picoJava-II™ Verification Guide

THE NETWORK IS THE COMPUTER™

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303 USA
650 960-1300

Part No.: 914-1681-02
March 1999

Contents

Figures

Tables

Code Examples

Preface

This guide provides a comprehensive suite of tests and a simulation environment to use in verifying the functionality of the picoJava-II core.

This guide does not cover performance validation or the verification of circuits, physical design, timing, and the library.

Organization of This Book

This guide is divided into two parts, with an index at the end.

- *Part I: picoJava-II Verification Environment and Tools* **contains these chapters:**
  - Chapter 1, *Overview*
  - Chapter 2, *Verification Environment*
  - Chapter 5, *External Tools*
  - Chapter 3, *Test Environment*
  - Chapter 4, *Test Scripts*
  - Chapter 6, *Monitors*
- *Part II: picoJava-II Verification Tests* **contains these chapters:**
  - Chapter 7, *Verification of the Instruction Cache Unit (ICU)*
  - Chapter 8, *Verification of the Integer Unit (IU)*
  - Chapter 9, *Verification of the Floating Point Unit (FPU)*
  - Chapter 10, *Verification of the Data Cache Unit (DCU)*
  - Chapter 11, *Verification of the Stack Manager Unit (SMU)*
  - Chapter 12, *Traps and Interrupts*

Related Books and References

Three books form the documentation set for the picoJava-II release:

- *picoJava-II Programmer's Reference Manual*
- *picoJava-II Microarchitecture Guide*
- *picoJava-II Verification Guide* (this book)

The following publications are reference material for the subject matter of the documentation set:

- Lindholm, Tim, and Frank Yellin: *The Java Virtual Machine Specification, Addison-Wesley, ISBN 0-201-63452-X*
- *IEEE Standard Test Access Port and Boundary-Scan Architecture, ANSI/IEEE Std. 1149.1-1990.*
- *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std. 754-1985.*
- Ungar, David: *ACM SIGPLAN Notices*, 19(5):157-167: *Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm*, April 1984.
- Wilson P., and T. Moher: *ACM SIGPLAN Notices*, 24(10):23-35: *A Card-marking Scheme For Controlling Intergenerational References In Generation-based Garbage Collection On Stock Hardware*, 1989.
- Steele, Guy L.: *Communications of the ACM*, 18(9): *Multiprocessing Compactifying Garbage Collection*, September 1975.
- Hudson, R., and J. E. B. Moss: *Proceedings of International Workshop on Memory Management*: *Incremental Garbage Collection For Mature Objects*, St. Malo, France, September 16-18, 1992.

Typographic Conventions

TABLE P-1 describes the typographic conventions used in this book.

TABLE P-1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, instructions, files, and directories; on-screen computer output; email addresses; URLs	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% You have mail.`

`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name%` **`su`**
		`Password:`

AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename*.
AaBbCc123	Book titles, section titles in cross-references, new words or terms, or emphasized words	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.

Sun Documents

The SunDocs^{SM} program provides more than 250 manuals from Sun Microsystems, Inc. If you live in the United States, Canada, Europe, or Japan, you can purchase documentation sets or individual manuals using this program.

For a list of documents and how to order them, see the catalog section of the SunExpress™ Internet site at `http://www.sun.com/sunexpress`.

Sun Documentation Online

The `docs.sun.com` Web site enables you to access Sun® technical documentation online. You can browse the `docs.sun.com` archive or search for a specific book title or subject. The URL is `http://docs.sun.com/`.

Disclaimer

The information in this manual is subject to change and will be revised from time to time. For up-to-date information, contact your Sun representative.

Feedback

Email your comments on this book to: `picojava-doc@sun.com`.

Acknowledgment

Many people—picoJava-II licensees, engineers, programmers, marketers—contributed to this book. We thank them for their input, feedback, and support.

PART I picoJava-II Verification Environment and Tools

CHAPTER **1**

Overview

As processor design becomes increasingly complex, verification plays an important role in the design cycle. This chapter summarizes the verification process for the picoJava-II CPU core system at both the register transfer language (RTL) level and the gate level. The chapter includes these sections:

1.1 Methodology

The picoJava-II CPU core, a reusable core for a variety of Java™ processors, is a synthesizable RTL netlist and can be targeted to different libraries.

There are two levels of verification:

- Stand-alone testing (SAT)
- Full core-level testing, the methodology for which is cosimulation between the architecture simulator and the RTL netlist

In addition, you can synthesize the RTL netlist into a gate-level netlist to ensure that the design meets the timing budget. Usually, several iterations are necessary to shorten the longest timing paths to meet the goals.

Finally, gate-level simulation is recommended to ensure that the synthesis is correct.

1.2 Strategy

You can use directed tests and Verilog-based random events to achieve high confidence within a reasonable time frame.

1.2.1 Directed Tests

Directed tests covers the major blocks in the design: the Instruction Cache Unit (ICU), the Data Cache Unit (DCU), the Integer Unit (IU), the microcode, the Stack Manager Unit (SMU), the trap, the Floating Point Unit (FPU), the interrupt, the power management unit, and exception logic.

See the various chapters in this book for details of testing the major blocks.

1.2.2 Random Events

The tests use random events to test features, such as interrupts, interrupt priority, and powerdown mode. For details of the algorithm, see *External Interrupt Controller* on page 64.

1.3 Tools and Utilities

TABLE 1-1 lists the external tools and utilities available for verification.

TABLE 1-1 External Tools and Utilities for Verification

Tool or Utility	Purpose
Verilog-XL	RTL or gate simulation
VCS	RTL or gate simulation
SignalScan	RTL or gate debugging
VeriCov	Measurement of test coverage
Radify	Optimization of Verilog designs
Tool Control Language (Tcl)	Scripting language for random instruction testing

Additional tools and utilities developed at Sun are listed in TABLE 1-2.

TABLE 1-2 Tools and Utilities Developed at Sun

Tool or Utility	Purpose
Class loader	Loading test programs into simulation memory
Runtime disassembler	Disassembler for test programs
PJSIM	Instruction simulator
Cosimulation	Cross-checking of results between PJSIM and RTL
Java-related tools, such as Java virtual machine, Java Assembler (JASM), and Java Bytecode Disassembler (JDIS)	Assembler and disassembler for picoJava-II instructions

CHAPTER **2**

Verification Environment

This chapter describes the RTL (Register Transfer Level) simulation environment. It contains the following sections:

2.1 Simulation Environment

The picoJava-II core simulation environment is called DSV (Decaf System Verification).

2.1.1 picoJava-II Processor Models

The picoJava-II processor core can be simulated at two different levels of abstraction:

- A detailed RTL model which can be simulated using a Verilog language simulator, or
- A coarse-grained instruction-accurate simulator written in C.

The RTL model is used for implementing the design in silicon and verifying its functionality prior to tapeout, while the instruction-accurate simulator (`ias`) is used for software development and debugging and as a golden reference model for verifying the RTL design.

The picoJava-II distribution contains a system simulation environment to run and debug programs on both the `ias` and RTL models. The simulation environment, called DSV (Decaf System Verification), is embedded in these two models, and provides functionality such as simulating memory accesses, loading class files into memory, and controlling simulation.

The RTL model is invoked using the command `pj2vlog` or `pj2vcs` (depending on which Verilog simulator you use). The instruction-accurate model is invoked using the command `ias`.

Note – Parts of the picoJava-II documentation may refer to the instruction simulator as PJSIM rather than `ias`. Both names refer to the same program.

Note that the picoJava-II environment does not contain a software model which is "cycle-accurate." That is, the software does not model the processor exactly at cycle boundaries. Programs for which cycle-accurate information is desired must be simulated on the RTL, which is slow, but produces exact results.

2.1.2 Memory Model

picoJava-II main memory functionality is simulated using a C library (`libloader.a`). The picoJava-II RTL testbench includes a simple memory controller, which acts as a broker between requests made by the picoJava-II core and the C memory model. The memory controller calls the C library using PLI (Programming Language Interface) functions. PLI functions are defined in the file `tools/ldr/src/rw.c`. For descriptions of the PLI functions, see *Programming Language Interface (PLI)* on page 13.

The structure of the C main memory model is such that exactly the same library is linked into the instruction-accurate simulator. `ias` uses the same interface functions to access main memory as the RTL. However, `ias` and RTL have independent implementations for cache memories. `ias` implements a high-level cache simulator (implemented in `tools/cache/src/cache.c`) while RTL models detailed cache control and datapath logic, using behavioral models of memory megacells to simulate storage for data, tag and status RAMs in both caches.

Some locations in the picoJava-II memory address space have special meaning in the simulation environment (for both `ias` and the RTL model). FIGURE 2-1 and TABLE 2-1 for a list of these special addresses.

```
/* there are 2 valid memory areas: from 0 to CM_SIZE - 1 and from
   SCRATCH_START to SCRATCH_START+ SCRATCH+SIZE - 1   */
EXTERN unsigned int CM_SIZE;/* not hard-coded any more, but a
variable in cm.c, default 16M */

#define SCRATCH_START 0x2fff0000 /* scratch memory area start */
#define SCRATCH_SIZE 0x20000     /* size of scratch memory area */
#define SYSCALL_MAGIC_ADDR 0xffe8 /* byte address */
#define BAD_MEMORY_START 0x2fffbad0  /* accesses to bad area return
error ack’s */
#define BAD_IO_START 0x3000bad0
#define BAD_SIZE 16                  /* size of bad area */
#define CONSOLE_ADDR 0x2f0000c0      /* writes to console address
print a character */

/* ias profiler addresses */
#define PROFILER_CMD_ADDR      0xffc0
#define PROFILER_DATA_ADDR     0xffc4

/* regions in common memory */
#define RESET_ADDRESS 0x0
#define TRAP_ADDRESS  0x00010000    /* trap table */
#define CLASS_ADDRESS 0x00020000    /* class loading area */
#define HEAP_ADDRESS  0x00100000    /* heap start */
#define STACK_BASE     0x00400000   /* initial stack bottom area */

/* memory addresses with special meaning for simulation */
#define FLUSH_LOCATION             0xfff0
#define RESETCODE_USE_LOCATION     0xffe8
#define TRAP_USE_LOCATION          0xffec
#define WATERMARK_LOCATION         0xffe4
#define PERF_COUNTER_ADDRESS       0xffdc
#define COSIM_LOCATION             0xffe0
#define END_OF_SIM_LOCATION        0xfffc
#define INTR_ACK_REGISTER          0xfff8
#define STOP_RANDOM_INTERRUPT      0xfff4

/* locations to store CP info in case of class not found error */
#define ERROR_CP_LOCATION          0xffd0
#define ERROR_CPINDEX_LOCATION     0xffd4
```

FIGURE 2-1 Memory Map

TABLE 2-1 lists special addresses for simulation.

TABLE 2-1 Special Addresses for Simulation

Address	Description
0xfffc	Writes to this address to end the simulation.
0xfff8	Writes 0xacc to this address to acknowledge the interrupt controller and reset the `pj_irl` pins to 0.
0xfff4	Writes to this address to disable random `irl` or `nmi` interrupts.
0xfff0	Writes 0xfxxx to this address to invalidate the data and instruction caches before running the reset code.
	The class loader writes 0xfxx to this address to indicate that it has loaded a class that has a `clinit` method.
	Writes 0xfx to this address to use the counter version (instead of the abort version) of the exception handler.
	Writes 0xf to this address to flush the stack and data and instruction caches at the end of simulation.
0xffec	Writes 0xffffffff to this address to indicate to trap handlers that the system should use handles while allocating and referencing objects.
0xffe4	Writes 0x10 to this address to indicate to reset code that the low and high watermarks for the dribbler should be set to 48 and 56, respectively.
	Writes 0x1 to indicate to reset code that the low and high watermarks for the dribbler should be set to 8 and 16, respectively.
0xffe0	Writes to this address to enable full compare during cosimulation. (The default is small compare.)
0xffdc	Writes 0x1 to this address to enable gathering of performance statistics by the performance monitor.
	Writes 0x0 to this address to disable gathering of performance statistics by the performance monitor.
0xffd4	Writes to this address to store the constant pool value for a missing class, field, or method.
0xffd0	Writes to this address to store the index of a missing class, field, or method.
0xf000 to 0xf384	Writes to an address in this range to store the value of the exception counter.

2.1.3 Class Loader

Classes are loaded into picoJava-II memory using a class loader. (See the `.c` files in `tools/src/ldr/src`.) The class loader code is shared between the RTL model and `ias.` Again, the class loader functionality is part of `libloader.a`, and the RTL accesses it using PLI functions, while `ias` calls the class loader functions directly.

The picoJava-II environment currently supports only static class loading. That is, *all* classes that are required during a run must be loaded in memory before the program begins execution. Classes are loaded in the RTL model by specifying the `+class+`*classname*`.class` option and in `ias` using the `_loadClass_` command.

The class loader reads class files and converts the class file data (such as methods, fields and constant pool information) to an internal runtime representation. This runtime representation is compatible with that needed by picoJava-II instructions that expect a certain data layout (such as `invoke_*_quick` instructions and array access instructions). The layout expected by these instructions is described in the *picoJava-II Programmer's Reference Manual*. The detailed class layout at runtime is described as C structures in the file `tools/ldr/src/decaf.h`. You can change this structure, but if so you must ensure that the new structure is compatible with that expected by picoJava-II hardware instructions.

In addition to loading `.class` files, the class loader loads data from files called *classname*`.init` and *classname*`.binit`. This feature allows tests to specify initial values for certain memory addresses. For the expected format of the `.init` and `.binit` files, see the Web pages included with the picoJava-II distribution.

The class loader does not perform any verification on class files it loads into memory.

2.1.4 Trap Handling

At startup, the class loader in both `ias` and the RTL model loads all trap handlers by default, and sets up the trap table to point to the trap handlers.

Each trap handler is read in as a `.class` file. However, a class file can have instructions specific to the picoJava-II instruction set, which are not present in the Java virtual machine instruction set. The trap table entry is set to point to the code for a static method inside the class called *main*, which should take no arguments and return a void. The pointer to the constant pool structure for this class is placed at the word in the trap table immediately following the pointer to the trap handler code. Each trap table entry is 8 bytes long. The position of the trap table is assumed to be fixed at a certain address. Initialization code must set the trapbase register in the processor to this fixed address before any traps are generated. For details on the trapbase register and the trap table, see Chapter 7, Traps and Interrupts, in the *picoJava-II Programmer's Reference Manual*.

Trap handler class files have predefined names for each trap. Trap handlers are usually written in the picoJava-II assembly language. Sample implementations are provided with the picoJava-II simulation environment in the directories `design/traps` and `design/exception`.

Note – The trap handlers provided with the picoJava-II environment are examples only. You must customize them for your system.

2.1.5 Simulation Options

The `pj2vcs` and `pj2vlog` commands accept options for controlling the simulation. Some of these options are conveyed to the test program by storing special values in predefined locations in memory. The test program checks these predefined locations and takes the appropriate action. For example, to set low and high watermarks for the dribbler, you can use the `+minwm` and `+maxwm` flags. Using these flags creates an initialization in both `ias` and RTL memory at location 0xffe4 before the test program begins execution. When the default reset code begins executing, it reads memory location 0xffec and sets up the watermarks accordingly.

When a test is run only on `ias`, the flags must be set up manually or specified in the `.iasrc` file.

Other options specified to `pj2vcs` or `pj2vlog` control the behavior of the hardware simulation environment. For example, the `+int_random` switch causes external interrupts to be signalled to the processor at random. As another example, the `+rand_ack1` option controls the timing of memory controller acknowledgements to the CPU.

The complete list of flags accepted by the RTL model is described in the Web pages included with the picoJava-II distribution. Also, you can type `pj2vcs -usage` or `pj2vlog -usage` for a list of options.

2.1.6 Runtime Classes

A limited implementation of some of the basic runtime classes needed to run Java programs is provided with the picoJava-II distribution.

Note – Runtime classes are provided for the sole purpose of running picoJava-II tests. These classes are not meant to be a complete (or compatible) implementation of the runtime classes required by the Java platform.

The picoJava-II environment provides some basic operating system functionality such as memory allocation using a small kernel embedded in the emulation trap handlers. There is no support for multi-threading and garbage collection in this runtime environment.

2.2 RTL Verification

The RTL model is verified through self-checking tests, cosimulation, and RTL monitors.

2.2.1 Cosimulation

The primary means of verifying the picoJava-II RTL is by running test programs on the RTL model. Most test programs developed for the RTL model are self-checking. That is, they generate a result indicating success or failure. In addition to self-checking tests, cosimulation is a powerful way to verify the RTL design. In cosimulation, the RTL model is run in lock-step with the `ias` model running the same program, while comparing the architectural state between the `ias` and the RTL model after every instruction. This allows tests which are not self-checking to be used to verify the RTL model. Of course, self-checking tests can also be run on either the RTL model or on `ias` standalone.

The RTL model runs in cosimulation mode if you specify the `+cosim+ias` option to `pj2vlog` or `pj2vcs`. The RTL model then starts up a slave `ias` process. As instructions complete in the RTL pipeline, the RTL model sends commands to `ias` to step forward for the same number of instructions. (The RTL model can complete up to 4 instructions at a time, because it folds up to 4 instructions together. For information about folding, see the *picoJava-II Microarchitecture Guide*.) Along with every step instruction, the RTL model also sends `ias` its own register values and top-of-stack entries after those instructions have completed execution. After `ias` has stepped the same number of instructions as the RTL model, it compares the state sent across by the RTL model with its own state, and flags any mismatches. At the end of simulation, caches of both models are flushed and the contents are compared.

For answers to frequently-asked questions about cosimulation, see the Web pages included with the picoJava-II distribution.

2.2.2 RTL Monitors

Another way to verify the RTL design is by using the monitors included in the picoJava-II RTL simulation environment. Monitors are assertion checkers that run concurrently with the RTL simulation, and constantly check if properties which should be true in the system are indeed true. If there is an assertion failure, the monitor halts the RTL simulation with an error code.

You can also use the monitors to count the occurrences of certain events. These statistics can be used to analyze performance or to estimate power consumption. For information about the monitors, see Chapter 6, *Monitors*.

2.3 Running Simulations

The picoJava-II distribution provides a test suite containing subsuites that exhaustively test all functionality in the processor. A test is usually a program written in picoJava-II assembly language and assembled using the picoJava-II assembler, or a program written in the Java programming language, and compiled using a Java compiler. The tests may be run either on `ias` alone, or on the RTL model and `ias` together (in cosimulation mode).

To run all the tests in the verification suite, you need to perform the following basic steps:

1. **Build `ias`.**

   The distribution provides a default binary for SPARC/Solaris platforms.

2. **If you want to run the tests on RTL, build the RTL model.**

   This creates a binary called `pj2vcs` or `pj2vlog`.

3. **Compile the test suites you would like to run.**

   The test suites are provided in source form (either picoJava-II assembly language or the Java programming language). The source must be compiled to create `.class` files that can then be loaded in the simulators and run.

4. **Run the tests.**

   Run the tests on either the `ias` or the RTL model directly, or use the Steam script to run tests. Steam is a perl script that provides a flexible environment in which to run large batches of tests and report their results. For information about Steam, see Chapter 4, *Test Scripts*.

For more detailed information about running simulations, see the Web pages included with the picoJava-II distribution.

2.4 Programming Language Interface (PLI)

This section describes the routines for the PLI functions.

2.4.1 `$decaf_cm_load(`*classfile, needRes*`)`

The following function—

```
$decaf_cm_load(classfile, needRes)
reg[8*SIZE:0] classFile;
integer needRes;
```

—loads a class file and its superclasses (if required) into the class area in the common memory.

The arguments are:

- *classfile*: the name of either a class file in the current work directory or a DSV system class file
- *needRes*: not a valid argument for now

To load all the class files, call this function multiple times.

The class paths are the current working directory and `$DSVHOME/class`.

2.4.2 `$decaf_cm_read(`*address, data, size*`)`

The following function—

```
$decaf_cm_read(address, data, size)
reg[31:0] addr;
reg[31:0] data;
integer size;
```

—performs a common memory read from the specified address (*address*). It reads the data and puts them into the second argument (*data*). The size of the read is 1, 2, or 4 bytes depending on the size parameter being 0, 1, or 2, respectively.

Call this function when the picoJava-II core performs a memory read. This function extracts the address and updates the data bus according to the contents of memory at that address.

2.4.3 `$decaf_cm_write(`*address*, *data*, *size*`)`

The following function—

```
$decaf_cm_write(address, data, size)
reg[31:0] addr;
reg[31:0] data;
integer size;
```

—performs a common memory write from the specified address (*address*). It reads the data and puts them into the second argument (*data*). The size of the write is 1, 2, or 4 bytes depending on the size parameter being 0, 1, or 2, respectively.

Call this function when the picoJava-II core performs a memory write. This function extracts the address and data values and updates the contents of memory at that address.

2.4.4 `$decaf_cm_dump()`

The `$decaf_cm_dump()` function dumps the class structure in the common memory in `stdout`. No argument is necessary.

2.4.5 `$decaf_cm_direct_dump(`*address*, *count*, *filename*`)`

The following function—

```
$decaf_cm_direct_dump(address, count, filename)
reg[31:0] addr;
integer count;
string filename;
```

—dumps the common memory content from the address to the file as specified. The count is in bytes.

2.4.6 `$decaf_cm_load_method(`*classFile*, *index*, *location*`)`

The following function—

```
$decaf_cm_load_method(classFile, index, location)
string classFile;
integer index;
integer location;
```

—loads a method to the common memory.

The arguments are:

- *classFile*: the name of the class file
- *index*: the index of the method to be loaded
- *location*: the memory location where the method is stored

2.4.7 `$decaf_cosim(`*simulator*`)`

The following function—

```
$decaf_cosim(simulator)
string simulator;
```

—sets up the cosimulation environment by creating the instruction simulator process and connecting it with UNIX pipes.

The argument *simulator* is the standard plus argument with the leading word `+cosim`. The next plus precedes the executable name of the instruction simulator. For example, `+cosim+ias` means that the executable `ias` is the instruction simulator (PJSIM).

2.4.8 `$decaf_cosim_cntl(`*tclCmd*`)`

The following function—

```
$decaf_cosim_cntl(tclCmd)
string tclCmd;
```

—sends the Tcl commands as specified to the instruction simulator. The communication is synchronous; the function waits until it receives an acknowledgment. This function is used extensively for cosimulation.

2.4.9 `$decaf_disasm(`*clkCount*, *address*, *trpFlag*`)`

The following function—

```
$decaf_disasm(clkCount, address, trpFlag)
integer clkCount;
reg [31:0] address;
integer trpFlag;
```

—is the entry point for the real address disassembler.

The arguments are:

- *clkCount*: the current clock count
- *address*: the address of current pc
- *trpFlag*: the flag that indicates whether this is a trapped instruction

2.4.10 `$decaf_cosim_compare_memory_at_end()`

The `$decaf_cosim_compare_memory_at_end()` function performs a memory comparison from the memory transaction records. We usually call this function once at the end of simulation.

2.4.11 `$decaf_load_traphandlers()`

The `$decaf_load_traphandlers()` function loads all emulation traps and exceptions into the common memory. We usually call this function at the initial stage.

2.4.12 `$decaf_tam_start(`*simulator*`)`

The following function—

```
$decaf_tam_start(simulator)
string simulator;
```

—sets up the environment by creating the instruction simulator process and connecting it with UNIX pipes.

The argument *simulator* is the standard plus argument with the leading word `+tam`. The next plus precedes the executable name of the instruction simulator. For example, `+tam+tam` means that the executable `tam` is the instruction simulator.

2.4.13 `$decaf_tam_memread(`*address*, *data*, *size*`)`

The following function—

```
$decaf_tam_memread(address, data, size)
reg [31:0] address;
reg [31:0] data;
integer size;
```

—performs a memory read from the specified address (*address*). It reads the data and puts them into the second argument (*data*). The size is the power of 2, such as 0, 1, and 2.

2.4.14 `$decaf_tam_memwrite(`*address*, *data*, *size*`)`

The following function—

```
$decaf_tam_memwrite(address, data, size)
reg [31:0] address;
reg [31:0] data;
integer size;
```

—performs a common memory write to the specified address (*address*). It reads the data from the second argument (*data*). The size is the power of 2, such as 0, 1, and 2.

2.4.15 `$decaf_tam_exit()`

The `$decaf_tam_exit()` function directs the instruction simulator to exit.

2.4.16 `$decaf_tam_poll(`*address*, *data*, *size*, *result*`)`

The following function—

```
$decaf_tam_poll(address, data, size, result)
reg [31:0] address;
reg [31:0] data;
integer size;
reg [31:0] result;
```

—polls the instruction simulator for the necessary execution status and initiates a transaction cycle accordingly.

2.4.17 `$decaf_tam_intr(`*irl*`)`

The following function—

```
$decaf_tam_intr(irl)
integer irl;
```

—notifies the instruction simulator of an interrupt. The argument *irl* stands for interrupt level.

CHAPTER 3

Test Environment

This chapter describes the picoJava-II test environment. It contains the following sections:

- *RT Tests*
- *RC Tests* on page 24

There are two types of tests in the picoJava-II core verification environment: *RT* (reset test) and *RC* (run code) tests. RT tests run as raw reset code which the picoJava processor is simulated to execute at poweron. RC tests are usually larger tests that depend on standard startup code to run and to initialize the chip before they are invoked.

3.1 RT Tests

RT tests run as part of reset code. To make it easy to write RT tests, there is a standard template for the reset code (in `sim/include/reset.code`) which initializes chip registers, reports status at the end of the test, and so on. The template jumps to a label called *Main* and the test body must push the status code on the stack and jump back to a label called *Check*.

If you need custom initialization to write an RT test, you can either write the complete reset code and assemble it using the JASM assembler, or modify the template and use that template in your test. Test cases written to use the template usually have a `.code` extension.

A set of scripts converts the `.code` file (after concatenating the template, expanding macros, and so on) to a `.jasm` file, which is compiled by the JASM assembler to a `.class` file. For a description of the reset process, see *Reset Process* on page 25.

Examples of `.code` test cases are available under `sim/test/pico_vts/basic`. Standard makefiles in these directories have rules to convert the `.code` files to `.class` files, which can then be run on the software simulator (`ias`) or the RTL model.

A class file compiled from an RT test must be present as `reset.class` in the directory where the test is run. If you use the Steam script, this is automatically done for you. For information about Steam, see Section 4.1, *Steam Script Overview*, on page 27.

3.1.1 Examples

This section provides examples of a test with custom reset code, and a test that uses the template.

RT Test with Custom Reset Code

1. **Create a test called** `my_test.jasm` **that includes all the functionality required to run as reset code.**

   That is, it includes the initialization for the machine registers, enables the dribbler, and so on.

2. **Create** `my_test.class`

   ```
   % jasm my_test.jasm
   ```

3. **Copy** `my_test.class` **to** `reset.class`**.**

   ```
   % cp my_test.class reset.class
   ```

4. **Run** `ias`**. This loads** `reset.class` **at address 0, as reset code.**

   ```
   % ias
   ```

   Note that the simulation environment (for both RTL and `ias`) loads `reset.class` from the current directory or if not there, from `$DSVHOME/class`. For information about class loading, see Section 2.1.3, *Class Loader*, on page 9.

RT Test Using the Template

You can write a test called `my_test.code` which includes only the functionality required by your test, but the test still has to run as part of reset code. In this case, you must provide a label called Main in your test, and jump back at the end of the test to a label called Check. The top-of-stack element when you jump back must be an integer signifying whether the test passed or not. (A 0 value indicates success; a non-0 value indicates failure.) The MakeTest script converts the `.code` file by expanding global macros and concatenating the reset template into a `.class` file. You need to have perl installed in order to run Maketest.

You can follow these steps:

```
% mkdir class
% setenv TEST_ENV_INCLUDE sim/include
% cd class
% Maketest my_test $TEST_ENV_INCLUDE/macro.inc $TEST_ENV_INCLUDE/
reset.code
% cp my_test.class reset.class
% ias (loads up my_test.class as reset.class)
```

In most test suite directories, these actions are automatically performed by the Makefile in the directory when you build the tests. For information on how to compile tests, see the online documentation.

3.1.2 Standard Files

TABLE 3-1 describes the standard files for the test environment:

TABLE 3-1 Test Environment Standard Files

Name	Description
/sim/include/MakeTest	This script merges the reset code with the test code. It also interprets and executes any compilation directives and processes the macro include files.
/sim/include/macro.inc	This file contains the generic M4 macro definitions and is the global macro library that contains macros that can be used by all test cases. An example is IPush(), which pushes a 32-bit integer value onto the stack. This file provides a set of macros for uniformity and a better understanding of the tests. Macros improve readability and speed up test development.
/sim/include/reset.code	This file contains the skeleton of the reset code that initializes the registers and starts the test, as well as code that writes to the end location to end the test. A marker in the reset code acts as the insertion point for the user-defined test code. Default register definitions at the top of reset code are based on the values in the basic tests. This way, even if you do not define your own register values, your test still compiles.
/config/Makefile	This makefile controls class file builds from the .code, .java, or .jasm files. The source files reside in the Makefile.in file in each directory. If the test case is a .code file, it executes MakeTest to build the class file; otherwise, it executes jasm or javac for the build.
test_config.inc[1]	This file lists the test-specific macros that add to or override the macros in macro.inc. It should be in each test directory where there are .code files.
Makefile.in[1]	This file lists the files that make builds. It can override the details in the generic makefile.

[1]Create this file in your test directory. All other files in this table are in the default setup.

3.1.3 Compilation Directives

Verification engineers write tests in a variety of ways. Some use `cpp`; others use `m4`, shell, or Perl scripts; still others create C or C++ programs. To accommodate these preferences, compilation directives are provided.

By embedding the following constructs in your test or macro files, you can control the processing of your tests or macros and the compilation of your tests into a `.jasm` file:

- `.compile_single` *command* — Applies *command* to the current input file, but not to other input files. Merging occurs after execution of this command on the current input file.
- `.compile_single_2` *command* — Applies *command*, but not to the current input file. This command can be useful for combining the result of an external event with the input stream.
- `.compile_all` *command* — Applies *command* to all the input files. This command is executed after merging is complete.
- `.no_merge` — Does not perform the merge phase.
- `.no_jasm` — Does not compile to a `.class` file but leaves as a `.jasm` file when compilation is complete.
- `.no_concat` — Does not concatenate `macro.inc`, `test_config.inc`, and the reset code. With this command, you must concatenate the files manually.
- `.no_global_compile` — Does not execute any of the `.compile_all` commands.
- `.inject_main [...]` — Inserts the code inside the square brackets at the beginning of the `main` method when a `.push_injection` occurs. Using multiple `.inject_main` directives may cause errors.
- `.pull_injection` — Extracts the code in the `.inject_main` directive from the input stream.
- `.push_injection` — Pushes the code that was previously extracted back into the input stream at the beginning of the `main` method.

Note – All of the compilation directives must be the first characters of a line. You can use as many directives as you desire; however, using multiple `.inject_main` directives results in undefined behavior.

Currently, the `.compile_all cpp -B -C -P` and the `.compile_all m4` command lines in `macro.inc` ensure that `m4` filters all the files to pick up and use the `m4` macros. Only `compile_all` causes all other files to recognize the macros that are defined in any one file.

You can add command-line arguments to the directives. For example, to run `cpp` with the `-B` and `-C` options, include a `.compile_single cpp -B -C` or `.compile_all cpp -B -C` line in your code.

The compiler executes commands in the following manner:

- From top to bottom. Therefore, to pass your file through multiple filters, keep in mind this order.
- According to your current path. Thus, to run a version of `m4` independent of your current path setting, you must provide the full path name of that `m4` executable.

3.1.4 `make` Command

For RT tests, the `make` command performs the following steps:

1. **Extract directives for and execute all `.compile_single` and `.compile_single_2` directives on the files: `macro.inc`, `test_config.inc`, and `reset.code`, as well as on your test case (`.code`).**
2. **Concatenate the `macro.inc`, `test_config.inc`, and `reset.code` files (in that order).**

   To skip this step, specify `.no_concat`.
3. **Merge in your test case at the appropriate point in the reset code.**

   To skip this step, specify `.no_merge`.
4. **Execute all `.compile_all` directives on the new file and create a `.jasm` file.**

   To skip this step, specify `.no_global_compile`.
5. **Assemble the `.jasm` file and create a `.class` file.**

   To skip this step, specify `.no_jasm`.

3.2 RC Tests

RC tests have a main program. They use a standard reset code, which calls `clinit` methods for all classes loaded, and then jumps to the first static method in any class called `main` that has the signature `main ()I`. To write an RC test, you can write a standard Java program (with the main method signature as described) and compile it with a regular Java compiler such as `javac`.

3.2.1 Memory Map

Refer to FIGURE 2-1 on page 7 and TABLE 2-1 on page 8 for the memory map and simulation flags used by the picoJava-II core verification environment.

3.2.2 Reset Process

The reset handler performs the following steps:

1. **Load 0x003FFFFC to the** `VARS` **register.**
2. **Load 0x003FFFFC to the** `SC_BOTTOM` **register.**
3. **Load 0x003FFFFC to the** `OPTOP` **register**
4. **Load 0x0 to the** `FRAME` **register.**
5. **Load 0x0 to the** `CONST_POOL` **register.**
6. **Load 0x0 to the** `LOCKCOUNT0` **register.**
7. **Load 0x0 to the** `LOCKCOUNT1` **register.**
8. **Load 0x0 to the** `LOCKADDR0` **register.**
9. **Load 0x0 to the** `LOCKADDR1` **register.**
10. **Load 0x0 to the** `BRK1A` **register.**
11. **Load 0x0 to the** `BRK2A` **register.**
12. **Load 0x0 to the** `BRK12C` **register.**
13. **Load 0x0 to the** `GLOBAL0` **register.**
14. **Load 0x0 to the** `GLOBAL1` **register.**
15. **Load 0x0 to the** `GLOBAL2` **register.**
16. **Load 0x0 to the** `GLOBAL3` **register.**
17. **Load 0x00300000 to the** `OPLIM` **register.**
18. **Load 0x00010000 to the** `TRAPBASE` **register.**
19. **Load 0xFFFF0000 to the** `USERRANGE1` **register.**
20. **Load 0xFFFF0000 to the** `USERRANGE2` **register.**
21. *Optional*. **Invalidate the instruction and data caches, using diagnostic writes.**

22. **Set up the `PSR`, with optional watermarks if specified.**
23. **Initialize the heap structure.**
24. **Search for and invoke all `clinit` methods.**
25. **Search for the `main` method.**
26. **Turn on the performance monitor.**
27. **Invoke the `main` method.**
28. **Turn off the performance monitor.**
29. **Disable the `NMI` and `IRL` interrupts.**
30. *Optional*. **Flush the stack and data caches.**
31. **Check the return value from `main` and assign a pass or fail to the test.**

CHAPTER **4**

Test Scripts

This chapter describes the following test scripts:

- The Standardized Test Executor and Monitor (Steam) script, which executes test cases for the picoJava-II simulation environment
- The `gen_sst_control` script, which parses log files from simulation runs

Steam runs an individual test case or a series of test cases in predetermined behavioral environments. You can automate certain pretest, midtest, and posttest functions, as well as control how test results are generated and displayed.

This chapter contains the following sections:

4.1 Steam Script Overview

Steam is a perl script for running tests in the picoJava-II environment. Steam can run tests in several modes: on the instruction simulator (ias) only; on the RTL model only; or in cosimulation mode, where both `ias` and RTL run the same program in lock-step, and the cosimulation environment compares the processor state between `ias` and RTL after every instruction.

Steam provides a flexible environment for running tests and reporting results. Individual tests can customize the environment and options used to run them, using control files for each test suite. The control file for a test suite contains information Steam needs to run the test suite, such as: the names of the tests in the suite, any supporting classes that need to be loaded, cosimulation flags for a given test, and so on. For more information about suite control files, see *Suite Control File* on page 33.

4.1.1 Syntax

The general syntax of the `steam` command is:

`steam` *options tests*

where *options* represents a list of control arguments, operands, and/or cosimulation arguments; and *tests* represents a list of test names and/or directories containing tests. You can specify test cases, directory names, and options in any order. Steam executes tests according to their order (left to right) in the command line.

For a list of control arguments, see Section 4.2.3, *Control Arguments*, on page 36. For a list of operands, see Section 4.2.4, *Operands*, on page 40. For a list of cosimulation arguments, see Section 4.2.5, *Cosimulation Arguments*, on page 41.

Listing Steam Options

To see a list of options, enter the following command:

```
steam -help
```

4.1.2 Before Running Steam

Before using Steam to run tests, you need to :

- Set environment variables
- Check for syntax errors (optional)
- Customize (optional)

Setting Environment Variables

Before using Steam, you need to set environment variables as follows:

```
setenv PICOJAVAHOME <directory where you untar'd the distribution>
setenv PROJECT picoJava-II
setenv DSVHOME $PICOJAVAHOME/$PROJECT/tools
setenv CLASSPATH ${DSVHOME}:.
setenv STEAM_PLUSARGS $PICOJAVAHOME/$PROJECT/sim/config
setenv VFILES_PATH $PICOJAVAHOME/$PROJECT/sim/env/vfiles
setenv VLOG_NAME pj2vlog
setenv VCS_NAME pj2vcs
setenv PATH $DSVHOME/bin:$PATH
```

For more information about environment variables used by Steam, see Section 4.2.9, *Environment Variables*, on page 49.

Checking Syntax

Before executing any tests, check for syntax errors by running Steam using the `-n` option. When you use the `-n` option, Steam reports any syntax problems but does not run tests or change any files.

Customizing

Optionally, you may customize the simulation in one of the following ways:

- Edit the following files:
  - `pj2vlog` program and `steam.plusargs` file
  - `pj2vcs` program and `steam.plusargs` file
  - Suite control file

or

- Specify options at the command line to override the settings in the suite control file

The control arguments, operands, cosimulation arguments, suite control file and `steam.plusargs` file are described later in this chapter.

4.1.3 Steam Examples

This section provides the following examples:

- Running a single test
- Running `ias` only
- Running a suite of tests
- Using a non-default suite control file

Running a Single Test

To use Steam to run a single test, complete the following steps:

1. **Change directory to the directory containing the suite control file.**

   For example:

```
cd $PICOJAVAHOME/$PROJECT/sim/test/pico_vts/basic_java
```

2. **Run Steam.**

   For example:

   ```
   steam allinst -flush -outdir test1dir
   ```

   This example runs the test case `allinst` with the `-flush` cosimulation option and places the results in the `test1dir` directory.

Running Tests on `ias` Only

By default, Steam runs tests in cosimulation mode. To use Steam to run tests on `ias` only, use the `-ias` option as in the following example.

1. **Change directory to the directory containing the suite control file.**

   For example:

   ```
   cd $PICOJAVAHOME/$PROJECT/sim/test/pico_vts/basic_java
   ```

2. **Run Steam.**

   For example:

   ```
   steam allinst -ias -outdir test1dir
   ```

   This example runs the test case `allinst` on `ias` only and places the results in the `test1dir` directory.

Running a Suite of Tests

To use Steam to run a suite of tests, complete the following steps:

1. **Change directory to the directory containing the suite control file.**

   For example:

   ```
   cd $PICOJAVAHOME/$PROJECT/sim/test/pico_vts/basic_java
   ```

2. **Run Steam.**

   For example:

   ```
   steam . allinst -flush -outdir test1dir
   ```

   This example runs all tests in the `pico_vts/basic_java` directory with the `-flush` option and places the results in the `test1dir` directory.

Using a Non-Default Suite Control File

To use Steam to run tests using a non-default suite control file, complete the following steps:

1. **Copy and edit an existing suite control file.**
2. **Change directory to the directory containing the suite control file.**

   For example:

   ```
   cd $PICOJAVAHOME/$PROJECT/sim/test/pico_vts/basic_java
   ```

3. **Run Steam.**

   For example:

   ```
   steam . -scf new.control -outdir test1dir
   ```

   This example runs all tests in the `pico_vts/basic_java` directory using the `new.control` suite control file. The results are placed in the `test1dir` directory.

4.2 Steam Operation

This section provides more detailed information about using the Steam script. It contains the following sections:

- *Execution and Process* on page 31
- *Suite Control File* on page 33
- *Control Arguments* on page 36
- *Operands* on page 40
- *Cosimulation Arguments* on page 41
- *Test Coverage and Statistics* on page 43
- *Log Files* on page 48
- *Timeout Control* on page 48
- *Environment Variables* on page 49
- *Exit Status* on page 50

4.2.1 Execution and Process

Steam execution consists of three phases:

- Pretest (the build-and-copy phase)
- Midtest (the run phase)
- Posttest (the check phase)

Note – To specify one or a combination of phases for Steam, use the following options as appropriate: **`-build`**, **`-run`**, or **`-check_only`**.

Pretest (Build-and-Copy Phase)

Prior to executing tests, Steam sets up the conditions, as follows:

1. **Parses the command line and prints errors or warnings, if any.**

   In case of errors, Steam halts.

2. **Parses the suite control files and stores the information for execution.**

   In case of errors, Steam halts.

3. **Copies the class files and any additional files for the test to the current directory or the output directory if one is in the command line.**

   If it cannot find one or more files, Steam posts an error and halts.

4. *Optional*. **Builds Verilog Chronologic Simulation (VCS) if that option is in the command line.**

5. *Optional*. **Builds the test cases if the build options are in the command line.**

   If the build fails, Steam does not run those test cases.

6. **Assembles the commands to run the tests.**

7. *Optional*. **Creates `runme` and `.iasrc` files if the relevant options are in the command line.**

Midtest (Run Phase)

Steam executes tests in the midtest phase, as follows:

1. **Obtains the current timestamp.**

2. **If Steam has permission to utilize the Dynamic Resource Allocation Manager (DReAM) environment, it submits the jobs.**

   While running in DReAM, if the `-exit` option is in the command line, Steam terminates the script to allow the jobs it has submitted to continue their execution in the background. Otherwise, it polls or executes all the jobs.

   If Steam is not running in DReAM, it submits only the first job and obtains the current timestamp. It repeats this step until it completes all the jobs.

3. **Obtains the final timestamp.**

Posttest (Check Phase)

After completion of all test cases, some postprocessing may be necessary. In the posttest phase, Steam collates information according to the command line and removes unnecessary or temporary information, as follows:

1. **Streamlines the test directories according to the posttest control options in the command line:** `-keepdirs`**,** `-nokeepdirs`**, or** `-keepfailures`**.**
2. **Collects statistics for the passes, failures, and abnormal terminations.**
3. **Creates and formats a summary report according to the command line.**

   In the absence of a summary string in the command line, Steam uses a default format.
4. **Creates three files:** `list.notrun`**,** `list.failure`**, and** `list.passfile`**, which contain the test cases that were not run, failed, or passed, respectively.**

4.2.2 Suite Control File

Each test suite must contain a suite control file, called `suite.control`, which contains:

- A list of all the subsuites — All the subdirectories, under the current directory, which contain test cases
- A list of all the test parameters that Steam applies to the test cases
- A list of the immediate test cases under the current directory (*not* the test cases of the subsuites)
- The parameters and extraneous files that are required for the test cases

FIGURE 4-1 and FIGURE 4-2 illustrate two examples of a suite control file.

```
# sample suite.control file for iu tests
suite
  tags |= listA listB
  subsuites = microcode hazard boundary fold
  subsuites = other_tests
```

FIGURE 4-1 An Example of a Suite Control File

The file shown in FIGURE 4-1 describes the test suites in the directed tests for the Integer Unit (IU). It assigns all the tests in its hierarchy to be elements of the tags, `listA` and `listB`, and enumerates the list of subsuite directories underneath the current suite.

Note – Place only the word `suite` and the test case names in column 1 of the suite control file. Indent all other lines with one or more white space characters.

```
# sample suite.control file for directed/iu/back2back
suite
  files |= class/$this.class
    tags |= back2back
    flags |= -rt
  imul_b2b
  store_load_b2b
  zero_line_b2b
  cache_flush_b2b
  cache_index_flush_b2b
  putstatic_quick_b2b
    flags = -rc -flush
  putstatic2_quick_b2b
    flags = -rc -flush
  getstatic_quick_b2b
    flags = -rc -flush
  getstatic2_quick_b2b
    flags = -rc -flush
  load_word_b2b
```

FIGURE 4-2 Another Example of a Suite Control File

The suite control file shown in FIGURE 4-2 contains the suite specification for the `sim/test/pico_vts/directed/iu/back2back` directory. The suite-specific items appear first, followed by the test cases.

The operator `|=` denotes that the current field is extended from its parents and inherits attributes. The operator = denotes that no inheritance applies.

The variable `$runtime` expands to `$DSVHOME/class`. You can preserve the subdirectory hierarchy when Steam copies files by enclosing in square brackets the part of the hierarchy to be preserved. For example:

```
files |= $runtime|[java/lang]/foo.class
```

copies the file `foo.class` from `$DSVHOME/class/java/lang` to a subdirectory `java/lang` from the current directory.

Keywords

TABLE 4-1 lists and defines the suite control file keywords.

TABLE 4-1 Suite Control File Keywords

Keyword	Definition
`tags`	The set of tags to which the test suites or test cases are bound, that is, how to configure the test lists for use by the `-setlist` option.
`subsuites`	The set of directories in the current directory that contain additional tests. These directories must contain `suite.control` files. Steam does not allow subsuite inheritance via the `|=` operator.
`flags`	The set of command-line options you should apply for a default execution of the test case. You can add both test script control arguments and cosimulation arguments.
`files`	The list of files that this test requires. These files should be in the current directory and form the complete list of files that test execution needs. If a file contains the extension `.class`, Steam assumes it to be an additional class file for the test.
`suite-prerun-command`	An optional command that you can run before executing the test suite. You cannot apply subsuite inheritance (via the `|=` operator) for this directive.
`suite-postrun-command`	An optional command that you can run after executing the test suite. You cannot apply subsuite inheritance (via the `|=` operator) for this directive.
`test-prerun-command`	An optional command that you can run before executing the test case. You cannot apply subsuite inheritance (via the `|=` operator) for this directive.
`test-postrun-command`	An optional command that you can run after executing the test case. You cannot apply subsuite inheritance (via the `|=` operator) for this directive.

4.2.3 Control Arguments

TABLE 4-2 lists and defines the Steam control arguments that may be specified either at the command line, or with the `flags` keyword in the suite control file.

TABLE 4-2 Control Arguments

Option	Description
Test type	
`-rt`	Considers the test case independent reset code. You cannot use it with `-rc`.
`-rc`	Considers the test case a class file that requires external reset code. You cannot use it with `-rt`.
DReAM control	
`-nodream`	Runs the test cases without dispatching jobs to DReAM. This is the default. You cannot use this option with `-dream`.
`-dream`	Runs the test cases by dispatching jobs to DReAM. You cannot use it with `-nodream`.
Cosimulation control	
`-ias`	Runs test cases in IAS (`pj_sim`) standalone mode and hence does not exercise the RTL model. You cannot be use this option with `-rtl` or `-cosim`.
`-pjsim`	Same as `-ias`.
`-rtl`	Runs test cases in RTL only and exercises the Verilog model directly. Does not run any tests with the instruction simulator (IAS or `pj_sim`). You cannot use this option with `-ias`, `-pjsim`, or `-cosim`.
`-cosim`	Runs test cases in the cosimulation environment, with IAS (`pj_sim`) and RTL models executing simultaneously. This is the default. You cannot use it simultaneously with `-ias`, `-pjsim`, or `-rtl`.
RTL model execution control	
`-vlog`	Runs test cases with Verilog XL instead of VCS. You cannot use this option with `-vcs`. Steam ignores this option if you specify `-ias` or `-pjsim`.
`-vcs`	Runs test cases with VCS instead of Verilog XL. This is the default. You cannot use it with `-vlog`. Steam ignores this option if you specify `-ias` or `-pjsim`.

TABLE 4-2 Control Arguments

Option	Description
DReAM job watch control	
`-poll`	Checks job completion at regular intervals. Once the command verifies that the jobs are complete, Steam displays the results and then terminates. Steam ignores this option if you do not run jobs in DReAM mode.
`-exit`	Exits immediately after dispatching jobs. Steam ignores this option if you do not run jobs in DReAM mode.
`-wait`	Executes `drmwait`(1) to wait for all jobs to complete before exiting. This is the default. Steam ignores it if you do not run jobs in DReAM mode.
Suite control file usage	
`-scf` *filename*	Uses a suite control file (*filename*) to control the behavior of running tests.
Configuration file	
`-config` *config_file*	Uses *config_file* as the configuration file for additional class files to be loaded at runtime. See *Operands* on page 40 for the function and behavior of *config_file*.
Test frequency control	
`-once`	Runs the test once only. You cannot be use this option with `-twice`.
`-twice`	Runs the test once. If it fails, runs the test again with the `TPL_DBG` cosimulation option. Exits if the test passes. This is the default. You cannot use this option with `-once`.
Second run	
`-dump_before` *num*	During the second run, generates the `sst` file *num* cycles before the first error or mismatch occurs.
`-dump_after` *num*	During the second run, generates the `sst` file *num* cycles after the first error or mismatch occurs.
`-no_second_sst`	Reruns the failed test but does not generate an `sst` file on the second run.
Error control	
`-nommcnt`	Does not count the number of register and memory mismatches between RTL and IAS (`pjsim`) during simulation. This option serves as a summary option only. Steam ignores this option if the cosimulation mode is not in use.

TABLE 4-2 Control Arguments

Option	Description
`-noerrcnt`	Does not count the number of RTL monitor error messages that Steam produces during test execution. This option serves as a summary option only; Steam ignores it if you use `-ias` or `-pjsim`.
Test result log	
`-logfilename` *log_name*	Assigns each test case the same log file name (*log_name).*The default log file name is *testcase*`.log`, where *testcase* is the name of the test up to the first + symbol in the name.
Pretest control	
`-build`	Builds the test cases before executing them by running `make` in the test directories.
`-buildvcs`	Builds a new version of VCS before executing tests. You can still use this option even if you do not run the tests with VCS, but it becomes moot in that case.
`-build_only`	Builds the test cases but does not execute them.
`-runme`	Generates `runme` files, which are short shell scripts that contain the invocation to `decafvcs` or `decafvlog`. You can run the `runme` scripts later for a quick reexecution of the test case.
`-no-rtime-classes`	Does not load Java runtime classes prior to executing the test (for rc tests only).
Posttest control	
`-keepdirs`	Preserves the test case, log file, and any other test-specific information in a run directory for each test case. This is the default. You cannot use it with `-nokeepdirs` or `-keepfailures`.
`-nokeepdirs`	Does not preserve run directories; instead, this option copies the log files back to the current directory upon completing the test, then removes the run directories. You cannot run this option with `-keepdirs` or `-keepfailures` or when you run multiple tests with `-logfilename` or `TPL_DBG`.
`-keepfailures`	Preserves run directories for only the test cases that are failing. If the test passes, this option copies the log file back to the current directory. You cannot use it with `-keepdirs` or `-nokeepdirs`.
`-merge_monitor_coverage`	Browses the test log files for lines that begin with `COVERAGE` and merges the results. See *Merging of Statistical Results* on page 44 for more information.

TABLE 4-2 Control Arguments

Option	Description
`-no_merge_rtl_stats`	Browses the test log files for lines that begin with `RTL -STAT` and merges the results. See *Merging of Statistical Results* on page 44 for more information.
Summary reporting of test results	
`-summary`	Produces a summary of the runs upon completion of test cases. You cannot use it with `-nosummary`.
`-nosummary`	Does not produce a final summary report. You cannot use it with `-summary`.
`-format_string` *string*	Customizes the summary report by printing it in the *string* format. See *Customization of the Summary Report* on page 45 for more information.
Timeout control	
`-timeout` *time_limit*	Aborts the test run if the total elapsed time exceeds *time_limit*. See *Operands* on page 40 for the format of *time_limit*.
Selective test execution control	
`-setlist` *set*	Runs only tests that fall within *set*. See *Operands* on page 40 for a description of this option and its format.
Debugging	
`-debug0`	Prints some debugging messages while Steam is running.
`-debug1`	Prints verbose debugging messages while Steam is running.
`-debug2`	Prints temporary debugging messages.
Miscellaneous	
`-jobc` *constraint_file*	Submits DReAM jobs with *constraint_file*. See *Operands* on page 40 for a description of the behavior of this option.
`-jobh` *dir_name*	Submits DReAM jobs with *path*, which is the directory name for constraint files. The default is `sim/env`.
`-n`	Parses the command line for correct syntax and reports any errors, discrepancies, or missing files. This option does not run tests or change files.
`-M` *mailing_list*	Emails the addressees (*mailing_list*) on completion of the test. If Steam generates a summary report, it becomes part of the email.
`-outdir` *dir_name*	Writes all output files to *dir_name* and collects the test cases, log files, and other data from a point rooted at the current directory.

TABLE 4-2 Control Arguments

Option	Description
`-q`	Runs in quiet mode and does not generate output messages, except for the summary report. To *not* produce this report, add the `-nosummary` option.
`-gate`	Uses the gate-level simulation paradigm instead of the RTL simulation paradigm.
`-vcov`	Generates VeriCov coverage results for the test cases as they are run.
`-help`	Displays information on how to use Steam.
`-h`	Same as `-help`.

4.2.4 Operands

TABLE 4-3 lists and defines the Steam operands that may be given at the command line.

TABLE 4-3 Operands

Option	Description
config_file	A file that lists the classes Steam loads, one class per line, each preceded by a + symbol and without the `.class` extension. For an example, see `sim/test/pico_vts/directed/iu/microcode/test_config.txt`.
log_name	The name of the file that Steam generates as the output of a cosimulation, RTL-standalone, or IAS-standalone run. Steam produces one such log file for each test case that it executes.
format_string	The string that describes how to print and align the rows and columns of the output summary report. The syntax and implementation of *format_string* are as yet not determined.
time_limit	A string of the format `h:mm` or `hh:mm`, which indicates in hours and minutes the maximum permissible time for test execution. The time limit begins when execution or job dispatch starts. If the time limit is exceeded, Steam terminates the tests and posts a notice in the summary.
set	Any combination of the `set` names that are available for one or more test cases. Steam determines the `set` names by parsing the test suite control file and allows any legal combination of the sets. `set` operators include~~:~~ + (union), * (intersection), ! (complement), and - (difference), which have standard meanings from the `set` theory.

TABLE 4-3 Operands

Option	Description
mailing_list	The list of email addresses or aliases to which to send the test run results. To list multiple names or aliases, enclose the entire list in quotation marks so that the shell does not consider each name as a separate argument to the script.
dir_name	The name of a directory, which can be either relative or absolute.
num	Any nonnegative integer.
string	Any string that is enclosed in either single or double quotation marks on the command line.
constraint_file	Any text file that DReAM can interpret as a constraint file. This is the name of a file, not a path.
filename	The name of a text file, not a path.

4.2.5 Cosimulation Arguments

Steam can be given arguments to use in running a simulation. In cosimulation or RTL mode, these options are passed along to the `pj2vlog` or `pj2vcs` command. All these options must be defined in the `steam.plusargs` file described in the following section. If they are not present in the `steam.plusargs` file, they are ignored.

Cosimulation arguments may be specified at the command line, or in a suite control file with the `flags` keyword.

All cosimulation arguments have an equal and opposite argument: `no_`*option*. You can use the opposite arguments at the command line to override the behavior that the suite control files specify.

Because suite control files may employ inheritance, it is a good practice to check for inherited options and/or negate any unwanted arguments by specifying `-no_`*option* at the command line.

`steam.plusargs` File

The `steam.plusargs` file determines which cosimulation arguments Steam recognizes. It contains the list of all valid arguments. Arguments that are not in this file are ignored.

TABLE 4-4 lists the arguments in the default `steam.plusargs` file.

TABLE 4-4 Arguments in the Default `steam.plusargs` file

Option	Description
`-boot8`	Enables the 8-bit boot mode.
`-nofpu`	Runs without the Floating Point Unit (FPU).
`-nosmu`	Runs without the Stack Manager Unit (SMU).
`-flush`	Flushes the stack and data caches after completing the test.
`-expcount`	Causes the exception handlers to increment a counter and return instead of aborting the test.
`-cacheinvalidate`	Invalidates all cache entries at the beginning of the test.
`-handle`	Runs the test with enabled handles. For details, see the *picoJava-II Microarchitecture Guide.*
`-minwm`	Runs with dribbler watermark settings at 8 and 16, the smallest legal values.
`-maxwm`	Runs with dribbler watermark settings at 48 and 56, the largest legal values.
`-pj_halt`	Forces the `pj_halt` signal to high for the duration of the test.
`-sst_control`	Creates a `signalscan` dump file for a set of clock-cycle ranges. Before you use this argument, create a file named `sst_control` in the current directory that contains clock-cycle values in hexadecimal, one value per line.
`-single_step`	Tests instruction single stepping, that is, disables the clock for 100 cycles.
`-fldg_mon`	Enables the instruction folding monitor.
`-int_random`	Enables random interrupts (`nmi`, `irl1`,...,`irl15`) and schedules them at random intervals.
`-int_cntl`	Enables a preset pattern of interrupts to test interrupt nesting and priority.
`-rand_ack1`, `-rand_ack2`	Uses a seed when initializing the random memory control acknowledgment logic.
`-s1`, `-s2`, `-s3`, `-s4`, `-s5`, `-s6`	Sets one of six random seeds to be used by the random memory control generator.
`-hold_seed_2`, `-hold_seed_3`	Sets one of two random seeds to initialize the random SMU hold generator.
`-no_ucode_mon`	Disables the microcode monitor.
`-ibuf_mon`	Enables the instruction buffer monitor.

TABLE 4-4 Arguments in the Default `steam.plusargs` file *(Continued)*

Option	Description
`-powerdown`	Enables random powerdown (`irl_15`).
`-int_cmd`	Enables interrupts at times in the `int_cmd_file` file.
`-record`	Records the pin assertions and dumps them to `pico_p{in, out}.tape`.
`-statistics`	Keeps statistics on the stack size and timing paths.
`-smu_hold`	Enables the random SMU hold signal in the full-chip level.
`-dcu_debug`	Enables the DCU debug monitor, which prints out all memory accesses.

Customized `steam.plusargs` File

If you customize the simulation, you need to add the new option definitions to the `steam.plusargs` file in addition to modifying the `pj2vcs` or `pj2vlog` program. The default is located in `sim/config/steam.plusargs`.

To add arguments to the `steam.plusargs` file, list one argument per line and include the following information for each argument:

- The name of the argument
- The name of the project that uses the argument
- A description of the argument

Separate the information with colons. For example:
`-fldg_mon:decaf:enable the instruction folding monitor`.

To direct Steam to use a file other than the default, use the environment variable `$STEAM_PLUSARGS` . If the file specified by `$STEAM_PLUSARGS` does not exist, Steam prints a warning message and uses the default file. If Steam cannot locate the default file, it prints an error message and terminates the script.

4.2.6 Test Coverage and Statistics

This section describes the Steam processes that generate coverage and statistical results from test runs.

Merging of Coverage Results

Steam can merge the coverage results that are generated through multiple test cases.

A COVERAGE line, displayed by a Verilog monitor, contains a string of 1s and 0s, which Steam can interpret. For example:

```
COVERAGE: ibuffer  001011101100
```

- The keyword COVERAGE: at the beginning of a line shows Steam that it is a valid coverage statistic.
- The token ibuffer describes the type of coverage and can be any string that does not contain white space.
- The string of 1s and 0s is a predetermined set of events for which the monitor is verifying coverage.

For each event monitored:

- 1 indicates that the item was covered in the test.
- 0 indicates that the item was *not* covered in the test.

To merge coverage results:

- **Add the** -merge_monitor_coverage **option to the command line.**

Steam then performs the following steps:

1. **Browses the log files of the tests that were run and grabs the lines that begin with** COVERAGE:**.**
2. **Uses the bitwise logical** or **command to add together the numerical strings in the** COVERAGE **lines of a statistical token (such as** ibuffer**) to produce the total coverage for that token.**
3. **Logs the totals in the** steam.log **file.**

Merging of Statistical Results

To be recognized by Steam, a statistical result must contain the following:

- The keyword RTL-STAT: at the beginning of a line
- A statistical token (such as CLOCK_CYCLES) that does not contain white space
- An unsigned decimal number that follows the token

An RTL-STAT line is displayed by a Verilog monitor, usually at the end of simulation. Unlike the COVERAGE line, it contains numerical information about a specific event instead of a binary string, which may indicate coverage of multiple events. For example:

```
RTL-STAT: CLOCK_CYCLES  73456
```

Steam merges statistical results by default, as follows:

1. **Browses the log files of the tests that were run and grabs the lines that begin with** `RTL-STAT:`**.**
2. **Adds together the numerical values in the lines that contain statistical tokens (such as** `CLOCK_CYCLES`**) to produce the sum for the tokens.**
3. **Logs the totals in the** `steam.log` **file.**

To disable merging of statistical results:

- **Add the** `-no_merge_rtl_stats` **option to the command line.**

Customization of the Summary Report

By default, Steam displays a summary report after all the tests are complete.

Summary Report Format

The summary report contains basic information about the terminal disposition of test cases in three sections:

- The top section — A title and pre-report information that you specify
- The middle section — Specifics that pertain to the test cases, including the case names, whether the tests passed, failed, or did not run, and the number of instructions and clock cycles in the test cases
- The bottom section — Total statistics

Customization Process

You can customize the report format. Steam utilizes the string `format_string` to dictate the format of the report.

To define `format_string`, do one of the following:

- **Set the environment variable** `FORMAT_STRING`**.**

or

- **Use the** `-format_string` **command-line option.**

Note – In either case, be sure to enclose the format string in single or double quotes so as not to confuse the shell.

---

format_string, just a string itself, must contain formatting information for all three sections of the report. You insert two instances of a separator marker //: one to divide the top section from the middle section, the other to divide the middle section from the bottom section.

Steam posts an error if format_string does not contain these divisions.

format_string can contain any ASCII character. Steam interprets it literally, with several exceptions, to allow the placement of statistical information into the report. In following much the same theory as the printf command in C/C++, you can insert special character sequences into the string; the report formatter then substitutes them with special pieces of information. TABLE 4-5 summarizes these special characters.

TABLE 4-5 Special Characters for format_string

Character	Description	Recommended Section of Report
%c	The clock cycle count for the current test case	Middle
%i	The instruction count for the current test case	Middle
%f	The number of instructions that have been folded away for the current test case	Middle
%q	Elapsed (wall clock) time to execute the test case	Middle
%r	Test result: passed, failed, or not run	Middle
%m	The number of cosimulation mismatches in the current test case	Middle
%e	The number of error messages for the current test case	Middle
%l	The timeout limit assigned to the current test case	Middle
%w	The whole name of the test case, which contains a relative or absolute path to it	Middle
%y	The amount of CPU time it took to run the current test case	Middle
%t	The name of the test case (excluding the path to it)	Middle
%u	The directory in which the test case was run	Middle
%P	The total number of test cases that passed	Bottom
%F	The total number of test cases that failed	Bottom
%N	The total number of test cases that were not run	Bottom
%C	The total number of clock cycles for all tests	Bottom
%I	The total number of instructions that were executed for all tests	Bottom
%X	The total number of instructions that were folded away for all tests	Bottom

TABLE 4-5 Special Characters for `format_string` *(Continued)*

Character	Description	Recommended Section of Report
`%Y`	The total CPU time for all tests	Bottom
`%O`	The top-level directory from which all the run directories are rooted	Top or bottom
`%T`	The total number of tests	Bottom
`%%`	The percent (%) character	Any
`\n`	The newline character	Any

Note – You can use items that are recommended for the top or bottom section of the report in any section with valid results. However, If you use the items for the middle section of the report in the top or bottom section, Steam produces unknown values for them.

We suggest that you format your reports to make them intuitive, such as by frequent inserts of newline characters to space out information, such as the default format string that Steam uses in the following example. Remember that `format_string` is a single line of text; because of line-feed constraints, we cannot present it as such in this example.

```
Test Summary Report:\n\n//%w %r (err=%e, mm=%m) -- cyc: %c, inst:
%i, fold: %f\n//\n Total cycles: %C\nTotal instructions: %I\nTotal
instructions folded away: %X\nTotal tests passed: %P\n Total tests
failed: %F\nTotal tests not run: %N\nTotal tests: %T\nTotal cpu
time: %Y
seconds\n
```

Here is a sample report that the above string produces:

```
Test Summary Report:

./test1 passed (err=0, mm=0) -- cyc: 632, inst: 142, fold: 20
./test2 not run (err=0, mm=0) -- cyc: 0, inst: 0, fold: 0
./test3 failed (err=1, mm=5) -- cyc: 512, inst: 123, fold: 18

Total cycles: 1144
Total instructions: 265
Total instructions folded away: 38
Total tests passed: 1
Total tests failed: 1
Total tests not run: 1
Total tests: 3
Total cpu time: 8.24 seconds
```

4.2.7 Log Files

Check the `steam.log` and *outdir/test*`.log` files for details about the test activity.

4.2.8 Timeout Control

At times, it is convenient to limit the amount of runtime for a test case.

You can use the `-timeout` option in Steam, either inside the suite control file or in the command line, to specify the maximum length of time for which to run a test. If the test is run in DReAM, this time limit includes both the time in the queue and that for running the test.

The `-timeout` option takes an argument of the form `hh:mm` (or `h:mm`) to indicate the maximum amount of time (in hours and minutes) for running the test case.

If a time limit is exceeded, Steam does the following:

- Terminates the test that is running or waiting to be run (if it is in the DReAM queue)
- Posts a notice in the summary report that the test timed out and was therefore terminated
- Assigns `not run` as the result for the test

Always use `-timeout` in conjunction with `-poll`. Due to the nature of timeouts, the script must poll for the completion of the test and check it against the current amount of elapsed time. In case of a `drmwait` command, no checking can occur since `drmwait` returns *only* when all jobs have completed.

4.2.9 Environment Variables

The test execution script uses the environment variables in TABLE 4-6. The script may terminate in error if some of these variables are not set or are set incorrectly.

Note – You can override some of these environment variables with corresponding command-line options.

TABLE 4-6 Environment Variables in the Test Execution Script

Environment Variable	Description
`HOME`	Your home directory
`FORMAT_STRING`	The default string for report formatting. To override this default, specify the string on the command line.
`TIME_LIMIT`	The default time limit for test execution. If it is not set, Steam assumes that all tests can run for an unlimited time. To override this default, specify the time limit on the command line.
`PATH`	The name of the default directory for your UNIX commands, DReAM commands, and Perl scripts
`DSVHOME`	The default path of runtime classes and other class files that are not in the test directories
`CDMS_LOCALDIR`	The location of the local copy of the CDMS tree for the current project
`PROJECT`	The name of the current project
`VCS_NAME`	The name of the VCS executable. The default is `pj2vcs`.
`VLOG_NAME`	The name of the `verilog-XL` executable. The default is `pj2vlog`.
`IAS_NAME`	The name of the instruction simulator. The default is `ias`.
`STEAM_PLUSARGS`	The name of the directory that contains the `steam_plusargs` file
`VFILES_PATH`	The location of the `vfiles` file that `verilog-XL` uses

TABLE 4-6 Environment Variables in the Test Execution Script *(Continued)*

Environment Variable	Description
`JOBFILE_HOME`	The path to the directory that contains the DReAM job constraint files. The `-jobh` option overrides this environment variable.
`JOBFILE_CONSTRAINT`	The name of the job constraint file. The `-jobc` option overrides this environment variable.
`VCOV_DIR`	The directory for `pj2Vcov.cov`, `pj2Vcov.db`, `mayaVcov.inv`, and `cf.run`. If `$PROJECT` has a different setting, these file names reflect that change.
`VCS_GATE_NAME`	The name of the VCS executable, which can be used for running a gate-level simulation
`VCS_VCOV_NAME`	The name of the VCS executable, which can be used for generating VeriCov coverage results
`MAKEFILE_LOC`	The path to the Makefile, which can be used to build tests
`MAKEFILE_NAME`	The name of the Makefile, which can be used to build tests
`PWD`	The current working directory
`SHELL`	The shell in which to run Steam

4.2.10 Exit Status

TABLE 4-7 defines the three types of exit status for Steam.

TABLE 4-7 Exit Status Types for the Test Execution Script

Status	Description
0	The script ran and terminated normally.
1	A command-line parser error occurred. Steam writes error and warning messages to `stderr`.
255	The script terminated due to a problem during environment setup, such as the case where Steam cannot locate a file.

4.3 `gen_sst_control` Script

The `gen_sst_control` script does the following:

- Parses a log file from a simulation run
- Determines the clock cycles at which failures occur
- Generates a `sst_control` file, so that SignalScan can in turn create a trace file around the points of failure

If the log file does not contain any errors or mismatches, `gen_sst_control` creates an empty `sst_control` file.

4.3.1 `sst_control` File

At the beginning of simulation, the test bench reads the `sst_control` file to determine the clock cycle at which to start and stop the logging of signals. Because signal logging is time consuming, it is desirable to log information *only* during the pertinent time of a previously known failure point, not during the entire simulation.

In the current test environment, you must specify `+sst_control` and `TPL_DBG` for the logging to take effect according to the numbers in the `sst_control` file.

Since Verilog can read hexadecimal values, `gen_sst_control` calculates which clock cycle to start and stop the dumping, converts the result to hexadecimal, and prints the values into the `sst_control` file. Each line in that file contains a pair of hexadecimal numbers that indicate the clock cycle to start and stop the dump, respectively.

4.3.2 Syntax and Options

The syntax is:

`gen_sst_control` *options logfile*

TABLE 4-8 summarizes the command-line options.

TABLE 4-8 `gen_sst_control` Command Line Options

Option	Description	Default
`-error`	Checks for error messages only	False
`-mismatch`	Checks for mismatches only	False
`-before` *num*	Begins dump *num* cycles before the point of failure	100
`-after` *num*	Begins dump *num* cycles after the point of failure	100
`-all`	Dumps for all errors and mismatches, not just the first one	False

By default, `gen_sst_control` determines the points of failure by looking for *both* the error and mismatch messages. To direct `gen_sst_control`:

- To look for error messages only, use the `-error` option
- To look for mismatch messages only, use the `-mismatch` option

Also, since most debuggers work with only the first point of failure of a test case, `gen_sst_control` generates only the information for the first error or mismatch by default. Using the `-all` option generates an `sst_control` file that contains logging information for *all* points of failure.

The `-before` and `-after` options specify how much time before and after the point of failure to dump information. Although the default value is 100 clock cycles for both options, you may want to increase this number in certain circumstances, especially before the point of failure.

Often, the *real* point of failure occurs long before it is flagged as an error or mismatch. Conversely, we recommend that you be conservative with the dump size, not just for efficiency in generating the dump, but also because the maximum size of the `TPL.sst` file is 50 Mbytes only. The earlier information in dumps that require more space is truncated.

CHAPTER **5**

External Tools

This chapter describes the use of the following verification tools that are developed outside of Sun:

- Radify (from Synopsys) — For optimizing Verilog designs (see the next section)
- VeriCov (from Summit Design) — For measuring test coverage of the module and expressions (see page 54)

These tools are optional. For information about Radify, go to `www.synopsys.com`. For information about VeriCov, go to `www.summit-design.com`.

5.1 Radify

Radify optimizes Verilog designs for runtime simulation. It performs semantically preserved transformations on the original design in the source and writes out the Verilog code for the modified design to a file. The transformations ensure that the modified design runs faster and is more compact.

You can use some Verilog-XL arguments to start Radify. Radify ignores + arguments that it does not recognize.

▼ To Run Radify

Type at the system prompt:

```
% radify -f vfiles
```

where *vfiles* is a list of Verilog files. The default output file is `rad.v`.

▼ To Run Simulation with VCS

- **Build a `vcs` executable with `rad.v` (instead of the Verilog source).**

  Here is a sample VCS build script for the picoJava-II core:

```
#!/bin/csh -f

setenv DSVHOME /home/picoJava-II/tools
vcs -Mupdate=1 rad.v -P /home/picoJava-II/sim/env/vcs.tab \
   -o pj2vcs $DSVHOME/lib/libloader.a $DSVHOME/lib/libtamv.a \
   -lXt -lX11
```

▼ To Run Simulation with Verilog-XL

- **Provide `rad.v` to Verilog as input (instead of the Verilog source). For example, type the following command:**

```
% pj2vlog rad.v +cosim+ias
```

5.2 VeriCov

VeriCov measures the quality of simulation tests that are applied to a Verilog design and provides a quantitative measure of how well simulation test benches exercise the design.

To run the process, VeriCov uses VCS (Chronologic) or Verilog-XL (Cadence) simulators.

5.2.1 Key Features

VeriCov generates the following coverage results:

- Coverage for blocks — Provides measurement for blocks in the design. Blocks are sequences of procedural statements between `begin-end` pairs that contain a blocking statement, including `if-else`, `for`, `repeat`, and `case` statements.
- Coverage for paths — Provides measurement for the complete set of procedural paths that exists within each initial and always block and within the `forever`, `while`, `repeat`, and `for` loop statements.

- Coverage for expressions — Provides measurement for logical primitives, behavioral expressions, and continuous assignments, including first-level expression term coverage for `not`, `and`, `or`, `xor`, and `xnor` logical operations.

5.2.2 Build Script

The following script builds `pj2vcs` with VeriCov.

```
#!/bin/csh -f

setenv VERICOVHOME /home/vericov/vericov,v1.2.2/5.x
    setenv DSVHOME /home/picoJava-II/tools

    echo "compile -f /home/picoJava-II/sim/env/vfiles" > vcovCompile.in
    echo "new db pj2Vcov.db" >> vcovCompile.in
    echo "libselon" >> vcovCompile.in
    echo "new cov_table" >> vcovCompile.in
    echo "modcovon -bpen *" >> vcovCompile.in
    echo "save instru_code pj2Vcov.inv" >> vcovCompile.in
    echo "save cov_table pj2Vcov.cov" >> vcovCompile.in
    echo "quit" >> vcovCompile.in

    vcov +vcov+config+vcovCompile.in | tee vcovCompile.log
    vcs pj2Vcov.inv -Mupdate=1 -P $VERICOVHOME/pli/vcs2_2/vcovpli.tab \
    -P /home/picoJava-II/sim/env/vcs.tab -o pj2vcs -Mdir=csrc.vcov \
    $VERICOVHOME/pli/vcovpli.o $VERICOVHOME/lib/libtcl.a \
    $DSVHOME/lib/libloader.a $DSVHOME/lib/libtamv.a -lXt -lX11
```

Also, this build script creates two other files, `pj2Vcov.db` and `pj2Vcov.cov`, which are used to generate coverage data during simulation.

5.2.3 Simulation

To perform a simulation task:

1. **Set the environment variable, as follows:**

   `% `**`setenv VCOV_DUMP $`***testname***`.cov`**

2. **Set up links in the test directories:**

   a. **Link to** `pj2Vcov.db`**, the design database.**

   b. **Link to** `pj2Vcov.cov`**, the coverage file.**

c. **Link to a** `vericov` **command file.**

Here is the command file (`cf.run`) for the picoJava-II core:

```
load db pj2Vcov.db
load cov_table pj2Vcov.cov
save cov_table $env(VCOV_DUMP)
```

3. **Execute the task with the following command:**
   ```
   % pj2vcs.vcov +vcov+config+cf.run
   ```

5.2.4 Coverage Report

To generate a coverage report:

1. **Create a** `cf.merge` **file to indicate to** `vcovrpt` **which database to load and which coverage files to merge.**

   Here is a sample `cf.merge` file:

```
load db pj2Vcov.db
load cov_table \
basic/arithm_int.cov \
basic/arithm_long.cov \
basic/locvar_float.cov
union merge.cov
rptgen
```

`vcovrpt` then generates the coverage data.

2. **Create a coverage report with the following command:**
   ```
   % vcovrpt +vcov+config+cf.merge
   ```
   The report contains a summary and details of the instance blocks, paths, and expressions.

Note – If the design contains muxes with default statements that are never executed, the coverage number from `vericov` is lower than the actual coverage. When reading the coverage report, therefore, be sure to determine if a lower coverage number is caused by insufficient tests or by these extraneous statements.

5.2.5 Coverage Numbers

This section gives the coverage numbers obtained by running the full test suite.

TABLE 5-1 shows the CPU coverage numbers.

TABLE 5-1 CPU Test Coverage

Coverage Type	Coverage (%)
Block	93
Path	91
Expression	85

TABLE 5-2 shows the block coverage for each unit of the core.

TABLE 5-2 Block Coverage for Core Units

Unit	Sub-Unit	Block Coverage (%)
IU		93
	ex	92
	ucode	93
	IFU	94
	RCU	95
	pipe	99
DCU		96
ICU		91
FPU		91
SMU		93
PCSU		100

CHAPTER **6**

Monitors

This chapter describes the monitors the picoJava-II verification team uses to observe and sample signals in the CPU core. After an general description of monitors—

- *Overview* on page 59

—the chapter discusses the following monitors:

- *I-Buffer Monitor* on page 60
- *SMU Monitor* on page 60
- *FPU Monitor* on page 61
- *Powerdown Monitor* on page 61
- *Microcode Monitor* on page 62
- *Folding Monitor* on page 63

This chapter also explains in the following sections how interrupt events are scheduled, how the pipeline is frozen, and how statistics for reports are generated.

- *External Interrupt Controller* on page 64
- *Random SMU Hold Generator* on page 66
- *Statistics Monitor* on page 67

6.1 Overview

Monitor functions are cycle-accurate and complement cosimulation testing, which is only instruction-accurate. Some monitors not only sample signal transactions but also generate events from outside the core, such as interrupts and powerdowns.

Monitors report error conditions by displaying to standard output:

```
ERROR:
```

which indicates to the test run script that the test has failed.

For details and source code, see the `.v` and `.h` files in `sim/env`. Instantiations of the monitors are in the monitor module in the `monitor.v` file.

6.2 I-Buffer Monitor

The monitor for the I-Buffer, called the `ibuf_monitor` module, checks the `valid`<15:0> signal (valid byte) in the I-Buffer control, `iu_shift_d` <7:0> (number of read bytes), and `ic_fill_sel`<15:0> events. It performs the following tasks:

- Monitors the I-Buffer state for each cycle
- Sets a flag for each I-Buffer state or pattern
- Checks the coverage
- Posts an error for any pattern that does not match a particular group of patterns and stops the simulation

If the monitor detects a pattern that combines one or more zeroes, then the design team must create a test to reproduce that pattern to ensure complete coverage of the verification for the I-Buffer states.

This monitor operates as follows:

- If the I-Buffer produces a pattern that does not fall under the group, the I-Buffer monitor stops the simulation. A debug process must follow.
- The I-Buffer monitor does not test the boot mode.

6.3 SMU Monitor

Always enabled, the monitor for the SMU operates at the rising edge of each clock cycle.

The SMU monitor posts an error message if:

- The SMU asserts:
  - A fill signal when there are more valid entries than the low watermark in the stack cache
  - A spill signal when there are less valid entries than the high watermark in the stack cache
- There are more than 60 entries in the stack cache and the SMU has not asserted an overflow.

- No stall occurs in the IU pipeline when there are less than six valid entries in the stack cache.
- The SMU writes an undefined value in the stack cache.
- The DCU writes an undefined value in the stack cache.
- The SMU writes an undefined value in the data cache.
- The SMU asserts a spill signal and a fill signal at the same time.
- The SMU and the IU are writing to the same address at the same time.

6.4 FPU Monitor

The monitor for the FPU checks for protocol violations at the IU-FPU interface and samples some critical signals for the purpose of debugging.

The violations include:

- An unexpected assertion or deassertion of signals on the IU-FPU interface
- Signals that go to Xs (invalid conditions)
- An undue length of an FPU operation

The FPU monitor samples and displays:

- Valid FPU opcode
- Valid FPU input and output

6.5 Powerdown Monitor

The powerdown monitor interfaces with the PCSU. It performs the following tasks:

- Tests powerdown features
- Records powerdown cycles and acts as a low-power performance meter

The powerdown monitor operates as follows:

- After the PCSU has asserted the `pcsu_powerdown` signal, `POWERDOWN_WAIT_NUMBER` becomes the upper limit for the wait cycles. If a function unit cannot enter the powerdown state upon reaching this limit, the powerdown monitor posts a warning message to indicate which unit it is.
- At the end of simulation, the powerdown monitor displays:
  - The total of standby cycles
  - The ratio of the number of standby cycles to the total of system cycles

6.6 Microcode Monitor

The microcode monitor, enabled by default, tests several conditions on the interface between the IU and the microcode.

To disable this monitor:

● **Add `+no_ucode_mon` to the command line.**

The rules for this interface are based on the IU-microcode interface (see the *picoJava-II Microarchitecture Guide),* as follows:

- Microcode cannot read and write at the same stack cache address simultaneously.
- A microcode special request, made by the IU, contains the data from the data cache that the IU writes to the stack cache. If the IU makes such a request, the microcode register `u_f03` must indicate a data cache read.
- In a read or write request from the IU to the stack cache or data cache, if one of the following four exceptions occurs—
  - `runtime_IndexOutOfBoundsException`
  - `runtime_NullPtrException`
  - `gc_notify`
  - `ClasscastException`

  —then the states in the registers, the stack cache, and the data cache do *not* change; also, the `u_done` signal goes high.
- The picoJava-II core throws no exceptions in a microcode stall, that is, when `ie_stall_ucode` is high.
- Microcode asserts the `u_done` signal during the last microcode cycle.
- Microcode cannot update the `OPTOP` and `VARS` registers in the last cycle of its operation.
- `iu_trap_r` and `ifu_op_valid_r` are *not* active at the same time.
- If the `VARS`, `FRAME`, or `cp` registers have changed, microcode cannot make any more data cache read requests.

Any time any of the above rules are not met, the microcode monitor posts an error message.

6.7 Folding Monitor

The folding monitor checks the combinations of folding instructions by probing signals from the folding logic and I-Buffer logic to determine whether foldings are correct.

In cases where folding does not occur, the folding monitor keeps track of the causes in a performance counter. The causes are:

- The `psr.fle` signal is off, that is, folding is disabled.
- The I-Buffer is dirty.
- A stack cache miss has occurred.
- Traps or branches have occurred.
- Decoding is not valid; that is, the I-Buffer contains incomplete instructions.

The folding monitor performs the following tasks:

- Decode instructions from the I-Buffer and perform folding.
- Compare the output from the folding logic against that from the folding monitor. Following are the nine valid folding combinations from the folding logic:

`LV LV OP MEM`	`LV OP MEM`	`LV OP`
`LV LV OP`	`LV BG2`	`LV MEM`
`LV LV BG2`	`LV BG1`	`OP MEM`

- Turn on debug mode and post error messages if the folding logic output shows incorrect folding combinations or combinations that differ from those in the folding monitor. The error messages are:

```
IB contents and dirty bits.

Actual folding logic output.

All inputs which can alter the result of the folding logic
output at the unit boundaries.
```

- Collect performance statistics on why folding does not occur.

Here is a sample of the output from the folding monitor at the end of a test:

```
Folding_statistics_begin
--------------------------------------------------------------------------------------
                                       psr.fle   IB      Stack    Traps/     Decode
                  Folded    Missed     Disable   Dirty   Missed   Branches   !valid
--------------------------------------------------------------------------------------
LV LV OP MEM      0         0          0         0       0        0          0
LV LV OP          2         0          0         0       0        0          0
LV LV BG2         0         0          0         0       0        0          0
LV OP MEM         0         0          0         0       0        0          0
LV BG2            0         1          1         0       0        0          0
LV BG1            2         12         11        0       0        0          1
LV OP             1         3          2         0       0        0          1
LV MEM            0         2          2         0       0        0          0
OP MEM            0         0          0         0       0        0          0
--------------------------------------------------------------------------------------
Folding_statistics_end
--------------------------------------------------------------------------------------
```

6.8 External Interrupt Controller

The external interrupt controller is inside the powerdown monitor module and shares the `pj_nmi` and `pj_irl` signals. It performs the following tasks:

- Schedules `pj_nmi` and pj_`irl` events
- Tests interrupt features
- Uses `pj_irl` [15] as a powerdown interrupt

The command line takes the arguments listed in TABLE 6-1.

TABLE 6-1 External Interrupt Controller Arguments

Argument	**Default**
`+int_random`	`pj_nmi` and `pj_irl`[3:0] are low.
`+int_cntl`	The interrupt controller is off.
`+int_cmd`	The interrupt commands are off.
`+powerdown`	Does not generate `pj_irl` [15].

The interrupt protocol uses the following memory locations:

- PENDING_REGISTER, mapped to 30'h0000fff8
- STOP_RANDOM_INTERRUPT, mapped to 30'h0000fff4

Here is the recommended coding sequence for the interrupt handler:

CODE EXAMPLE 6-1 Recommended Coding Sequence for the Interrupt Handler

```
INT_Start:
              sipush 0x0ACC;       // Enters critical region {
              sipush 0xFFF8;
              sethi 0x0;
              ncstore_word;        // Stores 0xACC to 0x0000FFF8,
              sipush 0xFFF8;       // indicating ack to the interrupt controller
              sethi 0x0;
              ncload_word;         // Reads 0xACC from 0x0000FFF8

                                   //  } Exits critical region

              sipush 0xFFF4;       // Read/write to STOP_RANDOM_INTERRUPT (30'h0000fff4)
              sethi 0x0;           // if you need to stop assertion of random interrupts.
              ncload_word;

              priv_read_psr;       // Turns on interrupt enable bit
              sipush 0x0010;       // if nested interrupts are allowed for the current test case;
              ior;                 // otherwise, the IU turns it on at the return point.
              priv_write_psr;

                                   // Main_body begin {
Application:   .
               .
                                   // } end

              priv_ret_from_trap;  // Returns from trap
```

The external interrupt controller operates as follows:

- If +int_random is in the command line for simulation, the external interrupt controller asserts pj_nmi and pj_irl randomly. These signals remain active until the test software executes ncwrite_word to PENDING_REGISTER.

  The handler performs ncwrite prior to turning on PSR.IE. In some cases, however, ncwrite occurs so late on the bus that the IE is on and the RTL receives another interrupt, but the ias sees only one interrupt. Thus, ncload_word must follow ncwrite_word to enforce the ordering and ensure that PSR.IE is on only *after* ncwrite.

  The external interrupt controller continues to assert interrupts at random until the test software executes ncwrite_word to STOP_RANDOM_INTERRUPT.

- If +int_cntl is in the command line, it overrides +int_random. The external interrupt controller then becomes active and initiates the nested interrupt patterns by filling PENDING_REGISTER in the sequence of marching 1s.

- If +int_cmd is in the command line, it overrides +int_cntl and +int_random. A command file, int_cmd_file, then schedules the nmi and irl<3:0> events to check the priority of the traps, interrupts, and interrupt-related functions in other units.

This operation also verifies the interaction between the IU and FPU, the stack operation and the IU, the microcode, or the TRAP software when an interrupt occurs. You must prepare for this mode a force file named int_cmd_file in the working directory. CODE EXAMPLE 6-2 is an example.

CODE EXAMPLE 6-2 Example of Interrupt Handler Code in Verilog

```
// define the interrupt level (1-16), 16 is nmi
`define INT_LEVEL 16
// define the trigger signal
`define INT_TRIGGER (negedge sys.pico.iu.datapath.ucode.u_done)
always @`INT_TRIGGER begin
pending_reg[`INT_LEVEL] <= #1 1'b1;
```

6.9 Random SMU Hold Generator

The random SMU hold generator randomly activates the smu_hold signal inside the CPU core, even if an SMU stall condition is not really present. This is a useful way to accelerate testing in the presence of pipeline holds caused by the SMU. Since the SMU operates asynchronously to the execution of instructions in the pipeline, it can potentially cause a hold in any clock cycle. The random SMU hold mechanism allows us to hit corner cases involving the presence of an SMU stall that are not easily encountered during normal program execution.

To activate this generator:

- **Add the option** +smu_hold **to the command line.**

To specify a frequency for the smu_hold signal:

- **Add the option:** +hold_seed_2 **or** +hold_seed_3 **to the command line.**

6.10 Statistics Monitor

The statistics monitor performs sampling at the rising edge of each clock cycle. It gathers three types of information:

- The average number of stack cache entries. To obtain this information, the statistics monitor does the following:
  - Samples the numbers at each clock cycle.
  - Divides the aggregate sum by the total number of elapsed clock cycles at the end of the simulation.
- The number of times the `non-ignored-hold` signal is high, that is, the number of times SMU hold conditions (`smu_hold` and `pj_hold`) are not ignored

  To obtain this information, the statistics monitor keeps a count of whether `non-ignored-hold` is true at each clock cycle. If so, it adds 1 to the counter.
- The number of times that the top eight critical timing paths are hit. To obtain this information, the statistics monitor does the following:
  - Checks when a signal changes from its source to the end of the timing path.

    Instead of checking every point along the way, the monitor checks several points along the path, including the end point (the signal going into the flip-flop), thereby approximating whether the timing path is hit. We have identified the timing paths from the bucket report file from gate-level synthesis.
  - Prints a message at the time of execution and increments a counter for each hit.

The statistics monitor prints the totals of the above statistics at the end of simulation.

To enable the statistics monitor:

● **Add** `+statistics` **to the command line.**

6.11 Activity Monitor

The activity monitor counts occurrences of certain events that reflect the degree of activity in functional blocks. The power consumption of the design can be estimated based on these statistics. The activity monitor keeps track of the following events:

- ICU control is idle
- ICRAM is disabled
- I-Buffer is active

- DCU reads
- DCU writes
- DCU line fills
- DCRAM is disabled
- Write Buffer is idle
- DCU holds the pipe
- SMU holds the pipe
- ICU holds the pipe
- Microcode is busy
- FPU is idle
- I-Cache accesses
- I-Cache misses

PART II picoJava-II Verification Tests

CHAPTER 7

Verification of the Instruction Cache Unit (ICU)

The Instruction Cache Unit (ICU) fetches instructions from the instruction cache (I-Cache) for the decode block in the Integer Unit (IU). To separate the rest of the pipeline from the fetch stage, an instruction buffer (I-Buffer) holds the instructions fetched from memory until consumption by the IU.

This chapter contains the following sections:

7.1 Tests for Basic Functions

This section describes basic I-Cache-related test operations; the next section explains the tests for back-to-back cache operations.

TABLE 7-1 shows the ICU test coverage.

TABLE 7-1 Test Coverage for the ICU

Coverage	Read	Description
Alignment	2.1	Tests with different length of instructions go across word (4 bytes), IB fetch (7 bytes), and instruction cache line (16 bytes) boundaries (`icu_2_1.code`).

TABLE 7-1 Test Coverage for the ICU *(Continued)*

Cache hit	2.2	Reads with I-Cache hit in the first 4 bytes and the second 4 bytes in the cache line (`icu_2_2.code`).
Cache miss	2.3	Reads with an I-Cache miss, makes sure cache line fill happens (`icu_2_2.code`).
Disabled cache	2.4	Reads with a disabled I-Cache, makes sure it acts like NC read and that no instruction is loaded to I-Cache (`icu_2_4.code`).

7.2 Tests for Instruction Cache Functional Units

This section describes the tests for the I-Cache functional units.

7.2.1 I-Cache Control (`ic_cntl`)

We test I-Cache control as follows:

- Test accesses with different PC values (target PC, trap PC, or next PC).

  The test case for I-Cache hits is `icu_3_1_1.code`.

  The test cases for I-Cache misses are:

```
icu_3_1_2.code (ifeq)            icu_3_1_5.code (ifle)
icu_3_1_3_.code (ifne)           icu_3_1_6.code (ifgt)
icu_3_1_4.code (iflt)            icu_3_1_7.code (ifge)
```

- Test with `goto` instructions (`icu_3_1_8.code`).
- Test with two loops (`icu_3_1_9.code`).
- Test select with different buses through random test cases.

7.2.2 Cache Read Misses

We test cache read misses as follows:

- Test the I-Buffer with nine or more valid entries to stall the pipeline (`icu_3_2_1.code`) (`ibuf_full`).

- Test with a new request to a different line when the ICU is filling a cache (`icu_3_1_8.code` and `icu_3_1_9.code`).
- Test at random with a new request to the same line that is being filled.
- Test at random with external interrupt that goes high when the ICU is doing cache line fill.
- Test with a branch to a `PC` that has the same index as the current `PC`, but a different tag (`icu_3_1_10.code`).

7.2.3 Noncacheable (NC) Reads

We test noncacheable (NC) reads as follows:

- Test with NC reads and ensure that the data are not written into the I-Cache (`icu_2_4.code`) (I-Cache disabled).
- Test at random with NC reads with a different index. Random test cases with monitor hit on all of these cases.
- Test at random with an external interrupt equal to 1 at the same time that a noncacheable request is sent to the bus.

7.3 Tests for Instruction Buffer (I-Buffer)

We test all combinations of the I-Buffer with monitor and random test cases, the source for which is in `sim/tests/ibufcov`.

7.3.1 I-Buffer Control (`ibuf_cntl`)

We test I-Buffer control as follows:

- Test with tag miss and ensure that the valid bit is reset (`icu_3_4_1_1.code`).
- Access the same flush line and ensure that a cache miss occurs (`icu_3_4_1_2.code`).
- Test with a specific address: Diagnostic-write a valid bit to the address. Create a new line with the same index, then flush the address. Diagnostic-read if the old address's valid bit is clear (`icu_3_4_1_3.code`).

For a different index, use `icu_3_4_1_4.code`.

7.3.2 I-Cache Datapath (`icu_dpath`)

We test read and write data from the I-Cache with a different word and index.

The test cases are `icu_3_5_1.code` (I-Cache data) and `icu_3_5_2.code` (I-Cache tag).

7.3.3 Disabled I-Cache Instructions

The ICU treats all requests to the I-Cache as noncacheable requests and forwards them to the BIU.

We test with a line fill operation, followed by a disabled I-Cache instruction. Then, send a new request to the same line that is being filled (`icu_3_6_1.code`).

7.3.4 Invalidation of the Cache

If the address is present in the instruction cache, the ICU invalidates it. However, if the instruction cache is not enabled (`PSR.ICE = 0`), then the ICU ignores the invalidation request.

We test with `ICE = 0` and `ICE = 1`.

The test cases are `icu_4_4_1.code` (`PSR.ICE = 1`) and `icu_4_4_2.code` (`PSR.ICE = 0`).

7.3.5 Boot Mode

The test case for boot mode is `icu_4_5_1.code`.

CHAPTER **8**

Verification of the Integer Unit (IU)

This chapter describes the verification plan for the Integer Unit (IU) at both the functional and architectural levels. It contains the following sections:

8.1 Functional Tests

TABLE 8-1 lists the functional tests for the IU.

TABLE 8-1 Functional Tests for the IU

Number	Name	Description
1	`diag_pass.code`	Verifies the reset code and error reporting mechanism for `*.code` files.
2	`diag_detect.code`	Verifies a small set of instructions for self-checking diagnostics, such as `bipush`, `sipush`, `if_icmpeq`, and `if_icmpn`, during processor block integration.
3	`arithm_int.code`	Verifies the results of `iadd`, `isub`, `idiv`, `imul`, `irem`, and `ineg`. This test picks operands arbitrarily and performs one calculation for each instruction.

TABLE 8-1 Functional Tests for the IU *(Continued)*

Number	Name	Description
4	`arithm_long.code`	Verifies the results of `ladd`, `lsub`, `ldiv`, `lmul`, `lrem`, and `lneg`. This test picks operands arbitrarily and performs one calculation for each instruction.
5	`compare_branch.code`	Verifies the comparison and branching for `if`*cond*, `if_icmp`*condition*, `lcmp`, `if_acmp`*condition*, `goto`, `jsr`, `ret`, and `ret_from_sub`.
6	`convert_int.code`	Verifies the conversion operation for `i2b`, `i2c`, `i2s`, and `i2l`. This test covers sample cases where sign-extension must or must not occur for each conversion.
7	`convert_long.code`	Verifies the conversion operation for `l2i` and ensures that truncation of signed words occurs correctly.
8	`locvar_dbl.code`	Verifies the store to and load from a `locvar` of the type double. This test uses both forms of `dload_`*n*`|dstore_`*n* and `dload` *n*`|dstore` *n*.
9	`locvar_float.code`	Verifies the store to and load from a `locvar` of the type float. This test uses both forms of `fload_`*n*`|fstore_`*n* and `fload` *n*`|fstore` *n*.
10	`locvar_int.code`	Verifies the store to and load from a `locvar` of the type int. This test uses both forms of `iload_`*n*`|istore_`*n* and `iload` *n*`|istore` *n*.
11	`locvar_long.code`	Verifies the store to and load from a `locvar` of the type long. This test uses both forms of `lload_`*n*`|lstore_`*n* and `lload` *n*`|lstore` *n*.
12	`locvar_object.code`	Verifies the store to and load from a `locvar` of the type object. This test uses both forms of `a` `oload_`*n*`|ostore_`*n* and `oload` *n*`|ostore` *n*.
13	`logic_int.code`	Verifies the results of `ishl`, `ishr`, `iushr`, `iand`, `ior`, and `ixor`. This test picks operands arbitrarily, performs one calculation for each instruction, and tests the difference between `ishr` and `iushr`.

TABLE 8-1 Functional Tests for the IU *(Continued)*

Number	Name	Description
14	`logic_long.code`	Verifies the results of `lshl`, `lshr`, `lushr`, `land`, `lor`, and `lxor`. This test picks operands arbitrarily, performs one calculation for each instruction, and tests the difference between `lshr` and `lushr`.
15	`push_dconst.code`	Verifies that `dconst_0` and `dconst_1` push the correct values onto the stack.
16	`push_fconst.code`	Verifies that `fconst_0`, `fconst_1`, and `fconst_2` push the correct values onto the stack.
17	`push_iconst.code`	Verifies that `iconst_`*x* pushes the correct values onto the stack.
18	`push_lconst.code`	Verifies that `lconst_0` and `lconst_1` push the correct values onto the stack.
19	`stack_instr_a.code`	Sets up the stack and verifies that `nop`, `pop`, `pop2`, `dup`, `dup2`, and `swap` modifies it correctly.
20	`stack_instr_b.code`	Verifies the operation of `dup_x1`, `dup_x2`, `dup2_x1`, and `dup2_x2`.
21, 22	`store_load_byte.code` `ncstore_load_byte.code`	Verify the operation of `store_byte`, `load_byte`, and `load_ubyte`. These tests also verify the sign extension for `load_byte`, no sign extension for `load_ubyte`, data caching through diagnostic cache instructions, big and little endian addressing, and reads and writes to noncacheable address space.
23, 24	`store_load_short.code` `ncstore_load_short.code`	Verify the operations of `store_short`, `load_char`, and `load_short`. These tests also verify the sign extension for `load_short`, no sign extension for `load_char`, data caching through diagnostic cache instructions, big and little endian addressing, and reads and writes to noncacheable address space.
25, 26	`store_load_short_oe.code` `ncstore_load_short.code`	Verify the operations of `store_short_oe`, `load_char_oe`, and `load_short_oe`. These tests also verify the sign extension for `load_short_oe`, no sign extension for `load_char_oe`, data caching through diagnostic cache instructions, big and little endian addressing, and reads and writes to noncacheable address space.

TABLE 8-1 Functional Tests for the IU *(Continued)*

Number	Name	Description
27, 28	`store_load_word.code` `ncstore_load_word.code`	Verify the operations of `store_word` and `load_word`, data caching through diagnostic cache instructions, big and little endian addressing, and reads and writes to noncacheable address space.
29, 30	`store_load_word_oe.code` `ncstore_load_word_oe.code`	Verify the operations of `store_word_oe` and `load_word_oe`, data caching through diagnostic cache instructions, big and little endian addressing, and reads and writes to noncacheable address space.
31	`store_load_byte_index.code`	Verifies the operations of `store_byte_index`, `load_byte_index`, and `load_ubyte_index`. This test also verifies sign extension for `load_byte_index`, no sign extension for `load_ubyte_index`, data caching through diagnostic cache instructions, and reads and writes to noncacheable address space.
32	`store_load_short_index.code`	Verifies the operations of `store_short_index`, `load_short_index`, and `load_char_index`. This test also verifies sign extension for `load_short_index`, no sign extension for `load_char_index`, data caching through diagnostic cache instructions, big and little endian addressing, and reads and writes to noncacheable address space.
33, 34	`store_load_word_index.code` `nastore_word_index.code`	Verify the operations of `store_word_index`, `nastore_word_index`, and `load_word_index`. These tests also verify data caching through diagnostic cache instructions, big and little endian addressing, and reads and writes to noncacheable address space.
35, 36	`dneg_test.code`, `lneg_test.code`	Verify the `dneg` and `lneg` operations.
37	`iucmp_test.code`	Verifies the `iucmp` operation.
38	`zero_line.code`	Verifies that a particular line in the data cache is zeroed out. This test does not check the tags after issuing the instruction.
39	`priv_rw_reg.code`	Verifies that `write_`*regname*, `priv_write_`*regname*, `read_`*regname*, and `priv_read_`*regname* can write to the writable bits of the registers.

TABLE 8-1 Functional Tests for the IU *(Continued)*

Number	Name	Description
40	`sethi.code`	Verifies the operation of the `sethi` instruction.
41	`wide_test.code`	Verifies the operations of the *x*`load_w`|*x*`store_w` and `iinc_w` instructions.
42	`call_return.jasm`	Verifies the functionality of the `call`, `return0`, `return1`, and `return2` instructions.
43	`nonnull_quick.jasm`	Verifies the functionality of the `nonnull_quick` instruction.
44	`monitor_test.jasm`	Verifies the `monitorenter` and `monitorexit` instructions in conjunction with the `lockcount` and `lockaddress` registers.
45	`ldc_test.jasm`	Verifies the `ldc_quick`, `ldc_w_quick`, `aldc_quick`, and `aldc_w_quick` instructions.
46	`ldc2_test.jasm`	Verifies the `ldc2_w_quick` instruction.
47	`putget.jasm`	Verifies the functionality of the `putstatic_quick`, `putstatic2_quick`, `getstatic_quick`, `getstatic2_quick`, `putfield_quick`, `putfield2_quick`, `getfield_quick`, `getfield2_quick`, `putfield_quick_w`, and `getfield_quick_w` instructions.
48, 49	`aputaget_gcvector.jasm` `aputaget_gctrain.jasm`	Verify the `aputstatic_quick`, `agetstatic_quick`, `aputfield_quick`, and `agetfield_quick` instructions and associated garbage collection traps.
50	`aputaget_null.jasm`	Verifies `NullPointerException` with the `aputfield_quick` and `agetfield_quick` instructions.
51	`arrays.java`	Verifies the basic operation of all array instructions in *The Java Virtual Machine Specification*.
52	`returns.java`	Verifies the basic operation of all return instructions in *The Java Virtual Machine Specification*.
53	`invoking_banana.java`	Verifies the basic operation of all `invoke_*_quick` instructions that are implemented in hardware.
54	`fib_2.java`	Verifies method invocation and return with a simple recursive algorithm.

TABLE 8-1 Functional Tests for the IU *(Continued)*

Number	Name	Description
55	`allinst.jasm`	Verifies the functionality of RISC-type instructions in the instruction set.
56	`if_acmp.code`	Verifies `if_acmpne` and `if_acmpeq` operations.
57	`B2B_1.code`	Verifies that `fadd` and `dadd` occur back to back.
58	`allmul.code`	Verifies `dmul` and `fmul` operations.
59	`bigarray.code`	Tests load and store of large arrays (> 16-Mbyte elements).
60	`bigarray_handles.code`	Tests load and store of large arrays (> 16-Mbyte elements) with handles.
61	`inlining.java`	Verifies basic operation of `invokestatic` and `invokevirtual`.
62	`allinst_brk.jasm`	Verifies that breakpoint operations work with branches, basic instructions, register operations, and load / stores.
63	`monitor_test_2`	Triggers stack cache miss when `monitorenter` is executed by setting `VARS` 60 entries away from `OPTOP`.

For details, see the following source directories:

```
sim/test/pico_vts/basic
sim/test/pico_vts/basic_java
sim/test/pico_vts/instr_tests
```

Testing of instructions not in TABLE 8-1 is part of the verification for the cache or trap. See Chapter 7, *Verification of the Instruction Cache Unit (ICU)*, Chapter 10, *Verification of the Data Cache Unit (DCU)*, and Chapter 12, *Traps and Interrupts*.

Architecture tests for the IU verify the following features:

- Hazard detection and prevention
- Branch prediction
- Folding
- Handling of polymorphic instructions and boundary conditions

TABLE 8-2 lists the hazard tests. For details, see the sources at `sim/test/pico_vts/directed/iu/hazard`.

TABLE 8-2 Hazard Tests for the IU

Number	Name	Description
1-30	`alu_hzd_1-30.code`	Creates the test by setting `i` = 1, `j` = 1 to 6, and `k` = 1 to 6.
31-60	`stack_instr_hzd_1-30.code`	Creates the test by setting `i` = 2, `j` = 1 to 6, and `k` = 1 to 6.
61-90	`locvar_hzd_1-30.code`	Creates the test by setting `i` = 3, `j` = 1 to 6, and `k` = 1 to 6.
91-120	`convers_hzd_1-30.code`	Creates the test by setting `i` = 4, `j` = 1 to 6, and `k` = 1 to 6.
121-150	`stack_push_hzd_1-30.code`	Creates the test by setting `i` = 5, `j` = 1 to 6, and `k` = 1 to 6.
151-180	`fpu_hzd_1-30.code`	Creates the test by setting `i` = 6, `j` = 1 to 6, and `k` = 1 to 6.
181-182	`branch_hzd_1-2.code`	Creates the test for hazards with branch instructions.

8.2 Tests for Folding Combinations

The IU can “fold” up to four instructions to decrease the cycles per instruction (CPI). For details, see the sources at `sim/test/pico_vts/directed/iu/fold`.

There are a total of 374 tests to verify all possible folding combinations.

8.3 Tests for Back-to-Back Instructions

TABLE 8-3 lists the tests for back-to-back instructions.

TABLE 8-3 Tests for Back-to-Back Instructions

Name	Description
`cache_flush_b2b.code`	Tests the back-to-back `cache_flush` instruction.
`imul_b2b.code`	Tests the back-to-back `imul` instruction.
`store_load_b2b.code`	Tests the back-to-back `load` and `store` instructions.
`cache_index_flush_b2b.code`	Tests the back-to-back `cache_index_flush` instruction.
`load_word_b2b.code`	Tests the back-to-back `load_word` instruction.
`zero_line_b2b.code`	Tests the back-to-back `zero_line` instruction.
`getstatic2_quick_b2b.jasm`	Tests the back-to-back `getstatic2_quick` instruction.
`putstatic2_quick_b2b.jasm`	Tests the back-to-back `putstatic2_quick` instruction.
`getstatic_quick_b2b.jasm`	Tests the back-to-back `getstatic_quick` instruction.
`putstatic_quick_b2b.jasm`	Tests the back-to-back `putstatic_quick` instruction.

8.4 Tests for Polymorphic Instructions

A polymorphic instruction contains one functional specification, but several design forms for which the functionality must be correct.

8.4.1 Local Variable and Object Field Load or Store

A load or store from a local variable or a field in an object can have different forms. The following cases are tested:

- There is a stack cache hit during a read or write to a local variable.
- There is a stack cache miss but a data cache hit during a read or write to a local variable.
- There is a stack cache miss and a data cache miss during a read or write to a local variable.

- An object reference is a direct pointer to the memory allocated from the heap for the object data during a read or write to a field.
- An object reference is a pointer to a handle during a read or write to a field.

The tests for these cases are a subset of the functional tests in TABLE 8-1 and are available in `sim/test/pico_vts/basic` and `sim/test/pico_vts/basic_java`. Run these tests with various configurations to target all of the above cases.

8.4.2 Microcode Instructions

Tests for microcode instructions are necessary for flagging exceptions and stack cache misses during accesses of the stack data and object references (pointers to handles) as well as other conditions, such as interrupts.

TABLE 8-4 lists the microcode tests. It is a good idea to run these tests with different cache sizes, with and without references to handles, and with Verilog tasks to generate interrupts during microcode operations. For details, see the sources in `sim/test/pico_vts/directed/iu/microcode`.

TABLE 8-4 Microcode Tests for the IU

Number	Name	Description
1-8	`xaload_null.jasm`	Verifies that an array load instruction on a null arrayref causes a `NullPointerException` trap.
9-14	`xarray.jasm`	Verifies array operations, sign extensions on loads, and so on.
15-21	`xarray_ovf.jasm`	Verifies that an array operation out of bounds causes an `ArrayIndexOutOfBoundsException` trap.
22-29	`xastore_null.jasm`	Verifies that an array store instruction on a null arrayref causes a `NullPointerException` trap.
30-38	`return, xreturn.jasm`	Verifies stack deallocation, stack cache misses on restoring registers from the method frame, and back-to-back returns.
39	`invoke_null.jasm`	Verifies that `invoke_*_quick` with a null object causes a `NullPointerException` trap.
40	`invoker.jasm`	Verifies stack allocation, a large number of parameters causing stack cache miss while accessing object references, and so on for all `invoke_*_quick` instructions.

TABLE 8-4 Microcode Tests for the IU *(Continued)*

Number	Name	Description
41	call_return.jasm	Verifies stack allocation, deallocation for call, return0, return1, and return2. Checks for the stack cache miss case while restoring registers on a return.
42	get_curr_class.jasm	Verifies the correct operation of the get_current_class instruction.
43	checkcast_test.jasm	Verifies trapping and nontrapping modes of the checkcast_quick microcode.
44	exit_sync_meth.jasm	Verifies the operation of exit_sync_method and the return followed by the exit_sync_method.
45	instanceof_test.java	Verifies trapping and nontrapping modes of instanceof_quick.
46	tableswitch.java	Verifies switching with positive and negative indices, large and small offsets, and so on.
47	invokesuper_q.jasm	Verifies the invokesuper_quick mask off bits 1 and 0 of obj_Hint_Block.
48, 49	call_brkg goto_brk	Verify breakpoint operations involving brk1a, brk2a, brk12c.
50-54	aastore_q_1 aastore_q_2 aastore_q_3 aastore_q_4 aastore_q_5	Verify aastore_quick operations.
55	anewarray_null	Verifies creation of array using anewarray.
56	arraylength_null	Verifies arraylength operation on a null object.
57, 58	ucode_smiss_1 ucode_smiss_2	Tests for microcode accesses which miss in the stack cache.
59	larray_smiss	Verifies access to array of longs with stack cache miss.
60	invokesuper_q	Verifies invoke_super_quick operation.
61	aaload_instanceof	Verifies back-to-back operation of aaload and instanceof.
62-64	xarray_extended	Verifies array access operations for large arrays (> 16 Mbytes).
65-67	xarray_handle_extended	Verifies array access operations with handle bit set for large arrays (> 16 Mbytes).
68	barray_gl.jasm	Test case to check global register bypassing during microcode instruction in E stage.

TABLE 8-4 Microcode Tests for the IU *(Continued)*

Number	Name	Description
69	`barray_gl_sw1.jasm`	Test case to check global register bypassing with microcode instruction followed by `nop`.
70	`barray_gl_sw2.jasm`	Test case to check global register bypassing during microcode execution with `write_global` followed by `nop`.
71	`barray_brk_halt`	Verifies two breakpoint halts.
72	`ucode_ext_halt`	Test case for invoke instructions.

8.4.3 Boundary Conditions

Boundary condition tests verify corner cases in the design. Examples are:

- **Local variable load and store**: Storing a long or double local variable with one word in the stack cache and one word in the data cache.
- **Branch instructions**: Using maximum and minimum branch offsets.
- **Arithmetic, logic, and shift instructions**: Operands that cause overflow and underflow.
- **Local variable load or store:** Accessing the local variable 255 and using the wide index.

TABLE 8-5 lists the tests. For details, see the sources in `sim/test/pico_vts/directed/iu/boundary`.

TABLE 8-5 Boundary Tests for the IU

Number	Name	Description
1	`goto_bound.code`	Tests forward and backward branching with boundary values of the 2-byte signed jump offset.
2	`jsr_ret_bound.code`	Tests the `jsr` and `ret` instructions for wide offsets.
3	`if_w_1.code`	Tests the conditional branch `if`*condition* for the true condition with wide offset.
4	`if_w_2.code`	Tests the conditional branch `if`*condition* for the false condition with wide offset.
5	`if_w_3.code`	Tests the conditional branch `if_icmp`*condition* with wide offset.

8.5 Miscellaneous Tests

TABLE 8-6 lists the miscellaneous tests for the IU. For details, see the sources in `sim/test/pico_vts/directed/iu/other_tests`.

TABLE 8-6 Miscellaneous IU Tests

Number	Name	Description
1	`power_down.code`	Tests software-initiated powerdown with the `priv_powerdown` instruction in `irl_f.jasm` (exception handler). Runs the test with random `IRL`/`NMI` interrupts. `irl_f` puts the picoJava-II core in powerdown mode, and other interrupts then wake it up.
2	`priv_reset.code`	Tests software-initiated reset with the `priv_reset` instruction.
3	`dsub_test.jasm`	Verifies a corner case in the IU-SMU-DCU interface.
4	`iload_test.jasm`	Verifies a corner case in the IU-SMU-DCU interface.
5	`edo101.jasm`	Verifies a corner case bug caused by `fdiv` followed by the `cache_flush` instruction.
6	`boot_mode`	Verifies `boot8` mode.
7	`load_boundary`	Verifies long operations with half of the operands in the stack cache and the other half in the data cache.
8	`dcu_smu`	Verifies SMU load hits in the DCU and an extended load instruction with a simultaneous error acknowledgment.
9	`load_word_back.jasm`	Verifies back-to-back extended load instructions missing the data cache.
10	`load_word_lop.jasm`	Verifies a `load_word` operation followed by a long operation. The first half of an operand missing in the data cache is required from `load_word`; the other half is a hit in the data cache for the long operation.
11	`branch_trap`	Verifies the branch instructions followed by various trap instructions, and vice versa.
12	`load_boundary_mul.jasm`	Verifies the `load_word` operation followed by a multiplication. The operand for the multiplication is required from `load_word` and is missing in the data cache; the load_word address is equal to the `SC_BOTTOM` address.

Other miscellaneous tests under sim/test/pico_vts include:

- Tests with random code that hit bugs during the design of picoJava-II, located in the directories random_failure and maya_random_failure
- Tests that cover all possible instruction buffer states, located in the directory ibufcov
- Tests that stress the stack cache and the invoke_* instructions, located in the directory stress

CHAPTER 9

Verification of the Floating Point Unit (FPU)

The Floating Point Unit (FPU) in picoJava-II performs all the operations for floats and doubles.

This chapter contains the following sections:

9.1 Overview

The FPU interacts with the Integer Unit (IU) in the execution stage. It takes multiple cycles to complete an operation. The IU can hold the FPU in both the input and output stages; it can also terminate an FPU operation in the case of a trap or an interrupt.

The FPU does not throw exceptions for its operations.

9.2 Functional Tests

TABLE 9-1 describes the functional tests for the FPU.

TABLE 9-1 Functional Tests for the FPU

Number	Name	Description

1	`B2B_1.code`
	`B2B_2.code`
	`B2B_3.code`
	`B2B_4.code`
	`B2B_5.code`
	`B2B_6.code`
	`B2B_7.code`
	`B2B_8.code`
	`B2B_9.code`
	`B2B_10.code`
		Test all combinations of back-to-back operations in the FPU for both floats and doubles.

2	`BYPASS_1.code` `BYPASS_2.code`	Test the bypassing cases for the FPU operations in the execution stage.

3	`DADD_000.code`
	`DADD_001.code`
	`DADD_002.code`
	`DADD_003.code`
	`DADD_004.code`
	`DADD_005.code`
	`DADD_006.code`
	`DADD_010.code`
	`DADD_011.code`
	`DADD_012.code`
	`DADD_013.code`
	`DADD_014.code`
	`DADD_015.code`
		Test the DADD operation.

4	`FADD_000.code`
	`FADD_001.code`
	`FADD_002.code`
	`FADD_003.code`
	`FADD_004.code`
	`FADD_005.code`
	`FADD_006.code`
	`FADD_010.code`
	`FADD_011.code`
	`FADD_012.code`
	`FADD_013.code`
	`FADD_014.code`
	`FADD_015.code`
		Test the FADD operation.

TABLE 9-1 Functional Tests for the FPU *(Continued)*

Number	Name	Description

5	`FSUB_000.code`
	`FSUB_001.code`
	`FSUB_002.code`
	`FSUB_003.code`
	`FSUB_004.code`
	`FSUB_005.code`
	`FSUB_006.code`
	`FSUB_010.code`
	`FSUB_011.code`
	`FSUB_012.code`
	`FSUB_013.code`
	`FSUB_014.code`
	`FSUB_015.code`
		Test the FSUB operation.

6	`FCMPG_001.code`
	`FCMPG_002.code`
	`FCMPG_003.code`
	`FCMPG_004.code`
	`FCMPG_005.code`
	`FCMPG_006.code`
	`FCMPG_010.code`
	`FCMPG_011.code`
	`FCMPG_012.code`
	`FCMPG_013.code`
	`FCMPG_014.code`
	`FCMPG_015.code`
		Test the FCMPG operation.

7	`FCMPL_001.code`
	`FCMPL_002.code`
	`FCMPL_003.code`
	`FCMPL_004.code`
	`FCMPL_005.code`
	`FCMPL_006.code`
	`FCMPL_010.code`
	`FCMPL_011.code`
	`FCMPL_012.code`
	`FCMPL_013.code`
	`FCMPL_014.code`
	`FCMPL_015.code`
		Test the FCMPL operation.

TABLE 9-1 Functional Tests for the FPU *(Continued)*

Number	Name	Description
8	DSUB_000.code DSUB_001.code DSUB_002.code DSUB_003.code DSUB_004.code DSUB_005.code DSUB_006.code DSUB_010.code DSUB_011.code DSUB_012.code DSUB_013.code DSUB_014.code DSUB_015.code	Test the DSUB operation.
9	F2D_000.code F2D_010.code	Test the F2D operation.
10	F2I_000.code F2I_010.code	Test the F2I operation.
11	F2L_000.code F2L_010.code	Test F2L operation.
12	D2F_000.code D2F_010.code	Test D2F operation.
13	D2I_000.code D2I_010.code	Test D2I operation.
14	D2L_000.code D2L_010.code	Test D2L operation.
15	DCMPG_001.code DCMPG_002.code DCMPG_003.code DCMPG_004.code DCMPG_005.code DCMPG_006.code DCMPG_010.code DCMPG_011.code DCMPG_012.code DCMPG_013.code DCMPG_014.code DCMPG_015.code	Test the DCMPG operation.

TABLE 9-1 Functional Tests for the FPU *(Continued)*

Number	Name	Description
16	DCMPL_001.code DCMPL_002.code DCMPL_003.code DCMPL_004.code DCMPL_005.code DCMPL_006.code DCMPL_010.code DCMPL_011.code DCMPL_012.code DCMPL_013.code DCMPL_014.code DCMPL_015.code	Test the DCMPL operation.
17	I2D_000.code I2D_010.code	Test the I2D operation.
18	I2F_000.code I2F_010.code	Test the I2F operation.
19	L2D_000.code L2D_010.code	Test the L2D operation
20	L2F_000.code L2F_010.code	Test the L2F operation.
21	all_mul.code	Tests the DMUL operation.
22	ddiv_nan.code	Tests DDIV, including NaN cases.
23	drem_nan.code dmul_nan.code	Test DREM and DMUL, including NaN cases.
24	fmul_nan.code frem_nan.code fdiv_nan.code	Test FMUL, FREM, FDIV, including NaN cases.
25	divbypass.code mulbypass_1.code rembypass.code	Test bypass cases for DDIV, DMUL, and DREM.
26	DREM_010.code DREM_011.code DREM_012.code DREM_013.code DREM_014.code	Test the DREM operation.
27	FREM_010.code FREM_011.code FREM_012.code FREM_013.code FREM_014.code	Test the FREM operation.

TABLE 9-1 Functional Tests for the FPU *(Continued)*

Number	Name	Description
28	DDIV_010.code DDIV_011.code DDIV_012.code DDIV_013.code DDIV_014.code DDIV_015.code	Test the DDIV operation.
29	FDIV_010.code FDIV_011.code FDIV_012.code FDIV_013.code FDIV_014.code FDIV_015.code	Test the FDIV operation.
30	FMUL_010.code FMUL_011.code FMUL_012.code FMUL_013.code FMUL_014.code FMUL_015.code	Test the FMUL operation.
31	DMUL_010.code DMUL_011.code DMUL_012.code DMUL_013.code DMUL_014.code DMUL_015.code	Test the DMUL operation.
32	fmul_1.code	Tests FMUL and DMUL operation.

The above test cases verify the following FPU functionalities:

- Corner vectors such as not-a-numbers (NaNs), zeroes, and infinities
- Underflow and overflow
- Operations that result in rounding to zero
- Operations that result in rounding to the nearest mode
- Widening and narrowing conversions
- Operations that involve denormals
- Back-to-back FPU operations
- Bypassing of data logic for all FPU operations
- Holds while FPU operations are underway
- Interrupts while FPU operations are underway

CHAPTER **10**

Verification of the Data Cache Unit (DCU)

The picoJava-II Data Cache Unit (DCU) manages all requests to the data cache.

This chapter contains the following sections:

10.1 Test Coverage

TABLE 10-1 shows the verification coverage for the data cache.

TABLE 10-1 Test Coverage for the DCU

Scenario	Instructions	Description	Test Case

Alignment of load instructions	`load_ubyte`
	`load_byte`
	`load_short`
	`load_word`
	`load_char_oe`
	`load_short_oe`
	`load_word_oe`
	`load_word_index`
	`load_short_index`
	`load_char_index`
	`load_byte_index`
	`load_ubyte_index`
		Test all load instructions with the last byte address from 0 to F.	`exp_4_1_7_1.code` (exception verification plan)

Alignment of store instructions	`store_byte`
	`store_short`
	`store_word`
	`store_short_oe`
	`store_word_oe`
	`store_word_index`
	`nastore_word_index`
	`store_short_index`
	`store_byte_index`
		Test all store instructions with the last byte address from 0 to F.	`exp_4_1_7_2.code` (exception verification plan)

Alignment of cache operations	`zero_line`
	`cache_flush`
	`cache_index_flush`
		Test all cache operation instructions with the last byte address from 0 to F.	`dcu_2_3_2.code`

Loading of instructions with different endian modes	`load_ubyte`
	`load_byte`
	`load_char`
	`load_short`
	`load_word`
	`load_char_oe`
	`load_short_oe`
	`load_word_oe`
	`load_word_index`
	`load_short_index`
	`load_char_index`
	`load_byte_index`
	`load_ubyte_index`
		Test all load instructions with different endian modes.	`dcu_2_4_2.code`

TABLE 10-1 Test Coverage for the DCU *(Continued)*

Scenario	Instructions	Description	Test Case

Storing of instructions with different endian modes	`store_byte`
	`store_short`
	`store_word`
	`store_short_oe`
	`store_word_oe`
	`store_load_index`
	`nastore_word_index`
	`store_short_index`
	`store_byte_index`
		Test all store instructions with different endian modes.	`dcu_2_5_2.code`

Cache operation instructions with different endian modes	`zero_line`
	`cache_flush`
	`cache_index_flush`
		Test all cache operation instructions with big endian mode.	`dcu_2_6.code`

Loading with cache hits		Test load cache hits.	`dcu_2_7.code`
Storing with cache hits		Test store cache hits.	`dcu_2_8.code`
Cache operations with cache hits		Test a ache operation with a cache hit.	`dcu_2_9_1.code`
Loading with cache misses		Test a load with a cache miss to a clean line; ensure that a cache line fill occurs.	`dcu_2_7.code`
Storing with cache misses		Test a store with a cache miss to a clean line; ensure that a cache line fill occurs.	`dcu_2_8.code`

Cache operations with cache misses	`zero_line`
	`cache_flush`
	`cache_index_flush`
		Test a cache operation with a cache miss to a clean line; ensure that a cache line fill occurs.	`dcu_2_9_1.code`

Copybacks due to loads	`load_word`	Tests copyback caused by a load.	`dcu_2_13_2.code`
Copybacks due to stores	`store_word`	Tests copyback caused by a store.	`dcu_2_14_2.code`

Copybacks due to cache operations	`zero_line`
	`cache_flush`
	`cache_index_flush`
		Test copyback caused by a cache operation.	`dcu_2_15_2.code`

Update of the dirty bit by loads	`load_word`	Tests that a load updates the dirty bit.	`dcu_2_16_2.code`

TABLE 10-1 Test Coverage for the DCU *(Continued)*

Scenario	Instructions	Description	Test Case
Update of the dirty bit by stores	`store_word`	Tests that a store updates the dirty bit.	`dcu_2_17_2.code`
Nonupdate of the dirty bit by cache operations	`zero_line` updates a dirty bit; `cache_flush` and `cache_index_flush` do *not*.	Test that cache operations may update the dirty bit.	`dcu_2_18_2.code`
Nonupdate of the LRU bit by loads	`load_byte`	Tests that a load does *not* update the LRU bit.	`dcu_2_19_2.code`
Update of the LRU bit by stores	`store_short`	Tests that a store updates the LRU bit.	`dcu_2_20_2.code`
Update of the LRU bit by cache operations	`zero_line` `cache_flush` `cache_index_flush`	Test that a cache operation updates the LRU bit.	`dcu_2_21_2.code`
Loading with a disabled cache	`load_word`	Tests that a disabled cache acts like a noncacheable load.	`dcu_2_22_2.code`
Storing with a disabled cache	`store_word`	Tests that a disabled cache acts like a noncacheable store.	`dcu_2_23_2.code`
Cache operations with a disabled cache	`zero_line` `cache_flush` `cache_index_flush`	Test that cache flushes act like `nop` and trap on `zero_line`.	`dcu_2_24_2.code`

10.2 Functional Tests

This section describes the functional tests for the DCU.

10.2.1 Arbiter

To ensure correct arbiter operations, the test checks that the pipeline request has a higher priority than the dribbling manager’s request.

10.2.2 Address Control

The tests verify the following for address control:

- The LRU bit is updated on a cache line fill (`dcu_2_20_2.code`).
- The LRU bit is *not* updated on a load hit (`dcu_3_2_1.code`).
- The LRU bit is updated on a store hit in the cache (`dcu_3_2_2.code`).

10.2.3 Aligner Control

The tests verify aligner control as follows:

- Test traps for misaligned loads, stores and cache operations (see Chapter 12, *Traps and Interrupts*).
- Test at random different `pj_ack`<2:0> values returned from the BIU.

10.2.4 Miss Control

The tests check for misses randomly with an external interrupt equal to 1 while the DCU is performing a line fill. A line fill completes prior to acceptance of the external interrupt.

10.2.5 Writeback Control

The tests verify writebacks as follows:

- Test that when a miss is detected, the line to be replaced is determined by the invalid bit before the LRU bit is checked (`dcu_3_5_1.code`).
- Test that if both entries are valid, then the LRU bit is used to determine which line is to be replaced (`dcu_3_2_1.code` and `dcu_3_2_2.code`).
- Test that if both entries are invalid, then the LRU bit is *not* used to determine which line is to be replaced, and that the first line is chosen (`dcu_3_2_1.code` and `dcu_3_2_2.code`).
- Test that in case of a miss while the DCU is performing a copyback, the cache line fill waits for the writeback to be complete before it starts its process (`dcu_3_5_3.code`).
- Test randomly with an external interrupt equal to 1 while the DCU is performing a copyback. Verify that the copyback is complete prior to acceptance of the external interrupt.

- Test with a new request to the same dirty line while the latter is performing a writeback. Verify that the DCU waits for the writeback to be completed and then starts another cache line fill (`dcu_3_5_2.code`).
- Test back-to-back load and store instructions to the same line (`dcu_3_5_3.code`).

10.2.6 Data Cache Datapath

All test cases test this datapath.

10.3 Tests for Noncacheable Loads

Noncacheable loads force a cache miss and send a noncacheable request to the bus. Once available, the data are bypassed to the pipeline.

The tests are as follows:

- Test the access by byte, half-word, and word load instructions to different positions of a cache line (`exp_4_1_7_3.code`, read exception verification plan). The last byte of the address ranges from 0 to F.
- Test the little and big endian for noncacheable load instructions (`dcu_4_0_4.code`).
- Test a noncacheable load and verify that the data are not written into the data cache (`dcu_4_0_6.code`).
- Test a noncacheable load hit to a dirty line in the cache (`dcu_4_0_8.code`).

10.4 Tests for Noncacheable Stores

The tests for noncacheable stores are as follows:

- Test byte, half-word, and word store instructions access to different positions of a cache line (`exp_4_1_7_4.code` (read exception verification plan). The last byte of the address ranges from 0 to F.
- Test the big endian for noncacheable store instructions (`dcu_4_1_4.code`).
- Test a noncacheable store and verify that the DCU does not write the data into the data cache (`dcu_4_1_6.code`).

- Test a noncacheable store, followed by a cacheable load miss to another clean line. The DCU starts a cache line fill during the transaction of the noncacheable store (dcu_4_1_7.code).
- Test a noncacheable store, followed by a cacheable load miss to another dirty line. The DCU waits for the noncacheable store to finish before the new cache line fill starts (dcu_4_1_8.code).

10.5 Tests for Zeroing Out of Cache Lines

The tests for zeroing out of cache lines are as follows:

- Test zeroing a dirty line, followed by another request to the same line that was zeroed during the writeback transaction. It is a cache hit with 0 value data. Verify that the DCU forwards zeroes to the pipeline instead of the writeback data (dcu_4_2_1.code) (zero_line).
- Test zeroing a dirty line, followed by another miss request to the same index line during the writeback transaction. The DCU stalls the new request. Verify that the zeroes write back to the main memory (dcu_4_2_2.code) (zero_line).
- Test with a zero_line to both ways with the same tag (only one is valid) (dcu_4_2_3.code).

10.6 Tests for Flushing of Cache Comparisons

Flush comparisons accommodate self-modifying code. The tests are as follows:

- Test that flushing occurs *only* when the tag is hit (dcu_4_3_1.code) (cache_flush).
- Test that the DCU changes the LRU bit (dcu_4_3_2.code).
- Test with a new request to the same flushed line, which is dirty, and verify that the DCU performs a copyback for it. The DCU stalls the new request until the copyback and invalidation operations are complete (dcu_4_3_3.code).
- Test with cache_flush to both ways with the same tag; only one is valid (dcu_4_3_4.code).

10.7 Tests for Cache-Indexed Flushing

The tests for cache-indexed flushing are as follows:

- Test that indexed flushing does not compare the tag (`dcu_4_4_1.code`).
- Test with a new request to the same flushed line, which is dirty, and verify that the DCU performs a copyback for it. The DCU stalls the new request until the copyback and invalidation operations are complete (`dcu_4_4_2.code`).

10.8 Tests for Disabling of the Data Cache

The tests for disabling of the data cache are as follows:

- Test with a store to a clean line to flush and disable the DCU. Next, send a load request to the same address (`dcu_4_5_2.code`).
- Test with stores set to 0 and 1 with the same index, followed by another store with the same index to create a copyback from set 0. Next, disable the data cache and check if the `load_word` instruction has copied back the correct data from set 0 (`dcu_4_5_3.code`).

10.9 Tests for Diagnostic Reads and Writes

The tests check the diagnostic read and write to a different index and word. Use `dcu_4_6_1.code` for data cache data; use `dcu_4_6_2.code` for data cache tags.

10.10 Tests for Invalidation of the Cache

If the address is present in the data cache, the DCU invalidates it. However, if the data cache is not enabled (`PSR.DCE = 0`), the DCU ignores the invalidation request.

The tests for cache invalidation are as follows:

- Test cache invalidation with `DCE` = 0 and `DCE` = 1
  - `dcu_4_7_1.code` (`PSR.DCE` = 1)
  - `dcu_4_7_2.code` (`PSR.DCE` = 0)

- Test with cache_invalidate both ways with the same tag; only one is valid (dcu_4_7_3.code).
- Test with cache_invalidate and nastore_word_index (dcu_4_9_1.code).

10.11 Tests for Dispatches of Load or Store Instructions

Load or store instruction dispatches occur before the DCU completes the trapping instruction. Afterward, the DCU cancels these dispatches.

Test with load or store instructions before and after an lmul instruction (dcu_4_8_1.code).

10.12 Other Data Cache Tests

Other data cache tests include:

- Test for loading from a bad memory location (dcu_4_10_1.code).
- Test for cache_flush (dcu_flush.code).
- Test for cache invalidating flushing (dcu_inv.code).
- Test for cache_index_flush (dcu_ind_flush.code).
- Test for zeroing of cache lines (dcu_zero.code).
- Test for cache_index_flush that automatically adjusts to different cache size configurations (dcu_auto_ind_flush).
- Test for disabling the data cache; data previously written in the cache should still be present (diagnostic_nocache.code).
- Test for reading data from the cache that has been disabled. Read from both sets (diagnostic_nocache_1.code).

CHAPTER 11

Verification of the Stack Manager Unit (SMU)

The Stack Manager Unit (SMU) performs the following functions:

- Stores and provides the necessary operands to the Integer Unit (IU)
- Handles overflow and underflow conditions of the stack cache
- Stalls the pipeline in an overflow or underflow condition

TABLE 11-1 describes the SMU tests, which cover all the basic and corner-case SMU operations.

TABLE 11-1 Functional Tests for the SMU

Number	Name	Description
1	`push_pop_1.code`	256 pushes, 256 pops. This test verifies the normal conditions of the Stack Manager. In filling up the stack cache with 256 entries, the Dribble Manager fills and spills entries according to the settings of the high and low watermarks. The pipeline halts when the stack cache is full; `num_entries` contains the number of entries on the stack. SMU then writes data from the stack cache back to the data cache.
2	`push_pop_2.code`	Two pushes, one pop, repeat. This test performs in a similar way as the previous test but fills up the stack cache more slowly to allow the Dribble Manager time to spill entries. It runs until the stack entries reach a maximal number, and then it pops twice for every push until the stack is empty.
3	`push_pop_3.code`	256 pops, 256 pushes. This test functions exactly the same as the previous two tests but focuses on fill and underflow conditions. It pops entries off an empty stack and causes the Dribble Manager to fill new entries.

TABLE 11-1 Functional Tests for the SMU *(Continued)*

Number	Name	Description
4	`push_pop_4.code`	Several pops, then move `OPTOP` suddenly. This test executes several pops until the dribbler starts to fill entries. Then, it resets `OPTOP` to a higher value for a sudden stall of the pipeline. Since the filling takes place in the background, this sudden `OPTOP` move causes an unusual condition to occur.
5	`push_pop_5.code`	Several pops, then move `OPTOP` suddenly the other way. A variant of the previous test, this test resets `OPTOP` to a lower value, which causes an immediate overflow as the dribbler fills cache entries.
6	`push_pop_6.code`	256 pushes with `ncstore_byte` mixed in, 256 pops with more `ncstore`s. This test, similar to others, performs a series of repeated pushes or pops, which are intermixed with `ncstore_byte` operations to a single address in the scratch area. The target is the interaction between the SMU and the IU.
7	`push_pop_7.code`	256 `dup`s, then 256 `store_byte`/`dup`s. This test uses `dup` instructions instead of push instructions to fill up the stack cache faster and activates `dribble_stall`.
8	`push_pop_8.code`	This test is similar to the previous one but aims at the case where a store and the dribbling out of an address occur at the same time.
9	`push_pop_9.code`	This test is similar to the previous one but aims at the case where a store and the dribbling in of an address occur at the same time.
10	`read_write_1.code`	This test utilizes three read ports and two write ports simultaneously. It performs operations to the stack contents continuously to exercise these ports to verify that they can function simultaneously. Simultaneous read and write operations cannot occur to the same location in memory at the same time. However, you can force three reads and two writes by, for example, performing an addition operation (two reads, one write) while dribbling data in the background (one read, one write).

TABLE 11-1 Functional Tests for the SMU *(Continued)*

Number	Name	Description
11	`ovr_und_1.code`	This test generates an overflow or underflow condition by moving `OPTOP` and `SC_BOTTOM`. It causes a stack overflow by resetting the `OPTOP` register to a lower value. An overflow occurs when `SC_BOTTOM` > 60 + `OPTOP`; an underflow occurs when `SC_BOTTOM` < = `OPTOP` + 6. This test verifies that the pipeline has been stalled, that every entry in the stack cache is copied back to the data cache, and that `SC_BOTTOM` is reset to `OPTOP`. The SMU releases the pipe only after there are at least six entries in the cache.
12	`fib_1.code`	This test stresses the SMU by invoking a recursive function to calculate Fibonacci numbers. Since the recursive definition of the Fibonacci function depends on the results from the previous computation, this test verifies the maintenance of stack frames.
13	`fib_2.code`	This test is similar to the previous one. The only difference is that it initializes local variables in each stack frame.
14	`invoke_return_1.code`	This test performs a number of nested invokes followed by the same number of returns. It allocates a number of stack frames (which you can specify) through which it then returns values.
15	`invoke_return_2.code`	This test is similar to the previous one. The only difference is that it also allocates some local variables.
16	`invoke_hold.jasm`	This test performs a microcode invoke operation after the SMU has released the IU pipeline and executes successive `dup` commands. The instruction that follows an overflow is `invokespecial_quick`, `invokestatic_quick`, `invokevirtual_quick`, or `invokeinterface_quick`.
17	`load_store_index.code`	This test verifies local variable store misses and `load_word` bypasses in the same cycle.

CHAPTER **12**

Traps and Interrupts

Exceptions in picoJava-II operations are caused by:

- A subset of the Java virtual machine instructions which the picoJava-II core emulates, using software
- picoJava-specific exceptions, such as runtime exceptions and hardware errors
- Hardware-generated interrupts

This chapter contains the following sections:

For information on trap types and trap priorities, see the *picoJava-II Programmer's Reference Manual*.

12.1 Exception Handlers

When the picoJava-II core takes a trap for any reason, it sets up a trap frame and transfers control to the trap handler for that trap type.

The picoJava-II core verification environment provides reference implementations of Java virtual machine bytecodes which cause an emulation trap. These trap handlers are provided for the purpose of verification only.

The verification environment also provides reference implementations of handlers for other traps (such as exceptions and interrupts) for the purpose of verification. By default, these trap handlers simply abort the test that is running. However, they also have a mode to count how many times an exception occurred during a test. This mode is useful for testing exceptions. In this mode, the trap handlers, instead of aborting the program, simply increment a counter in a predefined memory area and return to the test program to continue execution. At the end of the test, the test program checks these counters and compares the result with the expected number of exceptions of each type.

The counters for each trap type are stored in memory from location 0xf000 onward, one word for each trap type (including unused trap types). The address of the counter for a particular trap type is (0xf000 + trap_type * 4).

To use the mode where the exception handlers count the number of times the exception occurred (instead of aborting the program), specify the `+expcount` option. This option stores the value 0xf0 to memory location 0xfff0. Exception handlers check this location to decide whether to increment the counter and return to the test, or abort it.

Note – When tests are run only on the simulator, this memory location must be explicitly initialized when the mode in which exception handlers count the number of exceptions is used.

To initialize this memory location, enter the following command:

```
% memPoke 0xfff0 0xf0
```

Note that cosimulation between the RTL and the simulator continues during the time the trap is taken and while the exception handler is being executed. However, the cosimulation environment cannot handle asynchronous exceptions, because it is nondeterministic when the exception will be reported in the RTL. This makes it difficult to compare state with the simulator after every instruction. For tests that exercise such exceptions, you can disable cosimulation and instead depend on self-checking tests running on the RTL standalone.

12.2 Traps and Exceptions

Exceptions fall into a number of categories, as defined in the following sections, which also list their test cases.

12.2.1 Power-On Reset (POR)

Triggered by an external reset request, a power-on reset (POR) causes a transfer of control to address 0. When a POR is active, the picoJava-II core ignores all other resets and traps.

You should test at the beginning of all test cases.

12.2.2 Asynchronous Error

In case of a store error or an SMU error, the picoJava-II core goes into the error state and signals itself to take an `asynchronous_error` trap at an offset of 0x8 from the TBA (Trap Base Address).

The test cases are described in TABLE 12-1.

TABLE 12-1 Tests for Asynchronous Errors

Test Case	Description
`exp_4_1_1_2.code`	Tests with stores to an erroneous cacheable memory location. An asynchronous error occurs.
`exp_4_1_1_3.code`	Tests with a `write_optop` instruction to an erroneous cacheable memory location. An asynchronous error occurs.
`exp_4_1_1_4.code`	Tests with a store to a local variable that is an erroneous cacheable memory location. An asynchronous error occurs.
`exp_4_1_1_7.code`	Tests with a store to an erroneous cacheable memory location when `PSR.AEM` = 1. The picoJava-II core should take no exceptions.
`exp_4_1_1_8.code`	Tests with a writeback to an erroneous cacheable memory location. An asynchronous error occurs.
`exp_4_1_1_9.code`	Tests with a `write_optop` instruction to an erroneous noncacheable location. An asynchronous error occurs.
`exp_4_1_1_10.code`	Tests with stores to an erroneous noncacheable location. An asynchronous error occurs.
`exp_4_1_1_11.code`	Tests with a store to a local variable that is an erroneous noncacheable location. An asynchronous error occurs.
`exp_4_1_1_12.code`	Tests with a store to an erroneous noncacheable location when `PSR.AEM` = 1. The picoJava-II core should take no exceptions.
`async_fold`	Tests arrival of asynchronous error during execution of folded group.

A table in Chapter 8, “Powerdown, Clock, Reset, and Scan Unit, (PCSU),” in the *picoJava-II Microarchitecture Guide* lists the machine states for asynchronous errors. This trap disturbs the minimum state.

12.2.3 Data Access Memory Errors

`data_access_mem_error` is an error exception that occurs on a data load from memory.

The test cases are described in TABLE 12-2.

TABLE 12-2 Tests for Data Access Memory Errors

Test Case	Description
`exp_4_1_1_1.code`	Tests with loads to an erroneous memory location.
`exp_4_1_1_5.code`	Tests with a `putstatic_quick` instruction to an erroneous memory location. A data access memory error exception occurs.
`exp_4_1_1_6.code`	Tests with a `tableswitch` instruction with some of its opcodes in erroneous locations. A data access memory error exception occurs.

12.2.4 Instruction Access Memory Errors

`instruction_access_mem_error` is an error exception that occurs on an instruction access from memory or I/O.

The test cases are described in TABLE 12-3.

TABLE 12-3 Tests for Instruction Access Memory Errors

Test Case	Description
`exp_4_1_2_1.code`	Tests with `write_pc` to an erroneous memory location.
`exp_4_1_2_2.code`	Tests with `write_pc` to an erroneous I/O location.
`exp_4_1_2_3.code`	Tests with `write_pc` at the border of good memory and bad memory and verifies that an instruction access memory error exception occurs afterward in bad memory.
`exp_4_1_2_4.code`	Tests with a `tableswitch` instruction in an erroneous location.

12.2.5 Privileged Instructions

`privileged_instruction` is an error that results from the execution of a privileged instruction when `PSR.SU` is equal to 0.

The test case is described in TABLE 12-4.

TABLE 12-4 Test for Privileged Instructions

Test Case	Description
exp_4_1_3_1.code	Tests with PSR.SU = 0 and executes all 42 privileged instructions to verify that the priv_reset instruction causes a privileged instruction exception instead of a reset exception.

12.2.6 Illegal Instructions

illegal_instruction is an error that results from the execution of an instruction with an unimplemented or reserved opcode.

The test cases are described in TABLE 12-5.

TABLE 12-5 Tests for Illegal Instructions

Test Case	Description
exp_4_1_4_1.code	Tests with all unimplemented and reserved opcodes (two bytes opcode and PSR.SU = 1).
exp_4_1_4_2.code	Tests with all unimplemented and reserved opcodes (two bytes opcode and PSR.SU = 0).

12.2.7 breakpoint1

breakpoint1 is a trap that occurs when either an instruction fetch memory address or a load or store data memory address matches the address in the Breakpoint1 address register.

The test cases are described in TABLE 12-6.

TABLE 12-6 Tests for `breakpoint1`

Test Case	Description
`exp_4_1_5_1.code` `(PSR.SU = 1, SUBRK1 = 0)`	Test with a load data memory address that matches the `Breakpoint1` address register.
`exp_4_1_5_2.code` `(PSR.SU = 1, SUBRK1 = 1)`	
`exp_4_1_5_3.code` `(PSR.SU = 0, SUBRK1 = 0, BRKM1 = b'1111111)`	
`exp_4_1_5_4.code` `(PSR.SU = 0, SUBRK1 = 0, BRKM1 = b'1111110)`	
`exp_4_1_5_5.code` `(PSR.SU = 0, SUBRK1 = 0, BRKM1 = b'1111100)`	
`exp_4_1_5_6.code` `(PSR.SU = 0, SUBRK1 = 0, BRKM1 = b'1111000)`	
`exp_4_1_5_7.code` `(PSR.SU = 0, SUBRK1 = 0, BRKM1 = b'1110000)`	
`exp_4_1_5_8.code` `(PSR.SU = 0, SUBRK1 = 0, BRKM1 = b'1100000)`	
`exp_4_1_5_9.code` `(PSR.SU = 0, SUBRK1 = 0, BRKM1 = b'1000000)`	
`exp_4_1_5_10.code` `(PSR.SU = 0, SUBRK1 = 1, BRKM1 = b'1111111)`	
`exp_4_1_5_11.code` `(PSR.SU = 0, SUBRK1 = 1, BRKM1 = b'1111110)`	
`exp_4_1_5_12.code` `(PSR.SU = 0, SUBRK1 = 1, BRKM1 = b'1111100)`	
`exp_4_1_5_13.code` `(PSR.SU = 0, SUBRK1 = 1, BRKM1 = b'1111000)`	
`exp_4_1_5_14.code` `(PSR.SU = 0, SUBRK1 = 1, BRKM1 = b'1110000)`	
`exp_4_1_5_15.code` `(PSR.SU = 0, SUBRK1 = 1, BRKM1 = b'1100000)`	
`exp_4_1_5_16.code` `(PSR.SU = 0, SUBRK1 = 1, BRKM1 = b'1000000)`	

TABLE 12-6 Tests for `breakpoint1` *(Continued)*

Test Case	Description
`exp_4_1_5_17.code` `(PSR.SU = 1, SUBRK1 = 0)`	Test with a store data memory address that matches the `Breakpoint1` address register.
`exp_4_1_5_18.code` `(PSR.SU = 1, SUBRK1 = 1)`	
`exp_4_1_5_19.code` `(PSR.SU = 0, SUBRK1 = 0, BRKM1 = b'1111111)`	
`exp_4_1_5_20.code` `(PSR.SU = 0, SUBRK1 = 0, BRKM1 = b'1111110)`	
`exp_4_1_5_21.code` `(PSR.SU = 0, SUBRK1 = 0, BRKM1 = b'1111100)`	
`exp_4_1_5_22.code` `(PSR.SU = 0, SUBRK1 = 0, BRKM1 = b'1111000)`	
`exp_4_1_5_23.code` `(PSR.SU = 0, SUBRK1 = 0, BRKM1 = b'1110000)`	
`exp_4_1_5_24.code` `(PSR.SU = 0, SUBRK1 = 0, BRKM1 = b'1100000)`	
`exp_4_1_5_25.code` `(PSR.SU = 0, SUBRK1 = 0, BRKM1 = b'1000100)`	
`exp_4_1_5_26.code` `(PSR.SU = 0, SUBRK1 = 0, BRKM1 = b'1111111)`	
`exp_4_1_5_27.code` `(PSR.SU = 0, SUBRK1 = 1, BRKM1 = b'1111110)`	
`exp_4_1_5_28.code` `(PSR.SU = 0, SUBRK1 = 1, BRKM1 = b'1111100)`	
`exp_4_1_5_29.code` `(PSR.SU = 0, SUBRK1 = 1, BRKM1 = b'1111000)`	
`exp_4_1_5_30.code` `(PSR.SU = 0, SUBRK1 = 1, BRKM1 = b'1110000)`	
`exp_4_1_5_31.code` `(PSR.SU = 0, SUBRK1 = 1, BRKM1 = b'1100000)`	
`exp_4_1_5_32.code` `(PSR.SU = 0, SUBRK1 = 1, BRKM1 = b'1000000)`	
`exp_4_1_5_33.code`	Tests with a fetching address that matches the `Breakpoint1` address register.
`exp_4_1_5_34.code`	Tests with a fetching address that matches the `Breakpoint1` address register, with the folding bit on.

TABLE 12-6 Tests for breakpoint1 *(Continued)*

Test Case	Description
exp_4_1_5_35.code	Tests the halt mode with a load_byte instruction that causes a data breakpoint1 exception.
exp_4_1_5_36.code	Tests the halt mode with a store_byte instruction that causes a data breakpoint1 exception.
exp_4_1_5_37.code	Tests the halt mode with an instruction breakpoint1 exception.
exp_4_1_5_38.code	Tests with a zero_line instruction that causes a data breakpoint1 exception with SRCBRK1 = 0.
exp_4_1_5_39.code	Tests with a cache_invalidate instruction that causes a data breakpoint1 exception with SRCBRK1 = 0.
exp_4_1_5_40.code	Tests with a zero_line instruction that causes a data breakpoint1 exception with SRCBRK1 = 1.
exp_4_1_5_41.code	Tests with a cache_invalidate instruction that causes a data breakpoint1 exception with SRCBRK1 = 1.
data_brk_1 data_brk_2 data_brk_3 data_brk_4	Test data breakpoints during stack cache miss for data load.
data_brk_halt_1 data_brk_halt_2 data_brk_halt_3 data_brk_halt_4	Test data breakpoints in halt mode.

12.2.8 breakpoint2

breakpoint2 is a trap that occurs when either an instruction fetch memory address or load or store data memory address matches the address in the Breakpoint2 address register.

The test cases are described in TABLE 12-7.

TABLE 12-7 Tests for `breakpoint2`

Test Case	Description
`exp_4_1_6_1.code` (PSR.SU = 1, SUBRK1 = 0)	Test with a load data memory address that matches the `Breakpoint2` address register.
`exp_4_1_6_2.code` `(PSR.SU = 1, SUBRK1 = 1)`	
`exp_4_1_6_3.code` `(PSR.SU = 0, SUBRK1 = 0, BRKM1 = b'1111111)`	
`exp_4_1_6_4.code` `(PSR.SU = 0, SUBRK1 = 0, BRKM1 = b'1111110)`	
`exp_4_1_6_5.code` `(PSR.SU = 0, SUBRK1 = 0, BRKM1 = b'1111100)`	
`exp_4_1_6_6.code` `(PSR.SU = 0, SUBRK1 = 0, BRKM1 = b'1111000)`	
`exp_4_1_6_7.code` `(PSR.SU = 0, SUBRK1 = 0, BRKM1 = b'1110000)`	
`exp_4_1_6_8.code` `(PSR.SU = 0, SUBRK1 = 0, BRKM1 = b'1100000)`	
`exp_4_1_6_9.code` `(PSR.SU = 0, SUBRK1 = 0, BRKM1 = b'1000000)`	
`exp_4_1_6_10.code` `(PSR.SU = 0, SUBRK1 = 1, BRKM1 = b'1111111)`	
`exp_4_1_6_11.code` `(PSR.SU = 0, SUBRK1 = 1, BRKM1 = b'1111110)`	
`exp_4_1_6_12.code` `(PSR.SU = 0, SUBRK1 = 1, BRKM1 = b'1111100)`	
`exp_4_1_6_13.code` `(PSR.SU = 0, SUBRK1 = 1, BRKM1 = b'1111000)`	
`exp_4_1_6_14.code` `(PSR.SU = 0, SUBRK1 = 1, BRKM1 = b'1110000)`	
`exp_4_1_6_15.code` `(PSR.SU = 0, SUBRK1 = 1, BRKM1 = b'1100000)`	
`exp_4_1_6_16.code` `(PSR.SU = 0, SUBRK1 = 1, BRKM1 = b'1000000)`	

TABLE 12-7 Tests for `breakpoint2` *(Continued)*

Test Case	**Description**
`exp_4_1_6_17.code` `(PSR.SU = 1, SUBRK1 = 0)`	Test with a store data memory address that matches the `Breakpoint2` address register.
`exp_4_1_6_18.code` `(PSR.SU = 1, SUBRK1 = 1)`	
`exp_4_1_6_19.code` `(PSR.SU = 0, SUBRK1 = 0, BRKM1 = b'1111111)`	
`exp_4_1_6_20.code` `(PSR.SU = 0, SUBRK1 = 0, BRKM1 = b'1111110)`	
`exp_4_1_6_21.code` `(PSR.SU = 0, SUBRK1 = 0, BRKM1 = b'1111100)`	
`exp_4_1_6_22.code` `(PSR.SU = 0, SUBRK1 = 0, BRKM1 = b'1111000)`	
`exp_4_1_6_23.code` `(PSR.SU = 0, SUBRK1 = 0, BRKM1 = b'1110000)`	
`exp_4_1_6_24.code` `(PSR.SU = 0, SUBRK1 = 0, BRKM1 = b'1100000)`	
`exp_4_1_6_25.code` `(PSR.SU = 0, SUBRK1 = 0, BRKM1 = b'1000100)`	
`exp_4_1_6_26.code` `(PSR.SU = 0, SUBRK1 = 0, BRKM1 = b'1111111)`	
`exp_4_1_6_27.code` `(PSR.SU = 0, SUBRK1 = 1, BRKM1 = b'1111110)`	
`exp_4_1_6_28.code` `(PSR.SU = 0, SUBRK1 = 1, BRKM1 = b'1111100)`	
`exp_4_1_6_29.code` `(PSR.SU = 0, SUBRK1 = 1, BRKM1 = b'1111000)`	
`exp_4_1_6_30.code` `(PSR.SU = 0, SUBRK1 = 1, BRKM1 = b'1110000)`	
`exp_4_1_6_31.code` `(PSR.SU = 0, SUBRK1 = 1, BRKM1 = b'1100000)`	
`exp_4_1_6_32.code` `(PSR.SU = 0, SUBRK1 = 1, BRKM1 = b'1000000)`	
`exp_4_1_6_33.code`	Tests with a fetching address that matches the `Breakpoint2` address register.
`exp_4_1_6_34.code`	Tests with a fetching address that matches the `Breakpoint2` address register, with the folding bit on.

TABLE 12-7 Tests for `breakpoint2` *(Continued)*

Test Case	Description
`exp_4_1_6_35.code`	Tests the halt mode with a `load_byte` instruction that causes a data `breakpoint2` exception.
`exp_4_1_6_36.code`	Tests the halt mode with a `store_byte` instruction that causes a data `breakpoint2` exception.
`exp_4_1_6_37.code`	Tests the halt mode with an instruction `breakpoint2` exception.
`exp_4_1_6_38.code`	Tests with a `zero_line` instruction that causes a data `breakpoint2` exception with `SRCBRK2` = 0.
`exp_4_1_6_39.code`	Tests with a `cache_invalidate` instruction that causes a data `breakpoint2` exception with SRCBRK2 = 0.
`exp_4_1_6_40.code`	Tests with a `zero_line` instruction that causes a data `breakpoint2` exception with `SRCBRK2` = 1.
`exp_4_1_6_41.code`	Tests with a `cache_invalidate` instruction that causes a data `breakpoint2` exception with `SRCBRK2` = 1.

12.2.9 Misalignment of Memory Addresses

`mem_address_not_aligned` is an error that occurs when a load or store instruction generates an address with which it is not properly aligned.

The test cases are described in TABLE 12-8.

TABLE 12-8 Tests for Memory Address Misalignment

Test Case	Description
exp_4_1_7_1.code (cacheable loads and load index instructions)	Test all load and store instructions with misaligned addresses.
exp_4_1_7_2.code (cacheable stores and store index instructions)	
exp_4_1_7_3.code (noncacheable stores)	
exp_4_1_7_4.code (noncacheable loads)	

12.2.10 Data Access I/O Errors

data_access_io_error is an error exception that occurs on a load data access from or to the I/O space.

Test with a load from an erroneous I/O (exp_4_1_8_1.code).

12.2.11 OPLIM Traps

An OPLIM trap occurs when an instruction causes OPTOP to be smaller than OPLIM. The trap handler must either grow the current stack area or allocate a new "chunk." When it returns from the trap, the IU must reexecute the trapped instruction.

The test cases are described in TABLE 12-9.

TABLE 12-9 Tests for OPLIM Traps

Test Case	Description
exp_4_1_9_1.code	Tests with write_optop.
exp_4_1_9_2.code	Tests with write_oplim.
exp_4_1_9_3.code	Tests priv_update_optop without an OPLIM trap during an atomic update, but with a new OPTOP and a new OPLIM that is greater than the old OPLIM.
exp_4_1_9_4.code	Tests priv_update_optop without an OPLIM trap during an atomic update, but with a new OPTOP and a new OPLIM that is smaller than the old OPLIM.
exp_4_1_9_5.code	Tests priv_update_optop with an OPLIM trap during an update.
exp_4_1_9_6.jasm	Tests invokenonvirtual_quick, which causes an OPLIM trap.

12.2.12 Soft Traps

Test with the `soft_trap` instruction (`exp_4_1_10_1.code`).

12.2.13 `ldiv`

Test with `ldiv` in the IU.

12.2.14 `lmul`

Test with `lmul` in the IU.

12.2.15 `lrem`

Test with `lrem` in the IU.

12.2.16 Runtime Arithmetic

`runtime_ArithmeticException` is an exceptional arithmetic situation, such as an integer division or remainder operation with zero.

Test with `idiv` and `irem` instructions (`exp_4_1_14_1.code`):

- To create this exception
- To take the following exceptions: `tt` = 0x6d, 0x71, which are `runtime_arithmetic` exceptions

12.2.17 Runtime Null Pointers

`runtime_NullPtrException` is an exception that occurs when a null reference instead of an object reference is used.

The test cases are described in TABLE 12-10.

TABLE 12-10 Tests for Runtime Null Pointers

Test Case	Description
exp_4_1_15_1.code	Tests with all aload instructions (a, b, c, d, i, l, f and s) to create runtime_NullPtrException.
exp_4_1_15_2.code	Tests with all astore instructions (a, b, c, d, i, l, f and s) to create runtime_NullPtrException.

12.2.18 Emulation of Zero Lines

The zero line emulation trap occurs when the zero_line instruction is executed when the data cache is off.

The test cases are described in TABLE 12-11.

TABLE 12-11 Tests for Zero Line Emulation

Test Case	Description
exp_4_1_16_1.code	Test a zero_line instruction with PSR.DCE = 0.
exp_4_1_16_2.code	Test a zero_line instruction with a noncacheable address and PSR.DCE = 1.

12.2.19 Emulation of FP Instructions

If either the FPE bit in the PSR or the FPP bit in the HCR is off, the Instruction Cache Unit (ICU) traps all floating-point (FP) instructions, which are:

fadd, dadd, fsub, dsub, fmul, dmul, fdiv, dddiv, frem, drem, i2f, i2l, l2f, l2d, f2i, f2l, d2i, d2l, f2d, d2f, fcmpg, fcmpl, dcmpg, and dcmpl

The test cases are described in TABLE 12-12.

TABLE 12-12 Tests for Emulation of FP Instructions

Test Case	Description
exp_4_1_17_1.code	Tests all FP instructions with PSR.FPE = 0 and HCR.FPP = 1.
exp_4_1_17_2.code	Tests all FP instructions with PSR.FPE = 1 and HCR.FPP = 1.
exp_4_1_17_4.code	Tests a drem trap with PSR.FPE/DRT = 1 and HCR.FPP = 1.

Note – FPU emulation handlers increment the trap counters and move on to the instructions that follow. No actual floating-point operations take place.

12.2.20 Breakpoint Handlers

The breakpoint handler exception occurs when the program counter (PC) register values matches the breakpoint address.

The test case is `exp_4_1_18_1.code`.

12.2.21 Unimplemented Instructions

Test all instructions that have not been implemented.

The test cases are described in TABLE 12-13.

TABLE 12-13 Tests for Unimplemented Instructions

Test Case	Description
`exp_4_1_19_1.code`	Tests with `PSR.SU` = 1.
`exp_4_1_19_2.code`	Tests with `PSR.SU` = 0.

12.2.22 Memory Protection Fault

The memory protection fault is a synchronous trap that occurs when the picoJava-II core accesses outside of the address ranges of the `USERRANGE1` and `USERRANGE2` registers.

Note – The picoJava-I core can take asynchronous traps when it accesses outside of the address ranges of the `URSERRANGE1` and `USERRANGE2` registers.

The test cases are described in TABLE 12-14.

TABLE 12-14 Tests for Memory Protection Faults

Test Case	Description
`exp_4_1_20_1.code` (load), `exp_4_1_20_2.code` (store)	Test all load and store instructions with `PSR.ACE` = 1 and `PSR.CAC` = 0 (outside `USERRANGE1` and `USERRANGE2`).
`exp_4_1_20_3.code` (load), `exp_4_1_20_4.code` (store)	Test all load and store instructions with `PSR.ACE` = 1 and `PSR.CAC` = 0 (outside `USERRANGE1` only).
`exp_4_1_20_5.code` (load), `exp_4_1_20_6.code` (store)	Test all load and store instructions with `PSR.ACE` = 1 and `PSR.CAC` = 0 (outside `USERRANGE2` only).
`exp_4_1_20_7.code` `exp_4_1_20_8.code`	Test all load and store instructions with `PSR.ACE` = 1 and `PSR.CAC` = 1 (outside `USERRANGE1` and `USERRANGE2`).
`exp_4_1_20_9.code` `exp_4_1_20_10.code`	Test all load and store instructions with `PSR.ACE` = 1 and `PSR.CAC` = 1 (outside `USERRANGE1` only).
`exp_4_1_20_11.code` `exp_4_1_20_12.code`	Test all load and store instructions with `PSR.ACE` = 1 and `PSR.CAC` = 1 (outside `USERRANGE2` only).
`exp_4_1_20_13.code`	Tests with a `putstatic_quick` access out of `USERRANGE`s.
`exp_4_1_20_14.code`	Tests with a `write_optop` access that is out of `USERRANGE`s.
`exp_4_1_20_15.code`	Tests with a `store_word_index` access that is out of `USERRANGE`s.
`exp_4_1_20_16.code`	Tests with a `zero_line` access that is out of `USERRANGE`s.
`exp_4_1_20_17.code`	Tests with a `cache_invalidate` access that is out of `USERRANGE`s.

12.2.23 Out-of-Bounds Runtime Index

`runtime_IndexOutOfBoundsException` is an out-of-range exception that is either an index or a subrange. It is specified by two indexes or by an index and a length.

The test cases are described in TABLE 12-15.

TABLE 12-15 Tests for Out-of-Bounds Runtime Index Exceptions

Test Case	Description
`exp_4_1_21_1.jasm`	Tests with all `aload` instructions (a, b, c, d, i, l, f, and s) to create `runtime_IndexOutOfBoundsException`.
`exp_4_1_21_2.jasm`	Tests with all `astore` instructions (a, b, c, d, i, l, f, and s) to create `runtime_IndexOutOfBoundsException`.

12.2.24 Lock Count Overflow Traps

The lock count overflow trap occurs if the `LOCKCOUNT` register overflows or underflows when incremented or decremented while entering or exiting a monitor.

The test cases are described in TABLE 12-16.

TABLE 12-16 Tests for Lock Count Overflow Traps

Test Case	Description
`exp_4_1_22_1.code`	Tests `LockCount`/`addr0` overflow and `LockCount`/`addr1` miss.
`exp_4_1_22_2.code`	Tests `LockCount`/`addr1` overflow and `LockCount`/`addr0` miss.
`exp_4_1_22_3.code`	Tests `LockCount`/`addr0` underflow and `LockCount`/`addr1` miss.
`exp_4_1_22_4.code`	Tests `LockCount`/`addr1` underflow and `LockCount`/`addr0` miss.

12.2.25 Lock Enter Miss Traps

The lock enter miss trap occurs if the monitor that is entering does not exist in either of the `LOCKADDR` registers (`Lockaddr0` and `Lockaddr1`).

The test case is `exp_4_1_23_1.code`.

12.2.26 Lock Exit Miss Traps

The lock exit miss trap occurs if the monitor that is exiting does not exist in either of the `LOCKADDR` registers (`Lockaddr0` and `Lockaddr1`).

The test case is `exp_4_1_24_1.code`.

12.2.27 Lock Release Traps

The lock release trap occurs if `LOCKCOUNT` equals 0 and the corresponding `LOCKWANT` bit is set.

The test cases are described in TABLE 12-17.

TABLE 12-17 Tests for Lock Release Traps

Test Case	Description
`exp_4_1_25_1.code`	Tests `Lockaddr0` and `Lockcount0`.
`exp_4_1_25_2.code`	Tests `Lockaddr1` and `Lockcount1`.
`exp_4_1_25_3.code`	Tests `Lockaddr0`, `Lockaddr1`, `Lockcount0`, and `Lockcount1`.

12.2.28 Garbage Collection Notify Traps

The `gc_notify` trap occurs when a copy is made of object references that the garbage collector has not yet processed. For details, see the chapter on garbage collection in the *picoJava-II Programmer's Reference Manual.*

The test cases are described in TABLE 12-18.

TABLE 12-18 Tests for Garbage Collection Notify Traps

Test Case	Description
`exp_4_1_26_1.code`	Tests `aputstatic_quick` and `aputfield_quick` with `GCE` = 0.
`exp_4_1_26_2.code`	Tests `aputstatic_quick` and `aputfield_quick` with `GCE` = 1.

12.2.29 Trap Priority Tests for Two Exceptions

TABLE 12-19 through TABLE 12-27 describe the trap priority tests for two exceptions.

TABLE 12-19 Tests for Memory Protection with Other Exceptions

Test Case	Description
`exp_5_5_1_1.code`	Tests memory protection with a data access memory error.
`exp_5_5_1_2.code`	Tests memory protection with `breakpoint1`.
`exp_5_5_1_3.code`	Tests memory protection with `breakpoint2`.
`exp_5_5_1_4.code`	Tests memory protection with a misaligned error.
`exp_5_5_1_5.code`	Tests memory protection with `data_access_io`.
`exp_5_5_1_6.code`	Tests memory protection with `OPLIM`.

TABLE 12-20 Tests for `breakpoint1` with Other Exceptions

Test Case	Description
`exp_5_5_2_1.code`	Tests `breakpoint1` with `breakpoint2`.
`exp_5_5_2_2.code`	Tests `breakpoint1` with `instruction_access_error`. Needs a special `breakpoint1.code`.
`exp_5_5_2_3.code`	Tests `breakpoint1` with `illegal_instruction`.
`exp_5_5_2_4.code`	Tests `breakpoint1` with `privileged_instruction`.
`exp_5_5_2_5.code`	Tests `breakpoint1` with `OPLIM`.
`exp_5_5_2_6.code`	Tests `breakpoint1` with `mem_address_not_aligned`.
`exp_5_5_2_7.code`	Tests `breakpoint1` with `data_access_mem_error`.
`exp_5_5_2_8.code`	Tests `breakpoint1` with `data_access_io_error`.
`exp_5_5_2_9.code`	Tests `breakpoint1` with `fpu-etrap`.
`exp_5_5_2_10.code`	Tests `breakpoint1` with `unimplemented_instr_0xed`.
`exp_5_5_2_11.code`	Tests `breakpoint1` with `ZeroLineEmulation`
`exp_5_5_2_12.code`	Tests `breakpoint1` with `runtime_Arithmetic`.
`exp_5_5_2_13.code`	Tests `breakpoint1` with `LockCountOverflowTrap`.
`exp_5_5_2_14.code`	Tests `breakpoint1` with `gc_notify`.

TABLE 12-21 Tests for `breakpoint2` with Other Exceptions

Test Case	Description
`exp_5_5_3_1.code`	Tests `breakpoint2` with `breakpoint1`.
`exp_5_5_3_3.code`	Tests `breakpoint2` with `illegal_instruction`.
`exp_5_5_3_4.code`	Tests `breakpoint2` with `privileged_instruction`.
`exp_5_5_3_5.code`	Tests `breakpoint2` with `OPLIM`.
`exp_5_5_3_6.code`	Tests `breakpoint2` with `mem_address_not_aligned`.
`exp_5_5_3_7.code`	Tests `breakpoint2` with `data_access_mem_error`.
`exp_5_5_3_8.code`	Tests `breakpoint2` with `data_access_io_error`.
`exp_5_5_3_9.code`	Tests `breakpoint2` with `fpu-etrap`.
`exp_5_5_3_10.code`	Tests `breakpoint2` with `softtrap`, `ldiv`, and `unimplemented_instr_0xed`.
`exp_5_5_3_11.code`	Tests `breakpoint2` with `ZeroLineEmulation`.
`exp_5_5_3_12.code`	Tests `breakpoint2` with `runtime_Arithmetic`.
`exp_5_5_3_13.code`	Tests `breakpoint2` with `LockCountOverflowTrap`.
`exp_5_5_3_14.code`	Tests `breakpoint2` with `gc_notify`.

TABLE 12-22 Tests for Instruction Access Errors with Other Exceptions

Test Case	Description
`exp_5_5_4_1.code`	Tests an instruction access error with an illegal instruction.
`exp_5_5_4_2.code`	Tests an instruction access error with a privileged instruction.
`exp_5_5_4_3.code`	Tests instruction access error with `OPLIM`.

TABLE 12-23 Tests for `OPLIM` with Other Exceptions

Test Case	Description
`exp_5_5_5_1.code`	Tests illegal instruction and `OPLIM` trap.
`exp_5_5_7_1.code`	Tests `OPLIM` with `mem_address_not_aligned`.
`exp_5_5_7_2.code`	Tests `OPLIM` with `data_access_mem_error`.
`exp_5_5_7_3.code`	Tests `OPLIM` with `data_access_io_error`.
`exp_5_5_7_4.code`	Tests `OPLIM` with `soft_trap`.

TABLE 12-23 Tests for `OPLIM` with Other Exceptions

Test Case	Description
`exp_5_5_7_5.code`	Tests `OPLIM` with `runtime_NullptrException`.
`exp_5_5_7_6.code`	Tests `OPLIM` with `LockEnterMissTrap`.
`exp_5_5_7_7.code`	Tests `OPLIM` with `gc_notify`.

TABLE 12-24 Tests for `Mem_address_not_aligned` with Other Exceptions

Test Case	Description
`exp_5_5_8_1.code`	Tests `mem_address_not_aligned` with `data_access_mem_error`.
`exp_5_5_8_2.code`	Tests `mem_address_not_aligned` with `data_access_io_error`.

TABLE 12-25 Test for Data Access Memory Error with Other Exceptions

Test Case	Description
`exp_5_5_9_1.code`	Tests access memory error with `ZeroLineEmulation`.

TABLE 12-26 Test for Data Access I/O Error with Other Exceptions

Test Case	Description
`exp_5_5_10_1.code`	Tests access I/O error with `ZeroLineEmulation`.

TABLE 12-27 Test for `NullPtr` and `IndexOutOfBnd`

Test Case	Description
`exp_5_5_11_1.code`	Verifies that in all array loads, `Arrayref` is null and the index is negative.
`exp_5_5_11_2.code`	Verifies that in all array stores, `Arrayref` is null and the index is negative.

12.2.30 Trap Priority Tests with Three Exceptions

TABLE 12-28, TABLE 12-29, and TABLE 12-30 describe the trap priority tests for three exceptions.

TABLE 12-28 Tests for Memory Protection Errors with Other Exceptions

Test Case	Description
`exp_5_6_1_1.code`	Tests a memory protection error with `breakpoint1` and `breakpoint2`.
`exp_5_6_1_2.code`	Tests a memory protection error with `breakpoint1` and `OPLIM`.
`exp_5_6_1_3.code`	Tests a memory protection error with `breakpoint1` and `mem_address_not_aligned`.
`exp_5_6_1_4.code`	Tests a memory protection error with `breakpoint1` and `data_access_mem`.
`exp_5_6_1_5.code`	Tests a memory protection error with `breakpoint1` and `data_access_io`.
`exp_5_6_1_6.code`	Tests a memory protection error with `OPLIM` and `mem_address_not_aligned`.
`exp_5_6_1_7.code`	Tests a memory protection error with `OPLIM` and `data_access_mem`.
`exp_5_6_1_8.code`	Tests a memory protection error with `OPLIM` and `data_access_io`.
`exp_5_6_1_9.code`	Tests a memory protection error with `mem_address_not_aligned` and `data_access_mem`.
`exp_5_6_1_10.code`	Tests a memory protection error with `mem_address_not_aligned` and `data_access_io`.

TABLE 12-29 Tests for `breakpoint1` with Other Exceptions

Test Case	Description
`exp_5_6_2_1.code`	Tests `breakpoint1` with `OPLIM` and `mem_address_not_aligned`.
`exp_5_6_2_2.code`	Tests `breakpoint1` with `OPLIM` and `data_access_mem`.
`exp_5_6_2_3.code`	Tests `breakpoint1` with `OPLIM` and `data_access_io`.

TABLE 12-30 Tests for `OPLIM` with Other Exceptions

Test Case	Description
`exp_5_6_3_1.code`	Tests with `OPLIM`, `mem_address_not_aligned`, and `data_access_mem`.
`exp_5_6_3_2.code`	Tests with `OPLIM`, `mem_address_not_aligned`, and `data_access_io`.

12.3 Interrupts

We test nonmaskable interrupts (`NMI`) and maskable interrupts (`IRL`) at random.

12.4 Corner Cases for Trapping Instructions

TABLE 12-31 through TABLE 12-44 describe the corner cases for trapping instructions.

TABLE 12-31 `Getstatic` and `Putstatic`

Test Case	Description
`trap_6_1_1.java` (int) `trap_6_1_2.java` (float) `trap_6_1_3.java` (double) `trap_6_1_4.java` (long)	Test the access of inherited static variables.
`trap_6_1_5.java` (int) `trap_6_1_6.java` (float) `trap_6_1_7.java` (double) `trap_6_1_8.java` (long)	Test the access and shadowing of static variables.
`trap_6_1_5.java` (int) `trap_6_1_6.java` (float) `trap_6_1_7.java` (double) `trap_6_1_8.java` (long)	Test the access of shadowed static variables (`super.aStaticVar`).
`trap_6_1_3.java` (double) `trap_6_1_4.java` (long)	Access long and double.

TABLE 12-32 `Getfield` and `Putfield`

Test Case	Description
`trap_6_2_1.java` (int) `trap_6_2_2.java` (float) `trap_6_2_3.java` (double) `trap_6_2_4.java` (long)	Test the access of inherited instance variables.
`trap_6_2_5.java` (int) `trap_6_2_6.java` (float) `trap_6_2_7.java` (double) `trap_6_2_8.java` (long)	Test the access and shadowing of instance variables.
`trap_6_2_5.java` (int and static)) `trap_6_2_6.java` (float and static) `trap_6_2_7.java` (double and static) `trap_6_2_8.java` (long and static) `trap_6_2_13.java` (int and double, nonstatic)	Test the access of shadowed static variables (`super.anInstanceVar`).
`trap_6_2_7.java` (double) `trap_6_2_8.java` (long)	Test the access of long and double.
`trap_6_2_9.java` (int) `trap_6_2_10.java` (float) `trap_6_2_11.java` (double) `trap_6_2_12.java` (long)	Test the access of `var` instances with index values that exceed 255 (`getfield_quick_w` and `putfield_quick_w`).
`trap_6_2_9.java` (int) `trap_6_2_10.java` (float) `trap_6_2_11.java` (double) `trap_6_2_12.java` (long)	Test the access of instance variables whose keys are the same.

TABLE 12-33 `Invokestatic`

Test Case	Description

`trap_6_3_1.java`
`trap_6_3_2.java`
`trap_6_3_3.java`
`trap_6_3_4.java`
`trap_6_3_5.java`
	Test the access of inherited static methods.

`trap_6_3_6.java`	Tests the access of inherited static methods with the same key.
`trap_6_3_7.java` (int) `trap_6_3_8.java` (double)	Test the access of the overriding static method.
`trap_6_3_9.java` (int) `trap_6_3_10.java` (double)	Test the access of the overridden static method.

TABLE 12-33 `Invokestatic` *(Continued)*

Test Case	Description
`trap_6_3_1.java`	Tests the access of the overloaded static method with the same method name, but a different number of arguments.
`trap_6_3_2.java`	Tests the access of the overloadedstatic method with the same method name, but different types of arguments (int and double).
`trap_6_3_3.java`	Tests the access of the overloaded static method with the same method name, but different types of arguments (int and float).
`trap_6_3_4.java`	Tests the access of the overloaded static method with the same method name, but different types of arguments (long and double).
`trap_6_3_5.java`	Tests the access of the overloaded static method with the same method name, but different types of arguments (int and double).
`trap_6_3_11.java`	Tests the access of the overloaded static method with the same method name, but a different order of arguments (int and double).
`trap_6_3_12.java`	Tests the access of superclass methods.

TABLE 12-34 `Invokevirtual`

Test Case	Description
`trap_6_4_1.java`	Tests the access of inherited instance methods with the same method name, but a different number of arguments.
`trap_6_4_2.java`	Tests the access of inherited instance methods with the same method name, but different types of arguments (int and double).
`trap_6_4_3.java`	Tests the access of inherited instance methods with the same method name, but different types of arguments (int and float).
`trap_6_4_4.java`	Tests the access of inherited instance methods with the same method name, but different types of arguments (long and double).
`trap_6_4_5.java`	Tests the access of inherited instance methods with the same method name, but different types of arguments (int and float).

TABLE 12-34 `Invokevirtual` *(Continued)*

Test Case	Description
`trap_6_4_6.java`	Tests the access of inherited instance methods with the same method name, but different types of returns (int and double).
`trap_6_4_7.java` (int) `trap_6_4_8.java` (double) `trap_6_4_13.java` (casts a subclass to its superclass and calls the overriding method)	Test the access of the overriding instance method.
`trap_6_3_9.java` (int) `trap_6_3_10.java` (double)	Test the access of the overridden instance method.
`trap_6_4_1.java`	Tests the access of the overloaded instance method with the same method name, but a different number of arguments.
`trap_6_4_2.java`	Tests the access of the overloaded instance method with the same method name, but different types of arguments (int and double).
`trap_6_4_3.java`	Tests the access of the overloaded instance method with the same method name, but different types of arguments (int and float).
`trap_6_4_4.java`	Tests the access of the overloaded instance method with the same method name, but different types of arguments (long and double).
`trap_6_4_5.java`	Tests the access of the overloaded instance method with the same method name, but different types of arguments (int and double).
`trap_6_4_6.java`	Tests the access of the overloaded instance method with the same method name, but different types of returns (int and double).
`trap_6_4_11.java`	Tests the resolution of the dynamic method.
`trap_6_4_12.java`	Tests the access of an instance method with an index that exceeds 255.
`trap_6_4_14.java`	Tests the access of superclass methods.

TABLE 12-35 `ldc2_w`

Test Case	Description

`trap_6_1_3.java`
`trap_6_1_7.java`
`trap_6_2_11.java`
`trap_6_2_3.java`
`trap_6_2_7.java`
`trap_6_3_10.java`
`trap_6_3_2.java`
`trap_6_3_5.java`
`trap_6_3_8.java`
`trap_6_4_10.java`
`trap_6_4_2.java`
`trap_6_4_4.java`
`trap_6_4_6.java`
`trap_6_4_8.java`
	Test with double.

`trap_6_1_4.java`
`trap_6_1_9.java`
`trap_6_2_12.java`
`trap_6_2_8.java`
`trap_6_4_5.java`
	Test with long.

TABLE 12-36 `ldc`

Test Case	Description
`trap_6_2_4.java`	Tests with int.

`trap_6_1_2.java`
`trap_6_1_6.java`
`trap_6_2_10.java`
`trap_6_2_2.java`
`trap_6_2_6.java`
`trap_6_3_3.java`
`trap_6_3_4.java`
`trap_6_4_3.java`
`trap_6_4_4.java`
	Test with float.

`trap_6_6_1.java`	Tests the string.

TABLE 12-37 `ldc_w`

Test Case	Description
`trap_6_7_1.java` `trap_6_7_2.java`	Test with int.
`trap_6_7_3.java`	Tests with float.

TABLE 12-38 `Lookupswitch`

Test Case	Description
`trap_6_8_1.java`	Tests with byte.
`trap_6_8_2.java`	Tests with char.
`trap_6_8_3.java`	Tests with short.
`trap_6_8_4.java`	Tests with int.

TABLE 12-39 `Newarray`

Test Case	Description
`trap_6_9_1.java`	Tests with byte.
`trap_6_9_2.java`	Tests with int.
`trap_6_9_3.java`	Tests with float.
`trap_6_9_4.java`	Tests with double.
`trap_6_9_5.java`	Tests with long.
`trap_6_9_6.java`	Tests with boolean.
`trap_6_9_7.java`	Tests with char.
`trap_6_9_8.java`	Tests with short.

TABLE 12-40 `Anewarray`

Test Case	Description
`trap_6_10_1.java`	Tests with an object.

TABLE 12-41 `Multianewarray`

Test Case	Description
`trap_6_11_1.java`	Tests with byte.
`trap_6_11_2.java`	Tests with int.
`trap_6_11_3.java`	Tests with float.
`trap_6_11_4.java`	Tests with double.
`trap_6_11_5.java`	Tests with long.
`trap_6_11_6.java`	Tests with boolean.
`trap_6_11_7.java`	Tests with char.

TABLE 12-41 Multianewarray

Test Case	Description
trap_6_11_8.java	Tests with short.
trap_6_11_9.java	Tests with object.
trap_6_11_10.java	Tests with a multianewarray call with less dimension than the declared type.

TABLE 12-42 Invokespecial

Test Case	Description
trap_6_12_1.java	Tests with invokenonvirtual_quick (init method).
trap_6_12_2.java trap_6_12_3.java	Test with invokesuper_quick (other method than init).

TABLE 12-43 Invokeinterface

Test Case	Description
trap_6_13_1.java	Tests the access of the method for the superclass with invokeinterface.
trap_6_13_2.java	Tests with a class that implements multiple interfaces.
trap_6_13_3.java	Tests with an interface that extends another interface.
trap_6_13_4.java	Tests with an interface that extends multiple interfaces at a time.

TABLE 12-44 athrow

Test Case	Description
trap_6_14_1.java	Tests basic athrow with try and catch statements.
trap_6_14_2.java	Tests the throw runtime class.
trap_6_14_3.java	Tests with throw multiple exception classes.

Index