CHAPTER 1 CSL Verification Components

TABLE 1.1 Chapter Overview

1.1 Definitions
1.2 CSL Verification Components Overview
1.3 CSL Verification Components Concepts
1.4 CSL Verification Components Examples

1.1 Definitions

TABLE 1.2 Definitions

Term used	Definition
Chip	Chip Level
Module	Module Level
tb	Test Bench
dut	Design Under Test
Vector	A set of values associated with an ordered set of signals captured at time n
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
Golden Vector	A known good vector
tx	transmit
rx	receive
alg_csim	Algorithmic C/C++ Simulator
beh_csim	Behavioral C/C++ Simulator
csim	Behavioral C++ Simulator
rsim	RTL Simulator

Term used	Definition
bfm	Bus Functional Model
state data	The state of a circuit at a certein moment in time. The content of all memory cells inside the circuit.
expected state data	The expected state of a circuit

1.2 CSL Verification Components Overview

The CSL verification components are vectors and state data. These components are used to verify if a DUT it is running correctly by comparing the expected vectors and/or state data with the output vectors and/or state data of the DUT. The verification components can be used in different ways to validate the DUT's response to stimulus. One way is loading the expected verification components into the testbench's memory and compare it with the DUT's output at runtime. Another way is to store the DUT's response in to a file and then compare this file with the expected verification components file.

A vector is a collection of a set of signal values in an ordered list. The vectors are stored in a file. Each line of the file contains one vector. The vector file is generated by a C++ simulator or driver code. The cslc produces a vector writer C++ class which the user instantiates in the user's C++ simulator or driver. The vector writer class first writes out the vector file header and then writes out each individual vector to the vector file when an event is received.

An ordered list of signals is assigned to the vector.

csl_Vector {

```
    csl_port a;
    csl_port b(3);
    csl_port c(4);
    csl_port d(3);
    csl_port e(2);
    csl_port f(5);

    csl_vector v = {a,b,c,d,e,f}
```

The cslc will check that the connecvity objects in the vector are all from the same pipestage. If the connectivity objects are not from the same pipestage cslc will issue a warning or error.

1.2.0.1

The output vector format depedends on the width of the signal and the radix selected for the vector. A three bit signal in binary is represented by 3 digits from the set {0,1} producing a binary vector

with the values of each signal in the following format:

```
<a[0:0]>_<b[2:0]>_<c[3:0]>_<d[2:0]>_<e[1:0]>_<f[4:0]>
```

A signal in hexidecimal is represented by 1 digit from the set {[0-9]A-F}. If the signal modulus 4 is not equal to zero then the bits have to be packed together and the values have to be shifted and ored to produce the

```
vector output value.
            upper lower
            bit   bit
```

TABLE 1.3

	name	pos	pos
1	x[0:0]	17	16
3	y[2:0]	16	14
4	x[3:0]	13	10
3	q[2:0]	9	7
2	p[1:0]	6	5
5	r[4:0]	4	0

```
<x[0:0]>_<y[2:0]>_<z[3:0]>_<q[2:0]>_<p[1:0]>_<r[4:0]>
```

Example vector:

```
csl_vector v = {x,y,z,q};
binary vector separated into signal bit groups
each signal is shown as a group of bits separated by an underscore.
// x   y   z    q
01 01 111 0111 // binary signal values
01_01_111_0111 // binary vector output

all binary bits are smashed together
01011110111    // concat the value bits together
text

convert the binary representation to a hexidecimal representation
010 1111 0111  // group in 4-bit chunks
  2    F    7  // coinvert 4 bit chunks to hex numbers
2F7            // create hexidecimal vector output
```

1.3 CSL Verification Components Concepts

1.3.1 Vector concepts

A verification vector is a set of values associated with a set of signals. A verification vector is a collection of numbers concatenated together. The order of the vector is determined by the order of the signals.
A set of verification vectors is comprised of verification vectors captured based on an event.
The format of the vector is defined using the CSL vector specification. The vector is specified as an ordered list of signal names and ranges.

Vectors which drive a RTL DUT interface are called stimulus vectors. Vectors which are compared to RTL DUT output signals are called expected results vectors. A vector is associated with an unit interface. The signal which is in the vector can be a subset of the unit's interface.

1.3.1.1 Verification vector Extraction Points

A verification vector is extracted from a file, C/C++ model simulator, or RTL model simulation. A file can contain a set of *golden verification vectors*. A golden set of verification vectors is a known good set of vectors, by known good set we mean vectors which describe the interface of an RTL model. The RTL models' vectors describe the correct behaviour of the RTL models interface. A C/C++ simulator or RTL testbench can extract the vectors from the file. Similarly the verification vectors can be extracted from a C/C++ simulator or RTL DUT.

1.3.1.2 Verification vector Insertion Points

A verification vector is written to a file, inserted into a C/C++ model simulator or RTL model simulation at any module interface point. A file can contain a set of golden verification vectors. A C/C++ simulator or RTL testbench can extract the vectors from the file. Similarly the verification vectors can be extracted from a C/C++ simulator or RTL DUT.

1.3.1.3 Verification vector Event Types

TABLE 1.4

	Event types	Definition
CA	Cycle_Accurate	depends on clock cycle
TA	TRANSACTION_ACCURATE	test_vectors generated or captured based on events
PA	PIPESTAGE_ACCURATE	test_vectors geneated based on the same latency as the RTL but the combinational logic may be in different pipestages

Associate an event with the test_vector type if the test_vector type is CA then associate a clock with the test_vector event else if the test_vector type is TA then associate an expression with the test_vector event else if the test_vector type is PA then associate a clock with the test_vector event.?
Example:
test_vector_object_name "." event_type "(" test_event_type ")" ";"

1.3.1.3.1 Turning on the vector record mechanism

Events are defined so that we can indicate points at which to turn on and turn off the verification vector recording. For example a test signal could be used to turn on or off the verification vector recording. If the verification vector record mechanism is turned on then when a verification vector event occurs (i.e. a clock edge or another signal or combinational logic equation is asserted) the vector is captured.

1.3.1.3.2 Capture mechanism

The verification vector is "captured" based on the verification vector type. Each time that a verification vector event occurs the associated verification vector is recorded in a file.The verification vector recording is turned on and off based on another set of events such as after reset or a unit is enabled.

1.3.1.3.3 Verification vector Capture Mechanism Type

There are several different types of verification vector capture mechanisms. The verification vector can be written out to a file. The verification vector can be written out to a socket interface. The verification vector can be written out to a function call. We call the capture mechanism the verification vector connection type. The verification vector is communicated from one object to another object using a medium such as a file, RPC call, or function call.

1.3.1.4 Using verification vectors

The simulators can run standalone using an executable test(no need for external stimulus) or the

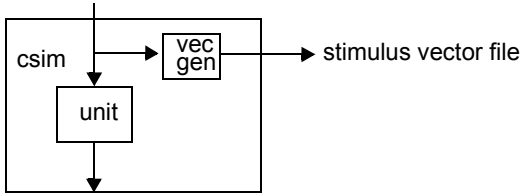
simulators can be driven with verification vectors. A standalone test, also known as a diagnostic test is executed on the design, computes the result, compares the computed result to a value(s) stored in the test. Verification vectors are extracted from verification vector extraction points which are defined in the CSL. The C++ Simulator can create a vector file and then the testbench loads it into a memory. From ths memory the vectors can be used to feed the DUT's inputs or to be compared with the results of the DUT. A socket PLI can be used in connection with the C++ Simulator or a file Input/ Output operation can take place to read more lines from a file at once.

1.3.2 Verification vectors

A verification vector is a set of values that is used to stimulate or verify an interface. The values match signals on the interface. The input stimulus vectors are generated on every clock. The output test-vectors are generated on every clock or on an event.

1.3.2.1 Stimulus vector generator

FIGURE 1.1 Adding the vector generators for stimulus vector file

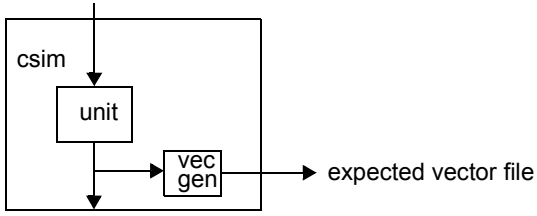


vec_gen is a module that extracts the vectors from the inputs of the C++ module at each simulation clock. The stimulus vectors are written to a file. During testbench simulation, the stimulus vectors are loaded into the vector memory in the testbench. From there they will be fed to the DUT's inputs.

1.3.2.2 Expect vector generator

The second vec gen is a module which extracts the expected vectors from the outputs at each event (clock or transaction).

FIGURE 1.2 Adding the vector generators for expected vector file



1.3.2.3 C++ Simulator Vector generators

C++ Simulator generates the stimulus and expected vectors. The vectors are written in vector files.

FIGURE 1.3 Vector extraction and vector file creation

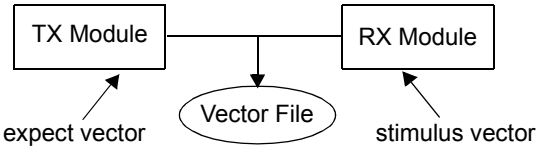
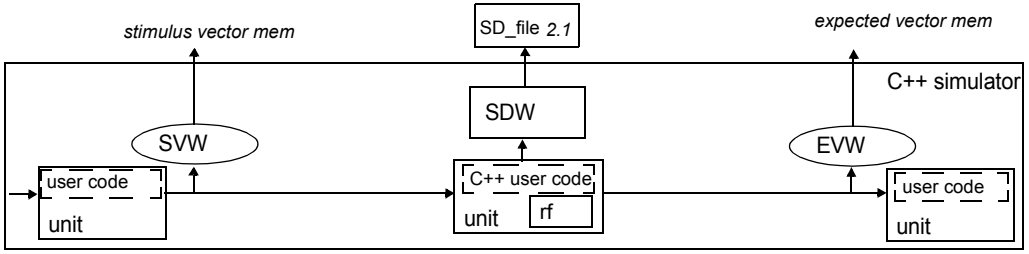


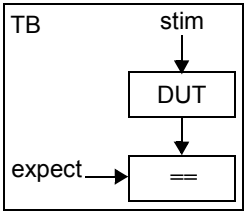
FIGURE 1.4 C++ Simulator



1.3.2.4 CSL vector Stim/Expect vector usage

DUT with stimulus vector(s) driving the DUT inputs and expected vectors compared against the DUT outputs

FIGURE 1.5 Stim/ Expect vector usage

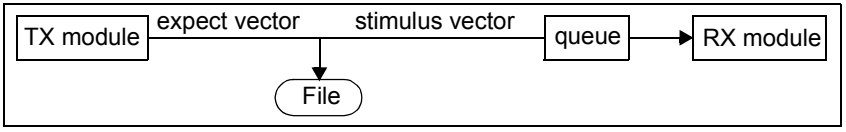


1.3.2.5 Vector file creation

The C++ simulator generates vector files or drives a socket PLI which is connected to the test-bench. Stim and expect vectors can be applied using:

- memories in the TB which are loaded with files using verilog readmem{b,h} or file I/O
- a C++ simulator that can run concurrently with the testbench and generate stim/ expect vectors which are applied to the DUT using a socket PLI

FIGURE 1.6 Creating vector files



1.3.2.6 VC file format

A vector file contains vector lines which are in either of the two formats:

- opcode_signal values separated by underscores
- valid_signal values separated by underscores

TABLE 1.5 Vector file sections

Section	Description
comment	contains the time and date the vector was created, start of header
header	contains the vector id,vector version, number of vectors, start of header and end of header.
vectors	

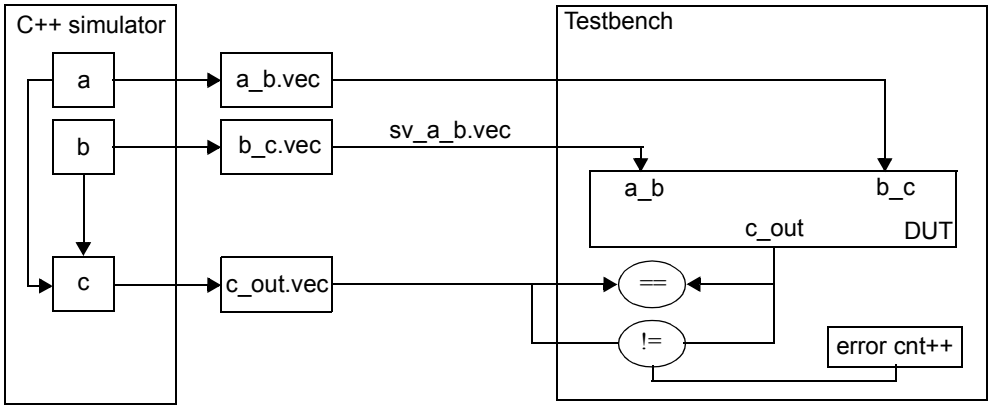
Each line in the *comment* section starts with "//". These lines are ignored by the testbench.
The *header* contains informations on the vector file. It contains on the first line an id (number) wich the testbench uses to verify that the correct vector has been read in. The id is set automatically. The user can set this id explicitly. On the second line of the header the version is specified.Then the number of vectors in the file is specified. The testbench needs this value and saves it. When the vector count equals the *NUMBER_VECTORS* then the simulation is over. The fourth line of the header is the start of vector header, then the end of vector header.The next line is a vector.
After the header, each line represents a vector value. The vector section of the file ends on the end of the file.

TABLE 1.6 Vector File Header

Line no.	Content
1	VEC_ID_NUM
2	VEC_VERSION
3	NUMBER_VECTORS
4	START_OF_HEADER
5	END_OF_HEADER

This testbench will compare the contents of the memories against the memory state read in from a file or PLI on every cycle or event.

FIGURE 1.7 Multiple inputs stimulus files



CSL testbench specifications are compiled by clsc to create RTL testbenches. Testbenches are generated based on csl_vector_definitions and csl_testbench statements . The figure above depicts the function of a testbench. The DUT (Design under Test) is tested in the testbench using the stimulus and expected vectors. The C++ simulator creates stimulus to drive the DUT and vectors to compare against the output of the DUT. The testbench can also perform state data comparison as well. The C++ simulator will generate the state of a memory at each event and either store the state in a file or send the state to the testbench via a PLI socket.

Example:

```
initial begin
   for (tb_b_mem_i=0; tb_b_mem_i<2048; tb_b_mem_i=tb_b_mem_i+1) begin
      b.mem[tb_b_mem_i]= init_val; // b is instantiated in the tb
   end
   $readmemb ("b_mem",b.mem); // load the contents of the file b_mem
                              // into the memory b.mem
end
```

For each vector in the testbench a memory is declared

```
reg [VEC_A_B_WIDTH:0] [0: VEC_A_B_MAX_NUM ]
$readmemh {""}
compare v_a_b_vec sv_a_b
```

where n equals number of signals in thre vector

Each event triggers the read_en and the a vector is read out of the vector memory

```
always @ (posedge ph_clk && mci.pu_reader_start) begin
if (compare_event_b_c!) begin
vec_bc_mem_adder=vec_bc_mem_adder+1;
vec_a_b_data=vec_a_b_mem  [vec_bc_mem_adder];
if (vec_bc_mem_adder==MAX) vec_b_c_last=1'b1;
```