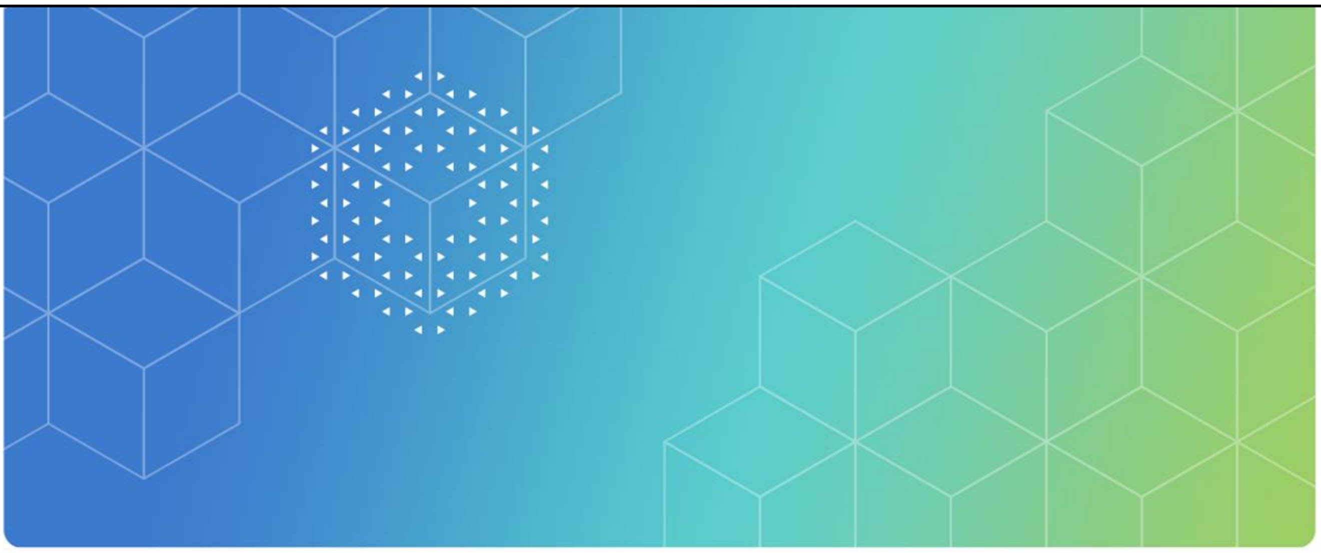
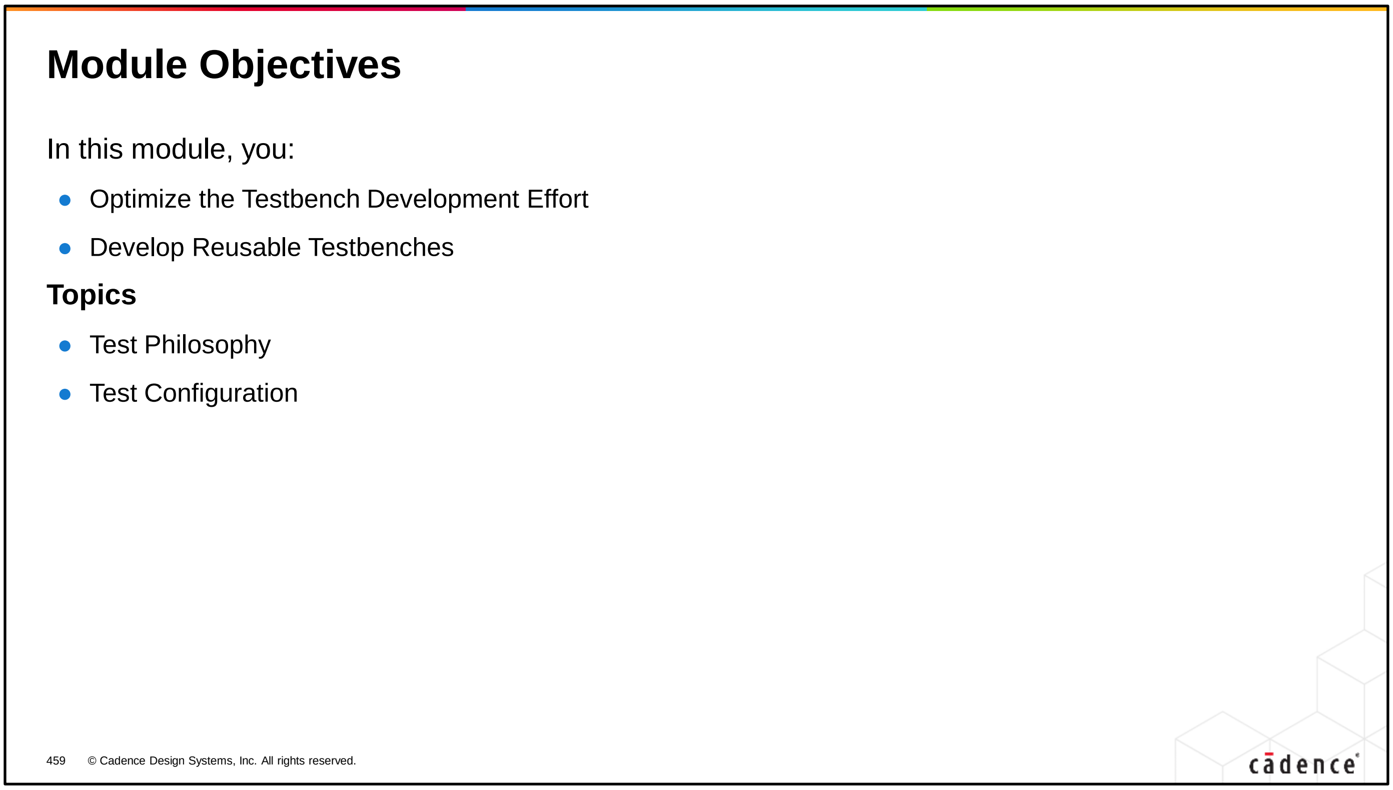


Module 23

Developing a Testbench

This module presents design verification concepts, reinforced with small working examples and illustrations.



Module Objectives

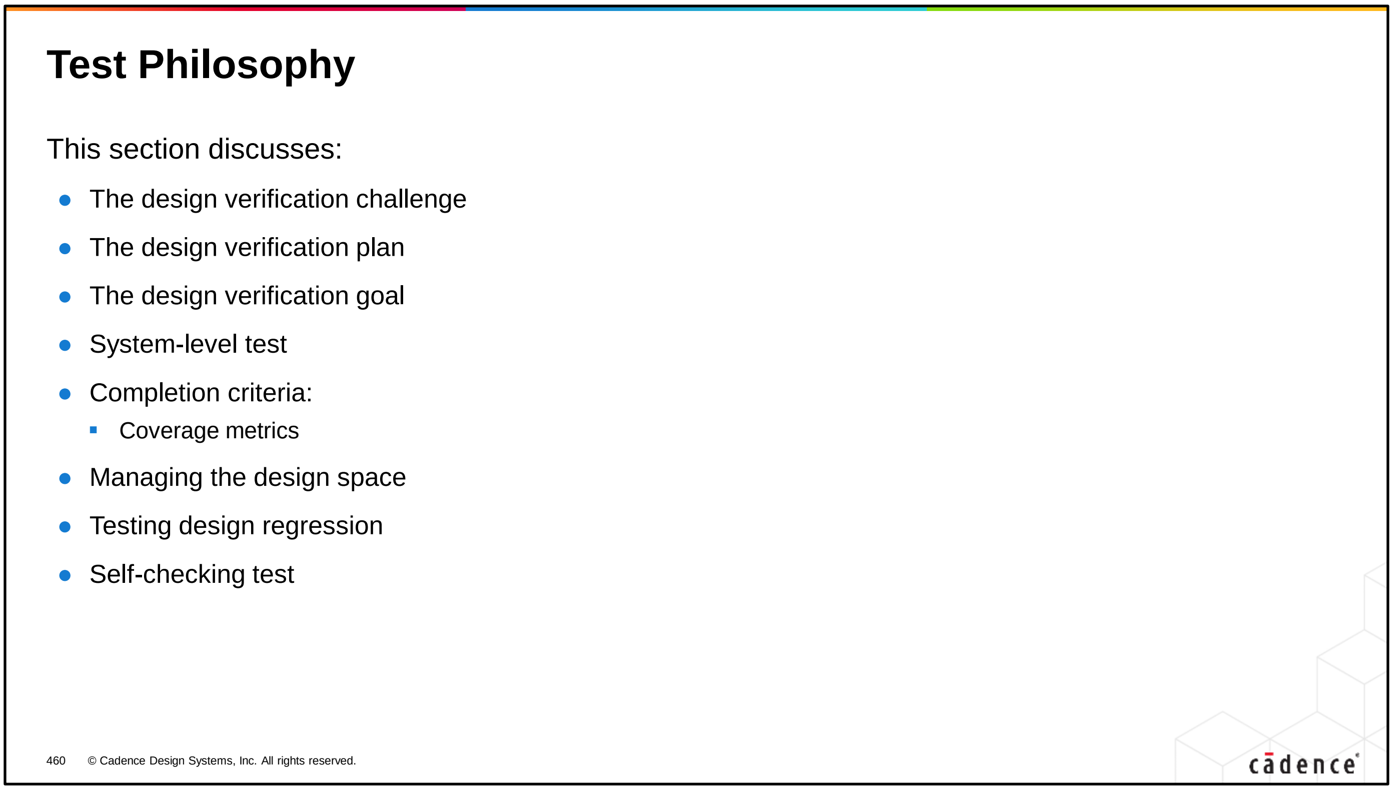
In this module, you:

- Optimize the Testbench Development Effort
- Develop Reusable Testbenches

Topics

- Test Philosophy
- Test Configuration

- Your first objective is to optimize the testbench development effort. The first part of this module supports that objective by exploring the philosophy of system-level test, including the test plan, completion criteria, and design data management.
- Your second objective is to develop reusable testbenches. The second part of this module supports that objective by presenting self-checking tests and various ways to configure the test environment.

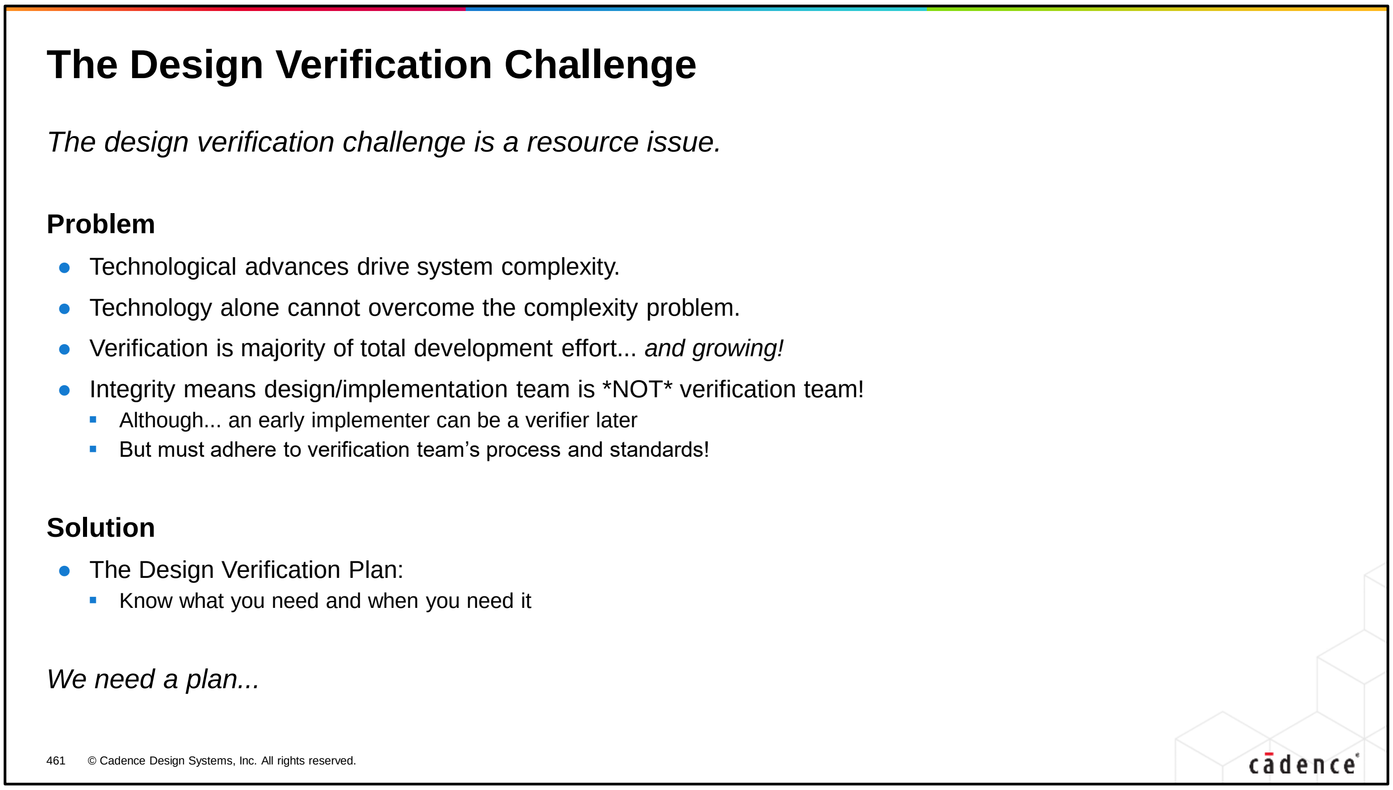


Test Philosophy

This section discusses:

- The design verification challenge
- The design verification plan
- The design verification goal
- System-level test
- Completion criteria:
  - Coverage metrics
- Managing the design space
- Testing design regression
- Self-checking test

This section addresses several aspects of test philosophy.



The Design Verification Challenge

The design verification challenge is a resource issue.

Problem

- Technological advances drive system complexity.
- Technology alone cannot overcome the complexity problem.
- Verification is majority of total development effort... *and growing!*
- Integrity means design/implementation team is *NOT* verification team!
  - Although... an early implementer can be a verifier later
  - But must adhere to verification team's process and standards!

Solution

- The Design Verification Plan:
  - Know what you need and when you need it

We need a plan...

461 © Cadence Design Systems, Inc. All rights reserved.

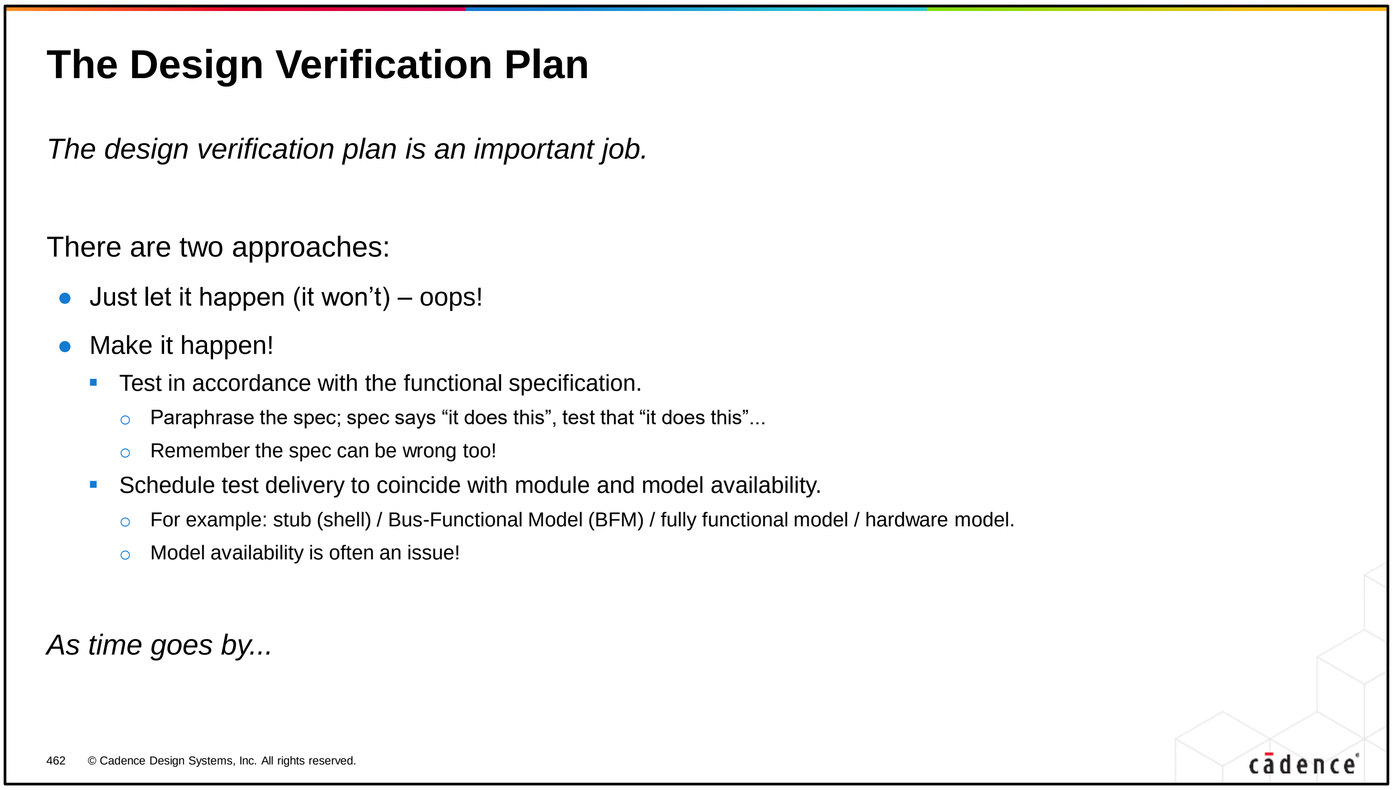
Design verification ensures that the design continues to comply with all system requirements throughout all levels of abstraction, from the system behavioral level to the block implementation level, and continuing through integration of the physical units.

You will need to integrate, structure, and coordinate your verification methods to minimize the probability that the development team misses errors, especially complex system-level design errors, until much later in the design process. Such large-scale redesign is much more costly than iteration confined to a single design phase.

The industry currently uses event-driven simulation as the most common design verification technology at the behavioral and the functional level. This technology is practical, well understood, and mature, but unable to keep up on its own with Moore's law: that design complexity grows at an exponential rate, approximately doubling every 12 to 18 months.

Faster technologies are available, for example, hardware acceleration, cycle-based simulation and emulation, but are less applicable at the higher levels of abstraction early in the design process.

The key to the design verification challenge is to not rely on technological advances alone, but to carefully plan and execute the overall design and design verification strategy.



The Design Verification Plan

The design verification plan is an important job.

There are two approaches:

- Just let it happen (it won't) – oops!
- Make it happen!
  - Test in accordance with the functional specification.
    - Paraphrase the spec; spec says "it does this", test that "it does this"...
    - Remember the spec can be wrong too!
  - Schedule test delivery to coincide with module and model availability.
    - For example: stub (shell) / Bus-Functional Model (BFM) / fully functional model / hardware model.
    - Model availability is often an issue!

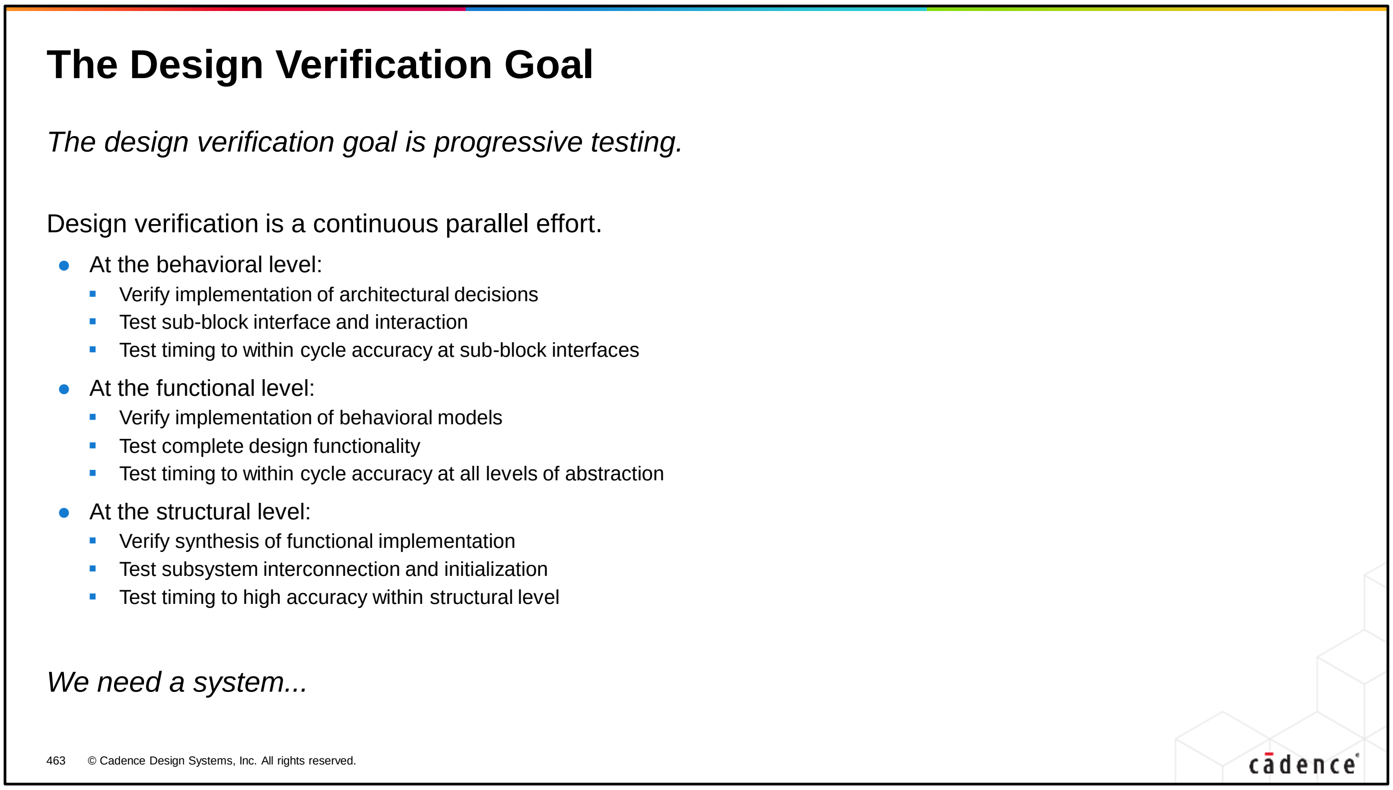
As time goes by...

462 © Cadence Design Systems, Inc. All rights reserved.

cadence

A viable design verification plan must include the following elements:

- 1st – Use of the appropriate technology for each phase of the design process
  - This requires understanding the available technologies, the needs of the project, and the verification strategy. Depending upon the design phase, appropriate technologies can be anything from C-based stochastic analysis to a hardware prototype.
- 2nd – Creation of appropriate, easily-selectable simulation configurations
  - Configurations must be designed to focus verification efforts on key partitions of the design, while filling in the rest of the system context with behavioral models. The number and design of configurations also heavily depends upon the resources available, both of hardware and of personnel.
- 3rd – Development of formal, well-structured testbench templates
  - The use of testbench templates facilitates test generation, maintenance, and modification. Test procedures, test data, and test timing should be separated, and good programming practices, such as parameterization, should be used.
- 4th – Generation of appropriate tests in a timely manner



The Design Verification Goal

The design verification goal is progressive testing.

Design verification is a continuous parallel effort.

- At the behavioral level:
  - Verify implementation of architectural decisions
  - Test sub-block interface and interaction
  - Test timing to within cycle accuracy at sub-block interfaces
- At the functional level:
  - Verify implementation of behavioral models
  - Test complete design functionality
  - Test timing to within cycle accuracy at all levels of abstraction
- At the structural level:
  - Verify synthesis of functional implementation
  - Test subsystem interconnection and initialization
  - Test timing to high accuracy within structural level

We need a system...

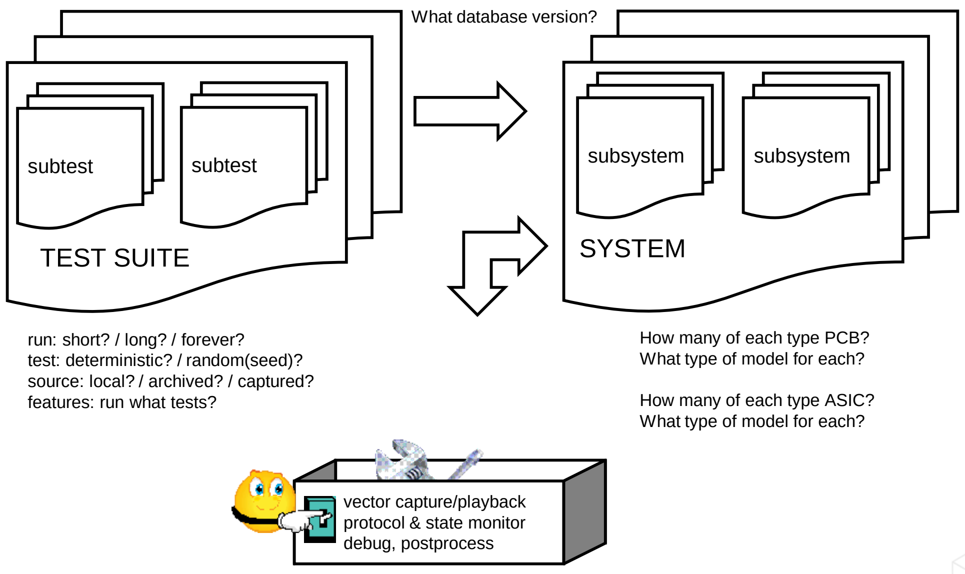
463 © Cadence Design Systems, Inc. All rights reserved.

All design verification strategies have these two overall goals:

- The first goal for high-level system design verification is to validate the design intent before implementation is begun. This involves:
  - Verification of system functionality – that algorithm selection is correct.
  - Verification that the system design meets performance and cost requirements.
  - Verification that subsystem partitioning and packaging, into hardware and software, for example, or between physical components, is practical and effective.
- The second goal is to verify the design implementation. This involves:
  - Verification that all components, whether newly designed, reused, or standard, off-the-shelf, work as expected within the context of the new system-level design.
  - Verification of all internal functionality of newly designed subsystems.
  - Verification of hardware and software interaction.

System-Level Test and Configurability

System-level test is about configurability.


What database version?
subtest
subtest
TEST SUITE
subsystem
subsystem
SYSTEM
run: short? / long? / forever?
test: deterministic? / random(seed)?
source: local? / archived? / captured?
features: run what tests?
How many of each type PCB?
What type of model for each?
How many of each type ASIC?
What type of model for each?
vector capture/playback
protocol & state monitor
debug, postprocess

System-level test is about configurability. A simulation configuration groups specific models of specific subsystems together for a defined verification purpose. The simulation configuration may also constrain the type, detail, and length of tests, or you may wish to postpone such test selection until runtime.

You should include in your testbench:

- Tools for vector capture and playback.
- Monitors of bus protocol, arbiter fairness and state machine transitions.
- Debug tools.
- Postprocessors for the ASIC foundry.

System-Level Test and Team Communication

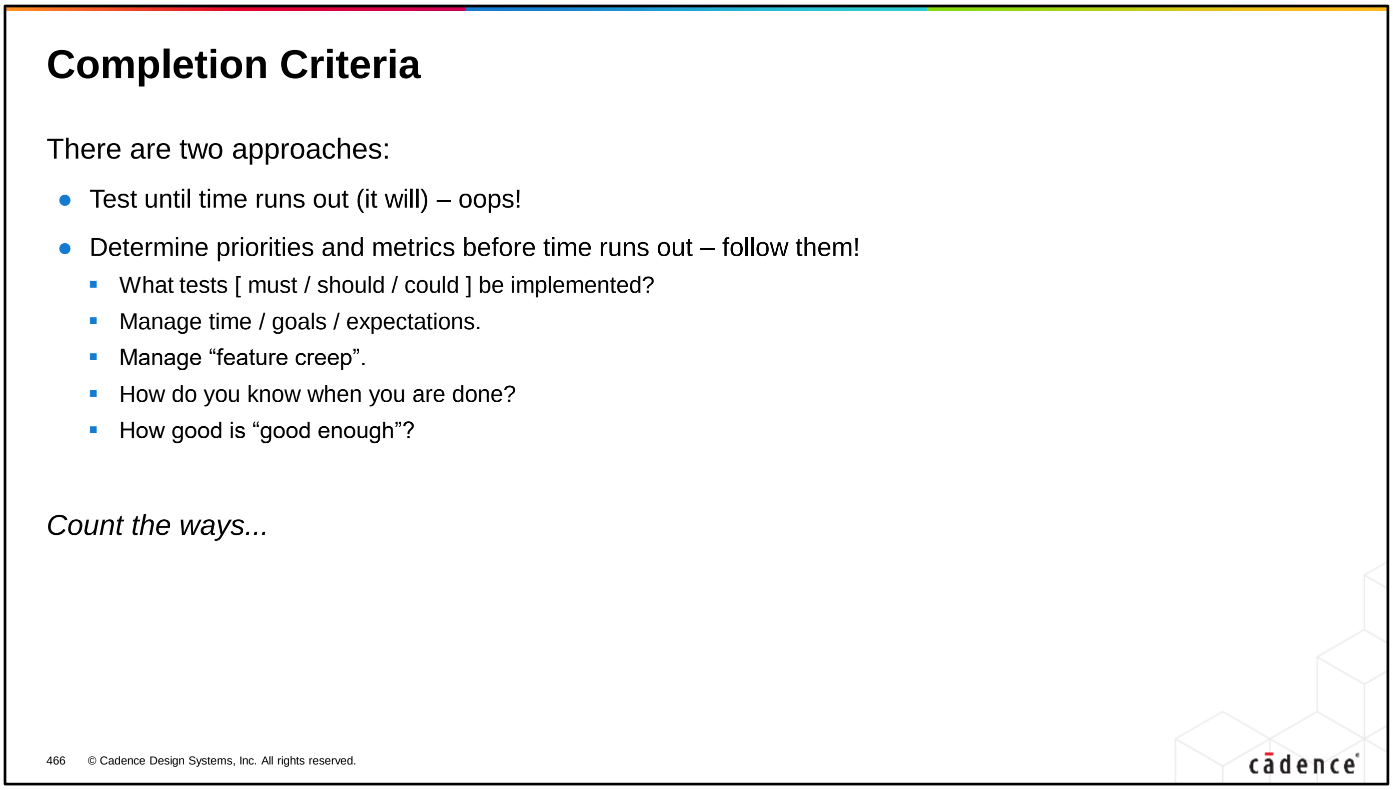
System-level test is about *communication!*

There are two approaches:

- System-level test:
  - Developed in parallel with hardware (tested "as a whole" from day 1)
  - Used (with reconfiguration) by everybody
  - Demands up-front planning and continued maintenance
  - Facilitates communication between designers and verifiers
  - Validates the verifier's interpretation of the specifications
  - Enforces a common test framework, strategy, and style
  - Conserves resources due to less duplication of effort
- Design-unit test:
  - Owned by the designer/implementer
  - "Ad hoc", quick and dirty, typically not communicated or maintained
  - Can be useful for preliminary "sanity" check

Ok, but when am I done?

System-level test is about communication. The vast majority of the design errors that persist beyond a single phase of the design cycle do so, not because the information required to prevent them is not known to the team at large, but because the information is not known to the persons responsible for the piece of the design containing the error. System-level test is a formidable ally to the design team in their constant war against miscommunication or lack of communication.



Completion Criteria

There are two approaches:

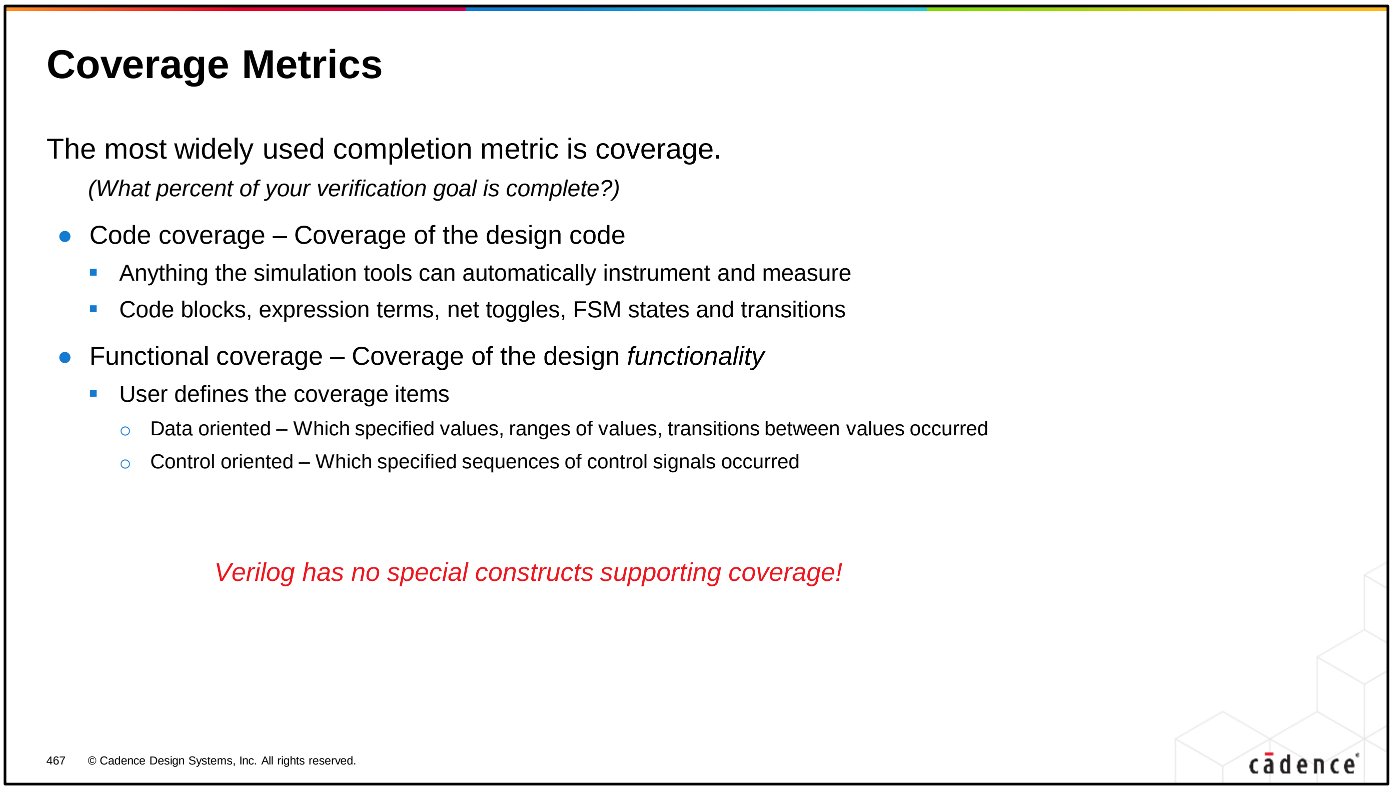
- Test until time runs out (it will) – oops!
- Determine priorities and metrics before time runs out – follow them!
  - What tests [must / should / could] be implemented?
  - Manage time / goals / expectations.
  - Manage “feature creep”.
  - How do you know when you are done?
  - How good is “good enough”?

Count the ways...

466 © Cadence Design Systems, Inc. All rights reserved.

cādence

An important part of the design verifier’s job is to provide meaningful project status information to the management team. The management team needs a reasonably firm test plan, against which they can analyze the project status trend, to do project-related planning. Although the test plan should retain some flexibility throughout the project, the design verifier should also make clear to the management team the time and resources trade-offs associated with any modifications.



Coverage Metrics

The most widely used completion metric is coverage.

(What percent of your verification goal is complete?)

- Code coverage – Coverage of the design code
  - Anything the simulation tools can automatically instrument and measure
  - Code blocks, expression terms, net toggles, FSM states and transitions
- Functional coverage – Coverage of the design *functionality*
  - User defines the coverage items
    - Data oriented – Which specified values, ranges of values, transitions between values occurred
    - Control oriented – Which specified sequences of control signals occurred

Verilog has no special constructs supporting coverage!

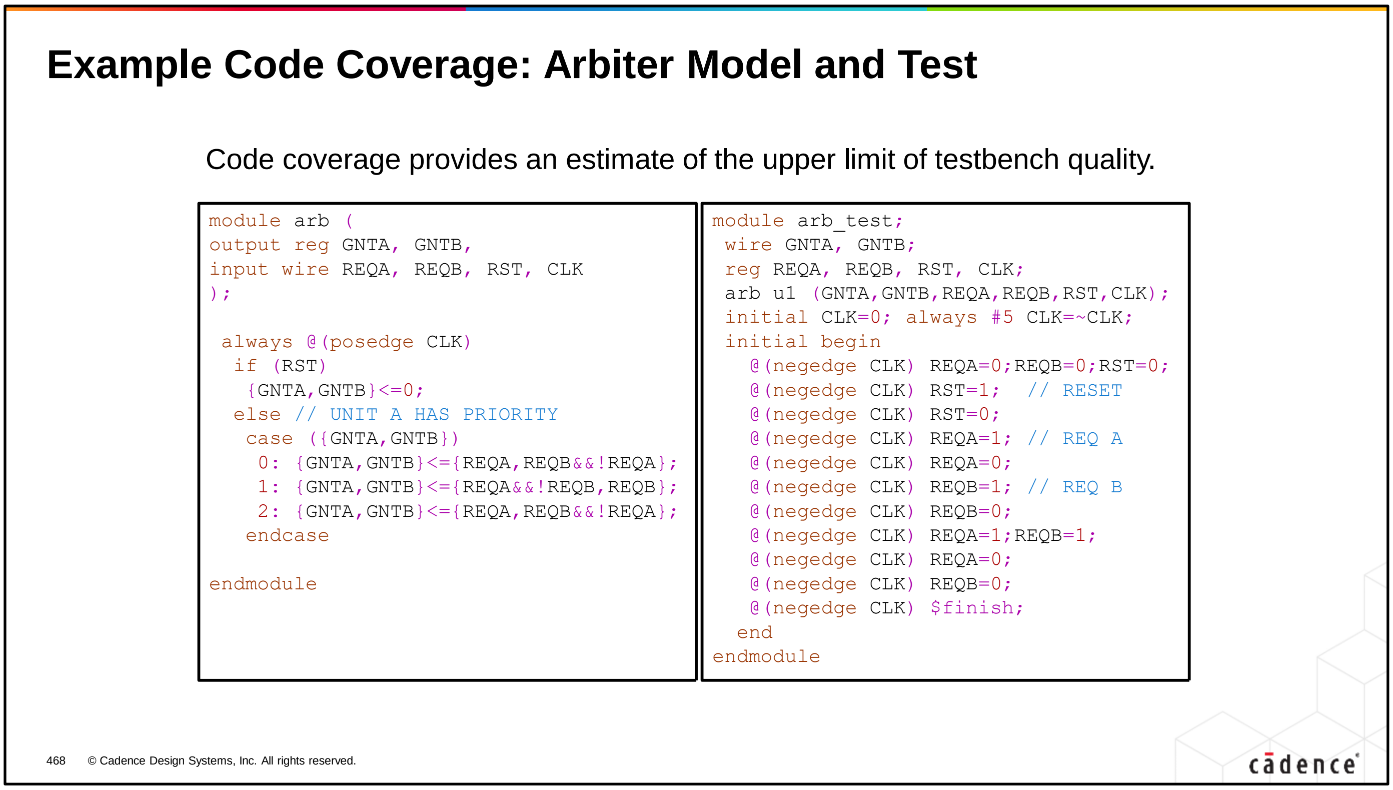
467 © Cadence Design Systems, Inc. All rights reserved.

cādence

The most widely used completion metric is coverage. Coverage is in two forms:

- Code coverage is a measure of how completely the testbench exercises the design code. Code coverage includes any code items that simulation tools can automatically instrument and measure. This might include for example code blocks, expression terms, net toggles, and FSM states and transitions. Code coverage does not say anything about design functionality.
- Functional coverage is a measure of how completely the testbench exercises the design functionality. You the user have to specify this functionality, so the coverage metric is only as valid as the test specification. Functional coverage is further divided into two forms:
  - Data-oriented functional coverage is a measure of which specified values, ranges of values, and transitions between values occurred.
  - Control-oriented functional coverage is a measure of which specified sequences of control signals occurred.

Verilog has no special constructs supporting coverage. For code coverage that the tool automatically instruments this is not an issue. For functional coverage you have to turn to high level verification languages, such as SystemVerilog.



Example Code Coverage: Arbiter Model and Test

Code coverage provides an estimate of the upper limit of testbench quality.

```
module arb (
output reg GNTA, GNTB,
input wire REQA, REQB, RST, CLK
);

 always @(posedge CLK)
  if (RST)
   {GNTA,GNTB}<=0;
  else // UNIT A HAS PRIORITY
   case ({GNTA,GNTB})
    0: {GNTA,GNTB}<={REQA,REQB&&!REQA};
    1: {GNTA,GNTB}<={REQA&&!REQB,REQB};
    2: {GNTA,GNTB}<={REQA,REQB&&!REQA};
   endcase

endmodule
```

```
module arb_test;
 wire GNTA, GNTB;
 reg REQA, REQB, RST, CLK;
 arb u1 (GNTA,GNTB,REQA,REQB,RST,CLK);
 initial CLK=0; always #5 CLK=~CLK;
 initial begin
   @(negedge CLK) REQA=0;REQB=0;RST=0;
   @(negedge CLK) RST=1;  // RESET
   @(negedge CLK) RST=0;
   @(negedge CLK) REQA=1; // REQ A
   @(negedge CLK) REQA=0;
   @(negedge CLK) REQB=1; // REQ B
   @(negedge CLK) REQB=0;
   @(negedge CLK) REQA=1;REQB=1;
   @(negedge CLK) REQA=0;
   @(negedge CLK) REQB=0;
   @(negedge CLK) $finish;
  end
endmodule
```

Code coverage provides an estimate of the upper limit of testbench quality. Block coverage, for example, indicates whether the covered code has been executed, and says nothing about how correct the code is. Code coverage is nonetheless very useful, for with fairly simple technology it can pinpoint an area of the design that the testbench has completely missed.

This arbiter simply responds with a grant to every request. The grant persists until the request drops. The arbiter gives priority to unit A if both units simultaneously make a request.

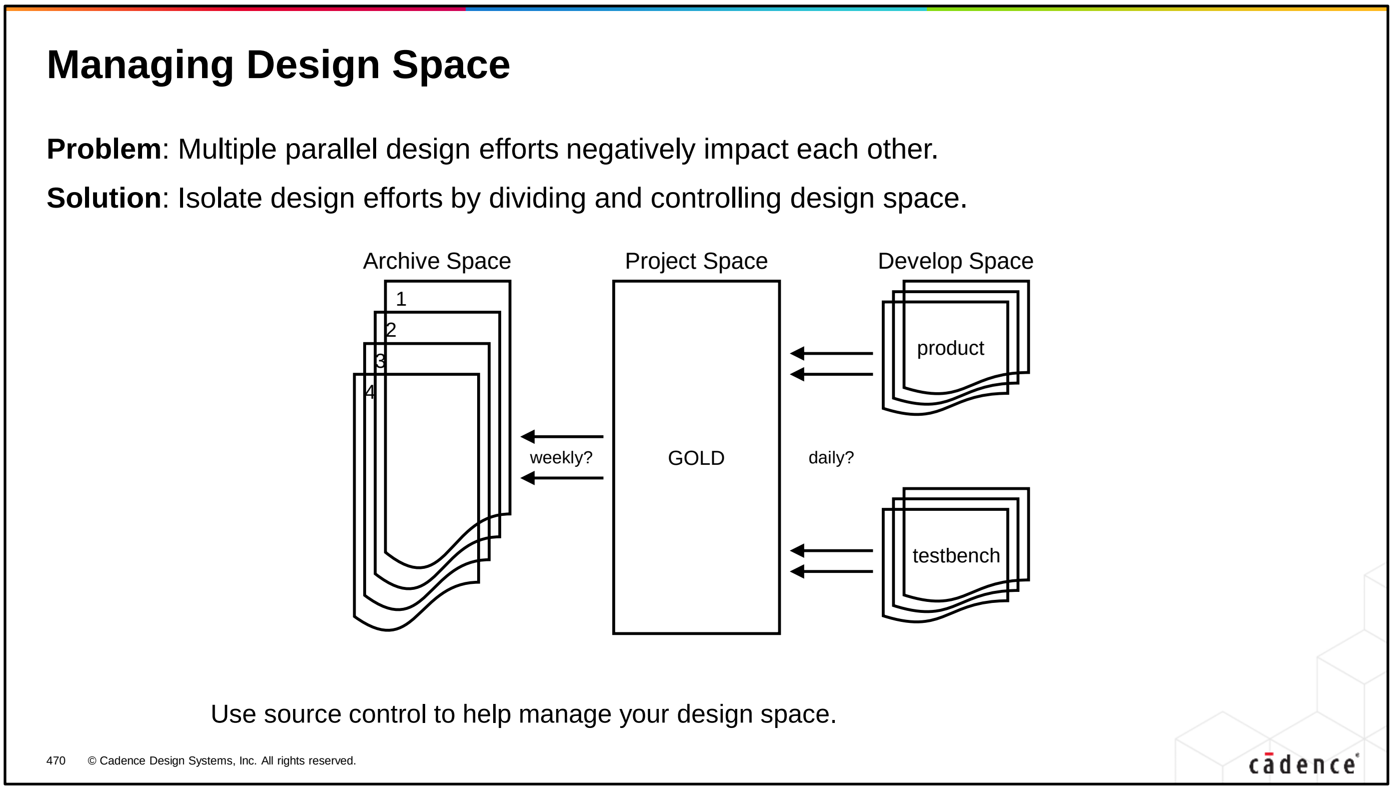
This short testbench resets the arbiter and then separately tests that it grants a unit A request and then a unit B request, and that upon simultaneous arrival of both requests, it grants the unit A request first.

Example Code Coverage – Block Coverage Report

```
<><><><><><><><><><><><><><><><><><><><><><><><><><><>
<> COVERAGE FILE: .../cov_work/design/test/icc.ucd  <>
<><><><><><><><><><><><><><><><><><><><><><><><><><><>
hit blk:  minimum of all scoring points on that source line
        Module name: arb;
        File name: .../test.v
line no    hit blk
----------------------------------------------
                        // .../test.v
     1                  module arb (
     2                  output reg GNTA, GNTB,
     3                  input wire REQA, REQB, RST, CLK
     4                  );
     5
     6                   always @(posedge CLK)
     7       1            if (RST)
     8       1             {GNTA,GNTB}<=0;
     9                    else // UNIT A HAS PRIORITY
    10       1             case ({GNTA,GNTB})
    11       1              0: {GNTA,GNTB}<={REQA,REQB&&!REQA};
    12       1              1: {GNTA,GNTB}<={REQA&&!REQB,REQB};
    13       1              2: {GNTA,GNTB}<={REQA,REQB&&!REQA};
    14                     endcase
    15
    16                  endmodule
```

This is annotated block coverage reported by the Cadence Xcelium™ Comprehensive Coverage analysis tool. The author edited it slightly to reduce long file names and remove unneeded white space to fit it on the page. A code block is a block of code that executes together – either none of it executes or all of it executes. The tool measures block coverage instead of line or statement coverage because Verilog is a free-form language, so line counts and statement counts are meaningless. You can see that the testbench caused each block to execute.

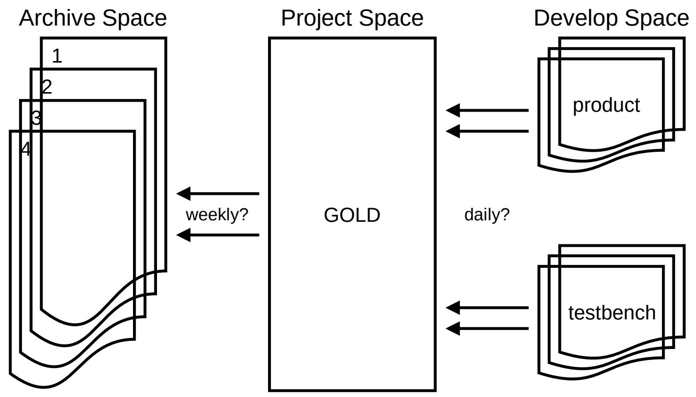
You can easily understand the block coverage reports, so you should start your code coverage analysis with block coverage. Upon achieving 100% block coverage, as this testbench did, you should then analyze the expression coverage. Expression coverage would reveal that this testbench did not test that a grant that is asserted will stay asserted as long as a request stays asserted.



Managing Design Space

Problem: Multiple parallel design efforts negatively impact each other.

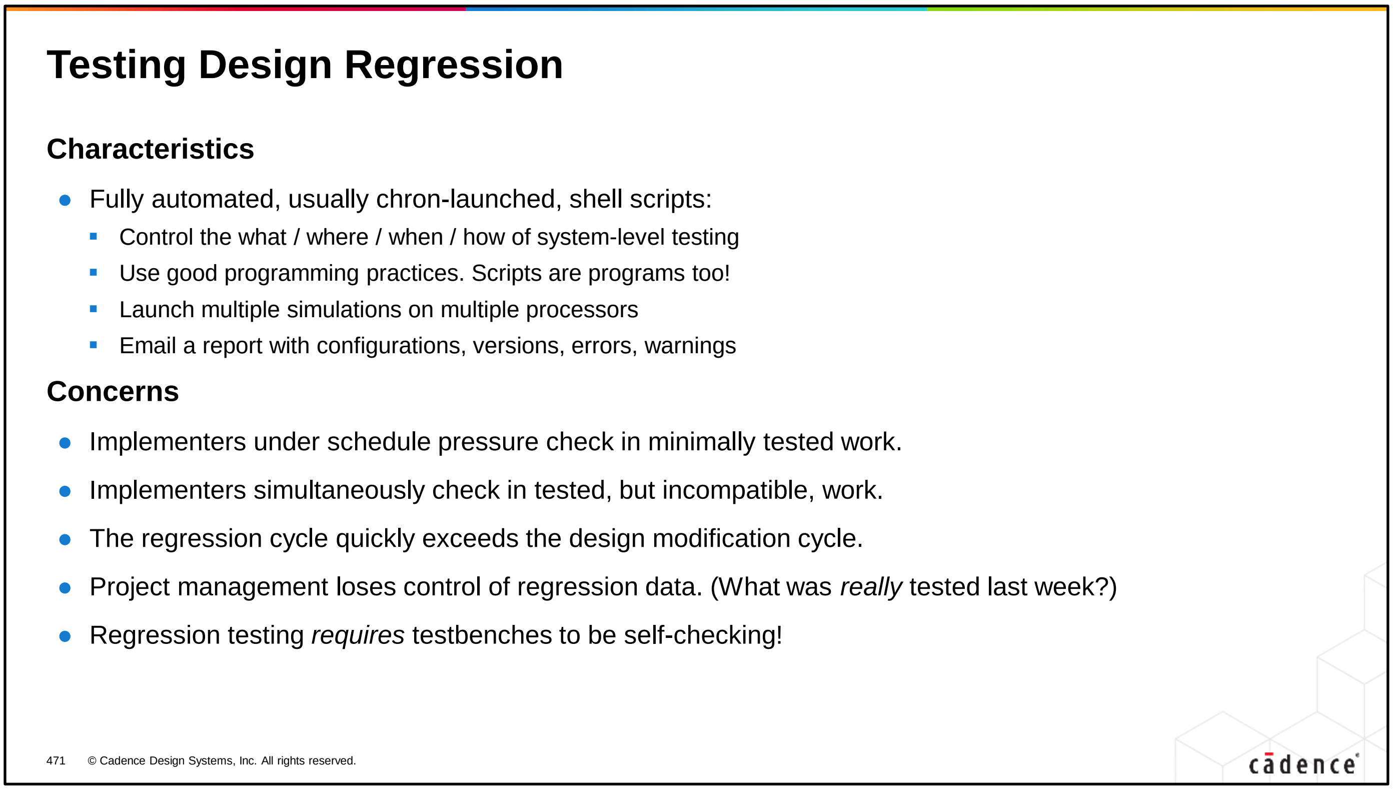
Solution: Isolate design efforts by dividing and controlling design space.



Use source control to help manage your design space.

Partition your design space into the following areas:

- 1st – The development space:
  - Copy elements to this private area for your individual design efforts;
  - Use the “golden” project space for all other design elements; and
  - Replace project space design elements only after thoroughly verifying them.
- 2nd – The project space:
  - Tag the in-progress elements to prevent duplication of design efforts;
  - Submit only thoroughly verified design elements; and
  - Continuously run regression tests.
- 3rd – The archive space:
  - Back up the project space here on a regular basis or upon completion of major features;
  - Continuously run the full test suite, incorporating random test sequences; and
  - Use this space for investigatory efforts such as timing analysis, fault analysis and so on...



Testing Design Regression

Characteristics

- Fully automated, usually chron-launched, shell scripts:
  - Control the what / where / when / how of system-level testing
  - Use good programming practices. Scripts are programs too!
  - Launch multiple simulations on multiple processors
  - Email a report with configurations, versions, errors, warnings

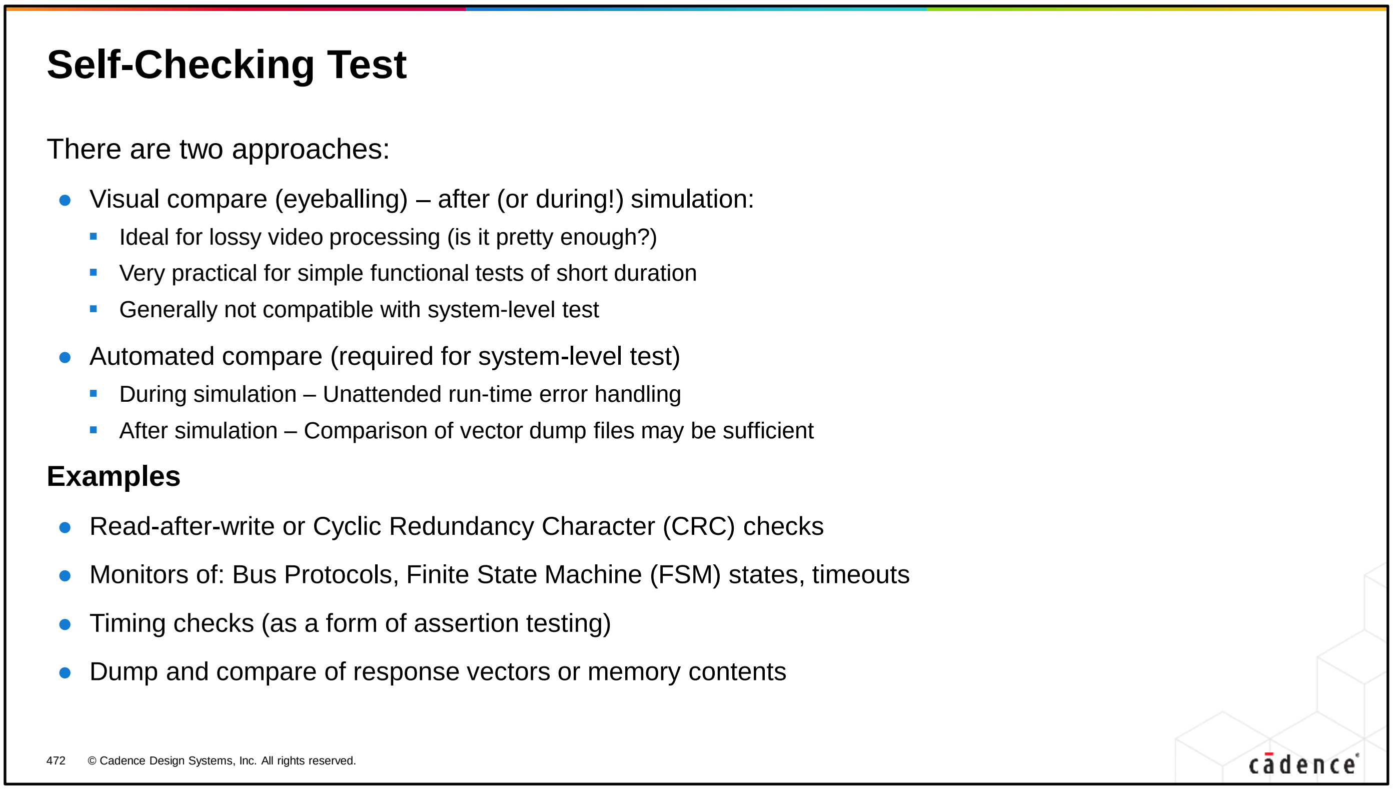
Concerns

- Implementers under schedule pressure check in minimally tested work.
- Implementers simultaneously check in tested, but incompatible, work.
- The regression cycle quickly exceeds the design modification cycle.
- Project management loses control of regression data. (What was *really* tested last week?)
- Regression testing *requires* testbenches to be self-checking!

You will want to continually in the background verify that addition of new features or modification of previous features does not cause test failure in the project space. This effort is termed “regression testing” or more confusingly just “regressions”. What this effort does is to detect the “regression” of the project to a less mature state.

Regression tests are usually fully automated chron-launched shell scripts that run multiple simulations on multiple processors, logging the tested configurations and versions, and emailing notification of any anomalies.

Regression testing of course requires that your testbenches be self-checking!



Self-Checking Test

There are two approaches:

- Visual compare (eyeballing) – after (or during!) simulation:
  - Ideal for lossy video processing (is it pretty enough?)
  - Very practical for simple functional tests of short duration
  - Generally not compatible with system-level test
- Automated compare (required for system-level test)
  - During simulation – Unattended run-time error handling
  - After simulation – Comparison of vector dump files may be sufficient

Examples

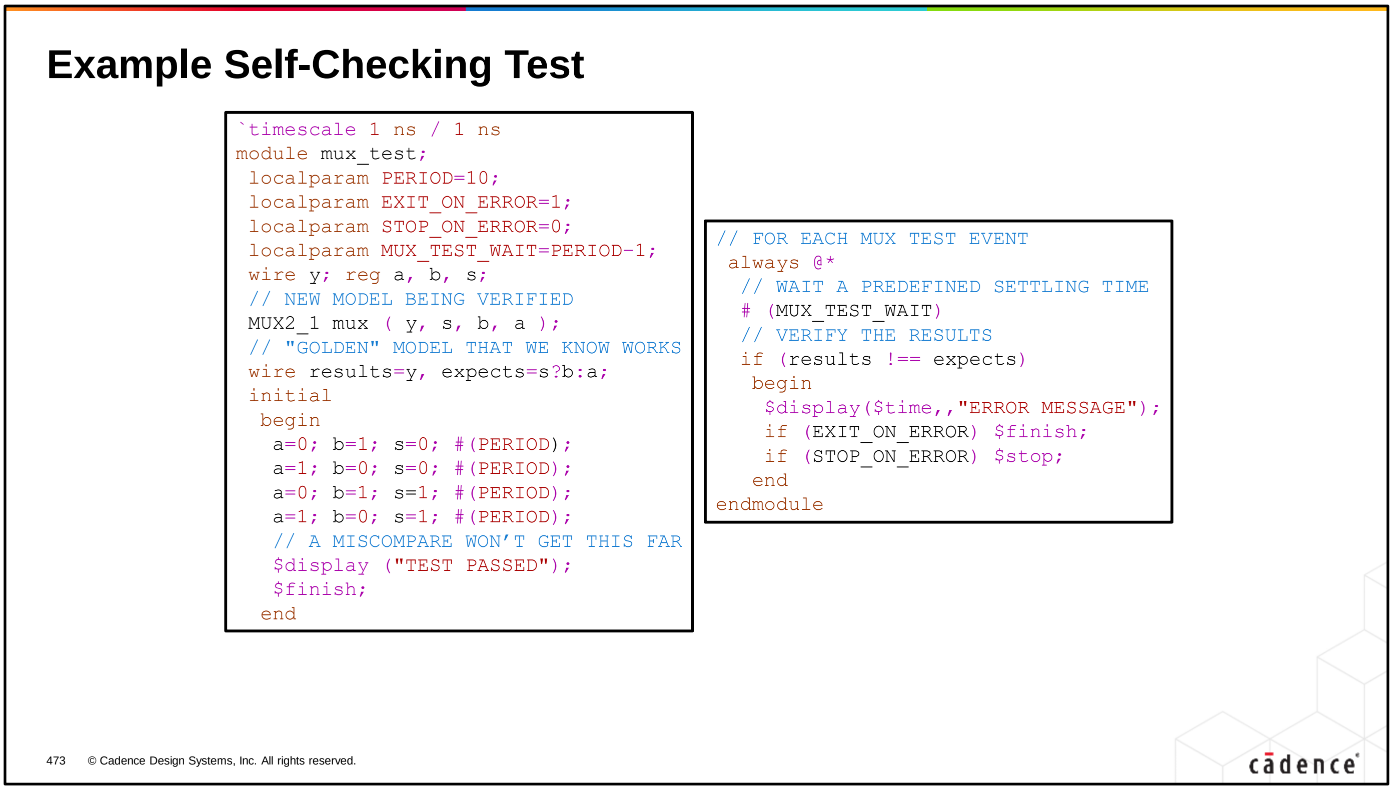
- Read-after-write or Cyclic Redundancy Character (CRC) checks
- Monitors of: Bus Protocols, Finite State Machine (FSM) states, timeouts
- Timing checks (as a form of assertion testing)
- Dump and compare of response vectors or memory contents

Testbench self-checking is in two forms:

- The most common form is to check data and perhaps even protocols on the fly in order to provide immediate diagnostic information pointing to the area and time of the problem.
- A second less-common form is to automate a post-processing comparison of dumped test vectors to those of a known-good design. This form may be useful if you want a very detailed check of functionality that is very unlikely to have changed.

Your testbench can be self-checking in many ways:

- You can read back data that you write to storage devices.
- You can compare check characters calculated on the transmission side and the reception side.
- You can code monitors for bus protocols, FSM state paths, and timeouts.
- You can imaginatively set up timing checks to check the timing relationship between two edges of the same or different signals.



Example Self-Checking Test

```
`timescale 1 ns / 1 ns
module mux_test;
 localparam PERIOD=10;
 localparam EXIT_ON_ERROR=1;
 localparam STOP_ON_ERROR=0;
 localparam MUX_TEST_WAIT=PERIOD-1;
 wire y; reg a, b, s;
 // NEW MODEL BEING VERIFIED
 MUX2_1 mux ( y, s, b, a );
 // "GOLDEN" MODEL THAT WE KNOW WORKS
 wire results=y, expects=s?b:a;
 initial
  begin
   a=0; b=1; s=0; #(PERIOD);
   a=1; b=0; s=0; #(PERIOD);
   a=0; b=1; s=1; #(PERIOD);
   a=1; b=0; s=1; #(PERIOD);
   // A MISCOMPARE WON'T GET THIS FAR
   $display ("TEST PASSED");
   $finish;
  end
```

```
// FOR EACH MUX TEST EVENT
 always @*
  // WAIT A PREDEFINED SETTLING TIME
  # (MUX_TEST_WAIT)
  // VERIFY THE RESULTS
  if (results !== expects)
   begin
    $display($time,,"ERROR MESSAGE");
    if (EXIT_ON_ERROR) $finish;
    if (STOP_ON_ERROR) $stop;
   end
endmodule
```

You first generate “expect” patterns using a model that you know works, such as:

- Your own knowledge of what the response should be at any given time.
- When modifying a model, the working model prior to modification.
- When generating a model of lower abstraction, the existing model of higher abstraction.

You then compare the model response to your previously generated “expect” patterns.

This example compares the results to “expect” patterns that a second model at a higher level of abstraction simultaneously generates. The known model of higher abstraction is a conditional expression.

A self-checking test may at first, before you fix it, falsely indicate miscomparisons:

- Perhaps due to different data types, especially if comparing data between two different simulators.
- Perhaps due to comparing data at different or unstable points in the clock cycle, especially when comparing a model with estimated timing to a simpler unit-delay model.
- Perhaps due to comparing data after different initialization sequences, as behavioral models often skip the prolonged initialization sequences the RTL and gate-level models must exercise.
- Perhaps due to comparing unknown or “don’t-care” data values.

Be careful to not confuse the logical equality (==) and case equality (===) operators. The logical equality operator produces an unknown result when comparing a high impedance or unknown state. The case equality operator always produces either a 1 or 0 result when comparing any two operands.

Example Self-Checking Test Alternative

```
`timescale 1 ns / 1 ns
module mux_test;
 localparam PERIOD=10;
 localparam EXIT_ON_ERROR=1;
 localparam STOP_ON_ERROR=0;
 wire y; reg a, b, s;

 // MODEL BEING VERIFIED
 MUX2_1 mux ( y, s, b, a );

 // VERIFY THE RESULTS
 task expect(input e);
  if (y !== e)
   begin
    $display($time,,"ERROR MESSAGE");
    if (EXIT_ON_ERROR) $finish;
    if (STOP_ON_ERROR) $stop;
   end
 endtask
```

```
// TEST THE MODEL
 initial
  begin
   {s,b,a}=3'b010; #(PERIOD) expect(1'b0);
   {s,b,a}=3'b001; #(PERIOD) expect(1'b1);
   {s,b,a}=3'b110; #(PERIOD) expect(1'b1);
   {s,b,a}=3'b101; #(PERIOD) expect(1'b0);
   // A MISCOMPARE WON'T GET THIS FAR
   $display ("TEST PASSED");
   $finish;
  end
endmodule
```

This example utilizes the same model and test as previously, but instead of comparing the model results to those of a “golden” model, periodically calls a task to verify the model results.

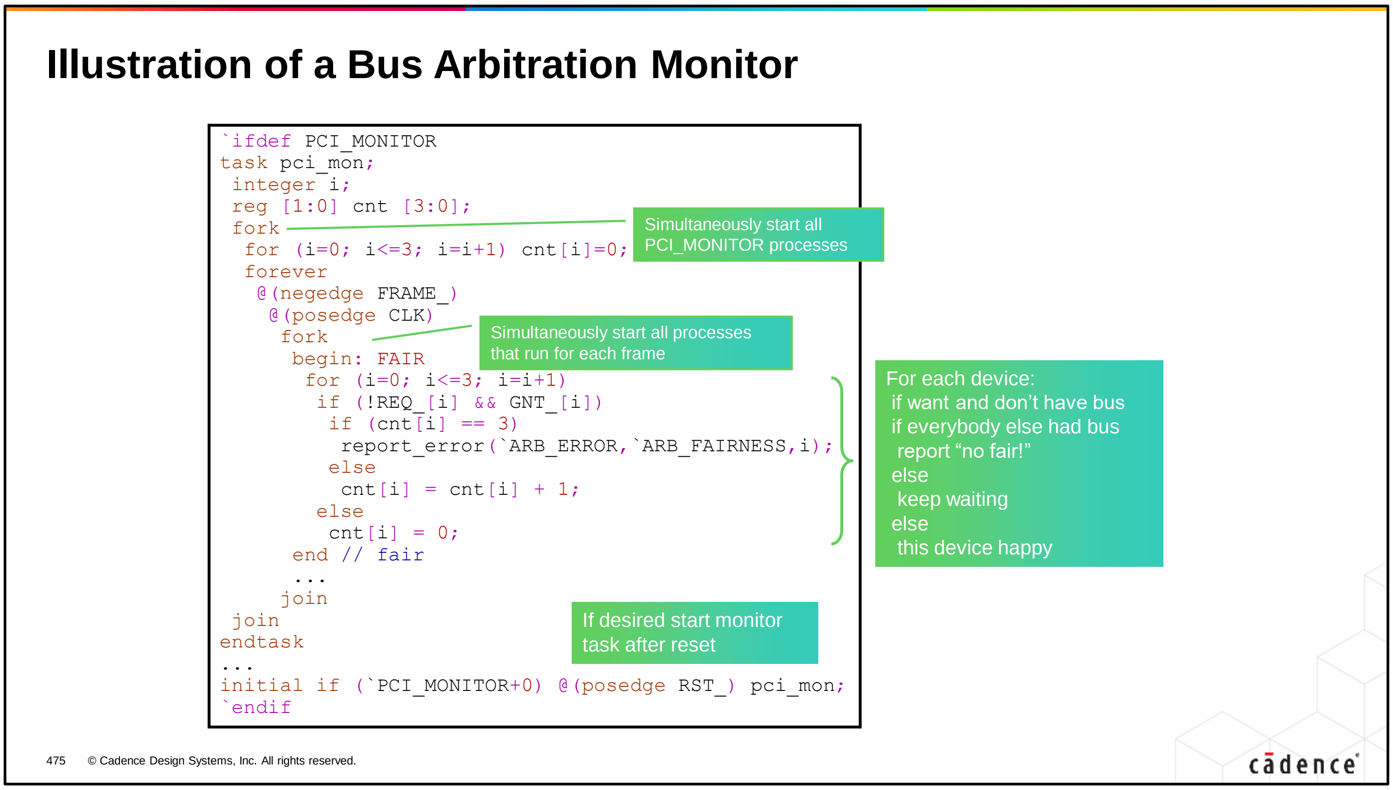


Illustration of a Bus Arbitration Monitor

```
`ifdef PCI_MONITOR
task pci_mon;
 integer i;
 reg [1:0] cnt [3:0];
 fork
  for (i=0; i<=3; i=i+1) cnt[i]=0;
  forever
   @(negedge FRAME_)
    @(posedge CLK)
     fork
      begin: FAIR
       for (i=0; i<=3; i=i+1)
        if (!REQ_[i] && GNT_[i])
         if (cnt[i] == 3)
          report_error(`ARB_ERROR,`ARB_FAIRNESS,i);
         else
          cnt[i] = cnt[i] + 1;
        else
         cnt[i] = 0;
      end // fair
      ...
     join
 join
endtask
...
initial if (`PCI_MONITOR+0) @(posedge RST_) pci_mon;
`endif
```

Simultaneously start all PCI_MONITOR processes

Simultaneously start all processes that run for each frame

For each device:
 if want and don't have bus
 if everybody else had bus
  report "no fair!"
 else
  keep waiting
 else
  this device happy

If desired start monitor task after reset

475 © Cadence Design Systems, Inc. All rights reserved.

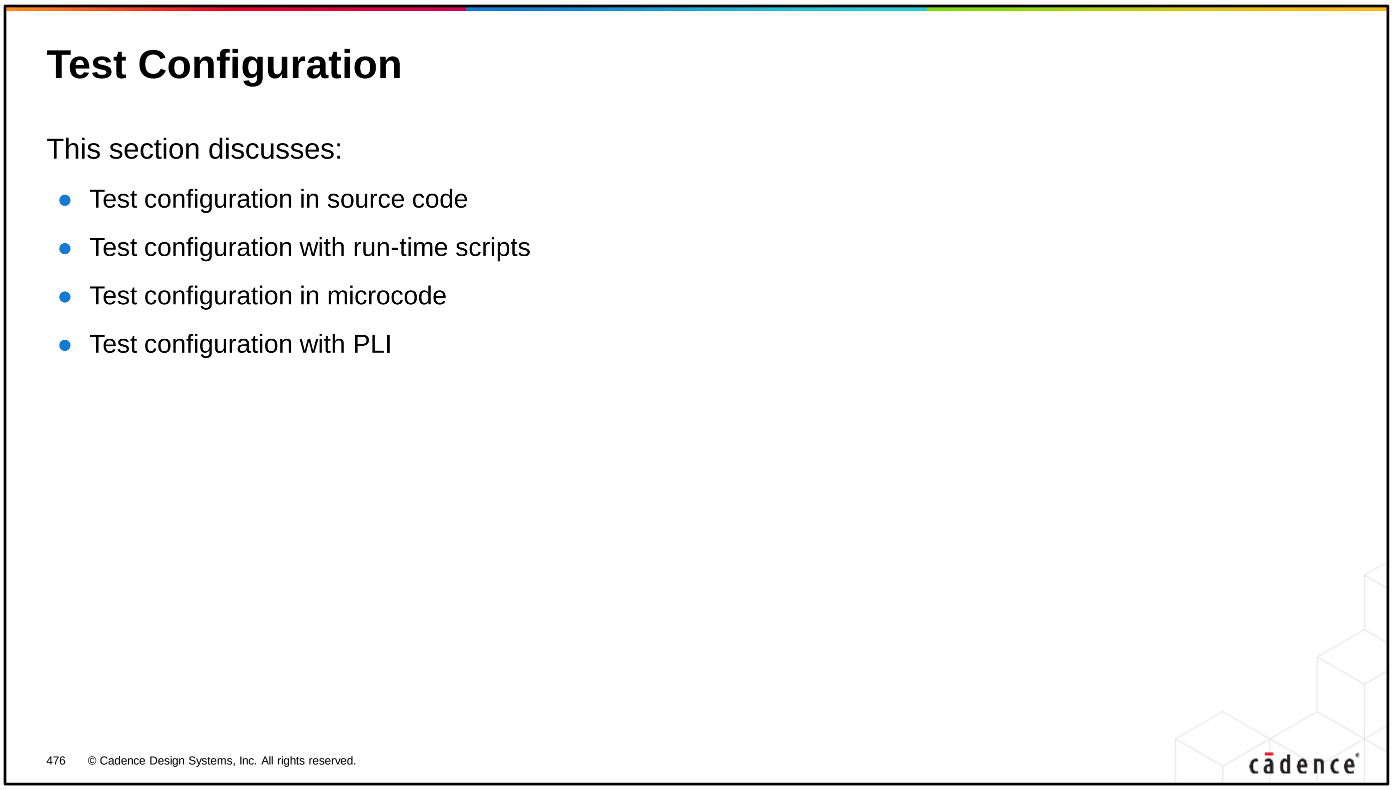
You can write a Verilog task to simultaneously monitor several bus protocol rules. Some examples of such Peripheral Component Interface (PCI) bus rules might be:

- That there are no bus contentions.
- That no master parks itself on the bus.
- That the bus does not become deadlocked.
- That a target lock is gained, held, and released in accordance with locking rules.

You can optionally configure the testbench to activate this "pci_mon" task upon initialization. The task can immediately fork independent, concurrent processes, only one of which is shown, that trigger on different sets of events. The process that activates upon the FRAME_ signal becoming true can again fork independent, concurrent processes, only one of which is shown. Finish this fork-join block before the arrival of the next frame.

The completed portion of the monitor task verifies that no request remains ungranted for longer than is required for all other devices to each acquire the bus once. You can modify the monitor to accommodate other priority schemes.

The monitor reports errors to a central error handler, using codes to represent the error types.



Test Configuration

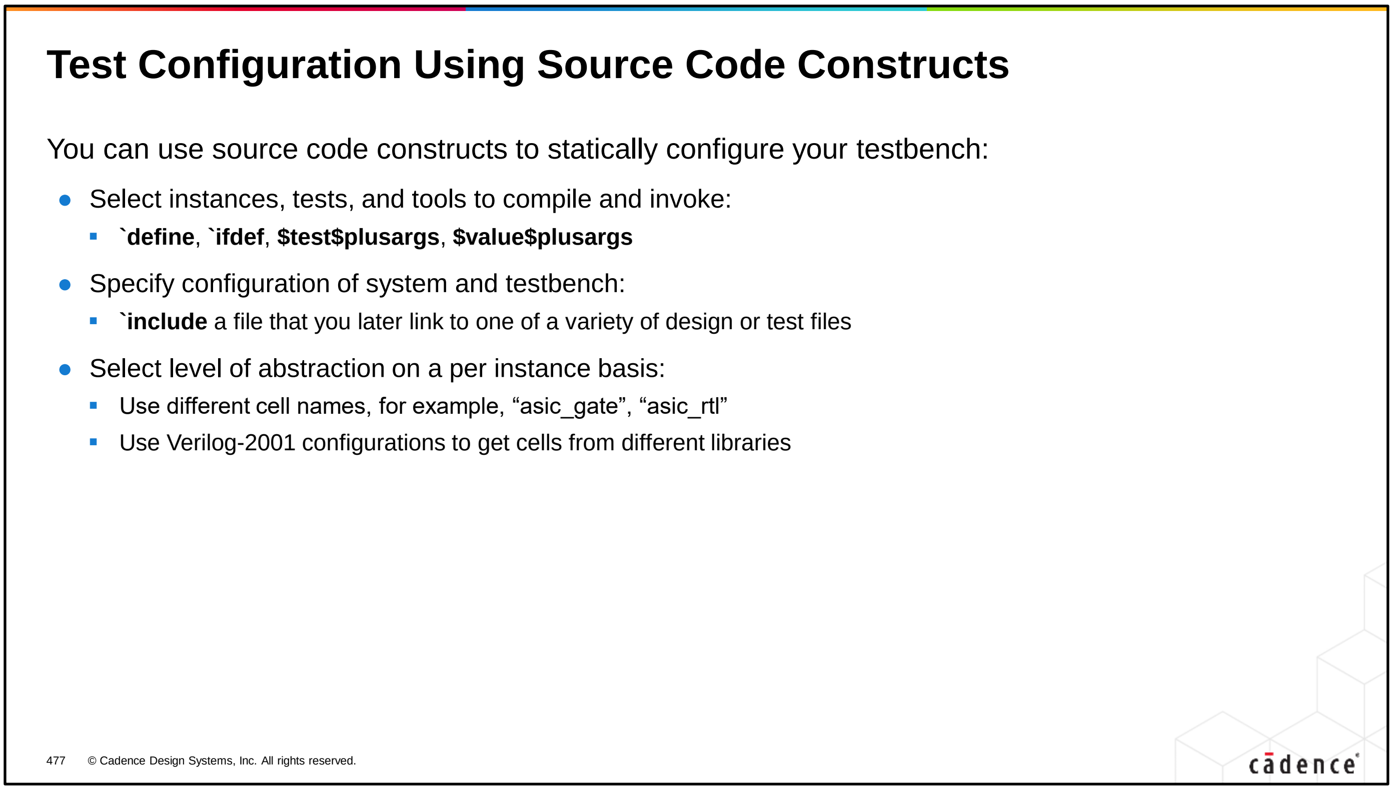
This section discusses:

- Test configuration in source code
- Test configuration with run-time scripts
- Test configuration in microcode
- Test configuration with PLI

476 © Cadence Design Systems, Inc. All rights reserved.

cadence

The following section discusses configuration of the test environment. It discusses test configuration in source code, with run-time scripts, in microcode, and with the Programming Language Interface (PLI).



Test Configuration Using Source Code Constructs

You can use source code constructs to statically configure your testbench:

- Select instances, tests, and tools to compile and invoke:
  - **`define**, **`ifdef**, **$test$plusargs**, **$value$plusargs**
- Specify configuration of system and testbench:
  - **`include** a file that you later link to one of a variety of design or test files
- Select level of abstraction on a per instance basis:
  - Use different cell names, for example, “asic_gate”, “asic_rtl”
  - Use Verilog-2001 configurations to get cells from different libraries

477 © Cadence Design Systems, Inc. All rights reserved.

As you will not be synthesizing the testbench, you can use the full range of Verilog constructs, choosing those that are most efficient or most easily coded for your application.

Some of the most common ways of presenting tests to the system are to put them in source code, run-time scripts, or pattern memory or microcode, perhaps applied via the PLI. Some of the methods you can use to configure the system and testbench in source code are:

- You can use the `define and `ifdef compiler directives to specify portions of the code to be compiled, thus selecting representations of instances, tests to be applied, and debug tools to be compiled into the simulation database.
- You can use the `include directive to include portions of code. The `include directive can itself be selected by the `ifdef directive, and the final resolution of the included file can be changed by modifying file-system links at the operating system level.
- You can use slightly different module names for different representations of any instantiated module, or retrieve the modules from different libraries.

Illustration of Test Configuration in Source Code (1)

```
// sys_config.h
  `define CPU_BASIC
//`define CPU_QUAD
  `define IO_BASIC
//`define IO_FAST
```

```
// system.v
`include "sys_config.h"
module system;
 `ifdef CPU_BASIC
   basic_cpu_pcb cpu(,,);
 `endif
 `ifdef CPU_QUAD
   quad_cpu_pcb cpu(,,);
 `endif
 `ifdef IO_BASIC
   basic_io_pcb io(,,);
 `endif
 `ifdef IO_FAST
   fast_io_pcb io(,,);
 `endif
 ...
endmodule
```

```
// test_config.h
  `define THIS_TEST
  `define THAT_TEST
//`define SOME_TEST
```

```
// test.v
`include "test_config.h"
module test;
 `ifdef THIS_TEST
   `include "this_test.v"
 `endif
 `ifdef THAT_TEST
   `include "that_test.v"
 `endif
 initial
  begin
   `ifdef THIS_TEST
     ;
   `endif
   `ifdef THAT_TEST
     that_test;
   `endif
  end
 ...
endmodule
```

This illustration shows how you can use text macros to configure the system under test. It uses text macros to select the system configuration and test configuration and launch the test tasks.

Illustration of Test Configuration in Source Code (2)

```
`ifdef RUN_FOREVER
  forever
`else
  `ifdef RUN_LONG
    repeat (`MEM_TEST_LONG)
  `else
    repeat (`MEM_TEST_SHORT)
  `endif
`endif

begin
 `ifdef MEM_TEST_RANDOM
   tdata = $random(tseed);
   taddr = $random(tseed);
 `else
   taddr = taddr + 1;
   tdata = taddr;
 `endif
 mem_write (taddr, tdata);
 mem_read (taddr, tdata);
end
```

This illustration shows how you can use text macros to configure the testbench. Text macro variables select the duration of the test, and whether the memory access is random or determinate.

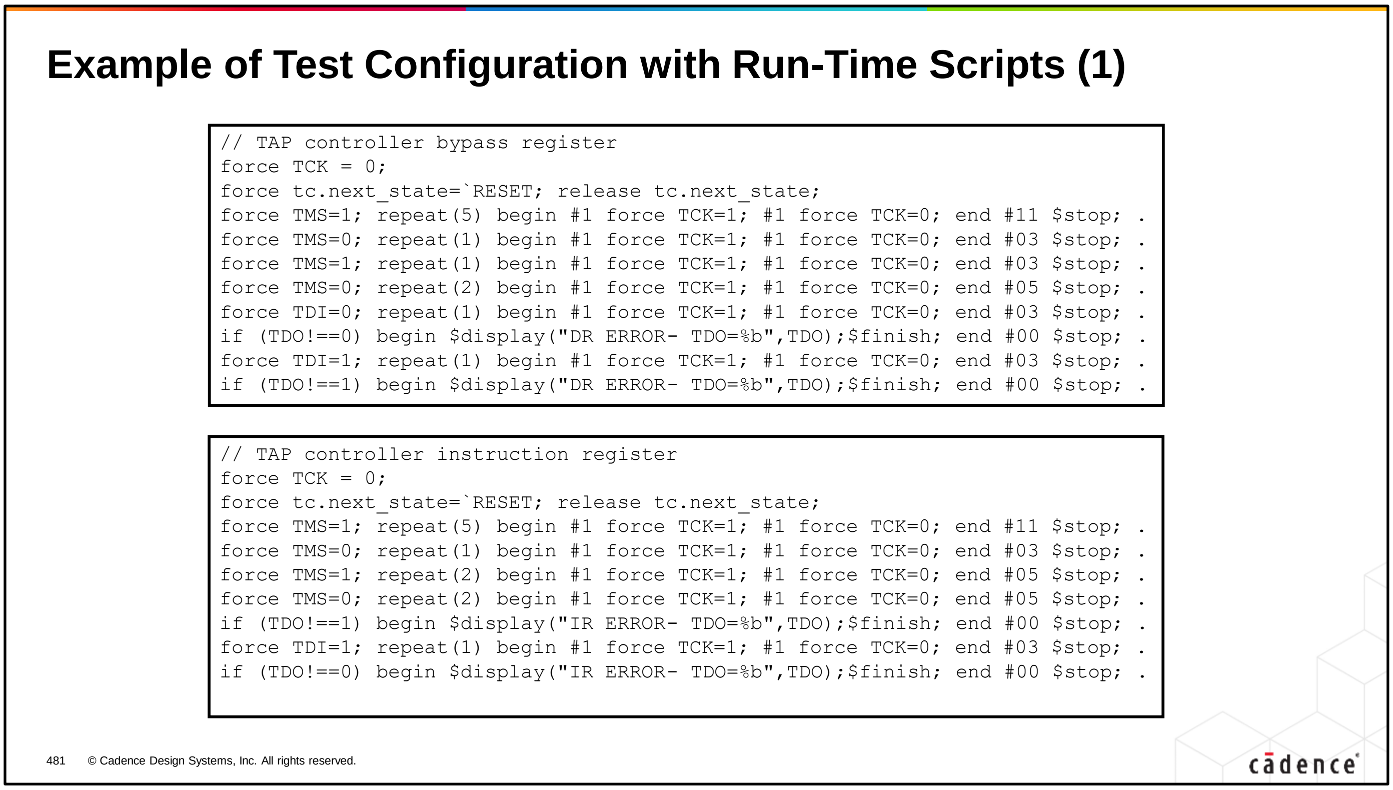
Test Configuration Using Run-Time Scripts

You can use run-time scripts to dynamically configure your testbench:

- Use platform utilities to construct a command line and run the simulator:
  - **awk**, **csh**, **make**, **perl**, **sed**, **sh**
- Use the scripting capabilities of the simulator interface:
  - **tcl**

You can use platform utilities to configure backplanes, check out the required components of a system configuration, build the simulator command line, and launch the simulator.

You can execute a file of run-time commands upon startup or at the command-line prompt. To test in this manner you must understand the progression of simulation time, and how the simulator performs interactive assignments to variables.



Example of Test Configuration with Run-Time Scripts (1)

```
// TAP controller bypass register
force TCK = 0;
force tc.next_state=`RESET; release tc.next_state;
force TMS=1; repeat(5) begin #1 force TCK=1; #1 force TCK=0; end #11 $stop; .
force TMS=0; repeat(1) begin #1 force TCK=1; #1 force TCK=0; end #03 $stop; .
force TMS=1; repeat(1) begin #1 force TCK=1; #1 force TCK=0; end #03 $stop; .
force TMS=0; repeat(2) begin #1 force TCK=1; #1 force TCK=0; end #05 $stop; .
force TDI=0; repeat(1) begin #1 force TCK=1; #1 force TCK=0; end #03 $stop; .
if (TDO!==0) begin $display("DR ERROR- TDO=%b",TDO);$finish; end #00 $stop; .
force TDI=1; repeat(1) begin #1 force TCK=1; #1 force TCK=0; end #03 $stop; .
if (TDO!==1) begin $display("DR ERROR- TDO=%b",TDO);$finish; end #00 $stop; .
```

```
// TAP controller instruction register
force TCK = 0;
force tc.next_state=`RESET; release tc.next_state;
force TMS=1; repeat(5) begin #1 force TCK=1; #1 force TCK=0; end #11 $stop; .
force TMS=0; repeat(1) begin #1 force TCK=1; #1 force TCK=0; end #03 $stop; .
force TMS=1; repeat(2) begin #1 force TCK=1; #1 force TCK=0; end #05 $stop; .
force TMS=0; repeat(2) begin #1 force TCK=1; #1 force TCK=0; end #05 $stop; .
if (TDO!==1) begin $display("IR ERROR- TDO=%b",TDO);$finish; end #00 $stop; .
force TDI=1; repeat(1) begin #1 force TCK=1; #1 force TCK=0; end #03 $stop; .
if (TDO!==0) begin $display("IR ERROR- TDO=%b",TDO);$finish; end #00 $stop; .
```

481 © Cadence Design Systems, Inc. All rights reserved.

cādence

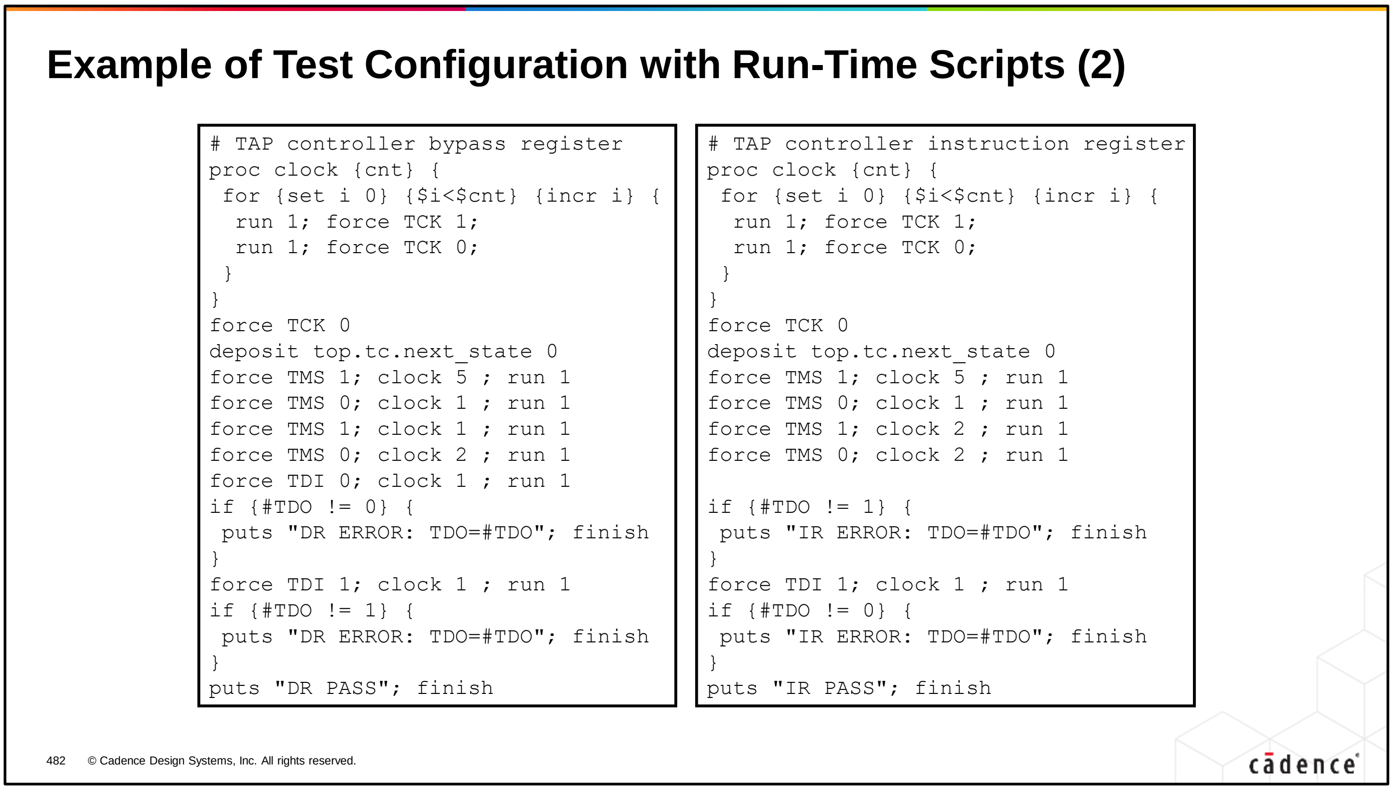
These examples test features of the Test Access Port (TAP) controller described by IEEE Standard 1149.1 as proposed by the Joint Test Action Group (JTAG). The two features tested are part of the universal feature set, and are valid for any TAP controller.

The TAP controller state machine is guaranteed to enter the reset state within five cycles of the Test Clock (TCK) pin if the Test Mode Select (TMS) pin is held high.

The first example steps the TAP controller state machine into the “shift data register” (SHIFT-DR) state and then shifts a zero followed by a one through the DR. The DR is selected by the Instruction Register (IR), which always selects the single-bit Bypass Register when reset, thus it is the Bypass Register that is tested.

The second example steps the TAP controller state machine into the “shift instruction register” (SHIFT-IR) state and then shifts a one into the multi-bit IR. This is sufficient to verify that the machine status captured in the IR contains 2'b01 in the two least significant bits, as required by the standard.

To run these examples “as is” the simulator must retain an interpretive engine. Purely compiled-code simulators cannot interpret Verilog code entered after the simulation starts.



Example of Test Configuration with Run-Time Scripts (2)

```
# TAP controller bypass register
proc clock {cnt} {
 for {set i 0} {$i<$cnt} {incr i} {
  run 1; force TCK 1;
  run 1; force TCK 0;
 }
}
force TCK 0
deposit top.tc.next_state 0
force TMS 1; clock 5 ; run 1
force TMS 0; clock 1 ; run 1
force TMS 1; clock 1 ; run 1
force TMS 0; clock 2 ; run 1
force TDI 0; clock 1 ; run 1
if {#TDO != 0} {
 puts "DR ERROR: TDO=#TDO"; finish
}
force TDI 1; clock 1 ; run 1
if {#TDO != 1} {
 puts "DR ERROR: TDO=#TDO"; finish
}
puts "DR PASS"; finish
```

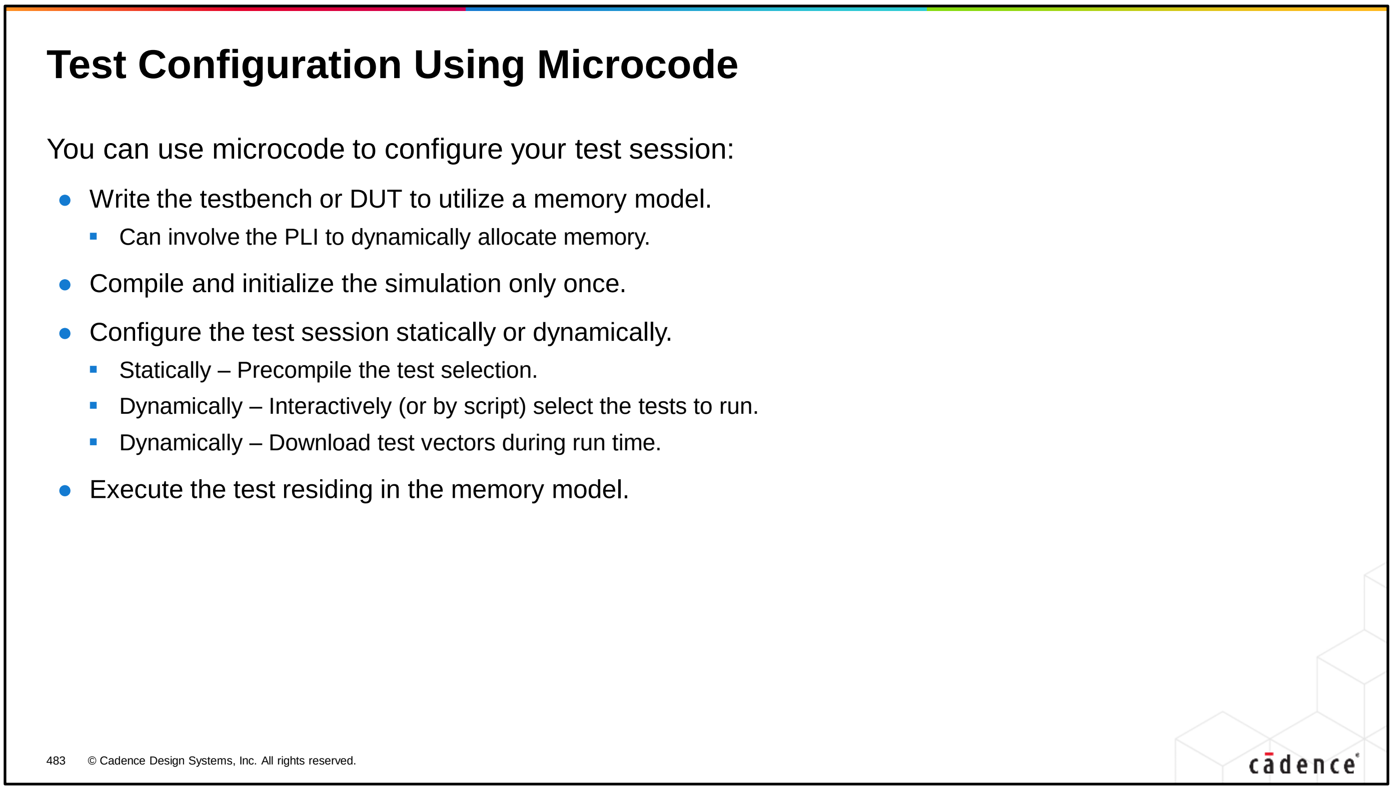
```
# TAP controller instruction register
proc clock {cnt} {
 for {set i 0} {$i<$cnt} {incr i} {
  run 1; force TCK 1;
  run 1; force TCK 0;
 }
}
force TCK 0
deposit top.tc.next_state 0
force TMS 1; clock 5 ; run 1
force TMS 0; clock 1 ; run 1
force TMS 1; clock 2 ; run 1
force TMS 0; clock 2 ; run 1

if {#TDO != 1} {
 puts "IR ERROR: TDO=#TDO"; finish
}
force TDI 1; clock 1 ; run 1
if {#TDO != 0} {
 puts "IR ERROR: TDO=#TDO"; finish
}
puts "IR PASS"; finish
```

This illustrates exactly the same script-based test, but utilizes the Tool Command Language (Tcl), an interactive language that many compiled-code simulators and other Electronic Design Automation (EDA) tools support.

This example runs on an implementation that provides its own non-standard interactive Tcl commands, for example:

- deposit – To deposit a value that persists only until another event replaces it.
- finish – Similar to the $finish system task.
- force – Similar to the force statement but forces a value and not a continuous assignment.
- run – To advance the simulation.
- The hash character (“#”) prefix to substitute the value of a simulation object.

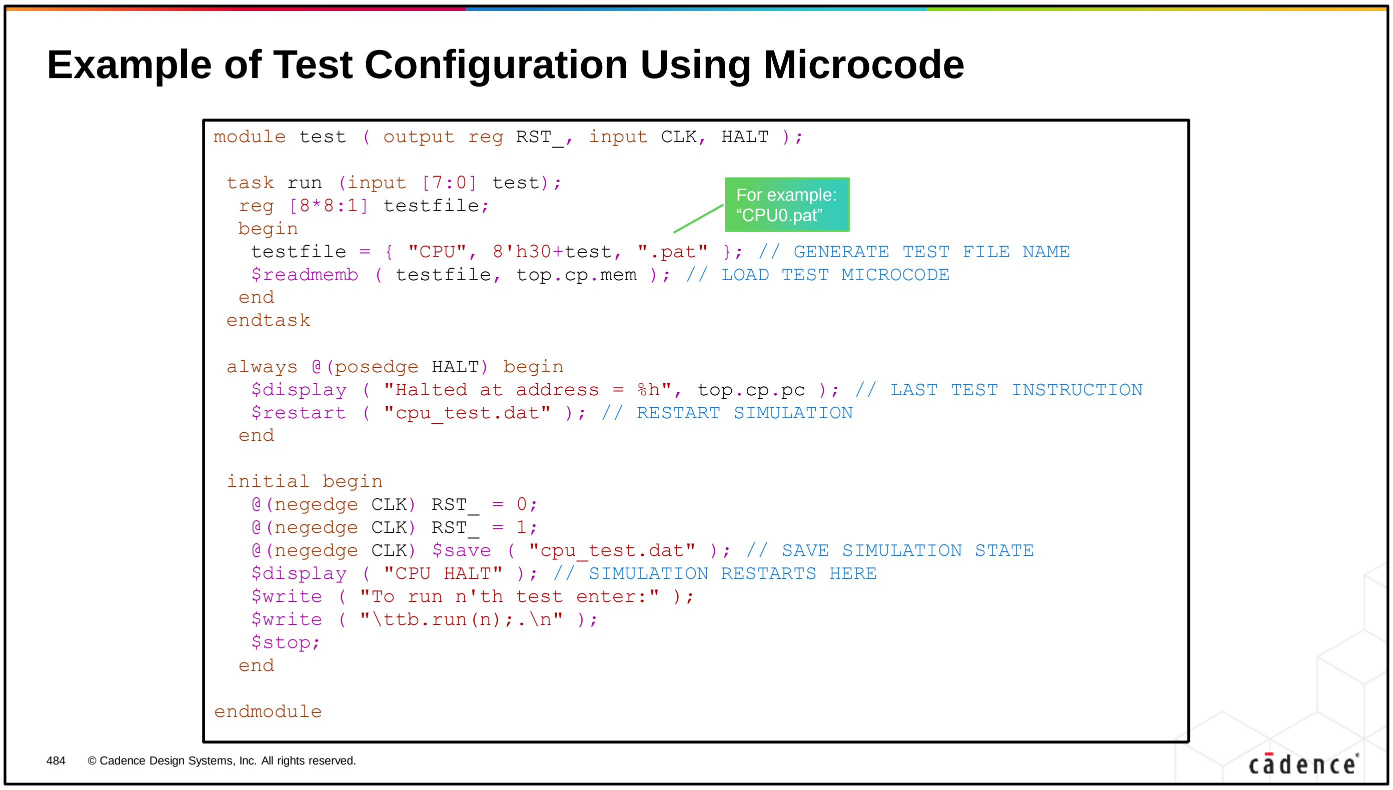


Test Configuration Using Microcode

You can use microcode to configure your test session:

- Write the testbench or DUT to utilize a memory model.
  - Can involve the PLI to dynamically allocate memory.
- Compile and initialize the simulation only once.
- Configure the test session statically or dynamically.
  - Statically – Precompile the test selection.
  - Dynamically – Interactively (or by script) select the tests to run.
  - Dynamically – Download test vectors during run time.
- Execute the test residing in the memory model.

Microcode execution is somewhat similar to vector playback. Both methods load and store test data, usually into pattern memory, which you may statically or dynamically allocate. The testbench can directly apply playback vectors to simulation objects, such as nets and registers. The testbench can generate and apply test vectors by executing microcode in a machine that is either in the testbench or part of the design under test. A microcoded test is usually significantly more compact than the same test represented with raw vectors.



Example of Test Configuration Using Microcode

```
module test ( output reg RST_, input CLK, HALT );

 task run (input [7:0] test);
  reg [8*8:1] testfile;
  begin
   testfile = { "CPU", 8'h30+test, ".pat" }; // GENERATE TEST FILE NAME
   $readmemb ( testfile, top.cp.mem ); // LOAD TEST MICROCODE
  end
 endtask

 always @(posedge HALT) begin
   $display ( "Halted at address = %h", top.cp.pc ); // LAST TEST INSTRUCTION
   $restart ( "cpu_test.dat" ); // RESTART SIMULATION
  end

 initial begin
   @(negedge CLK) RST_ = 0;
   @(negedge CLK) RST_ = 1;
   @(negedge CLK) $save ( "cpu_test.dat" ); // SAVE SIMULATION STATE
   $display ( "CPU HALT" ); // SIMULATION RESTARTS HERE
   $write ( "To run n'th test enter:" );
   $write ( "\ttb.run(n);.\n" );
   $stop;
  end

endmodule
```

For example:
“CPU0.pat”

484 © Cadence Design Systems, Inc. All rights reserved.

In this example, the testbench saves the entire simulation database immediately upon completing the initialization. The initialization sequence for a typical design could require significant simulation time that may not be necessary to repeat for each subtest.

This example generates a unique filename from the “run” task argument, and reads the vectors in that file into memory to exercise the CPU model.

The $restart system task reloads the simulation database. The state of the simulation upon restarting is exactly the state saved by the $save system task.

The IEEE Standard 1364-2001 Verilog describes but does not require the $save and $restart system tasks. Not all simulators support these tasks, and some offer this feature with some other proprietary interactive command.

To run these examples “as is”, the simulator must retain an interpretive engine. Purely compiled-code simulators cannot interpret Verilog code entered after the simulation starts.

Program Code Which Runs the Test

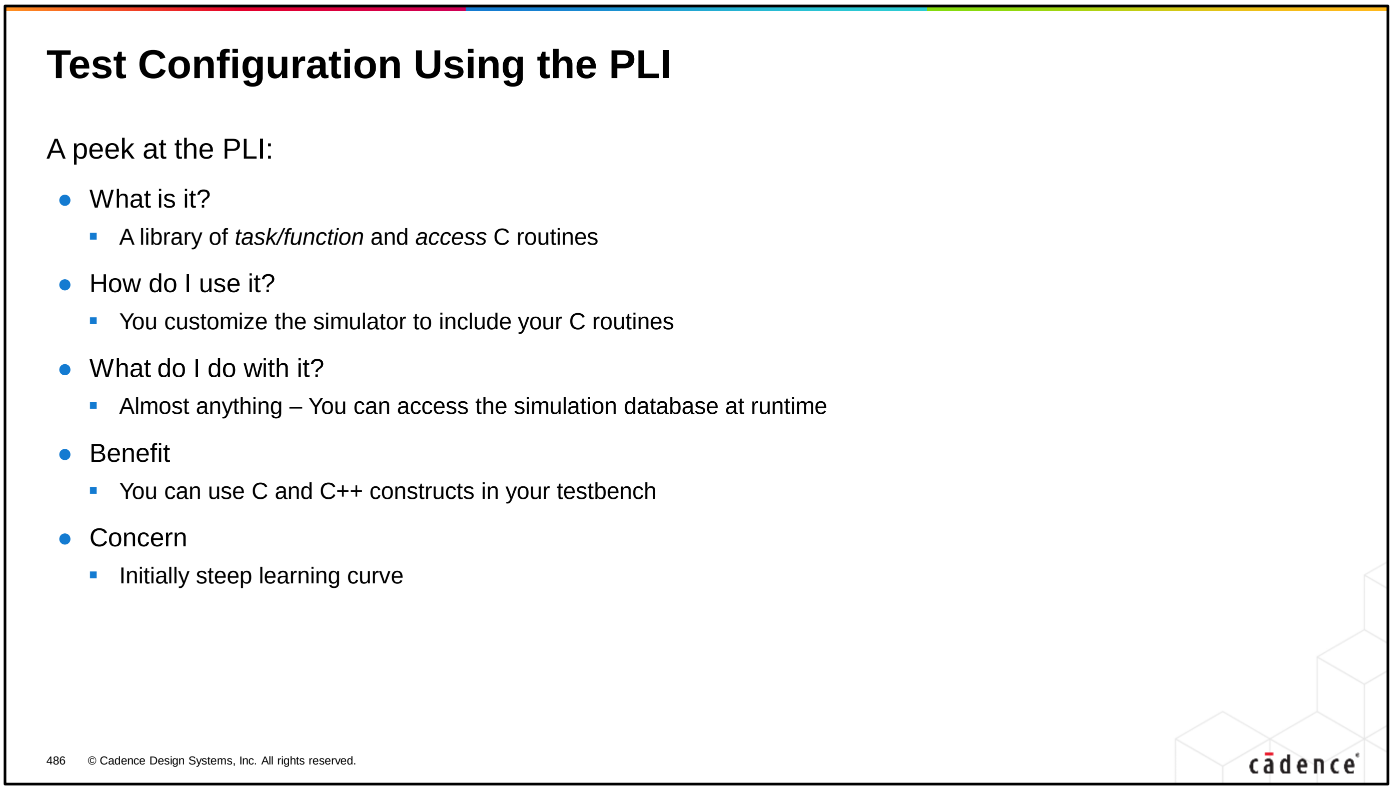
CPU0.pat

```
/****************************************************************************************************************************
 * Test program for the VeriRISC CPU system
 *
 * This diagnostic program tests the basic instruction set of the VeriRISC system. If the system executes each
 * instruction correctly, then it should halt when the HLT instruction at address 17(hex) is executed.
 * If the system halts at any other location, then an instruction did not execute properly.  Refer to the
 * comments in this file to see which instruction failed.
 ****************************************************************************************************************************/
//opcode_operand         // addr                          assembly code
//---------------------// --------------------------------------------------------------------------------------------
@00 111_11110            //  00   BEGIN:     JMP TST_JMP
    000_00000            //  01              HLT           //JMP did not work at all
    000_00000            //  02              HLT           //JMP did not load PC, it skipped
    101_11010            //  03   JMP_OK:     LDA DATA_1
    001_00000            //  04               SKZ
    000_00000            //  05               HLT          //SKZ or LDA did not work
    101_11011            //  06              LDA DATA_2
    001_00000            //  07              SKZ
    111_01010            //  08              JMP SKZ_OK
    000_00000            //  09              HLT           //SKZ or LDA did not work
    110_11100            //  0A   SKZ_OK:    STO TEMP      //store non-zero value in TEMP
    101_11010            //  0B              LDA DATA_1
    110_11100            //  0C              STO TEMP      //store zero value in TEMP
    101_11100            //  0D              LDA TEMP
    001_00000            //  0E              SKZ           //check to see if STO worked
    000_00000            //  0F              HLT           //STO did not work
    100_11011            //  10              XOR DATA_2
    001_00000            //  11              SKZ           //check to see if XOR worked
    111_10100            //  12              JMP XOR_OK
    000_00000            //  13              HLT           //XOR did not work at all
    100_11011            //  14   XOR_OK:    XOR DATA_2
    001_00000            //  15              SKZ
    000_00000            //  16              HLT           //XOR did not switch all bits
    000_00000            //  17   END:       HLT           //CONGRATULATIONS - TEST1 PASSED!
    111_00000            //  18              JMP BEGIN     //run test again

@1A 00000000             //  1A   DATA_1:                   //constant 00(hex)
    11111111             //  1B   DATA_2:                   //constant FF(hex)
    10101010             //  1C   TEMP:                    //variable - starts with AA(hex)

@1E 111_00011            //  1E   TST_JMP:  JMP JMP_OK
    000_00000            //  1F             HLT            //JMP is broken
```

This page does not contain notes.



Test Configuration Using the PLI

A peek at the PLI:

- What is it?
  - A library of *task/function* and *access* C routines
- How do I use it?
  - You customize the simulator to include your C routines
- What do I do with it?
  - Almost anything – You can access the simulation database at runtime
- Benefit
  - You can use C and C++ constructs in your testbench
- Concern
  - Initially steep learning curve

486 © Cadence Design Systems, Inc. All rights reserved.

The Programming Language Interface (PLI) is an Application Program Interface (API) to your simulator environment. It contains utility and access routines that you call from your C programming language functions to interact with instantiated simulation objects. With the Verilog Procedural Interface (VPI), which evolved from the Open Verilog International (OVI) PLI version 2.0, you can also access behavioral constructs such as processes, blocks, and statements.

You compile your C routines with the PLI or VPI routine libraries and simulator core to create a customized version of the simulator. The customized simulator recognizes calls from your Verilog source to your C routines.

Some things you can do with the PLI are to:

- Determine what simulation objects of a specified type the design contains, where they are, and what their fanins and fanouts are.
- Calculate and annotate delays.
- Set values and monitor value changes and bus contention during simulation.
- Use dynamically allocated memory in your design.
- Access UNIX utilities, do file I/O and IPC, co-simulate, and import 3rd-party models.

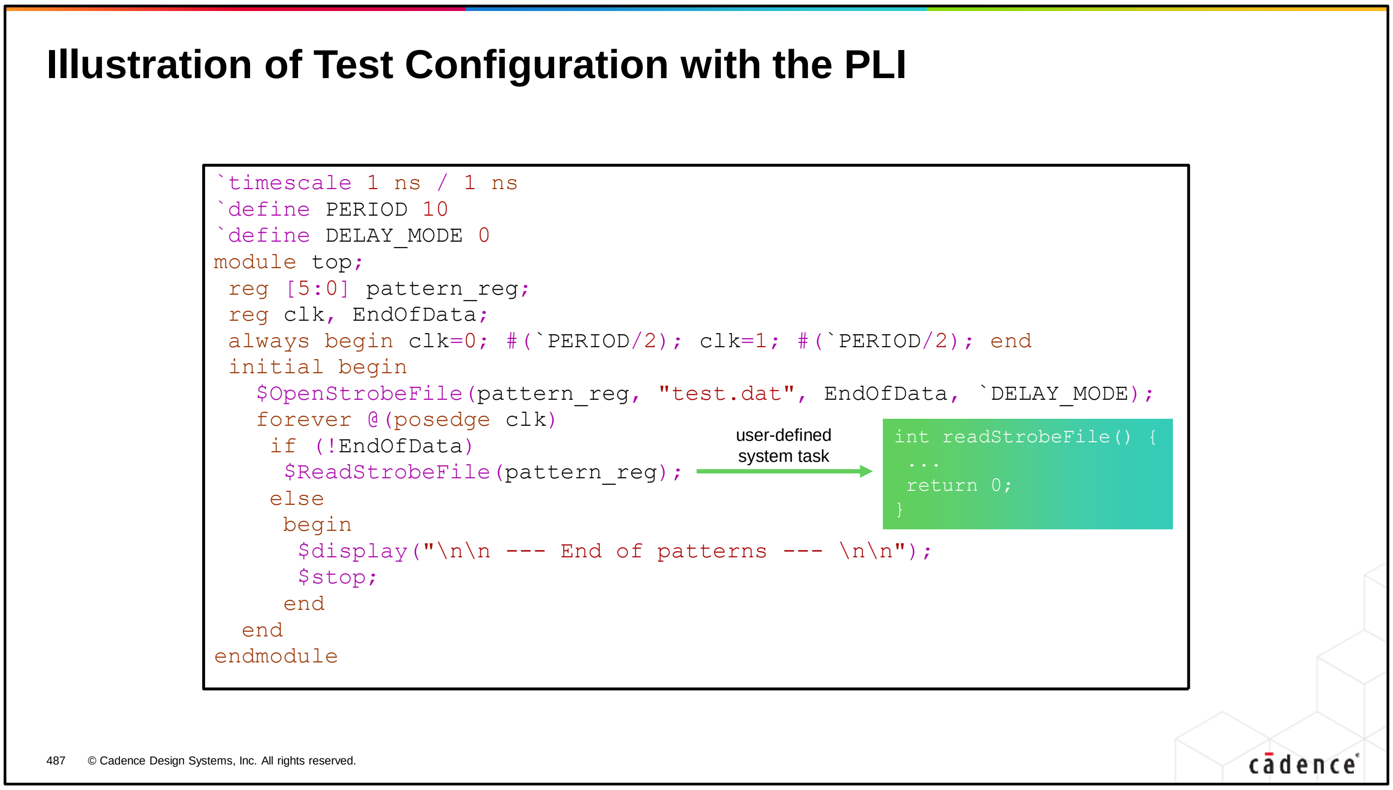


Illustration of Test Configuration with the PLI

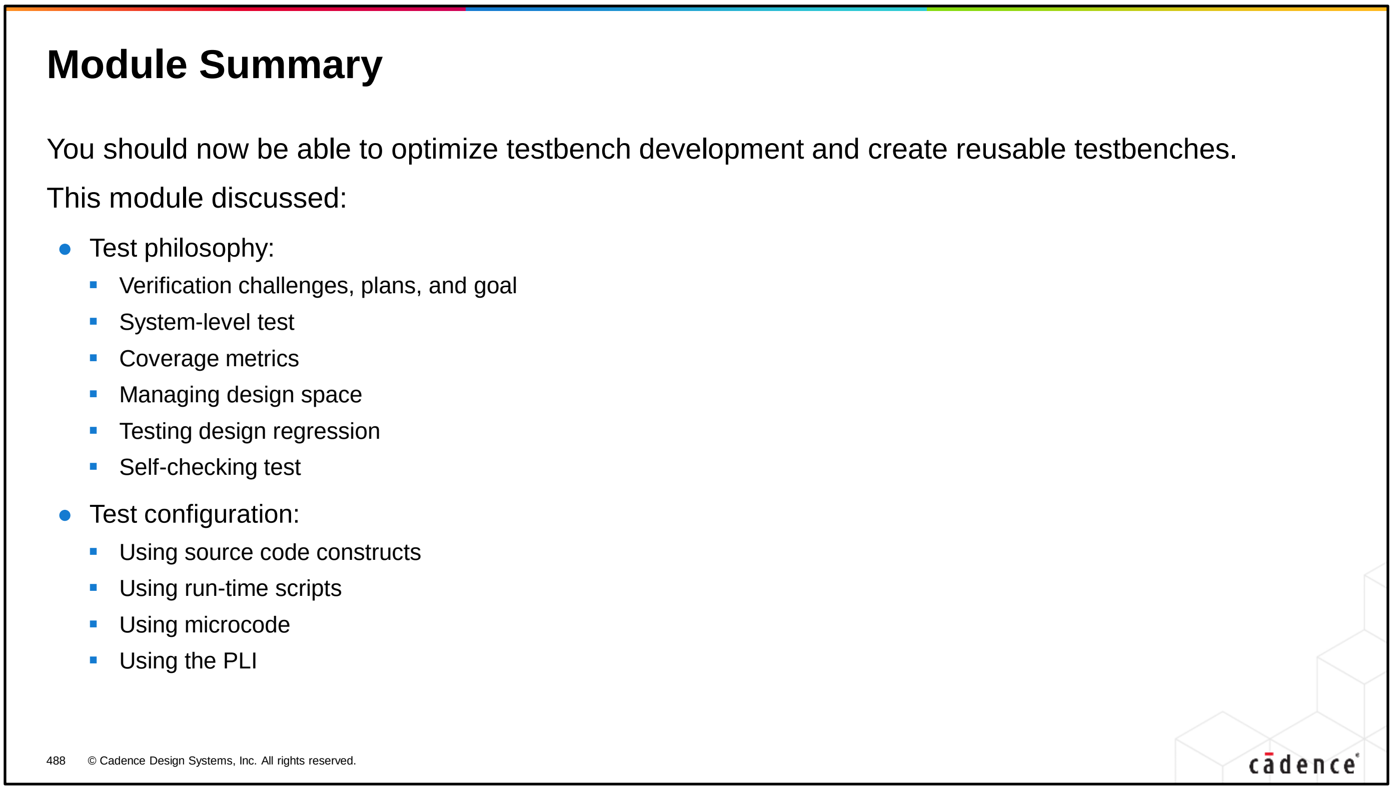
```
`timescale 1 ns / 1 ns
`define PERIOD 10
`define DELAY_MODE 0
module top;
 reg [5:0] pattern_reg;
 reg clk, EndOfData;
 always begin clk=0; #(`PERIOD/2); clk=1; #(`PERIOD/2); end
 initial begin
   $OpenStrobeFile(pattern_reg, "test.dat", EndOfData, `DELAY_MODE);
   forever @(posedge clk)
    if (!EndOfData)
     $ReadStrobeFile(pattern_reg);
    else
     begin
      $display("\n\n --- End of patterns --- \n\n");
      $stop;
     end
   end
endmodule
```

```
int readStrobeFile() {
 ...
 return 0;
}
```

The testbench calls the user-defined $OpenStrobeFile system task to open a channel to the “test.dat” pattern file, giving the application the pattern register, pattern file name, flag register, and delay mode. The PLI application uses the delay mode to signify whether it should update the pattern register immediately upon each $ReadStrobeFile call, or after applying a pattern delay time.

The testbench then continues to update the pattern register upon each clock cycle, stopping when the “EndOfData” register asserts. The application asserts the EndOfData register when the testbench attempts to read pattern data past the end of the pattern file.

The readStrobeFile() C routine associated with the $ReadStrobeFile user-defined system task reads and applies the next pattern and updates the state of the EndOfData register.



Module Summary

You should now be able to optimize testbench development and create reusable testbenches.

This module discussed:

- Test philosophy:
  - Verification challenges, plans, and goal
  - System-level test
  - Coverage metrics
  - Managing design space
  - Testing design regression
  - Self-checking test
- Test configuration:
  - Using source code constructs
  - Using run-time scripts
  - Using microcode
  - Using the PLI

You should now be able to optimize testbench development and create reusable testbenches.

This module presented design verification concepts, reinforced with small working examples and illustrations. The first part of this module explored the philosophy of system-level test, including the test plan, completion criteria, and design data management. The second part described self-checking test methods and various ways to configure the test environment.

Module Review

1. What are some characteristics of a system-level test?
2. Which are some characteristics of a regression test?
3. Suggest some ways to dynamically configure the test while the simulator is running.

This page does not contain notes.

Module Review Solutions

1. What are some characteristics of a system-level test?
   - System-level test requires up-front planning and some continued maintenance and restricts the style of the individual designer, but also facilitates communication among the development team and reduces duplication of verification efforts.
2. Which are some characteristics of a regression test?
   - A regression test suite is typically self-checking and launched by some automatic process that logs and reports the tested configuration and the results.
3. Suggest some ways to dynamically configure the test while the simulator is running.
   - You can write testbench code to dynamically react to keyboard or script input and to execute instructions from a file. The testbench code can be in Verilog or in the C programming language.

This page does not contain notes.

Lab

Lab 23-1 Testing a VeriRISC CPU Model

- For this lab, you interactively select and download test microcode and run it
- You generate a test that asks the user which program to run. You load that program into the CPU memory, reset the CPU, and let it run.
- You report the address where the program halts and whether that address is correct.

Your objective for this lab is to interactively select and download test microcode and run it.

The lab model is a VeriRISC CPU. The CPU utilizes a two-port memory so that it can fetch and execute an instruction on each clock cycle. The 8-bit CPU instruction consists of a 3-bit leftmost operation code encoding eight instructions, and a 5-bit rightmost operand addressing up to 32 words.

The lab provides three programs with incremental levels of complexity.

The CPU has only the HALT output, thus the test environment can detect program failure only by detecting that the CPU halted at an incorrect address.

For this lab, you generate a test that asks the user which program to run. You load that program into the CPU model memory, reset the CPU, and let it run. You report the address where the program halts and whether that address is correct.