CHAPTER 1 CSL Testbench

TABLE 1.1 Chapter Outline

1.1 Definitions
1.2 CSL Testbench Overview
1.3 CSL Testbench Concepts
1.4 CSL Testbench Examples

1.1 Definitions

The following table defines the terms used in the CSL testbench document.

TABLE 1.2 Definitions

Term used	Definition
tb	Test Bench
tx	transmit
rx	receive
dut	Design under test
csim	Behavioral C++ Simulator
vsim	Verilog Simulator
bfm	Bus functional model
vector	A set of values
stimulus vector	A set of values applied on the inputs of a design under test
expected vector	A set of values compared against the outputs of a design under test
architectural state	The state of a circuit at a certein moment in time. The content of all memory cells inside the circuit.
expected architectural state	The expected state of a circuit

A testbench is a structure designed to verify the functionality of an RTL design. The components (unit instances) of the DUT are instantiated in a testbench using the same CSL syntax used to include an instance in a unit. DUT memories can be initialized. Stimulus vector(s) are connected to the DUT(s) inputs. The DUT outputs are connected to the comparators which are also connected to expected vector objects.
Vector - you associate a Vector with a module name. In a Testbech you instanciate a Vector that is connected to a module instance.
StateData - takes a HID that points to a memory instance.

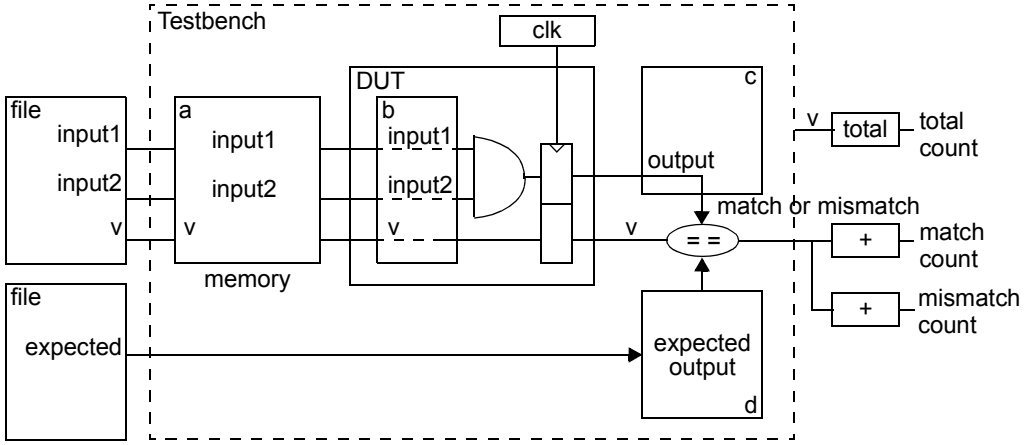
1.2 CSL Testbench Overview

Reset and clock signal generators are added to the testbench; other signal generators are added to the testbench as needed. State data objects and comparators are instantiated and connected to the DUT(s).
A testbench creates an operating environment around the design under test (DUT). The testbench emulates the behaviour of the DUT input pins that the DUT “sees” in real operating mode.
The testbenches are used by hardware engineers to verify that the design’s RTL is functionally corect. To test the design means to verify whether it provides correct outputs for the given inputs. That means that the outputs provided by the design when driven by a given set of inputs should match the expected outputs wich are known good set of outputs.

This type of testing requires the use of a C++ simulator, previously generated vectors or some other model which generates stimulus and expected vectors. The stimulus inputs and expected outputs are generated by another model and used to stimulate the DUT and check the DUT’s outputs. The C++ simulator will generate two types of vectors: stimulus vectors which contain the inputs and expected vectors which contain the known good outputs. In the example in Figure 1.1, the stimulus vectors correspond to **a** and the expect vectors correspond to **d**. In the same figure **e** is the C++ gen erator for stimulus and expected vectors.

In Figure 1.1 **v** is the event which triggers the compare between the DUT output vector and the expected vector. The DUT consists of an AND gate with two inputs. The first set of inputs are (0 1) and the expected output is 0 (0 & 1). If the output generated by the AND gate is 0 (matches the expected output) additional inputs can then be applied (1 0, 0 0, 1 1). If not, something went wrong and an error message is issued.

FIGURE 1.1 Simple Testbench



As a conclusion to this example, make a firm distinction between these 4 concepts(see Figure 1.1):

1.A set of inputs(**a**) (test input values) that drive the design under test;

2.Inputs(**b**) are the actual inputs of the design (they are connected to the given inputs);

3.Outputs(**c**) represent the values provided by the design under test for the given set of inputs (can be right or wrong);

4.Expected outputs(**d**) are the known good values that the design under test should provide. The table below shows the values of the stimulus and expected vectors.

TABLE 1.3 Stimulus and expected vectors values

No.	Stim vector	Expected vector
1	0_0	0
2	0_1	0
3	1_0	0
4	1_1	1

The stimulus vectors reflect all possible input combinations the design should be able to handle. The expected vectors contain all the outputs the design should yield. If the design under test had 2 outputs, instead of just one like in this case, the expect vectors would each have 2 values, one for each output.

The C++ simulator is a software model of the DUT that accurately reproduces its behaviour. In our case, a C++ function that simulates the logical AND operation. The signals in the stimulus vector generated by the C++ simulator are then used to drive the DUT(one at a time), which will provide

output signals organised in an output vector. In our case one signal per vector, because the simple design that we chose has only one output.

The output vector contains whatever the real design generates(c. in the picture), it is not the same as the expect vector(d. in the picture).
The output vector is then compared against the expect vector. If they match, then the design works correctly. If they don't, one of the following situations occured:

1.The DUT does not work properly

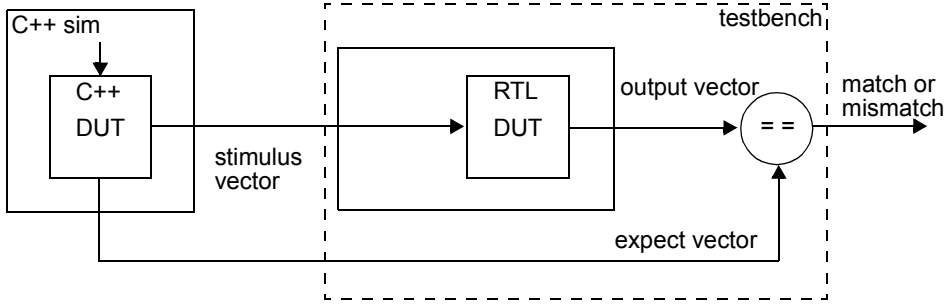
2.The C++ simulator does not work properly

3.The 2 vectors(output and expect) were not aligned or not synchronised correctly

4.The DUT produced more output vectors than expected

Figure 1.2 has been redrawn for a more general case. A vector is represented by a block arrow, because it stands for a bundle of signals(one or more).

FIGURE 1.2 Testbench using C++ simulator



1.2.1 CSL Vector Comparison

The expected vector produced by the software model of the DUT is compared against the output vector produced by the actual DUT(both the DUT model and the actual DUT are driven by the stimulus vector). This comparison has several stages:

1.The C++ simulator generates stimulus vectors which are written to a file;

2.A stimulus vector is extracted from the file and set as input for both the software simulated DUT and the real one;

3.Every clock edge or every time a specific event occurs.

The expected vector (generated by the C++ simulator) is stored in a file (expect file). The output vector (produced by the real DUT) is captured and stored in another file(output file). The two vectors in the two files are compared and if they match, another stimulus vector is extracted from the stimulus

file and set as input for both the simulated DUT and the real one, if they don't match, an mismatch counter is incremented; if the max number of mismatches has been reached,the process stops and an error message is generated if not, the next stim vector is extracted and the comparison goes on.

Back to the AND gate example and follow the steps above (assuming the max no. of errors is 1):

1.The stimulus file(containing the stimuls vectors) looks as follows: 00, 01, 10, 11;

2.The first stimvect is extracted 00. The simulated DUT produces the following expect vector: 0 .This is stored in file expect1; Let's say the output vector produced by the real DUT is also 0. This is stored in file output1. The vectors match, therefore, step b is resumed:

3.The next stimvect is extracted 01.

4.The simulated DUT generates the expected vector 0 stored in file expect2, and let's suppose the real DUT's output is stored in file output2

The vectors in the two files(expect2 and output2) do not match so the algorithm is stopped and an error message is generated. As seen before, the capture and comparison of the two vectors can happen at clock edge or every time a specific event occurs.

Let's say the specific event is the following: a signal inside the DUT shifts from 0 to 1 for the 10th time. A software counter will be implemented to keep track of the signal's evolution (increment a value every time the signal shifts from 0 to 1). Every time the signal value becomes 1 for the tenth time, the vectors are captured and compared, the counter is reset to 0, and the tracking of the signal is resumed. If something goes wrong and the DUT gets stuck in a loop, there is a good chance the counter will never reach ten(the signal will probably "get stuck" too). To prevent the testbench from endlessly waiting for the two vectors, the user can set a timeout value (set_timeout command). If the counter doesn't reach 10 faster than the value specified by the user, an error message is issued.

1.2.2 CSL Architectural State

The files(stimulus, expected and output) are stored in the memory elements automatically generated for that purpose. The contents of any of those files at a specific time(after a particular event occurs) is called architectural state.

For example, here's how the file with the output vector could look after the counter in the previous example has hit 10(this is its architectural state):

1011
1000
1100
1111

All the output signals generated by the DUT for one specific stimulus vector are included in this output vector. This example is far more complicated then the AND gate as it has 16 outputs instead of just one. This is not meant to complicate things, it's simpy more suggestive for illustrating the concept of architectural state. Each line stands for one register.(Meaning that the DUT has four regis-

ters.)

1.2.3 Inteface consistency

The **cslc** generates consistent interfaces for the following components: vector readers/writers (A), vectors and architectural states files (B), testbench memories (C), comparators (D), DUT interface (E). See Figure 1.3

FIGURE 1.3 Interface consistency between C++ Simulator and vsim object.

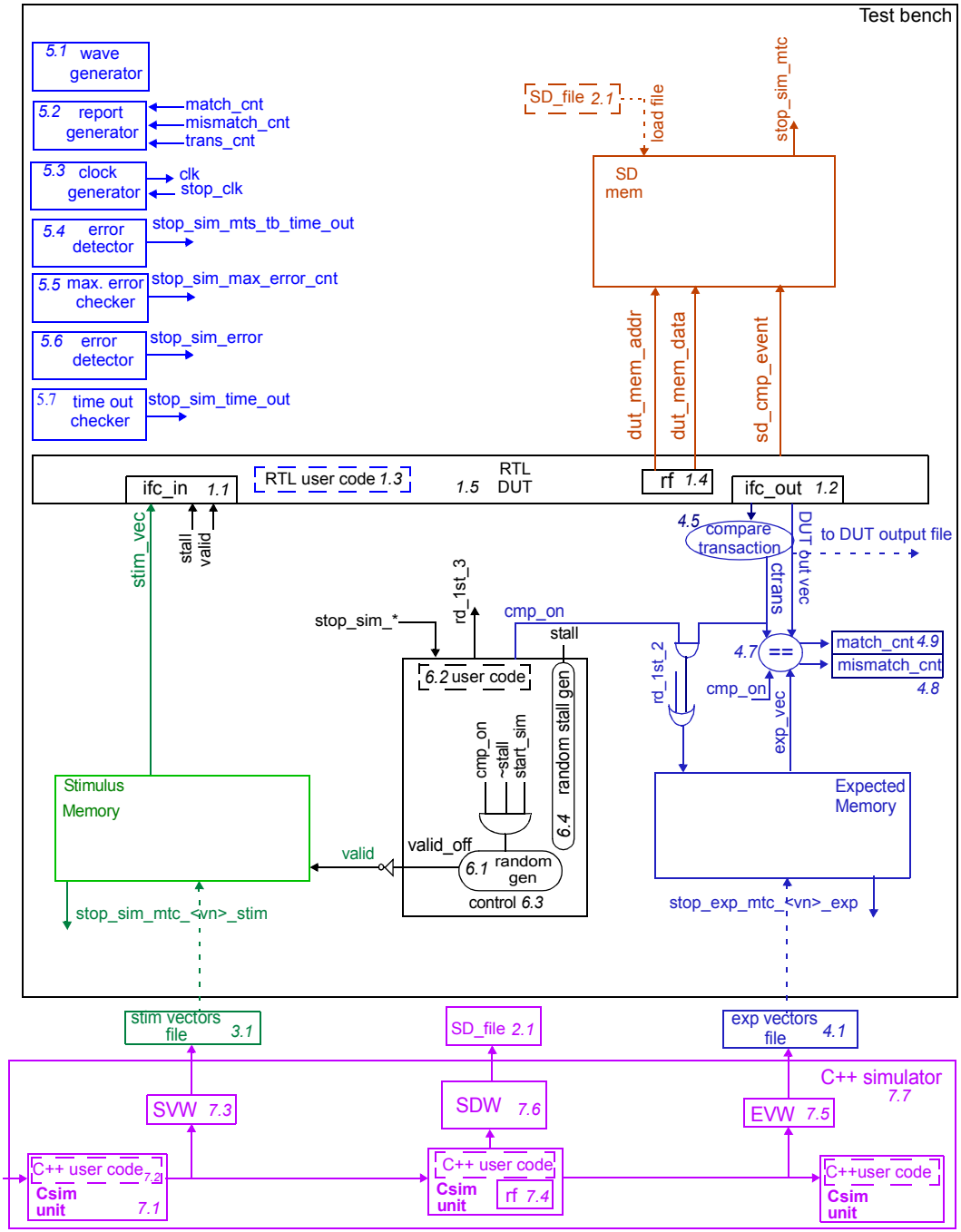


TABLE 1.4

Nr. of components	Description	
1.1	DUT input interface	user
1.2	DUT output interface	user
1.3	RTL user code	user
1.4	Register file	user
1.5	DUT(Design Under Test)	user
2.1	State data file	csim
2.2	State data memory address generator	adapter
2.3	Expected state data memory	adapter
2.4	Expected state data id/ version checker	adapter
2.5	State data compare transaction event	adapter
2.6	State data maximum transactions checker	adapter
2.7	DUT/expected state data compare	adapter
2.8	Expected state data match counter	adapter
2.9	Expected state data mismatch counter	adapter
2.10	Read address checker	adapter
2.11	Zero out mux	adapter
3.1	Stimulus vector file	csim
3.2	Stimulus vector memory address generator	adapter
3.3	Stimulus vector memory	adapter

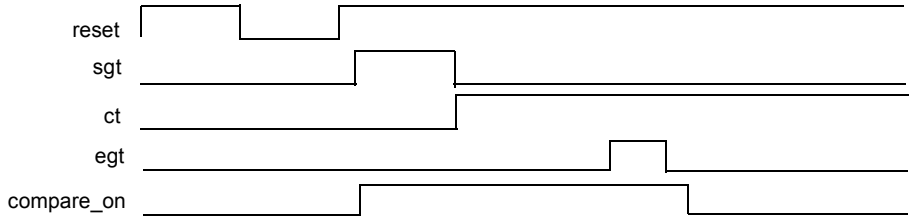
TABLE 1.4

Nr. of components	Description	
3.4	Stimulus vector id/version checker	
3.5	Read address checker	
3.6	Zero out mux	
4.1	Expected vector file	
4.2	Expected vector memory address generator	
4.3	Expected vector memory	
4.4	Expected vector id/version checker	
4.5	Expected vector compare transaction event	
4.6	Expected vector maximum transaction checker	
4.7	DUT/expected vector compare	
4.8	Expected vector mismatch counter	
4.9	Expected vector match counter	
5.1	Wave generator	
5.2	Report generator	
5.3	Clock generator	
5.4	Error detector	
5.5	Maximum errors checker	
5.6	Error detector	
5.7	Time out checker	
6.1	Random stall generator	

TABLE 1.4

Nr. of components	Description	
6.2	User code	
6.3	Testbench control unit	
6.4	Random stall generator	
7.1	Unit	
7.2	C++ user code	
7.3	Stimulus vector writer	
7.4	Register file	
7.5	Expected vector writer	
7.6	State data writer	
7.7	C++ simulator	

FIGURE 1.4 Waves



1.3 CSL Testbench Concepts

1.3.1 C++ Simulator

C++ Simulator is used in testbenches to create golden models and to generate vectors and architectural states. The C++ model behavior is compared against RTL model behavior created using VSim.

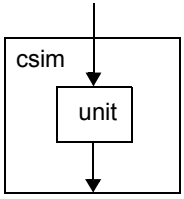
1.3.2 Golden Units

A golden unit or golden device is an ideal example of device against which all other devices are tested and judged. The Golden units are described using CSim and RTL units are compared against

those golden units.
The C++ Simulator -vm argument tells the C++ Simulator to create and connect PLI's for each of the verilog DUT's interfaces. C++ Simulator and verilog PLI's
Each legal verilog name maps to a set of PLI's in the C++ Simulator
The C++ Simulator contains a lookup table which maps the verilog name to a set of stimulus and expect vectors. Each vector which is specified in the table is mapped to a PLI

1.3.2.1 C++ simulator architecture

FIGURE 1.5 C++ simulator



The C++ Simulator is an equivalent model of the Verilog DUT at the chip, cluster and/or unit boundaries.A csim unit is a C++ simulator unit. It is equivalent in function to a RTL unit.The csim unit has an interface (inputs and outputs), logic and state(registers and register files and memories).

FIGURE 1.6 Define the verification point

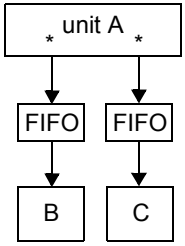
```
Top.csim_module (); //
A.csim_unit ();
B.csim_unit ();
C.csim_unit ();

Top.vector (in, <name>); // put all inputs in vectors
Top.vector (out, <name>); // put all outputs in vector
A.vector (sig_list, a_b);// put all signals driven by a and received by
b in vector a_b
```

Other names can be used for C++ modules and units also.

```
Top.csim_module (root);
A.csim_unit (aa);
B.csim_unit (bb);
C.csim_unit (cc);
```

1.3.2.2 Vector Generation

FIGURE 1.7 Multiple vectors from a single module



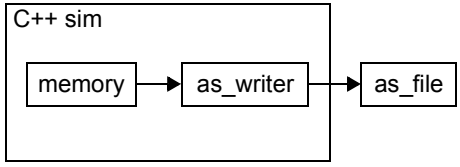
The vector formats are used in the following

- documentation
- c model unit
- c model vector
- testbench
- DUT
- Hw tester
- vector reader
- logic analyser

State elements have to be defined / same in Docs c.model DUT, c.model file, TB Reader,DUT.
Implementation need to be able to “backdoor“ load SES in c. model and verilog model.
CSL Vector
Stim/Expect vector generation
Stimulus vectors are generated every cycle and written to the vector file . Expect vectors are generated when a vector event occurs and are written to the file.

1.3.2.3 C++ Expected achitectural state writers

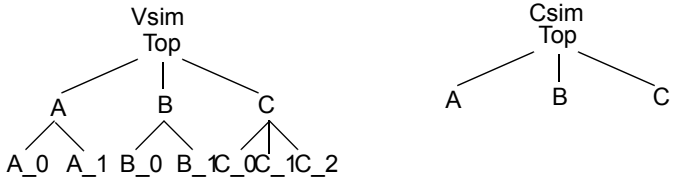
FIGURE 1.8 Architectural state writer



1.3.2.4 Example of C++ Simulator vs Vsim hierarchies

In Figure 1.4 are shown the differences between VSim and CSim hierarchies

FIGURE 1.9 Vsim and C++ Simulator Hierarchical Differences

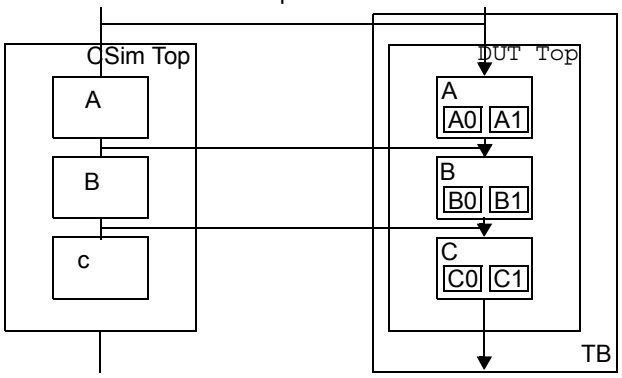


CSL CODE

```
csl_unit  TOP; //declares "TOP" csl_module
csl_unit   A; //declares "A" csl_module
csl_unit   B; //declares "B" csl_module
csl_unit   C; //declares "C" csl_module
csl_unit   A0; //declares "A0" csl_module
csl_unit   A1; //declares "A1" csl_module
csl_unit   B0; //declares "B0" csl_module
csl_unit   B1; //declares "B1" csl_module
csl_unit   C0; //declares "C0" csl_module
csl_unit   C1; //declares "C1" csl_module
csl_unit   C2; //declares "C2" csl_module
TOP.add_instance(A,A_); //add A as leaf for "TOP" csl module
TOP.add_instance(B,B_); //add B as leaf for "TOP" csl module
TOP.add_instance(C,C_); //add C as leaf for "TOP" csl module
A.add_instance(A0,A_0); //add A0 as leaf for "A" csl module
A.add_instance(A1,A_1); //add A1 as leaf for "A" csl module
B.add_instance(B0,B_0); //add B0 as leaf for "B" csl module
B.add_instance(B1,B_1); //add B1 as leaf for "B" csl module
C.add_instance(C0,C_0); //add C0 as leaf for "C" csl module
C.add_instance(C1,C_1); //add C1 as leaf for "C" csl module
C.add_instance(C2,C_2); //add C1 as leaf for "C" csl module
```

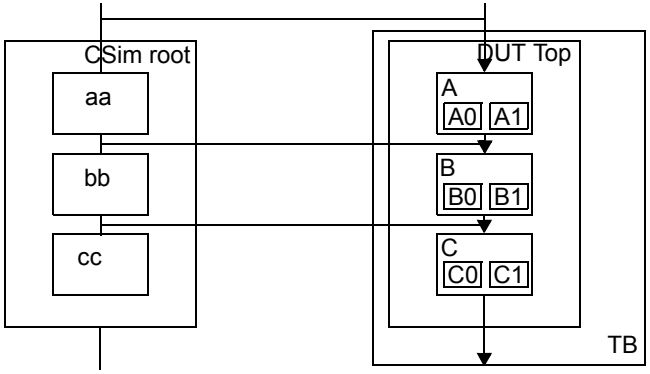
C++ Simulator modules are created with the Vsim lfc vector format. It uses existing unit Top, A, B, C and declares that there is a C++ Simulator module with the same name: A, B, C

FIGURE 1.10 Define the verification point




```
Top.csim_module (); //
A.csim_unit ();
B.csim_unit ();
C.csim_unit ();

Top.vector (in, <name>); // put all inputs in vectors
Top.vector (out, <name>); // put all outputs in vector
A.vector (sig_list, a_b);// put all signals driven by a and received by
b in vector a_b
```

FIGURE 1.11 Define the verification point

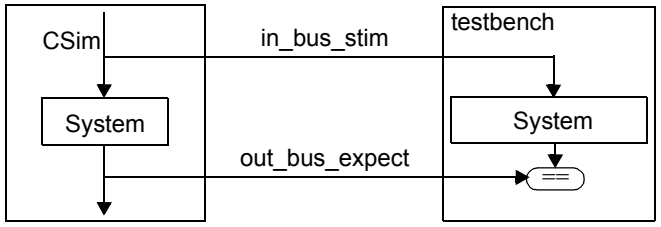



```
Top.csim_module (root);
A.csim_unit (aa);
B.csim_unit (bb);
C.csim_unit (cc);
```

1.3.2.5 The C++ simulator and RTL DUT compare levels are specified in CSL

The C++ Simulator does not need to model the complete design hierarchy. Instead, verification points in the design are chosen in the C++ Simulator generates vectors to verify those interfaces. See Figure 1.9 and Figure 1.10

FIGURE 1.12 Stimulus and expected vectors



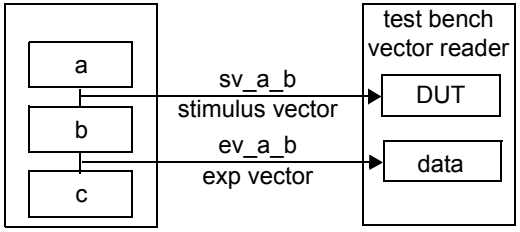
1.3.2.6 Test vector language

!!add language constructs:

- count
- branch
- loop

1.3.2.7 Common interfaces

FIGURE 1.13 Standard Testbench




```
void fsdb_dump_file( const char *pcszPath );
void fsdb_dump_enable( int en );

p_fsdb_scope fsdb_set_top_scope( p_fsdb_scope p_scope );
p_fsdb_scope fsdb_add_scope( p_fsdb_scope p_parent_scope, const char *pc_name, ... );
void fsdb_add_signal( p_fsdb_scope p_scope, p_fsdb_sig_bits pc_sig, int num_bits, const char
*pc_name, ... );
void fsdb_add_signal_pointer( p_fsdb_scope p_scope, pp_fsdb pp_sig, int num_bits, const char
```

*pc_name, ...);
void fsdb_update(double d_new_time);

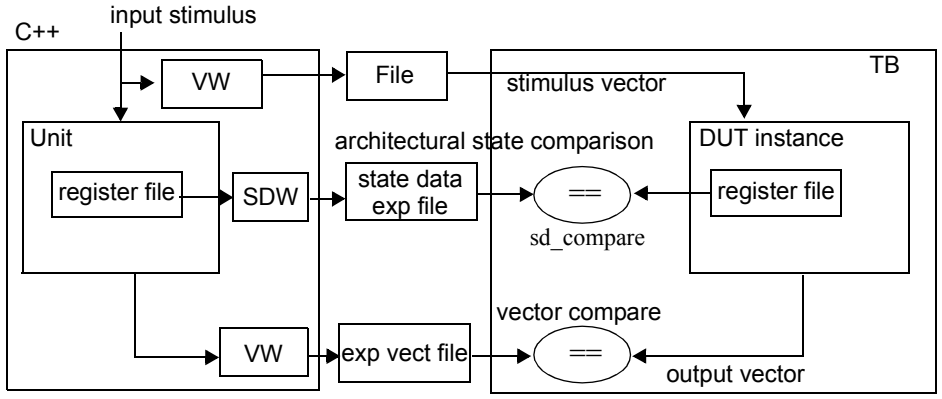
fsdb_dump_file is a file that the RTL testbench creates and is displayed in a waveform viewer;
fsdb_dump_enable - turns on the creation of the waveform;
p_fsdb_scope - level in the simulation model to dump waves form. This is an instance name expressed using an HID.
fsdb_set_top_scope - The top level of the scope to dump waves from Dump. All signals to waves below the top scope.
fsdb_add_scope - Adds another scope to the list of scopes.
fsdb_add_signal - adds a signal to the output wave file.
fsdb_add_signal_pointer

1.3.3 Random Temporal interface Testbench

The Temporal interface Testbench uses stimulus and expect vectors from the C++ Simulator to drive the module under test. The temporal interface Testbench “randomly“ asserts the control signals for the module under test to vary the relationships between the control signals.

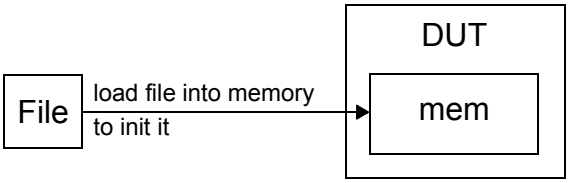
1.3.3.0.1 Architectural state comparison testbench

FIGURE 1.14 Architectural state comparison



1.3.3.0.2 Memory initialization

FIGURE 1.15 Memory Initialization



User can select memory initialization file at run time.

```
.mem_init mi_b;//
.module_name (b);
.memory_name (mem);
.mem_init_file (MIBASE_BIN);
.mem_init_base(MIBASE_BIN);
mem_id (374);// unique id for memory initialization
.random_select (PID);
RANDOM SEED_1
```

Vectors and states are assigned names for the purposes of selecting the vectors and states. A state is a collection of values which is stored in a set of state elements in the C++ simulator and the RTL design. The state changes either based on the clock or a transaction. A testbench can compare the state from the C++ sim with the state from the RTL design at each clock edge or transaction. An example of state comparison which is used would be the comparison of architecturally visible registers in processors, C++ and RTL models. The testbench performs a comparison between the two sets of architectural states.

1.3.4 Verification hierarchy

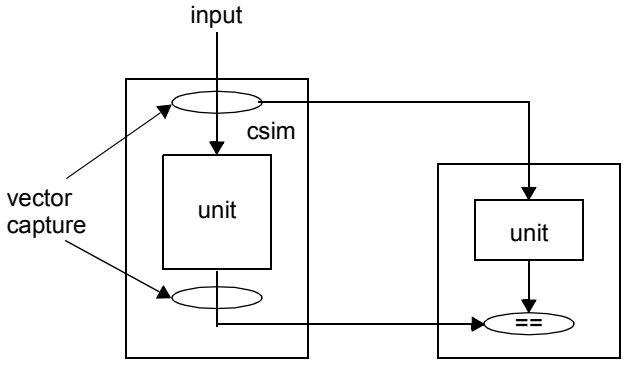
There are 6 compare levels in the verification hierarchy:

TABLE 1.5 Verification hierarchy

C++ Simulator	Vsim Testbenches
System	System
System	Chip
System	Unit
Chip	Chip
Chip	Unit
Unit	Unit

The diagrams in this section show the C++ Simulator and RTL compare levels.

FIGURE 1.16 Socket based Testbench

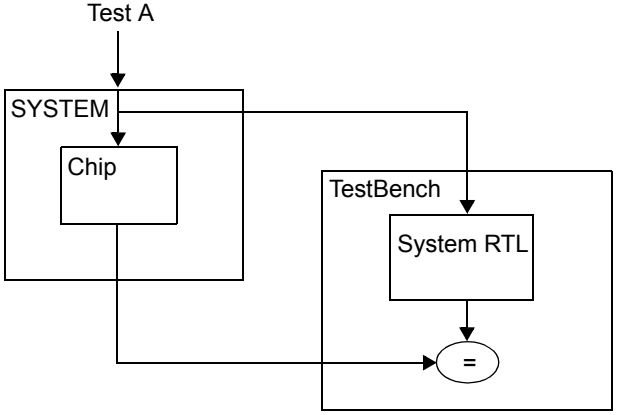


1.3.5 C++ Simulator vs. Vsim Testbenches

- System vs System
- System/Chip vs Chip
- System/Chip/Unit vs Unit
- Chip vs Chip
- Chip/Unit vs Unit
- Unit vs Unit

1.3.5.1 System vs System level simulation

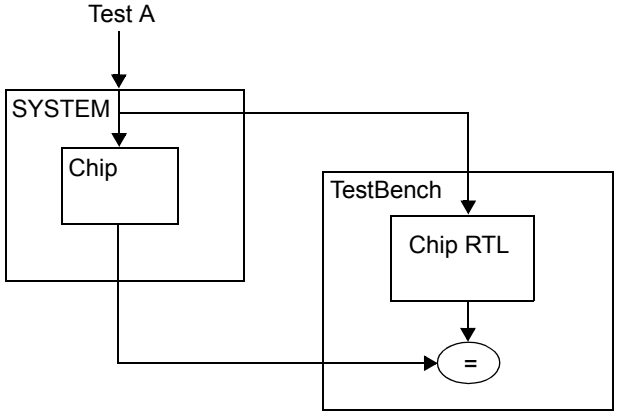
FIGURE 1.17 System and System



1.3.5.2 System/Chip vs Chip level simulation

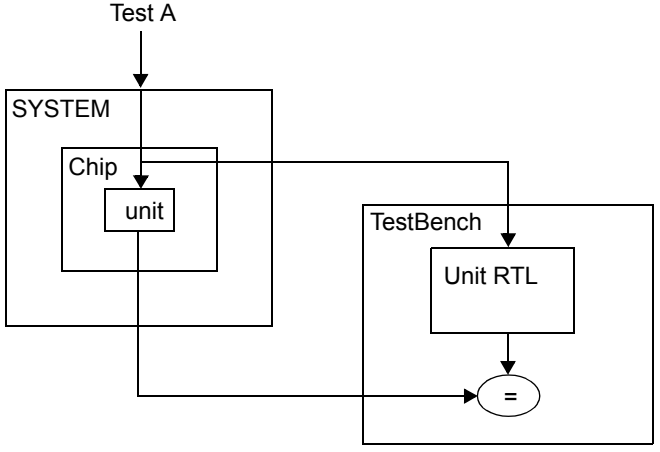
CREATING VECTORS FOR A VERILOG CHIP SIMULATOR

FIGURE 1.18 C++ Simulator testbench with chip level , stimulus and comparison



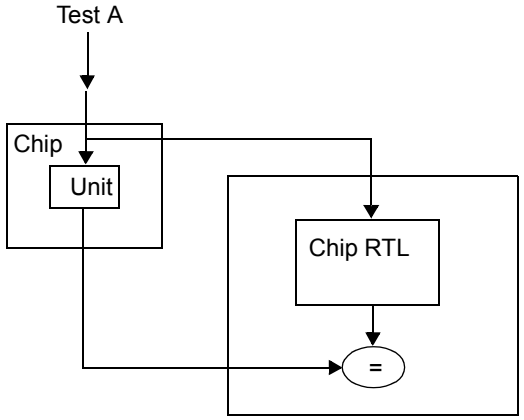
1.3.5.3 System/Chip/Unit vs Unit level Simulation

FIGURE 1.19 System and module



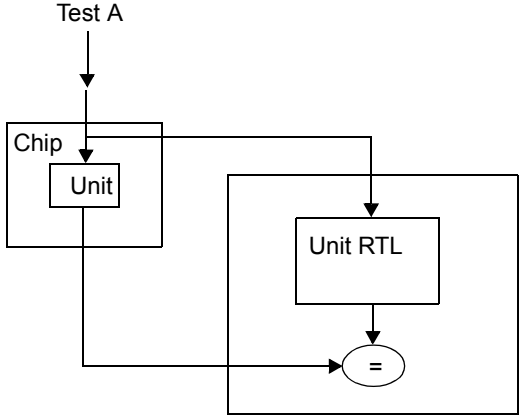
1.3.5.4 Chip vs Chip level simulation

FIGURE 1.20 Chip vs Chip



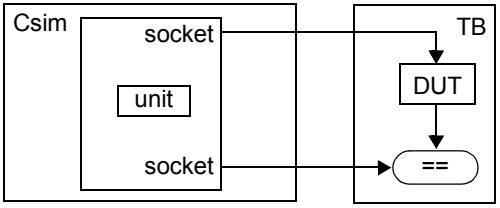
1.3.5.5 Chip/Unit vs Unit level simulation

FIGURE 1.21 C++ Simulator chip and RTL module



1.3.5.6 Unit vs unit simulation

FIGURE 1.22 Unit and unit



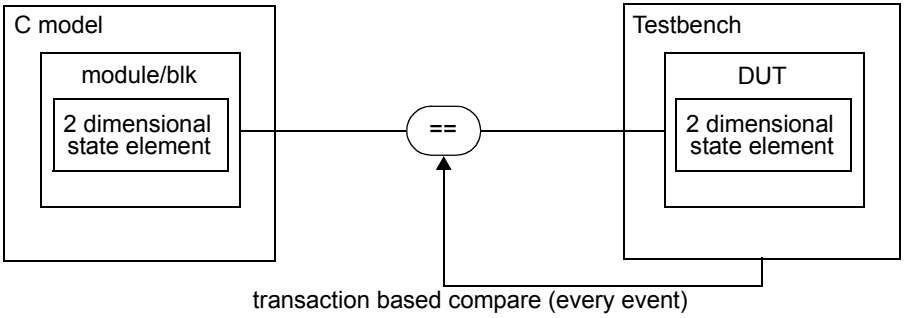
1.3.6 Testbenches

Test Bench vector bit length checker
If there is a bit string being loaded into a register from a testbench then there should be a length check to make sure that the two are of equal lenghts

1.3.7 Testbench Types

1.3.8 Architectural state comparisons

FIGURE 1.23 Testbench State element comparison

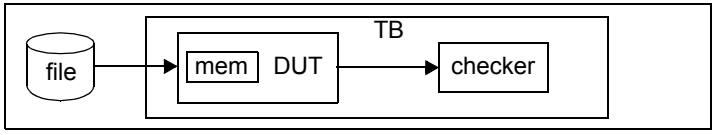


The C++ simulator will “backdoor“ load the state elements in the C++ and RTL memories prior to starting the simulation.

1.3.8.1 RTL selfchecking testbench

Use command line arguments to verilog simulator to specify the file name which the testbench should read in DUT memory

FIGURE 1.24 Self checking testbench



A self checking testbench loads a program into the DUT memory and then reads out the result from the DUT. No expected result vectors are needed. A checksum or other success indicator is required.

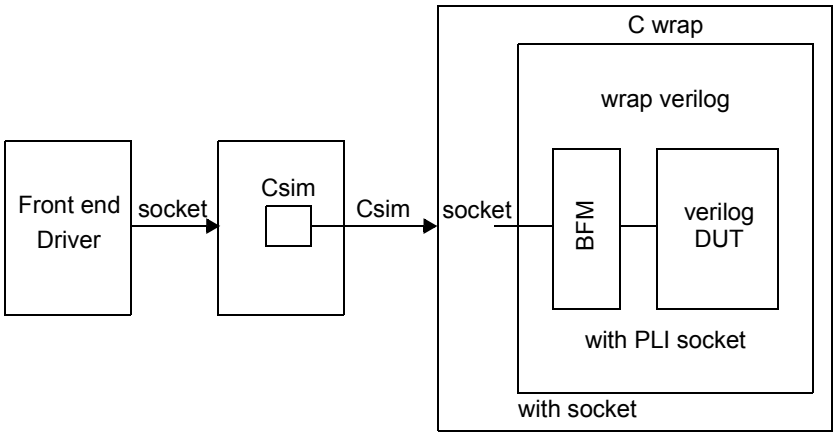
1.3.8.2 Socket based Testbench

Socket connection

FIGURE 1.25 Socket based communication

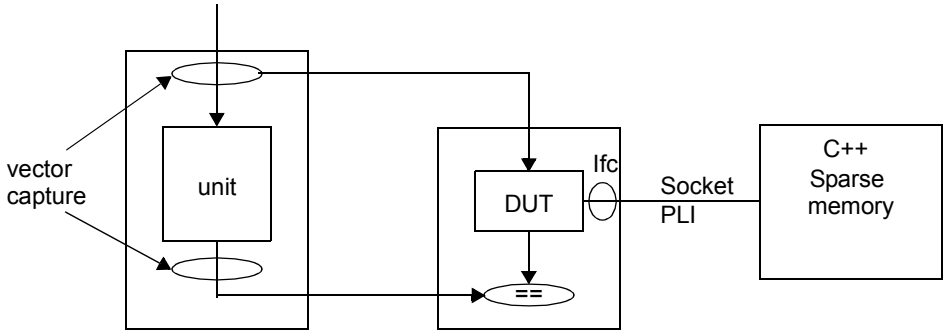
socket setup file connects signals in the C++ Simulator and vsim

FIGURE 1.26 Front end



The C++ Simulator must be able to drive the Verilog DUT at the system chip or unit levels

FIGURE 1.27 Socket based Testbench



Sparse memory model - used to model large memories by only allocating memory for the parts of the moemory where a memory word is used.

1.3.9 Testbench reports

Report files are used to describe the performance of the DUT.

1.3.9.1 Incident Report

Detailing for any test that failed. The report consist of all detailes of the incident such as actual and expected results, when it failed, and any suporting evidence that will help in it resolution. The report will also include an assessment of the impact upon testing of an incident.

1.3.9.2 Summary Report

Testbench generates the summary report for each state data and vector instance in the testbench. The report includes the number of transactions, number of matches and mismatches.
The report records what testing was done and how long it took. It also provides information about the quality of the Design under test and statistics delivered by the Incident Report. This report is used to indicate whether the DUT is fit for the purpose according to wheteher or not it has met acceptance citeria defined by project stakehorders.

1.3.10 Design Under test

The design under test (DUT) is an instantiation of a design inside a testbench.

1.3.11 Testbench clocks

There can be testbench driven clock, meaning that the clock signal is generated inside the testbench, or socket driven clock that are generated outside the testbench and connected throw a socket.
If the testbench has only one clock object then the testbench clock object's clk output is connected automatically to the other objects in the testbench with a clock. If two or more clock objects are added to the testbench then the clock objects need to be explicitly added to each testbench object.

```
initial begin
    clk = 1'b0;
end
    always #2 clk  = ~clk;
```

1.3.12 Testbench reset

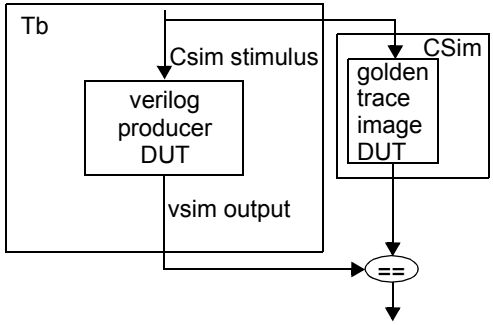
Same as in the case of the clock signal, if the testbench has only one reset signal then the testbench reset signal is connected automatically to the other objects in the testbench with a reset. If two or more reset signals are added to the testbench then the reset signals need to be explicitly added to each testbench object.

```
initial begin
    reset_ = 1'b1;
    @(posedge clk);
    reset_ = 1'b0;
    @(posedge clk);
    reset_ = 1'b1;
end
```

1.3.13 Golden Units

A golden unit or golden device is an ideal example of device against which all other devices are tested and judged. The Golden units are described using CSim and RTL units are compared against those golden units.

FIGURE 1.28 Module input stimulus and expected output !!to be fixed



The C++ Simulator -vm argument tells the C++ Simulator to create and connect PLI's for each of the verilog DUT's interfaces. C++ Simulator and verilog PLI's
Each legal verilog name maps to a set of PLI's in the C++ Simulator
The C++ Simulator contains a lookup table which maps the verilog name to a set of stimulus and expect vectors. Each vector which is specified in the table is mapped to a PLI

1.3.14 Testbench Components

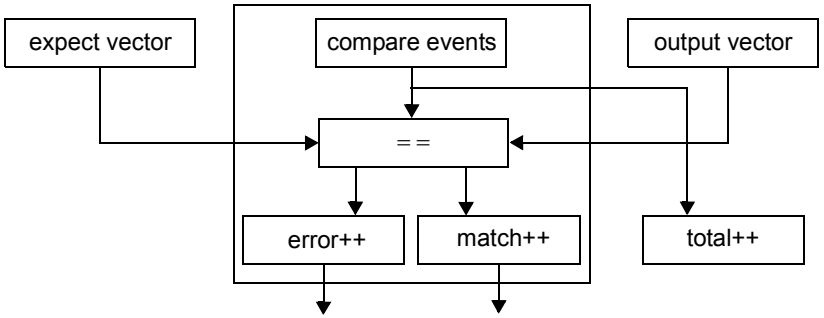
1.3.15 Compare logic components

- compare event logic
- compare logic
- error counter
- match counter

```
write (<addr>,<data>);
retval=read (<addr>);
compare (retval,<data>);
```

1.3.15.1 Compare logic

FIGURE 1.29 Compare logic



(PLI socket read memb + code to apply/compare vector DUT)
The vector name and the vector formats are the same in the following components

- documentation
- c model unit
- c model vector
- testbench vector reader
- DUT
- HW tester logic analyser
- regression environment (scripts + testlists)

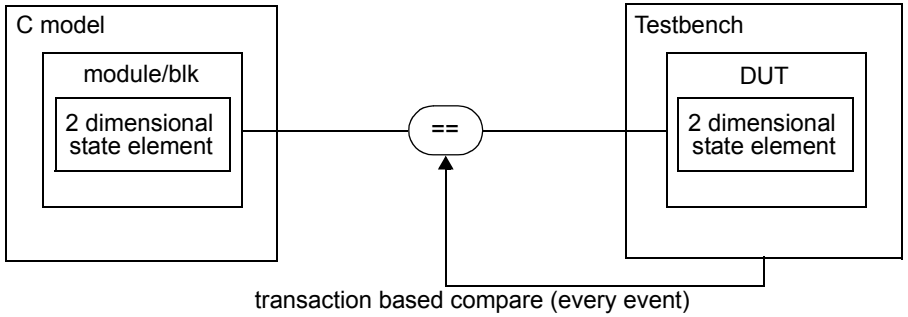
1.3.15.2 Error Counter

An error counter accumulates the total number of mismatches between the expected vectors and DUT output vector.

1.3.16 Architectural state comparisons

Figure 1.30 shows comparing the architectural state of the C++ Simulator state elements

FIGURE 1.30 Testbench State element comparison

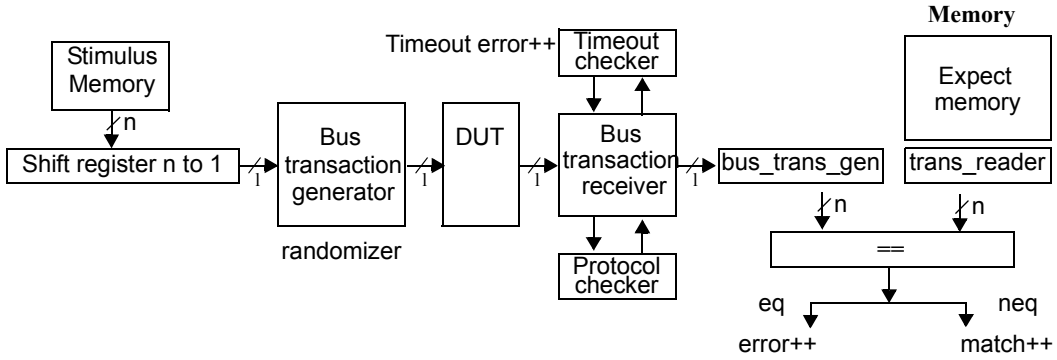


The C++ simulator will “backdoor“ load the state elements in the C++ and RTL memories prior to starting the simulation.

1.3.16.1 Random stall & valid injection

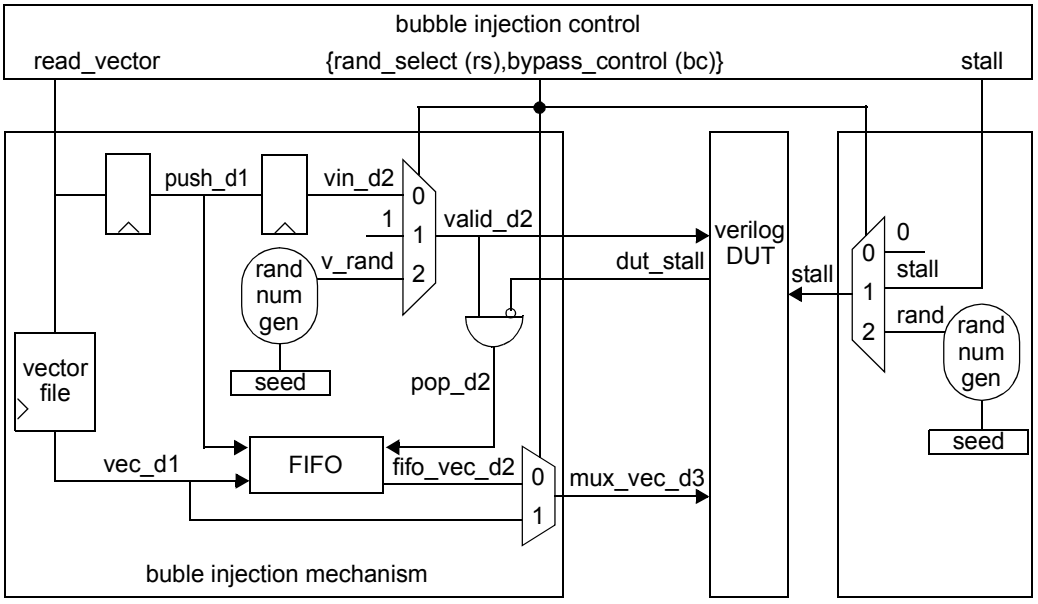
The temporal interface testbench injects bubbles (random vectors) between stimulus vectors which drive the model under test. It also “gathers“ transactions into a queue and sends the transactions in bursts to the DUT. The temporal interface testbench “randomly“ asserts the stall signals to the DUT to stress the output. Stalls can be shut off to test the max mem throughput of the DUT.

FIGURE 1.31 Serial bus testbench



The bubble injection mechanism randomly inserts delays into the input vector stream by stalling the DUT’s input interface.

FIGURE 1.32 Design for valid creation



C Based Testbench for Raw vector/state stimulus and expected result hardware emulation control. Delays between test vector application <bit_vector> <#<num>> Take the delay status out of the tb. Use a counter which uses a clk with a granularity of limit of the delay.

Testbenches use plus args to load the directory name and test name

```
    `include <tasks>.vh;
    get_test_dir(directory);
    get_test_name(test_name);
    // use plus args to read in the <dir>/<test>
    // +dir directory +testname test_name
    write (<addr>,<data>);
    retval=read (<addr>);
compare (retval,<data>);
```