microSPARC-IIep™ Validation Catalog

THE NETWORK IS THE COMPUTER™

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

Part No.: 851-2430
Revision: 01
July 1999

Contents

Preface

The book, *microSPARC-IIep Validation Catalog*, is one of a four-manual documentation set for the microSPARC-II technology. The other three manuals are:

- *Multiprocessor SPARC Architecture Simulator (MPSAS) User's Guide*, describes how to use the MPSAS and its associated programs.
- *Multiprocessor SPARC Architecture Simulator (MPSAS) Programmer's Guide* explains the facilities for creating or modifying MPSAS modules or otherwise extending MPSAS.
- *microSPARC-IIep Megacell Reference* explains how to build microSPARC-IIep megacells.

Organization of This Book

This book is intended for programmers who develop and debug programs to be run the microSPARC-IIep. It is arranged as follows:

Chapter 1, "Introduction," introduces the microSPARC-IIep validation catalog.

Chapter 2, "Diagnostic Environment," describes the diagnostic environment used for microSPARC-IIep verification.

Chapter 3, "Integer Unit Test," identifies the integer unit validation test suites.

Chapter 4, "Floating-point Unit Tests," covers the microSPARC-IIEp floating-point unit diagnostic tests.

Chapter 5, "Memory Management Unit Tests," describes the diagnostic tests for the memory management unit.

Chapter 6, "Instruction and Data Cache Tests," describes the instruction and data cache validation test suites.

Chapter 7, "Memory Interface Unit Tests," covers the memory interface unit diagnoztics.

Chapter 8, "PCI Controller Unit Tests," covers the PCI controller unit tests.

Chapter 9, "Random Diagnostic Tests," describes the diagnostic test randomly generated using the thrash process.

Prerequisite Knowledge

It is assumed that you are familiar with programming in the C language in the UNIX® environment and that you have a basic familiarity with computer architecture, in particular the SPARC architecture. For further information, see the list of documents in the following section.

Related Books and References

The following documents contain material that further explains or clarifies information presented in this guide.

The SPARC V8 Architecture Manual

SunOS Reference Manual: Section 1, User Commands

The C Programming Language by Brian W. Kernighan and Dennis M. Ritchie, Prentice Hall; ISBN: 0131103628

Typographic Conventions

The following table 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℠ 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 by 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.

CHAPTER 1

Introduction

The microSPARC-IIep is a highly integrated, low-cost, low-power implementation of the SPARC V8 RISC architecture. High performance is achieved by the high level of integration including on-chip instruction and data caches, and close coupling of the CPU with main memory. A full custom implementation allows for a target frequency of 70 MHz and 150 MHz, providing sustained performance of 42 and 84 Specmarks, respectively. The design is highly testable with the use of the full JTAG scan support. The microSPARC-IIep supports up to 256 Mbytes of DRAM and four 32-bit 33 MHz PCI slots.

1.1 microSPARC-IIep Blocks

This catalog contains descriptions of the diagnostics used in the architectural verification of microSPARC-IIep. Explanations are provided for the individual test cases.The diagnostics are divided into the following sections:

- Integer Unit
- Floating-Point Unit
- Memory Management Unit
- Instruction Cache and Data Cache
- Memory Interface Unit
- PCI Controller
- Random Tests

1.2 microSPARC-IIep Diagnostic Style

The microSPARC-IIep diagnostics can be classified into two categories:

- Non-asm2ver style: contains no notion of Reference MMU and thereby always runs with the MMU disabled.
- asm2ver style: contains a notion of Reference MMU. The MMU is enabled and the process allows for mapping of virtual pages.

Many of the diagnostics are self-checking and do not require Microprocessor SPARC Architecture Simulator (MPSAS) support (option `-n sas` on run script). MPSAS performs rudimentary modeling of the PCI (just a memory), so most tests of the PCI are self-checking for that reason.

1.3 microSPARC-IIep Diagnostic Modes

The microSPARC-IIep diagnostics make use of a concept called “mode.” A mode is a way of enabling or disabling some of the major functional blocks of microSPARC-IIep. The following modes are used in the microSPARC-IIep:

- `c_pa_reset`: cacheable physical address (`cpa`) prevents virtual to physical address translation.
- `tlb_reset`: translation look-aside buffer (`tlb`) register mode with one TLB entry. Causes thrashing of the TLB for extra traffic on the internal memory bus.
- `tlb32_reset`: translation look-aside buffer (`tlb`) register mode configured to operate with all 32 entries. The TLB replacement counter is enabled and the entries are replaced using a pseudorandom algorithm.
- `nc`: no cache (nc) mode with reset environment. In this mode, the microSPARC-IIep is configured so that the Instruction Cache and Data Cache are both disabled. All memory references miss the cache. This mode resets the Instruction Cache Enable and Data Cache Enable bits in the Processor Control Register.
- `nic`: no instruction cache (nic) mode with reset environment. In this mode, the microSPARC-IIep is configured so that the Instruction Cache is disabled. All instruction accesses to the cache cause a miss. This mode resets the Instruction Cache Enable bit in the Processor Control Register.
- `ndc`: no data cache (ndc) mode with reset environment. In this mode, the microSPARC-IIep is configured so that the Data Cache is disabled. All data accesses to the cache cause a miss. This mode resets the Data Cache Enable bit in the Processor Control Register.
- `fastrfr`: refresh (rfr) mode with reset environment. In this mode, the microSPARC-IIep is configured so that the external DRAM refresh rate is the fastest. The microSPARC-IIep generates a refresh cycle every 128 clocks. This mode resets the Refresh Control bits [1:0] in the Processor Control Register.

- `s`: standby (s) assert/deassert randomly the power-down mode pin. A Verilog task asserts the power-down mode pin. The deassert of the mode pin can happen by deasserting the pin itself, or by asserting a fixed interrupt (based on the power management specification). Memory protocol monitor and PCI bus protocol monitor are used to verify the quiescent state.
- `nbf`: no branch folding (nbf) causes the microSPARC-IIep not to fold branches.
- `pmc0`: page 0 mode control.
- `pmc1`: page 1 mode control.
- `pmc2`: page 2 mode control.
- `ptp2`: creates virtually tagged level-2 page table pointers.

CHAPTER 2

Diagnostic Environment

This chapter describes the diagnostic environment used for microSPARC-IIep verification.

2.1 Test Environment

The microSPARC-IIep test environment consists of a two models:

- Multiprocessor SPARC Architecture Simulator (MPSAS) model
- Verilog RTL (or gate-level) model as simulated by VCS

There are two ways of running MPSAS:

- I-I: instruction-to-instruction comparison, that is, MPSAS run with Verilog VCS. Comparisons are done after every instruction.
- Standalone: MPSAS runs through diagnostic to the end, before Verilog is run, and final state dumps are generated. Verilog VCS simulation then starts, and upon completion, generates final state dumps. The two sets of dumps are then compared. MPSAS can also be run standalone without running VCS.

Three options are available:

- `run -n sas`: run with neither I-I nor standalone (for self-checking diagnostics)
- `run -S`: run both I-I and standalone
- `run`: (unspecified option) run only standalone

2.2 Memory Environment

There are two perspectives to consider for the microSPARC-IIep address map:

- Hard-wired physical address spaces
- Virtual spaces setup by the asm2ver environment

The physical address spaces defined for microSPARC-IIep address bits 30:28 are listed in the *microSPARC-IIep User's Manual*. It defines eight possible address spaces, four of which are currently used:

PA[30:28]=000	DRAM Memory (256M)
PA[30:28]=001	Control Space (256M)
PA[30:28]=010	FlashPROM Space (256M)
PA[30:28]=011	PCI Space (256M)

2.2.1 Tests Using the Assembly to Verilog Tool

Tests that use the Assembly to Verilog tool (`asm2ver`) are prepended with `reset.s` and `traps.s` from `asm2ver` tests `~diag/include/asm2ver`. These code segments use `#define` statements to configure different modes, such as supervisor/user, alignment trap enable, etc. The `#define` definitions can be put at the beginning of the test as follows:

```
#define INIT_PSR SUPER
```

The preceding example instructs the `reset.s` code to set supervisor mode in the PSR without having to know the specific register or bit field. The `reset.s` and `traps.s` also build a default set of trap handlers and page tables. The code builds page tables for the code between the label `user_text_start` and `user_text_end`. A similar set of tables is built for the data segment between labels `user_data_start` and `user_data_end`.

```
#define IOTLB_ENABLE
```

In the preceding example, the `asm2ver` test initializes and enables the iotlb. This causes translations for PCI DMA to and from main memory. This is normally used as follows:

```
#define ENABLE_PCI
```

The preceding example sets up and enables the control registers for master and slave PCI traffic.

The IOTLB_ENABLE switch sets all 16 entries as follows:

```
.word    0x080010f4,    0x00000001    ! 0xf4100008,    0x01000000
.word    0x081010f4,    0x00800001    ! 0xf4101008,    0x01008000
.word    0x080020f4,    0x00000002    ! 0xf4200008,    0x02000000
.word    0x081020f4,    0x00800002    ! 0xf4201008,    0x02008000
.word    0x080030f4,    0x00000003    ! 0xf4300008,    0x03000000
.word    0x081030f4,    0x00800003    ! 0xf4301008,    0x03008000
.word    0x080040f4,    0x00000004    ! 0xf4400008,    0x04000000
.word    0x081040f4,    0x00800004    ! 0xf4401008,    0x04008000
.word    0x080050f4,    0x00000005    ! 0xf4500008,    0x05000000
.word    0x081050f4,    0x00800005    ! 0xf4501008,    0x05008000
.word    0x080060f4,    0x00000006    ! 0xf4600008,    0x06000000
.word    0x081060f4,    0x00800006    ! 0xf4601008,    0x06008000
.word    0x080070f4,    0x00000007    ! 0xf4700008,    0x07000000
.word    0x081070f4,    0x00800007    ! 0xf4701008,    0x07008000
.word    0x080080f4,    0x00000008    ! 0xf4800008,    0x08000000
.word    0x081080f4,    0x00800008    ! 0xf4801008,    0x08008000
```

The first word is the CAM position, second is the RAM portion. The words are byte-twisted to account for the translation that takes place going through PCIC in the default mode. The comments show the little endian equivalents.

The default memory maps for the `asm2ver` tests are listed in TABLE 2-1 and TABLE 2-2.

TABLE 2-1 Virtual Memory Map

0-11FF	traps
1590	xxxxxxxx
2000	user text (code)
4000	results area I
C000	results area II
14000	trap handlers (reset handler, trap data table, stack space)
15000	user data

TABLE 2-2 Physical Memory Map

0000	trap vector table
5000	trap handler data
100000	user text area
200000	user data area
300000	results area

To create a custom map, the function ASSIGN_MMU_PAGE is provided as follows:

```
ASSIGN_MMU_MAP (
PAGE_START_LABEL = user_text_start,
PAGE_END_LABEL = user_text_end.
PHYSICAL_PAGE_NUMBER = 0x30005,
PAGE_ACCESS_RIGHTS = ACCESS_RW,
PAGE_CUNTEXT_NUMBER = 55,
PAGE_SIZE = PAGE_4K,
PAGE_CACHEABLE,
PAGE_ENTRY_RESERVED = 0.
PAGE_ENTRY_PTP = 0,
PAGE_DATA_EXCLUDE
```

FIGURE 2-1 Custom Mapping of the ASSIGN_MMU_PAGE Function

TABLE 2-3 identifies the definitions of each field of the ASSIGN_MMU_PAGE function.

TABLE 2-3 ASSIGN_MMU_PAGE Function Field Definitions

Field	Description
PAGE_START_LABEL and PAGE_END_LABEL	Designates the beginning and end virtual address of a portion of code/data that is to be located somewhere in physical memory. It could point to mnemonic labels within the test program or refer to absolute addresses.
PAGE_ACCESS_RIGHTS	Assigns one of eight possible access rights to a set of pages between PAGE_START_LABEL and PAGE_END_LABEL. These access rights are described in the MMU specification. The mnemonics for these rights are: ACCESS_R, _RW, _RX, _RWX, _X, _S_RW, _S_RX, and _S_RWX for MMU codes 0-7, respectively.
PAGE_CONTEXT_NUMBER	Denotes the page context number the set of pages belongs to. The maximum context number allowed is 256.
PAGE_SIZE	Defines the size of the page the portion of code/data is assigned to. Four different page sizes are allowed and denoted by PAGE_4K, PAGE_256K, PAGE_16M, and PAGE_ROOT (4G). Multiple pages, of the required size, may be allotted automatically depending on the page size and the amount of information to be loaded into the pages. If the difference between the user_text_start and user_text_end labels in FIGURE 2-1 is 12 Kbytes and the PAGE_SIZE is defined to be PAGE_4K, three such pages will be used.
PAGE_CACHEABLE	When present, denotes whether the page is considered cacheable or not. If it is not included, noncacheable is assumed.
PHYSICAL_PAGE_NUMBER	Encodes the starting physical address for the pages being defined. This number is appended with three hexadecimal 0's to form the physical address.

TABLE 2-3 ASSIGN_MMU_PAGE Function Field Definitions *(Continued)*

Field	Description	
PAGE_ENTRY_VALID	Allows some control over the valid field as it appears in the PTEs and PTPs of the MMU. The following macros are available:	
	LEVEL_ALLVALID	Sets all entries valid
	LEVEL3_INVALID	Causes the third-level entry to be marked invalid
	LEVEL2_INVALID	Causes the second-level entry in the page table to be marked invalid
	LEVEL1_INVALID	Causes the first-level entry in the page table to be marked invalid
	LEVEL0_INVALID	Causes the zeroth-level entry in the page table to be marked invalid

2.3 PCI Environment

The verification setup for microSPARC-IIep consists of four RaviCad PCI masters, one RaviCad slave, and one LMC slave attached to the microSPARC-IIep PCI bus. PCI PIO traffic is performed by writing to the space three physical address range. This space is split into a hard-wired space and spaces that are potentially translated before they are sent to the PCI bus, as shown in TABLE 2-4.

TABLE 2-4 PCI Physical Address Space for PIO

Physical Address	Mapping
0x30000000 - 0x3000FFFF	Hard-wired IO space to PCI bus.
0x30001000 - 0x31000000	Contains configuration registers and untranslated memory.

Any space three addresses above this last range are subject to translation by the `smbar0`, `smbar1`, and `pibar` registers. Refer to *microSPARC-IIep User's Manual* for details. These registers allow for two different memory spaces and one IO space to be translated before going to the PCI bus. Any address in this range, not caught by the range of one of these registers, goes through untranslated as a memory cycle.

For all space three cases, the uppermost nibble, bits 31:28, do not get passed on to the PCI bus. For the fixed map cases, the nibble is stripped. For the translated cases, the upper bits presented on the PCI bus are determined by the register mappings.

Therefore, there is no simple mapping of the PIO address space above 0x30FFFFFF. It is set by the translation registers. The `crueltt_falc.s` and `traps.s` initialization code provide for default programming of these registers, as well as the DMA translation. To provide a consistent default test environment, the PCI slave models have been set to respond to the following PCI physical address ranges. FIGURE 2-2 illustrates the ranges.

```
lmc mem range 0     00100000-001FFFFF//LMC mem pages accessed by fixed map
lmc mem range 1     21000000-21002000//2 LMC mem pages accessed by (pmbar0=2) translation
lmc mem range 2     (still available)
lmc io range 0      00000000-00002000//2 LMC IO pages accesses by fixed map
lmc io range 1      13004000-13008000//4 LMC IO pages accessed by (pibar=1)
lmc io range 2      (still available)
ravi mem range      00800000-12004000 *//Large range in fixed mem+pages translated by (pmbar1=1)
ravi io range       00004000-13002000//Pages in fixed IO+2 pages contiguous with LMC IO (pibar=1)
```

FIGURE 2-2 PCI Physical Address Ranges

These ranges provide target addresses that can be used with both the fixed map, requiring no register setup, and the default settings of the translation registers.

Files `pcic.h` and `pcic.s` are in the PCI/include area. The `pcic.h` file has some `#defines` for PCIC registers. If you use `traps.s` or `crueltt_falc.s` in your test, you can use these instead of hardwired values.

TABLE 2-5 shows the six PCI base registers and six PCI size registers the `pcic.s` file initializes.

TABLE 2-5 PCI Base and PCI Size Registers

PCI Base	PCI Size Registers
PCIBASE0	Set to map PCI DMA starting at 0xF0000000
PCISIZE0	The space above covers a 256-Mbyte range
PCIBASE1	Set for 0xE0000000
PCISIZE1	Set for 256 bytes
PCIBASE2	Set for 0xE0000100
PCISIZE2	Set for 256 bytes
PCIBASE3	Set for 0xE0000200
PCISIZE3	Set for 256 bytes
PCIBASE4	Set for 0xE0000300
PCISIZE4	Set for 256 bytes

TABLE 2-5 PCI Base and PCI Size Registers *(Continued)*

PCI Base	PCI Size Registers
PCIBASE5	Set for 0xE0000400
PCISIZE5	Set for 256 bytes

One of the register sets maps master DMA space similarly to the way it was set up for the PCI DMA master tests, such as `burst_wr`. The remaining registers are parked into unique 256-byte spaces starting at 0xE0000000.

The `SMBAR` registers for afx to PCI space have new addresses. They have `#defines` in `pcic.h` as well. They default to the values listed in TABLE 2-6.

TABLE 2-6 `SMBAR` Registers Default Values

Register	Default Value
SMBAR0	0x01 … Bits 27:23 of virtual address
MSIZE0	0x0F … Compare all four bits
PMBAR0	0x20 Bits 28:23 on PCI bus
SMBAR1	0x02
MSIZE1	0x0F
PMBAR1	0x10

CHAPTER 3

Integer Unit Tests

This chapter identifies the integer unit validation test suites.

addrbyp

Data bypass is required when the instruction immediately following a load instruction (`LD`, `LDUB`, `LDSB`, `LDUH`, `LDSH`, and the alternate space variants) uses as an operand the destination register of the `load.Bypass` path 3 is used to pass a result from an R stage to the D stage in the pipeline. This diagnostic checks that the source 1 and source 2 inputs of the address `adder` are fed properly by the bypass paths from LD and LDD type instructions.

allInsts

A basic test to verify execution of each instruction. The program contains a number of nops to test instructions without any dependencies and interlocks. This test verifies illegal opcodes. This test, along with `FallInst_v.s` diagnostic, covers all the instructions. This is a self-checking diagnostic. The following instructions are verified:

`add`, `addcc`, `addx`, `addxcc`, `and`, `andcc`, `andn`, `andncc`, `annul`, `ba`, `bicc not taken`, `bicc taken`, `call`, `iflush`, `jmpl`, `ld`, `ldd`, `lddf`, `ldf`, `ldfsr`, `ldsb`, `ldsh`, `ldst`, `ldub`, `lduh`, `mulscc`, `or`, `orcc`, `orn`, `orncc`, `rdpsr`, `rdwim`, `rdy`, `restore`, `save`, `sra`, `std`, `swap`, `sethi`, `sll`, `srl`, `st`, `stb`, `stdfq`, `stf`, `stfsr`, `sth`, `sub`, `subcc`, `subx`, `subxcc`, `taddcc`, `taddcctv`, `ticc`, `tsubcc`, `tsubcctv`, `xnor`, `xor`, `xorcc`, `wrpsr`, `wrwim`, `wry`.

This test uses its own trap handler.

alu_tvs

This diagnostic puts predetermined sequences of patterns through the ALU. The tests consist of blocks of ADDcc, ADDXcc, SUBcc, ORNcc, ORcc, ANDNcc, ANDcc, XNORcc, and XORcc instructions. Each instruction takes as operands two words read from the data segment to produce a result and setting of the icc. If the flag CHECK_RES is set at compilation time, the result and correct icc image are also read and checked against the results. The result of the operation is then stored, to make it visible, on the output pins of the IU. The original test patterns as received from BIT had the various instruction patterns interspersed. For this diagnostic, each instruction type (for example, all ANDcc) has been grouped to run together in a loop. To guarantee the correct condition codes are used for each subblock (for example, a group of sequential ADDXccs before they were merged), additional instructions are added to correctly set the ccs.

anotherRett

A basic test to verify various rett conditions. The following cases are covered:

- ET = 1, S = 0, misaligned address, window underflow should get illegal instruction trap because ET = 1
- ET = 0, S = 0, misaligned address, window underflow should get privilege instruction trap because S = 0
- ET = 0, S = 1, misaligned address, window underflow should get window underflow trap
- ET = 0, S = 1, misaligned address should get misaligned address trap
- ET = 1, ticc 26 trap handler does ticc 27 to make sure TT does not change on second ticc.

This test causes a watch-dog reset and an endless loop in Verilog simulation.

atomic_h

A basic test to verify atomic operation under different conditions. The atomic instructions (swap and ldstub) will hit in the cache, miss in the cache, and a non-cached access is attempted. Six cases are covered.

bicccorner_h

A test to verify the corner conditions with bicc instructions. The test verifies the following cases:

The annulled instruction in the delay slot is udivcc. ALl 16 branch instructions are followed by a udivcc instruction. The udivcc instruction should be annulled in each case.

Causes an Instruction Cache miss for the delay slot instruction. The missing instruction is a `udivcc`.

Checks for `udivcc` for overflow condition.

br_displ.s

This diagnostic does plenty of negative displacements in `bicc`/call instructions. Displacements vary from 1 to 8.

brCombo

An exhaustive test to verify all the branch condition code combinations. The following tests are covered:

- Branch which `annul` single cycle `alu` instructions
- Branch which `annul` loads
- Branch which `annul` stores
- Branch which `annul` `ldstub` (atomic operation)
- Branch which `annul` floating-point loads
- Branch which `annul` branches
- Branch which `annul` `jmp`
- Branch which `annul` `save`
- Branch which `annul` `restore`
- Branch which `annul` `ticc`
- Branch which `annul` `rett`
- Taken branches with various instructions (`addcc`, `ld`, `jmp`, `ldd`, `st`, `std`, `rett`) in the delay slot and branch target is also a branch
- Branch followed by a branch combination. The target of first branch is also a branch. The first branch is taken.
- Not taken branch with various delay (`faddd`, `ld`, `ldd`, `st`, `std`, `ticc`, `taddcctv`) and target instructions.

branch_seq1

This diagnostic tests the branch sequence where a control transfer instruction (with its delay slot) has another control transfer instruction at or within seven instructions of the target. (The instructions before the second control transfer are nops in this diagnostic.)

The first control transfer is one of four possibilities: `call`, `jump link`, `branch` (`taken`, `no annul`), `branch` (`taken`, `annul`).

The second control transfer is one of six: call, jump link, branch (taken, no annul), branch (taken, annul), branch (not taken, no annul), branch (not taken, annul).

Also, the first and the second control transfer can be on even or odd addresses (independent of each other). Each set of combinations is further expanded to give the target transfer at one of the first eight locations at the target.

The body of tests is in branch_seq.tb and is generated from branch_seq.im by the script branch_seq.make. The total number of test sets is: (2 * 4) * (2 * 6) * (8) = 768 with the first odd/even second odd/even and eight target locations.

Due to the large size of this diagnostic, it has been broken into three parts that run independently. The other two diagnostics are called branch_seq2 and branch_seq3.

branch_seq2

This diagnostic tests the branch sequence where a control transfer instruction (with its delay slot) has another control transfer instruction at or within seven instructions of the target. (The instructions before the second control transfer are nops in this diagnostic.)

The first control transfer is one of four possibilities: call, jump link, branch (taken, no annul), branch (taken, annul).

The second control transfer is one of six: call, jump link, branch (taken, no annul), branch (taken, annul), branch (not taken, no annul), branch (not taken, annul).

Also, the first and the second control transfer can be on even or odd addresses (independent of each other). Each set of combinations is further expanded to give the target transfer at one of the first eight locations at the target.

The body of tests is in branch_seq.tb and is generated from branch_seq.im using the script branch_seq.make. The total number of test sets is: (2 * 4) * (2 * 6) * (8) = 768. First odd/even second odd/even eight target locations.

Because of the large size of this diagnostic, it has been broken into three parts that run independently. The other two diagnostics are called branch_seq1 and branch_seq3.

branch_seq3

This diagnostic tests the branch sequence where a control transfer instruction (with its delay slot) has another control transfer instruction at or within seven instructions of the target. (The instructions before the second control transfer are nops in this diagnostic.)

The first control transfer is one of four possibilities: `call`, `jump link`, `branch` (`taken`, `no annul`), `branch` (`taken`, `annul`).

The second control transfer is one of six: `call`, `jump link`, `branch` (`taken`, `no annul`), `branch` (`taken`, `annul`), `branch` (`not taken`, `no annul`), `branch` (`not taken`, `annul`).

Also, the first and the second control transfer can be on even or odd addresses (independent of each other). Each set of combinations are further expanded to give the target transfer at one of the first eight locations at the target.

The body of tests is in `branch_seq.tb` and is generated from `branch_seq.im` using the script `branch_seq.make`. The total number of test sets is:
(2 * 4)   *   (2 * 6)   *   (8)   = 768. First odd/even second odd/even eight target locations.

Because of the large size of this diagnostic, it has been broken into three parts that run independently. The other two diagnostics are called `branch_seq1` and `branch_seq2`.

branche

A basic test to verify the 16 combinations of the `taken` and `not taken` branches.

bypass3

Data bypass is required when the instruction immediately following a load instruction (`LD`, `LDUB`, `LDSB`, `LDUH`, `LDSH`, and the alternate space variants) uses as an operand the destination register of the `load.Bypass` path 3 is used to pass a result from an R stage to the D stage in the pipeline. This diagnostic checks that the source 1 and source 2 inputs of the `adder` are fed properly by the bypass paths from `alu` type instructions.

This test is similar to the `mem_memCombo` test.

bypass3_win

Data bypass is required when the instruction immediately following a load instruction (LD, LDUB, LDSB, LDUH, LDSH, and the alternate space variants) uses, as an operand, the destination register of the load.Bypass path 3 to pass a result from the R stage to the D stage in the pipeline. This diagnostic checks that the source 1 and source 2 inputs of the adder are fed properly by the bypass paths from alu type instructions.

This test is similar to the bypass3 test. This test does not have the percent go no bypass test case.

callret

This is a short test to verify the call and return operations.

callrit

This is a basic test to verify register interlock after a call instruction. For example, if the instruction in the delay slot of a call instruction uses register %o7, there is an interlock so that the new value (written by the call instruction) is used. The instructions used in the delay slot are jmp, ld, ldd, ldstub, multiple call, restore, save, st, std, taddcctv, and ticc.

cti_couplest1

This program tests all the CTI combinations.

- Register r10 (window = 2) is used to log errors.
- Register r11 (window = 2) is used to log the various cases covered.
- A non-zero bit in r10 indicates an error.
- A non-zero in r11 indicate the cases covered by the test.

TABLE 3-1 identifies the bit assignment for r10/r11.

TABLE 3-1 Error Bit Assignment for CTI Combinations

Register Bits	Case Covered
01:00	case 1a
03:02	case 1b
05:04	case 1c
07:06	case 1d
09:08	case 2a
11:10	case 2b
13:12	case 3a
15:14	case 3b
17:16	case 4a
19:18	case 4b
21:20	case 7a
23:22	case 7b
25:24	case 8a
27:26	case 8b

Upon successful completion of the test, r10 (out2, location 42 dec. in RF) will have all 0’s, and r11 (out3, location 43 dec. in RF) will have all 1’s.

TABLE 3-2 CTI Combination Test Cases

Case No.	Cti # 1	Cti # 2
1a	cti taken	cti taken
1b	cti (a) taken	cti taken
1c	cti taken	cti (a) taken
1d	cti (a) taken	cti (a) taken
2a	cti taken	BA (a)
2b	cti (a) taken	BA (a)
3a	cti taken	cti untaken

TABLE 3-2 CTI Combination Test Cases *(Continued)*

Case No.	Cti # 1	Cti # 2
3b	cti (a) taken	cti untaken
4a	cti taken	cti (a) untaken
4b	cti (a) taken	cti (a) untaken
7a	BA (a)	cti taken
7b	BA (a)	cti (a) taken
8a	BA (a)	cti untaken
8b	BA (a)	cti (a) untaken

cwp

This diagnostic tests the various instructions and sequences that affect and depend upon the CWP. These include the save and restore instructions and the subroutine linkage sequence.

divTest

A short test to check `sdivcc` boundary case. It performs a number of divisions and checks the results against a precalculated result. It also tests the conditions code bits; it also checks for overflow conditions. The test branches to bad trap location in case of failure.

divTest1

A short test to check `sdivcc` and `udivcc` operations. It performs a number of divisions and checks the results against a precalculated result. It also tests the conditions code bits; it also checks for overflow conditions. The test branches to bad trap location in case of failure.

divTest2

A short test to check `sdivcc` and `udivcc` boundary case. It performs a number of divisions and checks the results against a precalculated result. It also tests the conditions code bits. It checks for division by zero (trap) and result of zero conditions. The test branches to bad trap location in case of failure.

divTest3

A basic test to check `sdivcc` boundary case. It performs a number of divisions and checks the results against a precalculated result. It also tests the conditions code bits; it checks for overflow conditions. The test branches to bad trap location in case of failure.

divTest4

A short test to check `sdiv` operations. It performs a number of divisions and checks the results against a precalculated result. It also tests the conditions code bits. The test branches to bad trap location in case of failure.

div_test

An exhaustive test to verify `udiv`, `udivcc`, `sdiv`, and `sdivcc` operations. The test causes a divide by zero trap, overflow conditions, and performs various divides for randomly generated operands.

dmove

A short test to verify basic load/store operations with interlocks. The following cases are covered:

- Load followed by a dependent store
- Load double followed by a dependent store even register
- Load double followed by a dependent store odd register
- Load even register followed by a dependent store double
- Load odd register followed by a dependent store double
- Load double register followed by a dependent store double
- Generic store followed by store double

dslot_ilock2

This diagnostic tests the cases where an interlock occurs in the delay slot of a branch. The interlock is set up by a taken CTI. The FPC is notified of the new PC via the DOUT bus, but it must interlock because an FPU operation wants to be sent at the same time. The new PC is sent first, causing the IU to interlock. This diagnostic (`dslot_ilock2`) considers half the cases: The first CTI is a taken branch (annulled and not annulled cases), the second CTI is a `FBicc` (taken, untaken, and the annulled versions). See `dslot_ilock3` for the other cases.

dslot_ilock3

This diagnostic tests the cases where an interlock occurs in the delay slot of a branch. The interlock is set up by a taken CTI. The FPC is notified of the new PC via the DOUT bus, but it must interlock because an FPU operation wants to be sent at the same time. The new PC is sent first, causing the IU to interlock. This diagnostic (`dslot_ilock3`) considers half the cases: The first CTI is a taken branch (annulled and not annulled cases), the second CTI is an `FBicc` (taken, untaken, and the annulled versions). See `dslot_ilock2` for the other cases.

forwsethi

A single operand corner case to verify forwarding (bypassing) into `sethi`. It doesn't affect anything else.

ia_da_mmu_miss_h

A quick test to verify the interaction between Integer Unit and Memory Management Unit for instruction access error trap and data access error trap. This test also toggles the software table walk mode bit.

idiv_ovf

A quick test to verify the divide overflow cases. The test covers the `udivcc` and `sdivcc` instructions.

idiv_zero

A quick test to verify the divide by zero cases. The test covers the `sdiv`, `sdivcc`, `udiv`, and `udivcc` instructions.

ifetch.s

A shorter diagnostic to reproduce bugs found by Kaos. Bug related to `iflush` instruction followed by a folded branch caused the PC and instruction to miscompare.

iflush1.s

Tests `iflush` 5 instruction window with a full queue.

The algorithm: loads a line in the cache. This line contains an `iflush` instruction. Modify the sixth instruction after `iflush` to jump to the next test. Branch back to the line in cache. Execute `iflush instn`. The sixth instruction should be the modified instruction to jump to the next test. The following cases are covered:

- `iflush` at odd address
- `iflush` at even address
- `iflush` at even address with a preceding multicycle instruction
- `iflush` at odd address with a preceding multicycle instruction
- `iflush` at odd address, nop followed by a folded branch
- `iflush` at even address, nop followed by a folded branch
- `iflush` at even address, nop followed by a folded branch and `iflush` preceded by a multicycle instruction
- `iflush` at odd address, nop followed by a folded branch and `iflush` preceded by a multicycle instruction

iflush2.s

Tests `iflush` instruction window with an empty queue. The algorithm is the same as `iflush1.s`. `iflush` instruction is executed with an empty queue by having `iflush` the target of a branch. The following cases are covered:

- `iflush` at even address
- `iflush` at odd address
- `iflush` at even address with a preceding multicycle instruction
- `iflush` at odd address with a preceding multicycle instruction

iflush3.s

Tests `iflush` instruction window with `iflush` in the delay slot of a folded branch. The algorithm is same as `iflush1.s`. `iflush` instruction is executed in delay slot of folded branch. Eight possible branch cases are considered.

iflush4.s

Tests `iflush` 5 instruction window with `iflush` preceding a folded branch. The algorithm is same as iflush1.s. `iflush` instruction is executed preceding a folded branch. Eight possible branch cases are considered.

iflush5.s

Tests `iflush 5` instuction window with `iflush` being the target of a folded branch. The algorithm is same as `iflush1.s`. `iflush` instruction is executed as the target of a folded taken branch. Four possible branch cases are considered and `iflush` at even and odd address.

iflush6.s

Tests `iflush 5` instruction window with `iflush` being the target of an untaken folded branch. The algorithm is same as `iflush1.s`. `iflush` instruction is executed as the target of an untaken folded branch. Four possible branch cases are considered and `iflush` at even and odd address.

iflush7.s

Tests `iflush 5` instruction window with an empty queue, followed by a folded branch and `iflush` at even and odd address. The algorithm is same as `iflush1.s`. The following cases are covered:

- Empty queue -> `iflush` -> nop -> folded branch (`iflush` at even address)
- Empty queue -> `iflush` -> nop -> folded branch (`iflush` at odd address)

iflush8.s

Tests `iflush 5` instruction window with `iflush` and `jmp` instructions. The algorithm is same as `iflush1.s`. The following cases are covered:

- `iflush` before and after a `jmp`
- `iflush` -> nop -> `jmp`
- `iflush` target of a `jmp` (even and odd address)

ill_opcode

This diagnostic creates instruction words that contain illegal opcodes (OP3). Not all illegal opcodes are covered. See the macro file `ill_opc_macro.s` for documentation for a list of the codes that are covered. Each illegal instruction traps to the handler `illegal_opcode` and increment a counter (`illop_ct`). At the end of execution, this `illop_ct` contains the number of illegal instructions encountered (NUM_ERRS). This value is checked at the end. This diagnostic also tests the now illegal LDF2, STF3, LDC3, and STC2. It does not check LDF3, STF2, LDC2, or STC3.

illegal_branches

This diagnostic tests combinations involving illegal combinations of branches. CTI couples will fail if the first `bicc` is not taken. The following cases are covered:

- Annulled branch not taken
- Floating-point annulled branch not taken

This diagnostic fails if run with branch folding. It is run without branch folding in regression.

imul_742_810

A quick test to verify the multiply corner case bugs 742 and 810. The test verifies a `multiply` instruction followed by a `mulscc` instruction. The test also verifies the following bypass sequence:

```
load instruction -dest_reg1
```

- Any instruction
- Add or subtract instruction with `dest_reg1` as one of the sources
- `Multiply` instruction

imul_idiv

A short test to verify multiply and divide operations using register operands. The test performs `umulcc`, `smulcc`, `umul`, `sdivcc`, `udivcc`. Eight operations are performed.

imul_idiv_imm

A short test to verify multiply and divide operations using immediate operands. The test performs `umulcc`, `smulcc`, `umul`, `sdivcc`, `udivcc`. Six operations are performed.

imul_simple

A short test to verify multiply operation using register operands. The test does one `umulcc`, `smulcc`, and `umul` operation.

interlock

This diagnostic tests IU register interlocks that occur following certain instructions.

If the instruction in the delay slot of a call instruction uses register `%07`. There is an interlock so that the new value written by the call instruction is used.

Write and Read Y register with a three-instruction delay interlock.

Write and Read PSR register with a three-instruction delay interlock. The implementation and version number are masked off.

Write and Read WIM register with a three-instruction delay interlock.

A simple interlock for load single word from memory.

intlddstd

A simple test case to verify register interlocks during load and store double word operands. Interlock is between a couple of load instructions. It also checks for load followed by store operation.(Read followed by Write to memory).

intldst

A simple test case to verify Load Store Unsigned Byte operation. All the `ldstub` operations causes an data access exception. The destination register in the test should be clearly restored upon data access exception to work.

intload

A simple test case to verify various load instructions: the load word, load unsigned halfword, load signed halfword, load unsigned byte, load signed byte for all address alignments. The load writes into all the registers of one window. There are no stores to verify that the loads into the register file are done correctly. To improve observability, use `intload2`.

intload2

A simple test case to verify various load instructions followed by store of all registers of interest: the load word, load unsigned halfword, load signed halfword, load unsigned byte, load signed byte for all address alignments. The loads writes into all the registers of one window. The individual registers are then stored back to memory to allow improved observability on the pins of microSPARC-I.

intma

A quick and simple test to verify load and store operation. A basic diagnostic which should be used during bring-up. No self-checking for the results written to register file or memory. To improve observability, use `intma_h`.

intma_h

A quick and simple test to verify load and store operation with self-checking. A basic diagnostic which should be used during bring-up. Self-checking is provided for the results written to the register file and memory.

intma_tbr

A quick and simple test to verify load and store operation. A basic diagnostic that should be used during bring-up. No self-checking for the results written to register file and memory. Only difference between this diagnostic and `intma` is that this writes a non-zero value into the upper bits of the `tbr` register.

iu_traps_h

A toggle coverage test to toggle all the bits of the `wim` register. The test also generates a single data access exception trap.

jmp

A short test to verify the jump and call sequence. The test does the `jmp` followed by a `jmp` followed by a call followed by a `jmp`.

jmpCombo

An exhaustive test to verify the combinations of jump with various other instructions. The following cases are covered:

- `jmp` followed by a `bicc` instruction
- `jmp` followed by a `call` instruction
- `jmp` followed by a `load` instruction
- `jmp` followed by a `ldstub` instruction
- `jmp` followed by a `load double` instruction
- `jmp` followed by a `store` instruction
- `jmp` followed by a `store double` instruction
- `jmp` followed by a branch followed by a call

- jmp followed by a jmp followed by a call
- jmp followed by a read y register
- jmp followed by a read psr register
- jmp followed by a read tbr register
- jmp followed by a read wim register

jmplRett_v

This diagnostic sets up an rett to return to a supervisor page in user mode (by setting PS=0) to a trap. The fetch of the rett target gets an instruction access exception.

jmplRett_v_h

A specific test case that sets up a rett to return to a supervisor page in user mode (by setting PS=0 to a trap). The fetch of the rett target gets an instruction access exception. This test also checks for an instruction cache miss between jmpl and rett.

jmplRettvu_h

A specific test case that sets up a rett to return to a user page from a supervisor mode (by setting PS=1 prior to a trap). The fetch of the rett target will not get an instruction access exception.

lotsatog

A toggle test to improve the toggle coverage. The test will do a series of load signed byte, load unsigned byte, load signed halfword, load unsigned halfword, addcc, and series of additions.

mem_brCombo

An exhaustive test to verify the memory load/store operations followed by a bicc instruction. The following cases are covered:

- Store word followed by bicc
- Store word followed by bicc,a not taken
- Store word followed by taken
- Store word followed by bicc,a taken
- Store word followed by a branch always
- Store word followed by a branch always, annulled

- Load word followed by `bicc` not taken
- Load word followed by `bicc`,a not taken
- Load word followed by taken
- Load word followed by `bicc`,a taken
- Load word followed by a branch always.
- Load word followed by a branch always, annulled
- `ldstub` followed by `bicc` not taken
- `ldstub` followed by `bicc`,a not taken
- `ldstub` followed by taken
- `ldstub` followed by `bicc`,a taken
- `ldstub` followed by a branch always
- `ldstub` followed by a branch always, annulled

mem_jmpRett

An exhaustive test that verifies various memory reference instructions followed by `jmp`, `rett`, `fpop` instruction sequence. The following cases are covered:

- Load word followed by a `jmp`, `rett`
- Store word followed by a `jmp`, `rett`
- Load double word followed by a `jmp`, `rett`
- Store double word followed by a `jmp`, `rett`
- `ldstub` followed by a `jmp`, `rett`
- load word followed by load floating-point word, followed by interlocking `fpop`

This test uses its own trap handler.

multicycleCombo

An uninterrupted sequence of instructions —`ld st ldstub ldd std jmp`, which is designed so that each of the following instructions is followed by any other sometime during the sequence:

- Load word.
- Load unsigned halfword.
- Store word (register interlock with previous load).
- Store halfword (register interlock with previous load unsigned halfword).
- Load store unsigned byte.
- Load store unsigned byte.
- Load double word.
- Load double word.
- Store double word (register interlock with first load double word).
- Store double word (register interlock with second load double word).

- `jmpl`.
- `ldub`, `ldstub`, `ldsh`, `ldd`, `ldsb`, `std` (register interlock with previous `ldd`), `ld`, `jmpl`, `stb`.
- `lduh`, `st`, `ldd`, `std` (register interlock with previous `ldd`), `std` (register interlock with previous `ldd`), `stb`, `jmpl`, `ldstub`.
- `st`, `save`, `st`, `ldstub`, `std`, `ldstub`, `jmpl`, `ldd`.
- `ldstub`, `ldd`, `jmpl`, `std` (register interlock with previous `ldd`), `ldd`.

muldivcor_high

A corner case test for multiply, divide, address generation operations and interlock with `%g0` register. The following is the test sequence:

- Generate a very large address and return back.
- Do `umulcc` with the same source and destination register.
- Perform back-to-back unsigned multiply and unsigned divide with same destination register. One of the source registers is the same as the destination register.
- Perform back-to-back signed multiply and signed divide with same destination register. One of the source registers is the same as the destination register.
- Interlock test for load double and store double with `%g0`. Only the `%g1` register should be updated.

This test is a superset of `muldivcorner_h`.

muldivcorner_h

A corner case test for multiply, divide operations and interlock with `%g0` register. The following is the test sequence:

- Do `umulcc` with the same source and destination register.
- Perform back-to-back unsigned multiply and unsigned divide with same destination register. One of the source registers is the same as the destination register.
- Perform back-to-back signed multiply and signed divide with same destination register. One of the source registers is the same as the destination register.
- Interlock test for load double and store double with `%g0`. Only the `%g1` register should be updated.

This test is a subset of `muldivcor_high`.

mulscc_corner_h

A directed corner case test used to verify the bug found by one of the random Kaos diagnostics. This test verifies the multiply operation by using `mulscc` instructions. A `umulcc` is followed by `mulscc` instruction.

nabalux_v

This test verifies the rule that SuperSPARC-I cannot use condition codes calculated in E0 as input to extended precision arithmetic in the same group. This rule inserts a break between the condition code computation and its use in extended arithmetic (`addx`, `addxcc`, `subx`, `subxcc`).

nabcRF

An exhaustive test to verify all the software-visible 120-integer unit registers. The test sequence is as follows:

- Verify all the global registers.
- Verify all the ins registers.
- Verify all the local registers.
- Repeat the ins and local test for other windows.
- Both 0 and 1 patterns are toggled.

`RF_test2` is another exhaustive test for register file.

nabcreg_v

A diagnostic that does back-to-back read of state register instructions (read `y`, read `wim`, read `tbr`). To prevent the registers from being read while they are being modified, the SuperSPARC-I processor forces an extra cycle between RDPSR instructions and a previous arithmetic operation that may have changed the condition code. It would be more interesting for SuperSPARC-I based on the grouping rules and issue restrictions. This was a quick way to get diagnostics with interlocking, back-to-back `mulscc` instructions.

nabmulscc_v

A diagnostic that does back-to-back `mulscc` operations. It would be more interesting for SuperSPARC-I based on the grouping rules and issue restrictions. This was a quick way to get diagnostics with interlocking, back-to-back `mulscc` instructions.

nabrule14_v

A diagnostic that does forwarding of cascaded ALU results for memory address calculation in SuperSPARC-I. It would be more interesting for SuperSPARC-I based on the grouping rules and issue restrictions.

nabrule2_v

A diagnostic that verifies the operation of various address ports of the register file. A rule which would prevent a group from using too many register file ports. It would be more interesting for SuperSPARC-I based on the grouping rules and issue restrictions. This was a quick way to get some additional weird diagnostics for microSPARC-I. FIGURE 3-1 illustrates the test sequence.

```
There are 2 MEMORY-ADDR-PORTS(MP) and 2 OPERAND-
PORTS(OP). MP are accessed 1 phase earlier. Store
uses OP. Load uses MP. If OP1 is used, even if
Immediate is used, OP2 is considered used as well.

Remember that LD/ST cannot be grouped with another
LD/ST because of 2 memrefs (rule 6). However, a JMPL
uses OP, so this is for test4 in this case.

test1 - 3-instr group. Verify tasks work.

test2 - LD,LD,LD. The 3rd LD cannot be in the same
group. Has used up the 2 MPs.

test3 - LD,LD,ST. Can they be in the same group
since LD uses MP and ST uses OP?
```

FIGURE 3-1 `nabrule2_v` Test Sequence

nabrule4_v

A diagnostic used to verify the operation of various `alu` ports. The memory address and target PC calculations do not use these ALU ports. It would be more interesting for SuperSPARC-I based on the grouping rules and issue restrictions.

nabrule9_v

A diagnostic that does forwarding of cascaded ALU results for memory address calculation in SuperSPARC-I. It would be more interesting for SuperSPARC-I based on the grouping rules and issue restrictions. The diagnostic uses the floating-point instructions `fitos` and `fstoi`.

nabsetcc2_v

A diagnostic that breaks the current instruction group before the second `setcc` instruction unless the second instruction is a `mulscc` (SuperSPARC-I allows only one condition code source, resulting in simplified exception handling). SuperSPARC-I executes `mulscc` either as a single instruction group, or it executes two `mulscc`'s as a group. It would be more interesting for SuperSPARC-I based on the grouping rules and issue restrictions.

nabticcA_v

A basic test to verify two of the `ticc` conditions. This test verifies the trap always and trap never operation. Only 1 of the 16 possible combinations is tested. The five remaining cases cannot be verified by means of the arithmetic instructions. They are verified with the `wrpsr` instruction.

This test is a subset of the `nabticc_v` test.

nabticcD_v

A basic test to verify two of the `ticc` conditions. This test verifies the trap on greater or equal and trap on less operation. Only 11 of the 16 possible combinations are tested. The five remaining cases cannot be verified by means of the arithmetic instructions. They are verified with the `wrpsr` instruction.

This test is a subset of the `nabticc_v` test.

nabticc_v

An exhaustive test to verify all the `ticc` cases. Sixteen conditions are possible. For each of the 16 conditions there are 16 states of the condition codes (although not required). Eleven states of the condition code are verified by means of arithmetic instructions. The remaining 5 states of condition codes are verified by the `wrpsr` instruction.

There are two subsets of this test called `nabticcA_v` and `nabticcD_v`. They verify two cases each.

no_bypass

A basic test used to verify where bypassing should not occur. The following cases are covered by this test:

- No bypassing on `%g0`

- No bypassing from annulled delay slots
- No bypassing from branch (taken and untaken) instructions where the bits in the branches opcode corresponding to a Type 3 `rd` field indicate a register that is subsequently used
- No bypassing from call instruction where the bits in the call opcode corresponding to a Type 3 `rd` field indicate a register that is subsequently used
- No bypassing from `sethi` instruction where the bits in the call opcode corresponding to a Type 3 `rd` field indicate a register that is subsequently used
- No bypassing between a store and `alu` where the read of the store is a read of a source of the ALU
- No bypassing between a store double and `alu` where the read of the store is a read of source of the ALU

no_del_rett

A basic test to verify the case where the delay slot of a `jmpl` is a `rett` and the `rett` in missing from the instruction buffers. Two cases are tested:

- When the `jmpl` and the `rett` transfer control to contiguous instructions.
- When the instructions are not contiguous.

Since `rett` does a window restore, a stack area is established and set up in the registers. Do a `save` prior to each `rett` to make sure that there is no window underflow trap. Also, `rett` requires that `ET` (enable trap) be 0 and it will set it to 1. Therefore, set `ET` to 0 prior to each `rett`; otherwise, an illegal instruction trap occurs. S (supervisor bit) should always be 1.

page_interlock_h

A corner case test to verify the following situation.

A load is followed by an interlocking (the destination register of load is one of the source registers) add. The two instructions are on two different pages. The load is the last instruction on a user page, and the add is the first instruction on the next supervisor page. The add instruction will cause an instruction access exception.

psr1

A basic test that verifies all instructions that change the current window pointer. The instructions are `wrpsr`, `save`, `restore`, `ticc`, and `rett`. The test sequence is as follows:

- Using `wrpsr`, check for current `cwp` – window five.
- Using `wrpsr`, check for current `cwp` – window four.
- Using `wrpsr`, check for current `cwp` – window three.

- Using `wrpsr`, check for current `cwp` – window two.
- Using `save` instruction, check for `cwp`.
- Using `restore` instruction, check for `cwp`.
- Using `trap always` instruction, check for `cwp`.
- Using `rett` instruction, check for `cwp`.
- A sequence of `call`, `save`, `restore` instructions.

This test uses its own trap handler.

rdspreg

A basic test used to verify the operation of the programmer visible state registers. The following cases are covered:

- Write `y` register, followed by read `y` register
- Write `psr` register, followed by read `psr` register
- Write `wim` register, followed by read `wim` register
- Write `tbr` register, followed by read `tbr` register
- Write `y` register
- Read `y` register, followed by interlocking load instruction
- Read `y` register, followed by interlocking store instruction
- Read `y` register, followed by interlocking load store unsigned byte instruction
- Read `y` register, followed by interlocking `jmpl` instruction
- Read `y` register, followed by interlocking load double instruction
- Read `y` register, followed by interlocking store double instruction
- Read `y` register, followed by floating point instruction
- Read `y` register, followed by interlocking save instruction
- Read `y` register, followed by interlocking restore instruction
- Read y register, followed by interlocking trap always instruction
- Read `y` register, followed by interlocking tagged add, modify `icc`, and trap on overflow instruction
- Read `y` register, followed by interlocking tagged subtract, modify `icc`, and trap on overflow instruction
- Read `y` register, followed by interlocking load followed by interlocking `jmpl` instruction

readspec

A quick test to verify write to programmer-visible state registers followed by read of the register. The test sequence is as follows:

- Write `psr`
- Write `wim`
- Write `tbr`

- Write `y`
- Read `psr`
- Read `y`
- Read `wim`
- Read `tbr`

The test also includes some `add` and `andn` instructions.

rett2

A basic test to verify combinations of jump followed by `rett` in various situations. The following cases are covered:

- All of the traps and the trap prioritizing that can occur on RETT:
  - Illegal instruction caused by ET = 1 and S = 1
  - Privileged instruction caused by S = 0 and ET = 1
  - Privileged instruction (converted to `reset`) caused by S = 0 and ET = 0
  - Window underflow (converted to `reset`)
  - Misaligned address (converted to `reset`)
- Multicycle and other interesting targets for JMP/RETT
  - `ld/st`
  - `ldd/std`
  - `ldst/ld`
  - not taken branch
  - `wrpsr`
  - save
  - restore

This test uses its own trap handler. This test causes a watch-dog reset. This test will cause an endless loop in Verilog simulation.

rett_sps

A basic test to verify `jmpl/rett` will fetch instructions with the correct permission (user or supervisor). The following cases are covered:

- Jump hits in the cache and `rett` misses.
- Target of `jmp` hits but the target of `rett` misses
- `jmp` and `rett` target misses

sdiv_byp_bug

A directed test to verify the divide bypass bug for the carry overflow bit. When a `sdivcc` instruction is followed by a `subx`/`addx`, the `subx`/`addx` instruction thinks the carry bit is 1, even though the `sdiv`/`udiv` instruction always sets the C bit to 0. This is an operand corner case for generation of C condition code.

The operands and sequence were discovered by a random Kaos program. The original failure occurred at 99k+ q cycles, and a directed test was written to verify the fix.

shift_test

A basic test to verify the functionality of the shift unit. It does `sll`, `srl`, and `sra`. It checks that the result has been shifted the proper amount by examining the final register contents. Cases of small (0 or 1), large (31 or 32) and normal (type 7, 11, 21) shift amounts are done. Since the shifter only examines the low 5 bits of the shift amount, large numbers with higher bits set should only cause a shift amount equal to the number contained in the low 5 bits. The test sequence is as follows:

- `sll` by small amount (0 or 1)
- `sll` by big amount (32 or 31)
- `sll` by normal amount (7, 11, 21)
- `srl` by small amount (0 or 1)
- `srl` by big amount (32 or 31)
- `srl` by normal amount (7, 11, 21)
- `sra` for positive number by small amount (0 or 1)
- `sra` for positive number by big amount (32 or 31)
- `srl` for positive number by normal amount (7, 11, 21)
- `sra` for negative number by small amount (0 or 1)
- `sra` for negative number by big amount (32 or 31)
- `srl` for negative number by normal amount (7, 11, 21)

st_divcombo

A basic test to verify the divide combination with loads and stores. The test checks the following sequences:

- Load followed by `sdiv`
- Load followed by interlocking `sdiv`
- `sdiv` followed by load
- Store followed by `sdiv`
- `sdiv` followed by interlocking store

test_Bicc

An exhaustive test to verify the Branch on Integer Condition Code instructions. For the 12 possible combinations of condition codes, test all of the 16 Bicc instructions.

t-s-g-i

A basic test to verify the special group (state register instructions). The test verifies the following cases:

- Write `tbr` followed by read `tbr`
- Write `psr` followed by read `psr`
- Write `wim` followed by read `wim`
- Write `y` followed by read `y`

taddcc_h

A corner case test to verify that if a `taddcctv` causes a tag overflow, a tag overflow trap is generated and the condition codes remain unchanged.

brfold.s*

Test different instruction categories (non-load/-store) in the delay slot of a folded branch. Following instructions are verified: add, addcc, bicc, call, mulscc, nop, rdy, restore, save, sdivcc, sethi, sll, smulcc, sub, subcc, ticc, udiv, umul, wry, xnorcc, xor

brfold1.s

A conditional taken branch.

brfold2.s

A conditional taken branch (`annul` on).

brfold3.s

A conditional untaken branch.

brfold4.s

A conditional untaken branch (`annul` on).

brfold5.s

An unconditional taken branch.

brfold6.s

An unconditional untaken branch.

brfold7.s

An unconditional taken branch (`annul` on).

brfold8.s

An unconditional untaken branch (`annul` on).

brfldst.s*

Tests load and store operations in the delay slot of a folded branch. The following instructions are verified. `st, ld, std, ldd, stb, ldub, sth, lduh, ldsb, ldsh, sta, lda, stda, ldda, stba, lduba, stha, lduha, ldsba, ldsha, ldstub, swap, ldstuba, swapa, stf, ldf, stdf, lddf, stfsr, ldfsr, stdfq, iflush, sta` flush operations.

brfldst1.s

A conditional taken branch.

brfldst2.s

A conditional taken branch (`annul` on).

brfldst3.s

A conditional untaken branch.

brfldst4.s

A conditional untaken branch (`annul` on).

brfldst5.s

An unconditional taken branch.

brfldst6.s

An unconditional untaken branch.

brfldst7.s

An unconditional taken branch (`annul` on).

brfldst8.s

An unconditional untaken branch (`annul` on).

brftag.s

Test tagged addition and subtraction instructions in the delay slot of a folded branch.

brftraps.s

Verifies traps generated in the delay slot of a folded branch. It uses its own trap handler. Not all trap types are tested in this diagnostic. The purpose is to verify the program counter of the IU.

brfatraps.s

A directed corner case test that verifies the bug found by one of the random Kaos diagnostics. This diagnostic verifies traps detected in the delay slot of a conditional untaken (`annul` on) folded branch. The diagnostic uses its own trap handler. The following traps are covered:

- `div-by-0`
- `instr-access-exception`
- `data-access-exception`
- Privileged instruction
- Illegal instruction
- `fp-disabled`
- `cp-disabled`
- `win-overflow`
- `win-underflow`
- `mem-addr-not-align`
- `fp-exception`
- `tag-overflow`
- Trap instruction

br_traps.s

This diagnostic verifies the program counter in the IU when traps are detected before and after a branch instruction. Since traps occur before a branch instruction, that particular branch instruction is not folded.

br_ilock.s

This diagnostic verifies interlock detection during unfolded `bicc/call` instruction. An unfolded `bicc/call` instruction is achieved by having this instruction in the delay slot of a folded branch. A `bicc/call` interlock is achieved by causing `rs1` or `rs2` field of the `bicc/call` to match the destination register of a previous load operation. This normally should not cause interlock, but since interlock is detected faster by not looking into the `bicc/call` opcode, this case of interlock is allowed.

ldst.s

Verifies that loads and stores work correctly including bypasses. The following cases are covered: `ld/ldd, ldd/ld, st/std, std/st, ld/st, ldd/st, ld/std, ldd/stdld/`nop`/st, ldd/`nop`/st, ld/`nop`/std, ldd/`nop`/std, ld/bicc/st, ldd/bicc/st, ld/bicc/std, ldd/bicc/std`

All above cases with `%g0` as destination register (no bypass).

atomic.s

Verifies that atomic operations, including bypasses, work correctly. The following cases are covered: `ld/ldstub, ldd/ldstub, st/ldstub, std/ldstub, ld/nop/ldstub, ldd/nop/ldstub, st/nop/ldstub, std/nop/ldstub, ld/bicc/ldstubld/swap, ldd/swap, st/swap, std/swap, ld/nop/swap, ldd/nop/swap, st/nop/swap, std/nop/swap, ld/bicc/swap`

All above cases with `%g0` as destination register (no bypass).

st_ilock.s

A basic test to verify bypasses during load single/store interlock with and without traps. This diagnostic uses its own trap handler. The following cases are covered: `ld/st, ld/std, ld/stb, ld/sth, ld/bicc/st, ld/bicc/std, ld/bicc/stb, ld/bicc/sth`

All above cases with `%g0` as destination register (no bypass).

ldd_ilock.s

A basic test to verify bypasses during load double/store interlock with and without traps. This diagnostic uses its own trap handler. The following cases are covered: `ldd/st, ldd/std, ldd/stb, ldd/sth, ldd/bicc/st, ldd/bicc/std, ldd/bicc/stb, ldd/bicc/sth`

All above cases with `%g0` as destination register (no bypass).

ldub_ilock.s

A basic test to verify bypasses during load unsigned byte/store interlock with and without traps. This diagnostic uses its own trap handler. The following cases are covered: `ldub/st, ldub/std, ldub/stb, ldub/sth, ldub/bicc/st, ldub/bicc/std, ldub/bicc/stb, ldub/bicc/sth`

All above cases with `%g0` as destination register (no bypass).

lduh_ilock.s

A basic test to verify bypasses during load unsigned halfword/store interlock with and without traps. This diagnostic uses its own trap handler. The following cases are covered: `lduh/st, lduh/std, lduh/stb, lduh/sth, lduh/bicc/st, lduh/bicc/std, lduh/bicc/stb, lduh/bicc/sth`

All above cases with `%g0` as destination register (no bypass).

ldsb_ilock.s

A basic test to verify bypasses during load signed byte/store interlock with and without traps. This diagnostic uses its own trap handler. The following cases are covered: `ldsb/st, ldsb/std, ldsb/stb, ldsb/sth, ldsb/bicc/st, ldsb/bicc/std, ldsb/bicc/stb, ldsb/bicc/sth`

All above cases with `%g0` as destination register (no bypass).

ldsh_ilock.s

A basic test to verify bypasses during load signed halfword/store interlock with and without traps. This diagnostic uses its own trap handler. The following cases are covered. `ldsh/st, ldsh/std, ldsh/stb, ldsh/sth, ldsh/bicc/st, ldsh/bicc/std, ldsh/bicc/stb, ldsh/bicc/sth`

All above cases with `%g0` as destination register (no bypass).

atomic_ilock.s

A basic test to verify bypasses during a load or store operation followed by an interlocked atomic operation that generates a trap. This diagnostic uses its own trap handler. The following cases are covered: `ld/ldstub, ldd/ldstub, st/ldstub, std/ldstub, ld/bicc/ldstub, ld/swap, ldd/swap, st/swap, std/swap, st/bicc/swap`

All above cases with `%g0` as destination register (no bypass).

synch_trap2.s

This diagnostic verifies trap priority logic for multiple traps issued by a single instruction. This diagnostic tests for `illegal_instruction` trap, `privileged` trap, `fp-disabled` trap, `mem_address_misalign` trap, and `data_access_exception` traps.

synch_trap3.s

This diagnostic verifies trap priority logic for multiple traps issued by a single instruction. This diagnostic tests for `illegal_instruction` trap, privileged trap, `fp-disabled` trap, `mem_address_misalign` trap and `data_access_exception` traps.

rett3.s

This diagnostic verifies the trap priority logic during execution of a `rett` instruction in the trap handler and causes IU to go to error mode. The reset handler for this diagnostic has been modified to verify the next case after the IU goes to error mode.

rett4.s

This diagnostic tests the ASI field on load/store instructions and instruction or data access exception traps around the `rett` instruction.

Case 1: Return in user mode from a trap handler (ps = 0). Target of the `jmp` must be a `ld/st` to a supervisor page only. This should cause data access exception. Target of the `jmp` must be a to a user page, and make sure no trap occurs.

Case 2: Return in user mode from a trap handler (ps = 0). Target of the `jmp` must be to a supervisor page; this should cause instruction access exception. Target of the `jmp` is to a user page, and no trap occurs.

RF8_test2.s

Diagnostic to test IU register file (8 windows). Covers all software-visible 120 Integer Unit registers. The following tests are done for each register:

- Write unique address into each location to verify decoding of addresses
- Write complement of unique address into each location
- Write checkerboard pattern (1010) into each location
- Write reverse checkerboard pattern into each location

RF8_test3.s

Shorter version of `RF8_test2.s`.

ticc_bicc.s

Tests branch folding for `ticc/bicc` pairs.

CHAPTER 4

Floating-point Unit Tests

This chapter covers the microSPARC-IIep floating-point unit diagnostic tests.

Fallinst_v

A basic test for all instructions for floating-point unit operations. It also includes integer multiply and divide instructions that are new to the SPARC Version 8 architecture. The test includes the following instructions:

- disable floating-point bit in the PSR
- enable floating-point bit in the PSR
- `fadds`, `faddd`, `fsubs`, `fsubd` – check result
- `fadds`, `udiv`, `fmuls`, `umul`, `fdivd`, `smul`, `fsubs`, `sdiv`, `fsmuld` (mix floating-point instructions with multicycle integer instructions) – check result.
- `faddx`, `fmulx` – unimplemented instruction - check result
- `fcmps`, `fbne` – check result
- `fcmps`, fcmpes, fcmpd, fcmped
- `fdivs`, `fadds`, `fmuls`, `fsubs`, `faddd`, `fsqrts`, `fsubd`, `fsqrtd`, `fmovs`, `fsmuld` (mix adder with multiplier) – check result
- floating-point divide by zero
- `fmovs` followed by interlocking store – check result
- `fmovs`, `fnegs`, `fabss`, `fmovs` – check result
- `fstoi`, `fdtoi`, `fitos`, `fdtos`, `fitod`, `fstod` – check result
- `fmuls` followed by interlocking `fsubs` – check result
- `fmuls`, `fdivs`, `fmuld`, `fdivd`, `fsqrtd` – check result
- `fsqrts`, `fsmuld` – check result
- load single floating-point register
- load double floating-point register
- load single floating-point register followed by interlocking `fmovs` – check result
- store all floating-point registers
- store `fsr`, load `fsr` – check result
- `udiv` by zero
- `wry`, `udiv`, `udivcc`, `sdiv`, `sdivcc` – check result
- `wry`, `umul`, `umulcc`, `smul`, `smulcc` – check result

This is a complementary test to `allInsts`. It covers all the floating-point instructions.

anl_ds_trap

This diagnostic checks the operation of floating-point instructions that would normally cause traps but are in the annulled delay slot of a branch. The FP traps checked in this diagnostic are IEEE exceptions and unfinished exceptions. The annulling branches are both IU and FP branches. If the FP trap is taken, the trap handler will write error number 1 to the log area and then exit.

- Try a delay slot that is annulled and when all TEM bits are set. Try IEEE exception. Branch is IU branch.
- Try the same thing as above except try a FP branch.
- Instead of an IEEE exception in the annulled delay slot, put in an instruction that generates an unfinished exception.

The microSPARC-IIep design does not signal any unfinished exception, so this diagnostic only signals an IEEE exception.

chain_double

This diagnostic is a long test for various floating-point operations with register interlock. The test covers double-precision floating-point add, multiply, and subtract. There are 18 sets of tests. The test starts with no delay between the interlocking instructions and ranges to having 18 nops between interlocking instructions. The test is self-checking and stores results to memory.

chain_double_sub

This diagnostic is a long test for various floating-point operations with register interlock. The test covers double-precision floating-point add, multiply, and subtract. There are nine sets of tests. The test starts with 10 nops between the interlocking instructions and ranges to having 18 nops between interlocking instructions. The test is self-checking and stores results to memory.

chain_single

This diagnostic is a long test for various floating-point operations with register interlock. The test covers single-precision floating-point add, multiply, and subtract. There are 16 sets of tests. The test starts with no delay between the interlocking instructions and ranges to having 16 nops between interlocking instructions. The test is self-checking and stores results to memory.

chain_single_lock

This diagnostic is a long test for various floating-point operations with register interlock. The test covers the floating-point double-to-single convert instruction and single-precision multiply instructions. There are 16 sets of tests. The test starts with no delay between the interlocking instructions and ranges to having 16 nops between interlocking instructions. The test is self-checking and stores results to memory.

depAluSrc

This diagnostic tests register dependencies in the floating-point unit when Fpops are followed by Fpops whose source1 or source2 operand is the first Fpop's result register. This diagnostic assumes that the chip's mechanism to keep track of busy registers is independent of the incoming instruction to be checked. With this assumption, it is unnecessary to test every possible Fpop followed by all possible Fpops. The complete set of `depAluSrc`*`XX`* diagnostics creates situations for all Fpops that have result registers, so that these Fpops are followed by Fpops that immediately use the result. They also make sure that each Fpop with register operands is preceded at least once by an Fpop that generates these operands. This diagnostic should be run with the standard SPARC register organization (RMODE=0).

The test stores the final result of each set of operation to memory. The instructions covered are `faddd, fabss, fmovs, fmuld`, and `fnegs.`

depAluSrc11

This diagnostic tests register dependencies in the floating-point unit when Fpops are followed by Fpops whose source_one operand is the first Fpop's result register. This diagnostic assumes that the chip's mechanism to keep track of busy registers is independent of the incoming instruction to be checked.With this assumption, it is unnecessary to test every possible Fpop followed by all possible Fpops. The complete set of `depAluSrc`*`XX`* diagnostics creates situations for all Fpops that have result registers, so that these Fpops are followed by Fpops that immediately use the result. They also make sure that each Fpop with register operands is preceded at least once by an Fpop that generates these operands. This diagnostic should be run with the standard SPARC register organization (RMODE=0).

The test stores the final result of each set of operation to memory. The instructions covered are: `fabss, faddd, fadds, fcmpd, fcmped, fcmps, fcmpes, fdivs, fdtos, fitod, fitos, fmovs, fmuld, fmuls, fnegs, fstod, fsubd, and fsubs`.

depAluSrc21

This diagnostic tests register dependencies in the floating-point unit when Fpops are followed by Fpops whose source-two operand is the first Fpop's result register. This diagnostic assumes that the chip's mechanism to keep track of busy registers is independent of the incoming instruction that is to be checked. With this assumption it is unnecessary to test every possible Fpop followed by all possible Fpops. The complete set of `depAluSrc`*`XX`* diagnostics creates situations for all Fpops that have result registers, so that these Fpops are followed by Fpops that immediately use the result. They also make sure that each Fpop with register operands is preceded at least once by an Fpop that generates these operands. This diagnostic should be run with the standard SPARC register organization (RMODE=0).

The test stores the final result of each set of operation to memory. The instruction covered are: `fabss, faddd, fadds, fcmpd, fcmped, fcmpes, fcmps, fdivd, fdivs, fdtos, fitod, fitos, fmovs, fmuld, fmuls, fnegs, fstod, fsubd, and fsubs.`

depAluSrc22

This diagnostic tests register dependencies in the floating-point unit when Fpops are followed by Fpops whose source-two operand is the first Fpop's result register. This diagnostic assumes that the chip's mechanism to keep track of busy registers is independent of the incoming instruction to be checked. With this assumption, it is unnecessary to test every possible Fpop followed by all possible Fpops. The complete set of `depAluSrc`*`XX`* diagnostics creates situations for all Fpops that have result registers, so that these Fpops are followed by Fpops that immediately use the result. They also make sure that each Fpop with register operands is preceded, at least once, by an Fpop that generates these operands. This diagnostic should be run with the standard SPARC register organization (RMODE=0).

The test stores the final result of each set of operation to memory. The instruction covered are `fabss`, `fdtoi`, `fdtos`, `fitod`, `fitos`, `fmovs`, `fnegs`, `fstod`, and `fstoi`.

dtoi_m

This diagnostic tests floating-point double-to-integer conversions. The test will cause inexact, no, and overflow exceptions. The operands are negative.

dtoi_m0

This diagnostic does floating-point double-to-integer conversions. The rounding direction is zero (nearest). The test will cause inexact, no exceptions. The operands are negative.

dtoi_m1

This diagnostic does floating-point double-to-integer conversions. The rounding direction is one (toward zero). The test will cause inexact, no exceptions. The operands are negative.

dtoi_m2

This diagnostic does floating-point double-to-integer conversions. The rounding direction is two (positive infinity). The test will cause inexact, no exceptions. The operands are negative.

dtoi_m3

This diagnostic does floating-point double-to-integer conversions. The rounding direction is three (negative infinity). The test will cause inexact, no exceptions. The operands are negative.

dtoi_p

This diagnostic tests floating-point double-to-integer conversions. The test will cause inexact, no, and overflow exceptions. The operands are positive.

dtoi_p0

This diagnostic does floating-point double-to-integer conversions. The rounding direction is zero (nearest). The test will cause inexact, no exceptions. The operands are positive.

dtoi_p1

This diagnostic does floating-point double-to-integer conversions. The rounding direction is one (toward zero). The test will cause inexact, no exceptions. The operands are positive.

dtoi_p2

This diagnostic does floating-point double-to-integer conversions. The rounding direction is two (positive infinity). The test will cause inexact, no exceptions. The operands are positive.

dtoi_p3

This diagnostic does floating-point double-to-integer conversions. The rounding direction is three (negative infinity). The test will cause inexact, no exceptions. The operands are positive.

fabort1_v

An IU takes an exception when one or more FPops are still pending.

fabort2_v

An IU takes an exception when one or more FPops are still pending.

A bug was found with the use of unassigned ASI, rollover ASI.

fabs_sp_v

A test for IEEE FP single-precision absolute value with all four rounding modes (nearest, zero, positive infinity and negative infinity). The test is a loop of 4 load doubles (each load double gets two operands for FP operation), 4 Fabss and 2 stores. The loop count is 20. The expected result and `fsr` register are stored and checked.

fadd_dp_v

A test for IEEE FP double-precision add operation with all four rounding modes (nearest, zero, positive infinity, and negative infinity). The test is a loop of 8 load doubles, 4 Faddd and 2 stores. The loop count is 300. The expected result and `fsr` register are stored and checked.

fadd_sp_v

A test for IEEE FP single-precision add operation with all four rounding modes (nearest, zero, +ve infinity, and -ve infinity). The test is a loop of 4 load doubles (each load double gets two operands for FP operation), 4 Fadds and 2 stores. The loop count is 300.The expected result and `fsr` register are stored and checked.

fall_opf

An exhaustive test to verify all possible 1,024 SPARC FPops, by varying the `opf` field of the instruction (512 type1, 512 type2). Only 23 of these should not cause an unimplemented exception. The legal FPops are: `fabss, faddd, fadds, fcmpd, fcmped, fcmpes, fcmps, fdivd, fdivs, fdtoi, fdtos, fitod, fitos, fmovs, fmuld, fmuls, fnegs, fsqrtd, fsqrts, fstod, fstoi, fsubd, and fsubs.`

The special fp trap handler was made as small as possible to reduce the runtime of the diagnostic. It will have to be changed for different implementations.

falu_simple_v

This diagnostic verifies basic single-precision and double-precision FP ALU operations. The test loads operands in to the FPU registers, executes `faddd, fadds fcmpd, fcmps, fdtoi, fdtos, fitod, fitos, fmovs, fstod, fstoi, fsubd, and fsubs,` and stores the results into memory.

farith_1_v

This diagnostic verifies all FP ALU operations with weird and extreme arithmetic numbers. This test is useful in verifying the circuit schematics for the datapath. The test loops on load of FP registers `%f0`–`%f16`, load `fsr`, `fadds`, `fsubs`, `fmovs`, `fabss`, `faddd`, `fsubd`, `fcmps`, followed by store of results into memory, followed by load of FP registers `%f0`–`%f16`, load `fsr`, `fstoi`, `fstod`, `fmovs`, `fdtoi`, `fdtos`, `fabss`, `fitos`, `fitod`, `fnegs`, `fcmpd`, followed by store of results into memory. This loop is repeated for about 10 sets of operands and verified in all four rounding modes.

fcexc_v

This diagnostic verifies that the cexc bits in the `fsr` are cleared by the FP instructions `fabss`, `fadds`, `fcmps`, `fmovs`, and `fnegs`.

fcmp_dp_v

This diagnostic is a test for IEEE FP double-precision compare operation with all four rounding modes (nearest, zero, +ve infinity, and -ve infinity). The test is a loop of 8 load doubles, 4 Fcmpd and 2 stores. The loop count is 360. The expected result and `fsr` register are stored and checked. The operands are equal.

fcmp_sp_v

This diagnostic is a test for IEEE FP single-precision compare operation with all four rounding modes (nearest, zero, +ve infinity, and -ve infinity). The test is a loop of 4 load doubles, 4 Fcmps and 2 stores. The loop count is 360. The expected result and `fsr` register are stored and checked. The operands are equal.

fd2si_v

This diagnostic is a test for IEEE FP double-precision convert operation with all four rounding modes (nearest, zero, +ve infinity, and -ve infinity). The test is a loop of 4 load doubles, 4 Fdtos, 4 Fdtoi, and 8 stores. The loop count is 68. The expected result and `fsr` register are stored and checked.

fdepend_1_v

This diagnostic verifies FP register dependencies. The test loads single- and double-precision operands from memory into registers. The first part of the test involves only single-precision operands, and the second part of the test involves double-precision operands.

The interlock operations are as follows:

- `fadds %f0`, `%f1`, `%f28`
- `fadds %f28`, `%f2`, `%f29`

The interlock is between the source_one register or source_two register of an instruction and the destination register of a previous instruction.

The test inserts one to four instructions between the interlocking instructions.

fdepend_2_v

This diagnostic verifies FP register dependencies. The test loads single- and double-precision operands from memory into registers. The first part of the test involves interlock operation between single-precision operands register with double-precision operands registers. The second part of the test involves double-precision operands registers with single-precision operands.

The interlock is between the source_one register or source_two register of an instruction and the destination register of a previous instruction.

fdepend_dp_v

This diagnostic verifies double-precision FP register dependencies. The test loads single- and double-precision operands from memory into registers. A complete mix of source_one, source_two, and destination and queue positions is used. This test is more appropriate for SuperSPARC-I than for microSPARC-IIep.

For different positions in the queue, the same source operands are used.

fdepend_sp_v

This diagnostic verifies single-precision FP register dependencies. The test loads single- and double-precision operands from memory into registers. A complete mix of source_one, source_two, and destination and queue positions is used. This test is more appropriate for SuperSPARC-I than for microSPARC-IIep.

For different positions in the queue, the same source operands are used.

fdiv1_v

This diagnostic verifies basic functionality of floating-point multiply, divide, and square root operations using single- and double-precision operands. The test loads single- and double-precision operands from memory into registers. A single operation involving `fmuls`, `fdivs`, `fsqrts`, `fmuld`, `fdivd`, and `fsqrtd` is executed in the test.

fldf_1_v

This diagnostic verifies floating-point compare operations with all mixes of integer and floating-point operations with load and store of integer of integer registers and floating-point registers. The test also checks for proper operation of `fcompare` and `fbfcc` instructions. The following is the test sequence:

- Load integer register followed by integer instruction followed by floating-point instruction.
- Move register value followed by integer instruction followed by load integer register followed by floating-point instruction.
- Store integer register followed by integer instruction followed by floating-point instruction.
- Move register value followed by integer instruction followed by store integer register followed by floating-point instruction.
- Load floating-point register followed by integer instruction followed by floating-point instruction.
- Move register value followed by integer instruction followed by load floating-point register followed by floating-point instruction.
- Store floating-point register followed by integer instruction followed by floating-point instruction.
- Move register value followed by integer instruction followed by store floating-point register followed by floating-point instruction.
- Load integer registers followed by floating-point instruction.
- Store integer registers followed by floating-point instruction
- Move floating-point register value followed by floating-point instruction followed by load integer register.
- Move floating-point register value followed by floating-point instruction followed by store integer register.
- Load floating-point registers followed by floating-point instruction.
- Store floating-point registers followed by floating-point instruction.
- Move floating-point register value followed by floating-point instruction followed by load floating-point register.
- Move floating-point register value followed by floating-point instruction followed by store floating-point register.
- Load integer registers followed by floating-point compare followed by floating-point condition code instruction.
- Move floating-point register followed by floating-point compare followed by integer load followed by floating-point condition code instruction.
- Store integer registers followed by floating-point compare followed by floating-point condition code instruction.
- Move floating-point register followed by floating-point compare followed by integer store followed by floating-point condition code instruction.
- Load floating-point registers followed by floating-point compare followed by floating-point condition code instruction.

- Move floating-point register followed by floating-point compare followed by load floating-point register followed by floating-point condition code instruction.
- Store floating-point registers followed by floating-point compare followed by floating-point condition code instruction.
- Move floating-point register followed by floating-point compare followed by store floating-point registers followed by floating-point condition code instruction.

fldf_2_v

This diagnostic verifies the store floating-point register followed by a floating-point operation. The important case to consider is when the floating-point instruction completes in the same cycle as the store floating-point register. The following is the test sequence:

- Store floating-point single followed by floating-point instruction (`fmovs`, `fstod`, `fdtos`, `fsqrtd`, `fadds`, `fsmuld`, `fmuld`, `fcmps`, and `fcmpd`) with no register dependency.
- Store floating-point single followed by floating-point instruction (`fmovs`, `fstod`, `fdtos`, `fsqrtd`, `fadds`, `fsmuld`, `fmuld`, `fcmps`, and `fcmpd`) with source_one register dependency, that is, the source register for store and floating-point instruction is the same.
- Store floating-point single followed by floating-point instruction (`fmovs`, `fstod`, `fdtos`, `fsqrtd`, `fadds`, `fsmuld`, `fmuld`, `fcmps`, and `fcmpd`) with destination register dependency, that is, the source register for store and destination register for floating-point instruction is the same.
- Floating-point instruction (`fmovs`, `fstod`, `fdtos`, `fsqrtd`, `fadds`, `fsmuld`, `fmuld`, `fcmps`, and `fcmpd`) followed by store floating register with no register dependency.
- Floating-point instruction (`fmovs`, `fstod`, `fdtos`, `fsqrtd`, `fadds`, `fsmuld`, `fmuld`, `fcmp`, and `fcmpd`) followed by store floating-point register with source_one register dependency, that is, the source register for store and floating-point instruction is the same.
- Floating-point instruction (`fmovs`, `fstod`, `fdtos`, `fsqrtd`, `fadds`, `fsmuld`, `fmuld`, `fcmps`, and `fcmpd`) followed by store floating-point register with destination register dependency, that is, the source register for store and destination register for floating-point instruction is the same.

`fsmuld` is an unimplemented instruction in the microSPARC-IIep technology.

fldf_basic_v

This simple diagnostic verifies the basic operation of load floating-point registers, store floating-point registers, and single-precision floating-point add. All the loads and stores are done with single-precision operands and registers. To prevent interlocks, additional nops are inserted.

fldf_depend_v

This diagnostic verifies the basic operation of load floating-point registers, store floating-point registers, and single-precision floating-point add. All the loads and stores are done with single-precision operands and registers. The test has some interlock checking. The test code sequence is:

- Load floating-point register_one.
- Load floating-point register_two.
- Floating-point add single-precision register_one, register_two, and destination_one.
- Store floating-point register_one.
- Store floating-point register_two.
- Store floating-point destination_one.

fldstfsr_v

This diagnostic verifies load and store of individual bits of the floating-point status register. The version number bits are not checked. The individual bits are toggled for 0 and 1 state.

fmix1_v

This basic diagnostic verifies the floating-point operations involving the `adder` and comparator logic blocks. The test initially loads the operands from memory into the floating-point registers. A single-precision operation is performed, and the result is stored into memory. The test later performs double-precision operation and the result is stored into memory. The test code sequence involves the following floating-point operations: `fcmps` (equal, less than, greater than, and unordered), `fitos`, `fstoi`, `fstod`, `fnegs`, `fabss`, `fcmpd` (equal, less than, greater than, and unordered), `faddd`, `fsubd`, `fitod`, `fdtoi`, `fdtos`.

fmix2_v

This exhaustive diagnostic verifies the dependency of floating-point adder and floating-point multiplier. A mix of all floating-point adder and multiplier instructions is used. Thirty-five cases are covered, based on the following matrix:

- `fm_ss2s`: Floating-point multiplier operation involving two single-precision source operands and a single-precision result. The instructions involved are `fmuls` and `fdivs`.
- `fm_dd2d`: Floating-point multiplier operation involving two double-precision source operands and a double-precision result.The instructions involved are `fmuld` and `fdivd`.
- `fm_s2s`: Floating-point multiplier operation involving a single-precision source operand and a single-precision result. Only the `fsqrts` instruction falls in this category.
- `fm_d2d`: Floating-point multiplier operation involving a double-precision source operand and a double-precision result. Only the `fsqrtd` instruction falls in this category.
- `fm_ss2d`: Floating-point multiplier operation involving two single-precision source operands and a double-precision result. Only the `fsmuld` instruction falls in this category.
- `fa_ss2s`: Floating-point adder operation involving two single-precision source operands and a single-precision result. The instructions involved are `fadds` and `fsubs`.
- `fa_dd2d`: Floating-point adder operation involving two double-precision source operands and a double-precision result. The instructions involved are `faddd` and `fsubd`.
- `fa_s2s`: Floating-point adder operation involving a single-precision source operand and a single-precision result. The instructions involved are `fmovs`, `fabss`, and `fnegs`.
- `fa_d2d`: Floating-point adder operation involving a double-precision source operand and a double-precision result.The instruction involved are `fmovs`, `fabss`, and `fnegs`.
- `fa_d2s`: Floating-point adder operation involving a double-precision source operand and a single-precision result.The instructions involved are `fdtos` and `fdtoi`.
- `fa_ss2-`: Floating-point multiplier operation involving two single-precision source operands only. Only the `fcmps` instruction falls in this category.

- `fa_dd2-`: Floating-point multiplier operation involving two double-precision source operands only. Only the `fcmpd` instruction falls in this category.

TABLE 4-1 Floating-point Adder and Multiplier Test Cases

Option	**`fm_ss2s`**	**`fm_dd2d`**	**`fm_s2s`**	**`fm_d2d`**	**`fm_ss2d`**
fa_ss2s	am_11	am_12	am_13	am_14	am_15
fa_dd2d	am_21	am_22	am_23	am_24	am_25
fa_s2s	am_31	am_32	am_32	am_33	am_34
fa_s2d	am_41	am_42	am_43	am_44	am_45
fa_d2s	am_51	am_52	am_53	am_54	am_55
fa_ss2-	am_61	am_62	am_63	am_64	am_65
fa_dd2-	am_71	am_72	am_73	am_74	am_75

fmult1_v

This diagnostic is a basic test to verify the floating-point add and multiply operation with exceptions. The test involves loading the floating-point register file with operands from memory. The `fmuls`, `fmuld`, `fadds`, and `faddd` instructions are used in the test.

fmult2_v

This diagnostic is a million-cycle microSPARC-II test to verify exhaustively all the floating-point multiplier block instructions like `fmuls`, `fmuld`, `fdivs`, `fdivd`, `fsqrts`, and `fsqrtd`. It involves all four rounding modes (nearest, zero, positive infinity, and negative infinity). The loop count of 500 will result in about million cycles. Normally the count is set to 5. The traps are disabled in this test.

fn3_single

This diagnostic is a long test to verify single-precision FPU operations. This test does not generate any IEEE exceptions. The test sequence is as follows:

- Check load `fsr` followed by store `fsr`.
- Check single-precision, floating-point numbers to integer conversion.
- Check single-precision, floating-point numbers to double-precision conversion.
- Check double-precision, floating-point numbers to single-precision conversion.
- Check integer-to-single conversion.

- Check single-precision moves (`fmovs`).
- Check double-precision moves (`fmovd`).
- Check single-precision negations (`fnegs`).
- Check single-precision absolute value (`fabss`).
- Check that less than / greater than comparison works.
- Check single-precision additions (`fadds`).
- Check single-precision subtractions (`fsubs`).
- Check single-precision multiplications (`fmuls`).
- Check single-precision divisions (`fdivs`).
- Check back-to-back `fadds`, `fmuls`, `fdivs` sequence.

fn4_double

This diagnostic is a long test to verify `FPU` operations. The test sequence is as follows:

- Check conversion of double-precision floating-point numbers to single-precision.
- Check conversion of single-precision floating-point numbers to double-precision.
- Check conversion of integer to double-precision.
- Check single-precision moves (`fmovs`).
- Check single-precision negations (`fnegs`).
- Check single-precision absolute value (`fabss`).
- Check single-precision moves (`fmovs`).
- Check that less than / greater than, equal and unordered comparison works.
- Check double-precision additions (`faddd`).
- Check double-precision subtractions (`fsubd`).
- Check double-precision multiplications (`fmuld`).
- Check double-precision divisions (`fdivd`).
- Check back-to-back `faddd`, `fmuld`, `fdivd` sequence.
- Check various floating-point trap conditions.

fnan_v

This diagnostic verifies the SNaNs and QNaNs for Meiko’s implementation of the IEEE standard. Refer to the microSPARC-IIep specification for all the details.

fneg_sp_v

This diagnostic is a test for the IEEE FP single-precision negate operation with all four rounding modes (nearest, zero, +ve infinity, and -ve infinity). The test is a loop of 4 load doubles (each load double gets two operands for FP operation), 8 Fnegs and 4 stores. The loop count is 20. The expected result and `fsr` register are stored and checked.

fp_dacc

This diagnostic is a basic test to verify data access error exception handling for floating-point instructions. The test sequence includes:

- Load floating-point single
- Load floating-point double
- Load floating-point `fsr`

The floating-point loads from an invalid page should not change the FP registers.

fp_excp

This diagnostic is a corner case test to verify floating-point exceptions that will cause a watch-dog reset. The test sequence is:

- A floating-point divide-by-zero trap occurs. The next instruction is a store floating-point register, which causes a data access exception. The floating-point controller is now in exception mode. As both exceptions occur at the same time, the microSPARC-IIep is put into error mode.

This test causes a watch-dog reset. This test will cause an endless loop in Verilog simulation.

fp_nan

This diagnostic verifies the untrapped floating-point resulting in the same and different formats (single, double, and integer).

fp_round

This diagnostic verifies the four rounding directions:

1. positive infinity
2. negative infinity
3. zero
4. nearest

When possible, the diagnostic checks large values (> Max), values near 0, and small values (< Min) for both the MPY and ALU FP units. (Getting values 0 < ABS(val) < Min is tough with the ALU, so it is skipped). The “correct” values used in this diagnostic come from SAS runs that use the 68881 math coprocessor.

- Check rounding to positive infinity for multiply and add -single-precision operation.
- Check rounding to negative infinity for multiply and add -single-precision operation.
- Check rounding to zero for multiply and add -single-precision operation.
- Check rounding to nearest for multiply and add -single-precision operation.

fp_sametime

This diagnostic tests cases when floating-point operations finish during the same cycle, when FPops finish on top of LDF ops, and combinations of the above with and without dependencies. Note that in generating these timing diagrams, it was assumed that the hold/ interlock mechanisms were “disabled” (that is, worst case failure). This ensures that these mechanisms work properly.

- Generate two FPops that complete at the same time. Assumes that `fmuld` takes four cycles and `fadds` takes two cycles to complete. There are no dependencies between the instructions.

This test is exactly the same as the preceding one except that there is a destination dependency between the instructions.

- Generate two FPops that complete at the same time along with LDFs that attempt to write the register file with destinations the same as that of the FPops. Note: this tests the interlock mechanism to prevent simultaneous write to a register file from different sources. The timing diagrams were drawn up assuming the mechanisms were disabled.

- Generate a single FPop that completes as a load comes in. There are no dependencies, so the two results should be written into the register file at the same time.

The following test case is exactly the same as the previous one except that the destination registers of the `fadds` and the `ldf` are the same. They should interlock, so two results are not written at the same time to the register file.

- Generate two FPops that finish at different times. However, there are two LDFs that occur when the FPops try to write the register file. These LDFs try to write the same destination registers that the FPops do.

- Generate two FPops with source dependency, and see if load conflict causes any problems.

- Generate a sequence of instructions that will cause the same register to appear in the following buses (assuming no interlocks): `rfile_rs1`, `rfile_rs2`, `rfile_rd`, and `rfile_ldaddr`. Actually, it does assume an interlock between the load and std.

This test causes the register file and the cache to be written to during the same cycle.

fp_stex

This diagnostic is a corner case test to verify floating-point store exceptions. The test sequence verifies the following cases:

- Store single exception
- Store double exception
- Store fsr exception
- Store floating-point queue exception. The fpu exception and memory exception at the same time

fpexc_1_v

A floating-point exception test ported from SuperSPARC-I. This program shows that SAS is correct in converting SNaN to QNaN. TI converts also, but the resulting QNaN is not correct. Only QNaN bit should be flipped to produce a QNaN; the remaining bits should remain the same. The same is true for QNaN to QNaN; the original QNaN is not maintained.

The Meiko implementation is slightly different than TI, but both are flavors of IEEE 754 standard.

fpop_dp_v

This simple diagnostic verifies the basic operation of double-precision load floating-point registers, store floating-point registers, and double-precision floating-point add. All the registers are verified in this test.

fpop_sp_v

This simple diagnostic verifies the basic operation of single-precision load floating-point registers, store floating-point registers, and single-precision floating-point add. All the registers are verified in this test.

fpop_word_v

This simple diagnostic verifies the basic operation of odd double-precision load floating-point registers, store floating-point registers, and double-precision floating-point add. All the registers are verified in this test. The odd register pair should be ignored by the design and work like the even pair. The assembler will complain about this code, and hence the test is written using the `.word` format.

fpu_regdep

This diagnostic tests a few cases of unique FP instruction operand dependencies. These include dependencies that occur between sequences of instructions that use both the FALU and FMPY. Parallel operation of the FALU and FMPY is assumed. Also, it is assumed that each unit can only operate on one instruction at a time and that double-precision divide takes much longer than double-precision add. This allows the diagnostic to test multiple dependencies.

The microSPARC-IIep FPU cannot execute FALU and FMPY in parallel.

fpu_special3

This diagnostic handles the outlined cases involving sequences of FPops and LD/ST fsr. Writes to the `%fsr` should not change the `ftt`, `qne`, and reserved fields.

fpu_togcov

This diagnostic is specially written to increase the toggle coverage of the microSPARC-IIep FPU/FPC. It performs the following tests:

- Generates a floating-point sequence error
- Executes instructions with high virtual addresses (this diagnostic initializes the `mmu/tlb`)
- Generates an underflow exception

This is a “nice” test in that it does not depend on the cycle time of the instructions (the instructions were chosen to interlock to guarantee things happen as planned).

fpx_1_v

A test ported from SuperSPARC-I to test floating-point exceptions. The intent of the test is to verify the `al` exceptions and trap modes using the floating-point single-precision and double-precision addition instruction (`fadds`, `faddd`).

fpx_2_v

A test ported from SuperSPARC-I to test floating-point exceptions. The intent of the test is to verify the `al` exception, trap modes, rounding modes. The unimplemented floating-point instructions are also tested. The following is the test sequence:

- Single-precision floating-point add (`fadds`).
- Double-precision floating-point add (`faddd`).
- Single-precision floating-point subtract (`fsubs`).
- Double-precision floating-point subtract (`fsubd`).
- Single-precision floating-point multiply (`fmuls`).
- Double-precision floating-point multiply (`fmuld`).
- Single-precision floating-point divide (`fdivs`).
- Double-precision floating-point divide (`fdivd`).
- Single-precision floating-point square root (`fsqrts`).
- Double-precision floating-point square root (`fsqrtd`).
- Convert single-precision to integer (`fstoi`).
- Convert double-precision to integer (`fdtoi`).
- Convert integer to single-precision (`fitos`).
- Convert double-precision to single-precision (`fdtos`).
- Convert integer to single-precision (`fitod`).
- Convert single-precision to double-precision (`fstod`).
- Execute the following unimplemented instructions: `faddx`, `fsubx`, `fmulx`, `fdivx`, `fsqrtx`, `fstox`, `fxtos`, `fdtox`, `fxtod`, `fitox`, `fxtoi`, `fcmpx`, `fcmpex`, `fdmulx`.

fpx_3_v

This diagnostic is a test ported from SuperSPARC-I to test floating-point exceptions. The intent of the test is to verify exceptions when the floating-point queue is empty to when the floating-point queue is full (SuperSPARC-I has a queue depth greater than 1; microSPARC-IIep queue is one deep). The following is the test sequence:

- `fstoi` resulting in overflow inexact without load register dependency
- `fdtoi` resulting in overflow inexact without load register dependency
- `fdivs` resulting in underflow inexact without load register dependency
- `fdivs` resulting in underflow inexact without load register dependency
- `fstoi` resulting in overflow inexact with load register dependency
- `fdtoi` resulting in overflow inexact with load register dependency
- `fdivs` resulting in underflow inexact with load register dependency

- `fstoi` resulting in exception followed by `fadds` without register dependency
- `fdtoi` resulting in exception followed by `fadds` without register dependency
- `fdivs` resulting in divide-by-zero exception followed by `fmuls` without register dependency
- `fdtoi` resulting in overflow inexact followed by `fadds` without register dependency
- `fstoi` resulting in exception followed by `fadds` with register dependency
- `fdtoi` resulting in exception followed by `fadds` with register dependency
- `fdivs` resulting in divide-by-zero exception followed by `fmuls` with register dependency
- `fdtoi` resulting in overflow inexact followed by `fadds` with register dependency
- Plus additional cases with floating point queue empty and full

fpx_4_v

This diagnostic is a test ported from SuperSPARC-I to test floating-point operand corner case. The intent of the test is to verify the corner cases related with zero, positive infinity, negative infinity, alpha, negative alpha, positive SNaN, negative SNaN, positive QNaN, and negative SNaN. The following is the test sequence:

- Exhaustive case with single-precision floating-point addition (`fadds`)
- Exhaustive case with single-precision floating-point subtract (`fsubs`)
- Exhaustive case with single-precision floating-point multiply (`fmuls`)
- Exhaustive case with single-precision floating-point divide (`fdivs`)
- Exhaustive case with double-precision floating-point addition (`fadds`)
- Exhaustive case with double-precision floating-point subtract (`fsubs`)
- Exhaustive case with double-precision floating-point multiply (`fmuls`)
- Exhaustive case with double-precision floating-point divide (`fdivs`)

fpx_5_v

This diagnostic is a test ported from SuperSPARC-I to test floating-point exceptions.The intent of the test is to verify exceptions when the floating-point queue is empty to floating-point queue is full (SuperSPARC-I has a queue depth greater than 1; microSPARC-IIep queue is one deep). The following is the test sequence:

- `fstoi` resulting in overflow inexact followed by load `fsr`
- `fdtoi` resulting in overflow inexact followed by store `fsr`
- `fdivs` resulting in underflow inexact followed by store `fsr`

- `fdivs` resulting in underflow inexact followed by load `fsr`
- `fstoi` resulting in exception followed by store double floating-point queue
- `fdtoi` resulting in exception followed by store double floating-point queue
- `fdivs` resulting in divide by zero exception followed by store double floating-point queue
- `fdtoi` resulting in overflow inexact followed by store double floating-point queue
- Plus additional cases with floating-point queue empty and full

fpx_6_v

This diagnostic is a test ported from SuperSPARC-I to test floating-point exceptions. The intent of the test is to verify the read pointers that are affected during exception and dependency, floating-point queue is full or has more than one instruction which did not start. Probably interesting test case for SuperSPARC-I.

fpx_7_v

This diagnostic is a test ported from SuperSPARC-I to test floating-point exceptions. The intent of the test is to verify the exceptions when the subsequent instruction is a load/store that would hit or miss in the cache and the load address may be misaligned.

fpx_8_v

This diagnostic is a test ported from SuperSPARC-I to test floating-point exceptions. This test is a collection of test sequences that detected bugs during the initial development of floating-point exception logic. The following is the test sequence:

- Unimplemented `fpops` and `udiv`
- `fqueue` and floating-point exception
- Unimplemented fpop
- Floating-point exception and load misaligned with no dependency
- Floating-point exception and load misaligned with dependency
- Floating-point add followed by floating-point load. The destination register is same for both the instructions
- Floating-point add followed by floating-point load with misaligned address. The destination register is same for both the instructions
- Plus additional cases with queue full and empty

fpx_8_v_sp.s

This diagnostic is a modified version of `fpx_8_v.s`. One of the test cases to test precise trap on `stdfq` instruction is commented out in `fpx_8_v.s`. This diagnostic has its own trap handler.

fpx_window_v

This diagnostic is a test ported from SuperSPARC-I to test the floating-point exception handler. The exception is caused inside the trap handler routine and once causing window overflow. It is more of a test to verify the Verilog-SAS interface.

fp_int_traps.s

This diagnostic tests for floating-point traps along with integer traps. The following cases are covered.

- Floating-point exception and data access exception
- Simultaneous floating-point exception and data access exception
- Simultaneous floating-point exception and memory address misaligned traps
- Floating-point exceptions and instruction access exceptions
- Floating-point exception and data access error traps

This diagnostic tests for timing-dependent floating-point traps. Enabling software check does not guarantee traps to occur at the right time.

fregdep_1_v

This diagnostic is an exhaustive test to verify floating-point load register interlock dependencies. The interlocking instruction sequence is:

- Load single floating-point register followed by `ldf`, `lddf`, `stf`, `stdf`, `fadds`, `faddd`, `fmovs`, `fabss`, `fnegs`, `fstoi`, `fstod`, `fdtos`, `fsmuld`, `fcmps`, `fcmpd`, `fba`, `ldfsr`, `stfsr`, `umul`.
- Load double floating-point register followed by `ldf`, `lddf`, `stf`, `stdf`, `fadds`, `faddd`, `fmovs`, `fabss`, `fnegs`, `fstoi`, `fstod`, `fdtos`, `fsmuld`, `fcmps`, `fcmpd`, `fba`, `ldfsr`, `stfsr`, `umul`.

fregdep_2_v

This diagnostic verifies FP register dependencies. The test loads single- and double-precision operands from memory into registers. The first part of the test involves interlock operation between single-precision operands register with double-precision operands registers. The second part of the test involves double-precision operands registers with double-precision operands.

The interlock is between the source_one register or source_two register of an instruction and the destination register of a previous instruction.

The test inserts one to four single-precision instructions between the interlocking instructions.

fregdep_3_v

This diagnostic verifies FP register dependencies. The test loads single- and double-precision operands from memory into registers. The test sequence is:

- Load single floating-point register followed by load double floating-point register with dependency.
- Load single floating-point register followed by store single floating-point register with dependency.
- Load two single floating-point registers followed by store double floating-point register with dependency.
- Load double floating-point register followed by store two single floating-point registers with dependency.
- Load double floating-point register followed by store double floating-point register with dependency.
- Load single/double followed by floating-point instruction (`fadds`, `fsubs`, `faddd`, `fsubd`), followed by store single, store double in various combinations.
- Plus back-to-back floating-point instructions (`fsubd`, `fabss`, `fmuld`, `fsubd`, `fmovs`, `fnegs`, `fmuls`, `faddd`, `fdtoi`, `fstoi`, `fitos`, `fadds`, `fitod`, `fdtoi`, `fstod`, `fdtos`).

fround_cvt_v

This diagnostic verifies the floating-point rounding operations. Using various operands the test will verify the following instructions:

- `fdtoi`, `fdtos`, `fstoi`, `fstod`, `fitos`, and `fitod`

All the four rounding modes are checked.

This test is a superset of the `fround_cvt_dp_v`, `fround_cvtf_sp_v`, `fround_cvti_v` tests.

fround_cvtf_dp_v

This diagnostic verifies the floating-point rounding operations. Using various operands, the test verifies the following instructions:

- `fdtoi`, `fdtos`, `fstoi`, `fstod`

All four rounding modes are checked.

This test is a subset of the `fround_cvt_v` test.

fround_cvtf_sp_v

This diagnostic verifies the floating-point rounding operations. Using various operands, the test verifies only the `fstoi` instruction. All four rounding modes are checked.

This test is a subset of the `fround_cvt_v` test.

fround_cvti_v

This diagnostic verifies the floating-point rounding operations. Using various operands, the test verifies the following instructions:

- `fitos`, `fitod`

All four rounding modes are checked.The operands are also negated. This test is a subset of the `fround_cvt_v` test.

fround_dp_v

This diagnostic verifies the floating-point rounding operations for double-precision instructions. Using various operands, the test verifies the following instructions:

- `faddd`, `fsubd`

All four rounding modes are checked.The operands are also negated.

fround_sp_v

This diagnostic verifies the floating-point rounding operations for single-precision instructions. Using various operands, the test verifies the following instructions:

- `fadds`, `fsubs`

All four rounding modes are checked. The operands are also negated.

fround_sq_div_v

This diagnostic verifies the floating-point rounding operations for single- and double-precision instructions. Using various operands, the test verifies the following instructions:

- `fmuls`, `fdivs`, `fsqrts`, `fmuld`, `fdivd`, `fsqrtd`

All four rounding modes are checked. The operands are also negated.

fround_stod_v

This diagnostic verifies floating-point rounding operations for single-precision QNaN operand to double-precision QNaN append conversion (`fstod`). The IEEE implementation will convert it to equivalent double-precision representation of QNaN. This diagnostic was ported from SuperSPARC-I. The TI floating-point will give a fixed QNaN result.

fsi2d_v

This diagnostic tests IEEE FP single-precision to double-precision operand format and integer to double-precision format. with all four rounding modes (nearest, zero, +ve infinity, and -ve infinity). The test is a loop of 2 load doubles (each load double gets two operands for FP operation), 4 Fstod and 4 Fitod and 8 stores. The loop count is 40. The expected result and `fsr` register are stored and checked.

fsi2is_v

This diagnostic tests IEEE FP single-precision to integer operand format and integer operand to FP single-precision operand format with all four rounding modes (nearest, zero, +ve infinity, and -ve infinity). The test is a loop of 4 load doubles (each load double gets two operands for FP operation), 8 Fstoi and 8 Fitos, and 8 stores. The loop count is 40. The expected result and `fsr` register are stored and checked.

fsqrt_dp_v

This diagnostic tests IEEE FP double-precision square root operation with all four rounding modes (nearest, zero, +ve infinity, and -ve infinity). The test is a loop of 4 load doubles (each load double gets two operands for FP operation), 4 Fsqrtd, and 4 stores. The loop count is 88. The expected result and `fsr` register are stored and checked.

fsqrt_sp_v

This diagnostic tests IEEE FP single-precision negate operation with all four rounding modes (nearest, zero, +ve infinity, and -ve infinity). The test is a loop of 4 load doubles (each load double gets two operands for FP operation), 8 Fsqrts, and 4 stores. The loop count is 80. The expected result and `fsr` register are stored and checked.

fsrA

This diagnostic exhaustively verifies load and store of floating-point status register (`fsr`) with interspersed floating-point operations. The test sequence is:

- Load the FPC pipe with compares, and verify all the condition code states and transitions.
- Load the FPC pipe with compares, and verify all the condition code states and transitions with traps.
- Load the FPC pipe with compares, and verify all the condition code states and transitions with traps taken by `FBfcc` instructions.
- Load the FPC pipe with compares, and verify all the condition code states and transitions with traps taken by `ldfsr` instructions, followed by series of load `fsr` register.

fsrtest

This basic diagnostic verifies the floating-point status register. The test sequence is:

- Verify `fsr` after reset.
- Write each `fsr` bit with 1 except the `cexc` field.(writing `cexc` field to 1 will result in a trap requiring a special trap handler).
- Write each `fsr` bit with 0.
- No other floating-point instruction in progress while reading the `fsr`.
- No other floating-point instruction in progress while writing the `fsr`.
- `fsr` write followed by `fsr` read verifying the new value.

fsub_dp_v

A test for IEEE FP double-precision subtract operation with all four rounding modes (nearest, zero, +ve infinity, and -ve infinity). The test is a loop of 8 load doubles (each load double gets two operands for FP operation), 4 Fsubd, and 4 stores. The loop count is 300. The expected result and `fsr` register are stored and checked.

fsub_sp_v

This diagnostic tests IEEE FP single-precision subtract operation with all 4 rounding modes (nearest, zero, +ve infinity and -ve infinity). The test is a loop of 4 load doubles (each load double gets two operands for FP operation), 4 Fsubs and 2 stores. The loop count is 300. The expected result and `fsr` register are stored and checked.

funimpl_v

This diagnostic is a test ported from SuperSPARC-I to verify unimplemented floating-point instructions. The test verifies the odd register pairing of floating-point registers for double-precision operations. The test sequence is:

- Use `%f1` instead of `%f0` for `faddd`

The assembler will signal error if the odd registers are used. To get around the problem, the instructions have been forced by using the `.word` command option.

linpack_1

This diagnostic is a small test to verify the inner loop of double-precision Linpack benchmark. The test executes the loop twice. The test was generated with version 1.3 of the Fortran compiler.

linpack_bench

This diagnostic is a large test to verify the inner loop of double-precision Linpack benchmark. The test executes the loop six times. This test has some start-up code in the beginning. The test was generated with version 1.4 of the Fortran compiler.

linpack_bench_big

This diagnostic is a large test to estimate the Linpack performance of microSPARC-II by executing the loop 100 times. The test is run with the ITBR mode enabled, fast page-mode, and memory refresh set at every 512 clocks.

monadic

This diagnostic examines monadic instruction chaining. Pseudo-monadic instructions (`fabss`, `fnegs`) and monadic instructions (`fmovs`, `fdtos`) do not use the `rs1` field of the instruction word. The assembler sets this field to 0. No chaining should be done on `%f0` for these monadic instructions. This diagnostic sets up a variety of cases where chaining would occur if the `rs1` field were decoded; the user should check that incorrect holds are not generated. This diagnostic also examines the dyadic `fcmp` instructions. `fcmp` does not decode `rd`, which the assembler sets to 0. Similarly, there should not be any chaining or holding on `%f0` for these instructions.

Not very interesting for microSPARC-IIep since it does not support chaining.

nabcRFf

This diagnostic exhaustively verifies all the software-visible 32 floating-point registers. The test sequence is as follows:

- Verify all registers with 0’s.
- Verify all registers with alternate 1’s and 0’s.
- Verify all registers with alternate 0’s and 1’s.

This megacell test will require some modification so that it can be used as part of the production vectors.

seq_err

This diagnostic is a corner case test to verify the sequence error floating-point trap. The test verifies all six types of nonstore floating-point instructions (`ldf`, `lddf`, `ldfsr`, `fpop`, `fcmp`, `FBfcc`) that can sequence errors.

test_fbcc

This diagnostic exhaustively verifies all the Branch on floating-point condition code instructions. The test sequence is less than condition, greater than condition, equal, and unordered condition.

tst_fpu_basic

This small diagnostic verifies basic floating-point operations. The following is the test sequence:

- Load single-precision operands.

- Load double-precision operands.
- Perform single-precision multiply (`fmuls`).
- Perform single-precision addition (`fadds`).
- Perform single-precision subtract (`fsubs`).
- Perform single-precision divide (`fdivs`).
- Perform single-precision less than compare test.
- Perform single-precision equal compare test.
- Perform single-precision greater than compare test.
- Perform single-precision double-precision another equal compare test.
- Perform floating-point integer to single conversion (`fitos`).
- Perform floating-point single to integer conversion (`fstoi`).
- Perform floating-point single to double conversion (`fstod`).
- Perform double-precision multiply (`fmuld`).
- Perform double-precision addition (`faddd`).
- Perform double-precision subtract (`fsubd`).
- Perform double-precision divide (`fdivd`).
- Perform double-precision less than compare test.
- Perform double-precision equal compare test.
- Perform double-precision greater than compare test.
- Perform double-precision another equal compare test.
- Perform floating-point integer-to-double conversion (`fitod`).
- Perform floating-point double-to-single conversion (`fdtos`).

CHAPTER 5

Memory Management Unit Tests

This chapter describes the diagnostic tests for the memory management unit.

asi_access.s

This diagnostic verifies illegal address during ASI 0x4 operations. Swift only cares about bits [12:8] of the virtual address and ignores the other bits.This diagnostic also tests for Dcache RAM access (ASi 0xF) with different sizes

br80pages4k_v

This basic diagnostic causes a branch across a 4-Kbyte page boundary. The test covers 80 pages.

datapath

This diagnostic verifies load and stores, separated by different number of cycles, different data types, and atomic operations. From different address regions (4k boundaries) do loads, isolated from each other by several nops, then stores to the same locations. It also does a few different operand size loads and stores, followed by `swap` and `ldstub` instructions. The test starts with the five nops delay between various load/store instructions. This delay is changed to four, three, two, and one during the course of the test.

The test covers the `std`, `stb`, `sth`, `ldub`, `lduh`, `ldstub`, `ld`, `ldd`, and `swap` instructions.

This test uses the `crueltt.s` or follows the `non-asm2ver` style of coding. The same test using `asm2ver` style of coding is called `datapath_virt`.

datapath_virt

This basic diagnostic verifies load and stores, separated by different number of cycles, different data types, and atomic operations. From different address regions (4k boundaries) do loads, isolated from each other by several nops, then stores to the same locations. It also does a few different operand size loads and stores, followed by `swap` and `ldstub` instructions. The test starts with the five nops delay between various load/store instructions. This delay is changed to four, three, two, and one during the course of the test.

The test covers the `std`, `stb`, `sth`, `ldub`, `lduh`, `ldstub`, `ld`, `ldd`, and `swap` instructions.

This test uses the `traps.s` or asm2ver style of coding. The same test using non-asm2ver style of coding is called `datapath`.

flushTlb_v

This diagnostic exhaustively verifies the TLB flush operation. The flush operation allows software invalidation of TLB entries. The basic test was ported from SuperSPARC-I. The flush operation is done using store alternate ASI 0x3. Both the page and entire flushes are verified.

iom_b979

This diagnostic is a directed bug test that verifies the IOMMU is only enabled after the MMU is turned on. If the IOMMU is turned on but MMU is disabled, then this diagnostic will cause a hang.

iom_fls_tlb

This basic diagnostic verifies IOMMU flush by using flush address register through control space. The test sequence is as follows:

- Set `TRCR.tc`, set up `tlb` entry 2-63 with `iopte` (`va = pa`)
- Verify IOMMU flush using flush address register

iom_fls_tlba

This basic diagnostic verifies IOMMU flush by using flush all `tlb` entries register through control space. The test sequence is as follows:

- Set `TRCR.tc`, set up `tlb` entry 2-63 with `iopte` (va = pa, ctx= 0x0).
- Verify IOMMU flush using flush all `tlb` entries register.

iom_fls_tlbcxt

Subset of `iom_fls_tlb`.

ld68pages4k_v

This basic diagnostic verifies load from various 4k pages. A total of 68 loads occurs from 68 different pages. Each load would miss in the `tlb`.

mfar_v

This basic diagnostic verifies synchronous fault address and status register (SFAR and SFSR) read and writes. The following is the test sequence:

- Check that `sfsr.fav` is 0 on instruction access exception.
- Check that `sfsr.fav` is 1 on data access exceptions.
- Series of write and read (for checking) with different data patterns (alternate 1's and 0's, 0's and 1's, all 1's, all 0's) to `sfar` using `ASI` accesses.

mfsrToErr_h

This diagnostic is a corner case test to verify unassigned (illegal) ASI accesses. The first part of the test executes the instruction a store double using ASI 0x2 and will cause a trap. The second part of the test executes an instruction swap ASI using ASI 0x2, which will cause a trap.

This test is basically the same as `mfsrToErr_v`. This test will work in the reset mode environment.

mfsrToErr_v

This diagnostic is a corner case test to verify SFSR after unassigned (illegal) ASI accesses. The first part of the test executes the instruction a store double using ASI 0x2, which will cause a trap. The second part of the test executes an instruction `swap` ASI using ASI 0x2, which will cause a trap. The SFSR is checked at the end. Also, the trapping address is checked.

This test is basically the same as `mfsrToErr_h`. This test has self-checking associated with it but does not run the reset environment.

mmuAsi

This basic diagnostic verifies exceptions generated by illegal size for ASI 0x6 TLB access. The test generates data access errors for byte, halfword, and double word ASI access of TLB. Only the word access is a valid one. If the address of the access is misaligned, then the memory address not aligned trap should be taken. (This is a higher priority than the data access error trap). The test sequence is:

- Data access error due to illegal size of double word.
- Verify the trap type.
- Data access error due to illegal size of halfword.
- Verify the trap type.
- Data access error due to illegal size of byte.
- Verify the trap type.
- Misaligned address for a double-word access.
- Verify the trap type based on priority.
- Misaligned address for a halfword access.
- Verify the trap type based on priority.
- Byte access cannot have misaligned address, so generate a data access error trap.
- Verify the trap type based on priority.

mmuContext

This diagnostic exhaustively verifies the use of context register and related logic (TLB tag/comparison, CTXR, and CTPR). The test sequence starts with TLB entry 1 and works all the way to 32.

mmuFsrCombo

This diagnostic exhaustively verifies the fault status register and the action taken when pairs of MMU faults occur without the initial fault being acknowledged by a read of the FSR. This test checks each possible pair for a variety of instructions. The three fault types of interest are data access exception, instruction access exception, and translation error.

mmuInvRsvd

This diagnostic exhaustively verifies that the SFSR and SFAR contain correct values and that the correct trap is taken in various accesses to Invalid (Entry type equal to 0) and Reserved (Entry type equal to 3) page tables. The test also verifies the special case of a level-3 PTP.

mmuParity

This diagnostic exhaustively verifies the checking of parity errors. Even or odd parity can be set. The following is the sequence:

- Instruction memory access — Set PE, FT=5, AT in SFSR, cause Instruction access Error Trap (D stage + one).
- IU, FPU read memory access — Set PE, ERR, CP, TYPE in MFSR, save PA in MFAR, cause Level 15 interrupt.
- IU, FPU write byte, halfword memory access (read modify write) — Set PE, ERR, CP, TYPE in MFSR, save PA in MFAR, cause Level 15 interrupt.
- Tablewalk on instruction memory access (translation error) — Set PE, FT=4, AT in SFSR, cause Instruction access Error Trap (D stage).
- Tablewalk on data memory access (translation error) — Set PE, FT=4, save `iu_dva` in SFAR, cause Data access Error Trap (R stage).
- IO DMA read memory access — Return SBus error acknowledge, set PE, ERR in MFSSR, save PA in MFAR, cause Level 15 interrupt.
- IO DMA write byte, halfword memory access (read modify write) — Return SBus error acknowledge, set PE, ERR in MFSSR, save PA in MFAR, cause Level 15 interrupt.
- Tablewalk on IO DMA memory access — Return SBus error acknowledge, set PE, ERR in MFSSR, save PA in MFAR, cause Level 15 interrupt.

This test is automatically generated by a script.

mmuSoftWalk

This diagnostic exhaustively verifies the MMU software tablewalk operation. This test would enable the instruction access MMU miss and data access MMU miss traps for instruction and data tablewalks to be done by software. This is a debug feature. The test sequence is:

- Set up for software tablewalk — Load PTE mappings (zero to 256K) into TLB RAM slot 31, load TAG mappings into TLB CAM slot 31, disable the `tlb` counter, enable software tablewalk.
- `ifetch` from invalid page.

- `ifetch` from privileged page.
- `ifetch` from protected page.
- `ifetch` from translation error page.
- Load from invalid page.
- Store to invalid page.
- Load from privileged page.
- Store to privileged page.
- Load from protected page.
- Store to protected page.
- Load from translation error page.
- Store to translation error page.

mmu_100_5

This basic diagnostic verifies TLB replacement control register bit 21–24 (memory speed and Sbus divide clock). Those bits are set from external (or by run mode).

- Set run mode, then read back expected bit setting

See also `mmu_125_3`, `mmu_70_3`, `mmu_85_4`.

mmu_D_accat

This diagnostic tests ACC/AT table on behalf of MMU. It does not check SFSR (see `mmu_D_accat_chk`), no error checking.

It sets up cacheable page with acc 0–7, sets up `tlb` entry to make sure each access will be MMU hit, do AT access from 0–7 with `acc` 0–7.

This is a subset of `mmu_D_accat_chk`, short diagnostic.

mmu_D_accat_chk

This diagnostic tests ACC/AT table on behalf of MMU and Dcache. It checks SFSR on expected protection/privilege trap, verifies expected results and Dtag on most no-fault cases.

It sets up cacheable page with `acc` 0–7, R bit and M bit, sets up TLB entry to make sure each access will be MMU hit, do AT access from 0–7 with acc 0–7.

mmu_I_accat

This diagnostic tests ACC / AT table (page 257 of SPARC architecture reference) on behalf of MMU. It does not check SFSR (see `mmu_I_accat_chk`), no error checking.

It sets up cacheable page with `acc` 0–7, sets up TLB entry to make sure each access will be MMU hit, do AT 2–3 access with `acc` 0–7.

This is a subset of `mmu_I_accat_chk`, short diagnostic.

mmu_I_accat_chk

This diagnostic tests ACC / AT table on behalf of MMU and `Icache`. It checks SFSR on expected protection / privilege trap, verifies expected results and Itag on no-fault cases.

It sets up cacheable page with `acc` 0–7, R bit and M bit, sets up TLB entry to make sure each access will be MMU hit, do AT 2-3 access with `acc` 0–7.

mmu_cregs2

This basic diagnostic verifies reads and writes to MMU control registers. The test sequence is:

- Write unique value to each of the registers (PCR, CTPR, CXR, and TRCR).The write (store asi) is followed by a series of nops so that there is no other resource requirement.
- Read back each of the registers and check if the value is maintained (the reserved field is masked off).

mmu_cspace_h

This basic diagnostic verifies reads and writes to MMU control space registers. The test sequence performs a write followed by a read, followed by a write of alternate 1's and 0's, followed by a read, followed by a write of alternate 0's and 1's, followed by a read, of the following registers:

- Processor control register (`pcr`)
- Context table pointer register (`ctpr`)
- Context register (`cxr`)
- Synchronous fault status register (`sfsr`)
- Synchronous fault address register (`sfar`)
- TLB replacement control register (`trcr`)
- Instruction translation buffer register PTE (`itbrp`)
- Instruction translation buffer register tag (`itbrt`)

- IO control register (iocr)
- IO base address register (ioba)
- Asynchronous fault status register (afsr)
- Asynchronous fault address register (afar)
- Memory fault status register (mfsr)
- Memory fault address register (mfar)
- MID register (mid)
- SBus slot configuration register 0 (sscr0)
- SBus slot configuration register 1 (sscr1)
- SBus slot configuration register 2 (sscr2)
- SBus slot configuration register 3 (sscr3)

This is amodification of the original mmu_cspace to run under the reset environment.

mmu_cxtctp

This basic diagnostic verifies some data patterns for context table pointer register (ctpr) and context register (ctr). The test sequence is:

- Map a data page to two different contexts.
- Check initial ctpr.
- Check initial cxr.
- Load context table ptp.
- Store context table ptp.
- Switch to user mode.
- Do a load, followed by store.
- Reconfigure MMU and return through other context number.
- Check new ctpr.
- Check new cxr.
- Switch to user mode.
- Do a load, followed by store.

mmu_cxtctp_3

This diagnostic is a toggle test to improve the toggle coverage for the ctpr and cxr register bits. Test pattern includes alternate 1’s and 0’s followed by alternate 0’s and 1’s.

mmu_cxtctp_4

This diagnostic is a toggle test to improve the toggle coverage for the `ctpr` register bits. Test pattern includes a 0x00ffffff followed by 0x00111111.

mmu_dmiss

This basic diagnostic verifies TLB miss on data reference. The test sequence is as follows:

- Initialize context table, level1 table, level2 table and level3 table with `ptp`'s and `pte` entries for pages 0, 1, 2, and 3 in the text segment, pages 4, 5, and 6 in the data segment.
- Initialize `ctx` and `ctpr` to 0 and start of context table.
- Enable the MMU for translation.
- Execute some nops on page 0, load from page 2, branches to page 3.

mmu_fls_type

Walking 1 on index 1, 2, and 3 of VFPA to verify different flush type, ACC <= 5.

- Set TRCR.tc, flush all `tlb` entries.
- For each test case, `tlb` is set up as follows:
  - Entry 2–21: Walking 1 on VA[31:12], and VA is equal to PA, default `pte` is level 3 `pte` with following attributes C=0, M=0, R=0, ACC=2.
  - Entry 22–23: `iopte` with following attributes C=0, M=0, R=0.
  - Entry 24: level 0 `ptp`.
  - Entry 25: level 1 `ptp`.
  - Entry 26: level 2 `ptp`.
  - Case 1: Page flush.
  - Set up `tlb` as mentioned above.
- Do two no-match (`vfpa` does not match any entry in `tlb`) page flush, and one match (`vfpa` matches one of `iopte` entry) page flush.
- Check PTP entry is valid. Do matched page flush on PTP entry, verify PTP entry still valid.
- Do page flush entry by entry, check current entry is flushed and next entry still is valid.
- Do a match page flush (matched with second `iopte`). Verify both `iopte`s are valid.
- Set up `tlb` entry as above with context number (0xa5, current context number 0x5a).
- Case 2: do match-page flush but context mismatch, verify entry is valid.

- Case 3: do match-segment flush but context mismatch, verify `pte/iopte` entry is valid, `ptp` entry is flushed.
- Case 4: do match-region flush but context mismatch, verify `pte/iopte` entry is valid, `ptp` entry is flushed.
- Case 5: do context flush but context mismatch, verify `pte/iopte` entry is valid, `ptp` entry is flushed.
  - Set up `tlb` entry with current context number.
- Case 6: segment flush with context match.
  - Do a match segment flush on `iopte` entry, an index 1 and 2 match flush (should flush entries with matched va[31:18]) and an irrelevant seg flush.
  - Verify entire with matched va[17:12] have been flushed, `iopte` entries are valid.
  - Do segment flush entry by entry with matched va[31:18], check next entry is valid.
  - Set up TLB entry with current context number.
- Case 7: region flush with context match.
  - Do region flush to invalidate `tlb` entry with matched va[31:24], do a no-match region flush.
  - Verify `iopte` entry is valid.
  - Do region flush entry by entry with matched va[31:24], check next entry is valid.
  - Set up `tlb` entry with current context.
- Case 8: context flush with context match.
  - Do context flush, verify `iopte` entry is valid, others are invalidated.

mmu_fls_type_S

Walking 1 on index 1, 2, and 3 of VFPA to verify different flush type, ACC >=6 (see table 15 of Swift specification rev 1.2).

- Set TRCR.tc, flush all `tlb` entry.
- For each test case, `tlb` is set up as follows:
  - Entry 2–21: Walking 1 on VA[31:12], and VA is equal to PA, default `pte` is level 3 pte with following attributes C=0, M=0, R=0.
  - Entry 22–23: `iopte` with following attributes C=0, M=0, R=0.
  - Entry 24: level 0 `ptp`.
  - Entry 25: level 1 `ptp`.
  - Entry 26: level 2 `ptp`.
  - Set up TLB as mentioned above.
  - Case 1: context flush with context match.

  - Do context flush, verify `iopte` entry is valid, others are invalidated.
  - Set up `tlb` entry with CTX = 0xa4, ACC = 6.
- Case 2: page flush with context match.
  - Do two no-match (`vfpa` does not match any entry in `tlb`) page flush, and one match (`vfpa` matches one of `iopte` entry) page flush.
  - Check `ptp` entry is valid.
  - Do page flush entry by entry, check current entry is flushed and next entry still is valid.
  - Do a match page flush (matched with second `iopte`). Verify both `iopte`s are valid.
  - Set up `tlb` entry with CTX = 0x55, ACC = 2.
- Case 3: page flush with context mismatch.
  - Do page flush entry by entry, check entry is not flushed.
- Case 4: context flush with context mismatch.
  - Verify all entries but `iopte` entry are invalidated.
  - Set up `tlb` entry with CTX = 0x55, ACC = 7.
- Case 5: page flush with context mismatch.
  - Repeat case 2.
  - Set up `tlb` entry with CTX = 0xaa, ACC = 6.
- Case 6: segment flush with context mismatch.
  - Repeat case 6 of `mmu_fls_type`, verify `ptp` is invalidated.
  - Set up `tlb` entry with CTX = 0x99, ACC = 7.
- Case 7: context flush with context mismatch.
  - Verify all entries but `iopte` entry are invalidated.
  - Set up `tlb` entry with CTX = 0x99, ACC = 7.
- Case 8: region flush with context mismatch.
  - Repeat case 7 of `mmu_fls_type`, verify `ptp` is invalidated.
  - Set up `tlb` entry with CTX = 0xa4, ACC = 7.
- Case 9: segment flush with context match.
  - Repeat case 6.
  - Set up `tlb` entry with CTX = 0xa4, ACC = 6.
- Case 10: region flush with context mismatch.
  - Repeat case 8.

mmu_fls_type_v

`mmu_fls_type` run in ptp2 mode (set `TRCR.ptp2`), the difference is level-2 `ptp` in entry 26 is a `vptp`. See `mmu_fls_type_v` for details.

mmu_fls_type_S_v

`mmu_fls_type_S` run in ptp2 mode (set `TRCR.ptp2`), the difference is level-2 `ptp` in entry 26 is a `vptp`. See `mmu_fls_type_S` for details.

mmu_ihit

This basic diagnostic verifies TLB hit on an instruction reference. The test sequence is as follows:

- Initialize the `tlb` with PTE entries for pages 0, 1, and 3 for text, pages 3, 4, and 5 for data.
- Initialize `ctx` and `ctpr` to 0 and start of context table.
- Enable the MMU for translation.
- Execute some nops on page 0, branches to page 1, should see `tlb_hit` when going to page 1.
- Execute some nops on page 1, branches to page 2, should see `tlb_hit` when going to page 2.

mmu_imiss

This basic diagnostic verifies TLB miss on an instruction reference. The test sequence is as follows:

- Initialize context table, level1 table, level2 table, and level3 table with `ptp`'s and `pte` entries for pages 0, 1, and 2 in the text segment.
- Initialize `ctx` and `ctpr` to 0 and start of context table.
- Enable the mmu for translation, `itbr` is disabled.
- Execute some nops on page 0 and then branches to page 1, followed by branches to page 2.

mmu_inv

This basic diagnostic verifies SFSR and SFAR when an invalid address error is generated on level-3 access (Entry type is zero). The test sequence is:

- Do a load on level 3 (cause a data access trap), check the SFAR and SFSR.

- Do a store on level 3 (cause a data access trap), check the SFAR and SFSR.
- Do an instruction fetch on level 3 (cause a instruction access trap), check the SFSR.
- Do a store on level 2 (cause a data access trap), check the SFAR and SFSR.
- Do a store on level 1 (cause a data access trap), check the SFAR and SFSR.

This is the basic test template. This test is identical to `mmu_s_inv_check`. The other test in this category is `mmu_u_inv_check`. This test is run in supervisor mode.

mmu_protpriv

This diagnostic exhaustively verifies all illegal accesses, protection violations in user mode. The test sequence is as follows:

- Write to user read-only page (cause data access trap), check for access resulted in a trap, trap to change to supervisor, check SFAR has the correct value, check SFSR has the correct value, and go to the next test.
- Execute from user read-only page (cause instruction access trap), check for access resulted in a trap, trap to change to supervisor, check SFSR has the correct value, and go to the next test.
- Execute from user read / write-only page (cause instruction access trap), check for access resulted in a trap, trap to change to supervisor, check SFSR has the correct value, and go to the next test.
- Write to user read / execute-only page (cause data access trap), check for access resulted in a trap, trap to change to supervisor, check SFAR has the correct value, check SFSR has the correct value, and go to the next test.
- Read from user execute-only page (cause data access trap), check for access resulted in a trap, trap to change to supervisor, check SFAR has the correct value, check SFSR has the correct value, and go to the next test.
- Write to user execute-only page (cause data access trap), check for access resulted in a trap, trap to change to supervisor, check SFAR has the correct value, check SFSR has the correct value, and go to the next test.
- Write to supervisor read / execute only page (cause data access trap), check for access resulted in a trap, check SFAR has the correct value, check SFSR has the correct value, and go to the next test.
- Execute from supervisor read / write-only page (cause instruction access trap), check for access resulted in a trap, trap to change to supervisor, check SFSR has the correct value, and go to the next test.
- Read from supervisor read / execute-only page (cause data access trap), check for access resulted in a trap, trap to change to supervisor, check SFAR has the correct value, check SFSR has the correct value, and go to the next test.

- Write to supervisor read / execute-only page (cause data access trap), check for access resulted in a trap, check SFAR has the correct value, check SFSR has the correct value, and go to the next test.
- Execute from supervisor read / execute-only page (cause instruction access trap), check for access resulted in a trap, trap to change to supervisor, check SFSR has the correct value, and go to the next test.
- Write (supervisor mode) to supervisor read / execute-only page (cause data access trap), check for access resulted in a trap, check SFAR has the correct value, check SFSR has the correct value, and go to the next test.
- Read from supervisor read / write / execute-only page (cause data access trap), check for access resulted in a trap, trap to change to supervisor, check SFAR has the correct value, check SFSR has the correct value, and go to the next test.
- Write to supervisor read / write / execute-only page (cause data access trap), check for access resulted in a trap, check SFAR has the correct value, check SFSR has the correct value, and go to the next test.
- Execute from supervisor read / write-only page (cause instruction access trap), check for access resulted in a trap, trap to change to supervisor, check SFSR has the correct value, and go to the next test.

This is the basic test template. This test is identical to `mmu_u_protpriv_check`. The other test in this category is `mmu_s_protpriv_check`. This test is run in user mode.

mmu_ptp2

This basic diagnostic verifies virtually tagged level 2 ptp. The test sequence is:

- Set TRCR.wp = 3, only `tlb` entry 0-3 can be used.
- Set TRCR.pl to lock `ptp` entry.
- Do Icache fill on level-3 page, check `tlb` entry 0-2, and level-2 `ptp` is virtually tagged.
- Change context number to 2.
- Do Icache fill on level-3 page, check `tlb` entry 2.
- Jump to next page, check `tlb` entry 2 unchanged, a virtual `ptp` hit is expected (ctx = 2, va[31:18] unchanged).
- Repeat above xaction.

mmu_s_hit_check

This basic diagnostic checks MMU translations at all levels for all load, store, and `ifetch` operations. The test sequence is as follows:

- level1_s and level1_m are mapped to regions.
- level2_s and level2_m are mapped to segments.
- level3_s and level3_m are mapped to pages.

- Load byte from level1_s, level2_s, and level3_s.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- check data; check pte, flush tlb; reset pte.

- Load halfword from level1_s, level2_s, and level3_s.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check pte, flush tlb; reset pte.

- Load word from level1_s, level2_s, and level3_s.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check pte; flush tlb; reset pte.

- Load double from level1_s, level2_s, and level3_s.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check pte; flush tlb; reset pte.

- Store byte to level1_m, level2_m, and level3_m.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check pte; flush tlb; reset pte.

- Store halfword to level1_m, level2_m, and level3_m.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check pte; flush tlb; reset pte.

- Store word to level1_m, level2_m, and level3_m.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check pte; flush tlb; reset pte.

- Store double to level1_m, level2_m, and level3_m.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check pte; flush tlb; reset pte.

- Swap to level1_m, level2_m, and level3_m.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check pte; flush tlb; reset pte.

- ldstub to level1_m, level2_m, and level3_m.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check pte; flush tlb; reset pte.

- ifetch from level1_s, level2_s, and level3_s.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check pte; flush tlb; reset pte.

This test is similar to mmu_xlate, mmu_u_hit_check, mmu_u_miss_check, mmu_s_miss_check. This is the hit case in supervisor mode. Some of the code gets conditionally executed.

mmu_s_inv_check

This diagnostic verifies SFSR and SFAR when an invalid address error is generated on level-3 access (Entry type is zero). The test sequence is:

- Do a load on level 3 (cause a data access trap), check the SFAR and SFSR.
- Do a store on level 3 (cause a data access trap), check the SFAR and SFSR.
- Do an instruction fetch on level 3 (cause a instruction access trap), check the SFSR.
- Do a store on level 2 (cause a data access trap), check the SFAR and SFSR.
- Do a store on level 1 (cause a data access trap), check the SFAR and SFSR.

This test is identical to `mmu_inv`. The other test in this category is `mmu_u_inv_check`. This test is run in supervisor mode.

mmu_s_miss_check

This basic diagnostic checks MMU translations at all levels for all load, store, and ifetch operations. The test sequence is as follows:

- `level1_s` and `level1_m` are mapped to regions.
- `level2_s` and `level2_m` are mapped to segments.
- `level3_s` and `level3_m` are mapped to pages.
- Load byte from `level1_s`, `level2_s`, and `level3_s`.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check `pte`, flush `tlb`; reset `pte`.
- Load halfword from `level1_s`, `level2_s`, and `level3_s`.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check `pte`, flush `tlb`; reset `pte`.
- Load word from `level1_s`, `level2_s`, and `level3_s`.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check `pte`; flush `tlb`; reset `pte`.
- Load double from `level1_s`, `level2_s`, and `level3_s`.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check `pte`; flush `tlb`; reset `pte`.
- Store byte to `level1_m`, `level2_m`, and `level3_m`.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check `pte`; flush `tlb`; reset `pte`.
- Store halfword to `level1_m`, `level2_m`, and `level3_m`.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check `pte`; flush `tlb`; reset `pte`.
- Store word to `level1_m`, `level2_m`, and `level3_m`.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check `pte`; flush `tlb`; reset `pte`.

- Store double to `level1_m`, `level2_m`, and `level3_m`.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check `pte`; flush `tlb`; reset `pte`.

- Swap to `level1_m`, `level2_m`, and `level3_m`.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check `pte`; flush `tlb`; reset pte.

- `ldstub` to `level1_m`, `level2_m`, and `level3_m`.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check `pte`; flush `tlb`; reset `pte`.

- `ifetch` from `level1_s`, `level2_s`, and `level3_s`.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check `pte`; flush `tlb`; reset `pte`.

This test is similar to `mmu_xlate`, `mmu_u_hit_check`, `mmu_u_miss_check`, `mmu_s_hit_check`. This is the miss case in supervisor mode. Some of the code gets conditionally executed.

mmu_s_protpriv_check

This diagnostic exhaustively verifies all illegal accesses, protection violations in user mode. The test sequence is as follows:

- Write to user read-only page (cause data access trap), check for access resulted in a trap, trap to change to supervisor, check SFAR has the correct value, check SFSR has the correct value, and go to the next test.
- Execute from user read-only page (cause instruction access trap), check for access resulted in a trap, trap to change to supervisor, check SFSR has the correct value, and go to the next test.
- Execute from user read / write-only page (cause instruction access trap), check for access resulted in a trap, trap to change to supervisor, check SFSR has the correct value, and go to the next test.
- Write to user read / execute-only page (cause data access trap), check for access resulted in a trap, trap to change to supervisor, check SFAR has the correct value, check SFSR has the correct value, and go to the next test.
- Read from user execute-only page (cause data access trap), check for access resulted in a trap, trap to change to supervisor, check SFAR has the correct value, check SFSR has the correct value, and go to the next test.
- Write to user execute-only page (cause data access trap), check for access resulted in a trap, trap to change to supervisor, check SFAR has the correct value, check SFSR has the correct value, and go to the next test.
- Write to supervisor read / execute-only page (cause data access trap), check for access resulted in a trap, check SFAR has the correct value, check SFSR has the correct value, and go to the next test.

- Execute from supervisor read / write-only page (cause instruction access trap), check for access resulted in a trap, trap to change to supervisor, check SFSR has the correct value, and go to the next test.
- Read from supervisor read / execute-only page (cause data access trap), check for access resulted in a trap, trap to change to supervisor, check SFAR has the correct value, check SFSR has the correct value, and go to the next test.
- Write to supervisor read / execute-only page (cause data access trap), check for access resulted in a trap, check SFAR has the correct value, check SFSR has the correct value and go to the next test.
- Execute from supervisor read / execute-only page (cause instruction access trap), check for access resulted in a trap, trap to change to supervisor, check SFSR has the correct value, and go to the next test.
- Write (supervisor mode) to supervisor read / execute-only page (cause data access trap), check for access resulted in a trap, check SFAR has the correct value, check SFSR has the correct value, and go to the next test.
- Read from supervisor read / write / execute-only page (cause data access trap), check for access resulted in a trap, trap to change to supervisor, check SFAR has the correct value, check SFSR has the correct value, and go to the next test.
- Write to supervisor read / write / execute-only page (cause data access trap), check for access resulted in a trap, check SFAR has the correct value, check SFSR has the correct value, and go to the next test.
- Execute from supervisor read / write-only page (cause instruction access trap), check for access resulted in a trap, trap to change to supervisor, check SFSR has the correct value, and go to the next test.

This test is similar to `mmu_protpriv` and `mmu_u_protpriv_check`. This test is run in supervisor mode.

mmu_test

This basic diagnostic verifies `tlb` miss on data references. The test sequence is as follows:

- Initialize context table, `lvl1_table`, `lvl2_table`, and `level3_table` with `ptp`s and `pte` entries for pages 0, 1, 2 and 3 in the text segment, pages 4, 5, and 6 in the data segment.
- Initialize `ctx` and `ctpr` to 1 and start of context table.
- Enable the MMU in supervisor mode.
- Do an `ld` and `st` on page 2 (should cause a `tlb` miss for data), do a probe entire to bring the next instruction page, trap to user mode.
- Do an `ld` and `st` in user mode (should cause a `tlb` miss for data) to make sure context match works.

mmu_test2_p

This basic diagnostic verifies the tablewalks at level 2 and level 3 for data pages in supervisor mode. The test sequence is as follows:

- Level 2 tablewalk, load word, load double word.
- Level 3 tablewalk, load word, load double word.
- Level 2 tablewalk, store word.
- Level 3 tablewalk, store word.
- Read back data to verify store.
- Read PTEs by probing (load ASI access).

mmu_test3_p

This basic diagnostic verifies the tablewalks at level 2 and level 3 for data pages in user mode. The test sequence is as follows:

- Level 2 tablewalk, load word, load double word.
- Level 3 tablewalk, load word, load double word.
- Level 2 tablewalk, store word.
- Level 3 tablewalk, store word.
- Read back data to verify store.
- Change to supervisor mode.
- Read PTEs by probing (load ASI access).

mmu_test4_p

This basic diagnostic verifies memory access in bypass mode in supervisor and user mode. The test sequence is as follows:

- Disable the MMU.
- Load double word in supervisor mode.
- Load word in supervisor mode.
- Load unsigned halfword in supervisor mode.
- Load unsigned byte in supervisor mode.
- Store double word in supervisor mode.
- Store word in supervisor mode.
- Store unsigned halfword in supervisor mode.
- Store unsigned byte in supervisor mode.
- Change to user mode.

- Load double word in user mode.
- Load word in user mode.
- Load unsigned halfword in user mode.
- Load unsigned byte in user mode.
- Store double word in user mode.
- Store word in user mode.
- Store unsigned halfword in user mode.
- Store unsigned byte in user mode.
- Plus self-checking on all the load and store location.

mmu_test5_p

This basic diagnostic verifies that memory access permissions are checked properly during access. The address translations are level-3 translations. The test sequence is:

- Set processor in supervisor mode.
- Test read-only type access when acc = 0.
- Test read/write type access when acc = 1.
- Test read for read/execute type access when acc = 2.
- Test read/write for read/write/execute type access when acc = 3.
- Test read/write for read/write type access when acc = 5.
- Test read for read/execute type access when acc = 6.
- Test read/write for read/write/execute type access when acc = 7.
- Set processor in user mode.
- Test read only type access when acc = 0.
- Test read/write type access when acc = 1.
- Test read for read/execute type access when acc = 2.
- Test read/write for read/write/execute type access when acc = 3.
- Test read/write for read/write type access when acc = 5.

There is no self-checking.

mmu_test6_p

This basic diagnostic verifies various cases of Instruction and Data translations occurring simultaneously and skewed by a cycle. Almost all the translations are level-3 pages, only one case of segment. The test sequence is as follows:

- Perform Instruction and Data tablewalks simultaneously with branch always.

- Perform Instruction and Data tablewalks simultaneously with conditional branch taken.
- Perform Data tablewalk one cycle before Instruction tablewalk simultaneously with conditional branch taken.
- Perform Instruction tablewalk one cycle before Data tablewalk with conditional branch taken.
- Perform Instruction tablewalk one cycle before Data tablewalk with conditional branch taken being on the odd boundary.
- Perform Instruction and Data tablewalks simultaneously but across segment boundary.
- Followed by checking of the `pte`s based on reading of the entries.

This is a pipeline timing-dependent test.

mmu_test7_p

This basic diagnostic verifies supervisor mode access checking. Almost all the translations are level-3 pages, only one case of segment. The data pages are mapped as supervisor read / write for all the test. The instruction access is described in the test. The test sequence is as follows:

- Perform Instruction and Data tablewalks simultaneously with branch always — access is user read / execute.
- Perform Instruction and Data tablewalks simultaneously with conditional branch taken — access is user read / write / execute.
- Perform Data tablewalk one cycle before Instruction tablewalk simultaneously with conditional branch taken — access is supervisor read / execute.
- Perform Instruction tablewalk one cycle before Data tablewalk with conditional branch taken — access is user execute-only.
- Perform Instruction tablewalk one cycle before Data tablewalk with conditional branch taken being on the odd boundary — access is supervisor read / write / execute.
- Perform Instruction and Data tablewalks simultaneously but across segment boundary — access is supervisor read / write / execute. Followed by checking of the pte’s based on probing of the entries.

mmu_test8_p

This basic diagnostic verifies user mode access checking. Almost all the translations are level-3 pages, only one case of segment. The data pages are mapped as supervisor read / write for all the test. The instruction access is described in the test. The test sequence is as follows:

- Perform Instruction and Data tablewalks simultaneously with branch always — access is user read / write / execute.
- Perform Instruction and Data tablewalks simultaneously with conditional branch taken — access is user read / write / execute.
- Perform Data tablewalk one cycle before Instruction tablewalk simultaneously with conditional branch taken — access is user read / write / execute.
- Perform Instruction tablewalk one cycle before Data tablewalk with conditional branch taken — access is user read / write / execute-only.
- Perform Instruction tablewalk one cycle before Data tablewalk with conditional branch taken being on the odd boundary — access is user read / write / execute.
- Perform Instruction and Data tablewalks simultaneously but across segment boundary — access is user read / write / execute.
- Followed by checking of the pte's based on probing of the entries.

mmu_u_hit_check

This basic diagnostic checks MMU translations at all levels for all load, store, and ifetch operations. The test sequence is as follows:

- `level1_s` and `level1_m` are mapped to regions.
- `level2_s` and `level2_m` are mapped to segments.
- `level3_s` and `level3_m` are mapped to pages.

- Load byte from `level1_s`, `level2_s`, and `level3_s`.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check `pte`, flush `tlb`; reset `pte`.

- Load halfword from `level1_s`, `level2_s`, and `level3_s`.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check `pte`, flush `tlb`; reset `pte`.

- Load word from `level1_s`, `level2_s`, and `level3_s`.
- PTE's are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check `pte`; flush `tlb`; reset `pte`.

- Load double from `level1_s`, `level2_s`, and `level3_s`.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check `pte`; flush `tlb`; reset `pte`.

- Store byte to `level1_m`, `level2_m`, and `level3_m`.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check `pte`; flush `tlb`; reset `pte`.

- Store halfword to `level1_m`, `level2_m`, and `level3_m`.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check `pte`; flush `tlb`; reset `pte`.

- Store word to `level1_m`, `level2_m`, and `level3_m`.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).

- Check data; check `pte`; flush `tlb`; reset `pte`.

- Store double to `level1_m`, `level2_m`, and `level3_m`.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check `pte`; flush `tlb`; reset `pte`.

- Swap to `level1_m`, `level2_m`, and `level3_m`.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check `pte`; flush `tlb`; reset `pte`.

- ldstub to `level1_m`, `level2_m`, and `level3_m`.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- check data; check `pte`; flush `tlb`; reset `pte`.

- ifetch from `level1_s`, `level2_s`, and `level3_s`.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check `pte`; flush `tlb`; reset `pte`.

This test is similar to `mmu_xlate`, `mmu_u_miss_check`, `mmu_s_hit_check`, `mmu_s_miss_check`. This is the hit case in user mode. Some of the code gets conditionally executed.

mmu_u_inv_check

This basic diagnostic verifies SFSR and SFAR when an invalid address error is generated on level-3 access (Entry type is zero). The test sequence is:

- Do a load on level 3 (cause a data access trap), check the SFAR and SFSR.
- Do a store on level 3 (cause a data access trap), check the SFAR and SFSR.
- Do a instruction fetch on level 3 (cause an instruction access trap), check the SFSR.
- Do a store on level 2 (cause a data access trap), check the SFAR and SFSR.
- Do a store on level 1 (cause a data access trap), check the SFAR and SFSR.

This test is similar to `mmu_inv` and `mmu_s_inv_check`. This test is run in user mode.

mmu_u_miss_check

This basic diagnostic checks MMU translations at all levels for all load, store, and `ifetch` operations. The test sequence is as follows:

- `level1_s` and `level1_m` are mapped to regions.
- `level2_s` and `level2_m` are mapped to segments.
- `level3_s` and `level3_m` are mapped to pages.

- Load byte from `level1_s`, `level2_s`, and `level3_s`.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check `pte`, flush `tlb`; reset `pte`.

- Load halfword from `level1_s`, `level2_s`, and `level3_s`.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check `pte`, flush `tlb`; reset `pte`.

- Load word from `level1_s`, `level2_s`, and `level3_s`.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check `pte`; flush `tlb`; reset `pte`.

- Load double from `level1_s`, `level2_s`, and `level3_s`.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check `pte`; flush `tlb`; reset `pte`.

- Store byte to `level1_m`, `level2_m`, and `level3_m`.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check `pte`; flush `tlb`; reset `pte`.

- Store halfword to `level1_m`, `level2_m`, and `level3_m`.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check `pte`; flush `tlb`; reset `pte`.

- Store word to `level1_m`, `level2_m`, and `level3_m`.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check `pte`; flush `tlb`; reset `pte`.

- Store double to `level1_m`, `level2_m`, and `level3_m`.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check `pte`; flush `tlb`; reset `pte`.

- Swap to `level1_m`, `level2_m`, and `level3_m`.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check `pte`; flush `tlb`; reset `pte`.

- `ldstub` to `level1_m`, `level2_m`, and `level3_m`.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check `pte`; flush `tlb`; reset `pte`.

- `ifetch` from `level1_s`, `level2_s`, and `level3_s`.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check `pte`; flush `tlb`; reset `pte`.

This test is similar to `mmu_xlate`, `mmu_u_hit_check`, `mmu_s_hit_check`, `mmu_s_miss_check`. This is the miss case in user mode. Some of the code is conditionally executed.

mmu_u_protpriv_check

This diagnostic exhaustively verifies that all illegal accesses, protection violations in user mode.The test sequence is as follows:

- Write to user read-only page (cause data access trap), check for access resulted in a trap, trap to change to supervisor, check SFAR has the correct value, check SFSR has the correct value, and go to the next test.

- Execute from user read-only page (cause instruction access trap), check for access resulted in a trap, trap to change to supervisor, check SFSR has the correct value, and go to the next test.
- Execute from user read/write-only page (cause instruction access trap), check for access resulted in a trap, trap to change to supervisor, check SFSR has the correct value, and go to the next test.
- Write to user read/execute-only page (cause data access trap), check for access resulted in a trap, trap to change to supervisor, check SFAR has the correct value, check SFSR has the correct value, and go to the next test.
- Read from user execute-only page (cause data access trap), check for access resulted in a trap, trap to change to supervisor, check SFAR has the correct value, check SFSR has the correct value, and go to the next test.
- Write to user execute-only page (cause data access trap), check for access resulted in a trap, trap to change to supervisor, check SFAR has the correct value, check SFSR has the correct value, and go to the next test.
- Write to supervisor read/execute-only page (cause data access trap), check for access resulted in a trap, check SFAR has the correct value, check SFSR has the correct value, and go to the next test.
- Execute from supervisor read/write-only page (cause instruction access trap), check for access resulted in a trap, trap to change to supervisor, check SFSR has the correct value, and go to the next test.
- Read from supervisor read/execute-only page (cause data access trap), check for access resulted in a trap, trap to change to supervisor, check SFAR has the correct value, check SFSR has the correct value, and go to the next test.
- Write to supervisor read/execute-only page (cause data access trap), check for access resulted in a trap, check SFAR has the correct value, check SFSR has the correct value, and go to the next test.
- Execute from supervisor read/execute-only page (cause instruction access trap), check for access resulted in a trap, trap to change to supervisor, check SFSR has the correct value, and go to the next test.
- Write (supervisor mode) to supervisor read/execute-only page (cause data access trap), check for access resulted in a trap, check SFAR has the correct value, check SFSR has the correct value, and go to the next test.
- Read from supervisor read/write/execute-only page (cause data access trap), check for access resulted in a trap, trap to change to supervisor, check SFAR has the correct value, check SFSR has the correct value, and go to the next test.
- Write to supervisor read/write/execute-only page (cause data access trap), check for access resulted in a trap, check SFAR has the correct value, check SFSR has the correct value, and go to the next test.
- Execute from supervisor read/write-only page (cause instruction access trap), check for access resulted in a trap, trap to change to supervisor, check SFSR has the correct value, and go to the next test.

This test is identical to mmu_protpriv. The other test in this category is mmu_s_protpriv_check. This test is run in user mode.

mmu_wp

This basic diagnostic verifies ptp lock and wraparound point in TLB replacement control register. The test sequence is as follows:

- Set TRCR[15], flush tlb all, do a Dcache fill with level-2 pte, verify tlb entry 1 with expected upper tag, lower tag, and level-1 ptp.
- Flush all tlb entries, do an Icache fill with level-3 pte, verify tlb entry 1.
- Set TRCR[14], flush all tlb entries, do an Icache fill with level-3 pte, verify tlb entry 0.
- Set TRCR[16], flush all tlb entries, do an Icache fill with level-3 pte, verify tlb entry 2.
- Set TRCR[14-16] to lock level 0-2 ptp, flush all tlb entries, do an Icache fill with level-3 pte, verify tlb entry 0-2 with expected ptp.
- WP test:
  - Set up 64 level-3 reserved pte (ET=3) in memory for data access.
  - Clear TRCR[14-16], flush all tlb entry, set TRCR.wp = 62 (so entry 0-62 can be replaced) first, set up tlb entry 63 to make MMU hit at Dcache fill (this page is mapped as reserved in memory), do an MMU flush to knock out instruction page, also flush Icache to make next Icache fill cause Icache fill along with MMU miss. Do a simultaneous Icache fill with mmu miss and Dcache fill with MMU hit. TRCR.wp is set up to make sure tablewalk for Icache fill will not replace data page's pte, because data page pte is always set up next to wraparound point. If data page's pte is replaced unexpectedly, diagnostic will trap on translation error, then jump to BAD_TRAP.
  - TRCR.wp is 62 first, it is decremented by 1 until it reaches 1. It verifies loaded data after each data access.

mmu_xlate

A basic test to check MMU translations at all levels for all load, store, and ifetch operations. The test sequence is as follows:

- level1_s and level1_m are mapped to regions.
- level2_s and level2_m are mapped to segments.
- level3_s and level3_m are mapped to pages.
- Load byte from level1_s, level2_s, and level3_s.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check pte, flush tlb; reset pte.
- Load halfword from level1_s, level2_s, and level3_s.

- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check `pte`, flush `tlb`; reset `pte`.

- Load word from `level1_s`, `level2_s`, and `level3_s`.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check `pte`; flush `tlb`; reset `pte`.

- Load double from `level1_s`, `level2_s`, and `level3_s`.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check `pte`; flush `tlb`; reset `pte`.

- Store byte to `level1_m`, `level2_m`, and `level3_m`.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check `pte`; flush `tlb`; reset `pte`.

- Store halfword to `level1_m`, `level2_m`, and `level3_m`.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check `pte`; flush `tlb`; reset `pte`.

- Store word to `level1_m`, `level2_m`, and `level3_m`.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check `pte`; flush `tlb`; reset `pte`.

- Store double to `level1_m`, `level2_m`, and `level3_m`.
- PTE's are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check `pte`; flush `tlb`; reset `pte`.

- Swap to `level1_m`, `level2_m`, and `level3_m`.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check `pte`; flush `tlb`; reset `pte`.

- `ldstub` to `level1_m`, `level2_m`, and `level3_m`.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check `pte`; flush `tlb`; reset `pte`.

- `ifetch` from `level1_s`, `level2_s`, and `level3_s`.
- PTEs are marked (!M, !R, !C, CXT=1, ACCESS).
- Check data; check `pte`; flush `tlb`; reset `pte`.

This test is the template for other tests. It is identical to `mmu_s_miss_check`.This test is similar to `mmu_u_hit_check`, `mmu_u_miss_check`, `mmu_s_hit_check`. This is the miss case in supervisor mode. Some of the code is conditionally executed.

nabpcr

This basic diagnostic verifies the processor control register. All the bits in the register are toggled so various modes are enabled and disabled. The test disables the MMU, Instruction Cache, Data Cache. It enables the Instruction translation Buffer Register, alternate cacheable bit, data view modes.

noFaultTab

This diagnostic exhaustively verifies the no-fault bit operation. The test sets the no-fault bit in the processor control register (`pcr`). The test generates translation error, invalid address error, privilege violation error, and protection error.

The diagnostic describes in great detail the relevant cases.

noFault_v

This basic diagnostic verifies the no-fault bit operation. The test sequence is as follows:

- Invalid translation with supervisor access, check `sfsr`.
- Invalid translation with user access, check `sfsr`.
- Illegal asi 0x15 access, check `sfsr`.

overWrite_v

This diagnostic was ported from SuperSPARC-I to verify setting of overwrite bit in the SFSR. An exhaustive cause to verify multiple faults is `mmuFsrCombo`.

owInvTrans_v

This diagnostic was ported from SuperSPARC-I to verify cases of overwrite bit setting due to invalid translation.The test sequence is:

- Invalid translation at level 3, check for invalid error, check trap type, check `sfar`, check data.
- Level-3 PTE has entry type as reserved, check for invalid error, check trap type, check `sfsr`, check `sfar`, check data.

probe_v

This diagnostic exhaustively verifies all the probe operations. The cases of interest are page, segment, region, context, and entire probes.

The test case is well defined in the diagnostic.

sizeErr_v

This diagnostic exhaustively verifies invalid size MMU ASI accesses. The ASI accesses of interest are 0x3 (reference MMU flush / probe), 0x4 (mmu registers), 0x6 (read and write of `tlb` entries and `itbr` register). The data access error should be reported for all the cases. The test sequence is:

- Probe using `ldsba`, check trap type, checking of `sfar`, followed by checking of `sfsr`.
- Probe using `ldsha`, check trap type, checking of `sfar`, followed by checking of `sfsr`.
- Probe using `lduba`, check trap type, checking of `sfar`, followed by checking of `sfsr`.
- Probe using `lduha`, check trap type, checking of `sfar`, followed by checking of `sfsr`.
- Probe using `ldda`, check trap type, checking of `sfar`, followed by checking of `sfsr`.
- Flush using `stba`, check trap type, checking of `sfar`, followed by checking of `sfsr`.
- Flush using `stha`, check trap type, checking of `sfar`, followed by checking of `sfsr`.
- Flush using `stda`, check trap type, checking of `sfar`, followed by checking of `sfsr`.
- Flush using `ldstuba`, check trap type, checking of `sfar`, followed by checking of `sfsr`.
- Access mmu register using `ldsba`, check trap type, checking of `sfar`, followed by checking of `sfsr`.
- Access mmu register using `ldsha`, check trap type, checking of `sfar`, followed by checking of `sfsr`.
- Access mmu register using `lduba`, check trap type, checking of `sfar`, followed by checking of `sfsr`.
- Access mmu register using `lduha`, check trap type, checking of `sfar`, followed by checking of `sfsr`.
- Access mmu register using `ldda`, check trap type, checking of `sfar`, followed by checking of `sfsr`.
- Access mmu register using `stba`, check trap type, checking of `sfar`, followed by checking of `sfsr`.
- Access mmu register using `stha`, check trap type, checking of `sfar`, followed by checking of `sfsr`.
- Access mmu register using `stda`, check trap type, checking of `sfar`, followed by checking of `sfsr`.

- Access mmu register using `ldstuba`, check trap type, checking of `sfar`, followed by checking of `sfsr`.
- Read `tlb` using `ldsba`, check trap type, checking of `sfar`, followed by checking of `sfsr`.
- Read `tlb` using `ldsha`, check trap type, checking of `sfar`, followed by checking of `sfsr`.
- Read `tlb` using `lduba`, check trap type, checking of `sfar`, followed by checking of `sfsr`.
- Read `tlb` using `lduha`, check trap type, checking of `sfar`, followed by checking of `sfsr`.
- Read `tlb` using `ldda`, check trap type, checking of `sfar`, followed by checking of `sfsr`.
- Write `tlb` using `stba`, check trap type, checking of `sfar`, followed by checking of `sfsr`.
- Write `tlb` using `stha`, check trap type, checking of `sfar`, followed by checking of `sfsr`.
- Write `tlb` using `stda`, check trap type, checking of `sfar`, followed by checking of `sfsr`.
- Write `tlb` using `ldstuba`, check trap type, checking of `sfar`, followed by checking of `sfsr`.

CHAPTER **6**

Instruction and Data Cache Tests

This chapter describes the Instruction and Data Cache Validation test suites.

atomic_parity.s

This test verifies parity errors during `ldstub` operations. In SuperSPARC-II, a parity error on a D$ fill causes lvl 15 interrupt to be signalled and the cache tag is invalidated if it is a cacheable access. If a parity error occurs on the load portion of an atomic operation, the store is still completed to the memory. The following cases are covered:

- Case 1: parity error on a noncacheable `ldstub` followed by a cacheable `ldstub` miss.
- Case 2: parity error on a cacheable `ldstub` miss followed by a cacheable `ldstub` miss.
- Case 3: parity error on a cacheable load miss followed by a cacheable `ldstub` miss.
- Case 4: parity error on noncacheable load miss followed by cacheable `ldstub` miss.
- Case 5: parity error on a cacheable load miss followed by cacheable `ldstub` miss with parity error.
- Case 6: parity error on noncacheable load miss followed by cacheable `ldstub` miss with parity error.
- Case 7: parity error on a cacheable load FP miss followed by cacheable `ldstub` miss.
- Case 8: parity error on cacheable load double FP miss followed by cacheable `ldstub` miss.
- Case 9: parity error on cacheable store halfword miss followed by cacheable `ldstub` miss.
- Case 10: parity error on noncacheable store halfword followed by cacheable `ldstub` miss.
- Case 11: parity error on noncacheable store byte followed by cacheable `ldstub` miss with parity error.

- Case 12: parity error on noncacheable store byte followed by cacheable `ldstub` with parity error.

atomic_parity2.s

Same as `atomic_parity.s` but executes `swap` instructions instead of `ldstub`.

cache_case3

This test case verifies the function of cache interlocks when two or more instructions reference the cache simultaneously during the various stages in the pipline.The test uses a double-word alignment. Both the even word and odd word cases are verified. The sequence is as follows:

- Store halfword followed by a load unsigned halfword.
- Store word followed by a load double word.
- Store byte followed by a load store unsigned byte.
- Swap followed by a store double word.
- Store word followed by a jump and link instruction.
- Store word followed by a store floating-point word.
- Store followed by a `branch/call` instruction.

cache_case4

This test case verifies the function of cache interlocks when two or more instructions reference the cache simultaneously during the various stages in the pipline.The test uses a double-word alignment, so both the even word and odd word cases are verified. The sequence is as follows:

- Store double word followed by a load word.
- Store double word followed by a load double word.
- Store double word followed by a store halfword.
- Store double word followed by a store double word.
- Store double word followed by a jump and link instruction.
- Store double word followed by a store floating-point word.
- Store followed by a `branch/call` instruction.

cache_case6

This test case verifies the function of cache interlocks when two or more instructions reference the cache simultaneously during the various stages in the pipline.The test uses a double-word alignment. Both the even word and odd word cases are verified. The sequence is as follows:

- Store floating-point double word followed by a load word.
- Store floating-point word followed by a load double word.
- Store floating-point word followed by a store byte.
- Store floating-point (`fsr`) followed by a store double word.
- Store floating-point double word followed by a jump and link instruction.
- Store floating-point word followed by a store floating-point word.
- Store floating-point word followed by a `branch/call` instruction.

cache_flush1

This exhaustive cache flush test focuses on tag's context. The test does walk 1 on context bit and does different flush.

- Context match / mismatch, page flush.
- Context match / mismatch, user flush.
- Context match / mismatch, context flush.
- Context match / mismatch, page flush, S=1.
- Context match / mismatch, user flush, S=1.
- Context match / mismatch, context flush, S=1.
- Context and page mismatch, context flush, S=1.

`cache_flush1.s` and `cache_flush2.s` are basic but exhaustive cache flush tests; the two diagnostics are self-checking and `asm2ver` style.

cache_flush2

This exhaustive cache flush test focuses on tag's address bit. The test does walk 1 on address bit and does different flush.

- Region match / mismatch, page flush.
- Region match / mismatch, region flush.
- Region match / mismatch, context flush.
- Region match / mismatch, user flush.
- Segment match / mismatch, user flush.
- Segment match / mismatch, context flush.
- Segment match / mismatch, region flush.
- Segment match / mismatch, segment flush.
- Segment match / mismatch, page flush.
- Page match / mismatch, user flush.
- Page match / mismatch, context flush.
- Page match / mismatch, region flush.
- Page match / mismatch, segment flush.
- Page match / mismatch, page flush.
- Bit 12 of Dcache, bit 13 of Icache page flush

cache_lvl.s

This basic test checks `level` field (level 1-3) in I$ and `acc` field (acc = 2) in D$.

I$ and D$ fill from address space mapped as 4K, 256K, and 16M with acc = 2. Check Icache tag `level` field and Dcache tag `acc` field.

cache_test1

This simple test case verifies branching with Instruction Cache operations. The test causes an Icache miss, followed by branch to self, forward, and backward.

icache_asi

This basic test case verifies Instruction Cache asi operations. The test cases covered in this diagnostic are as follows:

- Enable the Instruction Cache, jump to the top of the virtual page, execute its first block, and check the cache tags and data.
- Issue an `iflush`, make sure the Icache block zero valid bit is clear, and check the cache tags and data.
- Flash clear the Instruction Cache, make sure the block zero valid bit is clear, and check the cache tags and data.
- Clear the valid bit for block zero by writing to it, and check the cache tags and data.
- Modify the code in the Instruction Cache, execute the new code, and then check that the new code did work.

icache_tests

This basic test case verifies Instruction Cache operations. The test cases covered in this diagnostic are as follows:

- TLB address mismatch, Icache line invalid.
- Icache line invalid.
- Icache address mismatch.
- TLB address mismatch, the Icache page is noncacheable.
- `itbr` miss by fetching the target of a branch from a different page, simultaneously with a data load. The Ifetch will also cause a `tlb` miss and the load causes a Dcache miss.
- `itbr` miss by fetching the target of a branch from a different page, simultaneously with a data load. The Ifetch will hit the `tlb`, and load will also hit the Dcache.
- Store causes a first-write `tlb` miss, simultaneous with an `itbr` miss.

- Flush the `tlb`, modify the PTE for a page to read-only, and then attempt to execute from that page and write to that page.

iu_stream.s

`iu_stream.s` tests streaming logic for the Icache and Dcache. The following cases are covered:

- I$ streaming followed by flush of the present line and different cache line.
- I$ streaming followed by an access to same line or different lines.
- I$ streaming on a branch instruction within the same line.
- I$ streaming on a branch instruction to different lines.
- I$ streaming on an annulled branch.
- I$ streaming followed by exception in same line and different lines.
- D$ streaming followed by access to same line or different lines.
- D$ streaming followed by flush of same line and different lines.
- D$ streaming followed by an access to same line and different lines causing exceptions.
- D$ and I$ streaming in parallel.

mmu_IT_accat.s

`mmu_IT_accat.s` tests ACC/AT on behalf of Icache. It does not check SFSR (that checking is done in `mmu_I_accat_chk`), but it checks valid bit validity after every test.

If set up cacheable page with acc 0-7, set up Icache tag to make each access will be Icache hit, do AT access 2-3 with acc 0-7. Because `Icache` tag has only S bit instead of Acc field, this test only checks privilege error. If there is a protection error on cache line fill, Icache tag will remain invalid. It is not appropriate to test `Icache` hit with protection error.

This is a megacell test, more on `mmu_I_accat_chk`.

mmu_ca.s

`mmu_ca.s` tests bit 7 (cache allocation) on Process Control Register.

- Set bit 7 in `pcr`.
- Flush cache, store miss access, check Dtag still invalid.
- Load access, check Dtag validity.
- Store hit with `PCR.ca`, check Dtag and Ddata RAM with expected data.

nabD1

This basic test case verifies store operation for all operand sizes (double word, word, halfword and byte). Only the cache hit case is covered. The cache tags and data are verified by `asi` accesses. The test sequence also includes atomic operations like swap and load-store unsigned. The atomic operations cover both the hit and miss cases. The test is self-checking and uses ASI accesses.

nabD2

This basic test verifies cache hit and miss operations during loads and stores. The test does a load operation with a tag match but, invalid line and tag mismatch, but valid line. The test does a store operation with a tag match, but invalid line and tag mismatch, but valid line. The tests are self-checking and use ASI accesses.

nabD3

This basic test verifies some interesting sequences with data cache. It verifies multicycle operation like unsigned multiply during data cache miss fill sequence. The test does a load operation with tag mismatch, but valid line. The test attempts to do a store with the store buffer full. It also does some noncached data cache access. The test is self-checking and uses ASI accesses.

nabDcache

This basic test verifies Data Cache ASI operations. The test cases covered in this diagnostic are as follows:

- Perform a store operation, check the cache tag and data using ASI accesses.
- Execute Iflush instruction. The Data Cache should not be cleared.
- Clear data cache valid bit using `asi` access.
- Set data cache valid bits using `asi` access.
- Store double with cache disabled, check data tag and data RAM.
- Load double with cache disabled, check data tag and data RAM.
- Test write-through for a valid line in Data Cache.

nabI2

This basic test verifies Instruction Cache operations with branch operations. The test includes branching to self, branch out of fill line. The test checks both the cache tags and data.

nabpar

This basic test verifies parity errors. The test toggles the parity from odd to even, generates a parity error and makes sure that the line is invalid in the cache.

For the `nabcombo*` and `asmcombo*` diagnostics, the test code is identical. The `nabcombo*` diagnostics are non-`asm2ver`-style diagnostics. The asmcombo* diagnostics are `asm2ver`-style diagnostics.

tst_icache.s

This simple test verifies I$ megacell. It does a single read and write and tests the bypass path on the I$ megacell.

CHAPTER 7

Memory Interface Unit Tests

This chapter covers the memory interface unit diagnostics.

mem_adel_ns

This test verifies the memory address decode and evaluates logic. A walking 0's and 1's pattern is used. The address decode and evaluate logic (ADEL) gates the row/column address and memory control signals required for the current operation out to memory.

This test requires SAS, but instruction-to-instruction compare is not done.

mem_asm

This test verifies the memory arbitration state machine. The following sequence is checked:

- Read/write in paged/nonpaged mode with and without refresh.

mem_cbr.ref

This test verifies the CAS-before-RAS refresh cycle operation. A CAS-before-RAS refresh cycle is performed on all DRAM and VRAM banks.

This test should cover all the refresh cases.

mem_d.rd.8b

This test verifies reading of 8 bytes from memory. The 8 bytes are read from DRAM in a single operation, using a double-word read cycle. The read is paged or nonpaged, using the address supplied on the physical address bus. The read is either from the IU or SBus.

mem_d.rd.16b

This test verifies reading of 16 bytes from memory. This is used to fill one line of D-cache or do an SBus burst read of 16 bytes. The 16 bytes are read from DRAM in a single operation, using two double-word read cycles. The first read is paged or nonpaged, using the address supplied on the physical address bus. The next cycle is a paged read, where ADEL will increment or wrap the address to read a 16-byte aligned block from memory.

mem_d.rd.32b

This test verifies reading of 32 bytes from memory. This is used to fill one line of I-cache Instruction fetch. The 32 bytes are read from the DRAM in a single operation, using four double-word read cycles. The first read is paged or nonpaged, from the address given on the physical address bus. The following three reads are paged. The ADEL will supply the address for the next three reads, incrementing or wrapping it as necessary, in order to read a 32-byte aligned block and fill a whole Instruction Cache line.

mem_d.wr.4b

This basic test verifies writing of 4 bytes from memory. The 4 bytes are written to DRAM in a single operation, using a word write cycle. The write is paged or nonpaged, using the address supplied on the physical address bus. The write is either from the IU or SBus.

mem_d.wr.8b

This test verifies writing of 8 bytes from memory. The 8 bytes are written from DRAM in a single operation, using a double-word write cycle. The write is paged or nonpaged, using the address supplied on the physical address bus. The write is either from the IU or SBus.

mem_d.wr.16b

This test verifies writing of 16 bytes from SBC. The 8 bytes are written to DRAM in a single operation, using two double-word write cycles. The first write is paged or nonpaged, using the address supplied on the physical address bus. The second write is a paged write where ADEL will increment or wrap the address to write the next 8 bytes from SBC. The write is always from SBus for 16-byte burst write.

mem_d.rmw.b

This test verifies read-modify-write of a byte from memory. A paged or nonpaged read occurs followed by a paged word write, using the same address supplied on the physical address bus. The Memory Control Block (MCB) will perform the read and write cycles and will instruct the Data aligner-Parity Check generation logic (DPC) to latch the byte write data from the source, insert it in the appropriate byte of the word read from memory, and then gate it back on the memory data bus as the write data. The read-modify-write is either from the IU or SBus.

mem_d.rmw.2b

This test verifies read-modify-write of 2 bytes from memory. A paged or nonpaged read occurs followed by a paged word write, using the same address supplied on the physical address bus. The Memory Control Block (MCB) will perform the read and write cycles and will instruct the Data aligner-Parity Check (DPC) generation logic to latch the 2-byte write data from the source, insert it in the appropriate halfword of the word read from memory and then gate it back on the memory data bus as the write data. The read-modify-write is either from the IU or SBus.

mem_dpc

This test verifies data aligner logic, parity control generation and checker logic. The test toggles through various data patterns. The test covers the following cases:

- Even parity enabled
- Even parity disabled
- Odd parity enabled
- Odd parity disabled

mem_pnp

This test verifies the mmu page crossing accesses at page boundaries.

7.1 FlashPROM Controller 8-bit Tests

The 8-bit tests are designed to specifically test the 8-bit FlashPROM. To do this, the option `-boot flash8` must be used. The 8-bit FlashPROM only allows 8-bit reads, 8-bit writes, and 64-bit (dword) reads. This behavior can be seen with the cache off. With the cache on, the 8-bit reads appear as dword reads, and therefore only 8-bit writes and dword reads will be seen.

flash8_basic.s

`flash8_basic` consists of the following four tests:

- `flash8_afx1` — Conversion of `afx1` test to 8-bit writes/reads
- `flash8_afx2` — Conversion of `afx2` test to 8-bit writes/reads
- `flash8_multi`— Multiple 8-bit writes to the same location and then an 8-bit read
- `flash8_consec` — 8-bit writes to the consecutive locations and the 8-bit consecutive reads from those locations

flash8_adel.s

`flash8_adel` (FAIL — `rtl` does dword reads) marching pattern with 8-bit writes and reads.

flash8_dword.s

This test does byte writes and then dword reads of those locations.

flash8_256.s

This test is specifically designed to test the programming of the 8-bit FlashPROM. The test must be run with the `-flash_pm` mode to indicate that the PROM is programmed, that is, it recognizes the 0x40 on the data bus as a signal that the PROM should enter programming mode. The test first writes data of 0x40 to indicate programming mode, then writes the data to program, and finally ends by writing 0xC0 data to indicate the end of programming. This is done for 256 entries.

7.2 FlashPROM Controller 32-bit Tests

The 32-bit tests are designed to specifically test the 32-bit FlashPROM. The default boot option boots from the 32- bit FlashPROM; no command-line option is needed to specify the 32-bit FlashPROM. The 32-bit Flash PROM only allows 32-bit reads, 32-bit writes, and 64-bit (dword) reads. This behavior can be seen with the cache off. With the cache on, the 32-bit reads appear as dword reads, and therefore only 32-bit writes and dword reads will be seen.

flash32_basic.s

`flash32_basic` consists of the following four tests:

- `flash32_afx1` — Conversion of `afx1` test to 32-bit writes/reads
- `flash32_afx2` — Conversion of `afx2` test to 32-bit writes/reads
- `flash32_multi` (not implemented) — Multiple 32-bit writes to the same location and then n 32-bit read
- `flash32_consec` — 32-bit writes to the consecutive locations and then 32-bit consecutive reads from those locations

flash32_adel.s

`flash32_adel.s` performs marching pattern with 32-bit writes and reads (MPSAS fails when an incorrect pattern is returned).

flash32_dword.s

`flash32_dword.s` performs byte writes and then dword reads of those locations.

flashPromcombo.s

This test does mixed writes/reads of 32-bits to AFX and reads/writes of 8/16/32/64 from DRAM.

flashPromcomboQ1.s

This test does write/reads of 32-bits to AFX and 32-bit dword writes/reads to the same and different pages.

flashPromcombos3.s

This test does mixed writes/reads of 32-bits to AFX and reads/writes of 8/16/32/64 from DRAM. The accesses are read-modify-writes and are made to the same addresses, previous addresses, same pages, and different pages.

CHAPTER **8**

PCI Controller Unit Tests

This chapter covers the PCI controller unit tests.

The PCI diagnostic setup consists of four RaviCad (Virtual Chips) masters with one RaviCad slave and one LMC slave.

- The LMC slave supports command files, which makes it is easier to change parameters like `Trdy`, termination, and decodes. The source file has to be changed for changing parameters in the RaviCad model. The LMC slave also supports three memory address ranges and three I/O address ranges.
- The RaviCad slave supports one memory range and one I/O range. RERR generation is not functional in the LMC slave but is functional in the RaviCad slave, making it useful to maintain a RaviCad slave as well.
- The RaviCad masters support the following commands: memory read line, memory read multiple, memory write, and invalidate. Also, `Irdy` now has random delays.

In addition, both the RaviCad and the LMC bus monitors are used in the test environment.

arb3.s

`arb3.s` tests priority arbitration.

- Set 011 100 010 001 000 /3888 with LMC PCI master.

arb3_1.s

`arb3_1.s` tests priority arbitration.

- Set 000 001 010 011 100 /029C

Each PCI master does a 512-byte write and read-back and microSPARC-IIep as PCI master also does a 512-byte write and read-back. To which master finishes operations first (all these `arb3_x.s` test are the same pattern except for the priority setting) with LMC PCI master.

arb3_2.s

`arb3_2.s` tests priority arbitration.

- Set 010 011 000 001 100 / 260C with LMC PCI master.

arb3_3.s

`arb3_3.s` tests priority arbitration.

- Set 011 010 100 001 000 / 3508 with LMC PCI master.

arb3_4.s

`arb3_4.s` tests priority arbitration.

- Set 001 010 011 100 000 / 14E0 with LMC PCI master.

arb3_5.s

`arb3_5.s` tests priority arbitration.

- Set 011 100 001 000 010 / 3842 with LMC PCI master.

arb3_6.s

`arb3_6.s` tests priority arbitration.

- Set default 100 011 010 001 000 / 4688 with LMC PCI master.

addrbits.s

This test checks all address bits from AFX master into main memory. Does PCI DMA master writes into main memory, using a walking 1 address. The CPU side waits for non-zero semaphore value, then checks for correct value at each address. Values are also read back and compared from the DMA master.

For the second part of the test, the DMA master clears the test locations to 0, then performs the same test, write, read, write, read…. This activity starts while the CPU is still comparing the first part of the test. The CPU then checks for a second semaphore flag that indicates this second portion of the test is complete. The CPU does not check the values again for the second part of the test.

banks.s

`banks.s` tests AFX master to `memif` corner cases at bank boundaries (bit 24). DMA master performs 16 dword bursts squarely across bit 24 boundaries. The CPU polls for a unique flag that indicates when each of the bursts completes. The CPU then checks the data for each burst.

cheerio_pci.s

This diagnostic does multidata phase memory transactions from PCIC master to fast LMC slave model. In the run command, `-boot cheerio` is included.

burst_wr.s

`burst_wr.s` is a simple test of DMA burst write and read. The CPU side simply polls while DMA master performs a few 8-`dword` burst writes and reads. Data is only checked on the PCI master side.

bytes.s

`bytes.s` is a simple test of DMA byte write and read. The CPU side polls while DMA master performs a mixture of byte writes and reads. Data is only checked on the DMA master side.

collide.s

For `collide.s` get `memif` to access the same two memory locations from an FX master and the CPU at the same time. Attempt to tri-up `memif`. CPU and DMA write and read the same values to the same two address locations at the same time. Since CPU and DMA are writing the same values, test fails if the data gets mangled or confused with another address.

do4.s

For `do4.s`, make sure PCIC can accept a burst operation from each of the four masters. Program each of the four masters to perform a 4-dword burst to separate address ranges at the same time. DMA masters burst a write, then read, and then check the read data. CPU polls, waiting for all to complete. CPU checks burst results. This test has four command files, `do4_0.cmds` through `do4_3.cmds`, from which to generate transactions on the PCI bus from the four masters (0-3).

extarbit.s

`extarbit.s` tests the disabling of the microSPARC-IIep PCI arbiter and uses the external PCI arbiter.

falcon_pcim.s

`falcon_pcim.s` tests all the basic commands supported by microSPARC-IIep as a `pci` master. It also tests write burst mode and read prefetch mode.

falcon_pcim_bm.s

`falcon_pcim_bm.s` tests that all combinations of byte, halfword, word, double word can be written and read from the PCI bus.

falcon_pcim_mbase0.s

`falcon_pcim_mbase0.s` tests the memory base 0 registers.

falcon_pcim_mbase1.s

`falcon_pcim_base1.s` tests the memory base 1 registers.

falcon_pcim_ibase.s

`falcon_pcim_ibase.s` tests the I/O base registers. This diagnostic is not self-checking.

hshake.s

For `hshake.s`, to test illegal byte, enable in I/O space access, as generated by RaviCad PCI master.

illbe.s

`illbe.s` is a true interrupt mask test.

intr_msk_#-#.s

These tests are true interrupt mask tests. The interrupt mask ranges #-# are split into separate tests.

intr_lvl.s

This test checks PCI/IU interrupt signaling. Results are checked by data compare (MMU is disabled; default startup). The test walks a count on the four PCI EXT_INT pins (1-15). The data table contains at `expect_lvls`: the interrupt level expected for each EXT_INT pattern. This verifies that all levels can fire correctly and that the priority encoding is correct. Test 1: Enable interrupts, wait for a single interrupt from the PCIC; if trap occurs, set flag, clear interrupt request, and try next level.

iodma.s

`iodma.s` tests PCI I/O and memory space response when microSPARC-IIep as PCI slave setting:

- Set base2 as I/O space; range from 0xd0006000 to 0xd0006fff -- 4BK
- Set base3 as I/O space; range from 0xd000f000 to 0xd000ffff -- 4BK
- Set base4 as mem space; range from 0xd000a000 to 0xd000afff -- 4BK
- Set base5 as mem space; range from 0xd000e000 to 0xd000efff -- 4BK

- Tasks:
  - Master_0 write/read microSPARC-IIep io space base2 by byte, h-word, word, d-word, and 4/8-word burst
  - Master_3 write/read microSPARC-IIep io space base3 by byte, h-word, word, d-word, and 4-word burst
  - Master_1 write/read microSPARC-IIep mem space base4 by byte, h-word, word, d-word, and 4-word burst
  - Master_2 write/read microSPARC-IIep mem space base5 by byte, h-word, word, d-word, and 4-word burst

iosize.s

This test verifies the PCI DMA access to the IAFX DRAM, using the PCI address space size 256B. The test verifies the PCI DMA is accepted by the PCI slave interface for I/O operation, based on the PCIBASE0-5 along with the size specified by the PCISIZE0-5 registers.

iosize1.s

This test verifies the PCI DMA access to the IAFX DRAM, using the PCI address space size 1KB, 256B, 64KB, and 512KB. This test verifies the PCI DMA is accepted by the PCI slave interface for I/O operation, based on the PCIBASE0-5 along with the size specified by the PCISIZE0-5 registers. The size and DMA addresses are programmed such that the addresses cross over the next page boundary. The PCIBASE1 address register is initialized to respond to I/O command, and the PCIBASE3 is programmed to respond to memory.

iosize2.s

This test verifies the PCI DMA access to the IAFX DRAM, using the PCI address space size 256B. This test verifies the PCI DMA burst is accepted by the PCI slave interface for I/O operations when the DMA address is overlapped to match the content of the next PCIBASE address register. The PCIBASE0 is programmed to 0xF0001001, and the PCIBASE1 is programmed to 0xF0001101. The PCI Bus Transactor is set to request DMA burst transfer of 260 bytes to cross the PCISIZE0 (256-byte) boundary. The Command register in PCI Interface is programmed to respond to I/O.

iosize3.s

This test verifies the PCI DMA access to the IAFX DRAM, using the PCI address space size 256B. The test verifies the PCI DMA is accepted by the PCI slave interface for I/O operation only, based on the PCIBASE0-5 along with the size specified by the PCISIZE0-5 registers. The size and DMA addresses are programmed such that the addresses crosses over the next page boundary. The PCIBASE0, PCIBASE2, and PCIBASE4 are initialized to respond to I/O commands, and the PCIBASE1, PCIBASE3, and PCIBASE5 are initialized to respond to memory access. The PCI Bus transactors are programmed to issue I/O-command-type transactions only. Therefore, there should not be any DMA transactions accepted by the PCI save interface when crossing over the next page boundaries.

iotlb.s

`iotlb.s` tests the IOTLB's ability to perform an address translation using all 16 entries. It also checks IOTLB-miss level 15 interrupt. Uses `#define` ENABLE_IOTLB switch for default IOTLB setup. The DMA master does a DMA burst write and read to addresses that translate to unique spaces in main memory. The last DMA operation is an intentional miss that causes a level 15 interrupt. The level 15 handler on the CPU side checks these spaces for correct data.

iotlbmem.s

`iotlbmem.s` performs a simple RAM test on the 16 IOTLB entries, using the direct access registers. It puts a unique data pattern into each of the 16 IOTLB CAM and RAM locations, masking out the reserved fields. It reads back and compares values.

iotlb_flush.s

This test checks to see if individual lines of the IOTLB can be flushed.

lmcm.s

`lmcm.s` tests some PCI corner cases—dual address, address stepping, serr#, perr#, … with LMC PCI master.

long.s

`long.s` tests long burst (1 Kbyte) DMA writes and reads, measuring microSPARC-IIep DMA peak transfer rate.

memcmd.s1

This test does the following:

- Tests `WRTINVMEM`, `RDMULTIMEM`, and `RDLINEMEM` PCI commands response with microSPARC-IIep as PCI slave; burst length: 4/8/12-word.
- Tests DMA odd starting address (address start from 1, 2, or 3); scenario:
  - DMA burst starting byte enable could be 1110/1100/1000
  - DMA burst ending byte enable could be 0001/0011/0111
- Tests some memory spaces that do not belong to microSPARC-IIep -- belong to LMC slave or no one's space -- abort.
- Uses external PCI arbiter.

memsize.s

This test verifies the PCI DMA access to the IAFX DRAM, using the PCI address space size 256B, 512B, 1KB, 2KB, 4KB, and 8KB. This test verifies the PCI DMA is accepted by the PCI slave interface for memory operation, based on the PCIBASE0-5 along with the size specified by the PCISIZE0-5 registers.

memsize1.s

This test verifies the PCI DMA access to the IAFX DRAM, using the PCI address space size 16KB, 32KB, 64KB, 128KB, 256KB, and 512KB. This test verifies the PCI DMA is accepted by the PCI slave interface for memory operation, based on the PCIBASE0-5 along with the size specified by the PCISIZE0-5 registers.

memsize2.s

This test verifies the PCI DMA access to the IAFX DRAM, using the PCI address space size 256 bytes. It also verifies the PCI DMA burst is accepted by the PCI slave interface for memory operation when the DMA address is overlapped to match the content of the next PCIBASE address register. The PCIBASE0 is programmed to 0xF00010, and the PCIBASE1 is programmed to 0xF00011. The PCI Bus Transactor is set to request DMA burst transfer of 260 bytes to cross the PCISIZE0 (256-byte) boundary.

memsize3.s

This test verifies the PCI DMA access to the IAFX DRAM, using the PCI address space size 1MB, 2MB, 4MB, 8MB, 16MB, and 32MB. It also verifies the PCI DMA is accepted by the PCI slave interface for memory operation, based on the PCIBASE0-5 along with the size specified by the PCISIZE0-5 registers.

memsize4.s

This test verifies the PCI access to the IAFX DRAM, using the PCI address space size 64MB, 128MB, 256MB, 512MB, 1GB, and 256B. It also verifies the PCI DMA is accepted by the PCI slave interface for memory operation, based on the PCIBASE0-5 along with the size specified by the PCISIZE0-5 registers.

pci_abrtchk.s

This diagnostic checks the setting and clearing of bits in the status register and interrupt registers during the master abort and target abort PCI transactions with microSPARC-IIep as the master.

pci_comp_m1to4i.s

This diagnostic verifies master abort, target abort, target retry, and target disconnect for single data I/O transfers to fast, medium, slow, and subtractive slave targets. The transactions are from PCIC master to LMC PCI slave model.

pci_comp_m1to4m.s

This diagnostic verifies master abort, target abort, target retry, and target disconnect for single data configuration and memory transfers to fast, medium, slow, and subtractive slave targets. The transactions are from PCIC master to LMC PCI slave model.

pci_comp_m6to9.s

This diagnostic verifies master abort, target abort, target retry, and target disconnect for multidata phase configuration and memory transactions to fast, medium, slow, and subtractive slave targets. The transactions are from PCIC master to LMC PCI slave model.

pci_comp_m10m.s

This diagnostic verifies for the multidata phase memory transactions from PCIC master to LMC slave with different TRDY# latency. The transactions are from PCIC master to LMC PCI slave model.

pci_comp_m11.s

This diagnostic verifies when parity check enable bit is set in single and multidata phase configuration and memory transactions. It also checks for the read and clear of the bits related to the parity error in the PCIC status register. The transactions are from PCIC master to LMC PCI slave model.

pci_comp_m13.s

This diagnostic verifies that PCIC can terminate its transaction when its internal latency timer expires with GNT# inactive. Memory transactions from PCIC master to LMC slave model are performed.

pci_comp_s1to4.s

This diagnostic verifies the generation of SERR and PERR on the address and data parity error conditions. The direction of transfer is from the RaviCad PCI master to PCIC slave. This diagnostic checks for the parity error detection for I/O and memory data transfers.

pci_target.s

This diagnostic checks the response of the PCIC as a slave to intact cycle, special cycle, and special cycle with parity error. It also checks the response of PCIC for other commands like `write_invalidate`, `memory_readline`.

pcibars.s

`pcibars.s` tests the six sets of PCI to AFX translation registers.

pcibiend.s

`pcibiend.s` tests the biendianess of the PCI controller.

pcic_mbase1.s

This diagnostic verifies the reset values and the programmability of the PCI memory cycle translation registers (set 1). These registers are used to map the processor's 28-bit physical address (IAFX) bus into a 32-bit PCI physical address. The memory transactions with different settings on the translation registers is performed in this diagnostic.

pcic_pcis_io.s

This test performs an early check of some PCI functionality; performing I/O transfers. It writes the configuration data register of the target PCI device.

pcimisc.s

This diagnostic checks for the generation of a special cycle and interrupt acknowledge cycle. It also checks the status register and multidata phase PCI PIO memory transactions.

rascas.s

`rascas.s` tests `memif` corner cases. DMA master does rapid transaction across successive page boundaries to generate interesting cases for `memif`.

retry.s

- Tests PCIC master retry time-out counter — microSPARC-IIep as PCI master and the target gives microSPARC-IIep retry response more than PCIC retry counter value.
- Tests PCIC master TRDY_ time-out counter — microSPARC-IIep as PCI master and the target gives microSPARC-IIep TRDY_ wait states response more than PCIC TRDY_ time-out counter value.
- Tests disable microSPARC-IIep PCI arbiter, and using external PCI arbiter task is same as `falcon_pcim_bm.s` except for setting the disable pci arbiter bit.

slvranges.s

This test exercises all possible physical slave addresses on the PCI bus. It performs a store/load/compare to each of the physical slave addresses. It requires the use of the `smbar`, `iobar`, and fixed PCI spaces.

southbridge_pci.s

This diagnostic does multidata phase memory transactions from PCIC master to fast LMC slave model. In the run command, `-boot southbridge` is included.

standby.s

This test checks the standby feature as set by mid-register bit. It checks that the standby feature can be initiated and terminated via bit 11 of the mid-register. It also checks for proper wake-up from standby mode.

T_wrecks#.s

These diagnostics are just various iterations of TRASH tests.

testregs.s

This diagnostic checks the reset values and programmability of all the configuration and control registers of the PCIC.

trashpci.s

TRASH generated.

trashpci1.s

TRASH generated.

trash1.s

This test also tests Cache Line Wrap mode, Cache Line Toggle mode, and Reserved mode on memory write/read.

trash2.s

TRASH generated, nothing special.

twom.s

This test exercises arbitration between two pci masters doing byte operations. Two DMA masters compete for byte write-read-compares.

twoway.s

`twoway.s` tests PCI operations in both directions across the `pci` bridge at the same time. Runs the `falcon_pcim_bm` byte-mark test on the PIO side while a single DMA master does write-read-compare operations of various sizes.

twoway_ext.s

`twoway_ext.s` tests disabling of microSPARC-IIep PCI arbiter, and using external PCI arbiter task is same as `twoway.s` except for setting the disable PCI arbiter bit—that means microSPARC-IIep as PCI master does the PCI bus request; there are also two other PCI masters to request PCI bus, the three PCI master case.

type0_cfg.s

`typo0_cfg.s` tests type 0 PCI configuration read / write; generated all functional numbers, ad11--ad31 for idsel lines, and all base address registers access.

type1_cfg.s

`typo1_cfg.s` tests type 1 PCI configuration read / write; generated all bus / device numbers.

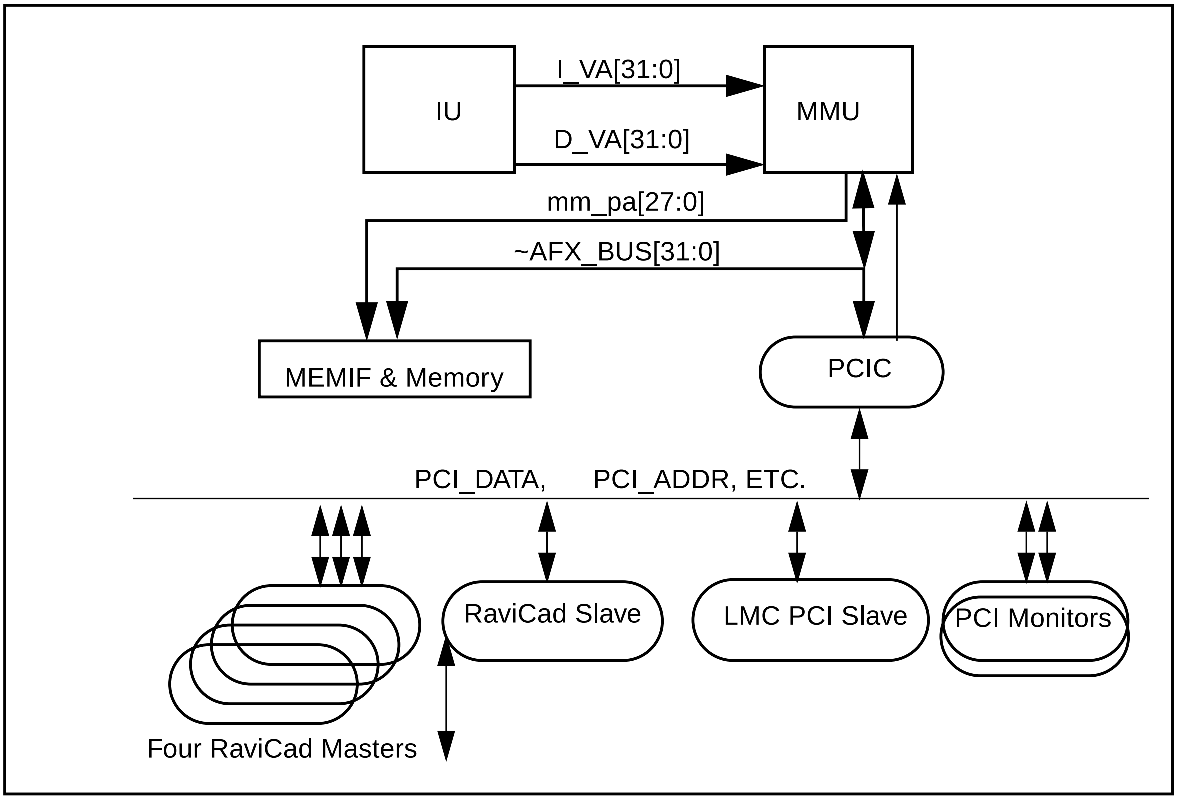

FIGURE 8-1 PCI Configuration

CHAPTER 9

Random Diagnostic Test

This chapter describes the diagnostic test randomly generated using either the thrash process. This diagnostic helps identify some corner cases in the design, MPSAS, and other tools. Most of the twister diagnostics are assembled using the `asm2ver` and random `io_act` mode.

dthrash

`dthrash` is a randomly generated diagnostic using the `dthrash` process from SuperSPARC-I. This diagnostic running in random `io_act` mode caused the design to hang.

Index