cādence™

Incisive® Enterprise Specman Elite® Testbench **Tutorial**

Version 8.1

Contents

1 Introduction

This chapter covers the following Specman tutorial concepts and tasks:

- "Tutorial Overview" on page 1-1
- "Tutorial Goals" on page 1-2
- "Setting up the Tutorial Environment" on page 1-4
- "Document Conventions" on page 1-5

Tutorial Overview

Incisive® Enterprise Specman Elite® Testbench provides benefits that result in:

- Reductions in the time and resources required for verification
- Improvements in product quality

The Specman system automates verification processes, provides functional coverage analysis, and raises the level of abstraction for functional coverage analysis from the RTL to the architectural/specification level. This means that you can:

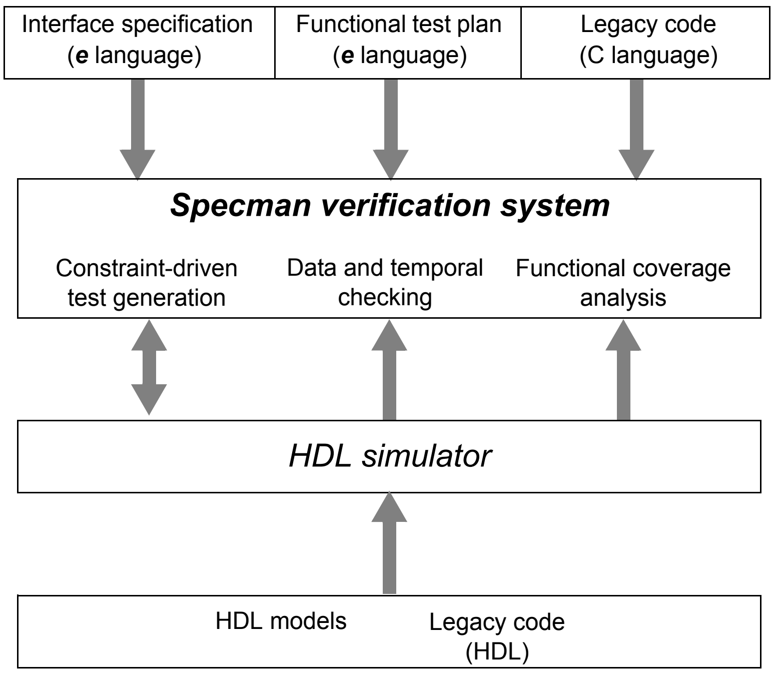
- Easily capture your design specifications to set up an accurate and appropriate verification environment
- Quickly and effectively create as many tests as you need
- Create self-checking modules that include protocols checking
- Accurately identify when your verification cycle is complete

The Specman system provides three main enabling technologies that enhance your productivity:

- **Constraint-driven test generation—**You control automatic test generation by capturing constraints from the interface specifications and the functional test plan. Capturing the constraints is easy and straightforward.
- **Data and temporal checking—**You can create self-checking modules that ensure data correctness and temporal conformance. For data checking, you can use a reference model or a rule-based approach.
- **Functional coverage analysis—**You avoid creating redundant tests that waste simulation cycles, because you can measure the progress of your verification effort against a functional test plan.

Figure 1-1 shows the main component technologies of the Specman system and its interface with an HDL simulator.

Figure 1-1 The Specman System Automates Verification


Interface specification (**e** language)
Functional test plan (**e** language)
Legacy code (C language)
Specman verification system
Constraint-driven test generation
Data and temporal checking
Functional coverage analysis
HDL simulator
HDL models
Legacy code (HDL)

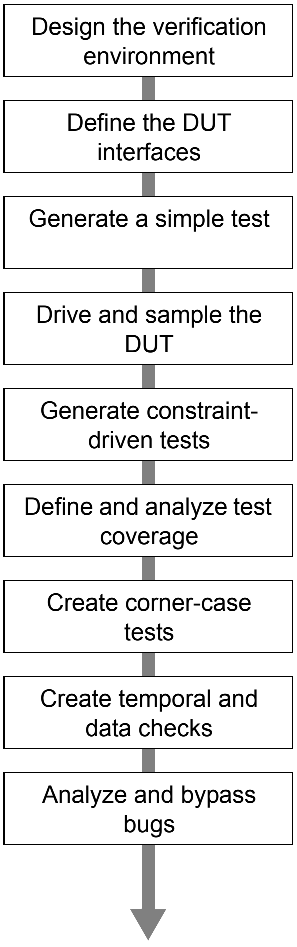
Tutorial Goals

This tutorial is designed for use with Specman Version 8.1.

The goal of this tutorial is to give you first-hand experience in how the Specman system effectively addresses functional verification challenges.

As you work through the tutorial, you follow the process described in Figure 1-2. The tutorial uses the Specman system to create a verification environment for a simple CPU design.

Figure 1-2 Tutorial Verification Task Flow


Design the verification environment
Define the DUT interfaces
Generate a simple test
Drive and sample the DUT
Generate constraint-driven tests
Define and analyze test coverage
Create corner-case tests
Create temporal and data checks
Analyze and bypass bugs

Setting up the Tutorial Environment

Before starting the design verification task flow shown in Figure 1-2 on page 1-3, you must prepare to use Specman and set up the tutorial environment.

Accessing the Specman Software

This tutorial assumes that you have access to the version of Specman that is indicated in the tutorial title. For example, if you are using the Specman Tutorial Version 8.1, this tutorial assumes you are running it with Specman Version 8.1.

If the Specman version you need is not installed in your environment, please contact your Cadence sales representative for help.

For information about installing Specman and preparing to use it, see the *Installation and Configuration Guide*.

Downloading and Installing the Tutorial

The tutorial files consist of the PDF version of this tutorial plus the ***e*** files that you use to run the tutorial. The PDF file and the ***e*** files are contained in a tarkit that is available on SourceLink. The name of the tutorial tarkit is sn_*release_number*_tutorial.tar.gz, for example sn_8.1_tutorial.tar.gz.

To download and install the tutorial, follow these steps:

1. Log on to http://sourcelink.cadence.com.
2. Select the "Specman" product and search on the keyword "tutorial".
3. Display the document for installing the Specman tutorial.
4. Follow the instructions to download and untar the tarfile.

   This creates the following directory structure:

   ```
   sn_release_number_tutorial/
   sn_release_number_tutorial.pdf
   ```

5. Change to the tutorial ***e*** file directory.

   ```
   % cd sn_release_number_tutorial/cpu/e
   ```

6. List the directory contents to see the file structure.

   ```
   % ls *

   gold:
   ```

```
cpu_bypass.e            cpu_dut.e             cpu_tst1.e
cpu_checker.e           cpu_instr.e           cpu_tst2.e
cpu_cover.e             cpu_misc.e            cpu_tst3.e
cpu_cover_extend.e      cpu_refmodel.e        regression_A.ecov
cpu_drive.e             cpu_top.e             regression_B.ecov

src:
cpu_bypass.e            cpu_dut.e             cpu_tst1.e
cpu_checker.e           cpu_instr.e           cpu_tst2.e
cpu_cover.e             cpu_misc.e            cpu_tst3.e
cpu_cover_extend.e      cpu_refmodel.e        regression_A.ecov
cpu_drive.e             cpu_top.e             regression_B.ecov
```

You can see that there are two sets of files. As you work through this tutorial, you will be modifying the files in the *src* directory. If you have trouble making the modifications correctly, you can view or use the files in the *gold* directory. The files in the *gold* directory are complete and correct.

Now that the files are installed, you are ready to proceed with the design verification task flow shown in Figure 1-2 on page 1-3. To start the first step in that flow, turn to Chapter 2 "Understanding the Environment". In this chapter, you review the DUT specifications and functional test plan for the CPU design and define the overall verification environment.

Document Conventions

This tutorial uses the document conventions described in Table 1-1.

Table 1-1 Document Conventions

Visual Cue	Meaning
`courier`	Specman or HDL code. For example, `keep opcode in [ADD, ADDI];`
`courier bold`	Text that you need to type exactly as it appears to complete a procedure or modify a file.
bold	In text, bold indicates Specman keywords. For example, in the phrase "the **verilog trace** statement," **verilog** and **trace** are keywords.
%	In examples that show commands being entered, the **%** symbol indicates the UNIX prompt.
SN>	In examples that show commands being entered in the Specman system, SN> indicates the Specman prompt.

2 Understanding the Environment

Goals for this Chapter

This tutorial uses a simple CPU design to illustrate the benefits of using the Specman system for functional verification. This chapter introduces the overall verification environment for the tutorial CPU design, based on the design specifications, interface specifications, and the functional test plan.

What You Will Learn

Part of the productivity gain provided by the Specman system derives from the ease with which you can capture the specifications and functional test plan in executable form. In this chapter, you become familiar with the design specifications, the interface specifications, and the functional test plan for the CPU design. You also become familiar with the overall CPU verification environment.

The following sections provide brief descriptions of the:

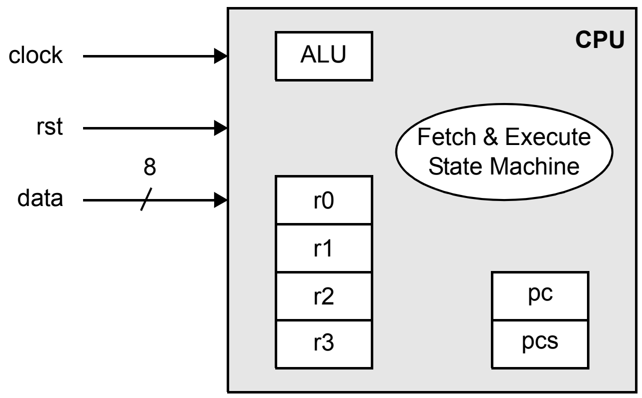
- Design specifications
- Interface specifications
- Functional test plan
- Overall verification environment

For more detailed information on the CPU instructions, the CPU interface, and the CPU's internal registers, see Appendix A "Design Specifications for the CPU".

The Design Specifications

The device under test (DUT) is an 8-bit CPU with a reduced instruction set (Figure 2-1).

Figure 2-1 CPU Block-Level Diagram



Available Instructions

Arithmetic:
ADD, ADDI, SUB, SUBI

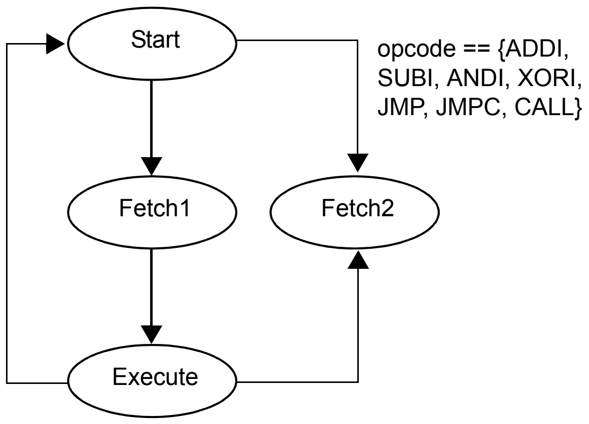
Logic:
AND, ANDI, XOR, XORI

Control Flow:
JMP, JMPC, CALL, RET

No-Operation:
NOP

The state machine diagram for the CPU is shown in Figure 2-2. The second fetch cycle is only for *immediate* instructions and for instructions that control execution flow.

Figure 2-2 CPU State Machine Diagram



There is a 1-bit signal associated with each state, *exec*, *fetch2*, *fetch1*, *strt*. If no reset occurs, the *fetch1* signal must be asserted exactly one cycle after entering the execute state.

The Interface Specifications

All instructions have a 4-bit opcode and two operands. The first operand specifies one of four 4-bit registers internal to the CPU. The second operand depends on the type of instruction:

- **Register instructions—**The second operand specifies another one of the four internal registers.

Figure 2-3 Register Instruction

byte	1							
bit	7	6	5	4	3	2	1	0
	opcode				op1		op2	

- **Immediate instructions—**The second operand is an 8-bit value. When the opcode is of type JMP, JMPC, or CALL, this operand must be a 4-bit memory location.

Figure 2-4 Immediate Instruction

byte	1								2							
bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
	opcode				op1		don't care		op2							

The Functional Test Plan

We need to create a series of tests that will result in adequate test coverage for most aspects of the design, including some rare corner cases. There will be three tests in this series.

Test 1

Test Objective

A simple go/no-go confirming that the verification environment is working properly.

Test Specifications

- Generate five instructions.
- Use either the ADD or ADDI opcode.

- Set op1 to REG0.
- Set op2 either to REG1 for a register instruction or to value 0x5 for an immediate instruction.

Test 2

Test Objective

Multiple random variations on a test gains high percentage coverage on commonly executed instructions.

Test Specifications

- Use constraints to direct random testing towards the more common arithmetic and logic operations rather than the control flow operations.
- Run the test many times, each time with a different random seed.

Test 3

Test Objective

Generation of a corner case test scenario that exercises JMPC opcode when the carry bit is asserted. Note that it is difficult to efficiently cover this scenario by purely random or purely directed tests.

Test Specifications

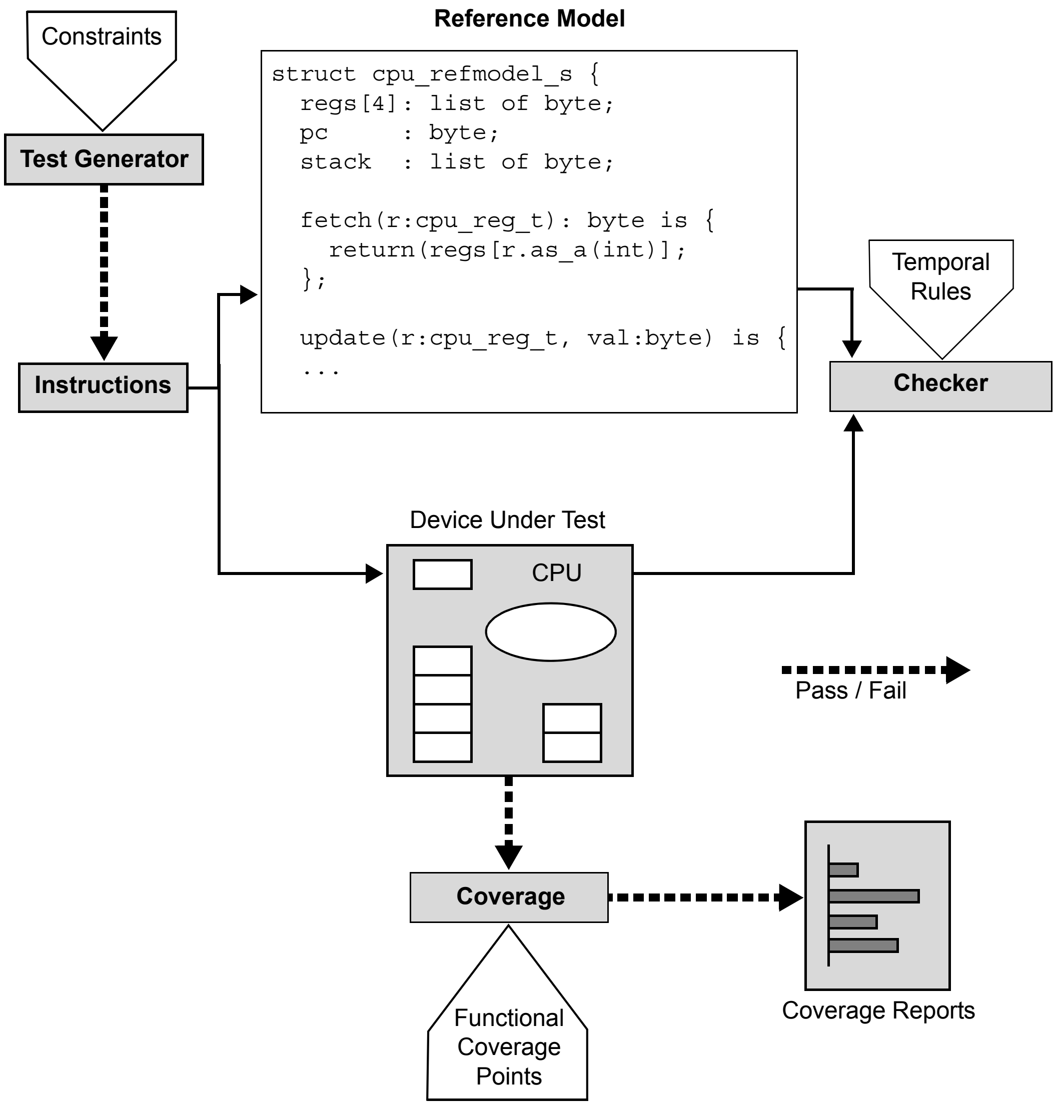
- Generate many arithmetic opcodes to increase the chances of carry bit assertion.
- Monitor the DUT and use on-the-fly generation to generate many JMPC opcodes when the carry signal is high.

Overview of the Verification Environment

The overall test strategy, shown in Figure 2-5, includes the following objectives:

- Constrain the Specman test generator to creation of valid CPU instructions.
- Compare the program counters in the CPU to those in a reference model.
- Define temporal rules to check the DUT behavior.
- Define coverage points for state machine transitions and instructions.

Figure 2-5 Design Verification Environment Block-Level Diagram



```
struct cpu_refmodel_s {
  regs[4]: list of byte;
  pc     : byte;
  stack  : list of byte;

  fetch(r:cpu_reg_t): byte is {
    return(regs[r.as_a(int)];
  };

  update(r:cpu_reg_t, val:byte) is {
  ...
```

Because the focus of this tutorial is the Specman system, we do not include an HDL simulator. Rather than instantiating an HDL DUT, we model the DUT in ***e*** and simulate it in Specman. The process you use to drive and sample the DUT in ***e*** is exactly the same as a DUT in HDL.

Now you are ready to create the first piece of the verification environment, the CPU instruction stream.

3 Creating the CPU Instruction Structure

Goals for this Chapter

The first task in the verification process is to set up the verification environment. In this chapter you start creating the environment by defining the inputs to the design, the CPU instructions.

What You Will Learn

In this chapter you learn how to create a data structure and define specification constraints that enable the Specman system to generate a legal instruction stream. By the end of this chapter, you will have created the core structure for the CPU instructions. This core structure will be used and extended in subsequent chapters to create the tests.

As you work through this chapter, you gain experience with one of the Specman system's enabling features—**easy specification capture**. Using a few constructs from the ***e*** language, you define the legal CPU instructions exactly as they are described in the interface specifications.

This chapter introduces the ***e*** constructs shown in Table 3-1.

Table 3-1 New Constructs Used in this Chapter

Construct	How the Construct is Used
<'...'>	Marks the beginning and end of ***e*** code.
struct	Creates a data structure to hold the CPU instructions.
extend	Adds the data structure containing the CPU instructions to the Specman system of data structures.

Table 3-1 New Constructs Used in this Chapter (continued)

Construct	How the Construct is Used
list of	Creates an array or list without having to keep track of pointers or allocate memory.
type	Defines an enumerated data type for the CPU instructions.
bits	Defines the width of an enumerated type.
keep	Specifies rules or constraints for the instruction fields.
when	Implements conditional constraints on the possible values of the instruction fields.

To create the CPU instruction structure, you must:

- Capture the interface specifications
- Create a list of instructions

The following sections explain how to perform these tasks.

Capturing the Specifications

In this task, you create the data structure for the instruction stream and constrain the test generator to generate only legal CPU instructions. Individual tests that you create later can constrain the generator even further to test some particular functionality of the CPU.

For a complete description of the legal CPU instructions, refer to Appendix A “Design Specifications for the CPU”.

Procedure

To capture the design specifications in ***e***:

1. Make a new working directory and copy the *src/cpu_instr.e* file to the working directory.
2. Open the *cpu_instr.e* file in an editor.

   The first part of the file has a summary of the design specifications for the CPU instructions.

```
   File name  cpu_instr.e
       Title  Basic structure of CPU instructions
     Project  Specman Tutorial
   Developer  Verisity Design, Inc.
Date created  12-1-2002
 Description  This module defines the basic structure of CPU instructions,
              according to the design and interface specifications.

               All instructions are defined as:
                 Opcode   Operand1   Operand2

               There are 2 types of instructions:

              Register Instruction:
                bit | 7 6 5 4 | 3 2 | 1 0 |
                    | opcode  | op1 | op2 |
                                      (reg)
                      :
              Immediate Instruction:
               byte | 1                     | 2               |
                bit | 7 6 5 4 | 3 2 | 1 0   | 7 6 5 4 3 2 1 0 |
                    | opcode  | op1 | don't | op2             |
                                    | care  |

              Register instructions are:
                ADD, SUB, AND, XOR, RET, NOP

              Immediate instructions are:
                ADDI, SUBI, ANDI, XORI, JMP, JMPC, CALL

              Registers are REG0, REG1, REG2, REG3
```

3. Find the portion of the file that starts with the `<'` ***e*** code delineator and review the constructs:

defines the legal opcodes as an enumerated type	`<'`
	`type cpu_opcode_t: [ // Opcodes`
	`    ADD, ADDI, SUB, SUBI,`
	`    AND, ANDI, XOR, XORI,`
	`    JMP, JMPC, CALL, RET,`
	`    NOP`
	`] (bits: 4);`

defines the internal registers	`type cpu_reg_t: [ // Register names`
	`    REG0, REG1, REG2, REG3`
	`] (bits:2);`

when complete, this structure defines a valid CPU instruction
 	
// indicates that rest of line is a comment
	`struct cpu_instr_s {`
	
	`    // defines 2nd op of reg instruction`
	
	`    // defines 2nd op of imm instruction`
	
	`    // defines legal opcodes for reg instr`
	
	`    // defines legal opcodes for imm instr`
	
	`    // ensures 4-bit addressing scheme`
	
	`};`

when complete, this construct adds the CPU instruction set to the Specman system	`extend sys {`
	`    // creates the stream of instructions`
	
	`};`
	`'>`

4. Define two fields in the *cpu_instr_s* structure, one to hold the opcode and one to hold the first operand.

   Use the enumerated types, *cpu_opcode_t* and *cpu_reg_t*, to define the types of these fields. To indicate that the Specman system must drive the values generated for these fields into the DUT, place a **%** character in front of the field name. You will see how this **%** character facilitates packing automation in Chapter 5 "Driving and Sampling the DUT".

   The structure definition should now look like this:

add these two lines into the file

```
struct cpu_instr_s {
    %opcode    :cpu_opcode_t;
    %op1       :cpu_reg_t;

    // defines 2nd op of reg instruction
.
.
.
};
```

5. Define a field for the second operand.

   The second operand is either a 2-bit register or an 8-bit memory location, depending on the kind of instruction, so you need to define a single field (*kind*) that specifies the two kinds of instructions. Because the generated values for *kind* are not driven into the design, do not put a **%** in front of the field name.

add this line to define the field 'kind' and define an enumerated type at the same time

```
struct cpu_instr_s {
    %opcode    :cpu_opcode_t;
    %op1       :cpu_reg_t;
    kind       :[imm, reg];

    // defines 2nd op of reg instruction
.
.
.
};
```

6. Define the conditions under which the second operand is a register and those under which it is a byte of data.

   You can use the **when** construct to do this.

```
struct cpu_instr_s {
    %opcode    :cpu_opcode_t;
    %op1       :cpu_reg_t;
    kind       :[imm, reg];

    // defines 2nd op of reg instruction
    when reg cpu_instr_s {
        %op2   :cpu_reg_t;
    };

    // defines 2nd op of imm instruction
    when imm cpu_instr_s {
        %op2   :byte;
    };
.
.
.
};
```

7. Constrain the opcodes for immediate instructions and register instructions to the proper values.

   Whenever the opcode is one of the register opcodes, then the *kind* field must be *reg*. Whenever the opcode is one of the immediate opcodes, then the *kind* field must be *imm*. You can use the **keep** construct with the implication operator => to easily create these complex constraints.

```
struct cpu_instr_s {
.
.
.
    // defines legal opcodes for reg instr
    keep opcode in [ADD, SUB, AND, XOR, RET, NOP]
        => kind == reg;

    // defines legal opcodes for imm instr
    keep opcode in [ADDI, SUBI, ANDI, XORI, JMP, JMPC, CALL]
        => kind == imm;

    // ensures 4-bit addressing scheme

};
```

8. Constrain the second operand to a valid memory location (less than 16) when the instruction is immediate.

   You can use the **when** construct together with **keep** and => to create this constraint.

```
struct cpu_instr_s {
.
.
.
    // ensures 4-bit addressing scheme
    when imm cpu_instr_s {
        keep opcode in [JMP, JMPC, CALL] => op2 < 16;
    };
};
```

9. Save the *cpu_instr.e* file.

   Now you have finished defining a legal CPU instruction.

Creating the List of Instructions

In this task, you create a CPU instruction structure by extending the Specman system (**sys**) to include a list of CPU instructions. **sys** is a built-in Specman structure that defines a generic verification environment.

Procedure

To create the list of instructions:

1. Within the same *cpu_instr.e* file, find the lines of code that extend the Specman system:

```
extend sys {
    // creates a stream of instructions

};
```

2. Create a field for the instruction data of type *cpu_instr_s*.

   When defining a field that is an array or a list, you must precede the field type with the keyword **list of**.

```
extend sys {
    // creates a stream of instructions
    !instrs: list of cpu_instr_s;
};
```

The exclamation point preceding the field name *instrs* tells the Specman system to create an empty data structure to hold the instructions. Then, each test tells the system when to generate values for the list, either before simulation (pre-run generation) or during simulation (on-the-fly generation). In this tutorial you use both types of generation.

3. Save the *cpu_instr.e* file.

   Now you have created the core definition of the CPU instructions. You are ready to extend this definition to create the first test.

4 Generating the First Test

Goals for this Chapter

In this chapter, you will generate the first test described in "The Functional Test Plan" on page 2-3. This first test is a simple test to confirm that the verification environment is set up correctly and that you can generate valid instructions for the CPU model.

You will also get a look at the Data Browser and the Generation Debugger GUIs, which are features that provide visibility into the data structure and the generation order of the ***e*** objects.

What You Will Learn

In this chapter, you learn how to create different types of tests easily by specifying test constraints in the Specman system. Test constraints direct the Specman generator to a specific test described in the functional test plan. This chapter illustrates how the Specman system can quickly generate an instruction stream. In the next chapter, you will learn how to drive this instruction stream to verify the DUT.

As you work through this chapter to create the first test, you gain experience with the following enabling features of the Specman system:

- **Extensibility—**This enables adding definitions, constraints, and methods to a struct in order to change or extend its original behavior without altering the original definition.
- **Constraint solver—**This is the core technology that intelligently resolves all specification constraints and test constraints and then generates the desired test.

This chapter shows new uses of the ***e*** constructs introduced in Chapter 3 "Creating the CPU Instruction Structure". It also introduces the Specman console menu commands shown in Table 4-1.

Note Starting in Specman release 8.1, the default console application is SimVision. The earlier Specview console is still available. In this tutorial SimVision commands are listed first and Specview commands are listed second in parenthesis. For example:

"In SimVision, click **File ›› Reload *e* Files** (in Specview, **File ›› Reload**) to reload the files for test 2."

Table 4-1 New Constructs and SimVision/Specview Menu Commands Used in this Chapter

Construct	How the Construct is Used
extend	Adds constraints to the **sys** and *cpu_instr_s* structs defined in Chapter 3 "Creating the CPU Instruction Structure".
keep	Limits the possible values of the instruction fields and the number of instructions generated for this test.
when	Defines conditional constraints.
SimVison & (Specview) Command	**How the Command is Used**
File ›› Load *e* File (File ›› Load)	Loads uncompiled ***e*** modules into the Specman system.
Verification ›› View *e* Modules (File ›› Modules)	Lists the ***e*** modules you have loaded into the Specman system.
Verification ›› Test (Test ›› Test)	Generates a test based on the constraints you specify.
Verification ›› Data Browser (Tools ›› Data Browser)	Opens the Data Browser GUI, in which you view the hierarchy of generated objects and their values.

Tip In most cases, the SimVision/Specview menu commands presented in this tutorial can be issued by clicking a button. For example, clicking SimVision's Load button (or Specview's Load button) is the same as clicking **File ›› Load *e* File** in Simvision (or **File ›› Load** in Specview). Similarly, you can click the SimVision or Specview Modules button instead of clicking **Verification ›› View *e* Modules** in SimVision (or clicking **File ›› Modules** in Specview).

The steps required to generate the first test for the CPU model are:

1. Defining the test constraints.
2. Loading the verification environment into the Specman system.
3. Generating the test.

The following sections explain how to perform these steps.

Defining the Test Constraints

The Functional Test Plan for the CPU design (see "The Functional Test Plan" on page 2-3) describes the objectives and specifications for this first test.

Test Objective

The objective is to confirm that the verification environment is working properly.

Test Specifications

To meet the test objective, the test should:

- Generate five instructions.
- Use either the ADD or ADDI opcode.
- Set op1 to REG0.
- Set op2 either to REG1 for a register instruction or to value 0x5 for an immediate instruction.

Procedure

To capture the test constraints in ***e***:

1. Copy the *src/cpu_tst1.e* to the working directory and open the *cpu_tst1.e* file in an editor.
2. Find the portion of the file that looks like this:

```
<'
import cpu_top;

extend cpu_instr_s {
    // test constraints

};

extend sys {
    // generate 5 instructions

};
.
.
.
```

3. Add lines below the comments as follows to constrain the opcode, operands, and number of instructions:

constrains the opcode and operands

constrains the number of instructions

```
<'
extend cpu_instr_s {
    //test constraints
    keep opcode in [ADD, ADDI];
    keep op1 == REG0;
    when reg cpu_instr_s { keep op2 == REG1; };
    when imm cpu_instr_s { keep op2 == 0x5; };
};

extend sys {
    //generate 5 instructions
    keep instrs.size() == 5;
};
```

4. Save the *cpu_tst1.e* file.

Loading the Verification Environment

To run the first test, you need the following files:

- **cpu_tst1.e**—imports (includes) *cpu_top.e* and contains the test constraints for the first test.
- **cpu_top.e**—imports *cpu_instr.e* and *cpu_misc.e*.
- **cpu_instr.e**—contains the definitions and specification constraints for CPU instructions.
- **cpu_misc.e**—configures settings for print and coverage display.

These files are called modules in the Specman system. Before the system can generate the test, you must load all the modules.

Procedure

To load all modules:

1. Copy the *src/cpu_top.e* file to the working directory.
2. Copy the *src/cpu_misc.e* file to the working directory.

   The working directory should now contain four files: *cpu_instr.e*, *cpu_misc.e*, *cpu_top.e*, and *cpu_tst1.e*

3. From the working directory, type the following command at the UNIX prompt to invoke Specman's graphical user interface, SimVision™:

   ```
   % specman -gui &
   ```

   Or, type the following command at the UNIX prompt to invoke Specman's graphical user interface, Specview™:

   ```
   % specview &
   ```

   Note If the Specman console (SimVision or Specview) (Figure 4-2 or Figure 4-2) does not appear, you have probably not defined the Specman environment variables correctly. To do so:

   a. Set the environment variable VRST_HOME to the Specman installation directory (the directory that contains the "components" directory).

      Example:

      ```
      VRST_HOME=/cad/tools/vpa
      ```

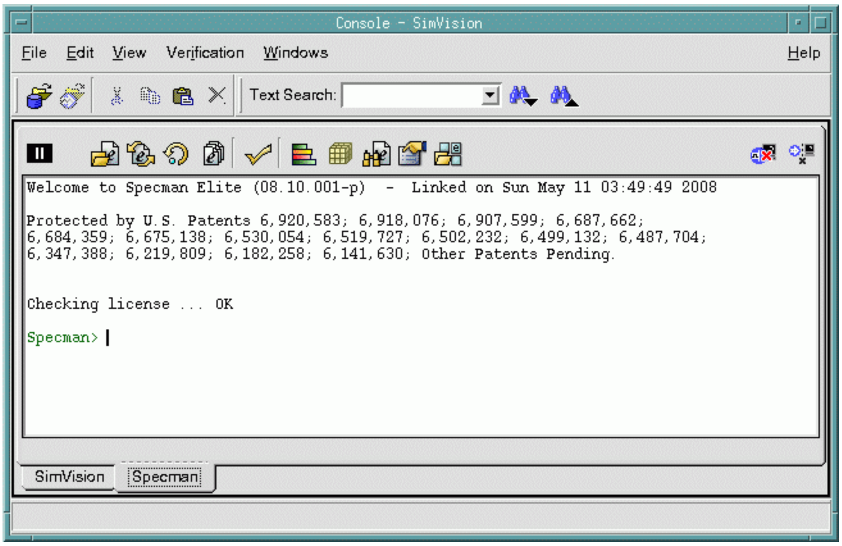
   b. Source $VRST_HOME/env.csh or $VRST_HOME/env.sh.

      Example:

      ```
      export VRST_HOM
      . $VRST_HOME/env.sh
      ```

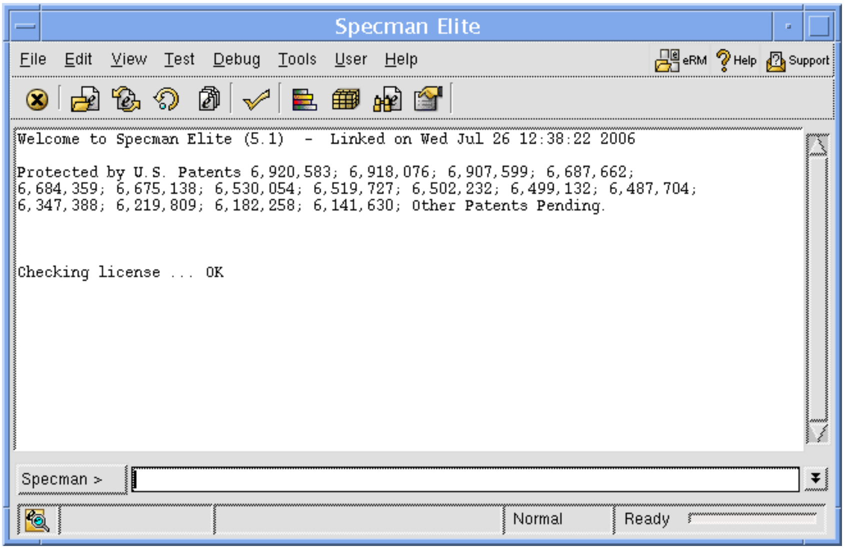
   For complete instructions for setting up Specman, see the *Installation and Configuration Guide*.

Figure 4-1 SimVison's Specman Tab



```
Welcome to Specman Elite (08.10.001-p)  -  Linked on Sun May 11 03:49:49 2008

Protected by U.S. Patents 6,920,583; 6,918,076; 6,907,599; 6,687,662;
6,684,359; 6,675,138; 6,530,054; 6,519,727; 6,502,232; 6,499,132; 6,487,704;
6,347,388; 6,219,809; 6,182,258; 6,141,630; Other Patents Pending.


Checking license ... OK

Specman>
```

Figure 4-2 Specview Main Window



```
Welcome to Specman Elite (5.1)  -  Linked on Wed Jul 26 12:38:22 2006

Protected by U.S. Patents 6,920,583; 6,918,076; 6,907,599; 6,687,662;
6,684,359; 6,675,138; 6,530,054; 6,519,727; 6,502,232; 6,499,132; 6,487,704;
6,347,388; 6,219,809; 6,182,258; 6,141,630; Other Patents Pending.


Checking license ... OK
```

4. Click **File ›› Load *e* File** (in Specview, **File ›› Load**).

   The Select A File dialog box appears.

5. In the Select A File dialog box, double-click *cpu_tst1.e* in the list of files.

   Specman automatically loads all four files contained in your working directory. In SimVision or Specview, you should see a message that looks as follows:

   ```
   Loading cpu_instr.e   (imported by cpu_top.e) ...
   Loading cpu_misc.e   (imported by cpu_top.e) ...
   Loading cpu_top.e   (imported by cpu_tst1.e) ...
   Loading cpu_tst1.e ...
   ```

Tip If the *cpu_tst1.e* file name does not appear in the dialog box, you probably did not invoke SimVision or Specview from the working directory. Use the list of directories in the dialog box to navigate to the working directory.

Tip If the *cpu_tst1.e* file does not load completely because of a syntax error, use the UNIX diff utility to compare your version of *cpu_tst1.e* to *tutorial/gold/cpu_tst1.e.* Fix the error and click the Reload button. Alternatively, you can click the blue hypertext link in the SimVision or Specview window, and the error location will be displayed in the Debugger window.

To see a list of loaded modules, click **Verification ›› *e* Modules** (in Specview, **File ›› Modules**).

There should be four modules loaded:

```
cpu_instr
cpu_misc
cpu_top
cpu_tst1
```

In the Modules window, click **File ›› Close** to close the window after you verify that all four modules are loaded.

Generating the Test

To generate the test:

1. Click **Verification ›› Generation Debugger ›› Collect Gen** (in Specview, **Tools ›› Generation Debugger ›› Collect Gen**).

   We perform this step in order to be able to view the generation order in the next procedure, "Analyzing Generation". Using **Collect Gen** is necessary when you want to use the Generation Debugger, but is not required for most Specman test runs.

2. Click **Verification ›› Test** (in Specview, **Test ›› Test)**.

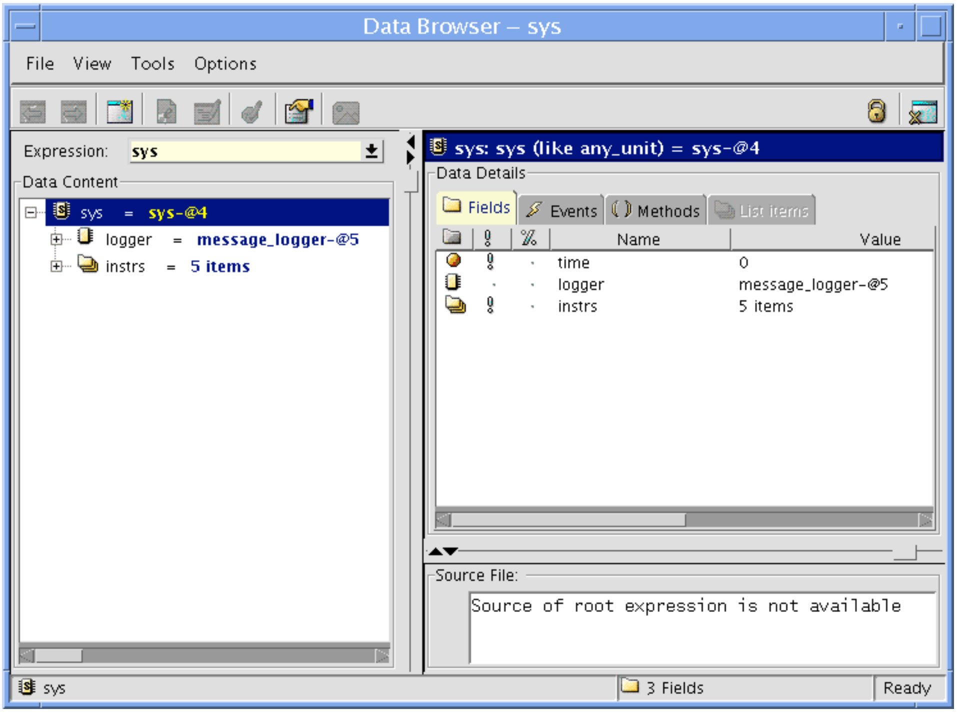
   You should see the following output in console window:

   ```
   Doing setup ...
   Generating the test using seed 1...
   ```

```
Starting the test ...
Running the test ...
No actual running requested.
Checking the test ...
Checking is complete - 0 DUT errors, 0 DUT warnings.
```

3. Click **Verification ›› Data Browser ›› Show Data Sys** (in Specview, **Tools ›› Data Browser ›› Show Data Sys**).

   The Data Browser is launched.

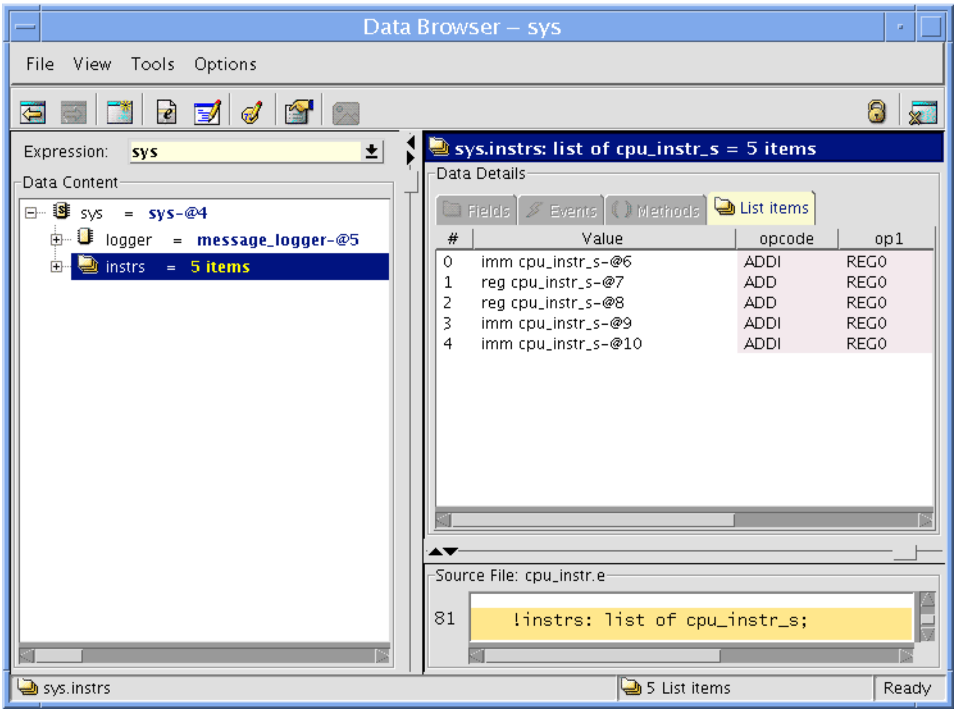


Name	Value
time	0
logger	message_logger-@5
instrs	5 items

```
Source of root expression is not available
```

4. Click the *5 items* link on the *instrs* line in the left pane.

   The list of five generated instructions appears in the top right pane.

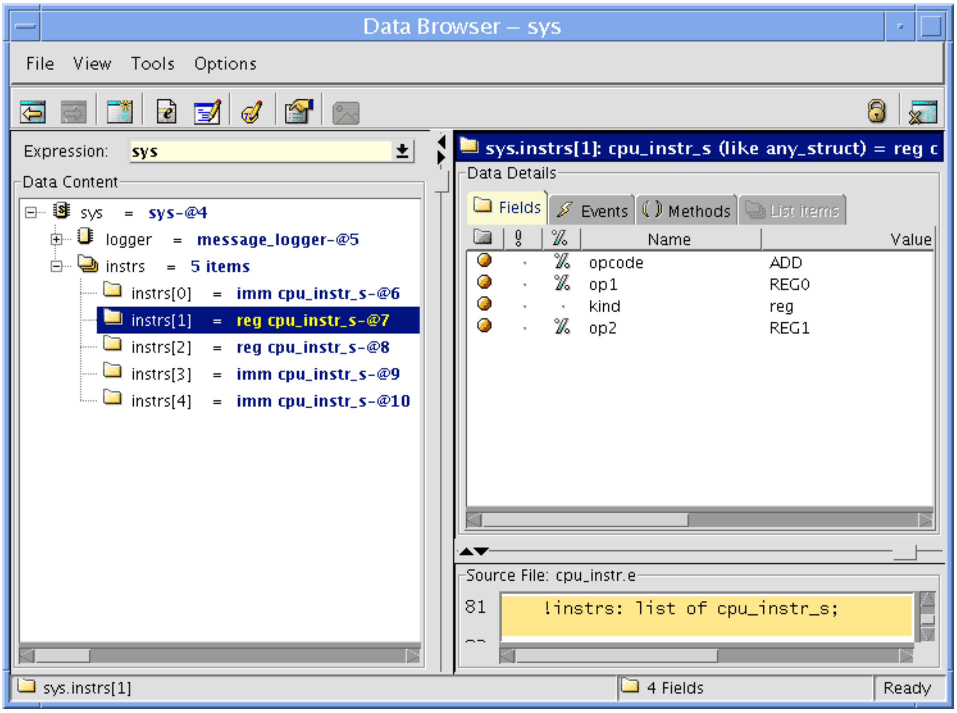


#	Value	opcode	op1
0	imm cpu_instr_s-@6	ADDI	REG0
1	reg cpu_instr_s-@7	ADD	REG0
2	reg cpu_instr_s-@8	ADD	REG0
3	imm cpu_instr_s-@9	ADDI	REG0
4	imm cpu_instr_s-@10	ADDI	REG0

```
81    !instrs: list of cpu_instr_s;
```

5. Double-click the first *reg cpu_instr_s* in the top right pane (list item #1).

   The generated values for the fields of the first *reg* instruction object appear in the right pane.


Data Browser – sys
File View Tools Options
Expression: sys
Data Content
sys = sys-@4
logger = message_logger-@5
instrs = 5 items
instrs[0] = imm cpu_instr_s-@6
instrs[1] = reg cpu_instr_s-@7
instrs[2] = reg cpu_instr_s-@8
instrs[3] = imm cpu_instr_s-@9
instrs[4] = imm cpu_instr_s-@10
sys.instrs[1]: cpu_instr_s (like any_struct) = reg c
Data Details
Fields Events Methods List items
Name Value
opcode ADD
op1 REG0
kind reg
op2 REG1
Source File: cpu_instr.e
81 !instrs: list of cpu_instr_s;
sys.instrs[1]
4 Fields
Ready

Tip If the results you see are significantly different from the results shown here, use the UNIX diff utility to compare your version of the ***e*** files to the files in the *gold* directory.

6. Click each of the other *instrs[n]* lines in the left panel and review their contents in the right panel to confirm that the instructions follow both the general constraints for CPU instructions and the constraints for this particular test.

7. Click **File** ›› **Close** or the Close Window button to close the Data Browser window.

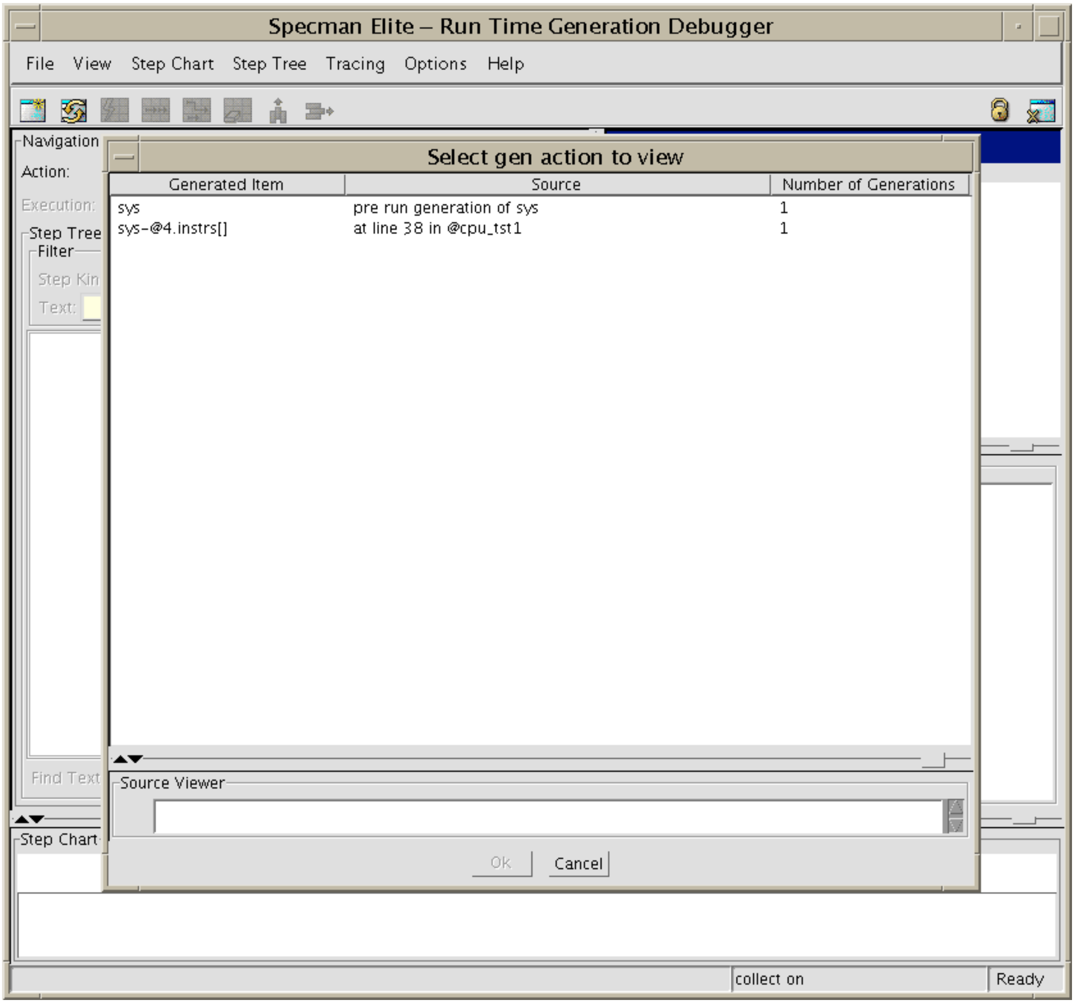
Based on the definition, specification constraints, and test constraints that you have provided, the Specman generator quickly generated the desired instruction stream.

Analyzing Generation

To open the Generation Debugger and view the generation order:

1. Click **Verification** ›› **Generation Debugger** ›› **Show Gen** (in Specview, **Tools** ›› **Generation Debugger** ›› **Show Gen**).

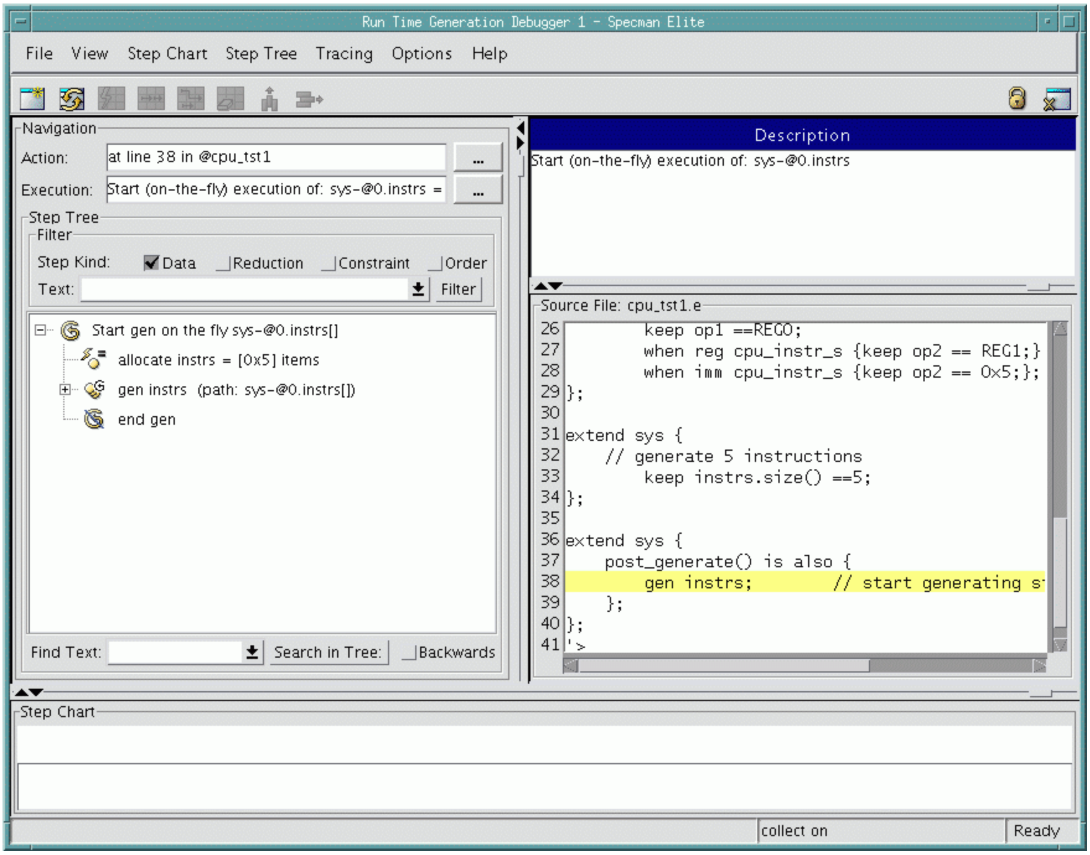
   Two windows open. The window on top shows a list of items that were generated. The window underneath is the Run Time Generation Debugger.


Specman Elite – Run Time Generation Debugger
File View Step Chart Step Tree Tracing Options Help
Navigation
Action:
Execution:
Step Tree
Filter
Step Kin
Text:
Find Text
Step Chart
Select gen action to view
Source Viewer
Ok Cancel
collect on
Ready

Generated Item	Source	Number of Generations
sys	pre run generation of sys	1
sys-@4.instrs[]	at line 38 in @cpu_tst1	1

2. In the top window, click the *sys-@4.instrs[]* line, and then click OK.

   The top window is closed, and the generation Step Tree information appears in the left pane of the Run Time Generation Debugger window.


Run Time Generation Debugger 1 - Specman Elite
File View Step Chart Step Tree Tracing Options Help
Navigation
Step Tree
Filter
Filter
Search in Tree:
Backwards
Description
Step Chart
Ready

Action: at line 38 in @cpu_tst1

Execution: Start (on-the-fly) execution of: sys-@0.instrs =

Step Kind: [✓ Data] Reduction Constraint Order

Text:

```
⊟ Start gen on the fly sys-@0.instrs[]
    allocate instrs = [0x5] items
  ⊞ gen instrs  (path: sys-@0.instrs[])
    end gen
```

Find Text:

Start (on-the-fly) execution of: sys-@0.instrs

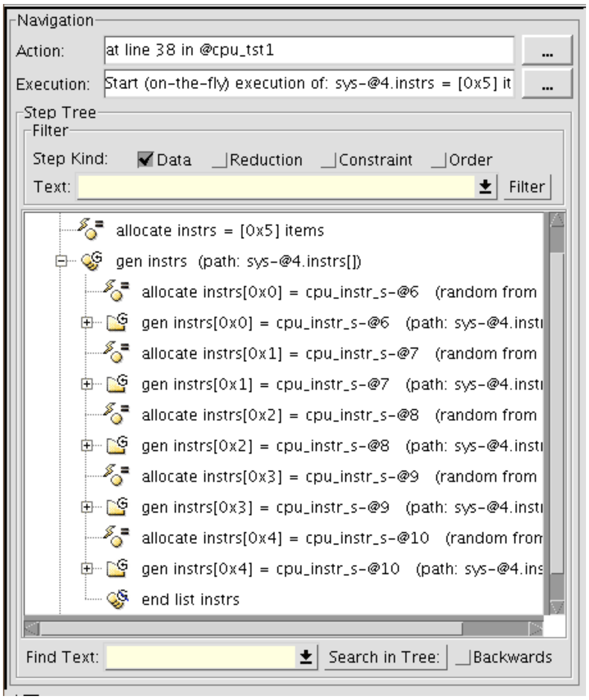
Source File: cpu_tst1.e

```
26          keep op1 ==REG0;
27          when reg cpu_instr_s {keep op2 == REG1;}
28          when imm cpu_instr_s {keep op2 == 0x5;};
29  };
30
31  extend sys {
32      // generate 5 instructions
33          keep instrs.size() ==5;
34  };
35
36  extend sys {
37      post_generate() is also {
38          gen instrs;         // start generating s
39      };
40  };
41  '>
```

collect on

3. In the Step Tree pane, click the + sign on the *gen instrs* line to expand the item.

   The list of *instr* instances appears.



Action: at line 38 in @cpu_tst1

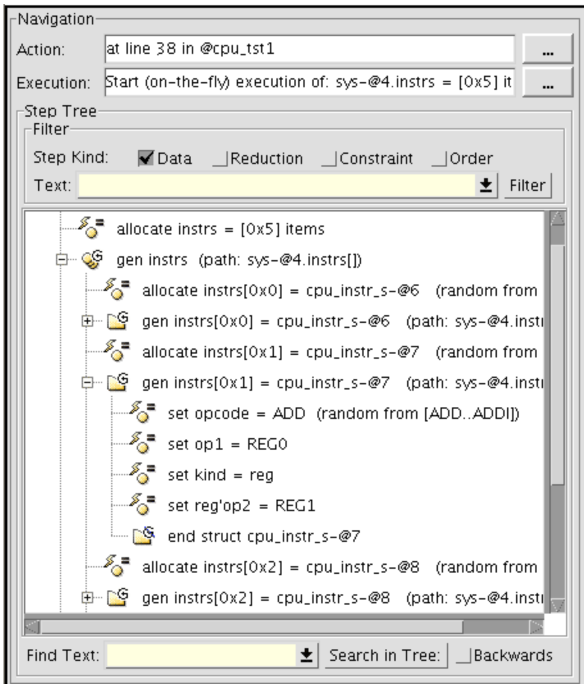
Execution: Start (on-the-fly) execution of: sys-@4.instrs = [0x5] it

Step Kind: [✓ Data]

```
allocate instrs = [0x5] items
gen instrs  (path: sys-@4.instrs[])
    allocate instrs[0x0] = cpu_instr_s-@6   (random from
    gen instrs[0x0] = cpu_instr_s-@6   (path: sys-@4.insti
    allocate instrs[0x1] = cpu_instr_s-@7   (random from
    gen instrs[0x1] = cpu_instr_s-@7   (path: sys-@4.insti
    allocate instrs[0x2] = cpu_instr_s-@8   (random from
    gen instrs[0x2] = cpu_instr_s-@8   (path: sys-@4.insti
    allocate instrs[0x3] = cpu_instr_s-@9   (random from
    gen instrs[0x3] = cpu_instr_s-@9   (path: sys-@4.insti
    allocate instrs[0x4] = cpu_instr_s-@10   (random fron
    gen instrs[0x4] = cpu_instr_s-@10   (path: sys-@4.ins
    end list instrs
```

4. Click the + sign on the *gen instrs[0x1]* line to expand the instance.

   The fields of the instance and their generated values appear. The order of the fields shown in the Step Tree pane is the order in which the fields were generated by Specman.



Action: at line 38 in @cpu_tst1

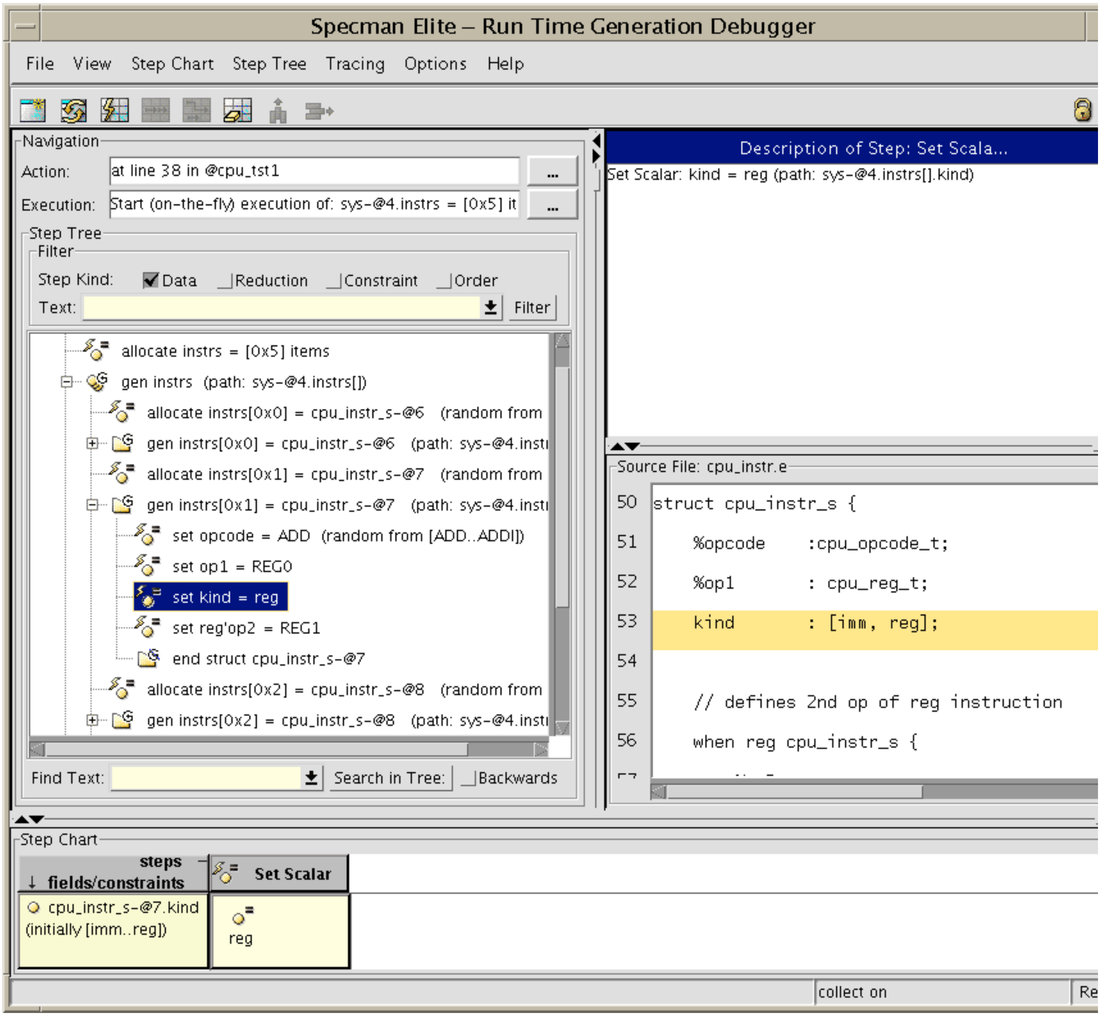
Execution: Start (on-the-fly) execution of: sys-@4.instrs = [0x5] it

Step Kind: [✓ Data]

```
allocate instrs = [0x5] items
gen instrs  (path: sys-@4.instrs[])
  allocate instrs[0x0] = cpu_instr_s-@6   (random from
  gen instrs[0x0] = cpu_instr_s-@6   (path: sys-@4.inst
  allocate instrs[0x1] = cpu_instr_s-@7   (random from
  gen instrs[0x1] = cpu_instr_s-@7   (path: sys-@4.inst
    set opcode = ADD  (random from [ADD..ADDI])
    set op1 = REG0
    set kind = reg
    set reg'op2 = REG1
    end struct cpu_instr_s-@7
  allocate instrs[0x2] = cpu_instr_s-@8   (random from
  gen instrs[0x2] = cpu_instr_s-@8   (path: sys-@4.inst
```

5. Double-click the *set kind = reg* line.

   A sequence of generation steps appears in the Step Chart in the bottom pane. This shows the step in the generation process at which Specman chose a value for the *kind* field. Viewed from left to right, the boxes in the Step Chart represent steps that Specman took to generate the values for fields in the instruction instance.


Specman Elite – Run Time Generation Debugger
File View Step Chart Step Tree Tracing Options Help
Navigation
Action:
Execution:
Step Tree
Filter
Step Kind: Data Reduction Constraint Order
Text: Filter
Find Text: Search in Tree: Backwards
Description of Step: Set Scala...
Source File: cpu_instr.e
Step Chart
steps
fields/constraints
Set Scalar
collect on
Re

Action: at line 38 in @cpu_tst1

Execution: Start (on-the-fly) execution of: sys-@4.instrs = [0x5] it

Step Kind: [✓ Data] Reduction Constraint Order

```
allocate instrs = [0x5] items
gen instrs  (path: sys-@4.instrs[])
   allocate instrs[0x0] = cpu_instr_s-@6   (random from
   gen instrs[0x0] = cpu_instr_s-@6   (path: sys-@4.instı
   allocate instrs[0x1] = cpu_instr_s-@7   (random from
   gen instrs[0x1] = cpu_instr_s-@7   (path: sys-@4.instı
      set opcode = ADD  (random from [ADD..ADDI])
      set op1 = REG0
      set kind = reg
      set reg'op2 = REG1
      end struct cpu_instr_s-@7
   allocate instrs[0x2] = cpu_instr_s-@8   (random from
   gen instrs[0x2] = cpu_instr_s-@8   (path: sys-@4.instı
```

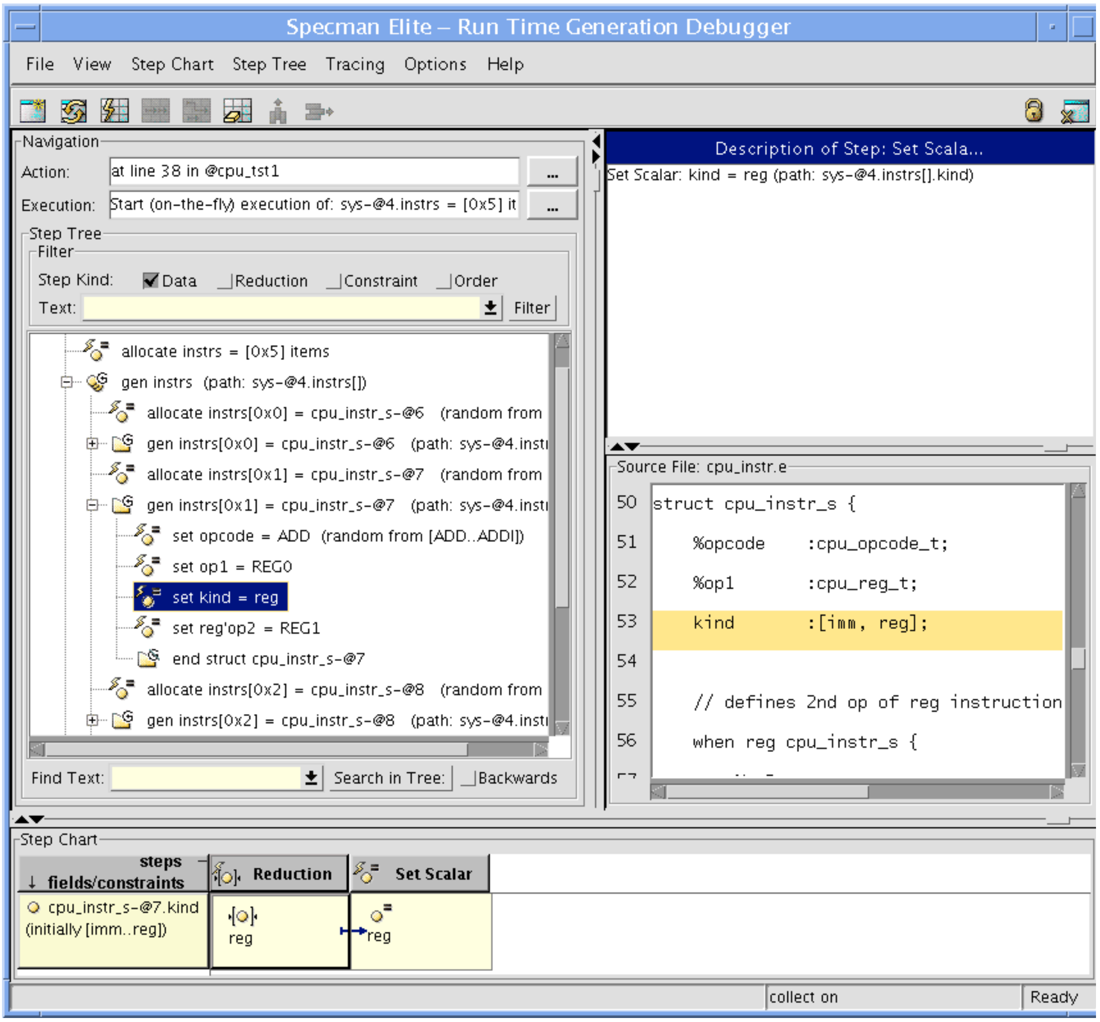
Set Scalar: kind = reg (path: sys-@4.instrs[].kind)

```
struct cpu_instr_s {
    %opcode    :cpu_opcode_t;
    %op1       : cpu_reg_t;
    kind       : [imm, reg];

    // defines 2nd op of reg instruction
    when reg cpu_instr_s {
```

steps / fields/constraints	Set Scalar
cpu_instr_s-@7.kind (initially [imm..reg])	reg

6. Double-click in the *reg* box under the Set Scalar step, to investigate why Specman chose *reg* as the value for the *kind* field.

   A Reduction step box appears, which indicates that constraints on the *kind* field resulted in its possible values being reduced so that it can only be *reg*.


Specman Elite – Run Time Generation Debugger
File View Step Chart Step Tree Tracing Options Help
Navigation
Step Tree
Filter
Filter
Find Text:
Search in Tree:
Backwards
Step Chart
Ready

Action: at line 38 in @cpu_tst1

Execution: Start (on-the-fly) execution of: sys-@4.instrs = [0x5] it

Step Kind: [✓ Data] Reduction Constraint Order

```
allocate instrs = [0x5] items
gen instrs (path: sys-@4.instrs[])
  allocate instrs[0x0] = cpu_instr_s-@6 (random from
  gen instrs[0x0] = cpu_instr_s-@6 (path: sys-@4.instr
  allocate instrs[0x1] = cpu_instr_s-@7 (random from
  gen instrs[0x1] = cpu_instr_s-@7 (path: sys-@4.instr
    set opcode = ADD (random from [ADD..ADDI])
    set op1 = REG0
    set kind = reg
    set reg'op2 = REG1
    end struct cpu_instr_s-@7
  allocate instrs[0x2] = cpu_instr_s-@8 (random from
  gen instrs[0x2] = cpu_instr_s-@8 (path: sys-@4.instr
```

Description of Step: Set Scala...

Set Scalar: kind = reg (path: sys-@4.instrs[].kind)

Source File: cpu_instr.e

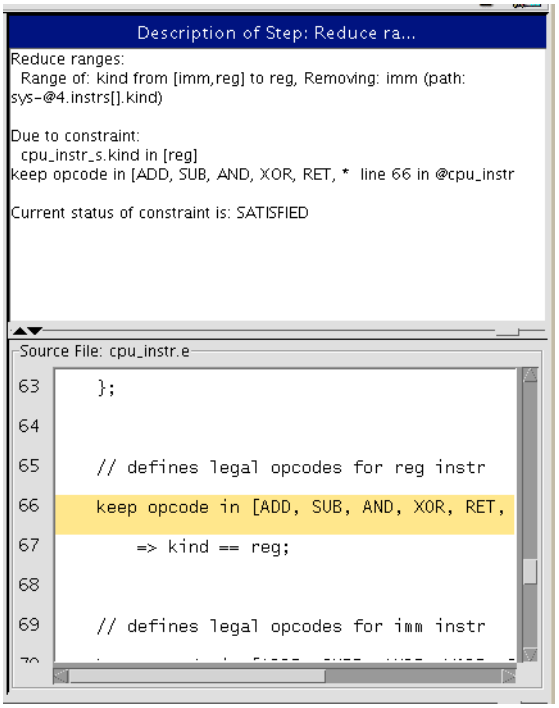
```
50  struct cpu_instr_s {
51      %opcode     :cpu_opcode_t;
52      %op1        :cpu_reg_t;
53      kind        :[imm, reg];
54
55      // defines 2nd op of reg instruction
56      when reg cpu_instr_s {
```

steps / fields/constraints	Reduction	Set Scalar
cpu_instr_s-@7.kind (initially [imm..reg])	reg	reg

collect on

7. Click the *Reduction* title at the top of the Reduction step box in the Step Chart.

   The constraint that caused the reduction is highlighted in the source code in the Source File pane in the middle right. We see that *kind* is constrained to be *reg* when *opcode* is ADD, which is the case for this particular *instr* instance (that is, the *instrs[0x1]* instance).



```
Reduce ranges:
  Range of: kind from [imm,reg] to reg, Removing: imm (path:
sys-@4.instrs[].kind)

Due to constraint:
  cpu_instr_s.kind in [reg]
keep opcode in [ADD, SUB, AND, XOR, RET, *  line 66 in @cpu_instr

Current status of constraint is: SATISFIED
```

Source File: cpu_instr.e

```
    };

    // defines legal opcodes for reg instr
    keep opcode in [ADD, SUB, AND, XOR, RET,
        => kind == reg;

    // defines legal opcodes for imm instr
```

8. Click the Close Window button to close Generation Debugger.

This procedure has shown you how constraints are applied during generation, and how to use the Generation Debugger analysis tool to easily investigate which constraint caused a particular generation result.

Now you are ready to drive this instruction stream into the DUT and run simulation.

5 Driving and Sampling the DUT

Goals for this Chapter

In this chapter, you will drive the DUT with the instruction stream you generated in the last chapter.

In a typical verification environment, where the DUT is modeled in an HDL, you need to link the Specman system with an HDL simulator before running simulation. To streamline this tutorial, we have modeled the DUT in ***e***.

What You Will Learn

In this chapter, you learn how to describe in ***e*** the protocols used to drive test data into the DUT. Although this tutorial does not use an HDL simulator, the process of driving and sampling a DUT written in HDL is the same as the process for a DUT written in ***e***.

As you work through this chapter, you gain experience with these features of the Specman verification system:

- **Time consuming methods (TCMs)—**You can write procedures in ***e*** that are synchronized to other TCMs or to an HDL clock. You can use these procedures to drive and sample test data.
- **DUT signal access—**You can easily access signals and variables in the DUT, either for driving and sampling test data or for synchronizing TCMs.
- **Simulator interface automation—**You can drive and sample a DUT without having to write PLI (Verilog simulators) or FLI/CLI (VHDL simulators) code. The Specman system automatically creates the necessary PLI/FLI calls for you.

This chapter introduces the ***e*** constructs shown in Table 5-1.

Table 5-1 New Constructs Used in this Chapter

Construct	How the Construct is Used
emit	Triggers a named event from within a TCM.
@	Synchronizes the TCMs with an event.
event	Creates a temporal object, in this case a clock, that is used to synchronize the TCMs.
'*hdl_signal_name*'	Accesses a signal in the DUT.
method **() is**…	Creates a procedure (method) that is a member of a struct and manipulates the fields of that struct. Methods can execute in a single point of time, or they can be time consuming methods (TCMs).
pack ()	Converts data from higher level ***e*** structs and fields into the bit or byte representation expected by the DUT.
wait	Suspends action in a TCM until the expression is true.

The steps for driving and sampling the DUT are:

1. Defining the protocols.
2. Running the simulation.

The following sections describe how to perform these steps.

Defining the Protocols

There are two protocols to define for the CPU:

- **Reset protocol—**drives the *rst* signal into the DUT.
- **Drive instructions protocol—**drives instructions into the DUT according to the correct protocol indicated by the *fetch1* and *fetch2* signals.

The drive instructions protocol has one TCM for *pre-run generation*, where the complete list of instructions is generated and then simulation starts. There is another TCM for *on-the-fly generation*, where signals in the DUT are sampled before the instruction is generated. The test in this chapter uses the simple methodology of pre-run generation, while subsequent tests in this tutorial use the more powerful on-the-fly generation.

The TCMs required to drive the CPU are described briefly in Table 5-2. A complete description of one of the TCMs follows the table. You can also view the *cpu_drive.e* file in the *src* directory, if you want to see the complete description of the other TCMs in ***e***.

Table 5-2 TCMs Required to Drive the CPU

Name	Function
drive_cpu()	Calls *reset_cpu ()*. Then, depending on whether the list of CPU instructions is empty or not, it calls *gen_and_drive_instrs ()* or *drive_pregen_instrs ()*.
reset_cpu()	Drives the *rst* signal in the DUT to low for one cycle, to high for five cycles, and then to low.
gen_and_drive_instrs()	Generates the next instruction, and then calls *drive_one_instr ()*.
drive_pregen_instrs()	Calls *drive_one_instr ()* for each generated instruction.
drive_one_instr()	Sends the instruction to the DUT. If the instruction is an immediate instruction, it also waits for the *fetch2* signal to rise, and then sends the second byte of data. Last, it waits for the *exec* signal to rise.

Figure 5-1 shows the ***e*** code for the *drive_one_instr ()* TCM. The CPU architecture requires that tests drive and sample the DUT on the falling edge of the clock. Therefore, all TCMs are synchronized to *cpuclk*, which is defined as follows:

```
extend sys {
    event cpuclk is (fall('top.clk')@sys.any);
};
```

Figure 5-1 The drive_one_instr () TCM

```
drive_one_instr(instr: cpu_instr_s) @sys.cpuclk is {
        var fill0 : uint(bits : 2) = 0b00;

        wait until rise('top.fetch1');

        emit instr.start_drv_DUT;

        if instr.kind == reg then {
            'top.data' = pack(packing.high, instr);
        } else {
        // immediate instruction
            'top.data' = pack(packing.high, instr.opcode,
                instr.op1, fill0);
            wait until rise('top.fetch2');
            'top.data' = pack(packing.high, instr.imm'op2);
        };

        wait until rise('top.exec');

        // execute instr in refmodel
        // sys.cpu_refmodel.execute(instr, sys.cpu_dut);
};
```

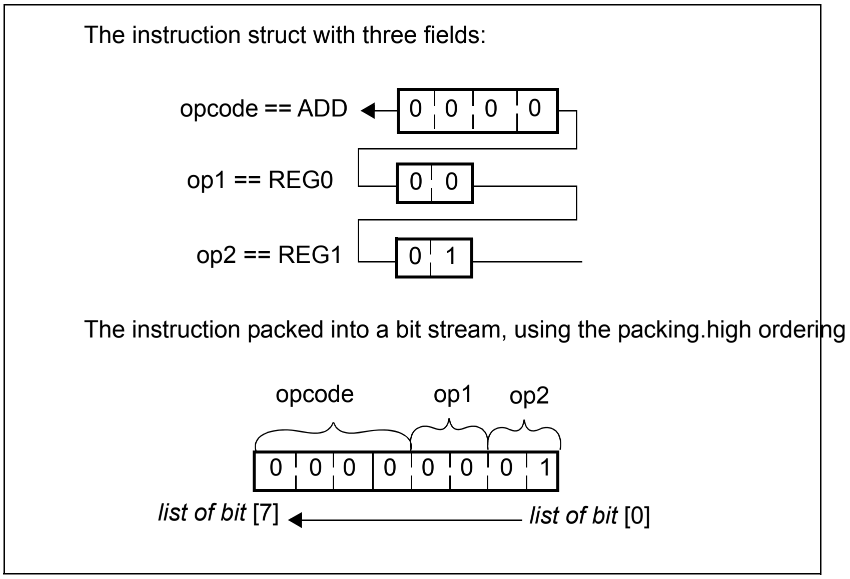
The assignment statements in Figure 5-1 show how to drive and sample signals in an HDL model. Each pair of single quotation marks identifies an object as an HDL signal.

The *start_drv_DUT* event emitted by *drive_one_instr* is not used by any of the TCMs that drive the CPU. You will use it in a later chapter to trigger functional coverage analysis.

The last line shown in Figure 5-1 executes the reference model and is commented out at the moment. You will use it in a later chapter to trigger data checking.

The **pack()** function is a Specman built-in function that facilitates the conversion from higher level data structure to the bit stream required by the DUT. In Chapter 3 “Creating the CPU Instruction Structure”, you used the **%** character to identify the fields that should be driven into the DUT. The **pack()** function intelligently and automatically performs the conversion, as shown in Figure 5-2.

Figure 5-2 A Register Instruction as Received by the DUT


The instruction struct with three fields:
opcode == ADD
0 0 0 0
op1 == REG0
0 0
op2 == REG1
0 1
The instruction packed into a bit stream, using the packing.high ordering
opcode
op1
op2
0 0 0 0 0 0 0 1
list of bit [7]
list of bit [0]

Running the Simulation

This procedure, which involves loading the appropriate files and clicking the Test button, is very similar to the procedure you used in the last chapter to generate the first test.

The difference is that this time you are including the DUT (contained in *cpu_dut.e*) and TCMs that drive it (contained in *cpu_drive.e*).

Procedure

Tip If you have exited SimVision/Specview, you must reinvoke it and load *cpu_tst1.e* again. To do so, enter the **specman -gui** command at the UNIX promt, click **File ›› Load *e* Files**, and select *cpu_tst1.e*. If you are using Specview, you must use the **specview** command at the UNIX prompt.

To run the simulation:

1. Copy the following files to the working directory:

```
src/cpu_dut.e
src/cpu_drive.e
```

2. Open the working directory's copy of the *cpu_top.e* file in an editor.

3. Find the lines in the file that look like this:

```
// Add dut and drive:
//import cpu_dut, cpu_drive;
```

4. Remove the comment characters in front of the *import* line so the lines look like this:

```
// Add dut and drive:
import cpu_dut, cpu_drive;
```

5. Save the *cpu_top.e* file.

6. Since you will not be using the Generation Debugger in the remainder of this tutorial, enter the following command in the Specman> command line in the SimVision/Specview window:

   collect generation off

7. Click **File ›› Reload *e* Files** (in Specview, **File ›› Reload**) to reload the files for test 1.

   Since you activated the "import cpu_dut, cpu_drive" line by removing the comment markers from that line, those two modules will be loaded along with the modules that were loaded in the previous procedure.

Tip If you see a message such as

```
*** Error: No match for 'cpu_dut.e'
```

you need to check whether the working directory contains the following files:

```
cpu_instr.e          cpu_drive.e
cpu_misc.e           cpu_top.e
cpu_dut.e            cpu_tst1.e
```

Add the missing file and then reload the modules.

8. Click **Verification ›› *e* Modules** (in Specview, **Files ›› Modules**) to confirm that six modules are loaded:

```
cpu_instr.e          cpu_drive.e
cpu_misc.e           cpu_top.e
cpu_dut.e            cpu_tst1.e
```

Tip If some of the modules are missing, first check whether you are loading the *cpu_top.e* file that you just modified. The modified *cpu_top.e* file must be in the working directory. Once the modified *cpu_top.e* file is in the working directory, click the Restore button. This action should remove all the currently loaded modules from the session. Then click Load and double-click *cpu_tst1.e* in the Select A File dialog box.

9. Click Close Window in the Modules window to close the window.

10. Click **Verification ›› Test** (in Specview, **Test ›› Test)** to run the simulation.

    You should see the following messages (or something similar) in the Specman console.

```
Doing setup…
Generating the test using seed 1…

Starting the test…
Running the test…
DUT executing instr 0 :    ADDI   REG0x0, @0x05
DUT executing instr 1 :    ADD    REG0x0, REG0x1
DUT executing instr 2 :    ADD    REG0x0, REG0x1
DUT executing instr 3 :    ADDI   REG0x0, @0x05
DUT executing instr 4 :    ADDI   REG0x0, @0x05
Last specman tick - stop_run() was called
Normal stop - stop_run() is completed
Checking the test…
Checking is complete - 0 DUT errors, 0 DUT warnings.
```

You can see from the output that five instructions were executed and no errors were found. It looks like the verification environment is working properly, so you are ready to generate a larger number of tests.

6 Generating Constraint-Driven Tests

Goals for this Chapter

In this chapter, you will run the second test described in "The Functional Test Plan" on page 2-3. To meet the objective of the second test, you must run the same test multiple times using constraints to direct random testing towards the more common operations of the CPU. Through this automatic test generation, we hope to gain high test coverage for the CPU instruction inputs.

What You Will Learn

In this chapter, you learn how to quickly generate different sets of tests by simply changing the seed used for constraint-driven test generation. You also learn how to use weights to control the distribution of the generated values to focus the testing on the common CPU instructions.

As you work through this chapter, you gain experience with two of the Specman verification system's enabling features:

- **Directed-random test generation—**This feature lets you apply constraints to focus random test generation on areas of the design that need to be exercised the most.
- **Random seed generation—**Changing the seed used for random generation enables the Specman system to quickly generate a whole new set of tests.

This chapter introduces the ***e*** constructs and SimVision/Specview menu commands shown in Table 6-1.

Table 6-1 New Constructs and SimVision/Specview Menu Commands Used in this Chapter

Construct	How the Construct is Used
keep soft	Specifies a soft constraint that is kept only if it does not conflict with other hard **keep** constraints.
select	Used with **keep soft** to control the distribution of the generated values.
SimVision & (Specview) Command	**How the Command is Used**
Verification ›› Specman Configuration (Tools ›› Config)	Used to access the Generation tab of the Specman Configuration Options window for creating a user-defined seed for random test generation.
Verification ›› Save Specman State (File ›› Save)	Saves the current test environment, including the random seed, to a .esv file. You can load this file with the **Verification ›› Restore Specman State** (in Specview, **File ›› Restore)** command.
Verification ›› Test with Random Seed (Test ›› Test with Random Seed)	Generates a set of tests with a new random seed.

The steps for generating random tests are:

1. Defining weights for random tests.
2. Generating and running tests with a user-specified seed.
3. Generating and running tests with a random seed.

The following sections describe these tasks in detail.

Defining Weights for Random Tests

Because of the way that CPUs are typically used, arithmetic and logical operations comprise a high percentage of the CPU instructions. You can use the **select** construct with **keep soft** to require the Specman system to generate a higher percentage of instructions for arithmetic and logical operations than for control flow.

Procedure

To see how weighted constraints are created in ***e***:

1. Copy the *src/cpu_tst2.e* file to the working directory.
2. Open the *cpu_tst2.e* file in an editor.
3. Find the portion of the file that looks as follows and review the **keep soft** constraint.

puts equal weight on arithmetic and logical operations and less weight on control flow operations

```
<'
import cpu_top;

extend cpu_instr_s {
    keep soft opcode == select {
        10 : [JMP, JMPC, CALL, RET, NOP];
        30 : [ADD, ADDI, SUB, SUBI];
        30 : [AND, ANDI, XOR, XORI];
    };
};

'>
```

Generating Tests With a User-Specified Seed

You can specify the random seed that the Specman system uses to generate tests.

Procedure

This procedure shows how to create a random seed:

1. In SimVision, click **Verification** ›› **Restore** Specman **State** ›› **Restore to Last State** (in Specview, **File** ›› **Restore** ›› **Restore to Last State)** to remove all of the ***e*** modules from the current session.
2. Click **File** ›› **Load *e* File** (in Specview, **File** ›› **Load)**. Then double-click the *cpu_tst2.e* file.

   The Specman system loads the *cpu_tst2.e* file along with its imported modules.
3. Click **Verification** ›› **Specman Configuration** (in Specview, **Tools** ›› **Config)**.

   The Specman Configuration Options window opens.
4. Click the Generation tab and then enter a number of your choice in the Seed text box.

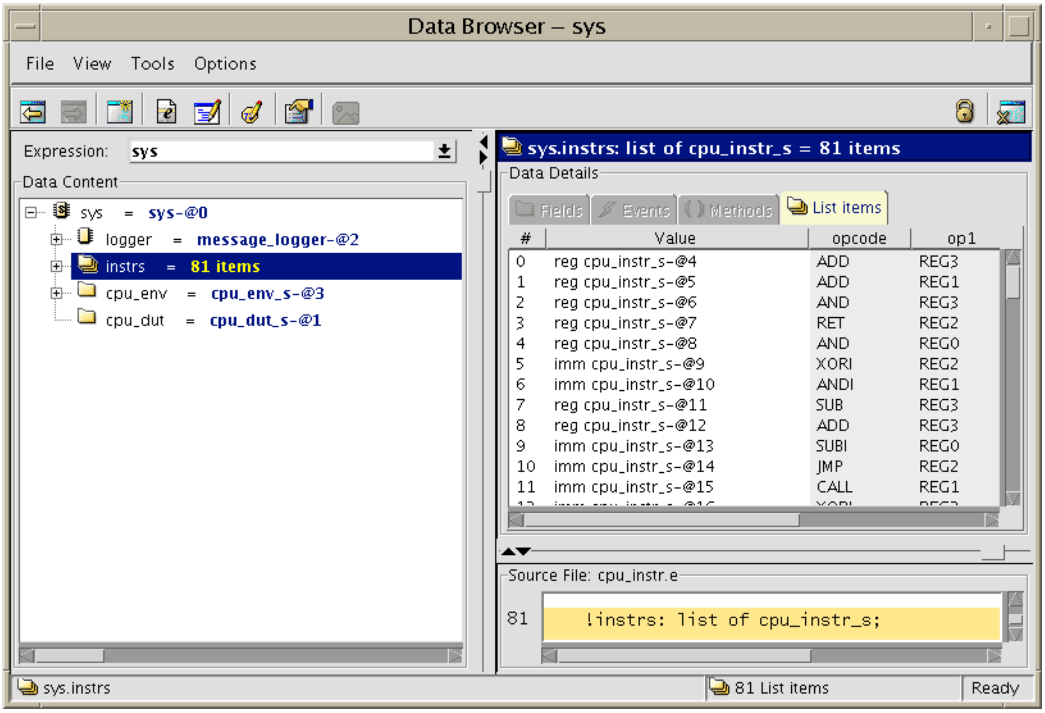
5. Click OK to save the settings and close the window.
6. In SimVision, click **Verification** ›› **Test** (in Specview, **Test** ›› **Test)**.

   The Specman system runs the test with the seed value you entered above, and reports the results.
7. Click **Verification** ›› **Data Browser** ›› **Show Data Sys** (in Specview, **Tools** ›› **Data Browser** ›› **Show Data Sys**).

   The Data Browser window appears.
8. Click the blue *x items* link following "instrs =" in the left pane (where *x* is the number of instruction instances that were generated).

   Instructions are listed in the top right pane.

   You should see an approximately equal distribution of arithmetic and logical operations, and about one-third as many control flow operations as there are either arithmetic or logical operations. That is, control flow, arithmetic, and logic operations are generated in a ratio of about 10:30:30. The more instruction instances are generated, the closer the distribution will be to the ratio specified in the **keep soft** opcode **== select** constraint.



#	Value	opcode	op1
0	reg cpu_instr_s-@4	ADD	REG3
1	reg cpu_instr_s-@5	ADD	REG1
2	reg cpu_instr_s-@6	AND	REG3
3	reg cpu_instr_s-@7	RET	REG2
4	reg cpu_instr_s-@8	AND	REG0
5	imm cpu_instr_s-@9	XORI	REG2
6	imm cpu_instr_s-@10	ANDI	REG1
7	reg cpu_instr_s-@11	SUB	REG3
8	reg cpu_instr_s-@12	ADD	REG3
9	imm cpu_instr_s-@13	SUBI	REG0
10	imm cpu_instr_s-@14	JMP	REG2
11	imm cpu_instr_s-@15	CALL	REG1

```
81    !instrs: list of cpu_instr_s;
```

Generating Tests With a Random Seed

You can require the Specman system to generate a random seed.

Procedure

To run a test using a Specman-generated random seed:

1. In SimVision, click **File ›› Reload *e* Files** (in Specview, **File ›› Reload)**.
2. Click Verification ›› **Test with Random Seed** (in Specview, **Test ›› Test with Random Seed)**.

   The Specman system runs the test with the random seed shown in the Specman console window and reports the results.
3. Review the results in the Data Browser, as in the previous procedure.

   You should again see an approximately equal distribution of arithmetic and logical operations, and about one-third as many control flow operations as there are either arithmetic or logical operations. The results should be different from the previous run.
4. Optionally you can repeat steps 1-3 several times to confirm that you see different results each time.

Tip If you see similar results in subsequent runs, it is likely that you forgot to reload the design before running the test. If you do not reload the design, the test is run with the current seed.

You can see that using different random seeds lets you easily generate many tests. Quickly analyzing the results of all these tests would be difficult without Specman's coverage analysis technology. The next two chapters show how to use coverage analysis to accurately measure the progress of your verification effort.

7 Defining Coverage

Goals for this Chapter

You can avoid redundant testing by measuring the progress of the verification effort with coverage statistics for your tests. This chapter explains how to define the test coverage statistics you want to collect.

What You Will Learn

In this chapter, you learn how to define which coverage information you want to collect for the DUT internal states, for the instruction stream, and for an intersection of DUT states and the instruction stream.

As you work through this chapter, you gain experience with another one of the Specman verification system's enabling features—the **Functional Coverage Analyzer**. The Specman coverage analysis feature lets you define exactly what functionality of the device you want to monitor and report. With coverage analysis, you can see whether generated tests meet the goals set in the functional test plan and whether these tests continue to be sufficient as the design develops, the design specifications change, and bug fixes are implemented.

This chapter introduces the ***e*** constructs shown in Table 7-1.

Table 7-1 New Constructs Used in this Chapter

Construct	How the Construct is Used
event	Defines a condition that triggers sampling of coverage data.
cover	Defines a group of data collection items.
item	Identifies an object to be sampled.

Table 7-1 New Constructs Used in this Chapter

Construct	How the Construct is Used
transition	Identifies an object whose current and previous values are to be collected when the sampling event occurs.

The three types of coverage data that you might want to collect are:

- Coverage data for the finite state machine (FSM).
- Coverage data for the generated instructions.
- Coverage data for the corner case.

The following sections describe how to define coverage for these three types of data.

Defining Coverage for the FSM

You can use the constructs shown in Table 7-1 to define coverage for the FSM:

- State machine register
- State machine transition

Procedure

To define coverage for the FSM:

1. Copy the *src/cpu_cover.e* file to the working directory and open *cpu_cover.e* in an editor.
2. Find the portion of the file that looks like the excerpt below and review the declaration that defines the sampling event for the FSM:

defines FSM sampling event

```
extend cpu_env_s {

    event cpu_fsm is @sys.cpuclk;

    // DUT Coverage: State Machine and State
    // Machine transition coverage
```

3. Add the coverage group and coverage items for state machine coverage.

The coverage group name (*cpu_fsm*) must be the same as the event name defined in Step 2 above. The **item** statement declares the name of the coverage item (*fsm*), its data type (*FSM_type_t*), and the object in the DUT to be sampled. The **transition** statement says that the current and previous values of *fsm* must be collected. This means that whenever the *sys.cpuclk* signal changes, the Specman system collects the current and previous values of *top.cpu.curr_FSM*.

defines the coverage group cpu_fsm

```
extend cpu_env_s {
    event cpu_fsm is @sys.cpuclk;

    // DUT Coverage: State Machine and State
    // Machine transition coverage
    cover cpu_fsm is {
      item fsm: cpu_FSM_type_t = 'top.cpu.curr_FSM';
      transition fsm;
    };
};
```

4. Save the *cpu_cover.e* file.

Defining Coverage for the Generated Instructions

You can use the constructs shown in Table 7-1 on page 7-1 to define coverage collection for the CPU instruction stream:

- opcode
- op1

This coverage group uses a sampling event that is declared and triggered in the *cpu_drive.e* file.

```
drive_one_instr(instr: cpu_instr_s) @sys.cpuclk is {
.
.
.
    emit instr.start_drv_DUT;
.
.
.
```

Thus data collection for the instruction stream occurs each time an instruction is driven into the DUT.

Procedure

To extend the *cpu_instr_s* struct to define coverage for the generated instructions:

1. Find the portion of the *cpu_cover.e* file that looks like the excerpt below and review the coverage group declaration.

defines coverage group

```
extend cpu_instr_s {

    cover start_drv_DUT is {

    };

};
```

2. Add *opcode* and *op1* items to the *start_drv_DUT* coverage group.

```
extend cpu_instr_s {

    cover start_drv_DUT is {
        item opcode;
        item op1;
    };
};
```

3. Save the *cpu_cover.e* file.

Defining Coverage for the Corner Case

Test 3 of the functional test plan (see “Test 3” on page 2-4) specifies the corner case that you want to cover. To test the behavior of the DUT when the JMPC (jump on carry) instruction opcode is issued, you need to be sure that the JMPC opcode is issued only when the carry signal is high. Here, you define a coverage group so you can determine how often that combination of conditions occurs.

Procedure

To define coverage data for the designated corner case:

1. Add a *carry* item to the *start_drv_DUT* coverage group.

```
extend cpu_instr_s {

    cover start_drv_DUT is {
        item opcode;
        item op1;
        item carry: bit = 'top.carry';
    };
};
```

2. Define a cross item between opcode and carry.

   Cross coverage lets you define the intersections of two or more coverage items, generating a more informative report. The cross coverage item defined here shows every combination of *carry* value and *opcode* that is generated in the test.

```
extend cpu_instr_s {

    cover start_drv_DUT is {
        item opcode;
        item op1;
        item carry: bit = 'top.carry';
        cross opcode, carry;
    };
};
```

3. Save the *cpu_cover.e* file.

Now that you have defined the coverage groups, you are ready to simulate and view the coverage reports.

8 Analyzing Coverage

Goals for this Chapter

The goals for this chapter are to

- Determine whether the tests you have generated meet the specifications in the functional test plan.
- Based on that information, decide whether additional tests must be created to complete design verification.

What You Will Learn

In this chapter, you learn how to display coverage reports for individual coverage items, exactly as you have defined them, and to merge reports for individual items so that you can easily analyze the progress of your design verification.

You will examine coverage grades for different types of coverage items. A coverage grade indicates how thoroughly the item was covered during a test or set of tests. The maximum grade is 1.00, which means that every possible value for that item occurred, or was "hit", during the tests. Incomplete coverage, or a "hole", is represented by a decimal fraction: A grade of 0.75, for example, means that three out of every four possible values were hit.

As you work through this chapter, you gain experience with these Specman features:

- **Help—**This helps you find the information you need in the Specman Online Documentation.
- **Coverage Extensibility—**This allows you to change coverage group and coverage item definitions.

This chapter introduces the SimVision/Specview menu commands shown in Table 8-1.

Table 8-1 New SimVision/Specview Menu Commands Used in this Chapter

SimVision & (Specview) Command	How the Command is Used
Verification ›› Coverage (Tools ›› Coverage)	Displays coverage reports and creates cross-coverage reports.
Help ›› Specman Component of IES (Help ›› Specman Help)	Invokes the Specman Online Documentation browser (Cadence Help).

The steps required to analyze test coverage for the CPU design are:

1. Running tests with coverage groups defined.
2. Viewing state machine coverage.
3. Viewing instruction stream coverage.
4. Viewing corner case coverage.

The following sections describe these tasks in detail.

Running Tests with Coverage Groups Defined

This procedure is similar to the procedure you have already used to run tests without coverage.

Procedure

To run tests with coverage groups defined:

1. Open the working directory's copy of the *cpu_top.e* file in an editor.
2. Find the lines in the file that look like this:

```
// Add Coverage:
//import cpu_cover;
```

3. Remove the comment characters in front of the **import** line so the lines look like this:

```
// Add Coverage:
import cpu_cover;
```

4. Save the *cpu_top.e* file.
5. In SimVision, click **File ›› Reload *e* Files** (in Specview, **File ›› Reload)** to reload the files for test 2.

Tip If you have exited SimVision/Specview, you must reinvoke it and load *cpu_tst2.e* again. To do so, execute the **specman -gui** command at the UNIX prompt, click **File ›› Load *e* File**, select the *cpu_tst2.e* file, and click OK. (If you are using Specview, execute the **specview** command at the UNIX prompt, click **File ›› Load**, select the *cpu_tst2.e* file, and click OK.)

6. Click **Verification ›› *e* Modules** (in Specview, **File ›› Modules**) to confirm that these seven modules are loaded:

```
cpu_instr
cpu_misc
cpu_dut
cpu_drive
cpu_cover
cpu_top
cpu_tst2
```

7. Click **Verification ›› Test** (in Specview, **Test ›› Test)**.

You should see something similar to the following in the Specman console window. The last line indicates that coverage data was written to an .ecov file (a coverage data file).

```
test
Doing setup…
Generating the test using seed 1
Starting the test…
Running the test…
DUT executing instr   0 :       ANDI   REG0x3, @0x8d
DUT executing instr   1 :       XOR    REG0x3, REG0x0
DUT executing instr   2 :       ADD    REG0x3, REG0x2
DUT executing instr   3 :       ANDI   REG0x3, @0x92
DUT executing instr   4 :       RET    REG0x3, REG0x1
.
.
.
Last specman tick - stop_run() was called
Normal stop - stop_run() is completed
Checking the test ...
Checking is complete - 0 DUT errors, 0 DUT warnings.
```

```
    Wrote 1 cover_struct to cpu_tst2_1.ecov
```

Viewing State Machine Coverage

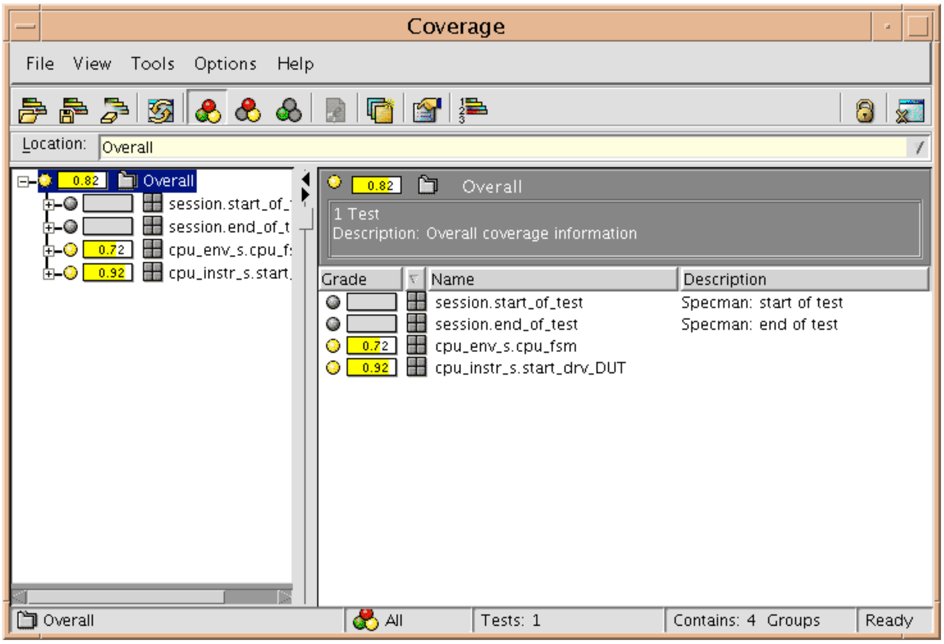
You have two reports to look at, the state machine register report and the state machine transition report.

If you are using a different seed, or a version of the Specman verification system other than the version for this tutorial, you might see different results in your coverage reports.

Procedure

1. In SimVision, click **Verification ›› Coverage ›› Show Cover** (in Specview, **Tools ›› Coverage ›› Show Cover)**.

   The Coverage window appears.



Grade	Name	Description
	session.start_of_test	Specman: start of test
	session.end_of_test	Specman: end of test
0.72	cpu_env_s.cpu_fsm	
0.92	cpu_instr_s.start_drv_DUT	

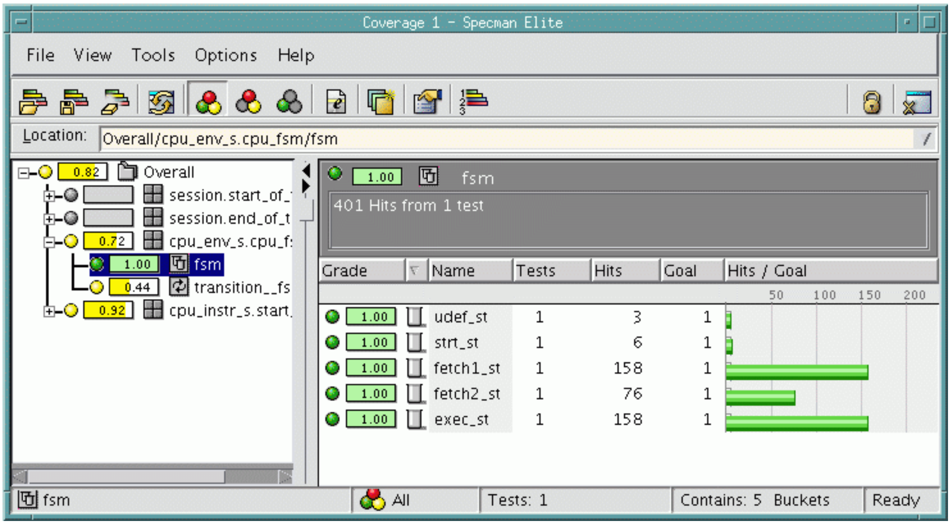
0.82 Overall
1 Test
Description: Overall coverage information

Overall | All | Tests: 1 | Contains: 4 Groups | Ready

2. In the left pane, click the + to the left of *cpu_env_s.cpu_fsm* and then click on *fsm*.

   The state machine register report appears in the right-hand pane.

   From the report it is easy to see that, for example, the *fetch1_st* state was entered 158 times in the 401 times sampled.



1.00 fsm

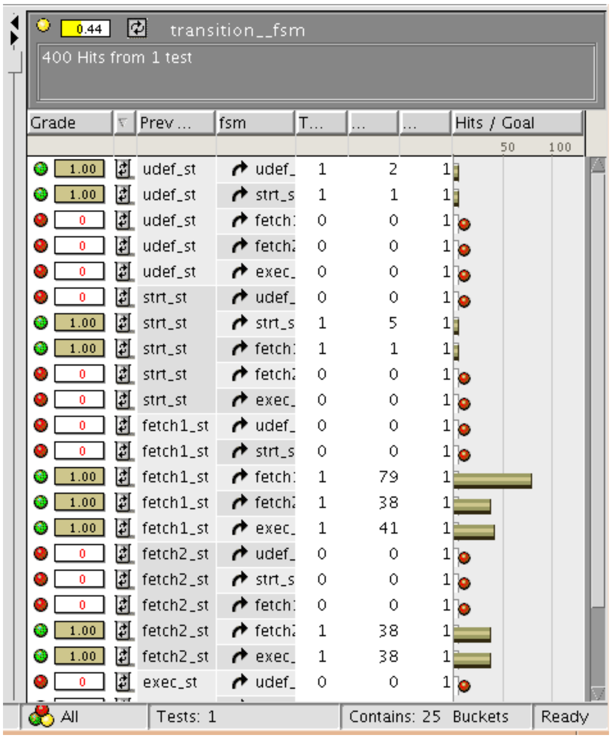
401 Hits from 1 test

Grade	Name	Tests	Hits	Goal	Hits / Goal
1.00	udef_st	1	3	1	
1.00	strt_st	1	6	1	
1.00	fetch1_st	1	158	1	
1.00	fetch2_st	1	76	1	
1.00	exec_st	1	158	1	

3. In the left pane, click on *transition_fsm.*

   The state machine transition report appears in the right-hand frames.

   As you scroll down the display, perhaps the first thing you notice about the state machine transition report is that there are a number of transitions that never occurred. This is because these transitions are illegal.



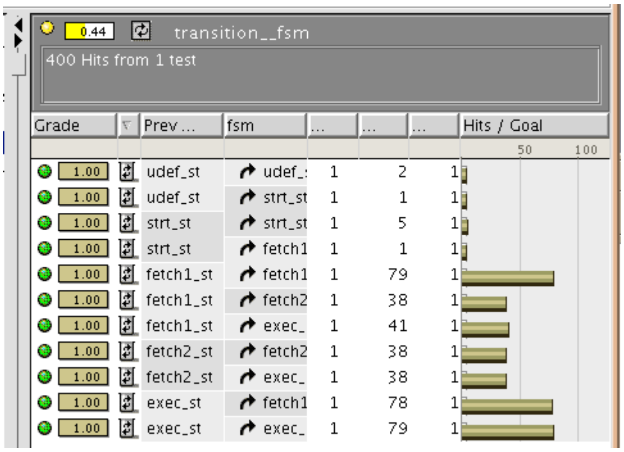
0.44 transition__fsm

400 Hits from 1 test

Grade	Prev ...	fsm	T...	...	...	Hits / Goal
1.00	udef_st	udef_	1	2	1	
1.00	udef_st	strt_s	1	1	1	
0	udef_st	fetch1	0	0	1	
0	udef_st	fetch2	0	0	1	
0	udef_st	exec_	0	0	1	
0	strt_st	udef_	0	0	1	
1.00	strt_st	strt_s	1	5	1	
1.00	strt_st	fetch1	1	1	1	
0	strt_st	fetch2	0	0	1	
0	strt_st	exec_	0	0	1	
0	fetch1_st	udef_	0	0	1	
0	fetch1_st	strt_s	0	0	1	
1.00	fetch1_st	fetch1	1	79	1	
1.00	fetch1_st	fetch2	1	38	1	
1.00	fetch1_st	exec_	1	41	1	
0	fetch2_st	udef_	0	0	1	
0	fetch2_st	strt_s	0	0	1	
0	fetch2_st	fetch1	0	0	1	
1.00	fetch2_st	fetch2	1	38	1	
1.00	fetch2_st	exec_	1	38	1	
0	exec_st	udef_	0	0	1	

All | Tests: 1 | Contains: 25 Buckets | Ready

You can change the display to show only data for transitions that have occurred by clicking **View** ›› **Full** in the Coverage GUI toolbar. (The **View** ›› **Full** option shows only data for transitions that have occurred; the **View** ›› **Holes Only** option shows only data for transitions that have not occurred.)



0.44 transition__fsm

400 Hits from 1 test

Grade	Prev ...	fsm	...	...	...	Hits / Goal
1.00	udef_st	udef_:	1	2	1	
1.00	udef_st	strt_st	1	1	1	
1.00	strt_st	strt_st	1	5	1	
1.00	strt_st	fetch1	1	1	1	
1.00	fetch1_st	fetch1	1	79	1	
1.00	fetch1_st	fetch2	1	38	1	
1.00	fetch1_st	exec_	1	41	1	
1.00	fetch2_st	fetch2	1	38	1	
1.00	fetch2_st	exec_	1	38	1	
1.00	exec_st	fetch1	1	78	1	
1.00	exec_st	exec_	1	79	1	

You can also define transitions as illegal so that they do not appear in the coverage report. Illegal transition definition is described in the following steps.

4. For an explanation of how to define transitions as illegal so that they do not appear in the coverage report, click **Help ›› Help Browser** in the Coverage GUI.

   The Cadence Help System’s navigation pane opens.

5. We want to search for *transition* only in the Specman manuals:

   a. Enter *transition* in the Search Term field.

   b. Click the down-pointing red arrow to the right of the Search Term field and click Search Selected.

   c. In the list of documentation sets, click Specman.

   d. Click the Search Selected magnifying glass next to the Search Term field.

   The **transition** construct is part of the syntax for coverage items. In this case, the first search hit result points to the **transition** construct syntax documentation.

6. Further down the search list, click on the “Defining Transition Coverage Items” link.

7. When the **transition** construct description appears:

   a. Scroll down the page to the **illegal** coverage item option description.

   b. In the **illegal** option description, click on the “illegal” link to display an example showing the use of the **illegal** option:

```
struct packet {
    packet_len: uint (bits: 12);
    event rcv_clk;
    cover rcv_clk is {
        item len: uint (bits: 12) = packet_len using
            ranges = {
                        range( [16..255], "small");
                        range( [256..3k-1], "medium");
                        range( [3k..4k-1], "big");
                     },
            illegal = (len < 16 or len > 4000);
    };
};
```

Viewing Instruction Stream Coverage

We will now look at the coverage for the CPU instruction stream. To provide more interesting results for examination, we will load the results of a set of regression tests. These regression tests were run with the second test and many different seeds.

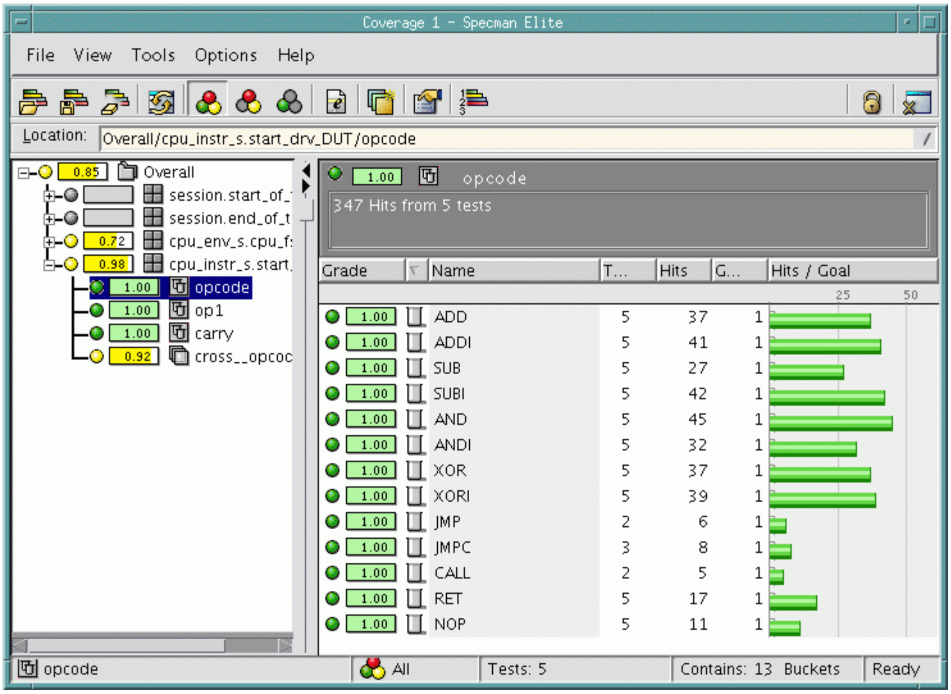
Procedure

To view instruction stream coverage:

1. Copy *src/regression_A.ecov* file to the working directory.
2. If you closed the coverage window after the last procedure, click the Coverage Analysis button in the SimVision/Specview toolbar to open it again.

   The Coverage window appears.
3. In the Coverage window, click **File** ›› **Clear Data** to delete the coverage data from the previous test.
4. Click **File** ›› **Read Data** to open the Read Files dialog box.
5. Select *regression_A.ecov* and click the Read button.
6. In the left pane, click the + by *cpu_instr_s.start_drv_DUT* to expand the item, and then click on *opcode*.

   The opcode coverage report appears in the right-hand panes. These results show that the current set of tests fulfill the requirement in the functional test plan to be weighted more toward arithmetic and logic operations than toward control flow operations.


Coverage 1 - Specman Elite

Location: Overall/cpu_instr_s.start_drv_DUT/opcode

0.85	Overall	
	session.start_of_	
	session.end_of_t	
0.72	cpu_env_s.cpu_f:	
0.98	cpu_instr_s.start	
1.00	opcode	
1.00	op1	
1.00	carry	
0.92	cross__opcoc	

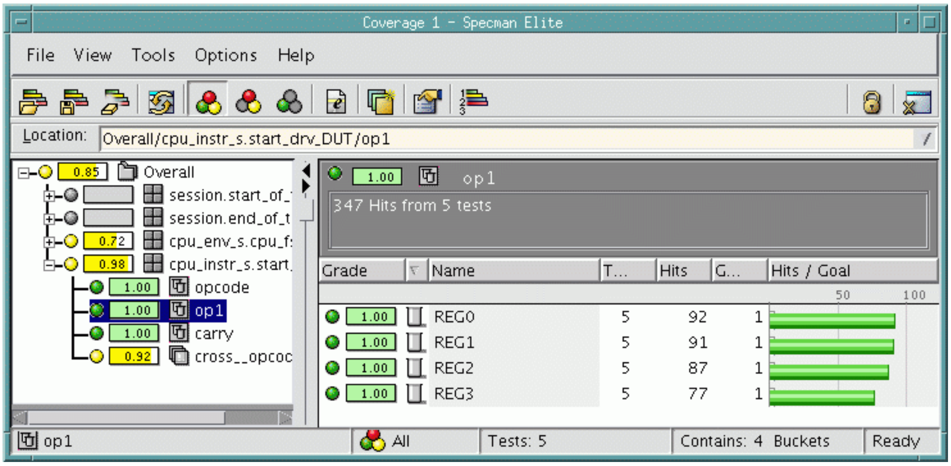
1.00 opcode
347 Hits from 5 tests

Grade	Name	T...	Hits	G...	Hits / Goal
1.00	ADD	5	37	1	
1.00	ADDI	5	41	1	
1.00	SUB	5	27	1	
1.00	SUBI	5	42	1	
1.00	AND	5	45	1	
1.00	ANDI	5	32	1	
1.00	XOR	5	37	1	
1.00	XORI	5	39	1	
1.00	JMP	2	6	1	
1.00	JMPC	3	8	1	
1.00	CALL	2	5	1	
1.00	RET	5	17	1	
1.00	NOP	5	11	1	

opcode | All | Tests: 5 | Contains: 13 Buckets | Ready

7. In the left pane, click on *op1*.

   The *op1* coverage report appears in the right-hand panes. All possible *op1* values are shown to be well covered.


Coverage 1 - Specman Elite

Location: Overall/cpu_instr_s.start_drv_DUT/op1

0.85	Overall	
	session.start_of_	
	session.end_of_t	
0.72	cpu_env_s.cpu_f:	
0.98	cpu_instr_s.start	
1.00	opcode	
1.00	op1	
1.00	carry	
0.92	cross__opcoc	

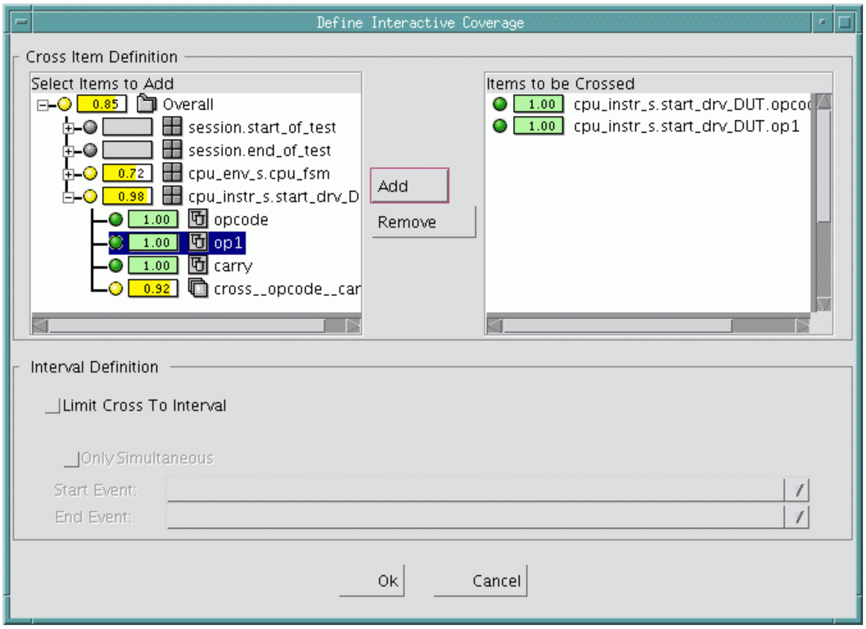
1.00 op1
347 Hits from 5 tests

Grade	Name	T...	Hits	G...	Hits / Goal
1.00	REG0	5	92	1	
1.00	REG1	5	91	1	
1.00	REG2	5	87	1	
1.00	REG3	5	77	1	

op1 | All | Tests: 5 | Contains: 4 Buckets | Ready

8. Click **Tools ›› Interactive Cross**.

9. When the Define Interactive Coverage dialog opens, click on the + next to *cpu_instr_s.start_drv_DUT* to expand it.
10. Click on *opcode*, and then click the Add button.
11. Click on *op1*, and then click the Add button again.



Select Items to Add

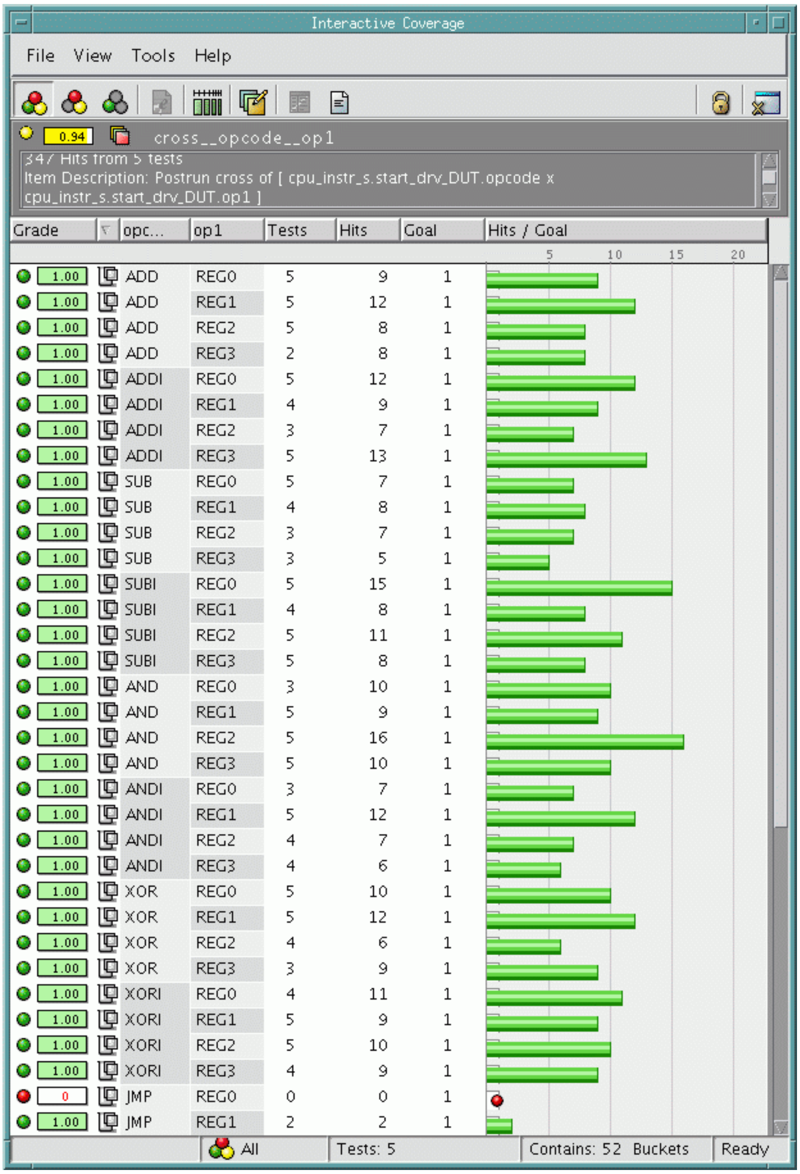
```
0.85  Overall
      session.start_of_test
      session.end_of_test
0.72  cpu_env_s.cpu_fsm
0.98  cpu_instr_s.start_drv_D
  1.00  opcode
  1.00  op1
  1.00  carry
  0.92  cross__opcode__car
```

Items to be Crossed

```
1.00  cpu_instr_s.start_drv_DUT.opcod
1.00  cpu_instr_s.start_drv_DUT.op1
```

Limit Cross To Interval
Only Simultaneous
Start Event:
End Event:

12. Click OK to display a coverage report of the new *cross__opcode__op1* item.



0.94 cross__opcode__op1

347 Hits from 5 tests
Item Description: Postrun cross of [cpu_instr_s.start_drv_DUT.opcode x cpu_instr_s.start_drv_DUT.op1]

Grade	opc...	op1	Tests	Hits	Goal	Hits / Goal
1.00	ADD	REG0	5	9	1	
1.00	ADD	REG1	5	12	1	
1.00	ADD	REG2	5	8	1	
1.00	ADD	REG3	2	8	1	
1.00	ADDI	REG0	5	12	1	
1.00	ADDI	REG1	4	9	1	
1.00	ADDI	REG2	3	7	1	
1.00	ADDI	REG3	5	13	1	
1.00	SUB	REG0	5	7	1	
1.00	SUB	REG1	4	8	1	
1.00	SUB	REG2	3	7	1	
1.00	SUB	REG3	3	5	1	
1.00	SUBI	REG0	5	15	1	
1.00	SUBI	REG1	4	8	1	
1.00	SUBI	REG2	5	11	1	
1.00	SUBI	REG3	5	8	1	
1.00	AND	REG0	3	10	1	
1.00	AND	REG1	5	9	1	
1.00	AND	REG2	5	16	1	
1.00	AND	REG3	5	10	1	
1.00	ANDI	REG0	3	7	1	
1.00	ANDI	REG1	5	12	1	
1.00	ANDI	REG2	4	7	1	
1.00	ANDI	REG3	4	6	1	
1.00	XOR	REG0	5	10	1	
1.00	XOR	REG1	5	12	1	
1.00	XOR	REG2	4	6	1	
1.00	XOR	REG3	3	9	1	
1.00	XORI	REG0	4	11	1	
1.00	XORI	REG1	5	9	1	
1.00	XORI	REG2	5	10	1	
1.00	XORI	REG3	4	9	1	
0	JMP	REG0	0	0	1	
1.00	JMP	REG1	2	2	1	

All | Tests: 5 | Contains: 52 Buckets | Ready

Extending Coverage

In this section the coverage group is extended by the addition of a new item, and by making an existing item a per-instance item, which allows us to see coverage separately for different subtypes.

Procedure

To extend a coverage group:

1. Copy the *src/cpu_cover_extend.e* file to the working directory and open the *cpu_cover_extend.e* file in an editor.
2. Find the lines in the file that look like this:

```
extend cpu_instr_s {

    //Extend the start_drv_DUT cover group with "is also"

        // Add the kind field to the cover group as a new item

        // Extend the op1 item to make it a per_instance item
};
```

3. Add the coverage group extension struct member. Do not forget the closing bracket.

   The syntax for a coverage group extension is the same as for the original coverage group definition, except that it uses **is also** instead of **is**.

```
extend cpu_instr_s {

    // Extend the start_drv_DUT cover group with "is also"
    cover start_drv_DUT is also {

        // Add the kind field to the cover group as a new item

        // Extend the op1 item to make it a per_instance item
    };
};
```

4. Add a new coverage item to cover the kind field of the cpu_instr_s struct.

```
extend cpu_instr_s {

    // Extend the start_drv_DUT cover group with "is also"
    cover start_drv_DUT is also {

        // Add the kind field to the cover group as a new item
        item kind;

        // Extend the op1 item to make it a per_instance item
    };
};
```

5. Extend the op1 item with **using also**, to make it a per_instance item.

   Since the op1 item can have one of the enumerated types REG0, REG1, REG2, or REG3, making this item a per_instance item will provide separate coverage for each of those four subtypes.

```
extend cpu_instr_s {

    // Extend the start_drv_DUT cover group with "is also"
    cover start_drv_DUT is also {

        // Add the kind field to the cover group as a new item
        item kind;

        // Extend the op1 item to make it a per_instance item
        item op1 using also per_instance;
    };
};
```

6. Save the *cpu_cover_extend.e* file.
7. Open the working directory's *cpu_top.e* file in an editor.
8. Find the lines in the file that look like this:

```
// Extend Coverage:
//import cpu_cover_extend;
```

9. Remove the comment characters in front of the *import* line so the lines look like this:

```
// Extend Coverage:
import cpu_cover_extend;
```

10. Save the *cpu_top.e* file.
11. In SimVision, click **File ›› Reload *e* Files** (in Specview, **File ›› Reload**) to reload the files for test 2.

Tip If you have exited SimVision, you must reinvoke it and load *cpu_tst2.e* again. To do so, execute the **specman -gui** command at the UNIX prompt, click the Load button, select the *cpu_tst2.e* file, and click the OK button. (If you are using Specview, you will need to execute the **specview** command at the UNIX prompt.)

12. Click **Verification ›› *e* Modules** (in Specview, **File ›› Modules**) to confirm that these eight modules are loaded:

```
cpu_instr
cpu_misc
cpu_dut
cpu_drive
cpu_cover
cpu_cover_extend
cpu_top
cpu_tst2
```

13. Click **Verification ›› Test** (in Specview, **Test ›› Test)**.

You should see something similar to the following in the Specman console window. The last line indicates that coverage data was written to an .ecov file (a coverage data file).

```
test
Doing setup…
Generating the test using seed 1
Starting the test…
Running the test…
DUT executing instr   0 :       ADD    REG0x3, REG0x0
DUT executing instr   1 :       ANDI   REG0x3, @0x20
DUT executing instr   2 :       XOR    REG0x3, REG0x2
DUT executing instr   3 :       ADD    REG0x3, REG0x1
DUT executing instr   4 :       SUBI   REG0x3, @0x9f
.
.
.
Last specman tick - stop_run() was called
Normal stop - stop_run() is completed
Checking the test ...
Checking is complete - 0 DUT errors, 0 DUT warnings.
```

```
    Wrote 1 cover_struct to cpu_tst2_1.ecov
```

The coverage data now includes information about the number of samples of each subtype (REG0, REG1, REG2, REG3) of the *cpu_instr_s* type. Each sample also includes information about the new *kind* item. In the next procedure, we view this new information for a series of tests run previously using different seeds.

Viewing Coverage Per Instance

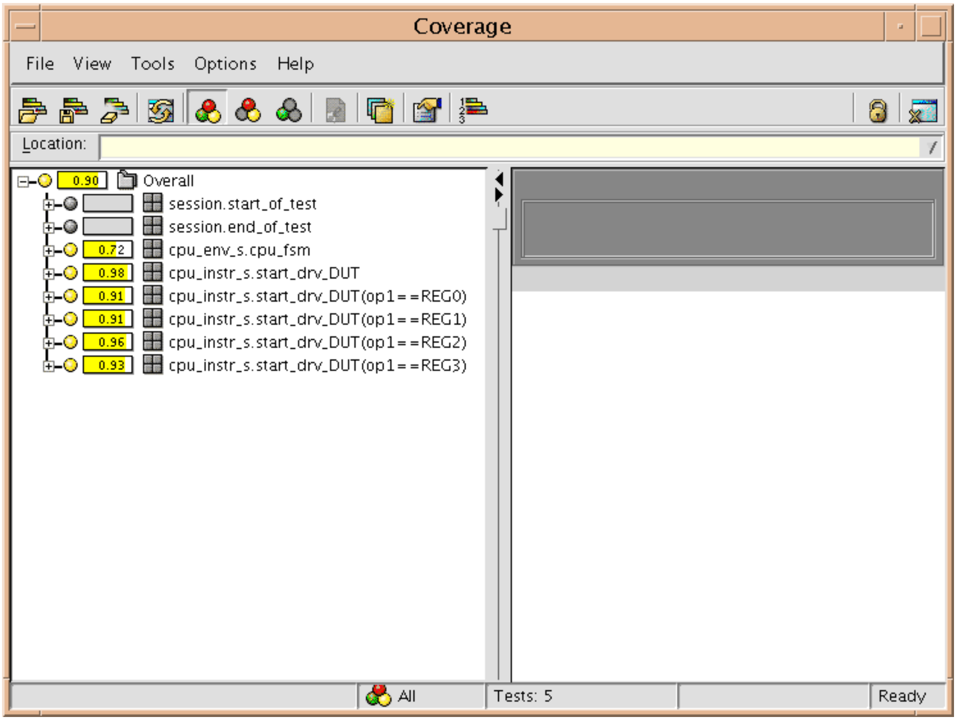
We now look at the per-instance coverage for the op1 subtypes. As in "Viewing Instruction Stream Coverage" on page 8-8, we will load the results of a set of regression tests. These regression tests were run with many different seeds. The results of each test were merged into a file named *regression_B.ecov*. In the following procedure, we load the *regression_B.ecov* file into the Coverage GUI and view the merged coverage data.

Procedure

To view coverage by op1 subtype of the cpu_instr_s instances:

1. Copy the file named *regression_B.ecov* from the *src* directory to your working directory.
2. If you closed the Coverage window after the previous procedure, click **Verification ›› Coverage ›› Show Cover** (in Specview, **Tools ›› Coverage ›› Show Cover)** to open it again.
3. Click **File ›› Clear Data** in the Coverage window to delete the previous test's coverage data.
4. Click **File ›› Read Data** to open the Read Files dialog box. Select *regression_B.ecov* and click the Read button.

   In the left pane, we now see that the instr.start_drv_DUT data has four additional entries: cpu_instr_s.start_drv_DUT(op1==REG0) through cpu_instr_s.start_drv_DUT(op1==REG3).



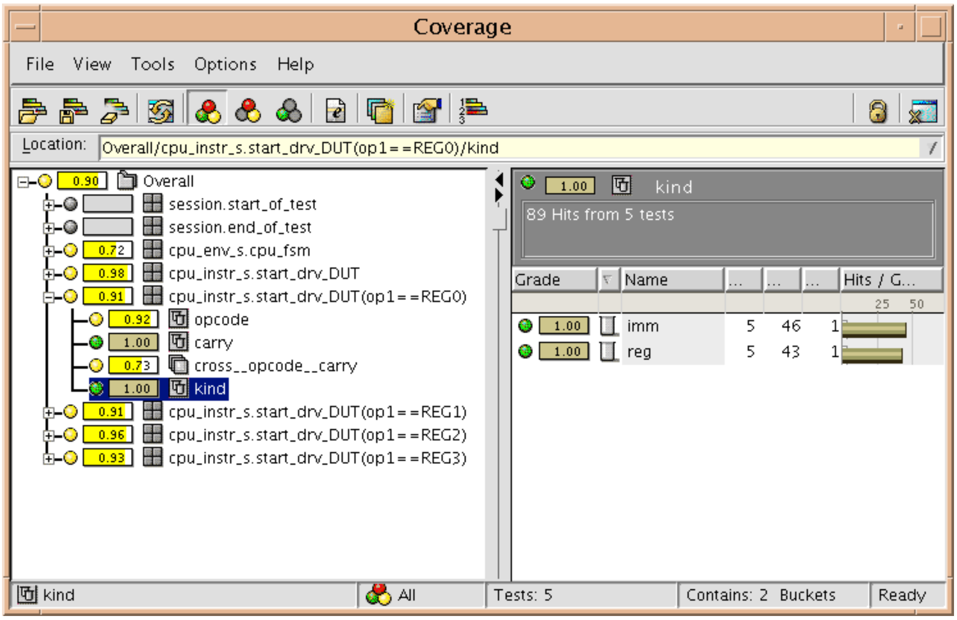
```
0.90  Overall
      session.start_of_test
      session.end_of_test
0.72  cpu_env_s.cpu_fsm
0.98  cpu_instr_s.start_drv_DUT
0.91  cpu_instr_s.start_drv_DUT(op1==REG0)
0.91  cpu_instr_s.start_drv_DUT(op1==REG1)
0.96  cpu_instr_s.start_drv_DUT(op1==REG2)
0.93  cpu_instr_s.start_drv_DUT(op1==REG3)
```

5. In the left pane, click the + to the left of *cpu_instr_s.start_drv_DUT(op1==REG0)*.

   We see that the kind item now appears in the cpu_instr_s.start_drv_DUT group.

6. Click on *kind*.

   The coverage data for the *imm* and *reg* values of kind appears in the right pane. These are the coverage results for kind when the op1 value is REG0, since we selected the *cpu_instr_s.start_drv_DUT(op1==REG0)* instance in the left pane.



```
Location: Overall/cpu_instr_s.start_drv_DUT(op1==REG0)/kind

0.90  Overall
      session.start_of_test
      session.end_of_test
0.72  cpu_env_s.cpu_fsm
0.98  cpu_instr_s.start_drv_DUT
0.91  cpu_instr_s.start_drv_DUT(op1==REG0)
  0.92  opcode
  1.00  carry
  0.73  cross__opcode__carry
  1.00  kind
0.91  cpu_instr_s.start_drv_DUT(op1==REG1)
0.96  cpu_instr_s.start_drv_DUT(op1==REG2)
0.93  cpu_instr_s.start_drv_DUT(op1==REG3)
```

```
1.00  kind
89 Hits from 5 tests
```

Grade	Name	...	...	...	Hits / G...
1.00	imm	5	46	1	
1.00	reg	5	43	1	

```
kind | All | Tests: 5 | Contains: 2 Buckets | Ready
```

7. In the left pane, click the + to the left of *cpu_instr_s.start_drv_DUT* and each of the instances (op1==REG1), (op1==REG2), (op1==REG3) to expand the top instance and all of the subtypes.

   We see that the *cross__opcode__carry* item has a different grade for each instance.

8. Click on each *cross__opcode__carry* item in turn to see which crosses of opcode and carry never occurred at all (shown under *cpu_instr_s.start_drv_DUT*), and which additional crosses never occurred under each particular subtype. The crosses that never occurred are shown in red, and with 0 in the Hits column.

Viewing Corner Case Coverage

Our corner case coverage shows how many times the JMPC opcode was issued when the carry bit was high (1).

Procedure

To view corner case coverage of the JMPC opcode:

1. If you closed the Coverage window after the last procedure, click **Verification ›› Coverage ›› Show Cover** (in Specview, **Tools ›› Coverage ›› Show Cover)** to open it again.

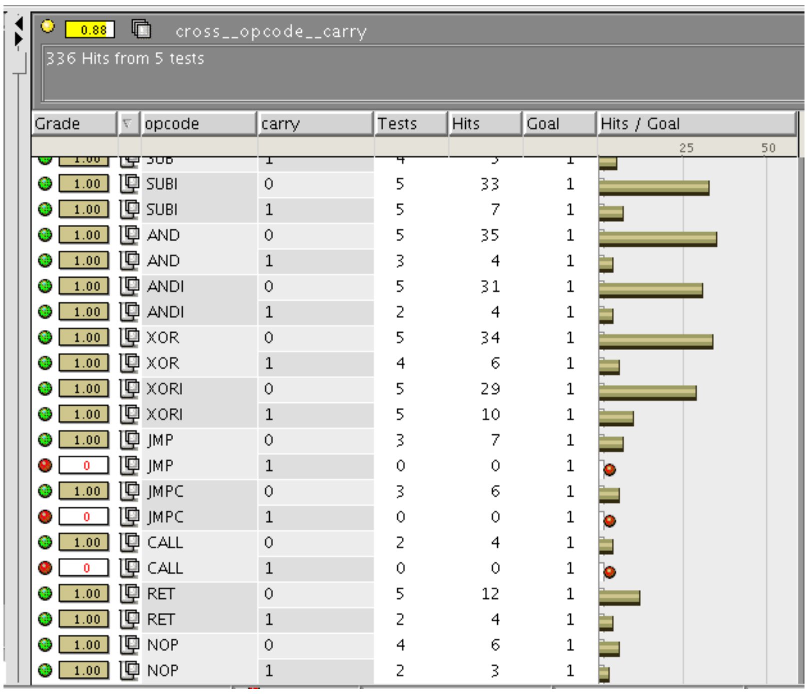
The Coverage window appears.

2. In the left pane, click the + to the left of *cpu_instr_s.start_drv_DUT* and then click on *cross__opcode__carry.*

   The cross-coverage report for opcode and carry appears in the right-hand pane.

3. Scroll down to the JMPC opcode.

   You can see, in the figure below, that carry was 0 each of the 6 times that opcode was JMPC. The combination of opcode JMPC and carry 1 never occurred in this set of 5 tests, which means that there is a coverage hole for that item and it has a grade of 0. The two other red items indicate other combinations of carry and opcode that never were hit in this set of tests. Any grade less than 1.00 is a hole. Grades less than 1.00 but more than 0 are shown in yellow in the coverage GUI.



0.88 cross__opcode__carry

336 Hits from 5 tests

Grade	opcode	carry	Tests	Hits	Goal	Hits / Goal
● 1.00	SUB	1	4	3	1	
● 1.00	SUBI	0	5	33	1	
● 1.00	SUBI	1	5	7	1	
● 1.00	AND	0	5	35	1	
● 1.00	AND	1	3	4	1	
● 1.00	ANDI	0	5	31	1	
● 1.00	ANDI	1	2	4	1	
● 1.00	XOR	0	5	34	1	
● 1.00	XOR	1	4	6	1	
● 1.00	XORI	0	5	29	1	
● 1.00	XORI	1	5	10	1	
● 1.00	JMP	0	3	7	1	
● 0	JMP	1	0	0	1	●
● 1.00	JMPC	0	3	6	1	
● 0	JMPC	1	0	0	1	●
● 1.00	CALL	0	2	4	1	
● 0	CALL	1	0	0	1	●
● 1.00	RET	0	5	12	1	
● 1.00	RET	1	2	4	1	
● 1.00	NOP	0	4	6	1	
● 1.00	NOP	1	2	3	1	

The ability to cross test input with the DUT's internal state yields the valuable information that the tests created so far do not truly test the JMPC opcode. You could raise the weight on JMPC and hope to achieve the goal. However, many simulation cycles would be wasted to cover this corner case. The Specman system lets you attack this type of corner case scenario much more efficiently. In the next chapter you learn how to do this.

In the next chapter, we see how to modify the test files to push the test into a corner that is not being covered well by the current test.

9 Writing a Corner Case Test

Goals for this Chapter

As described in the Functional Test Plan, you want to create a corner case test that generates the JMPC opcode when the carry signal is high.

What You Will Learn

As you work through this chapter, you learn an effective methodology for addressing corner case scenario testing. With Specman's **on-the-fly test generation**, you can direct the test to constantly monitor the state of signals in the DUT and to generate the right test data—at the right time—to reach a corner case scenario. This feature spares you the time-consuming effort required to write deterministic tests to reach the same result.

This chapter introduces the ***e*** constructs shown in Table 9-1.

Table 9-1 New Constructs Used in this Chapter

Construct	How the Construct is Used
'*signal*' * *weight* : *value*	Used as an expression containing a DUT signal within the **select** block of a **keep soft** constraint that controls the distribution of generated values.

The steps required to create the corner case test are:

- Increasing the probability of arithmetic operations.
- Linking JMPC generation to the DUT's carry signal.

The following section describes these tasks in detail.

Increasing the Probability of Arithmetic Operations

The goal of this test is to generate the JMPC opcode only when the carry signal is high (that is, when its value is 1). The carry signal can only be high when arithmetic operations are performed. Therefore, the test should favor generation of arithmetic operations over other types of operations.

Procedure

To increase the probability of arithmetic operations:

1. Copy the *src/cpu_tst3.e* file to the working directory and open the *cpu_tst3.e* file in an editor.
2. Find the portion of the file that contains the **keep soft** constraint.

```
extend cpu_instr_s {
    keep soft opcode == select {
        // high weights on arithmetic

        // generation of JMPC controlled by the carry
        // signal value

    };
};
```

3. Put a high weight on arithmetic operations and low weights on the others.

keeps high weight on arithmetic operations

```
extend cpu_instr_s {
    keep soft opcode == select {
        // high weights on arithmetic
        40 : [ADD, ADDI, SUB, SUBI];
        20 : [AND, ANDI, XOR, XORI];
        10 : [JMP, CALL, RET, NOP];

        // generation of JMPC controlled
        // by the carry signal value
    };
};
```

4. Save the *cpu_tst3.e* file.

Linking JMPC Generation to the Carry Signal

If you generate the list of instructions before simulation, there is only a low probability of driving a JMPC instruction into the DUT when the carry signal is asserted. A better approach is to monitor the carry signal and generate the JMPC instruction when the carry signal is known to be high.

This methodology lets you reach the corner case from multiple paths, in other words, from different opcodes issued prior to the JMPC opcode. This test shows how the DUT behaves under various sequences of opcodes.

Procedure

1. Find the portion of the *cpu_tst3.e* file that looks like this:

```
extend cpu_instr_s {
    keep soft opcode == select {
        // high weights on arithmetic
        40 : [ADD, ADDI, SUB, SUBI];
        20 : [AND, ANDI, XOR, XORI];
        10 : [JMP, CALL, RET, NOP];

        // generation of JMPC controlled by
        // the carry signal value
    };
};
```

2. On a separate line within the **select** block, enter a weight for the JMPC opcode, as a function of the carry signal (weight is 0 when carry = 0, or 90 when carry = 1).

```
extend cpu_instr_s {
    keep soft opcode == select {
        // high weights on arithmetic
        40 : [ADD, ADDI, SUB, SUBI];
        20 : [AND, ANDI, XOR, XORI];
        10 : [JMP, CALL, RET, NOP];

        // generation of JMPC controlled by
        // the carry signal value
        'top.carry' * 90 : JMPC;
    };
};
```

3. Save the *cpu_tst3.e* file.

You are now ready to run this test to create the corner case test scenario. Before running this test, however, you want to address another important part of functional verification: self-checking module creation. In the next chapter, you learn easy, self-checking module creation, another powerful feature provided by the Specman system.

10 Creating Temporal and Data Checks

Goals for this Chapter

In this chapter, you will check timing-related dependencies, and automate the detection of unexpected DUT behavior, by adding a self-checking module to the verification environment.

What You Will Learn

In this chapter, you will learn how to create temporal checks for the state machine control signals. You will also learn how to implement data checks using a reference model.

As you work through this chapter, you will gain experience with two of the Specman verification system's enabling features:

- Specman **temporal constructs—**These powerful constructs let you easily capture the DUT interface specifications, verify the protocols of the interfaces, and efficiently debug them. The temporal constructs minimize the size of complex self-checking modules and significantly reduce the time it takes to implement self-checking.
- Specman **data checking—**Data checking methodology can be flexibly implemented in the Specman system. For data-mover applications like switches or routers, you can use powerful built-in constructs for rule-based checking. For processor-type applications like the application used in this tutorial, reference model methodology is commonly implemented.

This chapter introduces the ***e*** constructs shown in Table 10-1.

Table 10-1 New Constructs Used in this Chapter

Construct	How the Construct is Used
expect	Checks that a temporal expression is true and, if not, reports an error.
check	Checks that a Boolean expression is true and, if not, reports an error.

The steps required to implement these checks are:

1. Creating the temporal checks.
2. Creating the data checks.
3. Running the test with checks.

The following sections describe these tasks in detail.

Creating the Temporal Checks

The design specifications for the CPU require that after entering the *exec_st* state, the *fetch1* signal must be asserted in the following cycle. This is a temporal check because it specifies the correct behavior of DUT signals across multiple cycles.

Procedure

To create the temporal check:

1. Copy the *src/cpu_checker.e* file to the working directory and open the *cpu_checker.e* file in an editor.
2. Find the portion of the file that looks like this:

defines start of exec state

```
// Temporal (Protocol) Checker
event enter_exec_st is
    (change('top.cpu.curr_FSM')and
    true('top.cpu.curr_FSM' == exec_st))
    @sys.cpuclk;
```

defines rise of fetch1

```
event fetch1_assert is
    (change('top.fetch1')and
    true('top.fetch1' == 1)) @sys.cpuclk;

//Interface Spec: After entering instruction
//execution state, fetch1 signal must be
//asserted in the following cycle.
```

3. Define a temporal check for the *enter_exec_st* event by creating an **expect** statement.

```
// Temporal (Protocol) Checker
event enter_exec_st is
    (change('top.cpu.curr_FSM')and
    true('top.cpu.curr_FSM' == exec_st))
    @sys.cpuclk;

event fetch1_assert is
        (change('top.fetch1')and
        true('top.fetch1' == 1)) @sys.cpuclk;
```

issues an error message if fetch1 does not rise exactly one cycle after entering execute state

```
//Interface Spec: After entering instruction
//execution state, fetch1 signal must be
//asserted in the following cycle.
expect @enter_exec_st => {@fetch1_assert}
    @sys.cpuclk else
    dut_error("PROTOCOL ERROR");
```

4. Save the *cpu_checker.e* file.

Creating Data Checks

To determine whether the CPU instructions are executing properly, you need to monitor the program counter, which is updated by many of the control flow operations.

Reference models are not required for data checking. You could use a rule-based methodology. However, reference models are part of a typical strategy for verifying CPU designs. The Specman system supports reference models written in Verilog, VHDL, C, or, as in this tutorial, ***e***. All you need to do is to create checks that compare the program counter in the DUT to their counterparts in the reference model.

Procedure

Creating data checks has two parts:

- Adding the data checks
- Synchronizing the reference model execution with the DUT

Adding the Data Checks

To add the data checks:

1. Find the portion of the *cpu_checker.e* file where the *exec_done* event is defined.

   Notice that there is an event, *exec_done*, and associated method, *on_exec_done*. The Specman system automatically creates an associated method for every event you define. The method is empty until you extend it. The method executes every time the event occurs.

event definition

method associated with event

```
// Data Checker
event exec_done is (fall('top.exec') and
    true('top.rst' == 0))@sys.cpuclk;

on_exec_done() is {
    // Compare PC - program counter
};
.
.
.
```

2. Add a check for the program counter by creating a **check** statement.

issues an error if there is a mismatch in the program counters of the DUT and the reference model

```
// Data Checker
event exec_done is (fall('top.exec') and
    true('top.rst' == 0))@sys.cpuclk;

on_exec_done() is {
    // Compare PC - program counter
    check that sys.cpu_dut.pc ==
        sys.cpu_refmodel.pc else
        dut_error("DATA MISMATCH(pc)");
```

3. Save the *cpu_checker.e* file.

Synchronizing the Reference Model with the DUT

To synchronize the reference model with the DUT:

1. Open the *cpu_drive.e* file in an editor.
2. At the top of the file, find the line that imports the CPU reference model and remove the comment characters from the *import* line.

imports the reference model

```
<'
import cpu_refmodel;

extend sys {
    event cpuclk is
(fall('top.clk')@tick_end);

    cpu_env : cpu_env_s;
    cpu_dut : cpu_dut_s;
    //cpu_refmodel : cpu_refmodel_s;
};
'>
```

3. Find the line that extends the Specman system by creating an instance of the CPU reference model and remove the comment characters.

creates an instance of the reference model

```
<'
import cpu_refmodel;

extend sys {
    event cpuclk is
       (fall('top.clk')@tick_end);

    cpu_env : cpu_env_s;
    cpu_dut : cpu_dut_s;
    cpu_refmodel : cpu_refmodel_s;
};
'>
```

4. Find the line in the *reset_cpu* TCM that resets the reference model and remove the comment characters.

resets the reference model

```
reset_cpu() @sys.cpuclk is {
    'top.rst' = 0;
    wait [1] * cycle;
    'top.rst' = 1;
    wait [5] * cycle;
    sys.cpu_refmodel.reset();
    'top.rst' = 0;
};
```

5. Find the line in the *drive_one_instr* TCM that executes the reference model when the DUT is in the execute state and remove the comment characters.

```
    // execute instr in refmodel
    sys.cpu_refmodel.execute(instr,sys.cpu_dut);
};
```

6. Save the *cpu_drive.e* file.

Running the Simulation

This procedure, which involves loading the appropriate files and then executing the test, is very similar to the procedure you used in previous chapters to generate other tests.

The only difference is that this time you will include the reference model and checks.

Procedure

To run the simulation:

1. Open the working directory's copy of the *cpu_top.e* file in an editor.
2. Find the lines in the file that look like this:

```
// Add Checking:
//import cpu_checker;
```

3. Remove the comment characters in front of the *import* line so the lines look like this:

```
// Add Checking:
import cpu_checker;
```

4. Save the *cpu_top.e* file.
5. Copy the *src/cpu_refmodel.e* file to the working directory.
6. Invoke Simvision, if it is not already running:

```
% specman -gui &
```

   If you are using Specview, execute the following command:

```
% specview &
```

7. Click **Verification ›› Restore Specman State** (in Specview, **File ›› Restore ›› Restore to Last State)** to remove any loaded modules from the current session.
8. Click **File ›› Load *e* File** (in Specview, **File ›› Load**), select the *cpu_tst3.e* file, and click OK.

   Remember, this is the test that you edited in Chapter 9 "Writing a Corner Case Test".

9. Click **Verification** ›› **Test** (in Specview, **Test** ›› **Test)** to run the simulation.

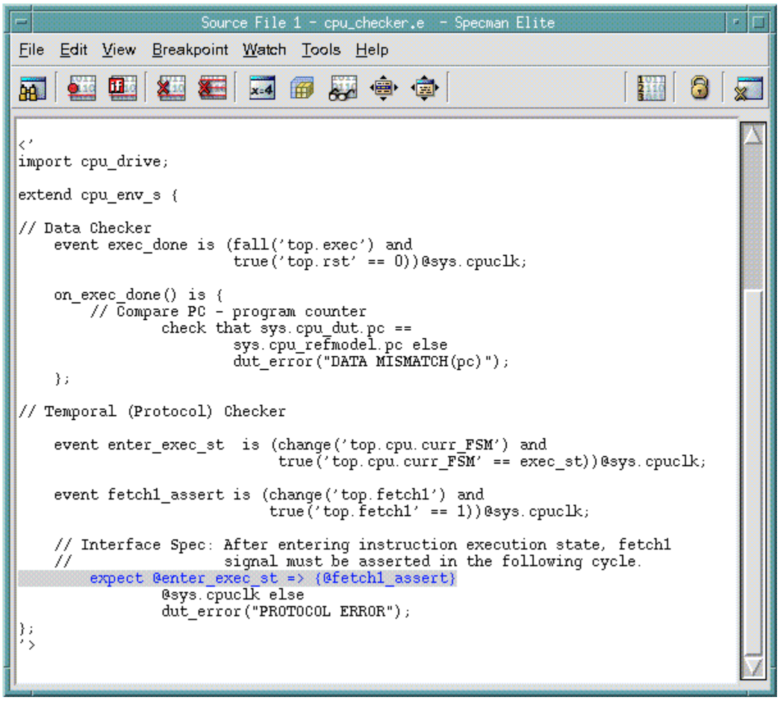
   It looks like we hit a bug here. The Specman system is reporting a protocol violation.

```
test
Doing setup ...
Generating the test using seed 7...
Starting the test ...
Running the test ...
DUT executing instr   0 :        XOR     REG0x0, REG0x2
DUT executing instr   1 :        ADD     REG0x2, REG0x3
DUT executing instr   2 :        CALL    REG0x1, @0x09
DUT executing instr   3 :        ANDI    REG0x1, @0x65
DUT executing instr   4 :        AND     REG0x1, REG0x0
DUT executing instr   5 :        ADD     REG0x0, REG0x2
DUT executing instr   6 :        SUB     REG0x0, REG0x1
DUT executing instr   7 :        XORI    REG0x3, @0xca
DUT executing instr   8 :        ADDI    REG0x3, @0xcb
DUT executing instr   9 :        ADDI    REG0x1, @0xc0
DUT executing instr  10 :        ANDI    REG0x1, @0xd6
DUT executing instr  11 :        SUB     REG0x0, REG0x2
.
.
.
*** Dut error at time 1346
        Checked at line 41 in @cpu_checker
        In cpu_env_s-@0:

PROTOCOL ERROR

Will stop execution immediately (check effect is ERROR)

   *** Error: A Dut error has occurred

   *** Error: Error during tick command
```

Tip If you are using a version of Specman that is not the same as the version of this tutorial, it is possible that the DUT error will not occur on the first test or that it will occur at a different time. If it does not occur, reload the test and specify a seed other than the default (1). When the error occurs, note the exact time when it occurred. You will use this information in the next chapter to debug the error.

10. Click the “Checked at line 41 in @cpu_checker” error hyperlink to view the line in the source that generated this message.

    This message comes from the checker module that you just created.


Source File 1 - cpu_checker.e - Specman Elite
File Edit View Breakpoint Watch Tools Help

```
<'
import cpu_drive;

extend cpu_env_s {

// Data Checker
    event exec_done is (fall('top.exec') and
                        true('top.rst' == 0))@sys.cpuclk;

    on_exec_done() is {
        // Compare PC - program counter
                check that sys.cpu_dut.pc ==
                        sys.cpu_refmodel.pc else
                        dut_error("DATA MISMATCH(pc)");
    };

// Temporal (Protocol) Checker

    event enter_exec_st  is (change('top.cpu.curr_FSM') and
                             true('top.cpu.curr_FSM' == exec_st))@sys.cpuclk;

    event fetch1_assert is (change('top.fetch1') and
                            true('top.fetch1' == 1))@sys.cpuclk;

    // Interface Spec: After entering instruction execution state, fetch1
    //                 signal must be asserted in the following cycle.
        expect @enter_exec_st => {@fetch1_assert}
                @sys.cpuclk else
                dut_error("PROTOCOL ERROR");
};
'>
```

11. Close the File cpu_checker.e window.

Do not quit SimVision/Specview at this time. The first procedure in the next chapter starts from this point.

In the next chapter, you learn how to identify the conditions under which this bug occurs and how to bypass the bug until it can be fixed.

11 Analyzing and Bypassing Bugs

Goals for this Chapter

The main goal for this chapter is to debug the temporal error generated during your previous tutorial session (Chapter 10 “Creating Temporal and Data Checks”). At the end of this chapter, you also learn how to direct the generator to bypass a test scenario that causes an error.

What You Will Learn

As you work through this chapter, you gain experience with two of the Specman system’s enabling features:

- **The Specman debugger—**Provides powerful debugging capability with visibility into the HDL design.
- **The Specman bypass feature—**Lets you temporarily prevent the Specman system from generating test data that reveals a bug in the design. With this feature you can continue testing while the bug is being fixed.

This chapter introduces the SimVision/Specview menu commands shown in Table 11-1 and the generation debugger menu commands shown in Table 11-2.

Table 11-1 New SimVision/Specview Menu Commands Used in this Chapter

SimVision & (Specview) Command	How the Command is Used
Verification ›› Specman Debugger ›› Open Thread Browser (Debug ›› Thread Browser)	Opens the Thread Browser, which displays all of the TCMs (threads) that are currently active.

Table 11-1 New SimVision/Specview Menu Commands Used in this Chapter

SimVision & (Specview) Command	How the Command is Used
Verification ›› Specman Debugger ›› Open Debug Window (Debug ›› Open Debug Window)	Opens the Debugger window, which displays the source for the current thread with the current line highlighted.

Table 11-2 Generation Debugger Menu Commands Used in this Chapter

Generation Debugger Command	How the Command is Used
View ›› Print	Displays the current value of an ***e*** variable.
Breakpoint ›› Set Breakpoint ›› Break	Sets a breakpoint on the currently highlighted line of ***e*** code.
Run ›› Step Any	Advances simulation to the next line of ***e*** code executed in any thread.
Run ›› Step	Advances simulation to the next line of ***e*** code executed in the current thread.

The steps for debugging the temporal error are:

1. Displaying DUT values.
2. Setting breakpoints.
3. Stepping the simulation.
4. Bypassing bugs.

The following sections describe how to perform these tasks.

Displaying DUT Values

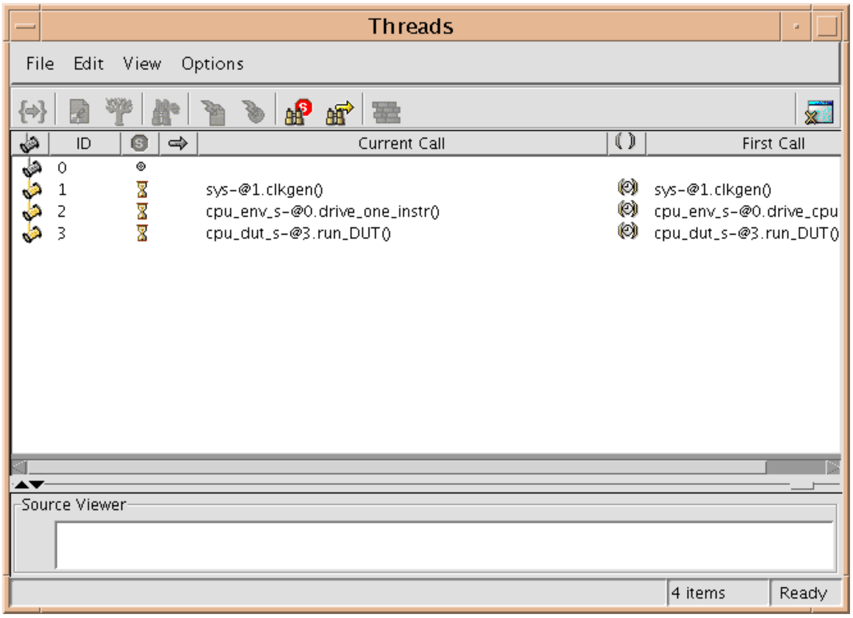
If you have just completed Chapter 10 “Creating Temporal and Data Checks”, the PROTOCOL ERROR message is still displayed on the Specman console window. If you exited SimVision/Specview, you will have to reinvoke SimVision/Specview and run the simulation again, as described in “Running the Simulation” on page 10-7. With SimVision/Specview running, continue with the procedure below.

Procedure

To display DUT values:

1. In SimVision, click **Verification ›› Specman Debugger ›› Open Thread Browser** (in Specview, **Debug ›› Thread Browser)**.

   The Thread Browser appears.



	ID			Current Call		First Call
	0					
	1			sys-@1.clkgen()		sys-@1.clkgen()
	2			cpu_env_s-@0.drive_one_instr()		cpu_env_s-@0.drive_cpu
	3			cpu_dut_s-@3.run_DUT()		cpu_dut_s-@3.run_DUT()

   The Thread Browser indicates the status of each TCM that is currently active in the Specman system:

   - Clock generation
   - Drive and Sample CPU
   - DUT

   To debug the error, look first at the TCM that drives the DUT.

2. Click on line 2, the line for *cpu_env_s-@0.drive_cpu*, to display the corresponding source file for this thread.

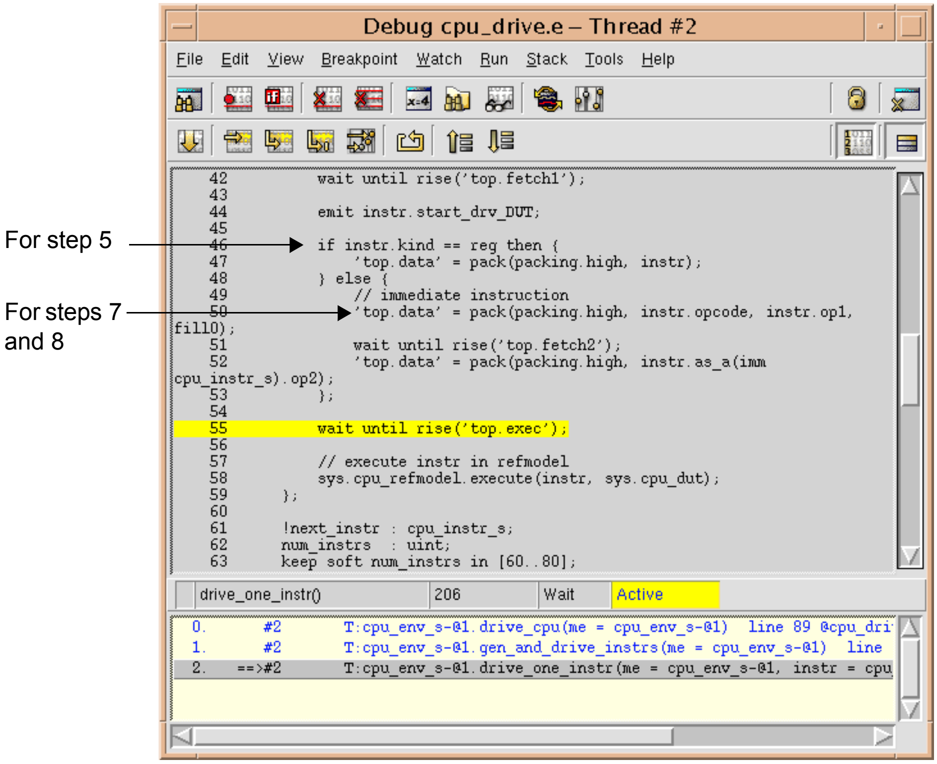
   The Source File window at the bottom of the screen displays a section of the *cpu_drive.e* ***e*** code module, with line 55 highlighted. The highlighted line shows that the *drive_one_instr* TCM is waiting for the *top.exec* signal to rise.

3. Click **View ›› Show Source** to open the Debug window.

   The Debug window appears, displaying the cpu_drive.e file.

4. Click **View ›› Line Numbers** to display line numbers.

   Line 55 is highlighted.


For step 5
For steps 7 and 8
Debug cpu_drive.e – Thread #2
File Edit View Breakpoint Watch Run Stack Tools Help

```
42          wait until rise('top.fetch1');
43
44          emit instr.start_drv_DUT;
45
46          if instr.kind == reg then {
47              'top.data' = pack(packing.high, instr);
48          } else {
49              // immediate instruction
50              'top.data' = pack(packing.high, instr.opcode, instr.op1,
fill0);
51              wait until rise('top.fetch2');
52              'top.data' = pack(packing.high, instr.as_a(imm
cpu_instr_s).op2);
53          };
54
55          wait until rise('top.exec');
56
57          // execute instr in refmodel
58          sys.cpu_refmodel.execute(instr, sys.cpu_dut);
59      };
60
61      !next_instr : cpu_instr_s;
62      num_instrs  : uint;
63      keep soft num_instrs in [60..80];
```

```
drive_one_instr()   206   Wait   Active
```

```
0.      #2      T:cpu_env_s-@1.drive_cpu(me = cpu_env_s-@1)  line 89 @cpu_dri
1.      #2      T:cpu_env_s-@1.gen_and_drive_instrs(me = cpu_env_s-@1)  line
2.  ==>#2       T:cpu_env_s-@1.drive_one_instr(me = cpu_env_s-@1, instr = cpu
```

5. To find out the current instruction type, use your mouse to highlight the phrase *instr.kind*, located nine lines above the highlighted **wait** statement (that is, at line 46).

6. Click the “Print selection in main window” button in the Debug window toolbar.

   Note To display the button title, hold the mouse over the button. The “Print selection in main window” button is the sixth from the left on the top row of buttons.

   The *instr.kind* value, instr.kind imm, is printed in the Specman console window.

7. In a similar fashion, highlight the phrase *instr.opcode* in line 50 and then click Print.

In the Specman console window, you can see that the value of opcode is JMPC.

8. Optionally you can find out the value of any HDL signals. For example, to display the value of *top.data*, highlight the phrase ‘*top.data*’, including the quotes, and then click Print.

Setting Breakpoints

You have determined that the bug appears on an immediate instruction when the opcode is JMPC. It may be possible to narrow down even further the conditions under which the bug occurs. You can set a breakpoint on the statement that drives the immediate instruction data into the DUT to see what the operands of the instruction are.

Procedure

To set a breakpoint:

1. Highlight any portion of the following line (line 52):

   ```
   'top.data' = pack(packing.high, instr.as_a(imm cpu_instr_s).op2);
   ```

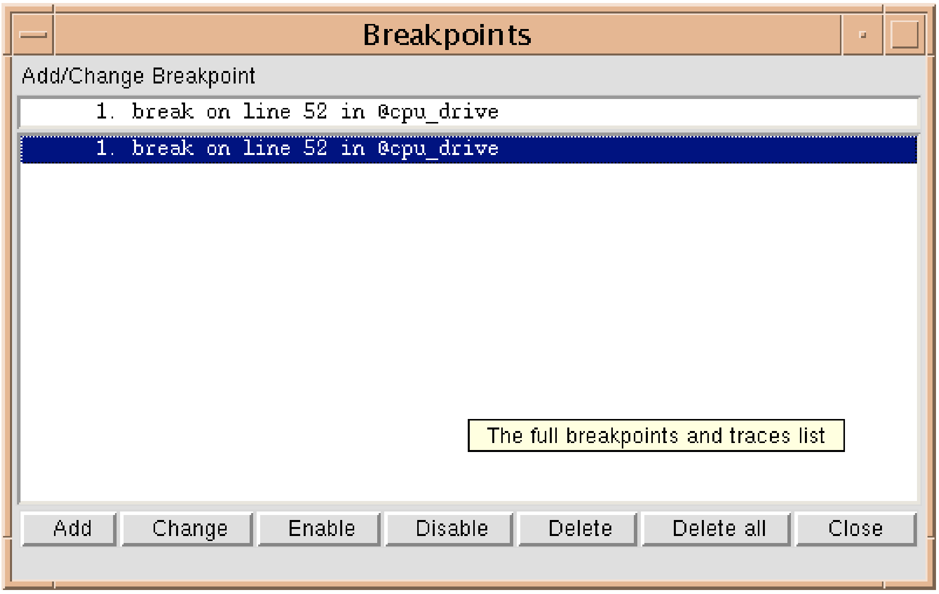
2. Click **Breakpoint ›› Set Breakpoint ›› Break**.

   The line should now be red and underlined to indicate a breakpoint has been set.

3. Click **Breakpoint ›› Show All Breakpoints** to activate the breakpoint just before the error occurs (at system time 1346).

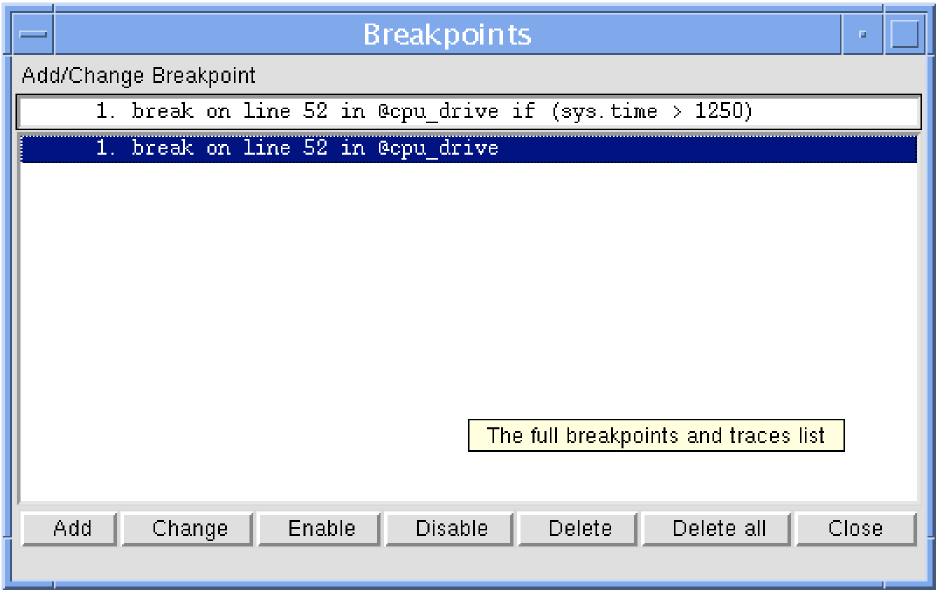
   The system time unit is displayed just below the code on the gray line, next to drive_one_instr().

   The Breakpoints window appears.



```
1. break on line 52 in @cpu_drive
1. break on line 52 in @cpu_drive
```

4. Edit the breakpoint description in the Breakpoints window to append the condition "if (sys.time > 1250)" as shown in the following figure, and click Change:



```
1. break on line 52 in @cpu_drive if (sys.time > 1250)
1. break on line 52 in @cpu_drive
```

When you click the Change button, the breakpoint is modified, and the condition you specified takes effect.

Tip If the error occurred at a simulation time other than 1346, choose a different value for the sys.time expression that is at least 50 time units before the time the error occurred.

Stepping the Simulation

You can trace the exact execution order of the ***e*** code by stepping through the simulation.

Procedure

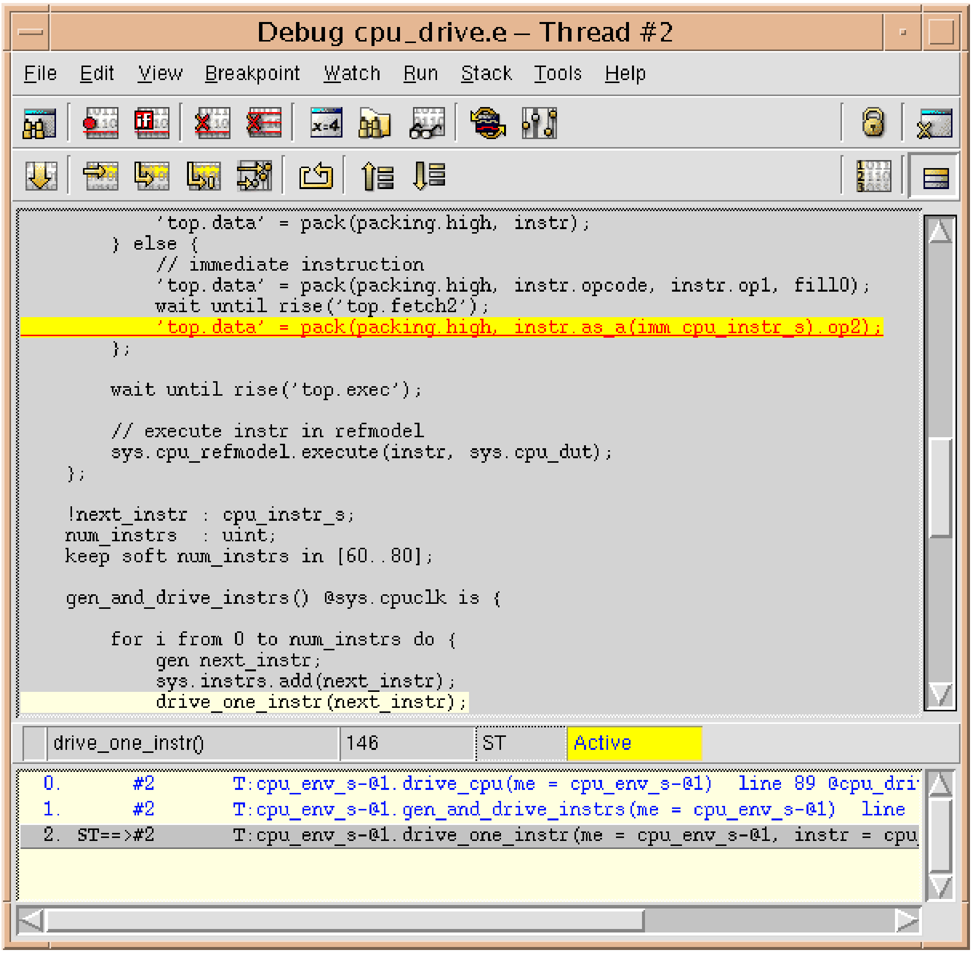
To step through the simulation:

1. In SimVision, click **File** ›› **Reload *e* Files** (in Specview, **File** ›› **Reload**) to run the simulation in debug mode.

   The Debug window closes when you reload the design.

2. In SimVision, click **Verification** ›› **Test** (in Specview, **Test** ›› **Test)**.

   The simulation stops at the breakpoint, and the Debug window opens.


Debug cpu_drive.e – Thread #2
File Edit View Breakpoint Watch Run Stack Tools Help

```
            'top.data' = pack(packing.high, instr);
        } else {
            // immediate instruction
            'top.data' = pack(packing.high, instr.opcode, instr.op1, fill0);
            wait until rise('top.fetch2');
            'top.data' = pack(packing.high, instr.as_a(imm cpu_instr_s).op2);
        };

        wait until rise('top.exec');

        // execute instr in refmodel
        sys.cpu_refmodel.execute(instr, sys.cpu_dut);
    };

    !next_instr  : cpu_instr_s;
    num_instrs   : uint;
    keep soft num_instrs in [60..80];

    gen_and_drive_instrs() @sys.cpuclk is {

        for i from 0 to num_instrs do {
            gen next_instr;
            sys.instrs.add(next_instr);
            drive_one_instr(next_instr);
```

```
drive_one_instr()    146    ST    Active
```

```
 0.       #2       T:cpu_env_s-@1.drive_cpu(me = cpu_env_s-@1)  line 89 @cpu_dri
 1.       #2       T:cpu_env_s-@1.gen_and_drive_instrs(me = cpu_env_s-@1)  line
 2.  ST==>#2       T:cpu_env_s-@1.drive_one_instr(me = cpu_env_s-@1, instr = cpu
```

3. To step to the next source line in any subsequent thread, click **Run** ›› **Step Any** in the Debug window.

4. Continue clicking **Run** ›› **Step Any** (or clicking the Step Any button) until the current thread is Thread #3 in the cpu_dut.e file, as indicated in the title bar at the top of the window.

   Note If the current thread switches to Thread #0 (the Specman tick thread), the source file is not visible. You should continue clicking **Run** ›› **Step Any**.

5. Click **Run** ›› **Step** in the Debug window to step through the simulation within the current thread (Thread #3).

6. Continue clicking **Run** ›› **Step** (or clicking the Step button) for about 35 to 40 steps until you hit the PROTOCOL error.

   The Step button is grayed out, and the PROTOCOL error is displayed in the Specman console window. For the purpose of simplifying this tutorial, we planted an obvious bug in the DUT. Whenever a JMPC instruction jumps to a location greater than 10, execution requires two extra cycles to complete.

Bypassing the Bug

A common problem in traditional test generation methodology is that when there is a bug in the design, verification cannot continue until the bug is fixed. There is no way to prevent the generator from creating tests that hit the bug.

The Specman system's extensibility feature, however, lets you temporarily prevent generation of the conditions that cause the bug to be revealed.

This particular bug seems to surface when the JMPC operation is performed using a memory location greater than 10. To continue testing other scenarios, you simply extend the test constraints to prevent the Specman system from generating this combination.

Procedure

To bypass the JMPC bug:

1. Copy the *src/cpu_bypass.e* file to the working directory.

2. Open the *cpu_bypass.e* file in an editor.

3. Review the **keep** constraint.

```
<'
extend imm cpu_instr_s {

    keep (opcode == JMPC) => op2 < 10 ;

};
'>
```

4. In SimVision, click **Verification** ›› **Specman Debugger** ›› **Delete All Breakpoints** (in Specview, **Debug** ›› **All Breakpoints** ›› **Delete All Breakpoints)**.

5. In SimVision, click **File** ›› **Reload *e* Files** (in Specview, **File** ›› **Reload**).

6. Click **File ›› Load *e* File** (in Specview, **File ›› Load**), select the *cpu_bypass.e* file, and click OK.
7. Click **Verification ›› Test** (in Specview, **Test ›› Test**).

   This time, the test runs to completion.

Tutorial Summary

Congratulations! You have successfully completed the major steps required to verify a device with the Specman verification system.

In this tutorial:

- You captured the interface specifications for the CPU instructions in ***e*** and created the instruction stream.
- You used specification constraints to ensure that only legal instructions were generated. You used test constraints to create a simple go/no-go test.
- You created a Specman TCM (time consuming method) to define the driver protocol and then drove the generated CPU instruction stream into the DUT. The results confirmed that you had generated the first test and driven it correctly into the design.
- Using Specman's powerful constraint-driven generator, you generated many sets of instructions. Using weight to control the generation value distribution, you effectively focused these sets of instructions on the commonly executed portion of the CPU DUT.
- Using Specman's unique Functional Coverage Analyzer, you accurately measured the effectiveness of the coverage of the regression tests. You identified a corner case "hole" by viewing the graphical coverage reports.
- To address the corner case scenario, you used Specman's powerful on-the-fly generation capability to generate a test based on the internal state of the design during simulation. Compared to the traditional deterministic test approach, this approach tests the corner case much more effectively from multiple paths.
- You then used the unique temporal constructs provided by the Specman system to create a self-checking monitor for verifying protocol conformance.
- When the self-checking monitor revealed a bug, the Specman debugger provided extensive features to debug the design efficiently.

Note You have created this verification environment, including self-checking modules and functional coverage analysis, in a short period of time. Once the environment is established, creating a large number of effective tests is merely one click away. The ultimate advantage of using the Specman system is a tremendous reduction in verification time and resources.

eRM Conventions in the Tutorial Code

The ***e*** code for this tutorial was created using the Cadence ***e*** Reuse Methodology (***e***RM). ***e***RM is a coding style that promotes reusability through standardization of the following elements of the code:

- Directory structure
- Inclusion of a PACKAGE_README.txt file
- Consistent prefixes for ***e*** code files (cpu_*module*.e)
- Consistent naming conventions for types, structs, and units (*type*_t, *struct*_s, *unit*_u)

Detailed information about ***e***RM coding and applications is available in the ***e** Reuse Methodology (**e**RM) Developer Manual*.

A Design Specifications for the CPU

This document contains the following specifications:

- CPU instructions
- CPU interface
- CPU register list

CPU Instructions

The instructions are from three main categories:

- **Arithmetic instructions**—ADD, ADDI, SUB, SUBI
- **Logic instructions**—AND, ANDI, XOR, XORI
- **Control flow instructions**—JMP, JMPC, CALL, RET
- **No-operation instructions**—NOP

All instructions have a 4-bit opcode and two operands. The first operand is one of four 4-bit registers internal to the CPU. This same register stores the result of the operation, in the case of arithmetic and logic instructions.

Based on the second operand, there are two categories of instructions:

- **Register instructions**—The second operand is another one of the four internal registers.
- **Immediate instructions**—The second operand is an 8-bit value contained in the next instruction. When the opcode is of type JMP, JMPC, or CALL, this operand must be a 4-bit memory location.

Figure A-1 Register Instruction

byte	1							
bit	7	6	5	4	3	2	1	0
	opcode				op1		op2	

Figure A-2 Immediate Instruction

byte	1								2							
bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
	opcode				op1		don't care		op2							

Table A-1 shows a summary description of the CPU instructions.

Table A-1 Summary of Instructions

Name	Opcode	Operands	Comments
ADD	0000	register, register	ADD; PC <- PC + 1
ADDI	0001	register, immediate	ADD immediate; PC <- PC + 2
SUB	0010	register, register	SUB; PC <- PC + 1
SUBI	0011	register, immediate	SUB immediate; PC <- PC + 2
AND	0100	register, register	AND; PC <- PC + 1
ANDI	0101	register, immediate	AND immediate; PC <- PC + 2
XOR	0110	register, register	XOR; PC <- PC + 1
XORI	0111	register, immediate	XOR immediate; PC <- PC + 2
JMP	1000	immediate	JUMP; PC <- immediate value

JMPC	1001	immediate	JUMP on carry;
			if carry = 1 PC <- immediate value
			else PC <- PC + 2

CALL	1010	immediate	Call subroutine;
			PC <- immediate value;
			PCS <- PC + 2

Table A-1 Summary of Instructions (continued)

Name	Opcode	Operands	Comments
RET	1011		Return from call; PC <- PCS
NOP	1100		Undefined command

CPU Interface

The CPU has three inputs and no outputs, as shown in Table A-2.

Table A-2 Interface List

Function	Direction	Width	Signal Name
CPU instruction	input	8 bits	data
clock	input	1 bit	clock
reset	input	1 bit	rst

When the CPU is reset by the *rst* signal, *rst* must return to its inactive value no sooner than *min_reset_duration* and no later than *max_reset_duration*.

CPU Register List

The CPU has six 8-bit registers and one 4-bit register, as shown in Table A-3.

Table A-3 Register List

Function	Width	Register Name
state machine register	4 bits	curr_FSM
program counter	8 bits	pc
program counter stack	8 bits	pcs
register 0	8 bits	r0
register 1	8 bits	r1

Table A-3 Register List

Function	Width	Register Name
register 2	8 bits	r2
register 3	8 bits	r3