CONVEX Processor Diagnostics Manual (C200 Series)

Document No. 760-000550-203

Third Edition
November 1988

CONVEX Computer Corporation
Richardson, Texas USA

CONVEX Processor Diagnostics Manual
(C200 Series)
Order No. DHW-081
Third Edition

Printed in the United States of America

Revision Sheet

CONVEX Processor Diagnostics Manual
(C200 Series)

Edition	Document No.	Date	Description
Third	760-000550-203	October 1988	Third release. C230/C240 updates.
Second	760-000550-202	June 1988	Second release. Multiprocessor updates.
First	760-000550-201	April 1988	First release.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

spu2000 Service Processor Peripheral Test

spu4000 Service Processor Interface Test

pia4000 PIA Functional Test

pi2_4000 PI2 Functional Test

mem4000 Memory Subsystem Test

cpx4000 CPX Functional Test

cpu4010 Referenced and Modified Bits

cpu4030 Scalar Building Block Test

cpu4040 Vector Concurrency Tests

cpu4041 Vector Instruction Tests

cpu4131 Privileged Instructions and Architectural Features

cpu4231 C200 Series Privileged Instruction & Architectural Features

cpu4232 Enhanced, Non-vector, Uni-processor Instruction Tests

cpu4233 Multiprocessor Diagnostics

cpu4241 Enhanced Vector Instruction Tests

Appendixes

List of Tables

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

Preface

Purpose and Intended Audience

This manual documents the Service Processor based processor diagnostics for the C200 Series CONVEX computers and is intended to be the primary source of information on how to use these diagnostics. Test programs for the Service Processor, main memory, the Central Processing Unit (CPU), the CPU Utilities Card(s) (CPX or CUE/CUO), and the Peripheral Interface Adapter (PIA or PI2) are described in this manual. I/O and peripheral test programs are documented in the *CONVEX PBUS I/O System Diagnostics Manual.*

This manual is not a tutorial, but rather a reference for Field Service and Manufacturing Test personnel, as well as CONVEX customers that perform their own system maintenance.

Scope

This manual applies to the C200 Series CONVEX computers.

Outline

Each chapter in this manual covers a specific diagnostic function. To identify the general contents of each chapter, prefixes appear before the page number. Test descriptions and the invocation procedures of each test are covered within each chapter. The organization is as follows:

- **Diagnostics Environment** — Contains an introduction to the theories and concepts that underlie processor diagnostics as well as the basic overview, philosophy, and structure of processor diagnostics
- **Dshell and Iscan Overview** — Provides a brief overview of and a general introduction to the *dshell* and *iscan* utilities
- **SPU** — Describes the Service Processor specific diagnostic tests
- **PIA** and **PI2** — Describes the Peripheral Interface Adapter (PIA or PI2)-specific diagnostic tests
- **Mem** — Describes the memory system-specific diagnostic tests
- **CPX** — Describes the CPU Utilities card(s) (CPX or CUE/CUO)-specific diagnostic tests
- **CPU** — Describes the processor-specific diagnostic tests
- **Appendix A** — Contains error messages that are common to all CPU-based diagnostic tests and errno messages explanations
- **Appendix B** — Contains common error messages that are associated with SPU UNIX

- **Appendix C** — Contains an explanation of the standard register dump display screen
- **Appendix D** — Contains a listing of machine opcodes in alphabetical order
- **Appendix E** — Contains a glossary of term used withing this manual
- **Appendix F** — Contains information concerning how to use the *contact* facility to report problems

Notational Conventions

The notational conventions used in this text are listed below:

- A *bit* is a single binary value or entity
- A *nibble* is 4 bits
- A *byte* is 8 bits
- A *halfword* is 16 bits
- A *word* is 32 bits
- A *longword* is 64 bits
- An *instruction* is a multi-halfword operand
- A bit is *set* when it contains a binary value of 1
- A bit is *clear* when it contains a binary value of 0
- All memory and I/O addresses are written in hexadecimal notation unless explicitly stated otherwise
- All register contents are written in hexadecimal notation unless explicitly stated otherwise
- A *register* is a programmer-visible hardware storage element internal to the processor
- *Physical memory* is the physical storage installed in the processor
- *Virtual memory* is the perceived amount of main memory as seen by the application programmer
- The symbol *K* is an abbreviation for *kilo* or 1,024
- The symbol *M* is an abbreviation for *mega* or 1,048,576
- The symbol *G* is an abbreviation for *giga* or 1,073,741,824
- TBD is an abbreviation for *To Be Determined*
- **Boldface** is user entered data

Warnings

The following is an example of a warning and its typical content and location as used in CONVEX documents:

WARNING

Warnings highlight procedures or information necessary to avoid injury to personnel. A warning immediately precedes the critical information and includes a description of the hazard.

Cautions

The following is an example of a caution and its typical content and location as used in CONVEX documents:

CAUTION

Cautions highlight procedures or information necessary to avoid damage to equipment, loss of data, or invalid test results. A caution immediately precedes the critical information and includes a description of the possible damage.

Notes

The following is an example of a note and its typical content and location as used in CONVEX documents:

NOTE

A note highlights useful information that is supplemental in nature. A note may immediately precede or follow the information that is being highlighted.

Test Reference Structure

The following guidelines outline the layout of the diagnostic tests:

To identify the general contents of each test, prefixes appear before the page number. The following prefixes identify the contents of the test they name:

TEST	
Prefix	**General Contents**
spu(nnnn)	Service Processor subsystem tests
pia(nnnn)	Peripheral Interface Adapter
pi2_(nnnn)	Peripheral Interface Adapter 2
mem(nnnn)	Memory subsystem tests
cpx(nnnn)	CPU Utility Card(s)
cpu(nnnn)	CPU subsystem tests
io(nnnn)	I/O subsystem tests
dev(nnnn)	Peripheral device tests

For example, information for specific CPU tests will be found on pages prefixed with *cpu*.

Tests are identified by a prefix consisting of three characters (identifying the subsystem) and four numbers (identifies the test). Tests are grouped according to subsystem, with tests appearing in numerical order, i.e., *cpu4000*, *cpu4010*, *cpu4030*, etc. For instance, to find test *cpu4040*, first find the section marked CPU. Then locate the pages prefixed with *cpu4040*.

Associated Documents

Readers should become familiar with both the glossary of technical terms and the technical notation conventions listed in the preface. A feedback form is found in the rear of this handbook, and readers are invited to comment on the service and clarity of this text.

Other related topics are detailed in:

- *CONVEX SPU UNIX Utilities Manual*, Order No. DHW-007
- *CONVEX PBUS I/O System Diagnostics Manual*, Order No. DHW-008
- *CONVEX Processor Operation Guide (C100 Series, C200 Series)*, Order No. DHW-015
- *CONVEX Diagnostic Utilities Manual (C1, C120)*, Order No. DHW-072
- *CONVEX Diagnostic Utilities Manual (C200 Series)*, Order No. DHW-082
- *CONVEX Architecture Reference*, Order No. DHW-005
- *CONVEX UNIX Tutorial Papers*, Order No. DSW-002
- *The C Programming Language*, Kernighan & Ritchie, Order No. DSW-046

Ordering Documentation

To obtain the most current version of any associated documentation, order using the product number. If the product number is not known, order by the exact title. In some situations the most current version is not desired. In order to receive a specific version of a manual, the manual must be ordered by a 12-digit document, or part, number, which can be provided by CONVEX.

The product number for this manual is DHW-081.

The document number for this manual is 760-000550-203.

CONVEX documents can be ordered by mail by sending a request to:

CONVEX Computer Corporation
Customer Service
PO Box 833851
Richardson TX 75083-3851 USA

Hardware and Software Support

Hardware and software support can be obtained through the CONVEX Technical Assistance Center (TAC). The TAC can be reached in Texas by calling (214)952-0379, or by calling 1(800)952-0379 from other locations in the continental United States. Customers outside the United States should contact their local CONVEX office.

Electronic Mail

The Hardware Documentation Group has an email address for documentation comments. Use this service to give us a quick response mechanism if you have special documentation questions that you would like addressed immediately. If you have a technical question, you should still contact the Technical Assistance Center, as described previously. To use email response service, just send mail addressed to:

```
cnvxhwdoc@convex.COM
```

We will read your comments and give you a personal reply.

What to Include in an Email Message

When you use the electronic mail service, please provide the following information:

- The reader's name and company name
- A return email address in INTERNET notation or UUCP (bang) notation
- The manual that is being critiqued
- The chapter and page number in question
- The comment
- Indicate if a personal response is required

Reader's Forum

If you wish to mail your comments to us, please use the form at the end of this manual and list the document page number with your questions and comments. Thank you.

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgments

I would like to thank the following people for their contributions to this document:

- Technical contributors: Brian Allison, Tom Ford, Steve Gardner, Tony Jones, Dave Rotheroe, Jeff Venters
- Document review team: Art Clark, John Clark, Don Davis, Ron Engleking, Steve Fieler, Art Fischman, Rick Miller, Craig Reed
- Hardware Documentation staff: Larry Bonura, Bruce Evans, Jimmie Holman, Barbara Morris, Randall Stiles, Louis Tallant

Without the efforts of all the aforementioned, this document would not have been possible.

David Massey, Lead Writer
CONVEX Hardware Documentation

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 1
Diagnostics Environment

1.1 Overview

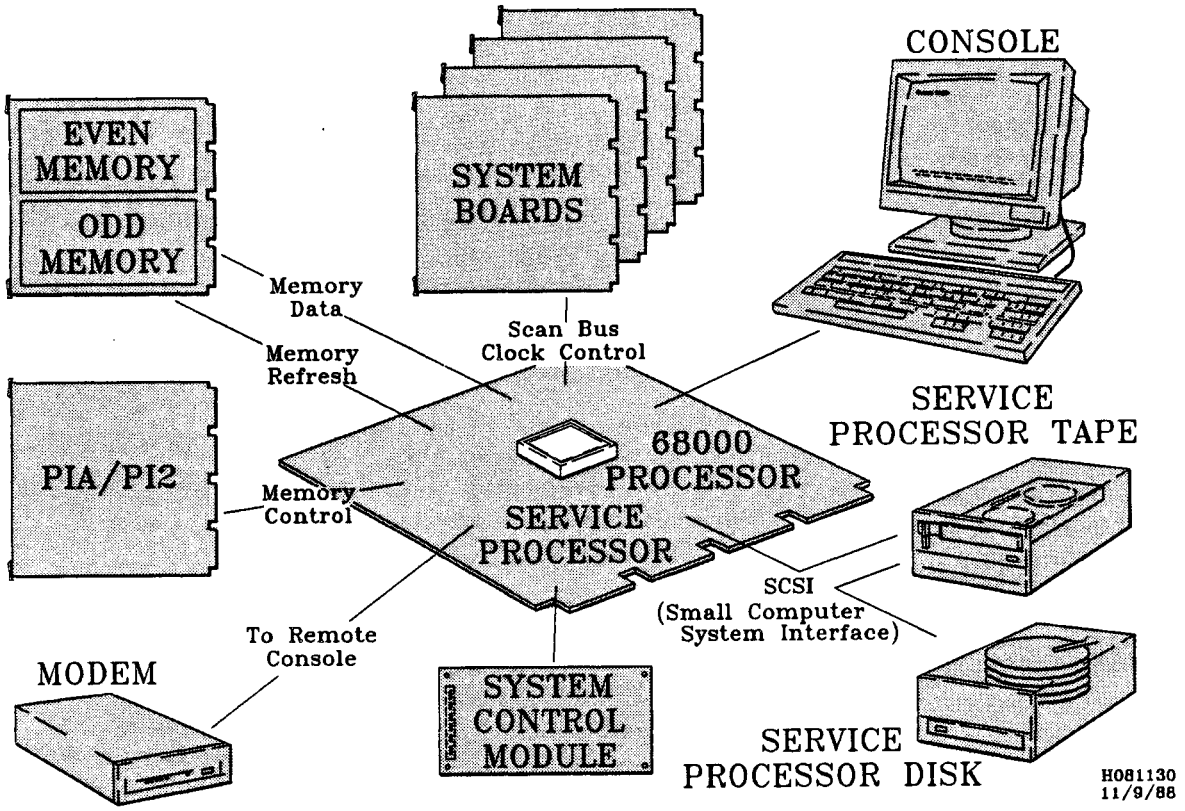
CONVEX system diagnostics consist of a suite of test programs designed (except where noted) to execute under the Service Processor operating system, SPU UNIX. These programs use the capabilities of the Service Processor to test the operation of one or more of the system functions and report any errors detected. All diagnostics in this manual are intended to be executed "off-line," that is, while CONVEX UNIX is not being executed by any of the Central Processing Units (CPUs) in the system.

The Service Processor, together with SPU UNIX and the various diagnostic utilities and test programs, comprise the CONVEX diagnostic environment. This chapter describes the hardware and software components of this environment and provides the background necessary to fully utilize the capabilities of the CONVEX processor diagnostics.

1.2 Hardware Overview

Each CONVEX computer system contains built-in test hardware designed for system troubleshooting. The test hardware consists of a single board microcomputer called the Service Processor and a number of serial scan buses that permit the Service Processor to access the internal state of the system. The basic features of the Service Processor hardware are shown in the following figure:

Figure 1–1, Basic Hardware Features



1.2.1 Service Processor Unit

The Service Processor Unit for the C200 Series systems is named SP2 for a Complex with one or two CPUs and SP4 for a Complex with three or four CPUs. The SP2/SP4 is a single-board microcomputer capable of operating on its own with minimum support from any other boards on the system. The SP2/SP4 consists of a Motorola 68000 microprocessor, 64 Kbytes of Erasable Programmable Read-Only Memory (EPROM), 2 Mbytes of Random Access Memory (RAM), two asynchronous serial ports, and a Small Computer System Interface (SCSI) bus. One of the asynchronous serial ports supports the system console and the other is used to support remote diagnostics (execution of diagnostics from a remote site). The SP2/SP4 has a 170-Mbyte, 5.25-inch Winchester disk and a 125-Mbyte, QIC-120 format streaming cartridge tape drive that are connected through the SCSI bus.

The SP2/SP4 controls all the clocks within the C200 Series systems. It does this by controlling the run signals that are sent from the SP2/SP4 to every board in the system. In addition to clock control, the SP2/SP4 can also interact with the other boards in the system by one of several system-wide buses. The SP2/SP4 communicates with the rest of the system through three main bus structures: the EBUS, the interrupt bus, and the scan bus.

1.2.2 EBUS

The EBUS connects the Service Processor to the memory system. The Service Processor has the ability to do a variety of memory operations on the main memory system. The Service Processor has a set of map registers that map the portions of the Service Processor logical address space to access main memory. There are enough of these map registers to allow the service processor access to 4 Mbytes of memory at one time.

1.2.3 Interrupt Bus

The interrupt bus communicates with each Central Processing Unit (CPU) and Channel Control Unit (CCU) within the system. There are 256 interrupts available within the system. The Service Processor has the ability to send all 256 interrupts and to receive interrupts 8–15. When the service processor sends an interrupt, any subsystem listening for that interrupt will receive it. Interrupts are used extensively during the diagnostics for communication between the Service Processor and the other processors within the system.

1.2.4 Scan Bus

The scan bus refers to a set of serial buses connected to each board in the system. Each system board has a *scan ring*, a series of latches connected together with the ability to operate in either a parallel or serial load mode. The scan bus permits the Service Processor to access and initialize the internal state of a system.

In the diagnostic mode, these rings operate in the serial load mode and appear to the Service Processor as a large shift register. The service processor controls these scan rings and can read and write them. The Service Processor can load and examine the state of any board in the system by writing and reading the scan rings. Scan rings are used extensively during system initialization for board initialization and control store loading, and are used by test programs and diagnostic utilities for internal state monitoring and manipulation.

1.2.5 Board Identification

Each board in the CPU chassis contains a serial Electrically-Erasable Programmable Read-Only Memory (EEPROM) that is accessible by the Service Processor via the scan bus. The EEPROM, frequently referred to as a COP chip, stores a variety of board identification information including part number, fabrication revision, and assembly revision. This information is used by the Service Processor to identify the scan ring configuration of a given board and to determine which boards are present in the CPU chassis.

1.2.6 Remote Diagnostics Port

In order to minimize hardware troubleshooting and problem isolation time, remote diagnostics capability is designed into the Service Processor. With a modem to the second asynchronous port on the Service Processor, it is possible for CONVEX Technical Assistance Center (TAC) personnel to execute diagnostics remotely. In this manner, it is frequently possible to identify the failing Field Replaceable Unit (FRU) before dispatching field service personnel to the customer site.

1.3 Software Overview

Software in the diagnostic environment is hierarchically structured as shown in the following table:

Table 1–1, Hierarchical Software Structure

LEVEL	DESCRIPTION
4	Test Program
3	Diagnostic Utilities and Diagnostic Shell
2	SPU UNIX
1	Soft Front Panel

At the lowest level of the hierarchy is the *soft front panel*. This is an EPROM-based program that permits specification of a variety of parameters related to system booting and operation. By entering the appropriate command while in the soft front panel, it is possible to boot the SPU operating system, SPU UNIX.

SPU UNIX is based on UNIX Version 7. Although more primitive than CONVEX UNIX, the user interface is recognizable to anyone familiar with the UNIX Bourne shell. SPU UNIX manages the SPU memory and peripheral resources, and provides the environment for the execution of system diagnostics.

The next level of the hierarchy consists of the diagnostic utilities. These utilities are a collection of programs developed to perform a variety of initialization and system monitoring tasks. Included in this list of programs are control store loaders, memory initialization and display tools, error loggers, and a diagnostic test executive called the Diagnostic Shell (*dshell*). The *dshell* is a CONVEX-developed command language interpreter specifically designed for controlling test program execution and error reporting. The *dshell* permits the user to specify portions of the test programs to be executed, the level of detail in the error report, and what action to take on the occurrence of an error (e.g., log multiple failures, pause on failure, loop on subtest, etc.).

The top level of the hierarchy is the test program itself. Each test program is designed to verify specific functions of the computer system or of a particular subsystem. The test program uses the diagnostic features of the Service Processor to control system clocking, load object modules into main memory or Channel Control Unit (CCU) memory, initiate execution of code by the CPU, and monitor the results.

Each component of the software environment hierarchy is discussed in more detail in the following sections.

1.4 Soft Front Panel

The soft front panel monitor can only be entered when the keyswitch is turned to the **LOCAL MAINTENANCE** position. From the soft front panel monitor, the user can select various modes of operation (such as diagnostic or the normal operating system), default boot device, auto reboot, and other flags from the prompt.

The soft front panel can be entered in three basic ways:

1. When the system is powered up.
2. When the **RESET** button is pushed.
3. When the */etc/reboot* command is executed from SPU UNIX.

Under normal operating procedures, the soft front panel monitor is bypassed by starting the system in the **SECURE EXECUTION** mode. But, when the keyswitch is turned to the **LOCAL MAINTENANCE** position, the soft front panel monitor is activated and becomes an interactive program that executes at the local system console. The soft front panel monitor can be used to examine and modify the settings of the switches used to configure the system and to invoke the SPU UNIX bootstrap program.

For more information on starting up and shutting down the system, refer to the *CONVEX System Manager's Guide* and the *CONVEX Processor Operation Guide (C100 Series, C200 Series)*.

The diagnostic environment begins at the soft front panel, which refers to the contents of an Electrically-Erasable Programmable Read-Only Memory (EEPROM) that is contained on the Service Processor. The contents of this EEPROM are modified interactively through the soft front panel code that is executed from the Service Processor Erasable Programmable Read-Only Memory (EPROM) whenever the system is powered up. Since the Service Processor stores the settings of the firmware switches in nonvolatile memory, they are preserved even when the system is powered off. When the monitor starts, it displays the current settings of the firmware switches and then prints its prompt, `(fp)>`. The soft front panel display is shown in the following figure:

Figure 1–2, Soft Front Panel Display, Example

```
123456789ABCDE
Convex Front Panel / Module Rev: 1.23, Version: 1 / CPU SN 17
mode-of-operation = normal-os            boot-device = disk
location-of-bootstrap = default          power-up-reboot = enable
automatic-reboot = enable                spu-selftest = enable
os-flags = 0                             remote-port-bps = 1200
(fp)>
```

1.4.1 Soft Front Panel Commands

Entering **help** in response to the `(fp)>` prompt displays a list of the soft front panel monitor commands. Commands available include the following:

> **NOTE**
>
> Usually it is necessary only to type the first letter of a command, or only enough of the command to uniquely identify it. For example, entering **s m=** works just as well as entering **set mode of operation=**.

1. **s**[*et*]

The *set* command sets a series of software switches to modify the front panel monitor operation. Available options include:

- **s**[*et*] **m**[*ode of operation*] =[*n*[*ormal-os*] | *d*[*iagnostic*] | a[*lternate-os*]]

  This command controls the level of operation the system achieves at start-up. The options available are:

  - *normal_os* — Starts the system with the standard version of multi-user CONVEX UNIX operation system enabled.
  - *alternate_os* — System prompts for special start-up modes, then boots the SPU according to those instructions.
  - *diagnostic* — System boots SPU UNIX operating system or an alternate diagnostic mode at start-up.

- **s**[*et*] **b**[*oot-device*] = [*d*[*isk*] | *t*[ape] | *i*[*omega*] | *o*[*ther2*]]

  This command selects the location of the boot device prior to booting the SPU. If *t*[*ape*] is selected, make sure to mount a copy of the SPU UNIX boot image tape in the tape unit before trying to boot the system. The *o*[*ther2*] option is currently undefined.

- **s**[*et*] **l**[*ocation-of-bootstrap*] = [*d*[*efault*] | *1*[*-copy*] | *2*[*-copy*] | *3*[*-copy*]]

  This command selects one of four redundant copies of the boot program used by the system. If errors are discovered in the default boot program, this command can be used.

- **s**[*et*] **p**[*ower-up-reboot*] = [*d*[*isable*] | [*e*[*nable*]]

  Enabling this switch allows the system to reboot to timesharing mode after any type of interruption in the power supplied to the system.

- **s**[*et*] **a**[*utomatic-reboot*] = [*d*[*isable*] | [*e*[*nable*]]

  This command permits rebooting in **SECURE EXECUTION** mode by entering the command sequence

  ```
  (spu)>/etc/reboot
  ```

  This command will not stop at the soft front panel prompt.

- **s**[*et*] **s**[*pu-selftest*] = [*d*[*isable*] | [*e*[*nable*]]

  This command enables the Service Processor self-test, *spu1000*. The self-test is an EPROM-based functional test that verifies operation of that portion of the Service Processor (SP2/SP4) necessary to execute SPU UNIX. This test

does not involve any hardware other than the service processor and its associated peripherals. The *spu1000* specifically tests the:

- 68000 microprocessor and its memory
- Service Processor logic, containing the timer chip and the two Universal Asynchronous Receiver-Transmitters (UARTs)
- boot devices, including the disk controller and cartridge tape controller
- Direct Memory Access Controller (DMAC)
- Small Computer System Interface (SCSI)
- Service Processor Real Time Clock registers.

The self-test typically fails only if there has been a hardware failure of some kind. Do not disable the Service Processor self-test with soft front panel monitor commands; the Service Processor self-test takes about a minute to run important diagnostic functions.

- **s**[*et*] **o**[*s-options*] = *number*

  This command is reserved for CONVEX use. It has no application in released software.

2. **p**[*reset*] [*number* | *s*[*tandard*] | *d*[*iagnostic*] | *i*[*nstall*]]

This command allows switches to be set in groups rather than individually, as is done with *set*. Each of the available modes (*standard*, *alternate*, *diagnostic*, and *install*) represents a default configuration for a set of soft front panel monitor switches. The *number* field is reserved for diagnostic use. Do not change the value of this field without instructions from the Technical Assistance Center. The following table summarizes the settings of the four *preset* modes:

Table 1–2, Switch Settings in the *preset* Mode

Preset Mode	Self-Test Enables?	Power-Up Reboot Enabled?	Auto-Boot Enabled?	Boot Device	Boot Mode
standard	Yes	Yes	Yes	Disk	*normal-os*
alternate	Yes	Yes	Yes	Disk	*alternate-os*
diagnostic	Yes	No	No	Disk	*diagnostic*
install	Yes	No	No	Tape	*install*

3. **b**[*oot*]

Terminates the soft front panel monitor and boots the SPU UNIX operating system.

4. **d**[*isplay*]

Displays the state of all soft front panel switches and settings.

5. **de**[*bug*]

Enables a debugger useful primarily to CONVEX Field Engineers and Manufacturing. The following table lists each *debug* command and a description of its use:

Table 1–3, *debug* Commands

COMMAND	DESCRIPTION
c	Used for real time clock calibration
m addr [*addr*]	Used to modify or dump memory data bytes (8 bit)
mw addr [*addr*]	Used to modify or dump memory data short integers (16 bit)
ml addr [*addr*]	Used to modify or dump memory data integers (32 bit)
q	Used to quit the *debug* utility

6. **h**[*elp*]

   Prints a one-page listing of available soft front panel monitor commands.

Once the soft front panel monitor begins, it prints a message similar to that shown in Figure 1–2, Soft Front Panel Display, Example. Each of the hexadecimal digits (123456789ABCDE) in the first line of the figure represents successive phases in the SPU self-test procedure. If the SPU self-test completes properly, the other lines shown in the figure are displayed on the system console, along with the soft front panel prompt, `(fp)>`.

The SPU self-test takes about a minute to complete and fails only if a hardware failure is detected. If a failure is detected during the self-test, the booting processes will halt and an error message may be displayed on the system console.

The last hexadecimal digit displayed on the system console indicates the self-test subtest that detected the error. For example, if the SPU self-test failed at Subtest 6 of the test, the system console would display `123456`, possibly followed by an error message. Refer to *spu1000* in this manual for more information on the SPU self-test.

1.5 SPU UNIX Operating System

The Service Processor Unit (named SP2 for a Complex with one or two CPUs or SP4 for a Complex with three or four CPUs) runs an operating system based on UNIX Version 7. Although more primitive than CONVEX UNIX, the user interface is recognizable to anyone familiar with the UNIX Bourne shell. Version 7 is a nondemand page version of UNIX, which means that an entire program must be resident in main memory before program execution can begin. If there is not enough memory for a program, the current program in memory will have to be swapped out. This swapping action can have disastrous effects on Service Processor performance. Because of this, be careful about trying to run many processes at once on the Service Processor. Specifically, avoid using the ampersand (&) extension for running background processes if possible.

Along with this version of UNIX comes many standard UNIX utilities such as *more*, *ls*, *cat*, etc. These utilities are found in the */bin* directory on the Service Processor. These UNIX utilities are described in the *CONVEX SPU UNIX Utilities Manual*.

1.6 Diagnostic Utilities

There are a variety of diagnostic utilities with many different uses. Many utilities are used for hardware initialization of various parts of the system; other utilities are meant for interactive use in manipulating hardware state within the system. Diagnostic utilities are generally located in directory */mnt/bin*.

Most of the utilities are meant to be run from the Bourne shell prompt in an interactive mode. Another set of utilities, known as the *error loggers*, are meant to be executed in background on the Service Processor while CONVEX UNIX is booted. The error logger utilities run on the service processor and monitor the various error sources within the CPU while the operating system is running.

1.6.1 Interactive Diagnostic Utilities

Several utility programs can be used to support C200 Series computer processes: cache loading and verifying, memory tools, system initialization, and hardware state tools. See the following table for available Service Processor diagnostic operation utilities and a description of their purposes:

Table 1–4, Available Diagnostic Utility Commands

DIAGNOSTIC UTILITY COMMANDS		
Category	**Command**	**Purpose**
Control store loaders	*cs*	Loads writable control store(s)
	dcache	Dumps the data cache
	icache	Loads, verifies, and dumps the instruction cache
	pte_cache	Dumps the PTE cache
Memory tools	*map*	Displays logical-to-physical mapping of main memory
	mm	Displays or modifies main memory
	mminit	Initializes main memory
	memld	Loads object file into system memory
System initialization	*boot_iop*	Program to cold boot an IOP or VIOP and download an object file to it
	boot_hsp	Program to cold boot an HSP and download an object file to it
	diaginit	Initializes diagnostic description files
	initall	Initializes control stores and main memory
	margin	Sets or reads power supply and system clock margins
	mkdiag_db	Maintain diagnostics configuration file
	scn_util	Hardware initialization utility
	scnlink	Intermediate scan ring definition file linker
	sysreset	Resets the computer system
	security_clear	Memory and cache purge
Hardware utilities	*ioputil*	Displays or modifies IOP memory or register locations
	cop	Displays board identification information found in COP chips
	cpureg	Initialize or display the CPU nonvector register state
	cpuvreg	Initialize or display the CPU vector register state
	dshell	System diagnostic test executive
	get_defects	Read manufacturer's defect map from an SMD drive
	hard_logger	Hard error logger
	hsputil	HSP register/memory utility
	iscn	Interactive Scan facility
	pup	Power-up bit read/write utility
	scn_ring	Interactive scan ring read/write/check utility
	sfpread	Read/modify the SPU soft front panel switches
	mm_sniff	Performs constant main memory error detection
	sp2util	Displays or modifies SP2/SP4 memory or register locations
	syshalt	Immediately halts the system on a subsystem basis
	vioputil	VIOP register and memory utility
	x	Hexadecimal/decimal calculator

1.6.2 Error Logging Utilities

While CONVEX UNIX is booted, there are two diagnostic utilities that are being executed on the Service Processor. These utilities detect and report hardware and environmental errors that occur during the operation of the system. These utilities are *errintd* and *mm_sniff*.

1.6.2.1 What *errintd* Does

The *errintd* utility is the error interrupt daemon. Its function is to monitor the various hardware and environmental errors and to invoke the correct routine to report the errors. The *errintd* utility interacts with the *prtlog* utility to print the messages to the console and into the error log file (*/mnt/errlog*).

1.6.2.2 What *mm_sniff* Does

The *mm_sniff* utility periodically reads through main memory looking for the occurrence of single-bit errors. It traverses all physical memory within a programmable time period. When *mm_sniff* detects a single-bit error, a scrub cycle is performed, which corrects the bit. It is important that single-bit errors are detected before they become double-bit errors. A single-bit error can be corrected by the error correcting circuitry on the Memory Control Module (MCM) whereas a double-bit error cannot.

When *mm_sniff* comes across a memory address that contains a single-bit error, *errintd* detects the error and decodes it.

1.6.2.3 Error Monitoring

During operation of the processor, the Service Processor monitors for error conditions that might need to be put into the error log. There are three types of errors monitored by the Service Processor:

- **Environmental errors** — These errors, detected by the System Control Monitor (SCM), relate to the physical environment of the machine, e.g., airflow, power supply, fans, etc. Depending on how critical the particular problem is, the error reported may be hard or soft.
- **Soft errors** — These errors will not bring the machine to a halt. Often they may be recoverable, as in the case of a soft memory error, where the MCM will correct a single-bit error.
- **Hard errors** — These are fatal errors that will stop the computer immediately. Hard errors may be remedied using the diagnostic tests described in this manual, however, some may be transient.

Each of the three error types has an error logger. Error conditions include: memory soft (single-bit) errors, memory hard (double-bit) errors, microstore parity errors, PTE cache parity errors, bus parity errors, and others. When the Service Processor detects one of these errors, *errintd* is activated which invokes the applicable error logger and activates the appropriate program to report the error.

After the system is booted, the Service Processor runs the *prtlog* (print log) program. The */mnt/os/prtlog* monitors all print requests generated by software running on the CCUs or CPUs as well as *errintd* print requests. Should a board send a message of importance that it wants to write to the console, *prtlog* detects the message, appends it to the error log, and prints it to the console. When the Service Processor hardware detects a hard, soft, or environmental error, the Service Processor signals the sleeping *errintd*, which manages these error interrupts. The *errintd* program (the error interrupt daemon) includes the interrupt service functions for soft, hard, and environmental errors.

When single-bit memory errors (soft errors) are detected by the hardware, the Service Processor enters the device and address (with the help of the Error Detection and Correction Code (EDC)) in */mnt/errlog*. If a particular RAM device starts having multiple failures, it becomes immediately apparent.

The soft error log file is named */mnt/softlog* and is written only in printable ASCII characters. The logger decodes the failing address (uninterleaved) and updates the soft error log file. See the following figure for a sample soft error log layout:

Figure 1–3, Soft Memory Error Log Layout

```
Log started: Sat Apr 19 15:35:20 1986
Log full: NO
Failed devices:     6              Total Failures: 2389884
M           MCM     B  B T
C          SERIAL   I  U /
M DEVICE   NUMBER   T  S S     FIRST FAIL       LAST FAIL        ADDR    #FAILS
- ------- --------- -- - - ----------------- ----------------- -------- ------
LATEST fail:
0 Za9030  xxxxxxxxx nn P S Aug  2 16:49 1986 Jan 17 08:36 1986 xxxxxxxx 999999
OLDEST fail:
0 Ua9040  xxxxxxxxx nn P T Apr 26 20:14 1986 Apr 26 20:14 1986 xxxxxxxx 000001
- ------  --------- -- - - ----------------- ----------------- -------- ------
1 Zb7090  xxxxxxxxx nn M T May  7 15:13 1986 May  7 15:13 1986 xxxxxxxx 000001
0 Za7030  xxxxxxxxx nn P S Aug  2 16:49 1986 Jan 17 08:36 1986 xxxxxxxx 999999
7 Zd4080  xxxxxxxxx nn P T Aug  5 14:32 1986 Aug  5 14:32 1986 xxxxxxxx 000001
0 Ua2070  xxxxxxxxx nn M T Aug 13 07:55 1986 Aug 13 07:55 1986 xxxxxxxx 000001
1 Zc9040  xxxxxxxxx nn M T Sep 28 16:02 1986 Sep 28 16:02 1986 xxxxxxxx 000001
```

When the hard error occurs, the subsystem experiencing the error asserts its hard error line and the processor is halted. The *errintd* starts the *hard_logger*. All *errintd* log messages are written to standard output that is piped through *prtlog*, which sends the message to */mnt/errlog* and the console.

Figure 1–4, Sample */mnt/errlog* Layout

```
[CCU7@19:11:22] Data overrun error
[CCU7@19:11:22] IOP 7 Multibus 1 CSR address ffd3c0 Port 0xd
[CPU_@10:04:52] NFS getattr failed for server rigel: RPC: Timed out
[CPU_@10:14:09] NFS getattr failed for server rigel: RPC: Timed out
[SPU_@14:09:57] Single bit EDC error
[SPU_@14:09:57]  MCU Master: CH7  -  MAU: 0   Device: L9-026 Address: 00ccfa18
  Syndrome: 54   Single bit EDC error
[SPU_@14:09:58] Logger unable to clear soft error interrupt
```

1.7 Diagnostic Test Programs

The C200 Series system diagnostics contain a functional test for every subsystem contained in the system. In addition, there are tests for each I/O controller and device that CONVEX supports. These system diagnostics are functional diagnostics. As such they indicate if a particular functional unit is operational and, in general, give no indication of the exact cause of failure (when they are not operational). Diagnostic test programs are contained in the directory */mnt/test*.

Diagnostic tests are executed under the Diagnostic Shell (*dshell*). Each test is divided into two logical groupings: subtests and classes. In general, *subtests* test a specific function in the subsystem being tested. A *class* is a group of related subtests. The division between classes and subtests varies greatly in the different functional tests.

1.7.1 Test Strategy

Functional tests verify the functional integrity of an entire system or a particular subsystem. Specifically, these tests verify the functionality of the Service Processor, the CPU, main memory, CCUs, I/O, and peripherals. For example, there is a CPU functional test that verifies proper instruction set execution. And there is a tape functional test that verifies the proper operation of a tape drive and its controller. Although the functional tests do not isolate failures to a specific board, the main memory functional test isolates a data pattern failure to the failing RAM.

All tests are designed to test from the least complex functions to the most complex functions. Likewise, the tests themselves should be run in an order in which the most basic system functions are tested first.

The following table shows the order in which diagnostic tests should be run to verify a system from the ground up—from the most basic function to the entire system:

Table 1–5, Suggested Order of Test Execution

SUGGESTED ORDER OF TEST EXECUTION		
Order	**Program Name**	**Test Name**
1	*spu1000*	Service Processor EPROM-Based Self-Test
2	*spu2000*	SPU Peripheral Test
3	*spu4000*	Service Processor Interface Test
4	*mem4000*	Memory Subsystem Test
5	*pia4000* *or pi2_4000*	PIA Functional Test PI2 Functional Test
6	*cpx4000*	CPX Functional Test
7	*cpu4030*	Scalar Building Block Test
8	*cpu4232*	Enhanced, Non-vector, Uni-processor Instruction Tests
9	*cpu4131*	Privileged Instructions and Architectural Features
10	*cpu4231*	C200 Series Privileged Instructions and Architectural Features
11	*cpu4010*	Reference and Modified Bits
12	*cpu4041*	Vector Instruction Tests
13	*cpu4241*	Enhanced Vector Instruction Tests
14	*cpu4233*	Multiprocessor Diagnostics
15	*cpu4040*	Vector Concurrency Tests
A	*io4000*	Multibus I/O Subsystem Test
A	*io4120*	HSP/HIA Subsystem Tests
A	*io5000*	VMEbus I/O Processor Test
B	*dev4100*	Multibus SMD Disk Test
B	*dev4110*	Multibus SMD Disk Formatter and Interactive Test
B	*dev4200*	Multibus STC Tape Unit Test
B	*dev4300*	Multibus Terminal Controller Test
B	*dev4400*	Multibus Line Printer Test
B	*dev4410*	Multibus Plotter Functional Test
B	*dev4500*	Multibus Ethernet Controller Test
B	*dev4510*	Multibus HYPERchannel Controller Test
B	*dev4600*	Multibus Emulator Controller Test
B	*dev5130*	VMEbus SMD/ESDI Disk Test and Formatter
B	*dev5500*	VMEbus Ethernet Controller Test

The tests are listed numerically in the order in which they should be executed. The device- and I/O-based tests are listed alphabetically in their suggested order of execution.

> **NOTE**
>
> The *cpu4131* test is applicable only for C1 and C120 machines. The *cpu4231* test is applicable for any C200 Series machine. The device and I/O tests should be selected depending on the components within the system under test. Chose a test from the A group, depending on what type of CCU is to be tested. Then choose a test from the B group, depending on the controller/device to be tested.

1.7.2 Test Structure

There are three basic test categories:

- PBUS I/O tests (e.g., *io4120, dev4110,* etc.)
- CPU tests (e.g., *cpu4030, cpu4010,* etc.)
- Scan-based tests (e.g., *pia4000, cpx4000,* etc.)

The PBUS I/O tests execute as a result of separate modules of test code running on a CCU, the controller under test, and the Service Processor. The CPU tests execute as a result of separate modules of test code running on both the CPU and the Service Processor. And, the scan-based tests only execute as a result of test code running on the Service Processor. The PBUS I/O and CPU tests both use an interprocessor communication structure that incorporates main memory and system interrupts to coordinate the separate modules of test code. Since scan-based tests only execute test code on the Service Processor, they do not need the interprocessor communication structure.

1.7.3 Test Program Naming Conventions

Test program names are in the form *cattypedevnn.suffix* where:

- *cat* is the subsystem being tested
- *type* is the type of test being performed, i.e, standalone, self-test, or offline functional test
- *device* is the device being tested, i.e., disk, tape, or printer. This segment of the test program name is used *only* if the category is a device.
- *nn* is a CONVEX code used for distinguishing between test programs
- *suffix* is one of three program identifiers:
  - All *.t* programs that execute on Service Processor
  - *x00* and *rnn* are object files for different target processors other than the Service Processor. The target processor depends on the subject of the test and it must have the test program category prior to it to determine what it is.

1.7.3.1 Test Program Categories

Test program categories include those tests for the CPU, peripheral devices, I/O system, memory system, Service Processor, and entire system. For example, *cpu4041* is a CPU vector instruction test while *mem4000* is a memory system functional test. The following table lists test program categories:

Table 1–6, Test Program Categories

TEST PROGRAM CATEGORIES	
Test Category (*cat*)	**Description**
cpu	CPU subsystem related test
dev	Peripheral device test
io	I/O subsystem related test
mem	Memory subsystem related test
spu	Service Processor subsystem related test
pia	PBUS Interface Adapter test
pi2	PBUS Interface Adapter 2 test
cpx	CPU Utilities Card(s) test

1.7.3.2 Test Program Types

Test program types describes whether the test is a standalone, self-test, kernel hardware test, or an offline or online functional test. See the following table for the numbering system and description of test program types:

Table 1–7, Test Program Types

TEST PROGRAM TYPES	
Number	**Description**
0	Standalone test
1	Self-test
2	Kernel hardware test
4, 5	Offline functional test

1.7.3.3 Test Program Device Types

Test programs will test disks, tapes, terminals, printers, and networks. See the following table for the numbering scheme and a description of the test program device types:

Table 1–8, Test Program Device Types

TEST PROGRAM DEVICE TYPES	
Number	**Description**
1	Disk
2	Tape
3	Terminal
4	Printer
5	Network

1.7.3.4 Examples of Test Program Names

The following table presents some examples using the naming conventions outlined above:

> NOTE
>
> In the following table, SOFF stands for Standard Object File Format.

Table 1–9, Example Test Program Names

EXAMPLE TEST PROGRAM NAMES	
Test Program Name	**Description**
cpu4041.t	Service Processor object code in *b.out* format for *cpu4041*
cpu4041.rnn	C210 or C220 machine object code in SOFF format (relocatable)
cpu4041.x00	C210 or C220 machine object code in SOFF format (linked to run in segment 0)
mem4000.t	Service Processor object code in *b.out* format for *mem4000*
mem4000.x00	C210 or C220 machine object code in SOFF format (linked to run in segment 0)
dev4100.t	Service Processor object code in *b.out* format for *dev4100*
dev4100.x00	IOP object code in *b.out* format

1.7.3.5 Current Test Program Name Assignments

See the following table for current test program name assignments:

Table 1–10, Test Program Name Assignments

TEST PROGRAM NAME ASSIGNMENTS	
Program Name	**Test Name**
cpu4010	Reference and Modified Bits
cpu4030	Scalar Building Block Test
cpu4040	Vector Concurrency Tests
cpu4041	Vector Instruction Tests
cpu4131	Privileged Instructions and Architectural Features
cpu4231	C200 Series Privileged Instructions and Architectural Features
cpu4232	Enhanced, Non-vector, Uni-processor Instruction Tests
cpu4233	Multiprocessor Diagnostics
cpu4241	Enhanced Vector Instruction Tests
dev4100	Multibus SMD Disk Test
dev4110	Multibus SMD Disk Formatter, and Interactive Test
dev4200	Multibus STC Tape Unit Test
dev4300	Multibus Terminal Controller Test
dev4400	Multibus Line Printer Test
dev4410	Multibus Plotter Functional Test
dev4500	Multibus Ethernet Controller Test
dev4510	Multibus HYPERchannel Controller Test
dev4600	Multibus Emulator Controller Test
dev5130	VMEbus SMD/ESDI Disk Test and Formatter
dev5500	VMEbus Ethernet Controller Test
pia4000	PIA Functional Test
pi2_4000	PI2 Functional Test
io1000	EPROM-Based IOP Self-Test
io1200	VIOP Self-Test
io4000	Multibus I/O Subsystem Test
io4120	HSP/HIA Subsystem Test
io5000	VMEbus I/O Processor Test
mem4000	Memory Subsystem Test
spu1000	Service Processor EPROM-Based Self-Test
spu4000	Service Processor Interface Test
cpx4000	CPX Functional Test

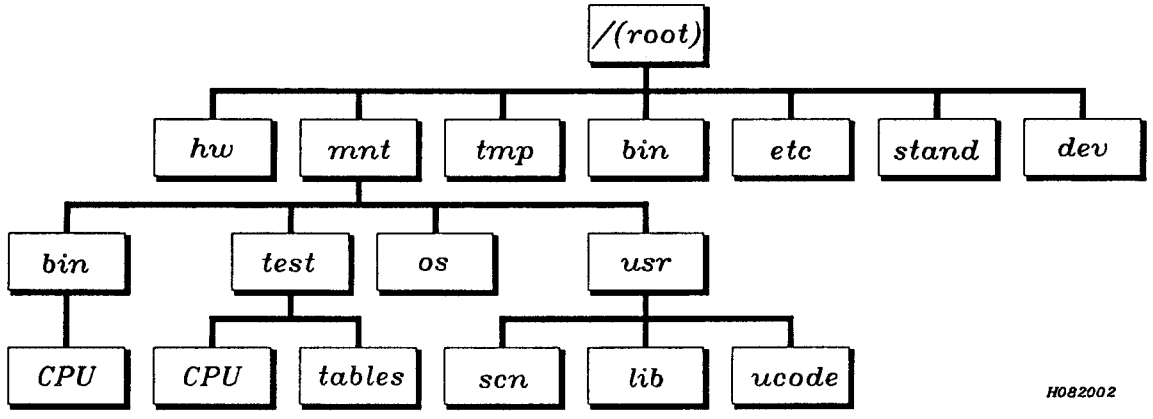
1.8 Directory Structure

The directory structure consists of the root directory (/), and the files that comprises its subdirectories. These files are distributed on four tapes and are discussed in a following section. The following list describes what is in each of the subdirectories:

- */hw* — Contains Interactive Scan (Iscan) scripts.
- */mnt* — A top-level directory that contains mountable files that contain diagnostic utilities
  - */mnt/bin* — Contains fundamental, frequently used diagnostic utilities.
    - */mnt/bin/CPU* — Contains Central Processing Unit (CPU) object files for any utility that requires it, such as *mminit*.
  - */mnt/test* — All *.t* test programs, SPU object codes, and all IOP code for Input/Output (I/O) (*io*) and peripheral device (*dev*) tests are included in this directory:
    - */mnt/test/CPU* — This contains CPU object codes used by test programs such as *cpu4030.rnn*, *cpu4041.x00*, and *mem4000.x00*.
    - */mnt/test/tables* — Contains subtests, classes, and timeout tables for CPU tests.
  - */mnt/os* — This contains CONVEX UNIX utilities and operating system, device drivers, EMACS, networking, *vmunix*, and other programs.
  - */mnt/usr* — A top-level directory for a whole group of files that are hardware specific
    - */mnt/usr/scn* — Contains Iscan ring definition files (such as *asp_rev1* and *vpd_rev1*), the output of the current *cop* configuration scheme (*cop.out*), and *scn_rings*, built by *scnlink*, which is the composite of the scan ring definition files.
    - */mnt/usr/lib* — This directory contains ASCII databases used by various programs, including *DB_cop*, and other files.
    - */mnt/usr/ucode* — Contains microcode for the vector and scalar processors.
- */temp* — This directory holds temporary files generated by a variety of programs, such as editors. It is used as scratch pad and is not backed up.
- */bin* — Contains an assortment of fundamental, frequently used SPU UNIX commands, such as *more* and *pwd*.
- */etc* — Contains system files and commands, the password file, maintenance files, and all other UNIX system administration utilities, such as *fsck* and *backup*.
- */stand* — Contains the standalone test programs, such as *spu2000*.
- */dev* — Contains all special device files, which are those the system uses to route data to particular peripheral devices (such as tape drives or disk drives) or to pseudo-devices (such as memory and communication channels).

The following figure shows each major component of the directory structure explained in the previous list.

Figure 1–5, Directory Structure



1.9 Software Distribution Tapes

The directory structure is built from four release tapes that are distributed with CONVEX computers. A brief description of the tapes and their contents are outlined below:

- System Diagnostics — This tape contains diagnostic utilities and their object files, test programs and their object files, and various diagnostic timeout tables. The following is a list of the contents of this tape:
  - */mnt/bin*
    - *CPU*
  - */mnt/test*
    - *CPU*
    - *tables*
- Diagnostic Database — This tape contains Interactive Scan (Iscan) scripts, scan ring definition files, ASCII databases, and microcode for vector scalar processors. The following is a list of the contents of this tape:
  - */hw*
  - */mnt/usr*
    - *scn*
    - *lib*
    - *ucode*
- SPU UNIX — The SPU UNIX tape contains SPU UNIX utilities, special device files, temporary files, system files, system commands, and the kernel. The following is a list of the contents of this tape:
  - */bin*
  - */stand*
  - */dev*

- */tmp*
- */etc*

- CONVEX UNIX Operating System — This tape contains CONVEX UNIX utilities, device drivers, EMACS, networking, operating system, *vmunix*, and other programs. The following is a list of the contents of this tape:
  - */mnt/os*

For installation procedures, see the release notes with the version number that matches the version number of the tape.

1.10 Powering Up the System

When the system is powered on, the final state that the machine will reach depends on the setting of the soft front panel flags that are stored in the EEPROM located on the Service Processor. If the *automatic-reboot* flag is disabled, then the Service Processor will stop at the soft front panel display. From here, any of the soft front panel commands can be entered.

1.10.1 Booting SPU UNIX

The Service Processor can be booted from either the disk or the tape drive. To boot from the tape drive, it is necessary to have a boot image format tape. Booting from tape is desirable only if the disk needs to be reformatted or restored. Booting from tape is never necessary during normal machine operation. Actually, it is impossible to boot SPU UNIX from anything but the disk drive. When booting from tape, the program that is actually booted is *spu2000*, which is the Service Processor disk formatter, tester, and file system utility.

To boot SPU UNIX, select the disk as the boot-device. Once this is done, enter **boot** to start the boot process. The *boot* command reads the boot track from the disk and prints a header describing the revision of the boot program. At this point, the program checks the status of the mode-of-operation flag in the Service Processor EEPROM.

If the mode of operation is diagnostic, then the *boot* program prints a colon and waits for the boot filename to be entered. The format of this must be entered in this format: *dk[major,minor]pathname*, refer to *dk(5)*. The diagnostic mode of operation is most useful if something besides the default program (*/unix*) is needed. In the field, this will be necessary only when running *spu2000*. If the mode of operation is normal, then the colon prompt will never be issued and */unix* will be booted on the Service Processor.

When SPU UNIX is booted, the first program executed is the Bourne shell script *.profile*. The general sequence of things done by this program are described below:

1. Performs a file system check of all mountable file systems (*/etc/fsck*).
2. Executes */mnt/bin/.diaginit*, if it exists.
3. Executes */mnt/os/.bootspu*, if it exists.

1.10.2 Using *fsck*

> **CAUTION**
>
> It is critical to the integrity of the disk files to correct *fsck* errors before attempting to modify the disks in any other way. Failure to do so will result in corrupted disk files.

If the Service Processor runs into a problem while running integrity checks on the Service Processor file systems, it is mandatory to run the *fsck* file system check program. Conditions that might cause such a problem include a hardware or software failure, or a previous power-down performed incorrectly. A typical message might be as indicated in the following figure:

Figure 1–6, Service Processor File Systems, Error Screen

```
SPU file system check in progress...
/dev/rdk0b: UNALLOCATED I=173 OWNER=root MODE=0
/dev/rdk0b: SIZE=0 MTIME=Jun 25 14:48 1985
/dev/rdk0b: NAME=a.out
/dev/rdk0b: UNEXPECTED INCONSISTENCY; RUN fsck MANUALLY
```

The phrase RUN `fsck` MANUALLY means that *fsck* was unable to determine how best to repair the problems it found on the disk. Consequently, it is necessary to run *fsck* manually; the problem is too difficult for *fsck* to repair without assistance.

To run *fsck* manually, enter:

(spu)> /etc/fsck [**file_system**]

where *file_system* is the name of the partition on which the file system is mounted (this name appears to the left of the UNEXPECTED INCONSISTENCY message). Except for the root file system, *fsck* should be run on the "raw" (nonbuffered) device of an unmounted file system, e.g., */dev/rdk0d*. For the root (/) file system, run *fsck* on the "block" (buffered) device, */dev/dk0b*. If **/etc/fsck** is executed without a *file_system* specified, all file systems in the */etc/fstab* file will be checked for inconsistencies or corruption.

Once *fsck* is invoked, messages similar to those in the following figure should print:

Figure 1–7, Messages After Invoking *fsck*

```
/dev/rdk0b
File System: (unknown)

** Checking /dev/rdk0b
** Phase 1 - Check Blocks and Sizes
** Phase 2 - Check Pathnames
UNALLOCATED I=173 OWNER=ROOT MODE=0
SIZE=0 MTIME=Jun 25 14:48 1985
NAME=/a.out
REMOVE ?
```

Entering a **y** in response to the question removes the bad file, in this case, *a.out*. The file must be removed to restore the file system to a consistent state. If **y** is the response to the query, the following output is produced:

Figure 1–8, Disk Restored Messages, After Removing Corrupted File

```
**Phase 3 - Check Connectivity
** Phase 4 - Check Reference Counts
** Phase 5 - Check Free List
131 files 1170 blocks 808 free
***** FILE SYSTEM WAS MODIFIED *****
***** REBOOT UNIX *****
```

These messages mean that *fsck* was able to restore the disk to a consistent state. Note, however, that the data on the disk does not necessarily match the copy of the data that might be in SPU UNIX memory.

If *fsck* modifies the root file system, it may mean that it is necessary to reboot the UNIX operating system. In this case, reboot with the */etc/reboot* program at the `(spu)>` prompt. The *reboot* command returns control to the soft front panel monitor (in the LOCAL or REMOTE MAINTENANCE keyswitch positions). Reboot the SPU UNIX operating system from there as specified earlier.

Often, *fsck* can safely repair problems without operator intervention. In these cases, *fsck* attempts to reboot SPU UNIX operating system after completing file system repairs.

1.10.3 Using *.diaginit*

The *.diaginit* utility is a Bourne shell script that initializes the files needed by the rest of the diagnostic utilities and test programs. The main file that must be created is the composite scan ring file, */mnt/usr/lib/scn_ring*. This file is a database of all scan rings in the system. It associates logical scan field names with information regarding how to access these fields.

When *.diaginit* is executed, it invokes the *pup* utility to determine the state of the power-up bit in the Service Processor EEPROM. If this bit is set, *.diaginit* is being executed after a power-up reboot. If the power up bit is not set, then *.diaginit* exits. If the system has been powered up, the file */mnt/usr/lib/cop.out* is moved to */mnt/usr/lib/cop.out.old* and the *cop* utility is invoked to create a new *cop.out* file.

The old and new *cop.out* files are compared; if they are different, the *scnlink* utility is executed. The *scnlink* utility examines the *cop.out* file for the ring revision of each board in the system, then links all ring revision files that correspond to the current system configuration. The state of the power-up bit in the Service Processor EEPROM is set to off, *mminit* initializes the system PCM, which initializes the system configuration files. Next *scn_util* generates */mnt/boot_db* and *.diaginit* is complete.

1.10.4 Using *.bootspu*

The *.bootspu* script determines if and how CONVEX UNIX should be booted. This routine examines the state of the mode-of-operation flag contained in the EEPROM on the Service Processor by way of the *sfpread* utility. If the mode-of-operation flag is set to *normal-os*, then the file */mnt/os/boot_cpu* is executed to boot CONVEX UNIX. If the mode-of-operation flag is set to *alternate-os*, a menu of available boot procedures is displayed and the user selects one. If the mode-of-operation is *diagnostic*, the script exits and a service processor prompt appears.

It can be seen from the above discussion the system completes power-up in one of the following states:

1. At the soft front panel prompt.
2. At the SPU UNIX prompt.
3. At the CONVEX UNIX login prompt.

The remainder of the commands described in this chapter pertain to actions that can be taken from the SPU UNIX prompt. For more information on what can be done at the soft front panel prompt, refer to the *spu1000* chapter in this manual, the *CONVEX System Manager's Guide*, and the *CONVEX Processor Operation Guide (C100 Series, C200 Series)*.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 2
Dshell and Iscan Overview

2.1 Overview

This chapter provides a brief overview of the *dshell* utility and the *iscan* utility. Included in this overview is an overall explanation of each utility and a list of each utility's commands. For a complete description of each of these utilities, refer to the *CONVEX Diagnostic Utilities Manual (C200 Series)*.

2.2 Diagnostic Shell (*dshell*) Overview

The Diagnostic Shell (*dshell*) is a command interface program that runs on the CONVEX Service Processor Unit (SPU). Most of the diagnostics available for CONVEX machines are interfaced through the *dshell*. Certain peripheral diagnostics are run as standalone tests. To determine whether a test can be run under the *dshell*, consult the appropriate test's chapter in this manual.

The *dshell* has two basic functions:

- Selecting diagnostics for execution
- Selecting test options
  - Pause on a failure or at the beginning or end of any specific subtest
  - Loop on a specific type of subtest or on a given set of subtests
  - Select subtest execution order
  - Direct test output to a file or to the screen (or both) to monitor the test as it runs or to analyze test results later
  - Select long or short error messages, or turn messages off
  - Execute either user-created or predefined command scripts

The following table list the various *dshell* commands and their functions.

Table 2–1, *dshell* Commands

COMMAND	FUNCTION
! [command]	This command is used to access, or *fork* a UNIX shell to execute the command that follows *!*.
exit	The *exit* command causes immediate termination of the *dshell* process and any test processes that may have been forked.
quit	The *quit* command causes immediate termination of the *dshell* process and any test processes that may have been forked.
^C	Returns user to the *dshell* command level if no subtest is running.
^B	Immediately terminate the *dshell* and any associated active processes. Core is dumped.
help	The *help* command causes a standard *help* menu to be displayed. The menu describes the correct command syntax for each *dshell* command and gives a terse description of what each command does.
status	The *status* command generates a report on the current state of the *dshell* command options. This report gives the name of each flag, its current value, and an explanation of its current effect.
log [options]	The *log* command provides a mechanism for specifying the number of failures that will be allowed to occur before a test or subtest terminates execution.
loop [options]	The *loop* command causes the *dshell* to repeat the execution of a test or subtest.
msgs [options]	The *msgs* command enables or disables different levels of test, class, and subtest result messages.
pause [options]	The *pause* command returns program control to the *dshell* to the beginning, end, or failure of all or specific subtests.
test [options]	The *test* executes specific tests, and displays test, class, and subtest menus.

2.3 Syntax Help for *dshell* Commands

The syntax for each *dshell* command can be obtained by typing the command with no options and pressing <CR>. For example, by entering **loop** and pressing <CR>, the syntax help in the following figure will be displayed on the screen:

Figure 2–1, Syntax Help for the *loop* Command

```
: loop
Proper syntax is:

loop off (-s) (-t)                    :disables loop modes
loop -s nnn                           :loop on subtest nnn
loop -t                               :loop on test
```

2.4 Interactive Scan (*iscan*) Overview

Interactive Scan (*iscan*) is a diagnostic utility designed to allow interactive alteration and display of the scan rings located throughout the C200 Series architectures. This utility is intended for persons with a detailed knowledge of the internal workings of the system.

The Interactive Scan language consists of a rich set of commands that are executed immediately when entered. These commands can be grouped into functions that are invoked whenever the name of the function is entered. Interactive Scan supports input file redirection as well as being able to include *iscan* files at any point during an *iscan* session. These capabilities allow the creation of a library of *iscan* scripts. By creating new *iscan* files, one is effectively adding to the capabilities of Interactive Scan.

In addition to the command mode, *iscan* contains a screen-oriented interactive scan field editor that allows the display and modification of scan fields located throughout the system.

2.5 Interactive Scan Commands

The following table is a comprehensive list of the *iscan* commands, with the short form or alias of each command, as well as the meaning of each command:

Table 2–2, Interactive Scan Commands

COMMAND	ALIAS	MEANING
!	*!*	Execute a UNIX command
adjust	*adjust*	Check scannability of all specified rings
bdrev	*br*	Check or display the revision level of a board
bdtype	*bt*	Check the presence of a board type in a slot
clear	*clr*	Clear the screen
clock	*c*	Generate a clock pulse
dump	*du*	Dump scan ring save buffers to a named file
edit	*e*	Invoke screen mode
even_parity	*even_parity*	Generate even parity
exit	*exit*	Leave Iscan
fetch	*fetch*	Fetch a field for placement in a register
fprint	*fpr*	Print to a file
get	*g*	Get a field value
halt	*hlt*	Take one or more boards out of the run state
help	*?*	Help
include	*in*	Read a specified file as input to the interpreter
iforce	*iforce*	Force a *scn_rd* operation on all gets
iupdate	*iu*	Manipulate the *iupdate* flag or force output after puts
list	*list*	List the fields associated with the named scan ring
loadscan	*ls*	Generate a scan command and a single clock pulse
log	*l*	Create a log file (this command cannot be logged)
logl	*ll*	Create a log (this command can be logged)
odd_parity	*odd_parity*	Generate odd parity
print	*pr*	Display a set of values on the screen
put	*p*	Put a field value
reset	*re*	Reset a subsystem
restore	*rs*	Restore a scan ring from a buffer
ritchie	*ri*	Terse error messages issued when toggled on
run	*r*	Put a slot in the run state
save	*sv*	Save the state of a scan ring to a buffer
scnclear	*sclr*	Clears a scan ring to all zeros
scnout	*so*	Create identical scan rings for multiple boards
screens	*sc*	Print out all names of defined screens
undump	*undu*	Read a named file to reconstruct a scan ring save buffer
verify	*v*	Enable read or compare verification

spu1000

Service Processor EPROM-Based Self-Test

Overview

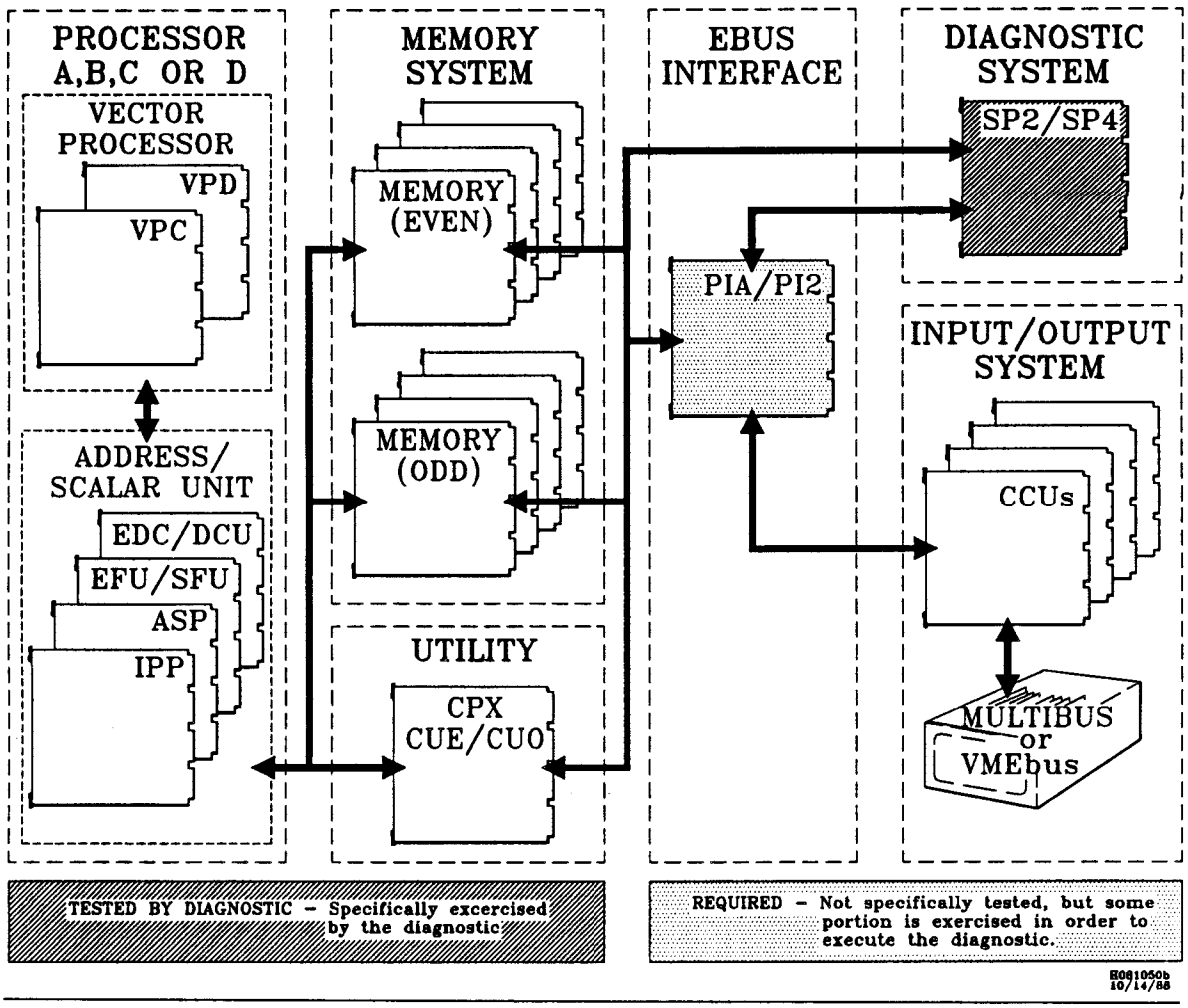
The *spu1000* functional test is an EPROM-based self-test that verifies operation of that portion of the Service Processor (SP2 for a Complex with one or two CPUs or SP4 for a Complex with three or four CPUs) necessary to execute SPU UNIX. This test does not involve any hardware other than the Service Processor and its associated peripherals. The EPROM-based self-test is divided into 14 subtests which are numbered from 1 to E in hexadecimal.

The specific hardware tested is divided into three categories:

- The 68000 and its local memory
- The Service Processor logic containing the timer chip and the two Universal Asynchronous Receiver-Transmitters (UARTs)
- The boot devices (the disk controller and the cartridge tape controller), the Direct Memory Access Controller (DMAC), and the Small Computer Standard Interface (SCSI) and Service Processor Real Time Clock registers

The *spu1000* tests the functions of the areas shown in the following figure:

Figure spu1000–1, Functional Areas Tested by *spu1000*



Prerequisites and Required Equipment

This test requires only a Service Processor (SP2 or SP4), a System Control Monitor (SCM for a Complex with one or two CPUs or ESM for a Complex with three or four CPUs) to control power, and a Peripheral Interface Adapter (PIA for a Complex with one or two CPUs or PI2 for a Complex with three or four CPUs) to provide clocks.

Test Invocation

The *spu1000* tests are stored in EPROM on the Service Processor and do not execute under SPU UNIX.

Although the EPROM-based self-test normally executes automatically whenever the **RESET** switch on the front panel is pressed, it is possible to manually invoke the test. The soft front panel selection menu allows the user to enable or disable the self-test function and to change

other functions. The front panel monitor displays the current state of all switches and displays `(fp)>` as the input prompt. To redisplay the switch selections, enter **display** <**CR**> at the soft front panel prompt.

To enable the self-test to run automatically, enter the following command at the `(fp)>` prompt:

set spu-selftest=enable

or enter the short form:

ss=e

When the self-test option is enabled, the test automatically executes when the **RESET** switch on the front panel is depressed. As each subtest executes, the corresponding test number or letter (hexadecimal digit) appears on the screen and is represented on the four red LEDs on the front panel. At the end of the test, the soft front panel menu is displayed on the screen.

When the self-test option is disabled (**set spu-selftest=disable**), the menu is displayed when the **RESET** switch is pressed. The test cannot be executed until the self-test function is set to enable.

Menu

The soft front panel menu is displayed on the screen at the end of the self-test, or when the **RESET** switch is pressed with the self-test option disabled.

The following figure illustrates the soft front panel format:

Figure spu1000–2, Soft Front Panel Selection

```
Convex Front Panel / Module Rev: X.X Version: X Class: XX / CPU SN XXXX

mode-of-operation = normal_os          boot-device = disk
location-of-bootstrap = default        power-up-reboot = enable
automatic-reboot = enable              spu-selftest = enable
os-flags = 0                           remote-port-bps = 1200
(fp)>
```

To display the front panel help screen, enter **help** at the front panel prompt. This screen is shown in the following figure:

Figure spu1000–3, Soft Front Panel Help Screen

```
DISPLAY THE CURRENT SWITCH SETTINGS:  display

REDEFINE THE VALUE OF A SWITCH:  set

    set mode-of-operation = { normal-os, alternate-os, diagnostic }
    set boot-device = { disk, tape, iomega, other2 }
    set location-of-bootstrap = { default, 1-copy, 2-copy, 3-copy }
    set power-up-reboot = { disable, enable }
    set automatic-reboot = { disable, enable }
    set spu-selftest = { disable, enable }
    set os-flags = <hex number>
    set remote-port-bps = { 1200, 300, 600, 110, 2400, 4800, 9600, 19200 }

PRESENT THE SWITCH REGISTER TO A KNOWN VALUE:  preset
    preset { standard, alternate, diagnostic, install, <hex number> }

BOOTSTRAP THE SYSTEM INTO OPERATION:  boot

SPECIAL HARDWARE DEBUG ROUTINE:  debug

Commands, switch names, and switch values may be abbreviated.
Numbers should be entered in hex with a leading digit.
```

To disable the automatic self-test mode, enter the following command at the `(fp)>` prompt:

set spu-selftest=disable

or the short form:

ss=d

If looping (running until **RESET** is pressed) is desired on the self-test, enter:

set os-flags=4

or the short form:

so=4

To start self-test looping, enter the following command at the `(fp)>` prompt:

boot

Entering *boot* without first changing the os-flag starts the normal boot procedure. To stop the looping mode and return to the soft front panel menu, press **RESET**.

To enable the self-test to run automatically, enter the following command at the `(fp)>` prompt:

set spu-selftest=enable

or the short form:

ss=e

To run the self-test, press **RESET** again. The self-test requires no user intervention during execution.

Default Sequence

When the self-test function is enabled, the test automatically executes when the **RESET** switch on the front panel is pressed.

When the self-test option is disabled, the menu is displayed when the **RESET** switch is pressed. The test does not execute until the self-test function is set to enable.

Class Descriptions

The EPROM-based self-tests are all one class. No separate class selections are provided.

Subtest Descriptions

The following table lists the test name and number and briefly describes each subtest performed. Also included are the names of the source files for each subtest.These files contain the code that is compiled or assembled and linked to form the EPROM code. The last column is the time required to run each subtest with normal defaults.

Table spu1000–1, Subtest Descriptions

SUBTEST	NAME	TEST PERFORMED	SOURCE FILE	DEFAULT TIME
1	CPU1	Verifies the 68000 instructions needed to correctly verify the EPROM checksum	*cpu1.a68*	0:01
2	ROM	Checksums the EPROM	*rom.a68*	0:01
3	CPU2	Verifies the 68000 instructions needed to correctly verify the RAM memory	*spu2.a68*	0:01
4	RAM1	Verifies the RAM buffer drivers, the functionality of the upper 4 Kbytes of Service Processor memory, and the RAM parity detection hardware	*ram1.a68*	0:01
5	CPU3	Verifies the 68000 instructions not previously tested	*cpu3.a68*	0:01
6	Timer	Verifies operation of the timer chip	*timer.c*	0:01
7	Console	Verifies operation of the console UART chip	*uart.c*	0:01
8	Remote	Verifies operation of the remote UART chip	*uart.c*	0:03
9	RAM2	Performs bit-pattern tests on all RAM memory not yet tested	*ram2.a68*	2:00
A	Mapper	Verifies bit patterns on the Service Processor map registers and tests memory map and protection features	*map.a68*	0:01
B	RAM3	Verifies the RAM that was masked by EPROM memory	*ram3.a68*	0:03
C	Boot	Performs a confidence check on boot devices	*periph.c, driver.c, disk.c, cart.c*	0:01
D	DMAC	Verifies DMAC	*dmac.a68, dmac_std.c*	0:01
E	Misc.	Checks the SCSC Control Register and Real Time Clock registers to ensure they contain reasonable values	*testmisc.c*	0:01

Service Processor Self-Test Error Reporting

If any of the subtests fail, the Service Processor itself is faulty and should be replaced. When a failure occurs, an error message displays on the console and the **ATTENTION** indicator on the front panel flashes. Since subtest identification (ID) numbers are displayed at the beginning of each subtest, the last ID displayed is the ID of the failing subtest. The ID number of the failing subtest is also displayed by the four test LEDs on the front edge of the SPU board. These subtests are displayed using the vertically arranged red LEDs to represent a different binary bit in the following format:

- <3> (top bit)
- <2>
- <1>
- <0> (bottom bit)

Many error messages are of the form:

subtest_name: error_code

The failing location, bank number, and bit position are displayed for a RAM-related failure.

Subtest 1, CPU1

The CPU1 test verifies the 68000 instructions that are used in the ROM checksum test. The strategy of this test is to access as little ROM as possible until the contents of ROM are verified by the ROM checksum test. If the CPU1 test fails, testing is terminated, and the following error message is displayed:

```
CPU Test Error:  CPU1-<XX>
```

where:

XX is the error code as identified in the following table:

Table spu1000–2, CPU1 Error Codes

NAME	SUBTEST	ERROR CODE	DESCRIPTION/(FAILED)
CPU1	1	10	Program control operations test (beq, bne)
		20	Integer arithmetic operations test (cmpi, cmpa, cmp)
		30	Register verification test (addl, lea, dbf)
		40	Checksum calculation capability test (addl, lea, dbf)

Subtest 2, ROM

The self-test object code is contained in the EPROM with a checksum stored in the last 4 bytes. The checksum test is structured so that the summation of the data contents of ROM on a 32-bit basis will result in a sum of zero. The ROM test performs this summation and verifies the result is zero. If the result is not zero, testing is terminated, and the following error message is displayed:

```
ROM Checksum Error
```

The following error table applies to the ROM subtest:

Table spu1000–3, ROM Error Codes

NAME	SUBTEST	ERROR CODE	DESCRIPTION/(FAILED)
ROM	2	N/A	ROM checksum test

Subtest 3, CPU2

The CPU2 test verifies the instructions used in the RAM1 test. If the CPU2 test fails, testing is terminated, and the following error message is displayed:

```
CPU Test Error:  CPU2-<XX>
```

where:

XX is the error code as identified in the following table:

Table spu1000–4, CPU2 Error Codes

NAME	SUBTEST	ERROR CODE	DESCRIPTION/(FAILED)
CPU2	3		**Register verification tests**
		10	Uniqueness test (do-d7, a0-a7, usp)
		11	Functionality test (d2-d7, a1-a7, usp)
		20	Addressing modes test
		30	Program control operations test (bcc)
		40	Data movement operations test (move, sr)
			Integer arithmetic operations test
		50	(cmpb, condition code generation)
		51	(cmpm)
		52	(addq, subq)
		60	Data movement operations test (exl, move, moveq)
		70	Shift and rotate operations test (rol)
		80	Bit manipulation operations test
		80	Register (bchg, bclr, bset, btst)

Subtest 4, RAM1

RAM1 tests for pin-to-pin shorts on the RAM data output lines by walking a one and then a zero across the memory word at each RAM boundary. If this test is successful, then the upper 4 Kbytes of RAM are tested for functionality. The tests are performed in the following order:

1. Output shorts
2. Address uniqueness
3. Bit functionality
   a. True/complement: 5555
   b. True/complement: AAAA
   c. True/complement: 5454
   d. True/complement: A8A8

Up to five errors will be logged before aborting the RAM1 test. This permits isolated bit failures to be distinguished from gross failures, such as output shorts. The RAM1 error message has the following format:

```
RAM Test Error

Loc: <XXXXXX>   Bank: <Y>   Bits: <ZZ ZZ ... ZZ>
```

where:

XXXXXX is the failing location in hex
Y is the failing bank number
ZZ are the failing bit numbers: <15..00>
PU — Indicates the upper byte parity failed
PL — Indicates the lower byte parity failed

If failures are detected during the RAM1 memory tests, but the failure count does not reach five, the RAM1 test is terminated after the last memory pattern has been executed. If no memory

errors are detected, RAM1 also verifies the parity error detection circuitry. If this test fails, testing is terminated, and the following error message is displayed:

```
Parity Interrupt Test Error
```

There are no error codes in the RAM1 subtest; however, a test description is provided in the following error table:

Table spu1000–5, RAM1 Error Codes

NAME	SUBTEST	ERROR CODE	DESCRIPTION/(FAILED)
RAM1	4	N/A	RAM shorts test, upper 4 Kbyte functionality test, parity interrupt test

Subtest 5, CPU3

The CPU3 test verifies all remaining 68000 instructions except *stop* and *reset*. CPU3 also verifies exception interrupt processing for a variety of conditions. If CPU3 detects an error, testing is terminated, and the following error message is displayed:

```
CPU Test Error:  CPU3-<XX>
```

where:

XX is the error code as identified in the following table:

Table spu1000–6, CPU3 Error Codes

NAME	SUBTEST	ERROR CODE	DESCRIPTION/(FAILED)
CPU3	5		**Program control operations tests**
		10	(bsr, jsr, rts, rtr)
			Data movement operations tests
		20	(link, unlink)
		21	(movem)
		22	(movep)
		23	(pea)
		24	(swap)
			Integer arithmetic operations tests
		30	(addb, condition code generation)
		31	(subb, condition code generation)
		32	(add, addq, addx, clr, sub, subq, subx)
		33	(ext, neg, negx)
		34	(divs)
		35	(divu)
		36	(muls)
		37	(mulu)
		38	(tas)
		39	(tst)
			Logical operations tests
		40	(and, or, eor, not)
		41	(andi, ori, eori)
			Shift and rotate operations tests
		50	Register (asl, asr, lsl, lsr, rol, ror, roxl, roxr)
		51	Memory (asl, asr, lsl, lsr, rol, ror, roxl, roxr)
			Bit manipulation operations tests
		60	Memory (bchg, bclr, bset, btst)
		70	Binary coded decimal operations tests (abcd, nbcd, sbcd)
		80	System control operations tests (chk, rte, trap, trapv) [*stop*, *trace* not tested]
			Miscellaneous exception condition tests
		90	(Divide by zero, illegal instruction)
		91	(Privilege violations) [Reset not tested]
		92	Address errors

Subtest 6, Timer

The timer test verifies the operation of the AMD 9513 timer chip. First, the RAM on the chip is tested by walking a column of ones and zeros through all the registers on the chip. Next, each frequency source to the timer is tested for accuracy. If the timer test detects an error, testing is terminated, and the following error message is displayed:

```
Timer Error: <XXXXXXXX>
```

where:

XXXXXXXX is the error code.

The following error table provides information on the timer test and its error codes:

Table spu1000–7, Timer Error Test

NAME	SUBTEST	ERROR CODE	DESCRIPTION/(FAILED)
Timer	6		**On-board timer test**
		1	Timer chip registers do not match after same number of clocks
		2	After same number of clocks the counters do not match each other
		3	Error tolerance exceeded for value in counters

Subtest 7, Console

The console test verifies the operation of the RS-232 port connected to the system console device. This tests only the local port, and gives no indication of the correct operation of the console device itself. First, the RS-232 port is placed in a loopback mode, allowing it to read each character that has been written to the port. All 256 possible byte values are written to and read from the port. Next, each of the interrupts on the port is tested. If the console test detects an error, testing is terminated, and the following error message is displayed:

```
Console Error: <XXXXXXXX>
```

where:

XXXXXXXX is the error code.

The following error table provides information for the console test:

Table spu1000–8, Console Error Test

NAME	SUBTEST	ERROR CODE	DESCRIPTION/(FAILED)
Console	7		**System console RS-232 port test**
		1	Loopback data does not match
		2	Timeout occurred on UART
		3	Framing error from UART
		4	Overrun error from UART
		5	Parity error from UART
		6	Interrupt not received

Subtest 8, Remote

The remote test verifies the operation of the RS-232 remote port. This tests only the remote port, and gives no indication of the correct operation of the remote connection itself. The RS-232

port is placed in the loopback mode, and tested in the same manner as the local console. If the remote test detects an error, testing is terminated, and the following error message is displayed:

```
Console Error: <XXXXXXXX>
```

where:

XXXXXXXX is the error code.

The following error table provides information for the remote test:

Table spu1000–9, Remote Error Codes

NAME	SUBTEST	ERROR CODE	DESCRIPTION/(FAILED)
Remote	8		**Remote RS-232 port test**
		1	Loopback data does not match
		2	Timeout occurred on UART
		3	Framing error from UART
		4	Overrun error from UART
		5	Parity error from UART
		6	Interrupt not received

Subtest 9, RAM2

The RAM2 test verifies all of Service Processor RAM above ROM. The following tests are performed:

1. Address uniqueness
2. Bit functionality
   a. True/complement: 5555
   b. True/complement: AAAA
   c. True/complement: 5454
   d. True/complement: A8A8

As in the RAM1 test, up to five errors will be logged before the RAM2 test is terminated. The RAM2 error message has the following format:

```
RAM Test Error

Loc: <XXXXXX>   Bank: <Y>   Bits <ZZ ZZ ... ZZ>
```

where:

XXXXXX is the failing address location in hex
Y is the failing bank number
ZZ are the failing bit numbers: <15..00>
PU — Indicates the upper byte parity failed
PL — Indicates the lower byte parity failed

If failures are detected during the RAM2 memory tests, but the failure count does not reach five,

the RAM2 test is terminated after the last memory pattern has been executed. If no memory errors are detected, RAM2 also verifies the parity error detection circuitry. If this test fails, testing is terminated, and the following error message is displayed:

```
Parity Interrupt Test Error
```

The following error table applies to the RAM2 subtest:

Table spu1000–10, RAM2 Error Codes

NAME	SUBTEST	ERROR CODE	DESCRIPTION/(FAILED)
RAM2	9	N/A	RAM functionality test for all RAM above ROM address space

Subtest A, Mapper

The map test verifies the memory mapper RAM and then tests mapper functionality. The following tests are performed on the memory mapper RAM:

1. Output shorts
2. Address uniqueness
3. Bit functionality
   a. True/complement: 5555
   b. True/complement: AAAA
   c. True/complement: 5454
   d. True/complement: A8A8

Up to five errors will be logged before aborting the mapper RAM test. If an error is detected in the mapper RAM, the following error message is displayed:

```
Map RAM Test Error:

Loc: <XXXXXX>   Bank: <Y>   Bits: <ZZ ZZ ... ZZ>
```

where:

XXXXXX is the failing address location in hex
Y is the failing bank number
ZZ are the failing bit numbers: <15..00>
PU — Indicates the upper byte parity failed
PL — Indicates the lower byte parity failed

After verifying the functionality of the mapper RAM, the map test performs a mapper operation test. This test verifies that memory mapping and the valid, read, write, and execute control bits function correctly. If an error is detected, the test is terminated and the following error message is displayed:

```
Map Test Error: <XX>
```

where:

XX is the error code as identified in the following table:

> **NOTE**
>
> In the following table, R, W, and E stand for read, write, and execute respectively.

Table spu1000–11, Mapper Error Codes

NAME	SUBTEST	ERROR CODE	DESCRIPTION/(FAILED)
Mapper	A	(printed message)	Map RAM functionality test
		00	Address translation test
			Access control tests
		10	(Supr R in 512 Kbyte-4 Mbyte range)
		11	(Supr W in 512 Kbyte-4 Mbyte range)
		12	(Supr E in 512 Kbyte-4 Mbyte range)
		20	(Supr R in nonvalid page)
		21	(Supr W in nonvalid page)
		22	(Supr E in nonvalid page)
		40	(User R in 4 Mbyte-16 Mbyte range, uio = 0)
		41	(User W in 4 Mbyte-16 Mbyte range, uio = 0)
		42	(User E in 4 Mbyte-16 Mbyte range, uio = 0)
		50	(User R in 4 Mbyte-16 Mbyte range, uio = 0)
		51	(User W in 4 Mbyte-16 Mbyte range, uio = 0)
		52	(User E in 4 Mbyte-16 Mbyte range, uio = 0)
		60	(User R in nonvalid page)
		61	(User W in nonvalid page)
		62	(User E in nonvalid page)
		70	(User R with read enabled)
		71	(User W with write enabled)
		72	(User E with execute enabled)
		80	(User R with read disabled)
		81	(User W with write disabled)
		82	(User E with execute disabled)

Subtest B, RAM3

The RAM3 test verifies the remainder of Service Processor RAM that occupies the same address space as Service Processor ROM. RAM3 uses the same memory tests and error message format used in RAM2. The following tests are performed:

1. Address uniqueness
2. Bit functionality
   a. True/complement: 5555
   b. True/complement: AAAA
   c. True/complement: 5454

d. True/complement: A8A8

As in the RAM1 and RAM2 tests, up to five errors will be logged before the RAM3 test is terminated. The RAM3 error message has the following format:

```
RAM Test Error

Loc: <XXXXXX>   Bank: <Y>   Bits: <ZZ ZZ ... ZZ>
```

where:

XXXXXX is the failing address location in hex
Y is the failing bank number
ZZ are the failing bit numbers: <15..00>
PU — Indicates the upper byte parity failed
PL — Indicates the lower byte parity failed

If failures are detected during the RAM2 memory tests, but the failure count does not reach five, the RAM2 test is terminated after the last memory pattern has been executed. If no memory errors are detected, RAM2 also verifies the parity error detection circuitry. If this test fails, testing is terminated, and the following error message is displayed:

```
Parity Interrupt Test Error
```

The following error table applies to the RAM3 subtest:

Table spu1000–12, RAM3 Error Codes

NAME	SUBTEST	ERROR CODE	DESCRIPTION/(FAILED)
RAM3	B	N/A	RAM functionality test for RAM in ROM address space

Subtest C, Boot Device

The boot device test verifies the interface to the Service Processor boot devices. The SCSI control interfaces are pattern tested with the SCSI bus reset asserted.. Functionality of each device is verified and error messages are displayed that describe the failing part of the Service Processor. The interface testing centers around the SCSI disk and cartridge tape interface.

Error messages may be self-explanatory or may require the user to look up an error code in either Table spu1000–14, Subtest Generated Error Codes, or Table spu1000–15, Controller Generated Error Codes.

Table spu1000–13, Boot Device Error Codes

NAME	SUBTEST	ERROR CODE	DESCRIPTION/(FAILED)
Boot Device	C	N/A	Boot device test

The following table contains error codes which are reported by the subtest itself:

Table spu1000–14, Subtest Generated Error Codes

ERROR CODE	DESCRIPTION
ffff0001	Controller took longer than 10 seconds to go busy after selected
ffff0002	Controller took longer than 1 second during acknowledge/request handshake during command transfer
ffff0003	Controller took longer than 1 second during acknowledge/request handshake during data input transfer
ffff0004	Controller took longer than 1 second during acknowledge/request handshake during data output transfer
ffff0005	Controller took longer than 10 seconds to give "message complete" status after command
ffff0006	Controller took longer than 1 second to give "message ready" status after command
ffff0007	Controller reported parity error
ffff0008	Error occurred during error processing
ffff0009	Data transfer did not complete specified length
ffff000a	Disk drive type specification is invalid
ffff000b	Block length specified in read or write is not multiple of sector size
ffff000c	Controller took longer than 1 second to go idle
ffff000d	Read/write compare failed in test
ffff000e	Controller timeout without request after drive select
ffff000f	Controller timeout without request after command

The following table contains error codes that are returned by the controller:

Table spu1000–15, Controller Generated Error Codes

ERROR CODE	DESCRIPTION
	Drive error codes
00	No sense
01	No index signal
02	No seek complete
03	Write fault
04	Drive not ready
06	No track 00
	Data medium errors codes
10	ID CRC error
11	Uncorrectable data error
12	ID address mark not found
13	Data address mark not found
14	Record not found
15	Seek error
18	Data check in no retry mode
19	ECC error during verify
1a	Interleave error
1c	Unformatted or bad format on drive
1d	Self-test failed
1e	Defective track (media errors)
	Service Processor Command error codes
20	Invalid command
21	Illegal block address
23	Volume overflow
24	Bad argument
25	Invalid logical unit number

Subtest D, DMAC

The Direct Memory Access Controller (DMAC) test performs memory-to-memory transfers to check the integrity of the DMAC. DMAC testing occurs as follows:

- A pattern test is performed on the DMAC'S General Control Register (GCR).
- The test cycles through the four DMAC channels; each channel is fully tested before the test continues with the next channel.
  - The registers associated with that channel (shown in the following list) are pattern tested.
    - DCR — Device Control Register
    - OCR — Operation Control Register
    - SCR — Sequence Control Register
    - MTCR — Memory Transfer Count Register

- MAR — Memory Address Register
- DAR — Device Address Register
- BTCR — Base Transfer Count Register
- BAR — Base Address Register
- NIVR — Normal Interrupt Vector Register
- EIVR — Error Interrupt Vector Register
- MFCR — Memory Function Code Register
- CPR — Channel Priority Register
- DFCR — Device Function Code Register
- BFCR — Base Function Code Register

- The test performs interrupt terminated memory-to-memory transfers to ensure that the DMAC moves data correctly.

If a register fails a pattern test, an error message of the following format is displayed:

```
DMAC Register Pattern Test Failed:
Register: <WWWWWWWW>   Address: <XXXXXXXX>
Pattern: <YY> (<ZZZZ>)   Mask: <AA>
Actual: <BB>   Expected: <CC>
```

where:

WWWWWWWW is the register name
XXXXXXXX is the register address
YY is the pattern value
ZZZZ is the pattern name
AA is the register mask (bits that exist in the failing register)
BB is the actual value read from the register
CC is the expected value to be read from the register
Rebound is a failed pattern test

Next, memory-to-memory moves are performed and interrupts used to indicate completion. Should an error occur when performing a move or checking a transfer for accuracy, an error message of the following format will be displayed:

```
<string>

channel: X   base address: YYYYYYYY
<register>   <initial value>   <current value>
<register>   <initial value>   <current value>
<register>   <initial value>   <current value>
.
.
.
```

where *string* is one of the following messages:

```
Timeout while waiting for DMAC interrupt
Received error interrupt from DMAC
Received wrong interrupt from DMAC
DMAC move buffer not as expected
```

where:

X is the channel number
YYYYYYYY is the channel address

The following error table applies to the DMAC subtest:

Table spu1000–16, DMAC Error Codes

NAME	SUBTEST	ERROR CODE	DESCRIPTION/(FAILED)
DMAC	D	N/A	DMAC controller test

Subtest E, Miscellaneous

The miscellaneous test checks the values contained in several registers to ensure they contain reasonable values. The SCSI Control Register and the Real Time Clock Registers are checked. Should a register be out of range, an error message of the following form will be displayed:

```
Register: <WWWWWWWWW>   Address: <XXXXXXXX>

Actual: <YYYY>   Minimum: <ZZZZ>   Maximum: <AAAA>
```

where:

WWWWWWWW is the register name
XXXXXXXX is the register address
YYYY is the actual value read
ZZZZ is the minimum allowable value
AAAA is the maximum allowable value

The following table lists the miscellaneous registers, their address, and a minimum and maximum value for each:

Table spu1000–17, Miscellaneous Registers

REGISTER	ADDRESS	MIN. VALUE	MAX. VALUE
SCSI Control Register	0xffd00c	0	0
RTC 1/100 Second	0xffd040	0	99
RTC Hours	0xffd042	0	23
RTC Minutes	0xffd044	0	59
RTC Seconds	0xffd046	0	59
RTC Month	0xffd048	0	12
RTC Day	0xffd049	0	31
RTC Year	0xffd04c	0	99
RTC Day of Week	0xffd04e	0	6

The following error table applies to the miscellaneous subtest:

Table spu1000–18, Miscellaneous Error Codes

NAME	SUBTEST	ERROR CODE	DESCRIPTION/(FAILED)
MISC	E	N/A	Miscellaneous register test

THIS PAGE INTENTIONALLY LEFT BLANK

spu2000

Service Processor Peripheral Test

Overview

Test program *spu2000* is a dual-purpose, standalone test/utility program. It performs two functions: peripheral test/format and SPU UNIX root partition restores. This program does not operate under the *dshell*. The test/utility function formats and tests the disk and cartridge tapes. The SPU UNIX root *restore* function uses tapes created by */etc/backup* to create a UNIX root on the Winchester or IOmega disk. This program tests the cartridge tape and disk interfaces.

Test Program Invocation

Dshell does not support this standalone test/utility program.

User Interface

One way to execute this test program is to boot a */etc/backup* format tape. The *backup* script copies the *spu2000* program to the boot location on the cartridge tape. Then, the test program boots and the first prompt displays.

A faster method of executing this test program can be used if root exists on the Winchester disk. First, change the C200 Series system switch to local. If SPU UNIX is running, type:

/etc/reboot<CR>

The soft front panel selection menu displays on the screen. (If SPU UNIX is NOT running, press the reset button to display the front panel menu.)

At the *(fp)>* prompt, type:

boot<CR>

The following displays:

```
SPU UNIX boot (Generated      )
:
```

Then enter the following:

dk(1,0)stand/spu2000

Whether *spu2000* is executed from tape or disk; the first-level prompt presents four options, as shown in the following figure:

Figure spu2000–1, Peripheral Test Selection

```
SPU Disk/Tape Diagnostic Utility $Revision: 1.3 $

     (U) for UNIX Root Restore
     (D) for Disk/Tape Utility
     (S) for SPU Hardware Utility
     (R) for Reboot SPU

Enter utility to execute -
```

NOTES

- Entering **U** restores the UNIX root.
- Entering **D** displays the disk/tape utility menu (refer to "Disk/Tape Format/Test Function").
- Entering **S** displays the menu for the Service Processor hardware utility.
- Entering **R** simulates a reset to the Service Processor and returns control of the Service Processor back to the EPROM code.

UNIX Root RESTORE Function

Once this mode is entered, the program reads and displays the date of the backup.

Figure spu2000–2, UNIX Root RESTORE Function Display

```
SPU UNIX Root Partition Restore
    reading bad block table...
       Attempting sector: 0 Successful.
    reading root date code ...
       Attempting sector: 0 Successful.

SPU UNIX root size = 2052 blocks.  Backed up Mon Aug 19 21:48:39 1985
```

After the backup date displays, the function asks whether to restore to a disk (Winchester) or IOmega disk.

```
Restore root onto disk or IOmega? [di] (d)
Define the device to recover
```

Then it asks for input disk parameters (or use defaults) for the selected disk display (refer to "Disk/Tape Format/Test Function). Finally, the screen prompts for restore permission. Either 0 or 1 can be selected for the recovery copy of the root from the tape.

```
Recover the root at this time? [yn] (n)
Recover copy 0 or 1? [01] (0)
```

Disk/Tape Format/Test Function

When *d* is selected as the peripheral test selection, the *disk/tape* menu displays as follows:

Figure spu2000–3, Disk/Tape Format/Test Function Display

```
SPU Disk/Tape Utility

     (D) for Disk (SPU Winchester)
     (T) for Tape (SPU cartridge)
     (I) for IOmega (SPU removable disk)
     (E) for Exit Test

Enter controller type/function -
```

WARNING

Please be careful when entering responses, as the format utility is data destructive.

To return to the main level prompt, enter **E**; otherwise, enter a selection for disk, tape, or Iomega. Entering **D**, **T**, or **I** causes the following prompts:

```
Format desired using standard defaults and no prompts [yn] (y)?
```

If *y* is entered, the user is given the opportunity to change the answer as the following prompt displays:

```
ALL PREVIOUS DATA WILL BE DESTROYED.  ARE YOU SURE [yn] (n)?
```

Replying *y* begins a format of the device without requiring any further intervention. When the format finishes, the following prompt displays, asking whether another format is desired:

```
Format another with defaults and no prompts [yn] (n)?
```

If *y* is entered, the automatic format operation repeats. Answering *n*, returns the user to the main level prompt.

If the first question `Format desired using standard defaults and no prompts)` is answered with *n*, the user is prompted for disk information. If a Winchester (disk) or IOmega is being used, the following disk parameter prompts display with the default shown in parentheses.

Figure spu2000–4, SPU Winchester Disk Parameters Display

```
SPU Winchester Disk Parameters:

     Number of heads ---------  (6)     ->
     Number of cylinders -----  (320)   ->
     Start of write precomp --  (128)   ->
     Step rate ---------------  (2)     ->
     Sector data size --------  (512)   ->
     Sectors per track/head --  (18)    ->
     Logical drive number ----  (0)     ->
     Sector interleave -------  (3)     ->

Are all inputs correct? - [yn] -> y
```

The optimum parameters (refer to **Table spu2000-1**) have been initialized in the program and are recommended as the defaults to the questions. (Although it is possible to change these parameters, it is not recommended.)

If a tape is used, the optimum parameters display, but cannot be changed.

The next prompt displays:

```
Format/test, Debug, or Abort operation [F,D,A]?
```

The *abort operation* option returns the user to the main prompt level; whereas, the *format/test* option moves the user to the *subtest enable* prompts, which are illustrated in **Figure spu2000-2**. (At this time, the *debug* option is nonfunctional and returns the user to the menu.)

Table spu2000–1, Service Processor Disk/Tape Format Defaults

Parameter	Tape	Winchester	Iomega
Number of heads	6	6	1
Number of cylinders	251	320	306
Start of write precomp	N/A	128	0
Step rate	N/A	2	0
Sector data size	1024	512	512
Sectors per track/head	17	18	64
Logical drive number	0	0	0
Sector interleave	1	3	16

Subtest Enable

Each subtest can be enabled separately. The tests selected can be looped on, and the maximum number of errors allowed can be modified. The test prompts display one at a time as shown in **Figure spu2000-2**.

Figure spu2000–5, Service Processor Peripheral Test Prompts

```
      Run maintenance track test? --  [yn]       (n)     ->
      Run format test? -------------  [yn]       (n)     ->
      Run write test? --------------  [yn]       (n)     ->
      Run read test? ---------------  [yn]       (n)     ->
      Run bad block fix? -----------  [yn]       (n)     ->
      Run random read test? --------  [yn]       (n)     ->
      Run seek test? ---------------  [yn]       (n)     ->
      LOOP ON TESTS? ---------------  [yn]       (n)     ->
      MAX NUMBER OF ERRORS? -------------        (1)     ->
Are all inputs correct? - [yn] ->
```

Run these tests only in the order shown. A write test must be selected before a read test can be attempted because the read test performs a 'data compare' using the pattern the write test generates.

Service Processor Hardware Utility

The Service Processor hardware utility functions the same as *sp2util* (refer to the *CONVEX Diagnostic Utilities Manual (C200 Series)*) but with reduced capability due to its standalone operation.

> **NOTE**
>
> This utility is menu driven and is not to be used by anyone except qualified CONVEX employees.

Class Descriptions

Because this test is all one class, no class selections are provided.

Subtest Descriptions

The following paragraphs describe the *spu2000* subtests in execution order, according to the peripheral test prompts as illustrated in Figure spu2000–5.

Maintenance Track Subtest

This subtest applies to IOmega disks only. It displays the data stored on the maintenance track and allows changes in some areas. The data that displays consists of five parts, as follows (**Figure spu2000-3** shows an example of the maintenance track subtest display):

Bad track log Indicates all tracks flagged as bad, along with the number of the alternate track being used.

Auto stop of spindle Specifies the amount of idle time before stopping the spindle. The time displayed is the time the drive remains spinning while not in use. The purpose of stopping the spindle is to prevent excessive wear of the media under the read/write head.

Write verify flag Provides for automatic read of any written sector and performance of a CRC check when the flag is *y*. Setting of the flag causes a slowdown of the I/O since each time a track is written, a second revolution of the disk is required to verify the written blocks. This technique, however, provides the advantage of detecting a write error at write time, which allows a retry of the write operation.

ECC checking flag Provides for automatic calculation of an ECC for the entire track whenever one or more sectors on a track are written—if the flag is *y*. Each time data is written to a track, the entire track must be read to recompute the ECC, and then the ECC block must be written. This can require as many as three revolutions of the disk. Having the flag set at *y* provides the capability to correct unrecoverable read errors. The ECC can recover an entire 256-byte block which has become unreadable.

Interleave value Indicates the number to be added to the last sector number. This number indicates the next consecutive sector to be used for a read or write operation. Interleaving sectors allow processing time between reads or writes. If set properly, the next desired sector should be coming under the read/write heads very shortly after processing of the last sector is completed.

Figure spu2000–6, Maintenance Track Data Display

```
                    Running Maintenance track subtest


                            MAINTENANCE TRACK DATA

Bad Track Log:
     bad        replacement
     track      track
     -----      -----
     No bad tracks have been flagged

Other Data:
     Idle time before auto stop of spindle (minutes) -----> 5
     Write verify ---------------------------------------->   y
     Check ECC ------------------------------------------->   y
     Interleave ------------------------------------------>   8

Change maintenance track data [yn] (n)? n
```

Any of the maintenance data except the bad track data can be changed. *If the ECC flag is changed to* **y** or the interleave value is changed, a format subtest automatically runs (whether specified or not). This is necessary to make the disk usable. If ECC is already set to *y*, the format subtest does not execute (unless enabled). The following figure shows an example of the input screen for the Maintenance Track Subtest.

Figure spu2000–7, Changing Maintenance Track Data

```
Change maintenance track data [yn] (n)? y
     Idle time before auto stop of spindle (minutes)
               [5/7.5/10/.../30/d (for disabled)]    (5)     ->
     Write verify
               [y/n]                                         (y)     ->
     Check ECC (Disk will be reformatted if change to 'y')
               [y/n]                                         (n)     ->
     Interleave (Disk will be reformatted if change)
               [1/2/4/8/16/32]                               (8)     16


WARNING - THE DISK WILL BE REFORMATTED!

Are all inputs correct [yn]? y

     maintenance track change underway ...

               Maintenance track subtest -------> passed    0:00:25
      Running Format subtest ------------------> passed     0:01:30
```

Format Subtest

This test formats Winchester, IOmega, or cartridge tape media. The test uses the format parameters selected or the defaults from the Maintenance Track Subtest, with one exception. If an IOmega is being formatted and the interleave value was changed in the previous Maintenance Track Subtest, then that interleave value is used during formatting.

When an IOmega is being formatted and either a *NO TRACK 0* or *NO INDEX SIGNAL* error occurs, the message:

```
Reinsert disk cartridge.  Press 'return' when ready
```

displays. Reinserting the disk and pressing RETURN forces the IOmega controller to reread the maintenance tracks in hopes that another attempt will succeed. If one of these two errors still occurs on the second read, the following message displays:

```
Error-maintenance tracks on IOmega cartridge may be bad.
Rebuilding these tracks results in loss of the current bad track history.
Rebuild maintenance tracks [yn] (n)
```

Rebuilding the maintenance tracks may allow the disk to be usable again.

Write Subtest

The write test repeats a fixed pattern, *0xe5a55a5e*, the number of times necessary to fill a block. This is written to every block which the drive parameters indicate exist. For instance, if only 100 tracks are specified on a 306 track IOmega, then only the first 100 tracks are written.

Read Subtest

This test reads each block which the drive parameters indicate exist and compares each block with the fixed pattern, *0xe5a55a5e*. The test keeps all blocks that generate errors in an internal table. These blocks are marked as bad during the Bad Block Fix Subtest so they will no longer be used.

Bad Block Fix Subtest

When this test begins, the user is prompted to enter manually the number of each bad block (if any) that was not previously flagged by the Read Subtest as being bad. Then the test sorts the blocks into ascending order and writes them to the disk/tape. After the blocks are logged on the disk/tape, they are no longer available for data storage. Instead, alternate blocks are used.

Random Read Subtest

This test performs 100 reads to randomly calculated block numbers. Before the test begins, the test determines the maximum number of blocks on the disk. Block numbers are calculated in the range zero to the maximum minus one.

Seek Subtest

This seek test performs an accordion seek in reverse. A seek is made from *min* to *max* track, then to *min+1*, *max-1*, *min+2*, *max-2*, etc. The accordion seek ends with a one-track seek. Each time the head comes to rest on a track, the test reads the middle block of the track to verify that the seek was successful.

Other Subtest Options

The final two options, as shown in **Figure spu2000-2**, allow the user to specify looping on tests and/or to specify the maximum number of errors that can occur before testing is aborted. Please note that if looping is chosen, the tests loop indefinitely, unless one of the following conditions occur:

1. **A** is pressed, which returns immediately to the main menu. Since this action may leave the drive in an unknown state, the alternative described in condition 2 should be used.
2. Press either **B** or **C**, which terminates the test in a controlled manner. Thus, a few seconds may pass before this termination actually occurs. If a format was in process, the subtest must complete before termination occurs.
3. The test reaches the maximum number of errors which causes test termination.

Execution Time

The following times are maximum under nominal conditions:

Table spu2000–2, Test Execution Time

Subtest	Number of Minutes		
	Tape	Disk	IOmega
Main. Track	N/A	N/A	1.0
Format	10.0	1.5	1.5
Write	10.0	9.5	17.0
Read	20.0	11.0	10.0
Bad Block Fix	0.5	1.5	1.5
Random Read	N/A	0.5	0.2
Seek	N/A	1.5	0.8
Total	**40.5**	**25.5**	**32.0**

Error Messages

The *spu2000* diagnostic reports I/O errors in a standard format. However, if additional data is available at the time of the error, that data is also reported. The basic format for the errors follows:

```
spu2000: <error desc> on <device> drive <dev.#> during command <command>.
Operation: <operation type>  [optional data displays here and on next line]
```

where:

<error desc>	Describes the type of error that occurred, such as, NO INDEX SIGNAL, DRIVE NOT READY, SEEK ERROR.
<device>	Is one of the following: Adaptec, IOmega, or Tape.
<dev.#>	Specifies the device number. Each device begins at 0 with additional drives of the same type being numbered 1, 2, ...
<command>	Identifies the I/O command to the drive which resulted in the error; for example, READ SECTOR, WRITE SECTOR, REZERO UNIT, SEEK.
<operation type>	Provides the name of the subtest, such as, Format, Read, Bad Block Fix, Disk Restore, Seek.
[optional data]	reports the block being accessed if using a disk (Adaptec or IOmega) and a read, write, or verify is underway.

The stream, segment, and sector is printed if using a tape and the error is detected by the controller or a data compare error was detected by the Service Processor.

A third line is printed, which contains the position of the error within the block (or sector on tape), the expected data, and the actual invalid data when using either a tape or disk and a data compare error occurs.

Examples of error messages that can occur are:

```
spu2000: NO INDEX SIGNAL on IOmega drive 0 during command REZERO UNIT.
Operation:  Format

spu2000: ID CRC ERROR on tape drive 0 during command READ SECTOR.
Operation:  Read    Stream:  3    Segment:  142    Sector:  14

spu2000: DATA COMPARE ERROR on Adaptec drive 1 during command READ SECTOR.
Operation:  Read       Block:  4523
Bytes into block:  24  Expected:  0xE5A55A5E   Actual: 0x00000000

spu2000: ID CRC ERROR &
RECORD NOT FOUND on tape drive 0 during command WRITE SECTOR.
Operation:  Write   Stream:  1   Segment:  200   Sector:  5
```

Notice that the last error message reports more than one error. The tape controller sets a series of bits to indicate which errors have occurred. The error message for each error prints.

Error Descriptions

This section provides an alphabetical list of errors reported by *spu2000*, with a brief description of the error.

BAD ARGUMENT

Peripheral: Winchester

The Adaptec Winchester drive detected a bad value in one of the fields of a command block.

BAD SEEK

Peripheral: Winchester, IOmega

The disk drive was unable to find the requested track.

COMMAND TIMEOUT

Peripheral: Tape

The tape drive has taken an excessively long time performing a head load, seek, or head positioning.

DATA ADDR MARK NOT FOUND

Peripheral: Winchester, IOmega

Each time a sector is written, a fixed pattern is written just before the sector; this is called the data address mark. If this pattern is not detected when an attempt is made to read the sector back, the drive has no idea where the data begins. Thus, the drive reports a data address mark not found.

DATA COMPARE ERROR

Peripheral: Winchester, IOmega, Tape

Data just read does not match with an expected data pattern. The data in error is printed.

DATA CRC ERROR

Peripheral: Winchester, IOmega, Tape

At write time, the Cyclic Redundancy Check (CRC) algorithm combines all the bytes in a sector and generates a 2-byte field which is written after the data. Upon read back, the bytes being read are again combined, using the same algorithm, and the result is compared with the 2 bytes at the end of the data. If the bytes do not match, this error occurs.

DATA XFER NOT COMPLETE

Peripheral: IOmega

The sector buffer in the drive either was not completely written to the disk or was not completely transferred to the Service Processor when reading from the disk.

DMA TIMEOUT

Peripheral: IOmega

Transfer of data to or from the IOmega's internal memory by the IOmega's direct memory access controller failed.

DRIVE ALREADY BUSY

Peripheral: Tape

The Service Processor is about to perform an I/O operation to the tape but finds that the tape controller is busy doing an unrequested operation.

DRIVE NOT READY

Peripheral: Winchester, IOmega, Tape

The peripheral is not ready to perform any I/O operations. This error generally occurs with the cartridge tape, although it can occur from any of the peripherals. Once a tape has been inserted, it takes a maximum of 3 minutes for the tape to reposition itself. The error message displays if a test was started on the tape several seconds before the tape was inserted, and the tape did not go ready before the timeout.

ECC ERROR DURING VERIFY

Peripheral: Winchester

The drive is unable to correct an error using the Error Correction Code.

FIFO WOULD NOT EMPTY

Peripheral: Tape

The First In-First Out (FIFO) data buffer on the tape controller was not empty at the completion of an I/O operation.

ID ADDR MARK NOT FOUND

Peripheral: Winchester, IOmega

The drive writes a fixed pattern at the beginning of each ID (header) called the ID address mark. Each time the controller reads or writes a sector, it checks this address mark for validity first. Since the controller automatically performs retries, multiple bad reads of an ID address mark have already occurred before this error is reported.

ID CRC ERROR

Peripheral: Winchester, Tape

The checksum calculated during the read of an ID does not match the Cyclic Redundancy Check sum that follows the ID. This error only prints when all multiple retries to read the ID fail.

ILLEGAL BLOCK ADDRESS

Peripheral: Winchester, IOmega

The drive was sent a block (sector) address outside the valid range for the peripheral.

ILLEGAL INTERLEAVE

Peripheral: Winchester, IOmega

An interleave factor other than 1, 2, 4, 8, 16, or 32 was requested for the IOmega. For the Winchester, an interleave less than one or greater than 17 was requested.

INSUFFICIENT CAPACITY

Peripheral: IOmega

No room remains to store data; the IOmega is full.

INVALID BLOCK NUMBER

Peripheral: Tape

The controller detected an attempt to perform I/O to an invalid block on the cartridge tape. Valid block numbers must be 0 to 25601.

INVALID COMMAND

Peripheral: Winchester, IOmega

The drive controller received an unrecognizable I/O command.

INVALID LOGICAL UNIT NO.

Peripheral: Winchester

An attempt was made to access a Winchester drive which does not exist.

INVALID STREAM NUMBER

Peripheral: Tape

An attempt was made to write to a stream number other than 1 - 6.

LOST DATA

Peripheral: Tape

The host sent data too rapidly to the tape controller causing a loss of some of the data.

MEDIA NOT LOADED

Peripheral: IOmega

This error occurs when the cartridge is not loaded with the door securely closed and an I/O attempt is made to an IOmega cartridge.

NO INDEX SIGNAL

Peripheral: Winchester, IOmega

The media is not spinning at the correct speed, or it is not spinning.

NO SEEK COMPLETE

Peripheral: Winchester

The drive detects a bad seek.

NO TRACK 0

Peripheral: Winchester, IOmega

On the Winchester, this message indicates a seek to track zero failed. For the IOmega, it means the maintenance track data (all 8 copies) is bad.

RECORD NOT FOUND

Peripheral: Winchester, IOmega, Tape

The ID field for the requested block could not be found. Generally, this means I/O errors are preventing the ID from being read.

UNCORRECTABLE DATA ERROR

Peripheral: Winchester, IOmega

An error in the data field of a block was detected. Either the ECC correction was not enabled or the error was too large for ECC correction to recreate the data.

UNFORMATTED OR BAD FORMAT

Peripheral: Winchester

The Winchester drive controller detected a flawed format and aborted the current I/O operation.

VOLUME OVERFLOW

Peripheral: Winchester

The disk is full; there is no room to store additional data.

WRITE FAULT

Peripheral: Winchester, IOmega

An internal drive error resulted in an inability to complete the requested write operation. This error will occur if the automatic write verify is turned on and the verify fails.

WRITE PROTECTED

Peripheral: IOmega, Tape

An attempt was made to write to an IOmega or tape cartridge that is write protected.

THIS PAGE INTENTIONALLY LEFT BLANK

spu4000

Service Processor Interface Test

Overview

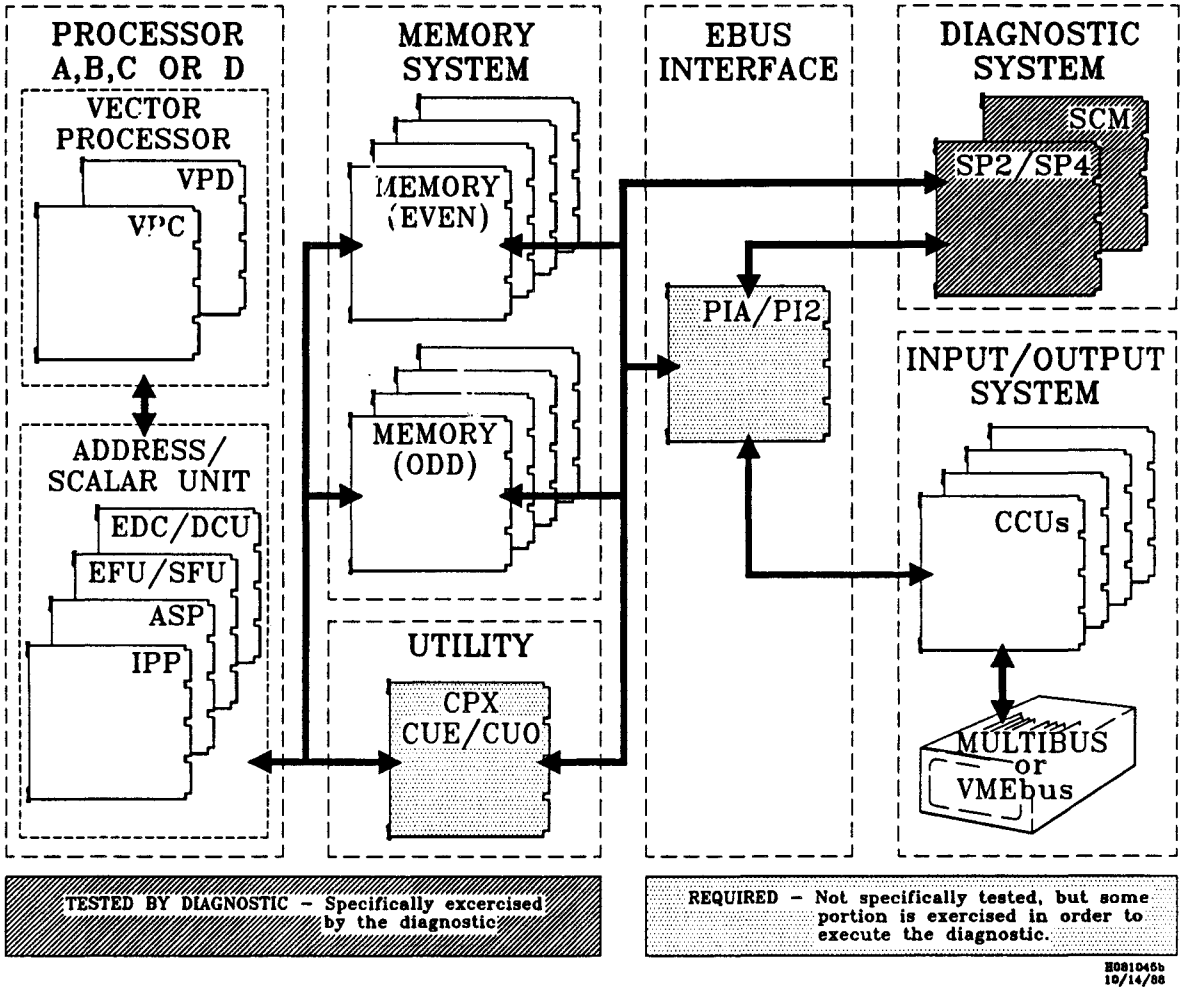
The Service Processor (SP) Interface Test (*spu4000*) verifies the interface between the Service Processor and various Field Replaceable Units (FRUs).

> **NOTE**
>
> The term "Service Processor" and "SP" are used generically to represent either SP2 or SP4 depending on the system configuration under test. The term "CPU Utility Board(s)" is used to represent either a CPX or a CUE / CUO combination depending on the system under test. Also, "PBUS Interface Board(s)" is used to represent either a PIA, a PI2 installed in the PIY slot, or two PI2s installed in the PIX and PIY slots, depending on the system configuration under test.

This test is designed to verify the hardware in a hierarchical fashion. First, control logic contained on the Service Processor is checked to ensure it is working correctly, then initial testing of the interfaces between the Service Processor and other parts of the system occurs. This is followed by more extensive testing of the Service Processor to system interfaces.

> **NOTE**
>
> The following figure represents the minimum configuration for running this test. Additional Field Replaceable Units (FRUs) installed in the system may be tested, depending on the response(s) made in the configuration menu. In general, each subtest requires only the Service Processor, the FRU which generates clocks, and possibly an FRU to be tested. There are exceptions to this, and the "PROBABLE FAULT LOCATION" column of each subtest table lists the minimum FRUs required to run each subtest.

Figure spu4000–1, Functional Areas Tested by *spu4000*



Prerequisites and Required Equipment

This test may be executed on any C200 Series machine. It should be run only after the EPROM Based Self-Test (*spu1000*) has passed.

For two CPU backplanes, the SCM, PIA, and Service Processor are required for all subtests. Also, each subtest that tests communication between a FRU and the Service Processor requires that FRU to be installed. In addition, subtest 1200 requires at least one MCM be installed in any memory slot, subtests 6103-6134 require a CPX, subtest 7200 requires both Memory Odd 0 (MO0) and Memory Even 0 (ME0), and subtest 8010 requires a CPX.

For four CPU backplanes, the ESM, CUO, and Service Processor are required for all subtests. Also, each subtest that tests communication between an FRU and the Service Processor requires that FRU to be installed. In addition, subtest 1200 requires at least one MCM be installed in any memory slot, subtests 6103-6134 require both CUE and CUO, subtest 7200 requires both Memory Odd 0 (MO0) and Memory Even 0 (ME0), and subtest 8010 requires a both CUE and CUO.

Many *spu4000* subtests check one of several interfaces between the Service Processor and another

FRU, and require the FRU to be installed to run the subtest. To provide for a wide variety of testable configurations, a method of specifying FRUs to be tested has been provided. Only subtests for FRUs specified will be attempted, subtests which require FRUs not contained in this test configuration will not be executed. The "Configuration Menu" section explains the method of FRU specification.

Test Invocation

To invoke the *spu4000* test, use the procedure shown in the following figure. All responses in **boldface** are entered by the user. The prompts and responses would appear sequentially on the screen, one line at a time. All the prompts and responses are shown in one figure for convenience.

> **NOTE**
>
> After this test is executed, the *initall* utility must be executed before invoking any other tests.

Figure spu4000–2, Test Invocation Sequence

```
(spu)>  cd /mnt/test
(spu)>  sysreset
(spu)>  dshell

CONVEX DIAGNOSTIC SHELL

: test spu4000 [-c [class numeral(s)]] [-s [subtest numeral(s)]] [+> filename]
```

> **NOTE**
>
> After entering **dshell**, specific *dshell* parameters may be changed. Refer to the "Dshell and Iscan Overview" chapter of this manual for more information.

Entering only **test spu4000** executes all applicable *spu4000* subtests sequentially. Execute a specific class(es) of subtest(s) or one or more individual subtests by using the *-c* or *-s* options, respectively. Detailed information for using these options can be found in the "Dshell and Iscan Overview" chapter of this manual. The [+>*filename*] option allows the test results to be appended to *filename*.

Configuration Menu

After *spu4000* is invoked, it prints a test heading, which consists of the test name and date, followed by the list of FRUs it believes are installed in the system. This list is called the *current configuration*, and contains only entries from the "SINGLE SLOT TERM" columns of Table spu4000-1, CPU FRU Configuration Terms and Table spu4000-2, Memory and I/O FRU Configuration Terms. The list is determined by examining information generated by *the scnlink* utility. Thus this list of FRUs should match the one in the file */mnt/user/scn/cop.out*, which is generated using the utility *cop*.

The current configuration may be edited to add or remove FRUs. Before performing any tests, the current configuration should be examined, and modified if necessary, since subtests which require hardware not in the specified configuration will not be attempted. After any modifications have been made to the current configuration, the list of FRUs to be tested is displayed. This list is called the *test configuration*. Only FRUs contained in the test configuration will be tested.

The "MULTIPLE SLOT TERM" columns of Table spu4000-1, CPU FRU Configuration Terms, and Table spu4000-2, Memory and I/O FRU Configuration Terms, are terms which represent multiple FRUs, or slots. These terms may be entered for convenience and *spu4000* will recognize them as input; however, *spu4000* uses only single-slot terms to list the FRUs in a configuration. All responses in **boldface** are entered by the user. The prompts and responses would appear sequentially on the screen, one line at a time. All the prompts and responses are shown in one figure for convenience.

Figure spu4000–3, Sample Configuration Menu

```
Test 'spu4000.t'                                    Tue Sep 15 00:00:00 1964

The current configuration includes the following slots:
  vdb  vcb  dcb  fub  ipb  asb  sp2  cpx  mo0  me0  pia  ccu0  ccu1

[command [boards]] ... (end) :
> remove all

The current configuration includes the following slots:

[command [boards]] ... (end) :
> add mcm pia cpx sp2

The current configuration includes the following slots:
  sp2  cpx  mo0  me0  mo1  me1  mo2  me2  mo3  me3  pia

[command [boards]] ... (end)  :
> remove mcm1 mcm2 mcm3e

The current configuration includes the following slots:
  sp2  cpx  mo0  me0  mo3  pia

[command [boards]] ... (end)  :
> add ccu0 add ccu1 remove mcm3o end

The test configuration includes the following slots:
  sp2  cpx  mo0  me0  pia  ccu0  ccu1

       Subtest 1000   0:00:00  passed
       Subtest 1100   0:00:00  passed
              .
              .
              .
```

As Figure spu4000-3, Sample Configuration shows, menu commands may be entered to alter the configuration only immediately following this prompt:

```
[command [boards]]... (end) :
>
```

If no attempt is made to change the configuration at this prompt, testing occurs. Otherwise the configuration is modified as requested, and the prompt re-issued, allowing the user to further alter the configuration.

Three commands are available to change the configuration: *add*, *remove*, and *end*. These commands may be used individually or in combination on an input line, with commands processed left to right. In addition, each command may be abbreviated by entering only the first letter.

1. The *add* command adds FRUs to the current configuration. The *add* command is illustrated in the previous figure where the mcm, pia, cpx, and sp2 terms are added to a configuration where no FRUs are specified. Adding these FRUs, as shown, returns a configuration which contains all eight MCMs as well as the pia, cpx and sp2. The *add* command may be followed by any number, including zero, of the terms contained in

Table spu4000-1, CPU FRU Configuration Terms and Table spu4000-2, Memory and I/O FRU Configuration Terms.

2. The *remove* command removes FRUs from the current configuration. The *remove* command is illustrated in the previous figure where the mcm1, mcm2, and mcm3e terms are removed from the configuration. Removing these terms, as shown, returns a configuration which no longer contains the mo1, me1, mo2, me2, and me3 FRUs. The *remove* command may be followed by any number, including zero, of the terms contained in Table spu4000-1, CPU FRU Configuration Terms and Table spu4000-2, Memory and I/O FRU Configuration Terms.

3. The *end* command indicates testing should begin, and that no other input should be processed or requested. The *end* command is used in the previous figure at the end of a line which contains add ccu0, add ccu1, remove mcm3o, end. This end command can be used on a blank line, or at the end of an input line containing one or more add or remove commands. The *end* command may also be indicated by pressing (RETURN) on a blank line. When the *end* command is entered, the machine will print the test configuration, and begin subtest execution.

Default Sequence

The following figure is a sample *spu4000* test invocation sequence, including test invocation and the default selection of all parameters. All responses in **boldface** are entered by the user. The prompts and responses would appear sequentially on the screen, one line at a time. All the prompts and responses are shown in one figure for convenience.

Figure spu4000–4, Sample Invocation and Default Sequence

```
(sp2)> cd /mnt/test
(sp2)> sysreset
(sp2)> dshell

CONVEX DIAGNOSTIC SHELL

: test spu4000

Test 'spu4000.t'                                    Tue Sep 15 00:00:00 1964

The current configuration includes the following slots:
  vdb  vcb  dcb  fub  ipb  asb  sp2  cpx  mo0  me0  pia  ccu0  ccu1

[command [boards]] ... (end) :
> (RETURN)

The test configuration includes the following slots:
  vdb  vcb  dcb  fub  ipb  asb  sp2  cpx  mo0  me0  pia  ccu0  ccu1

       Subtest 1000   0:00:00  passed
       Subtest 1100   0:00:00  passed
              .
              .
              .
```

To test the default set of FRUs, simply press return at the first prompt, as indicated by the carriage return, (RETURN) in the previous figure. This will result in *spu4000* attempting those subtests which apply to the FRUs in the test configuration.

The following tables indicate the possible terms which may be used to add or remove FRUs from the test configuration. Multiple-slot terms are for use only by the user, and are terms which can stand for multiple FRUs. Single-slot terms are terms which the *spu4000* test may print on a configuration line. To avoid ambiguity, each single-slot term can represent only one FRU. Either multiple-slot or single-slot terms may be entered by the user. Note that the terms listed below cannot apply all to any given system. Those terms which do not apply will be ignored.

Table spu4000–1, CPU FRU Configuration Terms

MULTIPLE SLOT TERMS	DESCRIPTION	SINGLE SLOT TERMS	DESCRIPTION
all	All FRUs		
cpu	All CPU FRUs	asa	AS, CPU A
cpua	All CPU A FRUs	dca	DC, CPU A
cpub	All CPU B FRUs	fua	FU, CPU A
cpuc	All CPU C FRUs	ipa	IP, CPU A
cpud	All CPU D FRUs	vca	VC, CPU A
as	All AS FRUs	vda	VD, CPU A
dc	All DC FRUs	asb	AS, CPU B
fu	All FU FRUs	dcb	DC, CPU B
ip	All IP FRUs	fub	FU, CPU B
vc	All VC FRUs	ipb	IP, CPU B
vd	All VD FRUs	vcb	VC, CPU B
		vdb	VD, CPU B
		asc	AS, CPU C
		dcc	DC, CPU C
		fuc	FU, CPU C
		ipc	IP, CPU C
		vcc	VC, CPU C
		vdc	VD, CPU C
		asd	AS, CPU D
		dcd	DC, CPU D
		fud	FU, CPU D
		ipd	IP, CPU D
		vcd	VC, CPU D
		vdd	VD, CPU D

Table spu4000–2, Memory and I/O FRU Configuration Terms

MULTIPLE SLOT TERMS	DESCRIPTION	SINGLE SLOT TERMS	DESCRIPTION
mcm	All Memory FRUs	mo0	Memory 0 Odd
mcme	All Memory Even FRUs	me0	Memory 0 Even
mcmo	All Memory Odd FRUs	mo1	Memory 1 Odd
mcm0	Both Memory 0 FRUs	me1	Memory 1 Even
mcm1	Both Memory 1 FRUs	mo2	Memory 2 Odd
mcm2	Both Memory 2 FRUs	me2	Memory 2 Even
mcm3	Both Memory 3 FRUs	mo3	Memory 3 Odd
		me3	Memory 3 Even
io	All PI, CU, and CCU FRUs	cpx	CPX
cu	All CPU Utility FRU(s)	cue	CUE
pi	All PBUS Interface FRU(s)	cuo	CUO
ccu	All CCU FRUs	pia	PIA
ccuy	All PBUS Y CCU FRUs	piy	PIY
ccux	All PBUS X CCU FRUs	pix	PIX
		ccu0	CCU 0, PBUS Y
		ccu1	CCU 1, PBUS Y
		ccu2	CCU 2, PBUS Y
		ccu3	CCU 3, PBUS Y
		ccu4	CCU 4, PBUS X
		ccu5	CCU 5, PBUS X
		ccu6	CCU 6, PBUS X
		ccu7	CCU 7, PBUS X

Class Descriptions

The Service Processor Interface Test is divided into nine classes. The following table identifies each of these classes, as well as their object module and source file. A description of the hardware tested by each class is given in the "TEST PERFORMED" column. The time shown in the "MAX TIME" column is the maximum amount of time this subtest (or class of subtests) should take to execute at nominal margin. Depending on system type and configuration, the actual time required may be significantly less. Margining the clocks will also affect execution time.

Table spu4000–3, Service Processor Interface Test Classes

CLASS	CLASS NAME	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	MAX TIME (min/sec)
1	SP Registers	Verifies most of the Service Processor registers	*spu4000.t*	*ctlreb.c ser_num.c run_arm.c rtc_count.c*	0:05
2	DBUS Interface Integrity	Verifies the scan ring interface, SP scan machine	*spu4000.t*	*spu_dbus_if.c dbus.c*	2:45
3	Board ID Test	Checks ability to read each board's COP	*spu4000.t*	*id_integ.c*	1:35
4	DBUS Integrity	Pattern tests scan rings on boards	*spu4000.t*	*dbus.c*	2:52
5	Hard-Error Test	Forces generation, and checks reporting of hard errors	*spu4000.t*	*err_chk.c*	1:54
6	Soft-Error Test	Forces generation, and checks reporting of soft errors	*spu4000.t*	*err_chk.c*	0:54
7	EBUS Interface	Checks SP interface with EBUS	*spu4000.t*	*ebus_ram.c ebus_popram.c ebus_ctrl.c ebus_poperr.c ebus_trans.c*	1:54
8	Interrupt Bus Integrity	Tests System Interrupt Bus operation	*spu4000.t*	*sibtest.c*	0:22
9	Margin Test	Tests ability to margin power supplies and clocks'	*spu4000.t*	*margin.c*	0:30

Subtest Descriptions

The following sections describe the subtests for each class, and explain the hardware function tested.

Each class section starts with a description of that class of subtests. Following this is a table which contains a line for each subtest in the class. After the "SUBTEST" and "TEST PERFORMED" columns of each table, a "PROBABLE FAULT LOCATION" column lists one or more Field Replaceable Units (FRUs) required to execute the subtest (in addition to the FRU which generates system clocks). The failure of any of the listed FRUs could cause a subtest to fail, however, it is possible, that the problem that causes a subtest to fail may not be located on any of the FRUs indicated in the "PROBABLE FAULT LOCATION" column. The next two columns of each table contain each subtest's object module and its source file. The "MAX TIME" column indicates the maximum amount of time the subtest should take to execute at

nominal margin. Depending on system type and configuration, the actual time required may be significantly less.

It is important to note that all subtests do not apply to all systems, and that only those subtests that apply to FRUs in the test configuration will be attempted.

Class 1 Subtests, Service Processor Registers

Class 1 subtests test some of the Service Processor's registers, the system serial number, the run-arm circuitry, and the Service Processor's real time clock.

Table spu4000–4, Class 1 Subtests

SUBTEST	TEST PERFORMED	PROBABLE FAULT LOCATION	OBJECT MODULE	SOURCE FILE	MAX TIME (min/sec)
1000	SP Control Register Pattern Test	SP	*spu4000.t*	*ctlreb.c*	0:01
1100	System Serial Number Validity	SP	*spu4000.t*	*ser_num.c*	0:01
1200	SP Run Arm Circuitry	SP, MCM	*spu4000.t*	*run_arm.c*	0:04
1300	SP Real Time Clock	SP	*spu4000.t*	*rtc_count.c*	0:01

Subtest 1000, Service Processor Control Register Pattern Test

Subtest 1000 writes various patterns to each of the registers indicated in the following table. The pattern written to a register could be one of the following:

- Alternating ones and zeros (10101010)
- Alternating zeros and ones (01010101)

For each pattern and register, the register is written with the pattern. Next, the register is read, and then the data is checked against the pattern to ensure the register is functioning correctly.

The following table lists the registers tested by Subtest 1000 and their acronyms:

Table spu4000–5, Registers Tested by Subtest 1000

ACRONYM	REGISTER NAME
CPR	Control Panel Register
DCON	Diagnostic Connect Register
DCR	Diagnostic Control Register
IO	I/O Run Register
MEM	Memory Run Register
MEM_LOG	Memory Log Run Register
MISC_LOG	I/O Log Run Register
ODENA	Output Data Enable Register
PBUS_X	PBUS X CCU Run Register (4 CPU backplane only)
PBUS_Y	PBUS Y CCU Run Register
PCR	Physical Configuration Register
PROC_A	CPU A Run Register
PROC_B	CPU B Run Register
PROC_C	CPU C Run Register (4 CPU backplane only)
PROC_D	CPU D Run Register (4 CPU backplane only)
RHR	Run Halt Register
SRR	System Reset Register
TRR	Test Result Register

Subtest 1100, System Serial Number Validity

Subtest 1100 tests the system serial number and ensures that the machine class is within an allowable range and the serial number is not all zeros or all ones.

Subtest 1200, Service Processor Run-Arm Circuitry

Subtest 1200 tests the Service Processor run-arm circuitry. This subtest requires that at least one MCM be installed in the system. The current state of the Service Processor is saved before testing starts, and restored after the test completes.

The following checks are made on the bits in the Diagnostic Control Register, in this order:

- Busy bit clear before hitting go
- Run_arm bit clear before hitting go
- Ecr_not_zero bit set before hitting go
- Run_arm bit sets within a reasonable period of time
- Busy bit set after run_arm bit set
- Ecr_not_zero bit set after run_arm bit set
- Busy bit clears within a reasonable period of time
- Run_arm bit set when busy cleared
- Ecr_not_zero bit cleared when busy cleared

Subtest 1300, Service Processor Real Time Clock

Subtest 1300 checks the real time clock over a fraction of a second to ensure it is operational.

Class 2 Subtests, DBUS Interface

Class 2 subtests verify the ability of the Service Processor to interface with various FRUs through the Diagnostics Bus (DBUS), which is used to access scan rings.

Table spu4000–6, Class 2 Service Processor and CPU FRU Subtests

SUBTEST	TEST PERFORMED	PROBABLE FAULT LOCATION	OBJECT MODULE	SOURCE FILE	MAX TIME (min/sec)
2000	SP Scan Loopback	SP	*spu4000.t*	*spu_dbus_if.c*	0:01
2100	ASA Scan Interface	ASA, SP	*spu4000.t*	*dbus.c*	0:03
2101	IPA Scan Interface	IPA, SP	*spu4000.t*	*dbus.c*	0:03
2102	FUA Scan Interface	FUA, SP	*spu4000.t*	*dbus.c*	0:03
2103	DCA Scan Interface	DCA, SP	*spu4000.t*	*dbus.c*	0:03
2104	VCA Scan Interface	VCA, SP	*spu4000.t*	*dbus.c*	0:03
2105	VDA Scan Interface	VDA, SP	*spu4000.t*	*dbus.c*	0:03
2110	ASB Scan Interface	ASB, SP	*spu4000.t*	*dbus.c*	0:03
2111	IPB Scan Interface	IPB, SP	*spu4000.t*	*dbus.c*	0:03
2112	FUB Scan Interface	FUB, SP	*spu4000.t*	*dbus.c*	0:03
2113	DCB Scan Interface	DCB, SP	*spu4000.t*	*dbus.c*	0:03
2114	VCB Scan Interface	VCB, SP	*spu4000.t*	*dbus.c*	0:03
2115	VDB Scan Interface	VDB, SP	*spu4000.t*	*dbus.c*	0:03
2120	ASC Scan Interface	ASC, SP	*spu4000.t*	*dbus.c*	0:03
2121	IPC Scan Interface	IPC, SP	*spu4000.t*	*dbus.c*	0:03
2122	FUC Scan Interface	FUC, SP	*spu4000.t*	*dbus.c*	0:03
2123	DCC Scan Interface	DCC, SP	*spu4000.t*	*dbus.c*	0:03
2124	VCC Scan Interface	VCC, SP	*spu4000.t*	*dbus.c*	0:03
2125	VDC Scan Interface	VDC, SP	*spu4000.t*	*dbus.c*	0:03
2130	ASD Scan Interface	ASD, SP	*spu4000.t*	*dbus.c*	0:03
2131	IPD Scan Interface	IPD, SP	*spu4000.t*	*dbus.c*	0:03
2132	FUD Scan Interface	FUD, SP	*spu4000.t*	*dbus.c*	0:03
2133	DCD Scan Interface	DCD, SP	*spu4000.t*	*dbus.c*	0:03
2134	VCD Scan Interface	VCD, SP	*spu4000.t*	*dbus.c*	0:03
2135	VDD Scan Interface	VDD, SP	*spu4000.t*	*dbus.c*	0:03

Table spu4000–7, Class 2 Memory and I/O FRU Subtests

SUBTEST	TEST PERFORMED	PROBABLE FAULT LOCATION	OBJECT MODULE	SOURCE FILE	MAX TIME (min/sec)
2200	Memory 0 Even Scan Interface	ME0, SP	*spu4000.t*	*dbus.c*	0:03
2202	Memory 0 Even Log Scan Interface	ME0, SP	*spu4000.t*	*dbus.c*	0:03
2205	Memory 0 Odd Scan Interface	MO0, SP	*spu4000.t*	*dbus.c*	0:03
2207	Memory 0 Odd Log Scan Interface	MO0, SP	*spu4000.t*	*dbus.c*	0:03
2210	Memory 1 Even Scan Interface	ME1, SP	*spu4000.t*	*dbus.c*	0:03
2212	Memory 1 Even Log Scan Interface	ME1, SP	*spu4000.t*	*dbus.c*	0:03
2215	Memory 1 Odd Scan Interface	MO1, SP	*spu4000.t*	*dbus.c*	0:03
2217	Memory 1 Odd Log Scan Interface	MO1, SP	*spu4000.t*	*dbus.c*	0:03
2220	Memory 2 Even Scan Interface	ME2, SP	*spu4000.t*	*dbus.c*	0:03
2222	Memory 2 Even Log Scan Interface	ME2, SP	*spu4000.t*	*dbus.c*	0:03
2225	Memory 2 Odd Scan Interface	MO2, SP	*spu4000.t*	*dbus.c*	0:03
2227	Memory 2 Odd Log Scan Interface	MO2, SP	*spu4000.t*	*dbus.c*	0:03
2230	Memory 3 Even Scan Interface	ME3, SP	*spu4000.t*	*dbus.c*	0:03
2232	Memory 3 Even Log Scan Interface	ME3, SP	*spu4000.t*	*dbus.c*	0:03
2235	Memory 3 Odd Scan Interface	MO3, SP	*spu4000.t*	*dbus.c*	0:03
2237	Memory 3 Odd Log Scan Interface	MO3, SP	*spu4000.t*	*dbus.c*	0:03
2310	CPX Scan Interface	CPX, SP	*spu4000.t*	*dbus.c*	0:03
2312	CPX Log Scan Interface	CPX, SP	*spu4000.t*	*dbus.c*	0:03
2320	CUE Scan Interface	CUE, SP	*spu4000.t*	*dbus.c*	0:03
2322	CUE Log Scan Interface	CUE, SP	*spu4000.t*	*dbus.c*	0:03
2330	CUO Scan Interface	CUO, SP	*spu4000.t*	*dbus.c*	0:03
2340	PIA Scan Interface	PIA, SP	*spu4000.t*	*dbus.c*	0:03
2342	PIA Log Scan Interface	PIA, SP	*spu4000.t*	*dbus.c*	0:03
2344	PIA Auxiliary Scan Interface	PIA, SP	*spu4000.t*	*dbus.c*	0:03
2350	PIY Scan Interface	PIY, SP	*spu4000.t*	*dbus.c*	0:03
2362	PIY Log Scan Interface	PIY, SP	*spu4000.t*	*dbus.c*	0:03
2354	PIY Auxiliary Scan Interface	PIY, SP	*spu4000.t*	*dbus.c*	0:03
2360	PIX Scan Interface	PIX, SP	*spu4000.t*	*dbus.c*	0:03
2362	PIX Log Scan Interface	PIX, SP	*spu4000.t*	*dbus.c*	0:03
2364	PIX Auxiliary Scan Interface	PIX, SP	*spu4000.t*	*dbus.c*	0:03
2400	CCU 0 Scan Interface	CCU 0, SP	*spu4000.t*	*dbus.c*	0:03
2401	CCU 1 Scan Interface	CCU 1, SP	*spu4000.t*	*dbus.c*	0:03
2402	CCU 2 Scan Interface	CCU 2, SP	*spu4000.t*	*dbus.c*	0:03
2403	CCU 3 Scan Interface	CCU 3, SP	*spu4000.t*	*dbus.c*	0:03
2404	CCU 4 Scan Interface	CCU 4, SP	*spu4000.t*	*dbus.c*	0:03
2405	CCU 5 Scan Interface	CCU 5, SP	*spu4000.t*	*dbus.c*	0:03
2406	CCU 6 Scan Interface	CCU 6, SP	*spu4000.t*	*dbus.c*	0:03
2407	CCU 7 Scan Interface	CCU 7, SP	*spu4000.t*	*dbus.c*	0:03

Subtest 2000, Service Processor Scan Loopback

Subtest 2000 checks the Service Processor's scan interface to the DBUS via loopback. When in loopback mode, the data shifted out of one end of the Scan Data Register is shifted into the other end of the register. A single set bit is rotated through the Scan Data Register in both directions.

Each of the 16 bits in the Scan Data Register is individually tested, with the rotate count varied from 0 to 16, for each scan output data enable (odena).

Subtests 2100-2407

Subtests 2100 to 2407 are the first subtests which actually send data to, and receive data from, a FRU using scan.

These class 2 subtests test the scan ring interface between the system FRUs and the Service Processor by scanning the Least Significant Bits (LSB) and the Most Significant Bits (MSB) of each scan ring with a 0 and a 1, as possible.

The following tests are performed in the indicated order for class 2 subtests:

- Fully bidirectional scan ring
  1. LSB, with a 0
  2. LSB, with a 1
  3. MSB, with a 0
  4. MSB, with a 1

- Partly bidirectional scan rings
  1. LSB, with a 0
  2. LSB, with a 1

- Fully unidirectional scan rings - neither the LSB nor the MSB is testable

Class 3 Subtests, Board ID

Each Class 3 Subtest tests the COP of a particular FRU. A COP is a non-volatile RAM which contains information about a FRU, which may include the FRU type, revision level, and serial number. These tests attempt to read the COP for each FRU in the configuration. If a COP read is successful, the subtest then checks to see if the type of FRU is known and if it is allowed in the backplane slot where it is found.

Table spu4000–8, Class 3 CPU FRU Subtests

SUBTEST	TEST PERFORMED	PROBABLE FAULT LOCATION	OBJECT MODULE	SOURCE FILE	MAX TIME (min/sec)
3100	ASA COP	ASA, SP	*spu4000.t*	*id_integ.c*	0:04
3101	IPA COP	IPA, SP	*spu4000.t*	*id_integ.c*	0:04
3102	FUA COP	FUA, SP	*spu4000.t*	*id_integ.c*	0:04
3103	DCA COP	DCA, SP	*spu4000.t*	*id_integ.c*	0:04
3104	VCA COP	VCA, SP	*spu4000.t*	*id_integ.c*	0:04
3105	VDA COP	VDA, SP	*spu4000.t*	*id_integ.c*	0:04
3110	ASB COP	ASB, SP	*spu4000.t*	*id_integ.c*	0:04
3111	IPB COP	IPB, SP	*spu4000.t*	*id_integ.c*	0:04
3112	FUB COP	FUB, SP	*spu4000.t*	*id_integ.c*	0:04
3113	DCB COP	DCB, SP	*spu4000.t*	*id_integ.c*	0:04
3114	VCB COP	VCB, SP	*spu4000.t*	*id_integ.c*	0:04
3115	VDB COP	VDB, SP	*spu4000.t*	*id_integ.c*	0:04
3120	ASC COP	ASC, SP	*spu4000.t*	*id_integ.c*	0:04
3121	IPC COP	IPC, SP	*spu4000.t*	*id_integ.c*	0:04
3122	FUC COP	FUC, SP	*spu4000.t*	*id_integ.c*	0:04
3123	DCC COP	DCC, SP	*spu4000.t*	*id_integ.c*	0:04
3124	VCC COP	VCC, SP	*spu4000.t*	*id_integ.c*	0:04
3125	VDC COP	VDC, SP	*spu4000.t*	*id_integ.c*	0:04
3130	ASD COP	ASD, SP	*spu4000.t*	*id_integ.c*	0:04
3131	IPD COP	IPD, SP	*spu4000.t*	*id_integ.c*	0:04
3132	FUD COP	FUD, SP	*spu4000.t*	*id_integ.c*	0:04
3133	DCD COP	DCD, SP	*spu4000.t*	*id_integ.c*	0:04
3134	VCD COP	VCD, SP	*spu4000.t*	*id_integ.c*	0:04
3135	VDD COP	VDD, SP	*spu4000.t*	*id_integ.c*	0:04

Table spu4000–9, Class 3 Memory and I/O FRU Subtests

SUBTEST	TEST PERFORMED	PROBABLE FAULT LOCATION	OBJECT MODULE	SOURCE FILE	MAX TIME (min/sec)
3200	Memory 0 Even COP	ME0, SP	*spu4000.t*	*id_integ.c*	0:04
3205	Memory 0 Odd COP	MO0, SP	*spu4000.t*	*id_integ.c*	0:04
3210	Memory 1 Even COP	ME1, SP	*spu4000.t*	*id_integ.c*	0:04
3215	Memory 1 Odd COP	MO1, SP	*spu4000.t*	*id_integ.c*	0:04
3220	Memory 2 Even COP	ME2, SP	*spu4000.t*	*id_integ.c*	0:04
3225	Memory 2 Odd COP	MO2, SP	*spu4000.t*	*id_integ.c*	0:04
3230	Memory 3 Even COP	ME3, SP	*spu4000.t*	*id_integ.c*	0:04
3235	Memory 3 Odd COP	MO3, SP	*spu4000.t*	*id_integ.c*	0:04
3300	SP COP	SP	*spu4000.t*	*id_integ.c*	0:04
3310	CPX COP	CPX, SP	*spu4000.t*	*id_integ.c*	0:04
3320	CUE COP	CUE, SP	*spu4000.t*	*id_integ.c*	0:04
3330	CUE COP	CUO, SP	*spu4000.t*	*id_integ.c*	0:04
3340	PIA COP	PIA, SP	*spu4000.t*	*id_integ.c*	0:04
3350	PIY COP	PIY, SP	*spu4000.t*	*id_integ.c*	0:04
3360	PIX COP	PIX, SP	*spu4000.t*	*id_integ.c*	0:04
3400	CCU 0 COP	CCU 0, SP	*spu4000.t*	*id_integ.c*	0:04
3401	CCU 1 COP	CCU 1, SP	*spu4000.t*	*id_integ.c*	0:04
3402	CCU 2 COP	CCU 2, SP	*spu4000.t*	*id_integ.c*	0:04
3403	CCU 3 COP	CCU 3, SP	*spu4000.t*	*id_integ.c*	0:04
3404	CCU 4 COP	CCU 4, SP	*spu4000.t*	*id_integ.c*	0:04
3405	CCU 5 COP	CCU 5, SP	*spu4000.t*	*id_integ.c*	0:04
3406	CCU 6 COP	CCU 6, SP	*spu4000.t*	*id_integ.c*	0:04
3407	CCU 7 COP	CCU 7, SP	*spu4000.t*	*id_integ.c*	0:04

Class 4 Subtests, Scan Ring Integrity

Class 4 subtests test the scan rings on the system FRUs by scanning patterns through them in both directions, as possible. Each pattern consists of all bits either set or clear, except bit 0, which is inverted compared to the rest. If the data read from a scan ring does not match the pattern written to it, failure isolation may be attempted.

The following list describes the isolation attempted for each of the three types of scan rings:

- Fully bidirectional - full isolation attempted for any failure
- Partly bidirectional - isolation attempted only when the initial failure is in the LEFT direction
- Fully unidirectional - no isolation attempted (none possible)

If isolation is possible, it is attempted by scanning a number of bits into the ring in the same direction the failure was detected, then scanning the same number of bits out in the opposite direction. The number of bits scanned starts at one, and is incremented until either the failing bit is detected (the data read does not match the data written), or the shift count equals the length of the bidirectional part of the scan ring.

For any failure, an appropriate error message is printed, describing the failure and the results of any isolation attempts.

The following list shows the order of testing:

1. Any bidirectional portion of the ring under test is scanned in the LEFT direction, with all bits of the scan ring set, except bit zero
2. Any bidirectional portion of the ring under test is scanned in the LEFT direction, with only bit zero set
3. The ring under test is scanned in the RIGHT direction, with all bits of the scan ring set, except bit zero
4. The ring under test is scanned in the RIGHT direction, with only bit zero set

Table spu4000–10, Class 4 CPU FRU Subtests

SUBTEST	TEST PERFORMED	PROBABLE FAULT LOCATION	OBJECT MODULE	SOURCE FILE	MAX TIME (min/sec)
4100	ASA Scan Ring Integrity	ASA, SP	*spu4000.t*	*dbus.c*	0:03
4101	IPA Scan Ring Integrity	IPA, SP	*spu4000.t*	*dbus.c*	0:03
4102	FUA Scan Ring Integrity	FUA, SP	*spu4000.t*	*dbus.c*	0:03
4103	DCA Scan Ring Integrity	DCA, SP	*spu4000.t*	*dbus.c*	0:03
4104	VCA Scan Ring Integrity	VCA, SP	*spu4000.t*	*dbus.c*	0:03
4105	VDA Scan Ring Integrity	VDA, SP	*spu4000.t*	*dbus.c*	0:03
4106	VDA CMOS Scan Ring Integrity	VDA, SP	*spu4000.t*	*dbus.c*	0:03
4110	ASB Scan Ring Integrity	ASB, SP	*spu4000.t*	*dbus.c*	0:03
4111	IPB Scan Ring Integrity	IPB, SP	*spu4000.t*	*dbus.c*	0:03
4112	FUB Scan Ring Integrity	FUB, SP	*spu4000.t*	*dbus.c*	0:03
4113	DCB Scan Ring Integrity	DCB, SP	*spu4000.t*	*dbus.c*	0:03
4114	VCB Scan Ring Integrity	VCB, SP	*spu4000.t*	*dbus.c*	0:03
4115	VDB Scan Ring Integrity	VDB, SP	*spu4000.t*	*dbus.c*	0:03
4116	VDB CMOS Scan Ring Integrity	VDB, SP	*spu4000.t*	*dbus.c*	0:03
4120	ASC Scan Ring Integrity	ASC, SP	*spu4000.t*	*dbus.c*	0:03
4121	IPC Scan Ring Integrity	IPC, SP	*spu4000.t*	*dbus.c*	0:03
4122	FUC Scan Ring Integrity	FUC, SP	*spu4000.t*	*dbus.c*	0:03
4123	DCC Scan Ring Integrity	DCC, SP	*spu4000.t*	*dbus.c*	0:03
4124	VCC Scan Ring Integrity	VCC, SP	*spu4000.t*	*dbus.c*	0:03
4125	VDC Scan Ring Integrity	VDC, SP	*spu4000.t*	*dbus.c*	0:03
4126	VDC CMOS Scan Ring Integrity	VDC, SP	*spu4000.t*	*dbus.c*	0:03
4130	ASD Scan Ring Integrity	ASD, SP	*spu4000.t*	*dbus.c*	0:03
4131	IPD Scan Ring Integrity	IPD, SP	*spu4000.t*	*dbus.c*	0:03
4132	FUD Scan Ring Integrity	FUD, SP	*spu4000.t*	*dbus.c*	0:03
4133	DCD Scan Ring Integrity	DCD, SP	*spu4000.t*	*dbus.c*	0:03
4134	VCD Scan Ring Integrity	VCD, SP	*spu4000.t*	*dbus.c*	0:03
4135	VDD Scan Ring Integrity	VDD, SP	*spu4000.t*	*dbus.c*	0:03
4136	VDD CMOS Scan Ring Integrity	VDD, SP	*spu4000.t*	*dbus.c*	0:03

Table spu4000–11, Class 4 Memory and I/O FRU Subtests

SUBTEST	TEST PERFORMED	PROBABLE FAULT LOCATION	OBJECT MODULE	SOURCE FILE	MAX TIME (min/sec)
4200	Memory 0 Even Scan Ring Integrity	ME0, SP	*spu4000.t*	*dbus.c*	0:03
4202	Memory 0 Even Log Scan Ring Integrity	ME0, SP	*spu4000.t*	*dbus.c*	0:03
4205	Memory 0 Odd Scan Ring Integrity	MO0, SP	*spu4000.t*	*dbus.c*	0:03
4207	Memory 0 Odd Log Scan Ring Integrity	MO0, SP	*spu4000.t*	*dbus.c*	0:03
4210	Memory 1 Even Scan Ring Integrity	ME1, SP	*spu4000.t*	*dbus.c*	0:03
4212	Memory 1 Even Log Scan Ring Integrity	ME1, SP	*spu4000.t*	*dbus.c*	0:03
4215	Memory 1 Odd Scan Ring Integrity	MO1, SP	*spu4000.t*	*dbus.c*	0:03
4217	Memory 1 Odd Log Scan Ring Integrity	MO1, SP	*spu4000.t*	*dbus.c*	0:03
4220	Memory 2 Even Scan Ring Integrity	ME2, SP	*spu4000.t*	*dbus.c*	0:03
4222	Memory 2 Even Log Scan Ring Integrity	ME2, SP	*spu4000.t*	*dbus.c*	0:03
4225	Memory 2 Odd Scan Ring Integrity	MO2, SP	*spu4000.t*	*dbus.c*	0:03
4227	Memory 2 Odd Log Scan Ring Integrity	MO2, SP	*spu4000.t*	*dbus.c*	0:03
4230	Memory 3 Even Scan Ring Integrity	ME3, SP	*spu4000.t*	*dbus.c*	0:03
4232	Memory 3 Even Log Scan Ring Integrity	ME3, SP	*spu4000.t*	*dbus.c*	0:03
4235	Memory 3 Odd Scan Ring Integrity	MO3, SP	*spu4000.t*	*dbus.c*	0:03
4237	Memory 3 Odd Log Scan Ring Integrity	MO3, SP	*spu4000.t*	*dbus.c*	0:03
4310	CPX Scan Ring Integrity	CPX, SP	*spu4000.t*	*dbus.c*	0:03
4312	CPX Log Scan Ring Integrity	CPX, SP	*spu4000.t*	*dbus.c*	0:03
4320	CUE Scan Ring Integrity	CUE, SP	*spu4000.t*	*dbus.c*	0:03
4322	CUE Log Scan Ring Integrity	CUE, SP	*spu4000.t*	*dbus.c*	0:03
4330	CUO Scan Ring Integrity	CUO, SP	*spu4000.t*	*dbus.c*	0:03
4340	PIA Scan Ring Integrity	PIA, SP	*spu4000.t*	*dbus.c*	0:03
4342	PIA Log Scan Ring Integrity	PIA, SP	*spu4000.t*	*dbus.c*	0:03
4344	PIA Auxiliary Scan Ring Integrity	PIA, SP	*spu4000.t*	*dbus.c*	0:03
4350	PIY Scan Ring Integrity	PIY, SP	*spu4000.t*	*dbus.c*	0:03
4362	PIY Log Scan Ring Integrity	PIY, SP	*spu4000.t*	*dbus.c*	0:03
4354	PIY Auxiliary Scan Ring Integrity	PIY, SP	*spu4000.t*	*dbus.c*	0:03
4360	PIX Scan Ring Integrity	PIX, SP	*spu4000.t*	*dbus.c*	0:03
4362	PIX Log Scan Ring Integrity	PIX, SP	*spu4000.t*	*dbus.c*	0:03
4364	PIX Auxiliary Scan Ring Integrity	PIX, SP	*spu4000.t*	*dbus.c*	0:03
4400	CCU 0 Scan Ring Integrity	CCU 0, SP	*spu4000.t*	*dbus.c*	0:03
4401	CCU 1 Scan Ring Integrity	CCU 1, SP	*spu4000.t*	*dbus.c*	0:03
4402	CCU 2 Scan Ring Integrity	CCU 2, SP	*spu4000.t*	*dbus.c*	0:03
4403	CCU 3 Scan Ring Integrity	CCU 3, SP	*spu4000.t*	*dbus.c*	0:03
4404	CCU 4 Scan Ring Integrity	CCU 4, SP	*spu4000.t*	*dbus.c*	0:03
4405	CCU 5 Scan Ring Integrity	CCU 5, SP	*spu4000.t*	*dbus.c*	0:03
4406	CCU 6 Scan Ring Integrity	CCU 6, SP	*spu4000.t*	*dbus.c*	0:03
4407	CCU 7 Scan Ring Integrity	CCU 7, SP	*spu4000.t*	*dbus.c*	0:03

Class 5 Subtests, Hard Errors

Subtests 5100 to 5360, the hard error subtests, verify that each FRU in the system capable of generating a hard error can cause the generation of a hard-error interrupt on the Service Processor, and that the Service Processor can correctly determine the functional unit sending the error.

Scan operations are used to set and clear a hard error condition on the FRU under test.

Table spu4000–12, Class 5 Subtests

SUBTEST	TEST PERFORMED	PROBABLE FAULT LOCATION	OBJECT MODULE	SOURCE FILE	MAX TIME (min/sec)
5100	ASA Hard Error	ASA, SP	*spu4000.t*	*err_chk.c*	0:04
5101	IPA Hard Error	IPA, SP	*spu4000.t*	*err_chk.c*	0:04
5103	DCA Hard Error	DCA, SP	*spu4000.t*	*err_chk.c*	0:04
5104	VCA Hard Error	VCA, SP	*spu4000.t*	*err_chk.c*	0:04
5105	VDA Hard Error	VDA, SP	*spu4000.t*	*err_chk.c*	0:04
5110	ASB Hard Error	ASB, SP	*spu4000.t*	*err_chk.c*	0:04
5111	IPB Hard Error	IPB, SP	*spu4000.t*	*err_chk.c*	0:04
5113	DCB Hard Error	DCB, SP	*spu4000.t*	*err_chk.c*	0:04
5114	VCB Hard Error	VCB, SP	*spu4000.t*	*err_chk.c*	0:04
5115	VDB Hard Error	VDB, SP	*spu4000.t*	*err_chk.c*	0:04
5120	ASC Hard Error	ASC, SP	*spu4000.t*	*err_chk.c*	0:04
5121	IPC Hard Error	IPC, SP	*spu4000.t*	*err_chk.c*	0:04
5123	DCC Hard Error	DCC, SP	*spu4000.t*	*err_chk.c*	0:04
5124	VCC Hard Error	VCC, SP	*spu4000.t*	*err_chk.c*	0:04
5125	VDC Hard Error	VDC, SP	*spu4000.t*	*err_chk.c*	0:04
5130	ASD Hard Error	ASD, SP	*spu4000.t*	*err_chk.c*	0:04
5131	IPD Hard Error	IPD, SP	*spu4000.t*	*err_chk.c*	0:04
5133	DCD Hard Error	DCD, SP	*spu4000.t*	*err_chk.c*	0:04
5134	VCD Hard Error	VCD, SP	*spu4000.t*	*err_chk.c*	0:04
5135	VDD Hard Error	VDD, SP	*spu4000.t*	*err_chk.c*	0:04
5200	Memory 0 Even Hard Error	ME0, SP	*spu4000.t*	*err_chk.c*	0:04
5205	Memory 0 Odd Hard Error	MO0, SP	*spu4000.t*	*err_chk.c*	0:04
5210	Memory 1 Even Hard Error	ME1, SP	*spu4000.t*	*err_chk.c*	0:04
5215	Memory 1 Odd Hard Error	MO1, SP	*spu4000.t*	*err_chk.c*	0:04
5220	Memory 2 Even Hard Error	ME2, SP	*spu4000.t*	*err_chk.c*	0:04
5225	Memory 2 Odd Hard Error	MO2, SP	*spu4000.t*	*err_chk.c*	0:04
5230	Memory 3 Even Hard Error	ME3, SP	*spu4000.t*	*err_chk.c*	0:04
5235	Memory 3 Odd Hard Error	MO3, SP	*spu4000.t*	*err_chk.c*	0:04
5310	CPX Hard Error	CPX, SP	*spu4000.t*	*err_chk.c*	0:04
5320	CUE Hard Error	CUE, SP	*spu4000.t*	*err_chk.c*	0:04
5330	CUO Hard Error	CUO, SP	*spu4000.t*	*err_chk.c*	0:04
5340	PIA Hard Error	PIA, SP	*spu4000.t*	*err_chk.c*	0:04
5350	PIY Hard Error	PIY, SP	*spu4000.t*	*err_chk.c*	0:04
5360	PIX Hard Error	PIX, SP	*spu4000.t*	*err_chk.c*	0:04

Class 6 Subtests, Soft Errors

The Class 6 subtests verify that each FRU in the system capable of pulling a soft error can cause the generation of a soft error interrupt on the Service Processor, and that the Service Processor

correctly determines the functional unit sending the error. These subtests use scan operations to set and clear a soft error condition on the FRU or FRUs under test.

The CPX or CUE are possible sources of failure for certain subtests, since CPU soft errors go through the CPU Utility Board(s).

Table spu4000–13, Class 6 Subtests

SUBTEST	TEST PERFORMED	PROBABLE FAULT LOCATION	OBJECT MODULE	SOURCE FILE	MAX TIME (min/sec)
6103	DCA Soft Error	DCA, CPX, SP	*spu4000.t*	*err_chk.c*	0:04
6104	VCA Soft Error	VCA, CPX or CUE, SP	*spu4000.t*	*err_chk.c*	0:04
6113	DCB Soft Error	DCB, CPX, SP	*spu4000.t*	*err_chk.c*	0:04
6114	VCB Soft Error	VCB, CPX or CUE, SP	*spu4000.t*	*err_chk.c*	0:04
6124	VCC Soft Error	VCC, CPX or CUE, SP	*spu4000.t*	*err_chk.c*	0:04
6134	VCD Soft Error	VCD, CPX or CUE, SP	*spu4000.t*	*err_chk.c*	0:04
6200	Memory 0 Even Soft Error	ME0, SP	*spu4000.t*	*err_chk.c*	0:04
6205	Memory 0 Odd Soft Error	MO0, SP	*spu4000.t*	*err_chk.c*	0:04
6210	Memory 1 Even Soft Error	ME1, SP	*spu4000.t*	*err_chk.c*	0:04
6215	Memory 1 Odd Soft Error	MO1, SP	*spu4000.t*	*err_chk.c*	0:04
6220	Memory 2 Even Soft Error	ME2, SP	*spu4000.t*	*err_chk.c*	0:04
6225	Memory 2 Odd Soft Error	MO2, SP	*spu4000.t*	*err_chk.c*	0:04
6230	Memory 3 Even Soft Error	ME3, SP	*spu4000.t*	*err_chk.c*	0:04
6235	Memory 3 Odd Soft Error	MO3, SP	*spu4000.t*	*err_chk.c*	0:04
6310	CPX Soft Error	CPX, SP	*spu4000.t*	*err_chk.c*	0:04
6320	CUE Soft Error	CUE, SP	*spu4000.t*	*err_chk.c*	0:04
6340	PIA Soft Error	PIA, SP	*spu4000.t*	*err_chk.c*	0:04
6350	PIY Soft Error	PIY, SP	*spu4000.t*	*err_chk.c*	0:04
6360	PIX Soft Error	PIX, SP	*spu4000.t*	*err_chk.c*	0:04

Class 7 Subtests, EBUS Interface

Class 7 subtests verify the Service Processor's EBUS interface. These tests check the functionality of the EBUS window map RAM, population map RAM, controller, address translation hardware, and population map. Class 7 subtest numbers are grouped into three categories, determined by the second digit of the subtest number. The first group pattern tests RAM in the Service Processor's I/O space, the second checks the functionality of hardware on the Service Processor, and the third attempts EBUS transactions between the Service Processor and main memory.

Table spu4000–14, Class 7 Subtests

SUBTEST	TEST PERFORMED	PROBABLE FAULT LOCATION	OBJECT MODULE	SOURCE FILE	MAX TIME (min/sec)
7000	EBUS Window RAM	SP	*spu4000.t*	*ebus_ram.c*	0:03
7010	EBUS Population Map RAM	SP	*spu4000.t*	*ebus_popram.c*	0:55
7100	EBUS Controller	SP	*spu4000.t*	*ebus_ctrl.c*	0:53
7110	EBUS Population Map Verification	SP	*spu4000.t*	*ebus_poperr.c*	0:03
7200	EBUS Transfer Test	SP, PIA or PIY, ME0, MO0	*spu4000.t*	*ebus_trans.c*	0:01

Subtest 7000, EBUS Window RAM

Subtest 7000 pattern tests the EBUS window map RAM with alternating zeros and ones (01010101) and alternating ones and zeros (10101010). If these pattern tests pass, an address uniqueness test is executed. The EBUS window map RAM is also tested for parity by writing bad parity to it and reading the result, which should generate a parity error. The EBUS window map RAM is saved before the test starts, and restored after testing completes.

Subtest 7010, EBUS Population Map RAM

Subtest 7010 tests the EBUS population map RAM by completing the following steps:

1. A zero is written to all EBUS population map locations, and the entire EBUS population map is read to ensure that every bit is set
2. For each location of the EBUS population map, the following occurs:
   - The location is set to one
   - Every EBUS population map location is read and checked to insure that each bit contains the expected value
   - The location is cleared to zero
3. A one is written to all EBUS population map locations, and the entire EBUS population map is read to ensure that every bit is set
4. For each location of the EBUS population map, the following occurs:
   - The location is set to zero
   - Every EBUS population map location is read and checked to insure that each bit contains the expected value
   - The location is cleared to one

The EBUS population map RAM is saved before the test starts and restored after testing completes.

Subtest 7100, EBUS Controller

Subtest 7100 tests the EBUS controller on the Service Processor. For each main memory window, two types of checks are made.

First, each EBUS window is configured to map into page zero of ring zero of main memory. The EBUS population RAM is written to indicate all of main memory exists. The functionality of the valid bit is then checked by attempting an invalid access.

Second, for each EBUS window, illegal combinations of the following bits are checked to ensure they are flagged as invalid:

- lw_wr
- scrub
- tac
- msync
- I/O
- tas

It is important to note that this subtest does not test the functionality of these bits, but only the invalidity of certain combinations. The EBUS window and population maps are saved before the test starts, and restored after testing completes.

Subtest 7110, EBUS Population Map Verification

Subtest 7110 clears the EBUS population map to indicate that no main memory is available. A main memory read is then attempted to each of the 1024 4-megabyte blocks in main memory. Each read attempt is checked to ensure that it generates an Service Processor bus error, and does not attempt to actually read RAM memory. The EBUS window and population maps are saved before the test starts, and restored after testing completes.

Subtest 7200, EBUS Transfer Test

Subtest 7200 is designed to insure that the EBUS windows can read and write to main memory. The test performs the following procedure:

1. Save the EBUS window and population RAM
2. Set each EBUS window to the first available main memory block, but have each page invalid
3. Set the population map to indicate only the first main memory block exists
4. For each EBUS window the following is performed:
   - Set the valid bit
   - Write data to main memory
   - Read the data back and verify the result
   - Perform a Test and Set (TAS) operation and verify the result
   - Perform a Test and Clear (TAC) operation and verify the result
   - Perform a scrub operation
   - Perform a long word write and verify the result
   - Reset the valid bit
5. Restore the EBUS window and population RAM

Class 8 Subtests, Interrupt Bus Integrity

Class 8 subtests verify system interrupt bus operation. Both subtests set up the Service Processor as the receiver of interrupts. Subtest 8000 has the Service Processor interrupt itself. Subtest 8010 has the programmable interrupt timer on the CPU Utility Board(s) interrupt the Service Processor. Each of the Service Processor's eight system interrupts is tested, in ascending order. Subtest 8010 tests both the reception and acknowledgement of interrupts. First, the interrupt is disabled in the Interrupt Enable Register (IER), then allowed to interrupt the 68000 and be processed.

Table spu4000–15, Class 8 Subtests

SUBTEST	TEST PERFORMED	PROBABLE FAULT LOCATION	OBJECT MODULE	SOURCE FILE	MAX TIME (min/sec)
8000	SP - SP Interrupt	SP, PIA	*spu4000.t*	*sib_test.c*	0:01
8010	PIT - SP Interrupt	SP, CPX or CUE / CUO, PIA or PIY	*spu4000.t*	*sib_test.c*	0:22

Class 9 Subtests, Margin Test

The margin subtests verify the SCM or ESM interface, and the ability of the Service Processor to margin the power supplies and the system clock rate. Class 9 subtests numbers are grouped into three categories, determined by the second digit of the subtest number.

If the second digit is a 0, the subtests test communication between the SCM or ESM and the Service Processor.

If the second digit is a 1, the subtests margin power supplies and insure the resulting voltage is within tolerance. For power margining, the third digit of the subtest number indicates the voltage being margined, and the fourth digit indicates the margin level.

If the second digit of a subtest number is a 2, the subtest margins the system clock rate and insures the resulting frequency is within tolerance. For clock margining, the fourth digit of the subtest number indicates the margin level.

Table spu4000–16, Class 9 Subtests

SUBTEST	TEST PERFORMED	PROBABLE FAULT LOCATION	OBJECT MODULE	SOURCE FILE	MAX TIME (min/sec)
9010	Check SP - SCM / ESM BUS	SP, SCM or ESM	*spu4000.t*	*margin.c*	0:01
9020	Check Local and Remote	SP, SCM or ESM	*spu4000.t*	*margin.c*	0:01
9100	Check +5 at Nominal Margin	SP, SCM or ESM	*spu4000.t*	*margin.c*	0:02
9104	Check +5 at Low Margin	SP, SCM or ESM	*spu4000.t*	*margin.c*	0:02
9105	Check +5 at High Margin	SP, SCM or ESM	*spu4000.t*	*margin.c*	0:02
9110	Check +12	SP, SCM or ESM	*spu4000.t*	*margin.c*	0:02
9120	Check -5	SP, SCM or ESM	*spu4000.t*	*margin.c*	0:02
9130	Check -12	SP, SCM or ESM	*spu4000.t*	*margin.c*	0:02
9140	Check -4.5 at Nominal Margin	SP, SCM or ESM	*spu4000.t*	*margin.c*	0:02
9144	Check -4.5 at Low Margin	SP, SCM or ESM	*spu4000.t*	*margin.c*	0:02
9145	Check -4.5 at High Margin	SP, SCM or ESM	*spu4000.t*	*margin.c*	0:02
9150	Check -2 at Nominal Margin	SP, SCM or ESM	*spu4000.t*	*margin.c*	0:02
9154	Check -2 at Low Margin	SP, SCM or ESM	*spu4000.t*	*margin.c*	0:02
9155	Check -2 at High Margin	SP, SCM or ESM	*spu4000.t*	*margin.c*	0:02
9200	Check Clocks at Nominal Margin	SP, PIA or CUO	*spu4000.t*	*margin.c*	0:01
9204	Check Clocks at Low Margin	SP, PIA or CUO	*spu4000.t*	*margin.c*	0:01
9205	Check Clocks at High Margin	SP, PIA or CUO	*spu4000.t*	*margin.c*	0:01
9206	Check Clocks at Extended Margin	SP, PIA or CUO	*spu4000.t*	*margin.c*	0:01

Subtest 9010, Check Service Processor - SCM / ESM BUS

Subtest 9010 checks the Service Processor to SCM interface bus by writing 0x00 and 0xFF to the SCM and reading 0x00 and 0xFF from the SCM. If the SCM detects a data bus failure when written to, it will turn off system power and display an error message on the front panel hex display. Refer to the *CONVEX System Manager's Guide* for more information.

Subtest 9020, Check Local and Remote

Subtest 9020 checks the SCM's ability to indicate local and remote operation. By giving commands to the SCM, the local and remote signals are toggled and verified on the Service Processor.

Subtests 9100-9155

These subtests check each power supply against specified maximum and minimum values for all possible margins.

- The nominal margin is checked from 97 to 103% of the voltage being tested
- The lower margin is checked from 93 to 100% of the voltage being tested
- The upper margin is checked from 100 to 107% of the voltage being tested

The +5, -2, and -4.5 voltage levels are checked at nominal, low, and high margins. The +12, -5, and -12 voltage levels are not marginable and for this reason are only checked for nominal range.

The marginable supplies are left in the margin state in which they entered the subtest.

Subtests 9200-9206

These subtests verify the ability of the Service Processor to margin the clocks. The Clock Frequency Register is used to select the clock frequency, and the interval timer is utilized to determine the actual frequency. For system clock margining, the system clock is checked to ensure it is within 5 percent of each of the following margins:

- Subtest 9200 — Nominal Margin (25 or 18.4 MHz)
- Subtest 9204 — Low Margin (22.9, 16.5 or 15 MHz)
- Subtest 9205 — High Margin (27.5 or 20.2 MHz)
- Subtest 9206 — Extended Margin (12.5 or 9.2 MHz)

The clock rate is restored to its initial rate after testing.

Test Error Messages

The following sections provide a complete description of the Service Processor Interface Test error messages. Sections are provided for each group of subtests, and their corresponding error messages. When any subtest fails, a halt code is not stored in a register; rather, a descriptive error message is displayed.

Initialization and General Messages

The following test message could result if the file */mnt/usr/scn/scn_rings* does not exist, or is corrupt:

```
spu4000:   scn_init call failed - test aborted
```

The following error message could result if the system serial number is invalid, which could result if this test was executed on the wrong type of system:

```
spu4000:   unknown machine class - test aborted
```

The following error message could result from an error in the invocation sequence:

```
spu4000:   spu4000 has no fault analyzer - test aborted
```

The following test messages can appear at any time during testing:

```
received unexpected bus error - test aborted
received segmentation violation - test aborted
```

Subtest 1000 Errors

The following error message could result from an error in this subtest:

```
service processor register failed pattern test
register: IIII   pattern: JJJJ
address: 0xKKKKKK   actual: 0xLLLL   expected: 0xMMMM
```

where:

IIII is the name of the register which failed
JJJJ is the pattern which failed
KKKKKK is the address of the failing register, in hex
LLLL is the actual value read from the register, in hex
MMMM is the expected value, in hex

Subtest 1100 Errors

The following error message could result from an error in this subtest:

```
system serial number invalid
actual: 0xIIII   expected: 0x2xxx, 0x3xxx, 0x4xxx, or 0x5xxx
```

where:

IIII is the actual system serial number, in hex
xxx is any hex number

Subtest 1200 Errors

Any of the following error messages could result from an error in this subtest:

```
busy bit set before hitting go
run_arm bit set before hitting go
ecr_not_zero bit clear before hitting go
run_arm bit did not set before timeout
busy bit clear after run_arm bit set
ecr_not_zero bit clear after run_arm bit set
busy bit did not clear before timeout
run_arm bit cleared when busy cleared
ecr_not_zero bit did not clear when busy cleared
```

Subtest 1300 Errors

The following error message could result from an error in this subtest:

```
service processor real time clock not counting
```

Subtest 2000 Errors

The following error message could result from an error in subtest 200:

```
scan loopback failed - direction: IIII
shift count: 0xJJ   actual: 0xKKKK   expected: 0xLLLL  odena: 0xM
```

where:

IIII is one of:
RIGHT
LEFT
JJ is the shift count, in hex
KKKK is the expected SDR contents, in hex
LLLL is the actual SDR contents, in hex
M is the odena value used, in hex

Subtests 2100-2407 and 4100-4407 Errors

The following error messages may occur in class 2 or class 4 subtests:

```
total ring length is shorter than bidirectional part
ring_length: 0xIIII (JJJJ)  bi_length: 0xKKKK (LLLL)
```

where:

IIII is the total ring length, in hex
JJJJ is the total ring length, in decimal
KKKK is the bidirectional ring length, in hex
LLLL is the bidirectional ring length, in decimal

or:

```
Unable to get ring into scannable state
```

or:

```
scn_rd returned error status II
```

or:

```
scn_wr returned error status II
```

where:

II is the error code

For Class 2 subtests only, the following error messages may appear:

```
LSB failure:   wrote I, read J
```

or:

```
MSB failure:   wrote I, read J
```

where:

I is the binary value written to the ring
J is the binary value read from the ring

For Class 4 subtests only, the following error messages may appear:

```
scan ring failure in bidirection part of scan ring isolated
failing bit (0 - 0xII (0 - JJ)): 0xKK (LL)
failure detected when writing and reading in the LEFT direction
isolation done by writing LEFT, reading RIGHT

expected buffer:
0xMM (NN) bit in buffer - 0xWW (XX) bits in MSW - LSB is lower right bit
<buffer, displayed as 16 bit hex values>

actual buffer:
0xMM (NN) bit in buffer - 0xWW (XX) bits in MSW - LSB is lower right bit
<buffer, displayed as 16 bit hex values>
```

or:

```
scan ring failure isolated
failing bit (0 - 0xII (0 - JJ)): 0xKK (LL)
failure detected when writing and reading in the RIGHT direction
isolation done by writing RIGHT, reading LEFT

expected buffer:
0xMM (NN) bit in buffer - 0xWW (XX) bits in MSW - LSB is lower right bit
<buffer, displayed as 16 bit hex values>

actual buffer:
0xMM (NN) bit in buffer - 0xWW (XX) bits in MSW - LSB is lower right bit
<buffer, displayed as 16 bit hex values>
```

where:

II is the MSB, in hex
JJ is the MSB, in decimal
KK is the failing bit, in hex
LL is the failing bit, in decimal
MM is the number of bits in the buffer, in hex
NN is the number of bits in the buffer, in decimal
WW is the number of bits displayed in the MSW, in hex
XX is the number of bits displayed in the MSW, in decimal

or:

```
unable to isolate scan ring failure in bidirection part of scan ring
failure detected when writing and reading in the LEFT direction
isolation attempted by writing LEFT, reading RIGHT
```

or:

```
unable to isolate scan ring failure
failure detected when writing and reading in the RIGHT direction
isolation attempted by writing RIGHT, reading LEFT
```

Subtests 3100-3407 Errors

The following error messages could result from an error in a Class 3 subtest:

```
unable to open data base file /mnt/usr/lib/DB_cop
```

or:

```
unable to read cop for slot IIII
```

or:

```
type of board installed in slot IIII unknown - port number JJJJ
```

or:

```
board type KKKK (part number JJJJ) not allowed in slot IIII
```

where:

IIII is the name of the slot under test
JJJJ is the port number read from the cop
KKKK is the board type read from file */mnt/usr/lib/DB_cop*

Subtests 5100-5360 & 6103-6360 Errors

Any of the following error messages could result if an error occurred while attempting to modify the state of an error:

```
mod_error returned I when attempting hard clear of JJJJ
mod_error returned I when attempting soft clear of JJJJ
mod_error returned I on clear of JJJJ
mod_error returned I on set of JJJJ
```

where:

JJJJ is the name of the ring which failed
I is on of the following error codes:

1	Illegal ring for specified function
8	Field not found in scan ring
4	Illegal function
16	Ring has zero length
-1	Scan error occurred

Any of the following error messages could result if an operation to modify the state of an error did not have the desired affect:

```
unable to disable all hard errors
actual ESR: 0xIIII   actual SEL: 0xJJJJ
```

or:

```
unable to disable all soft errors
actual ESR: 0xIIII   actual SEL: 0xJJJJ
```

or:

```
hard error set after clearing error
actual ESR: 0xIIII   actual SEL: 0xJJJJ
```

or:

```
soft error set after clearing error
actual ESR: 0xIIII   actual SEL: 0xJJJJ
```

or:

```
hard error not set when expected
actual ESR: 0xIIII   actual SEL: 0xJJJJ
```

or:

```
soft error not set when expected
actual ESR: 0xIIII   actual SEL: 0xJJJJ
```

or:

```
hard error set when only soft should be
actual ESR: 0xIIII   actual SEL: 0xJJJJ
```

or:

```
soft error set when only hard should be
actual ESR: 0xIIII   actual SEL: 0xJJJJ
```

or:

```
hard error set after (re)clearing error
actual ESR: 0xIIII   actual SEL: 0xJJJJ
```

or:

```
soft error set after (re)clearing error
actual ESR: 0xIIII   actual SEL: 0xJJJJ
```

where:

IIII is the actual ESR value, in hex
JJJJ is the actual SEL value, in hex

Subtest 7000 Errors

Any of the following error messages could result from an error in this subtest:

```
ebus ram data rebound failed - pattern: IIII
```

or:

```
ebus ram parity check failed - pattern: IIII
```

where:

IIII is the pattern which failed

Subtest 7010 Errors

The following error message could result from an error in this subtest:

```
ebus population map ram error detected
address: 0xIIIIII   actual: 0xJJJJ   expected: 0xKKKK
```

where:

IIIIII is the Service Processor I/O address of the population RAM
JJJJ is the actual value, in hex
KKKK is the expected value, in hex

Subtest 7100 Errors

The following error message could result from an error in this subtest:

```
ebus controller functionality error
window map address: 0xIIIIII   window map data: 0xJJJJJJJJ
actual BSR: 0xKKKK   expected BSR: 0xLLLL
```

where:

IIIIII is the Service Processor I/O address of the window map, in hex
JJJJJJJJ is the contents of the window map, in hex
KKKK is the actual Bus error Register, in hex
LLLL is the expected Bus Error Register, in hex

Subtest 7110 Errors

The following error message could result from an error in this subtest:

```
ebus population map functionality error detected
expected bus error did not occur
window map address: 0xIIIIII   window map data: 0xJJJJJJJJ
```

where:

IIIIII is the Service Processor I/O address of the window map, in hex
JJJJJJJJ is the contents of the window map, in hex

Subtest 7200 Errors

The following error messages could result from errors in this subtest:

```
ebus operation caused unexpected bus error
window map address: 0xIIIIII   window map data: 0xJJJJJJJJ
```

or:

```
ebus operation caused segmentation violation.
window map address: 0xIIIIII   window map data: 0xJJJJJJJJ
data written: 0xKKKK   data read: 0xLLLL
```

or:

```
normal (16 bit) ebus read or write failed
window map address: 0xIIIIII   window map data: 0xJJJJJJJJ
data written: 0xKKKK   data read: 0xLLLL
```

or:

```
ebus test and set (TAS) operation failed
window map address: 0xIIIIII   window map data: 0xJJJJJJJJ
data read during TAS: 0xMMMM   data read from memory after TAS: 0xNNNN
```

or:

```
ebus test and clear (TAC) operation failed
window map address: 0xIIIIII   window map data: 0xJJJJJJJJ
data read during TAC: 0xWWWW   data read from memory after TAC: 0xXXXX
```

or:

```
ebus long word write failed
window map address: 0xIIIIII   window map data: 0xJJJJJJJJ
actual: 0xYYYYYYYYYYYYYYYY   expected: 0xZZZZZZZZZZZZZZZZ
```

where:

IIIIII is the Service Processor I/O address of the window map, in hex
JJJJJJJJ is the contents of the window map, in hex
KKKK is the data written to memory, in hex
LLLL is the data read from memory, in hex
MMMM is the data read during a TAS operation, in hex
NNNN is the data read during a TAC operation, in hex
WWWW is the actual value of the memory location, in hex
XXXX is the actual value of the memory location, in hex
YYYYYYYYYYYYYYYY is the data read from memory, in hex
ZZZZZZZZZZZZZZZZ is the data written to memory, in hex

Subtests 8000-8010 Errors

The following error messages could result from errors in all Class 8 subtests: 801:

```
received unexpected interrupt 0xII while testing interrupt 0xJJ
IER: 0xKKKKKKKK
```

where:

II is the interrupt which was received, in hex
JJ is the interrupt under test, in hex
KKKKKKKK is the contents of the IER, in hex

or:

```
ISR not as expected while testing interrupt 0xII
actual: 0xJJJJJJJJ   expected: 0xKKKKKKKK
```

where:

II is the interrupt under test, in hex
JJJJJJJJ is the actual contents of the ISR, in hex
KKKKKKKK is the expected contents of the ISR, in hex

or:

```
incorrect interrupt received
received: 0xII   expected: 0xJJ
```

where:

II is the interrupt which was received, in hex
JJ is the interrupt which was expected, in hex

The following error message could result from an error in subtest 8010 only:

```
PIT never interrupted SP while testing interrupt 0xII
```

where:

II is the interrupt under test, in hex

Class 9 Subtest Errors

In addition to the errors described below for individual or subsets of Class 9 subtests, the following error messages may occur during any Class 9 subtest:

```
system monitor command failed
address: 0xIIIIII   operation: JJJJ
```

where:

IIIIII is the failing Service Processor I/O address, in hex
JJJJ is one of:

```
unknown
read
write
```

or:

```
system monitor command failed during call to IIII
```

where:

IIII is one of:

```
meas_voltage ()
pwr_marg_num ()
pwr_marg ()
```

Subtest 9010 Errors

The following error message could result from and error in this subtest:

```
read 0xII from system monitor, expected 0xJJ
```

where:

II is the data read from the SCM or ESM, in hex
JJ is expected data, in hex

Subtest 9020 Errors

Any combination of the following error messages could result from an error in this subtest:

```
wouldn't go into local mode, 1st attempt - cpr: 0xIIII
incorrectly in remote mode, 1st attempt - cpr: 0xIIII
wouldn't go into remote mode, 1st attempt - cpr: 0xIIII
incorrectly in local mode, 1st attempt - cpr: 0xIIII
wouldn't go into local mode, 2nd attempt - cpr: 0xIIII
incorrectly in remote mode, 2nd attempt - cpr: 0xIIII
wouldn't go into remote mode, 2nd attempt - cpr: 0xIIII
incorrectly in local mode, 2nd attempt - cpr: 0xIIII
couldn't restore local mode - cpr: 0xIIII
incorrectly in remote mode after restore - cpr: 0xIIII
couldn't restore remote mode - cpr: 0xIIII
incorrectly in local mode after restore - cpr: 0xIIII
```

where:

IIII is the actual control panel register value

Subtests 9100-9155 Errors

The following error message could result from an error in these subtests:

```
IIII voltage out of tolerance at JJJJ margin
actual: KKK.KK   minimum: LLL.LL   maximum: MMM.MM
```

where:

IIII is one of:
- +5
- +12
- -5
- -12
- -4.5
- -2

JJJJ is one of:
- nominal
- lower
- upper

KKK.KK is the actual voltage, in VDC
LLL.LL is the minimum allowable voltage, in VDC
MMM.MM is the maximum allowable voltage, in VDC

Subtest 9200-9206 Errors

These subtests verify the ability of the Service Processor to margin the clocks. The system clock is checked to ensure it is within five percent of a nominal, low, upper, and extended margin. The following error message could result:

```
clock out of tolerance at IIII margin
actual: JJJ.JJ   minimum: KKK.KK   maximum: LLL.LL
```

where:

IIII is one of:
- nominal
- lower
- upper

JJJ.JJ is the actual frequency, in MHz
KKK.KK is the minimum allowable frequency, in MHz
LLL.LL is the maximum allowable frequency, in MHz

SPU UNIX Error Messages

For a common list of possible error messages resulting from SPU UNIX, refer to the appendix on SPU UNIX Error Messages.

THIS PAGE INTENTIONALLY LEFT BLANK

pia4000

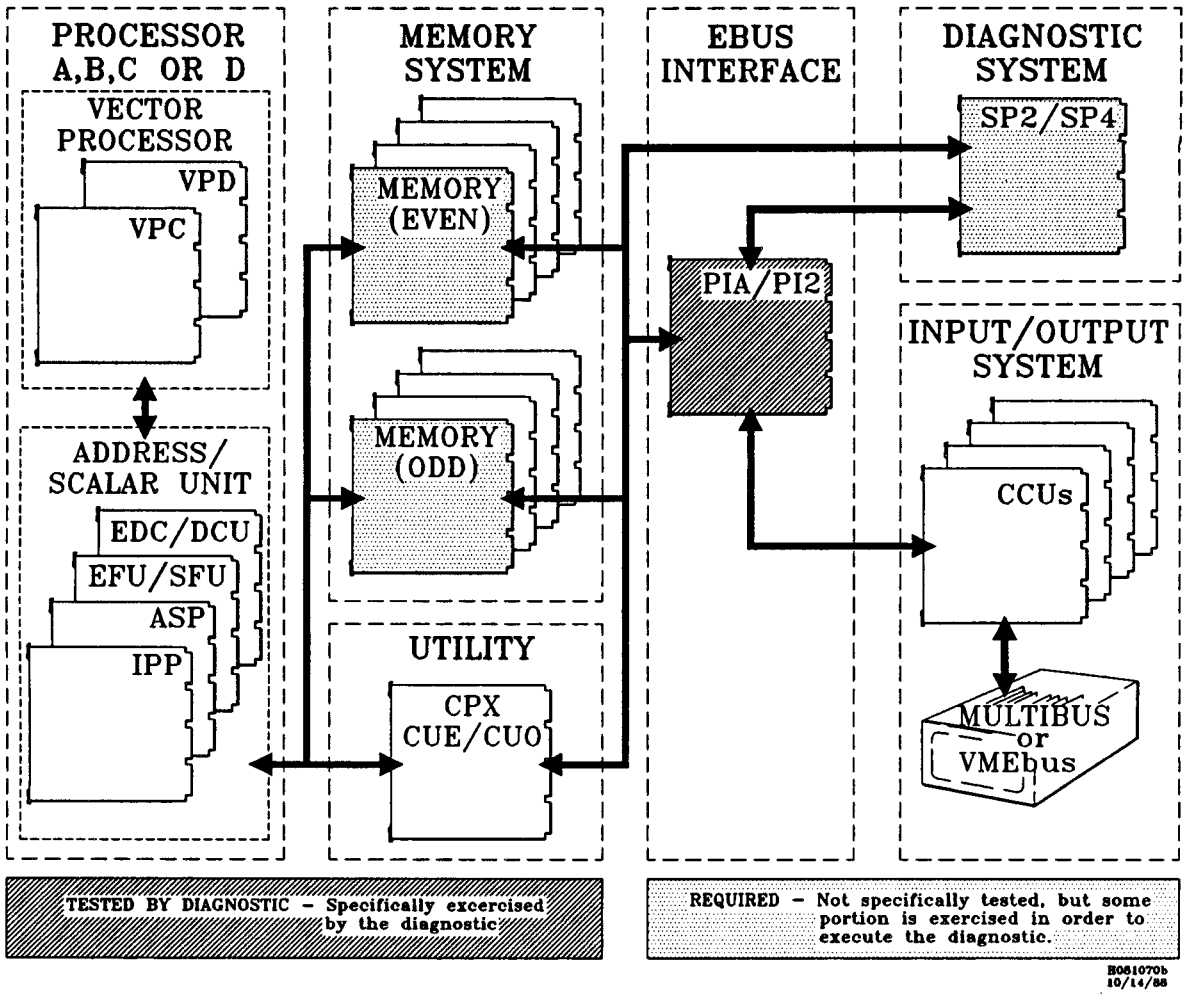
PIA Functional Test

Overview

The *pia4000* test verifies the functionality of the PBUS Interface Adapter (PIA) board. The PIA board functions as a bus converter, converting the old peripheral bus (PBUS) to the EBUS. The PIA is designed to allow existing Channel Control Units (CCUs) to continue normal operation without redesigning each CCU.

> **NOTE**
>
> This test is designed for testing the PBUS Interface Adapter (PIA). For testing the PBUS Interface Adapter 2 (PI2), use the *pi2_4000* diagnostic test.

Figure pia4000–1, Functional Areas Tested by *pia4000*



Prerequisites and Required Equipment

The boards listed in the following table must be operational. Also shown in the following table are the tests used to verify the required boards. No additional equipment is required to run this test.

Table pia4000–1, Required Functional Boards

BOARD	TEST TO VERIFY
Service Processor	spu1000, spu4000
Memory system	mem4000

NOTE

Memory system consists of a minimum of one pair of memory boards (one even and one odd).

Test Invocation

To invoke the *pia4000* test, use the procedure shown in the following figure. All responses in **boldface** are entered by the user. The prompts and responses in the following figure would appear sequentially on the screen, one line at a time. All the prompts and responses are shown in one figure for convenience.

CAUTION

The invocation sequence shown in the following figure is the typical invocation sequence. However, the *initall* utility must be executed in some cases. If the system has just been powered up, if *mem4000* was executed with failures, or if *spu4000* was executed, then *initall* must be executed prior to any test execution. Failure to execute *initall* in these circumstances could result in invalid test results.

NOTE

Running the *initall* utility requires two to three minutes to execute depending on if the control stores have been previously loaded. However, it is suggested that *initall* be executed in the event that the state of the system is unknown.

Figure pia4000–2, Test Invocation Sequence

```
(spu)>  cd /mnt/test
(spu)>  sysreset
(spu)>  dshell

CONVEX DIAGNOSTIC SHELL

: test pia4000 [-c [class number(s)]] [-s [subtest number(s)]] [+> filename]
```

NOTE

After entering **dshell**, specific *dshell* parameters may be changed. Please refer to the "Dshell and Iscan Overview" chapter of this manual for more information on *dshell*.

Entering only (*pia4000*) executes all *pia4000* subtests sequentially. Execute a specific class(es) of subtest(s) or one or more individual subtests by using the **-c** or **-s** options, respectively. Detailed information for using these options can be found in the "Dshell and Iscan Overview" chapter of this manual. The **[+>** *filename*] option allows the test results to be appended to *filename*.

Class Description

The PIA functional subtests encompass one class. The following table identifies each of the subtests within this class. An abbreviated description of the hardware tested by each subtest is given in the "TEST PERFORMED" column.

Table pia4000–2, PIA Functional Subtests

SUBTEST	TEST PERFORMED	DEFAULT TIME (min/sec)
50	PBUS Integrity Test	0:01
52	LPIA Freeze on Error Test	0:00
53	Reset/Align of Scan OK Bits	0:00
54	PIA/CCU Scan OK Sequence Test (Single clock steps)	0:00
55	PIA/CCU Scan OK Sequence Test (Multiclock steps)	0:00
100	PBUS Parity Checker Test	0:01
101	EBUS Parity Checker Test	0:02
150	Read Queue Register File Test	0:05
200	Read Queue Fill/Empty Logic Test	0:01
250	Write Control Queue Pattern Test	0:05
300	Write Control Queue Fill/Empty Logic Test	0:08
350	PCM RAM Test	0:34
400	NPM Edge Test	0:12
451	Bad PBUS Header Detection Test	2:28
550	Interrupt Arbiter State Machine Test	0:00
600	Forced PBUS Cycle Test:0:29	
650	SP2-EBUS Arbitration Test	0:04

NOTE

Some of the times above may vary plus or minus one second. Some times are zero because the subtests run faster than they can be clocked. This test has no timeout provisions and if any subtest takes substantially more time than is listed, the test may have failed.

Subtest Descriptions

The following subtests are described for the type of function performed by the subtest. If there are no errors detected during the test, the response for each subtest is **passed**. If any of the following subtests fail, the PIA board must be replaced.

> NOTE
>
> Accomplish **sysreset** before attempting to run *pia4000*. If the **sysreset** is not performed, some of the subtests could indicate a malfunction, when in fact none exists.

Subtest 50, PBUS Integrity Test

Subtest 50 is used to test the integrity of the path between input and output registers on the PBUS. The subtest first attempts to reset the PIA and check the PBUS for any indication of activity. If the PBUS is active, the subtest will abort immediately. To test the PBUS integrity, information is transferred to the PBREGOUT register and then read from the PBREGIN register and compared. The data should match.

Subtest 52, LPIA Freeze on Error Test

Subtest 52 is used to verify the loading of error conditions from the main ring to the log ring and to verify the correct functionality of the log ring. The test will first load the same type of data normally captured in an error situation, causing a bit to indicate that an error condition exists. When given a clock, this procedure will cause data to come through the main ring to the log ring. The log ring should then lock up. When additional clocks are exercised, the new data should not enter into the log ring.

Subtest 53, Reset/Align of Scan OK Bits

The scan bits test, Subtest 53, checks the status return by performing a *scan reset*. This subtest indicates whether the reset failed or if the bits are not scannable.

Subtest 54, PIA/CCU Scan OK Sequence Test (Single Clock)

Subtest 54 is used to check the state machine hardware that generates the two sets of clocks. To perform this test, an attempt is made to align the clocks to determine if components are scannable. This test also clocks the board and checks that the state transition in the clock sequential machine are correct.

Subtest 55, PIA/CCU Scan OK Sequence Test (Multi-Clock)

Subtest 55 performs the same functions as Subtest 54 and the same error messages are displayed. The difference between this subtest and Subtest 54 is the use of multiple clocks per step rather than a single clock per step.

Subtest 100, PBUS Parity Checker Test

Subtest 100 verifies the parity checker chip on the PBUS to determine the proper functionality. By entering a good set of values in the parity field and data field, the parity checker should indicate valid parity. If the parity checker indicates bad parity, then spurious bad parity is being detected.

A second set of invalid values is then entered in the fields that should cause an indication of bad parity by the parity checker. If bad parity is not detected during the test, then bad parity is being undetected. This subtest determines any malfunction resulting from failure to detect bad parity or an invalid detection of parity.

Subtest 101, EBUS Parity Checker Test

Subtest 101 is the same as Subtest 100 for the PBUS parity checker, except the test is performed against the EBUS parity checker. Error messages are identical as those displayed for the PBUS.

Subtest 150, Read Queue Register File Test

Subtest 150 will verify that the read queue register file does not corrupt data placed into it. The first phase of the test will attempt to reset the PIA. If reset is successful, the second portion of the subtest is initiated to test whether the data written to the read queue register file matches the data read back.

Subtest 200, Read Queue Fill/Empty Logic Test

The purpose of Subtest 200 is to test manipulation of the control logic via scan ring to fill or empty the queue and read the signals indicating that the queue is filled or emptied. This subtest sets up the condition for a memory read transfer, then clocks the read queue data for ten steps, verifies the header and trailer pointers for proper locations, and checks the queue counter for specific values.

Subtest 250, Write Control Queue Pattern Test

Subtest 250 verifies that the write control queue register file does not corrupt data entered. Bit patterns are written to the register file and read back to check data integrity.

Subtest 300, Write Control Queue Fill/Empty Logic Test

This subtest checks the fill and empty logic for the write control queue. This test manipulates the control logic via the scan ring to fill and empty the queue and read the signal(s) indicating that the queue has been filled or emptied.

Subtest 350, PCM RAM Test

Subtest 350 is used to verify that the Processor Configuration Memory (PCM) RAM can be properly loaded and that data integrity can be maintained. This subtest loads data and performs a pattern test on the PCM RAM chip using the manipulation of the scan ring.

Subtest 400, NPM Edge Test

Subtest 400 is used to verify that addresses on the edge of Non-Processor Memory (NPM) can be accessed without triggering an NPM fault and verify that crossing an NPM boundary does in fact generate a fault. This test manipulates the scan rings to generate the appropriate address and state signals to test the generation of the NPM fault signal.

Subtest 451, Bad PBUS Header Detection Test

Subtest 451 is used to determine whether an invalid PBUS transaction will be flagged. This test manipulates data for bad header parity on an otherwise good header, puts out an illegal header for detection, an I/O read with a bad start address, and a test and clear (TAC) or test and set (TAS) instruction with a bad byte count. The scope of this test is to start fake PBUS transfers and try to get the hardware to detect that a bad header is put out.

Subtest 550, Interrupt Arbiter State Machine Test

Subtest 550 is used to verify the Interrupt Arbiter (I-Arb) state machine. This test uses the scan ring to manipulate the inputs to the state machine consistent with a given bus cycle type. It then walks the machine through these states varying the inputs accordingly and checks that the next state or output generation is correct by reading from the scan ring.

Subtest 600, Forced PBUS Cycle Test

Subtest 600 exercises the various PBUS cycles by running a fake transfer of each type and checking the results. The test manipulates the scan ring and control bits in the simulation of a PBUS cycle. The transfers include a read, write, test and set, test and clear, and then checks that the proper signals transfer to the EBUS side.

Subtest 650, SP2-EBUS Arbitration Test

Subtest 650 is used to verify that the EBUS arbitration logic allows the SPU2 and CCUs to gain access to the EBUS. This test will scan in simulated requests and verify that the proper response occurs.

Subtest Error Descriptions

The following paragraphs indicate the error messages that may be returned when executing *pia4000*. Each error is described by the subtest that invokes the error message. If any of these errors are discovered by field engineers when troubleshooting, the corrective action is to replace the PIA board.

> **NOTE**
>
> The question marks (??) used in the following error messages indicate that values are output from the execution of the subtest.

Subtest 50 Errors

The PBUS must be inactive when executing the *pia4000* functional test. If the PBUS is active upon attempting to run this subtest, the following error message displays:

```
P-bus not inactive after reset
parity=?? data=??
```

During the integrity testing of the PBUS, if information transferred into the PBREGOUT register and read from the PBREGIN register conflicts, the following error message displays:

```
Bus xfer failure
Patterns applied at PBREGOUT
       DATA HI:  ??
       DATA LO:  ??
       PARITY:  ??
Patterns removed at PBREGIN
       DATA HI:  ??
       DATA LO:  ??
       PARITY:  ??
```

Subtest 52 Errors

This subtest attempts to load data from the main ring into the log ring, and determine if the log ring functions properly. An improper transfer of data or nontransfer of data creates an error condition and displays the following message:

```
Log ring FAILURE: unable to load data from main ring
Unable to continue with test
```

A second test within this subtest determines if the log ring is functioning as designed. It determines if the log ring can be locked to prevent entering additional data into the log ring. If for some reason the log ring is not locking properly, the following error message displays:

```
Error condition does not freeze log ring
Expected: ??  actual: ??
```

Subtest 53 Errors

The following error message displays when this subtest encounters problems attempting to align the log rings:

```
Unable to reset---scn_reset fails, (reason for error here)
```

The possible reasons displaying with the above message are:

```
Reset pia not scannable

Reset ccu not scannable
```

Subtest 54 Errors

The first error message to display from subtest 54 is the result of the clock hardware failing to properly reset the clocks:

```
Unable to reset---clock hardware failure
```

This subtest also checks if the state of the machine is functioning properly. The following error message displays the number of clocks from reset at the time of error detection. The initial state of the clocks before attempting alignment, and the current and expected state of clocks during the test also display. The PIA Scan OK Bit and CCU Scan OK Bit display with the error message for the current and expected state. Current and expected values display as HI or LO.

```
Bad state transition detected
       We are ?? clocks away from reset
       Initial state of two_clock was ??
       two_clock: current= ??  expected= ??
       pia_scok: current= ??  expected= ??
       ccu_scok: current= ??  expected= ??
```

Subtest 55 Errors

Subtest 55 performs the same functions as subtest 54 with identical error messages. The difference between this subtest and Subtest 54 is the use of multiple clocks rather than a single clock.

Subtest 100 Errors

When checking the PBUS for invalid parity detection or spurious parity detection, the following error messages display:

```
Parity checker failure (spurious bad parity)
Failed patterns follow:
       tv->parity = ??
       tv->hi = ??
       tv->lo = ??
       tv->xpout = ??
       tv->xperr = ??
       tv->apout = ??
       tv->aperr = ??

Parity checker failure (bad parity undetected)
Failed patterns follow: ??
       tv->parity = ??
       tv->hi = ??
       tv->lo = ??
       tv->xpout = ??
       tv->xperr = ??
       tv->apout = ??
       tv->aperr = ??
```

The *Parity* bit along with the *hi* word and *lo* word are input to the parity checker chip. The *xpout* field is the expected parity checker output, the *xperr* field is the expected parity error generated, the *apout* field is the actual parity checker output, and the *aperr* field is the actual parity error generated.

Subtest 101 Errors

Subtest 101 is the same as Subtest 100 for the PBUS parity checker, except the test is performed against the EBUS parity checker. Error messages are identical as those displayed for Subtest 100.

Subtest 150 Errors

An unsuccessful attempt at resetting the PIA board displays the following error message:

```
Unable to initialize pia to default state
```

The second phase of this subtest writes a location throughout the entire register queue. The register queue is corrupting the data entered if the information read back does not match. The following error message displays for this error condition:

```
Read queue ram failure
       Wrote: ?? ?? ?? ??
       Read back: ?? ?? ?? ??
```

The last phase of this subtest determines the proper functionality of entering additional data once previous data is rotated from the register in a sequenced pattern. Failure during this test generates the following error message:

```
Rotating pattern test failed
```

Subtest 200 Errors

The following error message displays when a read transfer is sent, but not detected in the return queue after ten cycles:

```
Fail: rq_ea_reg_full remains 0 during transfer
      rq_ea_req_full: Exp= 1 Act= ??
      Head pointer: Exp= 0 Act=??
      Tail pointer: Exp= 0 Act=??
      Bqueue count: Exp= 0 Act= ??
```

This error message displays when the return queue receives data before it is expected from memory:

```
Fail: R-queue has data too early in read cycle
      Head pointer: Exp= 0 Act=??
      Tail pointer: Exp= 0 Act=??
      Bqueue count: Exp= 0 Act= ??
```

This error message displays when the return queue does not receive expected return data:

```
Fail: Unable to load R-queue
      Head pointer: Exp= 1 Act=??
      Tail pointer: Exp= 0 Act=??
      Bqueue count: Exp= 1 Act= ??
      Pia: rq_pb_rqhd_b_a: Exp= NONZERO Act= ??
```

The following message displays when memory sends additional data to the return queue and a total of three elements are not detected within ten clocks:

```
Fail: load 2 elements and adjust in 10 clocks
      Bqueue count: Exp: 2 Act= ??
      Pia ?? scannable
```

The next error message is the result of expecting the return queue to still contain data, but does not:

```
Fail: R-queue data lost
      Head pointer: Exp= 1 Act=??
      Tail pointer: Exp= 0 Act=??
      Bqueue count: Exp= 1 Act= ??
      Pia: rq_pb_rqhd_b_a: Exp= NONZERO Act= ??
```

This error message displays when the subtest expects the return queue to be empty, but the return queue still contains data:

```
Fail: R-queue not empty after last transfer
      Head pointer: Exp= 1 Act=??
      Tail pointer: Exp= 0 Act=??
      Bqueue count: Exp= 1 Act= ??
      Pia: rq_pb_rqhd_b_a: Exp= NONZERO Act= ??
```

The following error message displays when the return queue fills sooner than the expected number of clocks:

```
Fail: Incorrect b_queue status
      Head pointer: Exp= f Act=??
      Tail pointer: Exp= 1 Act=??
      Bqueue count: Exp= e Act= ??
      Bqueue full flg: Exp= 1 Act= ??
```

This error message displays when the return queue contains invalid queue pointers for the data transferred:

```
Fail: Incorrect b_queue status
      Head pointer: Exp= 0 Act=??
      Tail pointer: Exp= 1 Act=??
      Bqueue count: Exp= f Act= ??
      Bqueue full flg: Exp= 0 Act= ??
```

This error message displays when attempting to fill the return queue within a given number of clocks and the queue did not fill:

```
Fail: Bqueue fill
      Head pointer: Exp= 0 Act=??
      Tail pointer: Exp= 1 Act=??
      Bqueue count: Exp= f Act= ??
      Bqueue full flg: Exp= 1 Act= ??
```

When executing another clock after the return queue is already full, any change in data detected (queue overflow) in the queue displays the following error message:

```
Fail: Bad b_queue status after fill
      Head pointer: Exp= 0 Act=??
      Tail pointer: Exp= 1 Act=??
      Bqueue count: Exp= f Act= ??
      Bqueue full flag: Exp= 1 Act= ??
```

With the return queue full, this test provides 16 clocks to transfer data out of the queue to get to the low-water mark. If a "not full" indication is detected before the given number of clocks, an error condition exists and the following error message displays:

```
Fail: Incorrect bq count at low water
      Bqueue full flag: Exp= 1 Act= ??
```

When the return queue is full, this test provides 16 clocks to transfer data out of the queue to the low-water mark. If the queue does not reduce to the low-water mark within the 16 clocks, an error condition exists and the following message displays:

```
Fail: Attempt to empty a full bqueue
      Head pointer: Exp= 0 Act=??
      Tail pointer: Exp= 1 Act=??
      Bqueue count: Exp= 8 Act= ??
      Bqueue full flag: Exp= 1 Act= ??
```

This error message displays when the return queue is at low-water mark and the indication is that the queue is still full:

```
Fail: Queue remains full at low-water mark
      Bqueue count: Exp= 7 Act= ??
      Bqueue full flag: Exp= 0 Act= ??
```

This error message displays when refilling the return queue and the queue fills to soon:

```
Fail: Queue refills at high water mark
      Bqueue count: Exp= 8 Act= ??
      Bqueue full flag: Exp= 1 Act= ??
```

The following error message displays when refilling the return queue and a full indication is not detected within the given number of clocks:

```
Fail: Low water mark handling
      Bqueue count: Exp= f Act= ??
      Bqueue full flag: Exp= 1 Act= ??
```

Subtest 250 Errors

An unsuccessful attempt at resetting the PIA board displays the following error message:

```
Unable to initialize pia to default state
```

The second phase of this subtest writes a location throughout the entire register queue. The register queue is corrupting the data entered if the information read back does not match. The following error message displays for this error condition:

```
Read queue ram failure
      Wrote: ?? ?? ?? ??
      Read back: ?? ?? ?? ??
```

The last phase of this subtest determines the proper functionality of entering additional data once previous data is rotated from the register in a sequenced pattern. Failure during this test generates the following error message:

```
Rotating pattern test failed
```

Subtest 300 Errors

The following error message displays when there is a change in the write/control queue after the adjustments to the log rings:

```
Aq_count changes after ring adjust
      Exp: f  Act: ??
```

This error message displays when the write/control queue cannot be filled within a given number of clocks:

```
Unable to fill Aq
      Exp: f  Act: ??
```

The following error message displays when the write/control queue is full and an attempt to signal the PBUS fails:

```
Unexpected value of mbav when aq is full
      Exp: 0  Act: ??
```

This error message indicates that upon a check of the pointers in the write/control queue, an error condition exists:

```
Unexpected Aq pointers when aq is full
      Exp: 0  Act: ??
      Exp: 1  Act: ??
```

This error message displays when the subtest is unable to take elements out of the write/control queue within a given number of cycles:

```
Unable to drain Aq to low-water mark
      Aq_count: Exp: c  Act: ??
```

The next two error messages display when an error condition exists in the signal to the PBUS that indicates the status of the write/control queue when entering or taking elements out of the queue. The first message indicates the queue is "not full" too soon is as follows:

```
Unexpected value of mbav when A-q at low water
      Exp: 0  Act: ??
      Aq_cnt: ?? Aq_head: ?? Aq_tail: ??
```

The second message displaying from this subtest, indicates the queue is "not full" too late is as follows:

```
Unexpected value of mbav when A-q at low water
      Exp: 1  Act: ??
      Aq_cnt: ?? Aq_head: ?? Aq_tail: ??
```

This error message displays when a check is made of the write/control queue when it is suppose to be empty but data is still detected in the queue:

```
Premature loading of A-q on writes
      Aq_count: Exp: 0  Act: ??
      Write_full: Exp: 0  Act: ??
```

This error message displays when sending a piece of data to the write/control queue and an error condition indicates the queue did not receive the data:

```
Premature loading of A-q on writes
      Aq_count: Exp: 1  Act: ??
      Write_full: Exp: 0  Act: ??
```

A total of 8 transfers are sent to fill the write/control queue. If the queue did not fill to the high-water mark, an error condition exists and the following error message displays:

```
Queue not filling to Hi-water mark (write)
      Aq_count: Exp: 8  Act: ??
      Write_full: Exp: 0  Act: ??
```

When sending transfers to the write/control queue, and less than eight transfers are detected in the queue, an error condition exists and the following error message displays:

```
Premature write_full signal (write)
       Aq_count: Exp: 8  Act: ??
       Write_full: Exp: 0  Act: ??
```

The first of three identical error messages displays when an error condition is detected when attempting to remove only one element from the write/control queue:

```
AQ not emptying properly after hi-water (write)
       Aq_count: Exp: 7  Act: ??
       Write_full: Exp: 1  Act: ??
```

By attempting to remove all elements in the write/control queue, but stopping just before the write full indication signals that the queue is empty, a premature indication that the queue is empty displays the following error message:

```
AQ not emptying properly after hi-water (write)
       Aq_count: Exp: 0  Act: ??
       Write_full: Exp: 1  Act: ??
```

If the subtest passes the first two conditions above, an extra clock is provided to determine if the write full changes properly and indicates an empty queue. If the write full does not show an empty indication, the following error message displays:

```
AQ not emptying properly after hi-water (write)
       Aq_count: Exp: 0  Act: ??
       Write_full: Exp: 0  Act: ??
```

With the write/control queue empty, an attempt is made to enter another element into the queue. If the element does not enter the queue properly, the following error message displays:

```
AQ not refilling properly after empty
       Aq_count: Exp: 1  Act: ??
       Write_full: Exp: 0  Act: ??
```

The following error message displays when a read request from the write/control queue is accepted while the memory is busy:

```
Queue empties to busy mem (on a Read)
       Cnt: Exp: 2  Act: ??
       rq_ea_reg_full Exp: 0  Act: ??
```

This error message displays when the write/control queue indicates unexpected data:

```
Premature rq_ea_reg_full when queue empties (on a Read)
       reg_full Exp: 1  Act: ??
```

This error message displays when the write/control queue fills sooner than the given number of clocks:

```
Premature Aqueue Full signal on read
       Aq_count Exp: f  Act: ??
       Aq_full Exp: 0  Act: ??
```

This error message displays when attempting to fill the write/control queue within a given number of clocks and the queue does not fill:

```
Inconsistent Aqueue Full or aq_count (on read)
      Aq_count Exp: f  Act: ??
      Aq_full Exp: 1  Act: ??
```

This error message displays when attempting to read out data from the write/control queue within a given number of clocks and the queue prematurely indicates "not full."

```
Inconsistent Aqueue count/aqueue_full (counting down on read)
      Exp: f  Act: ??
      Exp: 0  Act: ??
```

This error message displays when attempting to read out data from the write/control queue and the queue fails to indicate a "not full" condition in the proper number of clocks:

```
Inconsistent Aqueue count/aqueue_full (counting down on read)
      Exp: 8  Act: ??
      Exp: 0  Act: ??
```

This error message displays when data is read into the write/control queue and the queue refills before the given number of clocks:

```
Inconsistent Aqueue count/aqueue_full (counting down on read)
      Exp: 9  Act: ??
      Exp: 0  Act: ??
```

This error message displays if there is not at least one memory pair installed in the system:

```
Unable to setup memories for test
```

Subtest 350 Errors

An unsuccessful attempt at resetting the PIA board displays the following error message:

```
Unable to initialize pia to default state
```

The following error message displays when an attempt is made to write zeros to PCM RAM and an error condition is detected:

```
Failure to write zeros in pcm ram
      Failed location: ??
```

When attempting to write a pattern test to the PCM RAM and an error condition exists, the following error message displays:

```
Failed: simple pattern test ?? of pcm ram
      First failed Location: ??
      Wrote: ?? Read: ??
```

When the PCM is in an unscannable condition, the following error message displays:

```
PCM_RD: unable to scan
```

Subtest 400 Errors

The following error message indicates a failure to properly write data to the PCM RAM:

```
Failed: NPM setup
```

If this error message displays, it is recommended that subtest 350 be run in an attempt to isolate the problem further.

The next three error messages display when there is a failure to detect access to Non-Present Memory (NPM). The first error message is the result of an attempt to start a transfer in good memory and have it trespass into NPM by 1 byte:

```
Failed: NPM detection
        Transfer type: ??
        Start: present memory (??)
        End: 1 byte into NPM
```

The next error message is the result of an attempt to start a transfer in good memory and have it trespass into NPM by many bytes:

```
Failed:  NPM detection
        Transfer type:  ??
        Start:  present memory ??
        End:  many bytes into NPM (??)
```

This error message indicates a failure to detect access to NPM when attempting to start a transfer in NPM memory with sufficient length to carry it into present memory:

```
Failed: NPM detection
        Transfer type: ??
        Start: NPM (??)
        End: present memory (??)
```

The next three error messages display when there is spurious detection of NPM. The first error message displays spurious detection when starting to transfer data in the middle of good memory with short enough extent so that it should not trespass:

```
Spurious NPM detection
        Transfer type: ??
        Start: present memory (??)
        End: present memory (??)
        Number of long words transferred
        Exp: ?? Act: ??
```

The next error message indicates spurious error detection by attempting a transfer at the end of good memory that is suppose to walk up to the very end of good memory:

```
Spurious NPM detection
        Transfer type: ??
        Start: present memory (??)
        End: present memory at NPM edge (??)
        Number of long words transferred
        Exp: ?? Act: ??
```

The last error message for detecting a spurious error displays when attempting a transfer at the boundary of good memory that continues further into the good memory:

```
Spurious NPM detection
       Transfer type: ??
       Start: present memory edge of NPM (??)
       End: present memory (??)
       Number of long words transferred
       Exp: ?? Act: ??
```

This error message displays when the NPM RAM hardware fails in the middle of testing:

```
Failed: NPM reset
```

The next two error messages display if there is a problem attempting to setup NPM:

```
NPM_SETUP: unable to read pcm

NPM_SETUP: unable to write pcm
```

The following applicable message displays directly before the **Failed: NPM setup** message indicating the reason for the error in setup. If this error message displays, it is recommended that subtest 350 be run in an attempt to isolate the problem further.

The following error message is the last message in subtest 400 and displays when an error is encountered attempting to put the PCM back to the state it was in prior to the test:

```
NPM_RESET: unable to write pcm
```

Subtest 451 Errors

The following error message displays when detecting a bad I/O read header due to a bad address not detected:

```
Failed: Detection of illegal pbus header
Type: Iord transfer @ addr ??
```

This generic error message displays any time a bad header is detected, except for a parity error:

```
Failed  Detection of illegal pbus header
       Type: ??  Byte count: ??
```

The following two error messages display when an error is detected while attempting to determine PBUS acceptance of a non-operational instruction:

```
Failure to accept nop, byte count ??

Failure to accept nop, address: ??
```

The following error message indicates error detection when trying to transfer an otherwise good PBUS header containing bad parity:

```
Failed: Detection of pbus header parity error
```

Any IOWRT should fail. The following error message displays when there is an I/O error:

```
Failed: Detection of IOWRT pbus header
```

Subtest 550 Errors

The first part of this subtest determines that the device counter indicates the proper device causing an interrupt. If the device counter malfunctions, the following error message displays:

```
Failed: Interrupt arbitration
        ?? Clocks into arbitration sequence
        Dev_ctr= ?? (expected= ??)
        Next state= ?? (expected ??)
```

This error message displays when the device counter does not equal the requester when transitioning from state 0 to state 1:

```
Failed:
Iarb state transition:
        dev_ctr= ?? requester= ??
        Expected state: ??
        Actual: ??
```

This error message displays when the arbitration state gets out of sequence when transitioning through the other states:

```
Failed: Iarb state transition
        Next state: Exp: ??  Act: ??
```

When internal interrupt signals are sent out on the board and an interrupt is found to be active, the following error message displays:

```
BAD Iarb output:
        sibint: Exp: ??  Act: ??
```

It is an indication that the interrupt is not in the expected state.

This error message displays when an interrupt is received by a device and the device either acknowledged the interrupt when it shouldn't have or the device did not acknowledge the interrupt when it should have:

```
BAD Iarb output:
        sibcmplt: Exp: ??  Act: ??
```

Subtest 600 Errors

The following error messages display at intervals during the execution of Subtest 600 upon detecting an error condition. The error messages indicate the function of the test that has failed and displays the expected and actual results at that point in the test.

```
Bad ebus zone Addr= ??
       Even: Exp= ?? Act= ??
       Odd: Exp= ?? Act= ??

Bad TAS ebus data in data output phase (TAS)
       Hi: Exp= ?? Act= ??
       Lo: Exp= ?? Act= ??
       Parity: Exp= ?? Act= ??

Bad Ebus cycle in data output phase (TAS)
       Exp= ?? Act= ??

Ebus output enable in Data output phase (TAS)
       Exp= ?? Act= ??

Bad TAC ebus data in data output phase (TAC)
       Hi: Exp= ?? Act= ??
       Lo: Exp= ?? Act= ??
       Parity: Exp= ?? Act= ??

Bad Ebus cycle in data output phase (TAC)
       Exp= ?? Act= ??

Ebus output enable in Data output phase (TAC)
       Exp= ?? Act= ??

Bad Board Select/maddr generated during ?? xfer
       Addr= ??  Adr_mod= ?? mem_int= ??
       Board select: Exp= ?? Act= ??
       Maddr: Exp= ?? Act= ??

Bad ebus address(es) generated on ?? transfer
       Exp= ?? Act= ??

Data corrupted internally during ?? transfer
       Hi word: Exp= ?? Act= ??
       Lo word: Exp= ?? Act= ??
```

Subtest 650 Errors

The following error messages display at intervals during the execution of Subtest 650 upon detecting an error condition. The error messages indicate the function of the test that has failed and displays pertinent data for the appropriate error message.

```
Failed ebus arb: Spu unable to obtain ebus

Failed ebus arb: pbus unit unable to obtain ebus
pbus_flag: ??

Failed ebus arb:  unable to obtain ebus
pbus_flag: ?? ea_state: ??
ea_pb_send_header: ??

Ebus arb failure: Spu not given priority
```

SPU UNIX Error Messages

For a common list of possible error messages resulting from SPU UNIX, refer to the appendix on SPU UNIX Error Messages.

pi2_4000

PI2 Functional Test

Overview

The *pi2_4000* test is a functional test for the PBUS Interface Adapter 2 (PI2). All major functional units of the PI2 are tested including:

- PBUS interface logic
- RAMs
- Arbitration

In addition, other functionality that does not reside in any one area is also tested (ie., error handling). It is important to note the following areas of functionality that cannot be tested adequately:

- Arbitration logic can only be exhaustively tested when a full complement of PI2s is installed; therefore, a small amount of functionality is not tested in most configurations
- Paths between the backplain and the Scan registers are not testable with Scan-based tests alone (use *io4000* or other CCU tests to cover this functionality)
- The CCU-clock logic is not testable completely with Scan based tests (use *io4000* to cover this functionality)
- The Scan logic is also not tested and *spu4000* should be used to verify this functionality

This test offers two modes of operation. First, the traditional "fail / no-fail" testing that is commonly used is available. The second mode of operation includes the ability to single step (trace) through the entire Scan based portion of the test or up to the point of failure. Included in this mode is the capability to enter *iscan* or *sputil* before each step. When in the single step mode, the test will stop before each Scan operation (ie., clock, compare, write to field, etc...) and display the operation to be executed next. At that point, a shell can be forked so that *iscan* or *sputil* can be used to manipulate/examine the board during the diagnostic subtest or enter RETURN to continue.

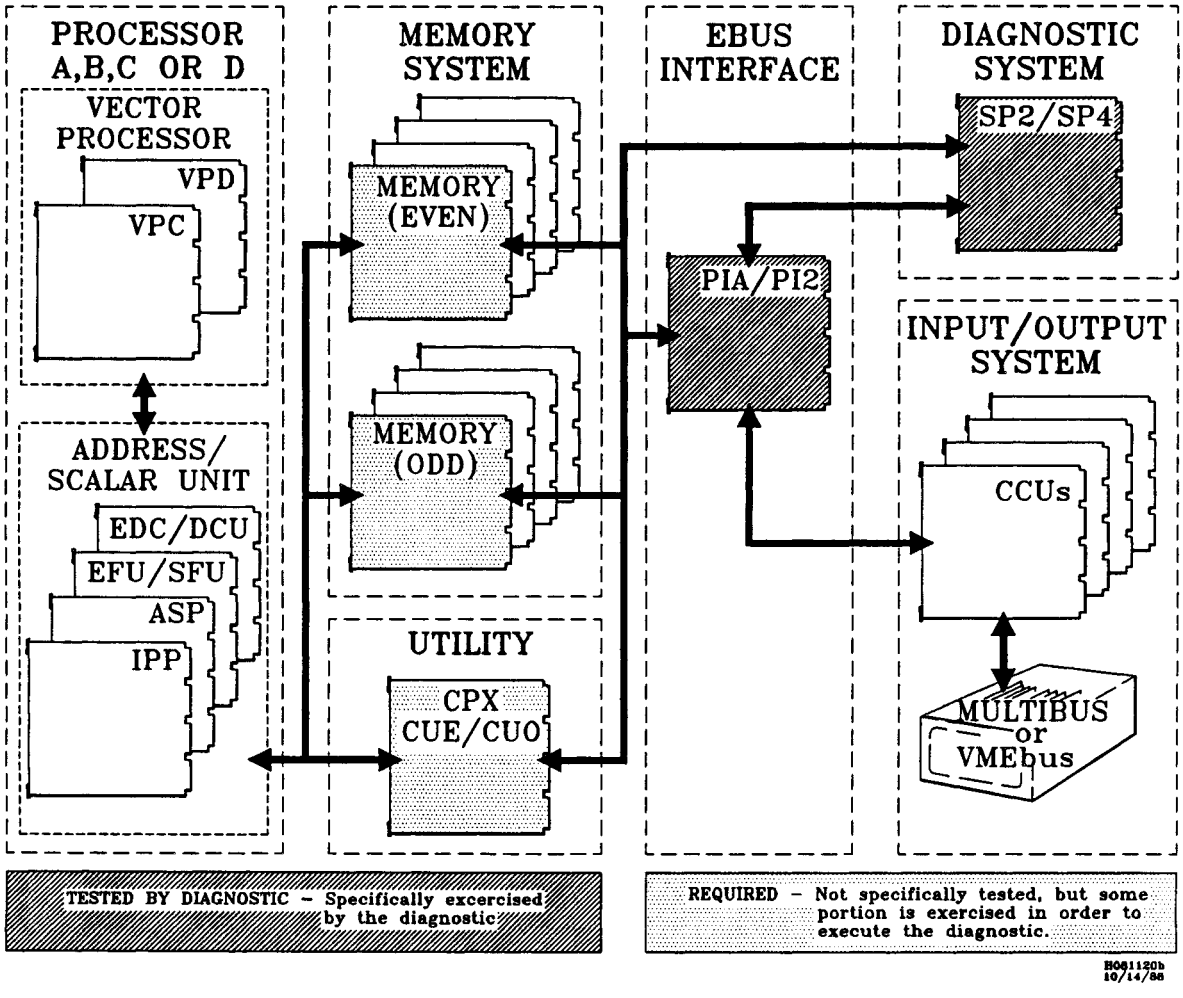
The Scan based portion of the test can be modified using a Scan-Language Interface that is described in the *CONVEX Processor Diagnostics Manual (C200 Series) Scan-Language Interface Supplement*. This ability is designed primarily for CONVEX Manufacturing and Engineering personnel for the purposes of debugging. The tools necessary to utilize the Scan-Language Interface are not shipped to field personnel and customers.

NOTE

The term "Service Processor" and "SP" are used generically to represent either SP2 or SP4 depending on the system configuration under test. The term "CPU Utility Board(s)" is used to represent either a CPX or a CUE / CUO combination depending on the system under test. Also, "PBUS Interface Board(s)" is used to represent either a PIA, a PI2 installed in the PIY slot, or two PI2s installed in the PIX and PIY slots, depending on the system configuration under test.

The following figure shows an overall view of what part of the system is being tested and what Field Replaceable Units (FRUs) are required to execute the test.

Figure pi2_4000–1, Functional Areas Tested by *pi2_4000*



Prerequisites and Required Equipment

The boards listed in the following table must be operational for test execution. Also shown in the following table are the tests used to verify the required boards. Since this test is designed to test a PI2, at least one PI2 must be present.

> **NOTE**
>
> In order to verify the full functionality, the system under test must contain as many PI2s as allowed in the system. However, only a very minimal portion of functionality is not tested when fewer than possible PI2s are present. A successful test will, in most cases, yield an acceptable level of confidence in the board(s).

Table pi2_4000–1, Required Functional Boards

BOARD	TEST TO VERIFY
Service Processor	*spu1000, spu4000*
CUO	*cpx4000*
Memory system	*mem4000*

> **NOTE**
>
> Memory system consists of a minimum of one pair of memory boards (one even and one odd).

Test Invocation

To invoke the *pi2_4000* test, use the procedure shown in the following figure. All responses in **boldface** are entered by the user. The prompts and responses in the following figure would appear sequentially on the screen, one line at a time. All the prompts and responses are shown in one figure for convenience.

CAUTION

The invocation sequence shown in the following figure is the typical invocation sequence. However, the *initall* utility must be executed in some cases. If the system has just been powered up, if *mem4000* was executed with failures, or if *spu4000* was executed, then *initall* must be executed prior to any test execution. Failure to execute *initall* in these circumstances could result in invalid test results.

NOTE

Running the *initall* utility requires two to three minutes to execute depending on if the control stores have been previously loaded. However, it is suggested that *initall* be executed in the event that the state of the system is unknown.

Figure pi2_4000–2, Test Invocation Sequence

```
(spu)>  cd /mnt/test
(spu)>  sysreset
(spu)>  dshell

CONVEX DIAGNOSTIC SHELL

: test pi2_4000 [-c [class number(s)]] [-s [subtest number(s)]] [+> filename]
```

NOTE

After entering **dshell**, specific *dshell* parameters may be changed. Please refer to the "Dshell and Iscan Overview" chapter of this manual for more information on *dshell*.

Entering only (*pi2_4000*) executes all *pi2_4000* subtests sequentially. Execute a specific class(es) of subtest(s) or one or more individual subtests by using the **-c** or **-s** options, respectively. Detailed information for using these options can be found in the "Dshell and Iscan Overview" chapter of this manual. The [+>*filename*] option allows the test results to be appended to *filename*.

Test Parameter Menu

Once the test is invoked, a test parameter menu prompt is presented allowing selection of default switches. The following figure shows all prompts, their possible answers (in brackets [ ]), and their default answers (in parentheses ( )). The prompts and responses in the following figure appear sequentially on the screen, one line at a time.

Figure pi2_4000–3, Test Parameter Menu

```
Test 'pi2_4000'                    Thu Nov 19 00:00:00 1965

                          ENTER TEST PARAMETERS

                  [ ]     Encloses allowed input ranges or values
                  ( )     Encloses the default value
                  ^       Returns to the previous prompt
                  :nn     Returns to the prompt # nn
                  :       Returns to the first unsatisfied prompt
                  :?      Reviews previous entries

1: Enter Pi2 to test (0=piy,1=pix): [0-1]                  (0) ->
2: Enter value of trace flag: [0-1]                        (0) ->
3: Enter value of extended test flag: [0-1]                (0) ->
4: Enter OK, or :NN to return to question NN [OK]
                                                           (OK) ->
```

Prompt Explanations

The following section describes each of the previous prompts.

`Enter Pi2 to test (0-piy,1-pix): [0-1] (0) ->`

This prompt allows selection of the particular PI2 board to be tested. Enter 0 for piy, and 1 for pix.

`Enter value of trace flag: [0-1] (0) ->`

This prompt allows the trace mode to be turned on or off. Enter 0 to turn trace mode off, and enter 1 to turn trace mode on. Refer to the previous explanation of trace mode in the Overview section of this chapter.

`Enter value of extended test flag: [0-1] (0) ->`

This prompt allows extended test mode to be turned on or off. Enter 0 to turn extended mode off, and enter 1 to turn extended mode on. Executing the diagnostic in extended mode requires more time than in regular mode because it causes the diagnostic to perform more extensive tests. In most cases, adequate fault coverage can be achieved without executing in the extended mode.

```
Enter OK, or :NN to return to question NN [OK]
                                          (OK) ->
```

If the **OK** default is selected, test execution begins. A particular prompt can be changed by entering its number here and the program would return to that prompt.

Class Description

The PI2 functional subtests encompass only one class. This class contains six different types of subtests that test a logical section(s) of the board. The six types of subtests are associated with the following areas:

- Clock
- PBUS
- RAM
- Transfer
- Error
- Arbitration

In most cases the subtests associated with transfers have a tendency to test all areas working together. The following table identifies each of the subtests within this class. An abbreviated description of the hardware tested by each subtest is given in the "TEST PERFORMED" column.

Table pi2_4000–2, PI2 Functional Subtests

SUBTEST	TEST PERFORMED	DEFAULT TIME REG. MODE (min/sec)	DEFAULT TIME EXT. MODE (min/sec)
100	Clock Align Test	0:00	0:00
150	Clock State Machine Test	0:01:0:01	
200	PBUS Integrity Test	0:02	0:02
250	Log Ring Lock-on-Error Test	0:01	0:01
300	PBUS Parity Checker Test	0:12	0:12
350	PBUS Arbitration Test	0:02	0:02
400	PBUS Illegal Header Test	0:35	0:35
450	Memory Base Pointer Test	0:07	0:07
500	PCM RAM Test	0:20	2:48
550	Write/Control Queue RAM Test	0:59	5:02
600	Write Transfer Test	0:38	3:13
650	Arbitration Queue RAM Test	0:20	0:42
700	Return Queue RAM Test	0:27	2:17
750	EBUS Parity Checker Test	0:05	0:05
800	Read Transfer Test	0:27	0:27
850	TAM Transfer Test	0:05	0:05
900	Memory Test	0:38	0:38
950	Hard Error Test	0:02	0:02
1000	Non-present Memory Test	0:30	0:30
1050	Write Data Parity Error Test	0:06	0:06
1150	Interrupt Function Test	0:05	0:05

NOTE

Some of the times above may vary plus or minus one second. Some times are zero because the subtests run faster than they can be clocked. This test has no timeout provisions and if any subtest takes substantially more time than is listed, the test may have failed.

Subtest Descriptions

The following subtests are described for the type of function performed by the subtest. If there are no errors detected during the test, the response for each subtest is **passed**. If any of the following subtests fail, the PI2 board(s) must be replaced.

> **NOTE**
>
> Accomplish **sysreset** before attempting to run *pi2_4000*. If the **sysreset** is not performed, some of the subtests could indicate a malfunction, when in fact none exists.

Subtest 100, Clock Alignment Test

This subtest verifies that the clock logic on the Service Processor and the PI2 can be properly aligned and that this alignment is preserved when the rings are clocked. It will test the scannability of the 10Mhz ring in the following circumstances:

- After sysreset has been issued
- After each of 5 single 10Mhz clocks (single step clocks)
- After groups of 1, 2, 3, 4, and 5 10Mhz clocks (multi-step clocks)

Subtest 150, Clock State Machine Test

This subtest is almost identical to the clock alignment subtest except that instead of checking the Service Processor's Scan-OK bits, in the Scan Feedback Register (SFR), it will look at the internal state of the clock logic via look-a-side Scan. The cases tested are as follows:

- After sysreset has been issued
- After each of 5 single 10Mhz clocks (single step clocks)
- After groups of 1, 2, 3, 4, and 5 10Mhz clocks (multi-step clocks)

Subtest 200, PBUS Integrity Test

This subtest is broken into two parts:

- Verification of a properly floating bus on reset
- Pattern testing of the PBUS in loopback mode

The patterns tested are the following:

- 64 bit word of all 0's
- 64 bit word of all f's
- 64 bit word of all a's
- 64 bit word of all 5's

- 0x123456789abcdef0

Subtest 250, Log Ring Lock-on-Error Test

This subtest verifies that the Log Ring actually "freezes up" in an error condition. Both EBUS and PBUS sections are checked to verify that the lock bits (and only the lock bits) do indeed lock their respective sections.

Subtest 300, PBUS Parity Checker Test

This subtest verifies that the parity checker on the PBUS does correctly detect bad parity and does not spuriously indicate bad parity. The following cases are examined:

- Verify good parity for data all 0's, parity all f's
- Walk a zero through parity for data all 0's
- Walk a one through all bytes of the data words with f's in the parity

Subtest 350, PBUS Arbitration Test

This subtest verifies that single and multiple PBUS requests are properly arbitrated. A request is set up in Scan and arbitration logic is clocked to verify that it does indeed arbitrate correctly. Both single request arbitration and multiple request arbitration are verified.

Subtest 400, PBUS Illegal Header Test

This subtest checks that the PI2 properly detects incorrectly constructed PBUS headers. The following cases are tested:

- Header bad parity detection
- TAS (Test and Set) transfer with byte count not equal to one
- TAC (Test and Clear) transfer with byte count not equal to one
- I/O write transfer
- Unimplemented PBUS type
- I/O read with illegal address

Subtest 450, Memory Base Pointer (MBP) Read Test

This subtest verifies the integrity of the MBP data paths using 0's, f's, a's and 5's patterns on the PBUS.

Subtest 500, PCM RAM Test

This subtest verifies that all possible patterns (0,1,2,3) can be written and read from the PCM RAM.

Subtest 550, Write/Control Queue RAM Test

This subtest verifies that the patterns (0, f, 3, 5, 6, 9, a, c) can be written to and read from all RAM locations.

This subtest consists of two verifications:

- Verify all data values can be written to location 0.
- Verify that the patterns (0 , f, 3, 5, 6, 9, a, c) can be written to all RAM locations.

Subtest 600, Write Transfer Test

Subtest 600 verifies the following:

- Longword aligned single transfer
- Longword aligned dual transfer
- Write abort
- Service Processor write
- The zone is calculated correctly for all possible zone combinations
- Burst mode
- Queue full/empty conditions

Subtest 650, Arbitration Queue RAM Test

This subtest consists of two verifications:

- Verify all data values can be written to location 0
- Verify that the patterns (0, f, 3, 5, 6, 9, a, c) can be written to all RAM locations

Subtest 700, Return Queue RAM Test:

Subtest 700 verifies that patterns of (0, f, 3, 5, 6, 9, a, c) repeated in all nibbles can be read and written to the RAM.

Subtest 750, EBUS Parity Checker Test

This subtest checks that the parity checker on the EBUS does in fact correctly detect bad parity and does not spuriously indicate bad parity. The following cases are examined:

- Verify good parity for data all 0's, parity all f's
- Walk a zero through parity for data all zeros
- Walk a one through all bytes of the data words with f's in the parity

Subtest 800, Read Transfer Test

Subtest 800 verifies the following:

- Single word transfer
- Dual transfer
- Burst mode operation
- Queue full/empty handling
- Transfer abort handling
- Simulated full 64k-byte transfer

Subtest 850, Test and Modify (TAM) Transfer Test

Subtest 850 verifies the following:

- Test and Set (TAS) transfer
- Test and Clear (TAC) transfer
- Service Processor Test and Modify (TAM) transfer

Subtest 900, Memory Test

This subtest sets up a transfer in Scan and lets it finish without checking any machine state. The object is to see if the data is correctly read/written from/to memory. The cases tested are as follows:

- Service Processor read, Service Processor write
- PBUS write, Service Processor read
- Service Processor write, PBUS read
- PBUS write, PBUS read

Subtest 950, Hard Error Test

This subtest checks the mechanisms for generating hard errors. It attempts to verify the following:

- PCM parity error detection
- Interrupt arbiter error detection
- Memory data error detection

Subtest 1000, Non-present Memory (NPM) Test

The following cases of NPM detection are verified:

- Transfer with starting address in NPM

- Transfer ending in NPM
- Transfer with starting address just after NPM
- Transfer ending at address just before NPM

Subtest 1050, Write Data Parity Error Test

In this subtest, a transfer is set up and a parity error is forced in each of the following circumstances:

- Error on first write
- Error on last write
- Error on an intermediate write

Subtest 1150, Interrupt Function Test

In this subtest, the interrupt state machine will be verified as well as the grant generation circuitry.

Modes of Operation and Source Files

The *pi2_4000* diagnostic has two modes of execution. The test can be executed in the traditional "fail / no-fail manner" in which the test is executed and if any failures occur, failure messages are output. The second mode of execution allows the user to step through the test in a tracing mode. This single step method allows the user to step through the test up to the point of failure or to set break points in a failing diagnostic.

The diagnostic itself has two sets of source code. There are both C-language source files and Scan-Language scripts. The Scan-Language scripts are compiled together into a single object file which is read into the diagnostic's data space. This single object file (*pi2_4000.x00*) is actually a description of threaded code to be executed by the C-language routine (*pia_4000.t*) of the diagnostic.

Scan Language Test Modification

The ability to modify and re-compile the individual Scan script files is provided to CONVEX manufacturing technicians and engineers. This capability is not shipped to field personnel or customers. Refer to the *CONVEX Processor Diagnostics Manual (C200 Series) Scan-Language Interface Supplement* for more details on this procedure.

pi2_4000 Specific Error Messages

There are two basic types of error messages created by *pi2_4000*. The first type of error message is in a standard format. In most cases, this type of error message is associated with a compare function. The standard type of error message takes the following format:

```
================================================================================
                    MISMATCH      EXP       ACT    COMMENT
================================================================================
Mismatch_String                   XX        YY     Explanatory_Statement
================================================================================
```

Where:

Mismatch_String is a description of the field or register that did or did not match

XX is the expected value from the field

YY is the actual value of the field

Explanatory_Statement is an explanatory statement about the comparison

> **NOTE**
>
> The above message format is displayed every time there is a comparison made. The only time it should be regarded as an error message is when the actual and expected values do not match.

All other *pi2_4000* specific error messages are specific in nature and are displayed in the following sections.

Subtest 300 Error Messages

Subtest 300 returns an error message that contains both a standard comparison format table (shown previously) and the following message:

```
1. st_300_2 Pattern= XXXXXXXX: failed
```

Where:

XXXXXXXX is an hexadecimal pattern that caused the failure. The pattern should be in the form of 0x01010101, 0x02020202, ... 0x08080808.

Subtest 350 Error Messages

Subtest 350 returns the following possible error messages:

```
Incorrect PBUS grant single request
Requestor: PBUS device YYYY
```

Where:

YYYY is NONE, 0, 1, 2, or 3

Or:

```
Incorrect PBUS grant multiple request
Requestors: PBUS devices [<list of devices chosen from 0-3> ]
Last grantee: PBUS device X
Grantee: PBUS device YYYY
Expected grantee: PBUS device ZZZZ
```

Where:

X is 0, 1, 2, or 3

YYYY is NONE, 0, 1, 2, or 3

ZZZZ is NONE, 0, 1, 2, or 3

Subtest 400 Error Messages

Subtest 400 returns a combination of the standard compare table and information about the address used and parity. The following message depicts the format of the address and parity information:

```
Addr: XXXXXXXX parity= YYYYYYYY
Byte count: IIIIIIII Parity JJJJJJJJ
```

Where:

XXXXXXXX, YYYYYYYY, IIIIIIII, and JJJJJJJJ are hexadecimal numbers

Subtest 450 Error Message

The following error message is returned from Subtest 450:

```
st_450_0 Pattern= XXXXXXXX input parity= YYYYYYYY output parity= ZZZZZZZZ
```

Where:

XXXXXXXX, YYYYYYYY, and ZZZZZZZZ are hexadecimal numerals

Subtest 500 Error Messages

The following error messages are returned by Subtest 500:

```
PCM RAM error SEQUENCE: DDDDDDDD
RAM addr= XXXXXXXX Actual value= YYYYYYYY Expected value: ZZZZZZZZ
```

Where:

DDDDDDDD is a decimal numeral
XXXXXXXX, YYYYYYYY, and ZZZZZZZZ are hexadecimal numerals

Or:

```
PCM RAM
RAM addr= XXXXXXXX Actual value= YYYYYYYY Expected value: ZZZZZZZZ
```

Where:

XXXXXXXX, YYYYYYYY, and ZZZZZZZZ are all hexadecimal numerals

Subtest 550 Error Messages

The following error message is returned from Subtest 550"

```
st_550: Write cntrl q address= XXXXXXXX
```

Where:

XXXXXXXX is an hexadecimal numeral

Subtest 600 Error Message

The following error message is returned by Subtest 600:

```
st_600_2: failed
taddr= XXXXXXXX byte count= YYYYYYYY
```

Where:

XXXXXXXX and YYYYYYYY are hexadecimal numerals

Or:

```
st_600_3: failed
taddr= XXXXXXXX byte count= YYYYYYYY
```

Where:

XXXXXXXX and YYYYYYYY are hexadecimal numerals

Subtest 650 Error Messages

The following error messages are returned by Subtest 650:

```
pat= XXXXXXXX Addr= YYYYYYYY old_pat= ZZZZZZZZ
```

Where:

XXXXXXXX, YYYYYYYY, and ZZZZZZZZ are all hexadecimal numerals

Or:

```
Error in reading back PREVIOUS DATA
```

> **NOTE**
>
> This error message as well as the following on are followed by the standard comparison table.

Or:

```
Error in reading back CURRENT DATA
```

Subtest 700 Error Message

The following error message is returned by Subtest 700:

```
st_700: addr= XXXXXXXX
```

Where:

XXXXXXXX is an hexadecimal numeral

Subtest 750 Error Message

The following error message is returned by subtest 750:

```
st_750_1 Pattern= XXXXXXXX: failed
```

Where:

XXXXXXXX is an hexadecimal numeral

Subtest 800 Error Messages

The following error messages are returned by Subtest 800:

```
addr= XXXXXXXX hdr par= YYYYYYYY hdr hi= ZZZZZZZZ hdr lo= IIIIIIII
```

Where:

XXXXXXXX, YYYYYYYY, ZZZZZZZZ, and IIIIIIII arre hexadecimal numerals

Or:

```
byte count= DDDDDDDD (=XXXXXXXX)
```

Where:

DDDDDDDD is a decimal numeral
XXXXXXXX is an hexadecimal numeral

Or:

```
data hi= XXXXXXXX data lo= YYYYYYYY data par= ZZZZZZZZ  maddr= IIIIIIII
```

Where:

XXXXXXXX, YYYYYYYY, ZZZZZZZZ, and IIIIIIII are hexadecimal numerals

Subtest 850 Error messages

The following error messages are returned from Subtest 850:

```
Unexpected value in mem after TAC
       Exp: XXXXXXXX Actual: YYYYYYYY
```

Where:

XXXXXXXX and YYYYYYYY are hexadecimal numerals

Or:

```
Dump of entire long word:
       XXXXXXXX
```

Where:

XXXXXXXX is a hexadecimal numeral

Or:

```
Unexpected TAS return
       Exp: XXXXXXXX Actual: YYYYYYYY
```

Where:

XXXXXXXX and YYYYYYYY are hexadecimal numerals

Or:

```
Unexpected TAS set value
       Exp: XXXXXXXX Actual: YYYYYYYY
```

Where:

XXXXXXXX and YYYYYYYY are hexadecimal numerals

Or:

```
Unexpected TAC return
       Exp: XXXXXXXX Actual: YYYYYYYY
```

Where:

```
Unexpected TAC clr value
       Exp: XXXXXXXX Actual: YYYYYYYY
```

Where:

XXXXXXXX and YYYYYYYY are hexadecimal numerals

Subtest 900 Error Messages

Any of the following error messages can be returned by Subtest 900:

```
Data line test: (SPU read) ADDR: XXXXXXXX
       Expected data: hi= IIIIIIII lo= JJJJJJJJ
       Actual data:   hi= KKKKKKKK lo= LLLLLLLL
```

Or:

```
Data line test: (PBUS write/read) ADDR: XXXXXXXX
       Expected data: hi= IIIIIIII lo= JJJJJJJJ
       Actual data:   hi= KKKKKKKK lo= LLLLLLLL
```

Or:

```
Data line test: (PBUS read) ADDR: XXXXXXXX
```

```
        Expected data: hi= IIIIIIII lo= JJJJJJJJ
        Actual data:   hi= KKKKKKKK lo= LLLLLLLL
```

Or:

```
Data line test: (SPU read/write) ADDR: XXXXXXXX
        Expected data: hi= IIIIIIII lo= JJJJJJJJ
        Actual data:   hi= KKKKKKKK lo= LLLLLLLL
```

Or:

```
Address line test: (SPU read) ADDR: XXXXXXXX
        Expected data: hi= IIIIIIII lo= JJJJJJJJ
        Actual data:   hi= KKKKKKKK lo= LLLLLLLL
```

Or:

```
Address line test: (PBUS write/read) ADDR: XXXXXXXX
        Expected data: hi= IIIIIIII lo= JJJJJJJJ
        Actual data:   hi= KKKKKKKK lo= LLLLLLLL
```

Or:

```
Address line test: (PBUS read) ADDR: XXXXXXXX
        Expected data: hi= IIIIIIII lo= JJJJJJJJ
        Actual data:   hi= KKKKKKKK lo= LLLLLLLL
```

Or:

```
Address line test: (SPU read/write) ADDR: XXXXXXXX
        Expected data: hi= IIIIIIII lo= JJJJJJJJ
        Actual data:   hi= KKKKKKKK lo= LLLLLLLL
```

Subtest 1150 Error Message

The following error messages is returned from Subtest 1150

```
Exhaustive req/dev_ctr test
Requestor: DDDDDDDD Step: EEEEEEEE
State: expected: FFFFFFFF actual: GGGGGGGG
dev_ctr exp: HHHHHHHH (XXXXXXXX) act: IIIIIIII (YYYYYYYY)
```

Where:

DDDDDDDD, EEEEEEEE, FFFFFFFF, GGGGGGGG, HHHHHHHHH, and IIIIIII are decimal numerals

XXXXXXXX and YYYYYYYY are hexadecimal numerals

SPU UNIX Error Messages

For a common list of possible error messages resulting from SPU UNIX, refer to the appendix on SPU UNIX Error Messages.

mem4000

Memory Subsystem Test

Overview

The *mem4000* test verifies the control features of the memory system as well as the memory arrays. This test will be executed from the Service Processor and is initiated under the Diagnostic Shell (Dshell).

The memory system test may be accessed via 5 different ports: the I/O port (port E) and each of 4 CPU ports (ports A-D).

The memory test is designed to be a tool for use by hardware personnel during development of the memory system, by manufacturing personnel during board test, and by field personnel during diagnostics testing. The following figure shows an overall view of what part of the system is being tested and which field replaceable units are required for the test to run.

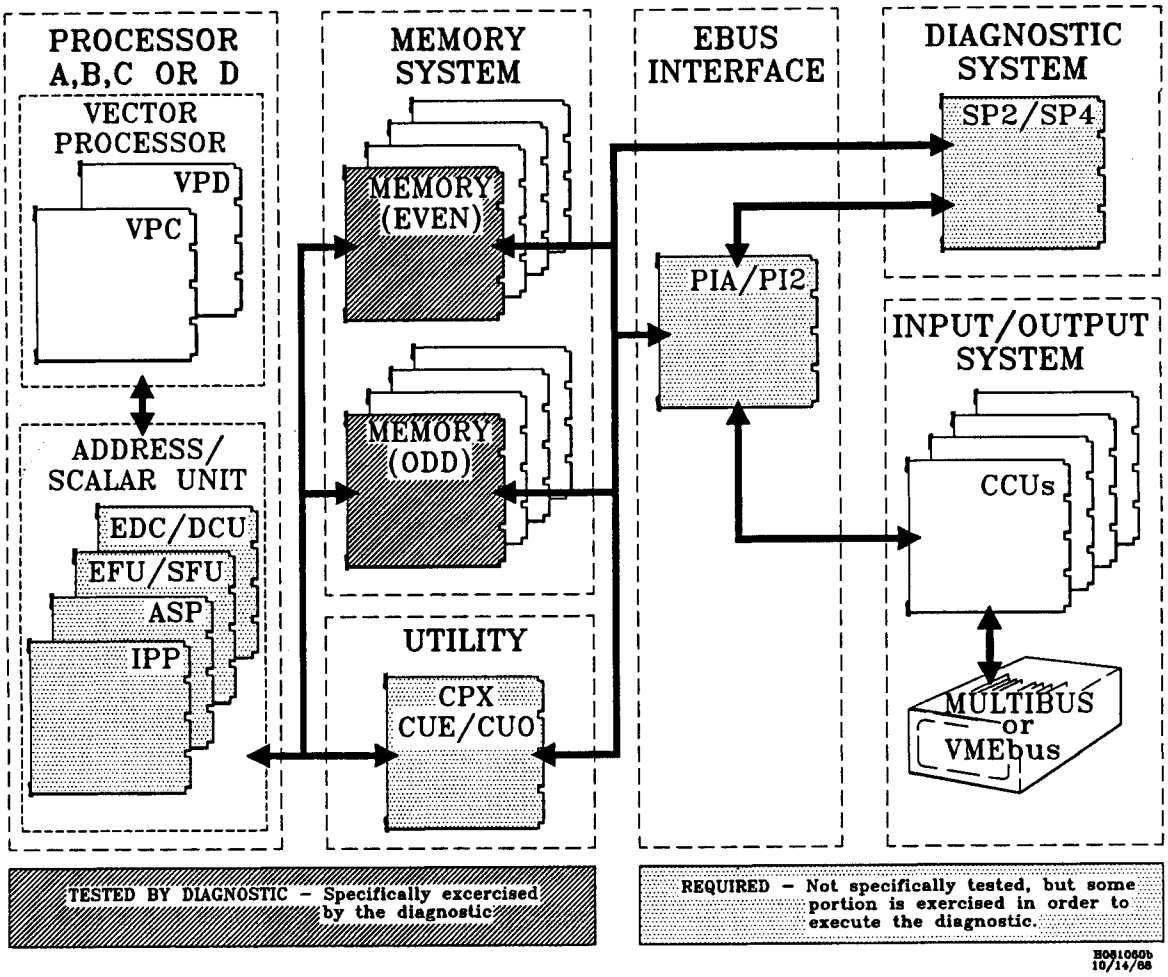
NOTE

If this test is executed and failures occur, the *initall* utility must be executed prior to any other test invocation. Failure to execute the *initall* utility in this circumstance could result in invalid test results.

NOTE

The term "Service Processor" and "SP" are used generically to represent either SP2 or SP4 depending on the system configuration under test. The term "CPU Utility Board(s)" is used to represent either a CPX or a CUE / CUO combination depending on the system under test. Also, "PBUS Interface Board(s)" is used to represent either a PIA, a PI2 installed in the PIY slot, or two PI2s installed in the PIX and PIY slots, depending on the system configuration under test.

Figure mem4000–1, Functional Areas Tested by mem4000



NOTE

This test requires a memory system which consists of one pair of memory boards (one even and one odd).

Prerequisites and Required Equipment

The Service Processor, CPU (in CPU-based subtests), CPX or CUE/CUO, PIA or PI2, memory scan rings, and EBUS must be installed and functional in order to run this test.

Table mem4000–1, Required Functional Boards

BOARDS FOR CLASS 1 & 2	TESTS TO VERIFY
Service Processor	*spu1000, spu4000*
PBUS Interface Adapter (PIA or PI2)	*pia4000* or *pi2_4000*

BOARDS FOR CLASS 3 & 4	TESTS TO VERIFY
Initial Block of Main Memory and common MCM functions	*mem4000* Class 1 & 2
Service Processor	*spu1000, spu4000*
PBUS Interface Adapter (PIA or PI2)	*pia4000* or *pi2_4000*
Instruction Processor Unit (IPP)	*cpu4030*
Address Scalar Processor (ASP)	*cpu4030*
Scalar Function Unit (SFU or EFU)	*cpu4030*
Data Cache Unit (DCU or EDC)	*cpu4030*
CPU Utility Board(s) (CPX or CUE/CUO)	*cpx4000*

Test Invocation

To invoke the *mem4000* test, use the procedure shown in the following figure. All responses in **boldface** are entered by the user. The prompts and responses appear sequentially on the screen, one line at a time. All prompts and responses are shown in one figure for convenience.

CAUTION

The invocation sequence shown in the following figure is the typical invocation sequence. However, the *initall* utility must be executed in some cases. If the system has just been powered up, if *mem4000* was executed with failures, or if *spu4000* was executed, then *initall* must be executed prior to any test execution. Failure to execute *initall* in these circumstances could result in invalid test results.

NOTE

Running the *initall* utility requires two to three minutes to execute depending on if the control stores have been previously loaded. However, it is suggested that *initall* be executed in the event that the state of the system is unknown.

Figure mem4000–2, Test Invocation Sequence

```
(spu)>  cd /mnt/test
(spu)>  sysreset
(spu)>  dshell

CONVEX DIAGNOSTIC SHELL

:  test mem4000 [-c [class number(s)]] [-s [subtest number(s)]] [+> filename]
```

NOTE

After entering response, "**dshell**", specific *dshell* parameters may be changed. Refer to the "Dshell and Iscan Overview" chapter of this manual for more information.

Entering only **test mem4000** executes all *mem4000* subtests sequentially. Execute a specific class(es) of subtest(s) or one or more individual subtests by using the *-c* or *-s* options, respectively. Detailed information for using these options can be found in the "Dshell and Iscan Overview" chapter of this manual. The **[+>***filename***]** option allows the test results to be appended to *filename*.

Test Parameter Menu

Once the test is invoked, a test parameter menu is presented allowing selection of default switches. If the test is run with all defaults invoked (answer **y** to the first prompt), no other prompts are displayed. If the user answers **n** to the first prompt (run test without default switches), then a series of prompts are presented. The following figure shows all possible prompts, possible responses (in brackets [ ]), and default responses (in parentheses ( )). The prompts and responses in the following figure appear sequentially on the screen, one line at a time. All the prompts and responses are shown in one figure for clarity.

Figure mem4000–3, Test Parameter Menu

```
Test 'mem4000.t'                                    Tue sep 13 04:30:15 1955


                          ENTER TEST PARAMETERS

                  [ ]    Encloses allowed input ranges or values
                  ( )    Encloses the default value
                  ^       Returns to the previous prompt
                  :nn     Returns to the prompt # nn
                  :       Returns to the first unsatisfied prompt
                  :?      Reviews previous entries

1:  Use default test parameters [y,n]                          (y) ->
2:  Wait time for refresh tests []                            (10) ->
3:  SPU tests start address (Physical) []                    (0x0) ->
4:  SPU tests end address + 1 (Physical)
    [0x0-0x7ffffff]                                    (0xXXXXX) ->
5:  CPU tests start address (Logical)
    [0xXXXXX-0xYYYYYYYY]                              (0xYYYYYYYY) ->
6:  CPU tests end address + 1 (Logical)
    [0xXXXXX-0xYYYYYYYY]                              (0xYYYYYYYY) ->
7:  CPU for CPU based tests [0,1,2,3]                          (0) ->
8:  Enter slot mask to use []                               (0xff) ->
9:  Enter OK, or :NN to return to question NN [OK]
                                                            (OK) ->
```

Prompt Explanations

A description of the meaning of each prompt follows:

```
Use default test parameters [y,n]                          (y) ->
```

If the user responds with **y** or <**CR**>, no additional test parameter prompts are displayed and testing begins. If, however, a negative response is supplied, additional test parameter prompts are displayed allowing modification of the default selections.

The following prompts are displayed and answered only if the first prompt is answered with **n**:

```
Wait time for refresh tests []                            (10) ->
```

Enter a decimal number having a maximum of five digits that specifies the amount of time to use between pattern tests for the refresh test (refer to Subtest 102).

```
SPU tests start address (Physical) []                    (0x0) ->
```

Enter a hex number having a maximum of eight digits that specifies the beginning address of all Service Processor-based pattern testing. The addresses referenced from the Service Processor are given in physical addresses. Any non-existent blocks within

the specified address range are skipped. All of physical memory can be tested from the Service Processor if desired.

```
SPU tests end address + 1 (Physical)
[0x0-0x7ffffff]                    (0xXXXXX) ->
```

Enter a hex number larger than the beginning address that specifies the ending address of all Service Processor-based pattern testing.

```
CPU tests start address (Logical)
  [0xXXXXX-0xYYYYYYYY]                    (0xYYYYY) ->
```

Enter a hex number within the range 0xXXXXX-0xYYYYYYYY that specifies the beginning address of all CPU-based pattern testing. During CPU subtests, all memory is mapped to be logically contiguous and the addresses referenced from the CPU are logical.

```
CPU tests end address + 1 (Logical)
  [0xXXXXX-0xYYYYYYYY]                    (0xYYYYYYYY) ->
```

Enter a hex number within the range 0xXXXXX-0xYYYYYYYY that specifies the ending address for all CPU-based pattern testing.

```
CPU for CPU based tests [0,1,2,3] (0) ->
```

Enter the number corresponding to the CPU to be tested.

```
Enter slot mask to use []              (0xZZ) ->
```

Enter a two-digit hex number, ZZ, that specifies that memory boards to test. This prompt is only applicable for the scan-based subtests (10-25, 125, 150-158). If the slot mask is anything except the default value, some of the CPU tests fail. The intent of this slot mask is to enable/disable boards for scan-based testing. The following table lists the valid slot masks and the memory pairs that they reference.

Table mem4000–2, Slot Mask Descriptions

SLOT MASK	MEMORY PAIRS TO TEST			
	me3 mo3	me2 mo2	me1 mo1	me0 mo0
03	No	No	No	Yes
0C	No	No	Yes	No
0F	No	No	Yes	Yes
30	No	Yes	No	No
33	No	Yes	No	Yes
3C	No	Yes	Yes	No
3F	No	Yes	Yes	Yes
C0	Yes	No	No	No
C3	Yes	No	No	Yes
CC	Yes	No	Yes	No
CF	Yes	No	Yes	Yes
F0	Yes	Yes	No	No
F3	Yes	Yes	No	Yes
FC	Yes	Yes	Yes	No
FF	Yes	Yes	Yes	Yes

```
Enter OK, or :NN to return to question NN [OK]
                                                (OK) ->
```

Enter **OK** or simply <**CR**> to ignore this prompt, or may enter **NN** (where NN is any of the parameter numbers 1 to 7) to return to any of the seven previous parameter questions.

When all prompts have been answered, the screen displays a test parameter summary which echos the prompts that have been answered. The following figure illustrates an example of a Test Parameter Summary screen. The actual values and responses vary according to the input.

Figure mem4000–4, Sample Test Parameter Summary

```
                    TEST PARAMETER SUMMARY

Use default test parameters                          : n
Wait time for refresh tests                          : 10
SPU test start address (Physical)                    : 0x0
SPU tests end address + 1 (Physical)                 : 0x25000
CPU tests start address (Logical)                    : 0x25000
CPU test end address + 1 (Logical)                   : 0x80000000
CPU for CPU based tests [0,1,2,3]                    : 0
Enter Slot mask to use                               : 0xff
Enter OK, or :NN to return to question NN            : OK
```

Default Subtest Sequence

The following list depicts the default order in which the subtests are executed:

10	152	403
20	153	300
25	154	301
200	155	302
100	156	303
101	157	350
104	158	370
102	160	500
103	161	501
125	170	502
126	400	503
150	401	504
151	402	505

Class Descriptions

The following four classes are defined for the memory test:

- **Class 1** — Service Processor Based Tests of Basic Functionality
- **Class 2** — Service Processor Based Tests of ECC Functionality
- **Class 3** — CPU Based Tests of Basic Functionality
- **Class 4** — CPU Based Tests of ECC Functionality

Subtests in Class 1 and Class 2 use the Service Processor EBUS interface and the memory system scan rings to verify the operation of the memory system. These subtests verify an initial block of main memory as well as the basic memory functions required to run the CPU-based tests.

Subtests in Class 3 and Class 4 use the Service Processor in conjunction with a CPU to verify the

operation of main memory. These tests use the CPU Scalar Processor Unit to execute data pattern tests. Each subtest is described in the sections that follow.

Class 1 Subtests, Service Processor-Based Tests of Basic Functionality

The Class 1 Subtests normally verify only an initial block of main memory and the functions required to run the CPU based subtests. There is an option that allows the Service Processor-based subtests to verify all of main memory. It should be noted, however, that this takes a considerable amount of time on large systems. The Service Processor-based subtests execute exclusively with Service Processor code; they do not require the CPU to execute any instructions. They use the Service Processor interface to main memory (port E) and scan operations to perform their testing. Included in the Service Processor-based subtests, are tests of the following functions:

- Reset Function
- Arbitration Logic
- Crossbar Operation
- Initial Block of Main Memory Pattern Tests
- Zones/Unaligned Addresses
- TAS/TAC

The following table lists the Class 1 subtests with a brief description, as well as the object module and source file for each. Also included is the minimum (using two boards) and maximum time (using eight boards) needed to run each subtest.

Table mem4000–3, Class 1 Subtests

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME 4 MEMORY BOARDS (min/sec)	NOMINAL TIME 8 MEMORY BOARDS (min/sec)
10	SPU testing reset of scan rings	*mem4000.t*	*st_010.c*	00:19	00:40
20	Arbitration Win Queue	*mem4000.t*	*st_020.c*	00:02	00:08
25	Arbitration Win Logic	*mem4000.t*	*st_025.c*	00:07	00:25
100	Main Memory Testing by EBUS. Data=Alternating	*mem4000.t*	*st_100.c*	00:08	00:40
101	Main Memory Testing by EBUS. Address=Data	*mem4000.t*	*st_101.c*	00:02	00:08
102	Main Memory Testing by EBUS. Refresh	*mem4000.t*	*st_102.c*	00:14	00:20
103	Main Memory Testing by EBUS. Zones/Unaligned Addresses	*mem4000.t*	*st_103.c*	00:11	00:26
104	Main Memory Testing by EBUS. Address=Walk 1	*mem4000.t*	*st_104.c*	00:14	00:14
125	TAM via Scan Operations	*mem4000.t*	*st_125.c*	00:02	00:08
126	TAM via EBUS Operations	*mem4000.t*	*st_126.c*	00:01	00:08
170	ECC RAM Testing by EBUS. Data=ECC Patterns	*mem4000.t*	*st_170.c*	00:05	00:20
200	Crossbar Write/Read Latching	*mem4000.t*	*st_200.c*	00:11	00:40

Subtest 10, Reset Subtest

Subtest 10 resets all MCMs with a reset signal. All fields that should have a known state after reset are checked to ensure that they contain the correct information.

Subtest 20, Arbitration Win Queue

Subtest 20 pattern tests the Arbitration Gate Array win queue. The subtest first tests constant patterns to set and clear all bits, and then performs an address uniqueness test. This subtest is executed on each memory card (up to eight possible) with Error Correcting Code (ECC) and parity checking disabled.

Subtest 25, Arbitration Win Logic

Subtest 25 verifies that the Arbitration Gate Array correctly chooses between the 5 ports (A through E) during write operations. Included are the following combinations:

- All ports individually
- Port E with each of the others, A through D

- Ports A and B
- Ports C and D
- Ports A through D
- Ports A through E

This subtest is executed on each memory card with ECC and parity checking disabled.

Subtest 100, Main Memory Testing by EBUS (Data=Alternating)

Subtest 100 verifies that all bits of the memory devices can be set and cleared. This subtest uses patterns of alternating 1's and 0's then a pattern of alternating 0's and 1's for all memory locations selected. This subtest is executed with ECC and parity checking disabled.

Subtest 101, Main Memory Testing by EBUS (Address=Data)

Subtest 101 verifies the ability to uniquely address each memory location. This subtest uses an address equals data pattern, and is executed with ECC and parity checking disabled. This subtest is executed with ECC and parity checking disabled.

Subtest 102, Main Memory Testing by EBUS (Refresh)

Subtest 102 checks that the refresh circuitry is operational. This subtest writes a pattern of alternating 1's and 0's to memory, waits for a specified time period, then retrieves the data from memory and verifies that there is no deterioration to the data. This subtest is executed with ECC and parity checking disabled.

Subtest 103, Main Memory Testing by EBUS (Zones/Unaligned Addr.)

Subtest 103 verifies the ability of the memory system to perform partial word writes for all the relevant zone patterns. This subtest is executed for each memory card with ECC and parity checking disabled.

Subtest 104, Main Memory Testing by EBUS (Address=Walk 1)

Subtest 104 checks the ability to address each of the address bits of the memory subsystem. This subtest walks a one across the address bits of the RAMs (bits <31>,<28>,<26...6>) of each board pair. Then the subtest checks a memory location of each bank (8 banks, bits <5...3>) on both the even and odd slots of the pair. Slots in the configuration are tested to ensure they return correct data and transfer status. Empty slots are tested to ensure they return an illegal EBUS address from the Service Processor. This subtest is executed with ECC and parity checking disabled.

Subtest 125, TAM via Scan Operations

For each byte of a main memory address, Subtest 125 initializes the memory address, then performs a Test and Modify (TAM) operation on the byte of the word under test. Then both the

returned data and memory location where the operation was performed are verified.

Checks for the following conditions are included:

- Ability of Port E to write to each of the four bytes of a word
- Ability of Ports A-D to write to a different byte of a word
- Ability of Ports A-E to write to the same byte of a word

Subtest 125 tests TAM operation for all the memory cards with ECC and parity checking disabled.

Subtest 126, TAM via EBUS Operations

Subtest 126 checks TAS/TAC operations via EBUS transfers. Each of the four bytes of a word are verified to correctly set and clear. Subtest 126 checks this operation for all the memory cards with ECC and parity checking disabled.

Subtest 170, ECC RAM Testing by EBUS (Data=ECC Patterns)

Subtest 170 verifies that all bits of the ECC RAM memory devices can be set and cleared. This subtest uses patterns designed to set and then clear all seven ECC check-bits for all the memory locations selected. This subtest is executed with ECC and parity enabled.

Subtest 200, Crossbar Write/Read Latching

Subtest 200 verifies the Arbitration Gate Array correctly selects the win sequence between the five ports on reads. It also verifies the read and write pointers in each of the crossbar gate arrays. This subtest is executed on each memory card with ECC and parity checking disabled.

Class 2 Subtests, Service Processor Based Subtests of ECC Functionality

The Service Processor based Error Correcting Code (ECC) subtests verify the operation of the ECC capability of the memory subsystem. These subtests run exclusively with Service Processor-based code. They verify the functions by performing reads and writes, tests and modifies, and scrub operations. These operations are performed either through scan or EBUS accesses to memory with forced ECC and parity setups.

The following table lists the Class 2 Subtests as well as their minimum and maximum time to execute in minutes and seconds, and object module and source file:

Table mem4000–4, Class 2 Subtests

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME 4 MEMORY BOARDS (min/sec)	NOMINAL TIME 8 MEMORY BOARDS (min/sec)
150	Scan Testing of Normal ECC/Parity	*mem4000.t*	*st_150.c*	01:16	06:04
151	Scan Testing of Write Parity Error Detection	*mem4000.t*	*st_151.c*	00:04	00:20
152	Scan Testing of Single-Bit ECC Detection, Data-Bits	*mem4000.t*	*st_152.c*	06:57	31:00
153	Scan Testing of Single-Bit ECC Detection, Check-Bits	*mem4000.t*	*st_153.c*	02:19	11:00
154	Scan Testing of Double-Bit ECC Detection, Data-Bits	*mem4000.t*	*st_154.c*	01:26	06:40
155	Scan Testing of Double-Bit ECC Detection, Check-Bits	*mem4000.t*	*st_155.c*	00:35	03:00
156	Scan Testing of Single-Bit ECC Detection, Partial Writes	*mem4000.t*	*st_156.c*	00:29	02:00
157	Scan Testing of Single-Bit ECC Detection, TAM	*mem4000.t*	*st_157.c*	00:28	02:00
158	Scan Testing of Scrub Operation	*mem4000.t*	*st_158.c*	00:13	01:00
160	Ebus Testing of Normal ECC/Parity	*mem4000.t*	*st_160.c*	00:21	02:00
161	Ebus Testing of Read Parity Error Detection	*mem4000.t*	*st_161.c*	00:06	00:24

Subtest 150, Scan Testing of Normal ECC/Parity

This subtest verifies the normal passage of data through the ECC and parity circuitry of the memory system. It enables ECC and writes a data pattern of walking 1's via scan operations. It then performs scan read operations and verifies that no errors are detected and that the data read equals the data written.

This subtest tests all the ECC and parity generators and checkers for all the memory cards that are installed. The number of possible ECC/parity generators/checkers in the system is eight per card with a maximum of eight cards. This subtest is executed with ECC and parity enabled.

Subtest 151, Scan Testing of Write Parity Error Detection

This subtest verifies the ability of the memory system to detect parity errors on write data. A data pattern of walking 1's for each byte and parity of all 0's is written. This causes a parity error for each of the parity bits of the 4 bytes. The subtest verifies that the memory system detects the error by checking that the hard error interrupt occurs and that the proper byte is tagged as being in error.

This subtest tests all the parity generators/checkers for all the memory cards that are installed with ECC and parity enabled. The number of possible parity generators/checkers in the system is eight per card with a maximum of eight cards.

Subtest 152, Scan Testing of Single-Bit ECC Detection, Data-Bits

This subtest verifies the ability of the memory system to detect and correct single-bit errors. It writes a data pattern of walking 1's that has the ECC codes of a 0 data pattern. The pattern is written via scan operations. The ECC code is forced on writes which places the bad ECC code in memory. The subtest then performs a scan read operation. A verification is made that the correct single-bit error is detected for all 32 bits through the soft error interrupt of the Service Processor and the soft error scan ring information of the memory soft log. The test also verifies that the data read equals all 0's since the bit is toggled. The data is then reread and it is reverified that the soft error occurs since the data in memory is not corrected.

The test is then executed with the ECC code forced on reads. The same results are expected. This verifies the ECC generation logic on writing the data and the generation of the ECC code used to check what was written in memory against what was read.

The next data pattern is a walking 0 with an ECC code for all 1's. The same results are expected.

This subtest tests all the ECC generators/checkers for all memory cards that are installed with ECC and parity enabled. The number of possible ECC generators/checkers in the system is eight per card with a maximum of eight cards.

Subtest 153, Scan Testing of Single-Bit ECC Detection, Check-Bits

Subtest 153 verifies the ability of the memory system to detect and correct single-bit ECC errors in the check-bits. The subtest uses a pattern of all 0's data and the forced ECC code is selected to cause an ECC error in each of the check-bits. The subtest verifies that the correct single-bit error is detected for all seven check-bits through the soft error interrupt of the Service Processor and the soft error scan ring information of the memory soft log. It also verifies the data read is all 0's (since the error is in the check bits). The data is reread and the same results are expected.

The test is then executed with the ECC code forced on reads. The same results are expected. This verifies the ECC generation logic on writing the data and the generation of the ECC code used to check what was written in memory against what was read.

The next data pattern is 1's with ECC codes selected to cause an ECC check-bit error. The test verifies that the bit in error can be toggled in either direction.

This subtest tests all the ECC generators/checkers for all memory cards that are installed with ECC and parity enabled. The number of possible ECC generators/checkers in the system is eight per card with a maximum of eight cards.

Subtest 154, Scan Testing of Double-Bit ECC Detection, Data-Bits

Subtest 154 verifies the ability of the memory system to detect double-bit errors. The subtest writes a pattern similar to walking 1's, except it has two bits set. An ECC code for 0 data is forced on writes that places the bad ECC code in memory. The subtest then performs a main memory read operation via scan. A verification is made that the correct double-bit error is detected for all 32 bits through the hard error interrupt of the Service Processor and the hard error scan ring information of the memory soft log. Hard error interrupts are expected and multibit error information is expected in the scan results. In this subtest, data is not checked.

This subtest tests all the ECC generators/checkers for all memory cards that are installed with ECC and parity enabled. The number of possible ECC generators/checkers in the system is eight per card with a maximum of eight cards.

Subtest 155, Scan Testing of Double-Bit ECC Detection, Check-Bits

Subtest 155 verifies the ability of the memory system to detect double-bit errors in the ECC check-bits. It uses a forced ECC code, which causes a double-bit error. Hard error interrupts are expected and multibit error information is expected in the scan results. In this subtest, data is not checked.

This subtest tests all the ECC generators/checkers for all memory cards that are installed with ECC and parity enabled. The number of possible ECC generators/checkers in the system is eight per card with a maximum of eight cards.

Subtest 156, Scan Testing of Single-Bit ECC Detection, Partial Writes

Subtest 156 checks for correct operation when partial writes are performed. The subtest uses a pattern of constant 1 with an ECC code for all 0's. The three test conditions are as follows:

- Check results when pattern forces bad ECC on writes twice, then reads and checks data.
- Check results when test forces bad ECC code on first write, uses normal ECC on second write, and then reads data.
- Check results when test forces bad ECC code on first write, uses normal ECC on next two writes and then reads data.

For the first case, the subtest expects normal return from the first write since it gets the result of the first read returned. It then expects the correct soft error information on the second write which gets the result of the first write as read data. The subtest also expects soft errors on the read operation as it is getting the result of the second write returned when it reads. The data is then checked for all 0's.

For the second case, the subtest expects normal return from the first write since it gets the result of the first read returned. It then expects the correct soft error information on the second write which gets the result of the first write as read data. Since the second write is performed with good ECC, the subtest expects normal return with no ECC errors on the read operation. The data is then checked for the constant 1 pattern.

For the third case, the subtest expects normal return from the first write since it gets the result of the first read returned. It then expects the correct soft error information on the second write which gets the result of the first write as read data. Since the second write is performed with good ECC the subtest expects normal return with no ECC errors on the third write operation. Since the third write is performed with good ECC, the subtest expects normal return with no ECC errors for the read operation. The data is then checked for the constant 1 data. This subtest is executed with ECC and parity enabled.

Subtest 157, Scan Testing of Single-Bit ECC Detection, TAM

Subtest 157 checks for correct operation when test and modify operations are performed. The subtest uses a pattern of constant 1 with an ECC code for all 0's. The three test conditions are as follows:

- Check results when pattern forces bad ECC on test and modifies twice, then reads and checks data.
- Check results when test forces bad ECC code on first test and modify, uses normal ECC on second test and modify, and then reads data.
- Check results when test forces bad ECC code on first test and modify, uses normal ECC on next two Test and Modifys and then reads data.

For the first case, the subtest expects normal return from the first test and modify since it gets the result of the first read returned. It then expects the correct soft error information on the second test and modify which gets the result of the first test and modify as read data. The subtest also expects soft errors on the read operation as it is getting the result of the second test and modify returned when it reads. The data is then checked for all 0's.

For the second case, the subtest expects normal return from the first test and modify since it gets the result of the first read returned. It then expects the correct soft error information on the second test and modify which gets the result of the first test and modify as read data. Since the second test and modify is performed with good ECC, the subtest expects normal return with no ECC errors on the read operation. The data is then checked for the constant 1 pattern.

For the third case, the subtest expects normal return from the first test and modify since it gets the result of the first read returned. It then expects the correct soft error information on the second test and modify which gets the result of the first test and modify as read data. Since the second test and modify is performed with good ECC, the subtest expects normal return with no ECC errors on the third test and modify operation. Since the third test and modify is performed with good ECC, the subtest expects normal return with no ECC errors for the read operation. The data is then checked for the constant 1 data. This subtest is executed with ECC and parity enabled.

Subtest 158, Scan Testing of Scrub Operation

Subtest 158 verifies the scrub operation. First, a pattern of walking 1 is written with forced bad ECC codes to memory via scan. The subtest then reads from memory via scan and verifies the correct soft error occurs and the data is correct. The subtest then performs a scrub operation to memory, rereads the data and verifies that a normal return occurs, after which the data is verified.

This subtest is executed with ECC and parity enabled.

Subtest 160, EBUS Testing of Normal ECC Parity

Subtest 160 verifies the normal passage of data through the ECC and parity circuitry of the memory system. It enables ECC and writes a pattern of walking 1's data via EBUS transfers. It then performs read operations via EBUS transfers and verifies that no errors are detected and that the data read equals the data written.

This subtest tests all the ECC and parity generators and checkers for all memory cards that are installed with ECC and parity enabled. The number of possible ECC/parity generators/checkers in the system is eight per card for a maximum of eight cards.

Subtest 161, EBUS Testing of Read Parity Error Detection

Subtest 161 verifies the ability of the Service Processor to detect parity errors on reads from the memory subsystem. The subtest writes data with good parity to memory via the EBUS. It then sets the forced read parity-bit of the scan ring and performs a read operation via the EBUS. The subtest verifies that the Service Processor EBUS logic detects the parity error and returns this information in the EBUS log register. This subtest is executed with ECC and parity enabled.

Class 3 Subtests, CPU-based Subtests of Basic Functionality

The CPU-based subtests verify the bulk of main memory. They use a combination of Service Processor and CPU code to execute. The Service Processor code is primarily responsible for determining the test areas that need to be tested, the setup of CPU pattern parameters and the monitoring and checking of CPU test result codes.

The CPU code for these subtests is rather basic in nature. It receives pattern description information from the Service Processor. This information is essentially the pattern to utilize and the address range to test. The CPU code is then capable of performing write/read/verify sequences using these parameters.

The following table lists the Class 3 subtests, including the object module and source file for each:

Table mem4000–5, Class 3 Subtests

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME 4 MEMORY BOARDS (min/sec)	NOMINAL TIME 8 MEMORY BOARDS (min/sec)	TIMEOUT LIMIT (min/sec)
300	Main Memory Testing by CPU. Data=Alternating	*mem4000.t*	*st_300.c*	00:34	02:10	2:40
301	Main Memory Testing by CPU. Data=Switching	*mem4000.t*	*st_300.c*	00:24	01:36	2:40
302	Main Memory Testing by CPU. Address=Data	*mem4000.t*	*st_300.c*	00:11	00:44	2:40
303	Main Memory Testing by CPU. Refresh	*mem4000.t*	*st_300.c*	00:11	00:44	2:40
350	CPU Testing of Memory Bank Independence	*mem4000.t*	*st_350.c*	00:08	00:15	2:40
370	ECC Ram Testing By CPU. Data=ECC Patterns	*mem4000.t*	*st_370.c*	00:22	01:30	2:40
500	CPU Testing: Addressing Word Write, Address Increment=Byte	*mem4000.t*	*st_500.c*	00:13	01:00	2:40
501	CPU Testing: Addressing Word Write, Address Increment=Half Word	*mem4000.t*	*st_500.c*	00:15	01:00	2:40
502	CPU Testing: Addressing Word Write, Address Increment=Word	*mem4000.t*	*st_500.c*	00:26	02:00	2:40
503	CPU Testing: Addressing Word Write, Address Increment=Long Word	*mem4000.t*	*st_500.c*	00:14	01:00	2:40
504	CPU Testing: Addressing Word Write, Address Increment.=Bank	*mem4000.t*	*st_500.c*	00:03	00:15	2:40
505	CPU Testing: Addressing Word Write, Address Increment=Row	*mem4000.t*	*st_500.c*	00:01	00:10	2:40

Subtest 300, Main Memory Testing by CPU (Data=Alternating)

Subtest 300 verifies the ability to set and clear all bits of the memory devices. This subtest uses a pattern of alternating 1's and 0's, a pattern of alternating 0's and 1's, and an odd parity pattern for all memory locations. ECC and parity are enabled in these subtests.

Subtest 301, Main Memory Testing by CPU (Data=Switching)

Subtest 301 verifies the ability to alternate all 1's data with all 0's data between addresses on the same memory card. ECC/Parity is enabled in this subtest.

Subtest 302, Main Memory Testing by CPU (Address=Data)

Subtest 302 verifies the ability to uniquely address each memory location. This subtest uses an address equals data pattern. ECC/Parity is enabled in this subtest.

Subtest 303, Main Memory Testing by CPU (Refresh)

Subtest 303 checks that the refresh circuitry is operational. This subtest writes a pattern of alternating 1's and 0's to memory, waits for a specified time period, then retrieves the data from memory and verifies that no deterioration of the data has occurred. This subtest is executed with ECC and parity enabled.

Subtest 350, CPU Testing of Memory Bank Independence

Subtest 350 initializes all eight addresses that are at the same chip address. It then writes a data pattern to a selected chip address. A read is then made of the selected chip address for all eight banks of the memory card. The subtest verifies only that the selected write occurred. The testing is repeated for all eight banks. This subtest is executed with ECC and parity enabled.

Subtest 370, ECC RAM Testing by CPU (Data=ECC Patterns)

Subtest 370 verifies the ability to set and clear all bits of the memory devices. This subtest uses patterns designed to set and clear all the ECC check-bits in the ECC RAMS. ECC and parity are enabled in these subtests.

Subtests 500-505, Memory Addressing Subtests

Subtests 500-505 attempt to find addressing problems that could occur in the memory system. They use selected addresses related to the memory system layout.

The addressing subtests initialize a section of memory and then write a pattern in one of the following ways:

- Distributed by Byte
- Distributed by Half Word
- Distributed by Word
- Distributed by Two Words (Long Word)
- Distributed by Bank
- Distributed by Row

The subtests then read all the data back and verify the results. These subtests are executed with ECC and parity enabled.

Class 4 Subtests, CPU-based Tests of ECC Functionality

The following table is a complete list of the Class 4 subtests, with object module, source file, and a minimum and maximum test time in seconds for each:

Table mem4000–6, Class 4 Subtests

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME 4 MEMORY BOARDS (min/sec)	NOMINAL TIME 8 MEMORY BOARDS (min/sec)	TIMEOUT LIMIT (min/sec)
400	CPU Testing of Normal ECC/Parity	*mem4000.t*	*st_400.c*	00:05	00:10	2:40
401	CPU Testing of Read Parity Error Detection	*mem4000.t*	*st_401.c*	00:04	00:10	2:40
402	CPU Testing of Single Bit ECC Detection	*mem4000.t*	*st_402.c*	00:02	00:05	2:40
403	CPU Testing of Double Bit ECC Detection	*mem4000.t*	*st_403.c*	00:02	00:05	2:40

Subtest 400, CPU Testing of Normal ECC/Parity

Subtest 400 verifies normal passage of data through the ECC and parity circuitry of the memory system. This subtest writes a pattern of walking 1's data. It then reads the data and verifies that no parity or ECC errors are detected and that the data read equals the data written. This subtest is executed with ECC and parity enabled.

Subtest 401, CPU Testing of Parity Error Detection

Subtest 401 verifies the ability of the CPU to detect parity errors on read data from the memory subsystem. This subtest writes a pattern to memory and then sets the forced read parity-bit of the scan ring. It then starts the CPU and verifies that a hard error occurs and that the error is caused by a parity error. This subtest is executed with ECC and parity enabled.

Subtest 402, CPU Testing of Single-Bit ECC Detection

Subtest 402 verifies the ability of the memory system to detect and correct single-bit errors. It tests for one single-bit error in each of the memory banks. A pattern having incorrect ECC codes is written via scan. The subtest then starts the CPU and reads the data. It verifies that the correct single-bit error is detected for all the memory banks through the soft error interrupt and the soft error scan ring information of the soft log. This subtest is executed with ECC and parity enabled.

Subtest 403, CPU Testing of Double-Bit ECC Detection

This subtest verifies the ability of the memory system to detect double-bit errors. A double-bit error is written to memory via scan operations. The subtest starts the CPU which reads the address with the error and verifies that a double-bit error is detected through the hard error interrupt and the soft error scan ring information of the soft log. This subtest is executed with ECC and parity enabled.

Test Error Messages

The following format is used to display *mem4000* error messages:

Figure mem4000–5, Sample Test Error Message Format

```
*****  Thu Dec 30 09:24:02  *****
Test:    mem4000.t  1.6    Class: 2    Subtest: 152 1.3   Count: 1   Error: 0
Failed:  Scan testing of single bit ECC detection. Data Bits

Checking forced write ECC codes
Pattern: Walk 1; ECC for data = 0
Slot: 4 Bank: 0
loop cnt: 0

 FAIL    OFFSET/
 TYPE     ADDR       EXP         ACT       COMMENTS
ECC     08000000  08000000   40000000  MCM ecc error addr

paused at fail in subtest 152
```

The first three lines in the figure above are *LIB TEST* information, which includes the day, date, time and year, followed by the test name, *mem4000.t*, the test revision number (1.6), the Subtest Class (2), the Subtest number (152), and the Subtest revision number (1.3). The count and error information can be ignored. The next line contains the subtest name, which is preceded by the Failed: header.

The next four lines in the figure are the set-up information, and contain what the test was testing when it failed. In this case, the test was checking forced write ECC codes. The pattern was a Walk 1, the ECC for data was 0, the slot was 4, the memory bank, 0, and the loop count was 0.

The next lines contain the actual error message information. The following headers are used: the fail type, the offset or address, the expected value, the actual value, and a comment describing the failure.

The last two lines simply give a brief explanation of the failure. The fail type in the actual error information line may be any of the following:

- DATA — Data value miscompared
- PARITY — Parity value miscompared
- ECC — Error Correcting Code (ECC) miscompared
- MM WR — Unexpected return code from an EBUS write operation
- MM RD — Unexpected return code from an EBUS read operation
- IER — Unexpected Interrupt Error Register
- ESR — Unexpected Interrupt Source Register
- MM TAM — Unexpected return code from an EBUS Test and Modify operation
- ADDR — Address returned in error

The comments portion of the error message information gives further explanation of what the test expected or did not expect when the test failed. There are 18 possible comment lines, each of which are listed and explained here.

- `hard_err failure` — Hard error scan field returned wrong data
- `par_err failure` — Parity error scan field returned wrong data
- `wr_dat failure` — Write data scan field returned wrong data
- `wr_par failure` — Write parity scan field returned wrong data
- `log cycle type failure` — Cycle type scan field returned wrong data

- `xfr from MM` — Unexpected status from an EBUS transfer on an EBUS write operation
- `xfr to MM` — Unexpected status from an EBUS transfer on an EBUS read operation
- `TAM of MM` — Unexpected status from an EBUS transfer on a test and modify of memory operation

- `MCM hard err type` — MCM had unexpected hard error type
- `MCM soft err type` — MCM had unexpected soft error type
- `SEL: no memory soft error` — Failure to receive memory soft error in Soft Error Log Register
- `MCM ecc value` — Unexpected value for ECC code

- `ISR:no soft err` — Failure to receive soft error in Interrupt Status Register
- `ISR: no soft err expected` — Soft error received by Interrupt Status Register when no error was expected
- `ISR: no hard error` — Failure to receive hard error in ISR register
- `ESR: no MCM error` — Failure to receive MCM error in ESR register
- `ISR: unable to clear soft err` — Unable to clear ISR soft error
- `ISR: unable to clear hard err` — Unable to clear ISR hard error

SPU UNIX Error Messages

For a common list of possible error messages resulting from SPU UNIX, refer to the appendix on SPU UNIX Error Messages.

cpx4000

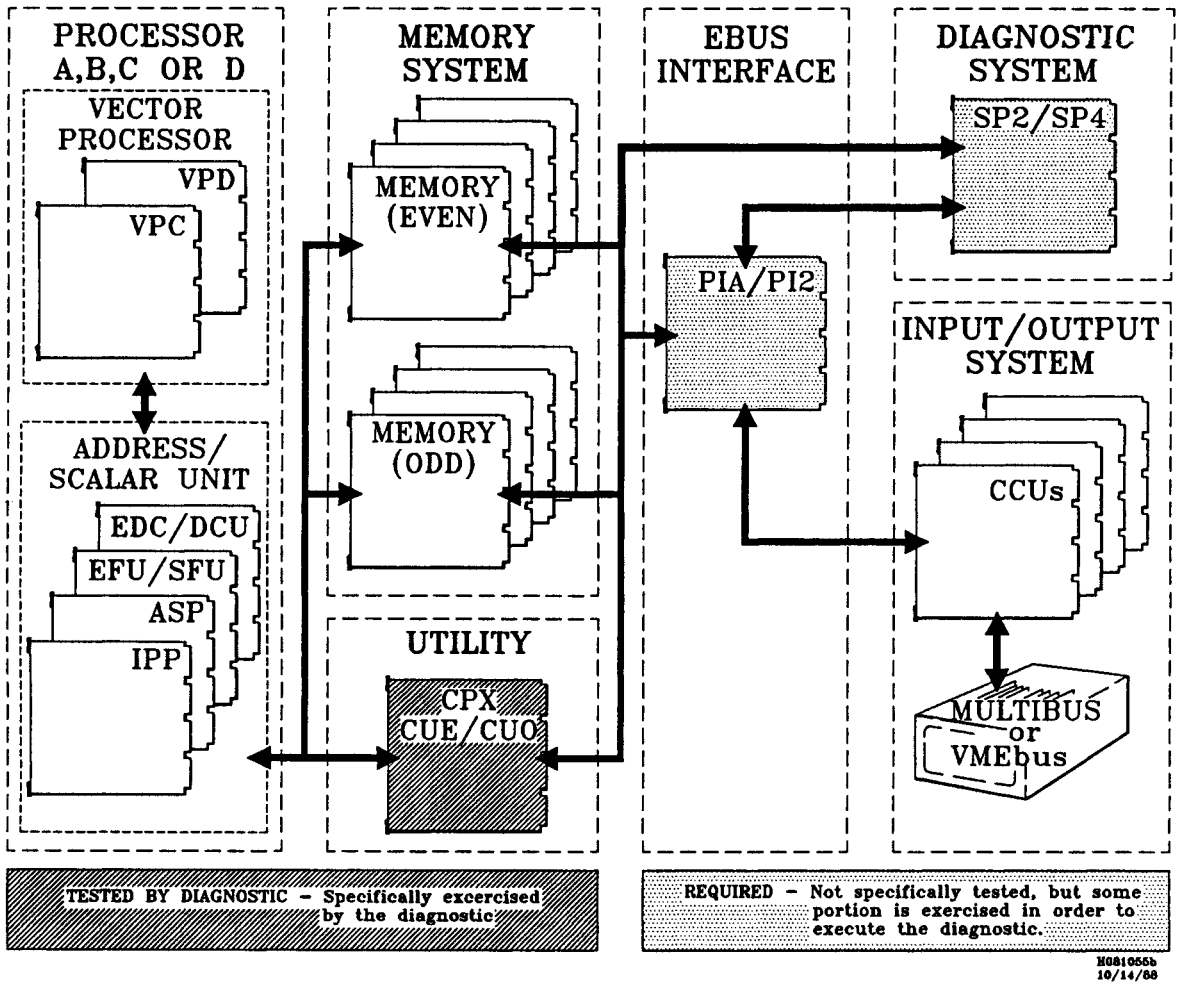
CPX Functional Test

Overview

The *cpx4000* is a stand-alone diagnostic for CPXs, CUEs, and CUOs. All functional blocks of the CPU Utility Board(s) are tested. This document assumes that the reader is familiar with these functional blocks.

> **NOTE**
>
> The term "Service Processor" and "SP" are used generically to represent either SP2 or SP4 depending on the system configuration under test. The term "CPU Utility Board(s)" is used to represent either a CPX or a CUE / CUO combination depending on the system under test. Also, "PBUS Interface Board(s)" is used to represent either a PIA, a PI2 installed in the PIY slot, or two PI2s installed in the PIX and PIY slots, depending on the system configuration under test.

Figure cpx4000–1, Functional Areas Tested by *cpx4000*



Prerequisites and Required Equipment

When cpx4000 is executed on a system containing a CPX, only a CPX, PIA, and SP2 are required. When executed on a system containing a CUE and CUO, only a CUE, CUO, PI2 (in the PIY slot), and SP4 are required. Any other boards installed in the system will not affect execution.

This test assumes *spu1000* subtests and applicable *spu4000* subtests pass. The PIA is required for both clock generation and EBUS arbitration, the PI2 (in the PIY) slot for EBUS arbitration only. A large number of *cpx4000* subtests require the EBUS to be functional.

Table cpx4000–1, Required Functional Boards

BOARDS	TESTS TO VERIFY
SP2 / SP4	*spu1000, spu4000*
PIA / PI2	*pia4000*

Test Invocation

The *cpx4000* test requires no input, other than the specification of individual subtests or classes. All subtests verify that the necessary hardware is present before attempting to execute. If the hardware is not present, a message is displayed indicating what missing hardware is required. Also, if a subtest is not applicable to the specific CPU Utilities Board(s) installed, the message "`not executed subtest not applicable`" will be displayed.

To invoke the *cpx4000* test, use the procedure shown in the following figure. All responses in **boldface** are entered by the user. The prompts and responses would appear sequentially on the screen, one line at a time. All the prompts and responses are shown in one figure for convenience.

Figure cpx4000–2, Test Invocation Sequence

```
(spu)>  cd /mnt/test
(spu)>  sysreset
(spu)>  dshell

CONVEX DIAGNOSTIC SHELL

:  test cpx4000 [-c [class numeral(s)]] [-s [subtest numeral(s)]] [+> filename]
```

NOTE

After entering **dshell**, specific *dshell* parameters may be changed. Refer to the "Dshell and Iscan Overview" chapter of this manual for more information.

Entering only **test cpx4000** executes all the *cpx4000* subtests sequentially. Execute a specific class(es) of subtest(s) or one or more individual subtests by using the **-c** or **-s** options, respectively. Detailed information for using these options can be found in the "Dshell and Iscan Overview" chapter of this manual. The [+>*filename*] option allows the test results to be appended to *filename*.

Test Parameter Menu

There is no test parameter menu with *cpx4000*; however, the test may be invoked using the **-c** and **-s** options to specify subtests.

Default Sequence

Those subtests specified upon test invocation will be executed in the order specified. Entering **test cpx4000** executes all subtests in order.

Class Descriptions

There are three classes of subtests in *cpx4000*.

Class 1 subtests give a quick overview of many (but not all) of the functional blocks of the CPU Utility Board(s).

Class 2 subtests give a necessary, but longer, overview of much of the remaining functionality of the CPU Utility Board(s). Both Class 1 and Class 2 subtests must be executed to have any confidence in the board(s).

Class 3 subtests further test functionality initially tested in Class 1 and Class 2 subtests, however they take significantly longer to execute. These subtests should be executed if the hardware integrity is at all suspect.

Subtest Descriptions

The following sections describe the *cpx4000* subtests. The following table lists each subtest, its class, object module, source file, and execution time.

> **NOTE**
>
> There are two execution time columns in the following table. The **CPX TIME** column represents the time required to execute the subtest on a system containing a CPX. The **CUE / CUO TIME** column represents the time required to execute the subtest on a system containing a CUE and CUO.

Table cpx4000–2, cpx4000 Subtests

SUBTEST	CLASS	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	CPX TIME	CUE / CUO TIME
100	1	Reset State Verification	*cpx4000.t*	*st_100.c*	0:04	0:02
105	1	Arbitration Win Logic	*cpx4000.t*	*st_105.c*	0:01	NA[2]
125	1	Low Level Error Processing	*cpx4000.t*	*st_125.c*	0:01	0:01
130	1	Crossbar Data Parity, 8 LSB	*cpx4000.t*	*st_130.c*	0:01	NA[2]
135	1	Crossbar Data Parity, 8 MSB	*cpx4000.t*	st_135.c	0:01	NA[2]
140	1	Timer and PCM Hard Error	*cpx4000.t*	*st_140.c*	0:01	0:01
155	1	Low Level Bad Bank Select Hard Errors	*cpx4000.t*	*st_155.c*	0:01	0:01
160	1	Processor Soft Errors	*cpx4000.t*	*st_160.c*	0:01	0:01
161	1	CU Soft Errors	*cpx4000.t*	*st_161.c*	0:01	0:03
165	1	Illegal I/O Address at Error Logic	*cpx4000.t*	*st_165.c*	0:01	0:01
170	1	Illegal I/O Address at Timers	*cpx4000.t*	*st_170.c*	0:01	0:01
175	1	Illegal I/O Address at PCM	*cpx4000.t*	*st_175.c*	0:01	0:01
180[1]	1	Overall I/O Address Test	*cpx4000.t*	*st_180.c*	0:25	0:21
181[1]	1	Address and Data Trapping Test	*cpx4000.t*	*st_181.c*	0:23	0:19
185[1]	1	Invalid PCM Reference	*cpx4000.t*	*st_185.c*	0:01	0:01
195	1	Communication Register Parity Error	*cpx4000.t*	*st_195.c*	0:01	0:01
200	1	High Level Bad Bank Select Hard Error	*cpx4000.t*	*st_200.c*	0:01	0:01
205[1]	1	Nonexhaustive PCM Pattern Test	*cpx4000.t*	*st_205.c*	0:21	0:20
206[1]	3	Exhaustive PCM Pattern Test	*cpx4000.t*	*st_206.c*	3:13	10:15
210[1]	1	General Multiple Bank R&M Bit Pattern Test	*cpx4000.t*	*st_210.c*	0:49	0:39
211[1]	2	Specific Multiple Bank R&M Bit Pattern Test	*cpx4000.t*	*st_210.c*	1:00	0:48
212[1]	2	General Individual Bank R&M Bit Pattern Test	*cpx4000.t*	*st_210.c*	3:58	6:20
213[1]	3	Specific Individual Bank R&M Bit Pattern Test	*cpx4000.t*	*st_210.c*	7:33	12:03
215	1	Communication Register Pattern Test	*cpx4000.t*	*st_215.c*	3:53	1:59
217	2	Communication Register Functionality Test	*cpx4000.t*	*st_217.c*	5:22	2:55
220[1]	1	TOC Functionality Test	*cpx4000.t*	*st_220.c*	0:26	0:21
225[1]	1	PIT Functionality Test	*cpx4000.t*	*st_225.c*	0:26	0:21
		Total Execution Time			28:22	37:05

[1] Requires functional EBUS arbitrator (PIA or PIY)
[2] Subtest is not currently applicable to these boards

Subtest 100, Reset Test

Subtest 100 is a scan based test that asserts backplane reset to every board installed in the system, then issues 40 normal clocks. After removing reset, every field on the CPU Utility Board(s) which should have a known state after a backplane reset, is checked to ensure that it contains the correct value. Then the CPU Utility Board(s) is defaulted, and the fields checked again.

Subtest 105, Arbitration Win Logic

Subtest 105 applies only to systems with a CPX. When attempted on other systems, the message "not executed subtest not applicable" is displayed.

Subtest 105 is a scan based test that verifies the arbitration gate array grants the correct port access, depending on what bus(es) are requesting, and the previous order of access. This subtest scans information into the arbitration gate array on the CPX, clocks the board and again scans the board to determine if the correct port was granted access.

Subtest 125, Low Level Error Processing

Subtest 125 is a scan based test that checks that the Service Processor (SP) can sense a hard error from all applicable CPU Utility Board(s). For each CPU utility board installed in a backplane, all possible combinations of the hard error and halt disable signals are manipulated to insure that a hard error is sensed (by the Service Processor) only when hard error is set and halt disable clear.

Subtest 130, Crossbar Data Parity (8 LSB)

Subtest 130 is a scan based test that applies only to systems with a CPX. When attempted on other systems, the message "not executed subtest not applicable" is displayed.

Subtest 130 verifies the parity checker for the lower 8 bits of write data coming out of the crossbar gate arrays. Various combinations of data and parity are used, with the Service Processor checking to insure it senses a hard error only when the data and parity do not match.

Subtest 135, Crossbar Data Parity (8 MSB)

Subtest 135 is a scan based test that applies only to systems with a CPX. When attempted on other systems, the message "not executed subtest not applicable" is displayed.

Subtest 135 is similar to subtest 130, except that it verifies the parity checker for the upper 8 bits of write data coming out of the crossbar gate arrays.

Subtest 140, Timer and PCM Hard Errors

Subtest 140 is a scan based test that tests the sending of the Programmable Interrupt Timer (PIT), Time of Century Counter (TOC) and Physical Configuration Map (PCM) hard errors. Various information is scanned into the appropriate scan ring, and the Service Processor checks that it senses a hard error only when one is expected.

Subtest 155, Bad Bank Select Hard Error

Subtest 155 is a scan based tests that checks that the Service Processor senses a hard error when any of the "illegal start" bits are set in the appropriate scan ring. Various combinations of the bits are checked.

Subtest 160, Processor Soft Errors

Subtest 160 is a scan based test that tests the ability of the Service Processor to sense a CPU soft error through the appropriate CPU Utility Board(s). All processor soft errors are verified, first by

setting the appropriate bit in the scan ring on the appropriate CPU Utility Board(s), then by setting bits on any installed CPU boards.

Subtest 161, CU Soft Errors

Subtest 161 is a scan based test that checks the ability of the Service Processor to sense modified bit soft errors on the CPU Utility Board(s). All possible legal and illegal conditions are checked, and the Service Processor checks to make sure it senses a soft error only when expected.

Subtest 165, Illegal I/O Address at Low Level Logic

Subtest 165 is scan based and tests the ability of the Service Processor to sense a hard error generated at the low level error logic's address checking circuitry. Various legal and illegal addresses are attempted.

Subtest 170, Illegal I/O Address at Timers

Subtest 170 is scan based and tests the ability of the Time of Century (TOC) and the Programmable Interrupt Timer (PIT) circuitry to detect an address error, and transmit it to the Service Processor. Various legal and illegal addresses are attempted, with the Service Processor checking to insure it senses an error only when one is expected.

Subtest 175, Illegal I/O Address at PCM

Subtest 175 is a scan based test that tests the PCM to insure the Service Processor can detect a hard error which results from an invalid address. Various legal and illegal addresses are attempted.

Subtest 180, Overall I/O Address Test

Subtest 180 tests the ability of the Service Processor to detect a hard error for various I/O space addresses over the entire range of I/O space. Both boundary and interior writes are attempted to locations in each legal and illegal section of system I/O space. This subtest enables clocks to an initialized I/O subsystem, and attempts accesses over the EBUS.

Subtest 181, Address and Data Trapping Test

Subtest 181 attempts the same I/O space writes as Subtest 180, but Subtest 181 ensures that the address (and data for the CPX) trapped by each illegal write are correct by reading the scan ring and checking what was actually trapped against the expected value(s). This subtest enables clocks to an initialized I/O subsystem, and attempts accesses over the EBUS, but also uses scan to check the operations.

Subtest 185, Invalid PCM Reference

Subtest 185 tests the ability of the Service Processor to sense a hard error that results from a reference to a main memory location zero, which is marked nonexistent by the PCM on the CPU Utility Board(s). Memory boards are not required for this subtest. This subtest is scan based, but does use system I/O space transactions over the EBUS to read and write the PCM on the CPU Utility Board(s).

Subtest 195, Communication Register Parity Error

Subtest 195 tests the ability of the Service Processor to detect the hard error which results from a parity error on communication register data. Various legal and illegal combinations of data and parity are scanned into the appropriate CPU Utility Board(s), the board clocked, and the Service Processor checks to insure it received and error only if one was expected. Each of the 10 parity checkers (8 for data bits and 2 for lock bits) are verified. This subtest is scan based.

Subtest 200, High Level Bad Bank Select Hard Error

Subtest 200 is a scan based test that checks that the Service Processor senses a hard error when an illegal start is generated by the arbitration gate array on the CPX or CUE. Each of the 5 illegal "segments" is individually checked, by scanning the start into the gate array and clocking the board. The Service Processor checks to insure it senses a hard error.

Subtest 205, Nonexhaustive PCM Pattern Test

Subtest 205 pattern tests the Physical Configuration Map (PCM). The PCM is saved before the test, and restored after the test completes. Other than initialization via scan, this subtest uses only system I/O space reads and writes.

The CPX version of the subtest tests the PCM RAM associated with each processor separately; processor B's PCM RAM is tested first, processor A's second. The CUE / CUE version tests each processor's odd and even bank of PCM RAM separately, in the following order:

- A even
- B even
- C even
- D even
- A odd
- B odd
- C odd
- D odd

First, a one is written to all the PCM locations, and then the entire PCM is read to ensure that every bit is set. Second, a zero is written to all the PCM locations, and then the entire PCM is read to ensure that every bit is zero. Third, a one is written to every seventh bit of the PCM, and the entire PCM is read to ensure that only every seventh bit is a one.

Subtest 206, Exhaustive PCM Pattern Test

Subtest 206 pattern tests the Physical Configuration Map (PCM). The PCM is saved before the test, and restored after the test completes. Other than initialization via scan, this subtest uses only system I/O space read and writes.

The CPX version of the subtest tests the PCM RAM associated with each processor separately; processor B's PCM RAM is tested first, processor A's second. The CUE / CUE version tests each processor's odd and even bank of PCM RAM separately in the following order:

- A even
- B even
- C even
- D even
- A odd
- B odd
- C odd
- D odd

First, a zero is written to all the PCM locations, and the entire PCM is read to ensure that every bit is clear. Second, for each location of the PCM bank under test, the following occurs:

- The location is set to one
- Every PCM location is read and checked to insure that each bit contains the expected value
- The location is cleared to zero

Subtests 210 & 212, General Multiple and Individual Bank R&M Bits

Subtests 210 and 212 pattern test the Referenced and Modified (R&M) bits. The CPX has 8 "banks" of R&M bits. The CUE / CUO pair has 16 banks of R&M bits. Subtest 210 tests all banks together, and Subtest 212 tests each of the banks separately.

The R&M bits are tested by writing a pattern to all locations, then reading it back and checking for errors. The first pattern is setting every location, the second clearing, and the third is setting every seventh bit and clearing all the others. Other than initialization via scan, this subtest uses only I/O space reads and writes.

Subtests 211 & 213, Specific Multiple & Individual Bank R&M Bits

Subtests 211 and 213 exhaustively test random R&M bits. The CPX has 8 "banks" of R&M bits. The CUE / CUO pair has 16 banks of R&M bits. Subtest 211 tests all banks together, and Subtest 213 tests each bank separately. Other than initialization via scan, this subtest uses only I/O space reads and writes.

First, all locations are cleared. Then, for each address to test, the following operations are performed:

- Set both the referenced and the modified bits by writing to the location in system I/O space
- Read every referenced and modified bit and verify that only the bits at the location set are set
- Clear any set referenced and modified bits

Subtest 215, Communication Register Pattern Test

Subtest 215 pattern tests the communication registers and their lock bits. The communication registers and their lock bits are read and written via scan.

First, all lock bits are cleared to zero, then checked to insure they are cleared. Second, for each lock bit, the following occurs:

- The lock bit is set to one
- That lock bit is read and checked to insure it is set
- The lock bit is cleared to zero

Third, the communication registers are pattern tested with the following patterns:

- Ones (0xFFFFFFFF)
- Ones/Zeros (0xAAAAAAAA)
- Zeros/Ones (0x55555555)
- Incrementing pattern (address uniqueness pattern)
- Zeros (0x00000000)

Subtest 217, Communication Register Functionality Test

Subtest 217 tests the functionality of each legal Communication Register operation. Only scan is used for this test. The order of testing is as follows:

1. All 16 modified bits are cleared and checked to ensure that they cleared
2. Each modified bit is individually checked by doing the following:
   - The bit under test is set
   - All 16 modified bits are read and checked to make sure that only the bit under test is set
   - The bit under test is cleared
3. Each communication register operation is tested with a register associated with each modified bit as follows:
   - The register, its lock bit, and the associated modified bit is cleared
   - The register is read to ensure it contains zeros
   - The operation to be checked is performed
   - Each of the 16 modified bits is checked (The one associated with the register under test is checked to verify it is in the expected state, the others are checked to ensure they are clear)
   - The register under test is read, and the data, lock bit, and modified bits are checked to ensure they are as expected

Subtest 220, TOC Functionality Test

Subtest 220 first pattern tests the TOC Data Register. Then, it insures the TOC increments when enabled, and does not increment when disabled. Next the subtest insures the TOC does not increment during multiple reads. It insures every 16 bit word of the TOC does increments properly.

Subtest 225, PIT Functionality Test

Subtest 225 first pattern tests the PIT registers. Then it insures the PIT increments properly when enabled, and does not increment when disabled. Next, the PIT is set to interrupt the Service Processor. The Service Processor insures it received the interrupt and attempts to clear the interrupt. The PIT's status register is tested to insure the full and overflow bits were set correctly. The status register is reread to insure it cleared correctly. The subtest checks that the interrupt number and next count registers of the PIT have not changed. It then allows multiple interrupts of the Service Processor and rechecks the functionality of the PIT status register.

Test Error Messages

The following sections provide a complete description of the CPX Functional Test error messages. Sections are provided for each group of subtests, and their corresponding error messages.

Subtest 100 Errors

The following error messages could result from an error in subtest 100:

```
CPX reset error after scn_reset
field: IIII   actual: 0xJJJJJJJJ   expected: 0xKKKKKKKK
```
or:
```
CPX reset error after cpx_default
field: IIII   actual: 0xJJJJJJJJ   expected: 0xKKKKKKKK
```
or:
```
CUE reset error after scn_reset
field: IIII   actual: 0xJJJJJJJJ   expected: 0xKKKKKKKK
```
or:
```
CUE reset error after cpx_default
field: IIII   actual: 0xJJJJJJJJ   expected: 0xKKKKKKKK
```
or:
```
CUO reset error after scn_reset
field: IIII   actual: 0xJJJJJJJJ   expected: 0xKKKKKKKK
```
or:
```
CUO reset error after cpx_default
field: IIII   actual: 0xJJJJJJJJ   expected: 0xKKKKKKKK
```
where:

IIII is the name of the field
JJJJJJJJ is the actual value of the field, in hex
KKKKKKKK is the expected value of the field, in hex

Subtest 105 Errors

The following error message could result from an error in subtest 105:

```
arbitration gate array win error
test: I   actual: J   expected: K
```
where:

I is a character representing the port under test
J is the actual win value, in hex
K is the expected win value, in hex

Subtest 125 Errors

Any of the following error messages could result from an error in subtest 125:

```
CPX halt_disable: I   hard_err: J
```
or:
```
CUE halt_disable: I   hard_err: J
```
or:
```
CUO halt_disable: I   hard_err: J
```
where:

I is the status of the halt disable bit
J is the status of the hard error bit

Subtest 130 Errors

Any of the following error messages could result from an error in subtest 130:

```
hard error set when expected clear
loop: I   data: 0xJJ   parity: K
```

or:

```
hard error clear when expected set
loop: I   data: 0xJJ   parity: K
```

or:

```
hard error set after attempting clear
loop: I   data: 0xJJ   parity: K
```

where:

I is the number of the times the loop has been executed, in decimal

JJ is the value written to the field, in hex

K is the parity written for that data, in decimal

Subtest 135 Errors

Any of the following error messages could result from an error in subtest 135:

```
hard error set when expected clear
loop: I   data: 0xJJ   parity: K
```

or:

```
hard error clear when expected set
loop: I   data: 0xJJ   parity: K
```

or:

```
hard error set after attempting clear
loop: I   data: 0xJJ   parity: K
```

where:

I is the number of the times the loop has been executed, in decimal

JJ is the value written to the field, in hex

K is the parity written for that data, in decimal

Subtest 140 Errors

Any of the following two line error messages could result from an error in subtest 140.

One of the following lines:

```
hard error clear when expected set
hard error set when expected clear
hard error set after attempting clear
```

Followed by one of the following lines:

```
CPX misc.pcmmerr: I
CPX pcm_pit_hard: 0xI
CUE pcm_rame_err: I
CUE pcm_meme_err: I
CUE ti_err: I
CUO pcm_ram_err: I
CUO pcm_mem_err: I
```

where:

I is the current value of the specified scan ring field, in hex

Subtest 155 Errors

Any of the following two line error messages could result from an error in subtest 155.

One of the following lines:

```
hard error clear when expected set
hard error set when expected clear
hard error set after attempting clear
```

Followed by one of the following lines:

```
CPX start_err: 0xI
CUE m2_start: I
CUE m4_start: I
CUE m5_start: I
CUE m6_start: I
CUE m7_start: I
```

where:

I is the current value of the specified scan ring field, in hex

Subtest 160 Errors

Any of the following multi-line error messages could result from an error in subtest 160.

One of the following lines:

```
soft error clear when expected set
soft error set after attempting clear
```

Followed by one of the following groups of line(s):

```
VC[1] softerr: I
CPX misc.vcb_soft: I
LCPX vpcb_soft: I

CPX misc.vcb_soft: I
LCPX vpcb_soft: I

DC[1] soft_err: I
CPX misc.dcb_soft: I
LCPX dcub_soft: I

CPX misc.dcb_soft: I
LCPX dcub_soft: I

VC[0] softerr: I
CPX misc.vca_soft: I
LCPX vpca_soft: I

CPX misc.vca_soft: I
LCPX vpca_soft: I

DC[0] soft_err: I
CPX misc.dca_soft: I
LCPX dcua_soft: I

CPX misc.dca_soft: I
LCPX dcua_soft: I

VC[0] softerr: I
LCUE vca_softerr: I

LCUE vca_softerr: I

VC[1] softerr: I
LCUE vcb_softerr: I

LCUE vcb_softerr: I

VC[2] softerr: I
LCUE vcc_softerr: I

LCUE vcc_softerr: I

VC[3] softerr: I
LCUE vcd_softerr: I

LCUE vcd_softerr: I
```

where:

I is the current value of the specified scan ring field, in hex

Subtest 161 Errors

Any of the following multi-line error messages could result from an error in subtest 161.

One of the following lines:

```
soft error clear when expected set
soft error set after attempting clear
```

Followed by one or more of the following lines:

```
CPX misc.rmserr: I
CUE ae0_mod_data_out: I
CUE ae0_par_data_out: I
CUE ae1_mod_data_out: I
CUE ae1_par_data_out: I
CUE ao0_mod_data_out: I
CUE ao0_par_data_out: I
CUE ao1_mod_data_out: I
CUE ao1_par_data_out: I
CUE be0_mod_data_out: I
CUE be0_par_data_out: I
CUE be1_mod_data_out: I
CUE be1_par_data_out: I
CUE bo0_mod_data_out: I
CUE bo0_par_data_out: I
CUE bo1_mod_data_out: I
CUE bo1_par_data_out: I
CUE ce0_mod_data_out: I
CUE ce0_par_data_out: I
CUE ce1_mod_data_out: I
CUE ce1_par_data_out: I
CUE co0_mod_data_out: I
CUE co0_par_data_out: I
CUE co1_mod_data_out: I
CUE co1_par_data_out: I
CUE de0_mod_data_out: I
CUE de0_par_data_out: I
CUE de1_mod_data_out: I
CUE de1_par_data_out: I
CUE do0_mod_data_out: I
CUE do0_par_data_out: I
CUE do1_mod_data_out: I
CUE do1_par_data_out: I
```

where:

I was the initial value of the specified scan ring field, in hex

Subtest 165 Errors

Any of the following multi-line error messages could result from an error in subtest 165.

One of the following lines:

```
hard error clear when expected set
hard error set when expected clear
hard error set after attempting clear
```

Followed by either of the following groups of lines:

```
CPX misc.mx_uadr: 0xIII
CPX misc.pcmstart: I
CPX misc.tistart: I
CPX misc.rmstart: I

CUE xbar_addr_28_17: 0xIII
CUE pcm_start_a: I
CUE ti_start0: I
CUE rm_start0: I
```

where:

I is the current value of the specified scan ring field, in hex

Subtest 170 Errors

Any of the following error messages could result from an error in subtest 170:

```
hard error set when expected clear
```

or:

```
hard error clear when expected set
```

or:

```
hard error set after attempting clear
```

Subtest 175 Errors

Any of the following two line error messages could result from an error in subtest 175.

One of the following lines:

```
hard error clear when expected set
hard error set when expected clear
hard error set after attempting clear
```

Followed by either of the following lines:

```
CPX initial pcm.mx_ad: 0xI   initial pcm.req2: 0xI
CUE initial pcm_xbar_addr_a<17..14>: 0xI
```

where:

I is the current value of the specified scan ring field, in hex

Subtest 180 Errors

Any of the following error messages could result from an error in subtest 180:

```
        ioctl allocate call returned error
```
or:
```
        ioctl free call returned error
```
or:
```
        did not receive expected signal - address:  0xIIIIIIII
```
or:
```
        received unexpected signal - address:  0xIIIIIIII
```
where:

IIIIIIII is the system I/O space address that failed, in hex

Subtest 181 Errors

Any of the following error messages could result from an error in subtest 180:

```
        ioctl allocate call returned error
```
or:
```
        ioctl free call returned error
```
or:
```
        did not receive expected signal - address:  0xIIIIIIII
```
or:
```
        received unexpected signal - address:  0xIIIIIIII
```
or:
```
        trapped data does not match written data
        actual data: 0xJJJJ   trapped data: 0xKKKK
```
or:
```
        trapped address (masked) does not match address written to
        actual address:    (masked): 0xLLLLLLLL   (unmasked): 0xIIIIIIII
        trapped address:   (masked): 0xMMMMMMMM
```
where:

IIIIIIII is the system I/O space address that failed, in hex
JJJJ is the data written to an illegal address, in hex
KKKK is the data trapped in the scan ring, in hex
LLLLLLLL is the masked system I/O space address that failed, in hex
MMMMMMMM is the masked system I/O space address trapped in the scan ring, in hex

Subtest 185 Errors

Any of the following one line error messages could result from an error in subtest 185:

```
could not read first pcm location
```

or:

```
could not write first pcm location
```

or:

```
could not restore first pcm location
```

In addition to the above, subtest 185 has several two-line error messages:

One of the following lines:

```
hard error clear when expected set
hard error set after attempting clear
```

Followed by any one the following lines:

```
CPX bank: A Even   address: 0x00000000
CUE a_me_req: I   a_me_addr: 0xI
CUE b_me_req: I   b_me_addr: 0xI
CUE c_me_req: I   c_me_addr: 0xI
CUE d_me_req: I   d_me_addr: 0xI
CUE a_mo_req: I   a_mo_addr: 0xI
CUE b_mo_req: I   b_mo_addr: 0xI
CUE c_mo_req: I   c_mo_addr: 0xI
CUE d_mo_req: I   d_mo_addr: 0xI
```

where:

I is the current value of the specified scan ring field, in hex

Subtest 195 Errors

Any of the following multi-line error messages could result from an error in subtest 195.

One of the following lines:

```
hard error clear when expected set
hard error set when expected clear
hard error set after attempting clear
```

Followed by either the following line or group of lines:

```
CPX creg.datparbad: 0xI   creg.lkparbad: 0xI

initial CUO rslt_par:     0xJJ
            r_tc_req_ok:  J
            tc_rslt_u:    J
            tc_rslt_l:    J
            comu_rd:      J
            coml_rd:      J
            cmod_par_out: J
            r_cmod_oe:    J
            r_ram_cs:     J
            lock_out:     J
            lock_par:     J
```

where:

I is the current value of the specified scan ring field, in hex
J was the initial value of the specified scan ring field, in hex

Subtest 200 Errors

Any of the following two line error messages could result from an error in subtest 200.

One of the following lines:

```
hard error clear when expected set
hard error set when expected clear
hard error set after attempting clear
```

Followed by either of the following lines:

```
CPX arb.is_bs[0] (select): I
CUE arb.is_bs[0] (select): I
```

where:

I was the bank initially selected, in hex

Subtest 205 Errors

Any of the following error messages could result from an error in subtest 205:

```
        pattern test unable to open windows
        <line>
or:
        pattern test unable to allocate windows
        <line>
or:
        pattern test unable to free windows
        <line>
or:
        hard error occurred during pattern test
        <line>
        failing address: 0xIIIIIIII   data mask: 0xJJJJ
        actual: 0xKKKK   expected: 0xLLLL
or:
        soft error occurred during pattern test
        <line>
        failing address: 0xIIIIIIII   data mask: 0xJJJJ
        actual: 0xKKKK   expected: 0xLLLL
or:
        PCM pattern test failed
        <line>
        failing address: 0xIIIIIIII   data mask: 0xJJJJ
        actual: 0xKKKK   expected: 0xLLLL
```

where:

<line> is one of:

```
        CPX PCM channel enabled: A
        CPX PCM channel enabled: B
        CUE PCM channel enabled: A even
        CUE PCM channel enabled: B even
        CUE PCM channel enabled: C even
        CUE PCM channel enabled: D even
        CUO PCM channel enabled: A odd
        CUO PCM channel enabled: B odd
        CUO PCM channel enabled: C odd
        CUO PCM channel enabled: D odd
```

IIIIIIII is the PCM address which failed, in hex
JJJJ is the mask for the actual and expected values, in hex
KKKK is the actual contents of that address, in hex
LLLL is the expected contents of that address, in hex

Subtest 206 Errors

Any of the following error messages could result from an error in subtest 205:

```
        mm_iowr call failed initial clear
        <line>
or:
        mm_iord call failed when checking initial clear
        <line>
or:
        mm_iowr call failed when setting PCM location at address 0xIIIIIIII
        <line>
or:
        mm_iord call failed when reading entire PCM
        <line>
or:
        mm_iowr call failed when clearing PCM location at address 0xIIIIIIII
        <line>
or:
        CPX PCM did not clear at address 0xIIIIIIII
        <line>
or:
        CUE / CUO PCM did not clear at address 0xIIIIIIII
        <line>
or:
        CPX PCM incorrectly clear at address 0xJJJJJJJJ
        address set: 0xIIIIIIII
        <line>
or:
        CUE / CUO PCM incorrectly clear at address 0xJJJJJJJJ
        address set: 0xIIIIIIII
        <line>
or:
        CPX PCM incorrectly set at address 0xJJJJJJJJ
        address set: 0xIIIIIIII
        <line>
or:
        CUE / CUO PCM incorrectly set at address 0xJJJJJJJJ
        address set: 0xIIIIIIII
        <line>
```

where:

`<line>` is one of:

```
CPX PCM channel enabled: A
CPX PCM channel enabled: B
CUE PCM channel enabled: A even
CUE PCM channel enabled: B even
CUE PCM channel enabled: C even
CUE PCM channel enabled: D even
CUO PCM channel enabled: A odd
CUO PCM channel enabled: B odd
CUO PCM channel enabled: C odd
CUO PCM channel enabled: D odd
```

`IIIIIIII` is the PCM address address under test, in hex

`JJJJJJJJ` is the PCM address which failed, in hex

Subtest 210, 211, 212, and 213 Errors

The following error messages could result from an error in any of these subtests:

```
        unable to open windows
or:
        unable to allocate a window
or:
        unable to free window
```

In addition to the common messages above, the following error messages could result from an error in subtest 210:

```
        CPX R & M bit not as expected
        address: 0xJJJJJJJJ   actual: 0xKKKK   expected: 0xLLLL
or:
        CUE / CUO R & M bit not as expected
        address: 0xJJJJJJJJ   actual: 0xKKKK   expected: 0xLLLL
or:
        hard error present after writing pattern
or:
        hard error present before writing pattern
or:
        soft error present after writing pattern
or:
        soft error present before writing pattern
or:
        hard error present after write of address: 0xJJJJJJJJ
or:
        hard error present after read of address: 0xJJJJJJJJ
or:
        soft error present after write of address: 0xJJJJJJJJ
or:
        soft error present after read of address: 0xJJJJJJJJ
```

In addition to the common messages above, the following error messages could result from an error in subtest 211:

```
        CPX R & M bit error
        address incorrectly set: 0xJJJJJJJJ   data: 0xKKKK
        address initially set:   0xMMMMMMMM   data: 0xNNNN
or:
        CPX R & M bit error
        address incorrectly clear: 0xJJJJJJJJ   data: 0xKKKK
        address initially set:     0xMMMMMMMM   data: 0xNNNN
or:
        CUE / CUO R & M bit error
        address incorrectly set: 0xJJJJJJJJ   data: 0xKKKK
        address initially set:   0xMMMMMMMM   data: 0xNNNN
or:
        CUE / CUO R & M bit error
        address incorrectly clear: 0xJJJJJJJJ   data: 0xKKKK
        address initially set:     0xMMMMMMMM   data: 0xNNNN
```

In addition to the common messages above, the following error messages could result from an error in subtest 212:

```
        CPX R & M bit not as expected
        bank: IIII   address: 0xJJJJJJJJ   actual: 0xKKKK   expected: 0xLLLL
or:
        CUE / CUO R & M bit not as expected
```

```
bank: IIII   address: 0xJJJJJJJJ   actual: 0xKKKK   expected: 0xLLLL
```

In addition to the common messages above, the following error messages could result from an error in subtest 213:

```
CPX R & M bit error   bank: IIII
address incorrectly set: 0xJJJJJJJJ   data: 0xKKKK
address initially set:   0xMMMMMMMM   data: 0xNNNN
```

or:

```
CPX R & M bit error   bank: IIII
address incorrectly clear: 0xJJJJJJJJ   data: 0xKKKK
address initially set:     0xMMMMMMMM   data: 0xNNNN
```

or:

```
CUE / CUO R & M bit error   bank: IIII
address incorrectly set: 0xJJJJJJJJ   data: 0xKKKK
address initially set:   0xMMMMMMMM   data: 0xNNNN
```

or:

```
CUE / CUO R & M bit error   bank: IIII
address incorrectly clear: 0xJJJJJJJJ   data: 0xKKKK
address initially set:     0xMMMMMMMM   data: 0xNNNN
```

where:

IIII represents one of the following strings which denotes the individual bank enabled, if not all banks enabled:

```
A even 0
A even 1
A odd 0
A odd 1
B even 0
B even 1
B odd 0
B odd 1
C even 0
C even 1
C odd 0
C odd 1
D even 0
D even 1
D odd 0
D odd 1
```

JJJJJJJJ is the failing address, in hex
KKKK is the data read from the failing address, in hex
LLLL is the expected contents of the failing address, in hex
MMMMMMMM is the address set, in hex
NNNN is the data written to the address set, in hex

Subtest 215 Errors

Any of the following error messages could result from an error in subtest 215:

```
        comm_reg returned error when clearing all lock bits
        register: OxIII   return value: J
or:
        comm_reg returned error when checking all lock bits clear
        register: OxIII   return value: J
or:
        initial clear of a lock bit failed - register: OxIII
or:
        comm_reg returned error when setting a lock bit
        register: OxIII   return value: J
or:
        comm_reg returned error when checking a lock bit
        register: OxIII   return value: J
or:
        lock bit clear when expected set - register: OxIII
or:
        lock bit clear when expected set - register: OxIII
        register initially set: OxKKK
or:
        comm_reg returned error when clearing lock bit previously set
        register: OxIII   return value: J
or:
        comm_reg returned error when writing data
        register: OxIII   return value: J
or:
        comm_reg returned error when checking data
        register: OxIII   return value: J
or:
        communication register pattern test error
        register: OxIII   data parity: OxLL
        upper data:   actual: OxMMMMMMMM   expected: OxNNNNNNNN
        lower data:   actual: OxWWWWWWWW   expected: OxXXXXXXXX
```

where:

III is the ram address of the failing communication register, in hex
J is the error code, in decimal
KKK is the address of the communication register written to, in hex
LL is the data parity read from the register, in hex
MMMMMMMM is the upper actual data read from the register, in hex
NNNNNNNN is the upper expected data read from the register, in hex
WWWWWWWW is the lower actual data read from the register, in hex
XXXXXXXX is the lower expected data read from the register, in hex

Subtest 217 Errors

The following error messages could result from an error in subtest 217:

```
        comm_reg returned error while initially clearing modified bits
        register: OxIII   return value: J
or:
        comm_reg returned error while initially checking modified bits clear
        register: OxIII   return value: J
or:
        initial clear of modified bit associated with register OxIII failed
or:
        comm_reg returned error while setting a modified bit
        register: OxIII   return value: J
or:
        comm_reg returned error while checking a modified bit
        register: OxIII   return value: J
or:
        modified bit associated with register OxIII clear when expected set
or:
        modified bit associated with register OxIII set when expected clear
        register set: OxKKK
or:
        comm_reg returned error while reclearing a modified bit
        register: OxIII   return value: J
or:
        comm_reg returned error while resetting a register
        register: OxIII   return value: J   loop count: LL
or:
        comm_reg returned error when initializing a modified bit
        register: OxIII   return value: J   loop count: LL
or:
        comm_reg returned error when checking reset
        register: OxIII   return value: J   loop count: LL
or:
        state before operation not as expected
        upper data:   actual: OxWWWWWWWW   expected: OxXXXXXXXX
        lower data:   actual: OxYYYYYYYY   expected: OxZZZZZZZZ
        actual data parity:   OxPP   lock data:    actual: M   expected: N
        register: OxIII   loop count: LL   operation: HHHH
or:
        comm_reg returned error when performing operation
        register: OxIII   return value: J   loop count: LL
or:
        comm_reg returned error when checking a modified bit
        register: OxIII   return value: J   loop count: LL
or:
        modified bit incorrectly clear after performing operation
        failing register: OxIII   test register: OxKKK
        loop count: LL   operation: HHHH
or:
        modified bit incorrectly set after performing operation
        failing register: OxIII   test register: OxKKK
        loop count: LL   operation: HHHH
or:
        comm_reg returned error when reading data
        register: OxIII   return value: J   loop count: LL
or:
```

```
state after operation not as expected
upper data:   actual: 0xWWWWWWWW   expected: 0xXXXXXXXX
lower data:   actual: 0xYYYYYYYY   expected: 0xZZZZZZZZ
actual data parity:   0xPP   lock data:   actual: M   expected: N
register: 0xIII   loop count: LL   operation: HHHH
```

where:

HHHH is the name of the operation which failed
III is the ram address of the failing register, in hex
J is the failing routines error code, in decimal
KKK is the address of the register previously accessed, in hex
LL is the number that corresponds to the index in the following table that shows what communication register operation failed, in decimal
M is the actual lock bit, in hex
N is the expected lock bit, in hex
PPPP is the actual data parity, in hex
WWWWWWWW is the actual upper data, in hex
XXXXXXXX is the expected upper data, in hex
YYYYYYYY is the actual lower data, in hex
ZZZZZZZZ is the expected lower data, in hex

Table cpx4000–3, Subtest 217 - Communication Register Operations

OPERATION	INDEXES
test	0-1
lock	2-5
unlock	6-9
put long	A-F
get long	10-15
send long	16-1B
receive long	1C-21
put lower	22-27
get lower	28-2D
send lower	2E-33
receive lower	34-39
put upper	3A-3F
get upper	40-45
send upper	46-4B
receive upper	4C-51
write modified	52-53
read modified	54-55
restore	46-5B

Subtest 220 Errors

Any of the following error messages could result from an error in subtest 220:

```
        pattern test unable to open windows
```
or:
```
        pattern test unable to allocate windows
```
or:
```
        pattern test unable to free windows
```
or:
```
        hard error occurred during pattern test
        failing address: 0xIIIIIIII   data mask: 0xJJJJ
        actual: 0xKKKK   expected: 0xLLLL
```
or:
```
        soft error occurred during pattern test
        failing address: 0xIIIIIIII   data mask: 0xJJJJ
        actual: 0xKKKK   expected: 0xLLLL
```
or:
```
        TOC register pattern test failed
        failing address: 0xIIIIIIII   data mask: 0xJJJJ
        actual: 0xKKKK   expected: 0xLLLL
```
where:

IIIIIIII is the address which failed, in hex
JJJJ is the data mask, in hex
KKKK is the data read from the address, in hex
LLLL is the expected data, in hex

or:
```
        TOC counting while off
        test: 0xM (N)   bytes I & J: 0xKKKK (LLLL)
```
or:
```
        TOC not counting while on
        test: 0xM (N)
```
or:
```
        TOC counting while off
        test: 0xM (N)
```
or:
```
        TOC holding lock failure - counting during multiple reads
        bytes I & J: 0xKKKK (LLLL)
```
or:
```
        TOC holding lock failure - not counting after reads complete
        test: 0xM (N)
```
where:

I & J are the bytes under test
KKKK is the non zero value, in hex
LLLL is the non zero value, in decimal
M is the test number, in hex
N is the test number, in decimal

or:
```
        TOC counting test failure
        bytes: I & J   loop: K   minimum: 0xLLLL   maximum: 0xMMMM   actual: 0xNNNN
```
where:

I & J are the bytes under test
K is the loop number, in decimal
LLLL is the minimum allowed value, in hex
MMMM is the maximum allowed value, in hex
NNNN is the actual value, in hex

Subtest 225 Errors

Any of the following error messages could result from an error in subtest 225:

```
        pattern test unable to open windows
or:
        pattern test unable to allocate windows
or:
        pattern test unable to free windows
or:
        hard error occurred during pattern test
        failing address: 0xIIIIIIII   data mask: 0xJJJJ
        actual: 0xKKKK   expected: 0xLLLL
or:
        soft error occurred during pattern test
        failing address: 0xIIIIIIII   data mask: 0xJJJJ
        actual: 0xKKKK   expected: 0xLLLL
or:
        IT register pattern test failed
        failing address: 0xIIIIIIII   data mask: 0xJJJJ
        actual: 0xKKKK   expected: 0xLLLL
```

where:

IIIIIIII is the address which failed, in hex
JJJJ is the data mask, in hex
KKKK is the data read from the address, in hex
LLLL is the expected data, in hex

```
or:
        IT counting while off
        test: 0xM (N)   ITC: 0xIIII (JJJJ)
or:
        IT counting while off
        test: 0xM (N)
or:
        IT not counting while on
        test: 0xM (N)
or:
        IT interrupt not received
        test: 0xM (N)
or:
        SIB interrupt will not clear - resetting system
        test: 0xM (N)
or:
        FULL bit clear when should be set - single interrupt, 1st read of ITSR
or:
        OVF bit set when should be clear - single interrupt, 1st read of ITSR
or:
        FULL bit set when should be clear - single interrupt, 2nd read of ITSR
or:
        OVF bit set when should be clear - single interrupt, 2nd read of ITSR
or:
        FULL bit clear when should be set - multiple interrupt, 1st read of ITSR
or:
        OVF bit clear when should be set - multiple interrupt, 1st read of ITSR
or:
        FULL bit set when should be clear - multiple interrupt, 2nd read of ITSR
```

or:

```
OVF bit set when should be clear - multiple interrupt, 2nd read of ITSR
```

or:

```
NITC value changed
NITC: 0xIIII (JJJJ)
```

or:

```
ITIN value changed
ITIN: 0xIIII (JJJJ)
```

or:

```
IT interrupt not received - single interrupt
test: 0xM (N)
```

where:

IIII is the non zero value from the specified register, in hex
JJJJ is the non zero value from the specified register, in decimal
M is the test number, in hex
N is the test number, in decimal

SPU UNIX Error Messages

For a common list of possible error messages resulting from SPU UNIX, refer to the appendix on SPU UNIX Error Messages.

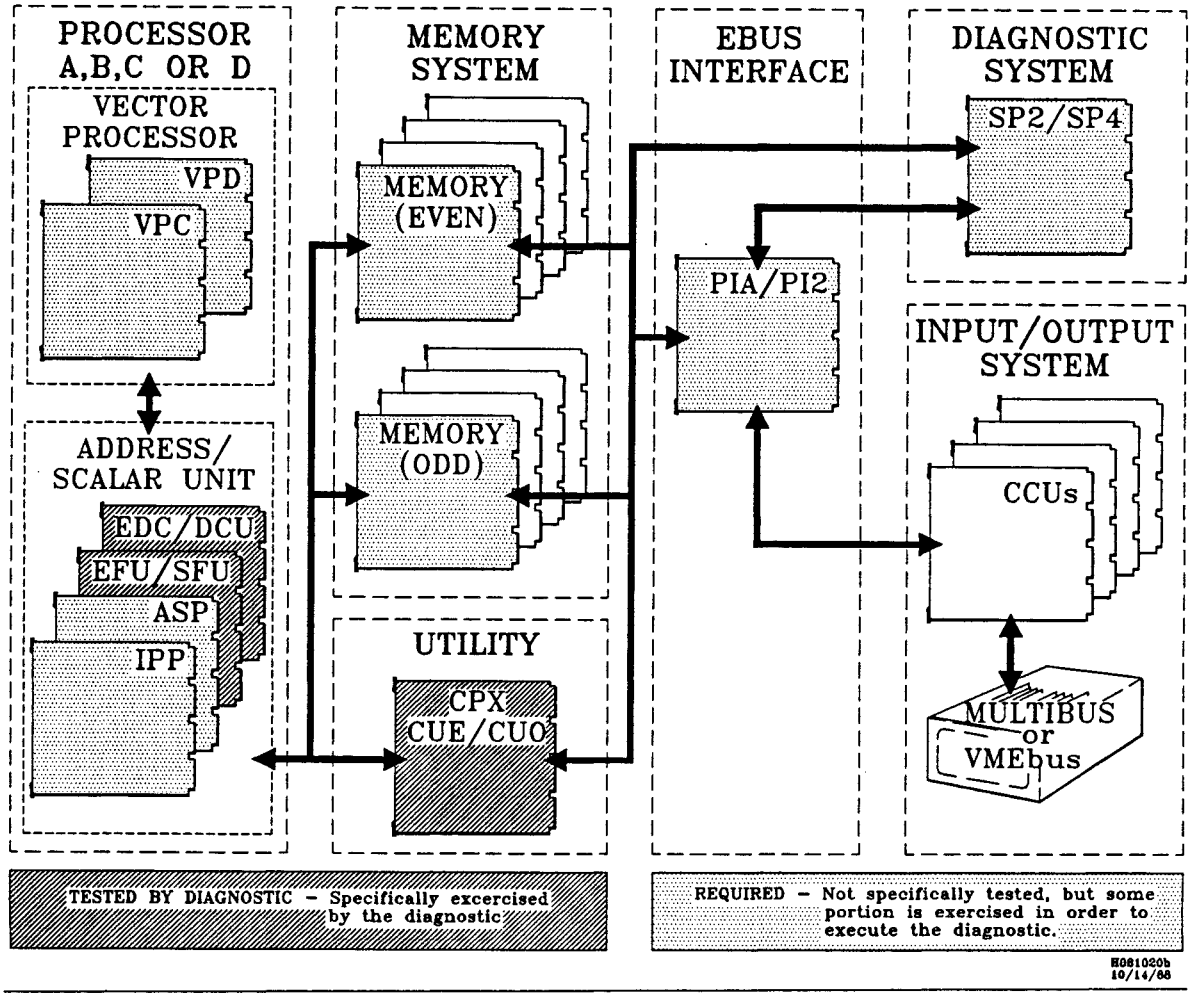
cpu4010

Referenced and Modified Bits

Overview

The *cpu4010* subtests are designed to exercise the Referenced and Modified (R&M) bits to verify their proper operation. The R&M bits are located on the CPU Utility Card(s) (CPX or CUE/CUO). This verification includes R&M bit uniqueness, proper memory page assignment, odd/even page encaching, and access using either byte or halfword accesses. A second function of the *cpu4010* subtests is to verify that the Physical Configuration Map (PCM) bits are unique and that they can be set and cleared accordingly. The following figure shows an overall view of what part of the system is being tested and which field replaceable units are required for the test to execute.

Figure cpu4010–1, Functional Areas Tested by *cpu4010*



Prerequisites and Required Equipment

In order to run the *cpu4010* test, the boards listed in the following table must be operational. The table shows the tests used to verify the required boards. No additional equipment is required to run this test.

Table cpu4010–1, Required Functional Boards

BOARD	TEST TO VERIFY
Service Processor (SP2 or SP4)	*spu1000, spu4000*
PBUS Interface Adapter (PIA or PI2)	*pia4000* or *pi2_4000*
Memory System	*mem4000*
Instruction Processor Unit (IPP)	*cpu4030*
Address Scalar Processor (ASP)	*cpu4030*
Scalar Function Unit (SFU or EFU)	*cpu4030*
Data Cache Unit (DCU or EDC)	*cpu4030*
CPU Utility Board(s) (CPX or CUE/CUO)	*cpx4000*
Vector Processor Control (VPC)	*cpu4041*
Vector Processor Data (VPD)	*cpu4041*

NOTE

Memory System consists of a minimum of one pair of memory boards (one even and one odd).

Test Invocation

To invoke the *cpu4010* test, use the procedure shown in the following figure. All responses in **boldface** are entered by the user. The prompts and responses in the following figure would appear sequentially on the screen, one line at a time. All prompts and responses are shown in one figure for convenience.

CAUTION

The invocation sequence shown in the following figure is the typical invocation sequence. However, the *initall* utility must be executed in some cases. If the system has just been powered up, if *mem4000* was executed with failures, or if *spu4000* was executed, then *initall* must be executed prior to any test execution. Failure to execute *initall* in these circumstances could result in invalid test results.

NOTE

Running the *initall* utility requires two to three minutes to execute depending on if the control stores have been previously loaded. However, it is suggested that *initall* be executed in the event that the state of the system is unknown.

Figure cpu4010–2, Test Invocation Sequence

```
(spu)>  cd /mnt/test
(spu)>  sysreset -l2
(spu)>  dshell

CONVEX DIAGNOSTIC SHELL

: test cpu4010 [-c [class numeral(s)] ] [-s [subtest numeral(s)]] [+> filename]
```

NOTE

After entering **dshell**, specific changes may be made to the *dshell* parameters. Please refer to the "Dshell and Iscan Overview" chapter of this manual for more information on *dshell*.

Entering only **test cpu4010** executes all subtests sequentially. The user can execute a specific class(es) of subtest(s) or one or more individual subtests by using the *-c* or *-s* options, respectively. Detailed information for using these options can be found in the "Dshell and Iscan Overview" chapter of this manual. The [+>*filename*] option allows the test results to be appended to *filename*.

Typical Test Sequence

Under normal conditions, the first six classes of subtests provide sufficient test results to verify the proper operations of the Referenced and Modified (R&M) bits. Classes 7 through 10 test the operation of the R&M bits more extensively. Therefore, classes 7 through 10 require a significant amount of time to execute (55 minutes per 32 Megabytes of memory), and should only be exercised when the first six classes of tests produce an error. The following invocation sequence is recommended for most cases.

test cpu4010 -s 1-1000 +> *filename*

The following table represent the time that each subtest requires to execute:

Table cpu4010–2, Subtest Execution Times

CLASS OF SUBTEST	TIME TO EXECUTE PER SUBTEST
1 - 5	5 sec. + 10 sec. per 128 Megabytes of memory
6	25 seconds
7	25 minutes per 32 Megabytes of memory
8	10 minutes per 32 Megabytes of memory
9	15 minutes per 32 Megabytes of memory
10	5 minutes per 32 Megabytes of memory

Test Parameter Menu

Once the test is invoked, a test parameter menu prompts for selection of default switches. If the test is run with all defaults invoked (user answers **y** to the first prompt), no other prompts are provided. If the user answers **n** to the first prompt (run test without default switches), then a series of prompts are presented. The following figure shows all possible prompts, their possible answers (in brackets [ ]), and their default answers (in parentheses ( )). The prompts and responses in the following figure would appear sequentially on the screen, one line at a time. All the prompts and responses are shown in one figure for convenience.

Figure cpu4010–3, Test Parameter Menu

```
Test 'cpu4010.t'                                    Thu Nov 19 06:18:15 1987

                              ENTER TEST PARAMETERS

                    [ ]    Encloses allowed input ranges or values
                    ( )    Encloses the default value
                    ^       Returns to the previous prompt
                    :nn     Returns to the prompt # nn
                    :       Returns to the first unsatisfied prompt
                    :?      Reviews previous entries

1:  Run default switches? [y,n]                                  (y) ->
2:  CPU to use in single head test? [ABCD]                       (X) ->
3:  CPUs to use in multiple head test? [ABCD]                    (ABCD) ->
4:  Sequential Execution? [y,n]                                  (n) ->
5:  Timeout Scale Factor Enabled? [1-100]                        (1) ->
6:  Dcache Enabled? [y,n]                                        (y) ->
```

NOTE

In prompts 2 and 3 in the previous figure, ABCD represents all available CPUs. Only the available CPUs within the machine under test will be displayed as possible choices. The variable **X** represents the first available CPU.

Prompt Explanations

A description of the meaning of each prompt follows:

```
Run default switches? [y/n]                                      (y) ->
```

A response of **y** or <**CR**> causes no additional test parameter prompts to be displayed and testing begins. However, if a negative response is supplied, additional test parameter prompts are displayed allowing modification of the default selections.

The following prompts are only displayed and answered if the first prompt is answered with **n**:

```
CPU to use in single head test? [ABCD]                           (X) ->
```

This prompt allows selection of the CPU to be used in the single head test. The possible selections, represented by ABCD, will consist of all available CPUs. The default, represented by X, will be the first CPU in the available list of CPUs.

```
CPUs to use in multiple head test? [ABCD]                        (ABCD) ->
```

This prompt allows selection of the CPU(s) to be used in the multiple head test. The possible selections, represented by ABCD, will consist of all available CPUs. The default for this prompt is

all available CPUs.

```
Sequential Execution? [y,n]                                    (n) ->
```

If set by entering **y**, the sequential bit in the PSW will be set to forced sequential execution mode.

```
Timeout Scale Factor [1-100]                                   (1) ->
```

This prompt allows adjustment of the timeout factor. The normal timeout factor is multiplied by the number selected to increase the timeout factor. For example, if **5** is entered, the normal timeout factor is multiplied by 5 and it will take the test five times as long to timeout. This factor should be used when sequential execution or disabling the Dcache is selected.

```
Dcache Enabled? [y,n]                                          (y) ->
```

The Dcache is normally enabled; however, if it is suspected broken, it can be disabled by entering **n** at this prompt.

When all prompts have been answered, the screen displays a test parameter summary which echos the prompts that have been answered. The following figure illustrates an example of a "Test Parameter Summary" screen. The actual values and responses vary according to the input.

Figure cpu4010–4, Sample Test Parameter Summary

```
                    Test Parameter Summary

Run default switches?                                  :  y
CPU to use in single head test?                        :  ab
CPUs to use in multiple head test?                     :  ab
Sequential Execution?                                  :  n
Timeout Scale Factor Enabled?                          :  1
Dcache Enabled?                                        :  y
```

Hardware Initialization Sequence

After the last prompt is entered (and prior to test code execution), the following events occur:

- The maximum and minimum physical memory installed in the system is determined from the Physical Configuration Map (PCM) on the Service Processor. This process generates an array which indicates which 2 megabyte blocks of memory are present (indicated by a 1 in the PCM) or are not present (indicated by a 0 in the PCM).
- The amount of memory which the source code will use is calculated.
- Page Table Entries for memory are calculated. Memory spans from address zero through the maximum installed address.

- Page Table Entries for I/O space are calculated.
- The Page Table Entries for I/O space and the test source code are mapped into memory. This mapping is determined by what test is to be executed. If the test requires the source code to reside in low memory, then the Page Table Entries for I/O space and memory are installed into low memory; otherwise the Page Table Entries are installed into high memory.
- Each test module is loaded into upper or lower memory based on which test is being run.
- A communication block is generated which will be used by the test code to determine what test to execute and how it is to be executed.
- The system is initialized (the memory system, CPX or CUE/CUO, PIA or PI2, and the CPU boards are reset).
- All CPU's scratch RAM is loaded with various values to control the execution of the test.
- For the first CPU selected, the initial Program Counter (PC) is loaded into register T0 and the initial Processor Status Word (PSW) is loaded into register T2.
- For the first CPU selected, control store is initialized to the cold-start location.
- For all CPUs except the first one selected, control store in initialized to the cold-start location.
- Clocks are turned on.

After all of the above events have occurred, the test code is started.

Class Descriptions

There are ten different classes of subtests for *cpu4010*. The following sections describe the different classes and each of their subtests. Each section contains a table listing each subtest in that class, a description of the subtest, the subtests executable test code (object module), and the subtest's source file (source code with comments).

Class 1 Subtests

Class 1 subtests verify the page boundary operations for the Referenced and Modified bits by the following procedure:

- An odd page is touched to encache its address into the Address Translation Unit (ATU).
- The Referenced and Modified bits for that page are then cleared and the last four bytes for the even page of the even/odd page pairs are written.
- The Referenced and Modified bits are then verified to ensure that only the even page Referenced and Modified bits set.

The following table describes each Class 1 subtest, its object module, and its source file. All subtests terminate with a halt command and store a halt code in address register A1. If the code stored in A1 is 0x100, then the subtest passed. Any other code in A1 indicates that the subtest failed.

> **NOTE**
>
> R&M refers to Referenced and Modified bits.

Table cpu4010–3, Class 1 Subtests

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	TIME
10	R&M boundary test with code in low memory	*cpu4010.rnn*	*rmbits.c*	Note 1
20	R&M boundary test with code in upper memory	*cpu4010.rnn*	*rmbits.c*	Note 1

NOTE 1: 5 sec. + 10 sec. per 128 Megabytes of memory

Class 2 Subtests

Class 2 subtests execute the following subtest procedures. All Class 2 subtests execute with the test code residing in low memory and are based on byte accesses to I/O space.

- All Referenced and Modified bits associated with accessible physical memory are cleared.
- The following events are repeated in a loop until all pages are operated on:
  - One Referenced and Modified bit is read from I/O space to verify that it is clear.
  - A load, store, or execute operation is performed on the first address of a page of accessible physical memory. This event should set the Referenced and Modified bits according to the instruction performed.
  - The Referenced and Modified bits for the page operated on are verified to ensure that they were set according to the operation performed.
  - The Referenced and Modified bits for the page operated on are cleared.
- After all pages are tested, the test verifies that all Referenced and Modified bits (R&M bits) were cleared. The following table describes each Class 2 subtest, its object module, and its source file.

All subtests terminate with a halt command and store a halt code in address register A1. If the code stored in A1 is 0x100, then the subtest passed. Any other code in A1 indicates that the subtest failed.

> **NOTE**
>
> R&M refers to Referenced and Modified bits.

Table cpu4010–4, Class 2 Subtests

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	TIME
110	R&M load test, even bank march, code in low memory, byte accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1
111	R&M load test, odd bank march, code in low memory, byte accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1
120	R&M store test, even bank march, code in low memory, byte accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1
121	R&M store test, odd bank march, code in low memory, byte accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1
130	R&M execute test, even bank march, code in low memory, byte accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1
131	R&M execute test, odd bank march, code in low memory, byte accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1

NOTE 1: 5 sec. + 10 sec. per 128 Megabytes of memory

Class 3 Subtests

Class 3 subtests execute the following subtest procedures. All Class 3 subtests execute with the test code residing in upper memory and are based on byte accesses to I/O space.

- All Referenced and Modified bits (R&M bits) associated with accessible physical memory are cleared.
- The following events are repeated in a loop until all pages are operated on:
  - One Referenced and Modified bit is read from I/O space to verify that it is clear.
  - A load, store, or execute operation is performed on the first address of a page of accessible physical memory. This event should set the Referenced and Modified bits according to the instruction performed.
  - The Referenced and Modified bits for the page operated on are verified to ensure that they were set according to the operation performed.
  - The Referenced and Modified bits for the page operated on are cleared.
- After all pages are tested, the test verifies that all Referenced and Modified bits were cleared. The following table describes each Class 3 subtest, its object module, and its source file.

All subtests terminate with a halt command and store a halt code in address register A1. If the code stored in A1 is 0x100, then the subtest passed. Any other code in A1 indicates that the subtest failed.

> **NOTE**
>
> R&M refers to Referenced and Modified bits.

Table cpu4010–5, Class 3 Subtests

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	TIME
210	R&M load test, even bank march, code in upper memory, byte accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1
211	R&M load test, odd bank march, code in upper memory, byte accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1
220	R&M store test, even bank march, code in upper memory, byte accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1
221	R&M store test, odd bank march, code in upper memory, byte accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1
230	R&M execute test, even bank march, code in upper memory, byte accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1
231	R&M execute test, odd bank march, code in upper memory, byte accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1

NOTE 1: 5 sec. + 10 sec. per 128 Megabytes of memory

Class 4 Subtests

Class 4 subtests execute the following subtest procedures. All Class 4 subtests execute with the test code residing in low memory and are based on halfword accesses to I/O space.

- All Referenced and Modified bits (R&M bits) associated with accessible physical memory are cleared.
- The following events are repeated in a loop until all pages are operated on:
  - One Referenced and Modified bit is read from I/O space to verify that it is clear.
  - A load, store, or execute operation is performed on the first address of a page of accessible physical memory. This event should set the Referenced and Modified bits according to the instruction performed.
  - The Referenced and Modified bits for the page operated on are verified to ensure that they were set according to the operation performed.
  - The Referenced and Modified bits for the page operated on are cleared.
- After all pages are tested, the test verifies that all Referenced and Modified bits were cleared. The following table describes each Class 4 subtest, its object module, and its source file.

All subtests terminate with a halt command and store a halt code in address register A1. If the code stored in A1 is 0x100, then the subtest passed. Any other code in A1 indicates that the subtest failed.

> **NOTE**
>
> R&M refers to Referenced and Modified bits.

Table cpu4010–6, Class 4 Subtests

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	TIME
310	R&M load test, even bank march, code in low memory, halfword accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1
311	R&M load test, odd bank march, code in low memory, halfword accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1
320	R&M store test, even bank march, code in low memory, halfword accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1
321	R&M store test, odd bank march, code in low memory, halfword accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1
330	R&M execute test, even bank march, code in low memory, halfword accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1
331	R&M execute test, odd bank march, code in low memory, halfword accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1

NOTE 1: 5 sec. + 10 sec. per 128 Megabytes of memory

Class 5 Subtests

Class 5 subtests execute the following subtest procedures. All Class 5 subtests execute with the test code residing in upper memory and are based on halfword accesses to I/O space.

- All Referenced and Modified bits (R&M bits) for the associated with accessible physical memory are cleared.
- The following events are repeated in a loop until all pages are operated on:
  - One Referenced and Modified bit is read from I/O space to verify that it is clear.
  - A load, store, or execute operation is performed on the first address of a page of accessible physical memory. This event should set the Referenced and Modified bits according to the instruction performed.
  - The Referenced and Modified bits for the page operated on are verified to ensure that they were set according to the operation performed.
  - The Referenced and Modified bits for the page operated on are cleared.
- After all pages are tested, the test verifies that all Referenced and Modified bits were cleared. The following table describes each Class 5 subtest, its object module, and its source file.

All subtests terminate with a halt command and store a halt code in address register A1. If the code stored in A1 is 0x100, then the subtest passed. Any other code in A1 indicates that the subtest failed.

> **NOTE**
>
> R&M refers to Referenced and Modified bits.

Table cpu4010–7, Class 5 Subtests

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	TIME
410	R&M load test, even bank march, code in upper memory, halfword accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1
411	R&M load test, odd bank march, code in upper memory, halfword accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1
420	R&M store test, even bank march, code in upper memory, halfword accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1
421	R&M store test, odd bank march, code in upper memory, halfword accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1
430	R&M execute test, even bank march, code in upper memory, halfword accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1
431	R&M execute test, odd bank march, code in upper memory, halfword accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1

NOTE 1: 5 sec. + 10 sec. per 128 Megabytes of memory

Class 6 Subtests

Class 6 subtests verify that the Physical Configuration Map's (PCM) bits can be set, verified, and cleared. Class 6 subtests execute in low memory based on byte and halfword accesses to I/O space, and in upper memory based on byte and halfword accesses to I/O space. Listed below are the events that occur to verify operation of the PCM's bits. The following table describes each Class 6 subtest, its object module and its source file.

- All PCM bits are cleared.
- The following steps are repeated in a loop until testing is complete:
  - The PCM's bits are set.
  - The PCM's bits are verified to ensure that they were set correctly.
  - The PCM's bits are cleared.
- The PCM's bits are checked to verify that they are clear.

All subtests terminate with a halt command and store a halt code in address register A1. If the code stored in A1 is 0x100, then the subtest passed. Any other code in A1 indicates that the subtest failed. Each Class 6 subtest requires 25 seconds to execute.

Table cpu4010–8, Class 6 Subtests

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	TIME (min/sec)
510	PCM march test, code in low memory, using byte accesses to I/O space	*cpu4010.rnn*	pcmbits.c	0:25
520	PCM march test, code in low memory, using halfword accesses to I/O space	*cpu4010.rnn*	pcmbits.c	0:25
610	PCM march test, code in upper memory, using byte accesses to I/O space	*cpu4010.rnn*	pcmbits.c	0:25
620	PCM march test, code in upper memory, using halfword accesses to I/O space	*cpu4010.rnn*	pcmbits.c	0:25

Class 7 Subtests

Class 7 subtests perform the operation of accessing memory in the same fashion as a Class 2 (byte accessing) and Class 4 (halfword accessing) subtests. However, the memory is accessed from different CPUs. This class allows verification that the Referenced and Modified Bits are correctly manipulated in a multiprocessor environment. This class is not executed if only one CPU is present.

All subtests terminate with a halt command and store a halt code in address register A1. If the code stored in A1 is 0x100, then the subtest passed. Any other code in A1 indicates that the subtest failed.

Table cpu4010–9, Class 7 Subtests

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	TIME
700	Multiple head concurrency loads	cpu4010.rnn	multi_test.c	0:10
710	Multiple head concurrency stores	cpu4010.rnn	mult_test.c	0:10

Class 8 Subtests

Class 8 subtests perform the same basic procedures of the Class 2 subtests. However, instead of performing operations only on the first address of a memory page, all addresses of memory are operated on. All Class 8 subtests execute with the test code residing in low memory and are based on byte accesses to I/O space. This test only needs to be executed if errors have resulted from running the first six classes of subtests. Each Class 8 subtest requires 25 minutes per 32 Megabytes of memory to execute. The following table describes each subtest, its object module, and its source file.

All subtests terminate with a halt command and store a halt code in address register A1. If the code stored in A1 is 0x100, then the subtest passed. Any other code in A1 indicates that the subtest failed.

> **NOTE**
>
> R&M refers to Referenced and Modified bits.

Table cpu4010–10, Class 8 Subtests

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	TIME
1110	R&M load test, even bank, code in low memory, byte accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1
1111	R&M load test, odd bank, code in low memory, byte accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1
1120	R&M store test, even bank, code in low memory, byte accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1
1121	R&M store test, odd bank, code in low memory, byte accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1
1130	R&M execute test, even bank, code in low memory, byte accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1
1131	R&M execute test, odd bank, code in low memory, byte accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1

NOTE 1: 25 minutes per 32 Megabytes of memory

Class 9 Subtests

Class 9 subtests perform the same basic procedures of the Class 3 subtests. However, instead of performing operations only on the first address of a memory page, all addresses of memory are operated on. All Class 9 subtests execute with the test code residing in upper memory and are based on byte accesses to I/O space. This test only needs to be executed if errors have resulted from running the first six classes of subtests. Each Class 9 subtest requires 10 minutes per 32 Megabytes of memory to execute. The following table describes each subtest, its object module, and its source file.

All subtests terminate with a halt command and store a halt code in address register A1. If the code stored in A1 is 0x100, then the subtest passed. Any other code in A1 indicates that the subtest failed.

> **NOTE**
>
> R&M refers to Referenced and Modified bits.

Table cpu4010–11, Class 9 Subtests

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	TIME
1210	R&M load test, even bank, code in upper memory, byte accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1
1211	R&M load test, odd bank, code in upper memory, byte accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1
1220	R&M store test, even bank, code in upper memory, byte accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1
1221	R&M store test, odd bank, code in upper memory, byte accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1
1230	R&M execute test, even bank, code in upper memory, byte accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1
1231	R&M execute test, odd bank, code in upper memory, byte accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1

NOTE 1: 10 minutes per 32 Megabytes of memory

Class 10 Subtests

Class 10 subtests perform the same basic procedures of the Class 4 subtests. However, instead of performing operations only on the first address of a memory page, all addresses of memory are operated on. All Class 10 subtests execute with the test code residing in low memory and are based on halfword accesses to I/O space. This test only needs to be executed if errors have resulted from running the first six classes of subtests. Each Class 10 subtest requires 15 minutes per 32 Megabytes of memory to execute. The following table describes each subtest, its object module, and its source file.

All subtests terminate with a halt command and store a halt code in address register A1. If the code stored in A1 is 0x100, then the subtest passed. Any other code in A1 indicates that the subtest failed.

> **NOTE**
>
> R&M refers to Referenced and Modified bits.

Table cpu4010–12, Class 10 Subtests

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	TIME
1310	R&M load test, even bank, code in low memory, halfword accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1
1311	R&M load test, odd bank, code in low memory, halfword accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1
1320	R&M store test, even bank, code in low memory, halfword accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1
1321	R&M store test, odd bank, code in low memory, halfword accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1
1330	R&M execute test, even bank, code in low memory, halfword accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1
1331	R&M execute test, odd bank, code in low memory, halfword accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1

NOTE 1: 15 minutes per 32 Megabytes of memory

Class 11 Subtests

Class 11 subtests perform the same basic procedures of the Class 5 subtests. However, instead of performing operations only on the first address of a memory page, all addresses of memory are operated on. All Class 11 subtests execute with the test code residing in upper memory and are based on halfword accesses to I/O space. This test only needs to be executed if errors have resulted from running the first six classes of subtests. Each Class 11 subtest requires 5 minutes per 32 Megabytes of memory to execute. The following table describes each subtest, its object module, and its source file.

All subtests terminate with a halt command and store a halt code in address register A1. If the code stored in A1 is 0x100, then the subtest passed. Any other code in A1 indicates that the subtest failed.

> **NOTE**
>
> R&M refers to Referenced and Modified bits.

Table cpu4010–13, Class 11 Subtests

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	TIME
1410	R&M load test, even bank, code in upper memory, halfword accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1
1411	R&M load test, odd bank, code in upper memory, halfword accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1
1420	R&M store test, even bank, code in upper memory, halfword accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1
1421	R&M store test, odd bank, code in upper memory, halfword accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1
1430	R&M execute test, even bank, code in upper memory, halfword accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1
1431	R&M execute test, odd bank, code in upper memory, halfword accesses to I/O space	*cpu4010.rnn*	*rmbits.c*	Note 1

NOTE 1: 5 minutes per 32 Megabytes of memory

Test Error Messages

Error messages for *cpu4010* are produced from two sources:

- Error messages specific to this test
- Common CPU test error messages

The error messages that are specific to *cpu4010* are presented below in alphabetical order with an explanation of each. For the CPU library error messages, refer to the "CPU Error Messages" appendix.

> **NOTE**
>
> In the following error messages, 'xxxxxxxx' represents the actual address that appears as part of the error message. Also, 'YYYY' and 'ZZZZ' represent the expected hexadecimal values of the Referenced and Modified bits for their respected memory page (refer to the *CONVEX Architecture Reference* for more information).

`Expected value: YYYY, Actual value: ZZZZ`

Indicates the expected value YYYY read for the referenced and modified bits when actually ZZZZ was read.

`Failed Boundary modified Bit check at i/o address: xxxxxxxx`

Indicates that with the odd page address encached in the ATU, a reference to an even page causes the modified bit for the odd page xxxxxxxx to be set.

`Failed Boundary Referenced Bit check at i/o address: xxxxxxxx`

Indicates that with the odd page address encached in the ATU, a reference to an even page causes the reference for the odd page xxxxxxxx to be set.

`Failed Boundary R/M Bit check at i/o address: xxxxxxxx`

Indicates that with the odd page address encached in the ATU, a reference to an even page causes the referenced and modified for the odd page xxxxxxxx to be set.

`Failed March Modified Bit check at i/o address: xxxxxxxx`

Indicates that during an up march test, the modified bit at i/o address xxxxxxxx was set when it should be clear.

`Failed March Referenced Bit check at i/o address: xxxxxxxx`

Indicates that during an up march test, the reference bit at i/o address xxxxxxxx was set when it should be clear.

`Failed March R/M Bit check at i/o address: xxxxxxxx`

Indicates that during an up march test, the Referenced and Modified bits at i/o address xxxxxxxx were set when it should be clear.

`Failed Modified Bit initialization at i/o address: xxxxxxxx`

Indicates that the code detected that the modified bit at i/o address xxxxxxxx was not properly cleared during initialization.

`Failed Non Resident Block ZZ Modified Bit at i/o address xxxxxxxx`

Indicates non resident memory PCM block ZZ has a modified bit set for the memory page indicated by xxxxxxxx.

`Failed Non Resident Block ZZ Referenced Bit at i/o address xxxxxxxx`

Indicates non resident memory PCM block ZZ has a referenced bit set for the memory page indicated by xxxxxxxx.

`Failed Non Resident Block ZZ Referenced and Modified Bit at i/o address xxxxxxxx`

Indicates non resident memory PCM block ZZ has both referenced and modified bits set for the memory page indicated by xxxxxxxx.

`Failed PCM Code Bits down march test at i/o address: xxxxxxxx`

Indicates that during a down march test, the PCM code at i/o address xxxxxxxx was incorrect for the machine's processor.

`Failed PCM Code Bits up march test at i/o address: xxxxxxxx`

Indicates that during an up march test, the PCM code at i/o address xxxxxxxx was incorrect for the machine's processor.

`Failed PCM down march test at i/o address: xxxxxxxx`

Indicates that during a down march test, the PCM value at i/o address xxxxxxxx was incorrect.

`Failed PCM Present Bit down march test at i/o address: xxxxxxxx`

Indicates that during a down march test, the PCM present bit at i/o address xxxxxxxx was clear when it should be set.

`Failed PCM Present Bit up march test at i/o address: xxxxxxxx`

Indicates that during an up march test, the pcm present bit at i/o address xxxxxxxx was set when it should be clear.

`Failed PCM up march test at i/o address: xxxxxxxx`

Indicates that during an up march test, the PCM value at i/o address xxxxxxxx was incorrect.

`Failed Reference Bit initialization at i/o address: xxxxxxxx`

Indicates that the code detected that the referenced bit at i/o address xxxxxxxx was not properly cleared during initialization.

`Failed Referenced/Modified Bit initialization at i/o address: xxxxxxxx`

Indicates that the code detected that the referenced and modified bits at i/o address xxxxxxxx were not properly cleared during initialization.

`Failed Touched Modified Bit check at i/o address: xxxxxxxx`

Indicates that a modified bit at i/o address xxxxxxxx was not set when the page it represents was accessed.

`Failed Touched Referenced Bit check at i/o address: xxxxxxxx`

Indicates that a referenced bit at i/o address xxxxxxxx was not set when the page it represents was accessed.

`Failed Untouched Modified Bit check at i/o address: xxxxxxxx`

Indicates that a modified bit at i/o address xxxxxxxx was set when the page it represents was not accessed.

`Failed Untouched Referenced Bit check at i/o address: xxxxxxxx`

Indicates that a referenced bit at i/o address xxxxxxxx was set when the page it represents was not accessed.

`Touched Block: YYYY, Touched Page: ZZZZ`

Indicates trying to touch page ZZZZ of pcm block YYYY.

SPU UNIX Error Messages

For a common list of possible error messages resulting from SPU UNIX, refer to the appendix on SPU UNIX Error Messages.

THIS PAGE INTENTIONALLY LEFT BLANK

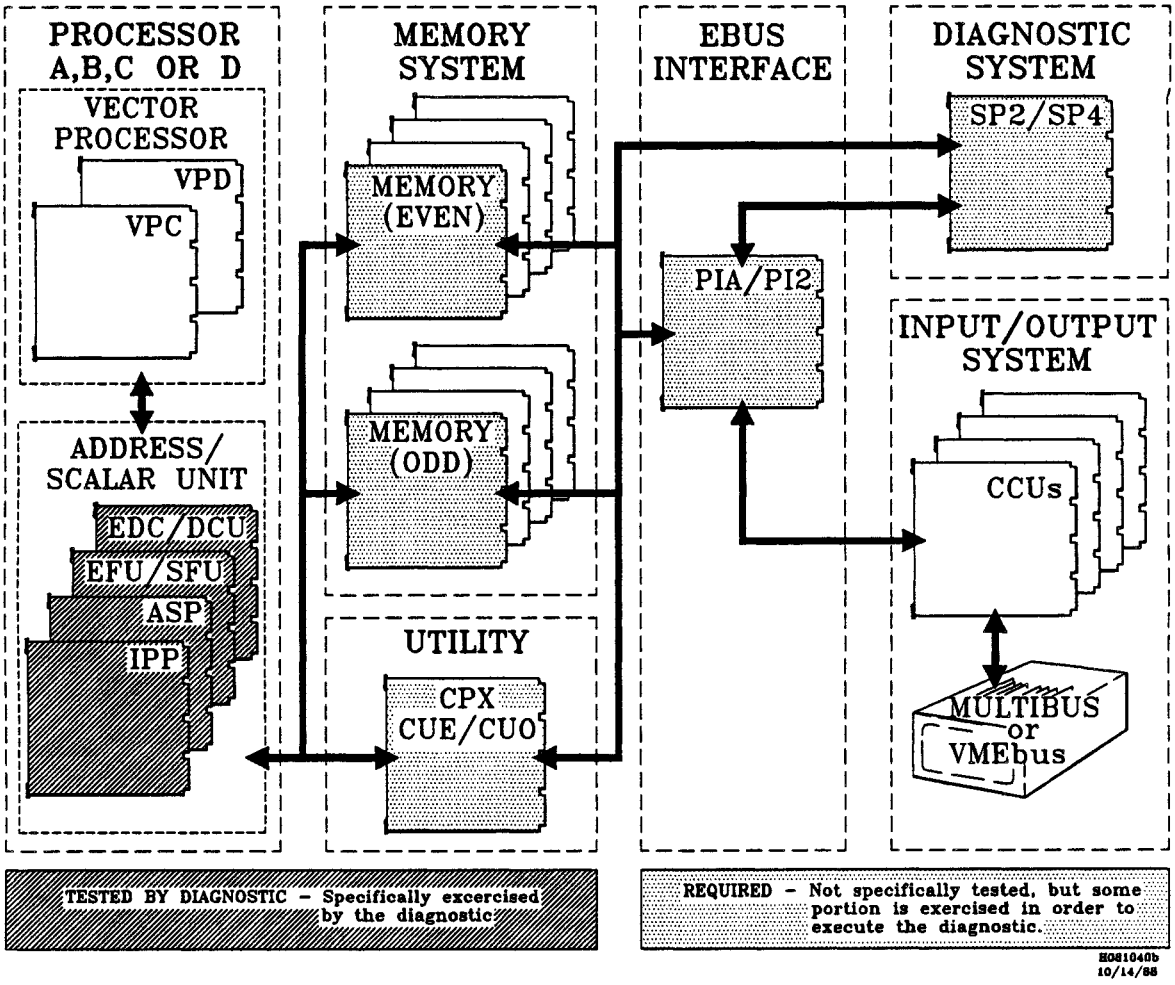
cpu4030

Scalar Building Block Test

Overview

The *cpu4030* test is the most basic of the CPU functional tests and is designed to test all nonprivileged scalar instructions in a nonexhaustive (verifies basic functionality) manner. The *cpu4030* test begins verification of the CPU by executing a variety of basic instructions out of the instruction cache (icache) only. No memory references are made and the only hardware initially tested are the DCU or EDC, IPP, SFU or EFU, and the ASP. Next, instructions are executed from the icache which require memory accesses for data. Finally, *cpu4030* performs operand fetches and data retrievals from memory. The following figure shows an overall view of what part of the system is being tested and which field replaceable units are required for the test to run.

Figure cpu4030–1, Functional Areas Tested by *cpu4030*



Prerequisites and Required Equipment

In order to run the *cpu4030* test, the Vector Processor Control (VPC) and the Vector Processor Data (VPD) must either be present in the machine or their slots must be terminated with terminators. Also, the boards listed in the following table must be operational. The table shows the tests used to verify the required boards. No additional equipment is required to run this test.

Table cpu4030–1, Required Functional Boards

BOARD	TEST TO VERIFY
Service Processor (SP2 or SP4)	*spu1000, spu4000*
PBUS Interface Adapter (PIA or PI2)	*pia4000* or *pi2_4000*
Memory System	*mem4000*
CPU Utility Board(s) (CPX or CUE/CUO)	*cpx4000*

NOTE

Memory System consists of a minimum of one pair of memory boards (one even and one odd).

Test Invocation

To invoke the *cpu4030* test, use the procedure shown in the following figure. All responses in **boldface** are entered by the user. The prompts and responses in the following figure would appear sequentially on the screen, one line at a time. All the prompts and responses are shown in one figure for convenience.

CAUTION

The invocation sequence shown in the following figure is the typical invocation sequence. However, the *initall* utility must be executed in some cases. If the system has just been powered up, if *mem4000* was executed with failures, or if *spu4000* was executed, then *initall* must be executed prior to any test execution. Failure to execute *initall* in these circumstances could result in invalid test results.

NOTE

Running the *initall* utility requires two to three minutes to execute depending on if the control stores have been previously loaded. However, it is suggested that *initall* be executed in the event that the state of the system is unknown.

Figure cpu4030–2, Test Invocation Sequence

```
(spu)>  cd /mnt/test
(spu)>  sysreset
(spu)>  dshell

CONVEX DIAGNOSTIC SHELL

:  test cpu4030 [-c [class numeral(s)]] [-s [subtest numeral(s)]] [+> filename]
```

NOTE

After entering the user response, "**dshell**", specific *dshell* parameters may be changed. Refer to the "Dshell and Iscan Overview" chapter of this manual for more information on *dshell*.

Entering only **test cpu4030** executes all *cpu4030* subtests sequentially. The user can execute a specific class(es) of subtest(s) or one or more individual subtests by using the *-c* or *-s* options, respectively. Detailed information for using these options can be found in the "Dshell and Iscan Overview" chapter of this manual. The **[+>***filename***]** option allows the test results to be appended to *filename*.

Test Parameter Menu

Once the test is invoked, a test parameter menu prompts for selection of default switches. If the test is run with all defaults invoked (user answers **y** to the first prompt), no other prompts are provided. If the user answers **n** to the first prompt (run test without default switches), then a series of prompts are presented. The following figure shows all possible prompts, their possible answers (shown in brackets [ ]), and their default answers (shown in parentheses ( )). The prompts and responses in the following figure would appear sequentially on the screen, one line at a time. All the prompts and responses are shown in one figure for convenience.

Figure cpu4030–3, *cpu4030* Test Parameter Menu

```
                        ENTER TEST PARAMETERS

                [ ]    Encloses allowed input ranges or values
                ( )    Encloses the default value
                ^       Returns to the previous prompt
                :nn     Returns to the prompt # nn
                :       Returns to the first unsatisfied prompt
                :?      Reviews previous entries

1:  Run default switches? [y,n]                              (y) ->
2:  CPUs to test: [ABCD]                                     (ABCD) ->
3:  Parallel test execution? [y,n]                           (n) ->
4:  Forced Faulting Enabled? [y,n]                           (n) ->
5:  Fault on Instruction Fetches? [y,n]                      (n) ->
6:  Sequential Execution? [y,n]                              (n) ->
7:  Timeout Scale Factor Enabled? [1-100]                    (1) ->
8:  Dcache Enabled? [y,n]                                    (y) ->
9:  Segment of Execution? [0-7]                              (0) ->
10: Loop Enabled? [y,n]                                      (n) ->
11: Chained Execution Mode? [y,n]                            (n) ->
12:  Hard Errors Enabled? [y,n]                              (y) ->
13:  Load CPU Code? [y,n]                                    (y) ->
```

> **NOTE**
>
> In prompt 2 in the previous figure, ABCD represents all available CPUs. Only the available CPUs within the machine under test will be displayed as possible choices.

Prompt Explanations

A description of what each prompt means follows:

```
Run default switches? [y/n]                                    (y) ->
```

If the user responds with **y** or <**CR**>, no additional test parameter prompts are displayed and testing begins. However, if a negative response is supplied, additional test parameter prompts are displayed allowing modification of the default selections.

The following prompts are only displayed and answered if the first prompt is answered with **n**:

```
CPUs to test: [ABCD]                                           (ABCD) ->
```

This prompt allows selection of the CPU(s) to be used in the test. The possible selections, represented by ABCD, will consist of all available CPUs. The default, ABCD, will consist of all available CPUs.

```
Parallel test execution?                                       (n) ->
```

This prompt allows parallel test execution to be enabled or disabled. This prompt is only displayed if the `CPUs to test:` prompt is answered with multiple CPUs.

```
Forced Faulting Enabled? [y,n]                                 (n) ->
```

If answered with **y**, normal force faulting occurs on all data references. The system will force a non-resident data exception to occur on every data reference. If this option is enabled, subtest execution time is increased and the Timeout Scale Factor requires adjustment to prevent the Service Processor from terminating the test prematurely. For an explanation of force faulting, refer to the glossary appendix of this manual.

```
Fault on Instruction Fetches? [y,n]                            (n) ->
```

In answered with **y**, force faulting occurs on instruction fetches in addition to data references. This prompt is only supplied to the user if the previous prompt is answered with **y**.

```
Sequential Execution? [y,n]                                    (n) ->
```

If set by entering **y**, the sequential bit in the PSW will be set to forced sequential execution mode.

```
Timeout Scale Factor [1-100]                    (1) ->
```

This prompt allows adjustment of the timeout factor. The normal timeout factor is multiplied by the number selected to increase the timeout factor. For example, if **5** is entered, the normal timeout factor is multiplied by 5 and it will take the test five times as long to timeout.

```
Dcache Enabled? [y,n]                           (y) ->
```

The Dcache is normally enabled; however, if it is suspected broken it can be disabled by entering **n** at this prompt.

```
Segment of Execution? [0-7]                     (0) ->
```

The segment of execution is contained in bits<31..29> of the Program Counter (PC). If **0** is entered, then bits<31..29> of the PC are 000 and the test is run in ring zero. If **1** is entered, then bits<31..29> of the PC are 001 and the test is run in ring one. If **2** is entered, then bits<31..29> of the PC are 010 and the test is run in ring two. If **3** is entered, then bits<31..29> of the PC are 011 and the test is run in ring three. If **4**, **5**, **6**, or **7** is entered, then bits<31..29> of the PC are 100, 101, 110, and 111 respectively and the test is run in ring four. Refer to the *CONVEX Architecture Reference* for more information concerning the meaning of rings in the machine architecture.

```
Loop Enabled? [y,n]                             (n) ->
```

If **y** is entered, hard looping on a specific subtest is enabled. When looping is enabled, the halt instruction of the specified subtest is changed to a branch back to the beginning of the subtest. This puts the subtest into an infinite loop which the user must break out of by typing **Ctrl- C**.

```
Chained Execution Mode? [y,n]                   (y) ->
```

With this option enabled, the test is executed in chained mode which causes the CPU to perform subtest sequencing. The Service Processor is unaware of the action of the CPU unless a subtest fails or unless all of the subtests pass. If this option is enabled, test execution time will be greatly reduced. However, the only information printed to the console upon completion or failure (regardless of the cause) is the message, "Subtest 1 passed," or "Subtest 1 failed." Also, this option can not be enabled if the **-c** or the **-s** options were used in the invocation procedure. This option disables all Class 4 subtests.

```
Hard Errors Enabled? [y,n]                      (y) ->
```

If this option is enabled (by entering **y**) and a hard error occurs, the clocks will be stopped and the test will fail. If this option is disabled (by entering **n**), parity errors and other sources of hard errors will go undetected. It is recommended that hard errors normally be enabled.

```
Load CPU Code? [y,n]                            (y) ->
```

If the CPU code for this test is already in memory, enter **n** for this prompt and the code will not be reloaded (thus saving time).

When all prompts have been answered, the user is shown a test parameter summary which echos the prompts that have been answered.

Figure cpu4030–4, Sample Test Parameter Summary

```
                    Test Parameter Summary

Run default switches?                          :  y
CPUs to test:                                  :  ab
Parallel test execution?                       :  n
Forced Faulting Enabled?                       :  y
Sequential Execution?                          :  n
Timeout Scale Factor Enabled?                  :  1
Dcache Enabled?                                :  y
Segment of Execution?                          :  0
Loop Enabled?                                  :  n
Hard Errors Enabled?                           :  y
Load CPU Code?                                 :  y
```

Hardware Initialization Sequence

After the last prompt is entered by the user (and before test code execution) the following events occur:

> **NOTE**
>
> The first two events are accomplished at the initial start of the *cpu4030* test. The remaining events are accomplished when each subtest is initialized.

- For each CPU, each module's page table entries are set up in main memory. Page table entries begin at the last available address in main memory and work toward low memory.
- For each CPU, each module is loaded into main memory following the logical to physical mapping described by the page tables. The exact logical to physical mapping that the modules are loaded into depends upon which segment of execution is selected by the user.
- The system is initialized (the memory system, CPX, PIA, and the CPU boards are reset).
- For each CPU, parity is initialized in the scalar processor by sending the scalar processor into a microcode routine and issuing clocks.
- Each CPU's scratch RAM is loaded with various values to control the execution of the test.
- For each CPU, the initial Program Counter (PC) is loaded into register T0 and the initial Program Status Word (PSW) is loaded into register T2.
- For each CPU, control store is initialized to the cold-start location.
- Clocks are turned on to the processor selected. If parallel execution is selected, each CPU's clocks are turned on at one time. If sequential execution is selected, each CPU's clocks are turned on one at a time.

Memory Allocation

Immediately before test code execution, a current memory allocation screen is displayed. The following figure is an example of the current memory allocation screen. The physical and logical addresses shown, as well as the filenames, are only representative; the actual addresses will vary depending on installed memory & file sizes.

The following list explains what each column the current memory allocation screen depicts:

- **First Column** — File Number - useful in conjunction with the mm(1d) utility.
- **Second Column** — Physical memory addresses where the specified file is loaded
- **Third Column** — Process identification
- **Fourth Column** — File name (actual path is *./filename* or /mnt/test/CPU/*filename*). The entries `pte1`, `pte2`, and `ptet` are not actual files but are instead indications of the page tables.
- **Fifth Column** — Logical starting address of the specified file

Figure cpu4030–5, Current Memory Allocation Screen

```
                    Current Memory Allocation

File No.  Physical  Address Pid File Name                    Logical Offset
      1   00000000-00027fff   0 p0r0_4030                   00000000
      2   00028000-000b1fff   0 cpu4030.rnn                 00022000
      1   000b2000-000d9fff   1 p0r0_4030                   00000000
      2   000da000-00163fff   1 cpu4030.rnn                 00022000
   ----   03ff7000-03ff9fff   1 ptet                        NA
   ----   03ffa000-03ffafff   1 pte2                        NA
   ----   03ffb000-03ffdfff   0 ptet                        NA
   ----   03ffe000-03ffefff   0 pte2                        NA
   ----   03fffc00-3fffffff   1 pte1                        NA
```

Class Descriptions

There are four classes of subtests for *cpu4030*. Each *cpu4030* subtest takes approximately two seconds to execute.

Class 1 Subtests

Class 1 subtests are the most basic of the CPU verification tests. These subtests require no access to main memory for data or instructions. They begin to verify the scalar functions of the CPU in a nonexhaustive manner out of the icache. Instructions are loaded into the icache via scan before the test begins. Since all instructions have been loaded into the icache, no requests to main memory for instructions are required. Data operands are supplied by extensive use of immediate

data supplied with the instructions. All subtests terminate with a halt command and store a halt code in address register A1. If the code stored in A1 is 0x100, then the subtest passed. Any other code in A1 indicates that the subtest failed.

Each Class 1 subtest is listed in the following figure with its brief description (instructions performed), its object module, and its source file. The object module is the actual executable code and the source file is the assembly language source.

> **NOTE**
>
> In the following tables, the **TEST PERFORMED** column lists each particular instruction that is tested. For more information on each instruction's meaning, refer to the "Opcodes Sorted by Name" appendix in this manual or the *CONVEX Architecture Reference*.

Table cpu4030–2, Class 1 Subtests

SUBTEST	TEST PERFORMED			OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
1	*ld.h #N,Ak*	*ld.w #N,Ak*		cpu4030.rnn	st_01.s	0:02	0:04
2	*ld.w #N,Sk*	*ld.w #N,Ak*		cpu4030.rnn	st_02.s	0:02	0:04
5	*mov Aj,Ak*	*mov Aj,Sk*		cpu4030.rnn	st_05.s	0:02	0:04
6	*mov Sj,Ak*	*mov.w Sj,Sk*	*mov.l Sj,Sk*	cpu4030.rnn	st_06.s	0:02	0:04
7	*mov PC,Ak*			cpu4030.rnn	st_07.s	0:02	0:04
10	*and Aj,Ak*	*and #N,Ak*		cpu4030.rnn	st_10.s	0:02	0:04
11	*and Sj,Sk*	*and #N,Sk*		cpu4030.rnn	st_11.s	0:02	0:04
14	*or Aj,Ak*	*or #N,Ak*		cpu4030.rnn	st_14.s	0:02	0:04
15	*or Sj,Sk*	*or #N,Sk*		cpu4030.rnn	st_15.s	0:02	0:04
16	*mov Ak,psw*	*mov psw,Ak*		cpu4030.rnn	st_16.s	0:02	0:04
18	*xor Aj,Ak*	*xor #N,Ak*		cpu4030.rnn	st_18.s	0:02	0:04
19	*xor Sj,Sk*	*xor #N,Sk*		cpu4030.rnn	st_19.s	0:02	0:04
20	*not Aj,Ak*			cpu4030.rnn	st_20.s	0:02	0:04
21	*not Sj,Sk*			cpu4030.rnn	st_21.s	0:02	0:04
24	*add.h Aj,Ak*	*add.w Aj,Ak*	*add.h #N,Ak*	cpu4030.rnn	st_24.s	0:02	0:04
	add.w #N,Ak	*0:04*					
25	*add.b Sj,Sk*			cpu4030.rnn	st_25.s	0:02	0:04
26	*add.h #N,Sk*	*add.h Sj,Sk*		cpu4030.rnn	st_26.s	0:02	0:04
27	*add.w #N,Sk*	*add.w Sj,Sk*	*add.w Sj,Ak*	cpu4030.rnn	st_27.s	0:02	0:04
28	*add.l Sj,Sk*			cpu4030.rnn	st_28.s	0:02	0:04
30	*sub.h #n,a*	*sub.h ai,Aj*		cpu4030.rnn	st_30.s	0:02	0:04
31	*sub.w #n,Ak*	*sub.w Aj,Ak*		cpu4030.rnn	st_31.s	0:02	0:04
32	*sub.b Sj,Sk*			cpu4030.rnn	st_32.s	0:02	0:04
33	*sub.h #n,Sj*	*sub.w Sj,Sk*		cpu4030.rnn	st_33.s	0:02	0:04
34	*sub.w #,Sk*	*sub.w Sj,Sk*		cpu4030.rnn	st_34.s	0:02	0:04
35	*sub.l Sj,Sk*			cpu4030.rnn	st_35.s	0:02	0:04
40	*neg.h Aj,Ak*			cpu4030.rnn	st_40.s	0:02	0:04
41	*neg.w Aj,Ak*			cpu4030.rnn	st_41.s	0:02	0:04
42	*neg.b Sj,Sk*			cpu4030.rnn	st_42.s	0:02	0:04
43	*neg.h Sj,Sk*			cpu4030.rnn	st_43.s	0:02	0:04
44	*neg.w Sj,Sk*			cpu4030.rnn	st_44.s	0:02	0:04
45	*neg.l Sj,Sk*			cpu4030.rnn	st_45.s	0:02	0:04
50	*branches*			cpu4030.rnn	st_50.s	0:02	0:04
52	*cmp.h Aj,Ak*			cpu4030.rnn	st_52.s	0:02	0:04
53	*cmp.w Sj,Ak*			cpu4030.rnn	st_53.s	0:02	0:04
54	*cmp.h #N,Ak*			cpu4030.rnn	st_54.s	0:02	0:04
55	*cmp.w #,Ak*			cpu4030.rnn	st_55.s	0:02	0:04

Table cpu4030-2, Class 1 Subtests (continued)

SUBTEST	TEST PERFORMED			OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
60	*cmp.b Sj,Sk*			cpu4030.rnn	st_60.s	0:02	0:04
61	*cmp.h Sj,Sk*			cpu4030.rnn	st_61.s	0:02	0:04
62	*cmp.w Sj,Sk*			cpu4030.rnn	st_62.s	0:02	0:04
63	*cmp.h #,Sk*			cpu4030.rnn	st_63.s	0:02	0:04
64	*cmp.w #N,Sk*			cpu4030.rnn	st_64.s	0:02	0:04
65	*cmp.l Sj,Sk*			cpu4030.rnn	st_65.s	0:02	0:04
66	*shf Aj,Ak*	*shf #n,Ak*	*shf #N,Ak*	cpu4030.rnn	st_66.s	0:02	0:04
67	*shf Sj,Sk*	*shf #N,Sk*		cpu4030.rnn	st_67.s	0:02	0:04
68	*tzc Sj,Sk*			cpu4030.rnn	st_68.s	0:02	0:04
69	*plc.t Sj,Sk*			cpu4030.rnn	st_69.s	0:02	0:04
70	*mul.h #n,Ak*	*mul.h #N,Ak*	*mul.h Aj,Ak*	cpu4030.rnn	st_70.s	0:02	0:04
71	*mul.w #n,Ak*	*mul.w #N,Ak*	*mul.w Aj,Ak*	cpu4030.rnn	st_71.s	0:02	0:04
72	*mul.b Sj,Sk*			cpu4030.rnn	st_72.s	0:02	0:04
73	*mul.h #N,Sk*	*mul.h Sj,Sk*		cpu4030.rnn	st_73.s	0:02	0:04
74	*mul.w #N,Sk*	*mul.w Sj,Sk*		cpu4030.rnn	st_74.s	0:02	0:04
75	*mul.l Sj,Sk*			cpu4030.rnn	st_75.s	0:02	0:04
80	*div.h #n,Ak*	*div.h #N,Ak*	*div.h Aj,Ak*	cpu4030.rnn	st_80.s	0:02	0:04
81	*div.w #n,Ak*	*div.w #N,Ak*	*div.w Aj,Ak*	cpu4030.rnn	st_81.s	0:02	0:04
82	*div.b Sj,Sk*			cpu4030.rnn	st_82.s	0:02	0:04
83	*div.h #n,Sk*	*div.h #N,Sk*	*div.h Sj,Sk*	cpu4030.rnn	st_83.s	0:02	0:04
84	*div.w #n,Sk*	*div.h #N,Sk*	*div.h Sj,Sk*	cpu4030.rnn	st_84.s	0:02	0:04
85	*div.l Sj,Sk*			cpu4030.rnn	st_85.s	0:02	0:04
90	*cvt(w.b,b.w,w.h,h.w) Aj,Ak*			cpu4030.rnn	st_90.s	0:02	0:04
91	*cvt(w.b,b.w,w.h,h.w,w.l,l.w) Sj,Sk*			cpu4030.rnn	st_91.s	0:02	0:04
100	*eq.s #n,Sk*	*eq.s #N,Sk*	*eq.s Sj,Sk*	cpu4030.rnn	st_100.s	0:02	0:04
101	*le.s #n,Sk*	*le.s #N,Sk*	*eq.s Sj,Sk*	cpu4030.rnn	st_101.s	0:02	0:04
102	*lt.s #n,Sk*	*lt.s #N,Sk*	*eq.s Sj,Sk*	cpu4030.rnn	st_102.s	0:02	0:04
103	*eq.d Sj,Sk*			cpu4030.rnn	st_103.s	0:02	0:04
104	*le.d Sj,Sk*			cpu4030.rnn	st_104.s	0:02	0:04
105	*lt.d Sj,Sk*			cpu4030.rnn	st_105.s	0:02	0:04
110	*add.s #N,Sk Sj,Sk*			cpu4030.rnn	st_110.s	0:02	0:04
111	*add.d Sj,Sk*			cpu4030.rnn	st_111.s	0:02	0:04
115	*sub.s #N,Sk Sj,Sk*			cpu4030.rnn	st_115.s	0:02	0:04
116	*sub.d Sj,Sk*			cpu4030.rnn	st_116.s	0:02	0:04
125	*neg.s Sj,Sk*			cpu4030.rnn	st_125.s	0:02	0:04
126	*neg.d Sj,Sk*			cpu4030.rnn	st_126.s	0:02	0:04
130	*mul.s #N,Sk*	*mul.s Sj,Sk*		cpu4030.rnn	st_130.s	0:02	0:04
131	*mul.d Sj,Sk*			cpu4030.rnn	st_131.s	0:02	0:04
132	*div.s #N,Sk*	*div.s Sj,Sk*		cpu4030.rnn	st_132.s	0:02	0:04
133	*div.d Sj,Sk*			cpu4030.rnn	st_133.s	0:02	0:04
140	*cvtw.s Sj,Sk*			cpu4030.rnn	st_140.s	0:02	0:04
142	*cvts.w Sj,Sk*			cpu4030.rnn	st_142.s	0:02	0:04
144	*cvtd.l Sj,Sk*			cpu4030.rnn	st_144.s	0:02	0:04
146	*cvtl.d Sj,Sk*			cpu4030.rnn	st_146.s	0:02	0:04
148	*cvtl.s Sj,Sk*			cpu4030.rnn	st_148.s	0:02	0:04
150	*cvts.l Sj,Sk*			cpu4030.rnn	st_150.s	0:02	0:04
152	*cvts.d Sj,Sk*			cpu4030.rnn	st_152.s	0:02	0:04
154	*cvtd.s Sj,Sk*			cpu4030.rnn	st_154.s	0:02	0:04

Table cpu4030-2, Class 1 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
1100	*eq.s #N sk eq.s sj sk (IEEE)*	cpu4030.rnn	st_1100.s	0:02	0:04
1101	*le.s #N sk le.s sj sk (IEEE)*	cpu4030.rnn	st_1101.s	0:02	0:04
1102	*lt.s #N sk lt.s sj sk (IEEE)*	cpu4030.rnn	st_1102.s	0:02	0:04
1103	*eq.d sj sk (IEEE)*	cpu4030.rnn	st_1103.s	0:02	0:04
1104	*le.d sj sk (IEEE)*	cpu4030.rnn	st_1104.s	0:02	0:04
1105	*lt.d sj sk (IEEE)*	cpu4030.rnn	st_1105.s	0:02	0:04
1110	*add.s #N sk add.s sj sk (IEEE)*	cpu4030.rnn	st_1110.s	0:02	0:04
1111	*add.d sj sk (IEEE)*	cpu4030.rnn	st_1111.s	0:02	0:04
1115	*sub.s sj sk sub.s #N sk (IEEE)*	cpu4030.rnn	st_1115.s	0:02	0:04
1116	*sub.d sj sk (IEEE)*	cpu4030.rnn	st_1116.s	0:02	0:04
1127	*neg.s sj sk (IEEE)*	cpu4030.rnn	st_1127.s	0:02	0:04
1128	*neg.d sj sk (IEEE)*	cpu4030.rnn	st_1128.s	0:02	0:04
1131	*mul.d sj sk (IEEE)*	cpu4030.rnn	st_1131.s	0:02	0:04
1132	*div.s #n sk div.s sj sk (IEEE)*	cpu4030.rnn	st_1132.s	0:02	0:04
1133	*div.d sj sk (IEEE)*	cpu4030.rnn	st_1133.s	0:02	0:04
1135	*mul.s #N sk mul.s sj sk (IEEE)*	cpu4030.rnn	st_1135.s	0:02	0:04
1140	*cvtw.s sj sk (IEEE)*	cpu4030.rnn	st_1140.s	0:02	0:04
1142	*cvtw.s sj sk (IEEE)*	cpu4030.rnn	st_1142.s	0:02	0:04
1144	*cvtd.l sj sk (IEEE)*	cpu4030.rnn	st_1144.s	0:02	0:04
1146	*cvtl.d sj sk (IEEE)*	cpu4030.rnn	st_1146.s	0:02	0:04
1148	*cvtl.s sj sk (IEEE)*	cpu4030.rnn	st_1148.s	0:02	0:04
1150	*cvts.l sj sk (IEEE)*	cpu4030.rnn	st_1150.s	0:02	0:04
1152	*cvts.d sj sk (IEEE)*	cpu4030.rnn	st_1152.s	0:02	0:04
1154	*cvtd.s sj sk (IEEE)*	cpu4030.rnn	st_1154.s	0:02	0:04

Class 2 Subtests

Class 2 subtests are icache based and require memory accesses for data. Each test accesses main memory for data, but not for instructions. Unlike Class 1, however, data used by these subtests are now loaded from and stored to main memory. All subtests terminate with a halt command and store a halt code in address register A1. If the code stored in A1 is 0x100, then the subtest passed. Any other code in A1 indicates that the subtest failed.

Each Class 2 subtest is listed in the following figure with its brief description (instructions performed), its object module, and its source file. The object module is the actual executable code and the source file is the commented code.

Table cpu4030–3, Class 2 Subtests

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
200	*ld.b <effa>,Ak byte boundaries*	cpu4030.rnn	st_200.s	0:02	0:04
201	*ld.h <effa>,Ak halfword boundaries*	cpu4030.rnn	st_201.s	0:02	0:04
202	*ld.w <effa>,Ak word boundaries*	cpu4030.rnn	st_202.s	0:02	0:04
205	*ld.b <effa>,Sk byte boundaries*	cpu4030.rnn	st_205.s	0:02	0:04
206	*ld.h <effa>,Sk halfword boundaries*	cpu4030.rnn	st_206.s	0:02	0:04
207	*ld.w <effa>,Sk word boundaries*	cpu4030.rnn	st_207.s	0:02	0:04
208	*ld.l <effa>,Sk word boundaries*	cpu4030.rnn	st_208.s	0:02	0:04
211	*ld.h <effa>,Ak unaligned*	cpu4030.rnn	st_211.s	0:02	0:04
212	*ld.w <effa>,Ak unaligned*	cpu4030.rnn	st_212.s	0:02	0:04
216	*ld.h <effa>,Sk unaligned*	cpu4030.rnn	st_216.s	0:02	0:04
217	*ld.w <effa>,Sk unaligned*	cpu4030.rnn	st_217.s	0:02	0:04
218	*ld.l <effa>,Sk unaligned*	cpu4030.rnn	st_218.s	0:02	0:04
220	*st.b Ak,<effa> byte boundaries*	cpu4030.rnn	st_220.s	0:02	0:04
221	*st.h Ak,<effa> halfword boundaries*	cpu4030.rnn	st_221.s	0:02	0:04
222	*st.w Ak,<effa> word boundaries*	cpu4030.rnn	st_222.s	0:02	0:04
225	*st.b Sk,<effa> byte boundaries*	cpu4030.rnn	st_225.s	0:02	0:04
226	*st.h Sk,<effa> halfword boundaries*	cpu4030.rnn	st_226.s	0:02	0:04
227	*st.w Sk,<effa> word boundaries*	cpu4030.rnn	st_227.s	0:02	0:04
228	*st.l Sk,<effa> word boundaries*	cpu4030.rnn	st_228.s	0:02	0:04
231	*st.h Ak,<effa> unaligned boundaries*	cpu4030.rnn	st_231.s	0:02	0:04
232	*st.w Ak,<effa> unaligned boundaries*	cpu4030.rnn	st_232.s	0:02	0:04
236	*st.h Sk,<effa> unaligned boundaries*	cpu4030.rnn	st_236.s	0:02	0:04
237	*st.w Sk,<effa> unaligned boundaries*	cpu4030.rnn	st_237.s	0:02	0:04
238	*st.l Sk,<effa> unaligned boundaries*	cpu4030.rnn	st_238.s	0:02	0:04
240	*psh.l Sk psh.w Sk psh.w Ak pop.w Ak pop.l Sk pop.w Sk pshea*	cpu4030.rnn	st_240.s	0:02	0:04
241	*psh.l Sk psh.w Sk psh.w Ak pop.w Ak pop.w Sk pop.l Sk pshea(unaligned)*	cpu4030.rnn	st_241.s	0:02	0:04
242	*tas <effa>*	cpu4030.rnn	st_242.s	0:02	0:04
245	*ldea <effa>,Ak*	cpu4030.rnn	st_245.s	0:02	0:04
250	*callq aligned stack rtnq aligned stack*	cpu4030.rnn	st_250.s	0:02	0:04
251	*callq unaligned stack rtnq unaligned stack*	cpu4030.rnn	st_251.s	0:02	0:04
252	*call aligned stack rtn aligned stack*	cpu4030.rnn	st_252.s	0:02	0:04
253	*call unaligned stack rtn unaligned stack*	cpu4030.rnn	st_253.s	0:02	0:04
254	*calls aligned stack rtn aligned stack*	cpu4030.rnn	st_254.s	0:02	0:04
255	*calls unaligned stack rtn unaligned stack*	cpu4030.rnn	st_255.s	0:02	0:04

Table cpu4030-2, Class 2 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
260	*st.b Ak,@ <effa> byte boundaries*	cpu4030.rnn	st_260.s	0:02	0:04
261	*st.h Ak,@ <effa> halfword boundaries*	cpu4030.rnn	st_261.s	0:02	0:04
262	*st.w Ak,@ <effa> word boundaries*	cpu4030.rnn	st_262.s	0:02	0:04
265	*st.b Sk,@ <effa> byte boundaries*	cpu4030.rnn	st_265.s	0:02	0:04
266	*st.h Sk,@ <effa> halfword boundaries*	cpu4030.rnn	st_266.s	0:02	0:04
267	*st.w Sk,@ <effa> word boundaries*	cpu4030.rnn	st_267.s	0:02	0:04
268	*st.l Sk,@ <effa> word boundaries*	cpu4030.rnn	st_268.s	0:02	0:04
271	*st.h Ak,@ <effa> unaligned boundaries*	cpu4030.rnn	st_271.s	0:02	0:04
272	*st.w Ak,@ <effa> unaligned boundaries*	cpu4030.rnn	st_272.s	0:02	0:04
276	*st.h Sk,@ <effa> unaligned boundaries*	cpu4030.rnn	st_276.s	0:02	0:04
277	*st.w Sk,@ <effa> unaligned boundaries*	cpu4030.rnn	st_277.s	0:02	0:04
278	*st.l Sk,@ <effa> unaligned boundaries*	cpu4030.rnn	st_278.s	0:02	0:04
280	*ld.b @ <effa>,Ak byte boundaries*	cpu4030.rnn	st_280.s	0:02	0:04
281	*ld.h @ <effa>,Ak halfword boundaries*	cpu4030.rnn	st_281.s	0:02	0:04
282	*ld.w @ <effa>,Ak word boundaries*	cpu4030.rnn	st_282.s	0:02	0:04
285	*ld.b @ <effa>,Sk byte boundaries*	cpu4030.rnn	st_285.s	0:02	0:04
286	*ld.h @ <effa>,Sk halfword boundaries*	cpu4030.rnn	st_286.s	0:02	0:04
287	*ld.w @ <effa>,Sk word boundaries*	cpu4030.rnn	st_287.s	0:02	0:04
288	*ld.l @ <effa>,Sk word boundaries*	cpu4030.rnn	st_288.s	0:02	0:04
291	*ld.h @ <effa>,Ak unaligned*	cpu4030.rnn	st_291.s	0:02	0:04
292	*ld.w @ <effa>,Ak unaligned*	cpu4030.rnn	st_292.s	0:02	0:04
296	*ld.h @ <effa>,Sk unaligned*	cpu4030.rnn	st_296.s	0:02	0:04
297	*ld.w @ <effa>,Sk unaligned*	cpu4030.rnn	st_297.s	0:02	0:04
298	*ld.l @ <effa>,Sk unaligned*	cpu4030.rnn	st_298.s	0:02	0:04

Class 3 Subtests

Class 3 subtests access main memory for both data and instructions. At this point, pre-loading the icache with instructions before the test begins does not occur. Instructions are fetched and executed as they would be during normal operation of the machine. All subtests terminate with a halt command and store a halt code in address register A1. If the code stored in A1 is 0x100, then the subtest passed. Any other code in A1 indicates that the subtest failed.

Each Class 3 subtest is listed in the following figure with its brief description (instructions performed), its object module, and its source file. The object module is the actual executable code and the source file is the commented code.

Table cpu4030–4, Class 3 Subtests

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
300	*load byte across page bnd*	cpu4030.rnn	st_300.s	0:02	0:04
301	*load halfword across page bnd*	cpu4030.rnn	st_301.s	0:02	0:04
302	*load word across page bnd*	cpu4030.rnn	st_302.s	0:02	0:04
303	*load longword across page bnd*	cpu4030.rnn	st_303.s	0:02	0:04
304	*store byte across page bnd*	cpu4030.rnn	st_304.s	0:02	0:04
305	*store halfword across page bnd*	cpu4030.rnn	st_305.s	0:02	0:04
306	*store word across page bnd*	cpu4030.rnn	st_306.s	0:02	0:04
307	*store longword across page bnd*	cpu4030.rnn	st_307.s	0:02	0:04
308	*load byte @, address at page bnd*	cpu4030.rnn	st_308.s	0:02	0:04
309	*load halfword @, address at page bnd*	cpu4030.rnn	st_309.s	0:02	0:04
310	*load word @, address at page bnd*	cpu4030.rnn	st_310.s	0:02	0:04
311	*load longword @, address at page bnd*	cpu4030.rnn	st_311.s	0:02	0:04
312	*load byte @, address and data at page bnd*	cpu4030.rnn	st_312.s	0:02	0:04
313	*load halfword @, address and data at page bnd*	cpu4030.rnn	st_313.s	0:02	0:04
314	*load word @, address and data at page bnd*	cpu4030.rnn	st_314.s	0:02	0:04
315	*load longword @, address and data at page bnd*	cpu4030.rnn	st_315.s	0:02	0:04
350	*execute at page bnd*	cpu4030.rnn	st_350.s	0:02	0:04
351	*execute different Size op codes at page bnd*	cpu4030.rnn	st_351.s	0:02	0:04
352	*execute different Size op codes at page bnd*	cpu4030.rnn	st_352.s	0:02	0:04
353	*branch near page bnd*	cpu4030.rnn	st_353.s	0:02	0:04

Class 4 Subtests

The *cpu4030* Class 4 subtests verify two basic capabilities of the machine that are not explicitly tested anywhere else. The first is the capability to execute code, branch forward and backward across all major carry addresses in the Program Counter (PC). The second is the verification of the ability to correctly wrap instructions within the current ring.

The carry testing is performed by subtests 500-515. Each of these subtests use the same object module. The module is loaded into a different logical address for each subtest, each module uses up two pages of logical address space (refer to table Class 4 Subtests). The logical addresses used are 0xf000, 0xff000, 0xfff000, and 0xffff000. Each subtest will execute code across all of the carries possible within its logical address space. For example subtest 500 will test carries from 0xe → 0x10, 0xfe → 0x100, 0xffe → 0x1000, and 0xfffe → 0x10000.

The ring wrapping is tested by executing code at the end of each segment. Code is mapped into the end of each segment (module wrapu_4030) and the end of each segment (wrapl_4030, p0r0_4030.1, or p0rN_4030.1). The module at the end of the segment is then executed. For segments 0-3, execution at the end of the segment should cause the PC to "wrap" back down to the beginning of the segment. This prevents the ability to violate architectural ring protection constraints. Segments that exist in ring 4, (4, 5, 6, 7) behave differently. These segments should behave as one ring. For example, it should be possible to execute code from segment 4 and end up in segment 5, segment 5 to segment 6 and segment 6 to segment 7. If code is executed at the end of segment 7, it should go back to the beginning of the ring and end up in segment 4.

Table cpu4030–5, Class 4 Subtests

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
500	Pc Carry Test I	pccarry_4030	pc_carry.s	0:02	0:04
505	Pc Carry Test II	pccarry_4030	pc_carry.s	0:02	0:04
510	Pc Carry Test III	pccarry_4030	pc_carry.s	0:02	0:04
515	Pc Carry Test IV	pccarry_4030	pc_carry.s	0:02	0:04
520	Pc Wraparound Test I	wrapu_4030	ring_wrapu.s	0:02	0:04
525	Pc Wraparound Test II	wrapu_4030	ring_wrapu.s	0:02	0:04
530	Pc Wraparound Test III	wrapu_4030	ring_wrapu.s	0:02	0:04
535	Pc Wraparound Test IV	wrapu_4030	ring_wrapu.s	0:02	0:04
540	Pc Wraparound Test V	wrapu_4030,wrapl_4030	ring_wrapu.s/ring_wrapl.s	0:02	0:04
545	Pc Wraparound Test VI	wrapu_4030,wrapl_4030	ring_wrapu.s/ring_wrapl.s	0:02	0:04
550	Pc Wraparound Test VII	wrapu_4030,wrapl_4030	ring_wrapu.s/ring_wrapl.s	0:02	0:04
555	Pc Wraparound Test VIII	wrapu_4030,wrapl_4030	ring_wrapu.s ring_wrapl.s	0:02	0:04

Test Error Messages

For a common list of error messages that could result from running this test, refer to the appendix on CPU Error Messages.

SPU UNIX Error Messages

For a common list of possible error messages resulting from SPU UNIX, refer to the appendix on SPU UNIX Error Messages.

THIS PAGE INTENTIONALLY LEFT BLANK

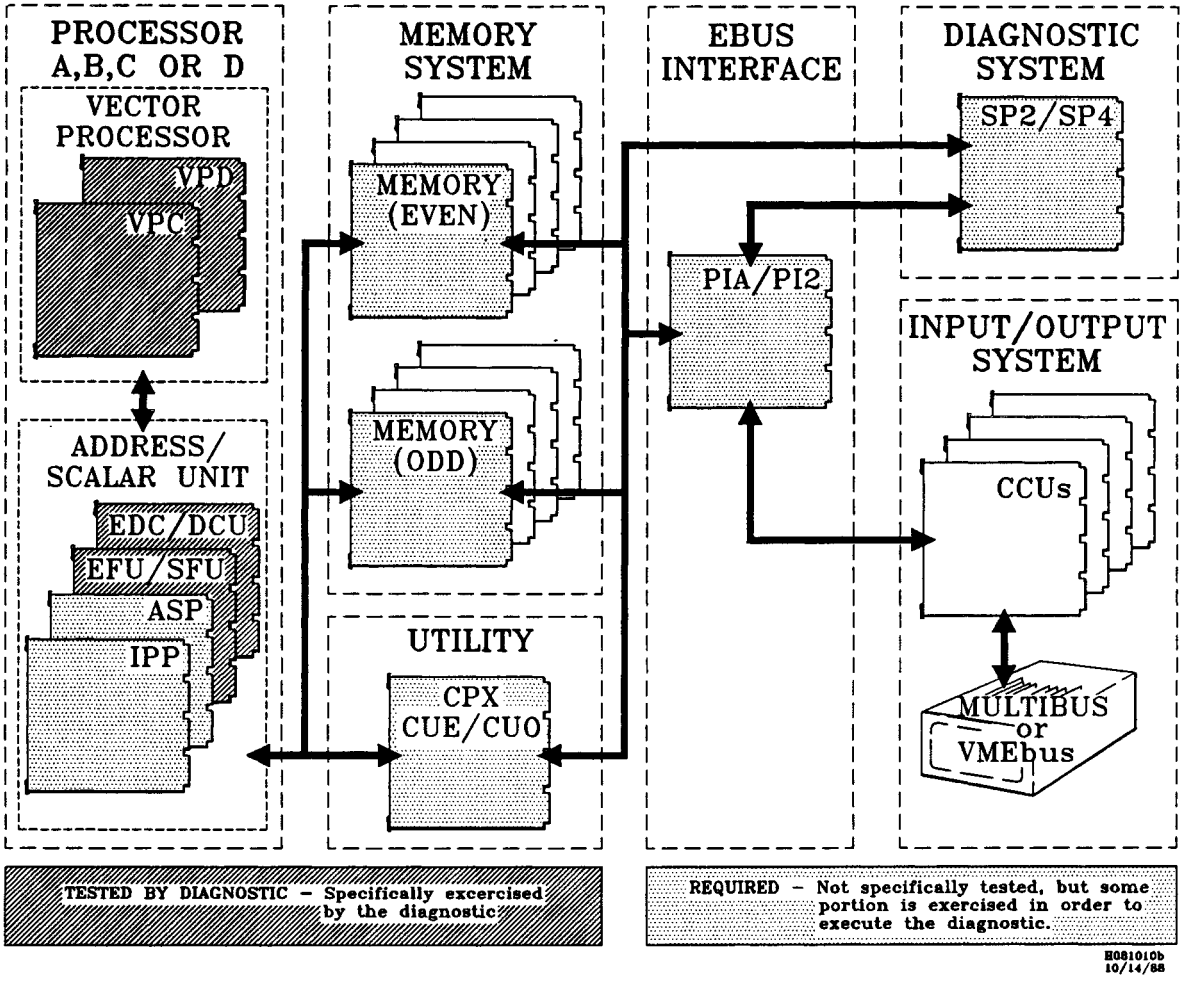
cpu4040

Vector Concurrency Tests

Overview

The *cpu4040* test verifies the Vector Processor Unit operation under all possible combinations of instructions from the three groups listed in Table cpu4040-5 Vector Instruction Groups. Each group of instructions corresponds to one of the three controllers on the Vector Process Unit: load/store, add/logical, and multiply/divide. The *cpu4040* assures that instructions from each set tested can execute concurrently and return the expected results. The following figure show an overall view of what part of the system is being tested and which Field Replaceable Units (FRUs) are required for the test to run:

Figure cpu4040–1, Functional Areas Tested by *cpu4040*



Prerequisites and Required Equipment

The Vector Processor Control (VPC) and Vector Processor Data (VPD) boards must be present, and are tested, when running the *cpu4040* test. The boards listed in the following table must be operational to verify the required boards. No additional equipment is required to run this test.

Table cpu4040–1, Required Functional Boards

BOARD	TEST TO VERIFY
Service Processor (SP2 or SP4)	*spu1000, spu4000*
Memory System	*mem4000*
Instruction Processor Unit (IPP)	*cpu4030*
Address Scalar Processor (ASP)	*cpu4030*
Scalar Function Unit (SFU or EFU)	*cpu4030*
Data Cache Unit (DCU or EDC)	*cpu4030*
Vector Processor Control (VPC)	*cpu4041*
Vector Processor Data (VPD)	*cpu4041*
CPU Utility Board(s) (CPX or CUE/CUO)	*cpx4000*

NOTE

The memory system consists of a minimum of one pair of memory boards (one even and one odd).

Because *cpu4040* checks test results against self-generated expected values, the scalar portion of the CPU is assumed to be operational. Also, enough of the vector unit must be operational to perform scalar-register to vector-register moves. This hardware can be verified by running *cpu4041*.

Test Invocation

To invoke the *cpu4040* test, use the procedure shown in the following figure. All responses in **boldface** are entered by the user. The prompts and responses appear sequentially on the screen, one line at a time. All prompts and responses are shown in one figure for convenience.

CAUTION

The invocation sequence shown in the following figure is the typical invocation sequence. However, the *initall* utility must be executed in some cases. If the system has just been powered up, if *mem4000* was executed with failures, or if *spu4000* was executed, then *initall* must be executed prior to any test execution. Failure to execute *initall* in these circumstances could result in invalid test results.

NOTE

Running the *initall* utility requires two to three minutes to execute depending on if the control stores have been previously loaded. However, it is suggested that *initall* be executed in the event that the state of the system is unknown.

Figure cpu4040–2, Test Invocation Sequence

```
(spu)>  cd /mnt/test
(spu)>  sysreset
(spu)>  dshell

CONVEX DIAGNOSTIC SHELL

:test cpu4040 [-c [class numeral(s)]] [-s [subtest numeral(s)]] [+> filename]
```

> **NOTE**
>
> After entering **dshell**, specific *dshell* parameters may be changed. Refer to the "Dshell and Iscan Overview" chapter of this manual for more information.

Entering only **test cpu4040** executes all *cpu4040* subtests sequentially. Execute a specific class(es) of subtest(s) or one or more individual subtests by using the *-c* or *-s* options, respectively. Detailed information for using these options can be found in the "Dshell and Iscan Overview" chapter of this manual. The [+>*filename*] option allows the test results to be appended to *filename*.

Test Parameter Menu

Once the test is invoked, a test parameter menu prompt is presented allowing selection of default switches. If the test is run with all defaults invoked (answer **y** to the first prompt), no other prompts are provided. If the user answers **n** to the first prompt (run test without default switches), then a series of prompts are presented. The following figure shows all prompts, their possible answers (in brackets [ ]), and their default answers (in parentheses ( )). The prompts and responses in the following figure appear sequentially on the screen, one line at a time. All the prompts and responses are shown in one figure for convenience.

> **NOTE**
>
> If test time is an issue, it is recommended not to run the test with all defaults enabled. Test time can be shortened by using only 1 value of *vl* (128) and one value of *vs* (4 for singles and 33 for doubles).

Figure cpu4040–3, Test Parameter Menu

```
Test 'cpu4040.t'                          Thu Nov 19 00:00:00 1965

                          ENTER TEST PARAMETERS

                  [ ]     Encloses allowed input ranges or values
                  ( )     Encloses the default value
                  ^       Returns to the previous prompt
                  :nn     Returns to the prompt # nn
                  :       Returns to the first unsatisfied prompt
                  :?      Reviews previous entries

 1: Run default switches? [y,n]                                  (y) ->
 2: Cpu to test: [abcd]                                          (a) ->
 3: SW test mode? [y,n]                                          (n) ->
 4: Debugger on? [y,n]                                           (y) ->
 5: Forced Fail Enable? [y,n]                                    (n) ->
 6: Force display of all regs on error? [y,n]                    (y) ->
 7: Display update mode? [f/n]                                   (f) ->
 8: Test display mode? [s,l]                                     (s) ->
 9: Dcache Enabled? [y,n]                                        (y) ->
10: Continuous faulting enabled? [y,n]                           (n) ->
11: Single faults enabled? [y,n]                                 (n) ->
Current group indexes:
12: Group:0[0-18]                                                (0) ->
13: Group:1[0-33]                                                (0) ->
14: Group:2[0-49]                                                (0) ->
15: Continuous loop enabled? [y,n]                               (n) ->
16: Enter number of vl values to check [0-30]                    (3) ->
Current set of vl values:
17: Vl:0 [0-128]                                                 (32) ->
18: Vl:1 [0-128]                                                 (64) ->
19: Vl:2 [0-128]                                                 (128) ->
20: Enter number of vs values to check [0-4]                     (2) ->
Current set of vs for words and singles:
21: Vs:0 [0-33]                                                  (4) ->
22: Vs:1 [0-33]                                                  (33) ->
Current set of vs for longs and doubles:
23: Vs:0 [0-33]                                                  (8) ->
24: Vs:1 [0-33]                                                  (33) ->
25: Enter OK, or :NN to return to question NN [OK]
                                                                 (OK) ->
```

Prompt Explanations

A description of the meaning of each prompt follows:

```
Run default switches? [y/n]                    (y) ->
```

If the user responds with **y** or <**CR**>, no additional test parameter prompts are displayed, and testing begins. However, if a negative response is supplied, additional test parameter prompts are displayed allowing modification of the default selections. Only enter the **n** response to restart a previous testing session at a specified point or to change any of the default parameters.

The following prompts are displayed and answered only if the first prompt is answered with **n**:

```
Cpu to test: [abcd]                    (abcd) ->
```

This prompt allows selection of the CPU to be used in the test. The possible selections, represented by **abcd**, will consist of all available CPUs. The default, **abcd**, will consist of all available CPUs.

```
SW test mode? [y,n]                    (n) ->
```

If answered with **y**, permutations of the selected instructions are disabled and the test will execute six times faster. *Only* answer with a **y** when performing verification of the test program.

```
Debugger on? [y,n]                     (y) ->
```

If answered with **y**, the program enters a *debug* mode whenever a test fails (refer to "*debugger* Description" in this section for more information). Enter **n** to bypass the *debugger* upon failure or to run this test under a script in the background.

```
Forced Fail Enable? [y,n]              (n) ->
```

If forced fail is enabled by entering **y**, the CPU acts as if a fail occurs at the end of each instruction group tested. This is normally used in conjunction with the *debugger* when debugging instruction sequences. Enabling forced fail allows access to the *debugger* even if all hardware components are working correctly (refer to the "*debugger* Description" in this section for more information).

```
Force display of all regs on error? [y,n]          (n) ->
```

Answering **y** to this prompt displays failing vector registers automatically on any failing instruction sequence. Answering **n** causes the test (upon failure) to stop at each failing vector register and to display the following prompts that must be answered:

```
Display initial?
        expected?
          actual?
```

For the above prompts, *initial* is the initialized condition of the register(s), *expected* is the expected value of the register(s), and *actual* is the actual value of the register(s) at the failure.

```
Display update mode [f,n]              (f) ->
```

Answering **n** (*normal*) to this prompt forces the CPU to delay further testing while instruction information is sent to the screen. If **n** is entered, the time to execute this test is extended. A response of **n** ensures that the screen output concurrently shows what the test is doing at all times. The **f** default (*fast*) allows the CPU to continue processing at the same time information is going to the screen. Since testing is not delayed in the **f** (*fast*) mode, if the CPU halts due to a failure, the screen output may not represent the current instruction information that the CPU was testing at the time of the failure.

```
Test display mode? [s,l]                    (s) ->
```

If the **s** *(short)* mode is selected, only the test *index numbers* are displayed along with the permutation that is currently being tested (see the section on "Instruction Permutations" and "Test Method" in this chapter for more information). For example:

```
Groups: X     Y     Z     Perm: N
```

In the above example:

- X represents the test *index number* for the load/store instruction performed
- Y represents the test *index number* for the add/logical instruction performed
- Z represents the test *index number* for the multiply/divide instruction performed

This display is written over itself on a CRT or scrolled on a hard copy terminal. If the **l** (*long*) mode is selected from this prompt, the instructions as well as the corresponding test *index numbers* of the three instructions being tested appear on the screen with the permutation number. The *long* display is useful for monitoring the progression of the test through the various instructions (refer to the section on "Instruction Permutations" and "Test Method" in this chapter for more information). For example:

```
Groups: X     Y     Z     Perm: N

Instructions:
             sum.w   v4
             mov     s0,s1,v0
             prod.l v1
```

In the above example:

- X represents the test *index number* for the load/store instruction performed
- Y represents the test *index number* for the add/logical instruction performed
- Z represents the test *index number* for the multiply/divide instruction performed

This display is scrolled on either a CRT or a hard copy terminal.

```
Dcache Enabled? [y,n]                       (y) ->
```

The Dcache is normally enabled; however, if it is suspected broken, it can be disabled by entering **n** at this prompt.

```
Continuous faulting enabled? [y,n]        (n) ->
```

To enable the continuous fault diagnostic mode of the CPU prior to the execution of the selected instructions, answer this prompt with **y**. As soon as the three instructions have executed, the mode is disabled. Since this mode forces a page fault to occur on every memory access, a check of the faulting mechanisms is performed, verifying each instruction's fault operation. A more detailed explanation of faulting is provided in the glossary appendix of this manual.

```
Single faults enabled? [y,n]              (n) ->
```

If this prompt is answered with **y**, all data pages are made nonresident at the start of each instruction. A fault is taken every time the test references a new page. A more detailed explanation of faulting is provided in the glossary appendix of this manual.

```
Current group indexes:
                Group:0[0-19]             (0)>
                Group:1[0-34]             (0)>
                Group:2[0-50]             (0)>
```

Enter the test *index number* of the desired instruction set after the prompt for each group. This option is useful to start the test at some point other than the beginning. If *cpu4040* has been interrupted, or if it has failed, the user can choose an instruction from each group (refer to Table cpu4040-5) to indicate the starting point for testing. If the test is running for the first time and all instructions are to be tested, enter the default of **0**.

```
Continuous loop enabled? [y,n]            (n)>
```

Respond with **y** whenever a specific set of instructions must be tested continuously. The initialization and execution of three selected instructions are *looped* or repeated until stopped with a **Ctrl-C**. If the user has enabled *debug* mode, the program goes into the *debug* mode as soon as the looping starts. The looping can then be stopped and the test resumed, if desired (refer to the "*debugger* Description" in this section for more information).

When a **y** is entered, the following four prompts are displayed and must be answered:

```
Enter permute number: [0-5]               (0) ->
```

After the three chosen instructions are permuted, the test assigns each permutation a number, starting at zero. If the desired permute number is entered, the instruction sequence is duplicated with the expected ordering.

```
Enter value for vl: [0-128]               (128) ->
```

This is the value that will be loaded into the Vector Length (*vl*) register prior to execution.

```
Enter vs for word/single instructions: [0-33]  (4) ->
```

This is the value that will be loaded into the *vs* register prior to execution if the instruction sequence uses only word and single operands.

```
Enter vs for long/double instructions: [0-33]  (8) ->
```

This is the value that will be loaded into the *vs* register prior to execution if the instruction sequence uses any double or long operands.

The following prompts only appear if **n** is answered to the `Continuous loop enabled` prompt:

```
Enter number of vl values to check: [0-30]  (3) ->
Current set of vl values:
            Vl:0 [0-128]                (32) ->
            Vl:1 [0-128]                (64) ->
            Vl:2 [0-128]                (128) ->
```

This test is executed with various values of *Vl*, which are stored in an array. The length of this array must be specified. In this example, the default of three was used which resulted in three additional prompts. If 30 is entered for the length of the array, 30 prompts would be displayed for the user to answer. The values of *Vl* are displayed one at a time. The user can change the value of any *Vl* element after it appears.

```
Enter number of vs values to check: [0-4]  (2) ->
Current set of vs for words and singles:
            Vs:0 [0-33]                 (4) ->
            Vs:1 [0-33]                 (33) ->
```

This test is executed with various values of *Vs*, which are stored in an array. The length of this array must be specified. In this example, the default of **2** was used which resulted in two prompts. If **4** were used, four prompts would be displayed. The values of *Vs* are displayed one at a time. The value of any *Vs* element can be changed after it appears. This is the vector stride that is used when the load store operand length is 4 bytes.

```
Current set of vs for longs and doubles:
            Vs:0 [0-33]                 (8) ->
            Vs:1 [0-33]                 (33) ->
```

This option displays the values of *Vs* that are used when an instruction sequence contains any double or long operand. The value of any *Vs* element can be changed after it appears. This is the vector stride that is used when the load store operand length is 8 bytes.

The following prompt is always supplied if any of the defaults have been changed:

```
Enter OK, or :NN to return to question NN [OK]
                                                (OK) ->
```

If the **OK** default is selected, test execution begins. A particular prompt can be changed by entering its number here and the program would return to that prompt.

When all prompts have been answered, a test parameter summary echos the prompts that have been answered. The following figure displays a sample test parameter summary. The actual summary varies depending on the answers to the prompts.

Figure cpu4040–4, Sample Test Parameter Summary

```
                    TEST PARAMETER SUMMARY

Run default switches?                              : n
Cpu to test:                                       : a
SW test mode?                                      : n
Debugger on?                                       : y
Forced Fail Enable?                                : n
Force display of all regs on error?                : n
Display update mode?                               : f
Test display mode?                                 : s
Dcache Enabled?                                    : y
Continuous faulting enabled?                       : n
Single faults enabled?                             : n
Current group indexes:
Group:0                                            : 0
Group:1                                            : 0
Group:2                                            : 0
Continuous loop enabled?                           : n
Enter number of vl values to check                 : 3
Current set of vl values:
Vl:0                                               : 32
Vl:1                                               : 64
Vl:2                                               : 128
Enter number of vs values to check                 : 2
Current set of vs for words and singles:
Vs:0                                               : 4
Vs:1                                               : 33
Current set of vs for longs and doubles:
Vs:0                                               : 8
Vs:1                                               : 33
Enter OK, or :NN to return to question NN          : OK
```

Hardware Initialization Sequence

After the last prompt is entered, and before test code execution, the following events occur:

> **NOTE**
>
> The first two events are accomplished at the initial start of the *cpu4040* test. The remaining events are accomplished when each subtest is initialized.

- Each module's Page Table Entries (PTEs) are set up in main memory. PTEs begin at the last available address in main memory and work toward low memory.
- Each module is loaded into main memory following the logical to physical mapping described by the page tables. The exact logical to physical mapping that the modules are loaded into depends upon which segment of execution is selected.

- The system is initialized (the memory system, CPX or CUE/CUO, PIA or PI2, and the CPU boards are reset).
- Parity is initialized in the scalar processor by sending the scalar processor into a microcode routine and issuing clocks.
- Scratch RAM is loaded with various values to control the execution of the test.
- The initial Program Counter (PC) is loaded into register T0 and the initial Program Status Word (PSW) is loaded into register T2.
- Control store is initialized to the cold-start location.
- Clocks are turned on.

Memory Allocation

Immediately before test code execution, a current memory allocation screen is displayed. The following figure is an example of the current memory allocation screen. The physical and logical addresses shown, as well as the filenames, are only representative; the actual addresses will vary depending on installed memory & file sizes.

The following list explains what each column the current memory allocation screen depicts:

- **First Column** — File Number - useful in conjunction with the mm(1d) utility.
- **Second Column** — Physical memory addresses where the specified file is loaded
- **Third Column** — Process identification
- **Fourth Column** — File name (actual path is *./filename* or /mnt/test/CPU/*filename*). The entries `pte1`, `pte2`, and `ptet` are not actual files but are instead indications of the page tables.
- **Fifth Column** — Logical starting address of the specified file

Figure cpu4040–5, Current Memory Allocation Screen

```
                    Current Memory Allocation

File No.  Physical  Address Pid File Name                    Logical Offset
      1   00000000-00027fff   0 p0r0_4040                   00000000
      2   00028000-000b1fff   0 cpu4040.rnn                 00022000
      1   000b2000-000d9fff   1 p0r0_4040                   00000000
      2   000da000-00163fff   1 cpu4040.rnn                 00022000
   ----   03ff7000-03ff9fff   1 ptet                        NA
   ----   03ffa000-03ffafff   1 pte2                        NA
   ----   03ffb000-03ffdfff   0 ptet                        NA
   ----   03ffe000-03ffefff   0 pte2                        NA
   ----   03fffc00-3fffffff   1 pte1                        NA
```

debugger Description

The *cpu4040* test provides an internal *debugger* that can be used as a system troubleshooting aid. The *debugger* displays the contents of registers and memory that are not normally displayed when a test fails.

When responding to the following prompt with **y**, the internal *debugger* is automatically invoked whenever a subtest fails.

```
Debugger on [y,n]                    (y) -> y
```

When invoked, the *debugger* displays a `dbug>` prompt.

debugger Commands

When the *debugger* utility is invoked, an online list of available commands is displayed by entering **?**. The *debugger* commands are listed in the following table:

Table cpu4040–2, *debugger* Commands

COMMAND	FUNCTION
c	Compares vector registers
d	Displays memory
f	Displays failing instructions
l	Starts looping on failed instructions
q	Quits (exits) debug mode and continues testing
r	Displays registers
s	Stops looping on failed instructions
?	Displays the *help* file

Data Type and Register Operands

Use the register data type and the register mnemonic operands listed in the following table with the *debugger* commands:

Table cpu4040–3, Data Type and Register Operands

DATA TYPE OPERANDS	MEANING	REGISTER OPERANDS	MEANING
a	Actual data	a	Address register
e	Expected data	psw	Process status register
i	Initialization data	s	Scalar register
		v	Vector register
		vl	Vector length register
		vm	Vector merge register
		vs	Vector stride register

debugger Command Descriptions

The following paragraphs briefly describe the use of each of the *debugger* commands:

Compare Vector Registers

c [*Register_Type Register_Number: Data_Operand*] [*Register_Type Register_Number: Data_Operand*]

or:

c [*Register_Operand: Data_Operand*] [*Register_Operand: Data_Operand*]

These commands allow comparison of the selected data types for the two registers specified where:

- [*Register_Number*] represents the number of the register desired which can range from **0** through **7**.
- [*Data_Operand*] represents the type of data operand desired. Possible responses include:
  - **a** — Actual data
  - **e** — Expected data
  - **i** — Initialization data
- [*Register_Operand*] represents the desired type of register which does not require a *Register_Number*. Possible responses include:
  - **psw** — Process status register
  - **vl** — Vector length register
  - **vm** — Vector merge register
  - **vs** — Vector stride register
- [*Register_Type*] — represents the desired type of register which requires a *Register_Number* to be supplied.
  - **a** — Address register
  - **s** — Scalar register
  - **v** — Vector register

Display Memory

d [*start_addr*] [*stop_addr*]

This command displays the contents of main memory from the start address (*start_addr*) to the ending address (*stop_addr*). Possible values for *start_addr* and *stop_addr* range from **00000000** to **FFFFFFFF** provided that the *start_addr* is less than the *stop_addr*.

Display Failing Instructions

f

Use this command to redisplay instructions that failed. If other *debug* commands have scrolled the failed instructions off the screen, this command can be used to redisplay them.

Loop On Failing Instructions

l

Use this command to start the CPU looping on the failed instructions. Only the **s** (*stop*) and **q** (*quit*) commands are operational during looping.

Quit

q

Use this command to quit (exit) debug mode and continue testing.

> **NOTE**
>
> Make sure that the *log -s* command is used at the *dshell* prompt to log multiple failures, otherwise, this command will exit the test and return to the *dshell* prompt. For more information on the *log -s* command, refer to the "Dshell and Iscan Overview" chapter of this manual.

Display Registers

r [*Data_Operand*] [*Register_Operand*]

This command displays the selected data for the specified register. The user can specify multiple registers on the command line where:

- [*Data_Operand*] represents the type of data operand desired. Possible responses include:
  - **a** — Actual data
  - **e** — Expected data
  - **i** — Initialization data
- [*Register_Operand*] represents the desired type of register. Possible responses include:
  - **a** — Address register, which must be followed by a register number that can range from **0** to **7**.
  - **psw** — Process status register
    s — Scalar register, which must be followed by a register number that can range from **0** to **7**.
  - **v** — Vector register, which must be followed by a register number that can range from **0** to **7**.
  - **vl** — Vector length register
  - **vm** — Vector merge register
  - **vs** — Vector stride register

Stop looping

s

Using this command terminates looping initiated via the *l* command and returns to the debugger prompt.

Class Descriptions

This test only has one class of subtests, Class 1. Class 1 subtests verify operation of the Vector Processor Unit.

Class 1 Subtests

The instruction sequences used in *cpu4040* fall into two basic groups according to how they interact under vector processing:

Table cpu4040–4, Class 1 Subtests

SUBTEST	CLASS	DESCRIPTION
10	1	Shared source-destination regs not allowed
20	1	Shared source-destination regs allowed

Test Method

An instruction is chosen from each column in the following table based on the current index number.

> **NOTE**
>
> For information on each instruction's meaning in the following table, refer to the "Opcodes Sorted by Name" appendix in this manual or the *CONVEX Architecture Reference*.

Table cpu4040–5, Vector Instruction Groups

INDEX	LOAD/STORE	ADD/LOGICAL	MULTIPLY/DIVIDE
0	*mov s0,s1,v0*	*sum.w v4*	*prod.w v0*
1	*mov v0,s0,s1*	*add.w v0,v1,v2*	*prod.l v1*
2	*mov a1,vl*	*add.w v0,v1,v4*	*mul.w v0,v1,v4*
3	*mov s1,s2,vm*	*add.w v0,v4,v4*	*mul.w v0,v1,v2*
4	*mov s1,vm,s2*	*add.w v1,v5,v6*	*mul.w v0,v4,v4*
5	*ldvi.w v5,v6*	*add.w v2,v3,v7*	*mul.w v1,v5,v2*
6	*ldvi.w v3,v7*	*add.w v2,v2,v6*	*mul.w v1,v6,v2*
7	*ldvi.l v0,v1*	*sum.l v5*	*mul.w v3,v3,v7*
8	*ldvi.l v0,v4*	*not v1,v5*	*mul.d v0,v1,v4*
9	*stvi.w s0,v0*	*not v0,v4*	*mul.d v0,v1,v2*
10	*stvi.l s1,v0*	*shf s4,v0*	*mul.d v0,v4,v4*
11	*stvi.w v0,v1*	*plc.t v0,v1*	*mul.d v3,v7,v1*
12	*stvi.w v1,v5*	*plc.t v2,v6*	*mul.d v1,v6,v2*
13	*stvi.l v2,v3*	*eq.w s4,v0*	*mul.d v1,v1,v5*
14	*stvi.l v3v7*	*lt.l s4,v3*	*div.w v0,v1,v4*
15	*ld.w (a2),v0*	*lt.w s4,v6*	*div.w v0,v1,v2*
16	*ld.l (a2),v0*	*eq.l s4,v4*	*div.w v0,v4,v4*
17	*st.w v0,(a2)*	*plc.t vm,s4*	*div.w v2,v6,v3*
18	*st.l v6,(a2)*	*add.l v0,v1,v2*	*div.w v3,v7,v0*
19		*add.l v0,v1,v4*	*div.w v1,v1,v5*
20		*add.l v0,v4,v4*	*div.l v0,v1,v4*
21		*add.l v3,v7,v5*	*div.l v0,v1,v2*
22		*add.l v7,v5,v1*	*div.l v0,v4,v4*
23		*add.l v2,v2,v6*	*div.l v1,v5,v2*
24		*eq.w v6,v7*	*div.l v2,v6,v7*
25		*eq.w v3,v7*	*div.l v3,v3,v7*
26		*eq.l v5,v6*	*mul.w v0,s5,v1*
27		*eq.l v1,v5*	*mul.w v0,s5,v4*
28		*max.w v0*	*mul.d v0,s5,v5*
29		*max.l v7*	*mul.d v2,s5,v6*
30		*add.w v0,s3,v1*	*div.w v0,s5,v1*
31		*add.w v2,s3,v6*	*div.w v0,s5,v4*
32		*add.l v6,s3,v7*	*div.l v0,s5,v5*
33		*add.l v1,s3,v5*	*div.l v2,s5,v6*
34			*merg.t v0,s1,v1*
35			*merg.t v0,s1,v4*
36			*mask.t v0,v1,v2*
37			*mask.t v0,v1,v4*
38			*mask.t v0,v0,v4*
39			*mask.t v0,v4,v3*
40			*mask.t v2,v0,v4*
41			*mask.t v0,v4,v4*
42			*mask.t v5,s1,v7*
43			*mask.t v2,s1,v6*
44			*cprs.t v0,v1*
45			*cprs.t v3,v7*
46			*merg.t v0,v1,v2*
47			*merg.t v0,v1,v4*
48			*merg.t v1,v5,v7*
49			*merg.t v3,v2,v6*

The current index number is determined by the test instruction's location in the test table. The selected instructions are permuted, and the registers are checked to see whether the instructions *chain*, which means that the destination register of one instruction is used as the source register of another. Depending on the subtest being run, the instructions are either tested or discarded (refer to individual subtest descriptions in this section).

Next, the registers in each of the register spaces are initialized, and the emulation routines are called to emulate the instructions under test. The registers are loaded with the initialized values, and the instructions are executed. The registers are then saved, and the *compare* routine compares the emulation and execution results. If the registers are equal, the test passes and it continues to the next permutation. But if they are not equal the test fails and the CPU halts.

When all six permutations have been checked, the table indexes are changed, and the next group of three instructions are tested. After all instructions have been checked, *vl* is changed, and all instructions are checked again. After all values of *vl* have been tested with all combinations of instructions, the value of *vs* is changed. The *cpu4040* tests all selected values of *vl*, all selected values of *vs*, and all possible instruction combinations before finishing.

Although all registers that are operands of the instructions being executed are initialized by a random number before use, these numbers are generated from a seed that is set at the beginning of each test sequence. This means that each permutation of instructions will have the registers initialized to different values. However, when an instruction sequence is looped on, the registers are always looped from the same initialized value. Of course, registers that contain the indexes in a vector of indexes instruction are not randomly initialized, and registers that contain the address for a vector load or store operation are not randomly initialized. In addition, vector registers used in *prod.l*, *prod.w*, and *mul.w vi,vj,vk* instructions have the input values limited to prevent overflow. These registers also are never initialized to zero.

Instruction Permutations

Generating a test sequence causes an instruction to be chosen from each column. These three instructions are then permuted and executed in each of the six possible orderings. For example, the following three instruction sets:

```
mov s0, s1, v0
sum.w v4
prod.w v0
```

permute into six different sets of instruction tests:

```
mov s0, s1, v0
sum.w v4
prod.w v0

mov s0, s1, v0
prod.w v0
sum.w v4

sum.w v4
prod.w v0
mov s0, s1, v0

prod.w v0
mov s0, s1, v0
sum.w v4
```

```
sum.w v4
mov s0, s1, v0
prod.w v0

prod.w v0
sum.w v4
mov s0, s1, v0
```

Under nominal conditions, the time it takes to execute *cpu4040* depends on several variables. A rough estimate of execution time (in minutes) can be obtained by the following formula:

$$I \times J \times K \times X \times Y \times 6 \times .005 \div 60 = \text{Minutes To Run Test}$$

The following table defines the factors in the preceding formula.

FACTORS	DEFINITION
I	Number of load/store instructions
J	Number of add/logical instructions
K	Number of multiply/divide instructions
X	Number of vl values
Y	Number of vs values
6	Number of permutations
.005	Seconds
60	Seconds/Minute

The number of instruction sequences tested is quite large. Besides checking the different combinations of instructions, each instruction sequence can be checked with different values of *vl* and *vs*. From the Vector Instruction Groups table, there are currently 18 instructions tested from the load/store column, 33 from the add/logic column, and 49 from the multiply/divide column. The number of instructions executed determines the amount of time required to execute this test. Assuming only 2 values of *vs*, and 3 values of *vl*:

$$18 \times 33 \times 49 \times 2 \times 3 \times 6 \times .005 \div 60 = 87.318 \text{ Minutes}$$

Nonchaining Instructions

Subtest 10 verifies instructions that do not chain together which means that the destination register of one instruction is *not* used as the source register of another. If subtest 10 generates a sequence of instructions that chain together, that combination is discarded and the test continues to the next instruction sequence in the table.

Chaining Instructions

Subtest 20 verifies instructions that chain together which means that the destination register of one instruction is used as the source register of another. If subtest 20 generates a sequence of instructions that do not chain together, that combination is discarded and the test continues to the next instruction sequence in the table.

> **NOTE**
>
> Almost all failures which occur when running the unchained instruction permutations (Subtest 10) will also be uncovered by running the chained instruction permutations (Subtest 20). However, if the failure is related to the chaining logic, Subtest 10 will pass while Subtest 20 will fail. If test time is an issue, it should be noted that the test time can significantly be reduced by only running Subtest 20 without significant reduction of fault coverage.

Test Error Messages

For a common list of error messages that could result from running this test, refer to the appendix on CPU Error Messages.

SPU UNIX Error Messages

For a common list of possible error messages resulting from SPU UNIX, refer to the appendix on SPU UNIX Error Messages.

THIS PAGE INTENTIONALLY LEFT BLANK

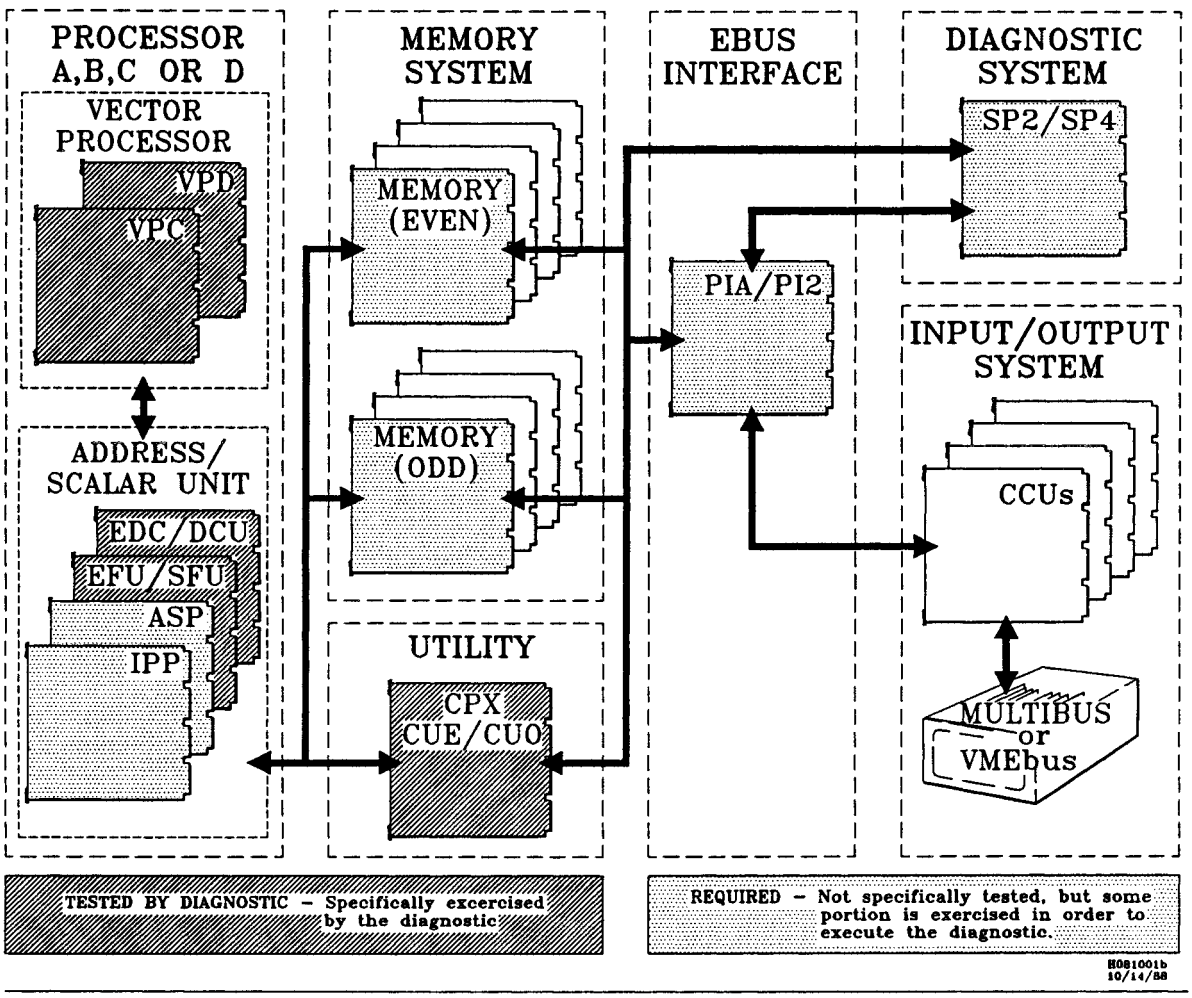
cpu4041

Vector Instruction Tests

Overview

The *cpu4041* test is a group of vector instruction tests used to verify the operation of the vector unit and its interfaces to the other C200 Series subsystems. The verification is performed by exercising each vector/vector and scalar/vector instruction while varying all of the parameters on which the instructions depend. The following figure shows an overall view of what part of the system is being tested and which Field Replaceable Units (FRUs) are required for the test to run:

Figure cpu4041–1, Functional Areas Tested by *cpu4041*



Prerequisites and Required Equipment

The Vector Processor Control (VPC) and Vector Processor Data (VPD) boards must be present, and are tested, when running the *cpu4041* test. The boards listed in the following table must be operational to verify the required boards. No additional equipment is required to run this test.

Table cpu4041–1, Required Functional Boards

BOARD	TEST TO VERIFY
Service Processor (SP2 or SP4)	*spu1000, spu4000*
PBUS Interface Adapter (PIA or PI2)	*pia4000* or *pi2_4000*
Memory System	*mem4000*
Instruction Processor Unit (IPP)	*cpu4030*
Address Scalar Processor (ASP)	*cpu4030*
Scalar Function Unit (SFU or EFU)	*cpu4030*
Data Cache Unit (DCU or EDC)	*cpu4030*
CPU Utility Board(s) (CPX or CUE/CUO)	*cpx4000*

NOTE

The memory system consists of a minimum of one pair of memory boards (one even and one odd).

Test Invocation

To invoke the *cpu4041* test, use the procedure shown in the following figure. All responses in **boldface** are entered by the user. The prompts and responses appear sequentially on the screen, one line at a time. All prompts and responses are shown in one figure for convenience.

CAUTION

The invocation sequence shown in the following figure is the typical invocation sequence. However, the *initall* utility must be executed in some cases. If the system has just been powered up, if *mem4000* was executed with failures, or if *spu4000* was executed, then *initall* must be executed prior to any test execution. Failure to execute *initall* in these circumstances could result in invalid test results.

NOTE

Running the *initall* utility requires two to three minutes to execute depending on if the control stores have been previously loaded. However, it is suggested that *initall* be executed in the event that the state of the system is unknown.

Figure cpu4041–2, Test Invocation Sequence

```
(spu)>  cd /mnt/test
(spu)>  sysreset
(spu)>  dshell

CONVEX DIAGNOSTIC SHELL

:test cpu4041 [-c [class numeral(s)]] [-s [subtest numeral(s)]] [+> filename]
```

NOTE

After entering **dshell**, specific *dshell* parameters may be changed. Refer to the "Dshell and Iscan Overview" chapter of this manual for more information.

Entering only **test cpu4041** executes all *cpu4041* subtests sequentially. To execute a specific class(es) of subtest(s) or one or more individual subtests, enter the *-c* or *-s* options, respectively. Detailed information for using these options can be found in the "Dshell and Iscan Overview" chapter of this manual. The [+>*filename*] option allows the test results to be appended to *filename*.

Test Parameter Menu

Once the test is invoked, a test parameter menu is presented allowing selection of default switches. If the test is run with all defaults invoked (answer **y** to the first prompt), no other prompts are provided. If the user answers **n** to the first prompt (run test without default switches), then a series of prompts are presented. The following figure shows all prompts, their possible answers (in brackets [ ]), and their default answers (in parenthesis ( )). The prompts and responses in the following figure appear sequentially on the screen, one line at a time. All the prompts and responses are shown in one figure for convenience.

Figure cpu4041–3, *cpu4041* Test Parameter Menu

```
Test 'cpu4041.t'                          Thu Nov 19 00:00:00 1965

                          ENTER TEST PARAMETERS

              [ ]    Encloses allowed input ranges or values
              ( )    Encloses the default value
              ^       Returns to the previous prompt
              :nn     Returns to the prompt # nn
              :       Returns to the first unsatisfied prompt
              :?      Reviews previous entries

1:  Run default switches? [y,n]                                  (y) ->
2:  CPUs to test: [ABCD]                                         (ABCD) ->
3:  Parallel Test Execution? [y,n]                               (n) ->
4:  Forced Faulting Enabled? [y,n]                               (n) ->
5:  Fault on Instruction Fetches? [y,n]                          (n) ->
6:  Sequential Execution? [y,n]                                  (n) ->
7:  Timeout Scale Factor Enabled? [1-100]                        (1) ->
8:  Dcache Enabled? [y,n]                                        (y) ->
9:  Segment of Execution? [0-7]                                  (0) ->
10: Loop Enabled? [y,n]                                          (n) ->
11: Chained Execution Mode? [y,n]                                (n) ->
12: Hard Errors Enabled? [y,n]                                   (y) ->
13: Number of vl values to test(0..vl)? [0-128]                  (128) ->
14: Load CPU Code? [y,n]                                         (y) ->
```

NOTE

In the second in the previous figure, ABCD represents all available CPUs. Only the available CPUs within the machine under test will be displayed as possible choices.

Prompt Explanations

A description of what each prompt means follows:

```
Run default switches? [y/n]                                      (y) ->
```

If a response of **y** or <**CR**> is entered, no additional test parameter prompts are displayed, and testing begins. However, if a negative response is supplied, additional test parameter prompts are displayed allowing changes to the default selections. Only enter the **n** response to restart a previous testing session at a specified point or to change any of the default parameters.

The following prompts are displayed and answered only if the first prompt is answered with **n**:

```
Cpus to test: [ABCD]                                             (ABCD) ->
```

This prompt allows selection of the CPU(s) to be used in the test. The possible selections,

represented by ABCD, will consist of all available CPUs. The default, ABCD, will consist of all available CPUs.

```
Parallel Test Execution? [y,n]                    (n) ->
```

This prompt allows parallel test execution to be enabled or disabled. This prompt is only displayed if the `CPUs to test:` prompt is answered with multiple CPUs.

```
Forced Faulting Enabled? [y,n]                    (n) ->
```

If answered with **y**, normal force faulting occurs only on data references. The system forces a nonresident data exception to occur on every data reference. For a more detailed explanation of forced faulting, refer to the glossary appendix of this manual.

```
Fault on Instruction Fetches? [y,n]               (n) ->
```

If answered with **y**, force faulting occurs on instruction fetches in addition to data references. This prompt is displayed only if the previous prompt is answered with **y**.

```
Sequential Execution? [y,n]                       (n) ->
```

If set by entering **y**, the sequential bit in the Processor Status Word (PSW) is set to forced sequential execution mode.

```
Timeout Scale Factor Enabled [1-100]              (1) ->
```

This prompt allows adjustment of the timeout factor. The normal timeout factor is multiplied by the number selected to increase the timeout factor. For example, if **5** is entered, the normal timeout factor is multiplied by 5 and it takes the test five times as long to timeout.

```
Dcache Enabled? [y,n]                             (y) ->
```

The Dcache is normally enabled; however, if it is suspected broken, it may be disabled by entering **n** at this prompt.

```
Segment of Execution? [0-7]                       (0) ->
```

This prompt is only displayed if forced faulting in not enabled. The segment of execution is contained in bits<31..29> of the Program Counter (PC). If **0** is entered, then bits<31..29> of the PC are 000 and the test is run in ring zero. If **1** is entered, then bits<31..29> of the PC are 001 and the test is run in ring one. If **2** is entered, then bits<31..29> of the PC are 010 and the test is run in ring two. If **3** is entered, then bits<31..29> of the PC are 011 and the test is run in ring three. If **4**, **5**, **6**, or **7** is entered, then bits<31..29> of the PC are 100, 101, 110, and 111, respectively, and the test is run in ring four. Refer to the *CONVEX Architecture Reference* for more information about the meaning of rings in the machine architecture.

```
Chained Execution Mode? [y,n]                     (n) ->
```

With this option enabled, the test is executed in chained mode which causes the CPU to perform subtest sequencing. The Service Processor is unaware of the action of the CPU unless a subtest fails or unless all of the subtests pass. If this option is enabled, test execution time will be greatly reduced. However, the only information printed to the console upon completion or failure

(regardless of the cause) is the message, "Subtest 1 passed," or "Subtest 1 failed." Also, this option can not be enabled if the **-c** or the **-s** options were used in the invocation procedure.

```
Loop Enabled? [y,n]                                    (n) ->
```

If **y** is entered, hard looping on a specific subtest is enabled. When looping is enabled, the halt instruction of the specified subtest is changed to branch back to the beginning of the subtest. This puts the subtest into an infinite loop which can be interrupted by entering **Ctrl-C**.

```
Hard Errors Enabled? [y,n]                             (y) ->
```

If this option is enabled and a parity error occurs, the clocks are stopped, and the test fails. If this option is disabled, parity errors and other sources of hard errors go undetected. It is recommended that hard errors normally be enabled.

```
Number of vl values to test (0..vl)? [0..128]          (128) ->
```

Each instruction in this test is executed twice, first with a Vector Length (vl) value of 128, and second with a vl value ranging from 0 to the vl count. The vl count will be 0 through 128 if the default of this prompt is selected. However, if time is an issue, enable forced faulting and select a vl count of 16 in response to this prompt. This allows decreased subtest execution times without a significant decrease in test coverage.

```
Load CPU Code? [y,n]                                   (y) ->
```

If the CPU code for this test is already in memory, the user can enter **n** for this prompt and the code is not reloaded (thus saving time).

When all prompts have been answered, the user is shown a test parameter summary displaying the prompts that have been answered. The following figure displays a sample test parameter summary. The actual summary varies according to answers given to the prompts.

Figure cpu4041–4, Sample Test Parameter Summary

```
                    TEST PARAMETER SUMMARY

Run default switches?                          :  n
Cpus to test:                                  :  ab
Parallel Test Execution?                       :  n
Forced Faulting Enabled?                       :  y
Fault on Instruction Fetches?                  :  y
Sequential Execution?                          :  y
Timeout Scale Factor Enabled?                  :  1
Dcache Enabled?                                :  y
Segment of Execution?                          :  0
Chained Execution Mode?                        :  n
Loop Enabled?                                  :  n
Hard Errors Enabled?                           :  y
Number of vl values to test(0..vl)?            :  128
Load CPU Code?                                 :  y
```

Hardware Initialization Sequence

After the last prompt is entered by the user (and before test code execution) the following events occur:

> **NOTE**
>
> The first two events are accomplished at the initial start of the test. The remaining events are accomplished when each subtest is initialized.

- For each CPU, each module's page table entries are set up in main memory. Page table entries begin at the last available address in main memory and work toward low memory.
- For each CPU, each module is loaded into main memory following the logical to physical mapping described by the page tables. The exact logical to physical mapping that the modules are loaded into depends upon which segment of execution is selected by the user.
- The system is initialized (the memory system, CPX or CUE/CUO, PIA or PI2, and the CPU boards are reset).
- For each CPU, parity is initialized in the scalar processor by sending the scalar processor into a microcode routine and issuing clocks.
- Each CPU's scratch RAM is loaded with various values to control the execution of the test.
- For each CPU, the initial Program Counter (PC) is loaded into register T0 and the initial Program Status Word (PSW) is loaded into register T2.

- For each CPU, control store is initialized to the cold-start location.
- Clocks are turned on to the processor selected. If parallel execution is selected, each CPU's clocks are turned on at one time. If sequential execution is selected, each CPU's clocks are turned on one at a time.

Memory Allocation

Immediately before test code execution, a current memory allocation screen is displayed. The following figure is an example of the current memory allocation screen. The physical and logical addresses shown, as well as the filenames, are only representative; the actual addresses will vary depending on installed memory & file sizes.

The following list explains what each column the current memory allocation screen depicts:

- **First Column** — File Number - useful in conjunction with the mm(1d) utility.
- **Second Column** — Physical memory addresses where the specified file is loaded
- **Third Column** — Process identification
- **Fourth Column** — File name (actual path is *./filename* or /mnt/test/CPU/*filename*). The entries `pte1`, `pte2`, and `ptet` are not actual files but are instead indications of the page tables.
- **Fifth Column** — Logical starting address of the specified file

Figure cpu4041–5, Current Memory Allocation Screen

```
                    Current Memory Allocation

File No.  Physical  Address Pid File Name                    Logical Offset
      1   00000000-00027fff   0 p0r0_4041                   00000000
      2   00028000-000b1fff   0 cpu4041.rnn                 00022000
      1   000b2000-000d9fff   1 p0r0_4041                   00000000
      2   000da000-00163fff   1 cpu4041.rnn                 00022000
   ----   03ff7000-03ff9fff   1 ptet                        NA
   ----   03ffa000-03ffafff   1 pte2                        NA
   ----   03ffb000-03ffdfff   0 ptet                        NA
   ----   03ffe000-03ffefff   0 pte2                        NA
   ----   03fffc00-3fffffff   1 pte1                        NA
```

Class Descriptions

There are four different classes of subtests for *cpu4041*. The following sections describe the different classes and each of their subtests. Each section contains a table listing each subtest in that class, a description of the subtest, the subtest executable test code (object module) and the subtest source code with comments (source file), the minimum time required to run the subtest (nominal time), and the worst case time that occurs as the result of running forced faults (maximum time).

> **NOTE**
>
> In the following tables, the **TEST PERFORMED** column lists each particular instruction that is tested. For more information on each instruction's meaning, refer to the "Opcodes Sorted by Name" appendix within this manual or the *CONVEX Architecture Reference*.

Class 1 Subtests

Class 1 subtests verify the operation of loading, storing, and modifying the vector unit control functions. Specifically, this class verifies the ability to alter and save the Vector Length (VL) register, Vector Stride (VS) register, and the Vector Merge (VM) register.

All subtests end with a halt command and store a halt code in address register A1. If the code stored in A1 is 0x100, then the subtest passed. Any other code in A1 indicates that the subtest failed. The following table lists the Class 1 subtests:

Table cpu4041–2, Class 1 Subtests

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
10	*ld.w #N,VL*	*cpu4041.rnn*	*st_10.s*	0:02	0:10
15	*ld.w #N,VS*	*cpu4041.rnn*	*st_15.s*	0:02	0:10
20	*mov.w Sk,VL*	*cpu4041.rnn*	*st_20.s*	0:02	0:10
25	*mov.w Sk,VS*	*cpu4041.rnn*	*st_25.s*	0:02	0:10
30	*mov Ak,VL*	*cpu4041.rnn*	*st_30.s*	0:02	0:10
35	*mov VL,Ak*	*cpu4041.rnn*	*st_35.s*	0:02	0:10
40	*mov Ak,VS*	*cpu4041.rnn*	*st_40.s*	0:02	0:10
45	*mov VS,Ak*	*cpu4041.rnn*	*st_45.s*	0:02	0:10
50	*ld.l <effa>,VLS*	*cpu4041.rnn*	*st_50.s*	0:02	0:10
51	*ld.l <effa>,VLS*	*cpu4041.rnn*	*st_51.s*	0:02	0:10
55	*st.l VLS,<effa>*	*cpu4041.rnn*	*st_55.s*	0:02	0:10
56	*st.l VLS,<effa>*	*cpu4041.rnn*	*st_56.s*	0:02	0:10
60	*mov Sj,Sk,VM*	*cpu4041.rnn*	*st_60.s*	0:02	0:10
65	*mov Sj,VM,Sk*	*cpu4041.rnn*	*st_65.s*	0:02	0:10
70	*ld.x <effa>,VM*	*cpu4041.rnn*	*st_70.s*	0:02	0:10
75	*st.x VM,<effa>*	*cpu4041.rnn*	*st_75.s*	0:02	0:10
80	*plc.t VM,Sk*	*cpu4041.rnn*	*st_80.s*	0:02	0:10
85	*plc.f VM,Sk*	*cpu4041.rnn*	*st_85.s*	0:02	0:10
195	*mov Si,Sj,Vk*	*cpu4041.rnn*	*st_195.s*	0:02	0:10
198	*mov Vi,Sj,Sk*	*cpu4041.rnn*	*st_198.s*	0:02	0:10
1000	*vector valid traps*	*cpu4041.rnn*	*st_1000.s*	0:02	0:10
1010	*dual vector loads*	*cpu4041.rnn*	*st_1010.s*	0:02	0:10

Class 2 Subtests

Class 2 subtests verify the operation of the logical and arithmetic pipes. Specifically, this class verifies vector/vector and scalar/vector comparisons, vector/vector and scalar/vector additions and subtractions, and vector reductions.

All subtests end with a halt command and store a halt code in address register A1. If the code stored in A1 is 0x100, then the subtest passed. Any other code in A1 indicates that the subtest failed. The following table lists the Class 2 subtests:

Table cpu4041–3, Class 2 Subtests

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
105	*le.b Sj,Vk*	*cpu4041.rnn*	*st_105.s*	0:02	0:10
106	*le.b Sj,Vk*	*cpu4041.rnn*	*st_106.s*	0:02	0:10
110	*le.h Sj,Vk*	*cpu4041.rnn*	*st_110.s*	0:02	0:10
111	*le.h Sj,Vk*	*cpu4041.rnn*	*st_111.s*	0:02	0:10
115	*le.w Sj,Vk*	*cpu4041.rnn*	*st_115.s*	0:02	0:10
116	*le.w Sj,Vk*	*cpu4041.rnn*	*st_116.s*	0:02	0:10
120	*le.l Sj,Vk*	*cpu4041.rnn*	*st_120.s*	0:02	0:10
121	*le.l Sj,Vk*	*cpu4041.rnn*	*st_121.s*	0:02	0:10
125	*le.s Sj,Vk (native mode)*	*cpu4041.rnn*	*st_125.s*	0:02	0:10
126	*le.s Sj,Vk (native mode)*	*cpu4041.rnn*	*st_126.s*	0:02	0:10
130	*le.d Sj,Vk (native mode)*	*cpu4041.rnn*	*st_130.s*	0:02	0:10
131	*le.d Sj,Vk (native mode)*	*cpu4041.rnn*	*st_131.s*	0:02	0:10
135	*lt.b Sj,Vk*	*cpu4041.rnn*	*st_135.s*	0:02	0:10
136	*lt.b Sj,Vk*	*cpu4041.rnn*	*st_136.s*	0:02	0:10
140	*lt.h Sj,Vk*	*cpu4041.rnn*	*st_140.s*	0:02	0:10
141	*lt.h Sj,Vk*	*cpu4041.rnn*	*st_141.s*	0:02	0:10
145	*lt.w Sj,Vk*	*cpu4041.rnn*	*st_145.s*	0:02	0:10
146	*lt.w Sj,Vk*	*cpu4041.rnn*	*st_146.s*	0:02	0:10
150	*lt.l Sj,Vk*	*cpu4041.rnn*	*st_150.s*	0:02	0:10
151	*lt.l Sj,Vk*	*cpu4041.rnn*	*st_151.s*	0:02	0:10
155	*lt.s Sj,Vk (native mode)*	*cpu4041.rnn*	*st_155.s*	0:02	0:10
156	*lt.s Sj,Vk (native mode)*	*cpu4041.rnn*	*st_156.s*	0:02	0:10
160	*lt.d Sj,Vk (native mode)*	*cpu4041.rnn*	*st_160.s*	0:02	0:10
161	*lt.d Sj,Vk (native mode)*	*cpu4041.rnn*	*st_161.s*	0:02	0:10
165	*eq.b Sj,Vk*	*cpu4041.rnn*	*st_165.s*	0:02	0:10
166	*eq.b Sj,Vk*	*cpu4041.rnn*	*st_166.s*	0:02	0:10
170	*eq.h Sj,Vk*	*cpu4041.rnn*	*st_170.s*	0:02	0:10
171	*eq.h Sj,Vk*	*cpu4041.rnn*	*st_171.s*	0:02	0:10
175	*eq.w Sj,Vk*	*cpu4041.rnn*	*st_175.s*	0:02	0:10
176	*eq.w Sj,Vk*	*cpu4041.rnn*	*st_176.s*	0:02	0:10
180	*eq.l Sj,Vk*	*cpu4041.rnn*	*st_180.s*	0:02	0:10
181	*eq.l Sj,Vk*	*cpu4041.rnn*	*st_181.s*	0:02	0:10
185	*eq.s Sj,Vk (native mode)*	*cpu4041.rnn*	*st_185.s*	0:02	0:10

Table cpu4041-4, Class 2 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
186	*eq.s Sj,Vk (native mode)*	*cpu4041.rnn*	*st_186.s*	0:02	0:10
190	*eq.d Sj,Vk (native mode)*	*cpu4041.rnn*	*st_190.s*	0:02	0:10
191	*eq.d Sj,Vk (native mode)*	*cpu4041.rnn*	*st_191.s*	0:02	0:10
250	*add.b Vi,Sj,Vk*	*cpu4041.rnn*	*st_250.s*	0:02	0:10
251	*add.b Vi,Sj,Vk*	*cpu4041.rnn*	*st_251.s*	0:04	0:10
255	*add.h Vi,Sj,Vk*	*cpu4041.rnn*	*st_255.s*	0:02	0:10
256	*add.h Vi,Sj,Vk*	*cpu4041.rnn*	*st_256.s*	0:02	0:10
260	*add.w Vi,Sj,Vk*	*cpu4041.rnn*	*st_260.s*	0:02	0:10
261	*add.w Vi,Sj,Vk*	*cpu4041.rnn*	*st_261.s*	0:04	0:10
265	*add.l Vi,Sj,Vk*	*cpu4041.rnn*	*st_265.s*	0:02	0:10
266	*add.l Vi,Sj,Vk*	*cpu4041.rnn*	*st_266.s*	0:04	0:10
270	*add.s Vi,Sj,Vk (native mode)*	*cpu4041.rnn*	*st_270.s*	0:02	0:10
271	*add.s Vi,Sj,Vk (native mode)*	*cpu4041.rnn*	*st_271.s*	0:02	0:10
275	*add.d Vi,Sj,Vk (native mode)*	*cpu4041.rnn*	*st_275.s*	0:02	0:10
276	*add.d Vi,Sj,Vk (native mode)*	*cpu4041.rnn*	*st_276.s*	0:02	0:10
280	*sub.b Vi,Sj,Vk*	*cpu4041.rnn*	*st_280.s*	0:02	0:10
281	*sub.b Vi,Sj,Vk*	*cpu4041.rnn*	*st_281.s*	0:04	0:10
285	*sub.h Vi,Sj,Vk*	*cpu4041.rnn*	*st_285.s*	0:02	0:10
286	*sub.h Vi,Sj,Vk*	*cpu4041.rnn*	*st_286.s*	0:02	0:10
290	*sub.w Vi,Sj,Vk*	*cpu4041.rnn*	*st_290.s*	0:02	0:10
291	*sub.w Vi,Sj,Vk*	*cpu4041.rnn*	*st_291.s*	0:04	0:10
295	*sub.l Vi,Sj,Vk*	*cpu4041.rnn*	*st_295.s*	0:02	0:10
296	*sub.l Vi,Sj,Vk*	*cpu4041.rnn*	*st_296.s*	0:04	0:10
300	*sub.s Vi,Sj,Vk (native mode)*	*cpu4041.rnn*	*st_300.s*	0:02	0:10
301	*sub.s Vi,Sj,Vk (native mode)*	*cpu4041.rnn*	*st_301.s*	0:02	0:10
305	*sub.d Vi,Sj,Vk (native mode)*	*cpu4041.rnn*	*st_305.s*	0:02	0:10
306	*sub.d Vi,Sj,Vk (native mode)*	*cpu4041.rnn*	*st_306.s*	0:02	0:10
310	*and Vi,Sj,Vk*	*cpu4041.rnn*	*st_310.s*	0:02	0:10
311	*and Vi,Sj,Vk*	*cpu4041.rnn*	*st_311.s*	0:02	0:10
315	*or Vi,Sj,Vk*	*cpu4041.rnn*	*st_315.s*	0:02	0:10
316	*or Vi,Sj,Vk*	*cpu4041.rnn*	*st_316.s*	0:02	0:10
320	*xor Vi,Sj,Vk*	*cpu4041.rnn*	*st_320.s*	0:02	0:10
321	*xor Vi,Sj,Vk*	*cpu4041.rnn*	*st_321.s*	0:04	0:10

Table cpu4041-4, Class 2 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
325	*shf Sj,Vk*	*cpu4041.rnn*	*st_325.s*	0:02	0:10
326	*shf Sj,Vk*	*cpu4041.rnn*	*st_326.s*	0:08	0:10
330	*le.b Vj,Vk*	*cpu4041.rnn*	*st_330.s*	0:02	0:10
331	*le.b Vj,Vk*	*cpu4041.rnn*	*st_331.s*	0:02	0:10
335	*le.h Vj,Vk*	*cpu4041.rnn*	*st_335.s*	0:02	0:10
336	*le.h Vj,Vk*	*cpu4041.rnn*	*st_336.s*	0:02	0:10
340	*le.w Vj,Vk*	*cpu4041.rnn*	*st_340.s*	0:02	0:10
341	*le.w Vj,Vk*	*cpu4041.rnn*	*st_341.s*	0:02	0:10
345	*le.l Vj,Vk*	*cpu4041.rnn*	*st_345.s*	0:02	0:10
346	*le.l Vj,Vk*	*cpu4041.rnn*	*st_346.s*	0:02	0:10
350	*le.s Vj,Vk (native mode)*	*cpu4041.rnn*	*st_350.s*	0:02	0:10
351	*le.s Vj,Vk (native mode)*	*cpu4041.rnn*	*st_351.s*	0:02	0:10
355	*le.d Vj,Vk (native mode)*	*cpu4041.rnn*	*st_355.s*	0:02	0:10
356	*le.d Vj,Vk (native mode)*	*cpu4041.rnn*	*st_356.s*	0:02	0:10
360	*lt.b Vj,Vk*	*cpu4041.rnn*	*st_360.s*	0:02	0:10
361	*lt.b Vj,Vk*	*cpu4041.rnn*	*st_361.s*	0:02	0:10
365	*lt.h Vj,Vk*	*cpu4041.rnn*	*st_365.s*	0:02	0:10
366	*lt.h Vj,Vk*	*cpu4041.rnn*	*st_366.s*	0:02	0:10
370	*lt.w Vj,Vk*	*cpu4041.rnn*	*st_370.s*	0:02	0:10
371	*lt.w Vj,Vk*	*cpu4041.rnn*	*st_371.s*	0:02	0:10
375	*lt.l Vj,Vk*	*cpu4041.rnn*	*st_375.s*	0:02	0:10
376	*lt.l Vj,Vk*	*cpu4041.rnn*	*st_376.s*	0:02	0:10
380	*lt.s Vj,Vk (native mode)*	*cpu4041.rnn*	*st_380.s*	0:02	0:10
381	*lt.s Vj,Vk (native mode)*	*cpu4041.rnn*	*st_381.s*	0:02	0:10
385	*lt.d Vj,Vk (native mode)*	*cpu4041.rnn*	*st_385.s*	0:02	0:10
386	*lt.d Vj,Vk (native mode)*	*cpu4041.rnn*	*st_386.s*	0:02	0:10
390	*eq.b Vj,Vk*	*cpu4041.rnn*	*st_390.s*	0:02	0:10
391	*eq.b Vj,Vk*	*cpu4041.rnn*	*st_391.s*	0:02	0:10
395	*eq.h Vj,Vk*	*cpu4041.rnn*	*st_395.s*	0:02	0:10
396	*eq.h Vj,Vk*	*cpu4041.rnn*	*st_396.s*	0:02	0:10
400	*eq.w Vj,Vk*	*cpu4041.rnn*	*st_400.s*	0:02	0:10
401	*eq.w Vj,Vk*	*cpu4041.rnn*	*st_401.s*	0:02	0:10
405	*eq.l Vj,Vk*	*cpu4041.rnn*	*st_405.s*	0:02	0:10

Table cpu4041-4, Class 2 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
406	*eq.l Vj,Vk*	*cpu4041.rnn*	*st_406.s*	0:02	0:10
410	*eq.s Vj,Vk (native mode)*	*cpu4041.rnn*	*st_410.s*	0:02	0:10
411	*eq.s Vj,Vk (native mode)*	*cpu4041.rnn*	*st_411.s*	0:02	0:10
415	*eq.d Vj,Vk (native mode)*	*cpu4041.rnn*	*st_415.s*	0:02	0:10
416	*eq.d Vj,Vk (native mode)*	*cpu4041.rnn*	*st_416.s*	0:02	0:10
430	*add.b Vi,Vj,Vk*	*cpu4041.rnn*	*st_430.s*	0:02	0:10
431	*add.b Vi,Vj,Vk*	*cpu4041.rnn*	*st_431.s*	0:05	0:10
435	*add.h Vi,Vj,Vk*	*cpu4041.rnn*	*st_435.s*	0:02	0:10
436	*add.h Vi,Vj,Vk*	*cpu4041.rnn*	*st_436.s*	0:02	0:10
440	*add.w Vi,Vj,Vk*	*cpu4041.rnn*	*st_440.s*	0:02	0:10
441	*add.w Vi,Vj,Vk*	*cpu4041.rnn*	*st_441.s*	0:05	0:10
445	*add.l Vi,Vj,Vk*	*cpu4041.rnn*	*st_445.s*	0:02	0:10
446	*add.l Vi,Vj,Vk*	*cpu4041.rnn*	*st_446.s*	0:04	0:10
450	*add.s Vi,Vj,Vk (native mode)*	*cpu4041.rnn*	*st_450.s*	0:02	0:10
451	*add.s Vi,Vj,Vk (native mode)*	*cpu4041.rnn*	*st_451.s*	0:02	0:10
455	*add.d Vi,Vj,Vk (native mode)*	*cpu4041.rnn*	*st_455.s*	0:02	0:10
456	*add.d Vi,Vj,Vk (native mode)*	*cpu4041.rnn*	*st_456.s*	0:02	0:10
460	*sub.b Vi,Vj,Vk*	*cpu4041.rnn*	*st_460.s*	0:02	0:10
461	*sub.b Vi,Vj,Vk*	*cpu4041.rnn*	*st_461.s*	0:05	0:10
465	*sub.h Vi,Vj,Vk*	*cpu4041.rnn*	*st_465.s*	0:02	0:10
466	*sub.h Vi,Vj,Vk*	*cpu4041.rnn*	*st_466.s*	0:02	0:10
470	*sub.w Vi,Vj,Vk*	*cpu4041.rnn*	*st_470.s*	0:02	0:10
471	*sub.w Vi,Vj,Vk*	*cpu4041.rnn*	*st_471.s*	0:04	0:10
475	*sub.l Vi,Vj,Vk*	*cpu4041.rnn*	*st_475.s*	0:02	0:10
476	*sub.l Vi,Vj,Vk*	*cpu4041.rnn*	*st_476.s*	0:04	0:10
480	*sub.s Vi,Vj,Vk (native mode)*	*cpu4041.rnn*	*st_480.s*	0:02	0:10
481	*sub.s Vi,Vj,Vk (native mode)*	*cpu4041.rnn*	*st_481.s*	0:02	0:10
485	*sub.d Vi,Vj,Vk (native mode)*	*cpu4041.rnn*	*st_485.s*	0:02	0:10
486	*sub.d Vi,Vj,Vk (native mode)*	*cpu4041.rnn*	*st_486.s*	0:02	0:10
490	*and Vi,Vj,Vk*	*cpu4041.rnn*	*st_490.s*	0:02	0:10
491	*and Vi,Vj,Vk*	*cpu4041.rnn*	*st_491.s*	0:02	0:10
495	*or Vi,Vj,Vk*	*cpu4041.rnn*	*st_495.s*	0:02	0:10
496	*or Vi,Vj,Vk*	*cpu4041.rnn*	*st_496.s*	0:02	0:10

Table cpu4041-4, Class 2 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
500	*xor Vi,Vj,Vk*	*cpu4041.rnn*	*st_500.s*	0:02	0:10
501	*xor Vi,Vj,Vk*	*cpu4041.rnn*	*st_501.s*	0:02	0:10
505	*not Vj,Vk*	*cpu4041.rnn*	*st_505.s*	0:02	0:10
506	*not Vj,Vk*	*cpu4041.rnn*	*st_506.s*	0:02	0:10
510	*neg.b Vj,Vk*	*cpu4041.rnn*	st_510.s	0:02	0:10
511	*neg.b Vj,Vk*	*cpu4041.rnn*	*st_511.s*	0:02	0:10
515	*neg.h Vj,Vk*	*cpu4041.rnn*	*st_515.s*	0:02	0:10
516	*neg.h Vj,Vk*	*cpu4041.rnn*	*st_516.s*	0:02	0:10
520	*neg.w Vj,Vk*	*cpu4041.rnn*	*st_520.s*	0:02	0:10
521	*neg.w Vj,Vk*	*cpu4041.rnn*	*st_521.s*	0:02	0:10
525	*neg.l Vj,Vk*	*cpu4041.rnn*	*st_525.s*	0:02	0:10
526	*neg.l Vj,Vk*	*cpu4041.rnn*	*st_526.s*	0:02	0:10
530	*neg.s Vj,Vk (native mode)*	*cpu4041.rnn*	*st_530.s*	0:02	0:10
531	*neg.s Vj,Vk (native mode)*	*cpu4041.rnn*	*st_531.s*	0:02	0:10
535	*neg.d Vj,Vk (native mode)*	*cpu4041.rnn*	*st_535.s*	0:02	0:10
536	*neg.d Vj,Vk (native mode)*	*cpu4041.rnn*	*st_536.s*	0:02	0:10
770	*merg.t Vi,Sj,Vk*	*cpu4041.rnn*	*st_770.s*	0:02	0:10
771	*merg.t Vi,Sj,Vk*	*cpu4041.rnn*	*st_771.s*	0:02	0:10
775	*merg.f Vi,Sj,Vk*	*cpu4041.rnn*	*st_775.s*	0:02	0:10
776	*merg.f Vi,Sj,Vk*	*cpu4041.rnn*	*st_776.s*	0:02	0:10
780	*mask.t Vi,Sj,Vk*	*cpu4041.rnn*	*st_780.s*	0:02	0:10
781	*mask.t Vi,Sj,Vk*	*cpu4041.rnn*	*st_781.s*	0:02	0:10
785	*mask.f Vi,Sj,Vk*	*cpu4041.rnn*	*st_785.s*	0:02	0:10
786	*mask.f Vi,Sj,Vk*	*cpu4041.rnn*	*st_786.s*	0:02	0:10
790	*merg.t Vi,Vj,Vk*	*cpu4041.rnn*	*st_790.s*	0:02	0:10
791	*merg.t Vi,Vj,Vk*	*cpu4041.rnn*	*st_791.s*	0:02	0:10
800	*mask.t Vi,Vj,Vk*	*cpu4041.rnn*	*st_800.s*	0:02	0:10
801	*mask.t Vi,Vj,Vk*	*cpu4041.rnn*	*st_801.s*	0:02	0:10
805	*plc.t Vj,Vk*	*cpu4041.rnn*	*st_805.s*	0:02	0:10
806	*plc.t Vj,Vk*	*cpu4041.rnn*	*st_806.s*	0:04	0:10
810	*cprs.t Vi,Vj,Vk*	*cpu4041.rnn*	*st_810.s*	0:02	0:10
811	*cprs.t Vi,Vj,Vk*	*cpu4041.rnn*	*st_811.s*	0:02	0:10
815	*cprs.t Vi,Vj,Vk*	*cpu4041.rnn*	*st_815.s*	0:02	0:10

Table cpu4041-4, Class 2 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
816	*cprs.t Vi,Vj,Vk*	*cpu4041.rnn*	*st_816.s*	0:02	0:10
820	*sum.b Vk*	*cpu4041.rnn*	*st_820.s*	0:02	0:10
821	*sum.b Vk*	*cpu4041.rnn*	*st_821.s*	0:04	0:10
825	*sum.h Vk*	*cpu4041.rnn*	*st_825.s*	0:02	0:10
826	*sum.h Vk*	*cpu4041.rnn*	*st_826.s*	0:04	0:10
830	*sum.w Vk*	*cpu4041.rnn*	*st_830.s*	0:02	0:10
831	*sum.w Vk*	*cpu4041.rnn*	*st_831.s*	0:04	0:10
835	*sum.l Vk*	*cpu4041.rnn*	*st_835.s*	0:02	0:10
836	*sum.l Vk*	*cpu4041.rnn*	*st_836.s*	0:04	0:10
840	*sum.s Vk (native mode)*	*cpu4041.rnn*	*st_840.s*	0:02	0:10
841	*sum.s Vk (native mode)*	*cpu4041.rnn*	*st_841.s*	0:04	0:10
845	*sum.d Vk (native mode)*	*cpu4041.rnn*	*st_845.s*	0:02	0:10
846	*sum.d Vk (native mode)*	*cpu4041.rnn*	*st_846.s*	0:04	0:10
880	*max.b Vk*	*cpu4041.rnn*	*st_880.s*	0:02	0:10
881	*max.b Vk*	*cpu4041.rnn*	*st_881.s*	0:15	0:20
885	*max.h Vk*	*cpu4041.rnn*	*st_885.s*	0:02	0:10
886	*max.h Vk*	*cpu4041.rnn*	*st_886.s*	0:15	0:20
890	*max.w Vk*	*cpu4041.rnn*	*st_890.s*	0:02	0:05
891	*max.w Vk*	*cpu4041.rnn*	*st_891.s*	0:15	0:20
895	*max.l Vk*	*cpu4041.rnn*	*st_895.s*	0:02	0:05
896	*max.l Vk*	*cpu4041.rnn*	*st_896.s*	0:15	0:20
900	*max.s Vk (native mode)*	*cpu4041.rnn*	*st_900.s*	0:02	0:15
901	*max.s Vk (native mode)*	*cpu4041.rnn*	*st_901.s*	0:15	0:20
905	*max.d Vk (native mode)*	*cpu4041.rnn*	*st_905.s*	0:02	0:10
906	*max.d Vk (native mode)*	*cpu4041.rnn*	*st_906.s*	0:15	0:20
910	*min.b Vk*	*cpu4041.rnn*	*st_910.s*	0:02	0:05
911	*min.b Vk*	*cpu4041.rnn*	*st_911.s*	0:15	0:20
915	*min.h Vk*	*cpu4041.rnn*	*st_915.s*	0:02	0:05
916	*min.h Vk*	*cpu4041.rnn*	*st_916.s*	0:15	0:20
920	*min.w Vk*	*cpu4041.rnn*	*st_920.s*	0:02	0:05
921	*min.w Vk*	*cpu4041.rnn*	*st_921.s*	0:15	0:20
925	*min.l Vk*	*cpu4041.rnn*	*st_925.s*	0:02	0:05
926	*min.l Vk*	*cpu4041.rnn*	*st_926.s*	0:15	0:20

Table cpu4041-4, Class 2 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
930	*min.s Vk (native mode)*	*cpu4041.rnn*	*st_930.s*	0:02	0:10
931	*min.s Vk (native mode)*	*cpu4041.rnn*	*st_931.s*	0:15	0:20
935	*min.d Vk (native mode)*	*cpu4041.rnn*	*st_935.s*	0:02	0:05
936	*min.d Vk (native mode)*	*cpu4041.rnn*	*st_936.s*	0:15	0:20
940	*all Vk*	*cpu4041.rnn*	*st_940.s*	0:02	0:05
941	*all Vk*	*cpu4041.rnn*	*st_941.s*	0:55	0:80
950	*any Vk*	*cpu4041.rnn*	*st_950.s*	0:02	0:05
951	*any Vk*	*cpu4041.rnn*	*st_951.s*	0:55	0:80
960	*parity Vk*	*cpu4041.rnn*	*st_960.s*	0:02	0:05
961	*parity Vk*	*cpu4041.rnn*	*st_961.s*	0:02	0:80
1125	*le.s Sj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1125.s*	0:02	0:10
1126	*le.s Sj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1126.s*	0:02	0:10
1130	*le.d Sj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1130.s*	0:02	0:10
1131	*le.d Sj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1131.s*	0:02	0:10
1155	*lt.s Sj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1155.s*	0:02	0:10
1156	*lt.s Sj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1156.s*	0:02	0:10
1160	*lt.d Sj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1160.s*	0:02	0:10
1161	*lt.d Sj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1161.s*	0:02	0:10
1185	*eq.s Sj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1185.s*	0:02	0:10
1186	*eq.s Sj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1186.s*	0:02	0:10
1190	*eq.d Sj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1190.s*	0:02	0:10
1191	*eq.d Sj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1191.s*	0:02	0:10
1270	*add.s Vi,Sj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1270.s*	0:02	0:10
1271	*add.s Vi,Sj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1271.s*	0:02	0:10
1275	*add.d Vi,Sj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1275.s*	0:02	0:10
1276	*add.d Vi,Sj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1276.s*	0:02	0:10
1300	*sub.s Vi,Sj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1300.s*	0:02	0:10
1301	*sub.s Vi,Sj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1301.s*	0:02	0:10
1305	*sub.d Vi,Sj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1305.s*	0:02	0:10
1306	*sub.d Vi,Sj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1306.s*	0:02	0:10
1350	*le.s Vj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1350.s*	0:02	0:10
1351	*le.s Vj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1351.s*	0:02	0:10
1355	*le.d Vj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1355.s*	0:02	0:10

Table cpu4041-4, Class 2 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
1356	*le.d Vj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1356.s*	0:02	0:10
1380	*lt.s Vj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1380.s*	0:02	0:10
1381	*lt.s Vj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1381.s*	0:02	0:10
1385	*lt.d Vj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1385.s*	0:02	0:10
1386	*lt.d Vj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1386.s*	0:02	0:10
1410	*eq.s Vj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1410.s*	0:02	0:10
1411	*eq.s Vj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1411.s*	0:02	0:10
1415	*eq.d Vj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1415.s*	0:02	0:10
1416	*eq.d Vj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1416.s*	0:02	0:10
1450	*add.s Vi,Vj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1450.s*	0:02	0:10
1451	*add.s Vi,Vj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1451.s*	0:02	0:10
1455	*add.d Vi,Vj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1455.s*	0:02	0:10
1456	*add.d Vi,Vj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1456.s*	0:02	0:10
1480	*sub.s Vi,Vj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1480.s*	0:02	0:10
1481	*sub.s Vi,Vj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1481.s*	0:02	0:10
1485	*sub.d Vi,Vj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1485.s*	0:02	0:10
1486	*sub.d Vi,Vj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1486.s*	0:02	0:10
1530	*neg.s Vj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1530.s*	0:02	0:10
1531	*neg.s Vj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1531.s*	0:02	0:10
1535	*neg.d Vj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1535.s*	0:02	0:10
1536	*neg.d Vj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1536.s*	0:02	0:10
1840	*sum.s Vk (IEEE mode)*	*cpu4041.rnn*	*st_1840.s*	0:02	0:10
1841	*sum.s Vk (IEEE mode)*	*cpu4041.rnn*	*st_1841.s*	0:04	0:10
1845	*sum.d Vk (IEEE mode)*	*cpu4041.rnn*	*st_1845.s*	0:02	0:10
1846	*sum.d Vk (IEEE mode)*	*cpu4041.rnn*	*st_1846.s*	0:04	0:10
1900	*max.s Vk (IEEE mode)*	*cpu4041.rnn*	*st_1900.s*	0:02	0:10
1901	*max.s Vk (IEEE mode)*	*cpu4041.rnn*	*st_1901.s*	0:15	0:20
1905	*max.d Vk (IEEE mode)*	*cpu4041.rnn*	*st_1905.s*	0:02	0:10
1906	*max.d Vk (IEEE mode)*	*cpu4041.rnn*	*st_1906.s*	0:15	0:20
1930	*min.s Vk (IEEE mode)*	*cpu4041.rnn*	*st_1930.s*	0:02	0:10
1931	*min.s Vk (IEEE mode)*	*cpu4041.rnn*	*st_1931.s*	0:15	0:20
1935	*min.d Vk (IEEE mode)*	*cpu4041.rnn*	*st_1935.s*	0:02	0:10
1936	*min.d Vk (IEEE mode)*	*cpu4041.rnn*	*st_1936.s*	0:15	0:10

Class 3 Subtests

Class 3 subtests verify the operation of the multiply and divide pipe. Specifically, this class verifies the vector/vector and scalar/vector multiplications and divisions.

All subtests end with a halt command and store a halt code in address register A1. If the code stored in A1 is 0x100, then the subtest passed. Any other code in A1 indicates that the subtest failed. The following table lists the Class 3 subtests:

Table cpu4041–4, Class 3 Subtests

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
630	*mul.b Vi,Sj,Vk*	*cpu4041.rnn*	*st_630.s*	0:02	0:10
631	*mul.b Vi,Sj,Vk*	*cpu4041.rnn*	*st_631.s*	0:04	0:10
635	*mul.h Vi,Sj,Vk*	*cpu4041.rnn*	*st_635.s*	0:02	0:10
636	*mul.h Vi,Sj,Vk*	*cpu4041.rnn*	*st_636.s*	0:04	0:10
640	*mul.w Vi,Sj,Vk*	*cpu4041.rnn*	*st_640.s*	0:02	0:10
641	*mul.w Vi,Sj,Vk*	*cpu4041.rnn*	*st_641.s*	0:04	0:10
645	*mul.l Vi,Sj,Vk*	*cpu4041.rnn*	*st_645.s*	0:02	0:10
646	*mul.l Vi,Sj,Vk*	*cpu4041.rnn*	*st_646.s*	0:04	0:10
650	*mul.s Vi,Sj,Vk (native mode)*	*cpu4041.rnn*	*st_650.s*	0:02	0:10
651	*mul.s Vi,Sj,Vk (native mode)*	*cpu4041.rnn*	*st_651.s*	0:02	0:10
655	*mul.d Vi,Sj,Vk (native mode)*	*cpu4041.rnn*	*st_655.s*	0:02	0:10
656	*mul.d Vi,Sj,Vk (native mode)*	*cpu4041.rnn*	*st_656.s*	0:02	0:10
660	*div.b Vi,Sj,Vk*	*cpu4041.rnn*	*st_660.s*	0:02	0:10
661	*div.b Vi,Sj,Vk*	*cpu4041.rnn*	*st_661.s*	0:02	0:10
665	*div.h Vi,Sj,Vk*	*cpu4041.rnn*	*st_665.s*	0:02	0:10
666	*div.h Vi,Sj,Vk*	*cpu4041.rnn*	*st_666.s*	0:02	0:10
670	*div.w Vi,Sj,Vk*	*cpu4041.rnn*	*st_670.s*	0:02	0:10
671	*div.w Vi,Sj,Vk*	*cpu4041.rnn*	*st_671.s*	0:02	0:10
675	*div.l Vi,Sj,Vk*	*cpu4041.rnn*	*st_675.s*	0:02	0:10
676	*div.l Vi,Sj,Vk*	*cpu4041.rnn*	*st_676.s*	0:04	0:10
680	*div.s Vi,Sj,Vk (native mode)*	*cpu4041.rnn*	*st_680.s*	0:02	0:10
681	*div.s Vi,Sj,Vk (native mode)*	*cpu4041.rnn*	*st_681.s*	0:02	0:10
685	*div.d Vi,Sj,Vk (native mode)*	*cpu4041.rnn*	*st_685.s*	0:02	0:10
686	*div.d Vi,Sj,Vk (native mode)*	*cpu4041.rnn*	*st_686.s*	0:02	0:10
700	*mul.b Vi,Vj,Vk*	*cpu4041.rnn*	*st_700.s*	0:02	0:10
701	*mul.b Vi,Vj,Vk*	*cpu4041.rnn*	*st_701.s*	0:04	0:10
705	*mul.h Vi,Vj,Vk*	*cpu4041.rnn*	*st_705.s*	0:02	0:10
706	*mul.h Vi,Vj,Vk*	*cpu4041.rnn*	*st_706.s*	0:02	0:10
710	*mul.w Vi,Vj,Vk*	*cpu4041.rnn*	*st_710.s*	0:02	0:10
711	*mul.w Vi,Vj,Vk*	*cpu4041.rnn*	*st_711.s*	0:04	0:10
715	*mul.l Vi,Vj,Vk*	*cpu4041.rnn*	*st_715.s*	0:02	0:10
716	*mul.l Vi,Vj,Vk*	*cpu4041.rnn*	*st_716.s*	0:04	0:10
720	*mul.s Vi,Vj,Vk (native mode)*	*cpu4041.rnn*	*st_720.s*	0:02	0:10

Table cpu4041-5, Class 3 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
721	*mul.s Vi,Vj,Vk (native mode)*	*cpu4041.rnn*	*st_721.s*	0:02	0:10
725	*mul.d Vi,Vj,Vk (native mode)*	*cpu4041.rnn*	*st_725.s*	0:02	0:10
726	*mul.d Vi,Vj,Vk (native mode)*	*cpu4041.rnn*	*st_726.s*	0:02	0:10
740	*div.b Vi,Vj,Vk*	*cpu4041.rnn*	*st_740.s*	0:02	0:10
741	*div.b Vi,Vj,Vk*	*cpu4041.rnn*	*st_741.s*	0:02	0:10
745	*div.h Vi,Vj,Vk*	*cpu4041.rnn*	*st_745.s*	0:02	0:10
746	*div.h Vi,Vj,Vk*	*cpu4041.rnn*	*st_746.s*	0:02	0:10
750	*div.w Vi,Vj,Vk*	*cpu4041.rnn*	*st_750.s*	0:02	0:10
751	*div.w Vi,Vj,Vk*	*cpu4041.rnn*	*st_751.s*	0:02	0:10
755	*div.l Vi,Vj,Vk*	*cpu4041.rnn*	*st_755.s*	0:02	0:10
756	*div.l Vi,Vj,Vk*	*cpu4041.rnn*	*st_756.s*	0:02	0:10
760	*div.s Vi,Vj,Vk (native mode)*	*cpu4041.rnn*	*st_760.s*	0:02	0:10
761	*div.s Vi,Vj,Vk (native mode)*	*cpu4041.rnn*	*st_761.s*	0:02	0:10
765	*div.d Vi,Vj,Vk (native mode)*	*cpu4041.rnn*	*st_765.s*	0:02	0:10
766	*div.d Vi,Vj,Vk (native mode)*	*cpu4041.rnn*	*st_766.s*	0:04	0:10
850	*prod.b Vk*	*cpu4041.rnn*	*st_850.s*	0:02	0:10
851	*prod.b Vk*	*cpu4041.rnn*	*st_851.s*	0:04	0:10
855	*prod.h Vk*	*cpu4041.rnn*	*st_855.s*	0:02	0:10
856	*prod.h Vk*	*cpu4041.rnn*	*st_856.s*	0:04	0:10
860	*prod.w Vk*	*cpu4041.rnn*	*st_860.s*	0:02	0:10
861	*prod.w Vk*	*cpu4041.rnn*	*st_861.s*	0:04	0:10
865	*prod.l Vk*	*cpu4041.rnn*	*st_865.s*	0:02	0:10
866	*prod.l Vk*	*cpu4041.rnn*	*st_866.s*	0:04	0:10
870	*prod.s Vk (native mode)*	*cpu4041.rnn*	*st_870.s*	0:02	0:10
871	*prod.s Vk (native mode)*	*cpu4041.rnn*	*st_871.s*	0:06	0:10
875	*prod.d Vk (native mode)*	*cpu4041.rnn*	*st_875.s*	0:02	0:10
876	*prod.d Vk (native mode)*	*cpu4041.rnn*	*st_876.s*	0:08	0:15
1650	*mul.s Vi,Sj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1650.s*	0:02	0:10
1651	*mul.s Vi,Sj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1651.s*	0:02	0:10
1655	*mul.d Vi,Sj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1655.s*	0:02	0:10
1656	*mul.d Vi,Sj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1656.s*	0:02	0:10
1680	*div.s Vi,Sj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1680.s*	0:02	0:10
1681	*div.s Vi,Sj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1681.s*	0:02	0:10

Table cpu4041-5, Class 3 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
1685	*div.d Vi,Sj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1685.s*	0:02	0:10
1686	*div.d Vi,Sj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1686.s*	0:02	0:10
1720	*mul.s Vi,Vj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1720.s*	0:02	0:10
1721	*mul.s Vi,Vj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1721.s*	0:02	0:10
1725	*mul.d Vi,Vj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1725.s*	0:02	0:10
1726	*mul.d Vi,Vj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1726.s*	0:02	0:10
1760	*div.s Vi,Vj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1760.s*	0:02	0:10
1761	*div.s Vi,Vj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1761.s*	0:02	0:10
1765	*div.d Vi,Vj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1765.s*	0:02	0:10
1766	*div.d Vi,Vj,Vk (IEEE mode)*	*cpu4041.rnn*	*st_1766.s*	0:02	0:10
1870	*prod.s Vk (IEEE mode)*	*cpu4041.rnn*	*st_1870.s*	0:02	0:10
1871	*prod.s Vk (IEEE mode)*	*cpu4041.rnn*	*st_1871.s*	0:08	0:10
1875	*prod.d Vk (IEEE mode)*	*cpu4041.rnn*	*st_1875.s*	0:02	0:10
1876	*prod.d Vk (IEEE mode)*	*cpu4041.rnn*	*st_1876.s*	0:02	0:10

Class 4 Subtests

Class 4 subtests verify the operation of loading and storing vector registers. Specifically, this class verifies loading and storing the vector using direct addressing and vector of indices, and storing of vectors and scalar registers using the extended operations.

All subtests end with a halt command and store a halt code in address register A1. If the code stored in A1 is 0x100, then the subtest passed. Any other code in A1 indicates that the subtest failed. The following table lists the Class 4 subtests:

Table cpu4041–5, Class 4 Subtests

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
200	*ld.b <effa>,Vk*	*cpu4041.rnn*	*st_200.s*	0:02	0:10
201	*ld.b <effa>,Vk*	*cpu4041.rnn*	*st_201.s*	0:02	0:10
205	*ld.h <effa>,Vk*	*cpu4041.rnn*	*st_205.s*	0:02	0:10
206	*ld.h <effa>,Vk*	*cpu4041.rnn*	*st_206.s*	0:02	0:10
210	*ld.w <effa>,Vk*	*cpu4041.rnn*	*st_210.s*	0:02	0:10
211	*ld.w <effa>,Vk*	*cpu4041.rnn*	*st_211.s*	0:02	0:10
215	*ld.l <effa>,Vk*	*cpu4041.rnn*	*st_215.s*	0:02	0:10
216	*ld.l <effa>,Vk*	*cpu4041.rnn*	*st_216.s*	0:02	0:10
230	*st.b Vk,<effa>*	*cpu4041.rnn*	*st_230.s*	0:02	0:10
231	*st.b Vk,<effa>*	*cpu4041.rnn*	*st_231.s*	0:02	0:10
235	*st.h Vk,<effa>*	*cpu4041.rnn*	*st_235.s*	0:02	0:10
236	*st.h Vk,<effa>*	*cpu4041.rnn*	*st_236.s*	0:02	0:10
240	*st.w Vk,<effa>*	*cpu4041.rnn*	*st_240.s*	0:02	0:10
241	*st.w Vk,<effa>*	*cpu4041.rnn*	*st_241.s*	0:04	0:10
245	*st.l Vk,<effa>*	*cpu4041.rnn*	*st_245.s*	0:02	0:10
246	*st.l Vk,<effa>*	*cpu4041.rnn*	*st_246.s*	0:05	0:10
550	*ldvi.b Vj,Vk*	*cpu4041.rnn*	*st_550.s*	0:02	0:10
551	*ldvi.b Vj,Vk*	*cpu4041.rnn*	*st_551.s*	0:06	0:10
555	*ldvi.h Vj,Vk*	*cpu4041.rnn*	*st_555.s*	0:02	0:10
556	*ldvi.h Vj,Vk*	*cpu4041.rnn*	*st_556.s*	0:06	0:10
560	*ldvi.w Vj,Vk*	*cpu4041.rnn*	*st_560.s*	0:02	0:10
561	*ldvi.w Vj,Vk*	*cpu4041.rnn*	*st_561.s*	0:06	0:10
565	*ldvi.l Vj,Vk*	*cpu4041.rnn*	*st_565.s*	0:02	0:10
566	*ldvi.l Vj,Vk*	*cpu4041.rnn*	*st_566.s*	0:08	0:15
570	*stvi.b Vi,Vj*	*cpu4041.rnn*	*st_570.s*	0:02	0:10
571	*stvi.b Vi,Vj*	*cpu4041.rnn*	*st_571.s*	0:02	0:10
575	*stvi.h Vi,Vj*	*cpu4041.rnn*	*st_575.s*	0:02	0:10
576	*stvi.h Vi,Vj*	*cpu4041.rnn*	*st_576.s*	0:02	0:10
580	*stvi.w Vi,Vj*	*cpu4041.rnn*	*st_580.s*	0:02	0:10
581	*stvi.w Vi,Vj*	*cpu4041.rnn*	*st_581.s*	0:02	0:10
585	*stvi.l Vi,Vj*	*cpu4041.rnn*	*st_585.s*	0:02	0:10
586	*stvi.l Vi,Vj*	*cpu4041.rnn*	*st_586.s*	0:02	0:10
590	*stvi.b Si,Vj*	*cpu4041.rnn*	*st_590.s*	0:02	0:10

Table cpu4041-6, Class 4 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
591	*stvi.b Si,Vj*	*cpu4041.rnn*	*st_591.s*	0:02	0:10
595	*stvi.h Si,Vj*	*cpu4041.rnn*	*st_595.s*	0:02	0:10
596	*stvi.h Si,Vj*	*cpu4041.rnn*	*st_506.s*	0:02	0:10
600	*stvi.w Si,Vj*	*cpu4041.rnn*	*st_600.s*	0:02	0:10
601	*stvi.w Si,Vj*	*cpu4041.rnn*	*st_601.s*	0:02	0:10
605	*stvi.l Si,Vj*	*cpu4041.rnn*	*st_605.s*	0:02	0:10
606	*stvi.l Si,Vj*	*cpu4041.rnn*	*st_606.s*	0:02	0:10
610	*ste.b Sk,<effa>*	*cpu4041.rnn*	*st_601.s*	0:02	0:10
611	*ste.b Sk,<effa>*	*cpu4041.rnn*	*st_611.s*	0:02	0:10
615	*ste.h Sk,<effa>*	*cpu4041.rnn*	*st_615.s*	0:02	0:10
616	*ste.h Sk,<effa>*	*cpu4041.rnn*	*st_616.s*	0:02	0:10
620	*ste.w Vk,<effa>*	*cpu4041.rnn*	*st_620.s*	0:02	0:10
621	*ste.w Vk,<effa>*	*cpu4041.rnn*	*st_621.s*	0:04	0:10
625	*ste.l Vk,<effa>*	*cpu4041.rnn*	*st_625.s*	0:02	0:10
626	*ste.l Vk,<effa>*	*cpu4041.rnn*	*st_626.s*	0:06	0:10

Test Error Messages

For a common list of error messages that could result from running this test, refer to the appendix on CPU Error Messages.

SPU UNIX Error Messages

For a common list of possible error messages resulting from SPU UNIX, refer to the appendix on SPU UNIX Error Messages.

THIS PAGE INTENTIONALLY LEFT BLANK

cpu4131

Privileged Instructions and Architectural Features

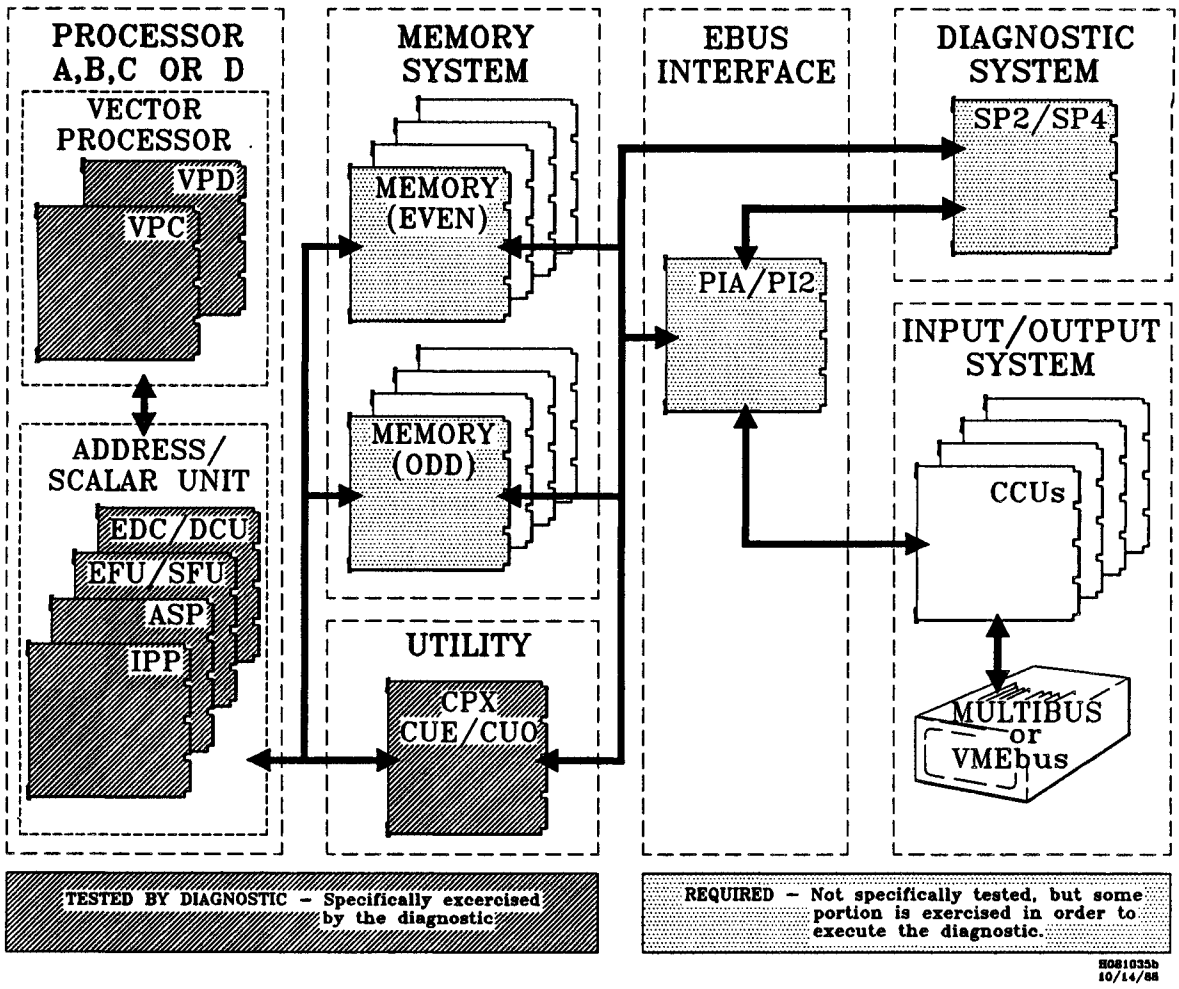
Overview

The *cpu4131* subtests are designed to test exceptions, privileged instructions, page faulting, interrupts, and all timers that are specific to the architecture of the machine.

> **NOTE**
>
> The *cpu4131* test is applicable only for C210A machines. This test should not be attempted on any other C200 Series machine. The *cpu4231* test is designed to replace this test for any C200 Series machines.

The following figure shows an overall view of what part of the system is being tested and which Field Replaceable Units (FRUs) are required for the test to execute.

Figure cpu4131–1, Functional Areas Tested by *cpu4131*



Prerequisites and Required Equipment

In order to run the *cpu4131* test, the boards listed in the following table must be operational. The table shows the tests used to verify the required boards. No additional equipment is required to run this test.

Table cpu4131–1, Required Functional Boards

BOARD	TEST TO VERIFY
Service Processor (SP2 or SP4)	*spu1000, spu4000*
PBUS Interface Adapter (PIA or PI2)	*pia4000* or *pi2_4000*
Memory System	*mem4000*
CPU Utility Board(s) (CPX or CUE/CUO)	*cpx4000*
Vector Processor Control (VPC)	*cpu4030*
Vector Processor Data (VPD)	*cpu4030*

NOTE

Memory System consists of a minimum of one pair of memory boards (one even and one odd).

Test Invocation

To invoke the *cpu4131* test, use the procedure shown in the following figure. All responses in **boldface** are entered by the user. The prompts and responses in the following figure would appear sequentially on the screen, one line at a time. All the prompts and responses are shown in one figure for convenience.

CAUTION

The invocation sequence shown in the following figure is the typical invocation sequence. However, the *initall* utility must be executed in some cases. If the system has just been powered up, if *mem4000* was executed with failures, or if *spu4000* was executed, then *initall* must be executed prior to any test execution. Failure to execute *initall* in these circumstances could result in invalid test results.

NOTE

Running the *initall* utility requires two to three minutes to execute depending on if the control stores have been previously loaded. However, it is suggested that *initall* be executed in the event that the state of the system is unknown.

Figure cpu4131–2, Test Invocation Sequence

```
(spu)>  cd /mnt/test
(spu)>  sysreset
(spu)>  dshell

CONVEX DIAGNOSTIC SHELL

:  test cpu4131 [-c [class numeral(s)]] [-s [subtest numeral(s)]] [+> filename]
```

NOTE

After entering the user response, "**dshell**", specific *dshell* parameters may be changed. Refer to the "Dshell and Iscan Overview" chapter of this manual for more information on *dshell*.

Entering only **test cpu4131** executes all *cpu4131* subtests sequentially. The user can execute a specific class(es) of subtest(s) or one or more individual subtests by using the *-c* or *-s* options, respectively. Detailed information for using these options can be found in the "Dshell and Iscan Overview" chapter of this manual. The **[+>***filename***]** option allows the test results to be appended to *filename*.

Test Parameter Menu

Once the test is invoked, a test parameter menu prompts for selection of default switches. If the test is run with all defaults invoked, (user answers **y** to the first prompt) no other prompts are provided. If the user answers **n** to the first prompt, (run test without default switches) then a series of prompts are presented. The following figure shows all possible prompts, their possible answers (in brackets [ ]), and their default answers (in parentheses ( )). The prompts and responses in the following figure would appear sequentially on the screen, one line at a time. All the prompts and responses are shown in one figure for convenience.

Figure cpu4131–3, Test Parameter Menu

```
                        ENTER TEST PARAMETERS

                [ ]    Encloses allowed input ranges or values
                ( )    Encloses the default value
                ^       Returns to the previous prompt
                :nn     Returns to the prompt # nn
                :       Returns to the first unsatisfied prompt
                :?      Reviews previous entries

 1: Run default switches? [y,n]                          (y) ->
 2: Forced Faulting Enabled? [y,n]                       (n) ->
 3: Fault on Instruction Fetches? [y,n]                  (n) ->
 4: Sequential Execution? [y,n]                          (n) ->
 5: Timeout Scale Factor Enabled? [1-100]                (1) ->
 6: Dcache Enabled? [y,n]                                (y) ->
 7: Segment of Execution? [0-7]                          (0) ->
 8: Loop Enabled? [y,n]                                  (n) ->
 9: Chained Execution Mode? [y,n]                        (n) ->
10:  Hard Errors Enabled? [y,n]                          (y) ->
11:  Load CPU Code? [y,n]                                (y) ->
```

NOTE

The third prompt in the above list is only supplied when the second prompt is answered with **y**.

Prompt Explanations

A description of what each prompt means follows:

```
Run default switches? [y/n]                              (y) ->
```

If the user responds with **y** or <CR>, no additional test parameter prompts are displayed and testing begins. However, if a negative response is supplied, additional test parameter prompts are displayed allowing modification of the default selections.

The following prompts are only displayed and answered if the first prompt is answered with **n**:

```
Forced Faulting Enabled? [y,n]                           (n) ->
```

If answered with **y**, normal force faulting occurs only on data references. The system will force a non-resident data exception to occur on every data reference. For a more detailed explanation of forced faulting, refer to the glossary appendix of this manual.

```
Fault on Instruction Fetches? [y,n]                (n) ->
```

In answered with **y**, force faulting occurs on instruction fetches in addition to data references. This prompt is only supplied to the user if the previous prompt is answered with **y**.

```
Sequential Execution? [y,n]                        (n) ->
```

If set by entering **y**, the sequential bit in the PSW will be set to forced sequential execution mode.

```
Timeout Scale Factor [1-100]                       (1) ->
```

This prompt allows adjustment of the timeout factor. The normal timeout factor is multiplied by the number selected to increase the timeout factor. For example, if **5** is entered, the normal timeout factor is multiplied by 5 and it will take the test five times as long to timeout.

```
Dcache Enabled? [y,n]                              (y) ->
```

The Dcache is normally enabled; however, if it is suspected broken, it can be disabled by entering **n** at this prompt.

```
Segment of Execution? [0-7]                        (0) ->
```

The segment of execution is contained in bits<31..29> of the Program Counter (PC). If **0** is entered, then bits<31..29> of the PC are 000 and the test is run in ring zero. If **1** is entered, then bits<31..29> of the PC are 001 and the test is run in ring one. If **2** is entered, then bits<31..29> of the PC are 010 and the test is run in ring two. If **3** is entered, then bits<31..29> of the PC are 011 and the test is run in ring three. If **4**, **5**, **6**, or **7** is entered, then bits<31..29> of the PC are 100, 101, 110, and 111 respectively and the test is run in ring four. Refer to the *CONVEX Architecture Reference* for more information concerning the meaning of rings in the machine architecture.

```
Loop Enabled? [y,n]                                (n) ->
```

If **y** is entered, hard looping on a specific subtest is enabled. When looping is enabled, the halt instruction of the specified subtest is changed to a branch back to the beginning of the subtest. This puts the subtest into an infinite loop which the user must break out of by typing **Ctrl- C**.

```
Chained Execution Mode? [y,n]                      (n) ->
```

With this option enabled, the test is executed in chained mode which causes the CPU to perform subtest sequencing. The Service Processor is unaware of the action of the CPU unless a subtest fails or unless all of the subtests pass. If this option is enabled, test execution time will be greatly reduced. However, the only information printed to the console upon completion or failure (regardless of the cause) is the message, "Subtest 1 passed." or "Subtest 1 failed." Also, this option can not be enabled if the **-c** or the **-s** options were used in the invocation procedure. When this mode is enabled, Subtests 560-570 are not executed.

```
Hard Errors Enabled? [y,n]                    (y) ->
```

If this option is enabled and a hard error occurs, the clocks will be stopped and the test will fail. If this option is disabled, parity errors and other sources of hard errors will go undetected. It is recommended that hard errors normally be enabled.

```
Load CPU Code? [y,n]                          (y) ->
```

If the CPU code for this test is already in memory, the user can enter **n** for this prompt and the code will not be reloaded (thus saving time).

When all prompts have been answered, the screen displays a test parameter summary which echos the prompts that have been answered. The following figure illustrates an example of a "Test Parameter Summary" screen. The actual values and responses vary according to the input.

Figure cpu4131–4, Sample Test Parameter Summary

```
                    Test Parameter Summary

Run default switches?                    : y
Forced Faulting Enabled?                 : n
Sequential Execution?                    : n
Timeout Scale Factor Enabled?            : 1
Dcache Enabled?                          : y
Segment of Execution?                    : 0
Loop Enabled?                            : n
Hard Errors Enabled?                     : y
Load CPU Code?                           : y
```

Hardware Initialization Sequence

After the last prompt is entered by the user (and before test code execution) the following events occur:

> **NOTE**
>
> The first two events are accomplished at the initial start of the *cpu4131* test. The remaining events are accomplished when each subtest is initialized.

- Each module's page table entries are set up in main memory. Page table entries begin at the last available address in main memory and work toward low memory.
- Each module is loaded into main memory following the logical to physical mapping described by the page tables. The exact logical to physical mapping that the modules are loaded into depends upon which segment of execution is selected by the user.

- The system is initialized (the memory system, CPX or CUE/CUO, PIA or PI2, and the CPU boards are reset).
- Parity is initialized in the scalar processor by sending the scalar processor into a microcode routine and issuing clocks.
- Scratch RAM is loaded with various values to control the execution of the test.
- The initial Program Counter (PC) is loaded into register T0 and the initial Program Status Word (PSW) is loaded into register T2.
- Control store is initialized to the cold-start location.
- Clocks are turned on.

Memory Allocation

Immediately before test code execution, a current memory allocation screen is displayed. The following figure is an example of the current memory allocation screen. The physical and logical addresses shown, as well as the filenames, are only representative; the actual addresses will vary depending on installed memory & file sizes.

The following list explains what each column the current memory allocation screen depicts:

- **First Column** — File Number - useful in conjunction with the mm(1d) utility.
- **Second Column** — Physical memory addresses where the specified file is loaded
- **Third Column** — Process identification
- **Fourth Column** — File name (actual path is *./filename* or /mnt/test/CPU/*filename*). The entries `pte1`, `pte2`, and `ptet` are not actual files but are instead indications of the page tables.
- **Fifth Column** — Logical starting address of the specified file

Figure cpu4131–5, Current Memory Allocation Screen

```
                    Current Memory Allocation

File No.  Physical  Address Pid File Name                    Logical Offset
      1   00000000-00027fff   0 p0r0_4131                   00000000
      2   00028000-000b1fff   0 cpu4131.rnn                 00022000
      1   000b2000-000d9fff   1 p0r0_4131                   00000000
      2   000da000-00163fff   1 cpu4131.rnn                 00022000
   ----   03ff7000-03ff9fff   1 ptet                        NA
   ----   03ffa000-03ffafff   1 pte2                        NA
   ----   03ffb000-03ffdfff   0 ptet                        NA
   ----   03ffe000-03ffefff   0 pte2                        NA
   ----   03fffc00-3fffffff   1 pte1                        NA
```

Class Descriptions

There are four classes of subtests for *cpu4131*.

Class 1 Subtests

Class one subtests verify a set of privileged instructions that are architecture dependent. All subtests terminate with a halt command and store a halt code in address register A1. If the code stored in A1 is 0x100, then the subtest passed. Any other code in A1 indicates that the subtest failed. The following table lists each Class 1 subtest, the instructions performed, its object module, its source file, and the nominal time that each subtest requires to execute.

Table cpu4131–2, Class 1 Subtests

SUBTEST	INSTRUCTIONS TESTED	OBJECT MODULE	SOURCE FILE	DEFAULT TIME (min/sec)	TIMEOUT LIMIT (min/sec)
10	system calls	*cpu4131.rnn*	*st_10.s*	0:02	0:05
20	ldpa aj,ak	*cpu4131.rnn*	*st_20.s*	0:02	0:05
21	patu	*cpu4131.rnn*	*st_21.s*	0:02	0:05
22	pate aj	*cpu4131.rnn*	*st_22.s*	0:02	0:05
25	pich	*cpu4131.rnn*	*st_25.s*	0:02	0:05
30	traps	*cpu4131.rnn*	*st_30.s*	0:02	0:05
35	exceptions	*cpu4131.rnn*	*st_35.s*	0:02	0:05
40	interrupts	*cpu4131.rnn*	*st_40.s*	0:02	0:05
41	interval timer test	*cpu4131.rnn*	*st_41.s*	0:02	0:05
42	timer ring cross check	*cpu4131.rnn*	*st_42.s*	0:02	0:05
45	privileged instruction check	*cpu4131.rnn*	*st_45.s*	0:02	0:05
600	test vector valid test	*cpu4131.rnn*	*st_600.s*	0:02	0:05
601	mov sk,vv test	*cpu4131.rnn*	*st_601.s*	0:02	0:05
602	vv pipeline test	*cpu4131.rnn*	*st_602.s*	0:02	0:05
603	putr4t/getr4t	*cpu4131.rnn*	*st_603.s*	0:02	0:05
604	mov toc,sk	*cpu4131.rnn*	*st_604.s*	0:02	0:05

Class 2 Subtests

Class 2 subtests take page faults while executing call and return instructions. All subtests terminate with a halt command and store a halt code in address register A1. If the code stored in A1 is 0x100, then the subtest passed. Any other code in A1 indicates that the subtest failed. The following table lists each Class 2 subtest, the instructions performed, its object module, its source file, and the nominal time that each subtest requires to execute.

> **NOTE**
>
> In the following tables, NR stands for non-resident page.

Table cpu4131–3, Class 2 Subtests

SUBTEST	INSTRUCTIONS TESTED	OBJECT MODULE	SOURCE FILE	DEFAULT TIME (min/sec)	TIMEOUT LIMIT (min/sec)
50	callq(abs/@) into non resident page	*cpu4131.rnn*	*st_50.s*	0:02	0:05
51	rtnq into *NR* page	*cpu4131.rnn*	*st_51.s*	0:02	0:05
52	callq with *NR* stack	*cpu4131.rnn*	*st_52.s*	0:02	0:05
53	rtnq with *NR* stack	*cpu4131.rnn*	*st_53.s*	0:02	0:05
60	call (abs/@) into non resident page	*cpu4131.rnn*	*st_60.s*	0:02	0:05
61	rtn into *NR* page	*cpu4131.rnn*	*st_61.s*	0:02	0:05
62	call(abs/@) with *NR* stack	*cpu4131.rnn*	*st_62.s*	0:02	0:05
63	rtn with *NR* stack	*cpu4131.rnn*	*st_63.s*	0:02	0:05
70	calls (abs/@) into non resident page	*cpu4131.rnn*	*st_70.s*	0:02	0:05
71	rtn into *NR* page	*cpu4131.rnn*	*st_71.s*	0:02	0:05
72	calls with *NR* stack	*cpu4131.rnn*	*st_72.s*	0:02	0:05
73	rtn with *NR* stack	*cpu4131.rnn*	*st_73.s*	0:02	0:05
80	*NR* stack,*NR* target (callq)	*cpu4131.rnn*	*st_80.s*	0:02	0:05
81	*NR* stack,*NR* target (calls)	*cpu4131.rnn*	*st_81.s*	0:02	0:05
82	*NR* stack,*NR* target (call)	*cpu4131.rnn*	*st_82.s*	0:02	0:05
85	indirect *NR*, *NR* target (callq)	*cpu4131.rnn*	*st_85.s*	0:02	0:05
86	indirect *NR*, *NR* target (calls)	*cpu4131.rnn*	*st_86.s*	0:02	0:05
87	indirect *NR*, *NR* target (call)	*cpu4131.rnn*	*st_87.s*	0:02	0:05
90	indirect *NR*, *NR* stack (callq)	*cpu4131.rnn*	*st_90.s*	0:02	0:05
91	indirect *NR*, *NR* stack (calls)	*cpu4131.rnn*	*st_50.s*	0:02	0:05
92	indirect *NR*, *NR* stack (call)	*cpu4131.rnn*	*st_92.s*	0:02	0:05
95	*NR* stack,*NR* return page (callq)	*cpu4131.rnn*	*st_95.s*	0:02	0:05
96	*NR* stack,*NR* return page (calls)	*cpu4131.rnn*	*st_96.s*	0:02	0:05
97	*NR* stack,*NR* return page (call)	*cpu4131.rnn*	*st_97.s*	0:02	0:05
100	*NR* stack, *NR* target, *NR* address (callq)	*cpu4131.rnn*	*st_100.s*	0:02	0:05
101	*NR* stack, *NR* target, *NR* indirect(calls)	*cpu4131.rnn*	*st_101.s*	0:02	0:05
102	*NR* stack, *NR* target, *NR* indirect(call)	*cpu4131.rnn*	*st_102.s*	0:02	0:05
200	halfword, *NR* page execute	*cpu4131.rnn*	*st_200.s*	0:02	0:05
201	word, *NR* page execute	*cpu4131.rnn*	*st_201.s*	0:02	0:05
202	3-halfword,*NR* page execute	*cpu4131.rnn*	*st_202.s*	0:02	0:05
322	ip lookahead faults	*cpu4131.rnn* and *support_4131*	*st_322.s*	0:02	0:05

Class 3 Subtests

Class 3 subtests perform loading and storing instructions with different alignments across non-resident page boundaries. All subtests terminate with a halt command and store a halt code in address register A1. If the code stored in A1 is 0x100, then the subtest passed. Any other code in A1 indicates that the subtest failed. The following table lists each Class 3 subtest, the instructions performed, its object module, its source file, and the nominal time that each subtest requires to execute.

Table cpu4131–4, Class 3 Subtests

SUBTEST	INSTRUCTIONS TESTED	OBJECT MODULE	SOURCE FILE	DEFAULT TIME (min/sec)	TIMEOUT LIMIT (min/sec)
210	byte loads *NR* page	*cpu4131.rnn*	*st_210.s*	0:02	0:05
220	load halfword *NR* page	*cpu4131.rnn*	*st_220.s*	0:02	0:05
221	load halfword *NR* page	*cpu4131.rnn*	*st_221.s*	0:02	0:05
222	load halfword *NR* page	*cpu4131.rnn*	*st_222.s*	0:02	0:05
223	load halfword *NR* page	*cpu4131.rnn*	*st_223.s*	0:02	0:05
224	load halfword *NR* page	*cpu4131.rnn*	*st_224.s*	0:02	0:05
225	load halfword *NR* page	*cpu4131.rnn*	*st_225.s*	0:02	0:05
226	load halfword *NR* page	*cpu4131.rnn*	*st_226.s*	0:02	0:05
227	load halfword *NR* page	*cpu4131.rnn*	*st_227.s*	0:02	0:05
228	load halfword *NR* page	*cpu4131.rnn*	*st_228.s*	0:02	0:05
230	load word *NR* page	*cpu4131.rnn*	*st_230.s*	0:02	0:05
231	load word *NR* page	*cpu4131.rnn*	*st_231.s*	0:02	0:05
232	load word *NR* page	*cpu4131.rnn*	*st_232.s*	0:02	0:05
233	load word *NR* page	*cpu4131.rnn*	*st_233.s*	0:02	0:05
234	load word *NR* page	*cpu4131.rnn*	*st_234.s*	0:02	0:05
235	load word *NR* page	*cpu4131.rnn*	*st_235.s*	0:02	0:05
236	load word *NR* page	*cpu4131.rnn*	*st_236.s*	0:02	0:05
237	load word *NR* page	*cpu4131.rnn*	*st_237.s*	0:02	0:05
238	load word *NR* page	*cpu4131.rnn*	*st_238.s*	0:02	0:05
240	load longword *NR* page	*cpu4131.rnn*	*st_240.s*	0:02	0:05
241	load longword *NR* page	*cpu4131.rnn*	*st_241.s*	0:02	0:05
242	load longword *NR* page	*cpu4131.rnn*	*st_242.s*	0:02	0:05
243	load longword *NR* page	*cpu4131.rnn*	*st_243.s*	0:02	0:05
244	load longword *NR* page	*cpu4131.rnn*	*st_244.s*	0:02	0:05
245	load longword *NR* page	*cpu4131.rnn*	*st_245.s*	0:02	0:05
246	load longword *NR* page	*cpu4131.rnn*	*st_246.s*	0:02	0:05
247	load longword *NR* page	*cpu4131.rnn*	*st_247.s*	0:02	0:05
248	load longword *NR* page	*cpu4131.rnn*	*st_248.s*	0:02	0:05
250	store byte *NR* page	*cpu4131.rnn*	*st_249.s*	0:02	0:05
260	store halfword *NR* page	*cpu4131.rnn*	*st_260.s*	0:02	0:05
261	store halfword *NR* page	*cpu4131.rnn*	*st_261.s*	0:02	0:05
270	store word *NR* page	*cpu4131.rnn*	*st_270.s*	0:02	0:05
271	store word *NR* page	*cpu4131.rnn*	*st_271.s*	0:02	0:05
272	store word *NR* page	*cpu4131.rnn*	*st_272.s*	0:02	0:05
273	store word *NR* page	*cpu4131.rnn*	*st_273.s*	0:02	0:05
274	store word *NR* page	*cpu4131.rnn*	*st_274.s*	0:02	0:05
275	store word *NR* page	*cpu4131.rnn*	*st_275.s*	0:02	0:05
276	store word *NR* page	*cpu4131.rnn*	*st_276.s*	0:02	0:05
277	store word *NR* page	*cpu4131.rnn*	*st_277.s*	0:02	0:05
278	store word *NR* page	*cpu4131.rnn*	*st_278.s*	0:02	0:05

Table cpu4131–5, Class 3 Subtests (continued)

SUBTEST	INSTRUCTIONS TESTED	OBJECT MODULE	SOURCE FILE	DEFAULT TIME (min/sec)	TIMEOUT LIMIT (min/sec)
280	store longword *NR* page	*cpu4131.rnn*	*st_280.s*	0:02	0:05
281	store longword *NR* page	*cpu4131.rnn*	*st_281.s*	0:02	0:05
282	store longword *NR* page	*cpu4131.rnn*	*st_282.s*	0:02	0:05
283	store longword *NR* page	*cpu4131.rnn*	*st_283.s*	0:02	0:05
284	store longword *NR* page	*cpu4131.rnn*	*st_284.s*	0:02	0:05
285	store longword *NR* page	*cpu4131.rnn*	*st_285.s*	0:02	0:05
286	store longword *NR* page	*cpu4131.rnn*	*st_286.s*	0:02	0:05
287	store longword *NR* page	*cpu4131.rnn*	*st_287.s*	0:02	0:05
288	store longword *NR* page	*cpu4131.rnn*	*st_288.s*	0:02	0:05
290	load byte indirect, address *NR*	*cpu4131.rnn*	*st_290.s*	0:02	0:05
291	load halfword indirect, address *NR*	*cpu4131.rnn*	*st_291.s*	0:02	0:05
292	load word indirect, address *NR*	*cpu4131.rnn*	*st_292.s*	0:02	0:05
293	load longword indirect, address *NR*	*cpu4131.rnn*	*st_293.s*	0:02	0:05
300	store byte indirect, address *NR*	*cpu4131.rnn*	*st_300.s*	0:02	0:05
301	store halfword indirect, address *NR*	*cpu4131.rnn*	*st_301.s*	0:02	0:05
302	store word indirect, address *NR*	*cpu4131.rnn*	*st_302.s*	0:02	0:05
303	store longword indirect, address *NR*	*cpu4131.rnn*	*st_303.s*	0:02	0:05
305	load byte indirect, address/data *NR*	*cpu4131.rnn*	*st_305.s*	0:02	0:05
306	load byte indirect, address/data *NR*	*cpu4131.rnn*	*st_306.s*	0:02	0:05
307	load byte indirect, address/data *NR*	*cpu4131.rnn*	*st_307.s*	0:02	0:05
308	load byte indirect, address/data *NR*	*cpu4131.rnn*	*st_308.s*	0:02	0:05
310	store byte indirect, address/data *NR*	*cpu4131.rnn*	*st_310.s*	0:02	0:05
311	store byte indirect, address/data *NR*	*cpu4131.rnn*	*st_311.s*	0:02	0:05
312	store byte indirect, address/data *NR*	*cpu4131.rnn*	*st_312.s*	0:02	0:05
313	store byte indirect, address/data *NR*	*cpu4131.rnn*	*st_313.s*	0:02	0:05
320	multiple fault test	*cpu4131.rnn*	*st_320.s*	0:02	0:05

Class 4 Subtests

Class 4 subtests are architecture dependent cache tests. All subtests terminate with a halt command and store a halt code in address register A1. If the code stored in A1 is 0x100, then the subtest passed. Any other code in A1 indicates that the subtest failed. The following table lists each Class 4 subtest, the instructions performed, its object module, its source file, and the nominal time that each subtest requires to execute.

Table cpu4131–6, Class 4 Subtests

SUBTEST	INSTRUCTIONS TESTED	OBJECT MODULE	SOURCE FILE	DEFAULT TIME (min/sec)	TIMEOUT LIMIT (min/sec)
400	pte cache test	*cpu4131.rnn*	*st_400.s*	0:02	0:05
500	icache test I	*cpu4131.rnn*	*st_500.s*	0:02	0:05
501	icache test II	*cpu4131.rnn*	*st_501.s*	0:02	0:05
502	icache test III	*cpu4131.rnn*	*st_501.s*	0:02	0:05
503	icache test IV	*cpu4131.rnn*	*st_503.s*	0:02	0:05
504	icache test V	*cpu4131.rnn*	*st_504.s*	0:02	0:05
505	icache test VI	*cpu4131.rnn*	*st_505.s*	0:02	0:05
506	icache test VII	*cpu4131.rnn*	*st_506.s*	0:02	0:05
510	dcache test I	*cpu4131.rnn*	*st_510.s*	0:02	0:05
511	dcache test II	*cpu4131.rnn*	*st_511.s*	0:02	0:05
512	dcache test III	*cpu4131.rnn*	*st_512.s*	0:02	0:05
513	dcache test IV	*cpu4131.rnn*	*st_513.s*	0:02	0:05
514	dcache test V	*cpu4131.rnn*	*st_514.s*	0:02	0:05
515	dcache test VI	*cpu4131.rnn*	*st_515.s*	0:02	0:05
516	dcache test VII	*cpu4131.rnn*	*st_516.s*	0:02	0:05
517	dcache test VIII	*cpu4131.rnn*	*st_517.s*	0:02	0:05
518	dcache test IX	*cpu4131.rnn*	*st_518.s*	0:02	0:05
519	dcache test X	*cpu4131.rnn*	*st_519.s*	0:02	0:05
560	Remote Invalidate I	*cpu4131.rnn*	*st_560.s*	0:25	0:35
565	Remote Invalidate II	*cpu4131.rnn*	*st_565.s*	0:25	0:35
570	Remote Invalidate III	*cpu4131.rnn*	*st_570.s*	0:25	0:35

Test Error Messages

For a common list of error messages that could result from running this test, refer to appendix on CPU Error Messages.

SPU UNIX Error Messages

For a common list of possible error messages resulting from SPU UNIX, refer to the appendix on SPU UNIX Error Messages.

THIS PAGE INTENTIONALLY LEFT BLANK

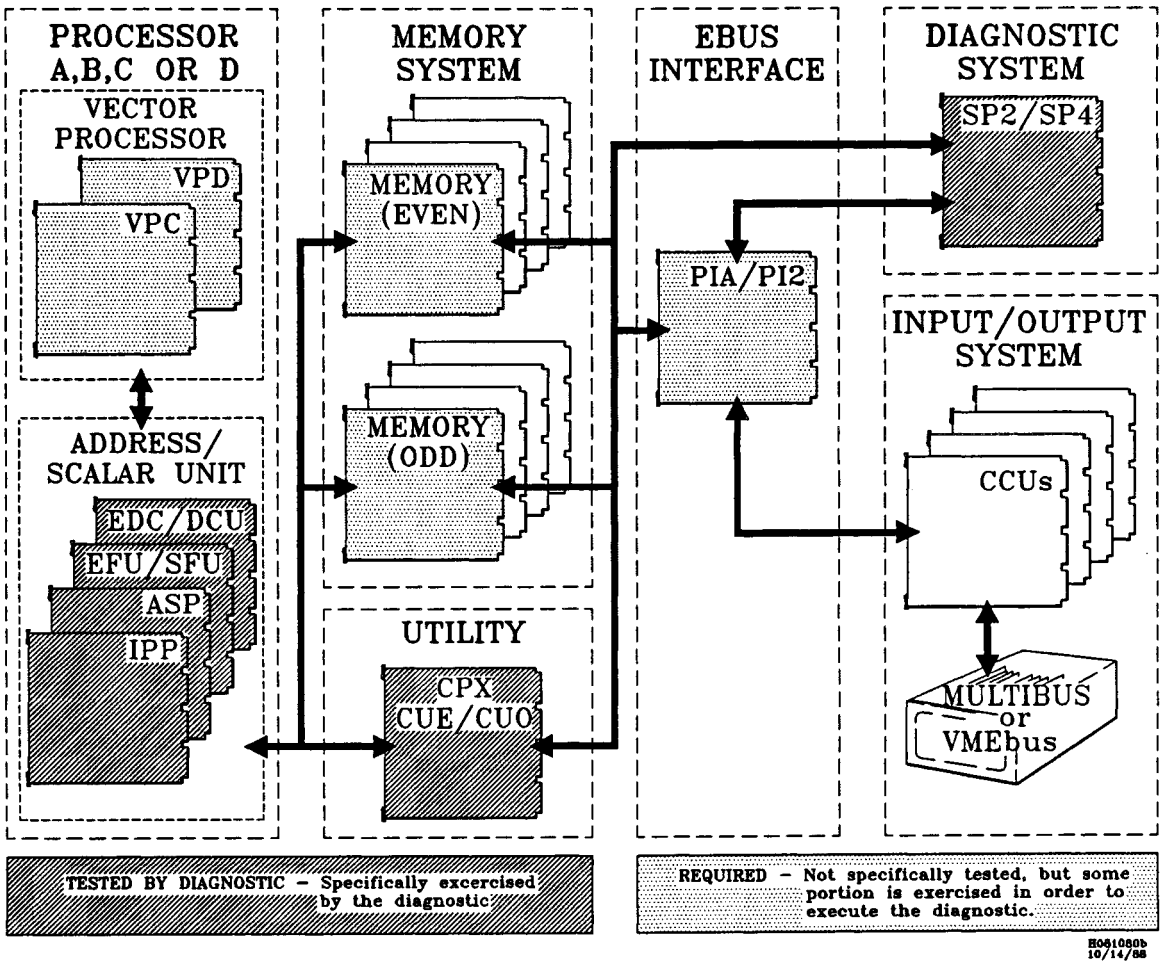
cpu4231

C200 Series Privileged Instruction & Architectural Features

Overview

The *cpu4231* test verifies non-vector, single-headed architectural features unique to the C200 Series processors. Included are tests of system exceptions, interrupts, privileged instructions, the various processor caches, remote invalidates, and non-resident memory pages.

Figure cpu4231–1, Functional Areas Tested by *cpu4231*



Prerequisites and Required Equipment

In order to run the *cpu4231* test, the Vector Processor Control (VPC) and the Vector Processor Data (VPD) must either be present in the machine or their slots must be terminated with terminators. Also, the boards listed in the following table must be operational. The table shows the tests used to verify the required boards. No additional equipment is required to run this test.

Table cpu4231–1, Required Functional Boards

BOARD	TEST TO VERIFY
Service Processor (SP2 or SP4)	*spu1000, spu4000*
Memory System	*mem4000*
Instruction Processor Unit (IPP)	*cpu4030*
Address Scalar Processor (ASP)	*cpu4030*
Scalar Function Unit (SFU or EFU)	*cpu4030*
Data Cache Unit (DCU or EDC)	*cpu4030*
Vector Processor Control (VPC)	*cpu4041*
Vector Processor Data (VPD)	*cpu4041*
CPU Utility Board (CPX or CUE/CUE)	*cpx4000*
PBUS Interface Adapter (PIA or PI2)	*pia4000*

NOTE

The memory system consists of a minimum of one pair of memory boards (one even and one odd).

Test Invocation

To invoke the *cpu4231* test, use the procedure shown in the following figure. All responses in **boldface** are entered by the user. The prompts and responses appear sequentially on the screen, one line at a time. All prompts and responses are shown in one figure for convenience.

CAUTION

The invocation sequence shown in the following figure is the typical invocation sequence. However, the *initall* utility must be executed in some cases. If the system has just been powered up, if *mem4000* was executed with failures, or if *spu4000* was executed, then *initall* must be executed prior to any test execution. Failure to execute *initall* in these circumstances could result in invalid test results.

NOTE

Running the *initall* utility requires two to three minutes to execute depending on if the control stores have been previously loaded. However, it is suggested that *initall* be executed in the event that the state of the system is unknown.

Figure cpu4231–2, Test Invocation Sequence

```
(spu)>  cd /mnt/test
(spu)>  sysreset
(spu)>  dshell

CONVEX DIAGNOSTIC SHELL

:test cpu4231 [-c [class numeral(s)]] [-s [subtest numeral(s)]] [+> filename]
```

NOTE

After entering **dshell**, specific *dshell* parameters may be changed. Refer to the "Dshell and Iscan Overview" chapter of this manual for more information.

Entering only **test cpu4231** executes all *cpu4231* subtests sequentially. Execute a specific class(es) of subtest(s) or one or more individual subtests by using the *-c* or *-s* options, respectively. Detailed information for using these options can be found in the "Dshell and Iscan Overview" chapter of this manual. The [+>*filename*] option allows the test results to be appended to *filename*.

Test Parameter Menu

Once the test is invoked, a test parameter menu is presented allowing selection of default switches. If the test is run with all defaults invoked (answer **y** to the first prompt), no other prompts are provided. If the user answers **n,** to the first prompt (run test without default switches), then a series of prompts are presented. The following figure shows all prompts, their possible answers (in brackets [ ]), and their default answers (in parentheses ( )). The prompts and responses in the following figure appear sequentially on the screen, one line at a time. All the prompts and responses are shown in one figure for convenience.

Figure cpu4231–3, Test Parameter Menu

```
Test 'cpu4231.t'                                        Thu Nov 19 00:00:00 1965

                              ENTER TEST PARAMETERS

                    [ ]     Encloses allowed input ranges or values
                    ( )     Encloses the default value
                    ^       Returns to the previous prompt
                    :nn     Returns to the prompt # nn
                    :       Returns to the first unsatisfied prompt
                    :?      Reviews previous entries


 1:  Run default switches? [y,n]                                    (y) ->
 2:  Cpus to test: [ABCD]                                           (ABCD) ->
 3:  Forced Faulting Enabled? [y,n]                                 (n) ->
 4:  Fault on Instruction Fetches? [y,n]                            (n) ->
 5:  Sequential Execution? [y,n]                                    (n) ->
 6:  Timeout Scale Factor Enabled? [1-100]                          (1) ->
 7:  Dcache Enabled? [y,n]                                          (y) ->
 8:  Segment of Execution? [0-7]                                    (0) ->
 9:  Chained Execution Mode? [y,n]                                  (n) ->
 10: Loop Enabled? [y,n]                                            (n) ->
 11: Hard Errors Enabled? [y,n]                                     (y) ->
 12: Load CPU Code? [y,n]                                           (y) ->
```

NOTE

In the second prompt in the previous figure, ABCD represents all available CPUs. Only the available CPUs within the machine under test will be displayed as possible choices. The fourth prompt in the previous list is only supplied when the third prompt is answered with **y**. The ninth prompt does not appear if the test was invoked with the -s or -c options.

Prompt Explanations

A description of the meaning of each prompt follows:

```
Run default switches? [y/n]                                    (y) ->
```

If a response of **y** or <**CR**> is given, no additional test parameter prompts are displayed and testing begins. However, if a negative response is supplied, additional test parameter prompts are displayed allowing modification of the default selections.

The following prompts are only displayed and answered if the first prompt is answered with **n**:

```
Cpus to test: [ABCD]                                    (ABCD) ->
```

This prompt allows selection of the CPUs to be used in the test. The possible selection, represented by ABCD, will consist of all available CPUs. The default, ABCD, will consist of all available CPUs.

```
Forced Faulting Enabled? [y,n]                          (n) ->
```

If answered with **y**, normal force faulting occurs only on data references. The system will force a non-resident data exception to occur on every data reference. For a more detailed explanation of forced faulting, refer to the glossary appendix of this manual.

```
Fault on Instruction Fetches? [y,n]                     (n) ->
```

In answered with **y**, force faulting occurs on instruction fetches in addition to data references. This prompt is only supplied to the user if the previous prompt is answered with **y**.

```
Sequential Execution? [y,n]                             (n) ->
```

If set by entering **y**, the sequential bit in the PSW will be set to forced sequential execution mode.

```
Timeout Scale Factor [1-100]                            (1) ->
```

This prompt allows adjustment of the timeout factor. The normal timeout factor is multiplied by the number selected to increase the timeout factor. For example, if **5** is entered, the normal timeout factor is multiplied by 5 and it will take the test five times as long to timeout.

```
Dcache Enabled? [y,n]                                   (y) ->
```

The Dcache is normally enabled; however, if it is suspected broken, it can be disabled by entering **n** at this prompt.

```
Segment of Execution? [0-7]                             (0) ->
```

The segment of execution is contained in bits<31..29> of the Program Counter (PC). If **0** is entered, then bits<31..29> of the PC are 000 and the test is run in ring zero. If **1** is entered, then bits<31..29> of the PC are 001 and the test is run in ring one. If **2** is entered, then bits<31..29> of the PC are 010 and the test is run in ring two. If **3** is entered, then bits<31..29> of the PC are 011 and the test is run in ring three. If **4**, **5**, **6**, or **7** is entered, then bits<31..29> of the PC are 100, 101, 110, and 111 respectively and the test is run in ring four. Refer to the *CONVEX Architecture Reference* for more information concerning the meaning of rings in the machine architecture.

```
Chained Execution Mode? [y,n]                           (n) ->
```

With this option enabled, the test is executed in chained mode which causes the CPU to perform subtest sequencing. The SPU is unaware of the action of the CPU unless a subtest fails or unless all of the subtests pass, so no subtest execution tracing occurs in this mode. All subtests are executed, however, and test execution time will be greatly reduced. Also, this option can not be enabled if the **-c** or the **-s** options were used in the invocation procedure.

```
Loop Enabled? [y,n]                                    (n) ->
```

If **y** is entered, hard looping on a specific subtest is enabled. When looping is enabled, the halt instruction of the specified subtest is changed to a branch back to the beginning of the subtest. This puts the subtest into an infinite loop which the user must break out of by typing **Ctrl- C**.

```
Hard Errors Enabled? [y,n]                    (y) ->
```

If this option is enabled and a hard error occurs, the clocks will be stopped and the test will fail. If this option is disabled, parity errors and other sources of hard errors will go undetected. It is recommended that hard errors normally be enabled.

```
Load CPU Code? [y,n]                          (y) ->
```

If the CPU code for this test is already in memory, the user can enter **n** for this prompt and the code will not be reloaded (thus saving time).

When all prompts have been answered, a test parameter summary echos the prompts that have been answered. The following figure displays a sample test parameter summary. The actual summary varies depending on the answers to the prompts.

Figure cpu4231–4, Sample Test Parameter Summary

```
                    TEST PARAMETER SUMMARY

Run default switches?                          :  n
Cpus to test:                                  :  ab
Forced Faulting Enabled?                       :  y
Sequential Execution?                          :  n
Timeout Scale Factor Enabled?                  :  1
Dcache Enabled?                                :  y
Segment of Execution?                          :  0
Chained Execution Mode?                        :  n
Loop Enabled?                                  :  n
Hard Errors Enabled?                           :  y
Load CPU Code?                                 :  y
```

Hardware Initialization Sequence

After the last prompt is entered, and before test code execution, the following events occur:

> **NOTE**
>
> The first two events are accomplished at the initial start of the test. The remaining events are accomplished when each subtest is initialized.

- For each CPU, each module's page table entries are set up in main memory. Page table entries begin at the last available address in main memory and work toward low memory.
- For each CPU, each module is loaded into main memory following the logical to physical mapping described by the page tables. The exact logical to physical mapping that the modules are loaded into depends upon which segment of execution is selected by the user.
- The system is initialized (the memory system, CPX, PIA, and the CPU boards are reset).
- For each CPU, parity is initialized in the scalar processor by sending the scalar processor into a microcode routine and issuing clocks.
- Each CPU's scratch RAM is loaded with various values to control the execution of the test.
- For each CPU, the initial Program Counter (PC) is loaded into register T0 and the initial Program Status Word (PSW) is loaded into register T2.
- For each CPU, control store is initialized to the cold-start location.
- Clocks are turned on for each selected CPU, one at a time.

Memory Allocation

Immediately before test code execution, a current memory allocation screen is displayed. The following figure is an example of the current memory allocation screen. The physical and logical addresses shown, as well as the filenames, are only representative; the actual addresses will vary depending on installed memory & file sizes.

The following list explains what each column the current memory allocation screen depicts:

- **First Column** — File Number - useful in conjunction with the mm(1d) utility.
- **Second Column** — Physical memory addresses where the specified file is loaded
- **Third Column** — Process identification
- **Fourth Column** — File name (actual path is *./filename* or /mnt/test/CPU/*filename*). The entries `pte1`, `pte2`, and `ptet` are not actual files but are instead indications of the page tables.
- **Fifth Column** — Logical starting address of the specified file

Figure cpu4231–5, Current Memory Allocation Screen

```
                    Current Memory Allocation

File No. Physical  Address Pid File Name                          Logical Offset
      1  00000000-00075fff   0 p0r0_4231                         00000000
      2  00076000-000e6fff   0 cpu4231.rnn                       00027000
      3  000e7000-008e6fff   0 support_4233                      e0000000
   ----  03ff4000-03ff5fff   0 pte2                              NA
   ----  03ff6000-03ffdfff   0 ptet                              NA
   ----  03ffe000-03ffefff   0 pte2                              NA
   ----  03fffc00-03ffffff   0 pte1                              NA
```

Class Descriptions

There are four different classes of subtests for *cpu4231*. The following sections describe the different classes and each of their subtests. Each section contains a table listing each subtest in that class, a description of the subtest, the subtest source code with comments (source file), the minimum time required to run the subtest (nominal time), and the timeout limit (maximum time). The object module for all *cpu4231* subtests is *cpu4231.rnn*.

> **NOTE**
>
> In the following tables, the **TEST PERFORMED** column lists either a description of what is tested or the instruction that is tested. For more information on an instruction's meaning, refer to the "Opcodes Sorted by Name" appendix within this manual or the *CONVEX Architecture Reference*.

Class 1 Subtests

This class of subtests verifies various instructions and features unique to the C200 Series architecture. Included are tests of system calls, C200 Series-specific non-vector instructions, thread-level addressing, exceptions, interval timers, and privileged instructions.

Table cpu4231–2, Class 1 Subtests

SUBTEST	TEST PERFORMED	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
10	*System calls*	*st_10.n*	0:01	0:05
20	*ldpa aj,ak*	*st_20.n*	0:01	0:05
21	*Patu instruction test*	*st_21.n*	0:01	0:05
22	*Pate aj instruction test*	*st_22.n*	0:01	0:05
23	*Thread addressing*	*st_23.n*	0:01	0:05
25	*Pich instruction test*	*st_25.n*	0:01	0:05
30	*Traps*	*st_30.n*	0:01	0:05
31	*Deadlocks*	*st_31.n*	0:01	0:05
32	*Unimplemented opcodes*	*st_32.n*	0:01	0:05
35	*Exceptions*	*st_35.n*	0:01	0:05
36	*Invalid cmr addresses*	*st_36.n*	0:01	0:05
37	*Trap instructions test*	*st_37.n*	0:01	0:05
40	*Interrupts*	*st_40.n*	0:01	0:05
41	*Interval timer test*	*st_41.n*	0:01	0:05
42	*Timer ring cross check*	*st_42.n*	0:01	0:05
43	*Timer CIR switch check*	*st_43.n*	0:01	0:05
45	*Privileged instruction check*	*st_45.n*	0:01	0:05
600	*Test vector valid test*	*st_600.n*	0:01	0:05
601	*Test mov sk,vv test*	*st_601.n*	0:01	0:05
602	*Test non-vector valid trapping just after setting vv*	*st_602.n*	0:01	0:05
604	*mov toc,sk*	*st_604.n*	0:01	0:05
700	*Ring crossings with trap bits set*	*st_700.n*	0:01	0:05
701	*Ring crossings with pbkpt bits set*	*st_701.n*	0:01	0:05

Class 2 Subtests

Class 2 subtests verify proper operation of page faults and non-resident calls, returns, and instructions. Class 2 subtests also verify proper operation of subroutine calls and returns for cases where the code and/or the stack are in non-resident memory.

Table cpu4231–3, Class 2 Subtests

SUBTEST	TEST PERFORMED	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
50	*Callq(abs/@) into non-resident page*	*st_50.n*	0:01	0:05
51	*Rtnq into non-resident page*	*st_51.n*	0:01	0:05
52	*Callq with non-resident stack*	*st_52.n*	0:01	0:05
53	*Rtnq with non-resident stack*	*st_53.n*	0:01	0:05
60	*Call (abs/@) into non-resident page*	*st_60.n*	0:01	0:05
61	*Rtn into non-resident page*	*st_61.n*	0:01	0:05
62	*Call(abs/@) with non-resident stack*	*st_62.n*	0:01	0:05
63	*Rtn with non-resident stack*	*st_63.n*	0:01	0:05
70	*Calls (abs/@) into non-resident page*	*st_70.n*	0:01	0:05
71	*Rtn into non-resident page*	*st_71.n*	0:01	0:05
72	*Calls with non-resident stack*	*st_72.n*	0:01	0:05
73	*Rtn with non-resident stack*	*st_73.n*	0:01	0:05
80	*Nr stack,nr target page (callq)*	*st_80.n*	0:01	0:05
81	*Nr stack,nr target page (calls)*	*st_81.n*	0:01	0:05
82	*Nr stack,nr target page (call)*	*st_82.n*	0:01	0:05
85	*Indirect nr, nr target page (callq)*	*st_85.n*	0:01	0:05
86	*Indirect nr, nr target page (calls)*	*st_86.n*	0:01	0:05
87	*Indirect nr, nr target page (call)*	*st_87.n*	0:01	0:05
90	*Indirect nr, nr stack (callq)*	*st_90.n*	0:01	0:05
91	*Indirect nr, nr stack (calls)*	*st_91.n*	0:01	0:05
92	*Indirect nr, nr stack (call)*	*st_92.n*	0:01	0:05
95	*Nr stack,nr return page (callq)*	*st_95.n*	0:01	0:05
96	*Nr stack,nr return page (calls)*		0:01	0:05
97	*Nr stack,nr return page (call)*	*st_97.n*	0:01	0:05
100	*Nr stack, nr target page, nr indirect address (callq)*	*st_100.n*	0:01	0:05
101	*Nr stack, nr target page, nr indirect address (calls)*	*st_101.n*	0:01	0:05
102	*Nr stack, nr target page, nr indirect address (call)*	*st_102.n*	0:01	0:05
200	*Halfword, nrpage execute*	*st_200.n*	0:01	0:05
201	*Word, nrpage execute*	*st_201.n*	0:01	0:05
202	*S-halfword,nrpage execute*	*st_202.n*	0:01	0:05
322	*Ip lookahead faults*	*st_322.n*	0:01	0:05

Class 3 Subtests

Class 3 subtests verify proper operation of memory operations (loads & stores) for various combinations of conditions (byte, halfword, word, longword, non-resident).

Table cpu4231–4, Class 3 Subtests

SUBTEST	TEST PERFORMED	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
210	*Byte loads nrpage*	*st_210.n*	0:01	0:05
220	*Load halfword nrpage*	*st_220.n*	0:01	0:05
221	*Load halfword nrpage*	*st_221.n*	0:01	0:05
222	*Load halfword nrpage*	*st_222.n*	0:01	0:05
223	*Load halfword nrpage*	*st_223.n*	0:01	0:05
224	*Load halfword nrpage*	*st_224.n*	0:01	0:05
225	*Load halfword nrpage*	*st_225.n*	0:01	0:05
226	*Load halfword nrpage*	*st_226.n*	0:01	0:05
227	*Load halfword nrpage*	*st_227.n*	0:01	0:05
228	*Load halfword nrpage*	*st_228.n*	0:01	0:05
230	*Load word nrpage*	*st_230.n*	0:01	0:05
231	*Load word nrpage*	*st_231.n*	0:01	0:05
232	*Load word nrpage*	*st_232.n*	0:01	0:05
233	*Load word nrpage*	*st_233.n*	0:01	0:05
234	*Load word nrpage*	*st_234.n*	0:01	0:05
235	*Load word nrpage*	*st_235.n*	0:01	0:05
236	*Load word nrpage*	*st_236.n*	0:01	0:05
237	*Load word nrpage*	*st_237.n*	0:01	0:05
238	*Load word nrpage*	*st_238.n*	0:01	0:05
240	*Load longword nrpage*	*st_240.n*	0:01	0:05
241	*Load longword nrpage*	*st_241.n*	0:01	0:05
242	*Load longword nrpage*	*st_242.n*	0:01	0:05
243	*Load longword nrpage*	*st_243.n*	0:01	0:05
244	*Load longword nrpage*	*st_244.n*	0:01	0:05
245	*Load longword nrpage*	*st_245.n*	0:01	0:05
246	*Load longword nrpage*	*st_246.n*	0:01	0:05
247	*Load longword nrpage*	*st_247.n*	0:01	0:05
248	*Load longword nrpage*	*st_248.n*	0:01	0:05
250	*Store byte nrpage*	*st_250.n*	0:01	0:05
260	*Store halfword nrpage*	*st_260.n*	0:01	0:05
261	*Store halfword nrpage*	*st_261.n*	0:01	0:05
270	*Store word nrpage*	*st_270.n*	0:01	0:05
271	*Store word nrpage*	*st_271.n*	0:01	0:05
272	*Store word nrpage*	*st_272.n*	0:01	0:05

Table cpu4231-4, Class 3 Subtests (continued)

SUBTEST	TEST PERFORMED	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
273	*Store word nrpage*	*st_273.n*	0:01	0:05
274	*Store word nrpage*	*st_274.n*	0:01	0:05
275	*Store word nrpage*	*st_275.n*	0:01	0:05
276	*Store word nrpage*	*st_276.n*	0:01	0:05
277	*Store word nrpage*	*st_277.n*	0:01	0:05
278	*Store word nrpage*	*st_278.n*	0:01	0:05
280	*Store longword nrpage*	*st_280.n*	0:01	0:05
281	*Store longword nrpage*	*st_281.n*	0:01	0:05
282	*Store longword nrpage*	*st_282.n*	0:01	0:05
283	*Store longword nrpage*	*st_283.n*	0:01	0:05
284	*Store longword nrpage*	*st_284.n*	0:01	0:05
285	*Store longword nrpage*	*st_285.n*	0:01	0:05
286	*Store longword nrpage*	*st_286.n*	0:01	0:05
287	*Store longword nrpage*	*st_287.n*	0:01	0:05
288	*Store longword nrpage*	*st_288.n*	0:01	0:05
290	*Load byte indirect, address non-resident*	*st_290.n*	0:01	0:05
291	*Load halfword indirect, address non-resident*	*st_291.n*	0:01	0:05
292	*Load word indirect, address non-resident*	*st_292.n*	0:01	0:05
293	*Load longword indirect, address non-resident*	*st_293.n*	0:01	0:05
300	*Store byte indirect, address non-resident*	*st_300.n*	0:01	0:05
301	*Store halfword indirect, address non-resident*	*st_301.n*	0:01	0:05
302	*Store word indirect, address non-resident*	*st_302.n*	0:01	0:05
303	*Store longword indirect, address non-resident*	*st_303.n*	0:01	0:05
305	*Load byte indirect, address and data non-resident*	*st_305.n*	0:01	0:05
306	*Load byte indirect, address and data non-resident*	*st_306.n*	0:01	0:05
307	*Load byte indirect, address and data non-resident*	*st_307.n*	0:01	0:05
308	*Load byte indirect, address and data non-resident*	*st_308.n*	0:01	0:05
310	*Store byte indirect, address and data non-resident*	*st_310.n*	0:01	0:05
311	*Store byte indirect, address and data non-resident*	*st_311.n*	0:01	0:05
312	*Store byte indirect, address and data non-resident*	*st_312.n*	0:01	0:05
313	*Store byte indirect, address and data non-resident*	*st_313.n*	0:01	0:05
320	*multiple fault test*	*st_320.n*	0:01	0:05

Class 4 Subtests

Class 4 subtests verify the proper operation of the various caches in the C200 Series architecture (pte cache, instruction cache, and data cache). These subtests also test remote invalidates between processors.

Table cpu4231–5, Class 4 Subtests

SUBTEST	TEST PERFORMED	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
400	*Pte cache test*	*st_400.n*	0:20	0:25
500	*Icache test I*	*st_500.n*	0:01	0:05
501	*Icache test II*	*st_501.n*	0:01	0:05
502	*Icache test III*	*st_502.n*	0:01	0:05
503	*Icache test IV*	*st_503.n*	0:01	0:05
504	*Icache test V*	*st_504.n*	0:01	0:05
505	*Icache test VI*	*st_505.n*	0:01	0:05
506	*Icache test VII*	*st_506.n*	0:01	0:05
510	*Dcache test I*	*st_510.n*	0:01	0:05
511	*Dcache test II*	*st_511.n*	0:01	0:05
512	*Dcache test III*	*st_512.n*	0:01	0:05
513	*Dcache test IV*	*st_513.n*	0:01	0:05
514	*Dcache test V*	*st_514.n*	0:01	0:05
515	*Dcache test VI*	*st_515.n*	0:01	0:05
516	*Dcache test VII*	*st_516.n*	0:01	0:05
517	*Dcache test VIII*	*st_517.n*	0:01	0:05
518	*Dcache test IX*	*st_518.n*	0:01	0:05
519	*Dcache test X*	*st_519.n*	0:01	0:05
560	*Remote Invalidate I*	*st_560.n*	0:20	0:35
565	*Remote Invalidate II*	*st_565.n*	0:20	0:35
570	*Remote Invalidate III*	*st_570.n*	0:20	0:35

Test Error Messages

For a common list of error messages that could result from running this test, refer to the appendix on CPU Error Messages.

SPU UNIX Error Messages

For a common list of possible error messages resulting from SPU UNIX, refer to the appendix on SPU UNIX Error Messages.

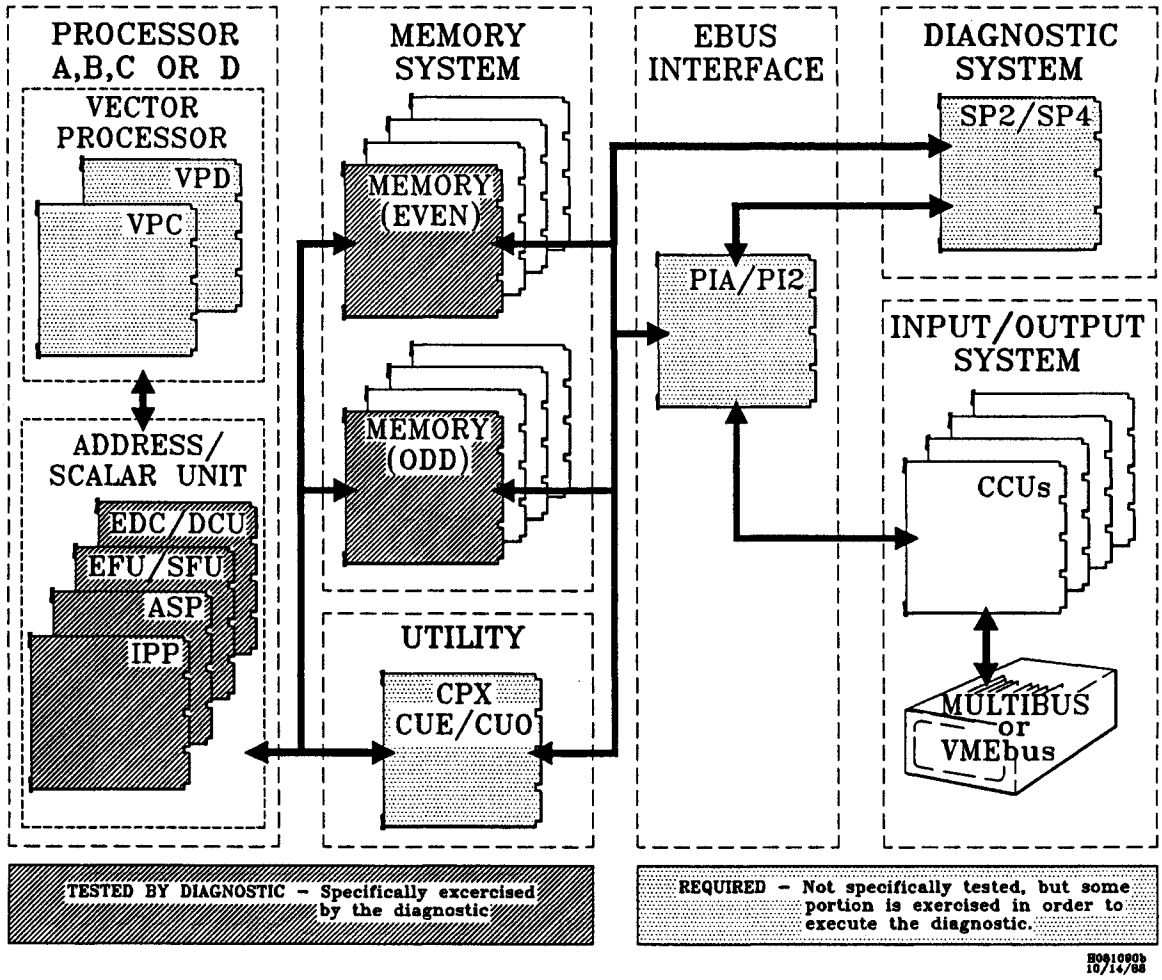
cpu4232

Enhanced, Non-vector, Uni-processor Instruction Tests

Overview

The *cpu4232* test is an extension of the building block test *cpu4030* for the CONVEX C200 Series machines. This test provides exhaustive, single CPU testing of the C200 Series scalar, address, and communication register instructions. Testing begins by first verifying the operation of the instructions which manipulate the communication registers. The locking/unlocking, putting/getting, sending/receiving, matching/testing, and incrementing of all eight communication register sets are verified to function correctly. Next, the sending/receiving, pushing/popping, matching/incrementing and testing-and-clearing operations on memory are verified. Third, the scalar instructions (scalar converts; loading/storing of the Communication Index Register (CIR), Thread Id Register (TID), Thread Timer Register (TTR); and loading of the Central Processing Unit IDentification (CPUID) are verified. Fourth, the multi-processor instructions *pfork*, *cfork*, *wfork*, *spawn*, *join*, and *idle* are verified. Last, the remaining instructions *trap*, *pbkpt*, *ldcmr*, *stcmr*, *ctrsl*, and *ctrsg* are verified. The following figure shows an overall view of what part of the system is being tested and which field replaceable units are required for the test to run.

Figure cpu4232–1, Functional Areas Tested by *cpu4232*



Prerequisites and Required Equipment

In order to run the *cpu4232* test, the Vector Processor Control (VPC) and the Vector Processor Data (VPD) must either be present in the machine or their slots must be terminated with terminators. Also, the boards listed in the following table must be operational. The table shows the tests used to verify the required boards. No additional equipment is required to run this test.

Table cpu4232–1, Required Functional Boards

BOARD	TEST TO VERIFY
Service Processor (SP2 or SP2)	*spu1000, spu4000*
Memory System	*mem4000*
CPU Utility Board(s) (CPX or CUE/CUO)	*cpx4000*
Instruction Processor Unit (IPP)	*cpu4030*
Address Scalar Processor (ASP)	*cpu4030*
Scalar Function Unit (SFU or EFU)	*cpu4030*
Data Cache Unit (DCU or EDC)	*cpu4030*
Vector Processor Control (VPC)	*cpu4041*
Vector Processor Data (VPD)	*cpu4041*

NOTE

The memory system consists of a minimum of one pair of memory boards (one even and one odd).

Test Invocation

To invoke the *cpu4232* test, use the procedure shown in the following figure. All responses in **boldface** are entered by the user. The prompts and responses appear sequentially on the screen, one line at a time. All prompts and responses are shown in one figure for convenience.

CAUTION

The invocation sequence shown in the following figure is the typical invocation sequence. However, the *initall* utility must be executed in some cases. If the system has just been powered up, if *mem4000* was executed with failures, or if *spu4000* was executed, then *initall* must be executed prior to any test execution. Failure to execute *initall* in these circumstances could result in invalid test results.

NOTE

Running the *initall* utility requires two to three minutes to execute depending on if the control stores have been previously loaded. However, it is suggested that *initall* be executed in the event that the state of the system is unknown.

Figure cpu4232–2, Test Invocation Sequence

```
(spu)>  cd /mnt/test
(spu)>  sysreset
(spu)>  dshell

CONVEX DIAGNOSTIC SHELL

:test cpu4232 [-c [class numeral(s)]] [-s [subtest numeral(s)]] [+> filename]
```

NOTE

After entering **dshell**, specific *dshell* parameters may be changed. Refer to the "Dshell and Iscan Overview" chapter of this manual for more information.

Entering only **test cpu4232** executes all *cpu4232* subtests sequentially. Execute a specific class(es) of subtest(s) or one or more individual subtests by using the *-c* or *-s* options, respectively. Detailed information for using these options can be found in the "Dshell and Iscan Overview" chapter of this manual. The [+>*filename*] option allows the test results to be appended to *filename*.

Test Parameter Menu

Once the test is invoked, a test parameter menu is presented allowing selection of default switches. If the test is run with all defaults invoked (answer **y** to the first prompt), no other prompts are provided. If the user answers **n** to the first prompt (run test without default switches), then a series of prompts are presented. The following figure shows all prompts, their possible answers (in brackets [ ]), and their default answers (in parentheses ( )). The prompts and responses in the following figure appear sequentially on the screen, one line at a time. All the prompts and responses are shown in one figure for convenience.

Figure cpu4232–3, *cpu4232* Test Parameter Menu

```
Test 'cpu4232.t'                                        Thu Nov 19 00:00:00 1965


                              ENTER TEST PARAMETERS

                    [ ]     Encloses allowed input ranges or values
                    ( )     Encloses the default value
                    ^       Returns to the previous prompt
                    :nn     Returns to the prompt # nn
                    :       Returns to the first unsatisfied prompt
                    :?      Reviews previous entries

 1:  Run default switches? [y,n]                                  (y) ->
 2:  CPUs to test: [ABCD]                                         (ABCD) ->
 3:  Forced Faulting Enabled? [y,n]                               (n) ->
 4:  Fault on Instruction Fetches? [y,n]                          (n) ->
 5:  Sequential Execution? [y,n]                                  (n) ->
 6:  Timeout Scale Factor Enabled? [1-100]                        (1) ->
 7:  Dcache Enabled? [y,n]                                        (y) ->
 8:  Segment of Execution? [0-7]                                  (0) ->
 9:  Chained Execution Mode? [y,n]                                (n) ->
 10: Loop Enabled? [y,n]                                          (n) ->
 11: Hard Errors Enabled? [y,n]                                   (y) ->
 12: Load CPU Code? [y,n]                                         (y) ->
```

NOTE

In the second prompt in the previous figure, ABCD represents all available CPUs. Only the available CPUs within the machine under test will be displayed as possible choices.

Prompt Explanations

A description of what each prompt means follows:

```
Run default switches? [y/n]                              (y) ->
```

If a response of **y** or <**CR**> is entered, no additional test parameter prompts are displayed, and

testing begins. However, if a negative response is supplied, additional test parameter prompts are displayed allowing changes to the default selections. Only enter the **n** response to restart a previous testing session at a specified point or to change any of the default parameters.

The following prompts are displayed and answered only if the first prompt is answered with **n**:

```
Cpus to test: [ABCD]                    (ABCD) ->
```

This prompt allows selection of the CPUs to be used in the test. The possible selection, represented by ABCD, will consist of all available CPUs. The default, ABCD, will consist of all available CPUs.

```
Forced Faulting Enabled? [y,n]                  (n) ->
```

If answered with **y**, normal force faulting occurs only on data references. The system forces a nonresident data exception to occur on every data reference. For a more detailed explanation of forced faulting, refer to the glossary appendix of this manual.

```
Fault on Instruction Fetches? [y,n]             (n) ->
```

If answered with **y**, force faulting occurs on instruction fetches in addition to data references. This prompt is displayed only if the previous prompt is answered with **y**.

```
Sequential Execution? [y,n]             (n) ->
```

If set by entering **y**, the sequential bit in the Processor Status Word (PSW) is set to forced sequential execution mode.

```
Timeout Scale Factor Enabled [1-100]            (1) ->
```

This prompt allows adjustment of the timeout factor. The normal timeout factor is multiplied by the number selected to increase the timeout factor. For example, if **5** is entered, the normal timeout factor is multiplied by 5 and it takes the test five times as long to timeout.

```
Dcache Enabled? [y,n]                           (y) ->
```

The Dcache is normally enabled; however, if it is suspected broken, it may be disabled by entering **n** at this prompt.

```
Segment of Execution? [0-7]             (0) ->
```

This prompt is only displayed if forced faults are not enabled. The segment of execution is contained in bits<31..29> of the Program Counter (PC). If **0** is entered, then bits<31..29> of the PC are 000 and the test is run in ring zero. If **1** is entered, then bits<31..29> of the PC are 001 and the test is run in ring one. If **2** is entered, then bits<31..29> of the PC are 010 and the test is run in ring two. If **3** is entered, then bits<31..29> of the PC are 011 and the test is run in ring three. If **4**, **5**, **6**, or **7** is entered, then bits<31..29> of the PC are 100, 101, 110, and 111, respectively, and the test is run in ring four. Refer to the *CONVEX Architecture Reference* for more information about the meaning of rings in the machine architecture.

```
Chained Execution Mode? [y,n]                    (n) ->
```

With this option enabled, the test is executed in chained mode which causes the CPU to perform subtest sequencing. The Service Processor is unaware of the action of the CPU unless a subtest fails or unless all of the subtests pass. If this option is enabled, test execution time will be greatly reduced. However, the only information printed to the console upon completion or failure (regardless of the cause) is the message, "Subtest 1 passed," or "Subtest 1 failed." Also, this option can not be enabled if the **-c** or the **-s** options were used in the invocation procedure.

```
Loop Enabled? [y,n]                              (n) ->
```

If **y** is entered, hard looping on a specific subtest is enabled. When looping is enabled, the halt instruction of the specified subtest is changed to branch back to the beginning of the subtest. This puts the subtest into an infinite loop which can be interrupted by entering **Ctrl-C**.

```
Hard Errors Enabled? [y,n]                       (y) ->
```

If this option is enabled and a parity error occurs, the clocks are stopped, and the test fails. If this option is disabled, parity errors and other sources of hard errors go undetected. It is recommended that hard errors normally be enabled.

```
Load CPU Code? [y,n]                             (y) ->
```

If the CPU code for this test is already in memory, the user can enter **n** for this prompt and the code is not reloaded (thus saving time).

When all prompts have been answered, the user is shown a test parameter summary displaying the prompts that have been answered. The following figure displays a sample test parameter summary. The actual summary varies according to answers given to the prompts.

Figure cpu4232–4, Sample Test Parameter Summary

```
                    TEST PARAMETER SUMMARY

Run default switches?                          :  n
Cpus to test:                                  :  ab
Forced Faulting Enabled?                       :  y
Fault on Instruction Fetches?                  :  y
Sequential Execution?                          :  n
Timeout Scale Factor Enabled?                  :  1
Dcache Enabled?                                :  y
Segment of Execution?                          :  0
Chained Execution Mode?                        :  n
Loop Enabled?                                  :  n
Hard Errors Enabled?                           :  y
Load CPU Code?                                 :  y
```

Hardware Initialization Sequence

After the last prompt is entered, and before test code execution, the following events occur:

> **NOTE**
>
> The first two events are accomplished at the initial start of the test. The remaining events are accomplished when each subtest is initialized.

- For each CPU, each module's page table entries are set up in main memory. Page table entries begin at the last available address in main memory and work toward low memory.
- For each CPU, each module is loaded into main memory following the logical to physical mapping described by the page tables. The exact logical to physical mapping that the modules are loaded into depends upon which segment of execution is selected by the user.
- The system is initialized (the memory system, CPX or CUE/CUO, PIA or PI2, and the CPU boards are reset).
- For each CPU, parity is initialized in the scalar processor by sending the scalar processor into a microcode routine and issuing clocks.
- Each CPU's scratch RAM is loaded with various values to control the execution of the test.
- For each CPU, the initial Program Counter (PC) is loaded into register T0 and the initial Program Status Word (PSW) is loaded into register T2.
- For each CPU, control store is initialized to the cold-start location.
- Clocks are turned on to the processor(s) selected, one at a time, in the order the CPU's are selected.

Memory Allocation

Immediately before test code execution, a current memory allocation screen is displayed. The following figure is an example of the current memory allocation screen. The physical and logical addresses shown, as well as the filenames, are only representative; the actual addresses will vary depending on installed memory & file sizes.

The following list explains what each column the current memory allocation screen depicts:

- **First Column** — File Number - useful in conjunction with the mm(1d) utility.
- **Second Column** — Physical memory addresses where the specified file is loaded
- **Third Column** — Process identification
- **Fourth Column** — File name (actual path is *./filename* or /mnt/test/CPU/*filename*). The entries `pte1`, `pte2`, and `ptet` are not actual files but are instead indications of the page tables.
- **Fifth Column** — Logical starting address of the specified file

Figure cpu4232–5, Current Memory Allocation Screen

```
                    Current Memory Allocation

File No.  Physical  Address Pid File Name                          Logical Offset
      1   00000000-00027fff   0 p0r0_4233                         00000000
      2   00028000-000b1fff   0 cpu4232.rnn                       00022000
      1   000b2000-000d9fff   1 p0r0_4233                         00000000
      2   000da000-00163fff   1 cpu4232.rnn                       00022000
   ----   03ff7000-03ff9fff   1 ptet                              NA
   ----   03ffa000-03ffafff   1 pte2                              NA
   ----   03ffb000-03ffdfff   0 ptet                              NA
   ----   03ffe000-03ffefff   0 pte2                              NA
   ----   03fffc00-3fffffff   1 pte1                              NA
```

Class Descriptions

There are five different classes of subtests for *cpu4232*. The following sections describe the different classes and each of their subtests. Each section contains a table listing each subtest in that class, a description of the subtest, the subtest executable test code (object module) and the subtest source code with comments (source file), the minimum time required to run the subtest (nominal time), and the worst case time (maximum time).

> **NOTE**
>
> In the following tables, the **INSTRUCTIONS TESTED** column lists each particular instruction that is tested. For more information on each instruction's meaning, refer to the "Opcodes Sorted by Name" appendix within this manual or the *CONVEX Architecture Reference*.

Class 1 Subtests

Class 1 subtests verify the ability of a single CPU to correctly manipulate each of the testable communication registers. For an explanation of valid manipulations on the communication registers, refer to the *CONVEX Architecture Reference*.

All subtests end with a halt command and store a halt code in address register A1. If the code stored in A1 is 0x100, then the subtest passed. Any other code in A1 indicates that the subtest failed. The following table lists the Class 1 subtests:

Table cpu4232–2, Class 1 Subtests

SUBTEST	INSTRUCTIONS TESTED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
100	*lck <Ceffa>*	*cpu4232.rnn*	*st_100.s*	0:01	0:10
105	*ulk <Ceffa>*	*cpu4232.rnn*	*st_105.s*	0:01	0:10
110	*put.w Ak,<Ceffa>*	*cpu4232.rnn*	*st_110.s*	0:01	0:10
115	*put.l Sk,<Ceffa>*	*cpu4232.rnn*	*st_115.s*	0:01	0:10
120	*get.w <Ceffa>,Ak*	*cpu4232.rnn*	*st_120.s*	0:01	0:10
125	*get.l <Ceffa>,Sk*	*cpu4232.rnn*	*st_125.s*	0:01	0:10
130	*snd.w Ak,<Ceffa>*	*cpu4232.rnn*	*st_130.s*	0:01	0:10
135	*snd.l Sk,<Ceffa>*	*cpu4232.rnn*	*st_135.s*	0:01	0:10
140	*rcv.w <Ceffa>,Ak*	*cpu4232.rnn*	*st_140.s*	0:01	0:10
145	*rcv.l <Ceffa>,Sk*	*cpu4232.rnn*	*st_145.s*	0:01	0:10
150	*mat.w Ak,<Ceffa>*	*cpu4232.rnn*	*st_150.s*	0:01	0:10
155	*mat.l Sk,<Ceffa>*	*cpu4232.rnn*	*st_155.s*	0:01	0:10
170	*tst <Ceffa>*	*cpu4232.rnn*	*st_170.s*	0:01	0:10
180	*inc.w <Ceffa>,Ak*	*cpu4232.rnn*	*st_180.s*	0:01	0:10
185	*inc.l <Ceffa>,Sk*	*cpu4232.rnn*	*st_185.s*	0:01	0:10

Class 2 Subtests

Class 2 subtests verifies the correctly manipulate the C200 Series memory structures in a single CPU environment. For an explanation of memory structures and how they can be manipulated, refer to the *CONVEX Architecture Reference*.

All subtests end with a halt command and store a halt code in address register A1. If the code stored in A1 is 0x100, then the subtest passed. Any other code in A1 indicates that the subtest failed. The following table lists the Class 2 subtests:

Table cpu4232–3, Class 2 Subtests

SUBTEST	INSTRUCTIONS TESTED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
131	*sndr.w Ak,<effa>*	*cpu4232.rnn*	*st_131.s*	0:01	0:10
136	*sndr.l Sk,<effa>*	*cpu4232.rnn*	*st_136.s*	0:01	0:10
141	*rcvr.w <effa>,Ak*	*cpu4232.rnn*	*st_141.s*	0:01	0:10
146	*rcvr.l <effa>,Sk*	*cpu4232.rnn*	*st_146.s*	0:01	0:10
160	*matm.w Ak,<effa>*	*cpu4232.rnn*	*st_160.s*	0:01	0:10
165	*matm.l Sk,<effa>*	*cpu4232.rnn*	*st_165.s*	0:01	0:10
175	*tac <effa>*	*cpu4232.rnn*	*st_175.s*	0:01	0:10
190	*pshr <effa>,Ak*	*cpu4232.rnn*	*st_190.s*	0:01	0:10
195	*popr Ak,<effa>*	*cpu4232.rnn*	*st_195.s*	0:01	0:10
200	*incr.w <effa>,Ak*	*cpu4232.rnn*	*st_200.s*	0:01	0:10
205	*incr.l <effa>,Sk*	*cpu4232.rnn*	*st_205.s*	0:01	0:10

Class 3 Subtests

Class 3 subtests verify the execution of the new C200 Series scalar instructions in a single CPU environment. For a detailed explanation of each of the new scalar instructions, refer to the *CONVEX Architecture Reference.*

All subtests end with a halt command and store a halt code in address register A1. If the code stored in A1 is 0x100, then the subtest passed. Any other code in A1 indicates that the subtest failed. The following table lists the Class 3 subtests:

Table cpu4232–4, Class 3 Subtests

SUBTEST	INSTRUCTIONS TESTED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
211	*mov Sk,TTR*	*cpu4232.rnn*	*st_211.s*	0:01	0:10
212	*mov TTR,Sk*	*cpu4232.rnn*	*st_212.s*	0:01	0:10
215	*mov CPUID,Sk*	*cpu4232.rnn*	*st_215.s*	0:01	0:10
225	*mov TID,Sk*	*cpu4232.rnn*	*st_225.s*	0:01	0:10
230	*mov Sk,TID*	*cpu4232.rnn*	*st_230.s*	0:01	0:10
290	*mov Sk,Cir*	*cpu4232.rnn*	*st_290.s*	0:01	0:10
295	*mov Cir,Sk*	*cpu4232.rnn*	*st_295.s*	0:01	0:10
305	*ldea <effa>,Sk*	*cpu4232.rnn*	*st_305.s*	0:01	0:10
310	*shf.w Sj,Sk*	*cpu4232.rnn*	*st_310.s*	0:01	0:10
400	*cvtd.w Sj,Sk (NATIVE)*	*cpu4232.rnn*	*st_400.s*	0:01	0:10
401	*cvtw.d Sj,Sk (NATIVE)*	*cpu4232.rnn*	*st_401.s*	0:01	0:10
410	*cvtd.w Sj,Sk (IEEE)*	*cpu4232.rnn*	*st_410.s*	0:01	0:10
411	*cvtw.d Sj,Sk (IEEE)*	*cpu4232.rnn*	*st_411.s*	0:01	0:10
420	*frint.s Sj,Sk (NATIVE)*	*cpu4232.rnn*	*st_420.s*	0:01	0:10
425	*frint.d Sj,Sk (NATIVE)*	*cpu4232.rnn*	*st_425.s*	0:01	0:10
500	*sqrt.s Sk (NATIVE)*	*cpu4232.rnn*	*st_500.s*	0:01	0:10
510	*atan.s Sk (NATIVE)*	*cpu4232.rnn*	*st_510.s*	0:01	0:10
520	*exp.s Sk (NATIVE)*	*cpu4232.rnn*	*st_520.s*	0:01	0:10
530	*ln.s Sk (NATIVE)*	*cpu4232.rnn*	*st_580.s*	0:01	0:10
540	*sin.s Sk (NATIVE)*	*cpu4232.rnn*	*st_540.s*	0:01	0:10
550	*cos.s Sk (NATIVE)*	*cpu4232.rnn*	*st_550.s*	0:01	0:10
600	*sqrt.d Sk (NATIVE)*	*cpu4232.rnn*	*st_600.s*	0:01	0:10
610	*atan.d Sk (NATIVE)*	*cpu4232.rnn*	*st_610.s*	0:01	0:10
620	*exp.d Sk (NATIVE)*	*cpu4232.rnn*	*st_620.s*	0:01	0:10
630	*ln.d Sk (NATIVE)*	*cpu4232.rnn*	*st_680.s*	0:01	0:10
640	*sin.d Sk (NATIVE)*	*cpu4232.rnn*	*st_640.s*	0:01	0:10
650	*cos.d Sk (NATIVE)*	*cpu4232.rnn*	*st_650.s*	0:01	0:10
1420	*frint.s Sj,Sk (IEEE)*	*cpu4232.rnn*	*st_1420.s*	0:01	0:10
1425	*frint.d Sj,Sk (IEEE)*	*cpu4232.rnn*	*st_1425.s*	0:01	0:10
1500	*sqrt.s Sk (IEEE)*	*cpu4232.rnn*	*st_1500.s*	0:01	0:10
1510	*atan.s Sk (IEEE)*	*cpu4232.rnn*	*st_1510.s*	0:01	0:10
1520	*exp.s Sk (IEEE)*	*cpu4232.rnn*	*st_1520.s*	0:01	0:10
1530	*ln.s Sk (IEEE)*	*cpu4232.rnn*	*st_1580.s*	0:01	0:10
1540	*sin.s Sk (IEEE)*	*cpu4232.rnn*	*st_1540.s*	0:01	0:10
1550	*cos.s Sk (IEEE)*	*cpu4232.rnn*	*st_1550.s*	0:01	0:10
1600	*sqrt.d Sk (IEEE)*	*cpu4232.rnn*	*st_1600.s*	0:01	0:10
1610	*atan.d Sk (IEEE)*	*cpu4232.rnn*	*st_1610.s*	0:01	0:10
1620	*exp.d Sk (IEEE)*	*cpu4232.rnn*	*st_1620.s*	0:01	0:10
1630	*ln.d Sk (IEEE)*	*cpu4232.rnn*	*st_1680.s*	0:01	0:10
1640	*sin.d Sk (IEEE)*	*cpu4232.rnn*	*st_1640.s*	0:01	0:10
1650	*cos.d Sk (IEEE)*	*cpu4232.rnn*	*st_1650.s*	0:01	0:10

Class 4 Subtests

Class 4 subtests verify the operation of the C200 Series process control instructions execution in a single CPU environment. For a detailed description of what the process control instructions are, and how they are required to function, refer to the *CONVEX Architecture Reference.*

All subtests end with a halt command and store a halt code in address register A1. If the code stored in A1 is 0x100, then the subtest passed. Any other code in A1 indicates that the subtest failed. The following table lists the Class 4 subtests:

Table cpu4232–5, Class 4 Subtests

SUBTEST	INSTRUCTIONS TESTED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
250	*pfork <effa>,Ak*	*cpu4232.rnn*	*st_205.s*	0:01	0:10
253	*spawn <effa>,Ak*	*cpu4232.rnn*	*st_258.s*	0:01	0:10
255	*cfork*	*cpu4232.rnn*	*st_255.s*	0:01	0:10
260	*wfork*	*cpu4232.rnn*	*st_260.s*	0:01	0:10
263	*join*	*cpu4232.rnn*	*st_263.s*	0:01	0:10
265	*idle Sk*	*cpu4232.rnn*	*st_265.s*	0:01	0:10

Class 5 Subtests

Class 5 subtests verify the operation of miscellaneous instructions in a single CPU environment. Specifically, process trapping instructions, loading and storing of the communication registers, and the CPU execution timer synchronization instructions are verified to function correctly in a single CPU environment.

All subtests end with a halt command and store a halt code in address register A1. If the code stored in A1 is 0x100, then the subtest passed. Any other code in A1 indicates that the subtest failed. The following table lists the Class 5 subtests:

Table cpu4232–6, Class 5 Subtests

SUBTEST	INSTRUCTIONS TESTED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
220	*trap #rm,#b*	*cpu4232.rnn*	*st_220.s*	0:01	0:10
235	*pbkpt*	*cpu4232.rnn*	*st_235.s*	0:01	0:10
240	*ctrsl*	*cpu4232.rnn*	*st_240.s*	0:01	0:10
245	*ctrsg*	*cpu4232.rnn*	*st_245.s*	0:01	0:10
270	*ldcmr <effa>,Ak*	*cpu4232.rnn*	*st_270.s*	0:01	0:45
275	*stcmr <effa>,Ak*	*cpu4232.rnn*	*st_275.s*	0:01	0:80
300	*cmr modify bit test*	*cpu4232.rnn*	*st_300.s*	0:01	0:20

Test Error Messages

For a common list of error messages that could result from running this test, refer to the appendix on CPU Error Messages.

SPU UNIX Error Messages

For a common list of possible error messages resulting from SPU UNIX, refer to the appendix on SPU UNIX Error Messages.

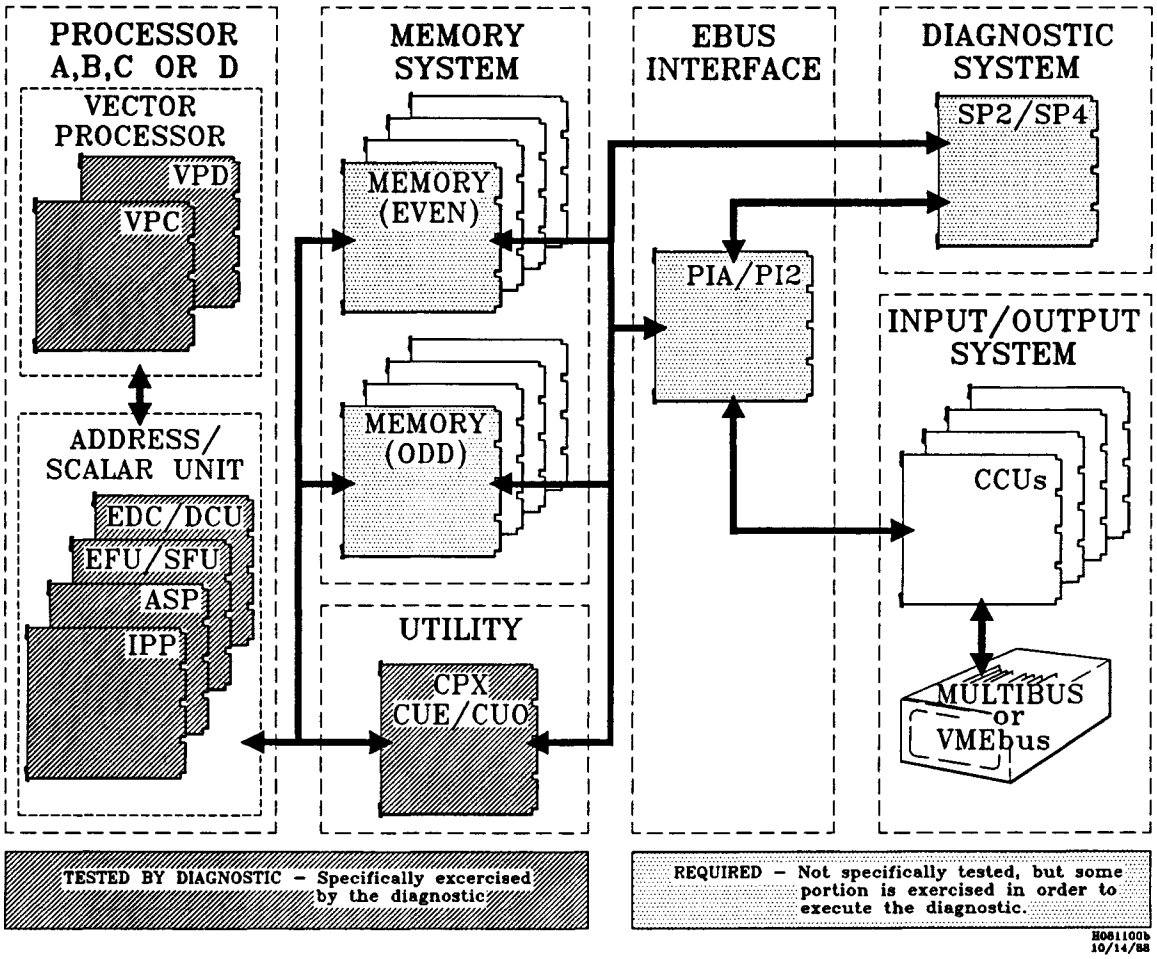
cpu4233

Multiprocessor Diagnostics

Overview

The *cpu4233* test verifies the operation of the CPU complex in a multi-headed environment. Included are tests for concurrent access and use of communication registers, memory, thread creation and termination instructions, interrupts, CPU execution timers, privileged instructions, and exceptions. Correct execution for processors executing in both the same and in different Communication Index Registers (CIRs) is also verified.

Figure cpu4233–1, Functional Areas Tested by *cpu4233*



Prerequisites and Required Equipment

In order to run the *cpu4232* test, the Vector Processor Control (VPC) and the Vector Processor Data (VPD) must either be present in the machine or their slots must be terminated with terminators. Also, the boards listed in the following table must be operational. The table shows the tests used to verify the required boards. No additional equipment is required to run this test.

Table cpu4233–1, Required Functional Boards

BOARD	TEST TO VERIFY
Service Processor (SP2 or SP4)	*spu1000, spu4000*
PBUS Interface Adapter (PIA or PI2)	*pia4000* or *pi2_4000*
Memory System	*mem4000*
Instruction Processor Unit (IPP)	*cpu4030*
Address Scalar Processor (ASP)	*cpu4030*
Scalar Function Unit (SFU or EFU)	*cpu4030*
Data Cache Unit (DCU or EDC)	*cpu4030*
Vector Processor Control (VPC)	*cpu4041*
Vector Processor Data (VPD)	*cpu4041*
CPU Utility Board(s) (CPX or CUE/CUO)	*cpx4000*

NOTE

The memory system consists of a minimum of one pair of memory boards (one even and one odd).

Test Invocation

To invoke the *cpu4233* test, use the procedure shown in the following figure. All responses in **boldface** are entered by the user. The prompts and responses appear sequentially on the screen, one line at a time. All prompts and responses are shown in one figure for convenience.

CAUTION

The invocation sequence shown in the following figure is the typical invocation sequence. However, the *initall* utility must be executed in some cases. If the system has just been powered up, if *mem4000* was executed with failures, or if *spu4000* was executed, then *initall* must be executed prior to any test execution. Failure to execute *initall* in these circumstances could result in invalid test results.

NOTE

Running the *initall* utility requires two to three minutes to execute depending on if the control stores have been previously loaded. However, it is suggested that *initall* be executed in the event that the state of the system is unknown.

Figure cpu4233–2, Test Invocation Sequence

```
(spu)>  cd /mnt/test
(spu)>  sysreset
(spu)>  dshell

CONVEX DIAGNOSTIC SHELL

:test cpu4233 [-c [class numeral(s)]] [-s [subtest numeral(s)]] [+> filename]
```

NOTE

After entering **dshell**, specific *dshell* parameters may be changed. Refer to the "Dshell and Iscan Overview" chapter of this manual for more information.

Entering only **test cpu4233** executes all *cpu4233* subtests sequentially. Execute a specific class(es) of subtest(s) or one or more individual subtests by using the *-c* or *-s* options, respectively. Detailed information for using these options can be found in the "Dshell and Iscan Overview" chapter of this manual. The [+>*filename*] option allows the test results to be appended to *filename*.

Test Parameter Menu

Once the test is invoked, a test parameter menu is presented allowing selection of default switches. If the test is run with all defaults invoked (answer **y** to the first prompt), no other prompts are provided. If the user answers **n** to the first prompt (run test without default switches), then a series of prompts are presented. The following figure shows all prompts, their possible answers (in brackets [ ]), and their default answers (in parentheses ( )). The prompts and responses in the following figure appear sequentially on the screen, one line at a time. All the prompts and responses are shown in one figure for convenience.

Figure cpu4233–3, Test Parameter Menu

```
Test 'cpu4233.t'                                    Thu Nov 19 00:00:00 1965

                          ENTER TEST PARAMETERS

                [ ]     Encloses allowed input ranges or values
                ( )     Encloses the default value
                ^       Returns to the previous prompt
                :nn     Returns to the prompt # nn
                :       Returns to the first unsatisfied prompt
                :?      Reviews previous entries


1:  Run default switches? [y,n]                              (y) ->
2:  Cpus to test: [ABCD]                                     (ABCD) ->
3:  Forced Faulting Enabled? [y,n]                           (n) ->
4:  Fault on Instruction Fetches? [y,n]                      (n) ->
5:  Sequential Execution? [y,n]                              (n) ->
6:  Timeout Scale Factor Enabled? [1-100]                    (1) ->
7:  Dcache Enabled? [y,n]                                    (y) ->
8:  Segment of Execution? [0-7]                              (0) ->
9:  Loop Enabled? [y,n]                                      (n) ->
10: Hard Errors Enabled? [y,n]                               (y) ->
11: Load CPU Code? [y,n]                                     (y) ->
```

NOTE

In the second prompt in the previous figure, ABCD represents all available CPUs. Only the available CPUs within the machine under test will be displayed as possible choices. The fourth prompt in the previous list is only supplied when the fifth prompt is answered with **y**.

Prompt Explanations

A description of the meaning of each prompt follows:

```
Run default switches? [y/n]                    (y) ->
```

If a response of **y** or **<CR>** is given, no additional test parameter prompts are displayed and testing begins. However, if a negative response is supplied, additional test parameter prompts are displayed allowing modification of the default selections.

The following prompts are only displayed and answered if the first prompt is answered with **n**:

```
Cpus to test: [ABCD]                          (ABCD) ->
```

This prompt allows selection of the CPUs to be used in the test. The possible selection, represented by ABCD, will consist of all available CPUs. The default, ABCD, will consist of all available CPUs. The first CPU in the list will be the *master* CPU for this test.

```
Forced Faulting Enabled? [y,n]                 (n) ->
```

If answered with **y**, normal force faulting occurs only on data references. The system will force a non-resident data exception to occur on every data reference. For a more detailed explanation of forced faulting, refer to the glossary appendix of this manual.

```
Fault on Instruction Fetches? [y,n]            (n) ->
```

In answered with **y**, force faulting occurs on instruction fetches in addition to data references. This prompt is only supplied to the user if the previous prompt is answered with **y**.

```
Sequential Execution? [y,n]                    (n) ->
```

If set by entering **y**, the sequential bit in the PSW will be set to forced sequential execution mode.

```
Timeout Scale Factor [1-100]                   (1) ->
```

This prompt allows adjustment of the timeout factor. The normal timeout factor is multiplied by the number selected to increase the timeout factor. For example, if **5** is entered, the normal timeout factor is multiplied by 5 and it will take the test five times as long to timeout.

```
Dcache Enabled? [y,n]                          (y) ->
```

The Dcache is normally enabled; however, if it is suspected broken, it can be disabled by entering **n** at this prompt.

```
Segment of Execution? [0-7]                    (0) ->
```

This prompt is only displayed if forced faulting is not enabled. The segment of execution is contained in bits<31..29> of the Program Counter (PC). If **0** is entered, then bits<31..29> of the PC are 000 and the test is run in ring zero. If **1** is entered, then bits<31..29> of the PC are 001 and the test is run in ring one. If **2** is entered, then bits<31..29> of the PC are 010 and the test is run in ring two. If **3** is entered, then bits<31..29> of the PC are 011 and the test is run in ring three. If **4**, **5**, **6**, or **7** is entered, then bits<31..29> of the PC are 100, 101, 110,

and 111 respectively and the test is run in ring four. Refer to the *CONVEX Architecture Reference* for more information concerning the meaning of rings in the machine architecture.

```
Loop Enabled? [y,n]                                    (n) ->
```

If **y** is entered, hard looping on a specific subtest is enabled. When looping is enabled, the halt instruction of the specified subtest is changed to a branch back to the beginning of the subtest. This puts the subtest into an infinite loop which the user must break out of by typing **Ctrl- C**.

```
Hard Errors Enabled? [y,n]                  (y) ->
```

If this option is enabled and a hard error occurs, the clocks will be stopped and the test will fail. If this option is disabled, parity errors and other sources of hard errors will go undetected. It is recommended that hard errors normally be enabled.

```
Load CPU Code? [y,n]                        (y) ->
```

If the CPU code for this test is already in memory, the user can enter **n** for this prompt and the code will not be reloaded (thus saving time).

When all prompts have been answered, a test parameter summary echos the prompts that have been answered. The following figure displays a sample test parameter summary. The actual summary varies depending on the answers to the prompts.

Figure cpu4233–4, Sample Test Parameter Summary

```
                    TEST PARAMETER SUMMARY

Run default switches?                    :  n
Cpus to test:                            :  ab
Parallel Test Execution?                 :  y
Forced Faulting Enabled?                 :  y
Sequential Execution?                    :  n
Timeout Scale Factor Enabled?            :  1
Dcache Enabled?                          :  y
Segment of Execution?                    :  0
Loop Enabled?                            :  n
Hard Errors Enabled?                     :  y
Load CPU Code?                           :  y
```

Hardware Initialization Sequence

After the last prompt is entered, and before test code execution, the following events occur:

> **NOTE**
>
> The first two events are accomplished at the initial start of the test. The remaining events are accomplished when each subtest is initialized.

- Each module's page table entries are set up in main memory. Page table entries begin at the last available address in main memory and work toward low memory.
- Each module is loaded into main memory following the logical to physical mapping described by the page tables. The exact logical to physical mapping that the modules are loaded into depends upon which segment of execution is selected by the user.
- The system is initialized (the memory system, CPX or CUE/CUO, PIA or PI2, and the CPU boards are reset).
- For each CPU, parity is initialized in the scalar processor by sending the scalar processor into a microcode routine and issuing clocks.
- Each CPU's scratch RAM is loaded with various values to control the execution of the test.
- For the *master* CPU, the initial Program Counter (PC) is loaded into register T0 and the initial Program Status Word (PSW) is loaded into register T2.
- For the *master* CPU, control store is initialized to the cold-start location. Other heads are placed in the idle loop.
- Clocks are turned on to all processors selected.

Memory Allocation

Immediately before test code execution, a current memory allocation screen is displayed. The following figure is an example of the current memory allocation screen. The physical and logical addresses shown, as well as the filenames, are only representative; the actual addresses will vary depending on installed memory & file sizes.

The following list explains what each column the current memory allocation screen depicts:

- **First Column** — File Number - useful in conjunction with the mm(1d) utility.
- **Second Column** — Physical memory addresses where the specified file is loaded
- **Third Column** — Process identification
- **Fourth Column** — File name (actual path is *./filename* or /mnt/test/CPU/*filename*). The entries `pte1`, `pte2`, and `ptet` are not actual files but are instead indications of the page tables.
- **Fifth Column** — Logical starting address of the specified file

Figure cpu4233–5, Current Memory Allocation Screen

```
                    Current Memory Allocation
File No. Physical  Address Pid File Name                          Logical Offset
      1  00000000-00075fff   0 p0r0_4233                         00000000
      2  00076000-000e6fff   0 cpu4233.rnn                       00027000
      3  000e7000-008e6fff   0 support_4233                      e0000000
   ----  03ff4000-03ff5fff   0 pte2                              NA
   ----  03ff6000-03ffdfff   0 ptet                              NA
   ----  03ffe000-03ffefff   0 pte2                              NA
   ----  03fffc00-03ffffff   0 pte1                              NA
```

NOTE

The previous figure is an example only. Sizes, offsets, and filenames may vary from release to release and on a specified segment of execution.

Class Descriptions

There are fourteen different classes of subtests for *cpu4233* which are listed in the following table. A description of each class of subtests is also given.

Table cpu4233–2, Class Descriptions

CLASS	DESCRIPTION
1	Communication register single headed tests
2	Communication register multi-headed tests (sync'd stepped, same cmr)
3	Communication register multi-headed tests (sync'd, same cmr)
4	Communication register multi-headed tests (sync'd, different same cmr)
5	Memory instruction singled headed tests
6	Memory instruction multi-headed tests (sync'd stepped, same cmr)
7	Memory instruction multi-headed tests (sync'd, same cmr)
8	Memory instruction multi-headed tests (sync'd, different same cmr)
9	Multi-headed parallel instruction tests
10	Communication register loading and storing tests
11	Multi_headed *trap* and *pbkpt* instruction tests
12	Multi-headed remote invalidates (testing same blocks)
13	Multi-headed remote invalidates (testing different memory blocks)
14	Multi-headed timer, interrupt, and exception processing test

Subtests

The following tables list each *cpu4233* subtest in its order of execution. The tables list each subtest, its class, a description of the subtest, the subtest source code with comments (source file), the minimum time required to run the subtest (nominal time), and the timeout limit (maximum time). The object module for all *cpu4233* subtests is *cpu4233.rnn*.

> **NOTE**
>
> In the following tables, the **TEST PERFORMED** column lists either a description of what is tested or the instruction that is tested. For more information on an instruction's meaning, refer to the "Opcodes Sorted by Name" appendix within this manual or the *CONVEX Architecture Reference*.

Table cpu4233–3, Subtests

SUBTEST	CLASS	TEST PERFORMED	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
101	1	*lck <Ceffa> (single head)*	*st_101.s*	0:02	0:10
102	2	*lck <Ceffa> (multiple heads, sync'd, stepped on same cmrs)*	*st_102.s*	0:01	0:10
103	3	*lck <Ceffa> (multiple heads, sync'd on same cmr locations)*	*st_103.s*	0:02	0:10
104	4	*lck <Ceffa> (multiple heads, sync'd on different cmrs)*	*st_104.s*	0:02	0:10
111	1	*ulk <Ceffa> (single head)*	*st_111.s*	0:01	0:10
112	2	*ulk <Ceffa> (multiple heads, sync'd, stepped on same cmrs)*	*st_112.s*	0:02	0:10
113	3	*ulk <Ceffa> (multiple heads, sync'd on same cmr locations)*	*st_113.s*	0:02	0:10
114	4	*ulk <Ceffa> (multiple heads, sync'd on different cmrs)*	*st_114.s*	0:01	0:10
121	1	*tst <Ceffa> (single head)*	*st_121.s*	0:01	0:10
122	2	*tst <Ceffa> (multiple heads, sync'd, stepped on same cmrs)*	*st_122.s*	0:02	0:10
123	3	*tst <Ceffa> (multiple heads, sync'd on same cmr locations)*	*st_123.s*	0:02	0:10
124	4	*tst <Ceffa> (multiple heads, sync'd on different cmrs)*	*st_124.s*	0:01	0:10
131	1	*get.w <Ceffa>,Ak (single head)*	*st_131.s*	0:02	0:10
132	2	*get.w <Ceffa>,Ak (multiple heads, sync'd, stepped on same cmrs)*	*st_132.s*	0:02	0:10
133	3	*get.w <Ceffa>,Ak (multiple heads, sync'd on same cmr locations)*	*st_133.s*	0:01	0:10
134	4	*get.w <Ceffa>,Ak (multiple heads, sync'd on different cmrs)*	*st_134.s*	0:01	0:10
141	1	*get.l <Ceffa>,Sk (single head)*	*st_141.s*	0:02	0:10

Table cpu4233-3, Subtests (continued)

SUBTEST	CLASS	TEST PERFORMED	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
142	2	*get.l <Ceffa>,Sk (multiple heads, sync'd, stepped on same cmrs)*	*st_142.s*	0:01	0:10
143	3	*get.l <Ceffa>,Sk (multiple heads, sync'd on same cmr locations)*	*st_143.s*	0:01	0:10
144	4	*get.l <Ceffa>,Sk (multiple heads, sync'd on different cmrs)*	*st_144.s*	0:02	0:10
151	1	*put.w Ak,<Ceffa> (single head)*	*st_151.s*	0:01	0:10
152	2	*put.w Ak,<Ceffa> (multiple heads, sync'd, stepped on same cmrs)*	*st_152.s*	0:01	0:10
153	3	*put.w Ak,<Ceffa> (multiple heads, sync'd on same cmr locations)*	*st_153.s*	0:02	0:10
154	4	*put.w Ak,<Ceffa> (multiple heads, sync'd on different cmrs)*	*st_154.s*	0:02	0:10
161	1	*put.l Sk,<Ceffa> (single head)*	*st_161.s*	0:01	0:10
162	2	*put.l Sk,<Ceffa> (multiple heads, sync'd, stepped on same cmrs)*	*st_162.s*	0:02	0:10
163	3	*put.l Sk,<Ceffa> (multiple heads, sync'd on same cmr locations)*	*st_163.s*	0:02	0:10
164	4	*put.l Sk,<Ceffa> (multiple heads, sync'd on different cmrs)*	*st_164.s*	0:01	0:10
171	1	*rcv.w <Ceffa>,Ak (single head)*	*st_171.s*	0:01	0:10
172	2	*rcv.w <Ceffa>,Ak (multiple heads, sync'd, stepped on same cmrs)*	*st_172.s*	0:02	0:10
173	3	*rcv.w <Ceffa>,Ak (multiple heads, sync'd on same cmr locations)*	*st_173.s*	0:01	0:10
174	4	*rcv.w <Ceffa>,Ak (multiple heads, sync'd on different cmrs)*	*st_174.s*	0:01	0:10
181	1	*rcv.l <Ceffa>,Sk (single head)*	*st_181.s*	0:02	0:10

Table cpu4233-3, Subtests (continued)

SUBTEST	CLASS	TEST PERFORMED	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
182	2	*rcv.l <Ceffa>,Sk (multiple heads, sync'd, stepped on same cmrs)*	*st_182*	s:01	0:10
183	3	*rcv.l <Ceffa>,Sk (multiple heads, sync'd on same cmr locations)*	*st_183.s*	0:01	0:10
184	4	*rcv.l <Ceffa>,Sk (multiple heads, sync'd on different cmrs)*	*st_184.s*	0:01	0:10
191	1	*snd.w Ak,<Ceffa> (single head)*	*st_191.s*	0:02	0:10
192	2	*snd.w Ak,<Ceffa> (multiple heads, sync'd, stepped on same cmrs)*	*st_192.s*	0:01	0:10
193	3	*snd.w Ak,<Ceffa> (multiple heads, sync'd on same cmr locations)*	*st_193.s*	0:01	0:10
194	4	*snd.w Ak,<Ceffa> (multiple heads, sync'd on different cmrs)*	*st_194.s*	0:02	0:10
201	1	*snd.l Sk,<Ceffa> (single head)*	*st_201.s*	0:02	0:10
202	2	*snd.l Sk.<Ceffa> (multiple heads, sync'd, stepped on same cmrs)*	*st_202.s*	0:01	0:10
203	3	*snd.l Sk,<Ceffa> (multiple heads, sync'd on same cmr locations)*	*st_203.s*	0:01	0:10
204	4	*snd.l Sk,<Ceffa> (multiple heads, sync'd on different cmrs)*	*st_204.s*	0:02	0:10
211	1	*inc.w <Ceffa>,Ak (single head)*	*st_211.s*	0:01	0:10
212	2	*inc.w <Ceffa>,Ak (multiple heads, sync'd, stepped on same cmrs)*	*st_212.s*	0:01	0:10
213	3	*inc.w <Ceffa>,Ak (multiple heads, sync'd on same cmr locations)*	*st_213.s*	0:02	0:10
214	4	*inc.w <Ceffa>,Ak (multiple heads, sync'd on different cmrs)*	*st_214.s*	0:02	0:10
221	1	*inc.l <Ceffa>,Sk (single head)*	*st_221.s*	0:01	0:10

Table cpu4233-3, Subtests (continued)

SUBTEST	CLASS	TEST PERFORMED	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
222	2	*inc.l <Ceffa>,Sk (multiple heads, sync'd, stepped on same cmrs)*	*st_222.s*	0:01	0:10
223	3	*inc.l <Ceffa>,Sk (multiple heads, sync'd on same cmr locations)*	*st_223.s*	0:02	0:10
224	4	*inc.l <Ceffa>,Sk (multiple heads, sync'd on different cmrs)*	*st_224.s*	0:02	0:10
231	1	*mat.w Ak,<Ceffa> (single head)*	*st_231.s*	0:01	0:10
232	2	*mat.w Ak,<Ceffa> (multiple heads, sync'd, stepped on same cmrs)*	*st_232.s*	0:02	0:10
233	3	*mat.w Ak,<Ceffa> (multiple heads, sync'd on same cmr locations)*	*st_233.s*	0:02	0:10
234	4	*mat.w Ak,<Ceffa> (multiple heads, sync'd on different cmrs)*	*st_234.s*	0:01	0:10
241	1	*mat.l Sk,<Ceffa> (single head)*	*st_241.s*	0:01	0:10
242	2	*mat.l Sk,<Ceffa> (multiple heads, sync'd, stepped on same cmrs)*	*st_242.s*	0:02	0:10
243	3	*mat.l Sk,<Ceffa> (multiple heads, sync'd on same cmr locations)*	*st_243.s*	0:02	0:10
244	4	*mat.l Sk,<Ceffa> (multiple heads, sync'd on different cmrs)*	*st_244.s*	0:01	0:10
251	5	*sndr.w Ak,<effa> (single head)*	*st_251.s*	0:01	0:10
252	6	*sndr.w Ak,<effa> (multiple heads, sync'd, stepped on same resource)*	*st_252.s*	0:02	0:10
253	7	*sndr.w Ak,<effa> (multiple heads, sync'd on same resources)*	*st_253.s*	0:02	0:10
254	8	*sndr.w Ak,<effa> (multiple heads, sync'd on different resources)*	*st_254.s*	0:01	0:10
261	5	*sndr.l Sk,<effa> (single head)*	*st_261.s*	0:01	0:10

Table cpu4233-3, Subtests (continued)

SUBTEST	CLASS	TEST PERFORMED	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
262	6	*sndr.l Sk,<effa> (multiple heads, sync'd, stepped on same resource)*	*st_262.s*	0:02	0:10
263	7	*sndr.l Sk,<effa> (multiple heads, sync'd on same resources)*	*st_263.s*	0:02	0:10
264	8	*sndr.l Sk,<effa> (multiple heads, sync'd on different resources)*	*st_264.s*	0:01	0:10
271	5	*rcvr.w <effa>,Ak (single head)*	*st_271.s*	0:02	0:10
272	6	*rcvr.w <effa>,Ak (multiple heads, sync'd, stepped on same resource)*	*st_272.s*	0:02	0:10
273	7	*rcvr.w <effa>,Ak (multiple heads, sync'd on same resources)*	*st_273.s*	0:01	0:10
274	8	*rcvr.w <effa>,Ak (multiple heads, sync'd on different resources)*	*st_274.s*	0:01	0:10
281	5	*rcvr.l <effa>,Sk (single head)*	*st_281.s*	0:02	0:10
282	6	*rcvr.l <effa>,Sk (multiple heads, sync'd, stepped on same resource)*	*st_282.s*	0:02	0:10
283	7	*rcvr.l <effa>,Sk (multiple heads, sync'd on same resources)*	*st_283.s*	0:01	0:10
284	8	*rcvr.l <effa>,Sk (multiple heads, sync'd on different resources)*	*st_284.s*	0:01	0:10
291	5	*matm.w Ak,<effa> (single head)*	*st_291.s*	0:02	0:10
292	6	*matm.w Ak,<effa> (multiple heads, sync'd, stepped on same resource)*	*st_292.s*	0:02	0:10
293	7	*matm.w Ak,<effa> (multiple heads, sync'd on same resources)*	*st_293.s*	0:01	0:10
294	8	*matm.w Ak,<effa> (multiple heads, sync'd on different resources)*	*st_294.s*	0:02	0:10
301	5	*matm.l Sk,<effa> (single head)*	*st_301.s*	0:02	0:10

Table cpu4233-3, Subtests (continued)

SUBTEST	CLASS	TEST PERFORMED	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
302	6	*matm.l Sk,<effa> (multiple heads, sync'd, stepped on same resource)*	*st_302.s*	0:02	0:10
303	7	*matm.l Sk,<effa> (multiple heads, sync'd on same resources)*	*st_303.s*	0:01	0:10
304	8	*matm.l Sk,<effa> (multiple heads, sync'd on different resources)*	*st_304.s*	0:02	0:10
311	5	*pshr Ak,<effa> (single head)*	*st_311.s*	0:02	0:10
312	6	*pshr Ak,<effa> (multiple heads, sync'd, stepped on same resource)*	*st_312.s*	0:02	0:10
313	7	*pshr Ak,<effa> (multiple heads, sync'd on same resources)*	*st_313.s*	0:01	0:10
314	8	*pshr Ak,<effa> (multiple heads, sync'd on different resources)*	*st_314.s*	0:02	0:10
321	5	*popr <effa>,Sk (single head)*	*st_321.s*	0:02	0:10
322	6	*popr <effa>,Sk (multiple heads, sync'd, stepped on same resource)*	*st_322.s*	0:02	0:10
323	7	*popr <effa>,Sk (multiple heads, sync'd on same resources)*	*st_323.s*	0:01	0:10
324	8	*popr <effa>,Sk (multiple heads, sync'd on different resources)*	*st_324.s*	0:01	0:10
331	5	*incr.w <effa>,Ak (single head)*	*st_331.s*	0:02	0:10
332	6	*incr.w <effa>,Ak (multiple heads, sync'd, stepped on same resource)*	*st_332.s*	0:01	0:10
333	7	*incr.w <effa>,Ak (multiple heads, sync'd on same resources)*	*st_333.s*	0:01	0:10
334	8	*incr.w <effa>,Ak (multiple heads, sync'd on different resources)*	*st_334.s*	0:02	0:10

Table cpu4233-3, Subtests (continued)

SUBTEST	CLASS	TEST PERFORMED	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
341	5	*incr.l <effa>,Sk (single head)*	*st_341.s*	0:02	0:10
342	6	*incr.l <effa>,Sk (multiple heads, sync'd, stepped on same resource)*	*st_342.s*	0:01	0:10
343	7	*incr.l <effa>,Sk (multiple heads, sync'd on same resources)*	*st_343.s*	0:01	0:10
344	8	*incr.l <effa>,Sk (multiple heads, sync'd on different resources)*	*st_344.s*	0:02	0:10
351	9	*pfork <effa>,Ak (multiple headed pfork with an outstanding fork request)*	*st_351.s*	0:01	0:10
352	9	*pfork <effa>,Ak (multiple headed pfork with no outstanding fork request)*	*st_352.s*	0:01	0:10
353	9	*pfork <effa>,Ak (multiple headed pfork with no outstanding fork request)*	*st_353.s*	0:02	0:10
354	9	*pfork <effa>,Ak (verify all threads can be uniquely picked up)*	*st_354.s*	0:01	0:10
361	9	*cfork (multiple headed cfork with no outstanding fork request)*	*st_361.s*	0:02	0:10
362	9	*cfork (multiple headed cfork with an outstanding fork request)*	*st_362.s*	0:01	0:10
371	9	*wfork (multiple headed wfork)*	*st_371.s*	0:01	0:10
372	9	*wfork (verify a wfork spins if thread register is unlocked)*	*st_372.s*	0:02	0:10
381	9	*spawn <effa>,Ak (multiple headed spawn with an outstanding fork request)*	*st_381.s*	0:01	0:10
382	9	*spawn <effa>,Ak (multiple headed spawn with no outstanding fork request)*	*st_382.s*	0:02	0:10
383	9	*spawn <effa>,Ak (multiple headed spawn with no outstanding fork request)*	*st_383.s*	0:01	0:10
384	9	*spawn <effa>,Ak (verify all threads can be uniquely picked up)*	*st_384.s*	0:01	0:10

Table cpu4233-3, Subtests (continued)

SUBTEST	CLASS	TEST PERFORMED	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
391	9	*join (multiple headed wfork)*	*st_391.s*	0:02	0:10
392	9	*join (verify a join spins if thread register is unlocked)*	*st_392.s*	0:01	0:10
401	9	*idle (multiple headed idle)*	*st_401.s*	0:02	0:10
431	5	*tas <effa> (single head)*	*st_431.s*	0:02	0:10
432	6	*tas <effa> (multiple heads, sync'd, stepped on same memory)*	*st_432.s*	0:01	0:10
433	7	*tas <effa> (multiple heads, sync'd on same memory locations)*	*st_433.s*	0:01	0:10
434	8	*tas <effa> (multiple heads, sync'd on different memory)*	*st_434.s*	0:02	0:10
441	5	*tac <effa> (single head)*	*st_441.s*	0:01	0:10
442	6	*tac <effa> (multiple heads, sync'd, stepped on same memory)*	*st_442.s*	0:01	0:10
443	7	*tac <effa> (multiple heads, sync'd on same memory locations)*	*st_443.s*	0:02	0:10
444	8	*tac <effa> (multiple heads, sync'd on different memory)*	*st_444.s*	0:02	0:10
451	10	*ldcmr <effa>,Ak (single head)*	*st_451.s*	0:02	0:10
452	10	*ldcmr <effa>,Ak (multiple heads, same cmr set)*	*st_452.s*	0:02	0:10
453	10	*ldcmr <effa>,Ak (multiple heads, different cmr set)*	*st_453.s*	0:02	0:10
461	10	*stcmr Ak,<effa> (single head)*	*st_461.s*	0:02	0:10
462	10	*stcmr Ak,<effa> (multiple heads, different cmr set)*	*st_462.s*	0:02	0:10
471	11	*trap #rm,#bit (multiple heads, single head execution, same cir)*	*st_471.s*	0:01	0:10

Table cpu4233-3, Subtests (continued)

SUBTEST	CLASS	TEST PERFORMED	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
472	11	*trap #rm,#bit (multiple heads, single head execution, diff cir)*	*st_472.s*	0:01	0:10
476	11	*pbkpt (multiple heads, single head execution, same cir)*	*st_476.s*	0:02	0:10
477	11	*pbkpt (multiple heads, single head execution, diff cir)*	*st_477.s*	0:01	0:10
500	12	*remote invalidate 1 (single head writing bytes, other heads spin on cmr)*	*st_500.s*	0:03	0:10
501	12	*remote invalidate 1 (single head writing half words, other heads spin on cmr)*	*st_501.s*	0:03	0:10
502	12	*remote invalidate 1 (single head writing words, other heads spin on cmr)*	*st_502.s*	0:02	0:10
503	12	*remote invalidate 1 (single head writing long words, other heads spin on cmr)*	*st_503.s*	0:02	0:10
510	12	*remote invalidate 2 (single head writing bytes, other heads spin on memory)*	*st_510.s*	0:03	0:10
511	12	*remote invalidate 2 (single head writing half words, other heads spin on memory)*	*st_511.s*	0:02	0:10
512	12	*remote invalidate 2 (single head writing words, other heads spin on memory*	*st_512.s*	0:02	0:10
513	12	*remote invalidate 2 (single head writing long words, other heads spin on memory*	*st_513.s*	0:02	0:10
520	12	*remote invalidate 3 (single head writing bytes, other heads delayed reading*	*st_520.s*	0:02	0:10
521	12	*remote invalidate 3 (single head writing half words, other heads delayed reading*	*st_521.s*	0:02	0:10

Table cpu4233-3, Subtests (continued)

SUBTEST	CLASS	TEST PERFORMED	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
522	12	*remote invalidate 8 (single head writing words, other heads delayed reading*	*st_522.s*	0:03	0:10
523	12	*remote invalidate 8 (single head writing long words, other heads delayed reading*	*st_523.s*	0:01	0:10
530	12	*remote invalidate 4 (multiple heads writing unique bytes in 4k block*	*st_530.s*	0:02	0:10
531	12	*remote invalidate 4 (multiple heads writing unique half words in 4k block*	*st_531.s*	0:02	0:10
532	12	*remote invalidate 4 (multiple heads writing unique words in 4k block*	*st_532.s*	0:02	0:10
533	12	*remote invalidate 4 (multiple heads writing unique long words in 4k block*	*st_533.s*	0:01	0:10
540	13	*remote invalidate 1 (single head writing bytes, other heads spin on cmr*	*st_540.s*	0:03	0:10
541	13	*remote invalidate 1 (single head writing half words, other heads spin on cmr*	*st_541.s*	0:02	0:10
542	13	*remote invalidate 1 (single head writing words, other heads spin on cmr*	*st_542.s*	0:02	0:10
543	13	*remote invalidate 1 (single head writing long words, other heads spin on cmr*	*st_543.s*	0:02	0:10
550	13	*remote invalidate 1 (single head writing bytes, other heads spin on cmr*	*st_550.s*	0:03	0:10
551	13	*remote invalidate 1 (single head writing half words, other heads spin on cmr*	*st_551.s*	0:03	0:10
552	13	*remote invalidate 1 (single head writing words, other heads spin on cmr*	*st_552.s*	0:02	0:10

Table cpu4233-3, Subtests (continued)

SUBTEST	CLASS	TEST PERFORMED	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
553	13	*remote invalidate 1 (single head writing long words, other heads spin on cmr*	*st_553.s*	0:02	0:10
560	13	*remote invalidate 1 (single head writing bytes, other heads spin on cmr*	*st_560.s*	0:02	0:10
561	13	*remote invalidate 1 (single head writing half words, other heads spin on cmr*	*st_561.s*	0:02	0:10
562	13	*remote invalidate 1 (single head writing words, other heads spin on cmr*	*st_562.s*	0:02	0:10
563	13	*remote invalidate 1 (single head writing long words, other heads spin on cmr*	*st_563.s*	0:01	0:10
570	13	*remote invalidates with pte missing*	*st_570.s*	0:01	0:20
571	13	*remote invalidates with Read fault*	*st_571.s*	0:02	0:20
1000	14	*interrupt test, multiple heads*	*st_1000.s*	0:01	0:10
1010	14	*deadlock test, multiple heads*	*st_1010.s*	0:02	0:10
1020	14	*patu test, multiple heads*	*st_1020.s*	0:02	0:10
1025	14	*pate test, multiple heads*	*st_1025.s*	0:02	0:10
1030	14	*ctrsl test, multiple heads*	*st_1030.s*	0:04	0:10
1035	14	*ctrsg test, multiple heads*	*st_1035.s*	0:02	0:10
1040	14	*timer ring cross test, multiple heads*	*st_1040.s*	0:02	0:10
1045	14	*CPU execution timer test, multiple heads*	*st_1045.s*	0:02	0:10

Test Error Messages

For a common list of error messages that could result from running this test, refer to the appendix on CPU Error Messages.

SPU UNIX Error Messages

For a common list of possible error messages resulting from SPU UNIX, refer to the appendix on SPU UNIX Error Messages.

THIS PAGE INTENTIONALLY LEFT BLANK

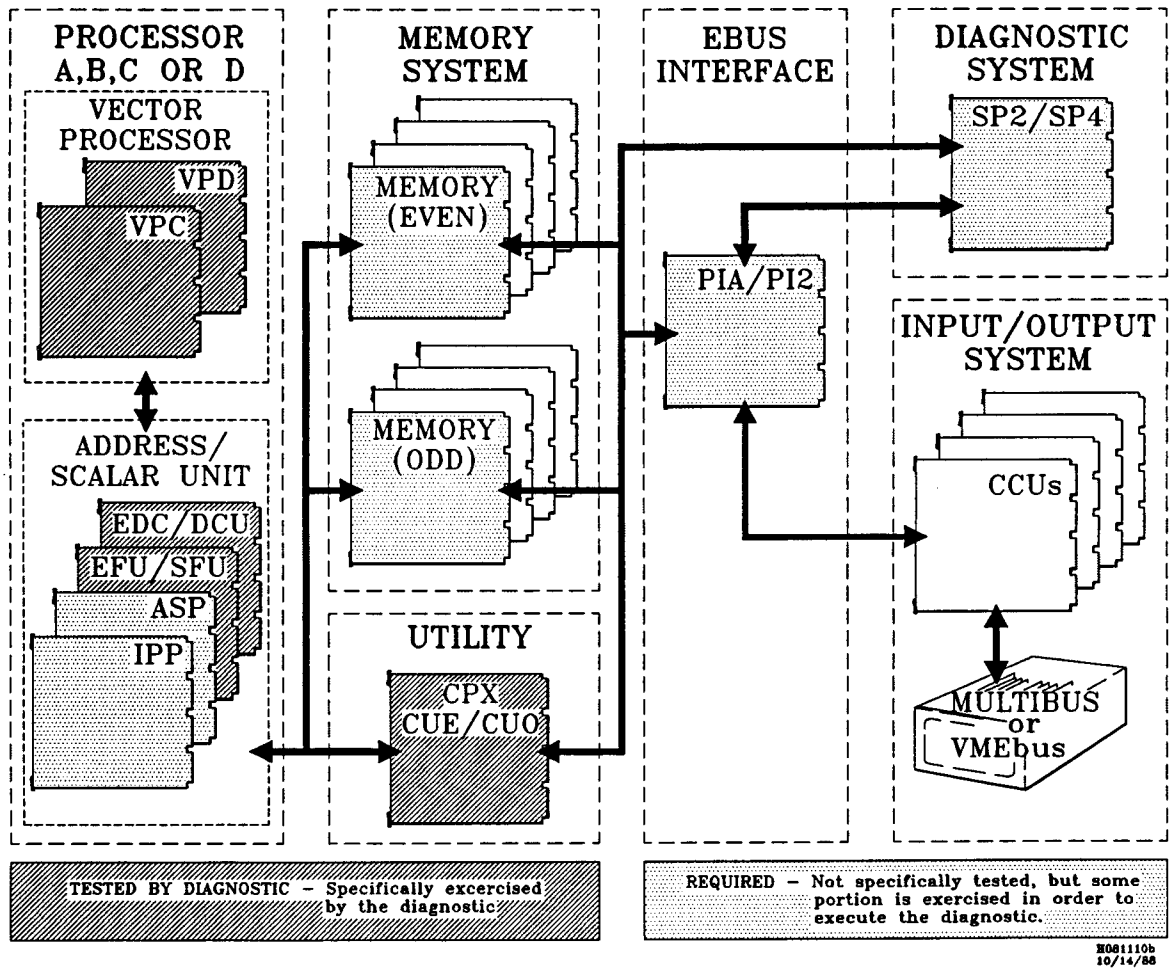
cpu4241

Enhanced Vector Instruction Tests

Overview

The *cpu4241* test is a group of vector instruction tests used to verify the operation of C200 Series vector and vector-under-mask instructions along with their interfaces to other C200 Series subsystems. The verification is performed by exercising each vector/vector and scalar/vector instruction while varying all of the parameters on which the instructions depend. The following figure shows an overall view of what part of the system is being tested and which Field Replaceable Units (FRUs) are required for the test to run:

Figure cpu4241–1, Functional Areas Tested by *cpu4241*



Prerequisites and Required Equipment

The Vector Processor Control (VPC) and Vector Processor Data (VPD) boards must be present, and are tested, when running the *cpu4241* test. The boards listed in the following table must be operational to verify the required boards. No additional equipment is required to run this test.

Table cpu4241–1, Required Functional Boards

BOARD	TEST TO VERIFY
Service Processor (SP2 or SP4)	*spu1000, spu4000*
PBUS Interface Adapter (PIA or PI2)	*pia4000*
CPU Utility Board(s) (CPX or CUE/CUO)	*cpx4000*
Memory System	*mem4000*
Instruction Processor Unit (IPP)	*cpu4030*
Address Scalar Processor (ASP)	*cpu4030*
Scalar Function Unit (SFU or EFU)	*cpu4030*
Data Cache Unit (DCU or EDC)	*cpu4030*

NOTE

The memory system consists of a minimum of one pair of memory boards (one even and one odd).

Test Invocation

To invoke the *cpu4241* test, use the procedure shown in the following figure. All responses in **boldface** are entered by the user. The prompts and responses appear sequentially on the screen, one line at a time. All prompts and responses are shown in one figure for convenience.

CAUTION

The invocation sequence shown in the following figure is the typical invocation sequence. However, the *initall* utility must be executed in some cases. If the system has just been powered up, if *mem4000* was executed with failures, or if *spu4000* was executed, then *initall* must be executed prior to any test execution. Failure to execute *initall* in these circumstances could result in invalid test results.

NOTE

Running the *initall* utility requires two to three minutes to execute depending on if the control stores have been previously loaded. However, it is suggested that *initall* be executed in the event that the state of the system is unknown.

Figure cpu4241–2, Test Invocation Sequence

```
(spu)>  cd /mnt/test
(spu)>  sysreset
(spu)>  dshell

CONVEX DIAGNOSTIC SHELL

:test cpu4241 [-c [class numeral(s)]] [-s [subtest numeral(s)]] [+> filename]
```

NOTE

After entering **dshell**, specific *dshell* parameters may be changed. Refer to the "Dshell and Iscan Overview" chapter of this manual for more information.

Entering only **test cpu4241** executes all *cpu4241* subtests sequentially. To execute a specific class(es) of subtest(s) or one or more individual subtests, enter the *-c* or *-s* options, respectively. Detailed information for using these options can be found in the "Dshell and Iscan Overview" chapter of this manual. The **[+>***filename***]** option allows the test results to be appended to *filename*.

Test Parameter Menu

Once the test is invoked, a test parameter menu is presented allowing selection of default switches. If the test is run with all defaults invoked (answer **y** to the first prompt), no other prompts are provided. If the user answers **n** to the first prompt (run test without default switches), then a series of prompts are presented. The following figure shows all prompts, their possible answers (in brackets [ ]), and their default answers (in parenthesis ( )). The prompts and responses in the following figure appear sequentially on the screen, one line at a time. All the prompts and responses are shown in one figure for convenience.

Figure cpu4241–3, *cpu4241* Test Parameter Menu

```
Test 'cpu4241.t'                          Thu Nov 19 00:00:00 1965

                          ENTER TEST PARAMETERS

                  [ ]     Encloses allowed input ranges or values
                  ( )     Encloses the default value
                  ^       Returns to the previous prompt
                  :nn     Returns to the prompt # nn
                  :       Returns to the first unsatisfied prompt
                  :?      Reviews previous entries


 1: Run default switches? [y,n]                              (y) ->
 2: Cpus to test: [ABCD]                                     (ABCD) ->
 3: Parallel Test Execution? [y,n]                           (y) ->
 4: Forced Faulting Enabled? [y,n]                           (n) ->
 5: Fault on Instruction Fetches? [y,n]                      (n) ->
 6: Sequential Execution? [y,n]                              (n) ->
 7: Timeout Scale Factor Enabled? [1-100]                    (1) ->
 8: Dcache Enabled? [y,n]                                    (y) ->
 9: Segment of Execution? [0-7]                              (0) ->
10: Chained Execution Mode? [y,n]                            (n) ->
11: Loop Enabled? [y,n]                                      (n) ->
12: Hard Errors Enabled? [y,n]                               (y) ->
13: Number of vl values to test(0..vl)? [0-128]              (128) ->
14: Load CPU Code? [y,n]                                     (y) ->
```

NOTE

In the second prompt in the previous figure, ABCD represents all available CPUs. Only the available CPUs within the machine under test will be displayed as possible choices. The fifth prompt in the previous list is only supplied when the fourth prompt is answered with **y**.

Prompt Explanations

A description of the meaning of each prompt follows:

```
Run default switches? [y/n]                                  (y) ->
```

If a response of **y** or <**CR**> is given, no additional test parameter prompts are displayed and testing begins. However, if a negative response is supplied, additional test parameter prompts are displayed allowing modification of the default selections. Only enter the **n** response to restart a previous testing session at a specified point or to change any of the default parameters.

The following prompts are only displayed and answered if the first prompt is answered with **n**:

```
Cpus to test: [ABCD]                                    (ABCD) ->
```

This prompt allows selection of the CPUs to be used in the test. The possible selection, represented by ABCD, will consist of all available CPUs. The default, ABCD, will consist of all available CPUs.

```
Parallel Test Execution? [y,n]                          (y) ->
```

If the system has more than one CPU available for testing, then this prompt is issued the user. If answered with **y**, then each CPU selected for testing will execute the tests concurrently; otherwise, each subtest is executed on the selected CPU's in the order of that the CPU's were selected.

```
Forced Faulting Enabled? [y,n]                          (n) ->
```

If answered with **y**, normal force faulting occurs only on data references. The system will force a non-resident data exception to occur on every data reference. For a more detailed explanation of forced faulting, refer to the glossary appendix of this manual.

```
Fault on Instruction Fetches? [y,n]                     (n) ->
```

In answered with **y**, force faulting occurs on instruction fetches in addition to data references. This prompt is only supplied to the user if the previous prompt is answered with **y**.

```
Sequential Execution? [y,n]                             (n) ->
```

If set by entering **y**, the sequential bit in the PSW will be set to forced sequential execution mode.

```
Timeout Scale Factor [1-100]                            (1) ->
```

This prompt allows adjustment of the timeout factor. The normal timeout factor is multiplied by the number selected to increase the timeout factor. For example, if **5** is entered, the normal timeout factor is multiplied by 5 and it will take the test five times as long to timeout.

```
Dcache Enabled? [y,n]                                   (y) ->
```

The Dcache is normally enabled; however, if it is suspected broken, it can be disabled by entering **n** at this prompt.

```
Segment of Execution? [0-7]                             (0) ->
```

This prompt is only displayed if forced faulting is not enabled. The segment of execution is contained in bits<31..29> of the Program Counter (PC). If **0** is entered, then bits<31..29> of the PC are 000 and the test is run in ring zero. If **1** is entered, then bits<31..29> of the PC are 001 and the test is run in ring one. If **2** is entered, then bits<31..29> of the PC are 010 and the test is run in ring two. If **3** is entered, then bits<31..29> of the PC are 011 and the test is run in ring three. If **4**, **5**, **6**, or **7** is entered, then bits<31..29> of the PC are 100, 101, 110, and 111 respectively and the test is run in ring four. Refer to the *CONVEX Architecture Reference* for more information concerning the meaning of rings in the machine architecture.

```
Chained Execution Mode? [y,n]                           (n) ->
```

With this option enabled, the test is executed in chained mode which causes the CPU to perform subtest sequencing. The Service Processor is unaware of the action of the CPU unless a subtest fails or unless all of the subtests pass. If this option is enabled, test execution time will be greatly

reduced. However, the only information printed to the console upon completion or failure (regardless of the cause) is the message, "Subtest 1 passed," or "Subtest 1 failed." Also, this option can not be enabled if the **-c** or the **-s** options were used in the invocation procedure. This option does not disable any subtests from begin executed.

```
Loop Enabled? [y,n]                                    (n) ->
```

If **y** is entered, hard looping on a specific subtest is enabled. When looping is enabled, the halt instruction of the specified subtest is changed to a branch back to the beginning of the subtest. This puts the subtest into an infinite loop which the user must break out of by typing **Ctrl- C**.

```
Hard Errors Enabled? [y,n]                             (y) ->
```

If this option is enabled and a hard error occurs, the clocks will be stopped and the test will fail. If this option is disabled, parity errors and other sources of hard errors will go undetected. It is recommended that hard errors normally be enabled.

```
Number of vl values to test (0..vl)? [0-128]           (128) ->
```

Each vector instruction which is dependent on the Vector Length (*vl*), has two separate subtests. The first subtest executes with a hard coded *vl* of 128. The second subtest uses this user supplied value as the initial *vl* value. The subtest is then executed and the *vl* decremented. This is repeated until the *vl* under test becomes less than zero.

```
Load CPU Code? [y,n]                                   (y) ->
```

If the CPU code for this test is already in memory, the user can enter **n** for this prompt and the code will not be reloaded (thus saving time).

When all prompts have been answered, a test parameter summary echos the prompts that have been answered. The following figure displays a sample test parameter summary. The actual summary varies depending on the answers to the prompts.

Figure cpu4241–4, Sample Test Parameter Summary

```
                    TEST PARAMETER SUMMARY

Run default switches?                          :  n
Cpus to test:                                  :  ab
Parallel Test Execution?                       :  y
Forced Faulting Enabled?                       :  y
Sequential Execution?                          :  n
Timeout Scale Factor Enabled?                  :  1
Dcache Enabled?                                :  y
Segment of Execution?                          :  0
Chained Execution Mode?                        :  n
Loop Enabled?                                  :  n
Hard Errors Enabled?                           :  y
Number of vl values to test (0..vl)?           :  128
Load CPU Code?                                 :  y
```

Hardware Initialization Sequence

After the last prompt is entered by the user (and before test code execution) the following events occur:

> **NOTE**
>
> The first two events are accomplished at the initial start of the test. The remaining events are accomplished when each subtest is initialized.

- For each CPU, each module's page table entries are set up in main memory. Page table entries begin at the last available address in main memory and work toward low memory.
- For each CPU, each module is loaded into main memory following the logical to physical mapping described by the page tables. The exact logical to physical mapping that the modules are loaded into depends upon which segment of execution is selected by the user.
- The system is initialized (the memory system, CPX or CUE/CUO, PIA or PI2, and the CPU boards are reset).
- For each CPU, parity is initialized in the scalar processor by sending the scalar processor into a microcode routine and issuing clocks.
- Each CPU's scratch RAM is loaded with various values to control the execution of the test.
- For each CPU, the initial Program Counter (PC) is loaded into register T0 and the initial Program Status Word (PSW) is loaded into register T2.
- For each CPU, control store is initialized to the cold-start location.

- Clocks are turned on to the processor selected. If parallel execution is selected, each CPU's clocks are turned on at one time. If sequential execution is selected, each CPU's clocks are turned on one at a time.

Memory Allocation

Immediately before test code execution, a current memory allocation screen is displayed. The following figure is an example of the current memory allocation screen. The physical and logical addresses shown, as well as the filenames, are only representative; the actual addresses will vary depending on installed memory & file sizes.

The following list explains what each column the current memory allocation screen depicts:

- **First Column** — File Number - useful in conjunction with the mm(1d) utility.
- **Second Column** — Physical memory addresses where the specified file is loaded
- **Third Column** — Process identification
- **Fourth Column** — File name (actual path is *./filename* or /mnt/test/CPU/*filename*). The entries `pte1`, `pte2`, and `ptet` are not actual files but are instead indications of the page tables.
- **Fifth Column** — Logical starting address of the specified file

Figure cpu4241–5, Memory Allocation Screen

```
                    Current Memory Allocation

File No.  Physical  Address Pid File Name                    Logical Offset
      1   00000000-00027fff   0 p0r0_4241                   00000000
      2   00028000-000b1fff   0 cpu4241.rnn                 00022000
      1   000b2000-000d9fff   1 p0r0_4241                   00000000
      2   000da000-00163fff   1 cpu4241.rnn                 00022000
   ----   03ff7000-03ff9fff   1 ptet                        NA
   ----   03ffa000-03ffafff   1 pte2                        NA
   ----   03ffb000-03ffdfff   0 ptet                        NA
   ----   03ffe000-03ffefff   0 pte2                        NA
   ----   03fffc00-3fffffff   1 pte1                        NA
```

Class Descriptions

There are four different classes of subtests for *cpu4241*. The following sections describe the different classes and each of their subtests. Each section contains a table listing each subtest in that class, a description of the subtest, the subtest executable test code (object module) and the subtest source code with comments (source file), the minimum time required to run the subtest (nominal time), and the timeout limit (maximum time).

> **NOTE**
>
> In the following tables, the **INSTRUCTIONS TESTED** column lists each particular instruction that is tested. For more information on each instruction's meaning, refer to the "Opcodes Sorted by Name" appendix within this manual or the *CONVEX Architecture Reference*.

Class 1 Subtests

Class 1 subtests verify the operation of loading, storing, and modifying the vector unit control functions. Specifically, this class verifies the ability to alter and save the Vector Length (VL) register, Vector Stride (VS) register, and the Vector Merge (VM) register.

All subtests end with a halt command and store a halt code in address register A1. If the code stored in A1 is 0x100, then the subtest passed. Any other code in A1 indicates that the subtest failed. The following table lists the Class 1 subtests:

Table cpu4241–2, Class 1 Subtests

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
21	*mov.w VL,Sk*	*cpu4241.rnn*	*st_021.s*	0:01	0:11
26	*mov.w VS,Sk*	*cpu4241.rnn*	*st_026.s*	0:01	0:11
61	*mov Sk,VM(U|L)*	*cpu4241.rnn*	*st_061.s*	0:01	0:11
62	*mov VM(U|L),Sk*	*cpu4241.rnn*	*st_062.s*	0:01	0:11
1005	*vector valid test*	*cpu4241.rnn*	*st_1005.s*	0:01	0:11

Class 2 Subtests

Class 2 subtests verify the operation of the logical and arithmetic pipes. Specifically, this class verifies vector/vector and scalar/vector comparisons, vector/vector and scalar/vector additions and subtractions, and vector reductions.

All subtests end with a halt command and store a halt code in address register A1. If the code stored in A1 is 0x100, then the subtest passed. Any other code in A1 indicates that the subtest failed. The following table lists the Class 2 subtests:

Table cpu4241–3, Class 2 Subtests

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
2105	*le.b.t Sj,Vk*	*cpu4241.rnn*	*st_105_106.s,st_105_execute.s*	0:01	0:11
2106	*le.b.t Sj,Vk*	*cpu4241.rnn*	*st_105_106.s,st_105_execute.s*	0:03	0:13
4105	*le.b.f Sj,Vk*	*cpu4241.rnn*	*st_105_106.s,st_105_execute.s*	0:01	0:11
4106	*le.b.f Sj,Vk*	*cpu4241.rnn*	*st_105_106.s,st_105_execute.s*	0:03	0:13
2110	*le.h.t Sj,Vk*	*cpu4241.rnn*	*st_110_111.s,st_110_execute.s*	0:01	0:11
2111	*le.h.t Sj,Vk*	*cpu4241.rnn*	*st_110_111.s,st_110_execute.s*	0:03	0:13
4110	*le.h.f Sj,Vk*	*cpu4241.rnn*	*st_110_111.s,st_110_execute.s*	0:01	0:11
4111	*le.h.f Sj,Vk*	*cpu4241.rnn*	*st_110_111.s,st_110_execute.s*	0:03	0:13
2115	*le.w.t Sj,Vk*	*cpu4241.rnn*	*st_115_116.s,st_115_execute.s*	0:02	0:12
2116	*le.w.t Sj,Vk*	*cpu4241.rnn*	*st_115_116.s,st_115_execute.s*	0:03	0:13
4115	*le.w.f Sj,Vk*	*cpu4241.rnn*	*st_115_116.s,st_115_execute.s*	0:01	0:11
4116	*le.w.f Sj,Vk*	*cpu4241.rnn*	*st_115_116.s,st_115_execute.s*	0:03	0:13
2120	*le.l.t Sj,Vk*	*cpu4241.rnn*	*st_120_121.s,st_120_execute.s*	0:01	0:11
2121	*le.l.t Sj,Vk*	*cpu4241.rnn*	*st_120_121.s,st_120_execute.s*	0:03	0:13
4120	*le.l.f Sj,Vk*	*cpu4241.rnn*	*st_120_121.s,st_120_execute.s*	0:01	0:11
4121	*le.l.f Sj,Vk*	*cpu4241.rnn*	*st_120_121.s,st_120_execute.s*	0:03	0:13
2125	*le.s.t Sj,Vk Native Mode*	*cpu4241.rnn*	*st_125_126.s,st_125_execute.s*	0:01	0:11
2126	*le.s.t Sj,Vk Native Mode*	*cpu4241.rnn*	*st_125_126.s,st_125_execute.s*	0:11	0:21
3125	*le.s.t Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_125_126.s,st_1125_execute.s*	0:01	0:11
3126	*le.s.t Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_125_126.s,st_1125_execute.s*	0:09	0:19
4125	*le.s.f Sj,Vk Native Mode*	*cpu4241.rnn*	*st_125_126.s,st_125_execute.s*	0:01	0:11
4126	*le.s.f Sj,Vk Native Mode*	*cpu4241.rnn*	*st_125_126.s,st_125_execute.s*	0:11	0:21
5125	*le.s.f Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_125_126.s,st_1125_execute.s*	0:01	0:11
5126	*le.s.f Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_125_126.s,st_1125_execute.s*	0:09	0:19
2130	*le.d.t Sj,Vk Native Mode*	*cpu4241.rnn*	*st_130_131.s,st_130_execute.s*	0:01	0:11
2131	*le.d.t Sj,Vk Native Mode*	*cpu4241.rnn*	*st_130_131.s,st_130_execute.s*	0:11	0:21
3130	*le.d.t Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_130_131.s,st_1130_execute.s*	0:01	0:11
3131	*le.d.t Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_130_131.s,st_1130_execute.s*	0:09	0:19
4130	*le.d.f Sj,Vk Native Mode*	*cpu4241.rnn*	*st_130_131.s,st_130_execute.s*	0:01	0:11
4131	*le.d.f Sj,Vk Native Mode*	*cpu4241.rnn*	*st_130_131.s,st_130_execute.s*	0:11	0:21
5130	*le.d.f Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_130_131.s,st_1130_execute.s*	0:01	0:11
5131	*le.d.f Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_130_131.s,st_1130_execute.s*	0:09	0:19
2135	*lt.b.t Sj,Vk*	*cpu4241.rnn*	*st_135_136.s,st_135_execute.s*	0:01	0:11
2136	*lt.b.t Sj,Vk*	*cpu4241.rnn*	*st_135_136.s,st_135_execute.s*	0:03	0:13
4135	*lt.b.f Sj,Vk*	*cpu4241.rnn*	*st_135_136.s,st_135_execute.s*	0:02	0:12
4136	*lt.b.f Sj,Vk*	*cpu4241.rnn*	*st_135_136.s,st_135_execute.s*	0:03	0:13
2140	*lt.h.t Sj,Vk*	*cpu4241.rnn*	*st_140_141.s,st_140_execute.s*	0:01	0:11
2141	*lt.h.t Sj,Vk*	*cpu4241.rnn*	*st_140_141.s,st_140_execute.s*	0:03	0:13

Table cpu4241-3, Class 2 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
4140	*lt.h.f Sj,Vk*	*cpu4241.rnn*	*st_140_141.s,st_140_execute.s*	0:01	0:11
4141	*lt.h.f Sj,Vk*	*cpu4241.rnn*	*st_140_141.s,st_140_execute.s*	0:03	0:13
2145	*lt.w.t Sj,Vk*	*cpu4241.rnn*	*st_145_146.s,st_145_execute.s*	0:01	0:11
2146	*lt.w.t Sj,Vk*	*cpu4241.rnn*	*st_145_146.s,st_145_execute.s*	0:03	0:13
4145	*lt.w.f Sj,Vk*	*cpu4241.rnn*	*st_145_146.s,st_145_execute.s*	0:01	0:11
4146	*lt.w.f Sj,Vk*	*cpu4241.rnn*	*st_145_146.s,st_145_execute.s*	0:03	0:13
2150	*lt.l.t Sj,Vk*	*cpu4241.rnn*	*st_150_151.s,st_150_execute.s*	0:01	0:11
2151	*lt.l.t Sj,Vk*	*cpu4241.rnn*	*st_150_151.s,st_150_execute.s*	0:03	0:13
4150	*lt.l.f Sj,Vk*	*cpu4241.rnn*	*st_150_151.s,st_150_execute.s*	0:01	0:11
4151	*lt.l.f Sj,Vk*	*cpu4241.rnn*	*st_150_151.s,st_150_execute.s*	0:03	0:13
2155	*lt.s.t Sj,Vk Native Mode*	*cpu4241.rnn*	*st_155_156.s,st_155_execute.s*	0:01	0:11
2156	*lt.s.t Sj,Vk Native Mode*	*cpu4241.rnn*	*st_155_156.s,st_155_execute.s*	0:10	0:20
3155	*lt.s.t Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_155_156.s,st_1155_execute.s*	0:02	0:12
3156	*lt.s.t Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_155_156.s,st_1155_execute.s*	0:09	0:19
4155	*lt.s.f Sj,Vk Native Mode*	*cpu4241.rnn*	*st_155_156.s,st_155_execute.s*	0:01	0:11
4156	*lt.s.f Sj,Vk Native Mode*	*cpu4241.rnn*	*st_155_156.s,st_155_execute.s*	0:10	0:20
5155	*lt.s.f Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_155_156.s,st_1155_execute.s*	0:02	0:12
5156	*lt.s.f Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_155_156.s,st_1155_execute.s*	0:09	0:19
2160	*lt.d.t Sj,Vk Native Mode*	*cpu4241.rnn*	*st_160_1611.s,st_160_execute.s*	0:01	0:11
2161	*lt.d.t Sj,Vk Native Mode*	*cpu4241.rnn*	*st_160_1611.s,st_160_execute.s*	0:10	0:20
3160	*lt.d.t Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_160_161.s,st_1160_execute.s*	0:01	0:11
3161	*lt.d.t Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_160_161.s,st_1160_execute.s*	0:09	0:19
4160	*lt.d.f Sj,Vk Native Mode*	*cpu4241.rnn*	*st_160_161.s,st_160_execute.s*	0:01	0:11
4161	*lt.d.f Sj,Vk Native Mode*	*cpu4241.rnn*	*st_160_161.s,st_160_execute.s*	0:11	0:21
5160	*lt.d.f Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_160_161.s,st_1160_execute.s*	0:01	0:11
5161	*lt.d.f Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_160_161.s,st_1160_execute.s*	0:09	0:19
2165	*eq.b.t Sj,Vk*	*cpu4241.rnn*	*st_165_166.s,st_165_execute.s*	0:01	0:11
2166	*eq.b.t Sj,Vk*	*cpu4241.rnn*	*st_165_166.s,st_165_execute.s*	0:03	0:13
4165	*eq.b.f Sj,Vk*	*cpu4241.rnn*	*st_165_166.s,st_165_execute.s*	0:01	0:11
4166	*eq.b.f Sj,Vk*	*cpu4241.rnn*	*st_165_166.s,st_165_execute.s*	0:03	0:13
2170	*eq.h.t Sj,Vk*	*cpu4241.rnn*	*st_170_171.s,st_170_execute.s*	0:01	0:11
2171	*eq.h.t Sj,Vk*	*cpu4241.rnn*	*st_170_171.s,st_170_execute.s*	0:03	0:13
4170	*eq.h.f Sj,Vk*	*cpu4241.rnn*	*st_170_171.s,st_170_execute.s*	0:01	0:11
4171	*eq.h.f Sj,Vk*	*cpu4241.rnn*	*st_170_171.s,st_170_execute.s*	0:03	0:13
2175	*eq.w.t Sj,Vk*	*cpu4241.rnn*	*st_175_176.s,st_175_execute.s*	0:02	0:12
2176	*eq.w.t Sj,Vk*	*cpu4241.rnn*	*st_175_176.s,st_175_execute.s*	0:03	0:13
4175	*eq.w.f Sj,Vk*	*cpu4241.rnn*	*st_175_176.s,st_175_execute.s*	0:01	0:11
4176	*eq.w.f Sj,Vk*	*cpu4241.rnn*	*st_175_176.s,st_175_execute.s*	0:03	0:13

Table cpu4241-3, Class 2 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
2180	*eq.l.t Sj,Vk*	*cpu4241.rnn*	*st_180_181.s,st_180_execute.s*	0:01	0:11
2181	*eq.l.t Sj,Vk*	*cpu4241.rnn*	*st_180_181.s,st_180_execute.s*	0:02	0:12
4180	*eq.l.f Sj,Vk*	*cpu4241.rnn*	*st_180_181.s,st_180_execute.s*	0:01	0:11
4181	*eq.l.f Sj,Vk*	*cpu4241.rnn*	*st_180_181.s,st_180_execute.s*	0:03	0:13
2185	*eq.s.t Sj,Vk Native Mode*	*cpu4241.rnn*	*st_185_186.s,st_185_execute.s*	0:02	0:12
2186	*eq.s.t Sj,Vk Native Mode*	*cpu4241.rnn*	*st_185_186.s,st_185_execute.s*	0:11	0:21
3185	*eq.s.t Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_185_186.s,st_1185_execute.s*	0:01	0:11
3186	*eq.s.t Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_185_186.s,st_1185_execute.s*	0:10	0:20
4185	*eq.s.f Sj,Vk Native Mode*	*cpu4241.rnn*	*st_185_186.s,st_185_execute.s*	0:01	0:11
4186	*eq.s.f Sj,Vk Native Mode*	*cpu4241.rnn*	*st_185_186.s,st_185_execute.s*	0:11	0:21
5185	*eq.s.f Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_185_186.s,st_1185_execute.s*	0:01	0:11
5186	*eq.s.f Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_185_186.s,st_1185_execute.s*	0:10	0:20
2190	*eq.d.t Sj,Vk Native Mode*	*cpu4241.rnn*	*st_190_191.s,st_190_execute.s*	0:01	0:11
2191	*eq.d.t Sj,Vk Native Mode*	*cpu4241.rnn*	*st_190_191.s,st_190_execute.s*	0:11	0:21
3190	*eq.d.t Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_190_191.s,st_1190_execute.s*	0:02	0:12
3191	*eq.d.t Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_190_191.s,st_1190_execute.s*	0:09	0:19
4190	*eq.d.f Sj,Vk Native Mode*	*cpu4241.rnn*	*st_190_191.s,st_190_execute.s*	0:01	0:11
4191	*eq.d.f Sj,Vk Native Mode*	*cpu4241.rnn*	*st_190_191.s,st_190_execute.s*	0:10	0:20
5190	*eq.d.f Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_190_191.s,st_1190_execute.s*	0:01	0:11
5191	*eq.d.f Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_190_191.s,st_1190_execute.s*	0:09	0:19
196	*shf Vi,Sj,Vk*	*cpu4241.rnn*	*st_196_197.s,st_196_execute.s*	0:01	0:11
197	*shf Vi,Sj,Vk*	*cpu4241.rnn*	*st_196_197.s,st_196_execute.s*	0:09	0:19
2196	*shf.t Vi,Sj,Vk*	*cpu4241.rnn*	*st_196_197.s,st_196_execute.s*	0:02	0:12
2197	*shf.t Vi,Sj,Vk*	*cpu4241.rnn*	*st_196_197.s,st_196_execute.s*	1:13	1:23
4196	*shf.f Vi,Sj,Vk*	*cpu4241.rnn*	*st_196_197.s,st_196_execute.s*	0:02	0:12
4197	*shf.f Vi,Sj,Vk*	*cpu4241.rnn*	*st_196_197.s,st_196_execute.s*	1:14	1:24
220	*tzc Vj,Vk*	*cpu4241.rnn*	*st_220_221.s,st_220_execute.s*	0:09	0:19
221	*tzc Vj,Vk*	*cpu4241.rnn*	*st_220_221.s,st_220_execute.s*	0:48	0:58
2220	*tzc.t Vj,Vk*	*cpu4241.rnn*	*st_220_221.s,st_220_execute.s*	0:09	0:19
2221	*tzc.t Vj,Vk*	*cpu4241.rnn*	*st_220_221.s,st_220_execute.s*	7:08	7:18
4220	*tzc.f Vj,Vk*	*cpu4241.rnn*	*st_220_221.s,st_220_execute.s*	0:10	0:20
4221	*tzc.f Vj,Vk*	*cpu4241.rnn*	*st_220_221.s,st_220_execute.s*	7:10	7:20
225	*tzc Vj,Vk*	*cpu4241.rnn*	*st_225_226.s,st_225_execute.s*	0:01	0:11
226	*tzc Vj,Vk*	*cpu4241.rnn*	*st_225_226.s,st_225_execute.s*	0:48	0:58
2225	*tzc.t Vj,Vk*	*cpu4241.rnn*	*st_225_226.s,st_225_execute.s*	0:10	0:20
2226	*tzc.t Vj,Vk*	*cpu4241.rnn*	*st_225_226.s,st_225_execute.s*	7:12	7:22
4225	*tzc.f Vj,Vk*	*cpu4241.rnn*	*st_225_226.s,st_225_execute.s*	0:10	0:20
4226	*tzc.f Vj,Vk*	*cpu4241.rnn*	*st_225_226.s,st_225_execute.s*	7:13	7:23
2250	*add.b.t Vi,Sj,Vk*	*cpu4241.rnn*	*st_250_251.s,st_250_execute.s*	0:01	0:11

Table cpu4241-3, Class 2 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
2251	*add.b.t Vi,Sj,Vk*	*cpu4241.rnn*	*st_250_251.s,st_250_execute.s*	0:19	0:29
4250	*add.b.f Vi,Sj,Vk*	*cpu4241.rnn*	*st_250_251.s,st_250_execute.s*	0:01	0:11
4251	*add.b.f Vi,Sj,Vk*	*cpu4241.rnn*	*st_250_251.s,st_250_execute.s*	0:19	0:29
2255	*add.h.t Vi,Sj,Vk*	*cpu4241.rnn*	*st_255_256.s,st_255_execute.s*	0:01	0:11
2256	*add.h.t Vi,Sj,Vk*	*cpu4241.rnn*	*st_255_256.s,st_255_execute.s*	0:19	0:29
4255	*add.h.f Vi,Sj,Vk*	*cpu4241.rnn*	*st_255_256.s,st_255_execute.s*	0:01	0:11
4256	*add.h.f Vi,Sj,Vk*	*cpu4241.rnn*	*st_255_256.s,st_255_execute.s*	0:19	0:29
2260	*add.w.t Vi,Sj,Vk*	*cpu4241.rnn*	*st_260_261.s,st_260_execute.s*	0:01	0:11
2261	*add.w.t Vi,Sj,Vk*	*cpu4241.rnn*	*st_260_261.s,st_260_execute.s*	0:19	0:29
4260	*add.w.f Vi,Sj,Vk*	*cpu4241.rnn*	*st_260_261.s,st_260_execute.s*	0:01	0:11
4261	*add.w.f Vi,Sj,Vk*	*cpu4241.rnn*	*st_260_261.s,st_260_execute.s*	0:19	0:29
2265	*add.l.t Vi,Sj,Vk*	*cpu4241.rnn*	*st_265_266.s,st_265_execute.s*	0:01	0:11
2266	*add.l.t Vi,Sj,Vk*	*cpu4241.rnn*	*st_265_266.s,st_265_execute.s*	0:19	0:29
4265	*add.l.f Vi,Sj,Vk*	*cpu4241.rnn*	*st_265_266.s,st_265_execute.s*	0:02	0:12
4266	*add.l.f Vi,Sj,Vk*	*cpu4241.rnn*	*st_265_266.s,st_265_execute.s*	0:19	0:29
2270	*add.s.t Vi,Sj,Vk Native Mode*	*cpu4241.rnn*	*st_270_271.s,st_270_execute.s*	0:01	0:11
2271	*add.s.t Vi,Sj,Vk Native Mode*	*cpu4241.rnn*	*st_270_271.s,st_270_execute.s*	0:12	0:22
3270	*add.s.t Vi,Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_270_271.s,st_1270_execute.s*	0:02	0:12
3271	*add.s.t Vi,Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_270_271.s,st_1270_execute.s*	0:15	0:25
4270	*add.s.f Vi,Sj,Vk Native Mode*	*cpu4241.rnn*	*st_270_271.s,st_270_execute.s*	0:01	0:11
4271	*add.s.f Vi,Sj,Vk Native Mode*	*cpu4241.rnn*	*st_270_271.s,st_270_execute.s*	0:12	0:22
5270	*add.s.f Vi,Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_270_27.s,st_1270_execute.s*	0:01	0:11
5271	*add.s.f Vi,Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_270_27.s,st_1270_execute.s*	0:15	0:25
2275	*add.d.t Vi,Sj,Vk Native Mode*	*cpu4241.rnn*	*st_275_276.s,st_275_execute.s*	0:01	0:11
2276	*add.d.t Vi,Sj,Vk Native Mode*	*cpu4241.rnn*	*st_275_276.s,st_275_execute.s*	0:09	0:19
3275	*add.d.t Vi,Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_275_276.s,st_1275_execute.s*	0:02	0:12
3276	*add.d.t Vi,Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_275_276.s,st_1275_execute.s*	0:15	0:25
4275	*add.d.f Vi,Sj,Vk Native Mode*	*cpu4241.rnn*	*st_275_276.s,st_275_execute.s*	0:01	0:11
4276	*add.d.f Vi,Sj,Vk Native Mode*	*cpu4241.rnn*	*st_275_276.s,st_275_execute.s*	0:09	0:19
5275	*add.d.f Vi,Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_275_276.s,st_1275_execute.s*	0:01	0:11
5276	*add.d.f Vi,Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_275_276.s,st_1275_execute.s*	0:15	0:25
2280	*sub.b.t Vi,Sj,Vk*	*cpu4241.rnn*	*st_280_281.s,st_280_execute.s*	0:01	0:11
2281	*sub.b.t Vi,Sj,Vk*	*cpu4241.rnn*	*st_280_281.s,st_280_execute.s*	0:19	0:29
4280	*sub.b.f Vi,Sj,Vk*	*cpu4241.rnn*	*st_280_281.s,st_280_execute.s*	0:01	0:11
4281	*sub.b.f Vi,Sj,Vk*	*cpu4241.rnn*	*st_280_281.s,st_280_execute.s*	0:19	0:29
2285	*sub.h.t Vi,Sj,Vk*	*cpu4241.rnn*	*st_285_286.s,st_285_execute.s*	0:01	0:11
2286	*sub.h.t Vi,Sj,Vk*	*cpu4241.rnn*	*st_285_286.s,st_285_execute.s*	0:19	0:29
4285	*sub.h.f Vi,Sj,Vk*	*cpu4241.rnn*	*st_285_286.s,st_285_execute.s*	0:01	0:11
4286	*sub.h.f Vi,Sj,Vk*	*cpu4241.rnn*	*st_285_286.s,st_285_execute.s*	0:19	0:29

Table cpu4241-3, Class 2 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
2290	*sub.w.t Vi,Sj,Vk*	*cpu4241.rnn*	*st_290_291.s,st_290_execute.s*	0:01	0:11
2291	*sub.w.t Vi,Sj,Vk*	*cpu4241.rnn*	*st_290_291.s,st_290_execute.s*	0:19	0:29
4290	*sub.w.f Vi,Sj,Vk*	*cpu4241.rnn*	*st_290_291.s,st_290_execute.s*	0:02	0:12
4291	*sub.w.f Vi,Sj,Vk*	*cpu4241.rnn*	*st_290_291.s,st_290_execute.s*	0:19	0:29
2295	*sub.l.t Vi,Sj,Vk*	*cpu4241.rnn*	*st_295_296.s,st_295_execute.s*	0:01	0:11
2296	*sub.l.t Vi,Sj,Vk*	*cpu4241.rnn*	*st_295_296.s,st_295_execute.s*	0:19	0:29
4295	*sub.l.f Vi,Sj,Vk*	*cpu4241.rnn*	*st_295_296.s,st_295_execute.s*	0:01	0:11
4296	*sub.l.f Vi,Sj,Vk*	*cpu4241.rnn*	*st_295_296.s,st_295_execute.s*	0:19	0:29
2300	*sub.s.t Vi,Sj,Vk Native Mode*	*cpu4241.rnn*	*st_300_301.s,st_300_execute.s*	0:01	0:11
2301	*sub.s.t Vi,Sj,Vk Native Mode*	*cpu4241.rnn*	*st_300_301.s,st_300_execute.s*	0:10	0:20
3300	*sub.s.t Vi,Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_300_301.s,st_1300_execute.s*	0:01	0:11
3301	*sub.s.t Vi,Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_300_301.s,st_1300_execute.s*	0:16	0:26
4300	*sub.s.f Vi,Sj,Vk Native Mode*	*cpu4241.rnn*	*st_300_301.s,st_300_execute.s*	0:01	0:11
4301	*sub.s.f Vi,Sj,Vk Native Mode*	*cpu4241.rnn*	*st_300_301.s,st_300_execute.s*	0:09	0:19
5300	*sub.s.f Vi,Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_300_301.s,st_1300_execute.s*	0:01	0:11
5301	*sub.s.f Vi,Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_300_301.s,st_1300_execute.s*	0:15	0:25
2305	*sub.d.t Vi,Sj,Vk Native Mode*	*cpu4241.rnn*	*st_305_306.s,st_305_execute.s*	0:01	0:11
2306	*sub.d.t Vi,Sj,Vk Native Mode*	*cpu4241.rnn*	*st_305_306.s,st_305_execute.s*	0:10	0:20
3305	*sub.d.t Vi,Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_305_306.s,st_1305_execute.s*	0:01	0:11
3306	*sub.d.t Vi,Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_305_306.s,st_1305_execute.s*	0:15	0:25
4305	*sub.d.f Vi,Sj,Vk Native Mode*	*cpu4241.rnn*	*st_305_306.s,st_305_execute.s*	0:01	0:11
4306	*sub.d.f Vi,Sj,Vk Native Mode*	*cpu4241.rnn*	*st_305_306.s,st_305_execute.s*	0:10	0:20
5305	*sub.d.f Vi,Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_305_306.s,st_1305_execute.s*	0:02	0:12
5306	*sub.d.f Vi,Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_305_306.s,st_1305_execute.s*	0:15	0:25
2310	*and.t Vi,Sj,Vk*	*cpu4241.rnn*	*st_310_311.s,st_310_execute.s*	0:01	0:11
2311	*and.t Vi,Sj,Vk*	*cpu4241.rnn*	*st_310_311.s,st_310_execute.s*	0:16	0:26
4310	*and.f Vi,Sj,Vk*	*cpu4241.rnn*	*st_310_311.s,st_310_execute.s*	0:01	0:11
4311	*and.f Vi,Sj,Vk*	*cpu4241.rnn*	*st_310_311.s,st_310_execute.s*	0:16	0:26
2315	*or.t Vi,Sj,Vk*	*cpu4241.rnn*	*st_315_316.s,st_315_execute.s*	0:01	0:11
2316	*or.t Vi,Sj,Vk*	*cpu4241.rnn*	*st_315_316.s,st_315_execute.s*	0:16	0:26
4315	*or.f Vi,Sj,Vk*	*cpu4241.rnn*	*st_315_316.s,st_315_execute.s*	0:01	0:12
4316	*or.f Vi,Sj,Vk*	*cpu4241.rnn*	*st_315_316.s,st_315_execute.s*	0:16	0:26
2320	*xor.t Vi,Sj,Vk*	*cpu4241.rnn*	*st_320_321.s,st_320_execute.s*	0:01	0:12
2321	*xor.t Vi,Sj,Vk*	*cpu4241.rnn*	*st_320_321.s,st_320_execute.s*	0:16	0:26

Table cpu4241-3, Class 2 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
4320	*xor.f Vi,Sj,Vk*	*cpu4241.rnn*	*st_320_32.s,st_320_execute.s*	0:01	0:12
4321	*xor.f Vi,Sj,Vk*	*cpu4241.rnn*	*st_320_32.s,st_320_execute.s*	0:16	0:26
2325	*shf.t Sj,Vk*	*cpu4241.rnn*	*st_325_326.s,st_325_execute.s*	0:02	0:12
2326	*shf.t Sj,Vk*	*cpu4241.rnn*	*st_325_326.s,st_325_execute.s*	0:55	1:05
4325	*shf.f Sj,Vk*	*cpu4241.rnn*	*st_325_326.s,st_325_execute.s*	0:02	0:12
4326	*shf.f Sj,Vk*	*cpu4241.rnn*	*st_325_326.s,st_325_execute.s*	0:55	1:05
2330	*le.b.t Vj,Vk*	*cpu4241.rnn*	*st_330_331.s,st_330_execute.s*	0:01	0:11
2331	*le.b.t Vj,Vk*	*cpu4241.rnn*	*st_330_331.s,st_330_execute.s*	0:03	0:13
4330	*le.b.f Vj,Vk*	*cpu4241.rnn*	*st_330_331.s,st_330_execute.s*	0:01	0:11
4331	*le.b.f Vj,Vk*	*cpu4241.rnn*	*st_330_331.s,st_330_execute.s*	0:03	0:13
2335	*le.h.t Vj,Vk*	*cpu4241.rnn*	*st_335_336.s,st_335_execute.s*	0:01	0:11
2336	*le.h.t Vj,Vk*	*cpu4241.rnn*	*st_335_336.s,st_335_execute.s*	0:03	0:13
4335	*le.h.f Vj,Vk*	*cpu4241.rnn*	*st_335_336.s,st_335_execute.s*	0:01	0:11
4336	*le.h.f Vj,Vk*	*cpu4241.rnn*	*st_335_336.s,st_335_execute.s*	0:03	0:13
2340	*le.w.t Vj,Vk*	*cpu4241.rnn*	*st_340_341.s,st_340_execute.s*	0:01	0:11
2341	*le.w.t Vj,Vk*	*cpu4241.rnn*	*st_340_341.s,st_340_execute.s*	0:03	0:13
4340	*le.w.f Vj,Vk*	*cpu4241.rnn*	*st_340_341.s,st_340_execute.s*	0:01	0:11
4341	*le.w.f Vj,Vk*	*cpu4241.rnn*	*st_340_341.s,st_340_execute.s*	0:03	0:13
2345	*le.l.t Vj,Vk*	*cpu4241.rnn*	*st_345_346.s,st_345_execute.s*	0:01	0:11
2346	*le.l.t Vj,Vk*	*cpu4241.rnn*	*st_345_346.s,st_345_execute.s*	0:03	0:13
4345	*le.l.f Vj,Vk*	*cpu4241.rnn*	*st_345_346.s,st_345_execute.s*	0:01	0:11
4346	*le.l.f Vj,Vk*	*cpu4241.rnn*	*st_345_346.s,st_345_execute.s*	0:03	0:13
2350	*le.s.t Vj,Vk Native Mode*	*cpu4241.rnn*	*st_350_351.s,st_350_execute.s*	0:01	0:11
2351	*le.s.t Vj,Vk Native Mode*	*cpu4241.rnn*	*st_350_351.s,st_350_execute.s*	0:09	0:19
3350	*le.s.t Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_350_351.s,st_1350_execute.s*	0:01	0:11
3351	*le.s.t Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_350_351.s,st_1350_execute.s*	0:13	0:23
4350	*le.s.f Vj,Vk Native Mode*	*cpu4241.rnn*	*st_350_351.s,st_350_execute.s*	0:01	0:11
4351	*le.s.f Vj,Vk Native Mode*	*cpu4241.rnn*	*st_350_351.s,st_350_execute.s*	0:08	0:18
5350	*le.s.f Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_350_351.s,st_1350_execute.s*	0:01	0:11
5351	*le.s.f Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_350_351.s,st_1350_execute.s*	0:13	0:13
2355	*le.d.t Vj,Vk Native Mode*	*cpu4241.rnn*	*st_355_356.s,st_355_execute.s*	0:02	0:12
2356	*le.d.t Vj,Vk Native Mode*	*cpu4241.rnn*	*st_355_356.s,st_355_execute.s*	0:09	0:19
3355	*le.d.t Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_355_356.s,st_1355_execute.s*	0:02	0:12
3356	*le.d.t Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_355_356.s,st_1355_execute.s*	0:13	0:23

Table cpu4241-3, Class 2 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
4355	*le.d.f Vj,Vk Native Mode*	*cpu4241.rnn*	*st_355_356.s,st_355_execute.s*	0:01	0:11
4356	*le.d.f Vj,Vk Native Mode*	*cpu4241.rnn*	*st_355_356.s,st_355_execute.s*	0:09	0:19
5355	*le.d.f Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_355_356.s,st_1355_execute.s*	0:01	0:11
5356	*le.d.f Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_355_356.s,st_1355_execute.s*	0:13	0:23
2360	*lt.b.t Vj,Vk*	*cpu4241.rnn*	*st_360_361.s,st_360_execute.s*	0:01	0:11
2361	*lt.b.t Vj,Vk*	*cpu4241.rnn*	*st_360_361.s,st_360_execute.s*	0:03	0:13
4360	*lt.b.f Vj,Vk*	*cpu4241.rnn*	*st_360_361.s,st_360_execute.s*	0:01	0:11
4361	*lt.b.f Vj,Vk*	*cpu4241.rnn*	*st_360_361.s,st_360_execute.s*	0:03	0:13
2365	*lt.h.t Vj,Vk*	*cpu4241.rnn*	*st_365_366.s,st_365_execute.s*	0:01	0:11
2366	*lt.h.t Vj,Vk*	*cpu4241.rnn*	*st_365_366.s,st_365_execute.s*	0:03	0:13
4365	*lt.h.f Vj,Vk*	*cpu4241.rnn*	*st_365_366.s,st_365_execute.s*	0:01	0:11
4366	*lt.h.f Vj,Vk*	*cpu4241.rnn*	*st_365_366.s,st_365_execute.s*	0:03	0:13
2370	*lt.w.t Vj,Vk*	*cpu4241.rnn*	*st_370_371.s,st_370_execute.s*	0:01	0:11
2371	*lt.w.t Vj,Vk*	*cpu4241.rnn*	*st_370_371.s,st_370_execute.s*	0:03	0:13
4370	*lt.w.f Vj,Vk*	*cpu4241.rnn*	*st_370_371.s,st_370_execute.s*	0:01	0:11
4371	*lt.w.f Vj,Vk*	*cpu4241.rnn*	*st_370_371.s,st_370_execute.s*	0:03	0:13
2375	*lt.l.t Vj,Vk*	*cpu4241.rnn*	*st_375_376.s,st_375_execute.s*	0:01	0:11
2376	*lt.l.t Vj,Vk*	*cpu4241.rnn*	*st_375_376.s,st_375_execute.s*	0:03	0:13
4375	*lt.l.f Vj,Vk*	*cpu4241.rnn*	*st_375_376.s,st_375_execute.s*	0:01	0:11
4376	*lt.l.f Vj,Vk*	*cpu4241.rnn*	*st_375_376.s,st_375_execute.s*	0:03	0:13
2380	*lt.s.t Vj,Vk Native Mode*	*cpu4241.rnn*	*st_380_381.s,st_380_execute.s*	0:01	0:11
2381	*lt.s.t Vj,Vk Native Mode*	*cpu4241.rnn*	*st_380_381.s,st_380_execute.s*	0:09	0:19
3380	*lt.s.t Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_380_381.s,st_1380_execute.s*	0:01	0:11
3381	*lt.s.t Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_380_381.s,st_1380_execute.s*	0:13	0:23
4380	*lt.s.f Vj,Vk Native Mode*	*cpu4241.rnn*	*st_380_381.s,st_380_execute.s*	0:01	0:11
4381	*lt.s.f Vj,Vk Native Mode*	*cpu4241.rnn*	*st_380_381.s,st_380_execute.s*	0:09	0:19
5380	*lt.s.f Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_380_381.s,st_1380_execute.s*	0:01	0:11
5381	*lt.s.f Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_380_381.s,st_1380_execute.s*	0:13	0:23
2385	*lt.d.t Vj,Vk Native Mode*	*cpu4241.rnn*	*st_385_386.s,st_385_execute.s*	0:01	0:11
2386	*lt.d.t Vj,Vk Native Mode*	*cpu4241.rnn*	*st_385_386.s,st_385_execute.s*	0:09	0:19
3385	*lt.d.t Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_385_386.s,st_1385_execute.s*	0:02	0:12
3386	*lt.d.t Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_385_386.s,st_1385_execute.s*	0:14	0:24
4385	*lt.d.f Vj,Vk Native Mode*	*cpu4241.rnn*	*st_385_386.s,st_385_execute.s*	0:01	0:11
4386	*lt.d.f Vj,Vk Native Mode*	*cpu4241.rnn*	*st_385_386.s,st_385_execute.s*	0:09	0:19
5385	*lt.d.f Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_385_386.s,st_1385_execute.s*	0:02	0:12
5386	*lt.d.f Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_385_386.s,st_1385_execute.s*	0:13	0:23
2390	*eq.b.t Vj,Vk*	*cpu4241.rnn*	*st_390_391.s,st_390_execute.s*	0:01	0:11
2391	*eq.b.t Vj,Vk*	*cpu4241.rnn*	*st_390_391.s,st_390_execute.s*	0:03	0:13

Table cpu4241-3, Class 2 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
4390	*eq.b.f Vj,Vk*	*cpu4241.rnn*	*st_390_391.s,st_390_execute.s*	0:01	0:11
4391	*eq.b.f Vj,Vk*	*cpu4241.rnn*	*st_390_391.s,st_390_execute.s*	0:03	0:13
2395	*eq.h.t Vj,Vk*	*cpu4241.rnn*	*st_395_396.s,st_395_execute.s*	0:04	0:14
2396	*eq.h.t Vj,Vk*	*cpu4241.rnn*	*st_395_396.s,st_395_execute.s*	0:03	0:13
4395	*eq.h.f Vj,Vk*	*cpu4241.rnn*	*st_395_396.s,st_395_execute.s*	0:03	0:13
4396	*eq.h.f Vj,Vk*	*cpu4241.rnn*	*st_395_396.s,st_395_execute.s*	0:03	0:13
2400	*eq.w.t Vj,Vk*	*cpu4241.rnn*	*st_400_401.s,st_400_execute.s*	0:01	0:11
2401	*eq.w.t Vj,Vk*	*cpu4241.rnn*	*st_400_401.s,st_400_execute.s*	0:03	0:13
4400	*eq.w.f Vj,Vk*	*cpu4241.rnn*	*st_400_401.s,st_400_execute.s*	0:02	0:12
4401	*eq.w.f Vj,Vk*	*cpu4241.rnn*	*st_400_401.s,st_400_execute.s*	0:03	0:13
2405	*eq.l.t Vj,Vk*	*cpu4241.rnn*	*st_405_406.s,st_405_execute.s*	0:01	0:11
2406	*eq.l.t Vj,Vk*	*cpu4241.rnn*	*st_405_406.s,st_405_execute.s*	0:03	0:13
4405	*eq.l.f Vj,Vk*	*cpu4241.rnn*	*st_405_406.s,st_405_execute.s*	0:01	0:11
4406	*eq.l.f Vj,Vk*	*cpu4241.rnn*	*st_405_406.s,st_405_execute.s*	0:03	0:13
2410	*eq.s.t Vj,Vk Native Mode*	*cpu4241.rnn*	*st_410_411.s,st_410_execute.s*	0:01	0:11
2411	*eq.s.t Vj,Vk Native Mode*	*cpu4241.rnn*	*st_410_411.s,st_410_execute.s*	0:09	0:19
3410	*eq.s.t Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_410_411.s,st_1410_execute.s*	0:01	0:11
3411	*eq.s.t Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_410_411.s,st_1410_execute.s*	0:13	0:23
4410	*eq.s.f Vj,Vk Native Mode*	*cpu4241.rnn*	*st_410_411.s,st_410_execute.s*	0:01	0:11
4411	*eq.s.f Vj,Vk Native Mode*	*cpu4241.rnn*	*st_410_411.s,st_410_execute.s*	0:09	0:19
5410	*eq.s.f Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_410_411.s,st_1410_execute.s*	0:01	0:11
5411	*eq.s.f Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_410_411.s,st_1410_execute.s*	0:13	0:23
2415	*eq.d.t Vj,Vk Native Mode*	*cpu4241.rnn*	*st_415_416.s,st_415_execute.s*	0:02	0:12
2416	*eq.d.t Vj,Vk Native Mode*	*cpu4241.rnn*	*st_415_416.s,st_415_execute.s*	0:09	0:19
3415	*eq.d.t Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_415_416.s,st_1415_execute.s*	0:02	0:12
3416	*eq.d.t Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_415_416.s,st_1415_execute.s*	0:13	0:23
4415	*eq.d.f Vj,Vk Native Mode*	*cpu4241.rnn*	*st_415_416.s,st_415_execute.s*	0:01	0:11
4416	*eq.d.f Vj,Vk Native Mode*	*cpu4241.rnn*	*st_415_416.s,st_415_execute.s*	0:09	0:19
5415	*eq.d.f Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_415_416.s,st_1415_execute.s*	0:01	0:11
5416	*eq.d.f Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_415_416.s,st_1415_execute.s*	0:13	0:23
420	*frint.s Vj,Vk Native Mode*	*cpu4241.rnn*	*st_420_421.s,st_420_execute.s*	0:01	0:11
421	*frint.s Vj,Vk Native Mode*	*cpu4241.rnn*	*st_420_421.s,st_420_execute.s*	0:03	0:13
1420	*frint.s Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_420_421.s,st_1420_execute.s*	0:01	0:11
1421	*frint.s Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_420_421.s,st_1420_execute.s*	0:04	0:14
2420	*frint.s.t Vj,Vk Native Mode*	*cpu4241.rnn*	*st_420_421.s,st_420_execute.s*	0:01	0:11
2421	*frint.s.t Vj,Vk Native Mode*	*cpu4241.rnn*	*st_420_421.s,st_420_execute.s*	0:18	0:28
3420	*frint.s.t Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_420_421.s,st_1420_execute.s*	0:01	0:11
3421	*frint.s.t Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_420_421.s,st_1420_execute.s*	0:21	0:31

Table cpu4241-3, Class 2 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
4420	*frint.s.f Vj,Vk Native Mode*	*cpu4241.rnn*	*st_420_421.s,st_420_execute.s*	0:01	0:11
4421	*frint.s.f Vj,Vk Native Mode*	*cpu4241.rnn*	*st_420_421.s,st_420_execute.s*	0:18	0:28
5420	*frint.s.f Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_420_421.s,st_1420_execute.s*	0:01	0:11
5421	*frint.s.f Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_420_421.s,st_1420_execute.s*	0:22	0:32
425	*frint.d Vj,Vk Native Mode*	*cpu4241.rnn*	*st_425_426.s,st_425_execute.s*	0:01	0:11
426	*frint.d Vj,Vk Native Mode*	*cpu4241.rnn*	*st_425_426.s,st_425_execute.s*	0:20	0:30
1425	*frint.d Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_425_426.s,st_1425_execute.s*	0:02	0:12
1426	*frint.d Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_425_426.s,st_1425_execute.s*	0:22	0:32
2425	*frint.d.t Vj,Vk Native Mode*	*cpu4241.rnn*	*st_425_426.s,st_425_execute.s*	0:04	0:14
2426	*frint.d.t Vj,Vk Native Mode*	*cpu4241.rnn*	*st_425_426.s,st_425_execute.s*	2:57	3:07
3425	*frint.d.t Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_425_426.s,st_1425_execute.s*	0:03	0:13
3426	*frint.d.t Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_425_426.s,st_1425_execute.s*	3:08	3:18
4425	*frint.d.f Vj,Vk Native Mode*	*cpu4241.rnn*	*st_425_426.s,st_425_execute.s*	0:03	0:13
4426	*frint.d.f Vj,Vk Native Mode*	*cpu4241.rnn*	*st_425_426.s,st_425_execute.s*	2:57	3:07
5425	*frint.d.f Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_425_426.s,st_1425_execute.s*	0:04	0:14
5426	*frint.d.f Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_425_426.s,st_1425_execute.s*	3:09	3:19
2430	*add.b.t Vi,Vj,Vk*	*cpu4241.rnn*	*st_430_431.s,st_430_execute.s*	0:01	0:11
2431	*add.b.t Vi,Vj,Vk*	*cpu4241.rnn*	*st_430_431.s,st_430_execute.s*	0:25	0:35
4430	*add.b.f Vi,Vj,Vk*	*cpu4241.rnn*	*st_430_431.s,st_430_execute.s*	0:01	0:11
4431	*add.b.f Vi,Vj,Vk*	*cpu4241.rnn*	*st_430_431.s,st_430_execute.s*	0:26	0:36
2435	*add.h.t Vi,Vj,Vk*	*cpu4241.rnn*	*st_435_436.s,st_435_execute.s*	0:02	0:12
2436	*add.h.t Vi,Vj,Vk*	*cpu4241.rnn*	*st_435_436.s,st_435_execute.s*	0:25	0:35
4435	*add.h.f Vi,Vj,Vk*	*cpu4241.rnn*	*st_435_436.s,st_435_execute.s*	0:01	0:11
4436	*add.h.f Vi,Vj,Vk*	*cpu4241.rnn*	*st_435_436.s,st_435_execute.s*	0:25	0:35
2440	*add.w.t Vi,Vj,Vk*	*cpu4241.rnn*	*st_440_441.s,st_440_execute.s*	0:01	0:11
2441	*add.w.t Vi,Vj,Vk*	*cpu4241.rnn*	*st_440_441.s,st_440_execute.s*	0:26	0:36
4440	*add.w.f Vi,Vj,Vk*	*cpu4241.rnn*	*st_440_441.s,st_440_execute.s*	0:02	0:12
4441	*add.w.f Vi,Vj,Vk*	*cpu4241.rnn*	*st_440_441.s,st_440_execute.s*	0:25	0:35
2445	*add.l.t Vi,Vj,Vk*	*cpu4241.rnn*	*st_445_446.s,st_445_execute.s*	0:01	0:11
2446	*add.l.t Vi,Vj,Vk*	*cpu4241.rnn*	*st_445_446.s,st_445_execute.s*	0:25	0:35
4445	*add.l.f Vi,Vj,Vk*	*cpu4241.rnn*	*st_445_446.s,st_445_execute.s*	0:01	0:11
4446	*add.l.f Vi,Vj,Vk*	*cpu4241.rnn*	*st_445_446.s,st_445_execute.s*	0:26	0:36
2450	*add.s.t Vi,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_450_451.s,st_450_execute.s*	0:01	0:11
2451	*add.s.t Vi,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_450_451.s,st_450_execute.s*	0:13	0:23
3450	*add.s.t Vi,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_450_451.s,st_1450_execute.s*	0:01	0:11
3451	*add.s.t Vi,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_450_451.s,st_1450_execute.s*	0:30	0:40
4450	*add.s.f Vi,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_450_451.s,st_450_execute.s*	0:01	0:11
4451	*add.s.f Vi,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_450_451.s,st_450_execute.s*	0:14	0:24
5450	*add.s.f Vi,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_450_451.s,st_1450_execute.s*	0:02	0:12

Table cpu4241-3, Class 2 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
5451	*add.s.f Vi,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_450_451.s,st_1450_execute.s*	0:30	0:40
2455	*add.d.t Vi,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_455_456.s,st_455_execute.s*	0:01	0:11
2456	*add.d.t Vi,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_455_456.s,st_455_execute.s*	0:13	0:23
3455	*add.d.t Vi,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_455_456.s,st_1455_execute.s*	0:02	0:12
3456	*add.d.t Vi,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_455_456.s,st_1455_execute.s*	0:30	0:40
4455	*add.d.f Vi,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_455_456.s,st_455_execute.s*	0:02	0:12
4456	*add.d.f Vi,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_455_456.s,st_455_execute.s*	0:13	0:23
5455	*add.d.f Vi,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_455_456.s,st_1455_execute.s*	0:01	0:11
5456	*add.d.f Vi,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_455_456.s,st_1455_execute.s*	0:30	0:40
2460	*sub.b.t Vi,Vj,Vk*	*cpu4241.rnn*	*st_460_461.s,st_460_execute.s*	0:01	0:11
2461	*sub.b.t Vi,Vj,Vk*	*cpu4241.rnn*	*st_460_461.s,st_460_execute.s*	0:25	0:35
4460	*sub.b.f Vi,Vj,Vk*	*cpu4241.rnn*	*st_460_461.s,st_460_execute.s*	0:01	0:11
4461	*sub.b.f Vi,Vj,Vk*	*cpu4241.rnn*	*st_460_461.s,st_460_execute.s*	0:26	0:36
2465	*sub.h.t Vi,Vj,Vk*	*cpu4241.rnn*	*st_465_466.s,st_465_execute.s*	0:02	0:12
2466	*sub.h.t Vi,Vj,Vk*	*cpu4241.rnn*	*st_465_466.s,st_465_execute.s*	0:25	0:35
4465	*sub.h.f Vi,Vj,Vk*	*cpu4241.rnn*	*st_465_466.s,st_465_execute.s*	0:02	0:12
4466	*sub.h.f Vi,Vj,Vk*	*cpu4241.rnn*	*st_465_466.s,st_465_execute.s*	0:26	0:36
2470	*sub.w.t Vi,Vj,Vk*	*cpu4241.rnn*	*st_470_471.s,st_470_execute.s*	0:01	0:11
2471	*sub.w.t Vi,Vj,Vk*	*cpu4241.rnn*	*st_470_471.s,st_470_execute.s*	0:25	0:35
4470	*sub.w.f Vi,Vj,Vk*	*cpu4241.rnn*	*st_470_471.s,st_470_execute.s*	0:02	0:12
4471	*sub.w.f Vi,Vj,Vk*	*cpu4241.rnn*	*st_470_471.s,st_470_execute.s*	0:26	0:36
2475	*sub.l.t Vi,Vj,Vk*	*cpu4241.rnn*	*st_475_476.s,st_475_execute.s*	0:02	0:12
2476	*sub.l.t Vi,Vj,Vk*	*cpu4241.rnn*	*st_475_476.s,st_475_execute.s*	0:25	0:35
4475	*sub.l.f Vi,Vj,Vk*	*cpu4241.rnn*	*st_475_476.s,st_475_execute.s*	0:01	0:11
4476	*sub.l.f Vi,Vj,Vk*	*cpu4241.rnn*	*st_475_476.s,st_475_execute.s*	0:25	0:35
2480	*sub.s.t Vi,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_480_481.s,st_480_execute.s*	0:01	0:11
2481	*sub.s.t Vi,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_480_481.s,st_480_execute.s*	0:15	0:25
3480	*sub.s.t Vi,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_480_481.s,st_1480_execute.s*	0:02	0:12
3481	*sub.s.t Vi,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_480_481.s,st_1480_execute.s*	0:31	0:41
4480	*sub.s.f Vi,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_480_481.s,st_480_execute.s*	0:01	0:11
4481	*sub.s.f Vi,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_480_481.s,st_480_execute.s*	0:15	0:25
5480	*sub.s.f Vi,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_480_481.s,st_1480_execute.s*	0:02	0:12
5481	*sub.s.f Vi,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_480_481.s,st_1480_execute.s*	0:31	0:41
2485	*sub.d.t Vi,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_485_486.s,st_485_execute.s*	0:01	0:11
2486	*sub.d.t Vi,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_485_486.s,st_485_execute.s*	0:15	0:25
3485	*sub.d.t Vi,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_485_486.s,st_1485_execute.s*	0:02	0:12
3486	*sub.d.t Vi,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_485_486.s,st_1485_execute.s*	0:31	0:41
4485	*sub.d.f Vi,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_485_486.s,st_485_execute.s*	0:01	0:11
4486	*sub.d.f Vi,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_485_486.s,st_485_execute.s*	0:15	0:25

Table cpu4241-3, Class 2 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
5485	*sub.d.f Vi,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_485_486.s,st_1485_execute.s*	0:02	0:12
5486	*sub.d.f Vi,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_485_486.s,st_1485_execute.s*	0:31	0:41
2490	*and.t Vi,Vj,Vk*	*cpu4241.rnn*	*st_490_491.s,st_490_execute.s*	0:01	0:11
2491	*and.t Vi,Vj,Vk*	*cpu4241.rnn*	*st_490_491.s,st_490_execute.s*	0:16	0:26
4490	*and.f Vi,Vj,Vk*	*cpu4241.rnn*	*st_490_491.s,st_490_execute.s*	0:01	0:11
4491	*and.f Vi,Vj,Vk*	*cpu4241.rnn*	*st_490_491.s,st_490_execute.s*	0:16	0:26
2495	*or.t Vi,Vj,Vk*	*cpu4241.rnn*	*st_495_496.s,st_495_execute.s*	0:02	0:12
2496	*or.t Vi,Vj,Vk*	*cpu4241.rnn*	*st_495_496.s,st_495_execute.s*	0:16	0:26
4495	*or.f Vi,Vj,Vk*	*cpu4241.rnn*	*st_495_496.s,st_495_execute.s*	0:01	0:11
4496	*or.f Vi,Vj,Vk*	*cpu4241.rnn*	*st_495_496.s,st_495_execute.s*	0:15	0:25
2500	*xor.t Vi,Vj,Vk*	*cpu4241.rnn*	*st_500_501.s,st_500_execute.s*	0:01	0:11
2501	*xor.t Vi,Vj,Vk*	*cpu4241.rnn*	*st_500_501.s,st_500_execute.s*	0:16	0:26
4500	*xor.f Vi,Vj,Vk*	*cpu4241.rnn*	*st_500_501.s,st_500_execute.s*	0:02	0:12
4501	*xor.f Vi,Vj,Vk*	*cpu4241.rnn*	*st_500_501.s,st_500_execute.s*	0:16	0:26
2505	*not.t Vj,Vk*	*cpu4241.rnn*	*st_505_506.s,st_505_execute.s*	0:01	0:11
2506	*not.t Vj,Vk*	*cpu4241.rnn*	*st_505_506.s,st_505_execute.s*	0:01	0:11
4505	*not.f Vj,Vk*	*cpu4241.rnn*	*st_505_506.s,st_505_execute.s*	0:01	0:11
4506	*not.f Vj,Vk*	*cpu4241.rnn*	*st_505_506.s,st_505_execute.s*	0:02	0:12
2510	*neg.b.t Vj,Vk*	*cpu4241.rnn*	*st_510_511.s,st_510_execute.s*	0:01	0:11
2511	*neg.b.t Vj,Vk*	*cpu4241.rnn*	*st_510_511.s,st_510_execute.s*	0:01	0:11
4510	*neg.b.f Vj,Vk*	*cpu4241.rnn*	*st_510_511.s,st_510_execute.s*	0:01	0:11
4511	*neg.b.f Vj,Vk*	*cpu4241.rnn*	*st_510_511.s,st_510_execute.s*	0:01	0:11
2515	*neg.h.t Vj,Vk*	*cpu4241.rnn*	*st_515_516.s,st_515_execute.s*	0:01	0:11
2516	*neg.h.t Vj,Vk*	*cpu4241.rnn*	*st_515_516.s,st_515_execute.s*	0:02	0:12

Table cpu4241-3, Class 2 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
4515	*neg.h.f Vj,Vk*	cpu4241.rnn	st_515_516.s,st_515_execute.s	0:01	0:11
4516	*neg.h.f Vj,Vk*	cpu4241.rnn	st_515_516.s,st_515_execute.s	0:02	0:12
2520	*neg.w.t Vj,Vk*	cpu4241.rnn	st_520_521.s,st_520_execute.s	0:01	0:11
2521	*neg.w.t Vj,Vk*	cpu4241.rnn	st_520_521.s,st_520_execute.s	0:01	0:11
4520	*neg.w.f Vj,Vk*	cpu4241.rnn	st_520_521.s,st_520_execute.s	0:01	0:11
4521	*neg.w.f Vj,Vk*	cpu4241.rnn	st_520_521.s,st_520_execute.s	0:01	0:11
2525	*neg.l.t Vj,Vk*	cpu4241.rnn	st_525_526.s,st_525_execute.s	0:01	0:11
2526	*neg.l.t Vj,Vk*	cpu4241.rnn	st_525_526.s,st_525_execute.s	0:02	0:12
4525	*neg.l.f Vj,Vk*	cpu4241.rnn	st_525_526.s,st_525_execute.s	0:01	0:11
4526	*neg.l.f Vj,Vk*	cpu4241.rnn	st_525_526.s,st_525_execute.s	0:02	0:12
2530	*neg.s.t Vj,Vk Native Mode*	cpu4241.rnn	st_530_531.s,st_530_execute.s	0:02	0:12
2531	*neg.s.t Vj,Vk Native Mode*	cpu4241.rnn	st_530_531.s,st_530_execute.s	0:14	0:24
3530	*neg.s.t Vj,Vk IEEE Mode*	cpu4241.rnn	st_530_531.s,st_1530_execute.s	0:01	0:11
3531	*neg.s.t Vj,Vk IEEE Mode*	cpu4241.rnn	st_530_531.s,st_1530_execute.s	0:09	0:19
4530	*neg.s.f Vj,Vk Native Mode*	cpu4241.rnn	st_530_531.s,st_530_execute.s	0:01	0:11
4531	*neg.s.f Vj,Vk Native Mode*	cpu4241.rnn	st_530_531.s,st_530_execute.s	0:13	0:23
5530	*neg.s.f Vj,Vk IEEE Mode*	cpu4241.rnn	st_530_531.s,st_1530_execute.s	0:01	0:11
5531	*neg.s.f Vj,Vk IEEE Mode*	cpu4241.rnn	st_530_531.s,st_1530_execute.s	0:09	0:19
2535	*neg.d.t Vj,Vk Native Mode*	cpu4241.rnn	st_535_536.s,st_535_execute.s	0:01	0:11
2536	*neg.d.t Vj,Vk Native Mode*	cpu4241.rnn	st_535_536.s,st_535_execute.s	0:14	0:24
3535	*neg.d.t Vj,Vk IEEE Mode*	cpu4241.rnn	st_535_536.s,st_1535_execute.s	0:02	0:12
3536	*neg.d.t Vj,Vk IEEE Mode*	cpu4241.rnn	st_535_536.s,st_1535_execute.s	0:09	0:19
4535	*neg.d.f Vj,Vk Native Mode*	cpu4241.rnn	st_535_536.s,st_535_execute.s	0:02	0:12
4536	*neg.d.f Vj,Vk Native Mode*	cpu4241.rnn	st_535_536.s,st_535_execute.s	0:14	0:24
5535	*neg.d.f Vj,Vk IEEE Mode*	cpu4241.rnn	st_535_536.s,st_1535_execute.s	0:01	0:11
5536	*neg.d.f Vj,Vk IEEE Mode*	cpu4241.rnn	st_535_536.s,st_1535_execute.s	0:09	0:19
540	*sub.s Si,Vj,Vk Native Mode*	cpu4241.rnn	st_540_541.s,st_540_execute.s	0:01	0:11
541	*sub.s Si,Vj,Vk Native Mode*	cpu4241.rnn	st_540_541.s,st_540_execute.s	0:02	0:12
1540	*sub.s Si,Vj,Vk IEEE Mode*	cpu4241.rnn	st_540_541.s,st_1540_execute.s	0:01	0:11
1541	*sub.s Si,Vj,Vk IEEE Mode*	cpu4241.rnn	st_540_541.s,st_1540_execute.s	0:03	0:13
2540	*sub.s.t Si,Vj,Vk Native Mode*	cpu4241.rnn	st_540_541.s,st_540_execute.s	0:01	0:11
2541	*sub.s.t Si,Vj,Vk Native Mode*	cpu4241.rnn	st_540_541.s,st_540_execute.s	0:11	0:21
3540	*sub.s.t Si,Vj,Vk IEEE Mode*	cpu4241.rnn	st_540_541.s,st_1540_execute.s	0:01	0:11
3541	*sub.s.t Si,Vj,Vk IEEE Mode*	cpu4241.rnn	st_540_541.s,st_1540_execute.s	0:15	0:25

Table cpu4241-3, Class 2 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
4540	*sub.s.f Si,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_540_541.s,st_540_execute.s*	0:01	0:11
4541	*sub.s.f Si,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_540_541.s,st_540_execute.s*	0:12	0:22
5540	*sub.s.f Si,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_540_541.s,st_1540_execute.s*	0:02	0:12
5541	*sub.s.f Si,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_540_541.s,st_1540_execute.s*	0:16	0:26
545	*sub.d Si,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_545_546.s,st_545_execute.s*	0:01	0:11
546	*sub.d Si,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_545_546.s,st_545_execute.s*	0:02	0:12
1545	*sub.d Si,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_545_546.s,st_1545_execute.s*	0:01	0:11
1546	*sub.d Si,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_545_546.s,st_1545_execute.s*	0:03	0:13
2545	*sub.d.t Si,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_545_546.s,st_545_execute.s*	0:01	0:11
2546	*sub.d.t Si,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_545_546.s,st_545_execute.s*	0:12	0:22
3545	*sub.d.t Si,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_545_546.s,st_1545_execute.s*	0:02	0:12
3546	*sub.d.t Si,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_545_546.s,st_1545_execute.s*	0:16	0:26
4545	*sub.d.f Si,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_545_546.s,st_545_execute.s*	0:01	0:11
4546	*sub.d.f Si,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_545_546.s,st_545_execute.s*	0:11	0:21
5545	*sub.d.f Si,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_545_546.s,st_1545_execute.s*	0:01	0:11
5546	*sub.d.f Si,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_545_546.s,st_1545_execute.s*	0:15	0:25
795	*shf Vi,Vj,Vk*	*cpu4241.rnn*	*st_795_796.s,st_795_execute.s*	0:01	0:11
796	*shf Vi,Vj,Vk*	*cpu4241.rnn*	*st_795_796.s,st_795_execute.s*	0:08	0:18
2795	*shf.t Vi,Vj,Vk*	*cpu4241.rnn*	*st_795_796.s,st_795_execute.s*	0:02	0:12
2796	*shf.t Vi,Vj,Vk*	*cpu4241.rnn*	*st_795_796.s,st_795_execute.s*	1:11	1:21
4795	*shf.f Vi,Vj,Vk*	*cpu4241.rnn*	*st_795_796.s,st_795_execute.s*	0:02	0:12
4796	*shf.f Vi,Vj,Vk*	*cpu4241.rnn*	*st_795_796.s,st_795_execute.s*	1:11	1:21
2805	*plc.t.t Vj,Vk*	*cpu4241.rnn*	*st_805_806.s,st_805_execute.s*	0:02	0:12
2806	*plc.t.t Vj,Vk*	*cpu4241.rnn*	*st_805_806.s,st_805_execute.s*	0:31	0:41

Table cpu4241-3, Class 2 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
4805	*plc.t.f Vj,Vk*	*cpu4241.rnn*	*st_805_806.s,st_805_execute.s*	0:01	0:11
4806	*plc.t.f Vj,Vk*	*cpu4241.rnn*	*st_805_806.s,st_805_execute.s*	0:31	0:41
2815	*xpnd.t.t Vj,Vk*	*cpu4241.rnn*	*st_815_816.s,st_815_execute.s*	0:01	0:11
2816	*xpnd.t.t Vj,Vk*	*cpu4241.rnn*	*st_815_816.s,st_815_execute.s*	0:02	0:12
4815	*xpnd.t.f Vj,Vk*	*cpu4241.rnn*	*st_815_816.s,st_815_execute.s*	0:01	0:11
4816	*xpnd.t.f Vj,Vk*	*cpu4241.rnn*	*st_815_816.s,st_815_execute.s*	0:01	0:11
2820	*sum.b.t Vk*	*cpu4241.rnn*	*st_820_821.s,st_820_execute.s*	0:01	0:11
2821	*sum.b.t Vk*	*cpu4241.rnn*	*st_820_821.s,st_820_execute.s*	0:42	0:52
4820	*sum.b.f Vk*	*cpu4241.rnn*	*st_820_821.s,st_820_execute.s*	0:01	0:11
4821	*sum.b.f Vk*	*cpu4241.rnn*	*st_820_821.s,st_820_execute.s*	0:42	0:52
2825	*sum.h.t Vk*	*cpu4241.rnn*	*st_825_826.s,st_825_execute.s*	0:01	0:11
2826	*sum.h.t Vk*	*cpu4241.rnn*	*st_825_826.s,st_825_execute.s*	0:42	0:52
4825	*sum.h.f Vk*	*cpu4241.rnn*	*st_825_826.s,st_825_execute.s*	0:01	0:11
4826	*sum.h.f Vk*	*cpu4241.rnn*	*st_825_826.s,st_825_execute.s*	0:42	0:52
2830	*sum.w.t Vk*	*cpu4241.rnn*	*st_830_831.s,st_830_execute.s*	0:02	0:12
2831	*sum.w.t Vk*	*cpu4241.rnn*	*st_830_831.s,st_830_execute.s*	0:42	0:52
4830	*sum.w.f Vk*	*cpu4241.rnn*	*st_830_831.s,st_830_execute.s*	0:01	0:11
4831	*sum.w.f Vk*	*cpu4241.rnn*	*st_830_831.s,st_830_execute.s*	0:42	0:52
2835	*sum.l.t Vk*	*cpu4241.rnn*	*st_835_836.s,st_835_execute.s*	0:01	0:11
2836	*sum.l.t Vk*	*cpu4241.rnn*	*st_835_836.s,st_835_execute.s*	0:42	0:52
4835	*sum.l.f Vk*	*cpu4241.rnn*	*st_835_836.s,st_835_execute.s*	0:01	0:11
4836	*sum.l.f Vk*	*cpu4241.rnn*	*st_835_836.s,st_835_execute.s*	0:42	0:52
2840	*sum.s.t Vk Native Mode*	*cpu4241.rnn*	*st_840_841.s,st_840_execute.s*	0:02	0:12
2841	*sum.s.t Vk Native Mode*	*cpu4241.rnn*	*st_840_841.s,st_840_execute.s*	1:05	1:15
3840	*sum.s.t Vk IEEE Mode*	*cpu4241.rnn*	*st_840_841.s,st_1840_execute.s*	0:02	0:12
3841	*sum.s.t Vk IEEE Mode*	*cpu4241.rnn*	*st_840_841.s,st_1840_execute.s*	1:08	1:18
4840	*sum.s.f Vk Native Mode*	*cpu4241.rnn*	*st_840_841.s,st_840_execute.s*	0:02	0:12
4841	*sum.s.f Vk Native Mode*	*cpu4241.rnn*	*st_840_841.s,st_840_execute.s*	1:04	1:14
5840	*sum.s.f Vk IEEE Mode*	*cpu4241.rnn*	*st_840_841.s,st_1840_execute.s*	0:02	0:12
5841	*sum.s.f Vk IEEE Mode*	*cpu4241.rnn*	*st_840_841.s,st_1840_execute.s*	1:09	1:19
2845	*sum.d.t Vk Native Mode*	*cpu4241.rnn*	*st_845_846.s,st_845_execute.s*	0:01	0:11
2846	*sum.d.t Vk Native Mode*	*cpu4241.rnn*	*st_845_846.s,st_845_execute.s*	1:05	1:15
3845	*sum.d.t Vk IEEE Mode*	*cpu4241.rnn*	*st_845_846.s,st_1845_execute.s*	0:02	0:12
3846	*sum.d.t Vk IEEE Mode*	*cpu4241.rnn*	*st_845_846.s,st_1845_execute.s*	1:10	1:10
4845	*sum.d.f Vk Native Mode*	*cpu4241.rnn*	*st_845_846.s,st_845_execute.s*	0:02	0:12
4846	*sum.d.f Vk Native Mode*	*cpu4241.rnn*	*st_845_846.s,st_845_execute.s*	1:05	1:15
5845	*sum.d.f Vk IEEE Mode*	*cpu4241.rnn*	*st_845_846.s,st_1845_execute.s*	0:02	0:12
5846	*sum.d.f Vk IEEE Mode*	*cpu4241.rnn*	*st_845_846.s,st_1845_execute.s*	1:10	1:20

Table cpu4241-3, Class 2 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
2880	*max.b.t Vk*	*cpu4241.rnn*	*st_880_881.s,st_880_execute.s*	0:04	0:14
2881	*max.b.t Vk*	*cpu4241.rnn*	*st_880_881.s,st_880_execute.s*	2:17	2:27
4880	*max.b.f Vk*	*cpu4241.rnn*	*st_880_881.s,st_880_execute.s*	0:04	0:14
4881	*max.b.f Vk*	*cpu4241.rnn*	*st_880_881.s,st_880_execute.s*	2:20	2:30
2885	*max.h.t Vk*	*cpu4241.rnn*	*st_885_886.s,st_885_execute.s*	0:03	0:13
2886	*max.h.t Vk*	*cpu4241.rnn*	*st_885_886.s,st_885_execute.s*	2:17	2:27
4885	*max.h.f Vk*	*cpu4241.rnn*	*st_885_886.s,st_885_execute.s*	0:04	0:14
4886	*max.h.f Vk*	*cpu4241.rnn*	*st_885_886.s,st_885_execute.s*	2:20	2:30
2890	*max.w.t Vk*	*cpu4241.rnn*	*st_890_891.s,st_890_execute.s*	0:04	0:14
2891	*max.w.t Vk*	*cpu4241.rnn*	*st_890_891.s,st_890_execute.s*	2:18	2:28
4890	*max.w.f Vk*	*cpu4241.rnn*	*st_890_891.s,st_890_execute.s*	0:03	0:13
4891	*max.w.f Vk*	*cpu4241.rnn*	*st_890_891.s,st_890_execute.s*	2:21	2:31
2895	*max.l.t Vk*	*cpu4241.rnn*	*st_895_896.s,st_895_execute.s*	0:04	0:14
2896	*max.l.t Vk*	*cpu4241.rnn*	*st_895_896.s,st_895_execute.s*	2:19	2:29
4895	*max.l.f Vk*	*cpu4241.rnn*	*st_895_896.s,st_895_execute.s*	0:04	0:14
4896	*max.l.f Vk*	*cpu4241.rnn*	*st_895_896.s,st_895_execute.s*	2:22	2:32
2900	*max.s.t Vk Native Mode*	*cpu4241.rnn*	*st_900_901.s,st_900_execute.s*	0:04	0:14
2901	*max.s.t Vk Native Mode*	*cpu4241.rnn*	*st_900_901.s,st_900_execute.s*	2:19	2:29
3900	*max.s.t Vk IEEE Mode*	*cpu4241.rnn*	*st_900_901.s,st_1900_execute.s*	0:03	0:13
3901	*max.s.t Vk IEEE Mode*	*cpu4241.rnn*	*st_900_901.s,st_1900_execute.s*	2:22	2:32
4900	*max.s.f Vk Native Mode*	*cpu4241.rnn*	*st_900_901.s,st_900_execute.s*	0:04	0:14
4901	*max.s.f Vk Native Mode*	*cpu4241.rnn*	*st_900_901.s,st_900_execute.s*	2:23	2:33
5900	*max.s.f Vk IEEE Mode*	*cpu4241.rnn*	*st_900_901.s,st_1900_execute.s*	0:04	0:14
5901	*max.s.f Vk IEEE Mode*	*cpu4241.rnn*	*st_900_901.s,st_1900_execute.s*	2:26	2:36
2905	*max.d.t Vk Native Mode*	*cpu4241.rnn*	*st_905_906.s,st_905_execute.s*	0:04	0:14
2906	*max.d.t Vk Native Mode*	*cpu4241.rnn*	*st_905_906.s,st_905_execute.s*	2:20	2:30
3905	*max.d.t Vk IEEE Mode*	*cpu4241.rnn*	*st_905_906.s,st_1905_execute.s*	0:04	0:14
3906	*max.d.t Vk IEEE Mode*	*cpu4241.rnn*	*st_905_906.s,st_1905_execute.s*	2:22	2:32
4905	*max.d.f Vk Native Mode*	*cpu4241.rnn*	*st_905_906.s,st_905_execute.s*	0:04	0:14
4906	*max.d.f Vk Native Mode*	*cpu4241.rnn*	*st_905_906.s,st_905_execute.s*	2:23	2:33
5905	*max.d.f Vk IEEE Mode*	*cpu4241.rnn*	*st_905_906.s,st_1905_execute.s*	0:04	0:14
5906	*max.d.f Vk IEEE Mode*	*cpu4241.rnn*	*st_905_906.s,st_1905_execute.s*	2:26	2:36
2910	*min.b.t Vk*	*cpu4241.rnn*	*st_910_911.s,st_910_execute.s*	0:03	0:13
2911	*min.b.t Vk*	*cpu4241.rnn*	*st_910_911.s,st_910_execute.s*	1:44	1:54
4910	*min.b.f Vk*	*cpu4241.rnn*	*st_910_911.s,st_910_execute.s*	0:03	0:13
4911	*min.b.f Vk*	*cpu4241.rnn*	*st_910_911.s,st_910_execute.s*	1:46	1:56

Table cpu4241-3, Class 2 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
2915	*min.h.t Vk*	*cpu4241.rnn*	*st_915_916.s,st_915_execute.s*	0:03	0:13
2916	*min.h.t Vk*	*cpu4241.rnn*	*st_915_916.s,st_915_execute.s*	2:16	2:26
4915	*min.h.f Vk*	*cpu4241.rnn*	*st_915_916.s,st_915_execute.s*	0:03	0:13
4916	*min.h.f Vk*	*cpu4241.rnn*	*st_915_916.s,st_915_execute.s*	2:20	2:30
2920	*min.w.t Vk*	*cpu4241.rnn*	*st_920_921.s,st_920_execute.s*	0:03	0:13
2921	*min.w.t Vk*	*cpu4241.rnn*	*st_920_921.s,st_920_execute.s*	2:16	2:26
4920	*min.w.f Vk*	*cpu4241.rnn*	*st_920_921.s,st_920_execute.s*	0:03	0:13
4921	*min.w.f Vk*	*cpu4241.rnn*	*st_920_921.s,st_920_execute.s*	2:19	2:29
2925	*min.l.t Vk*	*cpu4241.rnn*	*st_925_926.s,st_925_execute.s*	0:03	0:13
2926	*min.l.t Vk*	*cpu4241.rnn*	*st_925_926.s,st_925_execute.s*	2:18	2:28
4925	*min.l.f Vk*	*cpu4241.rnn*	*st_925_926.s,st_925_execute.s*	0:04	0:14
4926	*min.l.f Vk*	*cpu4241.rnn*	*st_925_926.s,st_925_execute.s*	2:22	2:32
2930	*min.s.t Vk Native Mode*	*cpu4241.rnn*	*st_930_931.s,st_930_execute.s*	0:03	0:13
2931	*min.s.t Vk Native Mode*	*cpu4241.rnn*	*st_930_931.s,st_930_execute.s*	2:19	2:29
3930	*min.s.t Vk IEEE Mode*	*cpu4241.rnn*	*st_930_931.s,st_1930_execute.s*	0:04	0:14
3931	*min.s.t Vk IEEE Mode*	*cpu4241.rnn*	*st_930_931.s,st_1930_execute.s*	2:22	2:32
4930	*min.s.f Vk Native Mode*	*cpu4241.rnn*	*st_930_931.s,st_930_execute.s*	0:04	0:14
4931	*min.s.f Vk Native Mode*	*cpu4241.rnn*	*st_930_931.s,st_930_execute.s*	2:22	2:32
5930	*min.s.f Vk IEEE Mode*	*cpu4241.rnn*	*st_930_931.s,st_1930_execute.s*	0:03	0:13
5931	*min.s.f Vk IEEE Mode*	*cpu4241.rnn*	*st_930_931.s,st_1930_execute.s*	2:25	2:35
2935	*min.d.t Vk Native Mode*	*cpu4241.rnn*	*st_935_936.s,st_935_execute.s*	0:04	0:14
2936	*min.d.t Vk Native Mode*	*cpu4241.rnn*	*st_935_936.s,st_935_execute.s*	2:18	2:28
3935	*min.d.t Vk IEEE Mode*	*cpu4241.rnn*	*st_935_936.s,st_1935_execute.s*	0:04	0:14
3936	*min.d.t Vk IEEE Mode*	*cpu4241.rnn*	*st_935_936.s,st_1935_execute.s*	2:22	2:32
4935	*min.d.f Vk Native Mode*	*cpu4241.rnn*	*st_935_936.s,st_935_execute.s*	0:03	0:13
4936	*min.d.f Vk Native Mode*	*cpu4241.rnn*	*st_935_936.s,st_935_execute.s*	2:21	2:31
5935	*min.d.f Vk IEEE Mode*	*cpu4241.rnn*	*st_935_936.s,st_1935_execute.s*	0:04	0:14
5936	*min.d.f Vk IEEE Mode*	*cpu4241.rnn*	*st_935_936.s,st_1935_execute.s*	2:25	2:35
2940	*all.t Vk*	*cpu4241.rnn*	*st_940_941.s,st_940_execute.s*	0:01	0:11
2941	*all.t Vk*	*cpu4241.rnn*	*st_940_941.s,st_940_execute.s*	0:35	0:45
4940	*all.f Vk*	*cpu4241.rnn*	*st_940_941.s,st_940_execute.s*	0:01	0:11
4941	*all.f Vk*	*cpu4241.rnn*	*st_940_941.s,st_940_execute.s*	0:35	0:45

Table cpu4241-3, Class 2 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
2945	*any.t Vk*	*cpu4241.rnn*	*st_945_946.s,st_945_execute.s*	0:01	0:11
2946	*any.t Vk*	*cpu4241.rnn*	*st_945_946.s,st_945_execute.s*	0:35	0:45
4945	*any.f Vk*	*cpu4241.rnn*	*st_945_946.s,st_945_execute.s*	0:02	0:12
4946	*any.f Vk*	*cpu4241.rnn*	*st_945_946.s,st_945_execute.s*	0:34	0:44
2950	*parity.t Vk*	*cpu4241.rnn*	*st_950_951.s,st_950_execute.s*	0:07	0:17
2951	*parity.t Vk*	*cpu4241.rnn*	*st_950_951.s,st_950_execute.s*	5:45	5:55
4950	*parity.f Vk*	*cpu4241.rnn*	*st_950_951.s,st_950_execute.s*	0:08	0:18
4951	*parity.f Vk*	*cpu4241.rnn*	*st_950_951.s,st_950_execute.s*	5:59	6:09
955	*cvtb.w Vj,Vk*	*cpu4241.rnn*	*st_955_956.s,st_955_execute.s*	0:01	0:11
956	*cvtb.w Vj,Vk*	*cpu4241.rnn*	*st_955_956.s,st_955_execute.s*	0:01	0:11
2955	*cvtb.w.t Vj,Vk*	*cpu4241.rnn*	*st_955_956.s,st_955_execute.s*	0:02	0:12
2956	*cvtb.w.t Vj,Vk*	*cpu4241.rnn*	*st_955_956.s,st_955_execute.s*	0:08	0:18
4955	*cvtb.w.f Vj,Vk*	*cpu4241.rnn*	*st_955_956.s,st_955_execute.s*	0:01	0:11
4956	*cvtb.w.f Vj,Vk*	*cpu4241.rnn*	*st_955_956.s,st_955_execute.s*	0:09	0:19
957	*cvth.w Vj,Vk*	*cpu4241.rnn*	*st_957_958.s,st_957_execute.s*	0:01	0:11
958	*cvth.w Vj,Vk*	*cpu4241.rnn*	*st_957_958.s,st_957_execute.s*	0:02	0:12
2957	*cvth.w.t Vj,Vk*	*cpu4241.rnn*	*st_957_958.s,st_957_execute.s*	0:01	0:11
2958	*cvth.w.t Vj,Vk*	*cpu4241.rnn*	*st_957_958.s,st_957_execute.s*	0:15	0:25
4957	*cvth.w.f Vj,Vk*	*cpu4241.rnn*	*st_957_958.s,st_957_execute.s*	0:01	0:11
4958	*cvth.w.f Vj,Vk*	*cpu4241.rnn*	*st_957_958.s,st_957_execute.s*	0:16	0:26
960	*cvtw.b Vj,Vk*	*cpu4241.rnn*	*st_960_961.s,st_960_execute.s*	0:01	0:11
961	*cvtw.b Vj,Vk*	*cpu4241.rnn*	*st_960_961.s,st_960_execute.s*	0:04	0:14
2960	*cvtw.b.t Vj,Vk*	*cpu4241.rnn*	*st_960_961.s,st_960_execute.s*	0:01	0:11
2961	*cvtw.b.t Vj,Vk*	*cpu4241.rnn*	*st_960_961.s,st_960_execute.s*	0:33	0:43
4960	*cvtw.b.f Vj,Vk*	*cpu4241.rnn*	*st_960_961.s,st_960_execute.s*	0:01	0:11
4961	*cvtw.b.f Vj,Vk*	*cpu4241.rnn*	*st_960_961.s,st_960_execute.s*	0:33	0:43
962	*cvtw.h Vj,Vk*	*cpu4241.rnn*	*st_962_963.s,st_962_execute.s*	0:01	0:11
963	*cvtw.h Vj,Vk*	*cpu4241.rnn*	*st_962_963.s,st_962_execute.s*	0:04	0:14
2962	*cvtw.h.t Vj,Vk*	*cpu4241.rnn*	*st_962_963.s,st_962_execute.s*	0:01	0:11
2963	*cvtw.h.t Vj,Vk*	*cpu4241.rnn*	*st_962_963.s,st_962_execute.s*	0:33	0:43
4962	*cvtw.h.f Vj,Vk*	*cpu4241.rnn*	*st_962_963.s,st_962_execute.s*	0:02	0:12
4963	*cvtw.h.f Vj,Vk*	*cpu4241.rnn*	*st_962_963.s,st_962_execute.s*	0:33	0:43
965	*cvtw.l Vj,Vk*	*cpu4241.rnn*	*st_965_966.s,st_965_execute.s*	0:01	0:11
966	*cvtw.l Vj,Vk*	*cpu4241.rnn*	*st_965_966.s,st_965_execute.s*	0:04	0:14
2965	*cvtw.l.t Vj,Vk*	*cpu4241.rnn*	*st_965_966.s,st_965_execute.s*	0:02	0:12
2966	*cvtw.l.t Vj,Vk*	*cpu4241.rnn*	*st_965_966.s,st_965_execute.s*	0:34	0:44

Table cpu4241-3, Class 2 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
4965	*cvtw.l.f Vj,Vk*	*cpu4241.rnn*	*st_965_966.s,st_965_execute.s*	0:02	0:12
4966	*cvtw.l.f Vj,Vk*	*cpu4241.rnn*	*st_965_966.s,st_965_execute.s*	0:35	0:45
967	*cvtw.s Vj,Vk Native Mode*	*cpu4241.rnn*	*st_967_968.s,st_967_execute.s*	0:01	0:11
968	*cvtw.s Vj,Vk Native Mode*	*cpu4241.rnn*	*st_967_968.s,st_967_execute.s*	0:05	0:15
1967	*cvtw.s Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_967_968.s,st_1967_execute.s*	0:01	0:11
1968	*cvtw.s Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_967_968.s,st_1967_execute.s*	0:05	0:15
2967	*cvtw.s.t Vj,Vk Native Mode*	*cpu4241.rnn*	*st_967_968.s,st_967_execute.s*	0:02	0:12
2968	*cvtw.s.t Vj,Vk Native Mode*	*cpu4241.rnn*	*st_967_968.s,st_967_execute.s*	0:39	0:49
3967	*cvtw.s.t Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_967_968.s,st_1967_execute.s*	0:02	0:12
3968	*cvtw.s.t Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_967_968.s,st_1967_execute.s*	0:39	0:49
4967	*cvtw.s.f Vj,Vk Native Mode*	*cpu4241.rnn*	*st_967_968.s,st_967_execute.s*	0:02	0:12
4968	*cvtw.s.f Vj,Vk Native Mode*	*cpu4241.rnn*	*st_967_968.s,st_967_execute.s*	0:39	0:49
5967	*cvtw.s.f Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_967_968.s,st_1967_execute.s*	0:01	0:11
5968	*cvtw.s.f Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_967_968.s,st_1967_execute.s*	0:40	0:50
970	*cvtw.d Vj,Vk Native Mode*	*cpu4241.rnn*	*st_970_971.s,st_970_execute.s*	0:01	0:11
971	*cvtw.d Vj,Vk Native Mode*	*cpu4241.rnn*	*st_970_971.s,st_970_execute.s*	0:05	0:15
1970	*cvtw.d Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_970_971.s,st_1970_execute.s*	0:01	0:11
1971	*cvtw.d Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_970_971.s,st_1970_execute.s*	0:05	0:15
2970	*cvtw.d.t Vj,Vk Native Mode*	*cpu4241.rnn*	*st_970_971.s,st_970_execute.s*	0:01	0:11
2971	*cvtw.d.t Vj,Vk Native Mode*	*cpu4241.rnn*	*st_970_971.s,st_970_execute.s*	0:40	0:50
3970	*cvtw.d.t Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_970_971.s,st_1970_execute.s*	0:01	0:11
3971	*cvtw.d.t Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_970_971.s,st_1970_execute.s*	0:39	0:49
4970	*cvtw.d.f Vj,Vk Native Mode*	*cpu4241.rnn*	*st_970_971.s,st_970_execute.s*	0:02	0:12
4971	*cvtw.d.f Vj,Vk Native Mode*	*cpu4241.rnn*	*st_970_971.s,st_970_execute.s*	0:39	0:49
5970	*cvtw.d.f Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_970_971.s,st_1970_execute.s*	0:02	0:12
5971	*cvtw.d.f Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_970_971.s,st_1970_execute.s*	0:39	0:49
975	*cvtl.w Vj,Vk*	*cpu4241.rnn*	*st_975_976.s,st_975_execute.s*	0:01	0:11
976	*cvtl.w Vj,Vk*	*cpu4241.rnn*	*st_975_976.s,st_975_execute.s*	0:09	0:19
2975	*cvtl.w.t Vj,Vk*	*cpu4241.rnn*	*st_975_976.s,st_975_execute.s*	0:02	0:12
2976	*cvtl.w.t Vj,Vk*	*cpu4241.rnn*	*st_975_976.s,st_975_execute.s*	1:15	1:25

Table cpu4241-3, Class 2 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
4975	*cvtl.w.f Vj,Vk*	*cpu4241.rnn*	*st_975_976.s,st_975_execute.s*	0:02	0:12
4976	*cvtl.w.f Vj,Vk*	*cpu4241.rnn*	*st_975_976.s,st_975_execute.s*	1:16	1:26
977	*cvtl.s Vj,Vk Native Mode*	*cpu4241.rnn*	*st_977_978.s,st_977_execute.s*	0:01	0:11
978	*cvtl.s Vj,Vk Native Mode*	*cpu4241.rnn*	*st_977_978.s,st_977_execute.s*	0:05	0:15
1977	*cvtl.s Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_977_978.s,st_1977_execute.s*	0:01	0:11
1978	*cvtl.s Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_977_978.s,st_1977_execute.s*	0:05	0:15
2977	*cvtl.s.t Vj,Vk Native Mode*	*cpu4241.rnn*	*st_977_978.s,st_977_execute.s*	0:01	0:11
2978	*cvtl.s.t Vj,Vk Native Mode*	*cpu4241.rnn*	*st_977_978.s,st_977_execute.s*	0:40	0:50
3977	*cvtl.s.t Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_977_978.s,st_1977_execute.s*	0:01	0:11
3978	*cvtl.s.t Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_977_978.s,st_1977_execute.s*	0:40	0:50
4977	*cvtl.s.f Vj,Vk Native Mode*	*cpu4241.rnn*	*st_977_978.s,st_977_execute.s*	0:01	0:11
4978	*cvtl.s.f Vj,Vk Native Mode*	*cpu4241.rnn*	*st_977_978.s,st_977_execute.s*	0:39	0:49
5977	*cvtl.s.f Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_977_978.s,st_1977_execute.s*	0:02	0:12
5978	*cvtl.s.f Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_977_978.s,st_1977_execute.s*	0:39	0:49
980	*cvtl.d Vj,Vk Native Mode*	*cpu4241.rnn*	*st_980_981.s,st_980_execute.s*	0:01	0:11
981	*cvtl.d Vj,Vk Native Mode*	*cpu4241.rnn*	*st_980_981.s,st_980_execute.s*	0:05	0:15
1980	*cvtl.d Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_980_981.s,st_1980_execute.s*	0:01	0:11
1981	*cvtl.d Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_980_981.s,st_1980_execute.s*	0:05	0:15
2980	*cvtl.d.t Vj,Vk Native Mode*	*cpu4241.rnn*	*st_980_981.s,st_980_execute.s*	0:02	0:12
2981	*cvtl.d.t Vj,Vk Native Mode*	*cpu4241.rnn*	*st_980_981.s,st_980_execute.s*	0:39	0:49
3980	*cvtl.d.t Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_980_981.s,st_1980_execute.s*	0:01	0:11
3981	*cvtl.d.t Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_980_981.s,st_1980_execute.s*	0:40	0:50
4980	*cvtl.d.f Vj,Vk Native Mode*	*cpu4241.rnn*	*st_980_981.s,st_980_execute.s*	0:01	0:11
4981	*cvtl.d.f Vj,Vk Native Mode*	*cpu4241.rnn*	*st_980_981.s,st_980_execute.s*	0:40	0:50
5980	*cvtl.d.f Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_980_981.s,st_1980_execute.s*	0:01	0:11
5981	*cvtl.d.f Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_980_981.s,st_1980_execute.s*	0:40	0:50
985	*cvts.w Vj,Vk Native Mode*	*cpu4241.rnn*	*st_985_986.s,st_985_execute.s*	0:01	0:11
986	*cvts.w Vj,Vk Native Mode*	*cpu4241.rnn*	*st_985_986.s,st_985_execute.s*	0:06	0:16
1985	*cvts.w Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_985_986.s,st_1985_execute.s*	0:01	0:11
1986	*cvts.w Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_985_986.s,st_1985_execute.s*	0:06	0:16
2985	*cvts.w.t Vj,Vk Native Mode*	*cpu4241.rnn*	*st_985_986.s,st_985_execute.s*	0:01	0:11
2986	*cvts.w.t Vj,Vk Native Mode*	*cpu4241.rnn*	*st_985_986.s,st_985_execute.s*	0:42	0:52
3985	*cvts.w.t Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_985_986.s,st_1985_execute.s*	0:02	0:12
3986	*cvts.w.t Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_985_986.s,st_1985_execute.s*	0:46	0:56

Table cpu4241-3, Class 2 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
4985	*cvts.w.f Vj,Vk Native Mode*	*cpu4241.rnn*	*st_985_986.s,st_985_execute.s*	0:02	0:12
4986	*cvts.w.f Vj,Vk Native Mode*	*cpu4241.rnn*	*st_985_986.s,st_985_execute.s*	0:42	0:52
5985	*cvts.w.f Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_985_986.s,st_1985_execute.s*	0:02	0:12
5986	*cvts.w.f Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_985_986.s,st_1985_execute.s*	0:46	0:56
987	*cvts.l Vj,Vk Native Mode*	*cpu4241.rnn*	*st_987_988.s,st_987_execute.s*	0:01	0:11
988	*cvts.l Vj,Vk Native Mode*	*cpu4241.rnn*	*st_987_988.s,st_987_execute.s*	0:06	0:16
1987	*cvts.l Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_987_988.s,st_1987_execute.s*	0:01	0:11
1988	*cvts.l Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_987_988.s,st_1987_execute.s*	0:06	0:16
2987	*cvts.l.t Vj,Vk Native Mode*	*cpu4241.rnn*	*st_987_988.s,st_987_execute.s*	0:01	0:11
2988	*cvts.l.t Vj,Vk Native Mode*	*cpu4241.rnn*	*st_987_988.s,st_987_execute.s*	0:42	0:52
3987	*cvts.l.t Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_987_988.s,st_1987_execute.s*	0:01	0:11
3988	*cvts.l.t Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_987_988.s,st_1987_execute.s*	0:47	0:57
4987	*cvts.l.f Vj,Vk Native Mode*	*cpu4241.rnn*	*st_987_988.s,st_987_execute.s*	0:01	0:11
4988	*cvts.l.f Vj,Vk Native Mode*	*cpu4241.rnn*	*st_987_988.s,st_987_execute.s*	0:42	0:52
5987	*cvts.l.f Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_987_988.s,st_1987_execute.s*	0:02	0:12
5988	*cvts.l.f Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_987_988.s,st_1987_execute.s*	0:46	0:56
990	*cvts.d Vj,Vk Native Mode*	*cpu4241.rnn*	*st_990_991.s,st_990_execute.s*	0:01	0:11
991	*cvts.d Vj,Vk Native Mode*	*cpu4241.rnn*	*st_990_991.s,st_990_execute.s*	0:05	0:15
1990	*cvts.d Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_990_991.s,st_1990_execute.s*	0:01	0:11
1991	*cvts.d Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_990_991.s,st_1990_execute.s*	0:06	0:16
2990	*cvts.d.t Vj,Vk Native Mode*	*cpu4241.rnn*	*st_990_991.s,st_990_execute.s*	0:02	0:12
2991	*cvts.d.t Vj,Vk Native Mode*	*cpu4241.rnn*	*st_990_991.s,st_990_execute.s*	0:41	0:51
3990	*cvts.d.t Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_990_991.s,st_1990_execute.s*	0:02	0:12
3991	*cvts.d.t Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_990_991.s,st_1990_execute.s*	0:47	0:57
4990	*cvts.d.f Vj,Vk Native Mode*	*cpu4241.rnn*	*st_990_991.s,st_990_execute.s*	0:02	0:12
4991	*cvts.d.f Vj,Vk Native Mode*	*cpu4241.rnn*	*st_990_991.s,st_990_execute.s*	0:41	0:51
5990	*cvts.d.f Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_990_991.s,st_1990_execute.s*	0:02	0:12
5991	*cvts.d.f Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_990_991.s,st_1990_execute.s*	0:47	0:57
992	*cvtd.w Vj,Vk Native Mode*	*cpu4241.rnn*	*st_992_993.s,st_992_execute.s*	0:01	0:11
993	*cvtd.w Vj,Vk Native Mode*	*cpu4241.rnn*	*st_992_993.s,st_992_execute.s*	0:06	0:16
1992	*cvtd.w Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_992_993.s,st_1992_execute.s*	0:01	0:11
1993	*cvtd.w Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_992_993.s,st_1992_execute.s*	0:06	0:16
2992	*cvtd.w.t Vj,Vk Native Mode*	*cpu4241.rnn*	*st_992_993.s,st_992_execute.s*	0:01	0:11
2993	*cvtd.w.t Vj,Vk Native Mode*	*cpu4241.rnn*	*st_992_993.s,st_992_execute.s*	0:42	0:52
3992	*cvtd.w.t Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_992_993.s,st_1992_execute.s*	0:02	0:12
3993	*cvtd.w.t Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_992_993.s,st_1992_execute.s*	0:46	0:56

Table cpu4241-3, Class 2 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
4992	*cvtd.w.f Vj,Vk Native Mode*	*cpu4241.rnn*	*st_992_998.s,st_992_execute.s*	0:01	0:11
4993	*cvtd.w.f Vj,Vk Native Mode*	*cpu4241.rnn*	*st_992_998.s,st_992_execute.s*	0:42	0:52
5992	*cvtd.w.f Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_992_998.s,st_1992_execute.s*	0:02	0:12
5993	*cvtd.w.f Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_992_998.s,st_1992_execute.s*	0:46	0:56
995	*cvtd.l Vj,Vk Native Mode*	*cpu4241.rnn*	*st_995_996.s,st_995_execute.s*	0:01	0:11
996	*cvtd.l Vj,Vk Native Mode*	*cpu4241.rnn*	*st_995_996.s,st_995_execute.s*	0:05	0:15
1995	*cvtd.l Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_995_996.s,st_1995_execute.s*	0:01	0:11
1996	*cvtd.l Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_995_996.s,st_1995_execute.s*	0:06	0:16
2995	*cvtd.l.t Vj,Vk Native Mode*	*cpu4241.rnn*	*st_995_996.s,st_995_execute.s*	0:02	0:12
2996	*cvtd.l.t Vj,Vk Native Mode*	*cpu4241.rnn*	*st_995_996.s,st_995_execute.s*	0:41	0:51
3995	*cvtd.l.t Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_995_996.s,st_1995_execute.s*	0:01	0:11
3996	*cvtd.l.t Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_995_996.s,st_1995_execute.s*	0:47	0:57
4995	*cvtd.l.f Vj,Vk Native Mode*	*cpu4241.rnn*	*st_995_996.s,st_995_execute.s*	0:02	0:12
4996	*cvtd.l.f Vj,Vk Native Mode*	*cpu4241.rnn*	*st_995_996.s,st_995_execute.s*	0:42	0:52
5995	*cvtd.l.f Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_995_996.s,st_1995_execute.s*	0:01	0:11
5996	*cvtd.l.f Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_995_996.s,st_1995_execute.s*	0:47	0:57
997	*cvtd.s Vj,Vk Native Mode*	*cpu4241.rnn*	*st_997_998.s,st_997_execute.s*	0:01	0:11
998	*cvtd.s Vj,Vk Native Mode*	*cpu4241.rnn*	*st_997_998.s,st_997_execute.s*	0:06	0:16
1997	*cvtd.s Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_997_998.s,st_1997_execute.s*	0:01	0:11
1998	*cvtd.s Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_997_998.s,st_1997_execute.s*	0:07	0:17
2997	*cvtd.s.t Vj,Vk Native Mode*	*cpu4241.rnn*	*st_997_998.s,st_997_execute.s*	0:01	0:11
2998	*cvtd.s.t Vj,Vk Native Mode*	*cpu4241.rnn*	*st_997_998.s,st_997_execute.s*	0:43	0:53
3997	*cvtd.s.t Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_997_998.s,st_1997_execute.s*	0:02	0:12
3998	*cvtd.s.t Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_997_998.s,st_1997_execute.s*	0:49	0:59
4997	*cvtd.s.f Vj,Vk Native Mode*	*cpu4241.rnn*	*st_997_998.s,st_997_execute.s*	0:01	0:11
4998	*cvtd.s.f Vj,Vk Native Mode*	*cpu4241.rnn*	*st_997_998.s,st_997_execute.s*	0:43	0:53
5997	*cvtd.s.f Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_997_998.s,st_1997_execute.s*	0:01	0:11
5998	*cvtd.s.f Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_997_998.s,st_1997_execute.s*	0:50	1:00

Class 3 Subtests

Class 3 subtests verify the operation of the multiply and divide pipe. Specifically, this class verifies the vector/vector and scalar/vector multiplications and divisions.

All subtests end with a halt command and store a halt code in address register A1. If the code stored in A1 is 0x100, then the subtest passed. Any other code in A1 indicates that the subtest failed. The following table lists the Class 3 subtests:

Table cpu4241–4, Class 3 Subtests

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
2630	*mul.b.t Vi,Sj,Vk*	*cpu4241.rnn*	*st_630_631.s,st_630_execute.s*	0:01	0:11
2631	*mul.b.t Vi,Sj,Vk*	*cpu4241.rnn*	*st_630_631.s,st_630_execute.s*	0:19	0:29
4630	*mul.b.f Vi,Sj,Vk*	*cpu4241.rnn*	*st_630_631.s,st_630_execute.s*	0:01	0:11
4631	*mul.b.f Vi,Sj,Vk*	*cpu4241.rnn*	*st_630_631.s,st_630_execute.s*	0:19	0:29
2635	*mul.h.t Vi,Sj,Vk*	*cpu4241.rnn*	*st_635_636.s,st_635_execute.s*	0:02	0:12
2636	*mul.h.t Vi,Sj,Vk*	*cpu4241.rnn*	*st_635_636.s,st_635_execute.s*	0:19	0:29
4635	*mul.h.f Vi,Sj,Vk*	*cpu4241.rnn*	*st_635_636.s,st_635_execute.s*	0:02	0:12
4636	*mul.h.f Vi,Sj,Vk*	*cpu4241.rnn*	*st_635_636.s,st_635_execute.s*	0:19	0:29
2640	*mul.w.t Vi,Sj,Vk*	*cpu4241.rnn*	*st_640_641.s,st_640_execute.s*	0:01	0:11
2641	*mul.w.t Vi,Sj,Vk*	*cpu4241.rnn*	*st_640_641.s,st_640_execute.s*	0:18	0:28
4640	*mul.w.f Vi,Sj,Vk*	*cpu4241.rnn*	*st_640_641.s,st_640_execute.s*	0:02	0:12
4641	*mul.w.f Vi,Sj,Vk*	*cpu4241.rnn*	*st_640_641.s,st_640_execute.s*	0:19	0:29
2645	*mul.l.t Vi,Sj,Vk*	*cpu4241.rnn*	*st_645_646.s,st_645_execute.s*	0:01	0:11
2646	*mul.l.t Vi,Sj,Vk*	*cpu4241.rnn*	*st_645_646.s,st_645_execute.s*	0:19	0:29
4645	*mul.l.f Vi,Sj,Vk*	*cpu4241.rnn*	*st_645_646.s,st_645_execute.s*	0:01	0:11
4646	*mul.l.f Vi,Sj,Vk*	*cpu4241.rnn*	*st_645_646.s,st_645_execute.s*	0:19	0:29
2650	*mul.s.t Vi,Sj,Vk Native Mode*	*cpu4241.rnn*	*st_650_651.s,st_650_execute.s*	0:01	0:11
2651	*mul.s.t Vi,Sj,Vk Native Mode*	*cpu4241.rnn*	*st_650_651.s,st_650_execute.s*	0:11	0:21
3650	*mul.s.t Vi,Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_650_651.s,st_1650_execute.s*	0:02	0:12
3651	*mul.s.t Vi,Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_650_651.s,st_1650_execute.s*	0:16	0:26
4650	*mul.s.f Vi,Sj,Vk Native Mode*	*cpu4241.rnn*	*st_650_651.s,st_650_execute.s*	0:01	0:11
4651	*mul.s.f Vi,Sj,Vk Native Mode*	*cpu4241.rnn*	*st_650_651.s,st_650_execute.s*	0:10	0:20
5650	*mul.s.f Vi,Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_650_651.s,st_1650_execute.s*	0:01	0:11
5651	*mul.s.f Vi,Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_650_651.s,st_1650_execute.s*	0:15	0:25
2655	*mul.d.t Vi,Sj,Vk Native Mode*	*cpu4241.rnn*	*st_655_656.s,st_655_execute.s*	0:01	0:11
2656	*mul.d.t Vi,Sj,Vk Native Mode*	*cpu4241.rnn*	*st_655_656.s,st_655_execute.s*	0:11	0:21

Table cpu4241-4, Class 3 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
3655	*mul.d.t Vi,Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_655_656.s,st_1655_execute.s*	0:02	0:12
3656	*mul.d.t Vi,Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_655_656.s,st_1655_execute.s*	0:16	0:26
4655	*mul.d.f Vi,Sj,Vk Native Mode*	*cpu4241.rnn*	*st_655_656.s,st_655_execute.s*	0:01	0:11
4656	*mul.d.f Vi,Sj,Vk Native Mode*	*cpu4241.rnn*	*st_655_656.s,st_655_execute.s*	0:11	0:21
5655	*mul.d.f Vi,Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_655_656.s,st_1655_execute.s*	0:01	0:11
5656	*mul.d.f Vi,Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_655_656.s,st_1655_execute.s*	0:16	0:26
2660	*div.b.t Vi,Sj,Vk*	*cpu4241.rnn*	*st_660_661.s,st_660_execute.s*	0:01	0:11
2661	*div.b.t Vi,Sj,Vk*	*cpu4241.rnn*	*st_660_661.s,st_660_execute.s*	0:10	0:20
4660	*div.b.f Vi,Sj,Vk*	*cpu4241.rnn*	*st_660_661.s,st_660_execute.s*	0:01	0:11
4661	*div.b.f Vi,Sj,Vk*	*cpu4241.rnn*	*st_660_661.s,st_660_execute.s*	0:11	0:21
2665	*div.h.t Vi,Sj,Vk*	*cpu4241.rnn*	*st_665_666.s,st_665_execute.s*	0:01	0:11
2666	*div.h.t Vi,Sj,Vk*	*cpu4241.rnn*	*st_665_666.s,st_665_execute.s*	0:12	0:22
4665	*div.h.f Vi,Sj,Vk*	*cpu4241.rnn*	*st_665_666.s,st_665_execute.s*	0:01	0:11
4666	*div.h.f Vi,Sj,Vk*	*cpu4241.rnn*	*st_665_666.s,st_665_execute.s*	0:12	0:22
2670	*div.w.t Vi,Sj,Vk*	*cpu4241.rnn*	*st_670_671.s,st_670_execute.s*	0:01	0:11
2671	*div.w.t Vi,Sj,Vk*	*cpu4241.rnn*	*st_670_671.s,st_670_execute.s*	0:14	0:24
4670	*div.w.f Vi,Sj,Vk*	*cpu4241.rnn*	*st_670_671.s,st_670_execute.s*	0:01	0:11
4671	*div.w.f Vi,Sj,Vk*	*cpu4241.rnn*	*st_670_671.s,st_670_execute.s*	0:15	0:25
2675	*div.l.t Vi,Sj,Vk*	*cpu4241.rnn*	*st_675_676.s,st_675_execute.s*	0:01	0:11
2676	*div.l.t Vi,Sj,Vk*	*cpu4241.rnn*	*st_675_676.s,st_675_execute.s*	0:20	0:30
4675	*div.l.f Vi,Sj,Vk*	*cpu4241.rnn*	*st_675_676.s,st_675_execute.s*	0:01	0:11
4676	*div.l.f Vi,Sj,Vk*	*cpu4241.rnn*	*st_675_676.s,st_675_execute.s*	0:20	0:30
2680	*div.s.t Vi,Sj,Vk Native Mode*	*cpu4241.rnn*	*st_680_681.s,st_680_execute.s*	0:01	0:11
2681	*div.s.t Vi,Sj,Vk Native Mode*	*cpu4241.rnn*	*st_680_681.s,st_680_execute.s*	0:13	0:23
3680	*div.s.t Vi,Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_680_681.s,st_1680_execute.s*	0:01	0:11
3681	*div.s.t Vi,Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_680_681.s,st_1680_execute.s*	0:21	0:31
4680	*div.s.f Vi,Sj,Vk Native Mode*	*cpu4241.rnn*	*st_680_681.s,st_680_execute.s*	0:01	0:11
4681	*div.s.f Vi,Sj,Vk Native Mode*	*cpu4241.rnn*	*st_680_681.s,st_680_execute.s*	0:14	0:24
5680	*div.s.f Vi,Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_680_681.s,st_1680_execute.s*	0:01	0:11
5681	*div.s.f Vi,Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_680_681.s,st_1680_execute.s*	0:22	0:32

Table cpu4241-4, Class 3 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
2685	*div.d.t Vi,Sj,Vk Native Mode*	*cpu4241.rnn*	*st_685_686.s,st_685_execute.s*	0:01	0:11
2686	*div.d.t Vi,Sj,Vk Native Mode*	*cpu4241.rnn*	*st_685_686.s,st_685_execute.s*	0:17	0:27
3685	*div.d.t Vi,Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_685_686.s,st_1685_execute.s*	0:02	0:12
3686	*div.d.t Vi,Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_685_686.s,st_1685_execute.s*	0:28	0:38
4685	*div.d.f Vi,Sj,Vk Native Mode*	*cpu4241.rnn*	*st_685_686.s,st_685_execute.s*	0:01	0:11
4686	*div.d.f Vi,Sj,Vk Native Mode*	*cpu4241.rnn*	*st_685_686.s,st_685_execute.s*	0:17	0:27
5685	*div.d.f Vi,Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_685_686.s,st_1685_execute.s*	0:02	0:12
5686	*div.d.f Vi,Sj,Vk IEEE Mode*	*cpu4241.rnn*	*st_685_686.s,st_1685_execute.s*	0:28	0:38
690	*div.s Si,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_690_691.s,st_690_execute.s*	0:01	0:11
691	*div.s Si,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_690_691.s,st_690_execute.s*	0:03	0:13
1690	*div.s Si,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_690_691.s,st_1690_execute.s*	0:01	0:11
1691	*div.s Si,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_690_691.s,st_1690_execute.s*	0:03	0:13
2690	*div.s.t Si,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_690_691.s,st_690_execute.s*	0:01	0:11
2691	*div.s.t Si,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_690_691.s,st_690_execute.s*	0:14	0:24
3690	*div.s.t Si,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_690_691.s,st_1690_execute.s*	0:01	0:11
3691	*div.s.t Si,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_690_691.s,st_1690_execute.s*	0:22	0:32
4690	*div.s.f Si,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_690_691.s,st_690_execute.s*	0:01	0:11
4691	*div.s.f Si,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_690_691.s,st_690_execute.s*	0:13	0:23
5690	*div.s.f Si,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_690_691.s,st_1690_execute.s*	0:01	0:11
5691	*div.s.f Si,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_690_691.s,st_1690_execute.s*	0:21	0:31
695	*div.d Si,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_695_696.s,st_695_execute.s*	0:01	0:11
696	*div.d Si,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_695_696.s,st_695_execute.s*	0:03	0:13
1695	*div.d Si,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_695_696.s,st_1695_execute.s*	0:01	0:11
1696	*div.d Si,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_695_696.s,st_1695_execute.s*	0:05	0:15
2695	*div.d.t Si,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_695_696.s,st_695_execute.s*	0:01	0:11
2696	*div.d.t Si,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_695_696.s,st_695_execute.s*	0:18	0:28
3695	*div.d.t Si,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_695_696.s,st_1695_execute.s*	0:01	0:11
3696	*div.d.t Si,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_695_696.s,st_1695_execute.s*	0:28	0:38
4695	*div.d.f Si,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_695_696.s,st_695_execute.s*	0:01	0:11
4696	*div.d.f Si,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_695_696.s,st_695_execute.s*	0:18	0:28

Table cpu4241-4, Class 3 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
5695	*div.d.f Si,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_695_696.s,st_1695_execute.s*	0:01	0:11
5696	*div.d.f Si,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_695_696.s,st_1695_execute.s*	0:28	0:38
2700	*mul.b.t Vi,Vj,Vk*	*cpu4241.rnn*	*st_700_701.s,st_700_execute.s*	0:01	0:11
2701	*mul.b.t Vi,Vj,Vk*	*cpu4241.rnn*	*st_700_701.s,st_700_execute.s*	0:25	0:35
4700	*mul.b.f Vi,Vj,Vk*	*cpu4241.rnn*	*st_700_701.s,st_700_execute.s*	0:02	0:12
4701	*mul.b.f Vi,Vj,Vk*	*cpu4241.rnn*	*st_700_701.s,st_700_execute.s*	0:25	0:35
2705	*mul.h.t Vi,Vj,Vk*	*cpu4241.rnn*	*st_705_706.s,st_705_execute.s*	0:02	0:12
2706	*mul.h.t Vi,Vj,Vk*	*cpu4241.rnn*	*st_705_706.s,st_705_execute.s*	0:25	0:35
4705	*mul.h.f Vi,Vj,Vk*	*cpu4241.rnn*	*st_705_706.s,st_705_execute.s*	0:02	0:12
4706	*mul.h.f Vi,Vj,Vk*	*cpu4241.rnn*	*st_705_706.s,st_705_execute.s*	0:26	0:36
2710	*mul.w.t Vi,Vj,Vk*	*cpu4241.rnn*	*st_710_711.s,st_710_execute.s*	0:01	0:11
2711	*mul.w.t Vi,Vj,Vk*	*cpu4241.rnn*	*st_710_711.s,st_710_execute.s*	0:26	0:36
4710	*mul.w.f Vi,Vj,Vk*	*cpu4241.rnn*	*st_710_711.s,st_710_execute.s*	0:02	0:12
4711	*mul.w.f Vi,Vj,Vk*	*cpu4241.rnn*	*st_710_711.s,st_710_execute.s*	0:26	0:36
2715	*mul.l.t Vi,Vj,Vk*	*cpu4241.rnn*	*st_715_716.s,st_715_execute.s*	0:02	0:12
2716	*mul.l.t Vi,Vj,Vk*	*cpu4241.rnn*	*st_715_716.s,st_715_execute.s*	0:26	0:36
4715	*mul.l.f Vi,Vj,Vk*	*cpu4241.rnn*	*st_715_716.s,st_715_execute.s*	0:02	0:12
4716	*mul.l.f Vi,Vj,Vk*	*cpu4241.rnn*	*st_715_716.s,st_715_execute.s*	0:25	0:35
2720	*mul.s.t Vi,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_720_721.s,st_720_execute.s*	0:01	0:11
2721	*mul.s.t Vi,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_720_721.s,st_720_execute.s*	0:13	0:23
3720	*mul.s.t Vi,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_720_721.s,st_1720_execute.s*	0:02	0:12
3721	*mul.s.t Vi,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_720_721.s,st_1720_execute.s*	0:30	0:40
4720	*mul.s.f Vi,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_720_721.s,st_720_execute.s*	0:01	0:11
4721	*mul.s.f Vi,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_720_721.s,st_720_execute.s*	0:13	0:23
5720	*mul.s.f Vi,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_720_721.s,st_1720_execute.s*	0:02	0:12
5721	*mul.s.f Vi,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_720_721.s,st_1720_execute.s*	0:31	0:41
2725	*mul.d.t Vi,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_725_726.s,st_725_execute.s*	0:01	0:11
2726	*mul.d.t Vi,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_725_726.s,st_725_execute.s*	0:14	0:24
3725	*mul.d.t Vi,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_725_726.s,st_1725_execute.s*	0:01	0:11
3726	*mul.d.t Vi,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_725_726.s,st_1725_execute.s*	0:31	0:41

Table cpu4241-4, Class 3 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
4725	*mul.d.f Vi,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_725_726.s,st_725_execute.s*	0:01	0:11
4726	*mul.d.f Vi,Vj,Vk Native Mode*	*cpu4241.rnn*	*st_725_726.s,st_725_execute.s*	0:13	0:23
5725	*mul.d.f Vi,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_725_726.s,st_1725_execute.s*	0:02	0:12
5726	*mul.d.f Vi,Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_725_726.s,st_1725_execute.s*	0:30	0:40
730	*sqrt.s Vj,Vk Native Mode*	*cpu4241.rnn*	*st_730_731.s,st_730_execute.s*	0:01	0:11
731	*sqrt.s Vj,Vk Native Mode*	*cpu4241.rnn*	*st_730_731.s,st_730_execute.s*	0:04	0:14
1730	*sqrt.s Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_730_731.s,st_1730_execute.s*	0:01	0:11
1731	*sqrt.s Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_730_731.s,st_1730_execute.s*	0:05	0:15
2730	*sqrt.s.t Vj,Vk Native Mode*	*cpu4241.rnn*	*st_730_731.s,st_730_execute.s*	0:01	0:11
2731	*sqrt.s.t Vj,Vk Native Mode*	*cpu4241.rnn*	*st_730_731.s,st_730_execute.s*	0:26	0:36
3730	*sqrt.s.t Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_730_731.s,st_1730_execute.s*	0:02	0:12
3731	*sqrt.s.t Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_730_731.s,st_1730_execute.s*	0:34	0:44
4730	*sqrt.s.f Vj,Vk Native Mode*	*cpu4241.rnn*	*st_730_731.s,st_730_execute.s*	0:02	0:12
4731	*sqrt.s.f Vj,Vk Native Mode*	*cpu4241.rnn*	*st_730_731.s,st_730_execute.s*	0:27	0:37
5730	*sqrt.s.f Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_730_731.s,st_1730_execute.s*	0:01	0:11
5731	*sqrt.s.f Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_730_731.s,st_1730_execute.s*	0:35	0:45
735	*sqrt.d Vj,Vk Native Mode*	*cpu4241.rnn*	*st_735_736.s,st_735_execute.s*	0:01	0:11
736	*sqrt.d Vj,Vk Native Mode*	*cpu4241.rnn*	*st_735_736.s,st_735_execute.s*	0:06	0:16
1735	*sqrt.d Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_735_736.s,st_1735_execute.s*	0:01	0:11
1736	*sqrt.d Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_735_736.s,st_1735_execute.s*	0:06	0:16
2735	*sqrt.d.t Vj,Vk Native Mode*	*cpu4241.rnn*	*st_735_736.s,st_735_execute.s*	0:02	0:12
2736	*sqrt.d.t Vj,Vk Native Mode*	*cpu4241.rnn*	*st_735_736.s,st_735_execute.s*	0:38	0:48
3735	*sqrt.d.t Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_735_736.s,st_1735_execute.s*	0:02	0:12
3736	*sqrt.d.t Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_735_736.s,st_1735_execute.s*	0:48	0:58
4735	*sqrt.d.f Vj,Vk Native Mode*	*cpu4241.rnn*	*st_735_736.s,st_735_execute.s*	0:01	0:11
4736	*sqrt.d.f Vj,Vk Native Mode*	*cpu4241.rnn*	*st_735_736.s,st_735_execute.s*	0:38	0:48
5735	*sqrt.d.f Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_735_736.s,st_1735_execute.s*	0:02	0:12
5736	*sqrt.d.f Vj,Vk IEEE Mode*	*cpu4241.rnn*	*st_735_736.s,st_1735_execute.s*	0:49	0:59
2740	*div.b.t Vi,Vj,Vk*	*cpu4241.rnn*	*st_740_741.s,st_740_execute.s*	0:01	0:11
2741	*div.b.t Vi,Vj,Vk*	*cpu4241.rnn*	*st_740_741.s,st_740_execute.s*	0:12	0:22

Table cpu4241-4, Class 3 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
4740	*div.b.f Vi, Vj, Vk*	*cpu4241.rnn*	*st_740_741.s,st_740_execute.s*	0:01	0:11
4741	*div.b.f Vi, Vj, Vk*	*cpu4241.rnn*	*st_740_741.s,st_740_execute.s*	0:11	0:21
2745	*div.h.t Vi, Vj, Vk*	*cpu4241.rnn*	*st_745_746.s,st_745_execute.s*	0:01	0:11
2746	*div.h.t Vi, Vj, Vk*	*cpu4241.rnn*	*st_745_746.s,st_745_execute.s*	0:12	0:22
4745	*div.h.f Vi, Vj, Vk*	*cpu4241.rnn*	*st_745_746.s,st_745_execute.s*	0:01	0:11
4746	*div.h.f Vi, Vj, Vk*	*cpu4241.rnn*	*st_745_746.s,st_745_execute.s*	0:12	0:22
2750	*div.w.t Vi, Vj, Vk*	*cpu4241.rnn*	*st_750_751.s,st_750_execute.s*	0:01	0:11
2751	*div.w.t Vi, Vj, Vk*	*cpu4241.rnn*	*st_750_751.s,st_750_execute.s*	0:14	0:24
4750	*div.w.f Vi, Vj, Vk*	*cpu4241.rnn*	*st_750_751.s,st_750_execute.s*	0:01	0:11
4751	*div.w.f Vi, Vj, Vk*	*cpu4241.rnn*	*st_750_751.s,st_750_execute.s*	0:14	0:24
2755	*div.l.t Vi, Vj, Vk*	*cpu4241.rnn*	*st_755_756.s,st_755_execute.s*	0:01	0:11
2756	*div.l.t Vi, Vj, Vk*	*cpu4241.rnn*	*st_755_756.s,st_755_execute.s*	0:18	0:28
4755	*div.l.f Vi, Vj, Vk*	*cpu4241.rnn*	*st_755_756.s,st_755_execute.s*	0:01	0:11
4756	*div.l.f Vi, Vj, Vk*	*cpu4241.rnn*	*st_755_756.s,st_755_execute.s*	0:18	0:28
2760	*div.s.t Vi, Vj, Vk Native Mode*	*cpu4241.rnn*	*st_760_761.s,st_760_execute.s*	0:02	0:12
2761	*div.s.t Vi, Vj, Vk Native Mode*	*cpu4241.rnn*	*st_760_761.s,st_760_execute.s*	0:15	0:25
3760	*div.s.t Vi, Vj, Vk IEEE Mode*	*cpu4241.rnn*	*st_760_761.s,st_1760_execute.s*	0:01	0:11
3761	*div.s.t Vi, Vj, Vk IEEE Mode*	*cpu4241.rnn*	*st_760_761.s,st_1760_execute.s*	0:41	0:51
4760	*div.s.f Vi, Vj, Vk Native Mode*	*cpu4241.rnn*	*st_760_761.s,st_760_execute.s*	0:02	0:12
4761	*div.s.f Vi, Vj, Vk Native Mode*	*cpu4241.rnn*	*st_760_761.s,st_760_execute.s*	0:15	0:25
5760	*div.s.f Vi, Vj, Vk IEEE Mode*	*cpu4241.rnn*	*st_760_761.s,st_1760_execute.s*	0:01	0:11
5761	*div.s.f Vi, Vj, Vk IEEE Mode*	*cpu4241.rnn*	*st_760_761.s,st_1760_execute.s*	0:41	0:51
2765	*div.d.t Vi, Vj, Vk Native Mode*	*cpu4241.rnn*	*st_765_766.s,st_765_execute.s*	0:02	0:12
2766	*div.d.t Vi, Vj, Vk Native Mode*	*cpu4241.rnn*	*st_765_766.s,st_765_execute.s*	0:20	0:30
3765	*div.d.t Vi, Vj, Vk IEEE Mode*	*cpu4241.rnn*	*st_765_766.s,st_1765_execute.s*	0:02	0:12
3766	*div.d.t Vi, Vj, Vk IEEE Mode*	*cpu4241.rnn*	*st_765_766.s,st_1765_execute.s*	0:53	1:03
4765	*div.d.f Vi, Vj, Vk Native Mode*	*cpu4241.rnn*	*st_765_766.s,st_765_execute.s*	0:02	0:12
4766	*div.d.f Vi, Vj, Vk Native Mode*	*cpu4241.rnn*	*st_765_766.s,st_765_execute.s*	0:20	0:30

Table cpu4241-4, Class 3 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
5765	*div.d.f Vi, Vj, Vk IEEE Mode*	*cpu4241.rnn*	*st_765_766.s,st_1765_execute.s*	0:02	0:12
5766	*div.d.f Vi, Vj, Vk IEEE Mode*	*cpu4241.rnn*	*st_765_766.s,st_1765_execute.s*	0:53	1:03
2850	*prod.b.t Vk*	*cpu4241.rnn*	*st_850_851.s,st_850_execute.s*	0:02	0:12
2851	*prod.b.t Vk*	*cpu4241.rnn*	*st_850_851.s,st_850_execute.s*	0:43	0:53
4850	*prod.b.f Vk*	*cpu4241.rnn*	*st_850_851.s,st_850_execute.s*	0:01	0:11
4851	*prod.b.f Vk*	*cpu4241.rnn*	*st_850_851.s,st_850_execute.s*	0:42	0:52
2855	*prod.h.t Vk*	*cpu4241.rnn*	*st_855_856.s,st_855_execute.s*	0:01	0:11
2856	*prod.h.t Vk*	*cpu4241.rnn*	*st_855_856.s,st_855_execute.s*	0:42	0:52
4855	*prod.h.f Vk*	*cpu4241.rnn*	*st_855_856.s,st_855_execute.s*	0:01	0:11
4856	*prod.h.f Vk*	*cpu4241.rnn*	*st_855_856.s,st_855_execute.s*	0:42	0:52
2860	*prod.w.t Vk*	*cpu4241.rnn*	*st_860_861.s,st_860_execute.s*	0:01	0:11
2861	*prod.w.t Vk*	*cpu4241.rnn*	*st_860_861.s,st_860_execute.s*	0:42	0:52
4860	*prod.w.f Vk*	*cpu4241.rnn*	*st_860_861.s,st_860_execute.s*	0:02	0:12
4861	*prod.w.f Vk*	*cpu4241.rnn*	*st_860_861.s,st_860_execute.s*	0:43	0:53
2865	*prod.l.t Vk*	*cpu4241.rnn*	*st_865_866.s,st_865_execute.s*	0:01	0:11
2866	*prod.l.t Vk*	*cpu4241.rnn*	*st_865_866.s,st_865_execute.s*	0:45	0:55
4865	*prod.l.f Vk*	*cpu4241.rnn*	*st_865_866.s,st_865_execute.s*	0:01	0:11
4866	*prod.l.f Vk*	*cpu4241.rnn*	*st_865_866.s,st_865_execute.s*	0:45	0:55
2870	*prod.s.t Vk Native Mode*	*cpu4241.rnn*	*st_870_871.s,st_870_execute.s*	0:02	0:12
2871	*prod.s.t Vk Native Mode*	*cpu4241.rnn*	*st_870_871.s,st_870_execute.s*	1:08	1:18
3870	*prod.s.t Vk IEEE Mode*	*cpu4241.rnn*	*st_870_871.s,st_1870_execute.s*	0:02	0:12
3871	*prod.s.t Vk IEEE Mode*	*cpu4241.rnn*	*st_870_871.s,st_1870_execute.s*	1:14	1:24
4870	*prod.s.f Vk Native Mode*	*cpu4241.rnn*	*st_870_871.s,st_870_execute.s*	0:02	0:12
4871	*prod.s.f Vk Native Mode*	*cpu4241.rnn*	*st_870_871.s,st_870_execute.s*	1:08	1:18
5870	*prod.s.f Vk IEEE Mode*	*cpu4241.rnn*	*st_870_871.s,st_1870_execute.s*	0:02	0:12
5871	*prod.s.f Vk IEEE Mode*	*cpu4241.rnn*	*st_870_871.s,st_1870_execute.s*	1:14	1:24
2875	*prod.d.t Vk Native Mode*	*cpu4241.rnn*	*st_875_876.s,st_875_execute.s*	0:01	0:11
2876	*prod.d.t Vk Native Mode*	*cpu4241.rnn*	*st_875_876.s,st_875_execute.s*	1:14	1:24
3875	*prod.d.t Vk IEEE Mode*	*cpu4241.rnn*	*st_875_876.s,st_1875_execute.s*	0:01	0:11
3876	*prod.d.t Vk IEEE Mode*	*cpu4241.rnn*	*st_875_876.s,st_1875_execute.s*	1:19	1:29

Table cpu4241-4, Class 3 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
4875	*prod.d.f Vk Native Mode*	*cpu4241.rnn*	*st_875_876.s,st_875_ezecute.s*	0:01	0:11
4876	*prod.d.f Vk Native Mode*	*cpu4241.rnn*	*st_875_876.s,st_875_ezecute.s*	1:15	1:25
5875	*prod.d.f Vk IEEE Mode*	*cpu4241.rnn*	*st_875_876.s,st_1875_ezecute.s*	0:02	0:12
5876	*prod.d.f Vk IEEE Mode*	*cpu4241.rnn*	*st_875_876.s,st_1875_ezecute.s*	1:20	1:30

Class 4 Subtests

Class 4 subtests verify the operation of loading and storing vector registers. Specifically, this class verifies loading and storing the vector using direct addressing and vector of indices, and storing of vectors and scalar registers using the extended operations.

All subtests end with a halt command and store a halt code in address register A1. If the code stored in A1 is 0x100, then the subtest passed. Any other code in A1 indicates that the subtest failed. The following table lists the Class 4 subtests:

Table cpu4241–5, Class 4 Subtests

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
2200	*ld.b.t <effa>,Vk*	*cpu4241.rnn*	*st_200_201.s,st_200_execute.s*	0:05	0:15
2201	*ld.b.t <effa>,Vk*	*cpu4241.rnn*	*st_200_201.s,st_200_execute.s*	0:04	0:14
4200	*ld.b.f <effa>,Vk*	*cpu4241.rnn*	*st_200_201.s,st_200_execute.s*	0:01	0:11
4201	*ld.b.f <effa>,Vk*	*cpu4241.rnn*	*st_200_201.s,st_200_execute.s*	0:05	0:15
2205	*ld.h.t <effa>,Vk*	*cpu4241.rnn*	*st_205_206.s,st_205_execute.s*	0:01	0:11
2206	*ld.h.t <effa>,Vk*	*cpu4241.rnn*	*st_205_206.s,st_205_execute.s*	0:05	0:15
4205	*ld.h.f <effa>,Vk*	*cpu4241.rnn*	*st_205_206.s,st_205_execute.s*	0:01	0:11
4206	*ld.h.f<effa>,Vk*	*cpu4241.rnn*	*st_205_206.s,st_205_execute.s*	0:06	0:16
2210	*ld.w.t <effa>,Vk*	*cpu4241.rnn*	*st_210_211.s,st_210_execute.s*	0:01	0:11
2211	*ld.w.t <effa>,Vk*	*cpu4241.rnn*	*st_210_211.s,st_210_execute.s*	0:05	0:15
4210	*ld.w.f <effa>,Vk*	*cpu4241.rnn*	*st_210_211.s,st_210_execute.s*	0:01	0:11
4211	*ld.w.f<effa>,Vk*	*cpu4241.rnn*	*st_210_211.s,st_210_execute.s*	0:06	0:16
2215	*ld.l.t <effa>,Vk*	*cpu4241.rnn*	*st_215_216.s,st_215_execute.s*	0:01	0:11
2216	*ld.l.t <effa>,Vk*	*cpu4241.rnn*	*st_215_216.s,st_215_execute.s*	0:08	0:18
4215	*ld.l.f <effa>,Vk*	*cpu4241.rnn*	*st_215_216.s,st_215_execute.s*	0:01	0:11
4216	*ld.l.f<effa>,Vk*	*cpu4241.rnn*	*st_215_216.s,st_215_execute.s*	0:08	0:18
2230	*st.b.t Vk,<effa>*	*cpu4241.rnn*	*st_230_231.s,st_230_execute.s*	0:01	0:11
2231	*st.b.t Vk,<effa>*	*cpu4241.rnn*	*st_230_231.s,st_230_execute.s*	0:28	0:38
4230	*st.b.f Vk,<effa>*	*cpu4241.rnn*	*st_230_231.s,st_230_execute.s*	0:01	0:11
4231	*st.b.f Vk,<effa>*	*cpu4241.rnn*	*st_230_231.s,st_230_execute.s*	0:28	0:38
2235	*st.h.t Vk,<effa>*	*cpu4241.rnn*	*st_235_236.s,st_235_execute.s*	0:01	0:11
2236	*st.h.t Vk,<effa>*	*cpu4241.rnn*	*st_235_236.s,st_235_execute.s*	0:32	0:42

Table cpu4241-5, Class 4 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
4235	*st.h.f Vk,<effa>*	*cpu4241.rnn*	*st_235_236.s,st_235_execute.s*	0:02	0:12
4236	*st.h.f Vk,<effa>*	*cpu4241.rnn*	*st_235_236.s,st_235_execute.s*	0:32	0:42
2240	*st.w.t Vk,<effa>*	*cpu4241.rnn*	*st_240_241.s,st_240_execute.s*	0:02	0:12
2241	*st.w.t Vk,<effa>*	*cpu4241.rnn*	*st_240_241.s,st_240_execute.s*	0:42	0:52
4240	*st.w.f Vk,<effa>*	*cpu4241.rnn*	*st_240_241.s,st_240_execute.s*	0:02	0:12
4241	*st.w.f Vk,<effa>*	*cpu4241.rnn*	*st_240_241.s,st_240_execute.s*	0:42	0:52
2245	*st.l.t Vk,<effa>*	*cpu4241.rnn*	*st_245_246.s,st_245_execute.s*	0:02	0:12
2246	*st.l.t Vk,<effa>*	*cpu4241.rnn*	*st_245_246.s,st_245_execute.s*	0:58	1:08
4245	*st.l.f Vk,<effa>*	*cpu4241.rnn*	*st_245_246.s,st_245_execute.s*	0:01	0:11
4246	*st.l.f Vk,<effa>*	*cpu4241.rnn*	*st_245_246.s,st_245_execute.s*	0:58	1:08
2550	*ldvi.b.t Vj,Vk*	*cpu4241.rnn*	*st_550_551.s,st_550_execute.s*	0:02	0:12
2551	*ldvi.b.t Vj,Vk*	*cpu4241.rnn*	*st_550_551.s,st_550_execute.s*	0:05	0:15
4550	*ldvi.b.f Vj,Vk*	*cpu4241.rnn*	*st_550_551.s,st_550_execute.s*	0:02	0:12
4551	*ldvi.b.f Vj,Vk*	*cpu4241.rnn*	*st_550_551.s,st_550_execute.s*	0:05	0:15
2555	*ldvi.h.t Vj,Vk*	*cpu4241.rnn*	*st_555_556.s,st_555_execute.s*	0:01	0:11
2556	*ldvi.h.t Vj,Vk*	*cpu4241.rnn*	*st_555_556.s,st_555_execute.s*	0:06	0:16
4555	*ldvi.h.f Vj,Vk*	*cpu4241.rnn*	*st_555_556.s,st_555_execute.s*	0:02	0:12
4556	*ldvi.h.f Vj,Vk*	*cpu4241.rnn*	*st_555_556.s,st_555_execute.s*	0:05	0:15
2560	*ldvi.w.t Vj,Vk*	*cpu4241.rnn*	*st_560_561.s,st_560_execute.s*	0:02	0:12
2561	*ldvi.w.t Vj,Vk*	*cpu4241.rnn*	*st_560_561.s,st_560_execute.s*	0:05	0:15
4560	*ldvi.w.f Vj,Vk*	*cpu4241.rnn*	*st_560_561.s,st_560_execute.s*	0:01	0:11
4561	*ldvi.w.f Vj,Vk*	*cpu4241.rnn*	*st_560_561.s,st_560_execute.s*	0:06	0:16
2565	*ldvi.l.t Vj,Vk*	*cpu4241.rnn*	*st_565_566.s,st_565_execute.s*	0:02	0:12
2566	*ldvi.l.t Vj,Vk*	*cpu4241.rnn*	*st_565_566.s,st_565_execute.s*	0:05	0:15
4565	*ldvi.l.f Vj,Vk*	*cpu4241.rnn*	*st_565_566.s,st_565_execute.s*	0:02	0:12
4566	*ldvi.l.f Vj,Vk*	*cpu4241.rnn*	*st_565_566.s,st_565_execute.s*	0:06	0:16
2570	*stvi.b.t Vi,Vj*	*cpu4241.rnn*	*st_570_571.s,st_570_execute.s*	0:01	0:11
2571	*stvi.b.t Vi,Vj*	*cpu4241.rnn*	*st_570_571.s,st_570_execute.s*	0:19	0:29
4570	*stvi.b.f Vi,Vj*	*cpu4241.rnn*	*st_570_571.s,st_570_execute.s*	0:01	0:11
4571	*stvi.b.f Vi,Vj*	*cpu4241.rnn*	*st_570_571.s,st_570_execute.s*	0:19	0:29

Table cpu4241-5, Class 4 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
2575	*stvi.h.t Vi,Vj*	*cpu4241.rnn*	*st_575_576.s,st_575_execute.s*	0:01	0:11
2576	*stvi.h.t Vi,Vj*	*cpu4241.rnn*	*st_575_576.s,st_575_execute.s*	0:18	0:28
4575	*stvi.h.f Vi,Vj*	*cpu4241.rnn*	*st_575_576.s,st_575_execute.s*	0:01	0:11
4576	*stvi.h.f Vi,Vj*	*cpu4241.rnn*	*st_575_576.s,st_575_execute.s*	0:18	0:28
2580	*stvi.w.t Vi,Vj*	*cpu4241.rnn*	*st_580_581.s,st_580_execute.s*	0:01	0:11
2581	*stvi.w.t Vi,Vj*	*cpu4241.rnn*	*st_580_581.s,st_580_execute.s*	0:18	0:28
4580	*stvi.w.f Vi,Vj*	*cpu4241.rnn*	*st_580_581.s,st_580_execute.s*	0:01	0:11
4581	*stvi.w.f Vi,Vj*	*cpu4241.rnn*	*st_580_581.s,st_580_execute.s*	0:18	0:28
2585	*stvi.l.t Vi,Vj*	*cpu4241.rnn*	*st_585_586.s,st_585_execute.s*	0:01	0:11
2586	*stvi.l.t Vi,Vj*	*cpu4241.rnn*	*st_585_586.s,st_585_execute.s*	0:19	0:29
4585	*stvi.l.f Vi,Vj*	*cpu4241.rnn*	*st_585_586.s,st_585_execute.s*	0:02	0:12
4586	*stvi.l.f Vi,Vj*	*cpu4241.rnn*	*st_585_586.s,st_585_execute.s*	0:18	0:28
2590	*stvi.b.t Si,Vj*	*cpu4241.rnn*	*st_590_591.s,st_590_execute.s*	0:01	0:11
2591	*stvi.b.t Si,Vj*	*cpu4241.rnn*	*st_590_591.s,st_590_execute.s*	0:17	0:27
4590	*stvi.b.f Si,Vj*	*cpu4241.rnn*	*st_590_591.s,st_590_execute.s*	0:01	0:11
4591	*stvi.b.f Si,Vj*	*cpu4241.rnn*	*st_590_591.s,st_590_execute.s*	0:17	0:27
2595	*stvi.h.t Si,Vj*	*cpu4241.rnn*	*st_595_596.s,st_595_execute.s*	0:01	0:11
2596	*stvi.h.t Si,Vj*	*cpu4241.rnn*	*st_595_596.s,st_595_execute.s*	0:17	0:27
4595	*stvi.h.f Si,Vj*	*cpu4241.rnn*	*st_595_596.s,st_595_execute.s*	0:01	0:11
4596	*stvi.h.f Si,Vj*	*cpu4241.rnn*	*st_595_596.s,st_595_execute.s*	0:17	0:27
2600	*stvi.w.t Si,Vj*	*cpu4241.rnn*	*st_600_601.s,st_600_execute.s*	0:01	0:11
2601	*stvi.w.t Si,Vj*	*cpu4241.rnn*	*st_600_601.s,st_600_execute.s*	0:17	0:27
4600	*stvi.w.f Si,Vj*	*cpu4241.rnn*	*st_600_601.s,st_600_execute.s*	0:01	0:11
4601	*stvi.w.f Si,Vj*	*cpu4241.rnn*	*st_600_601.s,st_600_execute.s*	0:17	0:27
2605	*stvi.l.t Si,Vj*	*cpu4241.rnn*	*st_605_606.s,st_605_execute.s*	0:01	0:11
2606	*stvi.l.t Si,Vj*	*cpu4241.rnn*	*st_605_606.s,st_605_execute.s*	0:17	0:27
4605	*stvi.l.f Si,Vj*	*cpu4241.rnn*	*st_605_606.s,st_605_execute.s*	0:01	0:11
4606	*stvi.l.f Si,Vj*	*cpu4241.rnn*	*st_605_606.s,st_605_execute.s*	0:17	0:27
2610	*ste.b.t Sk,<effa>*	*cpu4241.rnn*	*st_610_611.s,st_610_execute.s*	0:02	0:12
2611	*ste.b.t Sk,<effa>*	*cpu4241.rnn*	*st_610_611.s,st_610_execute.s*	0:28	0:38

Table cpu4241-5, Class 4 Subtests (continued)

SUBTEST	TEST PERFORMED	OBJECT MODULE	SOURCE FILE	NOMINAL TIME (min/sec)	TIMEOUT LIMIT (min/sec)
4610	*ste.b.f Sk,<effa>*	*cpu4241.rnn*	*st_610_611.s,st_610_execute.s*	0:02	0:12
4611	*ste.b.f Sk,<effa>*	*cpu4241.rnn*	*st_610_611.s,st_610_execute.s*	0:28	0:38
2615	*ste.h.t Sk,<effa>*	*cpu4241.rnn*	*st_615_616.s,st_615_execute.s*	0:01	0:11
2616	*ste.h.t Sk,<effa>*	*cpu4241.rnn*	*st_615_616.s,st_615_execute.s*	0:31	0:41
4615	*ste.h.f Sk,<effa>*	*cpu4241.rnn*	*st_615_616.s,st_615_execute.s*	0:02	0:12
4616	*ste.h.f Sk,<effa>*	*cpu4241.rnn*	*st_615_616.s,st_615_execute.s*	0:31	0:41
2620	*ste.w.t Vk,<effa>*	*cpu4241.rnn*	*st_620_621.s,st_620_execute.s*	0:02	0:12
2621	*ste.w.t Vk,<effa>*	*cpu4241.rnn*	*st_620_621.s,st_620_execute.s*	0:40	0:50
4620	*ste.w.f Vk,<effa>*	*cpu4241.rnn*	*st_620_621.s,st_620_execute.s*	0:02	0:12
4621	*ste.w.f Vk,<effa>*	*cpu4241.rnn*	*st_620_621.s,st_620_execute.s*	0:39	0:49
2625	*ste.l.t Vk,<effa>*	*cpu4241.rnn*	*st_625_626.s,st_625_execute.s*	0:01	0:11
2626	*ste.l.t Vk,<effa>*	*cpu4241.rnn*	*st_625_626.s,st_625_execute.s*	0:55	1:05
4625	*ste.l.f Vk,<effa>*	*cpu4241.rnn*	*st_625_626.s,st_625_execute.s*	0:02	0:12
4626	*ste.l.f Vk,<effa>*	*cpu4241.rnn*	*st_625_626.s,st_625_execute.s*	0:55	1:05

Test Error Messages

For a common list of error messages that could result from running this test, refer to the appendix on CPU Error Messages.

SPU UNIX Error Messages

For a common list of possible error messages resulting from SPU UNIX, refer to the appendix on SPU UNIX Error Messages.

Appendix A
CPU-Based Error Messages

A.1 Overview

This appendix lists the possible error messages that could result from running the various CPU diagnostics. These common CPU error messages come from a library of CPU error messages. Each message is message is individually numbered and prefixed with LCPU.

A.2 Notational Conventions Within This Appendix

The following notational conventions used throughout this appendix:

D stands for a decimal digit

X stands for a hexadecimal digit

I stands for the errno number (refer to explanation below)

VN stands for vector (V) and its number (N), which is a number from 0 to 7)

ROUTINE_NAME is the name of the routine that detected the error

<TEXT> stands for a text string

CCC stands for an octal number

A.3 Error Message Format

Each error message is displayed in the following format:

LCPU_DDD: ROUTINE_NAME : <TEXT>

A portion of the error messages are system call failures. These errors are detected during a system call and are displayed in the following format:

LCPU_DDD: ROUTINE_NAME : <TEXT>
errno: I <TEXT>

A description of each errno possibility is contained in the SP2 Intro(2) man page contained in the *SPU UNIX Utilities Manual*. This appendix also contains a table listing the information contained in the SP2 Intro(2) manpage. The description of the errno message can be found by locating the errno description in the Intro(2) man page or in this appendix that has the same number as given by the I (number) in the error message.

A.4 CPU Test Error Messages

The following section lists each CPU error message in numerical order. Each message is followed by a disposition statement which contains, if possible, suggested causes and remedies to the error message.

`LCPU_001:asp_default:no ASP present in CPU D.`

Disposition: Make sure that the ASP is in the designated CPU and that its COP is readable.

`LCPU_002:spar_init:time out on initialize of CPU:D.`

Disposition: The scan engine has hung trying to initialize the cpu. Try running sysreset, then try running code again.

`LCPU_003:spar_init:illegal ending upc. Expected:0xbfd, actual:XXXXXXXX on CPU:D.`

Disposition: When the CPU was sent into a microcode loop to initialize parity, it did not complete. Could be that micro code is not loaded or broken ASP. Load and try again.

`LCPU_004:cpu_to_prr:Invalid CPU:D.`

Disposition: Internal consistency check.

`LCPU_007:dcu_default:no DCU present in CPU D.`

Disposition: Make sure that the DCU is in the designated CPU and that its COP is readable.

`LCPU_008:vp_init: Unsupported serial number DDDDD for vpc[D].`

Disposition: A VPC has an unsupported serial number. Could be a misprogrammed COP chip, or it could be that the latest diagnostics are required to run the board. Check the COP chip and check to ensure that the latest diagnostics are installed.

`LCPU_009:ipp_default:no IPP present in CPU D.`

Disposition: Make sure that the IPP is in the designated CPU and that its COP is readable.

`LCPU_010:ROUTINE_NAME: Window driver iocntl error. Unable to allocate window errno:D <TEXT>`

Disposition: System Call failure

LCPU_011:ROUTINE_NAME: Window driver iocntl error. Unable to deallocate window errno:D <TEXT>

Disposition: System Call failure

LCPU_012:ROUTINE_NAME: Can't open /dev/wndw for access to main memory errno:D <TEXT>

Disposition: System Call failure

LCPU_013:ROUTINE_NAME: Not enough sp2 memory.

Disposition: There is not enough SP2 memory to run program. Report as a bug.

LCPU_014:ROUTINE_NAME: Not enough physical memory.

Disposition: There is not enough main memory for the CPU code to be correctly loaded. Report as a bug.

LCPU_015:ROUTINE_NAME: Invalid SDR (XXXXXXXX) for logical address XXXXXXXX

Disposition: During the translation from a logical address to a physical address, an invalid SDR was encountered. The value of the SDR is printed along with the logical address.

LCPU_016:ROUTINE_NAME: Invalid PTE1 (XXXXXXXX) for logical address XXXXXXXX at XXXXXXXX

Disposition: During the translation from a logical address to a physical address, an invalid first level PTE (PTE1) was encountered. The value of this PTE is printed along with the logical address that was being translated. The physical address of the PTE is also printed.

LCPU_017:ROUTINE_NAME: Invalid PTE2 (XXXXXXXX) for logical address XXXXXXXX at XXXXXXXX

Disposition: During the translation from a logical address to a physical address, an invalid second level pte (PTE2) was encountered. The value of this pte is printed along with the logical address that was being translated. The physical address of the pte is also printed.

LCPU_018:ROUTINE_NAME: Invalid PTET (XXXXXXXX) for logical address XXXXXXXX, thread:D

Disposition: During the translation from a logical address to a physical address, an invalid thread level PTE (PTET) was encountered. The value of this PTE is printed along with the logical address and thread that was being translated. The physical address of the PTE is also printed.

`LCPU_019:ROUTINE_NAME: Invalid magic number CCC (octal) on file <filename>`

Disposition: The file has an invalid magic number. Possible file corruption.

`LCPU_020:set_us_uw: Illegal type: D specified`

Disposition: Internal consistency check.

`LCPU_021:sfu_default: no SFU present in CPU D`

Disposition: Make sure that the SFU is in the designated CPU and that its COP is readable.

`LCPU_022:vp_default:Invalid CPU:D.`

Disposition: Internal consistency check.

`LCPU_023:vregrw: VPC or VPD not present in CPU:D`

Disposition: Make sure that the VPC and VPD are in the designated CPU and that their COPs are readable.

`LCPU_024:read_vector: parity error reading VN on CPU:D`

Disposition: A parity error was detected. Try running **sysreset cpus -l2** to clear out the parity errors.

`LCPU_025:xlink:<symbolname> still undefined`

Disposition: symbol is undefined. Software bug.

`LCPU_026:add_symbol: colliding on symbol <symbolname>`

Disposition: If this occurs during the running of a test program, it is a software bug and should be reported. It is possible to get this error with mm or ddb by using commands improperly. Basically this error means that two object files have symbols with the same name and they are being loaded into the same symbol table.

`LCPU_027:make_symbols: read error N errno:D <TEXT>`

Disposition: A system call error occurred while reading from a file.

`LCPU_028:add_relocatable_references:error on ordinal symbol D`

Disposition: Error locating a relocatable symbol. Probable file corruption.

`LCPU_029:sys_init:failed call to mcm_default`

Disposition: Internal consistency check.

`LCPU_030:sys_init:failed call to cpx_default`

Disposition: Probable hardware error. Scan engine broken, CPX can not scan.

`LCPU_031:sys_init:failed call to pia_default`

Disposition: Probable hardware error, scan machine broke, PIA can not scan.

`LCPU_032:ROUTINE_NAME: error reading file <filename>`
`errno:D <TEXT>`

Disposition: System call error reading from file.

`LCPU_033:ROUTINE_NAME: Can't open <filename> or <filename>`

Disposition: File can not be found in file system. Corrupt filesystem, files have been deleted. Incorrect mounting of partitions.

`LCPU_034:ROUTINE_NAME: error on reading opthdr from file <filename>.`

Disposition: An invalid number of bytes was read from the opthdr portion of a SOFF format file. Corrupt file.

`LCPU_035:ROUTINE_NAME: no optional header on SOFF file <filename>.`

Disposition: The optional header is missing on a SOFF format file. Probable file corruption.

`LCPU_036:ROUTINE_NAME: error reading SOFF section header from file <filename>.`
`errno:D <TEXT>`

Disposition: A system call error occurred during the reading of a SOFF file section header.

`LCPU_037:ROUTINE_NAME: error reading SOFF section header from file <filename>.`

Disposition: An invalid number of bytes was read from a section header in a SOFF format file. Corrupt file.

LCPU_038:ROUTINE_NAME: error reading SOFF string table from file <filename>.
errno:D <TEXT>

Disposition: A system call error occurred during the reading of a SOFF file string table.

LCPU_039:ROUTINE_NAME: error reading SOFF string table from file <filename>.

Disposition: An invalid number of bytes was read from a string table in a SOFF format file. Corrupt file.

LCPU_040:ROUTINE_NAME: error reading data from file <filename>.
errno:D <TEXT>

Disposition: A system call error occurred during the reading of a file segment from disk.

LCPU_041:ROUTINE_NAME: error reading data from file <filename>.

Disposition: An invalid number of bytes was read from a file segment on the disk. Probable file corruption.

LCPU_042:ROUTINE_NAME: can't read relocation buffer from file <filename>.
errno:D <TEXT>

Disposition: A system call error occurred during the reading of the relocation information from a file on disk.

LCPU_043:ROUTINE_NAME: can't read relocation buffer from file <filename>.

Disposition: The actual number of bytes read from the relocation section of a file was not the same as that listed in the file header. Probable file corruption.

LCPU_044:reloc: Unresolved external reference in file <filename>.

Disposition: There is an unresolved external reference in a file, probable file corruption or software bug.

LCPU_045:ROUTINE_NAME: symbol relocated relative to illegal segment type XXXXXXXX in file <filename>.

Disposition: There is an illegal segment type listed in the relocation information for the file currently being relocated. Possible file corruption, software bug.

LCPU_046:ROUTINE_NAME: illegal relocation offset (-1) for type X in file <filename>.

Disposition: There is an illegal relocation offset listed in the relocation information for the file currently being relocated.

Possible file corruption, software bug.

`LCPU_047:ROUTINE_NAME: illegal relocation length D in file <filename>`

Disposition: There is an illegal relocation length listed in the relocation information for the file currently being relocated. Possible file corruption, software bug.

`LCPU_048:Non-window:Segmentation violation`

Disposition: A segmentation violation occurred during the execution of this program that did not pertain to the window hardware on the SP2. This could be caused by software problems or a bad SP2.

`LCPU_049:ROUTINE_NAME: can not open file <filename> or <filename>`

Disposition: The specified files could not be found. Caused by corrupt filesystem, filesystem incorrectly mounted, specified files have been deleted.

`LCPU_050:ROUTINE_NAME: error reading magic number from file <filename>`
`errno:D <TEXT>`

Disposition: A system call error occurred during the reading of the magic number from the file.

`LCPU_051:ROUTINE_NAME: error reading a.out header from file <filename>.`
`errno:D <TEXT>`

Disposition: A system call error occurred during the reading of an a.out header from the specified file.

`LCPU_052:ROUTINE_NAME: error reading SOFF filehdr from file <filename>.`
`errno:D <TEXT>`

Disposition: A system call error occurred during the reading of the SOFF filehdr from a SOFF format file.

`LCPU_053:ROUTINE_NAME: error reading SOFF filehdr from file <filename>.`

Disposition: Invalid number of bytes returned from read of SOFF filehdr.

`LCPU_054:ROUTINE_NAME: error reading SOFF opthdr from file <filename>.`
`errno:D <TEXT>`

Disposition: A system call error occurred during the reading of the SOFF opthdr from a SOFF format file.

LCPU_055:ROUTINE_NAME:error reading SOFF opthdr from file <filename>.

Disposition: Invalid number of bytes returned when reading the opthdr from disk.

LCPU_056:ROUTINE_NAME: no optional header on SOFF file <filename>.

Disposition: No optional header found in a SOFF format file. Probable file corruption.

LCPU_057:ROUTINE_NAME: error reading SOFF scnhdr from file <filename>.
errno:D <TEXT>

Disposition: A system call error occurred during the reading of a section header from a SOFF format file.

LCPU_058:ROUTINE_NAME:error reading SOFF scnhdr from file <filename>.

Disposition: Invalid number of bytes returned when reading the scnhdr from disk.

LCPU_059:ROUTINE_NAME: error reading SOFF string table from file <filename>.
errno:D <TEXT>

Disposition: A system call error occurred during the reading of the SOFF string table.

LCPU_060:ROUTINE_NAME:error reading SOFF string table from file <filename>.

Disposition: Invalid number of bytes returned when reading the string table from disk.

LCPU_061:ROUTINE_NAME: error, specifying physical address range that does not exist

LCPU_062:ROUTINE_NAME:invalid number of threads:D.

Disposition: A file was being loaded with too many threads specified for this architecture. Probable software bug.

LCPU_063:ROUTINE_NAME: mapping on top of previously mapped logical address at XXXXXXXX

Disposition: A file has been loaded with an overlapping range of logical addresses to some other file that has already been loaded. Probable software bug although user brain damage can cause this from mm or ddb.

LCPU_064:ROUTINE_NAME: address overlap

Disposition: Internal consistency check failure.

LCPU_065:ROUTINE_NAME: address range XXXXXXXX-XXXXXXXX in use

Disposition: Internal consistency check failure.

LCPU_066:ROUTINE_NAME:can not get bus error info

LCPU_067:ROUTINE_NAME:error during ROUTINE_NAME

Disposition: A bus error or segmentation violation has occurred. A description will be given detailing what was going on when this happened. Probable software error although several SP2 problems could cause this.

LCPU_068:ROUTINE_NAME:No memory present or uninitialized pcm

Disposition: The SP2 detects a population configuration map of all zeros. Run mminit.

LCPU_069:ROUTINE_NAME:Can not create <filename> errno:D
errno:D <TEXT>

Disposition: Failed system call.

LCPU_070:s: Can not reopen <filename> errno:D
errno:D <TEXT>

Disposition: Failed system call.

LCPU_071:ROUTINE_NAME: Can not read page map file /mnt/test/CPU/pagemap from disk.
errno:D <TEXT>

Disposition: Failed system call.

LCPU_072:ROUTINE_NAME: Can not read page map file /mnt/test/CPU/pagemap from disk.

Disposition: Invalid number of bytes read.

LCPU_073:ROUTINE_NAME: Can not read header from page map file /mnt/test/CPU/pagemap.
errno:D <TEXT>

Disposition: Failed system call.

LCPU_074:ROUTINE_NAME: Can not read header from page map file /mnt/test/CPU/pagemap.

Disposition: Invalid number of bytes read.

LCPU_075:ROUTINE_NAME:Can not get semaphore file /mnt/test/CPU/pmsem.
errno:D <TEXT>

Disposition: Failed system call.

```
LCPU_076:close_pm_file:error writing file <filename>
  errno:D <TEXT>
```

Disposition: Failed system call.

```
LCPU_082:ROUTINE_NAME:pid D not found in current memory mapping
```

Disposition: The mapping for a process id was requested and the Process ID (PID) does not exist. Probable software bug, although a user could cause this error from mm or ddb.

```
LCPU_083:ROUTINE_NAME:page map file /mnt/test/CPU/pagemap does not exist.
```

Disposition: Probable software error or file corruption.

```
LCPU_085:<TEXT> Invalid parameter(s) detected
```

Disposition: Software error. A call to a routine was made with invalid parameters.

```
LCPU_100:ROUTINE_NAME: error on reading opthdr from file <filename>
  errno:D <TEXT>
```

Disposition: A system call error occurred during the reading of the opthdr portion of a SOFF format file.

```
LCPU_086:ROUTINE_NAME:ACCESS_TYPE
68000_ERROR_CODE
OPTIONAL_ERROR_SUBCODE
```

Disposition: An error occurred attempting an EBUS operation.

Where:

ROUTINE_NAME is the name of the routine that detected the error.

ACCESS_TYPE is one of the codes listed in **Table A-1, ACCESS_TYPE Codes Definitions.** The table also provides a description of each code.

68000_ERROR_CODE is one of the error messages listed in **Table A-2, 68000_ERROR_CODES** The error's disposition is also given in the table.

OPTIONAL_ERROR_SUBCODE is one of the error messages in **Table A-3, OPTIONAL_ERROR_SUBCODES**. Each message's disposition is also given in the table.

Table A–1, ACCESS_TYPE Code Definitions

ACCESS_TYPE CODE	MEANING
mmrd	Main memory read operation
mmtas	Main memory Test and Set (TAS) operation
iord	I/O space read operation
mmwr	Main memory write operation
iowr	I/O space write operation
mmsync	Main memory msync operation
mmtac	Main memory Test and Clear (TAC) operation
mmscrub	Main memory scrub operation
mmlwwr_ip	Main memory long word (lw) write with inverted parity
mmlwwr	Main memory long word (lw) write

Table A–2, 68000_ERROR_CODES

68000_ERROR_CODE	DISPOSITION
68000 Dtack timeout error.	A data transfer acknowledge (Dtack) timeout occurred. The transfer did not occur within the allocated time. If this occurs an EBUS error subcode will also be listed.
68000 PMAP parity error.	A parity error was detected in the population mapper rams.
68000 memory parity error.	A memory parity error was detected during a transfer.
68000 memory protection error.	An attempt to access memory not accessible to a user program was attempted.

Table A–3, OPTIONAL_ERROR_SUBCODES

OPTIONAL_ERROR_SUBCODE	DISPOSITION
EBUS received parity error.	Bad parity was received on the EBUS.
Illegal EBUS address.	An attempt to perform an EBUS operation to address space that does not have the corresponding PMAP bit set.
Invalid EBUS command.	An attempt to perform an EBUS operation with an illegal combination of bits set in the EBUS map register corresponding to the address of the operation.

Listed below are some typical examples of this type of error message:

```
LCPU_086:routine_name:mmwr
68000 Dtack timeout error.
EBUS received parity error.
```

In this example a parity error was detected on the EBUS during a main memory write operation. The EBUS received parity error is the cause of the 68000 Dtack timeout error.

```
LCPU_086:routine_name:mmwr
68000 Dtack Timeout error
Illegal EBUS address.
```

In this example a 68000 dtack timeout occurred during a write to main memory. An attempt was made to address memory that was not mapped in the SP2 Population Configuration Map (PCM) register.

A.5 Errno Message Information

This section contains information on errno messages that are explained in the SP2 Intro(2) manpage. The errno messages are printed with some of the CPU diagnostics error messages listed within this appendix. Each errno can be located by looking for the errno with the matching number contained in the error message. For example, if an error message contained:

```
errno:10 ECHILD No children
```

look in this section for message 10 or in the SP2 Intro(2) manpage for an explanation of the message.

> **NOTE**
>
> The SP2 Intro(2) manpage is located in the *SPU UNIX Utilities Manual.*

Table A–4, errno Explanations

errno Number	errno Message	Explanation
0	Error 0	Unused
1	EPERM Not owner	Typically this error indicates an attempt to modify a file in some way forbidden except to its owner or super-user. It is also returned for attempts by ordinary users to do things allowed only to the super-user.
2	ENOENT No such file or directory	This error occurs when a file name is specified and the file should exist but doesn't, or when one of the directories in a path name does not exist.
3	ESRCH No such process	The process whose number was given to *signal* and *ptrace* does not exist, or is already dead.
4	EINTR Interrupted system call	An asynchronous signal (such as interrupt or quit), which the user has elected to catch, occurred during a system call. If execution is resumed after processing the signal, it will appear as if the interrupted system call returned this error condition.
5	EIO I/O error	Some physical I/O error occurred during a *read* or *write*. This error may in some cases occur on a call following the one to which it actually applies.
6	ENXIO No such device or address	I/O on a special file refers to a subdevice that does not exist, or beyond the limits of the device. It may also occur when, for example, a tape drive is not dialed in or no disk pack is loaded on a drive.
7	E2BIG Arg list too long	An argument list longer than 5120 bytes is presented to *exec*.
8	ENOEXEC Exec format error	A request is made to execute a file which, although it has the appropriate permissions, does not start with a valid magic number, see **b.out(5)**.
9	EBADF Bad file number	Either a file descriptor refers to no open file, or a read (resp. write) request is made to a file that is open only for writing (resp. reading).
10	ECHILD No children	*Wait* and the process has no living or unwaited-for children.
11	EAGAIN No more processes	In a *fork*, the system's process table is full or the user is not allowed to create any more processes.
12	ENOMEM Not enough core	During an *exec* or *break*, a program asks for more core than the system is able to supply. This is not a temporary condition; the maximum core size is a system parameter. The error may also occur if the arrangement of text, data, and stack segments requires too many segmentation registers.
13	EACCES Permission denied	An attempt was made to access a file in a way forbidden by the protection system.

Table E-1, errno Explanations (continued)

errno Number	errno Message	Explanation
14	EFAULT Bad address	The system encountered a hardware fault in attempting to access the arguments of a system call.
15	ENOTBLK Block device required	A plain file was mentioned where a block device was required, e.g. in *mount*.
16	EBUSY Mount device busy	An attempt to mount a device that was already mounted or an attempt was made to dismount a device on which there is an active file (open file, *current* directory, mounted-on file, active text segment).
17	EEXIST File exists	An existing file was mentioned in an inappropriate con- text, e.g. *link*.
18	EXDEV Cross-device link	A link to a file on another device was attempted.
19	ENODEV No such device	An attempt was made to apply an inappropriate system call to a device; e.g. read a write-only device.
20	ENOTDIR Not a directory	A non-directory was specified where a directory is required, for example in a path name or as an argument to *chdir*.
21	EISDIR Is a directory	An attempt to write on a directory.
22	EINVAL Invalid argument	Some invalid argument: dismounting a non-mounted dev- ice, mentioning an unknown signal in *signal*, reading or writing a file for which *seek* has generated a negative pointer. Also set by math functions, see **intro(3)**.
23	ENFILE File table overflow	The system's table of open files is full, and temporarily no more *opens* can be accepted.

Table E-1, errno Explanations (continued)

errno Number	errno Message	Explanation
24	EMFILE Too many open files	Customary configuration limit is 20 per process.
25	ENOTTY Not a typewriter	The file mentioned in *stty* or itty is not a terminal or one of the other devices to which these calls apply.
26	ETXTBSY Text file busy	An attempt to execute a pure-procedure program that is currently open for writing (or reading!). Also an attempt to open for writing a pure-procedure program that is being executed.
27	EFBIG File too large	The size of a file exceeded the maximum (about 1.0E9 bytes).
28	ENOSPC No space left on device	During a *write* to an ordinary file, there is no free space left on the device.
29	ESPIPE Illegal seek	An *lseek* was issued to a pipe. This error should also be issued for other non-seekable devices.
30	EROFS Read-only file system	An attempt to modify a file or directory was made on a device mounted read-only.
31	EMLINK Too many links	An attempt to make more than 32767 links to a file.
32	EPIPE Broken pipe	A write on a pipe for which there is no process to read the data. This condition normally generates a signal; the error is returned if the signal is ignored.
33	EDOM Math argument	The argument of a function in the math package (3M) is out of the domain of the function.
34	ERANGE Result too large	The value of a function in the math package (3M) is unrepresentable within machine precision.

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix B
SPU UNIX Error Messages

B.1 Overview

This appendix lists SPU UNIX panic messages with a definition, cause, and disposition for each.

B.2 SPU UNIX Messages

The following messages are displayed by SPU UNIX when it encounters a problem with which it cannot cope.

`SPU_UNIX(1):pq_init: Processor queue error (head='H', last='L', indx='I')`

Description of Error: During a call to the pq_init routine, an error was detected within the queue structure.
'H' is a variable(hex) that contains the index of the head of the queue.
'L' is a variable(hex) that contains the index of the tail of the queue.
'I' is a variable(hex) that contains the index of the SPU's messages.

Cause and Disposition: This error is caused by a queue that resides in main memory being inconsistent. This could be due to any one of the processors (CPU/CCUs/SPU) overwriting this queue. It also could be due to the queue not being correctly initialized.

`SPU_UNIX(2):pq_recv: Queue head index invalid. (head='H')`

Description of Error: The head index of the current queue is larger than that possible for the system.
'H' is a variable(hex) that represents the invalid head index.

Cause and Disposition: This error is caused by a queue that resides in main memory being inconsistent. This could be due to any one of the processors (CPU/CCUs/SPU) overwriting this queue. It also could be due to the queue not being correctly initialized.

`SPU_UNIX(3):pq_recv: Not in queue or Not owner. (flg='F', own='O')`

Description of Error:
'F' is a variable(hex) that represents the status flag of the message.
'O' is a variable(hex) that represents the owner processor ID for the message.

Cause and Disposition: This error is caused by a queue that resides in main memory being inconsistent. This could be due to any one of the processors (CPU/CCUs/SPU) overwriting this queue. It also could be due to the queue not being correctly initialized.

`SPU_UNIX(4):pq_send: Send to invalid processor id 'I'`

Description of Error: A process on the SPU is trying to send a request to a processor that has a processor number higher than the maximum allowed in this system.
'I' is the processor number of the target processor.

Cause and Disposition: This is probably due to errant user software running on the SPU.

`SPU_UNIX(5):pq_send: Message free or not owner (flg='X' own='O')`

Description of Error: A message is trying to be sent that does not belong to this processor or message is still shown as free.
'F' is a variable(hex) that represents the status flag associated with this message.
'O' is a variable(hex) that represents the message owner processor number.

Cause and Disposition: Unknown

`SPU_UNIX(6):pq_send: Dest proc 'P' last index invalid ('I')`

Description of Error: A message is transmitted that has an incorrect value associated with its last index. The value is greater than the maximum allowed in the system.
'P' is a variable(hex) that represents the destination processor id.
'I' is a variable(hex) that represents the invalid index.

Cause and Disposition: This error is caused by a queue that resides in main memory being inconsistent. This could be due to any one of the processors (CPU/CCUs/SPU) overwriting this queue. It also could be due to the queue not being correctly initialized.

`SPU_UNIX(7):pq_fm_alloc: message 'M' allocated, not free`

Description of Error: A message is allocated from the pool of free messages. When the status of this message is checked, it is not marked free.
'M' is a variable(hex) that represents the message index.

Cause and Disposition: This error is caused by a queue that resides in main memory being inconsistent. This could be due to any one of the processors (CPU/CCUs/SPU) overwriting this queue. It also could be due to the queue not being correctly initialized.

`SPU_UNIX(8):pq_fm_free: freeing free message 'I'`

Description of Error: A message is returned to the free list that is already marked free.
'I' is a variable(hex) that represents the message index.

Cause and Disposition: This is probably due to errant user software.

`SPU_UNIX(9):pq_fm_free: free message list locked!`

Description of Error: A message is returned to the free list and the free list is locked. After waiting for some time, the list remains locked and the attempt is aborted.

Cause and Disposition: This error is caused by a queue that resides in main memory being inconsistent. This could be due to any one of the processors (CPU/CCUs/SPU) overwriting this queue. It also could be the queue not being correctly initialized.

`SPU_UNIX(10):panic:blkdev`

Description of Error: A call to the UNIX routine *getblk* has been made with a major number greater than supposedly possible for SPU UNIX

Cause and Disposition: The cause of this error is unknown. Reboot SPU UNIX, if it occurs repeatedly, replace SPU.

`SPU_UNIX(11):panic:devtab`

Description of Error: When trying to reference the buffer associated with a given block I/O device, a null pointer is referenced.

Cause and Disposition: The cause of this error is unknown. The pointer could have been corrupted by errant system software or a SPU memory problem. Reboot SPU UNIX, if it occurs repeatedly, replace SPU.

`SPU_UNIX(12):panic:IO err in swap`

Description of Error: When trying to swap (in or out) of memory to disk, an I/O error occurred.

Cause and Disposition: This message should be preceded by some type of I/O errors emanating from the disk driver. The I/O error could be helpful in finding this problem. A bad swap partition could be the cause.

`SPU_UNIX(13):physio: address wraparound error`

Description of Error: When performing a transfer to/from an I/O device, the number of bytes requested is so large that the transfer address wraps back around to kernel memory space.

Cause and Disposition: This is probably caused by an errant user program trying to do raw I/O with an extremely large number of bytes requested in the data transfer. If this error occurs frequently, try to determine the data transfer and what processes are running at the time of the error.

`SPU_UNIX(14):physio: transfer address out-of-bounds`

ts='TS' ds='DS' nb='NB' limit='L'

Description of Error: This error is cause by a request to perform an I/O transfer that will access an invalid address.
'Ts' is a variable(hex) that is the end address of the text segment.
'Ds' is a variable(hex) that is the end address of the data segment.
'Nb' is a variable(hex) that is the base address for I/O transfers.
'L' is a variable(hex) that is the remaining number of bytes in the current I/O request.
All addresses are logical addresses.

Cause and Disposition: This is probably caused by an errant user program with either an invalid start transfer address or an invalid number of bytes requested. If this error occurs frequently, try to determine the processes running at this time.

`SPU_UNIX(15):panic:ccucmd: can't receive message`

Description of Error: Unknown

Cause and Disposition: Unknown

`SPU_UNIX(16):ct(D): illegal transfer/partition, block='B', cnt='C'`

Description of Error: A request was made to transfer data specifying an invalid partition or for more blocks than is available for a given partition.
'D' is a variable(dec) that is the minor number of the device.
'B' is the requested block number.
'C' is the number of bytes requested.

Cause and Disposition: This error is probably due to a user program error or a bad/corrupt entry in the directory */dev*.

`SPU_UNIX(17):ct: ctlr busy during ctstart (fdc='X'), timeout = 'T'`

Description of Error: During the initiation of a command to the cartridge tape controller, the controller was busy, most likely completing a previous command.
'X' is a variable(hex) that contains the status of the floppy tape control register.
'T' is a variable(dec) that contains the timeout value remaining when the controller de-asserted the busy signal.

Cause and Disposition: This error is not cause for immediate concern. Under most circumstances, recovery will be automatic.

`SPU_UNIX(18):ct: ctlr timed out waiting for idle (fdc='X'), resetting...done`

Description of Error: This error occurs when the controller will not go idle on its own. A 4- to 5-second timeout value is used to wait for the controller to become ready. If it does not become ready during this time, this message will occur and the controller will become reset.
'X' is a variable(hex) that contains the contents of the tape control register when the timeout occurred.

Cause and Disposition: The cause of this error is a cartridge tape controller that does not become ready. This error is probably indicative of a potential SPU hardware problem.

`SPU_UNIX(19):ct: ctlr busy after reset (fdc='X')`

Description of Error: This message will be associated with error number 18. After the timeout occurs (waiting for the controller to go idle), the controller is reset. If the controller is still not ready, this message is displayed.
'X' is a variable(hex) that contains the contents of the tape control register when the timeout occurred.

Cause and Disposition: This error message is most likely caused by a bad SPU.

`SPU_UNIX(20):ct(D): Timeout limit exceeded 'TAPEERROR3'`

Description of Error: A command issued to the cartridge tape controller did not complete within the expected amount of time.
'D' is a variable(dec) that is the minor number of the device.

Cause and Disposition: This error is probably caused by a hardware problem with the SPU or the cartridge tape drive unit.

`SPU_UNIX(21):ct('D'):Bad block table overflow 'TAPEERROR3'`

Description of Error: There are more bad blocks detected on the tape than there is room for in the bad block table.
'D' is a variable(dec) that is the minor number of the device.

Cause and Disposition: This is probably caused by a bad cartridge tape, by dirty heads, or by other problems with the tape drive. Try a new tape first, clean the heads second, replace the drive third.

`SPU_UNIX(22):ct(D):68000 Bus error (X) 'TAPEERROR3'`

Description of Error: This error is caused when a bus error occurs during a cartridge tape I/O operation.
'D' is a variable(dec) that is the minor number of the device.

Cause and Disposition: The cause of this error is unknown.

`SPU_UNIX(23):ctgetaddr: failed on memory copy 'D' 'D'.`

Description of Error: An abnormal condition was detected by either the copyin (D=1) or the copyout (D=2) functions.

Cause and Disposition: The cause of this error is unknown.

`SPU_UNIX(24):ct('D'): 'D'): Seek error detected on read address, trk='T'`

Description of Error: A seek to a specified track was not successful.
'T' is a variable(dec) that is the track where the seek error is detected.
'D' is a variable(dec) that is the minor number of the device.

Cause and Disposition: This could be either a hardware error with the SPU or the cartridge tape driver or could be caused by errant user code trying illegal transfers.

`SPU_UNIX(25):ct('D'): Cartridge tape write protected!`

Description of Error: A write to a write-protected cartridge tape was attempted.
'D' is a variable(dec) that is the minor number of the device.

Cause and Disposition: The tape needs to be removed, the write-protection turned off, then the operation tried again. This could also be caused by a faulty or misadjusted write-protect detection mechanism in the tape drive.

`SPU_UNIX(26):ct('D'): Errors on bad block table 'B0' [,'B1','B2'...]`

Description of Error: Errors were found when reading the specified bad blocks. A list of the blocks are given.
'D' is a variable(dec) that is the minor number of the device.

Cause and Disposition: This is probably caused by a bad cartridge tape (the bad block table has become corrupted). Reformat a new tape and try again. If this occurs often, clean the heads on the drive or replace it.

`SPU_UNIX(27):dk('D'): illegal transfer, block='B', cnt='C'`

Description of Error: An illegal transfer request was made. The error is detected when either an invalid partition is referenced or a block number that is too large to be on the specified device is requested.
'D' is a variable(dec) that is the minor number of the device.
'B' is a variable(dec) that is the block number.
'C' is a variable(dec) that is the number of bytes requested.

Cause and Disposition: This is probably caused by an errant user program. It could also be caused by a corrupt entry in the */dev* directory.

`SPU_UNIX(28):dk('D'): Transfer timeout error, retries = 'R'`

Description of Error: The requested command did not complete within the expected amount of time.
'D' is a variable(dec) that is the minor number of the device.
'R' is a variable(dec) that is the number of times the command was attempted before aborting.

Cause and Disposition: See expanded list and discussion of disk errors below.

`SPU_UNIX(29):dk 68000 bus error ('X'), retries = 'R'`

Description of Error: A bus error occurred during a disk transfer operation.
'D' is a variable(dec) that is the minor number of the device.
'X' is a variable(hex) that is the address where the bus error was detected.
'R' is a variable(dec) that is the number of times the command was attempted before aborting.

Cause and Disposition: The cause of this error is generally unknown. The expanded error report for disk errors (below) could give some answers.

`SPU_UNIX(30):('D'): unexpected sasierror = 'U', retries = 'R'`

Description of Error:
'D' is a variable(dec) that is the minor number of the device.
'U' is a variable(dec) that is the error number returned for this transfer.
'R' is a variable(dec) that is the number of times the command was attempted before aborting.

Cause and Disposition: Unknown

`SPU_UNIX(31):dk('D'): Error status='E' ('S'), retries='R'`

Description of Error: An error was returned from the disk via the SCSI bus during some operation. The cause of the error is described in the message.
'D' is a variable(dec) that is the minor number of the device.
'R' is a variable(dec) that is the number of times the command was attempted before aborting.
'E' is a variable(hex) that is the error status read from the SCSI bus.
'S' is a variable(string) that contains a textual description of the error code.

The possible values of strings and error codes are:

0x00:No Error Status
0x01:No Index/Sector Signal
0x02:No Seek Complete
0x03:Write Fault
0x04:Drive not ready
0x05:Drive not selected
0x06:No Track00
0x10:ID CRC error
0x11:Uncorrectable data error
0x12:ID Address Mark not found
0x13:No address mark in data field
0x14:Record not found
0x15:Seek Error
0x18:Data Check in no retry mode
0x19:ECC error during verify
0x19:Defect list error
0x1a:Interleave error
0x1a:Parameter overrun
0x1c:Unformatted or bad format on drive
0x1c:Primary defect list not found
0x1d:Compare error
0x20:Illegal Command
0x21:Illegal block address
0x22:Illegal function for device type
0x23:Volume overflow
0x24:Bad argument
0x25:Invalid logical unit number
0x26:Invalid field in parameter list
0x27:Write protected
0x28:Media changed
0x29:Device was reset
0x2a:Mode select parameter changed
0x2c:Error count overflow
0x31:Medium format corrupted
0x40:Ram failure
0x43:Message reject error
0x44:Internal controller error
0x45:Select/reselect failed
0x47:Scsi interface parity error
0x48:Initiator detected error
0x49:Inappropriate/illegal message

Cause and Disposition: The causes of these errors are either the SPU or the appropriate disk drive or disk drive controller. For expanded information regarding interpretation of these error codes, refer to the manual for the controller/drive that is being used by the SPU.

Disk errors: For errors 28–31 a dump of the command will be sent to the screen. This dump looks like:

'S1' device 'MJ'/'MR' block 'B' XX XX XX XX XX XX

'S1' is a variable(string) that is either "Reading" or "Writing"
'MJ' is a variable(dec) that is the major number of the device.
'MR' is a variable(dec) that is the minor number of the device.
'B' is a variable(dec) that is the block number.

The string of 6 hex digits (XX XX XX XX XX XX) is the actual hex representation of the sasi command that was issued.

`SPU_UNIX(32):qic('D') Byte count not multiple of block size - trailing bytes ignored`

Description of Error: A command (read/write) was issued to the cartridge tape drive with a block size that is not a multiple of the inherent block size (512 bytes) of the drive.
The 'D' variable(dec) is the minor number of the device.

Cause and Disposition: The cause is probably errant user software. The transfer will complete using a truncated block size.

`SPU_UNIX(33):qic('D'): Transfer timeout error`

Description of Error: The requested command did not complete within the expected amount of time.
'D' is a variable(dec) that is the minor number of the device.

Cause and Disposition: Refer to the expanded list and discussion of QIC errors below.

`SPU_UNIX(34):qic('D'): 68000 bus error ('X')`

Description of Error: A bus error occurred during a QIC tape transfer operation.
'D' is a variable(dec) that is the minor number of the device.
'X' is a variable(hex) that is the address where the bus error was detected.

Cause and Disposition: The cause of this error is unknown.

`SPU_UNIX(35):qic('D'): unexpected sasierror = 'U'`

Description of Error: Unknown
'D' is a variable(dec) that is the minor number of the device.
'U' is a variable(dec) that is the error number returned for this transfer.

Cause and Disposition: Unknown

Tape errors: For errors 33–35 a dump of the command will be sent to the screen. This dump appears in the following format:

'S1' device 'MJ'/'MR' block 'B' XX XX XX XX XX XX

'S1' is a variable(string) that is either "Reading" or "Writing."
'MJ' is a variable(dec) that is the major number of the device.
'MR' is a variable(dec) that is the minor number of the device.
'B' is a variable(dec) that is the block number.

The string of 6 hex digits (XX XX XX XX XX XX) is the actual hex representation of the SCSI command that was issued.

`SPU_UNIX(36):qic('D'): Error status='E'`

Description of Error: An error was returned from the QIC tape drive via the SCSI bus during some operation. The cause of the error is described in one of the messages below (37–50).
'D' is a variable(dec) that represents the minor number of the device.
'E' is a variable(hex) that represents the returned SCSI error code.
The error code is the code available to a user program through the *iocntl* call. The possible values of the error codes are:

HARDWARE_ERROR 0x1
MEDIUM_ERROR 0x2
RECOVERED_ERROR 0x4
BLANK_CHECK 0x8
VOLUME_OVERFLOW 0x10

Cause and Disposition: Unknown

`SPU_UNIX(37):qic('D'):recovered error`

Description of Error: This error occurs when the QIC drive returns this status code in response to a check status command.
'D' is the minor number of the device.

Cause and Disposition: The last command completed successfully following recovery actions by the tape driver. A check condition was not issued.

`SPU_UNIX(38):qic('D'):not ready`

Description of Error: This error occurs when the QIC drive returns this status code in response to a check status command.
'D' is a variable(dec) that contains the minor number of the device.

Cause and Disposition: The tape drive cannot be accessed. Operator intervention may be required to correct this condition.

`SPU_UNIX(39):qic('D'):medium error`

Description of Error: This error occurs when the QIC drive returns this status code in response to a check status command.
'D' is a variable(dec) that contains the minor number of the device.

Cause and Disposition: The command terminated with a non-recoverable error that was probably caused by a flaw in the medium or an error in the recorded data.

`SPU_UNIX(40):qic('D'):hardware error`

Description of Error: This error occurs when the QIC drive returns this status code in response to a check status command.
'D' is a variable(dec) that contains the minor number of the device.

Cause and Disposition: The tape drive detected a non-recoverable hardware failure (parity, etc.) while performing the command.

`SPU_UNIX(41):qic('D'):illegal request - ignored`

Description of Error: This error occurs when the QIC drive returns this status code in response to a check status command.
'D' is a variable(dec) that contains the minor number of the device.

Cause and Disposition: The command descriptor block contained an illegal parameter. Probably a SPU UNIX software problem.

`SPU_UNIX(42):qic('D'):unit attention`

Description of Error: This error occurs when the QIC drive returns this status code in response to a check status command.
'D' is a variable(dec) that contains the minor number of the device.

Cause and Disposition: The tape drive has been reset or the cartridge has been changed.

`SPU_UNIX(43):qic('D'):data protect`

Description of Error: This error occurs when the QIC drive returns this status code in response to a check status command.
'D' is a variable(dec) that contains the minor number of the device.

Cause and Disposition: The cartridge is write-protected (**SAFE**) when a write operation is attempted.

`SPU_UNIX(44):qic('D'):blank check`

Description of Error: This error occurs when the QIC drive returns this status code in response to a check status command.
'D' is a variable(dec) that contains the minor number of the device.

Cause and Disposition: A no-data condition was encountered on the tape.

`SPU_UNIX(45):qic('D'):vendor unique error - ignored`

Description of Error: This message should not be seen since the drive supposedly will not issue it. If it is seen, the tape drive is probably bad.
'D' is a variable(dec) that contains the minor number of the device.

`SPU_UNIX(46):qic('D'):copy aborted error - ignored`

Description of Error: This message should not be seen since the drive supposedly will not issue it. If it is seen, the tape drive is probably bad.
'D' is a variable(dec) that contains the minor number of the device.

`SPU_UNIX(47):qic('D'):aborted command error - ignored`

Description of Error: This error occurs when the QIC drive returns this status code in response to a check status command.
'D' is a variable(dec) that contains the minor number of the device.

Cause and Disposition: The tape drive aborted the command. The initiator may be able to recover by trying the command again.

`SPU_UNIX(48):qic('D'):volume overflow`

Description of Error: This error occurs when the QIC drive returns this status code in response to a check status command.
'D' is a variable(dec) that contains the minor number of the device.

Cause and Disposition: The tape has reached the physical end-of-medium and data remains in the buffer that has not been written to the tape. Probably user program error.

`SPU_UNIX(49):qic('D'):data miscompare error - ignored`

Description of Error: This message should not be seen since the drive supposedly will not issue it. If it is seen, the tape drive is probably bad.
'D' is a variable(dec) that contains the minor number of the device.

`SPU_UNIX(50):qic('D'):reserved error code 0x0c - ignored`

Description of Error: This message should not be seen since the drive supposedly will not issue it. If it is seen, the tape drive is probably bad.
'D' is a variable(dec) that contains the minor number of the device.

`SPU_UNIX(51):qicioctl: unknown ioctl cmd='C'`

Description of Error: An ioctl system call was issued to the QIC device driver with an unknown command code.
'D' is a variable(dec) that contains the minor number of the device.

Cause and Disposition: This is probably caused by errant user code.

`SPU_UNIX(52):sasibusytest: timed out waiting for idle (cp='X'), resetting...done`

Description of Error: The SCSI bus took longer than allocated to become non-busy (go idle). The SCSI bus is then reset to try to get the bus to become idle.
'X' is a variable(hex) that contains the results of the SPU SCSI status register.

Cause and Disposition: This error is could be caused by either a bad SPU or bad device on the SCSI bus that is holding onto the busy signal.

`SPU_UNIX(53):sasibusytest: SASI busy after reset (cp='S')`
`SPU_UNIX(53):panic:sasibusytest: SASI controller busy after reset`

Description of Error: If after the reset, the bus is still busy, it is assumed that the hardware is in some way broken and a panic occurs.
'S' is a variable that contains the status of the SCSI controller register on the SPU after the timeout.

Cause and Disposition: This is probably caused by some I/O device on the SCSI being broken, a cabling problem or a bad SPU.

`SPU_UNIX(54):sasinitiate: timed out waiting for busy`

Description of Error: The SCSI bus took longer than expected to go busy after the issuance of a command on the bus. The bus is then reset and the command aborted.

Cause and Disposition: The cause of this is probably a device hanging on the SCSI bus or a bad SPU. Software can tolerate this error, so if it only happens occasionally then it is benign.

`SPU_UNIX(55):panic:sasitimeout Sasi_busy`

Description of Error: A condition in which the sasitimeout routine was called and the SCSI bus was not allocated to a device by the SPU.

Cause and Disposition: Unknown

`SPU_UNIX(56):sasiinit(): SASI controller busy! control port = 'S'`

Description of Error: A condition exists in which the SCSI bus is initialized and the bus shows busy during the initialization. This causes any operation pending on the bus to be aborted.
'S' is a variable that contains the status of the SPU's SCSI status register.

Cause and Disposition: Probably a device on the SCSI bus is hanging onto the busy line. This message will often be seen after error #52.

`SPU_UNIX(57):panic:ioccom canq`

Description of Error: Unknown

Cause and Disposition: Unknown

`SPU_UNIX(58):panic:ttyrub`

Description of Error: Unknown

Cause and Disposition: Unknown

`SPU_UNIX(59):ua overrun`

Description of Error: A received data overrun has been detected on the UART.

Cause and Disposition: Data is coming in too fast for the SPU to handle. This error is often seen when running the system at reduced margins where the data rate relative to the SPU's clock increases.

`SPU_UNIX(60):ua overrun`

Description of Error: A received data overrun has been detected on the UART.

Cause and Disposition: Data is coming in too fast for the SPU to handle. This error is often seen when running the system at reduced margins where the data rate relative to the SPU's clock increases.

`SPU_UNIX(61):DSR FALSE--ignored 'C' from #'D'`

Description of Error: The Data Set Ready (DSR) line has been dropped on one of the UARTs.
'C' is a variable(hex) that is the dropped character.
'D' is the uart number where DSR was de-asserted.

Cause and Disposition:
Check the terminal/modem and cabling connected to this port.

`SPU_UNIX(62):wdioctl: No free alloc table entries`

Description of Error: A window map for the current process could not be allocated. Since there is a one to one correspondence between window maps and available processes, this should not happen.

Cause and Disposition: Unknown. UNIX is corrupted.

`SPU_UNIX(63):wdioctl: PID error (exp='E', act='A')`

Description of Error: During a call to the window *iocntl* handler, a window map was allocated that did not belong to the current process.
'E' is a variable(dec) that represents the expected value of the process id (the current process).
'A' is a variable(dec) that represents the actual value of the process id for this window.

Cause and Disposition: This could be caused if a user program tries to do access through a window that has not been allocated yet.

`SPU_UNIX(64):Bad free count on dev 'MJ' 'MN'`

Description of Error: The free count on the specified disk is greater than the allowed maximum.
'MJ' is a variable(dec) that is the major number of the device.
'MN' is a variable(dec) that is the minor number of the device.

Cause and Disposition: The superblock is probably corrupted on the disk. Run *fsck* and try again.

`SPU_UNIX(65):No space on device 'MJ' 'MN'`

Description of Error: The specified device has no more usable free space.
'MJ' is a variable(dec) that is the major number of the device.
'MN' is a variable(dec) that is the minor number of the device.

Cause and Disposition: The device is probably full; remove some files. The filesystem could also be corrupted, run *fsck*.

`SPU_UNIX(66):Bad block on device 'MJ'/'MN'`

Description of Error: A requested block is outside of the range from the end of the ilist to the size of the device.
'MJ' is a variable(dec) that is the major number of the device.
'MN' is a variable(dec) that is the minor number of the device.

Cause and Disposition: A bad file system could have been mounted. Or the file system could be corrupted.

`SPU_UNIX(67):Out of inodes on device 'MJ'/'MN'`

Description of Error: There are no more inodes available on the specified device.
'MJ' is a variable(dec) that is the major number of the device.
'MN' is a variable(dec) that is the minor number of the device.

Cause and Disposition: Could be too many files in existence; delete some files. Filesystem could be corrupted; run *fsck*.

`SPU_UNIX(68):Bad count on device 'MJ'/'MN'`

Description of Error: Either the number of inodes available or the number of free blocks available is greater than the allowable maximum. Both of these values will then get set to zero, which will cause the no space message to be issued during the next transfer.
'MJ' is a variable(dec) that is the major number of the device.
'MN' is a variable(dec) that is the minor number of the device.

Cause and Disposition: Probably a corrupted filesystem. Run *fsck*.

`SPU_UNIX(69):panic:No fs`

Description of Error: The device for which a request is being made is not in the mount table.

Cause and Disposition: UNIX is corrupted. Reboot.

`SPU_UNIX(70):panic:Timeout table overflow`

Description of Error: During a call to the timeout function, there is no space in the callout structure to place the current request. There is nothing intelligent to be done if an entry will not fit, thus the panic.

Cause and Disposition: Probably a corrupted UNIX. Reboot and try again.

`SPU_UNIX(71):Detected illegal line clock of 'H' Hz, using 60 Hz`

Description of Error: SPU UNIX tries to determine the line frequency to know if it is running on a 60 Hz or 50 Hz system. If the measured value is not close to either 50 Hz or 60 Hz, this message will appear.
'H' is a variable that contains the measured clock frequency

Cause and Disposition: The cause of this could be a bad SPU timer chip which is used to calculate the line frequency. Also a bad SCM could be causing a low quality clock signal to reach the SPU. The electrical power utility serving the site could be having trouble with their generators, although this is unlikely.

`SPU_UNIX(72):No detectable line clock -- using 60 Hz`

Description of Error: During the time that the SPU is trying to measure the line clock frequency, no clock is detected.

Cause and Disposition: This could be caused by a bad timer chip on the SPU or a poor quality signal being generated by the System Control Module (SCM). If there truly is no line clock, SPU UNIX will hang shortly after this message appears.

`SPU_UNIX(73):Falloc:No file`

Description of Error: During a call to the *falloc* routine to allocate a file descriptor, no file descriptors/structures are available.

Cause and Disposition: This message is usually no cause for alarm. Generally, this message appears when too many files have been opened by one of the processes currently executing on the SPU. Try to determine which process has opened the excessive number of files.

`SPU_UNIX(74):panic:Fs not in mount table`

Description of Error: An inode is mounted, but the mounted file system is not found in the mount table.

Cause and Disposition: UNIX is corrupted. Reboot and try again.

`SPU_UNIX(75):Inode table overflow`

Description of Error: Too many inodes are currently in use (in core).

Cause and Disposition: Too many processes are running with too many inodes in use. The problem should be self-correcting. Check which processes are running.

`SPU_UNIX(76):Iaddress > 2^24`

Description of Error: A block address referenced inside an inode structure is $> 2^{24}$. This message is for information purposes only, since it appears that no other action is taken when this message occurs.

Cause and Disposition: The in-core copy of the inode is probably corrupt in some manner. Reboot.

`SPU_UNIX(77):Link error. &ftrap = 'X'fR`

`OR`

`SPU_UNIX(77):Link error. &fault = 'X'`

Description of Error: These errors will occur during booting the system if there is an address alignment error with either the fault or ftrap structures.
'X' is a variable(hex) that represents the address of the structure.

Cause and Disposition: This should only happen if the disk copy of UNIX is corrupted. Try a root restore.

`SPU_UNIX(78): panic:No room for scn_structs`

Description of Error: This is a SP2 message that only occurs during the boot process. There is not enough memory available to hold the structures for the system scan ring descriptions.

Cause and Disposition: Something is probably wrong with the version of SPU UNIX, or the SPU memory system could be broken in such a way that the memory sizing routine determines that there is less available memory than is actually installed.

`SPU_UNIX(79): panic:No room for ring structs`

Description of Error: This is a SP2 message that only occurs during the boot process. There is not enough memory available to hold the structures for the system scan ring descriptions.

Cause and Disposition: Something is probably wrong with the version of SPU UNIX, or the SPU memory system could be broken in such a way that the memory sizing routine determines that there is less available memory than is actually installed.

`SPU_UNIX(80):panic:Iinit can not read superblock`

Description of Error: During the initialization process, an I/O error occurs during the attempted read of the superblock.

Cause and Disposition: An I/O error will probably proceed this message. Take appropriate action based on the results of the I/O error.

`SPU_UNIX(81):Msg_exit: lost message chain from index %d`

Description of Error: Unknown

Cause and Disposition: Unknown

`SPU_UNIX(82):Msg_intr: Bus error in step 'S' ADDR='X'`

Description of Error: During the handling of a Message Based System (MBS) related system interrupt (system interrupt 11), a bus error occurred.
'S' is a variable(dec) that represents which step in the code was being executed when the bus error occurred.
'X' is a variable(hex) that represents the address where the bus error was detected.

Cause and Disposition: The cause of this error is generally not known. It is suspected that it could be a bad SPU. However a bad memory system could also cause this error since many of the mbs related functions are accessing main memory.

`SPU_UNIX(83):Msg_intr: Queue locked for 'P' passes. DISABLED`

Description of Error: After the receipt of an interrupt an attempt to de-queue a message fails because the queue remains locked.

Cause and Disposition: Could be an inadvertent interrupt or someone may have destroyed the queue structure in main memory.

`SPU_UNIX(85):Ptrace:Shared text`

Description of Error: A system *ptrace* system call was executed that tried to write into the text segment of a process that was using a shared text segment.

Cause and Disposition: Check for multiple invocations of a given process running. The adb utility is one of the few that use the *ptrace* system call.

`SPU_UNIX(86):Ptrace:usraccess 'X'`

Description of Error: A *ptrace* system call was made that tried to access kernel memory from user space.
'X' is a variable(hex) that contains the address of the attempted access.

Cause and Disposition: Probably errant user software incorrectly using the *ptrace* system call.

`SPU_UNIX(87):Ptrace:suiword failed`

Description of Error: During the execution of a *ptrace* system call, a copy of a word to user space failed.

Cause and Disposition: This is probably caused by an improper address passed to the *ptrace* call or to malfunctioning hardware on the SPU.

`SPU_UNIX(88):Proc on q`

Description of Error: A process is being added to the run queue that already exists on the run queue.

Cause and Disposition: Probably a corrupted UNIX. The process will not be added to the run queue so it may be possible to recover from this error.

`SPU_UNIX(89):panic:Running a dead proc`

Description of Error: A process is set to run that is in a zombie state (exited and waiting for parent to accept status with wait call).

Cause and Disposition: UNIX is corrupted. Reboot

`SPU_UNIX(90):panic:No procs`

Description of Error: During the execution of a *fork* system call a routine detects that there are no slots in the process table available. This is a panic at this point because prior to the point of the panic, the table was checked and there were slots in the process table.

Cause and Disposition: This is a transient UNIX-is-damaged type of error and will probably never appear. But if it does, reboot.

`SPU_UNIX(91):panic:Out of swap`

Description of Error: During the execution of an *exec* system call the system runs out of swap space.

Cause and Disposition: This error should not happen if the system is configured correctly. Could be that the entries in *dkparmtab* for */dev/swap* are corrupted. Try to reboot. If it appears again, do a root restore.

`SPU_UNIX(92):User access to kernel space 'X'`

Description of Error: During the execution of a read or write system call, a pointer is passed to the kernel that is outside of user space. The system call will then be aborted.
'X' is a variable(hex) that represents the address that was being referenced.

Cause and Disposition: This is probably caused by errant user code and can be ignored.

`SPU_UNIX(93):panic:Out of swap space`

Description of Error: During the swapping out of a process, the system runs out of swap space.

Cause and Disposition: This error should not happen if the system is configured correctly. Could be that the entries in *dkparmtab* for */dev/swap* are corrupted. Try to reboot. If it appears again, do a root restore.

`SPU_UNIX(94):Out of text`

Description of Error: During the establishment of the text region for a process, the text descriptor structure overflows.

Cause and Disposition: Probably too many processes are running. Or UNIX's internal structures are corrupted.

`SPU_UNIX(95):panic:Out of swap space`

Description of Error: During the creation of a text segment, the system runs out of swap space.

Cause and Disposition: This error should not happen if the system is configured correctly. Could be that the entries in *dkparmtabr* for */dev/swap* are corrupted. Try to reboot. If it appears again, do a root restore.

`SPU_UNIX(96):Trap: 'TRAPMESSAGE', 'MODE' mode`

Description of Error: These are system exceptions that occur during the execution of SPU UNIX. The user mode errors can be safely ignored. The system mode errors, however, will cause a panic.

TRAPMESSAGE is a variable(string) that describes the trap. The values this variable can take on are:

reset initial ssp?	level 1 autovector
reset initial pc?	level 2 autovector
bus error	level 3 autovector
address error	level 4 autovector
illegal instruction	level 5 autovector
zero divide	level 6 autovector
chk instruction	level 7 autovector
trapv instruction	trap #0
privilege violation	trap #1
trace	trap #2
line 1010 emulator	trap #3
line 1111 emulator	trap #4
reserved #1	trap #5
reserved #2	trap #6
reserved #3	trap #7
reserved #4	trap #8
reserved #5	trap #9
reserved #6	trap #A
reserved #7	trap #B
reserved #8	trap #C
reserved #9 trap #D	
reserved #10 trap #E	
reserved #11 trap #F	
reserved #12	
spurious interrupt	

These trap messages have a one to one correspondence with 68000 trap messages. For more information on these trap messages, refer to the *Motorola MC68000 User's Manual.*

'MODE' is a variable (string) that can take on the values "user" "kernel." This variable tells where the system was executing when the trap occurred. A trap will follow (message 98).

Cause and Disposition: The cause should be determined from the message. Kernel messages should not normally occur, could be a sign of bad SPU hardware or a bad copy of the kernel. Reboot. If there are still problems, replace the SPU.

`SPU_UNIX(97):Trap: interrupt vector 'X', 'MODE'`

Description of Error: A trap was taken for either an unassigned reserved vector or a user interrupt vector that was not expected. A trap will follow (message 98).

Cause and Disposition: Probably broken hardware on the SPU.

`SPU_UNIX(98):panic:trap`

Description of Error: This message is issued after either message number 96 or 97.

Cause and Disposition: Refer to message number 96 or 97

`SPU_UNIX(99):Trap: SCM Power Fail Interrupt`

Description of Error: The System Control Module (SCM) detected a power fail condition and sent an interrupt.

Cause and Disposition: If there is actually a power fail condition, the SPU will die immediately after issuing this message. Fix the power problem. If the SPU stays alive then the SCM or the SPU (or the interface) could be broken in such a way that the SPU is detecting these interrupts.

`SPU_UNIX(100):Trap: SPU memory parity error, 'MODE' mode`

Description of Error: A parity error has been detected in the SPU RAM. 'MODE' is a variable (string) that is either "user" or "kernel."

Cause and Disposition: If the parity error occurs in a user process, that process is sent a segmentation violation signal. If the parity error occurs in the kernel then a panic occurs (refer to message 101). If the parity errors are frequent, replace the SPU.

`SPU_UNIX(101):panic:SPU memory parity error in kernel`

Description of Error: A parity error occurred while executing in kernel space.

Cause and Disposition: Refer to message message 100.

`SPU_UNIX(102):Trap: kernel mode bus error`

OR

`SPU_UNIX(102):Trap: kernel mode address error`

Description of Error: A bus or an address error occurred while executing in kernel space. A panic will immediately follow.

Cause and Disposition: This could be caused by faulty hardware or corrupted software. Reboot. If the problem persists, replace the SPU.

`SPU_UNIX(103):panic:Kernel bus/address error`

Description of Error: This panic is caused by error 102.

Cause and Disposition: Refer to message 102.

`SPU_UNIX(104):Trap: spurious interrupt ignored`

Description of Error: The 68000 generated a spurious interrupt trap.

Cause and Disposition: This is caused when a bus error occurs during a 68000 interrupt acknowledge cycle. This is probably harmless unless it occurs often, in which case, replace the SPU.

`SPU_UNIX(105):Catchint:pid='P' signo='S',ix='I'`

Description of Error: A call to *catchint* has been made with a vector greater than 256, which is greater than possible on the SPU.
'P' is a variable(dec) that is the process number of the process making the call.
'S' is a variable(hex) that is the signal number trying to be assigned.
'I' is a variable(dec) that represents the index into an internal array of interrupts that can be caught.

Cause and Disposition: This is caused by errant user code.

`SPU_UNIX(106):Pitchint - invalid vector 'X'`

Description of Error: An interrupt occurred for a potentially catchable interrupt. But when the index of the interrupt is referenced it is not found.
'X' is a variable(hex) that is the 68000 vector.

Cause and Disposition: This is a corrupt UNIX error. Reboot and try again.

Discussion of Trap Messages Resulting from Hardware Interrupts

The following messages (error codes 107–126) are caused by the SPU receiving an interrupt from some hardware source that is normally under the control of a user program. SPU UNIX has a provision for mapping these hardware interrupts into signals through the two system calls *catchint* and *signal*.

Used in conjunction, these system calls allow for a user program to catch the occurrence of some hardware interrupt. In order for one of these interrupts to be accepted by the SPU, the bit in the SPU's interrupt enable register must be set to enable the particular interrupt. If the SPU detects one of these interrupts and no user program has previously tried to "catch" the interrupt via the *catchint* system call, one of these trap messages will be displayed.

Thus, for one of these messages to occur, the hardware must generate the source of the interrupt, the bit in the Interrupt Enable Register (IER) corresponding to the interrupt must be set, AND there must be no user program with an outstanding catch on that interrupt.

Getting one of these messages could be caused by faulty user code setting a bit in the IER incorrectly or faulty hardware generating or receiving unexpected interrupts.

`SPU_UNIX(107):Trap: SPU environment error`

Description of Error: Unknown

Cause and Disposition: Unknown

`SPU_UNIX(108):Trap: Hard error`

Description of Error: Unknown

Cause and Disposition: Unknown

`SPU_UNIX(109):Trap: Memory soft error`

Description of Error: Unknown

Cause and Disposition: Unknown

`SPU_UNIX(110):Trap: System interrupt acknowledge`

Description of Error: Unknown

Cause and Disposition: Unknown

`SPU_UNIX(111):Trap: System interrupt 15`

Description of Error: System interrupt 15 was received by the SPU when no user program was expecting it.

Cause and Disposition: There are several possibilities for this error. Any processor in the system can interrupt on this channel. An errant program in any of these processors (including the SPU) could send an interrupt to the SPU on this channel. The SPUs interrupt detection logic could be broken, as could any other processor's interrupt generation logic. Most of these messages have been traced to software problems on one of the processors, especially the SPU.

`SPU_UNIX(112):Trap: System interrupt 14`

Description of Error: System interrupt 14 was received by the SPU when no user program was expecting it.

Cause and Disposition: Refer to description for message 111.

`SPU_UNIX(113):Trap: System interrupt 13`

Description of Error: System interrupt 13 was received by the SPU when no user program was expecting it.

Cause and Disposition: Refer to description for message 111.

`SPU_UNIX(114):Trap: System interrupt 12`

Description of Error: System interrupt 12 was received by the SPU when no user program was expecting it.

Cause and Disposition: Refer to description for message 111.

`SPU_UNIX(115):Trap: System interrupt 11`

Description of Error: System interrupt 11 was received by the SPU when no user program was expecting it.

Cause and Disposition: Refer to description for message 111.

`SPU_UNIX(116):Trap: System interrupt 10`

Description of Error: System interrupt 10 was received by the SPU when no user program was expecting it.

Cause and Disposition: Refer to description for message 111.

`SPU_UNIX(117):Trap: System interrupt 09`

Description of Error: System interrupt 9 was received by the SPU when no user program was expecting it.

Cause and Disposition: Refer to description for message 111.

`SPU_UNIX(118):Trap: System interrupt 08`

Description of Error: System interrupt 8 was received by the SPU when no user program was expecting it.

Cause and Disposition: Refer to description for message 111.

`SPU_UNIX(119):Trap: DMA channel 3 normal interrupt`

Description of Error: An interrupt was generated by the DMA channel 3 interrupt circuitry that was not expected by any user program.

Cause and Disposition: This could be caused by an errant user program or by faulty DMA hardware.

`SPU_UNIX(120):Trap: DMA channel 3 error interrupt`

Description of Error: An interrupt was generated by the DMA channel 3 error interrupt circuitry that was not expected by any user program.

Cause and Disposition: This could be caused by an errant user program or by faulty DMA hardware.

`SPU_UNIX(121):Trap: DMA channel 2 normal interrupt`

Description of Error: An interrupt was generated by the DMA channel 2 interrupt circuitry that was not expected by any user program.

Cause and Disposition: This could be caused by an errant user program or by faulty DMA hardware.

`SPU_UNIX(122):Trap: DMA channel 2 error interrupt`

Description of Error: An interrupt was generated by the DMA channel 2 error interrupt circuitry that was not expected by any user program.

Cause and Disposition: This could be caused by an errant user program or by faulty DMA hardware.

`SPU_UNIX(123):Trap: DMA channel 1 normal interrupt`

Description of Error: An interrupt was generated by the DMA channel 1 interrupt circuitry that was not expected by any user program.

Cause and Disposition: This could be caused by an errant user program or by faulty DMA hardware.

`SPU_UNIX(124):Trap: DMA channel 1 error interrupt`

Description of Error: An interrupt was generated by the DMA channel 1 error interrupt circuitry that was not expected by any user program.

Cause and Disposition: This could be caused by an errant user program or by faulty DMA hardware.

`SPU_UNIX(125):Trap: DMA channel 0 normal interrupt`

Description of Error: An interrupt was generated by the DMA channel 0 interrupt circuitry that was not expected by any user program.

Cause and Disposition: This could be caused by an errant user program or by faulty DMA hardware.

`SPU_UNIX(126):Trap: DMA channel 0 error interrupt`

Description of Error: An interrupt was generated by the DMA channel 0 error interrupt circuitry that was not expected by any user program.

Cause and Disposition: This could be caused by an errant user program or by faulty DMA hardware.

`SPU_UNIX(127):ct('D'):Error:Pos 'TAPEERROR1' 'TAPEERROR3'`

Description of Error: Unknown

Cause and Disposition: Unknown

`SPU_UNIX(128):ct('D'):Error:Data 'TAPEERROR2' 'TAPEERROR3'`

Description of Error: Unknown

Cause and Disposition: (Expanded display for error messages 20, 21, 127, 128, and 22)

In the errors above, there are three variable fields not explained: 'TAPEERROR1', 'TAPEERROR2', and 'TAPEERROR3'. These three fields are described here.

The **'TAPEERROR1'** message consists of one or more messages. Each of these messages corresponds to a bit in the status register of the device. The possible messages and their meaning are:
FifoNtEmpty:
NotRdy:
WrtPrt:
B5:
SeekErr:
CrcErr:
B2:

The **'TAPEERROR2'** message consists of one or more messages. Each of these messages corresponds to a bit in the status register of the device. The possible messages and their meaning are:
FifoNtEmpty:
NotRdy:
WrtPrt:
DelData:
RcdNtFnd:
CrcErr:
LostData:
These strings are all printed separated by blanks on a single line.

The **'TAPEERROR3'** message consists of the following:
cterr='C', retries='R'
ct('D'): stream='S' segment='SG' 'V'='SV'
The 'C' variable(hex) is the error number returned by the low-level cartridge tape routines the possible values and meanings are:
The 'R' variable(dec) is the number of times that the command is tried before the error is reported
The 'D' variable(dec) is the minor number of the device
The 'S' variable(dec) is the stream number where the error occurred
The 'SG' variable(dec) is the segment number where the error was detected
The variable 'V' (string) is either "current segment" or "sector"
The variable 'SV' (dec) is either the sector number or the segment number depending on the value of 'V'

Appendix C
Register Dump Display Screen

C.1 Overview

This appendix explains the register dump display screen. The information that the register dump display screen contains can be important in troubleshooting and determining causes for errors. The following is an example of a typical register dump display screen.

```
Register dump for cpu:  1

a0: 00007000  s0: 0a004000 0100f000  t0: 40000000  pc:   00009b44
a1: 00000000  s1: 0a030000 08821000  t1: 00020002  psw:  02800040
a2: 0002e007  s2: 38004000 0810f000  t2: 0023e006  ipc:       1ad
a3: 4c00c000  s3: 08130002 38801000  t3: 0020c000  ccr:    604240
a4: 00080000  s4: 38924000 7890f001  t4: 00220000  cir:  0 .tid:  00
a5: 00000000  s5: 38910000 08121000  t5: 00220000  vl:  20   vs:  00000008
a6: 0001e000  s6: 0c004000 3490f000  t6: 0020e000 vm_u:  00000000 00000000
a7: 0000c000  s7: 30830005 00101000  t7: 4021c000 vm_l:  00000000 00000000
global_int_enab: 00  local_int_enab: 00  int_mode: 00  target_CPU: 0  ION: 0
```

NOTE

Systems with two heads will print this message out twice, once for CPU 1 and once for CPU 2.

The values that are represented in this example will vary depending on the circumstances. Each register's abbreviation is defined in the following table:

Table C–1, Register Definitions

REGISTER ABBREVIATION	MEANING
a0 - a7	Address registers
s0 - s7	Scalar registers
t0 - t7	Temporary registers
pc	Program Counter
psw	Program Status Word
ipc	Instruction Program Counter
ccr	Condition Code Register
cir	Communication Index Register
tid	Thread ID Register
vl	Vector Length
vs	Vector Stride
vm_u	Vecter Merge, upper 64 bits
vm_l	Vector Merge, lower 64 bits
global_int_enab	Gobal Interrupt Enable
local_int_enab	Local Interrupt Enable
int_mode	Interrupt Mode
target_CPU	Target CPU
ION	Interrupt On (0 for on, 1 for off)

Appendix D
Opcodes Sorted By Name

D.1 Overview

This appendix lists all machine opcodes sorted in alphabetical order by name. This appendix is especially useful for the CPU diagnostics.

When a diagnostic test fails, it will generally indicate the "instruction under test" at the moment that the test failed. It is important to note, however, that the instruction under test is not always the instruction that caused the test to fail.

The following table lists instruction abbreviations, a description of each, and which board is most likely to have the problem. If multiple boards are listed, they appear in decending order of probability.

D.2 CPU Instruction Glossary

Instruction Mnemonic	Instruction Description	Suspect Board(s)
add.b Sj,Sk	Add scalar/scalar integer byte	ASP
add.b Vi,Sj,Vk	Add vector/scalar integer byte	VPC, VPD, ASP
add.b Vi,Vj,Vk	Add vector/vector integer byte	VPC, VPD
add.b.f Vi,Sj,Vk	Add vector/scalar byte using VM	VPC, VPD, ASP
add.b.f Vi,Vj,Vk	Add vector/vector byte using not VM	VPC, VPD
add.b.t Vi,Sj,Vk	Add vector/scalar byte using VM	VPC, VPD, ASP
add.b.t Vi,Vj,Vk	Add vector/vector byte using VM	VPC, VPD
add.d Sj,Sk	Add scalar/scalar double float	SFU, ASP, IPP
add.d Vi,Sj,Vk	Add vector/scalar double float	VPC, VPD, ASP
add.d Vi,Vj,Vk	Add vector/vector double float	VPC, VPD
add.d.f Vi,Sj,Vk	Add vector/scalar double using not VM	VPC, VPD, ASP
add.d.f Vi,Vj,Vk	Add vector/vector double using not VM	VPC, VPD
add.d.t Vi,Sj,Vk	Add vector/scalar double using VM	VPC, VPD, ASP
add.d.t Vi,Vj,Vk	Add vector/vector double using VM	VPC, VPD
add.h #N,Ak	Add immediate address halfword	ASP, IPP
add.h #N,Sk	Add scalar/immediate integer halfword	ASP, IPP
add.h #n,Ak	Add short immediate address halfword	ASP, IPP
add.h Aj,Ak	Add address register halfword	ASP
add.h Sj,Sk	Add scalar/scalar integer halfword	ASP
add.h Vi,Sj,Vk	Add vector/scalar integer halfword	VPC, VPD, ASP

add.h Vi,Vj,Vk	Add vector/vector integer halfword	VPC, VPD
add.h.f Vi,Sj,Vk	Add vector/scalar halfword using not VM	VPC, VPD, ASP
add.h.f Vi,Vj,Vk	Add vector/vector halfword using not VM	VPC, VPD
add.h.t Vi,Sj,Vk	Add vector/scalar halfword using VM	VPC, VPD, ASP
add.h.t Vi,Vj,Vk	Add vector/vector halfword using VM	VPC, VPD
add.l Sj,Sk	Add scalar/scalar integer longword	ASP
add.l Vi,Sj,Vk	Add vector/scalar integer longword	VPC, VPD, ASP
add.l Vi,Vj,Vk	Add vector/vector integer longword	VPC, VPD
add.l.f Vi,Sj,Vk	Add vector/scalar longword using not VM	VPC, VPD, ASP
add.l.f Vi,Vj,Vk	Add vector/vector longword using not VM	VPC, VPD
add.l.t Vi,Sj,Vk	Add vector/scalar longword using VM	VPC, VPD, ASP
add.l.t Vi,Vj,Vk	Add vector/vector longword using VM	VPC, VPD
add.s #N,Sk	Add scalar/immediate single float	SFU, ASP, IPP
add.s Sj,Sk	Add scalar/scalar single float	SFU, ASP, IPP
add.s Vi,Sj,Vk	Add vector/scalar single float	VPC, VPD, ASP
add.s Vi,Vj,Vk	Add vector/vector single float	VPC, VPD
add.s.f Vi,Sj,Vk	Add vector/scalar single using not VM	VPC, VPD, ASP
add.s.f Vi,Vj,Vk	Add vector/vector single using not VM	VPC, VPD
add.s.t Vi,Sj,Vk	Add vector/scalar single using VM	VPC, VPD, ASP
add.s.t Vi,Vj,Vk	Add vector/vector single using VM	VPC, VPD
add.w #N,Ak	Add immediate address word	ASP, IPP
add.w #N,Sk	Add scalar/immediate integer word	ASP, IPP
add.w #n,Ak	Add short immediate address word	ASP, IPP
add.w Aj,Ak	Add address register word	ASP
add.w Sj,Ak	Add scalar to address word	ASP
add.w Sj,Sk	Add scalar/scalar integer word	ASP
add.w Vi,Sj,Vk	Add vector/scalar integer word	VPC, VPD, ASP
add.w Vi,Vj,Vk	Add vector/vector integer word	VPC, VPD
add.w.f Vi,Sj,Vk	Add vector/scalar word using not VM	VPC, VPD, ASP
add.w.f Vi,Vj,Vk	Add vector/vector word using not VM	VPC, VPD
add.w.t Vi,Sj,Vk	Add vector/scalar word using VM	VPC, VPD, ASP
add.w.t Vi,Vj,Vk	Add vector/vector word using VM	VPC, VPD
all Vk	AND reduce a vector	VPC, VPD, ASP
all.f Vk	AND reduce a vector using not VM	VPC, VPD, ASP
all.t Vk	AND reduce a vector using VM	VPC, VPD, ASP
and #N,Ak	AND immediate to address register	ASP, IPP
and #N,Sk	AND scalar/immediate	ASP, IPP
and Aj,Ak	AND address register	ASP
and Sj,Sk	AND scalar/scalar	ASP
and Vi,Sj,Vk	AND vector/scalar	VPC, VPD, ASP
and Vi,Vj,Vk	AND two vectors	VPC, VPD
and.f Vi,Sj,Vk	AND vector/scalar using not VM	VPC, VPD, ASP
and.f Vi,Vj,Vk	AND two vectors using not VM	VPC, VPD
and.t Vi,Sj,Vk	AND vector/scalar using VM	VPC, VPD, ASP
and.t Vi,Vj,Vk	AND two vectors using VM	VPC, VPD

any Vk	OR reduce a vector	VPC, VPD, ASP
any.f Vk	OR reduce a vector using not VM	VPC, VPD, ASP
any.t Vk	OR reduce a vector using VM	VPC, VPD, ASP
atan.d Sk	Arc-tangent of a double precision number	SFU, ASP, IPP
atan.s Sk	Arc-tangent of a single precision number	SFU, ASP, IPP
bkpt	Breakpoint	ASP, DCU, SFU, MCM
br	Branch always	IPP, ASP
bra.f	Branch on address carry false	IPP, ASP
bra.t	Branch on address carry true	IPP, ASP
bri.f	Branch on ION false	IPP, ASP
bri.t	Branch on ION true	IPP, ASP
brs.f	Branch on scalar carry false	IPP, ASP
brs.t	Branch on scalar carry true	IPP, ASP
call <effa>	Call a subroutine, long frame	ASP, DCU, SFU, MCM
callq <effa>	Push the PC and jump	ASP, DCU, SFU, MCM
calls <effa>	Call a subroutine, short frame	ASP, DCU, SFU, MCM
cfork	Clear a fork event	ASP, CPX, IPP
cos.d Sk	Cosine of a double precision number	SFU, ASP, IPP
cos.s Sk	Cosine of a single precision number	SFU, ASP, IPP
cprs.f Vj,Vk	Compress a vector using not VM	VPC, VPD
cprs.t Vj,Vk	Compress a vector using VM	VPC, VPD
ctrsg	Global synchronize CPU timers	ASP, CPX, IPP
cvtb.w Aj,Ak	Convert byte to word	ASP
cvtb.w Sj,Sk	Convert byte to word	ASP
cvtb.w Vj,Vk	Convert byte to word	VPC, VPD
cvtb.w.f Vj,Vk	Convert byte to word using not VM	VPC, VPD
cvtb.w.t Vj,Vk	Convert byte to word using VM	VPC, VPD
cvtd.l Sj,Sk	Convert double float to longword	SFU, ASP, IPP
cvtd.l Vj,Vk	Convert double float to longword	VPC, VPD
cvtd.l.f Vj,Vk	Convert double to longword using not VM	VPC, VPD
cvtd.l.t Vj,Vk	Convert double to longword using VM	VPC, VPD
cvtd.s Sj,Sk	Convert double float to single float	SFU, ASP, IPP
cvtd.s Vj,Vk	Convert double float to single float	VPC, VPD
cvtd.s.f Vj,Vk	Convert double to single using not VM	VPC, VPD
cvtd.s.t Vj,Vk	Convert double to single using VM	VPC, VPD
cvtd.w Sj,Sk	Convert double float to word	SFU, ASP, IPP
cvtd.w Vj,Vk	Convert double to word	VPC, VPD
cvtd.w.f Vj,Vk	Convert double to word using not VM	VPC, VPD
cvtd.w.t Vj,Vk	Convert double to word using VM	VPC, VPD
cvth.w Aj,Ak	Convert half to word	ASP
cvth.w Sj,Sk	Convert halfword to word	ASP
cvth.w Vj,Vk	Convert halfword to word	VPC, VPD
cvth.w.f Vj,Vk	Convert halfword to word using not VM	VPC, VPD
cvth.w.t Vj,Vk	Convert halfword to word using VM	VPC, VPD
cvtl.d Sj,Sk	Convert longword to double float	SFU, ASP, IPP

cvtl.d Vj,Vk	Convert longword to double float	VPC, VPD
cvtl.d.f Vj,Vk	Convert longword to double using not VM	VPC, VPD
cvtl.d.t Vj,Vk	Convert longword to double using VM	VPC, VPD
cvtl.s Sj,Sk	Convert longword to single float	SFU, ASP, IPP
cvtl.s Vj,Vk	Convert longword to single float	VPC, VPD
cvtl.s.f Vj,Vk	Convert longword to single using not VM	VPC, VPD
cvtl.s.t Vj,Vk	Convert longword to single using VM	VPC, VPD
cvtl.w Sj,Sk	Convert longword to word	ASP
cvtl.w Vj,Vk	Convert longword to word	VPC, VPD
cvtl.w.f Vj,Vk	Convert longword to word using not VM	VPC, VPD
cvtl.w.t Vj,Vk	Convert longword to word using VM	VPC, VPD
cvts.d Sj,Sk	Convert single float to double float	SFU, ASP, IPP
cvts.d Vj,Vk	Convert single float to double float	VPC, VPD
cvts.d.f Vj,Vk	Convert single to double using not VM	VPC, VPD
cvts.d.t Vj,Vk	Convert single to double using VM	VPC, VPD
cvts.l Sj,Sk	Convert single float to longword	SFU, ASP, IPP
cvts.l Vj,Vk	Convert single float to longword	VPC, VPD
cvts.l.f Vj,Vk	Convert single to longword using not VM	VPC, VPD
cvts.l.t Vj,Vk	Convert single to longword using VM	VPC, VPD
cvts.w Sj,Sk	Convert single float to word	SFU, ASP, IPP
cvts.w Vj,Vk	Convert single float to word	VPC, VPD
cvts.w.f Vj,Vk	Convert single to word using not VM	VPC, VPD
cvts.w.t Vj,Vk	Convert single to word using VM	VPC, VPD
cvtw.b Aj,Ak	Convert word to byte	ASP
cvtw.b Sj,Sk	Convert word to byte	ASP
cvtw.b Vj,Vk	Convert word to byte	VPC, VPD
cvtw.b.f Vj,Vk	Convert word to byte using not VM	VPC, VPD
cvtw.b.t Vj,Vk	Convert word to byte using VM	VPC, VPD
cvtw.d Sj,Sk	Convert word to double float	SFU, ASP, IPP
cvtw.d Vj,Vk	Convert word to double	VPC, VPD
cvtw.d.f Vj,Vk	Convert word to double using not VM	VPC, VPD
cvtw.d.t Vj,Vk	Convert word to double using VM	VPC, VPD
cvtw.h Aj,Ak	Convert word to halfword	ASP
cvtw.h Sj,Sk	Convert word to halfword	ASP
cvtw.h Vj,Vk	Convert word to halfword	VPC, VPD
cvtw.h.f Vj,Vk	Convert word to halfword using not VM	VPC, VPD
cvtw.h.t Vj,Vk	Convert word to halfword using VM	VPC, VPD
cvtw.l Sj,Sk	Convert word to longword	ASP
cvtw.l Vj,Vk	Convert word to longword	VPC, VPD
cvtw.l.f Vj,Vk	Convert word to longword using not VM	VPC, VPD
cvtw.l.t Vj,Vk	Convert word to longword using VM	VPC, VPD
cvtw.s Sj,Sk	Convert word to single float	SFU, ASP, IPP
cvtw.s Vj,Vk	Convert word to single float	VPC, VPD
cvtw.s.f Vj,Vk	Convert word to single using not VM	VPC, VPD
cvtw.s.t Vj,Vk	Convert word to single using VM	VPC, VPD

diag Ak	Execute nonstandard microcode sequence	ASP
div.b Sj,Sk	Divide scalar/scalar integer byte	SFU, ASP, IPP
div.b Vi,Sj,Vk	Divide vector/scalar integer byte	VPC, VPD, ASP
div.b Vi,Vj,Vk	Divide vector/vector integer byte	VPC, VPD
div.b.f Vi,Sj,Vk	Divide vector/scalar byte using not VM	VPC, VPD, ASP
div.b.f Vi,Vj,Vk	Divide byte vectors using not VM	VPC, VPD
div.b.t Vi,Sj,Vk	Divide vector/scalar byte using VM	VPC, VPD, ASP
div.b.t Vi,Vj,Vk	Divide byte vectors using VM	VPC, VPD
div.d Si,Vj,Vk	Divide scalar/vector double float	VPC, VPD, ASP
div.d Sj,Sk	Divide scalar/scalar double float	SFU, ASP, IPP
div.d Vi,Sj,Vk	Divide vector/scalar double float	VPC, VPD, ASP
div.d Vi,Vj,Vk	Divide vector/vector double float	VPC, VPD
div.d.f Si,Vj,Vk	Divide scalar/vector double using not VM	VPC, VPD, ASP
div.d.f Vi,Sj,Vk	Divide vector/scalar double using not VM	VPC, VPD, ASP
div.d.f Vi,Vj,Vk	Divide double vectors using not VM	VPC, VPD
div.d.t Si,Vj,Vk	Divide scalar/vector double using VM	VPC, VPD, ASP
div.d.t Vi,Sj,Vk	Divide vector/scalar double using VM	VPC, VPD, ASP
div.d.t Vi,Vj,Vk	Divide double vectors using VM	VPC, VPD
div.h #N,Ak	Divide immediate address halfword	SFU, ASP, IPP
div.h #N,Sk	Divide scalar/scalar integer halfword	SFU, ASP, IPP
div.h #n,Ak	Divide short immediate address halfword	SFU, ASP, IPP
div.h Aj,Ak	Divide address register halfword	SFU, ASP, IPP
div.h Sj,Sk	Divide scalar/scalar integer halfword	SFU, ASP, IPP
div.h Vi,Sj,Vk	Divide vector/scalar integer halfword	VPC, VPD, ASP
div.h Vi,Vj,Vk	Divide vector/vector integer halfword	VPC, VPD
div.h.f Vi,Sj,Vk	Divide vector/scalar halfword using not VM	VPC, VPD, ASP
div.h.f Vi,Vj,Vk	Divide halfword vectors using not VM	VPC, VPD
div.h.t Vi,Sj,Vk	Divide vector/scalar halfword using VM	VPC, VPD, ASP
div.h.t Vi,Vj,Vk	Divide halfword vectors using VM	VPC, VPD
div.l Sj,Sk	Divide scalar/scalar integer longword	SFU, ASP, IPP
div.l Vi,Sj,Vk	Divide vector/scalar integer longword	VPC, VPD, ASP
div.l Vi,Vj,Vk	Divide vector/vector integer longword	VPC, VPD_
div.l.f Vi,Sj,Vk	Divide vector/scalar longword using not VM	VPC, VPD, ASP
div.l.f Vi,Vj,Vk	Divide longword vectors using not VM	VPC, VPD_
div.l.t Vi,Sj,Vk	Divide vector/scalar longword using VM	VPC, VPD, ASP
div.l.t Vi,Vj,Vk	Divide longword vectors using VM	VPC, VPD_
div.s #N,Sk	Divide scalar/scalar single float	SFU, ASP, IPP
div.s Si,Vj,Vk	Divide scalar/vector single float	VPC, VPD, ASP
div.s Sj,Sk	Divide scalar/scalar single float	SFU, ASP, IPP
div.s Vi,Sj,Vk	Divide vector/scalar single float	VPC, VPD, ASP
div.s Vi,Vj,Vk	Divide vector/vector single float	VPC, VPD_
div.s.f Si,Vj,Vk	Divide scalar/vector single using not VM	VPC, VPD, ASP
div.s.f Vi,Sj,Vk	Divide vector/scalar single using not VM	VPC, VPD, ASP
div.s.f Vi,Vj,Vk	Divide single vectors using not VM	VPC, VPD_
div.s.t Si,Vj,Vk	Divide scalar/vector single using VM	VPC, VPD, ASP

div.s.t Vi,Sj,Vk	Divide vector/scalar single using VM	VPC, VPD, ASP
div.s.t Vi,Vj,Vk	Divide single vectors using VM	VPC, VPD_
div.w #N,Ak	Divide immediate address word	SFU, ASP, IPP
div.w #N,Sk	Divide scalar/scalar integer word	SFU, ASP, IPP
div.w #n,Ak	Divide short immediate address word	SFU, ASP, IPP
div.w Aj,Ak	Divide address register word	SFU, ASP, IPP
div.w Sj,Sk	Divide scalar/scalar integer word	SFU, ASP, IPP
div.w Vi,Sj,Vk	Divide vector/scalar integer word	VPC, VPD, ASP
div.w Vi,Vj,Vk	Divide vector/vector integer word	VPC, VPD_
div.w.f Vi,Sj,Vk	Divide vector/scalar word using not VM	VPC, VPD, ASP
div.w.f Vi,Vj,Vk	Divide word vectors using not VM	VPC, VPD_
div.w.t Vi,Sj,Vk	Divide vector/scalar word using VM	VPC, VPD, ASP
div.w.t Vi,Vj,Vk	Divide word vectors using VM	VPC, VPD_
dsi	Disable interrupts; reset ION to 0	ASP, CPX, IPP
enag Sj,Sk	Enable all global CPU interrupts	ASP, CPX, IPP
enal Sj,Sk	Enable local CPU interrupt	ASP, CPX, IPP
eni	Enable interrupts; set ION to 1	ASP, CPX, IPP
eq.b Sj,Sk	Compare equal byte	ASP
eq.b Sj,Vk	Compare equal byte	VPC, VPD, ASP
eq.b Vj,Vk	Compare equal byte	VPC, VPD
eq.b.f Sj,Vk	Cmp. equal byte using not VM	VPC, VPD, ASP
eq.b.f Vj,Vk	Cmp. equal byte using not VM	VPC, VPD
eq.b.t Sj,Vk	Cmp. equal byte using VM	VPC, VPD, ASP
eq.b.t Vj,Vk	Cmp. equal byte using VM	VPC, VPD
eq.d Sj,Sk	Compare equal double float	SFU, ASP, IPP
eq.d Sj,Vk	Compare equal double precision	VPC, VPD, ASP
eq.d Vj,Vk	Compare equal double precision	VPC, VPD
eq.d.f Sj,Vk	Cmp. equal double using not VM	VPC, VPD, ASP
eq.d.f Vj,Vk	Cmp. equal double using not VM	VPC, VPD
eq.d.t Sj,Vk	Cmp. equal double using VM	VPC, VPD, ASP
eq.d.t Vj,Vk	Cmp. equal double using VM	VPC, VPD
eq.h #N,Ak	Compare equal halfword	ASP, IPP
eq.h #N,Sk	Compare equal halfword	ASP, IPP
eq.h #n,Ak	Compare equal halfword	ASP, IPP
eq.h Aj,Ak	Compare equal halfword	ASP
eq.h Sj,Sk	Compare equal halfword	ASP
eq.h Sj,Vk	Compare equal halfword	VPC, VPD, ASP
eq.h Vj,Vk	Compare equal halfword	VPC, VPD
eq.h.f Sj,Vk	Cmp. equal halfword using not VM	VPC, VPD, ASP
eq.h.f Vj,Vk	Cmp. equal halfword using not VM	VPC, VPD
eq.h.t Sj,Vk	Cmp. equal halfword using VM	VPC, VPD, ASP
eq.h.t Vj,Vk	Cmp. equal halfword using VM	VPC, VPD
eq.l Sj,Sk	Compare equal longword	ASP
eq.l Sj,Vk	Compare equal longword	VPC, VPD, ASP
eq.l Vj,Vk	Compare equal longword	VPC, VPD

eq.l.f Sj,Vk	Cmp. equal long using not VM	VPC, VPD, ASP
eq.l.f Vj,Vk	Cmp. equal long using not VM	VPC, VPD
eq.l.t Sj,Vk	Cmp. equal long using VM	VPC, VPD, ASP
eq.l.t Vj,Vk	Cmp. equal long using VM	VPC, VPD
eq.s Sj,Sk	Compare equal single float	SFU, ASP, IPP
eq.s Sj,Vk	Compare equal single	VPC, VPD, ASP
eq.s Vj,Vk	Compare equal single	VPC, VPD
eq.s.f Sj,Vk	Cmp. equal single using not VM	VPC, VPD, ASP
eq.s.f Vj,Vk	Cmp. equal single using not VM	VPC, VPD
eq.s.t Sj,Vk	Cmp. equal single using VM	VPC, VPD, ASP
eq.s.t Vj,Vk	Cmp. equal single using VM	VPC, VPD
eq.w #N,Ak	Compare equal word	ASP, IPP
eq.w #N,Sk	Compare equal word	ASP, IPP
eq.w #n,Ak	Compare equal word	ASP, IPP
eq.w Aj,Ak	Compare equal word	ASP
eq.w Sj,Sk	Compare equal word	ASP
eq.w Sj,Vk	Compare equal word	VPC, VPD, ASP
eq.w Vj,Vk	Compare equal word	VPC, VPD
eq.w.f Sj,Vk	Cmp. equal word using not VM	VPC, VPD, ASP
eq.w.f Vj,Vk	Cmp. equal word using not VM	VPC, VPD
eq.w.t Sj,Vk	Cmp. equal word using VM	VPC, VPD, ASP
eq.w.t Vj,Vk	Cmp. equal word using VM	VPC, VPD
exit	Error exit instruction	DCU, SFU, ASP, MCM
exp.d Sk	Exponent of a double precision number	SFU, ASP, IPP
exp.s Sk	Exponent of a single precision number	SFU, ASP, IPP
frint.d Sj,Sk	Integerize float double scalar	SFU, ASP, IPP
frint.d Vj,Vk	Integerize float double vector	VPC, VPD
frint.d.f Vj,Vk	Integerize double vector using not VM	VPC, VPD
frint.d.t Vj,Vk	Integerize double vector using VM	VPC, VPD
frint.s Sj,Sk	Integerize float single scalar	SFU, ASP, IPP
frint.s Vj,Vk	Integerize float single vector	VPC, VPD
frint.s.f Vj,Vk	Integerize single vector using not VM	VPC, VPD
frint.s.t Vj,Vk	Integerize single vector using VM	VPC, VPD
get.l <Ceffa>,Sk	Get communication/scalar	ASP, CPX, IPP
get.w <Ceffa>,Ak	Get communication/address	ASP, CPX, IPP
halt #N,Ak	Halt the CPU	ASP, IPP
idle	Idle the CPU	ASP, CPX, IPP
inc.l <Ceffa>,Sk	Increment communication/scalar	ASP, CPX, IPP
inc.w <Ceffa>,Ak	Increment communication/address	ASP, CPX, IPP
incr.l <effa>,Sk	Increment long resource structure	DCU, SFU, ASP, MCM
incr.w <effa>,Ak	Increment resource structure data	DCU, SFU, ASP, MCM
jmp <effa>	Jump always	IPP, ASP
jmpa.f <effa>	Jump on address carry false	IPP, ASP
jmpa.t <effa>	Jump on address carry true	IPP, ASP
jmpi.f <effa>	Jump on ION false	IPP, ASP

jmpi.t <effa>	Jump on ION true	IPP, ASP
jmps.f <effa>	Jump on scalar carry false	IPP, ASP
jmps.t <effa>	Jump on scalar carry true	IPP, ASP
join	Join all threads	ASP, CPX, IPP
lck <Ceffa>	Lock communication register	ASP, CPX, IPP
ld.b <effa>,Ak	Load address register byte	DCU, SFU, ASP, MCM
ld.b <effa>,Sk	Load scalar byte	DCU, SFU, ASP, MCM
ld.b <effa>,Vk	Load vector byte	VPC, VPD, DCU, SFU, ASP, MCM
ld.b.f <effa>,Vk	Load vector byte using not VM	VPC, VPD, DCU, SFU, ASP, MCM
ld.b.t <effa>,Vk	Load vector byte using VM	VPC, VPD, DCU, SFU, ASP, MCM
ld.d #N,Sk	Load immediate upper 32 bits	ASP, IPP
ld.d <effa>,Sk	Load scalar double float	DCU, SFU, ASP, MCM
ld.d <effa>,Vk	Load vector double float	VPC, VPD, DCU, SFU, ASP, MCM
ld.d.f <effa>,Vk	Load vector double float using not VM	VPC, VPD, DCU, SFU, ASP, MCM
ld.d.t <effa>,Vk	Load vector double float using VM	VPC, VPD, DCU, SFU, ASP, MCM
ld.dl #N,Sk	Load 64-bit floating immediate, lower half	ASP, IPP
ld.du #N,Sk	Load 64-bit floating immediate, upper half	ASP, IPP
ld.h #N,Ak	Load halfword immediate into Ak	ASP, IPP
ld.h #n,Ak	Load short immediate into Ak	ASP, IPP
ld.h <effa>,Ak	Load address register halfword	DCU, SFU, ASP, MCM
ld.h <effa>,Sk	Load scalar halfword	DCU, SFU, ASP, MCM
ld.h <effa>,Vk	Load vector halfword	VPC, VPD, DCU, SFU, ASP, MCM
ld.h.f <effa>,Vk	Load vector halfword using not VM	VPC, VPD, DCU, SFU, ASP, MCM
ld.h.t <effa>,Vk	Load vector halfword using VM	VPC, VPD, DCU, SFU, ASP, MCM
ld.l #N,Sk	Load 32-bit immediate sign-extended to 64 bits	ASP, IPP
ld.l <effa>,Sk	Load scalar longword	DCU, SFU, ASP, MCM
ld.l <effa>,VLS	Load VS and VL from memory	ASP, DCU, SFU, VPC, MCM
ld.l <effa>,Vk	Load vector longword	VPC, VPD, DCU, SFU, ASP, MCM
ld.l.f <effa>,Vk	Load vector longword using not VM	VPC, VPD, DCU, SFU, ASP, MCM
ld.l.t <effa>,Vk	Load vector longword using VM	VPC, VPD, DCU, SFU, ASP, MCM
ld.ll #N,Sk	Load 64-bit integer immediate, lower half	ASP, IPP
ld.lu #N,Sk	Load 64-bit integer immediate, upper half	ASP, IPP
ld.s <effa>,Sk	Load scalar single float	DCU, SFU, ASP, MCM
ld.s <effa>,Vk	Load vector single float	VPC, VPD, DCU, SFU, ASP, MCM
ld.s.f <effa>,Vk	Load vector single float using not VM	VPC, VPD, DCU, SFU, ASP, MCM
ld.s.t <effa>,Vk	Load vector single float using VM	VPC, VPD, DCU, SFU, ASP, MCM
ld.u #N,Sk	Load immediate, upper half	ASP, IPP
ld.w #N,Ak	Load immediate into Ak	ASP, IPP
ld.w #N,Sk	Load a 32-bit immediate	ASP, IPP
ld.w #N,VL	Load VL with an immediate	ASP, VPC, SFU, IPP
ld.w #N,VS	Load VS from an immediate	ASP, SFU, IPP
ld.w #n,Ak	Load short immediate into Ak	ASP, IPP
ld.w <effa>,Ak	Load address register word	DCU, SFU, ASP, MCM
ld.w <effa>,Sk	Load scalar word	DCU, SFU, ASP, MCM
ld.w <effa>,Vk	Load vector word	VPC, VPD, DCU, SFU, ASP, MCM

ld.w.f <effa>,Vk	Load vector word using not VM	VPC, VPD, DCU, SFU, ASP, MCM
ld.w.t <effa>,Vk	Load vector word using VM	VPC, VPD, DCU, SFU, ASP, MCM
ld.x <effa>, VM	Load VM from memory	VPC, DCU, SFU, ASP, MCM
ldcmr <effa>,Ak	Load communication registers	DCU, SFU, ASP, MCM
ldea <effa>,Ak	Load effective address	ASP
ldea <effa>,Sk	Load effective address/scalar	ASP
ldkdr Ak	Load all eight SDRs	DCU, SFU, ASP, MCM
ldpa Aj,Ak	Load a physical byte address into Ak	DCU, SFU, ASP, MCM
ldsdr Ak	Load process SDRs	DCU, SFU, ASP, MCM
ldvi.b Vj,Vk	Index load vector byte	VPC, VPD, ASP, DCU, SFU, MCM
ldvi.b.f Vj,Vk	Index Load vector byte using not VM	VPC, VPD, ASP, DCU, SFU, MCM
ldvi.b.t Vj,Vk	Index Load vector byte using VM	VPC, VPD, ASP, DCU, SFU, MCM
ldvi.d Vj,Vk	Index load vector double float	VPC, VPD, ASP, DCU, SFU, MCM
ldvi.d.f Vj,Vk	Index Load vector double using not VM	VPC, VPD, ASP, DCU, SFU, MCM
ldvi.d.t Vj,Vk	Index Load vector double using VM	VPC, VPD, ASP, DCU, SFU, MCM
ldvi.h Vj,Vk	Index load vector halfword	VPC, VPD, ASP, DCU, SFU, MCM
ldvi.h.f Vj,Vk	Index Load vector halfword using not VM	VPC, VPD, ASP, DCU, SFU, MCM
ldvi.h.t Vj,Vk	Index Load vector halfword using VM	VPC, VPD, ASP, DCU, SFU, MCM
ldvi.l Vj,Vk	Index load vector longword	VPC, VPD, ASP, DCU, SFU, MCM
ldvi.l.f Vj,Vk	Index Load vector longword using not VM	VPC, VPD, ASP, DCU, SFU, MCM
ldvi.l.t Vj,Vk	Index Load vector longword using VM	VPC, VPD, ASP, DCU, SFU, MCM
ldvi.s Vj,Vk	Index load vector single float	VPC, VPD, ASP, DCU, SFU, MCM
ldvi.s.f Vj,Vk	Index Load vector single using not VM	VPC, VPD, ASP, DCU, SFU, MCM
ldvi.s.t Vj,Vk	Index Load vector single using VM	VPC, VPD, ASP, DCU, SFU, MCM
ldvi.w Vj,Vk	Index load vector word	VPC, VPD, ASP, DCU, SFU, MCM
ldvi.w.f Vj,Vk	Index Load vector word using not VM	VPC, VPD, ASP, DCU, SFU, MCM
ldvi.w.t Vj,Vk	Index Load vector word using VM	VPC, VPD, ASP, DCU, SFU, MCM
le.b Sj,Sk	Compare less than or equal byte	ASP
le.b Sj,Vk	Compare less than or equal byte	VPC, VPD, ASP
le.b Vj,Vk	Compare less than or equal byte	VPC, VPD
le.b.f Sj,Vk	Cmp. less than or equal byte using not VM	VPC, VPD, ASP
le.b.f Vj,Vk	Cmp. less than or equal byte using not VM	VPC, VPD
le.b.t Sj,Vk	Cmp. less than or equal byte using VM	VPC, VPD, ASP
le.b.t Vj,Vk	Cmp. less than or equal byte using VM	VPC, VPD
le.d Sj,Sk	Compare less than or equal double float	SFU, ASP, IPP
le.d Sj,Vk	Compare less than or equal double float	VPC, VPD, ASP
le.d Vj,Vk	Compare less than or equal double float	VPC, VPD
le.d.f Sj,Vk	Cmp. less than or equal double using not VM	VPC, VPD, ASP
le.d.f Vj,Vk	Cmp. less than or equal double using not VM	VPC, VPD
le.d.t Sj,Vk	Cmp. less than or equal double using VM	VPC, VPD, ASP
le.d.t Vj,Vk	Cmp. less than or equal double using VM	VPC, VPD
le.h #N,Ak	Compare less than or equal halfword	ASP, IPP
le.h #N,Sk	Compare less than or equal halfword	ASP, IPP
le.h #n,Ak	Compare less than or equal halfword	ASP, IPP
le.h Aj,Ak	Compare less than or equal signed halfword	ASP

le.h Sj,Sk	Compare less than or equal halfword	ASP
le.h Sj,Vk	Compare less than or equal halfword	VPC, VPD, ASP
le.h Vj,Vk	Compare less than or equal halfword	VPC, VPD
le.h.f Sj,Vk	Cmp. less than or equal half using not VM	VPC, VPD, ASP
le.h.f Vj,Vk	Cmp. less than or equal half using not VM	VPC, VPD
le.h.t Sj,Vk	Cmp. less than or equal half using VM	VPC, VPD, ASP
le.h.t Vj,Vk	Cmp. less than or equal half using VM	VPC, VPD
le.l Sj,Sk	Compare less than or equal longword	ASP
le.l Sj,Vk	Compare less than or equal longword	VPC, VPD, ASP
le.l Vj,Vk	Compare less than or equal longword	VPC, VPD
le.l.f Sj,Vk	Cmp. less than or equal long using not VM	VPC, VPD, ASP
le.l.f Vj,Vk	Cmp. less than or equal long using not VM	VPC, VPD
le.l.t Sj,Vk	Cmp. less than or equal long using VM	VPC, VPD, ASP
le.l.t Vj,Vk	Cmp. less than or equal long using VM	VPC, VPD
le.s #N,Sk	Compare less than or equal single	SFU, ASP, IPP
le.s Sj,Sk	Compare less than or equal single float	SFU, ASP, IPP
le.s Sj,Vk	Compare less than or equal single	VPC, VPD, ASP
le.s Vj,Vk	Compare less than or equal single	VPC, VPD
le.s.f Sj,Vk	Cmp. less than or equal single using not VM	VPC, VPD, ASP
le.s.f Vj,Vk	Cmp. less than or equal single using not VM	VPC, VPD
le.s.t Sj,Vk	Cmp. less than or equal single using VM	VPC, VPD, ASP
le.s.t Vj,Vk	Cmp. less than or equal single using VM	VPC, VPD
le.w #N,Ak	Compare less than or equal word	ASP, IPP
le.w #N,Sk	Compare less than or equal word	ASP, IPP
le.w #n,Ak	Compare less than or equal word	ASP, IPP
le.w Aj,Ak	Compare less than or equal signed word	ASP
le.w Sj,Sk	Compare less than or equal word	ASP
le.w Sj,Vk	Compare less than or equal word	VPC, VPD, ASP
le.w Vj,Vk	Compare less than or equal word	VPC, VPD
le.w.f Sj,Vk	Cmp. less than or equal word using not VM	VPC, VPD, ASP
le.w.f Vj,Vk	Cmp. less than or equal word using not VM	VPC, VPD
le.w.t Sj,Vk	Cmp. less than or equal word using VM	VPC, VPD, ASP
le.w.t Vj,Vk	Cmp. less than or equal word using VM	VPC, VPD
leu.b Sj,Sk	Compare less than or equal to byte	ASP
leu.h #N,Ak	Compare unsigned less than halfword	ASP, IPP
leu.h #N,Sk	Compare unsigned less than or equal to halfword	ASP, IPP
leu.h #n,Ak	Compare unsigned less than or equal halfword	ASP, IPP
leu.h Aj,Ak	Compare unsigned less than or equal to halfword	ASP
leu.h Sj,Sk	Compare less than or equal to halfword	ASP
leu.l Sj,Sk	Compare less than or equal to longword	ASP
leu.w #N,Ak	Compare unsigned less than word	ASP, IPP
leu.w #N,Sk	Compare unsigned less than or equal to word	ASP, IPP
leu.w #n,Ak	Compare unsigned less than or equal word	ASP, IPP
leu.w Aj,Ak	Compare unsigned less than or equal to word	ASP
leu.w Sj,Sk	Compare less than or equal to word	ASP

ln.d Sk	Natural logarithm of a double precision number	SFU, ASP, IPP
ln.s Sk	Natural logarithm of a single precision number	SFU, ASP, IPP
lop Sj,Sk	Count of leading zeros in Sj	SFU, ASP, IPP
lop Vj,Vk	Leading ones position vector	VPC, VPD
lop.f Vj,Vk	Leading ones position vector using not VM	VPC, VPD
lop.t Vj,Vk	Leading ones position vector using VM	VPC, VPD
lt.b Sj,Sk	Compare less than byte	ASP
lt.b Sj,Vk	Compare less than byte	VPC, VPD, ASP
lt.b Vj,Vk	Compare less than byte	VPC, VPD
lt.b.f Sj,Vk	Cmp. less than byte using not VM	VPC, VPD, ASP
lt.b.f Vj,Vk	Cmp. less than byte using not VM	VPC, VPD
lt.b.t Sj,Vk	Cmp. less than byte using VM	VPC, VPD, ASP
lt.b.t Vj,Vk	Cmp. less than byte using VM	VPC, VPD
lt.d Sj,Sk	Compare less than double float	SFU, ASP, IPP
lt.d Sj,Vk	Compare less than double float	VPC, VPD, ASP
lt.d Vj,Vk	Compare less than double float	VPC, VPD
lt.d.f Sj,Vk	Cmp. less than double using not VM	VPC, VPD, ASP
lt.d.f Vj,Vk	Cmp. less than double using not VM	VPC, VPD
lt.d.t Sj,Vk	Cmp. less than double using VM	VPC, VPD, ASP
lt.d.t Vj,Vk	Cmp. less than double using VM	VPC, VPD
lt.h #N,Ak	Compare less than halfword	ASP, IPP
lt.h #N,Sk	Compare less than halfword	ASP, IPP
lt.h #n,Ak	Compare less than halfword	ASP, IPP
lt.h Aj,Ak	Compare less than signed halfword	ASP
lt.h Sj,Sk	Compare less than halfword	ASP
lt.h Sj,Vk	Compare less than halfword	VPC, VPD, ASP
lt.h Vj,Vk	Compare less than halfword	VPC, VPD
lt.h.f Sj,Vk	Cmp. less than halfword using not VM	VPC, VPD, ASP
lt.h.f Vj,Vk	Cmp. less than halfword using not VM	VPC, VPD
lt.h.t Sj,Vk	Cmp. less than halfword using VM	VPC, VPD, ASP
lt.h.t Vj,Vk	Cmp. less than halfword using VM	VPC, VPD
lt.l Sj,Sk	Compare less than longword	ASP
lt.l Sj,Vk	Compare less than longword	VPC, VPD, ASP
lt.l Vj,Vk	Compare less than longword	VPC, VPD
lt.l.f Sj,Vk	Cmp. less than long using not VM	VPC, VPD, ASP
lt.l.f Vj,Vk	Cmp. less than long using not VM	VPC, VPD
lt.l.t Sj,Vk	Cmp. less than long using VM	VPC, VPD, ASP
lt.l.t Vj,Vk	Cmp. less than long using VM	VPC, VPD
lt.s #N,Sk	Compare less than single	SFU, ASP, IPP
lt.s Sj,Sk	Compare less than single float	SFU, ASP, IPP
lt.s Sj,Vk	Compare less than single	VPC, VPD, ASP
lt.s Vj,Vk	Compare less than single	VPC, VPD
lt.s.f Sj,Vk	Cmp. less than single using not VM	VPC, VPD, ASP
lt.s.f Vj,Vk	Cmp. less than single using not VM	VPC, VPD
lt.s.t Sj,Vk	Cmp. less than single using VM	VPC, VPD, ASP

lt.s.t Vj,Vk	Cmp. less than single using VM	VPC, VPD
lt.w #N,Ak	Compare less than word	ASP, IPP
lt.w #N,Sk	Compare less than word	ASP, IPP
lt.w #n,Ak	Compare less than word	ASP, IPP
lt.w Aj,Ak	Compare less than signed word	ASP
lt.w Sj,Sk	Compare less than word	ASP
lt.w Sj,Vk	Compare less than word	VPC, VPD, ASP
lt.w Vj,Vk	Compare less than word	VPC, VPD
lt.w.f Sj,Vk	Cmp. less than word using not VM	VPC, VPD, ASP
lt.w.f Vj,Vk	Cmp. less than word using not VM	VPC, VPD
lt.w.t Sj,Vk	Cmp. less than word using VM	VPC, VPD, ASP
lt.w.t Vj,Vk	Cmp. less than word using VM	VPC, VPD
ltu.b Sj,Sk	Compare less than byte	ASP
ltu.h #N,Ak	Compare unsigned less than halfword	ASP, IPP
ltu.h #N,Sk	Compare unsigned less than halfword	ASP, IPP
ltu.h #n,Ak	Compare unsigned less than halfword	ASP, IPP
ltu.h Aj,Ak	Compare unsigned less than halfword	ASP
ltu.h Sj,Sk	Compare less than halfword	ASP
ltu.l Sj,Sk	Compare less than longword	ASP
ltu.w #N,Ak	Compare unsigned less than word	ASP, IPP
ltu.w #N,Sk	Compare unsigned less than word	ASP, IPP
ltu.w #n,Ak	Compare unsigned less than word	ASP, IPP
ltu.w Aj,Ak	Compare unsigned less than word	ASP
ltu.w Sj,Sk	Compare less than word	ASP
mask.f Vi,Sj,Vk	Mask vector/scalar using not VM	VPC, VPD, ASP
mask.t Vi,Sj,Vk	Mask vector/scalar using VM	VPC, VPD, ASP
mask.t Vi,Vj,Vk	Mask vector/vector	VPC, VPD_
mat.l Sk,<Ceffa>	Match scalar/communication	ASP, CPX, IPP
mat.w Ak,<Ceffa>	Match address/communication	ASP, CPX, IPP
matm.l Sk,<effa>	Match scalar register and memory	DCU, SFU, ASP, MCM
matm.w Ak,<effa>	Match address register and memory	DCU, SFU, ASP, MCM
max.b Vk	Max of a vector of bytes	VPC, VPD, ASP
max.b.f Vk	Max of vector of bytes using not VM	VPC, VPD, ASP
max.b.t Vk	Max of vector of bytes using VM	VPC, VPD, ASP
max.d Vk	Max of a vector of double float	VPC, VPD, ASP
max.d.f Vk	Max of vector of doubles using not VM	VPC, VPD, ASP
max.d.t Vk	Max of vector of doubles using VM	VPC, VPD, ASP
max.h Vk	Max of a vector of halfwords	VPC, VPD, ASP
max.h.f Vk	Max of vector of halfwords using not VM	VPC, VPD, ASP
max.h.t Vk	Max of vector of halfwords using VM	VPC, VPD, ASP
max.l Vk	Max of a vector of longwords	VPC, VPD, ASP
max.l.f Vk	Max of vector of longwords using not VM	VPC, VPD, ASP
max.l.t Vk	Max of vector of longwords using VM	VPC, VPD, ASP
max.s Vk	Max of a vector of single float	VPC, VPD, ASP
max.s.f Vk	Max of vector of singles using not VM	VPC, VPD, ASP

max.s.t Vk	Max of vector of singles using VM	VPC, VPD, ASP
max.w Vk	Max of a vector of words	VPC, VPD, ASP
max.w.f Vk	Max of vector of words using not VM	VPC, VPD, ASP
max.w.t Vk	Max of vector of words using VM	VPC, VPD, ASP
merg.f Vi,Sj,Vk	Merge vector/scalar using not VM	VPC, VPD, ASP
merg.t Vi,Sj,Vk	Merge vector/scalar	VPC, VPD, ASP
merg.t Vi,Vj,Vk	Merge vector/vector	VPC, VPD_
min.b Vk	Min of a vector of bytes	VPC, VPD, ASP
min.b.f Vk	Min of vector of bytes using not VM	VPC, VPD, ASP
min.b.t Vk	Min of vector of bytes using VM	VPC, VPD, ASP
min.d Vk	Min of a vector of double float	VPC, VPD, ASP
min.d.f Vk	Min of vector of doubles using not VM	VPC, VPD, ASP
min.d.t Vk	Min of vector of doubles using VM	VPC, VPD, ASP
min.h Vk	Min of a vector of halfwords	VPC, VPD, ASP
min.h.f Vk	Min of vector of halfwords using not VM	VPC, VPD, ASP
min.h.t Vk	Min of vector of halfwords using VM	VPC, VPD, ASP
min.l Vk	Min of a vector of longwords	VPC, VPD, ASP
min.l.f Vk	Min of vector of longwords using not VM	VPC, VPD, ASP
min.l.t Vk	Min of vector of longwords using VM	VPC, VPD, ASP
min.s Vk	Min of a vector of single float	VPC, VPD, ASP
min.s.f Vk	Min of vector of singles using not VM	VPC, VPD, ASP
min.s.t Vk	Min of vector of singles using VM	VPC, VPD, ASP
min.w Vk	Min of a vector of words	VPC, VPD, ASP
min.w.f Vk	Min of vector of words using not VM	VPC, VPD, ASP
min.w.t Vk	Min of vector of words using VM	VPC, VPD, ASP
mov Aj,Ak	Move address register	ASP
mov Aj,Sk	Move an address to a scalar	ASP
mov Ak,PSW	Load an address register into the PSW	ASP, SFU, VPC
mov Ak,VL	Move Ak to VL	ASP, VPC, SFU
mov Ak,VS	Move Ak to VS	ASP, SFU
mov CIR,Sk	Move Comm Index Register to a scalar	ASP
mov CPUID,Sk	Move CPU identification to Scalar	ASP
mov ICR,Sk	Move Interrupt Control Register to Scalar	ASP, CPX, IPP
mov ITR,Sk	Move the ITC, ITSR, NITC into Sk	ASP, CPX, IPP
mov PC,Ak	Load next PC address	ASP, IPP
mov PSW,Ak	Store the PSW into an address register	ASP, SFU, VPC
mov Si,Sj,Vk	Move a scalar to a vector element	VPC, VPD, ASP
mov Sj,Ak	Move 32 bits of Sj into Ak	ASP
mov Sj,Sk,VM	Load VM(Sj) from Sk	VPC, ASP
mov Sj,VM,Sk	Load Sk from VM(Sj)	VPC, VPD, ASP
mov Sk,CIR	Move scalar to Comm Index Register	ASP, DCU
mov Sk,ICR	Move Scalar to Interrupt Control Register	ASP, CPX, IPP
mov Sk,ITR	Load NITC, ITC, ITSR from Sk	ASP, CPX, IPP
mov Sk,ITSR	Load ITSR with a scalar	ASP, CPX, IPP
mov Sk,TCPU	Move Scalar to Target CPU Register	ASP, CPX, IPP

mov Sk,TID	Load Thread ID from Scalar	ASP, DCU
mov Sk,TTR	Move scalar to thread timer	ASP, CPX, IPP
mov Sk,VML	Load VM<63..0> from Sk	VPC, ASP
mov Sk,VMU	Load VM<127..64> from Sk	VPC, ASP
mov Sk,VV	Move scalar to vector valid flag	ASP, IPP
mov TCPU,Sk	Move the Target CPU Register to Scalar	ASP, CPX, IPP
mov TID,Sk	Load scalar with Thread ID	ASP
mov TOC,Sk	Move TOC to a scalar	ASP, CPX, IPP
mov TTR,Sk	Move thread timer/scalar	ASP, CPX, IPP
mov VL,Ak	Move VL to Ak	ASP
mov VML,Sk	Load Sk from VM<63..0>	VPC, VPD, ASP
mov VMU,Sk	Load Sk from VM<127..64>	VPC, VPD, ASP
mov VS,Ak	Move VS to Ak	ASP
mov Vi,Sj,Sk	Move a vector element to a scalar	VPC, VPD, ASP
mov.d Sj,Sk	Move scalar register single float	ASP
mov.l Sj,Sk	Move scalar register longword	ASP
mov.s Sj,Sk	Move scalar register double float	ASP
mov.w Sj,Sk	Move scalar register word	ASP
mov.w Sk,VL	Move Sk to VL	ASP, VPC, SFU
mov.w Sk,VS	Move Sk to VS	ASP, SFU
mov.w VL,Sk	Move VL to Sk	ASP
mov.w VS,Sk	Move VS to Sk	ASP
mski Sk	Mask out interrupt	ASP, CPX, IPP
msync	Synchronize stores to memory	ASP, DCU, MCM
mul.b Sj,Sk	Multiply scalar/scalar integer byte	SFU, ASP, IPP
mul.b Vi,Sj,Vk	Multiply vector/scalar integer byte	VPC, VPD, ASP
mul.b Vi,Vj,Vk	Multiply vector/vector integer byte	VPC, VPD_
mul.b.f Vi,Sj,Vk	Multiply vector/scalar byte using not VM	VPC, VPD, ASP
mul.b.f Vi,Vj,Vk	Multiply byte vectors using not VM	VPC, VPD_
mul.b.t Vi,Sj,Vk	Multiply vector/scalar byte using VM	VPC, VPD, ASP
mul.b.t Vi,Vj,Vk	Multiply byte vectors using VM	VPC, VPD_
mul.d Sj,Sk	Multiply scalar/scalar double float	SFU, ASP, IPP
mul.d Vi,Sj,Vk	Multiply vector/scalar double float	VPC, VPD, ASP
mul.d Vi,Vj,Vk	Multiply vector/vector double float	VPC, VPD_
mul.d.f Vi,Sj,Vk	Multiply vector/scalar double using not VM	VPC, VPD, ASP
mul.d.f Vi,Vj,Vk	Multiply double vectors using not VM	VPC, VPD_
mul.d.t Vi,Sj,Vk	Multiply vector/scalar double using VM	VPC, VPD, ASP
mul.d.t Vi,Vj,Vk	Multiply double vectors using VM	VPC, VPD_
mul.h #N,Ak	Multiply immediate address halfword	SFU, ASP, IPP
mul.h #N,Sk	Multiply scalar/immediate integer halfword	SFU, ASP, IPP
mul.h #n,Ak	Multiply short immediate address halfword	SFU, ASP, IPP
mul.h Aj,Ak	Multiply address register halfword	SFU, ASP, IPP
mul.h Sj,Sk	Multiply scalar/scalar integer halfword	SFU, ASP, IPP
mul.h Vi,Sj,Vk	Multiply vector/scalar integer halfword	VPC, VPD, ASP
mul.h Vi,Vj,Vk	Multiply vector/vector integer halfword	VPC, VPD_

mul.h.f Vi,Sj,Vk	Multiply vector/scalar halfword using not VM	VPC, VPD, ASP
mul.h.f Vi,Vj,Vk	Multiply halfword vectors using not VM	VPC, VPD_
mul.h.t Vi,Sj,Vk	Multiply vector/scalar halfword using VM	VPC, VPD, ASP
mul.h.t Vi,Vj,Vk	Multiply halfword vectors using VM	VPC, VPD_
mul.l Sj,Sk	Multiply scalar/scalar integer longword	SFU, ASP, IPP
mul.l Vi,Sj,Vk	Multiply vector/scalar integer longword	VPC, VPD, ASP
mul.l Vi,Vj,Vk	Multiply vector/vector integer longword	VPC, VPD_
mul.l.f Vi,Sj,Vk	Multiply vector/scalar longword using not VM	VPC, VPD, ASP
mul.l.f Vi,Vj,Vk	Multiply longword vectors using not VM	VPC, VPD_
mul.l.t Vi,Sj,Vk	Multiply vector/scalar longword using VM	VPC, VPD, ASP
mul.l.t Vi,Vj,Vk	Multiply longword vectors using VM	VPC, VPD_
mul.s #N,Sk	Multiply scalar/immediate single float	SFU, ASP, IPP
mul.s Sj,Sk	Multiply scalar/scalar single float	SFU, ASP, IPP
mul.s Vi,Sj,Vk	Multiply vector/scalar single float	VPC, VPD, ASP
mul.s Vi,Vj,Vk	Multiply vector/vector single float	VPC, VPD_
mul.s.f Vi,Sj,Vk	Multiply vector/scalar single using not VM	VPC, VPD, ASP
mul.s.f Vi,Vj,Vk	Multiply single vectors using not VM	VPC, VPD_
mul.s.t Vi,Sj,Vk	Multiply vector/scalar single using VM	VPC, VPD, ASP
mul.s.t Vi,Vj,Vk	Multiply single vectors using VM	VPC, VPD_
mul.w #N,Ak	Multiply immediate address word	SFU, ASP, IPP
mul.w #N,Sk	Multiply scalar/immediate integer word	SFU, ASP, IPP
mul.w #n,Ak	Multiply short immediate address word	SFU, ASP, IPP
mul.w Aj,Ak	Multiply address register word	SFU, ASP, IPP
mul.w Sj,Sk	Multiply scalar/scalar integer word	SFU, ASP, IPP
mul.w Vi,Sj,Vk	Multiply vector/scalar integer word	VPC, VPD, ASP
mul.w Vi,Vj,Vk	Multiply vector/vector integer word	VPC, VPD_
mul.w.f Vi,Sj,Vk	Multiply vector/scalar word using not VM	VPC, VPD, ASP
mul.w.f Vi,Vj,Vk	Multiply word vectors using not VM	VPC, VPD_
mul.w.t Vi,Sj,Vk	Multiply vector/scalar word using VM	VPC, VPD, ASP
mul.w.t Vi,Vj,Vk	Multiply word vectors using VM	VPC, VPD_
neg.b Sj,Sk	Negate scalar/scalar integer byte	ASP
neg.b Vj,Vk	Negate vector/vector integer byte	VPC, VPD
neg.b.f Vj,Vk	Negate byte vector using not VM	VPC, VPD
neg.b.t Vj,Vk	Negate byte vector using VM	VPC, VPD
neg.d Sj,Sk	Negate scalar/scalar double float	SFU, ASP, IPP
neg.d Vj,Vk	Negate vector/vector double float	VPC, VPD
neg.d.f Vj,Vk	Negate double using not VM	VPC, VPD
neg.d.t Vj,Vk	Negate double using VM	VPC, VPD
neg.h Aj,Ak	Negate address register halfword	ASP
neg.h Sj,Sk	Negate scalar/scalar integer halfword	ASP
neg.h Vj,Vk	Negate vector/vector integer halfword	VPC, VPD
neg.h.f Vj,Vk	Negate halfword using not VM	VPC, VPD
neg.h.t Vj,Vk	Negate halfword using VM	VPC, VPD
neg.l Sj,Sk	Negate scalar/scalar integer longword	ASP
neg.l Vj,Vk	Negate vector/vector integer longword	VPC, VPD

neg.l.f Vj,Vk	Negate longword using not VM	VPC, VPD
neg.l.t Vj,Vk	Negate longword using VM	VPC, VPD
neg.s Sj,Sk	Negate scalar/scalar single float	SFU, ASP, IPP
neg.s Vj,Vk	Negate vector/vector single float	VPC, VPD
neg.s.f Vj,Vk	Negate single using not VM	VPC, VPD
neg.s.t Vj,Vk	Negate single using VM	VPC, VPD
neg.w Aj,Ak	Negate address register word	ASP
neg.w Sj,Sk	Negate scalar/scalar integer word	ASP
neg.w Vj,Vk	Negate vector/vector integer word	VPC, VPD
neg.w.f Vj,Vk	Negate word using not VM	VPC, VPD
neg.w.t Vj,Vk	Negate word using VM	VPC, VPD
nop	No operation (branch never)	IPP
not Aj,Ak	Complement address register	ASP
not Sj,Sk	Complement scalar/scalar	ASP
not Vj,Vk	Complement a vector	VPC, VPD
not.f Vj,Vk	Complement a vector using not VM	VPC, VPD
not.t Vj,Vk	Complement a vector using VM	VPC, VPD
or #N,Ak	OR immediate to address register	ASP, IPP
or #N,Sk	OR scalar/immediate	ASP, IPP
or Aj,Ak	OR address register	ASP
or Sj,Sk	OR scalar/scalar	ASP
or Vi,Sj,Vk	OR vector/scalar	VPC, VPD, ASP
or Vi,Vj,Vk	OR two vectors	VPC, VPD_
or.f Vi,Sj,Vk	OR vector/scalar using not VM	VPC, VPD, ASP
or.f Vi,Vj,Vk	OR two vectors using not VM	VPC, VPD_
or.t Vi,Sj,Vk	OR vector/scalar using VM	VPC, VPD, ASP
or.t Vi,Vj,Vk	OR two vectors using VM	VPC, VPD_
parity Vk	Exclusive OR reduce a vector	VPC, VPD, ASP
parity.f Vk	Exclusive OR reduce vector using not VM	VPC, VPD, ASP
parity.t Vk	Exclusive OR reduce vector using VM	VPC, VPD, ASP
pate Ak	Purge ATU entry	DCU, ASP, IPP, SFU
patu	Purge the entire ATU	DCU, ASP
pbkpt	Force process breakpoint exception	DCU, SFU, ASP, MCM
pfork <effa>,Ak	Post a fork event	ASP, CPX, IPP
pich	Purge the Icache	ASP, IPP
plc.f VM,Sk	Load the number of 0's in VM into Sk	VPC, VPD, SFU, ASP
plc.t Sj,Sk	Count the number of 1's in Sj	SFU, ASP, IPP
plc.t VM,Sk	Load the number of 1's in VM into Sk	VPC, VPD, SFU, ASP
plc.t Vj,Vk	Population count of a vector	VPC, VPD
plc.t.f Vj,Vk	Population count of vector using not VM	VPC, VPD
plc.t.t Vj,Vk	Population count of vector using VM	VPC, VPD
plch	Purge the Lcache	ASP
pop.l Sk	Pop Sk <63..0> from the stack	DCU, SFU, ASP, MCM
pop.w Ak	Pop word into address register	DCU, SFU, ASP, MCM
pop.w Sk	Pop Sk <31..0> from the stack	DCU, SFU, ASP, MCM

popr Ak,<effa>	Pop resource/address register	DCU, SFU, ASP, MCM
prod.b Vk	Multiply reduce a vector of bytes	VPC, VPD, ASP, SFU, IPP
prod.b.f Vk	Multiply reduce byte vector using not VM	VPC, VPD, ASP, SFU, IPP
prod.b.t Vk	Multiply reduce byte vector using VM	VPC, VPD, ASP, SFU, IPP
prod.d Vk	Multiply reduce a vector of double float	VPC, VPD, ASP, SFU, IPP
prod.d.f Vk	Multiply reduce double vector using not VM	VPC, VPD, ASP, SFU, IPP
prod.d.t Vk	Multiply reduce double vector using VM	VPC, VPD, ASP, SFU, IPP
prod.h Vk	Multiply reduce a vector of halfwords	VPC, VPD, ASP, SFU, IPP
prod.h.f Vk	Multiply reduce halfword vector using not VM	VPC, VPD, ASP, SFU, IPP
prod.h.t Vk	Multiply reduce halfword vector using VM	VPC, VPD, ASP, SFU, IPP
prod.l Vk	Multiply reduce a vector of longwords	VPC, VPD, ASP, SFU, IPP
prod.l.f Vk	Multiply reduce longword vector using not VM	VPC, VPD, ASP, SFU, IPP
prod.l.t Vk	Multiply reduce longword vector using VM	VPC, VPD, ASP, SFU, IPP
prod.s Vk	Multiply reduce a vector of single float	VPC, VPD, ASP, SFU, IPP
prod.s.f Vk	Multiply reduce single vector using not VM	VPC, VPD, ASP, SFU, IPP
prod.s.t Vk	Multiply reduce single vector using VM	VPC, VPD, ASP, SFU, IPP
prod.w Vk	Multiply reduce a vector of words	VPC, VPD, ASP, SFU, IPP
prod.w.f Vk	Multiply reduce word vector using not VM	VPC, VPD, ASP, SFU, IPP
prod.w.t Vk	Multiply reduce word vector using VM	VPC, VPD, ASP, SFU, IPP
psh.l Sk	Push Sk<63..0> onto the stack	DCU, ASP, MCM
psh.w Ak	Push an address register	DCU, ASP, MCM
psh.w Sk	Push Sk<31..0> onto the stack	DCU, ASP, MCM
pshea <effa>	Push effective address	DCU, ASP, MCM
pshr Ak,<effa>	Push address register/resource	DCU, SFU, ASP, MCM
put.l Sk,<Ceffa>	Put scalar/communication	ASP, CPX, IPP
put.w Ak,<Ceffa>	Put address/communication	ASP, CPX, IPP
rcv.l <Ceffa>,Sk	Receive communication/scalar	ASP, CPX, IPP
rcv.w <Ceffa>,Ak	Receive communication/address	ASP, CPX, IPP
rcvr.l <effa>,Sk	Receive scalar register/resource	DCU, SFU, ASP, MCM
rcvr.w <effa>,Ak	Receive address register/resource	DCU, SFU, ASP, MCM
rtn	Return from subroutine call	DCU, SFU, ASP, MCM
rtnc	Return from a context block	DCU, SFU, ASP, MCM
rtni	Return from base level interrupt	DCU, SFU, ASP, MCM
rtnq	Pop the PC and jump	DCU, SFU, ASP, MCM
shf #N,Ak	Logical shift immediate to address register	SFU, ASP, IPP
shf #N,Sk	Shift scalar/immediate	SFU, ASP, IPP
shf #n,Ak	Logical shift short immediate	ASP, IPP
shf Aj,Ak	Shift an address	SFU, ASP, IPP
shf Sj,Sk	Shift a scalar	SFU, ASP, IPP
shf Sj,Vk	Shift a vector accumulator	VPC, VPD, ASP
shf Vi,Sj,Vk	Shift a vector accumulator	VPC, VPD, ASP
shf Vi,Vj,Vk	Shift vector/vector	VPC, VPD_
shf.f Sj,Vk	Shift vector/scalar using not VM	VPC, VPD, ASP
shf.f Vi,Sj,Vk	Shift vector/scalar using not VM	VPC, VPD, ASP
shf.f Vi,Vj,Vk	Shift vector/vector using not VM	VPC, VPD_

shf.t Sj,Vk	Shift vector/scalar using VM	VPC, VPD, ASP
shf.t Vi,Sj,Vk	Shift vector/scalar using VM	VPC, VPD, ASP
shf.t Vi,Vj,Vk	Shift vector/vector using VM	VPC, VPD_
shf.w #N,Sk	Shift Scalar word/immediate	SFU, ASP, IPP
shf.w Sj,Sk	Shift a scalar word	SFU, ASP, IPP
sin.d Sk	Sine of a double precision number	SFU, ASP, IPP
sin.s Sk	Sine of a single precision number	SFU, ASP, IPP
snd.l Sk,<Ceffa>	Send scalar/communication	ASP, CPX, IPP
snd.w Ak,<Ceffa>	Send address/communication	ASP, CPX, IPP
sndr.l Sk,<effa>	Send scalar register/resource	DCU, SFU, ASP, MCM
sndr.w Ak,<effa>	Send address register/resource	DCU, SFU, ASP, MCM
spawn <effa>,Ak	Spawn a fork event	ASP, CPX, IPP
sqrt.d Sk	Square root of a double precision number	SFU, ASP, IPP
sqrt.d Vj,Vk	Square root double vector/vector	VPC, VPD
sqrt.d.f Vj,Vk	Square root double using not VM	VPC, VPD
sqrt.d.t Vj,Vk	Square root double using VM	VPC, VPD
sqrt.s Sk	Square root of a single precision number	SFU, ASP, IPP
sqrt.s Vj,Vk	Square root single vector/vector	VPC, VPD
sqrt.s.f Vj,Vk	Square root single using not VM	VPC, VPD
sqrt.s.t Vj,Vk	Square root single using VM	VPC, VPD
st.b Ak,<effa>	Store address register byte	DCU, ASP, MCM
st.b Sk,<effa>	Store scalar byte	DCU, ASP, MCM
st.b Vk,<effa>	Store vector byte	VPC, VPD, DCU, SFU, ASP, MCM
st.b.f Vk,<effa>	Store vector byte using not VM	VPC, VPD, DCU, SFU, ASP, MCM
st.b.t Vk,<effa>	Store vector byte using VM	VPC, VPD, DCU, SFU, ASP, MCM
st.d Sk,<effa>	Store scalar double float	DCU, ASP, MCM
st.d Vk,<effa>	Store vector double float	VPC, VPD, DCU, SFU, ASP, MCM
st.d.f Vk,<effa>	Store vector double float using not VM	VPC, VPD, DCU, SFU, ASP, MCM
st.d.t Vk,<effa>	Store vector double float using VM	VPC, VPD, DCU, SFU, ASP, MCM
st.h Ak,<effa>	Store address register halfword	DCU, ASP, MCM
st.h Sk,<effa>	Store scalar halfword	DCU, ASP, MCM
st.h Vk,<effa>	Store vector halfword	VPC, VPD, DCU, SFU, ASP, MCM
st.h.f Vk,<effa>	Store vector halfword using not VM	VPC, VPD, DCU, SFU, ASP, MCM
st.h.t Vk,<effa>	Store vector halfword using VM	VPC, VPD, DCU, SFU, ASP, MCM
st.l Sk,<effa>	Store scalar longword	DCU, ASP, MCM
st.l VLS,<effa>	Store VS and VL to memory	VPC, VPD, DCU, SFU, ASP, MCM
st.l Vk,<effa>	Store vector longword	VPC, VPD, DCU, SFU, ASP, MCM
st.l.f Vk,<effa>	Store vector longword using not VM	VPC, VPD, DCU, SFU, ASP, MCM
st.l.t Vk,<effa>	Store vector longword using VM	VPC, VPD, DCU, SFU, ASP, MCM
st.s Sk,<effa>	Store scalar single float	DCU, ASP, MCM
st.s Vk,<effa>	Store vector single float	VPC, VPD, DCU, SFU, ASP, MCM
st.s.f Vk,<effa>	Store vector single float using not VM	VPC, VPD, DCU, SFU, ASP, MCM
st.s.t Vk,<effa>	Store vector single float using VM	VPC, VPD, DCU, SFU, ASP, MCM
st.w Ak,<effa>	Store address register word	DCU, ASP, MCM
st.w Sk,<effa>	Store scalar word	DCU, ASP, MCM

st.w Vk,<effa>	Store vector word	VPC, VPD, DCU, SFU, ASP, MCM
st.w.f Vk,<effa>	Store vector word using not VM	VPC, VPD, DCU, SFU, ASP, MCM
st.w.t Vk,<effa>	Store vector word using VM	VPC, VPD, DCU, SFU, ASP, MCM
st.x VM,<effa>	Store VM into memory	VPC, VPD, DCU, ASP, MCM
stcmr Ak,<effa>	Store communication registers	DCU, SFU, ASP, CPX, IPP, MCM
ste.b Sk,<effa>	Store an extended scalar byte	DCU, SFU, ASP, VPC, MCM
ste.b.f Sk,<effa	Store extended scalar byte using not VM	DCU, SFU, ASP, VPC, MCM
ste.b.t Sk,<effa>	Store extended scalar byte using VM	DCU, SFU, ASP, VPC, MCM
ste.d Sk,<effa>	Store an extended scalar double float	DCU, SFU, ASP, VPC, MCM
ste.d.f Sk,<effa>	Store extended scalar double using not VM	DCU, SFU, ASP, VPC, MCM
ste.d.t Sk,<effa>	Store extended scalar double using VM	DCU, SFU, ASP, VPC, MCM
ste.h Sk,<effa>	Store an extended scalar halfword	DCU, SFU, ASP, VPC, MCM
ste.h.f Sk,<effa>	Store extended scalar halfword using not VM	DCU, SFU, ASP, VPC, MCM
ste.h.t Sk,<effa>	Store extended scalar halfword using VM	DCU, SFU, ASP, VPC, MCM
ste.l Sk,<effa>	Store an extended scalar longword	DCU, SFU, ASP, VPC, MCM
ste.l.f Sk,<effa>	Store extended scalar longword using not VM	DCU, SFU, ASP, VPC, MCM
ste.l.t Sk,<effa>	Store extended scalar longword using VM	DCU, SFU, ASP, VPC, MCM
ste.s Sk,<effa>	Store an extended scalar single float	DCU, SFU, ASP, VPC, MCM
ste.s.f Sk,<effa>	Store extended scalar single using not VM	DCU, SFU, ASP, VPC, MCM
ste.s.t Sk,<effa>	Store extended scalar single using VM	DCU, SFU, ASP, VPC, MCM
ste.w Sk,<effa>	Store an extended scalar word	DCU, SFU, ASP, VPC, MCM
ste.w.f Sk,<effa>	Store extended scalar word using not VM	DCU, SFU, ASP, VPC, MCM
ste.w.t Sk,<effa>	Store extended scalar word using VM	DCU, SFU, ASP, VPC, MCM
stop	Stop CPU clocks	ASP
stvi.b Sk,Vj	Scalar index store vector byte	VPC, VPD, ASP, DCU, MCM
stvi.b Vk,Vj	Index store vector byte	VPC, VPD, ASP, DCU, MCM
stvi.b.f Sk,Vj	Scalar index store vector byte using not VM	VPC, VPD, ASP, DCU, MCM
stvi.b.f Vk,Vj	Index store vector byte using not VM	VPC, VPD, ASP, DCU, MCM
stvi.b.t Sk,Vj	Scalar index store vector byte using VM	VPC, VPD, ASP, DCU, MCM
stvi.b.t Vk,Vj	Index store vector byte using VM	VPC, VPD, ASP, DCU, MCM
stvi.d Sk,Vj	Scalar index store vector double float	VPC, VPD, ASP, DCU, MCM
stvi.d Vk,Vj	Index store vector double float	VPC, VPD, ASP, DCU, MCM
stvi.d.f Sk,Vj	Scalar index store vector double using not VM	VPC, VPD, ASP, DCU, MCM
stvi.d.f Vk,Vj	Index store vector double using not VM	VPC, VPD, ASP, DCU, MCM
stvi.d.t Sk,Vj	Scalar index store vector double using VM	VPC, VPD, ASP, DCU, MCM
stvi.d.t Vk,Vj	Index store vector double using VM	VPC, VPD, ASP, DCU, MCM
stvi.h Sk,Vj	Scalar index store vector halfword	VPC, VPD, ASP, DCU, MCM
stvi.h Vk,Vj	Index store vector halfword	VPC, VPD, ASP, DCU, MCM
stvi.h.f Sk,Vj	Scalar index store vector half using not VM	VPC, VPD, ASP, DCU, MCM
stvi.h.f Vk,Vj	Index store vector halfword using not VM	VPC, VPD, ASP, DCU, MCM
stvi.h.t Sk,Vj	Scalar index store vector half using VM	VPC, VPD, ASP, DCU, MCM
stvi.h.t Vk,Vj	Index store vector halfword using VM	VPC, VPD, ASP, DCU, MCM
stvi.l Sk,Vj	Scalar index store vector longword	VPC, VPD, ASP, DCU, MCM
stvi.l Vk,Vj	Index store vector longword	VPC, VPD, ASP, DCU, MCM
stvi.l.f Sk,Vj	Scalar index store vector long using not VM	VPC, VPD, ASP, DCU, MCM

stvi.l.f Vk,Vj	Index store vector longword using not VM	VPC, VPD, ASP, DCU, MCM
stvi.l.t Sk,Vj	Scalar index store vector long using VM	VPC, VPD, ASP, DCU, MCM
stvi.l.t Vk,Vj	Index store vector longword using VM	VPC, VPD, ASP, DCU, MCM
stvi.s Sk,Vj	Scalar index store vector single float	VPC, VPD, ASP, DCU, MCM
stvi.s Vk,Vj	Index store vector single float	VPC, VPD, ASP, DCU, MCM
stvi.s.f Sk,Vj	Scalar index store vector single using not VM	VPC, VPD, ASP, DCU, MCM
stvi.s.f Vk,Vj	Index store vector single using not VM	VPC, VPD, ASP, DCU, MCM
stvi.s.t Sk,Vj	Scalar index store vector single using VM	VPC, VPD, ASP, DCU, MCM
stvi.s.t Vk,Vj	Index store vector single using VM	VPC, VPD, ASP, DCU, MCM
stvi.w Sk,Vj	Scalar index store vector word	VPC, VPD, ASP, DCU, MCM
stvi.w Vk,Vj	Index store vector word	VPC, VPD, ASP, DCU, MCM
stvi.w.f Sk,Vj	Scalar index store vector word using not VM	VPC, VPD, ASP, DCU, MCM
stvi.w.f Vk,Vj	Index store vector word using not VM	VPC, VPD, ASP, DCU, MCM
stvi.w.t Sk,Vj	Scalar index store vector word using VM	VPC, VPD, ASP, DCU, MCM
stvi.w.t Vk,Vj	Index store vector word using VM	VPC, VPD, ASP, DCU, MCM
sub.b Sj,Sk	Subtract scalar/scalar integer byte	ASP
sub.b Vi,Sj,Vk	Subtract vector/scalar integer byte	VPC, VPD, ASP
sub.b Vi,Vj,Vk	Subtract vector/vector integer byte	VPC, VPD_
sub.b.f Vi,Sj,Vk	Subtract vector/scalar byte using not VM	VPC, VPD, ASP
sub.b.f Vi,Vj,Vk	Subtract byte vectors using not VM	VPC, VPD_
sub.b.t Vi,Sj,Vk	Subtract vector/scalar byte using VM	VPC, VPD, ASP
sub.b.t Vi,Vj,Vk	Subtract byte vectors using VM	VPC, VPD_
sub.d Si,Vj,Vk	Subtract scalar/vector double float	VPC, VPD, ASP
sub.d Sj,Sk	Subtract scalar/scalar double float	SFU, ASP, IPP
sub.d Vi,Sj,Vk	Subtract vector/scalar double float	VPC, VPD, ASP
sub.d Vi,Vj,Vk	Subtract vector/vector double float	VPC, VPD_
sub.d.f Si,Vj,Vk	Subtract scalar/vector double using not VM	VPC, VPD, ASP
sub.d.f Vi,Sj,Vk	Subtract vector/scalar double using not VM	VPC, VPD, ASP
sub.d.f Vi,Vj,Vk	Subtract double vectors using not VM	VPC, VPD_
sub.d.t Si,Vj,Vk	Subtract scalar/vector double using VM	VPC, VPD, ASP
sub.d.t Vi,Sj,Vk	Subtract vector/scalar double using VM	VPC, VPD, ASP
sub.d.t Vi,Vj,Vk	Subtract double vectors using VM	VPC, VPD_
sub.h #N,Ak	Subtract immediate address halfword	ASP, IPP
sub.h #N,Sk	Subtract scalar/immediate integer halfword	ASP, IPP
sub.h #n,Ak	Subtract short immediate address halfword	ASP, IPP
sub.h Aj,Ak	Subtract address register halfword	ASP
sub.h Sj,Sk	Subtract scalar/scalar integer halfword	ASP
sub.h Vi,Sj,Vk	Subtract vector/scalar integer halfword	VPC, VPD, ASP
sub.h Vi,Vj,Vk	Subtract vector/vector integer halfword	VPC, VPD_
sub.h.f Vi,Sj,Vk	Subtract vector/scalar halfword using not VM	VPC, VPD, ASP
sub.h.f Vi,Vj,Vk	Subtract halfword vectors using not VM	VPC, VPD_
sub.h.t Vi,Sj,Vk	Subtract vector/scalar halfword using VM	VPC, VPD, ASP
sub.h.t Vi,Vj,Vk	Subtract halfword vectors using VM	VPC, VPD_
sub.l Sj,Sk	Subtract scalar/scalar integer longword	ASP
sub.l Vi,Sj,Vk	Subtract vector/scalar integer longword	VPC, VPD, ASP

sub.l Vi,Vj,Vk	Subtract vector/vector integer longword	VPC, VPD_
sub.l.f Vi,Sj,Vk	Subtract vector/scalar longword using not VM	VPC, VPD, ASP
sub.l.f Vi,Vj,Vk	Subtract longword vectors using not VM	VPC, VPD_
sub.l.t Vi,Sj,Vk	Subtract vector/scalar longword using VM	VPC, VPD, ASP
sub.l.t Vi,Vj,Vk	Subtract longword vectors using VM	VPC, VPD_
sub.s #N,Sk	Subtract scalar/immediate single float	SFU, ASP, IPP
sub.s Si,Vj,Vk	Subtract scalar/vector single float	VPC, VPD, ASP
sub.s Sj,Sk	Subtract scalar/scalar single float	SFU, ASP, IPP
sub.s Vi,Sj,Vk	Subtract vector/scalar single float	VPC, VPD, ASP
sub.s Vi,Vj,Vk	Subtract vector/vector single float	VPC, VPD_
sub.s.f Si,Vj,Vk	Subtract scalar/vector single using not VM	VPC, VPD, ASP
sub.s.f Vi,Sj,Vk	Subtract vector/scalar single using not VM	VPC, VPD, ASP
sub.s.f Vi,Vj,Vk	Subtract single vectors using not VM	VPC, VPD_
sub.s.t Si,Vj,Vk	Subtract scalar/vector single using VM	VPC, VPD, ASP
sub.s.t Vi,Sj,Vk	Subtract vector/scalar single using VM	VPC, VPD, ASP
sub.s.t Vi,Vj,Vk	Subtract single vectors using VM	VPC, VPD_
sub.w #N,Ak	Subtract immediate address word	ASP, IPP
sub.w #N,Sk	Subtract scalar/immediate integer word	ASP, IPP
sub.w #n,Ak	Subtract short immediate address word	ASP, IPP
sub.w Aj,Ak	Subtract address register word	ASP
sub.w Sj,Sk	Subtract scalar/scalar integer word	ASP
sub.w Vi,Sj,Vk	Subtract vector/scalar integer word	VPC, VPD, ASP
sub.w Vi,Vj,Vk	Subtract vector/vector integer word	VPC, VPD_
sub.w.f Vi,Sj,Vk	Subtract vector/scalar word using not VM	VPC, VPD, ASP
sub.w.f Vi,Vj,Vk	Subtract word vectors using not VM	VPC, VPD_
sub.w.t Vi,Sj,Vk	Subtract vector/scalar word using VM	VPC, VPD, ASP
sub.w.t Vi,Vj,Vk	Subtract word vectors using VM	VPC, VPD_
sum.b Vk	Sum a vector of bytes	VPC, VPD, ASP
sum.b.f Vk	Sum a vector of bytes using not VM	VPC, VPD, ASP
sum.b.t Vk	Sum a vector of bytes using VM	VPC, VPD, ASP
sum.d Vk	Sum a vector of double float	VPC, VPD, ASP, SFU, IPP
sum.d.f Vk	Sum a vector of double using not VM	VPC, VPD, ASP, SFU, IPP
sum.d.t Vk	Sum a vector of double using VM	VPC, VPD, ASP, SFU, IPP
sum.h Vk	Sum a vector of halfwords	VPC, VPD, ASP
sum.h.f Vk	Sum a vector of halfwords using not VM	VPC, VPD, ASP
sum.h.t Vk	Sum a vector of halfwords using VM	VPC, VPD, ASP
sum.l Vk	Sum a vector of longwords	VPC, VPD, ASP
sum.l.f Vk	Sum a vector of longwords using not VM	VPC, VPD, ASP
sum.l.t Vk	Sum a vector of longwords using VM	VPC, VPD, ASP
sum.s Vk	Sum a vector of single float	VPC, VPD, ASP, SFU, IPP
sum.s.f Vk	Sum a vector of single using not VM	VPC, VPD, ASP, SFU, IPP
sum.s.t Vk	Sum a vector of single using VM	VPC, VPD, ASP, SFU, IPP
sum.w Vk	Sum a vector of words	VPC, VPD, ASP
sum.w.f Vk	Sum a vector of words using not VM	VPC, VPD, ASP
sum.w.t Vk	Sum a vector of words using VM	VPC, VPD, ASP

sysc #r,#g	Perform a system call	ASP, DCU, SFU, IPP
tac <effa>	Test and clear a byte in memory	DCU, SFU, ASP, MCM
tas <effa>	Test and set a memory byte	DCU, SFU, ASP, MCM
trap #rm,#b	Force a trap system exception	ASP, CPX, DCU, SFU, IPP
tst <Ceffa>	Test communication register lock bit	ASP, CPX
tstvv	Test value of vector valid flag	ASP
tzc Sj,Sk	Count of trailing zeros in Sj	SFU, ASP, IPP
tzc Vj,Vk	Trailing zero count vector	VPC, VPD
tzc.f Vj,Vk	Trailing zero count vector using not VM	VPC, VPD
tzc.t Vj,Vk	Trailing zero count vector using VM	VPC, VPD
ulk <Ceffa>	Unlock communication register	ASP, CPX, IPP
wfork	Wait for a fork event	ASP, CPX, IPP
xmti Sk	Transmit interrupt	ASP, PIA, SP2, CPX
xor #N,Ak	Exclusive OR immediate to address register	ASP, IPP
xor #N,Sk	Exclusive OR scalar/immediate	ASP, IPP
xor Aj,Ak	Exclusive OR address register	ASP
xor Sj,Sk	Exclusive OR scalar/scalar	ASP
xor Vi,Sj,Vk	Exclusive OR vector/scalar	VPC, VPD, ASP
xor Vi,Vj,Vk	Exclusive OR two vectors	VPC, VPD_
xor.f Vi,Sj,Vk	Exclusive OR vector/scalar using not VM	VPC, VPD, ASP
xor.f Vi,Vj,Vk	Exclusive OR two vectors using not VM	VPC, VPD_
xor.t Vi,Sj,Vk	Exclusive OR vector/scalar using VM	VPC, VPD, ASP
xor.t Vi,Vj,Vk	Exclusive OR two vectors using VM	VPC, VPD_
xpnd.f Vj,Vk	Expand a vector using not VM	VPC, VPD
xpnd.t Vj,Vk	Expand a vector using VM	VPC, VPD

Appendix E
Glossary

E.1 Overview

The following terms are defined as they are used at CONVEX. These terms must be used in CONVEX documentation only as defined in this glossary. Standard acronyms are also included.

E.2 Terms

accumulator. A hardware register containing the results of arithmetic and logical operations.

context (processor). The entire, current state of the machine associated with the executing process.

exception. A hardware-detected event that disrupts the running of a program, process, or system. *See also* Fault.

fault. An exception that, while halting the instruction, leaves the registers and memory in a consistent state. The instruction can often resume its course when the cause of the fault is corrected.

forced faulting mode. A mode of operation where the CPU diagnostics cause simulated pagefaults to occur. In forced faults mode, a bit is set in hardware so that each time data is accessed in main memory, the entire context of the processor is saved off and then restored. This process thoroughly exercises the buses that are used to capture and restore the context of the machine as well as the entire memory system.

immediates. Operands that are contained within the instruction stream.

instruction. Instructions are used by the programmer to direct operations on the system's register set and memory.

interrupt. An occurrence, other than an exception, that changes the normal flow of instruction execution. An interrupt originates from hardware, such as an input or output (I/O) device.

interval timer. A privileged register. The interval timer is used to generate an interrupt based on the passage of a period of time.

logical address. Logical address space is that space seen by the application programmer.

modified bit. A bit within the CPU that records all valid write references to page frames. The modified bit is used by the operating system for memory management.

opcode. The code or sequence of bits in an instruction that determines the operation to be performed.

operand. A register or memory location referenced by an instruction.

page. A page is the unit of logical memory controlled by the memory management algorithms. In the C120, a page is 4 K (4,096) contiguous bytes.

pagefault. A pagefault occurs when a process requests data that is not currently in main memory. The machine first saves off the state of all controllers onto a context stack in main memory. The operating system will create a free page of physical memory to bring the data in from the disk. The appropriate Page Table Entries (PTEs) are set up so that the proper logical-to-physical translation occurs. The machine reads back from memory the state of the machine from the context stack, and restores the processor to the same state that it was in when it determined that the data it needed was nonresident. The CPU can then continue with normal operation of the process.

page frame. A page frame is the unit of physical (main) memory in which pages are placed. Associated with each page frame are referenced and modified bits to aid in memory management.

page table entry. A Page Table Entry (PTE) is an entry in a page table. A PTE is a word that contains various flags and fields that are used in the translation of logical-to-physical addresses. Address translation uses two levels of page table indexing. The first-level page table is referenced using bits <28> through <22> of a logical address. This is called the Index.1 field. The second-level page table is referenced using bits <21> through <12> of a logical address. This is called the Index.2 field.

physical address. Hardware-identified address in physical (main) memory consisting of a page frame number and the number of a byte within the page.

privileged instruction. An instruction used by the operating system or privileged systems programs. It must execute in ring0, or an exception occurs.

processor status word. The Processor Status Word (PSW) is a word that contains control flags used to control and indicate the state of various computations and sequences within the processor.

PSW. *See* Processor status word.

PTE. *See* Page table entry.

register. A hardware entity that is used to contain addresses, operands, and status.

segment. The basic partition of the logical memory space, equal to 512 megabytes.

Appendix F
Problem Reporting

F.1 Overview

The *contact* utility is the recommended way to report minor hardware deficiencies and technical documentation problems to the Technical Assistance Center (TAC). This utility is an interactive tool that prompts the user for the information to properly file a problem report.

> NOTE
>
> The *contact* utility is not intended for requesting customer service for hardware failures. To restore your CONVEX equipment to operational status, faster service can be obtained by directly telephoning the TAC (refer to "Technical Assistance" in the Preface).

To use the *contact* utility, there must be a phone connection to the TAC. UNIX-to-UNIX Communication Protocols (UUCP) allow communication between UNIX systems by either dial-in or hard-wired communication lines. For more information, refer to *uucp(1)* or to the *info(1)* entry in the UNIX man pages.

The name and version number of the product involved is required. Use the *vers* command to ascertain the program or utility name and version. The syntax for the command is **vers** *filename*, where *filename* is the full pathname of the program. If the full pathname of the program is not known, enter **which** *program*. For more information, refer to the *vers(1)* and *which(1)* entries in the UNIX man pages.

F.2 Information Required to Report a Problem

The *contact* utility requires the following information:

1. The customer name, title, phone number, and corporate name
2. The hardware nomenclature, part number, and revision level, or the technical manual name, document number, and version

> NOTE
>
> Use *vers* and *which* to identify product name and version.

3. A short (one line) summary of the problem

4. The more information provided, the more quickly the problem can be isolated and solved. At a minimum, include a detailed description of the problem (including page references, if applicable), the source code, and a stack backtrace whenever possible.

> **NOTE**
>
> See the *adb(1)* or *csd(1)* man pages for information on obtaining stack backtraces.

5. The priority of the problem, selected from a list of six levels
6. Instructions on how to reproduce the problem, including the command syntax used, any flags invoked, or anything else attempted to make the program run
7. Any other comments about the problem or files to be submitted

The *contact* user has a chance to review and edit the report prior to submitting it. If the user decides to delay submitting the report, the session can be aborted. The report is automatically saved in the user's top-level directory in a file named *dead.report*.

See the following figure for a sample *contact* session. User input is in bold lettering, and the system response is in monospace type.

Figure F–1, Sample *contact* Session

```
%contact (RETURN)
Welcome to contact version 0.11 ()

Enter your name, title, phone number, and corporate name (^D to terminate)
> Margaret Atwood, systems programmer, 814-4444, University r
> of Chicago (RETURN)
> (CTRL-D)

Enter the name of the product involved
> CONVEX UNIX Programmer's Manual, Part I (RETURN)

Enter the version number (in the form X.X or X.X.X.X) of the product
> Revision 4.0 (RETURN)

Enter a short (1 line) summary of the problem
> The finger command manual page lists nonexistent bug (RETURN)

Enter a detailed description of the problem (^D to terminate)
> The finger(1) man page says, under the BUGS section, that "Only the first
line of the .project file is printed."  Happily, this is not true! (RETURN)
> (CTRL-D)

Enter a problem priority, based on the following:
1) Critical    - work cannot proceed until the problem is resolved.
2) Serious     - work can proceed around the problem, with difficulty.
3) Necessary   - problem has to be fixed.
4) Annoying    - problem is bothersome.
5) Enhancement - requested enhancement.
6) Informative - for informational purposes only.
> 4 (RETURN)

Enter the instructions by which the problem may be reproduced (^D to terminate)
> a) put more than one line in .project (RETURN)
> b) read the man page for finger(1) (RETURN)
> (CTRL-D)

Enter any comments that are applicable (^D to terminate) (RETURN)
> (CTRL-D)

Do you have any suggestions or comments on the documentation that you
referenced when you were trying to resolve your problem (for example,
additions, corrections organization, accessibility)? (^D to terminate)
> The man page should be updated. (RETURN)
> (CTRL-D)

Are there any files that should be included in this report (yes | no)?
> no (RETURN)

Please select one of the following options:
1) Review the problem report.
2) Edit the problem report.
3) Submit the problem report.
4) Abort the problem report.
> 3 (RETURN)

Problem report submitted.
%
```

THIS PAGE INTENTIONALLY LEFT BLANK

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

THIS PAGE INTENTIONALLY LEFT BLANK

Electronic Mail

The Hardware Documentation Group has an email address for documentation comments. Use this service to give us a quick response mechanism if you have special documentation questions that you would like addressed immediately. If you have a technical question, you should still contact the Technical Assistance Center, as described previously. To use email response service, just send mail addressed to:

```
cnvxhwdoc@convex.COM
```

We will read your comments and give you a personal reply.

What to Include in an Email Message

When you use the electronic mail service, please provide the following information:

- The reader's name and company name
- A return email address in INTERNET notation or UUCP (bang) notation
- The manual that is being critiqued
- The chapter and page number in question
- The comment
- Indicate if a personal response is required

Reader's Forum

If you wish to mail your comments to us, please use the form on the next page and list the document page number with your questions and comments. Thank you.

THIS PAGE INTENTIONALLY LEFT BLANK

CONVEX Processor Diagnostics Manual
(C200 Series)
Document No. 760-000550-203, Third Edition

Reader's Forum

_____

_____

_____

_____

_____

_____

_____

_____

From:

Name_____ Title_____

Company_____ Date_____

Address and Phone No._____

FOR ADDITIONAL INFORMATION OR DOCUMENTATION:

Location	Phone Number
In Texas	(214)952-0200
Other continental locations	1(800)952-0379
Outside continental US	Contact local CONVEX office

Direct mail orders to:

CONVEX Computer Corporation
Customer Service
PO Box 833851
Richardson TX 75083-3851 USA

(Fold Here First)

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1046 RICHARDSON, TEXAS

POSTAGE WILL BE PAID BY ADDRESSEE

CUSTOMER SERVICE
CONVEX Computer Corp.
P.O. Box 833851
Richardson, TX 75083-3851

(Fold Here Second)

(Tape or Staple)