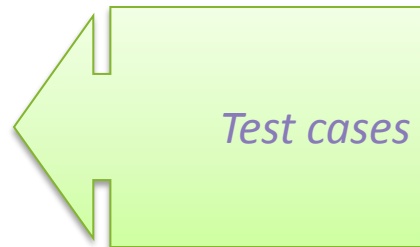
```
class device0 extends device;
  function logic getCe_N();
    return vintf.CBcntrlr.ce0_N;
  endfunction
  task waitCe_N();
    @vintf.CBcntrlr.ce0_N;
  endtask
endclass
```

```
class device1 extends device;
  function logic getCe_N();
    return vintf.CBcntrlr.ce1_N;
  endfunction
  task waitCe_N();
    @vintf.CBcntrlr.ce1_N;
  endtask
endclass
```

Testing: Creating a Program

```
program cntrlr_test (cntrlr_intf intf);  
  `include "device.v"  
  device0 d0 = new;  
  device1 d1 = new;  
  device2 d2 = new;  
  device3 d3 = new;  
  virtual cntrlr_intf vintf;  
  initial begin  
    vintf = intf;  
    @vintf.CBcntrlr;  
    resetSequence();  
    resetCheck();  
    checkSimpleReadWrite();  
    checkAllAddresses();  
    $finish;  
  end  
end
```

...



Testing: Reset Sequence & Test

//reset sequence

```
task resetSequence ();  
    $write("Task resetSequence entered\n");  
    vintf.CBcntrlr.reset <= 1'b1;  
    vintf.CBcntrlr.ramData <= 8'bzzzzzzzz;  
    repeat (2) @vintf.CBcntrlr;  
    vintf.CBcntrlr.reset <= 1'b0;  
endtask
```

//Check state of controller after reset

```
task resetCheck ();  
    $write("Task resetCheck entered to check reset values\n");  
    //all chip enables must be deasserted  
    expect(@(vintf.CBcntrlr) ##[0:10] vintf.CBcntrlr.ce0_N === 1'b1);  
    assert (vintf.CBcntrlr.ce1_N == 1'b1);  
    assert (vintf.CBcntrlr.ce2_N == 1'b1);  
    assert (vintf.CBcntrlr.ce3_N == 1'b1);  
endtask
```

Testing: Read & Write Operations

// low level task to drive a read onto the bus

```
task readOp (bit [7:0] adx);  
    $write("Task readOp : address %0h\n", adx);  
    vintf.CBcntrlr.busAddr <= adx;  
    vintf.CBcntrlr.busRdWr_N <= 1'b1;  
    vintf.CBcntrlr.adxStrb <= 1'b1;  
    @vintf.CBcntrlr vintf.CBcntrlr.adxStrb <= 1'b0;  
endtask
```

//low level task to drive a write onto the bus

```
task writeOp (bit [7:0] adx, bit [7:0] data);  
    $write("Task writeOp : address %0h data %0h\n", adx, data);  
    @vintf.CBcntrlr  
    vintf.CBcntrlr.busAddr <= adx;  
    vintf.CBcntrlr.busData <= data;  
    vintf.CBcntrlr.adxStrb <= 1'b1;  
    vintf.CBcntrlr.busRdWr_N <= 1'b0;  
    @vintf.CBcntrlr vintf.CBcntrlr.adxStrb <= 1'b0;  
    vintf.CBcntrlr.busRdWr_N <= 1'b1;  
    vintf.CBcntrlr.busData <= 8'bzzzzzzzz;  
endtask
```

Testing: More Checkers

```
//Checker to Verify sram write on a particular
// device meets the timing
task checkSramWrite ( device device_id,
                    bit [5:0] adx, bit [7:0] data);
    expect (@(vintf.CBcntrlr) ##[1:5]
           device_id.getRamAddr() == adx);
    expect (@(vintf.CBcntrlr) ##[0:2]
           device_id.getRamData() == data);
    assert(device_id.getRdWr_N() == 1'b0);
    assert(device_id.getCe_N() == 1'b0);
    assert(device_id.getRamData() == data);
    assert(device_id.getRamAddr() == adx);
    @vintf.CBcntrlr
    assert(device_id.getRdWr_N() == 1'b1);
    $write("Task checkSramWrite:
          Address %0h data %0h\n",
          vintf.CBcntrlr.ramAddr, vintf.CBcntrlr.ramData);
    assert(device_id.getCe_N() == 1'b1);
    assert(device_id.getRamData() == data);
    assert(device_id.getRamAddr() == adx);
endtask
```

```
task checkAllAddresses ();
    device dev;
    bit [7:0] index;
    bit [7:0] data;
    $write("Task checkAllAddresses entered\n");
    for (int i = 0; i < 256; i++) begin
        $write("Expect6: Index %0d time %0d\n", i, $time);
        index = i;
        data = 8'h5a;
        writeOp (index, data);
        case (index[7:6])
            2'b00: dev = d0;
            2'b01: dev = d1;
            2'b10: dev = d2;
            2'b11: dev = d3;
        endcase
        checkSramWrite (dev, index[5:0], data);
        readOp(index);
        checkSramRead (dev, index[5:0], data);
    end
endtask
```

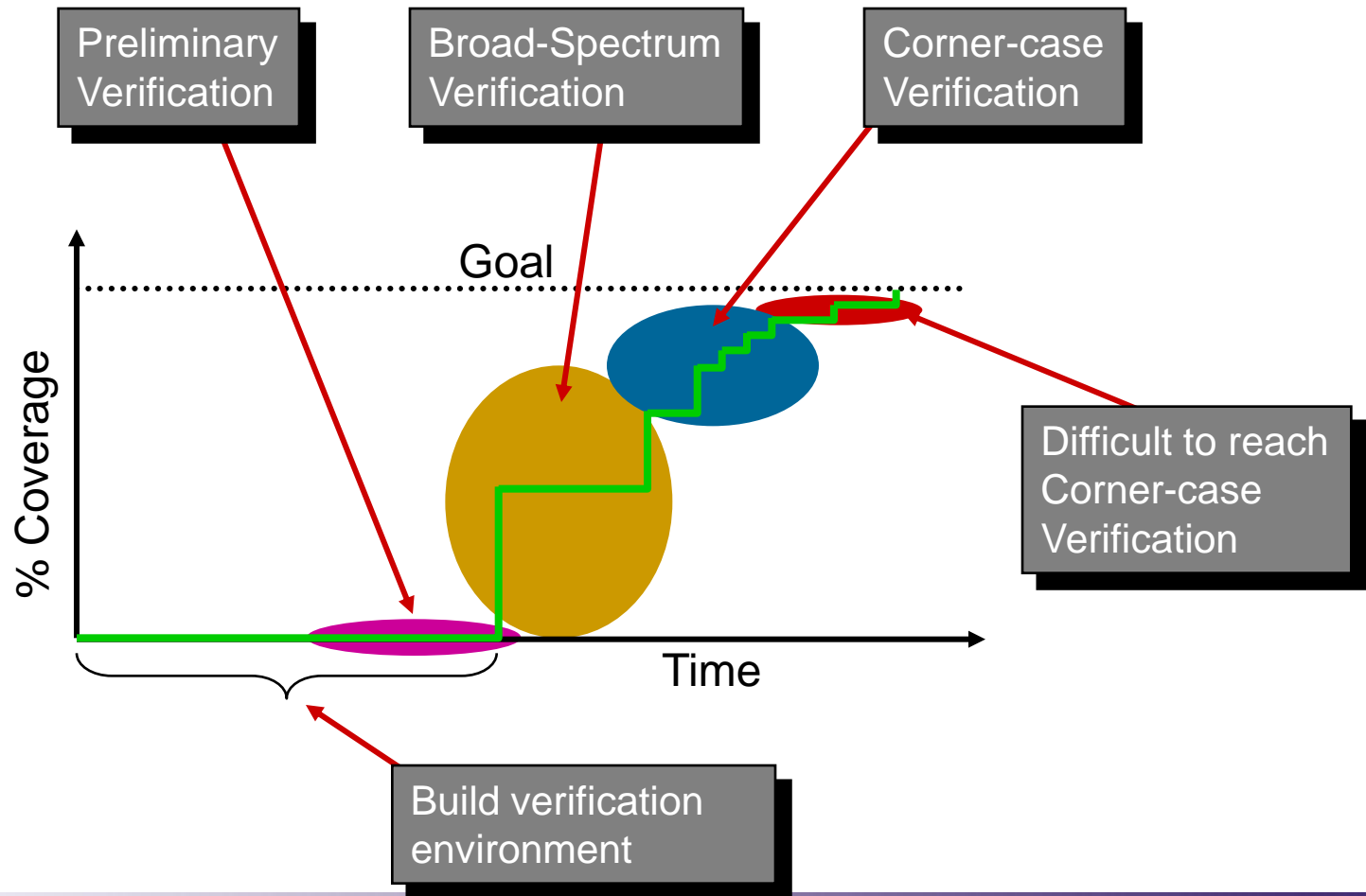

How to get full example

- The files for this tutorial can be accessed in the following directory:
`$VCS_HOME/doc/examples/testbench/sv/tutorial.`
- Don't forget to check out full system testbench for more on threading and coverage tests.

Bonus Track

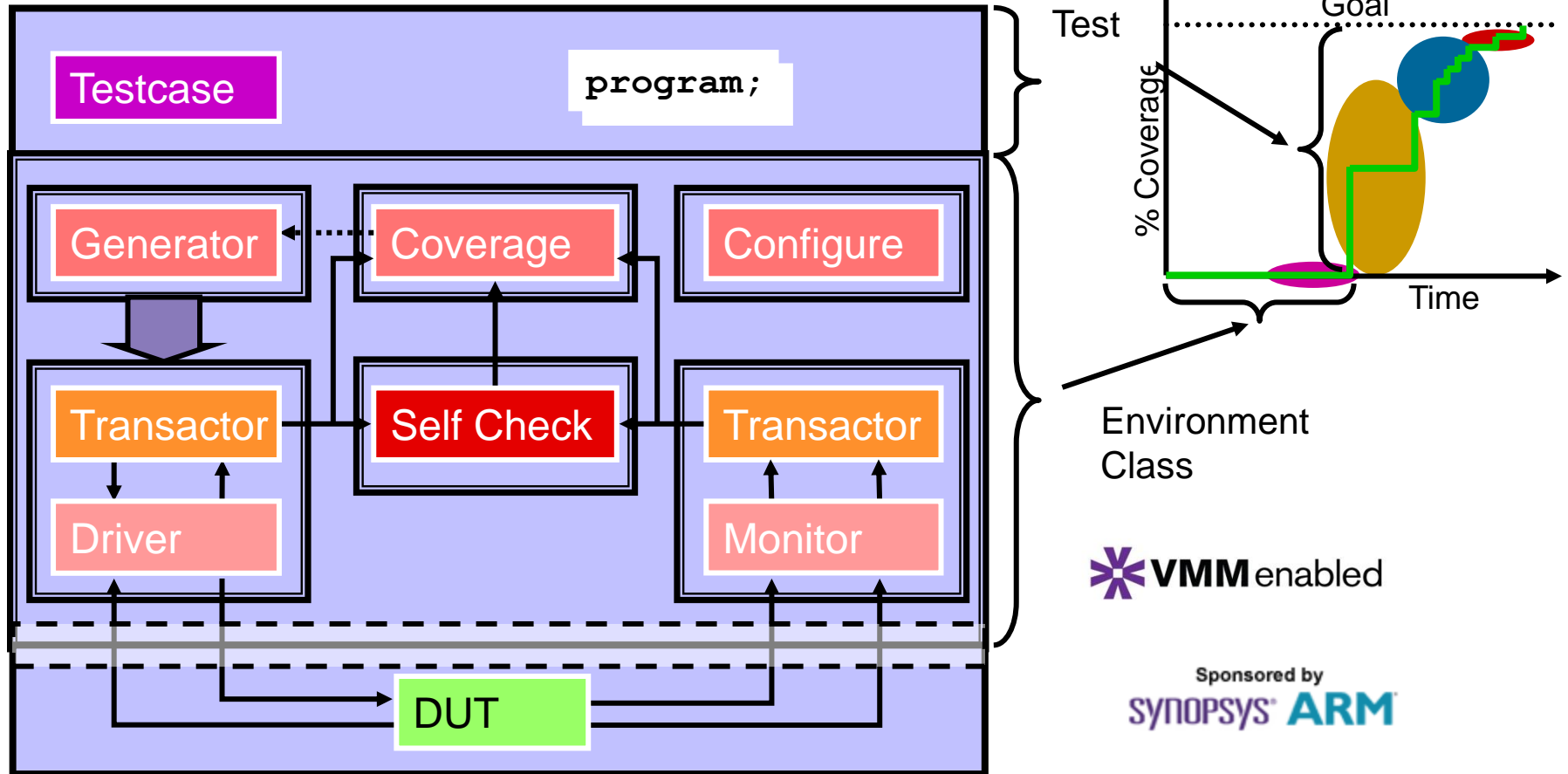
Coverage-Driven Verification

Phases of random stimulus based verification

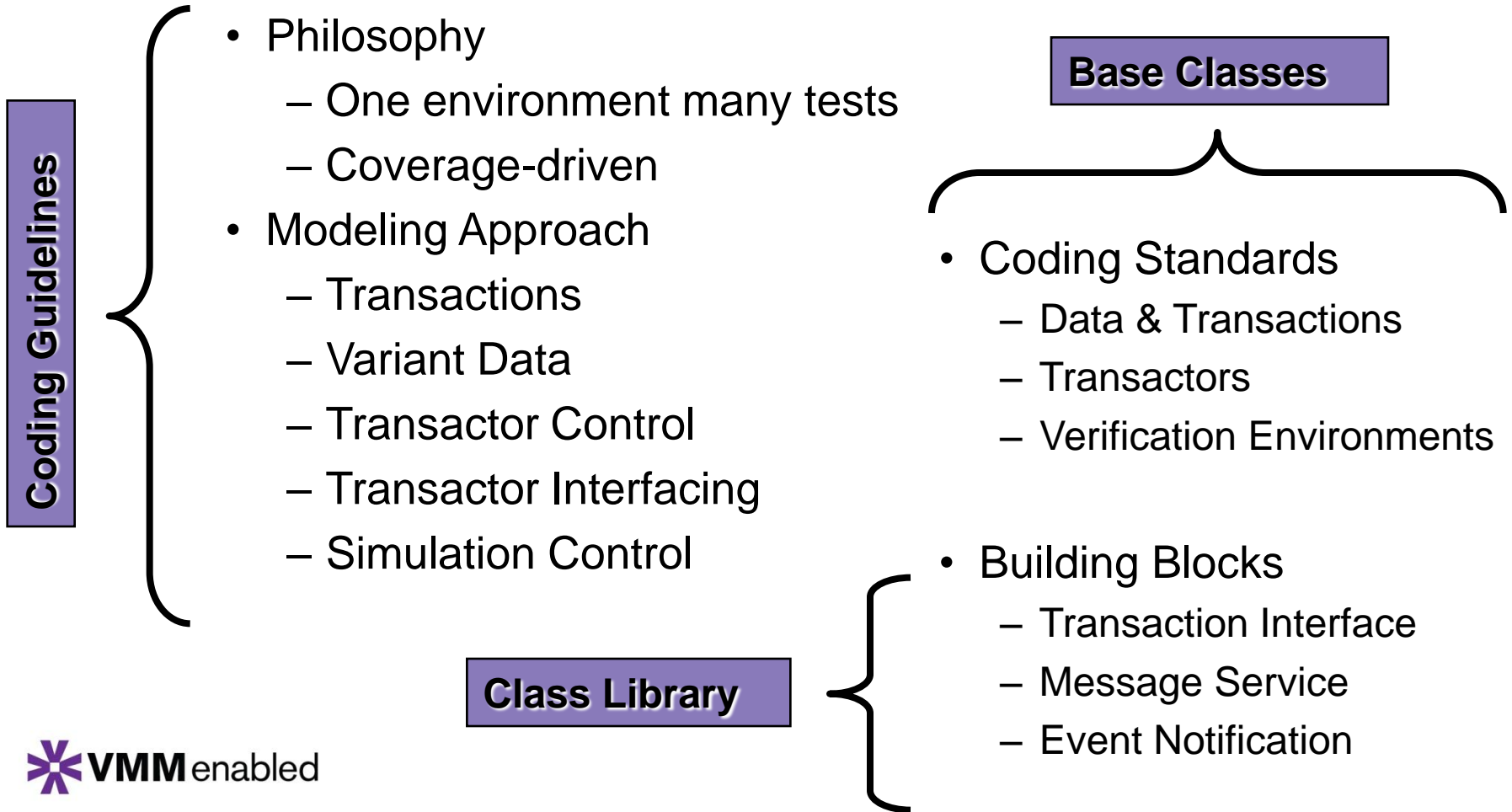


Verification Methodology Manual (VMM)

- Encapsulate testbench components in OOP Class



What Does VMM Provide?



References

- "IEEE Standard for System Verilog-Unified Hardware Design, Specification, and Verification Language," *IEEE STD 1800-2009* , vol., no., pp.C1-1285, 2009
- Bergeron, J., Cerny, E., Hunter, A., Nightingale, A., "Verification Methodology Manual for System Verilog", Springer 2006, XVIII, 510p
- VCS® MX / VCS® MXi Online documentation, Synopsys, June 2010.

Simulation with VCS

Outline

- VCS introduction
 - VCS Basics
 - DVE GUI Basic
- HDL Debug with DVE
 - Overview
 - Controlling the Simulation
 - Waveform Features
 - Features for Debugging

VCS INTRODUCTION

Agenda

- VCS Basics
- DVE GUI Basic

VCS MX Supports Two Major Flows

- 2-step flow for pure-Verilog users
 - Compilation, Simulation
- 3-step flow for mixed-language users
 - Analysis, Compilation, Simulation

- Why have 2 flows?
 - VHDL requires bottom-up analysis
 - Many Verilog users are familiar with traditional “Verilog-XL” flow

VCS MX Setup

- `${VCS_HOME}` should point to the root of the VCS installation

```
setenv VCS_HOME /tools/vcs/vcs2009.06-3
export VCS_HOME=/tools/vcs/vcs2009.06-3
```

- Optionally add `${VCS_HOME}/bin` to your path
- `${LD_LIBRARY_PATH}` should point to the license server
 - Optionally, you could use `${SNPSLMD_LICENSE_FILE}`

Flow Overview: Mixed-Language

3-Step Flow (UUM)

Map Logical Libraries
`synopsys_sim.setup`

Analyze Source Files

```
vlogan <files.v>  
vhdlan <files.vhd>  
syscan <files.cpp>
```

Elaborate/Compile Design

```
vcs <option> <design_top>
```

Simulate

```
simv <option>
```

- Map VHDL Logical Libraries
 - `synopsys_sim.setup`
- Analyze all Verilog source
 - *Command: vlogan*
- Analyze VHDL source
 - bottom-up
 - *Command: vhdlan*
- Compile the design
 - *Command: vcs*
- Simulate the design
 - *Command: simv*

VCS MX Setup File

example synopsys_sim.setup file

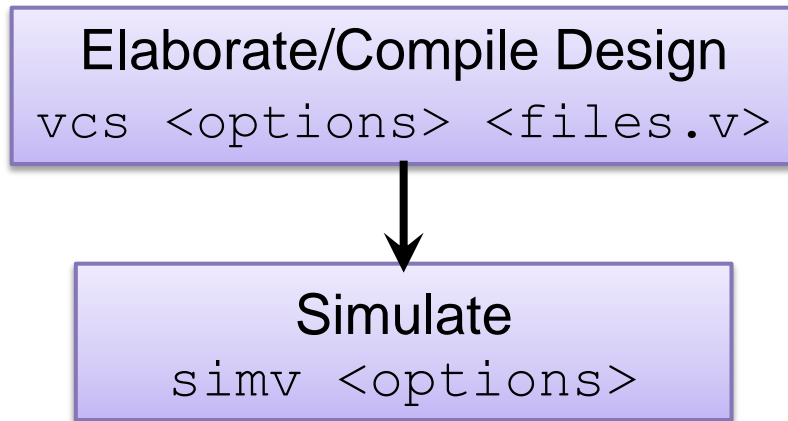
```
-- Example synopsys_sim.setup
-- see ${VCS_HOME}/bin/synopsys_sim.setup

-- Logical Library Mappings
WORK      > TB_LIB
TB_LIB    : /prj/libs/tb_lib
DUT_LIB   : /prj/libs/dut_lib
IP_BLOCK : ${VENDOR_LIB_PATH}

-- Simulator Variable Settings
ASSERT_STOP = ERROR
ASSERT_IGNORE = WARNING
TIME_RESOLUTION = 10 ps
```

Flow Overview: Pure Verilog

2-step Flow



- Compile the design
 - Specify all Verilog source code
 - Command: `vcs`
- Simulate the design
 - Command: `simv`
- Notes:
 - No setup file is needed
 - Verilog has no concept of logical libraries

Generating the executable

```
vcs entity_or_config_or_module <options>
```

- Elaboration and compile in a single step
- Elaboration
 - Binds the design hierarchy
 - Final reference resolving
- Compile
 - Code generation, Optimizations
 - Creates statically linked simulator executable (**simv**)

Common MX Elaboration options

- o <simv_name>
 - ucli
 - +incdir+<directory>
 - l <logfile>
 - R
 - gui
 - P pli.tab
 - sverilog
 - <.c|.o files>
 - debug_all | -debug | -debug_pp
- output user defined simulation name
 - enable command line interface
 - search paths for `include
 - creates runtime logfile
 - runs the **simv** immediately after compile
 - starts **simv** in DVE after compile
 - compiles user-defined system tasks
 - Selects Verilog version IEEE1800
 - Adds C or object files to compile or link
 - enable debug capabilities

use '**vcs -help**' other options

VCS Compilation Command Format

```
% vcs [compile_time_options] source_files
```

- `compile_time_options`
 - Controls how VCS compiles the source files
 - Critical for debug and performance
- `source_files`
 - Verilog source files: DUT and Testbench (SystemVerilog)
 - Vera
 - C/C++ source files
- Generates default executable binary named “`simv`”

Handling Different Verilog Versions

- VCS supports several Verilog versions
 - You can get caught in legacy code
 - “byte” is a reserved keyword in SystemVerilog
- Tell VCS which version by file extension:

```
% vcs -sverilog +verilog2001ext+.v2k +verilog1995ext+.v95
```

Assumes SV

```
% vcs +v2k +systemverilogext+.sv +verilog1995ext+.v95
```

Assumes v2k

```
% vcs +verilog2001ext+.v2k +systemverilogext+.sv
```

Assumes v95

Interactive Mode

- Interactive is single user mode
- Starting from a compilation

```
% vcs <universal> <options> -R -gui -debug_all
```

-R Starts simulation immediately after compilation

-gui Enables DVE to start at runtime, stops at time 0

- Run the Simulation (either Verilog or MX)
 - Batch/regression mode

```
% simv
```

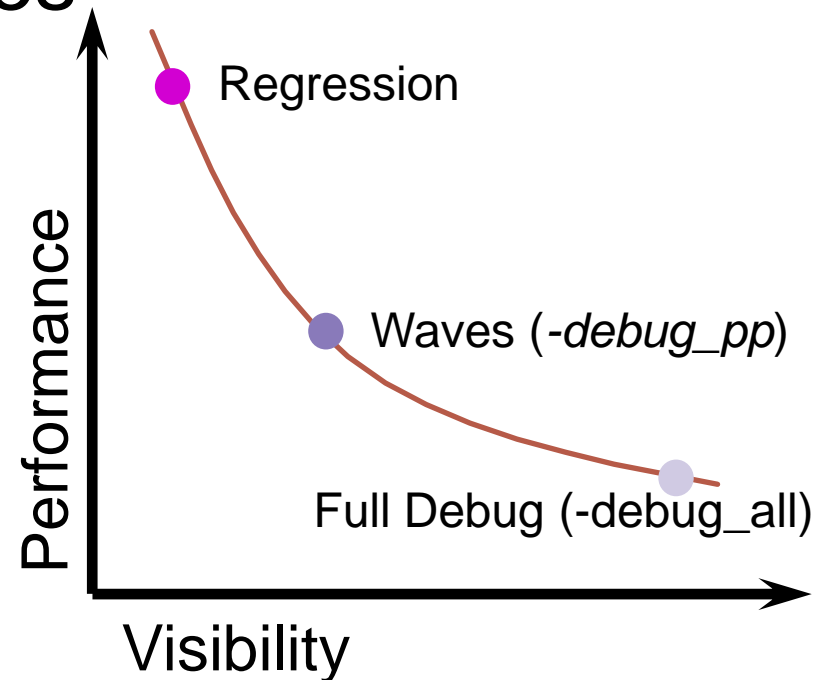
- Interactive mode: with DVE simulation GUI

```
% simv -gui <options>
```

Performance Considerations

What Affects Simulation Speed?

- Excessive I/O
- Inefficient PLI
- Enabling debug features
- Coding styles
- Compile time options
- 32-bit vs. 64-bit



Additional Resources

- Help for executable commands

```
%> command_name -help
```

- VCS/MX Documentation
(Start with chapter “Migrating to VCS MX”).

```
%> vcs -doc
```

- SNUG Papers and Tutorials

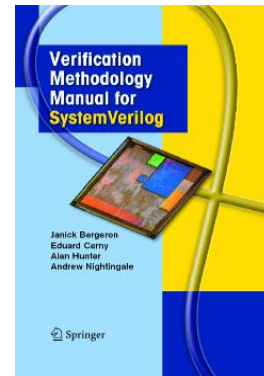
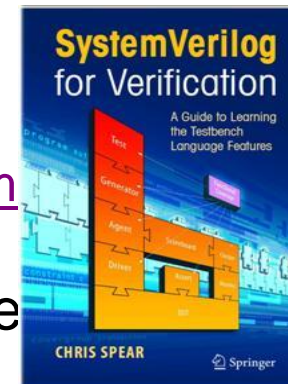
<http://www.snug-universal.org/papers/papers.htm>

- Self service using the Knowledge Database

<http://solvnet.synopsys.com>

- Examples

```
`${VCS_HOME}/doc/examples
```



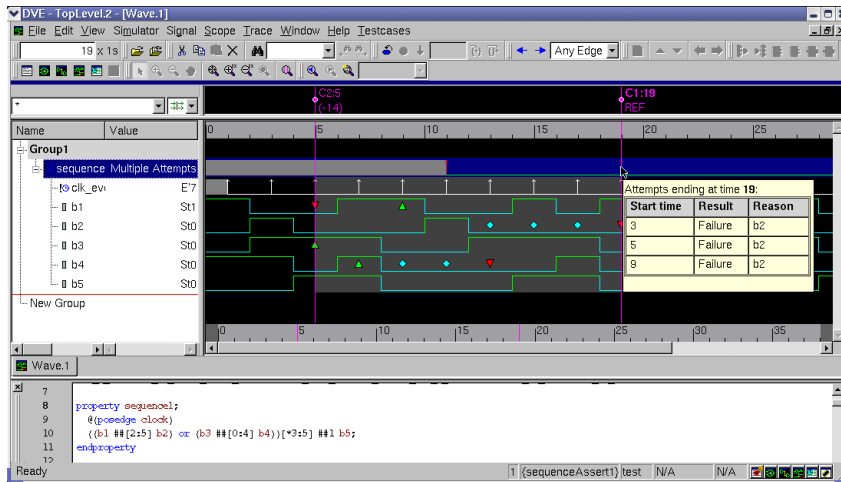
Questions and Help:

VCS_Support@Synopsys.com

Agenda

- VCS Basics
- DVE GUI Basic

Discovery Visual Environment



Intuitive GUI to Quickly Find Bugs

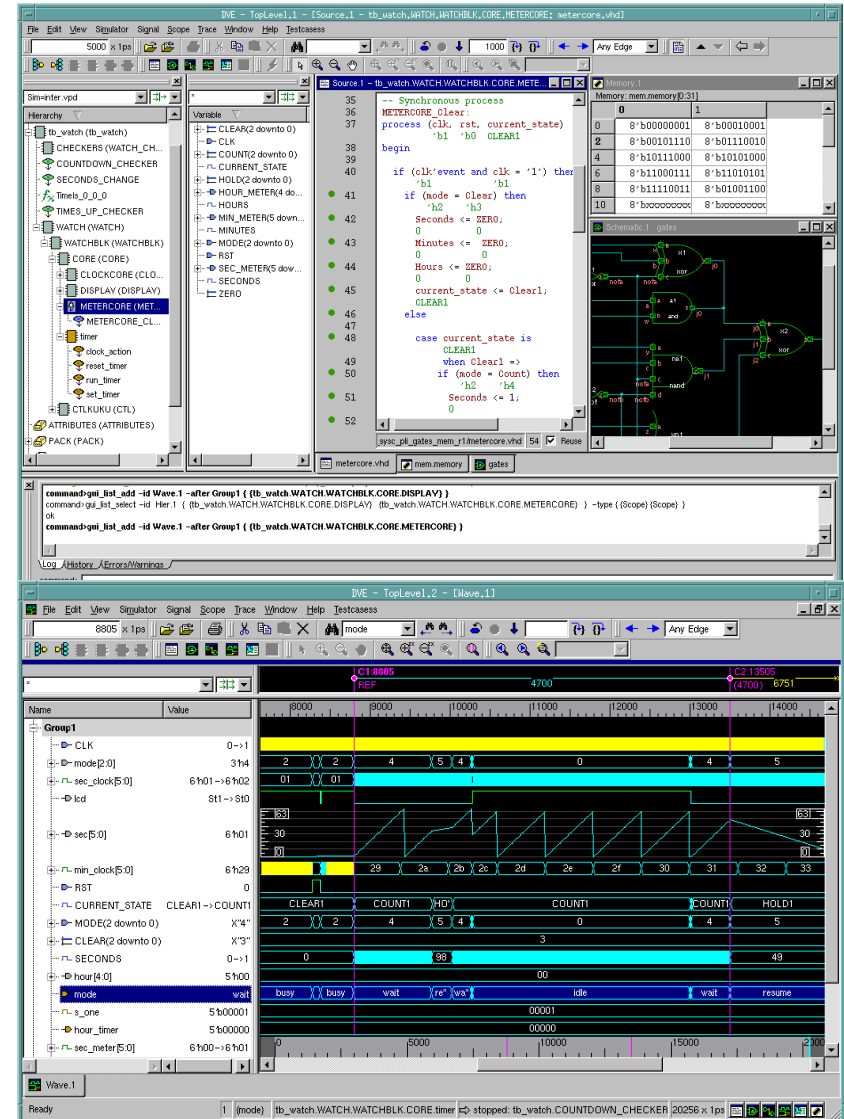
RTL or Gate
 Assertions
 Testbench
 Coverage

Multiple Languages

Verilog
 VHDL
 C/C++
 SystemC
 SystemVerilog
 OpenVera
 Analog

Supported Flows

Interactive
 Post-simulation analysis



Discovery Visual Environment

Overview of Primary Panes

Unified design hierarchy

Local variables and ports

Source code - annotated

UCLI commands - logged or typed

Tabbed or floating windows

The screenshot displays the Discovery Visual Environment interface with several panes:

- Hierarchy:** A tree view showing the design structure, including modules like `alu1 (alu)`.
- Variable:** A table listing variables and their types:

Variable	Type
<code>..accum[7:0]</code>	Wire
<code>..alu_out[7:0]</code>	Reg
<code>..clk</code>	Wire
<code>..data[7:0]</code>	Wire
<code>..opcode[2:0]</code>	Wire
<code>..zero</code>	Reg
- Source code:** A window showing the source code for the `alu` module, annotated with comments and including a testbench.

```
3 module alu (alu_out, zero, opcode, data, accum, clk);
4
5   input [7:0]  data, accum;
6   input [2:0]  opcode;
7   input
8
9   output [7:0] alu_out;
10  reg [7:0]    alu_out;
11
12  output      zero;
13  reg         zero;
14
15  initial $display("**Test passed\n");
16
17  initial
18     zero = 0;
19
20  `include "../source/opcodes.v"
```
- UCLI:** A command-line interface showing typed commands and their output:

```
command>gui_list_select -id
ok
command>gui_list_select -id
ok
command>gui_list_action -id
Log /History /Errors/Warnings /
command>
```
- Waveform:** A timing diagram showing signals like `clk2`, `address[4:0]`, `accum[7:0]`, `load_accum`, `inc_pc`, `fetch`, and `reset`.

Expression eval
Bus builder
Driver/load tracing

Waves:
- HDL
- SystemC
- C/C++
- TestBench
- Analog

Testbench Thread Debugging

SystemVerilog Testbench

- Effective debugging of threads and inter-process communication
 - Put breakpoints on lines, threads activation, semaphore & mailbox statuses
 - Display class contents and any other dynamic objects
 - Automatically updates as simulation progresses
 - Waveform support of global and static variables

The screenshot displays the Synopsys simulator interface for testbench thread debugging. The main window shows the testbench source code with a breakpoint set at line 18. The interface includes several panels:

- Local "watch" window:** Located at the top left, it shows a table of variables and their values. The current context is 'design_env::1'.

Variable	Value
this	class design_env
m	class vip_master
a1	0
b	0
- Thread browser:** Located on the left side, it shows a hierarchy of modules and threads. The current thread is 'if1 (vip_if)'.
- Interactive UCLI commands:** Located at the bottom left, it shows the command prompt with the following commands and output:

```
dve> thread -running
dve> thread -current
thread #4 : (parent: #<root>) CURRENT
1 : -line 18 -file orig_eg-from-Phil.v -scope {top.tb1.
```
- Testbench source code:** The main window displays the testbench source code with a breakpoint at line 18.

```
2 program testbench;
3 class vip_master;
4 int a;
5 virtual vip_if ii;
6 function new(string instance,
7 virtual vip_if bus);
8
9 this.ii = bus;
10 a = 1;
11 endfunction
12 endclass
13
14 class design_env;
15 vip_master m;
16 int b;
17 virtual function void build();
18 this.m = new("Host", top.if1);
19 b = 2;
20 endfunction
21 endclass
22 design_env e = new();
23 initial begin
24 e.build();
25
```
- Global "watch" window:** Located at the bottom right, it shows a table of global variables and their values.

Expression	Value
top.clk	0
{top.dut.busdrv[1:4]}	x
top.tb1.design_env::build.this.m	<not-active>

Discovery Visual Environment

Two methods of debugging

1. Interactive Debug

- Source browsing, line stepping, breakpoints, etc

2. Post-simulation Debug

- Generate a VPD (VCD+) containing all waveforms
- Debug simulation after-the-fact
 - Speeds up the overall debug process!
 - Instant access to all values at all times during the simulation
- Makes better use of your simulation licenses
 - Standalone GUI does not use a simulation runtime license

Selected Online DVE Training Videos

- Testbench debugging with DVE:
<https://solvnet.synopsys.com/retrieve/023564.html>
- DVE Flows : RTL Debug
<https://solvnet.synopsys.com/retrieve/025671.html>
- DVE : Driver Tracing
<https://solvnet.synopsys.com/retrieve/021729.html>
- DVE FAQ
<https://solvnet.synopsys.com/retrieve/019017.html>

HDL DEBUG WITH DVE

Documentation

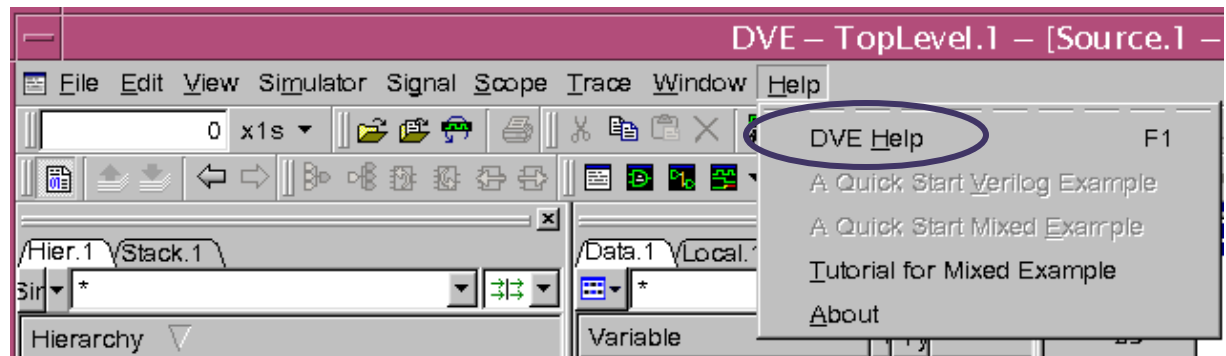


- User reference manual in html format
 - Now viewable in any web-browser with easy expand/collapse listings, tabs for Index, Contents, Search and Favorites
 - Point browser to `$VCS_HOME/doc/UserGuide/userguide_html`
 - `vcs -doc`
- Release notes (DVE)
 - `$VCS_HOME/gui/dve/doc/DVEReleaseNotes.txt`
- Quick start example
 - `$VCS_HOME/gui/dve/examples/tutorial/quickstart/quickStart.html`
 - Help-> Tutorial (for Mixed HDL)

- Within DVE:

or

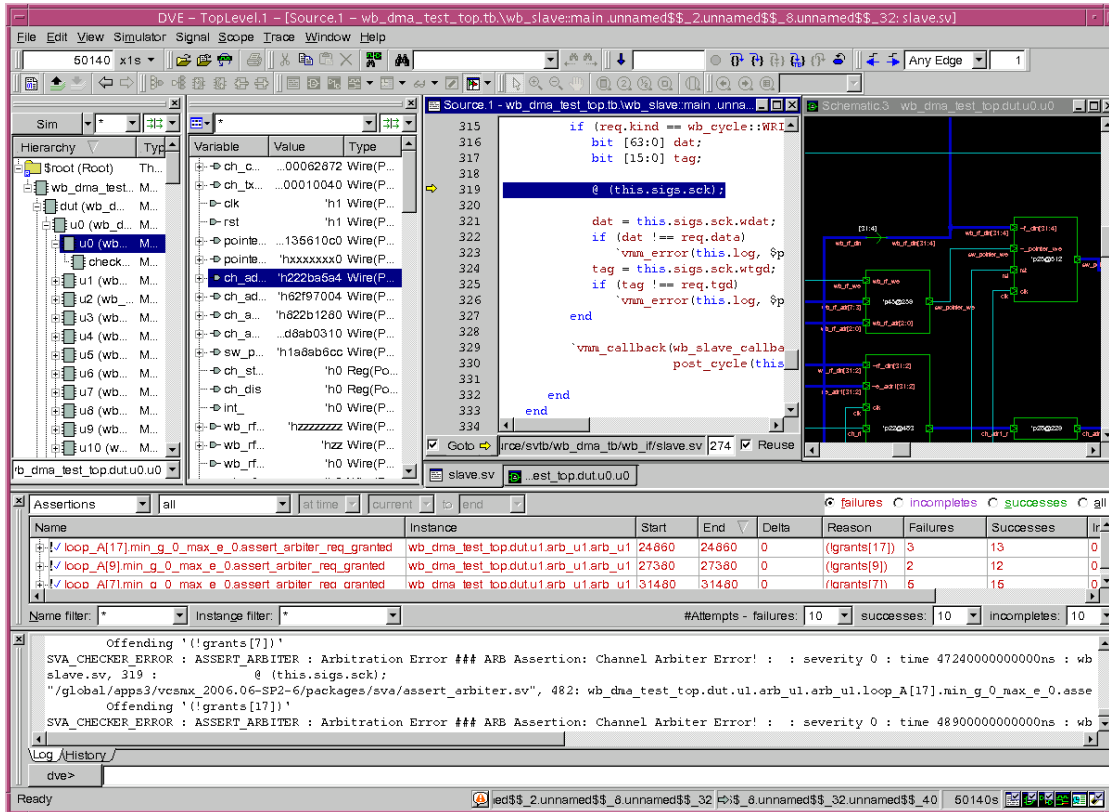
```
% vcs -doc
```



Agenda

- Overview
- Controlling the Simulation
- Waveform Features
- Features for Debugging

Design Debug Productivity

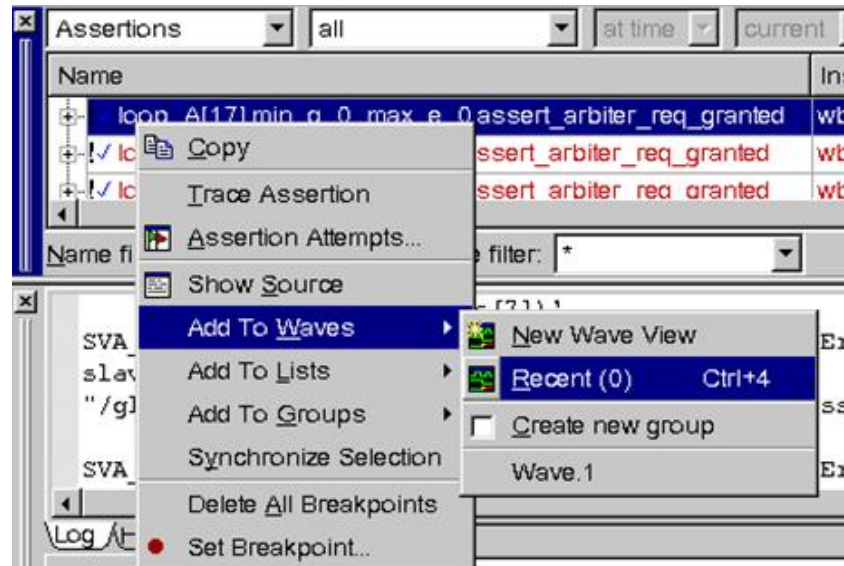


- Intuitive and Easy to Use
- Quickly Find Bugs
 - RTL or Gate
 - Assertions
 - Testbench
- Supports
 - Interactive and
 - Post-simulation analysis
- Multiple Languages
 - Verilog
 - VHDL
 - C/C++
 - SystemC
 - SystemVerilog
 - OpenVera

Docked windows inside workspace boundaries

Context Sensitive Menus (CSM)

- Point at an object
 - Signals, instances, ports, panes, and assertions.
 - Configure main toolbar
- Click Right Mouse Button (RMB) down
 - Menu appears with relevant options
- Click on choice



Object Selection

- Objects
 - Instance, Signal, Class, Assertion, etc...
- Drag and Drop
 - Point at an object in a pane or window
 - Hold LMB down
 - Drag object to a new location and release
- Select Multiple Items
 - LMB and Control key (to add or remove an item to selection)
 - LMB and Shift key (to group select)
 - LMB and drag to select a group of objects

Invoking DVE

Interactive Mode

- Starting from compilation

```
% vcs source.sv -R -gui -debug_all
```

-R	Runs executable immediately after compilation (optional)
-gui	Enables DVE
-debug	Enables command line debugging (no line stepping)
-debug_all	Enables command line debug including line stepping (optional)
-ucli	Forces runtime to go into UCLI debugger mode (optional)

- Start DVE from existing simulation executable

```
% simv -gui
```

Invoking DVE

Post-Processing Mode

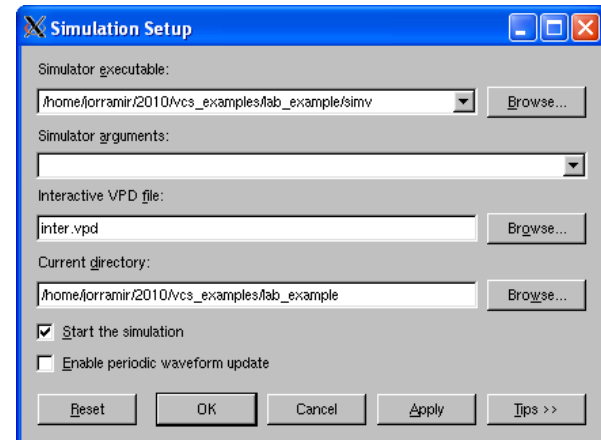
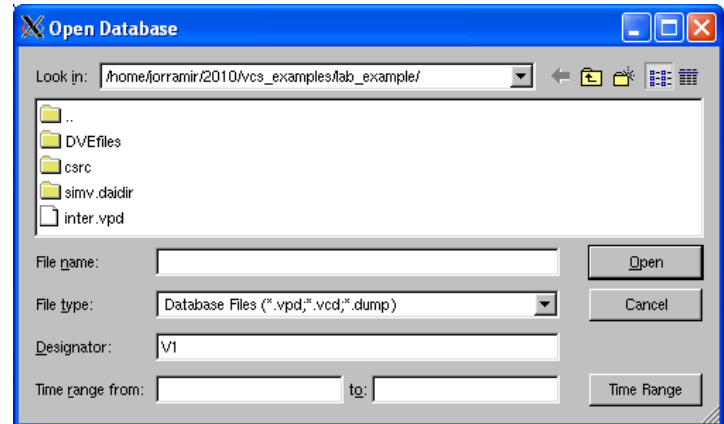
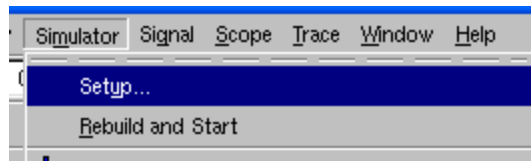
- Launch DVE GUI

```
% dve &
```

- Open database (vcd,vpd)
 - Click the Open Database icon open dialog box



- Open simulation file



DVE Top Level Window

Movable columns

Close window or pane

Filters

Source Window

Data Pane

Hierarchy Pane

Assertion Pane

TCL command line

Toggle console window on/off

Status

panes



The screenshot displays the DVE Top Level Window with several panes and components:

- Hierarchy Pane:** Shows a tree view of the design hierarchy, including modules like `u0 (wb_dma_rf)` and `u0 (wb_dma_ch_rf)`.
- Data Pane:** Displays a table of variables and their values, such as `ch_csr[31:0]` with value `...006 0866`.
- Source Window:** Shows Verilog code, including assignments like `assign ch_txsz[31:0] = CH_EN & de_txsz_we & (ch_sel==CH_NO);`.
- Assertion Pane:** Displays a table of assertions, including `loop_A[19].min_g_0_max_e_0.assert_arbiter_req_granted`.
- TCL Command Line:** Shows the command `dve>` and simulation metrics like `CPU Time: 9.250 seconds`.
- Status:** Shows the simulator status at the bottom, indicating `The simulator is not active: N/A`.
- panes:** A list of available panes on the right side, including Console, Hierarchy, Data, Signal Groups, DriverLoad, Stack, Local, Watch, and Assertion.

Agenda

- Overview
- Controlling the Simulation
- Waveform Features
- Features for Debugging

Interactive Simulation Control - (1/3)




- Simulation execution
 - Click the continue icon  to “start/continue”
 - Click the stop icon  to stop
 - Enter a ucli command 7
 - ucli% **run** (run until break point)
 - ucli% **run 100** (run for 100 time units)
 - ucli% **run 100ms** (run for 100 ms)
 - ucli% **stop -assert chkRstSeq -any** (assertion break point)
 - ucli% **run -posedge wb_ack_i** (run until positive of wb_ack_i)
 - Use simulator controls to set a simulation break point and run

- “Step Time”

- “Go To Time”

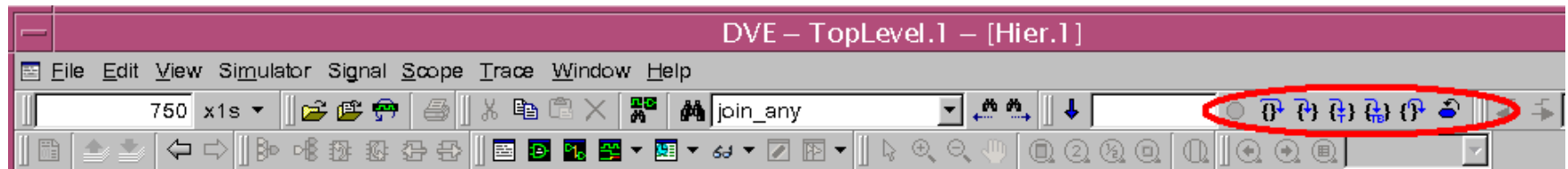


Interactive Simulation Control - (2/3)

- Simulation controls
 - Click step icon  to simulate to next executable line
 - Click next icon  to step over tasks and functions
 - Click restart icon  to reset simulation to time zero
 - ucli commands
 - ucli% step
 - ucli% next
 - ucli% restart
- Finishing the simulation
 - Tcl command “finish” works the same as \$finish system task

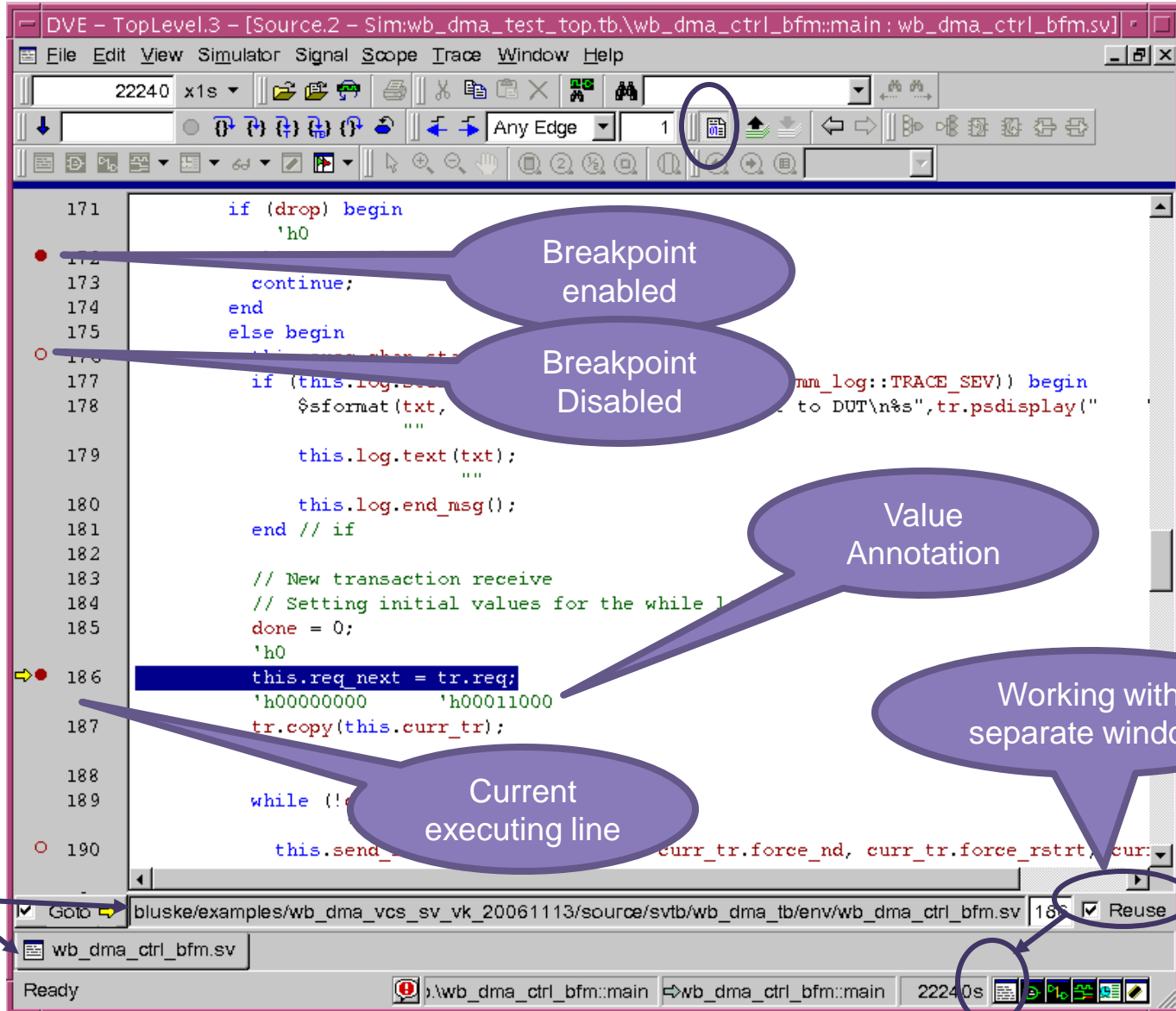
Interactive Simulation Control - (3/3)

- Stepping source (-debug_all)
 - ↓ Start / Continue
 - ↻ Step over tasks and functions
 - { } Simulate next executable line
 - { } Step into active thread (*testbench specific*)
 - { } Step into any testbench thread (*testbench specific*)
 - { } Step out of tasks and functions
 - ↺ Restart simulation
 - Stop simulation



Interactive Debug – Source Window

Overview



- Set Breakpoint
- Disable Breakpoint
- Delete Breakpoint
- Properties...
- Delete All Breakpoints
- Breakpoints...
- Line Number

Breakpoint CSM

Source file & location

Agenda

- Overview
- Controlling the Simulation
- **Waveform Features**
- Features for Debugging

Wave Window

Overview

Zoom gestures:
Zoom in 2X (up-right)
Zoom out 2X (down-right)
Zoom full (up or down)
Last zoom (down-left)
Next zoom (up-left)

Set time

Search icons

Viewing time range (current range display)

Signal groups

Drag Zoom

Absolute time range (entire range display)

Marker location

Find: Wave.1
Find: wb*
Find Next
Match case Wrap around
Match whole word only Search backwards
Use: Wildcards
Field: <Any>


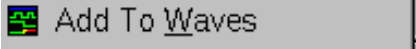
DVE - TopLevel.2 - [Wave.1]
153004500 x 10ps
wb*
Any Edge
Any Edge
Rising
Falling
Failure
Success
Value...

Name	Value
wb_dma_mast	
clk	St0->St1
St1	St1
St0	St0
wb_addr_o[31:0]	32'h0000 0410
wishboneMstChecks	
WB_MAS_3_25	Success
WB_MAS_3_75_4	Success
WB_MAS_3_75_5	Success
CLK_I	St0->St1
STB_O	St0
MAX_LATENCY[31...	50
ACK_I	St0
ADD_O[31:0]	32'h0000 0410
ADD_WIDTH[31:0]	32
CYC_O	St1

Wave.1
Ready

Wave Window


Managing Signals

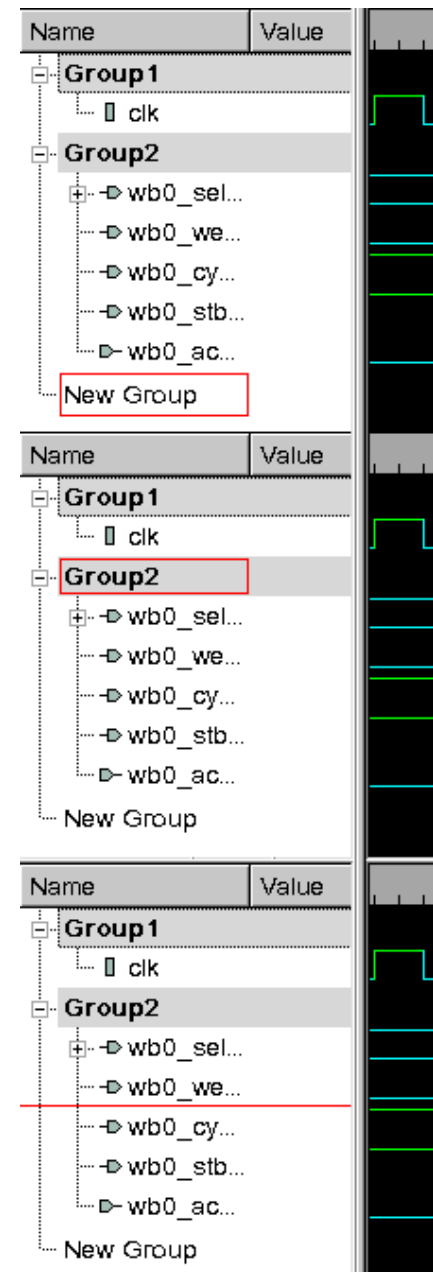
- Viewing signals
 - Select object (signals, scopes or assertions)
 - Click the wave icon  to add objects to a wave window
 - Use CSM and select  item
 - Double click on a failing assertion summary tab
 - Or drag and drop object to open wave window
- Grouping signals
 - Select object (signals, scopes or assertions)
 - Use CSM and select
 - Or drag and drop object to open signal groups pane
 - Or drag and drop object to desired group in open wave window

Wave Window

Insertion Bar

Middle mouse
button

- **MMB**  **on group or signal name**
 - MMB on new group to add signals to a new group
 - MMB on existing group to append signals to end of existing group
 - MMB on signal name to insert signals after desired signal
- **Add signals as normal**
 - Add waves icon
 - CSM
 - Hotkey (default: Ctrl+4)



The image displays three sequential screenshots of the Wave Window interface, illustrating the effect of the Middle Mouse Button (MMB) on signal insertion. Each screenshot shows a tree view with columns for 'Name' and 'Value'. The tree structure includes 'Group1' (containing 'clk') and 'Group2' (containing 'wb0_sel...', 'wb0_we...', 'wb0_cy...', 'wb0_stb...', and 'wb0_ac...').

- Top Screenshot:** A red box highlights the 'New Group' entry at the bottom of the tree, indicating the initial state before insertion.
- Middle Screenshot:** A red box highlights 'Group2', showing that signals have been appended to the end of this group.
- Bottom Screenshot:** A red box highlights the 'wb0_cy...' signal within 'Group2', showing that a new signal has been inserted after this specific signal.

Agenda

- Overview
- Controlling the Simulation
- Waveform Features
- Features for Debugging

User Defined Radices

- User Defined Radices

- **Toolbar menu:** Signal -> Set Radix->User-Defined->Edit

- Import or export user types: IDLE 11'b00000000001 -- file format

The screenshot shows the DVE (Digital Verification Environment) interface. The main window displays a signal trace for 'state[10:0]' with a value of 'READ->WRITE'. The trace shows a sequence of operations: READ, WRITE, READ, WRITE, READ, WRITE. The 'Edit User-Defined Radix' dialog box is open, showing a list of user-defined radices. The 'User-Defined Radix' field is set to 'dma_type'. The list includes: IDLE, READ, WRITE, UPDATE, LD_DESC1, LD_DESC2, LD_DESC3, LD_DESC4, LDDEC_5, WB, and PAUSE. The 'New' button is highlighted, indicating the process of creating a new radix.

To create a user-defined radix, click New, enter a radix name, then press Return.

Waveform Compare Tool

- Compare two signals, scopes or designs
 - **Toolbar menu: Signal -> Compare**
 - A new signal is created for each compare point

Name	Value
Group1	
ch_sel[4:0] <> ch_sel[4:0]	mismatch
ch_sel[4:0]	5'h1c
ch_sel[4:0]	5'h10
ch_sel[4] <> ch_sel[4]	NA
ch_sel[3] <> ch_sel[3]	NA
ch_sel[3]	S11
ch_sel[3]	S10
ch_sel[2] <> ch_sel[2]	NA
ch_sel[2]	S11
ch_sel[2]	S10
ch_sel[1] <> ch_sel[1]	NA
ch_sel[0] <> ch_sel[0]	NA

Waveform Compare

Compare selection:

Reference waveform: Design: Sim=test_1_600.vpd

Test waveform: Design: V1=test_1_234567.vpd

Signals/Scopes: {Sim:wb_drra_test_top.dut.u0.ch_sel[4:0]}

Signals/Scopes: {V1:wb_drra_test_top.dut.u0.ch_sel[4:0]}

Offset: 0 x 1s

Offset: 0 x 1s

Load Ref Signals/Scopes ...

Same signals/scopes as Reference

Results summary:

Compare results Summary:

- Number of signals compared: 1
- Number of values compared: 148
- Number of mismatches found: 12

More Options >> OK Cancel Apply Reset Tips>>

Example – Comparing Interactive simulation signal (design 1) to post processed reference simulation (design 2)

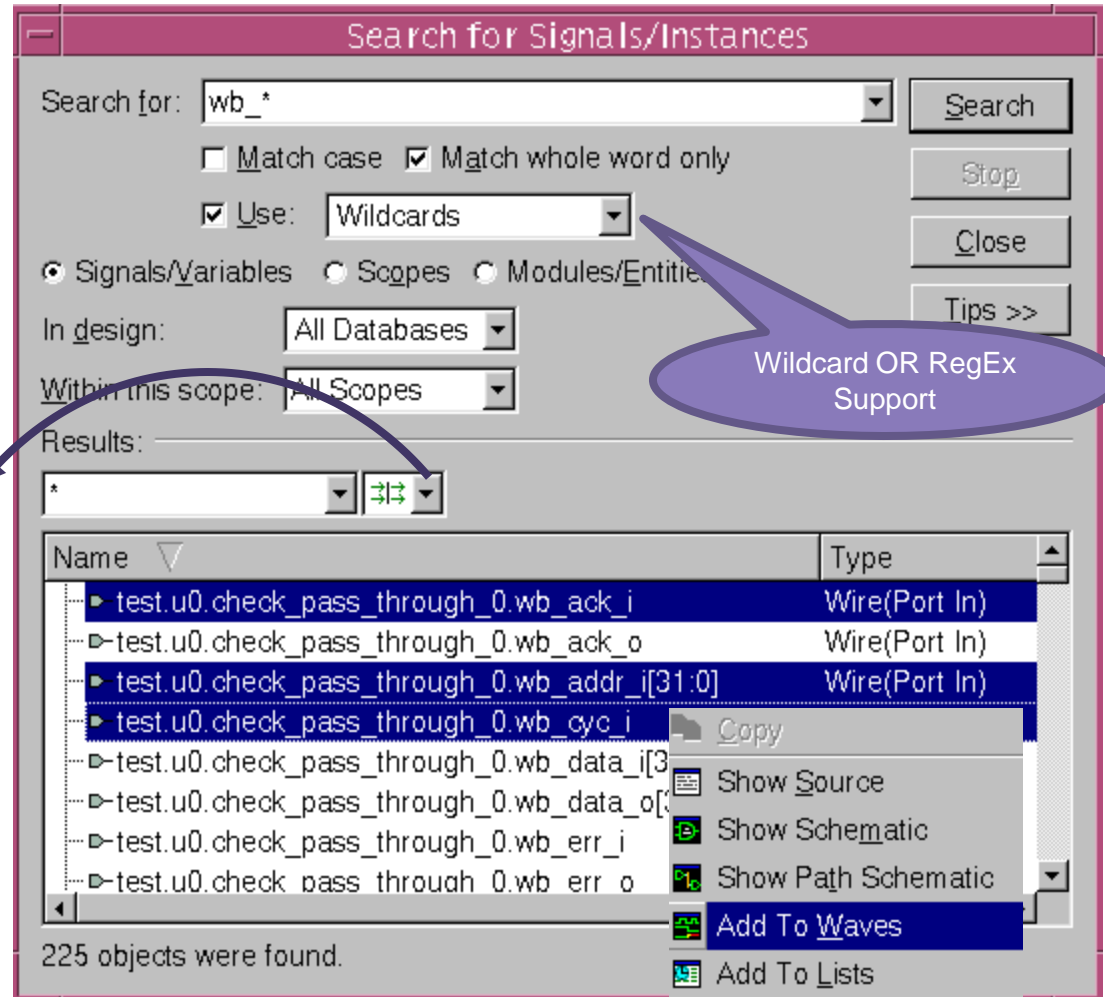
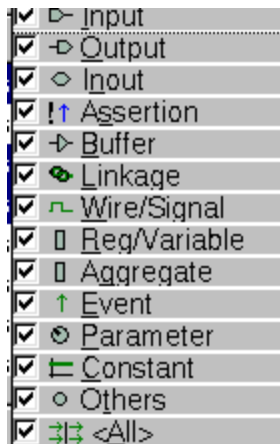
Searching for Objects

- Toolbar menu: Edit -> Search for Signals/ Instances or click 

– Viewing objects





- Select objects
- Right click to activate CSM
- Select window type
 - e.g. Wave

Filter results



Select All

Tracing Drivers & Loads

- Problem
 - A number of signals exhibiting less than desirable values
- Solution
 - Perform a “*backtrace*”
 - Displays a list of active drivers/loads at specified time
 - Trace back to the earliest unwanted signal transition or value
 - Identify signal responsible for the erring behavior
 - Reapply procedure, and eventually locate source of misbehavior
- Displaying Drivers/Loads
 - In any DVE analysis window highlight or select a signal
 - Then simply click either the  driver or  load icon
 - Or in a Wave or list window double click on signal
 - Then use next  and previous  instances icons

Driver / Load Pane

The screenshot displays the Synopsys Design Verification Environment (DVE) interface. The main window shows a Verilog code snippet with the following content:

```

561 channel select
562 always @(posedge clk or negedge rst)
563   if(!rst) ch_sel_r <= #1 0;
564   else
565     if(de_start) ch_sel_r <= #1 ch_sel_d;
566
567 assign ch_sel = !dma_busy ? ch_sel_d : ch_sel_r;
568
569 //////////////////////////////////////
570
571 // ... based on arbiter
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000

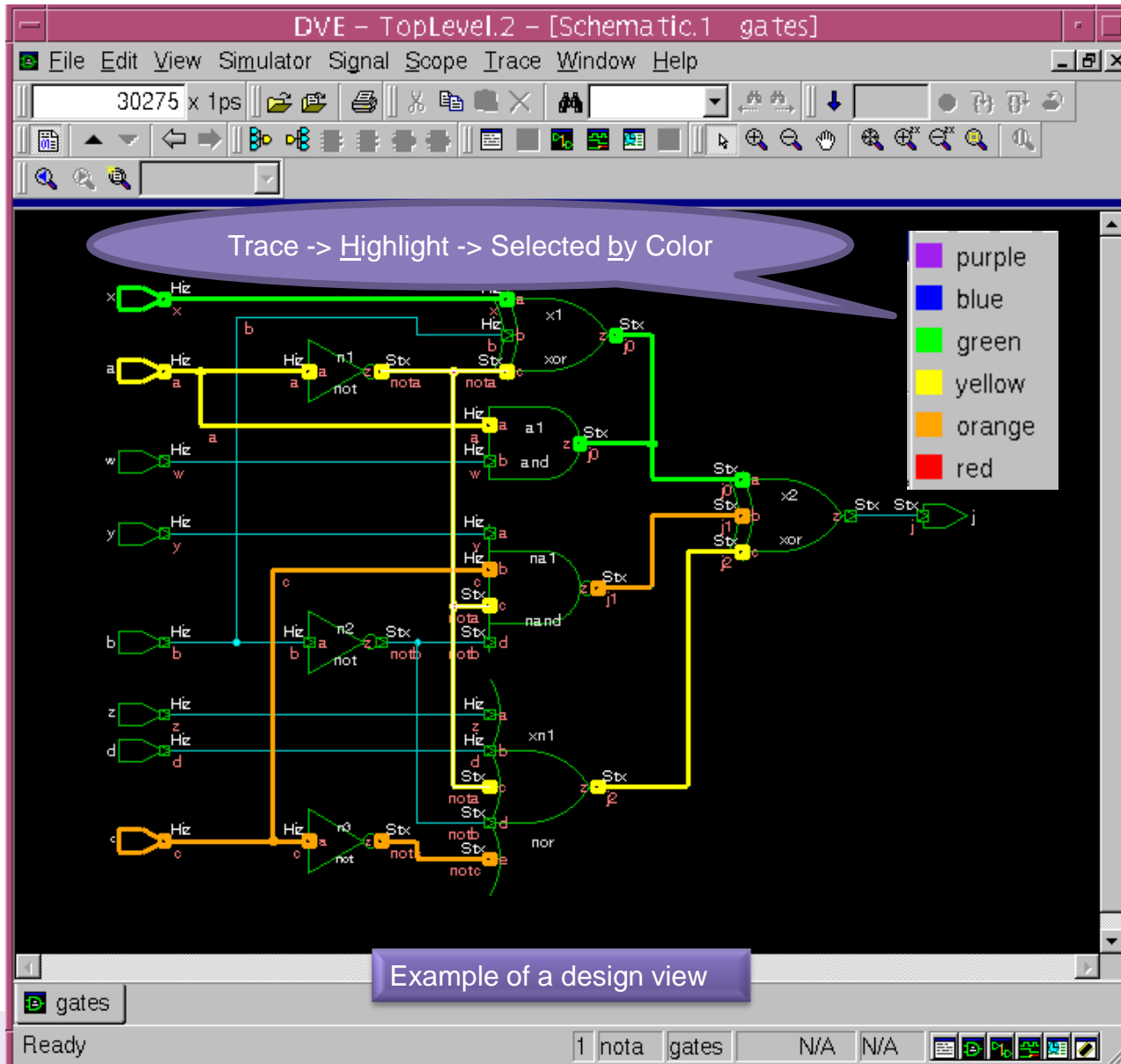
```

The interface includes several panes:

- Hierarchy:** Shows a tree view of the design components, with `u0 (wb_dma_ch...)` selected.
- Variable:** Lists variables and their types, with `ch_sel[4:0]` selected.
- Code Editor:** Displays the Verilog code with line numbers 561-999. Callouts point to line 563 (Current Instance) and line 565 (Previous Instance).
- Driver / Load Pane:** A table showing the current state of signals and drivers at time 128560. Callout points to the pane header.
- Log/History:** Shows simulation logs, including messages like "1 driver(s) found for signal".

Signals/Drivers/Loads	Value	Time	Line/Code
ch_sel_r[4:0]	NA	128560	
if(!rst) ch_sel_r <= #1 0;	NA	128560	563 ./source/verilog/wb_dma_dut/w
if(de_start) ch_sel_r <= #1 ch_sel_d;	NA	128560	565 ./source/verilog/wb_dma_dut/w
ch_sel[4:0]	NA	128560	
assign ch_sel = !dma_busy ? ch_sel_d : ch_sel_r;	2h'07	128560	567 ./source/verilog/wb_dma_dut/w
ch_sel[4:0]	NA	128560	
assign ch_sel = !dma_busy ? ch_sel_d : ch_sel_r;	2h'07	128560	567 ./source/verilog/wb_dma_dut/w
ch_stop	NA	128560	
ch_stop <= #1 CH_EN & ch_csr_we & wb_rf_din[...	NA	128560	354 ./source/verilog/wb_dma_dut/w

Schematic Window



Path Tracing

DVE - TopLevel.2 - [Schematic.1 tb_watch.hour]

File Edit View Simulator Signal Scope Trace Window Help

30275 x 1ps

Trace -> Follow Signal

Cell:
tb_watch.WATCH.WATCHBLK.CORE.DISPLAY.*p1@71 (*t23)

Double click on pin to expand path

Hierarchy crossings

Hierarchy Crossing Up:
tb_watch.WATCH.WATCHBLK.CORE.min

Net:
tb_watch.sec

Example of a path view

Ready

1 tb_watch.WATCH.WATCHBLK.CORE.min

N/A N/A



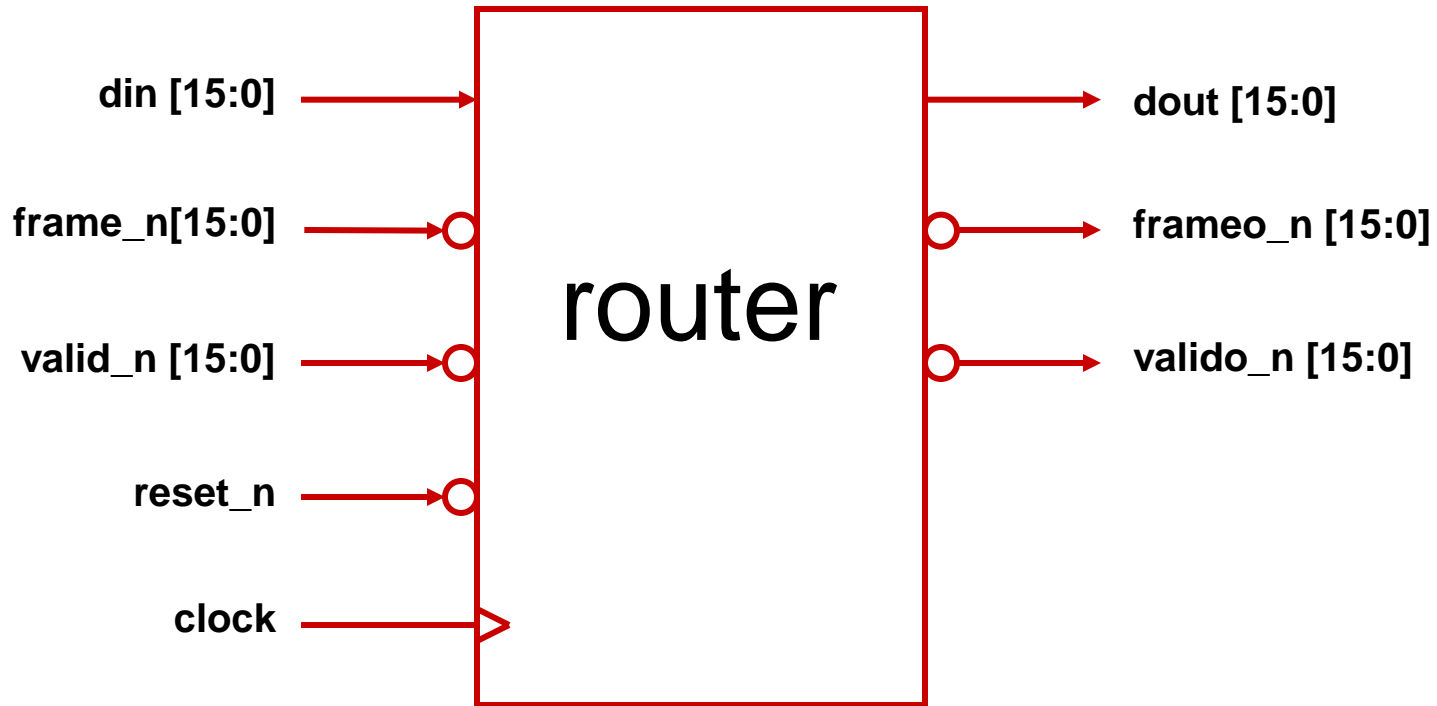
Lab Time!

- We will finish with a 30 min lab to test the basics.
- Individual work.
- When finish you check out:
 - `$VCS_HOME/doc/examples/testbench/sv/tutorial`
- We will be around for at least 60 min for questions.

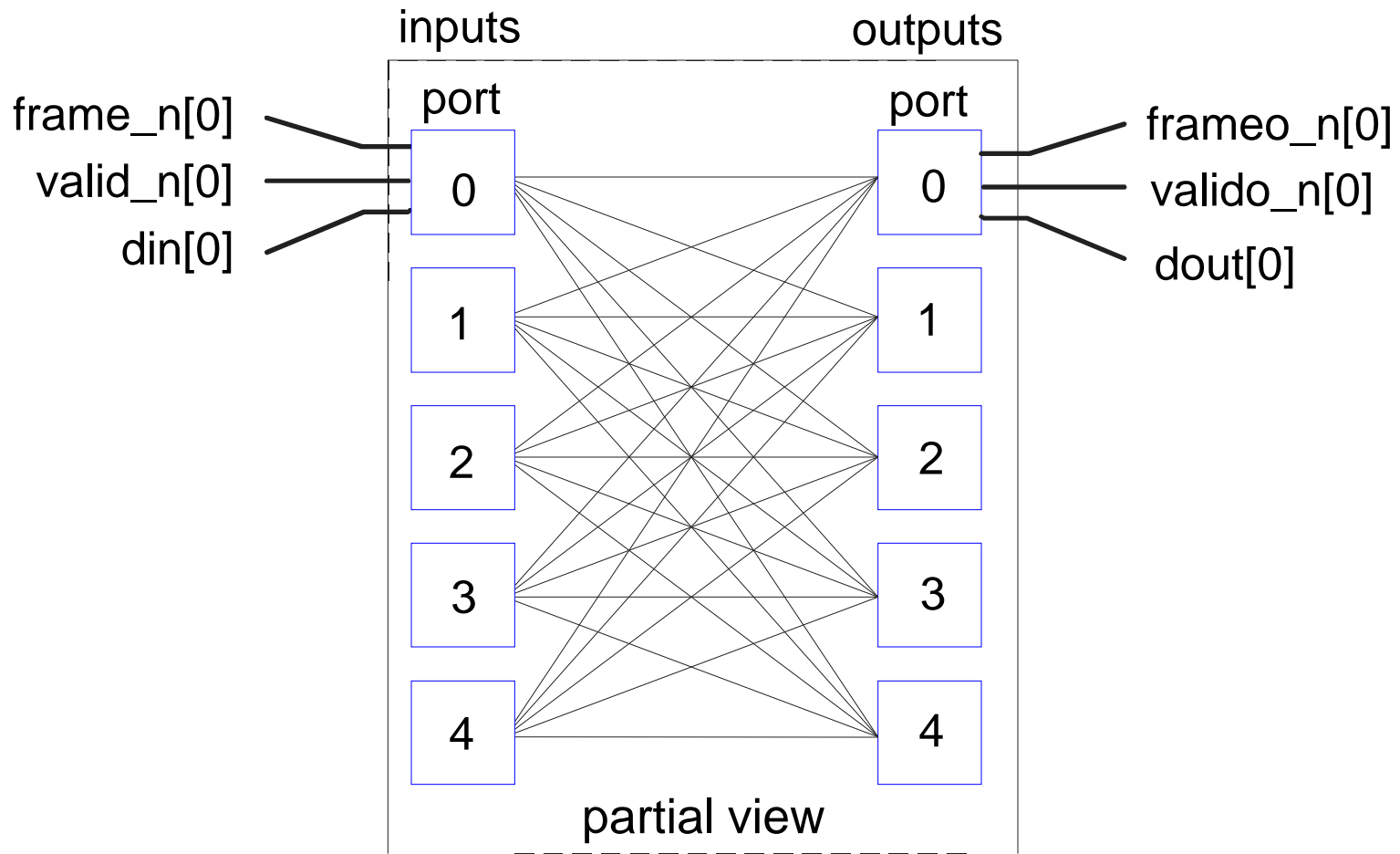
What Is the Device Under Test?

A router:

16 x 16 crosspoint switch



A Functional Perspective

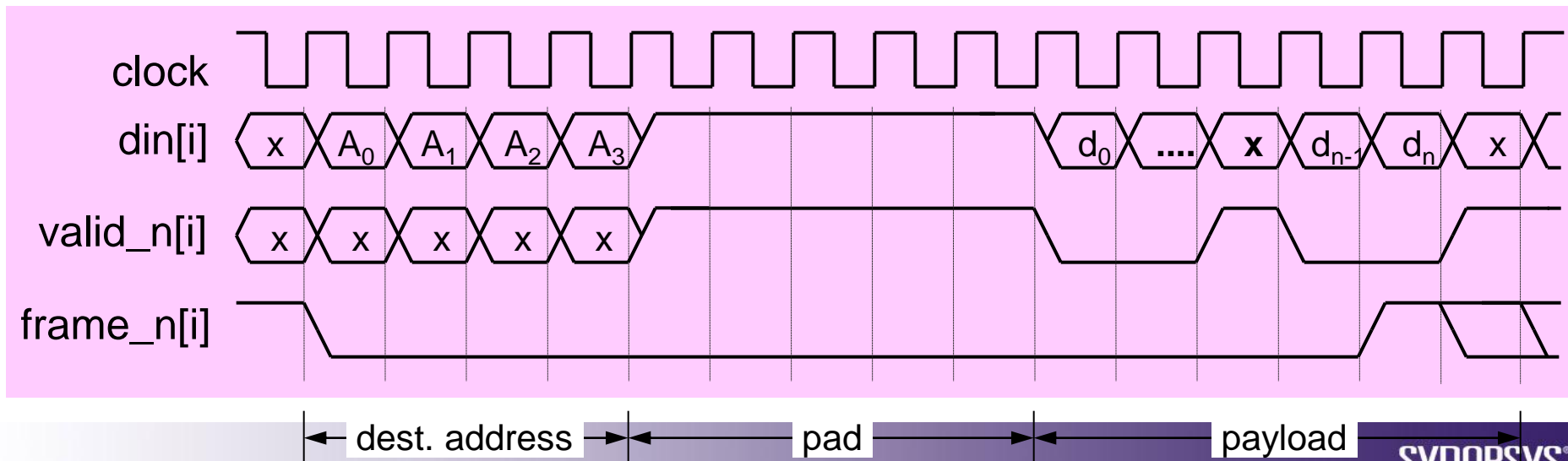


The Router Description

- Single positive-edge clock
- Input and output data are serial (1 bit / clock)
- Packets are sent through in variable length:
 - Each packet is composed of two parts
 - Header
 - Payload
- Packets can be routed from any input port to any output port on a packet-by-packet basis
- No internal buffering or broadcasting (1-to-N)

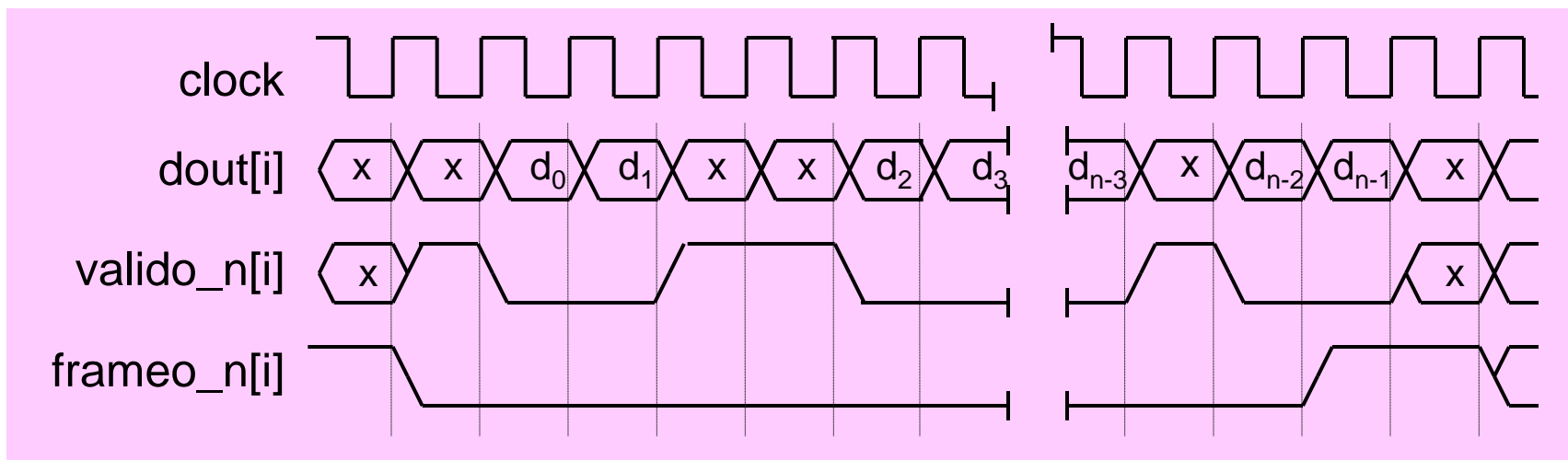
Input Packet Structure

- `frame_n`:
 - Falling edge indicates first bit of packet
 - Rising edge indicates last bit of packet
- `din`:
 - Header (destination address & padding bits) and payload
- `valid_n`:
 - `valid_n` is low if payload bit is valid, high otherwise



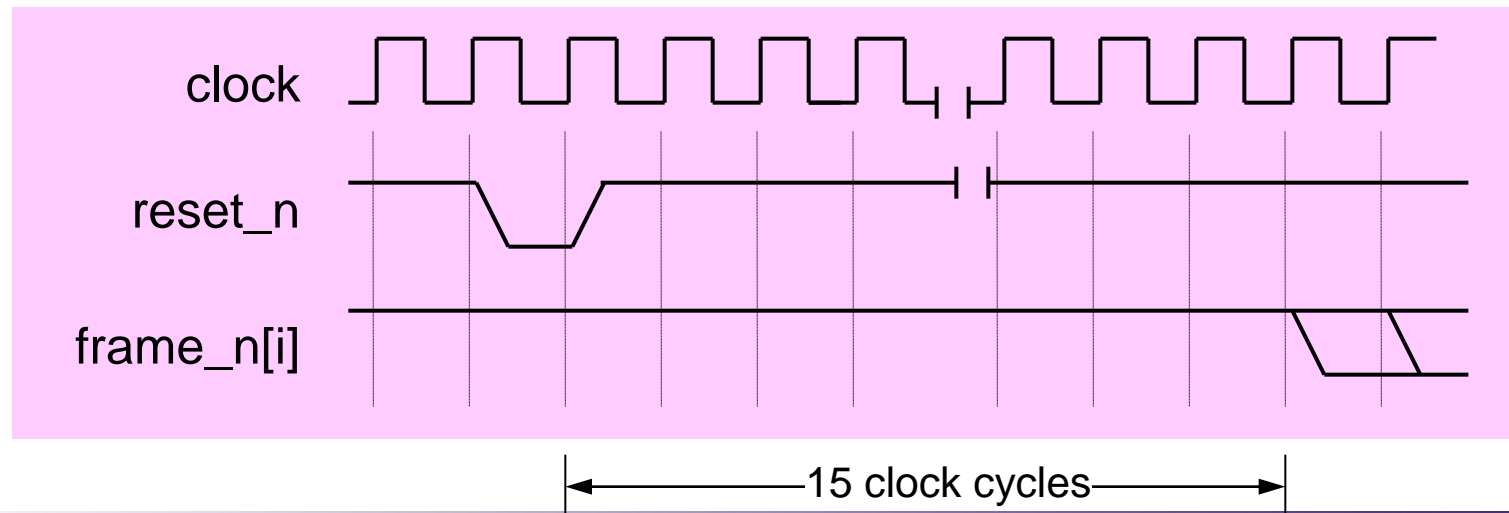
Output Packet Structure

- Output activity is indicated by: `frameo_n`, `valido_n`, and `dout`
- Data is valid only when:
 - `frameo_n` output is low (except for last bit)
 - `valido_n` output is low
- Header field is stripped



Reset Signal

- While asserting `reset_n`, `frame_n` and `valid_n` must be de-asserted
- `reset_n` is asserted for at least one clock cycle
- After de-asserting `reset_n`, wait for 15 clocks before sending a packet through the router





Lab Time!

- We will finish with a 30 min lab to test the basics.
- Individual work.
- When finish you check out:
 - `$VCS_HOME/doc/examples/testbench/sv/tutorial`
- We will be around for at least 60 min for questions.