Verisity®

Specman Elite™

Tutorial

Version 4.0.1

Table of Contents

1 Introduction

Overview

The Specman™ Elite™ verification system provides benefits that result in:

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

The Specman Elite 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

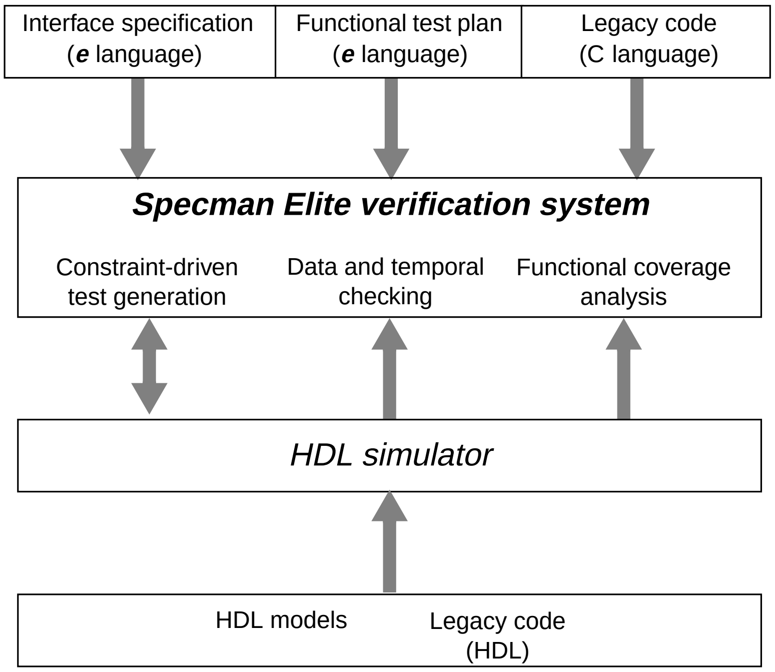
The Specman Elite system provides three main enabling technologies that enhance your productivity:

- **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 Elite system and its interface with an HDL simulator.

Figure 1-1 The Specman Elite System Automates Verification


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

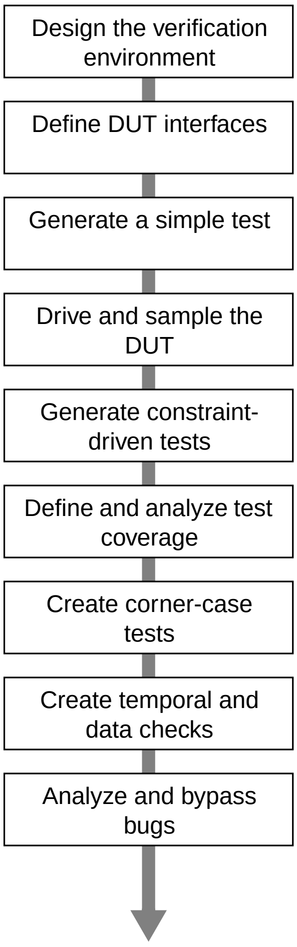
Tutorial Goals

This tutorial is for use with Specman Elite version 4.0 and higher.

The goal of this tutorial is to give you first-hand experience in how the Specman Elite 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 Elite system to create a verification environment for a simple CPU design.

Figure 1-2 Tutorial Verification Task Flow


Design the verification environment
Define 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 set up the tutorial environment.

To set up the tutorial environment:

1. Download the Specman Elite software and tutorial files (see “Setting up the Tutorial Environment” on page A-1).
2. Install the Specman Elite software.
3. Install the tutorial files.

See Appendix A, “Setting up the Tutorial Environment”, for detailed instructions.

Note Even if Specman Elite software is currently installed in your environment, you still have to download and install the tutorial files.

Document Conventions

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

Table 1-1 Document Conventions

Visual Cue	Meaning
`courier`	Specman Elite 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 Elite 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 Elite system, SN> indicates the Specman Elite prompt.

2 Understanding the Environment

Goals for this Chapter

This tutorial uses a simple CPU design to illustrate the benefits of using the Specman Elite 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 Elite 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:

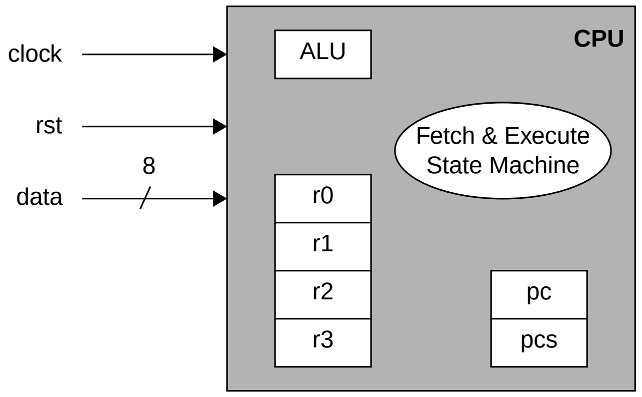
- Design specifications
- Interface specifications
- Functional test plan
- Overall verification environment

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

The Design Specifications

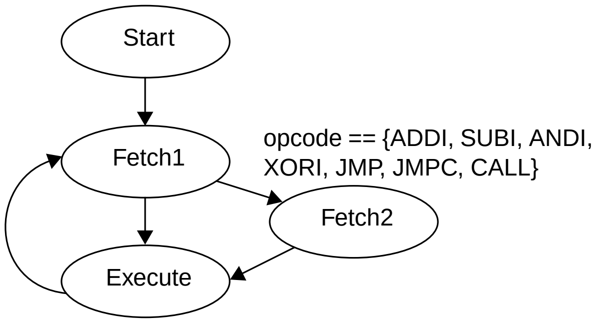
The device under test (DUT) is an 8-bit CPU with a reduced instruction set (Figure 2-1).

Figure 2-1 CPU Block-Level Diagram



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*, *start*. 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 is one of four 4-bit registers internal to the CPU. The second operand is determined by the type of instruction:

- **Register instructions—**The second operand is 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 test to confirm 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 to gain 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 15 times, each time with a different random seed.

Test 3

Test Objective

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

Test Specifications

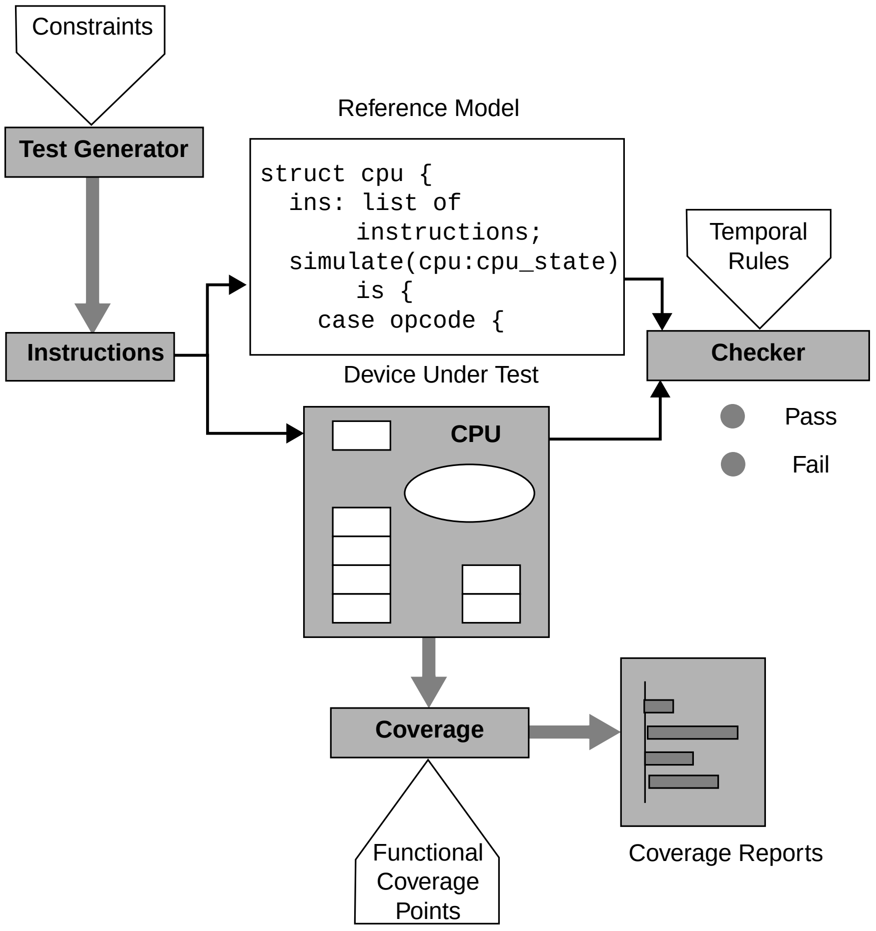
- Generate many arithmetic opcodes to increase the chances of carry bit assertion.
- Monitor the DUT and use on-the-fly generation to generate many JMPC opcodes when the carry signal is high.

Overview of the Verification Environment

The overall test strategy, shown in Figure 2-5, is to:

- Constrain the Specman Elite test generator to create 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 {
  ins: list of
       instructions;
  simulate(cpu:cpu_state)
       is {
    case opcode {
```

Because the focus of this tutorial is the Specman Elite 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 Elite. 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 Elite 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 Elite 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 for the CPU instructions to the Specman Elite system of data structures.
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 B, “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.

```
CPU_instr.e: Basic structure of CPU instructions
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

defines the internal registers

when complete, this structure defines a valid CPU instruction

// indicates that rest of line is a comment

when complete, this construct adds the CPU instruction set to the Specman Elite system

```
<'
type cpu_opcode: [ // Opcodes
    ADD, ADDI, SUB, SUBI,
    AND, ANDI, XOR, XORI,
    JMP, JMPC, CALL, RET,
    NOP
] (bits: 4);

type reg: [ // Register names
    REG0, REG1, REG2, REG3
] (bits:2);

struct instr {

    // 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

};

extend sys {
    // creates the stream of instructions

};
'>
```

---

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

   Use the enumerated types, *cpu_opcode* and *reg*, to define the types of these fields. To indicate that the Specman Elite 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 instr {
    %opcode    :cpu_opcode;
    %op1       :reg;

    // 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 instr {
    %opcode    :cpu_opcode;
    %op1       :reg;
    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 instr {
    %opcode    :cpu_opcode;
    %op1       :reg;
    kind       :[imm, reg];

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

    // defines 2nd op of imm instruction
    when imm instr {
        %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 instr {
.
.
.
    // 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 instr {
.
.
.
    // ensures 4-bit addressing scheme
    when imm instr {
        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 Elite system (**sys**) to include a list of CPU instructions. **sys** is a built-in Specman Elite 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 Elite system:

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

};
```

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

   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 instr;
};
```

   The exclamation point preceding the field name *instrs* tells the Specman Elite 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.

What You Will Learn

In this chapter, you learn how to create different types of tests easily by specifying test constraints in the Specman Elite system. Test constraints direct the Specman Elite generator to a specific test described in the functional test plan. This chapter illustrates how the Specman Elite 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 Elite 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 Specview menu commands shown in Table 4-1.

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

Construct	How the Construct is Used
extend	Adds constraints to the **sys** and *instr* 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.
Command	**How the Command is Used**
File>>Load	Loads uncompiled ***e*** modules into the Specman Elite system.
File>>Modules	Lists the ***e*** modules you have loaded into the Specman Elite system.
Test>>Test	Generates a test based on the constraints you specify.
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 Specview menu commands presented in this tutorial can be issued by clicking a button. For example, clicking the Load button in the Specview main window is the same as choosing Load from the File menu. Similarly, you can click the Modules button instead of choosing Modules from the File menu or click Test instead of choosing Test from the Test menu.

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 Elite 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 instr {
    // 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 instr {
    //test constraints
    keep opcode in [ADD, ADDI];
    keep op1 == REG0;
    when reg instr { keep op2 == REG1; };
    when imm instr { 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 Elite system. Before the system can generate the test, you must load all the modules.

Procedure

To load all modules:

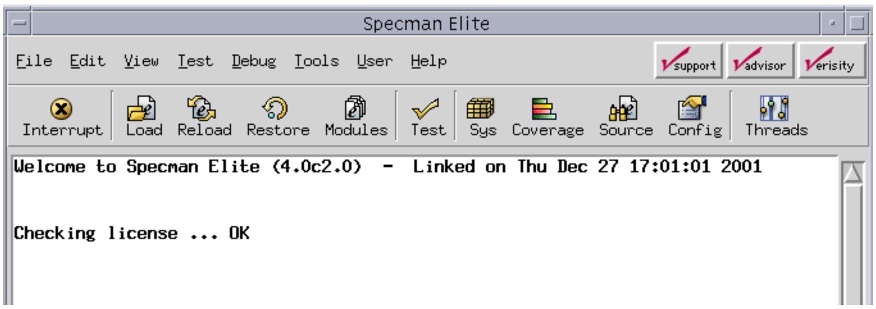
1. Copy the *src/CPU_top.e* file to the working directory.
2. Copy the *src/CPU_misc.e* file to the working directory.

   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 Elite’s graphical user interface, Specview™:

   ```
   % specview &
   ```

Tip If the Specview main window (Figure 4-1) does not appear, make sure that you have defined the Specman Elite environment variables correctly. You can source the *install_dir*/*release_number*/*env.csh* file to set these variables.

Figure 4-1 Specview Main Window


Specman Elite
File Edit View Test Debug Tools User Help
support advisor Verisity
Interrupt Load Reload Restore Modules Test Sys Coverage Source Config Threads

```
Welcome to Specman Elite (4.0c2.0)  -  Linked on Thu Dec 27 17:01:01 2001


Checking license ... OK
```

4. Choose Load from the File menu or click the Load button.

   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.

   Specview automatically loads all four files contained in your working directory. In the Specview main window, 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 /tutorial/CPU_tst1.e ...
```

Tip If the *CPU_tst1.e* file name does not appear in the dialog box, you probably did not invoke 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 Specview main window, and the error location will be displayed in the Debugger window.

To see a list of loaded modules, choose Modules from the File menu or click the Modules button.

There should be four modules loaded:

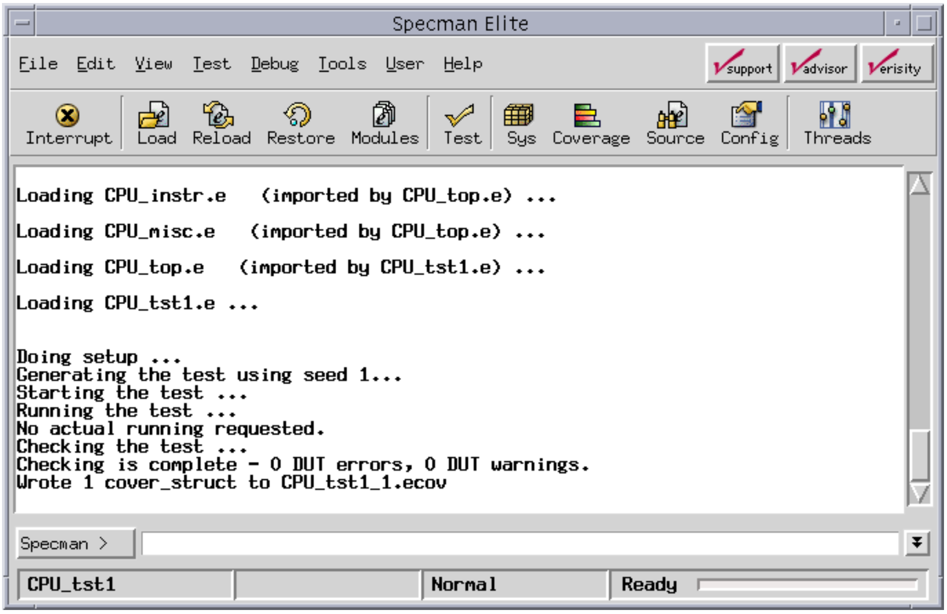
```
CPU_instr
CPU_misc
CPU_top
CPU_tst1
```

Generating the Test

To generate the test:

1. In the Specview main window, click the Test button.

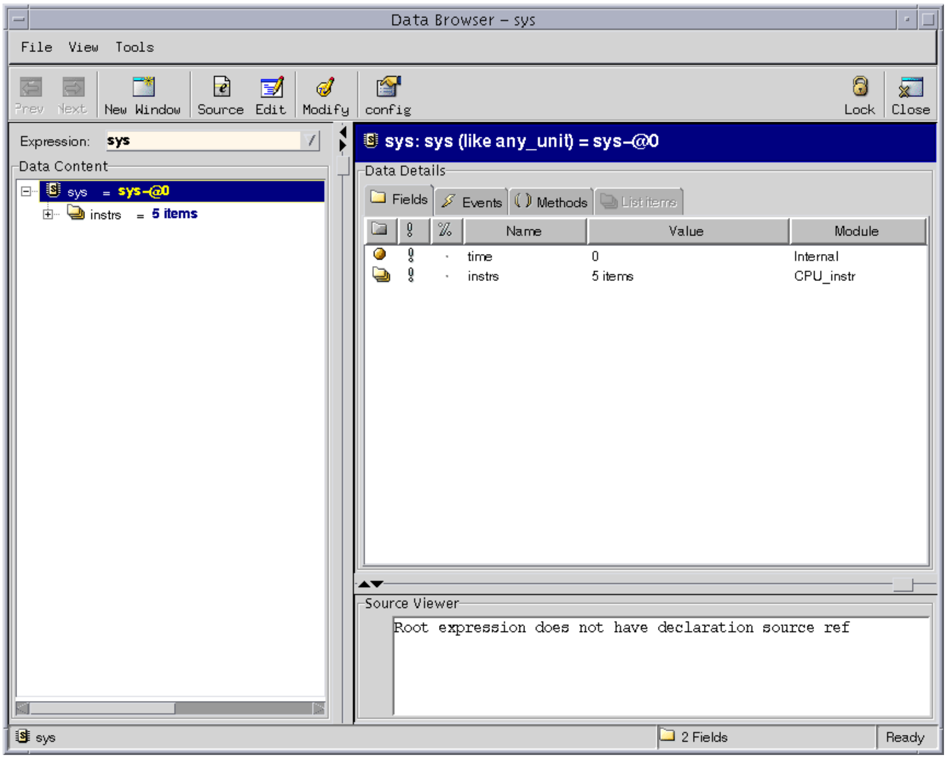
   You should see the following output in the Specview main window.


Specman Elite
File Edit View Test Debug Tools User Help
support advisor Verisity
Interrupt Load Reload Restore Modules Test Sys Coverage Source Config Threads
Specman >
CPU_tst1 Normal Ready

```
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 ...

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.
Wrote 1 cover_struct to CPU_tst1_1.ecov
```

2. In the Tools menu, choose Data Browser, and then choose Show Data Sys.

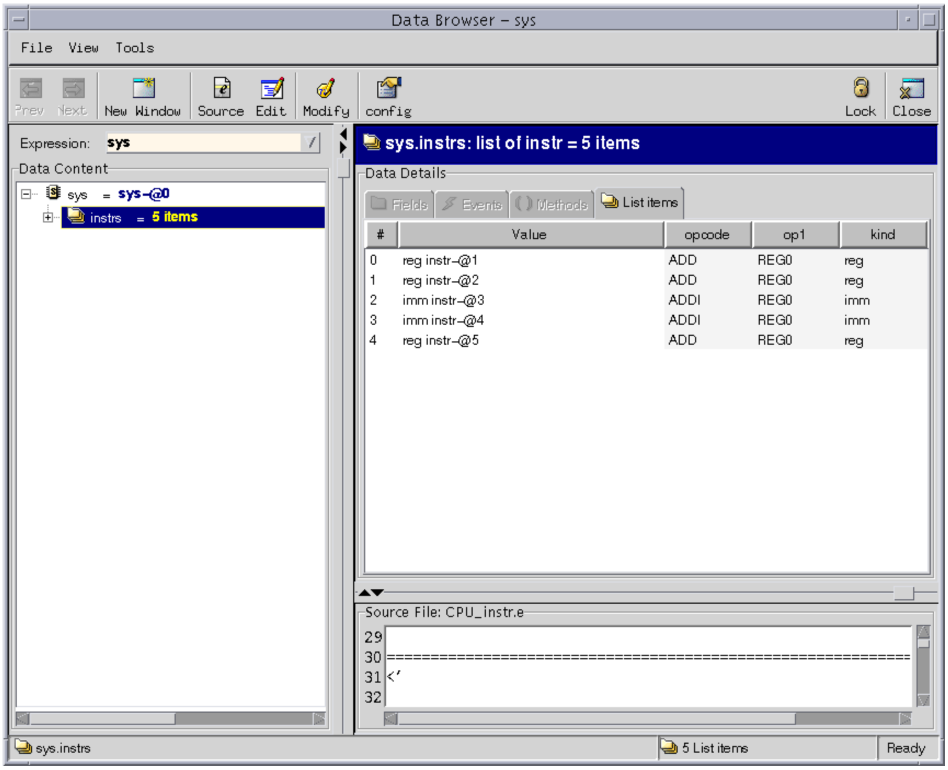
The Data Browser GUI appears.



```
Root expression does not have declaration source ref
```

3. Click the blue *instrs = 5 items* link in the left panel.

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



sys = sys-@0
instrs = 5 items

sys.instrs: list of instr = 5 items

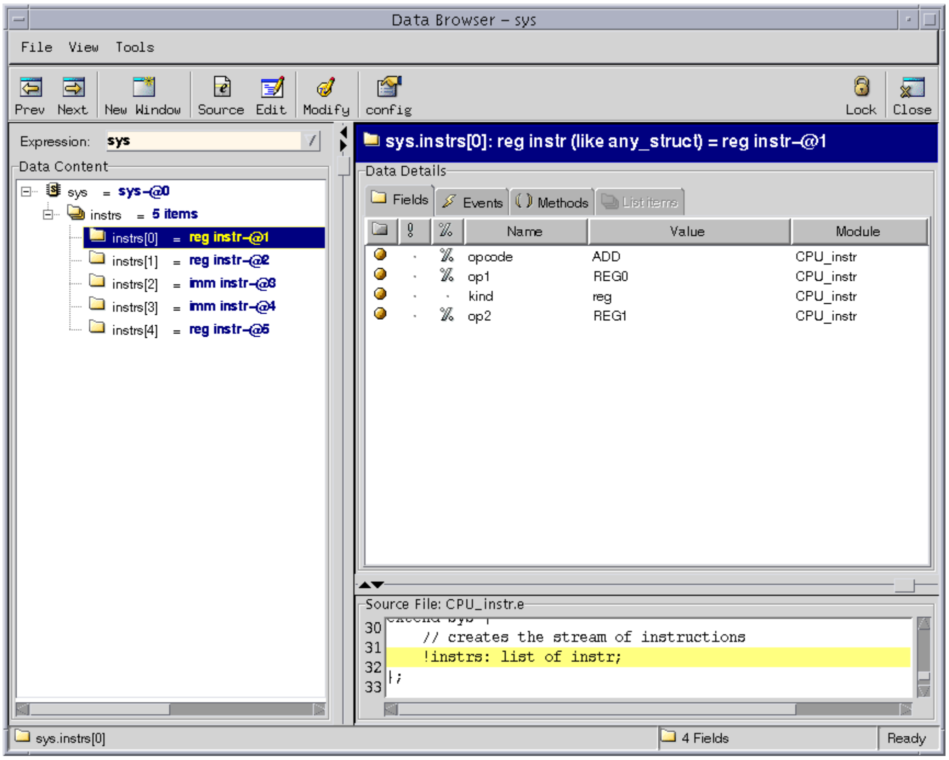
#	Value	opcode	op1	kind
0	reg instr-@1	ADD	REG0	reg
1	reg instr-@2	ADD	REG0	reg
2	imm instr-@3	ADDI	REG0	imm
3	imm instr-@4	ADDI	REG0	imm
4	reg instr-@5	ADD	REG0	reg

```

==============================================================
<'

```

sys.instrs 5 List items

4. Double-click the blue *reg instr-@1* link in the top right panel.

The generated values for the fields of the first instruction object appear in the right panel.



```
sys = sys-@0
  instrs = 5 items
    instrs[0] = reg instr-@1
    instrs[1] = reg instr-@2
    instrs[2] = imm instr-@3
    instrs[3] = imm instr-@4
    instrs[4] = reg instr-@5
```

sys.instrs[0]: reg instr (like any_struct) = reg instr-@1

Name	Value	Module
opcode	ADD	CPU_instr
op1	REG0	CPU_instr
kind	reg	CPU_instr
op2	REG1	CPU_instr

```
      // creates the stream of instructions
      !instrs: list of instr;
};
```

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 *tutorial*/*gold*/.

5. 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.

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

- **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 Elite system automatically creates the necessary PLI/FLI calls for you.

- **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.

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

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.

All 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, 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 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, also waits for the *fetch2* signal to rise and sends the second byte of data. Then 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: instr) @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);
};
```

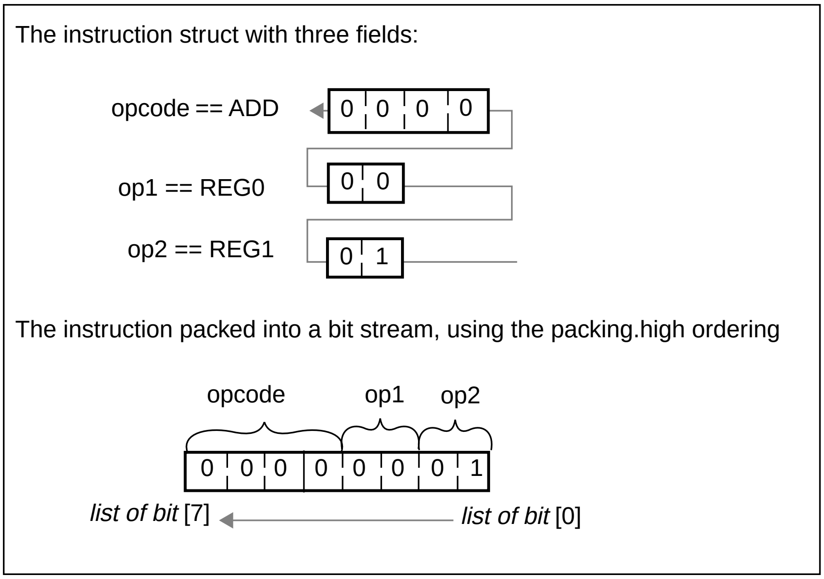
The assignment statements in Figure 5-1 show how to drive and sample signals in an HDL model. Each pair of single quotation marks identifies an object as an HDL signal.

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 Elite 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

To run the simulation:

1. Copy the *src/CPU_dut.e* to the working directory.
2. Copy the *src/CPU_drive.e* to the working directory.
3. Open the working directory's copy of the *CPU_top.e* file in an editor.

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

```
// Add dut and drive:
//import CPU_dut, CPU_drive;
```

5. 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;
```

6. Save the *CPU_top.e* file.

7. Click the Reload button to reload the files for test 1.

Tip If you have exited Specview, you must reinvoke it and load *CPU_tst1.e* again. To do so, enter the **specview** command at the UNIX prompt, click on the Load button, and choose *CPU_tst1.e*.

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 click the Reload button.

8. Click the Modules button to confirm that six modules are loaded:

```
CPU_instr        CPU_drive
CPU_misc         CPU_top
CPU_dut          CPU_tst1
```

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 choose *CPU_tst1.e* in the Select A File dialog box.

9. Click Test to run the simulation.

   You should see the following messages (or something similar) in the Specview main window.

```
Doing setup…
Generating the test using seed 0x1…
Starting the test…
Running the test…
DUT executing instr 0 :    ADD    REG0x0, REG0x1
DUT executing instr 1 :    ADD    REG0x0, REG0x1
DUT executing instr 2 :    ADDI   REG0x0, @0x05
DUT executing instr 3 :    ADDI   REG0x0, @0x05
DUT executing instr 4 :    ADD    REG0x0, 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_tst1_1.ecov
```

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 large 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 Elite 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 Elite system to quickly generate a whole new set of tests.

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

Table 6-1 New Constructs and 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.
Command	**How the Command is Used**
Tools>>Config	Used to access the Generation tab of the Specman Elite Configuration Options window for creating a user-defined seed for random test generation.
File>>Save	Saves the current test environment, including the random seed, to a .esv file. You can load this file with the File>>Restore command.
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 Elite 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

```
<'

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

'>
```

Generating Tests With a User-Specified Seed

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

Procedure

This procedure shows how to create a random seed:

1. In the Specview main window, click Restore to remove all the ***e*** modules from the current session.
2. Click Load. Then double-click the *CPU_tst2.e* file.

   The Specman Elite system loads the *CPU_tst2.e* file along with its imported modules.
3. Click Modules and confirm that the following modules are loaded:

```
CPU_instr    CPU_drive
CPU_misc     CPU_top
CPU_dut      CPU_tst2
```

4. Click the Config button or choose Config from the Tools menu.

   The Specman Elite Configuration Options window opens.

5. Choose the Generation tab and then enter a number of your choice in the text box under Seed.

6. Click OK to save the settings and close the window.

7. Click the Test button on the Specview main window.

   The Specman Elite system runs the test with the your seed and reports the results.

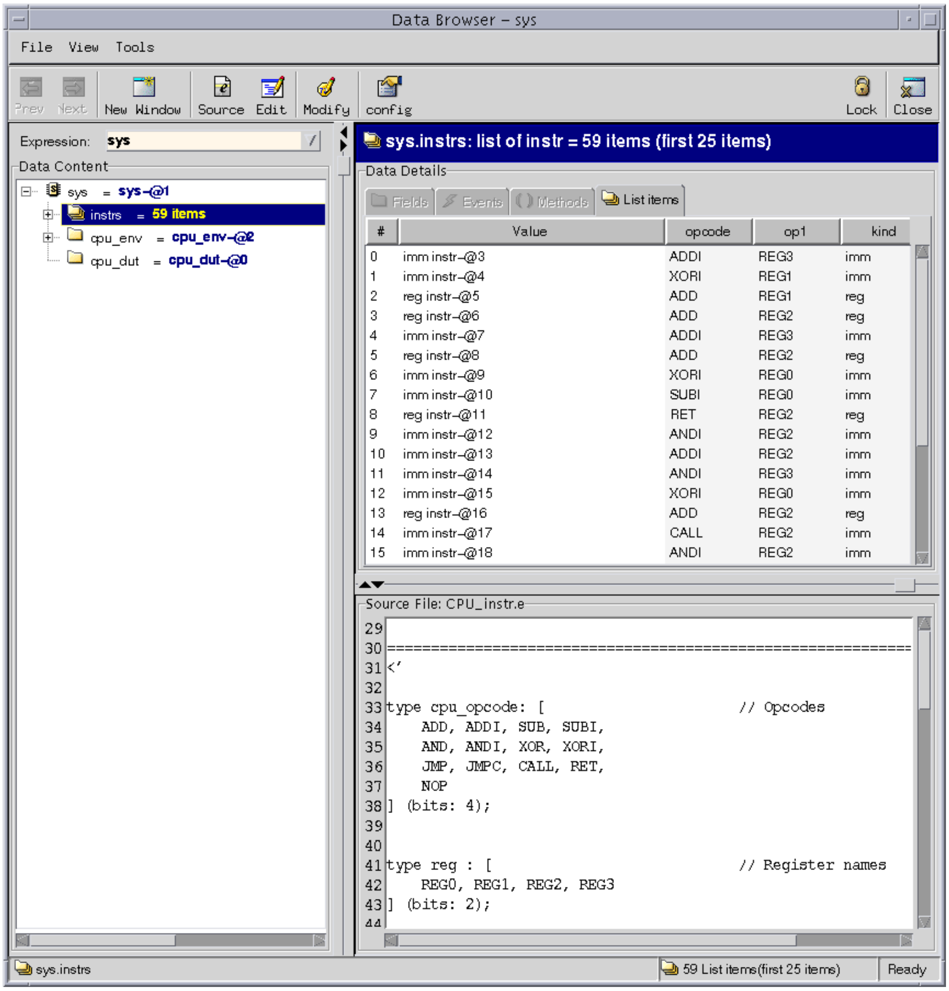
8. In the Tools menu, choose Data Browser and then choose Show Data Sys.

   The Data Browser GUI appears.

9. Click the blue *instrs = 59 items* link in the left panel.

   Instructions are listed in the top right panel. By default, only the first 25 instructions are listed. You can click the Config button in the Data Browser, and then change the number of list items to 59 to list all of the instructions.

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.



#	Value	opcode	op1	kind
0	imm instr-@3	ADDI	REG3	imm
1	imm instr-@4	XORI	REG1	imm
2	reg instr-@5	ADD	REG1	reg
3	reg instr-@6	ADD	REG2	reg
4	imm instr-@7	ADDI	REG3	imm
5	reg instr-@8	ADD	REG2	reg
6	imm instr-@9	XORI	REG0	imm
7	imm instr-@10	SUBI	REG0	imm
8	reg instr-@11	RET	REG2	reg
9	imm instr-@12	ANDI	REG2	imm
10	imm instr-@13	ADDI	REG2	imm
11	imm instr-@14	ANDI	REG3	imm
12	imm instr-@15	XORI	REG0	imm
13	reg instr-@16	ADD	REG2	reg
14	imm instr-@17	CALL	REG2	imm
15	imm instr-@18	ANDI	REG2	imm

```

===============================================================
<'

type cpu_opcode: [                         // Opcodes
    ADD, ADDI, SUB, SUBI,
    AND, ANDI, XOR, XORI,
    JMP, JMPC, CALL, RET,
    NOP
]  (bits: 4);


type reg : [                               // Register names
    REG0, REG1, REG2, REG3
]  (bits: 2);
```

Generating Tests With a Random Seed

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

Procedure

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

1. In the Specview main window, click the Reload button.
2. Choose Test>>Test With Random Seed.

   The Specman Elite system runs the test with the random seed shown in the Specview main 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 Elite'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 Elite verification system's enabling features—the **Functional Coverage Analyzer**. The Specman Elite 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.
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 {

    event cpu_fsm is @sys.cpuclk;
    .
    .
    .
};
```

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*), 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 Elite system collects the current and previous values of *top.cpu.curr_FSM*.

defines the coverage group cpu_fsm

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

    // DUT Coverage: State Machine and State
    // Machine transition coverage
    cover cpu_fsm is {
      item fsm: FSM_type = '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 7-2 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: instr) @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 *instr* 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 instr {

    cover start_drv_DUT is {

    };

};
```

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

```
extend instr {

    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 instr {

    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 instr {

    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 and use that information to 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.

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

- **Cross Coverage—**This lets you view the intersections of two or more coverage items.
- **Help—**This helps you find the information you need in the Specman Elite Online Documentation.
- **Coverage Extensibility—**This allows you to change coverage group and coverage item definitions.

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

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

Command	How the Command is Used
Tools>>Coverage	Displays coverage reports and creates cross-coverage reports.
Help>>Help Browser	Invokes the Specman Elite Online Documentation browser.

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. Click Reload to reload the files for test 2.

Tip If you have exited Specview, you must reinvoke it and load *CPU_tst2.e* again. To do so, enter the **specview** command at the UNIX prompt, click on the Load button, and choose *CPU_tst2.e*.

5. Click Modules to confirm that seven modules are loaded:

```
CPU_instr
CPU_misc
CPU_dut
CPU_drive
CPU_cover
CPU_top
CPU_tst2
```

6. Click Test.

   You should see something similar to the following in the Specview main 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 0x1
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
```

Viewing State Machine Coverage

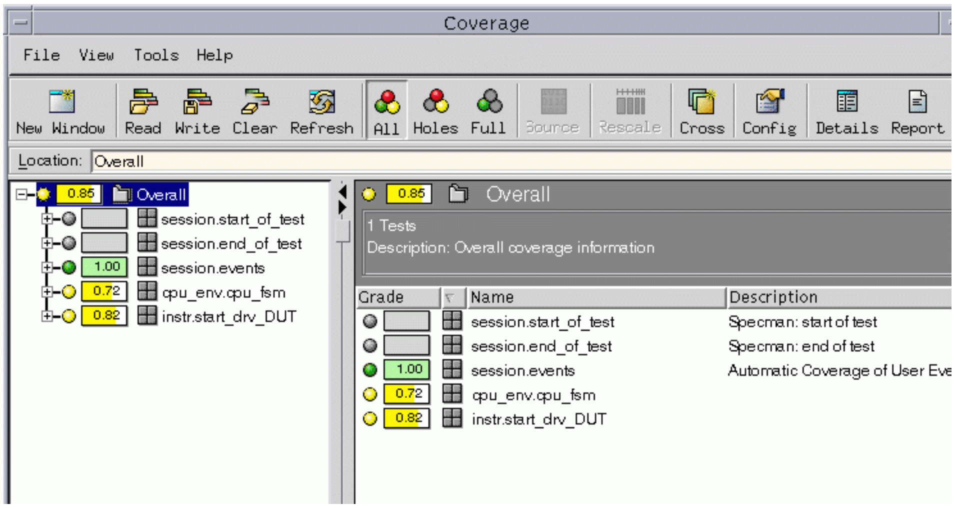
You have two reports to look at, the state machine register report and the state machine transition report.

If you are using a different seed or a version of the Specman Elite verification system other than 4.0, you may see different results in your coverage reports.

Procedure

1. Click the Coverage button in the Specview main window.

   The Coverage window appears.



Location: Overall

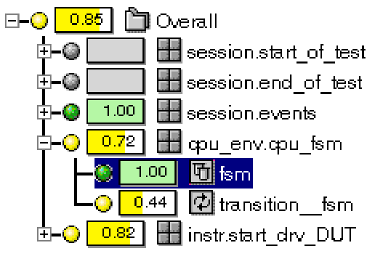
Grade		Name
0.85		Overall
		session.start_of_test
		session.end_of_test
1.00		session.events
0.72		cpu_env.cpu_fsm
0.82		instr.start_drv_DUT

0.85 Overall

1 Tests
Description: Overall coverage information

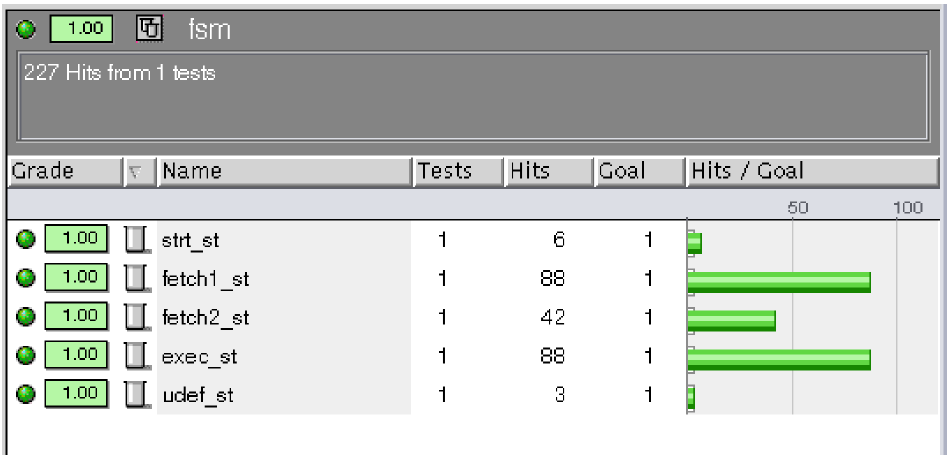
Grade		Name	Description
		session.start_of_test	Specman: start of test
		session.end_of_test	Specman: end of test
1.00		session.events	Automatic Coverage of User Eve
0.72		cpu_env.cpu_fsm	
0.82		instr.start_drv_DUT	

2. In the Group frame on the left, click the + to the left of *cpu_env.cpu_fsm* and then choose *fsm*.



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

From the report it is easy to see that, for example, the *fetch1* state was entered 88 times in the 227 times sampled.



1.00 fsm

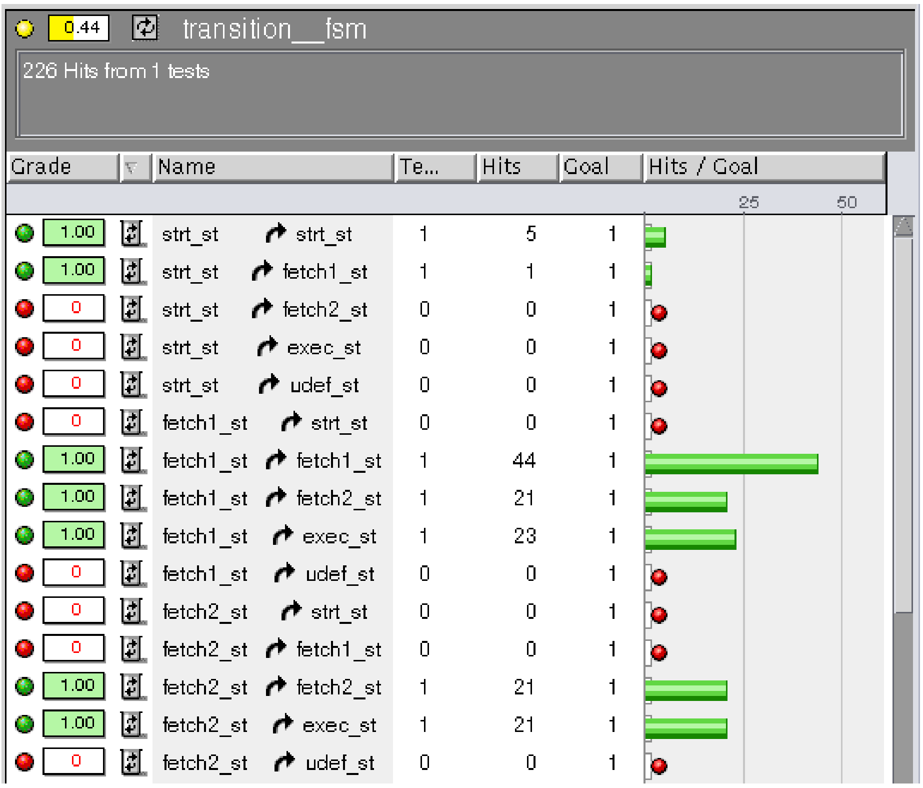
227 Hits from 1 tests

Grade	Name	Tests	Hits	Goal	Hits / Goal
1.00	strt_st	1	6	1	
1.00	fetch1_st	1	88	1	
1.00	fetch2_st	1	42	1	
1.00	exec_st	1	88	1	
1.00	udef_st	1	3	1	

3. In the Group frame on the left, choose *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.

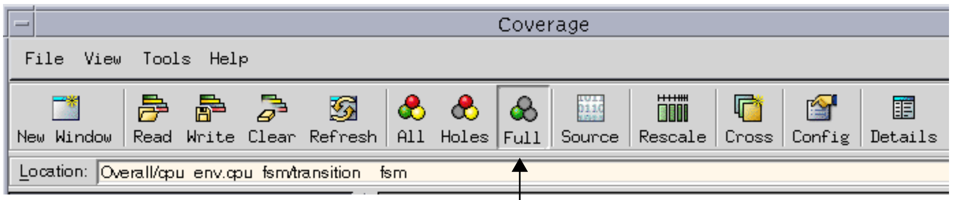


0.44 transition__fsm

226 Hits from 1 tests

Grade	Name	Te...	Hits	Goal
1.00	strt_st ↱ strt_st	1	5	1
1.00	strt_st ↱ fetch1_st	1	1	1
0	strt_st ↱ fetch2_st	0	0	1
0	strt_st ↱ exec_st	0	0	1
0	strt_st ↱ udef_st	0	0	1
0	fetch1_st ↱ strt_st	0	0	1
1.00	fetch1_st ↱ fetch1_st	1	44	1
1.00	fetch1_st ↱ fetch2_st	1	21	1
1.00	fetch1_st ↱ exec_st	1	23	1
0	fetch1_st ↱ udef_st	0	0	1
0	fetch2_st ↱ strt_st	0	0	1
0	fetch2_st ↱ fetch1_st	0	0	1
1.00	fetch2_st ↱ fetch2_st	1	21	1
1.00	fetch2_st ↱ exec_st	1	21	1
0	fetch2_st ↱ udef_st	0	0	1

You can change the display to show data for transitions that have occurred only by clicking the Full button at the top of the coverage window. (The All button shows data for all transitions, and the Holes button shows data for transitions that have not occurred.)

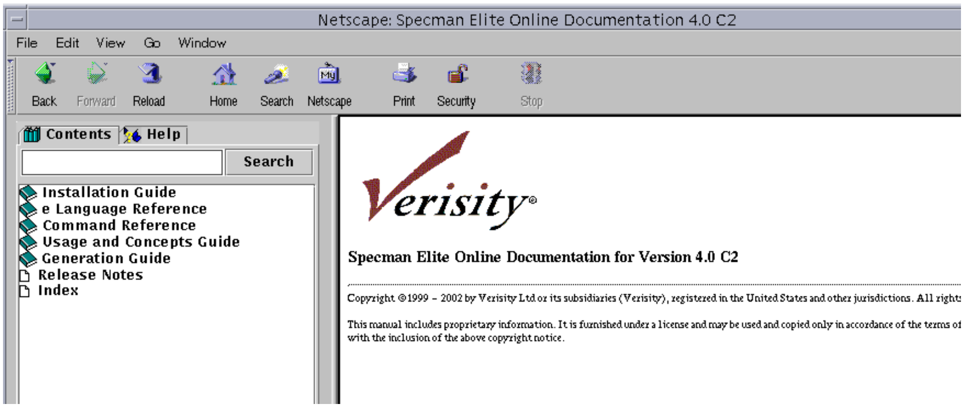


Full button

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

4. To see how to define transitions as illegal so that they do not appear in the coverage report, choose Help Browser from the Help menu in the Specview main window (or you can click the large Verisity button).

   The Specman Elite Online Documentation browser appears.



Specman Elite Online Documentation for Version 4.0 C2

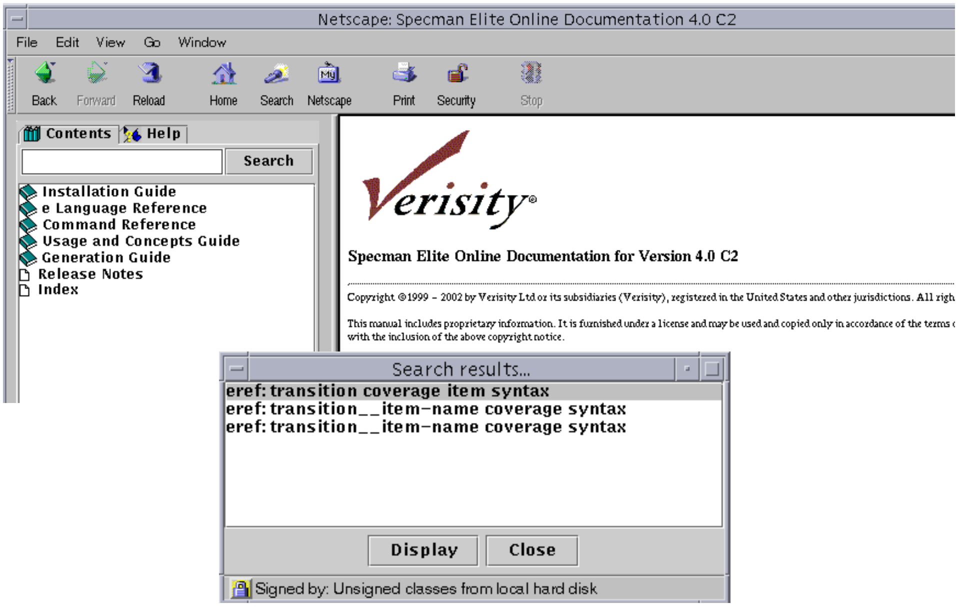
Copyright ©1999 – 2002 by Verisity Ltd or its subsidiaries (Verisity), registered in the United States and other jurisdictions. All rights

This manual includes proprietary information. It is furnished under a license and may be used and copied only in accordance of the terms of with the inclusion of the above copyright notice.

5. Enter the words *transition cover item syntax* in the Search field and press Return.

   The **transition** construct is a coverage item, so this search will find the description of the correct syntax for this construct.

6. When the list of topics that describe coverage item options appears, choose the first item in the list, *eref: transition cover item syntax*.



Specman Elite Online Documentation for Version 4.0 C2

Copyright ©1999 – 2002 by Verisity Ltd or its subsidiaries (Verisity), registered in the United States and other jurisdictions. All righ

This manual includes proprietary information. It is furnished under a license and may be used and copied only in accordance of the terms o with the inclusion of the above copyright notice.

eref: transition coverage item syntax
eref: transition__item-name coverage syntax
eref: transition__item-name coverage syntax

Signed by: Unsigned classes from local hard disk

The tag *eref* indicates that this document is part of the ***e** Language Reference Manual*.

7. When the **transition** construct description appears, scroll down the page to the **illegal** coverage item option description.

8. Continue scrolling down to the Examples section, and you will find an example showing the use of the **illegal** option:

```
cover state_change is {
    item st;
    transition st using illegal =
        not ((prev_st == START and st == FETCH1)
            or (prev_st == FETCH1 and st == FETCH2)
            or (prev_st == FETCH1 and st == EXEC)
            or (prev_st == FETCH2 and st == EXEC)
            or (prev_st == EXEC and st == START));
};
```

If you like, you can follow this example to enhance the **transition** statement in *CPU_cover.e* to ignore the illegal transitions.

Viewing Instruction Stream Coverage

We 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.

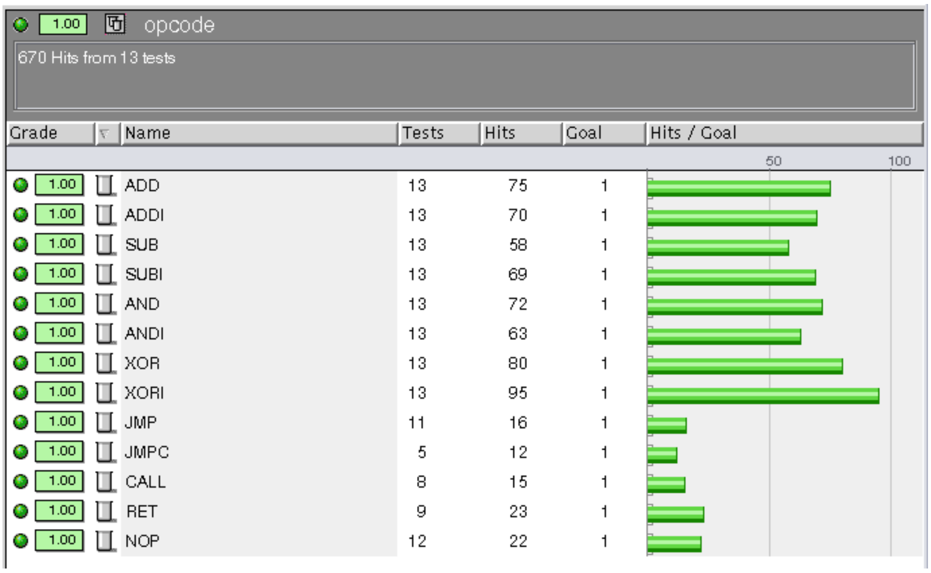
Procedure

To view instruction stream coverage:

1. Copy *src/regression_4.0.ecov* file to the working directory.
2. Click Coverage in the Specview main window.

   The Coverage window appears.
3. In the File menu in the Coverage window, choose Clear Data to remove the coverage data from the previous test.
4. In the Coverage window, click Read to open the Read Files dialog box. Select *regression_4.0.ecov*, then click Read in the Read Files dialog box to read in the file.
5. In the Group frame on the left, click the **+** to the left of *instr.start_drv_DUT* and then choose *opcode*.

The opcode coverage report appears in the right-hand frames. These results show that the current set of tests fulfill the requirement in the functional test plan to focus on arithmetic and logic operations rather than control flow operations.

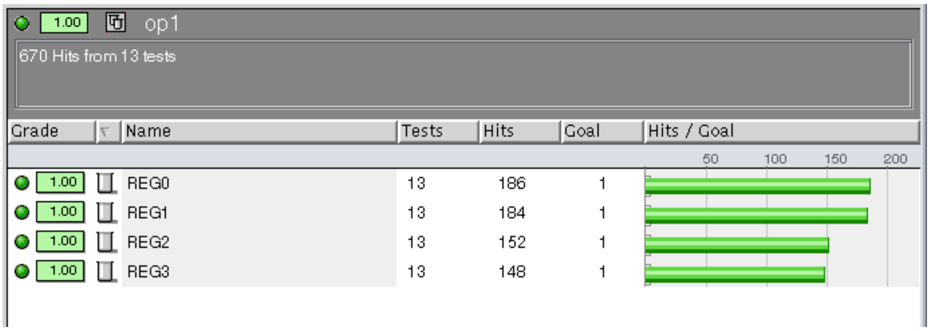


670 Hits from 13 tests

Grade	Name	Tests	Hits	Goal	Hits / Goal
1.00	ADD	13	75	1	
1.00	ADDI	13	70	1	
1.00	SUB	13	58	1	
1.00	SUBI	13	69	1	
1.00	AND	13	72	1	
1.00	ANDI	13	63	1	
1.00	XOR	13	80	1	
1.00	XORI	13	95	1	
1.00	JMP	11	16	1	
1.00	JMPC	5	12	1	
1.00	CALL	8	15	1	
1.00	RET	9	23	1	
1.00	NOP	12	22	1	

6. In the Group frame on the left, choose *op1*.

   The *op1* coverage report appears in the right-hand frames. All possible *op1* values appear to be well covered.



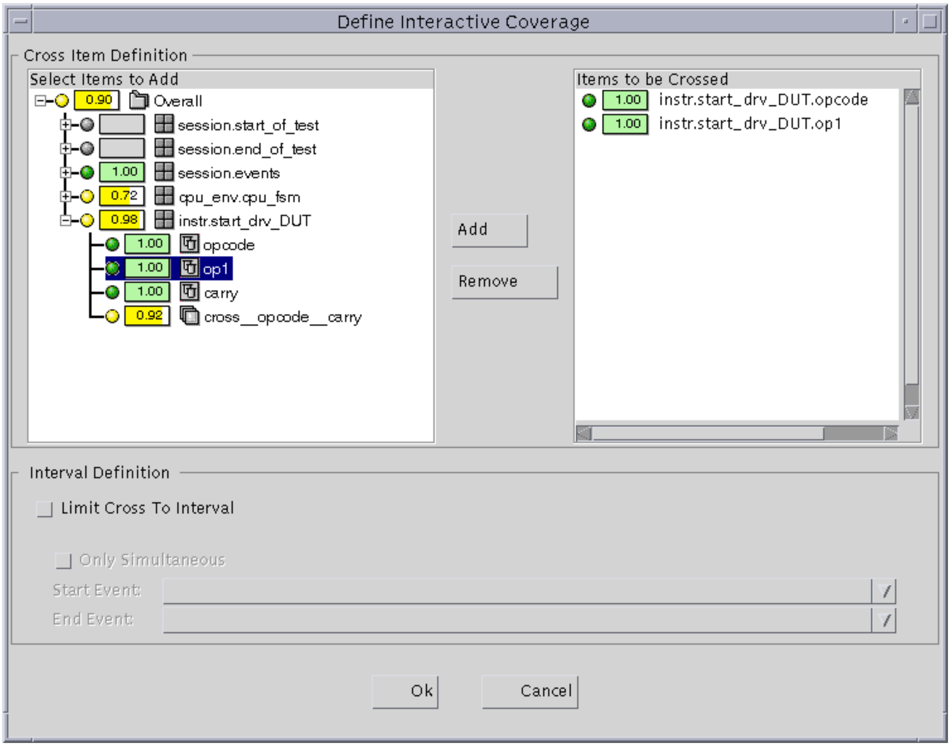
670 Hits from 13 tests

Grade	Name	Tests	Hits	Goal	Hits / Goal
1.00	REG0	13	186	1	
1.00	REG1	13	184	1	
1.00	REG2	13	152	1	
1.00	REG3	13	148	1	

7. On the Coverage window toolbar, click Cross.

8. When the Define Interactive Coverage dialog box appears, click on the + next to *instr.start_drv_DUT* to expand it.

9. Under *Select Items to Add*, choose *opcode* and click the Add button. Then choose *op1* and click Add again.

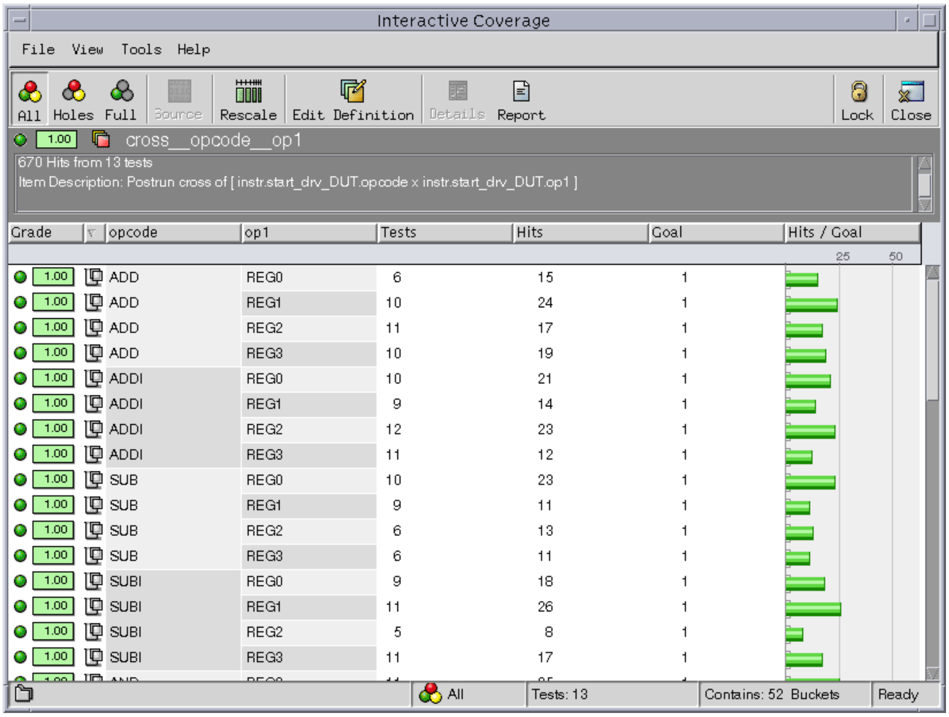


```
0.90 Overall
     session.start_of_test
     session.end_of_test
1.00 session.events
0.72 cpu_env.cpu_fsm
0.98 instr.start_drv_DUT
  1.00 opcode
  1.00 op1
  1.00 carry
  0.92 cross__opcode__carry
```

```
1.00 instr.start_drv_DUT.opcode
1.00 instr.start_drv_DUT.op1
```

10. Click OK to display a coverage report of the new *cross_opcode_op1* item.

    This coverage report shows whether the tests have covered every possible combination of opcode and register.


Interactive Coverage
File View Tools Help
All Holes Full Source Rescale Edit Definition Details Report Lock Close
1.00 cross__opcode__op1

670 Hits from 13 tests
Item Description: Postrun cross of [instr.start_drv_DUT.opcode x instr.start_drv_DUT.op1]

Grade	opcode	op1	Tests	Hits	Goal	Hits / Goal
1.00	ADD	REG0	6	15	1	
1.00	ADD	REG1	10	24	1	
1.00	ADD	REG2	11	17	1	
1.00	ADD	REG3	10	19	1	
1.00	ADDI	REG0	10	21	1	
1.00	ADDI	REG1	9	14	1	
1.00	ADDI	REG2	12	23	1	
1.00	ADDI	REG3	11	12	1	
1.00	SUB	REG0	10	23	1	
1.00	SUB	REG1	9	11	1	
1.00	SUB	REG2	6	13	1	
1.00	SUB	REG3	6	11	1	
1.00	SUBI	REG0	9	18	1	
1.00	SUBI	REG1	11	26	1	
1.00	SUBI	REG2	5	8	1	
1.00	SUBI	REG3	11	17	1	

Extending Coverage

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 instr {

    // 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, but **is also** instead of just **is**.

```
extend instr {

    // 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 instr struct.

```
extend instr {

    // 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 instr {

    // 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. Click the Reload button to reload the files for test 2.

Tip If you have exited Specview, you must reinvoke it and load *CPU_tst2.e* again. To do so, enter the **specview** command at the UNIX prompt, click on the Load button, and choose *CPU_tst2.e*.

12. Click Modules to confirm that eight modules are loaded:

```
CPU_instr
CPU_misc
CPU_dut
CPU_drive
CPU_cover
CPU_cover_extend
CPU_top
CPU_tst2
```

13. Click Test.

    You should see something similar to the following in the Specview main 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 0x1
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 *instr* 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 8-9, 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_4.0.ecov*. In the following procedure, we load the *regression_4.0.ecov* file into the Coverage GUI and view the merged coverage data.

Procedure

To view coverage by op1 subtype of the instr instances:

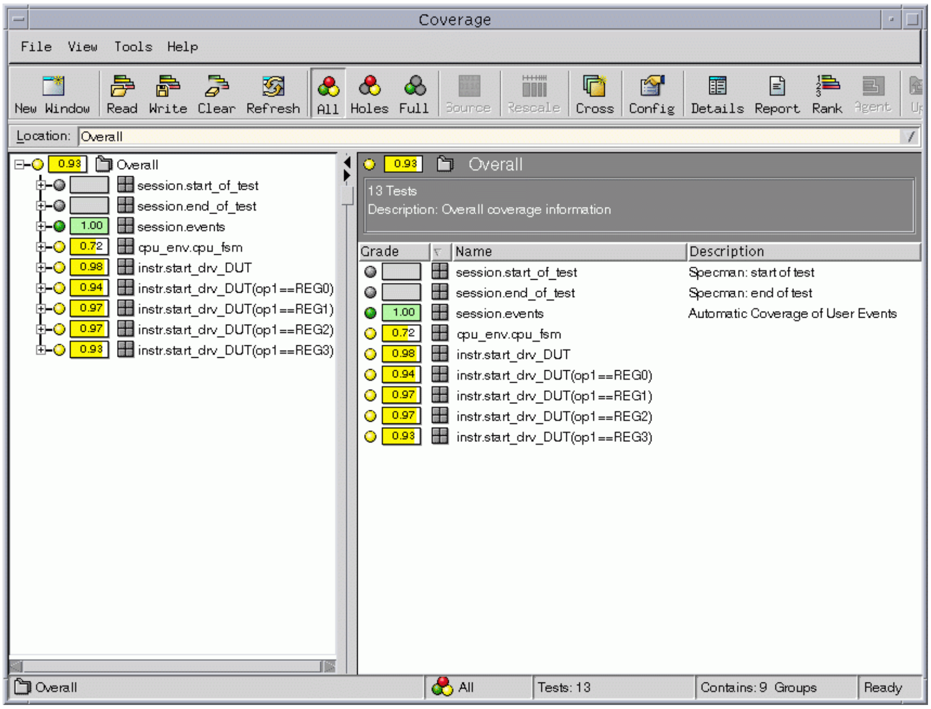
1. Click Coverage in the Specview main window.

   The Coverage window appears.

2. In the File menu in the Coverage window, choose Clear Data to remove the coverage data from the previous test.

3. In the Coverage window, click Read to open the Read Files dialog box. Select *regression_4.0.ecov*, then click OK in the Read Files dialog box to read in the file.

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


Coverage
File View Tools Help
New Window Read Write Clear Refresh All Holes Full Source Rescale Cross Config Details Report Rank Agent
Location: Overall
Overall
All
Tests: 13
Contains: 9 Groups
Ready

0.93 Overall
13 Tests
Description: Overall coverage information

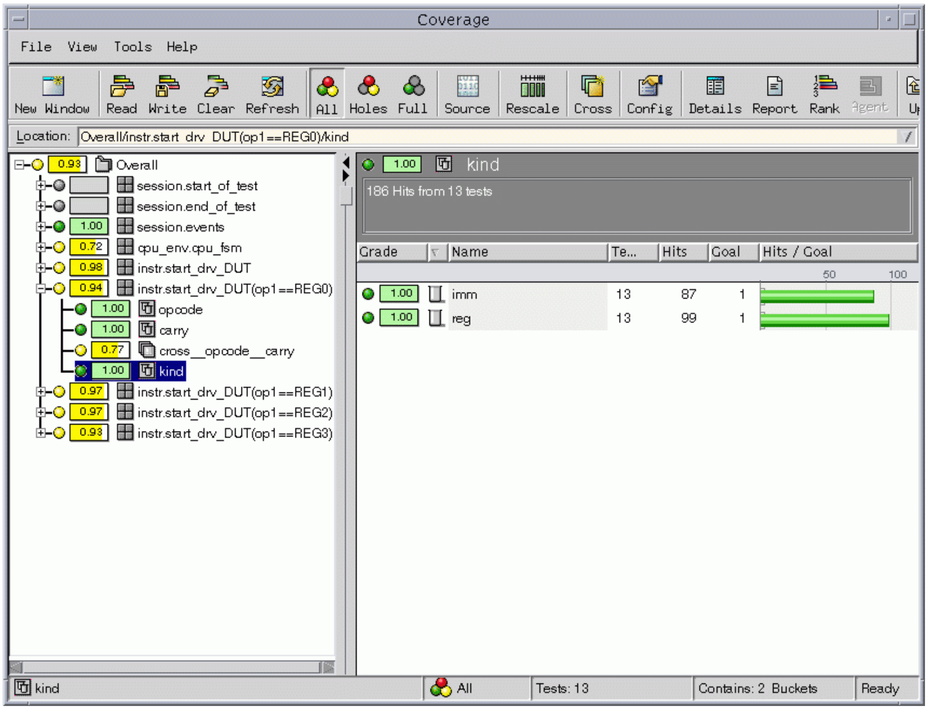
Grade	Name	Description
	session.start_of_test	Specman: start of test
	session.end_of_test	Specman: end of test
1.00	session.events	Automatic Coverage of User Events
0.72	cpu_env.cpu_fsm	
0.98	instr.start_drv_DUT	
0.94	instr.start_drv_DUT(op1==REG0)	
0.97	instr.start_drv_DUT(op1==REG1)	
0.97	instr.start_drv_DUT(op1==REG2)	
0.93	instr.start_drv_DUT(op1==REG3)	

4. In the Group frame on the left, click the + to the left of *instr.start_drv_DUT(op1==REG0)*.

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

5. Choose *kind*.

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



Location: Overall/instr.start drv DUT(op1==REG0)/kind

0.93	Overall
	session.start_of_test
	session.end_of_test
1.00	session.events
0.72	cpu_env.cpu_fsm
0.98	instr.start_drv_DUT
0.94	instr.start_drv_DUT(op1==REG0)
1.00	opcode
1.00	carry
0.77	cross__opcode__carry
1.00	kind
0.97	instr.start_drv_DUT(op1==REG1)
0.97	instr.start_drv_DUT(op1==REG2)
0.93	instr.start_drv_DUT(op1==REG3)

1.00 kind

186 Hits from 13 tests

Grade	Name	Te...	Hits	Goal	Hits / Goal
1.00	imm	13	87	1	
1.00	reg	13	99	1	

kind | All | Tests: 13 | Contains: 2 Buckets | Ready

6. In the Group frame on the left, click the **+** to the left of *instr.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.

7. Choose each *cross__opcode__carry* item in turn to see which crosses of opcode and carry never occurred at all (shown under *instr.start_drv_DUT*), and which additional crosses never occurred under each particular subtype.

8. In the Coverage window, click Close to close the window.

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, as shown by the results above and by the following procedure.

Viewing Corner Case Coverage

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

Procedure

To view corner case coverage of the JMPC opcode:

1. Click Coverage in the Specview main window.

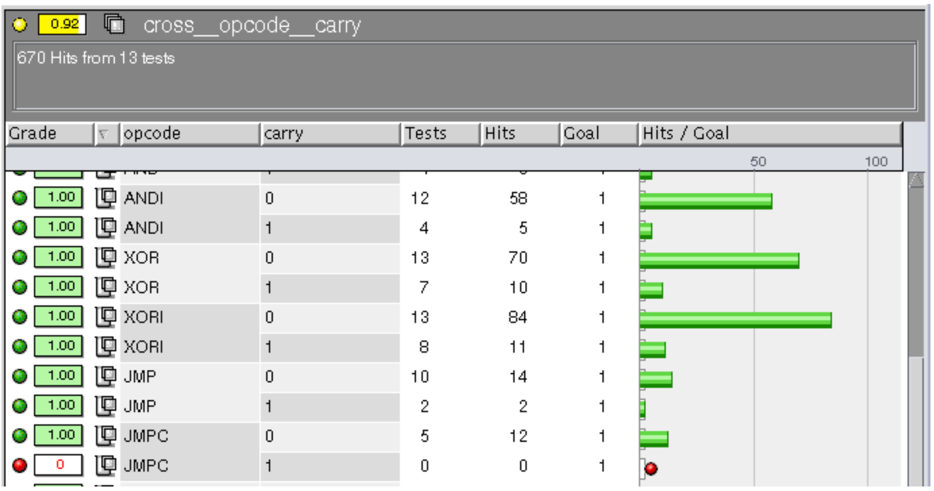
   The Coverage window appears.

2. In the Group frame on the left, click the + to the left of *instr.start_drv_DUT* and then choose *cross__opcode__carry.*

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

3. Scroll down to the JMPC opcode.

   You can see that the JMPC code was issued 12 times, and that carry was low each time.



0.92 cross__opcode__carry

670 Hits from 13 tests

Grade	opcode	carry	Tests	Hits	Goal	Hits / Goal
1.00	ANDI	0	12	58	1	
1.00	ANDI	1	4	5	1	
1.00	XOR	0	13	70	1	
1.00	XOR	1	7	10	1	
1.00	XORI	0	13	84	1	
1.00	XORI	1	8	11	1	
1.00	JMP	0	10	14	1	
1.00	JMP	1	2	2	1	
1.00	JMPC	0	5	12	1	
0	JMPC	1	0	0	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 Elite system lets you attack this type of corner case scenario much more efficiently. In the next chapter you learn how to do this.

9 Writing a Corner Case Test

Goals for this Chapter

As described in the Functional Test Plan, you want to create one 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 Elite'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. 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 instr {
    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 instr {
    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 instr {
    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 instr {
    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 Elite system.

10 Creating Temporal and Data Checks

Goals for this Chapter

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

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

- **Specman Elite 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 Elite data checking—**Data checking methodology can be flexibly implemented in the Specman Elite 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 *execute* 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:

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

                    event fetch1_assert is
 defines rise of        (change('top.fetch1')and
          fetch1        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;

                    //Interface Spec: After entering instruction
                    //execution state, fetch1 signal must be
                    //asserted in the following cycle.
 issues an error    expect @enter_exec_st => {@fetch1_assert}
 message if fetch1      @sys.cpuclk else
    does not rise       dut_error("PROTOCOL ERROR");
exactly one cycle
   after entering
    execute state
```

---

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 Elite 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 Elite 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.

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

             method  on_exec_done() is {
    associated with      // Compare PC - program counter
              event  };
                     .
                     .
                     .
```

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 the working directory.
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;
    cpu_dut : cpu_dut;
    //cpu_refmodel : cpu_refmodel;
};
'>
```

3. Find the line that extends the Specman Elite 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;
    cpu_dut : cpu_dut;
    cpu_refmodel : cpu_refmodel;
};
'>
```

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 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 Specview, if it is not already running:

```
% specview &
```

7. Click Restore to remove any loaded modules from the current session.
8. Click Load and load *CPU_tst3.e*.

   Remember that this is the test that you wrote in Chapter 9, “Writing a Corner Case Test”.
9. Click Test to run the simulation.

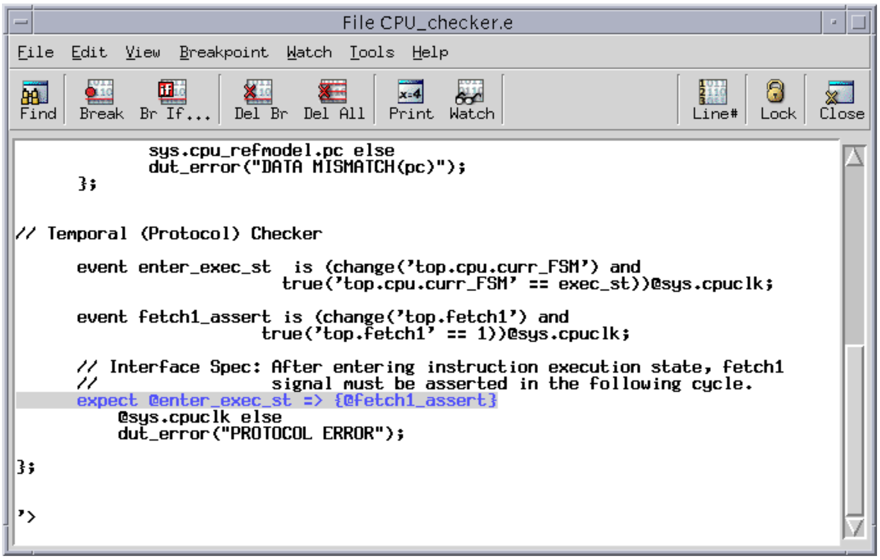
It looks like we hit a bug here. The Specman Elite system is reporting a protocol violation.

```
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 2246
        Checked at line 42 in @CPU_checker
        In cpu_env-@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 Elite that is not 4.0, 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 error hyperlink to view the line in the source that generated this message.

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


File CPU_checker.e
File Edit View Breakpoint Watch Tools Help
Find Break Br If... Del Br Del All Print Watch Line# Lock Close

```
                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");

};

'>
```

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 Elite system’s enabling features:

- **The Specman Elite debugger—**Provides powerful debugging capability with visibility into the HDL design.
- **The Specman Elite bypass feature—**Lets you temporarily prevent the Specman Elite 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 Specview menu commands shown in Table 11-1.

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

Command	How the Command is Used
Debug>>Thread Browser	Opens the Thread Browser, which displays all the TCMs (threads) that are currently active.
Debug>>Open Debug Window	Opens the Debugger window, which displays the source for the current thread with the current line highlighted.
Debugger: View>>Print	Displays the current value of an ***e*** variable.
Debugger: Breakpoint>> Set Breakpoint>>Break	Sets a breakpoint on the currently highlighted line of ***e*** code.
Debugger: Run>>Step Any	Advances simulation to the next line of ***e*** code executed in any thread.
Debugger: 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

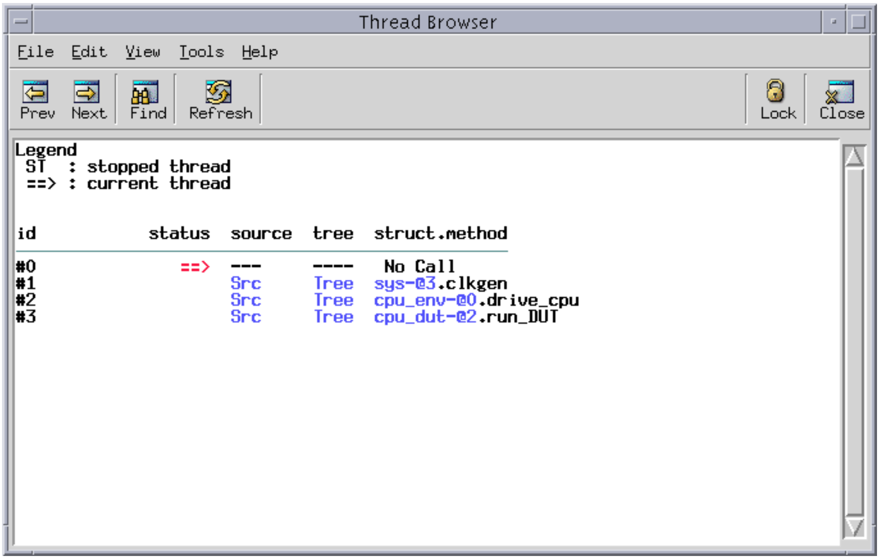
If you have just completed Chapter 10, “Creating Temporal and Data Checks”, the PROTOCOL ERROR message is still displayed on the Main Specman window. If you exited Specview, you will have to reinvoke Specview and run the simulation again, as described in “Running the Simulation” on page 10-7. Then continue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

Procedure

To display DUT values:

1. In the Specview main window, click the Threads button or choose Thread Browser from the Debug menu.

   The Thread Browser appears.



```
Legend
 ST  : stopped thread
 ==> : current thread

id              status  source  tree   struct.method
#0                ==>    ---    ----    No Call
#1                       Src    Tree   sys-@3.clkgen
#2                       Src    Tree   cpu_env-@0.drive_cpu
#3                       Src    Tree   cpu_dut-@2.run_DUT
```

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

   - Clock generation
   - Drive and Sample CPU
   - DUT

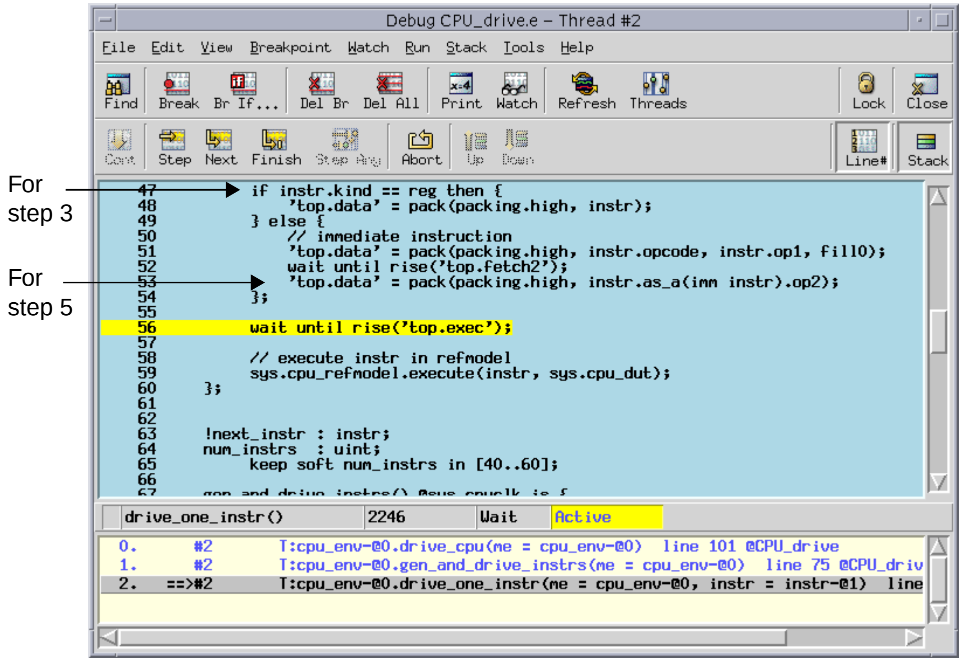
   To debug the error, look first at the TCM that drives the DUT.

2. On the line for *cpu_env-@0.drive_cpu*, click on *Src* to bring up the corresponding source file for this thread.

   The Debugger window appears, showing *CPU_drive.e*.

3. Click the Line # button in the Debugger window to display line numbers.

   The highlighted line (line 56) shows that the *drive_one_instr* TCM is waiting for the *top.exec* signal to rise.


For step 3
For step 5
Debug CPU_drive.e – Thread #2
File Edit View Breakpoint Watch Run Stack Tools Help
Find Break Br If... Del Br Del All Print Watch Refresh Threads Lock Close
Cont Step Next Finish Step Any Abort Up Down Line# Stack

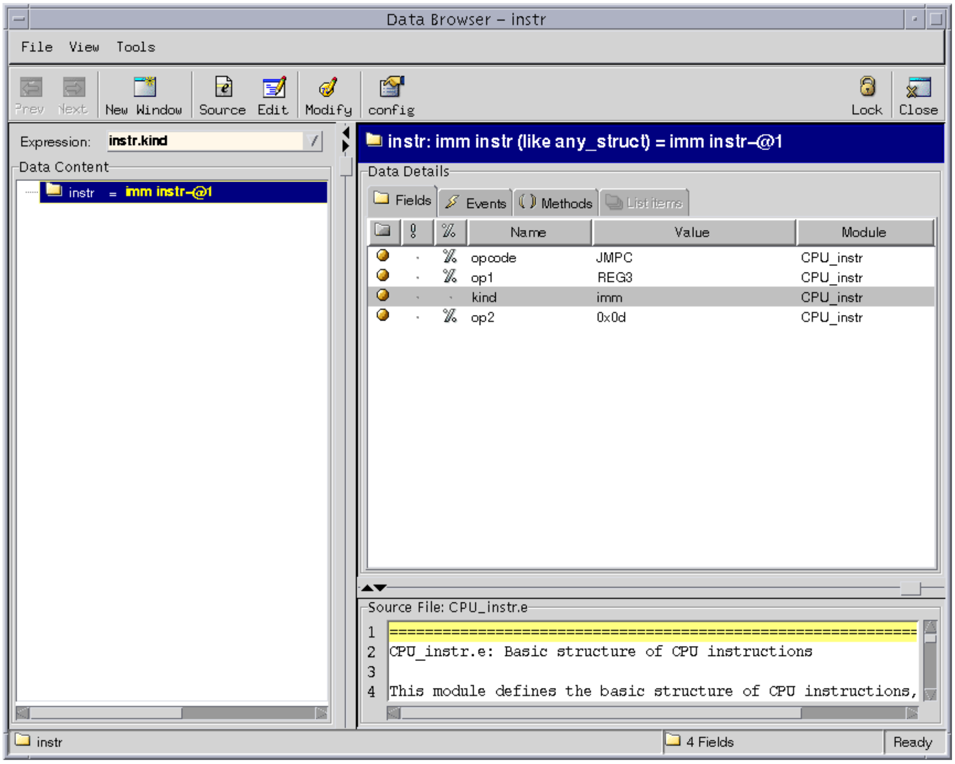
```
47          if instr.kind == reg then {
48              'top.data' = pack(packing.high, instr);
49          } else {
50              // immediate instruction
51              'top.data' = pack(packing.high, instr.opcode, instr.op1, fill0);
52              wait until rise('top.fetch2');
53              'top.data' = pack(packing.high, instr.as_a(imm instr).op2);
54          };
55
56          wait until rise('top.exec');
57
58          // execute instr in refmodel
59          sys.cpu_refmodel.execute(instr, sys.cpu_dut);
60      };
61
62
63      !next_instr : instr;
64      num_instrs  : uint;
65          keep soft num_instrs in [40..60];
66
```

```
drive_one_instr()    2246    Wait    Active
```

```
0.      #2      T:cpu_env-@0.drive_cpu(me = cpu_env-@0)   line 101 @CPU_drive
1.      #2      T:cpu_env-@0.gen_and_drive_instrs(me = cpu_env-@0)   line 75 @CPU_driv
2.   ==>#2      T:cpu_env-@0.drive_one_instr(me = cpu_env-@0, instr = instr-@1)   line
```

4. To find out the current instruction type, highlight the phrase *instr.kind*, located 9 lines above (line 47) the highlighted **wait** statement and click Print.

The Data Browser opens and shows that the *instr.kind* is an immediate instruction.



			Name	Value	Module
●	·	%	opcode	JMPC	CPU_instr
●	·	%	op1	REG3	CPU_instr
●	·	·	kind	imm	CPU_instr
●	·	%	op2	0x0d	CPU_instr

```
==============================================================
CPU_instr.e: Basic structure of CPU instructions

This module defines the basic structure of CPU instructions,
```

5. In a similar fashion, highlight the phrase *instr.opcode* in the line four lines above (line 51) the **wait** statement and click Print.

   In the Data Browser, you can see that the value of opcode is JMPC.

6. 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*’ and 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 line:

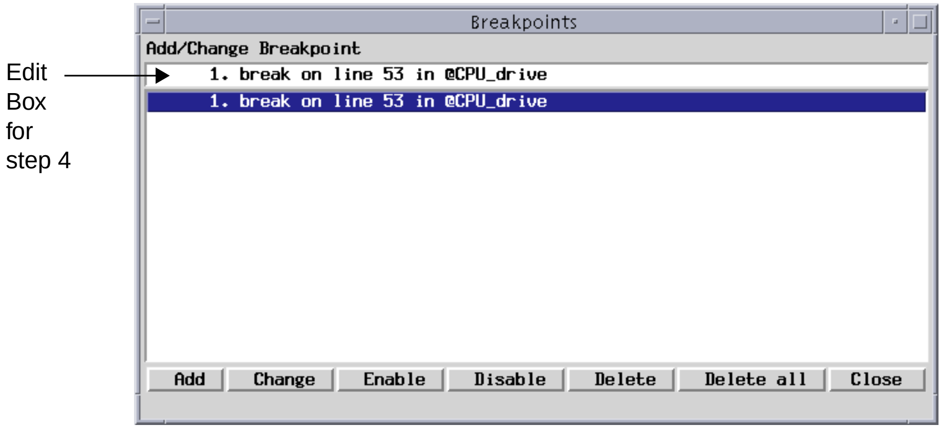
   ```
   'top.data' = pack(packing.high, instr.as_a(imm instr).op2);
   ```

2. On the Breakpoint menu, choose Set Breakpoint and then choose Break.

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

3. On the Breakpoint menu, choose Show All Breakpoints to activate the breakpoint just before the error occurs (at system time 2246).

   The Breakpoints window appears.



```
1. break on line 53 in @CPU_drive
1. break on line 53 in @CPU_drive
```

4. In the edit box at the top of the window, modify the current breakpoint by adding the condition "if (sys.time > 2200)", as follows:

   ```
   break on line 53 in @CPU_drive if (sys.time > 2200)
   ```

Tip If the error occurred at a simulation time other than 2246, choose a different value for the sys.time expression that is at least 46 time units before the error occurred.

5. Click *Change* to save the changes.

Stepping the Simulation

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

Procedure

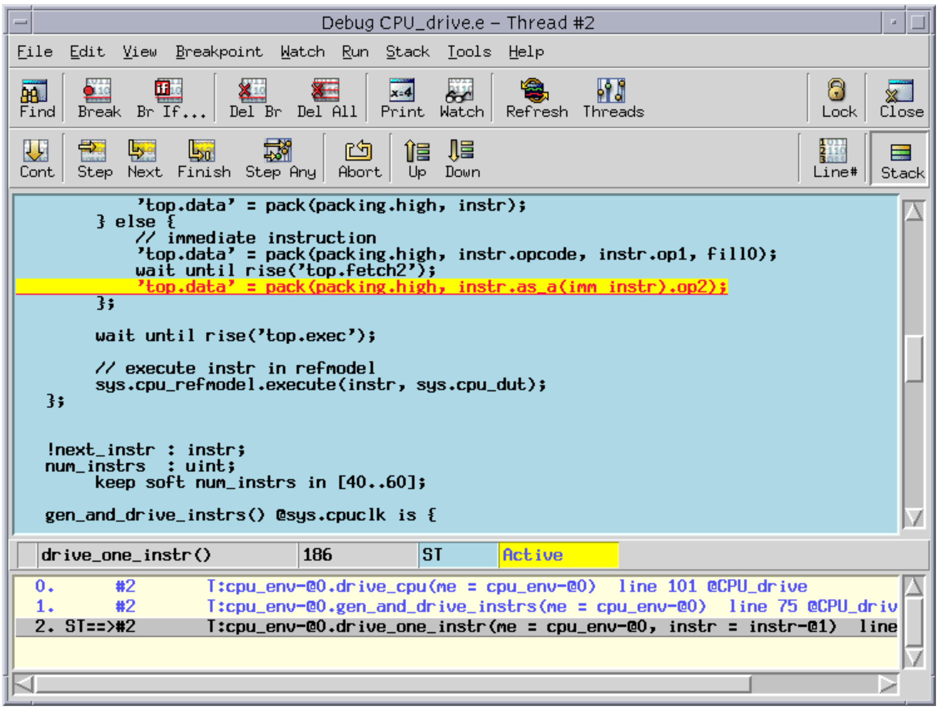
To step the simulation:

1. Click Reload in the Specview main window to run the simulation in debug mode.

   The Debugger window closes when you reload the design.

2. Click Test.

   The simulation stops at the breakpoint.



```
            '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 instr).op2);
        };

        wait until rise('top.exec');

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


   !next_instr : instr;
   num_instrs  : uint;
        keep soft num_instrs in [40..60];

   gen_and_drive_instrs() @sys.cpuclk is {
```

```
drive_one_instr()          186          ST          Active
```

```
 0.       #2         T:cpu_env-@0.drive_cpu(me = cpu_env-@0)   line 101 @CPU_drive
 1.       #2         T:cpu_env-@0.gen_and_drive_instrs(me = cpu_env-@0)   line 75 @CPU_driv
 2. ST==>#2          T:cpu_env-@0.drive_one_instr(me = cpu_env-@0, instr = instr-@1)   line
```

3. From the Run menu in the Debugger window, choose Step Any (or click the Step Any button) to advance to the next source line in any subsequent thread.

4. Continue clicking Step Any 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 Step Any.

5. In the Debugger window, click the Step button to step through the simulation within the current thread, Thread #3.

6. Continue clicking Step for about 35 to 40 steps until you hit the PROTOCOL error.

   The Step button is greyed out, and the PROTOCOL error is displayed in the Specview main 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 Elite 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 Elite 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 the editor.

3. Review the **keep** constraint.

```
<'
extend imm instr {

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

};
'>
```

4. In the Specview main window, choose Debug>>All Breakpoints>>Delete All Breakpoints.
5. Click Reload.
6. Click Load and load the *CPU_bypass.e* file.
7. Click 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 Elite 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 Elite 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 Elite’s powerful constraint-driven generator, you generated 15 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 Elite’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 Elite’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 Elite system to create a self-checking monitor for verifying protocol conformance.
- When the self-checking monitor revealed a bug, the Specview debugger provided extensive features to debug the design efficiently.

Note that 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 Elite system is a tremendous reduction in verification time and resources.

A Setting up the Tutorial Environment

To set up the tutorial environment, you need a Specman Elite license. You can get one by sending an email to *info@verisity.com* or by calling Verisity customer support at (650) 934-6890.

There are three procedures involved in setting up the tutorial environment:

- Downloading the Specman Elite software and tutorial files
- Installing the Specman Elite software
- Installing the tutorial files

These procedures are described in this appendix.

Note that even if Specman Elite software has already been installed in your environment, you still have to download and install the tutorial files.

Downloading the Required Files

To download the Specman Elite software and the tutorial files from the Verisity *ftp* site:

1. Change directory to the directory where you want to store the downloaded files.
2. Log in to the Verisity *ftp* site in the United States or Israel.

   United States:

```
% ftp ftp.verisity.com
Connected to ftp.verisity.com…
Name (ftp.verisity.com:your_name): anonymous
331 Guest login ok, send ident as password
Password: your-complete-email-address
230 Guest login ok, access restrictions apply.
```

Israel:

```
% ftp ftp-il.verisity.com
Connected to ftp.verisity.com…
Name (ftp.verisity.com:your_name): anonymous
331 Guest login ok, send ident as password
Password: your-complete-email-address
230 Guest login ok, access restrictions apply.
```

3. Change directory to the private/tutors directory.

```
ftp> cd private/tutors
250 CWD command successful
```

Note The private/tutors directory contains a README file that describes the contents of the directory.

4. Change the format type to **binary**.

```
ftp> bin
200 Type set to I.
```

5. Get the Specman Elite software.

```
ftp> get install_specmanrelease_number.sh
200 PORT command successful.
150 Opening BINARY mode data connection for
install_specmanrelease_number.sh
226 Transfer complete…
ftp> get sn_relrelease_number.main.tar.gz
200 PORT command successful.
150 Opening BINARY mode data connection for
sn_relrelease_number.main.tar.gz…
226 Transfer complete…
ftp> get sn_relrelease_number.OS.tar.gz
150 Opening BINARY mode data connection for
sn_relrelease_number.OS.tar.gz…
226 Transfer complete…
ftp> get sn_relrelease_number.docs.tar.gz
150 Opening BINARY mode data connection for
sn_relrelease_number.docs.tar.gz…
```

```
226 Transfer complete…
```

where *OS* is one of the platforms that Specman Elite supports, either **solaris** or **hpux**.

6. Get the tutorial files. For Specman Elite version 3.3.x, use 3.3 for *release_number*. For Specman Elite version 4.x, use 4.0 for *release_number*.

   ```
   ftp> get se_tutorrelease_number.tar.gz
   200 PORT command successful.
   150 Opening BINARY mode data connection for se_tutor.tar.gz…
   226 Transfer complete…
   ftp>
   ```

7. Log out of ftp.

   ```
   ftp> quit
   221 Goodbye.
   ```

Installing the Specman Elite Software

To set the environment variables, install the files, and start up the license manager:

1. Log in to the machine where you want to install the Specman Elite software.

   ```
   % rlogin solaris-machine | hpux-machine
   ```

2. Run the installation script.

   ```
   % sh ./install_specmanrelease_number.sh
   ```

3. When the Specman Elite Install Script Menu appears, choose option 1, “Complete installation”.

   After you have installed the machine-independent files and the machine-dependent files, the script will ask you to choose a license installation step.

4. Choose option 1, “Install license server” from the “License handling” menu.

   The script creates a new license file based on the license file you obtained via e-mail, activates the license server, updates the SPECMAN_LICENSE_FILE environment variable, and optionally creates the “rc.specman” file.

5. After the license handling procedure is completed, install the online docs.

   After you have installed the online docs, exit the installation script.

6. Source the Specman Elite environment file (env.csh or env.sh), for example:

   ```
   % source install_dir/release_number/env.csh
   ```

7. Make sure the Specman Elite object in your PATH is the one you have just installed.

```
% which specman
install_dir/OS/specman
%
```

8. To check the installation, start the Specman Elite graphical interface.

```
% specview &
```

Tip If you have difficulty starting Specview or obtaining a license, call Verisity customer support at (650) 934-6890.

Installing the Tutorial Files

To install the tutorial files.

1. Change directory to the directory where you want to install the tutorial files.

```
% cd tutor_dir
%
```

2. Unzip and untar the se_tutor.tar.gz file.

```
% gunzip se_tutorrelease_number.tar.gz
% tar -xvf se_tutor.tar
```

3. 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_4.0.ecov
CPU_drive.e         CPU_top.e

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_4.0.ecov
CPU_drive.e         CPU_top.e
%
```

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.

B 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 B-1 Register Instruction

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

Figure B-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 B-1 shows a summary description of the CPU instructions.

Table B-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

Table B-1 Summary of Instructions (continued)

Name	Opcode	Operands	Comments
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

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 B-2.

Table B-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 B-3.

Table B-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
register 2	8 bits	r2
register 3	8 bits	r3