Specman Elite™ Tutorial

Revision 1.1

Trademarks

Confidentiality Notice

RESTRICTED RIGHTS LEGEND

Destination Control Statement

Table of Contents

1 Introduction

Overview

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

- Drastic reductions in the time and resources required for verification
- Significant 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

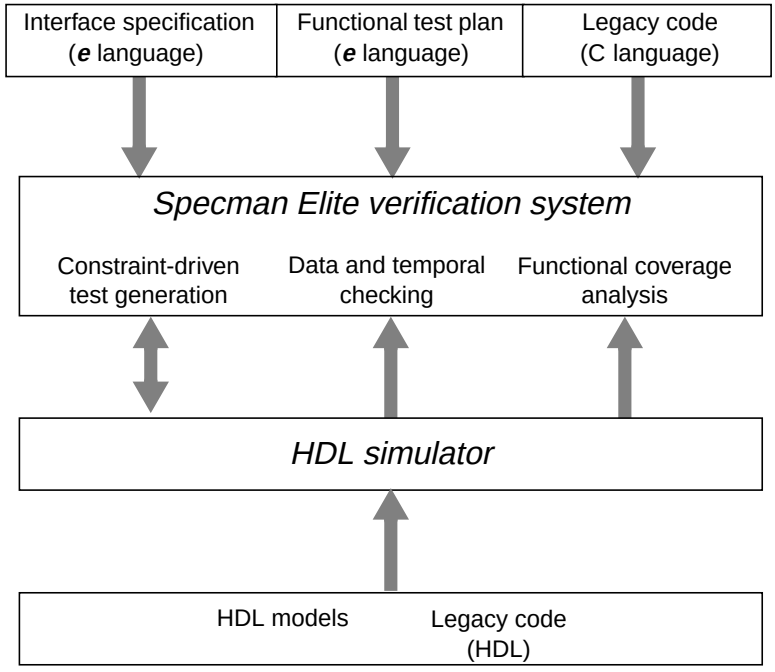
The Specman Elite system provides three main enabling technologies that drastically increase your productivity:

- **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 quickly 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** — because you can measure the progress of your verification effort against a functional test plan, you avoid creating redundant tests that waste simulation cycles.

Figure 1-1 on page 1-2 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

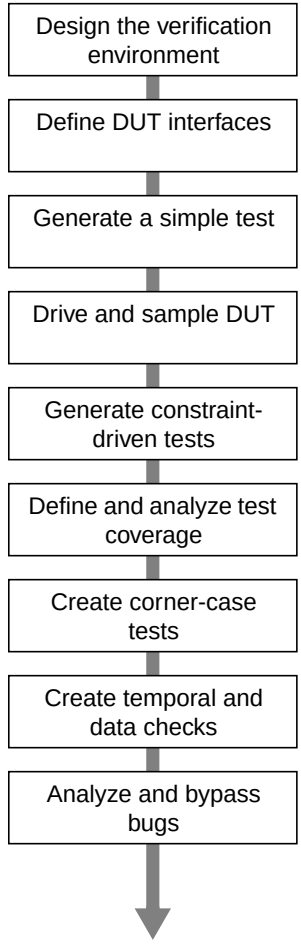


Tutorial Goals

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 on page 1-3. The tutorial uses the Specman Elite system to create a verification environment for a simple CPU design.

Figure 1-2 Tutorial Verification Task Flow



Setting up the Tutorial Environment

Before starting the design verification task flow shown in Figure 1-2 on page 1-3, you need to set up the tutorial environment. You must perform three main steps to set up the tutorial environment:

1. Downloading the Specman Elite software and tutorial files
2. Installing the Specman Elite software
3. Installing the tutorial files

Appendix A, “Setting up the Tutorial Environment” describes how to perform these steps.

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

Document Conventions

The tutorial uses the document conventions described in Table 1-1 on page 1-4.

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.

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 results 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:

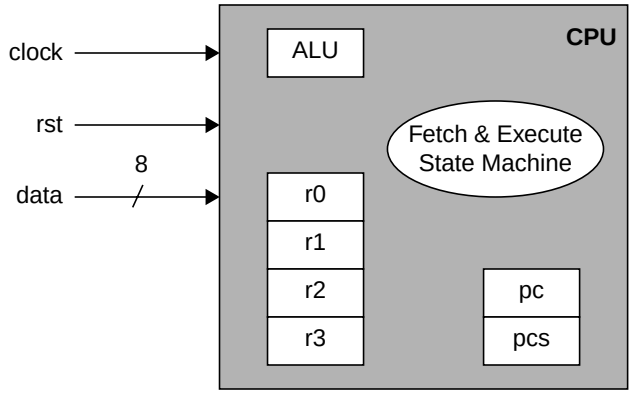
- Design specifications
- Interface specifications
- Functional test plan
- Overall verification environment

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

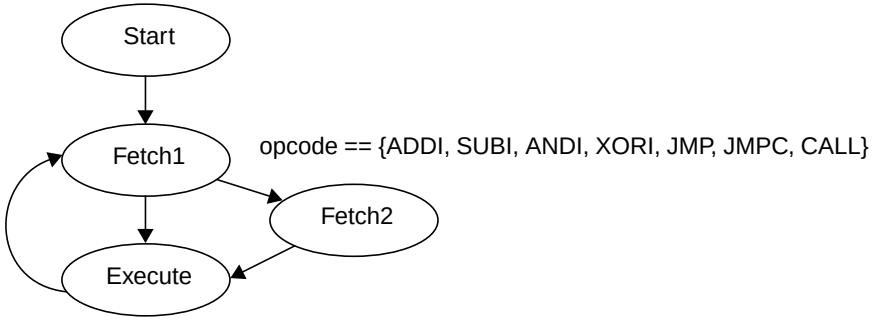
The device under test (DUT) is an 8-bit CPU with a reduced instruction set.

Figure 2-1 CPU Block-Level Diagram



The state machine diagram for the CPU is shown in Figure 2-2 on page 2-2. The second fetch cycle is only for *immediate* instructions.

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 instructions:

- **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

Create a simple go-no-go test to confirm that the verification environment is working properly.

Test Strategy

- 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

Create multiple random tests to gain high percentage coverage on commonly executed instructions.

Test Strategy

- Use constraints to drive random testing towards the more common arithmetic and logic operations rather than the control flow operations.
- Create 15 sets of tests

Test 3

Test Objective

Generate 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 Strategy

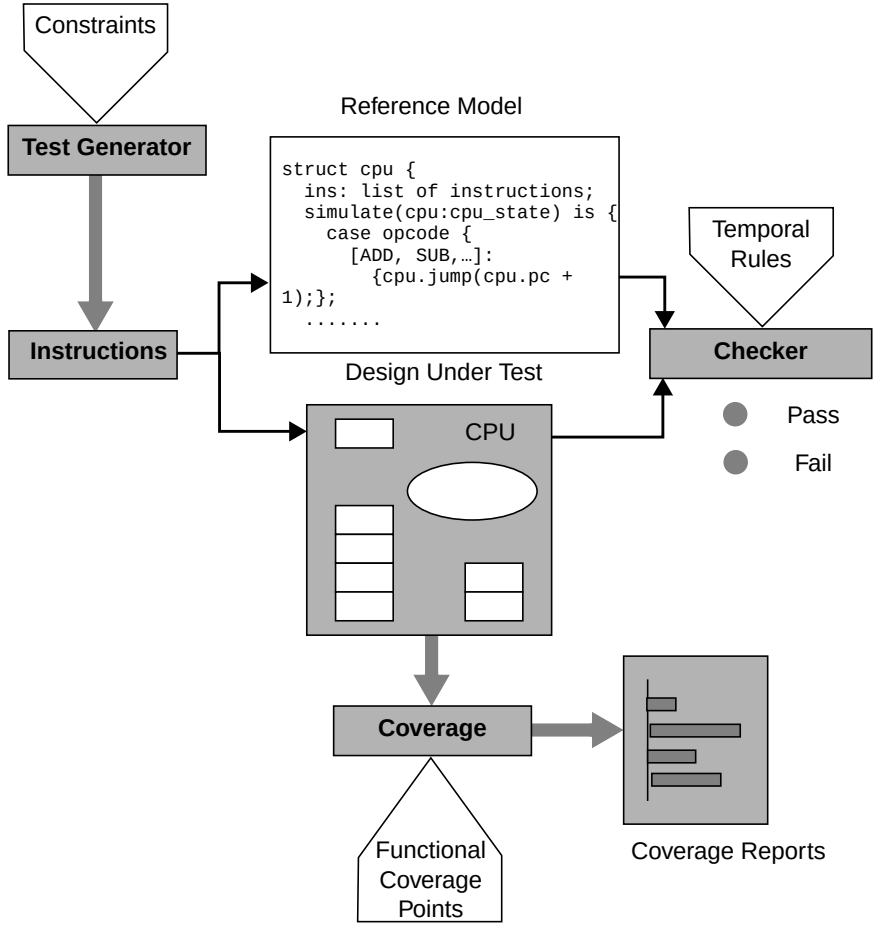
- Generate many arithmetic opcodes to increase the chances of carry bit assertion.
- Monitor the DUT and use on-the-fly generation to generate many JMPC opcodes when the carry signal is high.

Overview of the Verification Environment

The overall test strategy, which is shown in Figure 2-5 on page 2-5, is to

- Constrain the Specman Elite test generator to create valid CPU instructions
- Compare the program counter 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



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**. With just a few constructs from the ***e*** language you can define legal CPU instructions exactly as they are described in the interface specifications.

This chapter introduces the ***e*** constructs shown in Table 3-1 on page 3-2.

Table 3-1 Constructs Used in this Chapter

***e* Construct**	**How the Construct is Used in this Chapter**
<'…'>	Delineates the beginning and end of ***e*** code.
bits	Defines the width of an enumerated type.
extend	Adds the data structure for the CPU instructions to the Specman Elite system of data structures.
keep	Specifies rules or constraints for the instruction fields.
list of	Creates an array or list without having to keep track of pointers or allocate memory.
struct	Creates a data structure to hold the CPU instructions.
type	Defines an enumerated data type for the CPU instructions.
when	Implements conditional constraints on the possible values of the instruction fields.

To create the CPU instruction structure, you need to

- Capture the interface specifications
- Create a list of instructions

The following sections describe how to perform these tasks.

Capturing the Specifications

In this step, 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

Follow this procedure to capture the design specifications in ***e***:

1. Make a new working directory.
2. Copy the *tutorial/src/CPU_instr.e* file to the working directory.
3. 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
```

4. 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

lines beginning with // are comments

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 a stream of instructions

};
'>
```

5. 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 Specman Elite 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
.
.
.
};
```

6. 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
.
.
.
};
```

7. 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;
   };
.
.
.
};
```

8. 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

};
```

9. 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;
   };
};
```

   Now you have finished defining a legal CPU instruction.

Creating the List of Instructions

To create a CPU instruction structure, you extend 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.

1. 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 Specman Elite to create an empty data structure to hold the instructions. Then, each test tells Specman Elite 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

As described in the functional test plan, the 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 Specman Elite. Test constraints target 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** — the ability to add definitions, constraints, and methods to a struct in order to change or extend its original behavior, without modifying the original definition.
- **Constraint solver** — the core technology that intelligently resolves all specification constraints and test constraints and then generates the desired test.

This chapter shows new uses of the ***e*** constructs introduced in Chapter 3, “Creating the CPU Instruction Structure”. It also introduces the Specman Elite commands shown in Table 4-1 on page 4-2.

Table 4-1 Constructs and Commands Used in this Chapter

***e* Constructs**	**How the Construct is Used in this Chapter**
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.
Specman Elite Commands	
File->Load	Loads uncompiled ***e*** modules into Specman Elite.
Objects->Modules	Lists the ***e*** modules you have loaded into Specman Elite.
Test->Test	Generates a test based on the constraints you specify.

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

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

The following sections describe how to perform these steps.

Defining the Test Constraints

The Functional Test Plan for the CPU design describes the objectives and specifications for this first test.

Test Objectives

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

Test Specifications

To meet the test objectives, 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

Follow this procedure to capture the test constraints in ***e***:

1. Copy the *tutorial/src/CPU_tst1.e* to the working directory.
2. Open the *CPU_tst1.e* file in an editor.
3. Find the portion of the file that looks like this:

```
<'
import CPU_top;

extend instr {
   // test constraints

};

extend sys {
   // generate 5 instructions

};
.
.
.
```

4. Add lines below the comments 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;
};
```

5. 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.

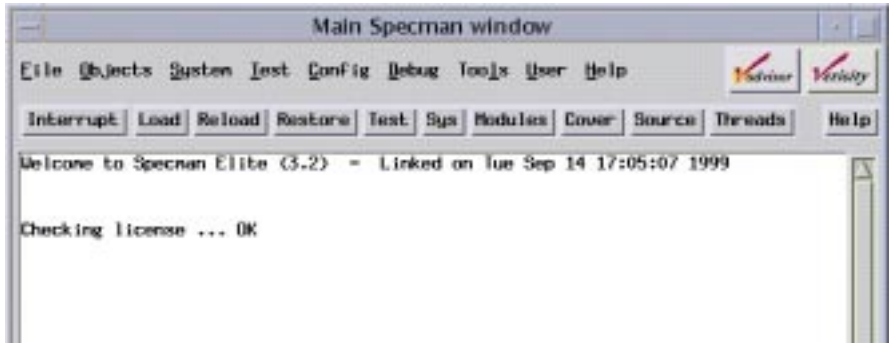
These files are called modules in Specman Elite. Before Specman Elite can generate the test, you must load all the modules. Here's how to do this:

1. Copy the *tutorial/src/CPU_top.e* file to the working directory.
2. Copy the *tutorial/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 in the following command at the UNIX prompt to invoke Specman Elite's graphical user interface, Specview™:

   ```
   % specview &
   ```



✓ If the Main Specman window 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.

4. Select *File->Load* from the menu or click the Load button.

5. When the *Select a File* window appears, double-click on the *CPU_tst1.e* file name in the list of files.

   You should see a message in the Main Specman window:

   ```
   load /tutorial/CPU_tst1.e…
   Load complete
   ```

✓ 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 and select the correct file.

✓ 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 on the blue hypertext link in the Main Specman window, and the error location will be displayed in the Debugger window.

6. To see a list of loaded modules, select *Objects->Modules* or click the Modules button.

   There should be four modules loaded:

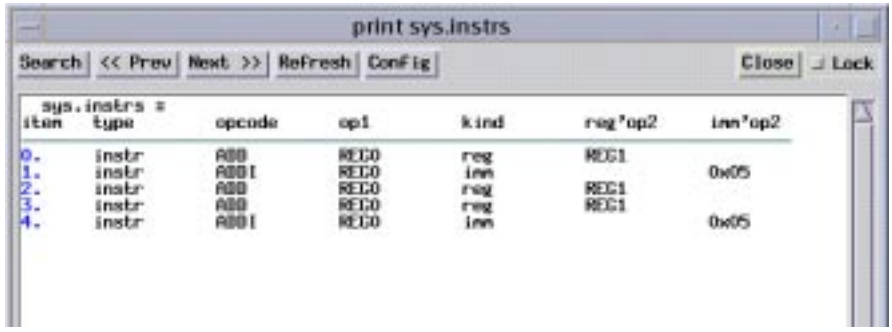
   ```
   CPU_instr
   CPU_misc
   CPU_top
   CPU_tst1
   ```

Generating the Test

Here's how to generate the test:

1. In the Main Specman window, select *Test->Test* or click the Test button.

   The *print sys.instrs* window appears, showing the generated list of five instructions.



✓ 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/*.

2. Review the list 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

The objective of this chapter is to 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. In order to streamline this tutorial, however, 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 on page 5-2.

Table 5-1 New Constructs

e **Constructs**	**How the Constructs are Used in this Chapter**
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.

These are the steps for driving and sampling the DUT:

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

The following sections describe these tasks in detail.

Defining the Protocols

There are two protocols to define for the CPU:

- **Reset protocol** — drives the *rst* signal in the DUT.
- **Drive instruction 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 significantly more powerful on-the-fly generation.

All the TCMs required to drive the CPU are described briefly in Table 5-2 on page 5-3. A complete description of one of the TCMs follows the table. You can also view the *CPU_drive.e* file in the *tutorial/src* directory, if you would like 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 ()* and 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 and, if the instruction is an immediate instruction, waits for the *fetch2* signal to rise and sends the second byte of data. Then this TCM waits for the *exec* signal to rise.

Figure 5-1 on page 5-4 shows the ***e*** code for the *drive_one_instr ()* TCM. The CPU architecture requires that the tests drive and sample the DUT on the falling edge of the clock, so all TCMs are synchronized to *cpuclk* which is defined as follows:

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

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);
};
```

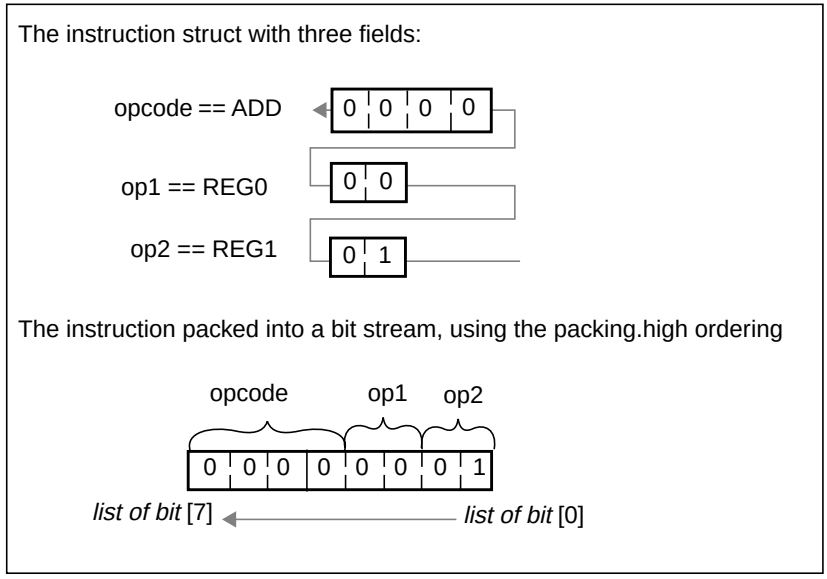
The assignment statements in Figure 5-1 on page 5-4 show how to drive and sample signals in an HDL model. Each pair of single quotation marks identifies an object as an HDL signal.

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 on page 5-4 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. The **%** character you used in Chapter 3, “Creating the CPU Instruction Structure”, denotes the fields that should be driven into the DUT. The **pack()** function intelligently and automatically performs the conversion, as shown in Figure 5-2 on page 5-5.

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



Running the Simulation

This procedure, which involves loading the appropriate files and clicking the Test button, is exactly the same as 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

1. Copy the *tutorial/src/CPU_drive.e* to the working directory.
2. Copy the *tutorial/src/CPU_dut.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.

✓ 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
```

✓ 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 the Load button and select *CPU_tst1.e* in the Select A File dialog box.

9. Click the Test button to run the simulation.

   You should see the following messages (or something similar) in the Main Specman 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 :       ADDI   REG0x0, @0x05
DUT executing instr 2 :       ADD    REG0x0, REG0x1
DUT executing instr 3 :       ADD    REG0x0, REG0x1
DUT executing instr 4 :       ADDI   REG0x0, @0x05
Last specman tick - stop_run() was called
Normal stop - stop_run() is completed
Checking the test…
Checking is complete - 0 DUT errors, 0 DUT warnings.
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

As described in the functional test plan, the second test requires you to generate multiple tests using constraint-driven test generation. 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:

- **Constraint-driven random test generation** — this feature lets you focus random test generation on the areas of the design that need to be exercised the most.
- **Random seed generation** — by changing the seed used for random generation, you can quickly cause the Specman Elite system to generate a whole new set of tests.

This chapter introduces the ***e*** constructs and Specview commands shown in Table 6-1 on page 6-2.

Table 6-1 New Constructs and Commands

***e* Constructs**	**How the Construct is Used in this Chapter**
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.
Specview Commands	
Config->Generation	Used to create 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 required to generate random tests are:

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

Follow this procedure to see how to create weighted constraints in ***e***:

1. Copy the *tutorial/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 like this 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:

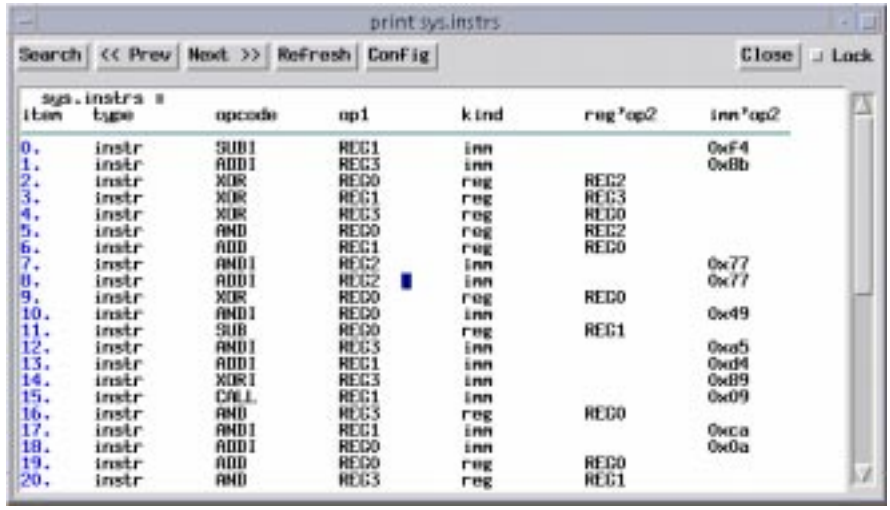
1. Click on the Restore button in the Main Specman window to remove all the ***e*** modules from the current session.
2. Click on the Load button and load the *CPU_tst2.e* file.
3. Click on the Modules button and confirm that the following modules are loaded:

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

4. Select *Config->Generation* in the Main Specman window.
5. In the Seed field of the Specman Elite Configuration Options window, enter your lucky number.
6. Close the Specman Elite Configuration Options window by clicking OK.
7. Click on the Test button.

8. Review the instructions in the *print sys.instrs* window.

   You should see an equal distribution of arithmetic and logic operations and fewer control flow operations.



Generating Tests With a Random Seed

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

Procedure

This procedure generates tests using a Specman Elite-generated seed:

1. Select Reload button.
2. Select *Test->Test With Random Seed* in the Main Specman window.
3. Review the results in the *print sys.instrs* window.

   You should see an equal distribution of arithmetic and logic operations and fewer control flow operations. The results should be different from the previous run.

4. Optionally you could repeat steps 1-3 several times to confirm that you see different results each time.

✓ 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 Points

Goals for this Chapter

The goal for this chapter is to avoid redundant testing and to measure the progress of the verification effort by generating effective test coverage statistics.

What You Will Learn

In this chapter, you learn how to define coverage points 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 on page 7-2.

Table 7-1 New Constructs

***e* Constructs**	**How the Construct is Used in this Chapter**
cover	Defines a group of data collection items.
event	Defines a sampling point when coverage data collection occurs.
item	Identifies an object to be sampled.
transition	Identifies an object whose current and previous values are to be collected when the sampling point occurs.

The steps required to define coverage points are:

1. Defining coverage points for the finite state machine (FSM).
2. Defining coverage points for the generated instructions.
3. Defining coverage points for the corner case.

The following sections describe these tasks in detail.

Defining Coverage Points for the FSM

You can use the constructs shown in Table 7-1 on page 7-2 to define the following coverage points for the FSM:

- State machine register
- State machine transition

Procedure

Follow this procedure to define coverage points for the FSM:

1. Copy the *tutorial/src/CPU_cover.e* file to the working directory.
2. Open the *CPU_cover.e* file in an editor.

3. Find the portion of the file that looks like the excerpt below and review the event declaration that defines the sampling point for the FSM:

defines FSM sampling point

```
extend cpu_env {

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

4. Add the cover group and cover items for state machine coverage.

   The cover group name (*cpu_fsm*) must be the same as the event name defined in step 3 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. In summary this means that whenever the *sys.cpuclk* signal changes, Specman Elite collects the current and previous values of *top.cpu.curr_FSM*.

defines the cover group cpu_fsm

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

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

Defining Coverage Points for the Generated Instructions

You can use the constructs shown in Table 7-1 on page 7-2 to define the following coverage points for the CPU instruction stream:

- opcode
- op1

This coverage group uses as a sampling point an event 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

Follow this procedure to extend the *instr* struct to define these coverage points:

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

```
                extend instr {

      defines       cover start_drv_DUT is {
coverage group
                      };

                };
```

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

```
extend instr {

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

Defining Coverage Points for the Corner Case

In order to test the behavior of the DUT when the JMPC (jump on carry) instruction opcode is issued, we need to be sure that the carry signal is high when the JMPC opcode is issued. Here, you define a coverage group so you can determine how often that sequence of events occurs.

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 coverage point between opcode and carry.

   Cross coverage lets you define the intersections of two or more coverage items, generating a more informative report.

```
extend instr {

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

3. Save the *CPU_cover.e* file.

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

8 Analyzing Coverage

Goals for this Chapter

The goal for this chapter is to determine whether the tests you have generated meet the goals set in the functional test plan and 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 cover 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** — lets you view the intersections of two or more cover items.
- **Apropos** — helps you find the information you need in the Specman Elite Online Documentation.

This chapter introduces the Specview commands shown in Table 8-1 on page 8-2.

Table 8-1 New Commands

Specview Commands	
Tools -> Coverage	Displays coverage reports and creates cross-coverage reports.
Help	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

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. Invoke Specman Elite, if it is not already running.

   ```
   % specview &
   ```

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

6. Click the Modules button to confirm that seven modules are loaded:

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

7. Click the Test button.

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

   ```
   test
   Doing setup…
   Generating the test using seed 0x1
   Starting the test…
   Running the test…
   DUT executing instr   0 :       RET     REG0x3, REG0x0
   DUT executing instr   1 :       JMPC    REG0x3, @0x02
   DUT executing instr   2 :       ADD     REG0x3, REG0x2
   DUT executing instr   3 :       AND     REG0x3, REG0x1
   DUT executing instr   4 :       JMPC    REG0x3, @0x09
   .
   .
   .
   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 3.2, you may see different results in your coverage reports.

Procedure

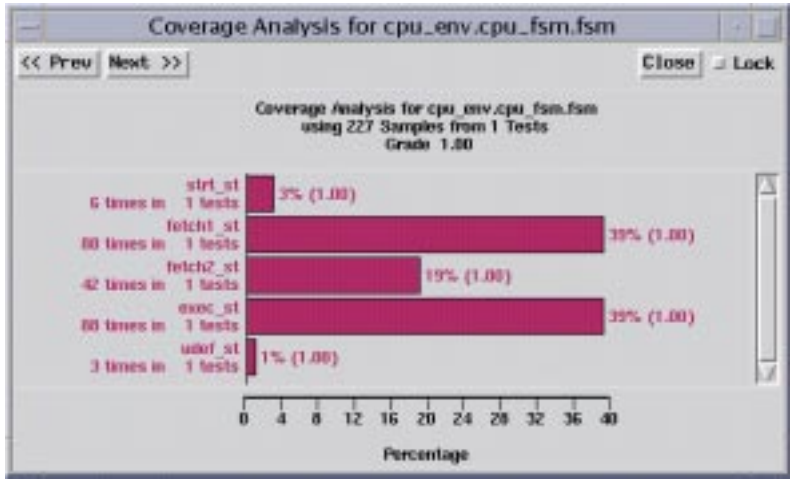
1. Click the Cover button in the Main Specman window.

   The Show Coverage dialog box appears.

2. In the Group field, select *cpu_env.cpu_fsm*.

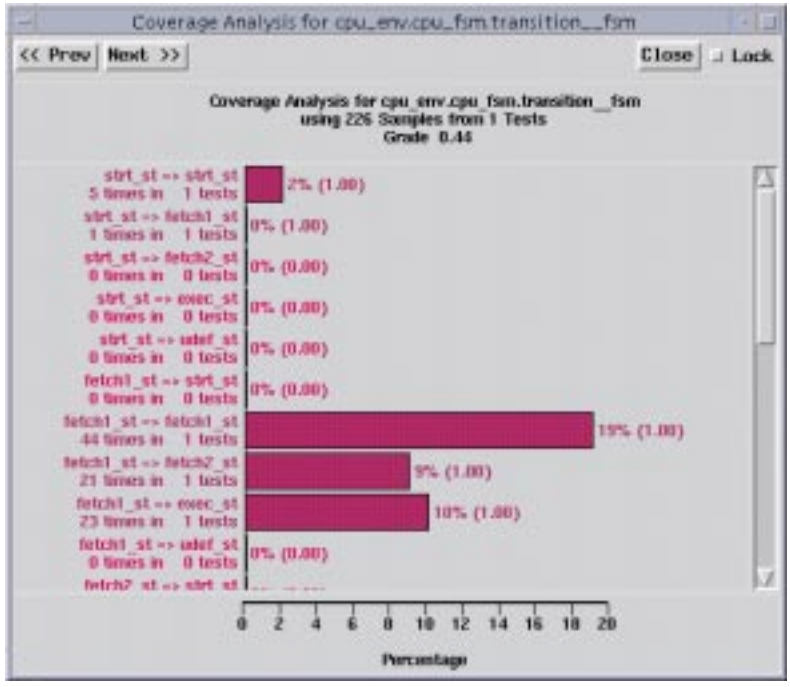
3. In the Items field, select *fsm* and click the Display Item button.

   The state machine register report appears in the Coverage Analysis window. From the report it is easy to see that, for example, the *fetch1* state was entered 88 times in the 227 times sampled.



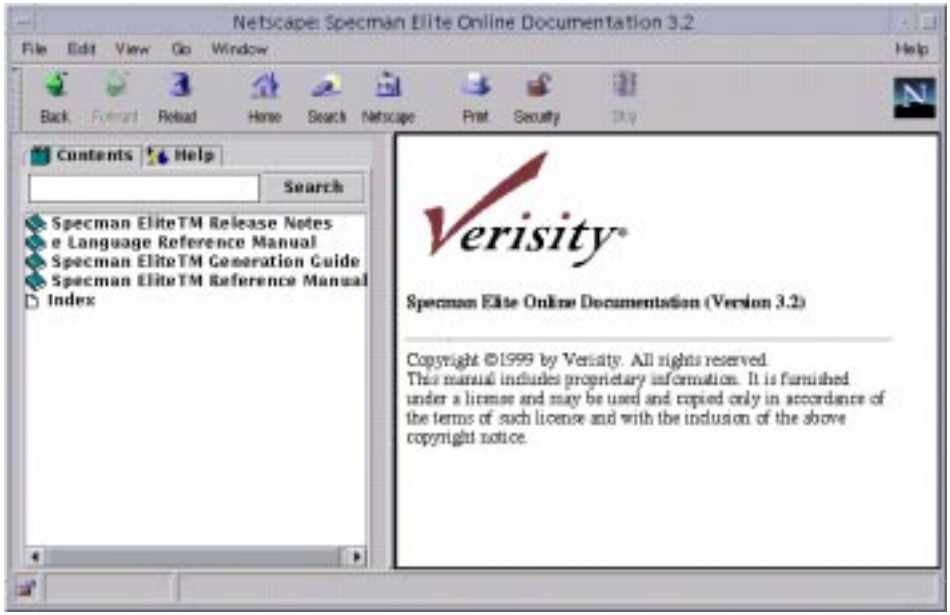
4. In the Items field, select *transition_fsm* and click the Display Item button.

   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.



5. To see how to define transitions as illegal so that they do not appear in the coverage report, click the Help button in the Main Specman window.

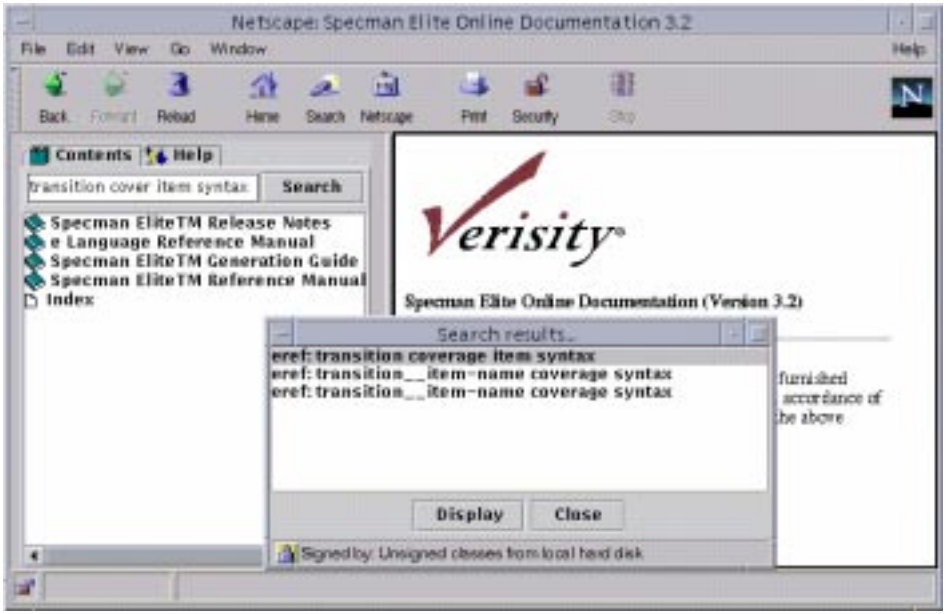
   The Specman Elite Online Documentation browser appears.



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

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

7. When the list of files that describe cover item options appears, select the first item in the list, *eref: transition cover item syntax*.



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

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

9. 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

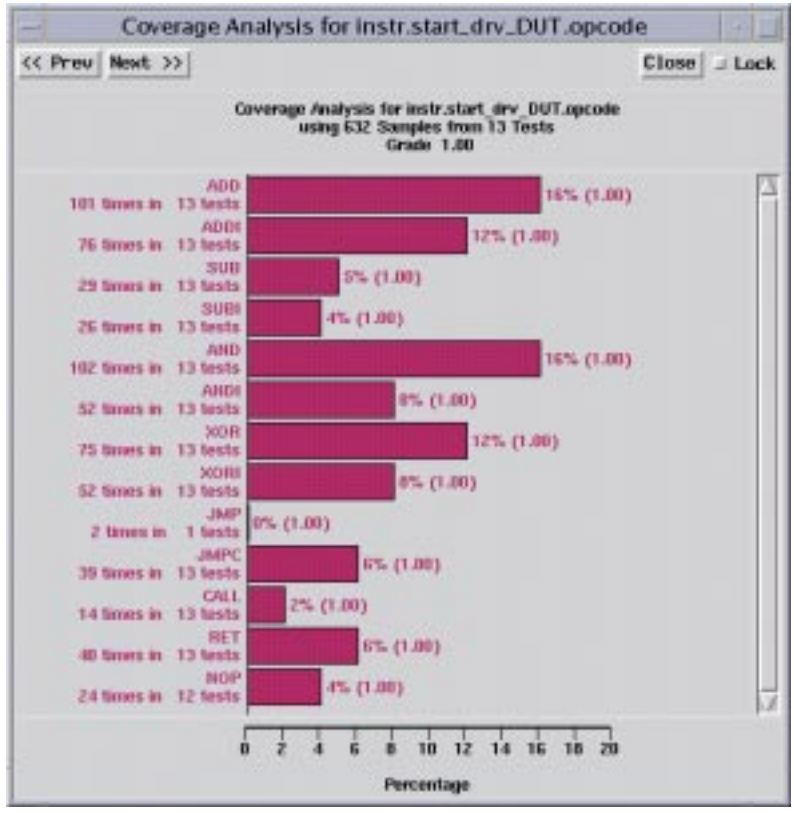
Now we'll look at the coverage for the CPU instruction stream. To provide more interesting results to look at, we'll load the results of a set of regression tests. These regression tests were run with the second test and many different random seeds.

Procedure

1. Copy *tutorial/src/regression_3.2.ecov* file to the working directory.
2. Click the Cover button in the Main Specman window.

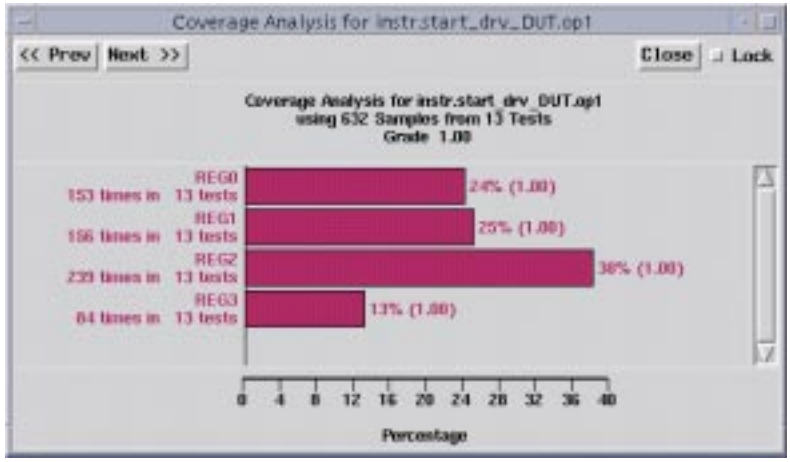
   The Show Coverage dialog box appears.
3. Click the Read... button in the Show Coverage dialog box and select the *regression_3.2.ecov* file.
4. Select *instr.start_drv_DUT* from the scroll list of the Group field.
5. In the Items field, select *opcode* and click the Display Item button.

The opcode coverage report appears in the Coverage Analysis window. 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.



6. In the Items field, select *op1* and click the Display Item button.

   The *op1* coverage report appears in the Coverage Analysis window. All possible *op1* values appear to be well covered.



7. In the Items field, select *opcode* and click the Add button under Cross Items.
8. In the Items field, select *op1* and click the Add button under Cross Items.
9. Click the Display Cross button to display the intersection of *opcode* and *op1*.

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

Viewing Corner Case Coverage

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

Procedure

1. Click the Cover button in the Main Specman window.

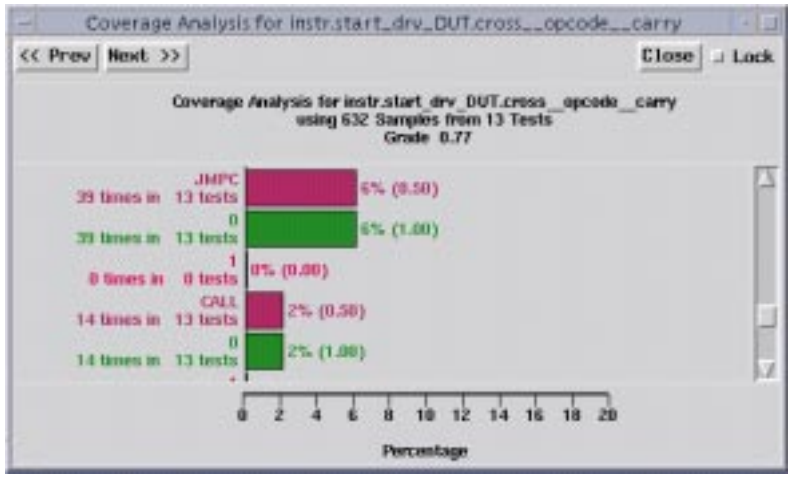
   The Show Coverage dialog box appears.

2. Select *instr.start_drv_DUT* from the scroll list of the Group field.
3. In the Items field, select *cross__opcode__carry* and click the Display Item button.

The cross coverage report for opcode and carry appears in the Coverage Analysis window.

4. Scroll down to the JMPC opcode. (Opcodes are shown in red.)

   You can see that the JMPC code was issued 39 times, and that *carry* was low each time.



The ability to cross test input with the DUT's internal state yields valuable information — 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 your corner case scenario testing. With the Specman Elite **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 lets you effectively generate a corner case test scenario, saving 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 on page 9-1.

Table 9-1 New Constructs

e Constructs	How the Construct is Used in this Chapter
'signal' * weight : value	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:

1. Increasing the probability of arithmetic operations.
2. 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 possibly be high when arithmetic operations are performed, so the test first has to increase generation of arithmetic operations over other types of operations.

Procedure

1. Copy the *tutorial/src/CPU_tst3.e* file to the working directory.
2. Open the *CPU_tst3.e* file in an editor.
3. 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

    };
};
```

4. 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
   };
};
```

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 enables you to 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, together with the name of the carry signal.

```
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, 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

The goal for this chapter is to check timing-related dependencies and to 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 on page 10-2.

Table 10-1 New Constructs and Commands

***e* Constructs**	**How the Construct is Used in this Chapter**
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 create 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

Follow this procedure to create the check:

1. Copy the *tutorial/src/CPU_checker.e* file to the working directory.
2. Open the *CPU_checker.e* file in an editor.

3. Find the portion of the file that looks like this:

defines start of exec state

defines rise of fetch1

```
// 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.
```

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

issues an error message if fetch1 doesn't rise exactly one cycle after entering execute state

```
// 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");
```

5. 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 can use a rule-based methodology instead. 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 create checks that compare the program counter in the DUT to their counterparts in the reference model.

Procedure

This procedure has two parts:

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

Adding 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.

event definition

method associated with event

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

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

2. Add a check for the program counter by creating a **check** statement and removing the comment characters in front of *dut_error*.

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

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.
3. 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;
};
'>
```

4. Find the line that extends the Specman Elite system by creating an instance of the CPU reference model.

5. Remove the comment characters.

```
                    <'
                    import CPU_refmodel;

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

                        cpu_env : cpu_env;
creates an          cpu_dut : cpu_dut;
instance of the     cpu_refmodel : cpu_refmodel;
reference model     };
                    '>
```

6. Find the line in the *reset_cpu* TCM that resets the reference model.
7. Remove the comment characters.

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

8. Find the line that executes the reference model when the DUT is in the execute state.
9. Remove the comment characters.

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

10. Save the *CPU_drive.e* file.

Running the Simulation

This procedure, which involves loading the appropriate files and clicking the Test button, is exactly the same as the procedure you used in previous chapters to generate other tests.

The difference is that this time you are including the reference model and checks.

Procedure

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 *tutorial/src/CPU_refmodel.e* file to the working directory.
6. Invoke the Specman Elite system, if it is not already running.

```
% specview &
```

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

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

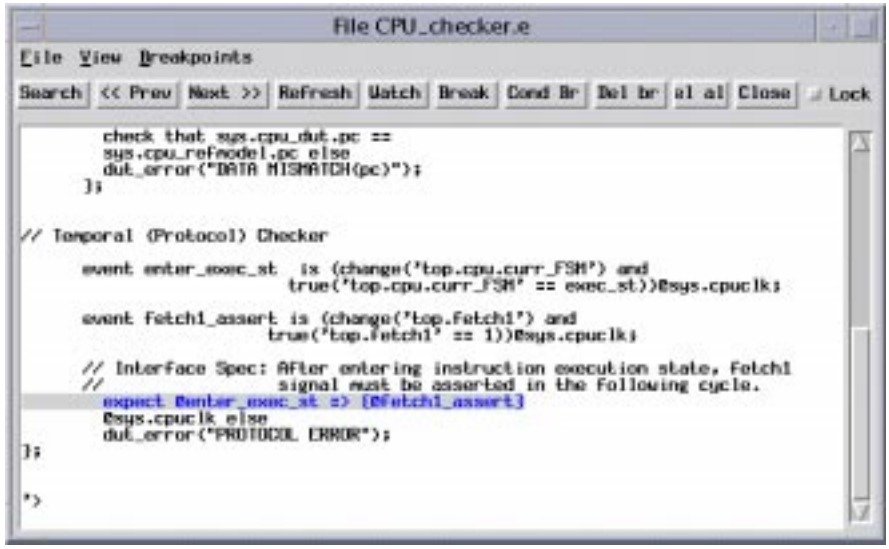
9. Click the Test button to run the simulation.

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

```
test
Doing setup ...
Generating the test using seed 0x7...
Starting the test ...
Running the test ...
DUT executing instr   0 :       SUB     REG0x0, REG0x2
DUT executing instr   1 :       NOP     REG0x2, REG0x3
DUT executing instr   2 :       ADDI    REG0x1, @0x9e
DUT executing instr   3 :       SUBI    REG0x1, @0x65
DUT executing instr   4 :       SUB     REG0x1, REG0x0
DUT executing instr   5 :       ADD     REG0x0, REG0x2
DUT executing instr   6 :       SUB     REG0x0, REG0x1
DUT executing instr   7 :       SUBI    REG0x3, @0xca



   *** Dut error at time 525
        Checked at line 41 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
```

10. Click on 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.



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 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** — Powerful debugging features with visibility into the HDL design.
- **The Specman Elite bypass feature** — You can temporarily prevent the Specman Elite system from generating test data that reveals a bug in the design. This feature lets you continue testing while the bug is being fixed.

This chapter introduces the Specview commands shown in Table 11-1 on page 11-2.

Table 11-1 New Commands

Specview Commands	
Debug -> Thread	Opens the Thread Browser, which displays all the TCMs (threads) that are currently active.
Debug -> Show Thread Source…	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	Sets a breakpoint on the currently highlighted line of ***e*** code.
Debugger: Run -> Step	Advances simulation to the next line of ***e*** code.

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.

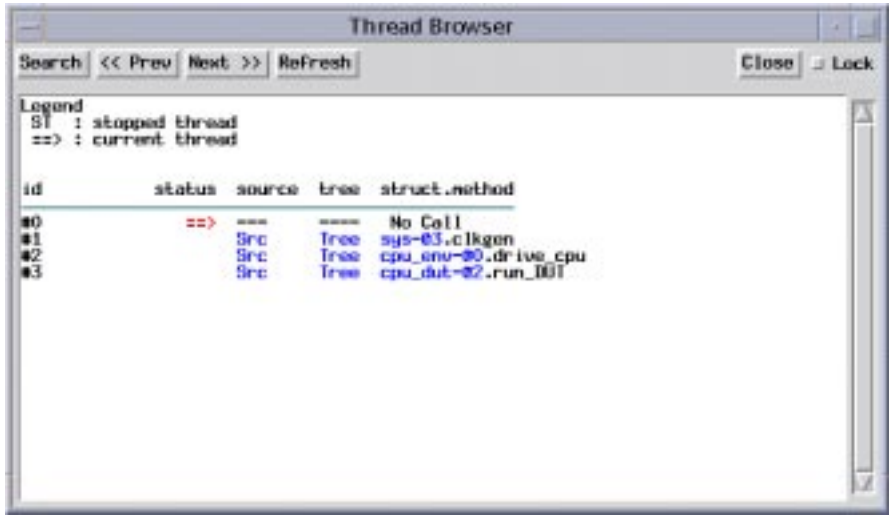
Displaying DUT Values

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

1. In the Main Specman window, select *Debug->Threads*.

   The Thread Browser appears.



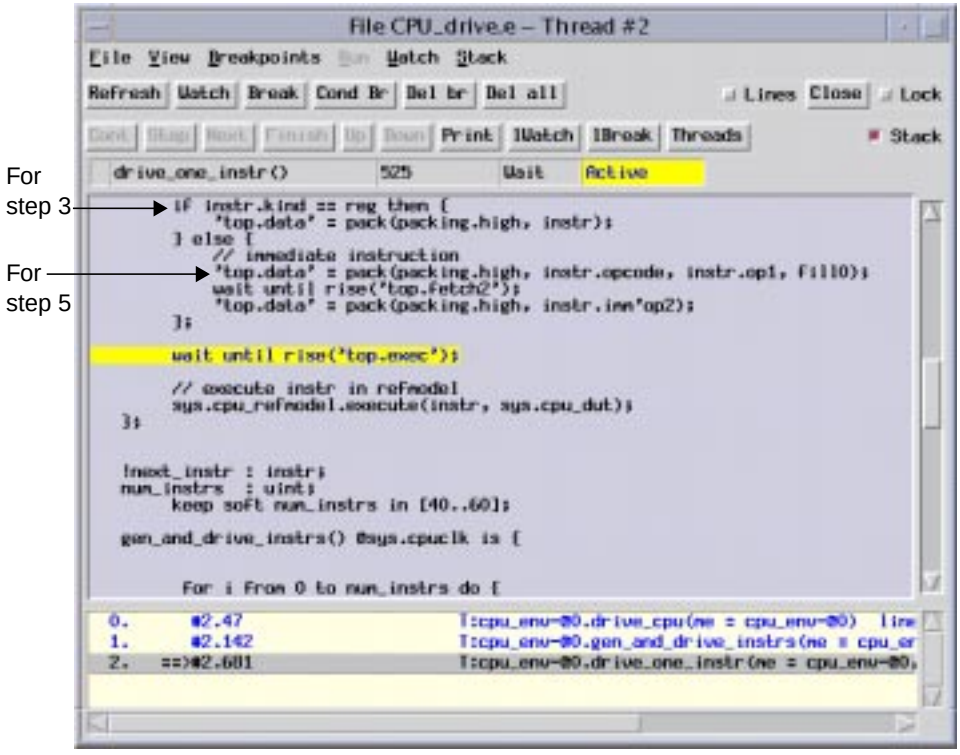
   The Thread Browser indicates the status of each TCM that is currently active in Specman Elite:

   - 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*, select *Src* to bring up the corresponding source file for this thread.

   The Debugger window appears, showing *CPU_drive.e*. The highlighted line shows that the *drive_one_instr* TCM is waiting for the *top.exec* signal to rise.



3. To find out the current instruction type, highlight the phrase *instr.kind*, located 9 lines above the highlighted **wait** statement.

4. Click the print button.

   The value of *instr.kind* shows an immediate instruction.

5. In a similar fashion, find out the current opcode: *instr.opcode*.

   The value of opcode is JMPC.

6. Optionally you could find out the value of any HDL signals. For example, to display the value of *top.data*, highlight the phrase '*top.data*' and click the print button.

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

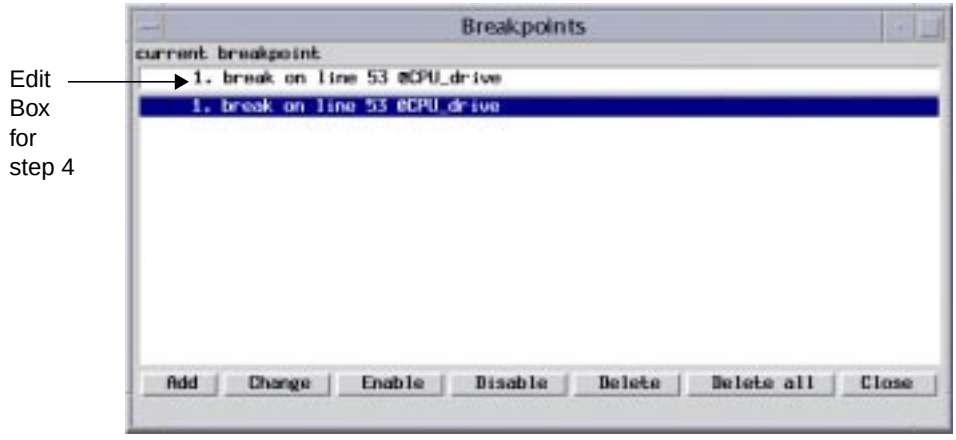
1. Highlight any portion of the line:
   ```
   'top.data' = pack(packing.high, instr.imm'op2);
   ```

2. Select *Breakpoints -> Set Breakpoint*.

   This line should now be underlined to indicate a breakpoint has been set.

3. To activate the breakpoint just before the error occurs (at system time 525), in the Debugger window, select *Breakpoint -> Show Breakpoints*.

   The Breakpoints window appears.



4. In the edit box at the top of the window, modify the current breakpoint as follows:

   ```
   break on line 53 @CPU_drive if (sys.time > 475)
   ```

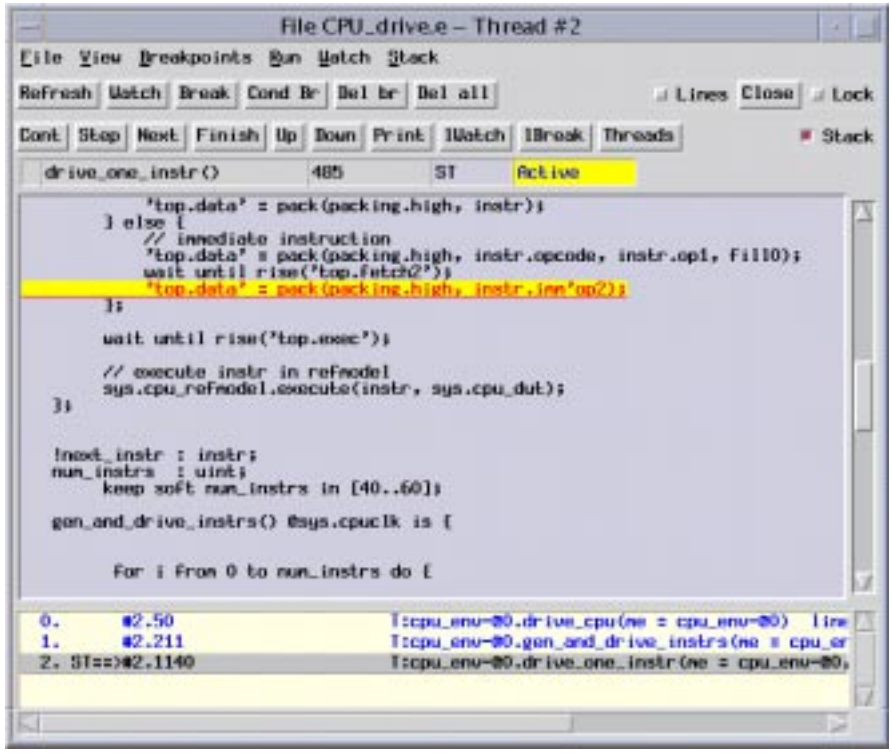
   Click *Change* and *Close* the window.

Stepping the Simulation

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

Procedure

1. To run the simulation in debug mode, click the Reload button in the Main Specman window.
2. Click the Test button.
3. The simulation stops at the breakpoint.



4. In the Debugger window, select *Run -> Step Anywhere* to advance to the next source line in any subsequent thread.
5. Repeat the *Run -> Step Anywhere* 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.

6. In the Debugger window, continue to click the *Step* button to step through the simulation.

   After about 22 steps, you will hit the PROTOCOL error. The PROTOCOL error is displayed in the Main Specman window and the Step button is greyed out.

   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.

Specman Elite's extensibility feature, however, lets you temporarily prevent Specman Elite from generating 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. In order to continue testing other scenarios, you simply extend the test constraints to prevent Specman Elite from generating this combination.

Procedure

Here is the procedure for bypassing the JMPC bug.

1. Copy the *tutorial/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 instr {

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

};
'>
```

4. In the Main Specman window, select *Debug->Delete All Breakpoints* to exit debug mode.
5. Click the Reload button.
6. Click the Load button and load the *CPU_bypass.e* file.
7. Click on the Test button to launch the test.

   This time the test runs to completion.

Tutorial Summary

Congratulations! You have successfully completed the major steps required to verify a design with the Specman Elite verification system. Here's a review of what you have accomplished 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.
- Then you used the unique temporal constructs provided by Specman Elite to create a self-checking monitor that verifies protocol conformance.

- When the self-checking monitor revealed a bug, the GUI 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 result of using Specman Elite is a drastic reduction in verification time and resources.

A Setting up the Tutorial Environment

In order to set up the tutorial environment, you need a Specman Elite license. You can get one by sending 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 Specman Elite Files

Here's the procedure for downloading 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.

```
% 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-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
```

4. Change the format type to *binary.*

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

5. Get the Specman Elite software.

```
ftp> get install_specman<release_number>.sh
200 PORT command successful.
150 Opening BINARY mode data connection for
install_specman<release_number>.sh
226 Transfer complete…
ftp> get sn_rel<release_number>.main.tar.gz
200 PORT command successful.
150 Opening BINARY mode data connection for
sn_rel<release_number>.main.tar.gz…
226 Transfer complete…
ftp> get sn_rel<release_number>.<OS>.tar.gz
150 Opening BINARY mode data connection for
sn_rel<release_number>.solaris.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.

```
ftp> get se_tutor.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

Here's the procedure for setting the environment variables, installing the files, and starting up the license manager. Please refer to README file for installation updates.

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

   ```
   % rlogin solaris/hpux-machine
   ```

2. Run the installation script.

   ```
   % sh ./install_specman<release_number>.sh
   ```

3. When the Specman Elite Install Script Menu appears, select option 1, "Complete installation".

   After you have installed the machine-independent files (including the online docs) and the machine-dependent files, the script will ask you to select 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. Exit the installation script after the license handling procedure is completed.

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 &
   ```

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

Installing the Tutorial Files

Here's the procedure for installing 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_tutor.tar.gz
% tar -xvf se_tutor.tar
```

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

```
% ls *

gold:
CPU_bypass.e        CPU_instr.e         CPU_tst2.e
CPU_checker.e       CPU_misc.e          CPU_tst3.e
CPU_cover.e         CPU_refmodel.e      regression_3.2.ecov
CPU_drive.e         CPU_top.e
CPU_dut.e           CPU_tst1.e

src:
CPU_bypass.e        CPU_instr.e         CPU_tst2.e
CPU_checker.e       CPU_misc.e          CPU_tst3.e
CPU_cover.e         CPU_refmodel.e      regression_3.2.ecov
CPU_drive.e         CPU_top.e
CPU_dut.e           CPU_tst1.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 design 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

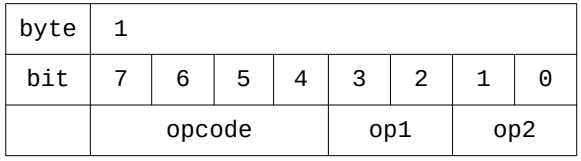


Figure B-2 Immediate Instruction

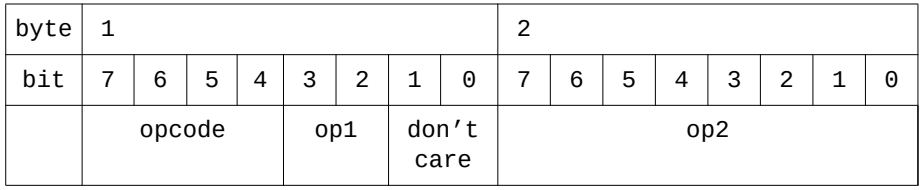


Table B-1 on page B-2 shows a summary description of the CPU instructions.

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 on page B-3.

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*, *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 on page B-4.

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