cadence

Modeling for Stimulus Generation EZ-Start Guide

August 2006

Cadence Design Systems, Inc., 555 River Oaks Parkway, San Jose, CA 95134, USA

Modeling for Stimulus Generation EZ-Start Guide

The procedures described in this document are deliberately broad and generic. Your specific design might require procedures that are slightly different from those described here. However, most of the concepts described in this document are valid for various application domains.

About EZ-Start Guides

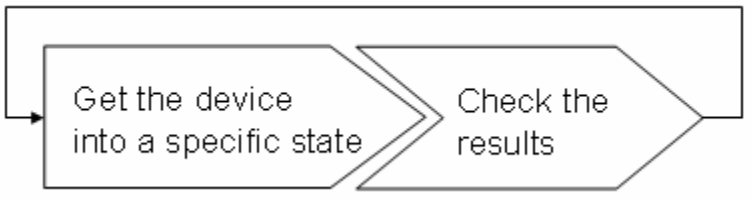
EZ-Start guides are provided by Cadence Design Systems as entry point tutorials for various technologies. EZ-Start guides are meant to quickly review high-level concepts of a specific technology, and allow the user to independently experience and explore it. For a deeper understanding of how to architect a quality verification environment, consult a Cadence methodology specialist or refer to the *Plan-to-Closure Methodology User Guide*.

Introduction

In general, there are three essential components to efficient verification: stimulus generation, coverage, and checking. Cadence Design Systems, Inc. provides companion EZ-Start packages for each of these three areas. In this EZ-Start package, you will learn how to model data items for stimulus generation.

A typical verification process (as illustrated below) involves repeatedly getting a device into a specific state, applying stimulus to the design, and then checking the results of the simulation.

Figure 1 Verification Process



Given the size and complexity of today's designs, the verification process introduces several challenges surrounding stimulus generation:

How do you identify every corner-case state?

How do you target the simulation for both internal and external scenarios?
How do you uncover bugs that are buried within the design?

This document describes how, by using SystemVerilog constructs, you can model your data items so that they address each of these challenges.

Note: You can apply the concepts covered in this document to different domains. For example, you can use these concepts for data-com where you have packets, cells, and frames; for graphical devices, where you generate register operations for different configurations; or, for CPU verification where you need to generate instructions and CPU interrupts.

Coverage-Driven Verification (CDV)

Coverage-driven verification (CDV) is a Cadence methodology that enables efficient verification. In general, CDV involves:

Random generation
Self checking
Metrics to track progress

The following rules incorporate the three aspects of CDV listed above. These rules are crucial in creating an implementation strategy that results in productive verification.

Organize your verification goals right away. These goals can include a combination of functional code and assertion coverage. Organizing these plans upfront ensures that your designs are verified quickly and thoroughly, and lets you track the verification progress until verification convergence.

The testbench must be smart enough to automatically generate legal stimuli. In such a testbench, the various tests are layered on top of the infrastructure and touch only specific areas. This results in more focused testing, thus making the tests shorter and easier to write, read, and maintain.

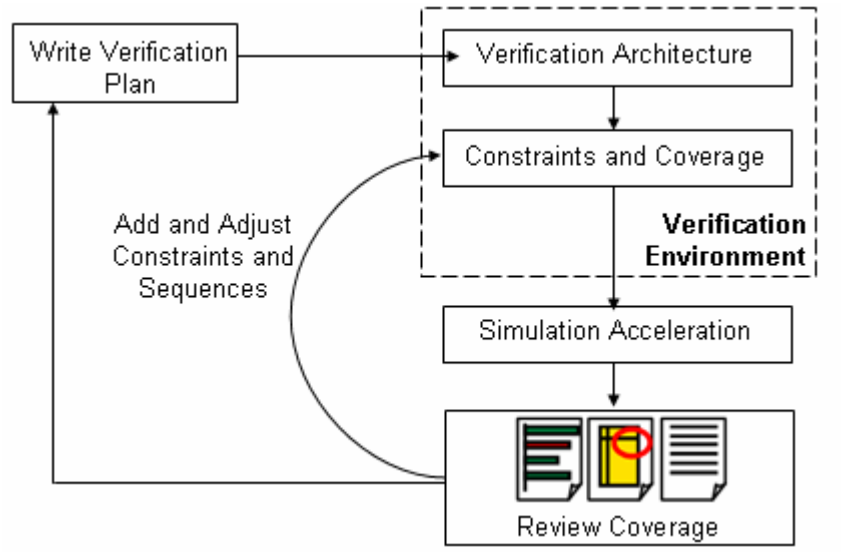
The infrastructure must allow for the creation of fully-directed tests that explicitly describe the injected stimuli and random tests. The level of randomization is a function of the defined constraints.

Do not overconstrain your tests. Your tests must be as random as possible, so that the generation engine is free to explore all values without any biased assumptions. Raising the level of randomization in your tests can help find unanticipated bugs.

Verification must start with random tests. Limit directed tests to cover only the remaining untested functionality. Automatic test generation decreases the amount of time spent creating tests manually.

The following figure illustrates a typical CDV flow, and gives the general process.

Figure 2 CDV Flow



Note: This figure gives just an overview of the CDV process. For more information on CDV and the Cadence tools that enable this process, refer to the *Plan-to-Closure Methodology User Guide*.

Modeling Stimuli Data Items

When modeling interaction within a testbench, you should be working at the transaction level. Working at this level has the following advantages:

1. Many tests involve transaction-level control. For example, a test can call for two consecutive packets that are sent to the same address.
2. Transactions hide a lot of protocol-specific details and allow you to control a scenario, while reducing the knowledge required to create tests.
3. Transactions can encapsulate assumptions and attribute interdependencies.

Note: Since the term *transaction* is broad, this document refers to generated traffic as *data items*. Data items include: packets, cells, frames for data-com devices, instructions or interrupts for CPU verification, and bus transfers or bursts for modeling buses. The following sections discuss how to model data items using SystemVerilog classes and constraints.

Examples 1-5 are also provided in ASCII form. To run these examples, open the `instruction.sv` file located in the `/examples` directory provided with this document.

Using Classes when Modeling Data Items

SystemVerilog introduces a new `class` construct, which is used in object-oriented (OO) programming. A class is a user-defined data type that can encapsulate data members, methods, and other properties. The collection of the data members and class properties define the functionality and characteristics of an object.

The following is an example of a CPU instruction that consists of an opcode and two operands. The first operand is a CPU register value, and the second is a byte.

Example 1 CPU Instruction Example

```
1  package instruction_pkg; // Package encapsulates user-defined types
2
3  typedef enum {REG0, REG1, REG2, REG3} register_e;
4  typedef enum {ADD, ADDI, SUB, SUBI, JMP, JMPC, CALL,RETURN}opcode_e;
5
6  class instruction_c;
7     rand opcode_e opcode;
8     rand register_e operand1;
9     rand byte operand2;
10
11    function void do_print(); // Encapsulates data and functionality
12       $display("Instruction: opcode=%s, register=%s, operand2=%h",
13                  opcode.name(), operand1.name(), operand2);
14    endfunction
15
16 endclass: instruction_c
17
18 endpackage: instruction_pkg
```

In this example:

Line 1	Uses a package called `instruction_pkg` to encapsulate the user-defined types. Packages avoid name-space cluttering and enable reuse.
Lines 3 and 4	Uses enumerated data types to group and name items. Enumerated types make the code easier to read and to debug. **Note:** This document uses the `*_e` naming convention for enumerated data types and `*_c` for classes.
Lines 7 to 9	Uses the `rand` keyword to declare the instruction attributes as

random variables.

Line 11 — Implements a service function called `do_print()` for the class data items. This function will print the class data.

Now that we have our class defined, we can use this definition to generate our desired number of instructions. The following code generates one hundred random CPU instructions.

Example 2 Generating CPU Instructions

```
1  import instruction_pkg::*;
2
3  program test;
4     instruction_c cur_instruction;
5
6     initial
7         begin
8         cur_instruction = new;
9         for (int i = 0; i < 100; i++)
10            begin
11                assert(cur_instruction.randomize());
12                cur_instruction.do_print();
13            end
14        end
15 endprogram
```

In this example:

Line 1 — Imports the package defined in Example 1 in order to access the package's attributes and declarations.

Line 3 — Uses a program block to encapsulate our test information.

Line 4 — Before you can use the class defined in Example 1, you need to create a variable using that class as its data type. This line creates a variable called `cur_instruction`, which is a handle to class instance `instruction_c`.

Line 8 — Before you can use the variable `cur_instruction`, it must be initialized. This line uses the `new` function to initialize the variable `cur_instruction` to an instance of class `instruction_c`.

Line 11 — Uses the built-in `randomize()` method to generate random values for all active random variables within an object. The `randomize()`

method returns 1 if it successfully assigns all of the random variables within an object to a valid value. Otherwise, it returns 0. This line also uses the `assert` construct to ensure that the correct values are assigned to the generated instruction.

Using Constraints when Modeling Data Items

Modeling real data items in real protocol rarely involves pure random attribute generation. There are dependencies between fields that make some data items illegal. For example, to make a legal cell, its checksum must be equal to an arithmetic calculation of its other fields. To capture such dependencies, you can use constraints. A constraint describes a property of a field and directs a solver, which can be embedded in a simulator, to process the constraint and choose a value that satisfies the properties of the constraint.

This section provides short examples for the following types of constraints:

Specification constraints—Constraints derived from a specification that defines the legal data items.

Default and preference constraints—Constraints that specify that a particular combination of legal values occur more than others. For example, you might have a default distribution of 95% legal packets and 5% illegal packets. You can override this default specification using tests, if you need to meet a certain verification goal.

Test constraints—Constraints that are layered on top of an existing environment in order to steer the generation towards unexplored areas.

Specification Constraints

For example, a specification indicates that "`JMP` and `JPMC` must use the value `REG0` as their branching address." The following illustrates the implementation of this specification.

Example 3 Specification Constraint

```
class instruction_c;
   rand opcode_e opcode;
   rand register_e operand1;
   rand byte operand2;
   rand inst_kind_e kind;

   //Spec: JMP and JMPC should use the address from REG0
   constraint jmp_reg0 {opcode inside {JMP, JMPC} -> operand1 == REG0;}
```

```
endclass: instruction_c
```

Note: Specification constraints can be overridden during testing. Overriding constraints is useful in cases where the specification describes the expected behavior for corrupted input or to test device recovery.

Default and Preference Constraints

For efficient verification, you can create constraints that emulate regular traffic behavior or specify how you want to distribute traffic. For example, the typical traffic for an Ethernet usually contains only a few erroneous packets. The following implements this type of behavior. Specifically, the following code specifies that `CALL` and `RETURN` opcodes occur less often than other opcodes.

Example 4 Default and Preference Constraints

```
class instruction_c;
   rand opcode_e opcode;
   rand register_e operand1;
   rand byte operand2;
   rand inst_kind_e kind;

   // By default, the following provides less CALL and RETURN opcodes
   constraint less_call_ret {
            opcode dist {[ADD:JMPC]:=10,CALL:=1,RETURN:=1};}
endclass: instruction_c
```

Test Constraints

When you are modeling data items, it is uselful to add extra control fields that provide a user-friendly interface. *Control fields* are class attributes that are not a part of the physical data time, but can simplify test creation and are useful for coverage. For example, a packet might have a control field that determines if a cyclic redundant check (CRC) result is legal. Although this Boolean field is not sent to the design under test (DUT), this field can help test writers specify how legal and illegal packets should be distributed—without having to know the underlying packet implementation. Control fields can help make tests shorter and easier to write and understand.

For example, the following is taken from specification and functional coverage planning:

“There are two types of instructions, arithmetic and flow control. The user will most likely want only one type of instructuion in a given scenario. How can the developer environment assist the test writer to iteratively achieve this preference?”

To address this, a new type is added to the instruction package and is then used in the class to differentiate between the different instruction types.

Example 5 Control Fields

```
package instruction_pkg;
...
// New control field for instruction kind
typedef enum {ARITHMETIC, FLOW_CONTROL, ... } inst_kind_e;
...
class instruction_c;
   rand opcode_e opcode;
   rand register_e operand1;
   rand byte operand2;
   rand inst_kind_e kind;  // Not part of the physical instruction data

   constraint kind_knob {
     (kind == ARITHMETIC) -> opcode inside {ADDI, ADD, SUB,SUBI};
     (kind == FLOW_CONTROL) -> opcode inside {JMP, JMPC, CALL, RETURN};
   }
...   // Other instruction class definitions
endclass: instruction_c
...
endpackage: instruction_pkg
```

Now, the test writer can add a constraint that calls for only arithmetic instructions or that further specifies the distribution.

Modeling for Stimuli Generation Lab

In this section, you will use the concepts discussed in the previous sections to model a data item.

You should be using IUS version 05.81 p002 or higher. To verify that you are using the correct version of the simulator, use the following command:

```
% ncverilog -version
```

You should see a message similar to the following:

```
    TOOL: ncverilog  05.70-p002
```

You can go to http://downloads.cadence.com/ to find and download the latest IUS release.

About the Data Item (BTM Cell)

You will be modeling a data item called a BTM cell. The BTM cell has 53 bytes:

The first byte is for the header of the cell.

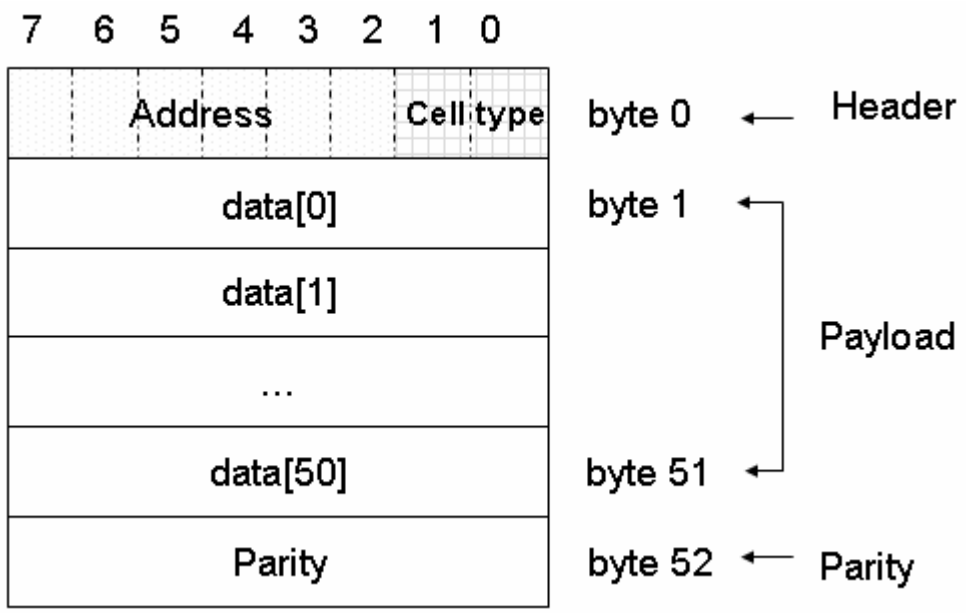
- The first two bits of the header contain the cell type (this can be data, control or reserved).
- The next six bits of the header contain the random address.

The next 51 bytes are for data or payload.

The last byte is a parity byte that contains a parity calculation of the first 52 bytes.

The following figure illustrates the structure of the BTM cell.

Figure 3 BTM Cell Structure



Modeling the BTM Cell

This section walks you through the steps necessary to model this BTM cell.

To model this BTM cell:

1. Create a file and give it an `.sv` extension. For example, `btm.sv`.
2. Within the `btm.sv` file, define a package that contains:
   a. An enumeration data type that captures the following data types: `DATA`, `CONTROL`, and `RESERVED`.

b. A class called `btm_cell_c` . This class has the following fields:

- Enumerated cell types
- A six-bit address
- 408-bit data
- A parity byte

Note: All of these fields must be randomized. Therefore, tag all of these fields with the `rand` keyword.

c. A `do_print() void` function to that prints the cell's type, address, and parity fields.

3. Import the package definition to the global scope.
4. Define a program that allocates one cell.
5. Using a loop, randomize the values and call the `do_print()` void function for ten cells.
6. Save the file.
7. Compile the file.

Use the following command to compile and run SystemVerilog code using batch mode:

```
% ncverilog +sv btm.sv
```

Or, use the following command to compile and run IUS in GUI mode:

```
% ncverilog +sv +gui btm.sv
```

The solution for this exercise is in the `exercise1.sv` file of the `/solutions` directory, which was provided with this document.

Constraining the BTM Cell

Now that you have modeled the structure of the BTM cell, you need to add the following constraints:

A constraint that avoids the `RESERVED` cell type.

Note: Your constraint should not specify default reserved cell type.

A constraint that constrains the value of the address so that it is byte aligned.

A constraint that constrains the last two bits of the address such that they are zero.

The solution for this exercise is in the `exercise2.sv` file of the `/solutions` directory, which was provided with this document.