Multiprocessor SPARC™ Architecture Simulator (MPSAS) User's Guide

THE NETWORK IS THE COMPUTER™

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

Part Number: 800-6795-03

Contents

Figures

Tables

Preface

This book, *Multiprocessor SPARC™ Architecture Simulator (MPSAS) User's Guide*, is one of a four-manual documentation set for the microSPARC-II™ technology. The other three manuals are:

- *Multiprocessor SPARC Architecture Simulator (MPSAS) Programmer's Guide*, which explains the facilities for creating or modifying MPSAS modules or otherwise extending MPSAS
- *microSPARC-IIep Validation Catalog*, which describes a suite of validation tests for the microSPARC-IIep technology
- *microSPARC-IIep Megacell Reference*, which explains how to build microSPARC-IIep megacells

Organization of This Book

This book is intended for programmers who develop and debug programs to be run on hardware that is simulated by MPSAS. It contains the following chapters and appendixes:

Chapter 1, *Overview*, introduces MPSAS and a number of concepts used in the rest of the manual.

Chapter 2, *Tutorial*, contains examples on how to use MPSAS to accomplish common simulation tasks.

Chapter 3, *User Interface*, discusses the MPSAS user interface, details the command-line interface, and briefly describes each command.

Chapter 4, *Sample Architectures*, describes the sample architectures provided with MPSAS and lists the modules, features, and configurations.

Chapter 5, *Configuration File*, tells how to configure MPSAS and includes the syntax and semantics of MPSAS configuration files.

Chapter 6, *Message Types*, describes the messages the modules use to communicate with each other.

Chapter 7, *Modules*, describes the purpose and behavior of the MPSAS modules, how to control them through the user interface and configuration file, how the modules interface with each other, and which source files implement them.

Chapter 8, *Trace Tools*, discusses a support program for MPSAS that helps process trace output that the simulator produces.

Appendix A, *Error Messages*, lists common error messages and suggests ways to troubleshoot errors.

Appendix B, *Command Manual Pages*, describes MPSAS user-interface commands in detail.

Appendix C, *The stand Directory*, describes tools that are provided with MPSAS for programming on bare SPARC hardware, which is the environment simulated by MPSAS.

At the end of the book is an index.

Prerequisite Knowledge

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

Related Books and References

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

The SPARC Architecture Manual/Version 8 by David Weaver, Prentice Hall; ISBN: 0138250014

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

Typographic Conventions

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

TABLE P-1 Typographic Conventions

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

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

AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename*.
AaBbCc123	Book titles, section titles in cross-references, new words or terms, or emphasized words	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.
<>	A bit number or colon-separated range of bit numbers within a field; bit 0 is the least significant bit.	`WB_VECTOR`<15:0>

Sun Documents

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

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

Sun Documentation Online

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

Disclaimer

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

CHAPTER **1**

Overview

The MPSAS behavioral simulator is a software tool with which you can model computer systems based on the SPARC technology at the instruction or transaction level. The system being simulated is independent of the computer on which the simulation is running. When run on a SPARCstation™ machine, MPSAS typically simulates thousands of SPARC instructions per second.

The goal of instruction-level simulation is to model the programmer’s view of a computer system. This process entails simulations of the execution of processor instructions, system and device registers, interrupts, and the memory hierarchy. The structure of the simulator need not directly correspond to the structure of the hardware.

The goals of transaction-level simulation are to model the major components of the computer system hardware (for example, each ASIC and bus), model the transactions between those components, and support the programmer’s view. Transaction-level models are more difficult to develop and run more slowly than instruction-level models, but they model the computer hardware more accurately.

The sample architectures supplied with MPSAS are modeled at the instruction level.

The simulator aids the following activities:

- Porting machine-dependent code, for example, UNIX kernel, boot PROM, and diagnostics. The target computer system hardware need not be available before code testing starts. Also, the simulator provides a software environment that is more easily controlled and observed than that of real hardware.
- Estimating performance. Accurate instruction counts of a benchmark program can be obtained with an instruction-level simulation. Accurate cycle counts require a timing-accurate, transaction-level simulation.
- Verifying the correctness of a design, for example, testing a cache coherence protocol.

- Verifying the correctness of a lower-level simulation, for example, gate-level. The lower-level simulation and MPSAS simulation of the same computer system are run simultaneously with the same test program, and their results are compared periodically.
- Exploring design trade-offs by simulating the different configurations of interest.

The MPSAS user interface supports a set of commands that control and observe the simulation. A batch facility is provided to combine commands in a text file. Online help is available for the commands.

1.1 Modular Structure

To build a system, a hardware design engineer selects instances of integrated circuits from a catalog and connects their pins via printed circuit board traces. Similarly, a user of MPSAS selects instances of module classes compiled into the simulator executable and connects their interfaces together. All of this information about instances of module classes and interface connections is located in a configuration file.

A module class is the code that simulates some hardware. An MPSAS module class may represent part of an integrated circuit or several boards full of integrated circuits. Instances of a module class are called module instances. This manual uses the term module by itself in many cases, where the distinction between class and instance is either unimportant or obvious.

Modules are written as necessary to simulate a computer system. For simplicity, several sample modules are available with MPSAS, including a SPARC integer unit (IU), a SPARC floating-point unit (FPU), memory management units (MMUs), memory, I/O devices, and assorted buses. You can use the sample modules along with custom modules in a simulation.

The configuration file is an ASCII text file. You can simulate different systems from the same set of modules by invoking the same simulator executable with different configuration files. For example, you can include a cache module in, or exclude it from, a simulation by using different configuration files.

In addition to the code for the various module classes, a simulator executable contains code that sets up the module classes and their instances, calls each module instance every cycle to do its next increment of work, handles communications between module instances, and provides an interface by which you can control and observe the simulated system. This software is collectively called “the framework.” A cycle is the notion of simulation time used in MPSAS. Its closest hardware analogy is the system clock.

An architecture is a simulator executable combined with one or more configuration files and any files that are required by the modules in the configuration files. Each configuration file specifies a system that uses module classes included in an architecture. MPSAS provides four sample architectures:

- sun4c – SPARCstation 1
- sun4e – SPARCengine™ 1
- `mbus` – a generic multiprocessor Mbus machine
- `simple` – a “bare bones” machine (CPU, FPU, RAM, and serial ports)

The files for each architecture are located in a directory with the same name as the architecture.

1.2 Communications

Communications are effected as follows.

1.2.1 Messages

Module instances communicate by sending messages to each other. For example, a module that simulates a processor can send a message to a module that simulates a memory device to fetch an instruction.

Message passing takes place over a channel, which is a mechanism provided by the framework to transport messages. There are two types of channels: simulation and debug. The simulation channel is only active when the simulation is running (the simulator cycles are being incremented). It is used by module instances to transfer information, such as a memory request, that is required by the simulation. The debug channel is always active and is used by module instances to support user interface commands.

Messages can contain data. No message size or content restrictions are enforced by the framework. However, there is a set of message formats that most modules use.

Each message has an associated type (a 32-bit value). The type implies the message protocol (format of message data and its use) and is used as an aid in the debugging modules and to multiplex multiple protocols on a connection.

Associated with each message is an optional cycle delay, which allows a module instance to send a message to another instance such that it is not received immediately but, rather, is delayed by a specified number of cycles.

1.2.2 Interfaces

Interfaces provide module instances with access to the simulation and debug channels. An interface is similar in concept to a group of pins on an integrated circuit. Interfaces are visible to you in the configuration file and in some user-interface commands.

Each interface belongs to a module instance. For two module instances to communicate, their interfaces must be connected together. FIGURE 1-1 shows two module instances, each with an interface (the circles) that are connected together. The line between the interfaces represents the simulation and debug channels. Module instance A owns interface α; module instance B owns interface β.

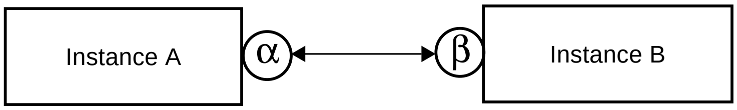

FIGURE 1-1 Interface Connections

When module instance A sends a message to its α interface, it is received by module instance B on its β interface.

Each interface has an associated name and type (for example, bus of type slave). Each module supports a fixed number of interface types. A module instance can have multiple interfaces of the same type; the name uniquely identifies each interface. Modules can enforce restrictions on the number of interfaces of each type.

CHAPTER 2

Tutorial

This chapter shows MPSAS being used to run a simple tutorial program, which is included with the product.

The tutorial introduces a number of different commands along with some other features of the user interface. (Chapter 3, *User Interface*, details the command syntax.) It uses the version of MPSAS and support files of the `simple` architecture.

Chapter 4, *Sample Architectures*, describes the system simulated by the `simple` architecture. For the purpose of this tutorial, you need know only that the simulated system contains a SPARC processor module, called `cpu1`, a memory module called `ram1`, and user interface-related modules.

In this tutorial, an overview tells you how to use MPSAS. The directions explain what you will accomplish without specifying the exact commands to type. Following those directions is a screen image that shows the commands and the responses from the simulator.

In this tutorial, you execute the simulator twice; the two sections, *The Start* on page 5 and *A More Sophisticated Session* on page 9, correspond to those two executions of the simulator.

2.1 The Start

In this section, you learn how to start the simulator, load and run a simple program, and examine and change the contents of memory. You should have a shell running, and your current directory should be the one in which MPSAS was installed on your machine. To begin, change directory to the one that contains the `simple` architecture (type **`cd sparctools/MPSAS/simple`**) and then type **`mpsas`** to execute the simulator.

At this point, the last few lines on the screen should look something like the following. (The characters in boldface are your input; the rest of the example is simulator output.)

```
hostname% cd sparctools/MPSAS/simple
hostname% mpsas
Simple architecture - Mpsas Release 1.1
Preprocessing configuration file "config_file".
Parsing configuration file "config_file".
Creating C module classes.
Creating module instances and interfaces.
Performing interface configurations.
Performing interface configuration verifications.
serial1: Serial Port A is /dev/ttyp9
serial1: Serial Port B is /dev/ttypa
Negative phase is inactive.

ui1:
```

MPSAS takes a few seconds to initialize, displaying the messages shown above as it does so. The first line identifies the architecture and the release number of the simulator. The second and third lines show that the simulator is preprocessing and reading configuration information from file `config_file` in the current directory. The simulator then builds the system, as described in the configuration file. The two lines that begin with `serial1:` (yours may differ slightly from those shown here) are the result of the initialization of module instance `serial1`, which you do not use in this tutorial.

The `ui1:` prompt is from the user interface module, which is waiting for you to type a command and press Return. This prompt indicates that the simulator is currently focused on module instance `ui1`. When the user interface requires a module instance name, you can omit it if it is the currently focused module instance.

Now load the program named `tutorial`, located in the `stand` directory, as shown in the following screen image. First, type **`help`** to display the usage of the `load` command and then use the `load` command to load the `tutorial` program from the MPSAS `stand` directory into address 0 of module instance `ram1`. The simulator also loads the program's symbol table.

```
ui1: help load
<load address> <instance> <file> [<symtable start> [<symtable end>]]
   This command loads data into a module instance. It is used to
   initialize modules that contain memory (e.g., RAM, ROM, etc.).
   It reads data from SPARC a.out and ELF files and loads it into a
   module instance at the specified load address. The load address is
   interpreted by ....

ui1: load 0 ram1 ../stand/tutorial

ui1:
```

MPSAS simulates a bare machine, with no operating system, monitor, or other code besides that which you load into it. Therefore, the tutorial program, like any program that you run on MPSAS, contains its own trap tables and processor initialization code.

You will be executing a number of commands that are specific to the SPARC processor module, so tell the simulator to `focus` on `cpu1`. Notice that the prompt changes to show the new focus. To ensure that the program is loaded properly, use the `read` command to display five words of memory, starting at address 0, as instructions. You should see the beginning of the trap table.

```
ui1: focus cpu1

cpu1: read inst 0 5
_trap_table:            :   ba        _start
_trap_table+0x4:        :   mov       %psr, %l0
_trap_table+0x8:        :   nop
_trap_table+0xc:        :   nop
_trap_table+0x10:       :   mov       0x1, %l3

cpu1:
```

Use the `sh` command to invoke a shell with a command to show the source code for the `tutorial` program, `tutorial.c`, in the `stand` directory. The source code for the trap table, trap handlers, and other support code is also in that directory; in this tutorial, however, you do not work with the support code. The `tutorial` program computes a function of the number in global variable `num`, leaving the result in global variable `result`. Write a number into `num`, run the program until it stops, and then read `result`. The support code stops the simulation when the program finishes by issuing a special SPARC trap instruction, which is understood by the simulator to be a request to stop the simulation.

```
cpu1: sh more ../stand/tutorial.c
/*
 * tutorial.c -- A tiny program used in the tutorial.
 * You set num to something, run the program, and result contains the answer.
 */

unsigned num;           /* you set this before you start */
unsigned result;        /* the answer will be here */

/*
 * fact(u) returns the factorial of u.
 */
unsigned
fact(u)
        unsigned u;
{
        unsigned factorial = 1;
        while (u)
                factorial *= u--;
        return factorial;
}

/*
 * main() just computes some useless number using fact().
 */
int
main()
{
        result  =  fact(num+1) - fact(num);
}

cpu1: write word &num 3

cpu1: run
trap1: cpu connected to me stopped the simulation.

cpu1: read word &result 1
_result            0x00000012 ....

cpu1:
```

The line of output beginning with `trap1:` is the indicator that this special trap instruction was executed by the program, stopping the simulation.

The four dots to the right of the hex value of `&result`, returned by the `read` command, are an ASCII display of the four characters that make up that word. Because all four characters are unprintable, they print as periods.

Type **list** to show what variables are defined by the currently focused module instance, in this case, cpu1. An entry suffixed with an asterisk contains members (variables within the variable), which you see when you print the entry. Try printing some variables by using the print command, like the SPARC processor module's pc and psr. Finally, type **quit** to exit from the simulator and return to the shell prompt.

```
cpu1: list
cpu1 Variables:
annul         annulled_count            bytes         cc            chars
doubles       exec*         executed_count              ext_trap_pending
external_count              floats        fp            fpu_ea        g0
g1            g2            g3            g4            g5            g6
g7            hwords        i0            i1            i2            i3
i4            i5            i6            i7            instructions  intr_count
irl           l0            l1            l2            l3            l4
l5            l6            l7            latest_instr  latest_instr_addr
latest_mem_addr             latest_mem_addr_valid       latest_mem_data
latest_mem_data_size        latest_trap_instr           latest_trap_num
latest_trap_pc              lwords        master_rcv_routine          npc
nwins         o0            o1            o2            o3            o4
o5            o6            o7            pc            prefetch
prefetch_instr              prefetch_instr_pc           prefetch_valid
psr*          sanitycheck   sp            stop_on_reset               tbr*
trap_count    watchexec     watchexternal               watchintr     watchtrap
wim           words         y

cpu1: print pc
0x00001aa4 &__exit+0x4

cpu1: print psr
impl=0x1 ver=0x1 n=0 z=1 v=0 c=0 ec=0 ef=1 pil=0x0 s=1 ps=0 et=1 cwp=0x0

cpu1: quit
hostname%
```

2.2 A More Sophisticated Session

Now run the same program again, but with a few more commands and features this time.

To restart a simulation, exit the simulator and restart it. While it may work in some cases to simply reset the SPARC processor's pc and npc variables, these steps are sometimes problematic because the simulator does not reset all the module instances. Exiting and restarting the simulator is one sure way to take the simulator to a known state. Therefore, if the simulator is still running from the first part of the tutorial, quit it now and restart.

In this section, you run script `tutorial_cmds` in the `simple` architecture directory to set up the tutorial program. (A script is a file that contains MPSAS commands.) Create a user-interface alias for the command `sh more` and then use the alias to display the script.

Scripts and aliases are both mechanisms for reducing the amount of typing you have to do and the number of things you have to remember. If you put your `alias` commands in a file called `.MPSASrc` in your home directory, those aliases are immediately available every time you execute MPSAS. As MPSAS starts up, it looks for such a file and executes any commands in it when it is started. Refer to Appendix B, *Command Manual Pages*, for details.

```
hostname% mpsas
Simple architecture - Mpsas Release 1.1
Preprocessing configuration file "config_file".
Parsing configuration file "config_file".
Creating C module classes.
Creating module instances and interfaces.
Performing interface configurations.
Performing interface configuration verifications.
serial1: Serial Port A is /dev/ttyp9
serial1: Serial Port B is /dev/ttypa
Negative phase is inactive.

ui1: alias m sh more

ui1: m tutorial_cmds
# tutorial_cmds - MPSAS script to load and set up tutorial program.
# usage: tutorial_cmds <value for "num" variable>
if ($argc != 1) { \
        echo Usage: $0 \\<value\\>; flush \
}

load 0 ram1 ../stand/tutorial
focus cpu1
write word &num $1
echo Tutorial program has been loaded and "num" has been initialized to $1.

ui1:
```

MPSAS ignores those lines in a script that begin with # as well as the blank lines. The `if` command ensures that one argument is specified (`$argc == 1`). The `load` and `focus` commands are the same as before, but the `write` command sets the value of num to $1, which is a notation that is replaced by the first argument to the command that invokes the script. Finally, the echo command writes its arguments to the screen or window so that scripts can display arbitrary messages.

Invoke the script with the file command, with 3 as an argument. Now that the program is loaded and variable num is set, type **fork** to create a copy of the simulator.

```
ui1: file tutorial_cmds 3
Tutorial program has been loaded and "num" has been initialized to 3.

cpu1: fork
MPSAS process 2934 active
MPSAS process 2932 waiting

cpu1:
```

At this point, you are interacting with the copy (a child process), and any change in state does not affect the original simulator. Later, you can return to the original simulator and continue from this saved state without exiting, restarting the simulator, and redoing your setup.

Next, find out how to set a breakpoint. Type **help** without any arguments to display the usage statement for each command and pipe the result to the UNIX grep utility to show only those lines that contain the word break.

Set a breakpoint on the tutorial program's fact function, which computes the factorial of a number, and run the program. When you hit the breakpoint, type **where** for a stack backtrace.

```
cpu1: help | grep break
breakpoint [add <address> | delete <number>] - breakpoints
breakpoint [add <address> | delete <number>] - breakpoints

cpu1: breakpoint add &fact

cpu1: run
cpu1: breakpoint 1 at _fact (0x1000) encountered.

cpu1: where
_fact(0x3 0x0 0x0 0x0 0x0 0x0)
_main+0x14(0x10a0 0x0 0x0 0x0 0x0 0x0)
_start+0x40(0x0 0x0 0x0 0x0 0x0 0x0)

cpu1:
```

The simulator does not know how many arguments each function has or their types, so it just displays cpu1's registers in0 (%i0) through in5 (%i5) in hexadecimal.

Using the `set` command, set `cpu1`'s `watchexec` variable to 1 and `step` a few instructions. When `watchexec` is true (nonzero), the `cpu1` module instance prints instructions in disassembled form after it executes them. As you advance through the program with the `step` command, the displayed instructions have been completely executed.

```
cpu1: set watchexec=1

cpu1: step 4
cpu1.watchexec(31): _fact              : sethi%hi(_end+0xffffbb70), %g1
cpu1.watchexec(32): _fact+0x4          : add%g1, 0x398, %g1
cpu1.watchexec(33): _fact+0x8          : save%sp, %g1, %sp
cpu1.watchexec(35): _fact+0xc          : st%i0, [%fp + 0x44]

cpu1:
```

Turn the `watchexec` flag off; you have to type only enough characters in a variable name to unambiguously identify it. Use the `when` command to have the simulator print the current window pointer in the `cpu1`'s `psr` register whenever it changes, along with the `pc`, then continue execution.

```
cpu1: set watchexe=0

cpu1: when psr.cwp changes {print -v cpu1.psr.cwp,cpu1.pc}

cpu1: run
cpu1.psr.cwp: 0x6
cpu1.pc:      0x00001084        &_main+0x1c
cpu1: breakpoint 1 at _fact (0x1000) encountered.

cpu1:
```

Notice that `cpu1.psr.cwp` changed before you hit the breakpoint because the simulator performed the `print` operation. Show all `cpu1` registers by typing **`regs`**.

```
cpu1: regs
Window: 6
     INS           LOCALS        OUTS          GLOBALS
0:   0x000010a0    0x00000000    0x00000004    0x00000000
1:   0x00000000    0x00000000    0x00000000    0xffffff98
2:   0x00000000    0x00000000    0x00000000    0x00000002
3:   0x00000000    0x00000000    0x00000000    0x00000000
4:   0x00000000    0x00000000    0x00000000    0x00000000
5:   0x00000006    0x00000000    0x00000000    0x00000000
6:   0x0000af80    0x00000000    0x0000af20    0x00000000
7:   0x00001a98    0x00000000    0x00001094    0x00000000

y:   0x00600000
pc:  0x00001000  _fact
npc: 0x00001004  _fact+0x4
sp:  0x0000af20  _end+0x6e90
fp:  0x0000af80  _end+0x6ef0

psr: 0x114010a6
     impl ver n z v c ec ef pil s ps et cwp
      1    1  - Z - - 0  1   0  1 0  1   6
wim: 0x00000002 (.....X.)  ['.' is valid, 'X' is invalid]
tbr  0x00000000 (tba=0x0 tt=0x0)
cpu1:
```

Now type **run** again to continue execution.

```
cpu1: run
cpu1.psr.cwp:   0x5
cpu1.pc:        0x0000100c      &_fact+0xc
cpu1.psr.cwp:   0x6
cpu1.pc:        0x0000109c      &_main+0x34
cpu1.psr.cwp:   0x0
cpu1.pc:        0x00001aa0      &__exit
trap1: cpu connected to me stopped the simulation.

cpu1:
```

Use the simulator's `expr` command to evaluate the result and compare it with the value of the tutorial program's `result` variable. `expr` prints the result in hexadecimal, decimal, symbolic (in case the value is an address), and character formats for convenience.

Now `quit` out of the copy of the simulator, which then returns the original simulator. The simulator's state is now identical to its state before `fork`. From here, you can run the program again, or fork and run the program in another copy of the simulator. For now, type **quit** again to exit the simulator.

```
cpu1: expr 4 * 3 * 2 - 3 * 2
0x12    18     &_trap_table+0x12       '\022'

cpu1: read word &result 1
_result            0x00000012 ....

cpu1: quit
Child exiting
MPSAS process 2934 terminated
MPSAS process 2932 active
No more children running. Next quit terminates the simulation.

cpu1: quit
hostname%
```

There are many more commands and features than those you have used here, but you are now acquainted with the basics and can experiment with more. Reading Chapter 3, *User Interface*, will increase your familiarity with the commands. Afterward, you may want to start up the tutorial program again and try out the commands.

For more details, see Appendix B, *Command Manual Pages*, or online help.

CHAPTER **3**

User Interface

This chapter provides a brief description of each of the MPSAS commands. Appendix B, "Command Manual Pages" contains the details.

MPSAS has an online help facility. Type **`help`** to show the commands and their operations.

At any time, the user interface is focused on a particular module instance. The name of that module instance, followed by a colon, is your prompt. Being focused on a module instance has the following ramifications:

- While many commands are independent of the modules and are universally available, some are associated with a particular module class; you can only execute them with regard to an instance of that class. By focusing on a module instance, you temporarily add any of its commands to the user interface's repertoire.
- The `help` command shows commands defined by the currently focused module instance in addition to those that are universally available.
- In referring to the currently focused module instance's variables, you need not specify the module instance name.

You can create the MPSAS `MODULE_CMD_PATH` environment variable to specify the order of module instances the user interface examines to match for module commands. If `MODULE_CMD_PATH` does not exist, the user interface examines only the currently focused module instance.

MPSAS initially has its focus on the instance of the `ui` (user interface) module, but you can change the focus with the `focus` command.

Note – If the `ui` module's `isconstprompt` variable value is set to `true`, the prompt changes to `mpsas:`, followed by a newline, which is more suitable for interfacing with other programs. In this case, the `focus` command does not change the prompt.

3.1 Universally Available Commands

This section briefly describes the universally available commands in related groups—roughly in an order that a user learning to use MPSAS may find useful. Refer to Appendix B, *Command Manual Pages*, for details.

- The `help` and `focus` commands, as mentioned above, are universally available.
- The `quit` command causes the simulator to exit (with a particular return code, if desired), returning control to the shell from which it was invoked.
- The `version` command gives information about the version of MPSAS you are running, along with some details on who built it and how.
- The `list` command prints different types of information, such as module variable names, the names of configured module instances, and the names of message types.
- The `load` command loads an executable in ELF format into a memory module, that is, into simulated memory.
- The `load_section` command loads only a section, such as text or data, from an executable. Both commands also load the executable's symbol table into the framework.
- The `symtab` command loads only the symbols from an executable into the framework.
- The `run` command starts the simulated machine once you have loaded a program into the memory of the simulated machine. Execution continues until one of the following situations occurs:
  - The program that runs on the simulated computer hits a breakpoint.
  - The program executes a special trap, which is recognized by the simulator as a request to stop the simulation.
  - You type the `tty` interrupt character (usually Control-C).
  - You type the `stop` command.
  - The module stops the simulation (usually due to an error).
- The `cycle` command causes the simulation to run for only a certain number of cycles. (To step by instruction, use the `cpu` module `step` command, discussed in the next section.)
- The `print` command displays variables and is useful for showing the internal state of modules.
- The `window` command displays the output of user-interface commands in a window.
- The `set` command changes the value of a variable, which you can use to change many aspects of a module's state.

- In general, wherever a value is required by a command, you can use an expression. MPSAS contains a expression evaluator (described in *Expressions* on page 25). If you simply want to know what an expression evaluates to or how it is parsed, use the `expr` command. If you want to know what the numeric value of some SPARC instruction disassembles to, use the `dasm` command.
- MPSAS provides a facility for executing commands when certain conditions, called triggers, occur. The `when` command specifies a trigger to be checked at the end of each cycle, along with a list of commands to be executed if the trigger "fires."
- The `snoop` command specifies a trigger and a list of commands. In this case, the trigger is checked when a message is transmitted between modules. This command is useful for debugging new modules because, when combined with the `msg` command, it displays messages. You can also use `snoop` and `set` to modify the contents of messages.
- The `onstop` command specifies a list of commands to be executed when the simulation stops.
- The `status` command lists the events currently known to the simulator. An event is the order to perform a set of commands when certain conditions are met.
- The `delete` command deletes an event.
- If you do not want an event to be active but would like to recall it later, you can disable it with the `disable` command and then reinstate it with the `enable` command.
- The `msg` command shows the format of a message type or the contents of a message being snooped.
- You can create your own variables with the `var` command and use them to record interesting values, for example, when triggers fire.

The event facility, the expression evaluator for event triggers, user-defined variables, and the set of variables defined by the various modules combine to give you a great deal of power in analyzing your simulated system and the program that runs on it. For example, the following command arranges for the simulator to display the string `user` when `cpu1` enters user mode and `super` when it enters supervisor mode, and then stop in either case:

```
when cpu1.psr.s changes {expr cpu1.psr.s ? "super" : "user"; stop}
```

The following commands arrange for the simulator to count the number of traps of each type caused by the program; `traps` is an array of 256 counters.

```
var add traps unsigned Word 256
when cpu1.trap_count changes \
{set traps(cpu1.latest_trap_num) = traps(cpu1.latest_trap_num) + 1}
```

- The following commands provide features similar to those available in various UNIX shells:
  - The `setenv` command associates text with a name; thereafter, you can expand the name in a command to that text. The `echo` command prints out its arguments to the screen.
  - The `alias` command associates a list of commands with a name, in effect creating a new command.
  - The `unalias` command removes an alias.
  - The `history` command displays the last few MPSAS commands you typed.
- You can store MPSAS commands in files, called scripts, and then execute them, with parameters, with the `file` command.

  The `echo` command causes commands from scripts to be echoed as they are executed.

  The `if` command, particularly useful in scripts, conditionally executes a list of commands. If a script encounters problems, the `flush` command can prevent ensuing commands in the script from being executed.
- The user interface is active even while the simulation is running (although no prompt is displayed). If you want a command to be executed when the simulation stops (rather than at the time the command is entered), precede it with the `wait` command. Typically, a `run` in a script is followed by a `wait` if the `run` is not the last command in the script.
- When tests are automated with the `file` command, the potential problem exists that the program may do something unexpected and never hit any of the breakpoints set for it. For example, it may end up in an infinite loop. To ensure that a test terminates in a reasonable amount of time, you can set watchdog timers with the commands `rtimer` (real time) and `vtimer` (virtual time—only the time the computer spends running MPSAS is considered).
- The `sh` command executes Bourne shell commands or starts a shell.
- If you are running a large program, such as an operating system, on the simulator, you may find a problem that occurs only after many minutes of simulation or after some substantial setup. In that case, you can use the `state` command to save the state of the simulator to the file system right before the interesting part and then restore it later, eliminating the lengthy setup.

  Also, you can direct the simulator to `fork` a copy of itself. After the fork, you will work in the copy until you quit from it, at which point the original simulator returns, looking just like it did before the fork. Using this mechanism, you can quickly perform a number of experiments from that known state.
- You may want to run a program on the simulated machine and capture information about what the machine is doing in a file for later analysis by a program, for example, to see how effective your cache is at keeping the processor off the bus. In this case, you can use the `group` command to associate a name

with a collection of state information and message fields, and then `dump` groups out to a trace file or trace-analyzer program. The `trace` command manages such files and programs.

- The `time` command displays information about the time and other resources the simulator has consumed, as well as information about the rate at which cycles and instructions of the simulated machine are being simulated.
- The `option` command displays or sets parameters that control the behavior of MPSAS in certain situations, such as the printing of expression values. Unlike setting a module variable with the `set` command, which only affects the module that owns the variable, changing an option may affect the entire simulator.

3.2 Module Commands

Modules can add to the base set of commands described in the previous section. These commands are immediately available whenever you are focused on an instance of the module that defines them, or they are present in a module instance specified in the `MODULE_CMD_PATH` environment variable.

The `MODULE_CMD_PATH` environment variable is composed of a list of module instances, separated by colons. When the user interface encounters a module command, that is, the command does not match one of the built-in commands, it checks the module instances specified in `MODULE_CMD_PATH` in the order they are listed. You can include the currently focused module instance in `MODULE_CMD_PATH` by using the special module instance name of period (`.`).

You can execute a module command independently of the focus or `MODULE_CMD_PATH` environment variable by preceding it with the name of the module instance you want to execute the command and a period. In the following example, the `cpu1`'s `breakpoint` command is invoked with the three possible forms:

```
cpu1: breakpoint
No breakpoints currently set.

cpu1: focus ram1

ram1: breakpoint
Unknown command: "breakpoint"

ram1: cpu1.breakpoint
No breakpoints currently set.

ram1: setenv MODULE_CMD_PATH cpu1:.

ram1: breakpoint
No breakpoints currently set.
```

A module command is actually handled by the specified instance of the module, so that, for example, setting a breakpoint by means of `cpu1.breakpoint` does not affect `cpu2` in a configuration with multiple instances of the `cpu` module.

This section only briefly describes each module's commands. Refer to Chapter 7, *Modules*, for details and for information about the modules with which these commands are associated.

3.2.1 `cpu` Commands

The `cpu` module's `breakpoint` command sets, deletes, and lists breakpoints for a CPU. A breakpoint stops the CPU just before executing the instruction at a particular virtual address. The `where` command displays a stack backtrace for the program that is running on the simulated computer.

Whereas the universally available `cycle` command directs the simulation to advance a specified number of cycles, the `cpu` module's `step` command directs the simulation to advance a certain number of instructions. Software developers usually are not interested in what the CPU does on a cycle-by-cycle basis and prefer to see the effects of whole instructions instead.

You can display the in registers (`i0` – `i7`) of a `cpu` module with the `ins` command. Similarly, the `locals`, `outs`, and `globals` commands print out the other register groups. The `regs` command prints out all of the above, along with all of the registers that are accessible in supervisor mode.

The `allstages` command shows the contents of each pipeline stage.

Each cpu module keeps track of all its accesses to memory for each ASI. You can display this information with the accesses command. Similarly, the profile command arranges for counts to be kept of the loads and stores to various address ranges.

The read command displays the contents of virtual memory in various formats, including disassembly. The write command changes the contents of virtual memory.

Sometimes a program that runs on your simulated computer reaches an address without showing how it arrived there. Rather than tracing through the instructions that led the program to that point, you can use the itrace command to set the number of instructions you want to record, then run the simulated system until it arrives at that address and use the itrace command again to print out the most recently executed instructions.

3.2.2 fcpu Commands

The fcpu module (fast CPU) supports all of the cpu module commands, described in the previous section.

3.2.3 fpu Commands

The fpu module's show_fpq command displays the contents of the floating-point queue. The finish_fpop command finishes execution of the oldest floating-point operation in that queue.

3.2.4 sun4c and sun4e mmu Commands

The mmu module used in the sun4c and sun4e architectures defines a single command, xlate, which takes a virtual address and mmu context number and displays the physical address and memory type to which they map.

3.2.5 cmu Commands

The commands for the cmu module of the mbus architecture are:

- xlate, which translates a virtual address to the corresponding physical address. In addition, a number of commands display the various internal and external data structures used by cmu.

- `tables`, which displays translation tables
- `ranges`, which shows how the tables map memory
- `contexts`, which shows `mmu` contexts that have valid mappings; `lines` shows valid cache lines. The `cmu` module keeps extensive statistics.
- `sstat`, which displays statistics about the activity on the processor bus and Mbus
- `clstat`, which shows statistics for all cache lines, such as hits and misses for reads, writes, and read-modify-writes
- `cstat`, which shows the same statistics for the cache as a whole
- `alias_cnt`, which tells how many virtual address aliases were detected on reads and writes
- `mstat`, which displays statistics about `mmu`, such as how many table walks it had to do at each level and the hit rate on the TLB

3.3 Processing of Command Lines

The user interface attaches special meanings to these characters:

```
> | $ ; \
```

To suppress that special meaning, precede the character with a backslash. Also, a backslash as the very last character in a line causes the next line to be considered a continuation of the command (with the backslash and newline removed).

Each time a command is executed, MPSAS processes it as follows:

1. **If the command is read from a script and echoing of script commands has been turned on, MPSAS echoes the command to the screen.**
2. **MPSAS looks at the command name (everything up to the first space, tab, or semicolon) to see if it is an alias; if so, MPSAS replaces the alias name with the alias definition.**
3. **MPSAS looks for a semicolon that represents the end of this command. If it finds one, it cuts off the remainder of the command after the semicolon and processes it later.**
4. **MPSAS expands references to names defined with `setenv` (such as `$foo`). If no such variable exists, the reference expands to nothing.**
5. **MPSAS looks for redirection symbols similar to those in the Bourne shell:**

   a. *mpsas_command* `>` *path* **causes the specified file to be created and the output of the command to be redirected into it.**

b. **_mpsas_command_ `>>` _path_ causes the specified file to be opened and the output of the command to be redirected into it.**

c. **_mpsas_command_ `|` _shell-command_ causes the specified Bourne shell command to be run with the output of the MPSAS command piped into it. A common example is `help | more`.**

If MPSAS finds such a redirection construct, it sets up the specified redirection and then strips the construct from the command.

6. **MPSAS replaces `\`_char_ with _char_.**
7. **MPSAS locates the code for the specified command and executes it.**

If you type a command, the following steps occur prior to those above.

1. **The command is added to the key file and the log file.**
2. **The history is replaced.**
3. **Quick substitution is performed.**
4. **If either of the previous two steps takes place, the resultant command is echoed.**
5. **The command is added to the history buffer.**

Note – The user interface is case-sensitive.

3.3.1 History Mechanism

The MPSAS command `history` facility provides a subset of the features provided by the UNIX C shell `history` mechanism.

TABLE 3-1 shows the `history` event specifiers.

TABLE 3-1 `history` Event Specifiers

Syntax	Interpretation
`!!`	Previous event
`!`_n_	Event _n_

TABLE 3-1 `history` Event Specifiers *(Continued)*

Syntax	Interpretation
`!-`*n*	Current event less *n*
`!`*str*	Most recent event that starts with the pattern *str*
`!?`*str*`?`	Most recent event that contains the pattern *str*

You can suffix the event specifier with an argument operator, which starts with a colon (`:`). TABLE 3-2 shows the argument operators that `history` accepts.

TABLE 3-2 `history` Argument Operators

Syntax	Interpretation
`*`	All arguments
`$`	Last argument
n	Argument *n*, 0 being the command, 1 the first argument, and so on

You can omit the colon for the `*` and `$` operators. If you omit the second exclamation point in the event specifier and specify an argument operator, the event defaults to the previous one. For example, `!$`, `!!$`, and `!!:$` all specify the last argument of the previous command.

To modify the maximum number of events you can store in the history buffer, set the user interface module's `history` variable, which defaults to 100. To display the history buffer, type the `ui` module `history` command.

3.3.2 Quick Substitution

The MPSAS quick substitution facility is similar to the UNIX C shell quick substitution mechanism. The syntax is:

`^`*old-pattern*`^`*new-pattern*`[^]`

MPSAS replaces *old-pattern* in the previous command the first time it is found with *new-pattern*. If *new-pattern* is omitted, MPSAS deletes *old-pattern*.

3.4 Expressions

Many of the commands take expressions as arguments. MPSAS includes an expression-handling package that is patterned after, but not identical to, expression handling for the C programming language.

3.4.1 Data Types

MPSAS supports the four basic data types:

- Unsigned (unsigned 64-bit integers)
- Signed (signed 64-bit integers)
- Float (IEEE double-precision floating point)
- String (dynamically allocated null-terminated char arrays)

3.4.2 Constants

You can enter integer constants in the usual C bases, as characters, and as the words `false` and `true`. For example:

```
ui1: expr 17, 017, 0x17, ’A’, ’\\n’, ’\\102’, false, true0x11  17
’\021’
0xf     15      ’\017’
0x17    23      ’\027’
0x41    65      ’A’
0xa     10      ’\n’
0x42    66      ’B’
0x0     0       ’\000’
0x1     1       ’\001’
```

Notice, however, that you must double each backslash. With that caveat, you enter floating-point and string constants, also, as in C. For example:

```
ui1: expr .75, 4e12, -9.3E-6, "hello, world\\n", "\\"\\\\"
    0.750000
 4.00000e+12
-9.30000e-06
"hello, world\n"
"\"\\"
```

If you have loaded symbols from some executable (for example, with the `load` command), you can reference them in expressions as &*symbol*, for example, `&main`. Because most C compilers prepend each C identifier with an underscore to create the actual symbol name, the expression package looks first for the symbol as you specified it and then, if that label does not exist, for the same label with a leading underscore.

Note – MPSAS loads only relocatable symbols, such as assembly language labels, but not absolute symbols. Automatic variables in C programs do not generate relocatable symbols.

3.4.3 Operators

TABLE 3-3 enumerates the operators supported within expressions. The first column shows the operator, the data types it takes in each argument position, and the data type it returns. For example,

usfc = *us* **?** *usfc1* **:** *usfc2*

means that the `?:` operator returns a result that may be unsigned (u), signed (s), floating-point (f), or string (c), that the first argument must be either unsigned or signed, and that the other arguments may be unsigned, signed, floating point, or string. The last two operands are numbered 1 and 2 so that they can be referred to in the description of the operator, “if *us* is 0 then *usfc2* else *usfc1*.”

The table is arranged in order of increasing precedence; entries not separated by lines are of equal precedence. You can use parentheses in the usual way to change the bindings. To see how an expression is parsed, use the command: `expr -v` *expression*.

All binary operators are left-associative; the `?:` operator is right-associative.

TABLE 3-3 Expression Operators

Operator	Meaning
usfc =*us* ? *usfc1* **:** *usfc2*	If *us* is 0 then *usfc2* else *usfc1*
u =*us1* `||` *us2*	Logical OR of *us1* and *us2* (0 if both are 0, else 1)
u =*us1* && *us2*	Logical AND of *us1* and *us2* (0 if either is 0, else 1)
u =*us1* `|` *us2*	Bitwise OR of *us1* and *us2*. `|` must be escaped so as not to be interpreted as a pipe symbol by user interface command processing.
u =*us1* ^ *us2*	Bitwise XOR of *us1* and *us2*

TABLE 3-3 Expression Operators *(Continued)*

Operator	Meaning
u =*us1* & *us2*	Bitwise AND of *us1* and *us2*
u =*usfc1* **==** *usfc2*	1 if *usfc1* and *usfc2* have the same value, else 0
u =*usfc1* != *usfc2*	1 if *usfc1* and *usfc2* have different values, else 0
u =*usf* `in` *ranges*	1 if *usf* is within one of the specified *ranges* (see below), else 0
u =*usfc1* **<** *usfc2*	1 if *usfc1* is less than *usfc2,* else 0
u =*usfc1* **<=** *usfc2*	1 if *usfc1* is less than or equal to *usfc2*, else 0
u =*usfc1* **>** *usfc2*	1 if *usfc1* is greater than *usfc2,* else 0. > must be escaped so as not to be taken as a redirect symbol.
u =*usfc1* **>=** *usfc2*	1 if *usfc1* is greater than or equal to *usfc2*, else 0
us =*us* **<<** *u*	*us* shifted left *u* modulo 64 bits
us =*us* **>>** *u*	*us* shifted right *u* modulo 64 bits. >> must be escaped so as not to be taken as a Redirect symbol. >> does arithmetic shift if *us* is signed.
usf =*usf1* **+** *usf2*	*usf1* plus *usf2*
usf =*usf1* **-** *usf2*	*usf1* minus *usf2*
usf =*usf1* ***** *usf2*	*usf1* times *usf2* (u/s operands must fit in 32 bits)
usf =*usf1* **/** *usf2*	*usf1* divided by *usf2* (u/s operands must fit in 32 bits)
us =*us1* **%** *us2*	Remainder after dividing *usf1* by *usf2*; operands must fit in 32 bits
sf =**-** *usf*	Zero minus *usf*
u =**!** *us*	Logical negation of *us* (1 if *us* is 0, else 0)
u =**~** *us*	Bitwise negation (1’s complement) of *us*
u =`as_unsigned` *usf*	*usf* reinterpreted as an unsigned integer
s =`as_signed` *usf*	*usf* reinterpreted as a signed integer
f =`as_float` *usf*	*usf* reinterpreted as a floating-point number
u =`to_unsigned` *usf*	*usf* converted to an unsigned integer
s =`to_signed` *usf*	*usf* converted to a signed integer
f =`to_float` *usf*	*usf* converted to a floating-point number
u =*usfc* `changes`	1 if *usfc* has changed from previous value, else 0
u =`asm(`*us*`,` *c*`)`	The numeric instruction that results from assembling at address *us* the assembly language instruction in string *c*

The difference between operators `as_float` and `to_float` (and the differences between their counterparts for the other data types) may not be obvious. `as_float` causes the argument’s bit pattern to be reinterpreted as the bit pattern for a floating-point number. `to_float` causes the argument’s value to be converted to floating-point format.

For example, suppose x is an `UNSIGNED` Word variable and y is a `FLOAT` Word variable. The expression `x == y` does not parse correctly because the expression parser does not allow direct comparison of integers to floating-point numbers. If you know that x is currently being used to hold an IEEE single-precision floating-point number, then to have it interpreted that way, the expression should be `as_float x`

`== y`. If, however, you want to know whether the integer value in x, when converted to floating point, matches y, then the expression should be `to_float x == y`. This expression would be `true`, for example, if x were 4 and y were 4.0.

The operator `as_string` (similar to `as_unsigned`) causes its operand to be reinterpreted as a string. However, reinterpretation of numbers as strings, and vice versa, is only allowed with variables that are written to support the feature, and exactly what "reinterpreted as a string" means depends on how the support for that variable was written. No such variables are included with MPSAS, hence `as_string` is of no use with the simulator.

For the `in` operator, *ranges* is a comma-separated list of ranges, surrounded by square brackets. A range is either a single expression whose value is to be matched or two expressions separated by a colon, in which case the range is the first expression's value through the second expression's value, inclusive. The expression:

```
pc in [&func1, &func3, &func6:&func9-1]
```

is `true` if the value of variable `pc` matches label `func1` or label `func3`, or lies anywhere between label `func6` and one less than label `func9`, inclusive. All expressions in the list must be of the same data type as the first operand.

In general, operators require that their operands be of matching types. For example, adding integers and floating-point operands requires a conversion:

```
ui1: expr (17 % 3) + 2.5
expression parser: cannot mix integers and floating point

ui1: expr to_float(17 % 3) + 2.5
     4.50000
```

However, the expression handler generally converts unsigned operands to signed when they are mixed with other signed operands. Also, the second operand of >> and <<, if signed, is converted to unsigned.

3.4.4 Variables

Names that refer to values that can change are called variables, which you can use freely in expressions. A variable can be any of the following:

- *name* — A user-defined variable or group (see the `var` command on page 218 and the `group` command on page 188)
- [*instance*.]*name* — Part of the state of some module instance

  If no module instance is specified, the currently focused instance is used.
- *msgtype* — `1` if the current message is of the specified message type, else `0`
- *msgtype.field* — A field in a message

Note – Messages are generally of interest only in the development of new modules.

When you supply a variable name, the expression package first looks for a user-defined variable by that name. If there is none, it looks for a user-defined group by that name. And so on down the list.

Within each category, MPSAS performs a minimum-ambiguity lookup. You need not type the full name of the variable, but only enough to unambiguously identify it within the category. For example, if there are exactly two user-defined variables, `num_traps` and `next_value`, then:

- `n` is ambiguous, causing the search to fail with a message to that effect.
- `nu` matches `num_traps`.
- `na` is not found, causing the search to continue with the next category.

This minimum-ambiguity feature is only applied to the last period-delimited segment of a name. For example, `ab` does not match `abc.def`.

Every variable has a data type (as described in *Data Types* on page 25) and belongs to a particular class. The class of a variable generally implies its size, how it looks when printed, how it is set, what it evaluates to when used in an expression, and so on. You can add new classes. The following classes are provided by the framework for general use and show up in various modules:

- `byte` (8 bits)
- `hword` (16 bits)
- `word` (32 bits)
- `lword` (64 bits)
- `bitfield` (part of one of the above)
- `string` (character strings)

The `list` command can show the variables available for a particular module instance. The `msg` command can show the variables (fields) in a particular message type.

Some variables can take parameters, which are expressions separated by commas and placed within parentheses following the variable name. For example, the following expression accesses a particular byte of memory:

```
ram1.bytes(cpu1.l2+3)
```

A variable name plus associated parameters (if any) is called a variable expression; for simplicity, we often refer to variable expressions simply as variables, ignoring the distinction. Parameters can be arbitrarily complex expressions. It is possible to skip a parameter, as in this example, which refers to all of the bytes between two addresses:

```
ram1.bytes(cpu1.l2+3,,cpu1.l3)
```

The number, data types, and allowed values of parameters differ from one parameter to the next. Refer to the description of the variable for details.

The value of an array is the exclusive-OR of the values of all of its elements. This exclusive-OR value serves as a `checksum` on the entire array, allowing it to be used with the `changes` operator to trigger an event if any member of the array changes. Similarly, the value of a group is the exclusive-OR of the values of all of its members.

Note – The exclusive-OR `checksum` may not change if multiple elements or members change in complementary ways between checks; in such situations, the `changes` operator may not notice a change.

3.4.5 Message Information

Four special variable names provide information about messages that are sent between modules, namely:

- `msgsize` — Number of bytes in the message
- `msgdata` — The value of the message data pointer
- `msgdelay` — The delay left for the message (–1 for debug channel)
- `msgtype` —The value of the message type pointer

You can use these message variables only in `snoop` command trigger expressions or in the command list of a snoop. For example, if you want to snoop on all simulation channel packets of size 50 bytes, you can use the following command:

```
snoop msgsize == 50 { msg -v }
```

3.4.6 Invalid Expressions

An expression can be invalid; that is, it has no value at the time. Division by zero produces an invalid result, as does a reference to a message type different from that of the current message (the message that triggers a `snoop` event, causing expressions to be evaluated). Integer `*`, `/`, and `%` that use operands greater than 32 bits also generate an invalid result. A reference to an array element in which the index is too large is invalid. A group with an invalid member is invalid. There may be cases in which part of a module instance's state does not have any meaning; the variable that represents that state may generate an invalid result when evaluated.

Each operator attempts to handle invalid operands in a way that makes sense. For example, arithmetic on an invalid operand yields an invalid result. Logical operators treat an invalid operand the same as `false`. The `?:` operator treats its first operand as a Boolean (so that if it is invalid, then it is `false`) and passes on the selected operand, including its validity or invalidity, ignoring the remaining operand's validity. The `changes` operator never reports a change when its operand is invalid.

Where commands use an expression as a boolean (such as the `if` command), an invalid expression is treated as `false`.

CHAPTER 4

Sample Architectures

MPSAS provides four sample architectures: `simple`, sun4c, sun4e, and `mbus`. The files for each are contained in directories whose names match that of the architecture. These directories are located in the top of the MPSAS source tree. Each directory contains:

- C++ language source code for any module classes specific to that architecture
- A makefile (called `Makefile`) that builds the simulator executable
- A configuration file for each system that the architecture models
- Setup files that are required by the module instances declared in the configuration files

To build a simulator executable, type **`cd`** to change to the architecture's directory and type **`make`**. This step produces an executable file, called `mpsas`. To start the simulator, type its name at the operating system prompt. The simulator uses the system configuration file called `config_file` unless otherwise specified.

4.1 `simple` Architecture

The `simple` architecture contains one computer system simulation, which simulates no specific computer but contains the minimum module instances to create a SPARC computer and run useful programs. It should be chosen for simple SPARC programs that do not need the facilities of an MMU or any special devices. The `simple` system is the fastest of all computer systems included with MPSAS.

The `simple` system contains the following module instances. (Module classes are in parentheses.)

- `serial1` (`serial`)
- `ram1` (`ram`), `trap1` (`trap`)
- `fpu1` (`fpu`), `cpu1` (`fcpu`)

- sigio1 (sigio)
- ui1 (ui)

FIGURE 4-1 illustrates the system. The boxes are module instances. The lines between the boxes represent one or more interfaces. The ui1 and sigio1 module instances are not shown since they are considered part of the framework.

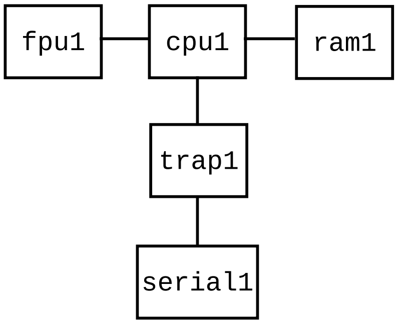

FIGURE 4-1 simple System

Note the following:

Module Instance	Remarks
serial1	Accessible only through the trap1 module and is not directly accessible to cpu1. Does not support interrupts.
ram1	1 Mbyte
trap1	Supports all processor external traps, except the disk operation trap (0xd2).
fpu1	SPARC FPU version 7
cpu1	• SPARC version 8 • Seven register windows • The PSR implementation field is 0x1 • The PSR version field is 0x1 • Prefetching is enabled • Automatic start (default) • Supports interrupts

TABLE 4-1 shows the simple system memory map.

TABLE 4-1 simple System Memory Map

Address Range	Description
00000000 – 000fffff	ram1

4.2 sun4c Architecture

The sun4c architecture contains one computer system simulation. There is no simulation for the sun4c cache and many of the I/O devices.

The sun4c system contains the following module instances. (Module classes are in parentheses.)

- `serial1` (`serial`)
- `simdisk1` (`simdisk`)
- `rom1` (`rom`)
- `eeprom1` and `ram1` (`ram`)
- `mmu1` (`mmu`)
- `trap1` (`trap`)
- `cpu1` (`cpu`)
- `sys1` (`sys4c`)
- `sigio1` (`sigio`)
- `ui1` (`ui`)

FIGURE 4-2 illustrates the system.

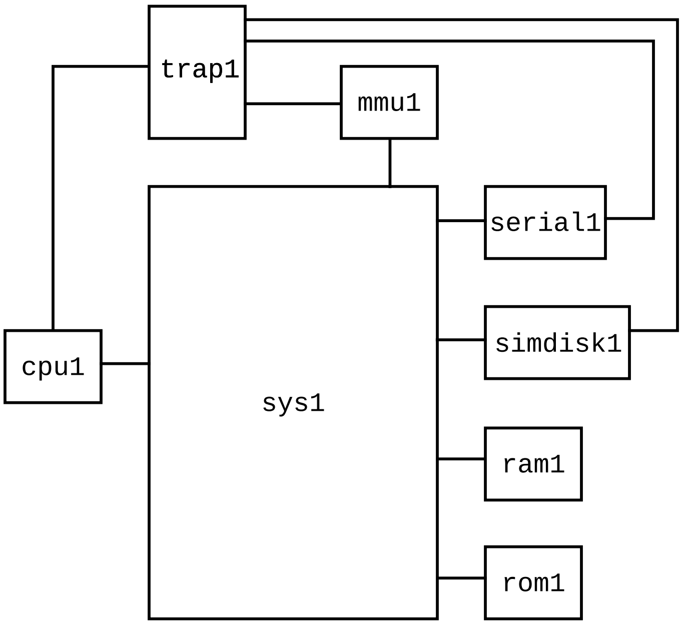

FIGURE 4-2 sun4c System

Note the following:

Module Instance	Remarks
`serial1`	Located in the processor's type 1 device space and system space. Interrupts are on level 12.
`simdisk1`	The initialization file is called `simdisk.init`. This module instance is in the processor's system space. Interrupts are on level 12.
`rom1`	128 Kbytes. Located in the processor's type 1 device space.
`eeprom1`	2 Kbytes. Located in the processor's type 1 device space.
`ram1`	4 Mbytes. Located in the processor's type 0 device space, starting at address 0x0.
`mmu1`	The initialization file is called `mmu.init`. Supports all processor external traps.
`trap1`	Supports all processor external traps.
`cpu1`	• SPARC version 7 • Seven register windows • The PSR implementation field is 0x1 • The PSR version field is 0x1 • Prefetching is enabled • Supports interrupts
`sys1`	No caches are connected.

TABLE 4-2, TABLE 4-3, and TABLE 4-4 define the sun4c system memory maps.

TABLE 4-2 sun4c System Memory Map (Type 1 Device Space)

Address Range	Description
f1000000 – f1ffffff	`serial1` registers
f2000000 – f2ffffff	`eeprom1`
f3000000 – f3ffffff	`sys1` counter and timer registers
f5000000 – f5ffffff	`sys1` interrupt controller registers
f6000000 – f6ffffff	`rom1`

TABLE 4-3 sun4c System Memory Map (Type 0 Device Space)

Address Range	Description
00000000 – 003fffff	`ram1`

TABLE 4-4 sun4c System Memory Map (System Space)

Address Range	Description
30000000 – 3fffffff	`sys1` context register
40000000 – 4fffffff	`sys1` enable register
60000000 – 6fffffff	`sys1` bus error register
a0000000 – afffffff	`simdisk1` registers
f0000000 – ffffffff	`serial1` registers

4.3 sun4e Architecture

The sun4e architecture contains one computer system simulation. There is no simulation for the sun4e cache and many of the I/O devices.

The sun4e system contains the following module instances. (Module classes are in parentheses.)

- `serial1` (`serial`)
- `simdisk1` (`simdisk`)
- `rom1` (`rom`)
- `vram1` and `ram1` (`ram`)
- `vmebus1` (`vmebus`), `s4vme1` (`s4vme`)
- `mmu1` (`mmu`)
- `trap1` (`trap`)
- `cpu1` (`cpu`)
- `sys1` (`sys4e`)
- `sigio1` (`sigio`)
- `ui1` (`ui`)

FIGURE 4-3 illustrates the system.

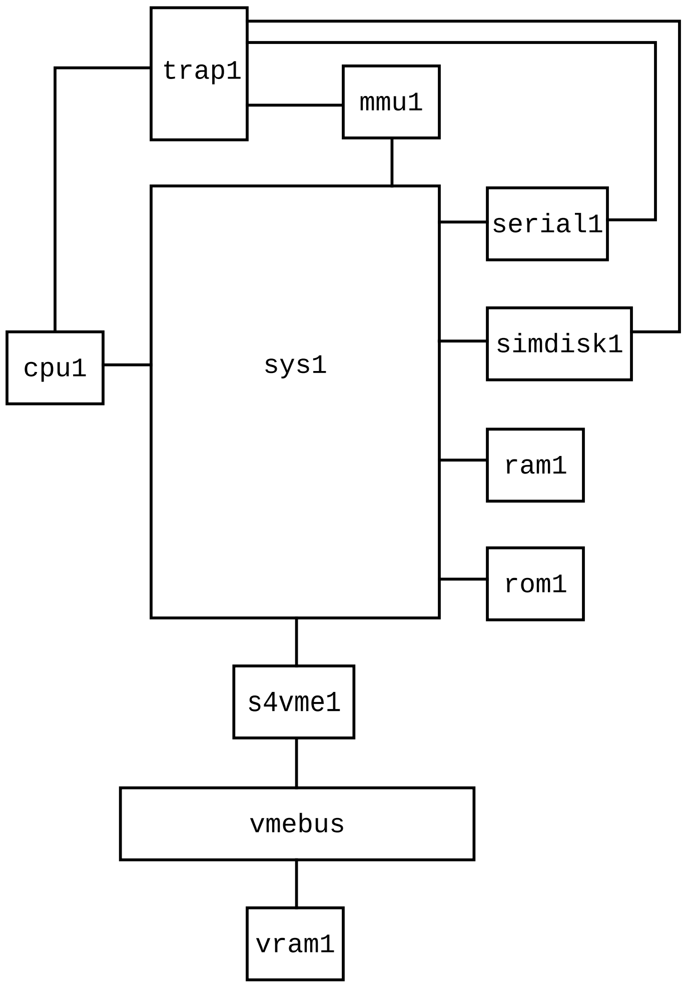

FIGURE 4-3 sun4e System

Note the following:

Module Instance	Remarks
`serial1`	Located in the processor's type 1 device space and system space. Interrupts are on level 12.
`simdisk1`	The initialization file is called `simdisk.init`. This module instance is in the processor's system space. Interrupts are on level 12.
`rom1`	128 Kbytes. Located in the processor's type 1 device space.

Module Instance	Remarks
vram1	8 Kbytes. Located on the VME bus, starting at address 0x00800000.
ram1	4 Mbytes. Located in the processor's type 0 device space, starting at address 0x0.
vmebus1	Contains one 32-bit slave (vram1) and one 16-bit slave (s4vme1).
mmu1	The initialization file is called mmu.init.
trap1	Supports all processor external traps.
cpu1	• SPARC version 7 • Seven register windows • The PSR implementation field is 0x1 • The PSR version field is 0x1 • Prefetching is enabled • Supports interrupts
sys1	No caches are connected.

TABLE 4-5, TABLE 4-6, and TABLE 4-7 define the sun4e system memory maps.

TABLE 4-5 sun4e System Memory Map (Type 1 Device Space)

Address Range	Description
e2000000 – e2ffffff	serial1 registers
e6000000 – e6ffffff	sys1 counter and timer registers
ea000000 – eaffffff	sys1 interrupt controller registers
ec000000 – ecffffff	rom1

TABLE 4-6 sun4e System Memory Map (Type 0 Device Space)

Address Range	Description
00000000 – 003fffff	ram1

TABLE 4-7 sun4e System Memory Map (System Space)

Address Range	Description
30000000 – 3fffffff	sys1 context register
40000000 – 4fffffff	sys1 enable register
60000000 – 6fffffff	sys1 bus error register
a0000000 – afffffff	simdisk1 registers
f0000000 – ffffffff	serial1 registers

4.4 mbus Architecture

The mbus architecture contains two computer system simulations:

- SP (single processor), configuration file: config_file
- DP (dual processor), configuration file: dp_config_file

The SP and DP mbus systems simulate no specific computer; they contain simulations of Mbus, SPARC reference MMU, and a virtual cache.

The SP system contains the following module instances. (Module classes are in parentheses.)

- intr1 (intr)
- timer1 (timer)
- simdisk1 (simdisk)
- serial1 (serial)
- rom1 (rom)
- ram1 (ram)
- trap1 (trap)
- mbus1 (mbus)
- cmu1 (cmu)
- fpu1 (fpu)
- cpu1 (fcpu)
- sigio1 (sigio)
- ui1 (ui)

The DP system contains all of the module instances of the SP system plus the following: trap2 (trap), cmu2 (cmu), fpu2 (fpu), and cpu2 (fcpu).

FIGURE 4-4 illustrates the DP system. The dotted lines in the mbus module instance are *not* paths in the mbus; they are just lines that pass under it. The SP system looks the same as the DP system, except it does not have the extra module instances, as represented by the shaded boxes.

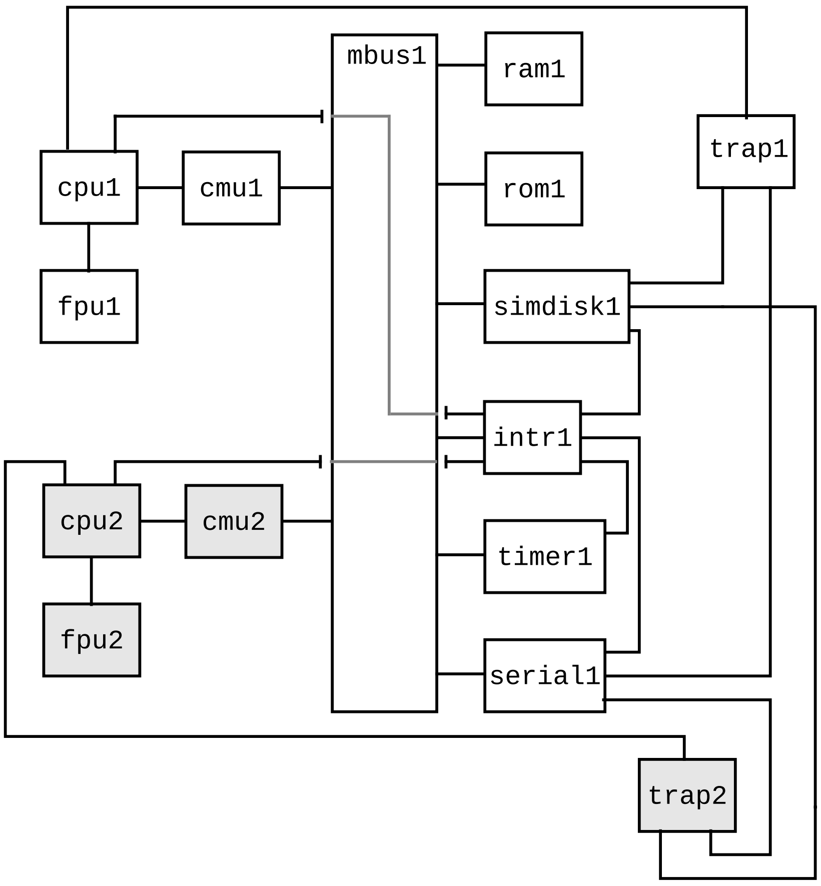

FIGURE 4-4 Mbus Dual Processor (DP) System

Note the following:

Module Instance	Remarks
`intr1`	• The `serial1` device uses bit 1 (mask 0x2) in the interrupt registers. • The `simdisk1` device uses bit 2 (mask 0x4) in the interrupt registers. • The `timer1` device uses bit 3 (mask 0x8) in the interrupt registers.
`timer1`	Contains one timer. Increments the count for each cycle. Directly accessible to processors. Interrupts are on level 14.
`simdisk1`	The initialization file is called `simdisk.init`. Directly accessible to processors. Interrupts are on level 4.

Module Instance	Remarks
`serial1`	Directly accessible to processors. Interrupts are on level 12.
`rom1`	256 Kbytes
`ram1`	2 Mbytes
`trap1` and `trap2`	Supports all processor external traps.
`mbus1`	An internal arbiter enables the register. Processor 0 (`cpu1`) is enabled by default; processor 1 (`cpu2`) is disabled; `simdisk1` is disabled.
`cmu1` and `cmu2`	Cypress type CY605. Mbus module ID of 8 for `cmu1`, 9 for `cmu2`. 64 TLB entries.
`fpu1` and `fpu2`	SPARC FPU version 7
`cpu1` and `cpu2`	• SPARC version 8 • Eight register windows • The PSR implementation field is 0x1 • The PSR version field is 0x1 • Automatic start (default) • Prefetching is enabled • Supports interrupts

TABLE 4-8 define the Mbus SP and DP system memory map.

TABLE 4-8 Mbus SP and DP System Memory Map

Address Range	Description
000000000 – 0001fffff	`ram1`
fe0001008 – fe000100b	`mbus1` arbiter enable register
ff0000000 – ff003ffff	`rom1`
ff1100000 – ff11fffff	`serial1` registers
ff1200000 – ff12fffff	`simdisk1` registers
ff1300000 – ff13fffff	`timer1` registers
ff1400000 – ff14fffff	`intr1` registers]

CHAPTER **5**

Configuration File

When the simulator starts, it parses a configuration file, a readable ASCII text file that specifies which module instances are involved in the simulation and how they are connected.

5.1 Overview

The configuration file is composed of module class declarations, which contain module instance declarations. The classes must be chosen from the available module classes in a simulator binary. Most classes can have any number of instance declarations.

Each instance declaration comprises three optional components:

- **Instance arguments**, which provide a mechanism for the configuration file to control features of a module instance. For example, a memory module may have an instance argument to specify the size of its memory and allow different instances of the same memory module class to have different sizes.
- **Interface declarations**, which specify the interfaces of a module instance. Besides a name and a type, interface declarations are composed of connectivity information, optional arguments, and optional read-only or write-only flags.
- **Global declarations**, which provide a mechanism for modules to share portions of their state.

Following is an example of a configuration file entry for a fictional `processor` module class.

```
module processor {
        instance processor1 {
                args "NUM_REGISTER_WINDOWS 7";
                args "PREFETCH enabled";

                interface ram of type master {
                        connected to ram1:slave;
                        args "ASI 0-250";
                }
                    interface control of type master {
                        connected to control1:slave;
                        args "ASI 251-255";
                }
                interface interrupt of type interrupt {
                        unconnected;
                }
                interface cmd_done of type cmd_done {
                        connected to ui1:cmd_done;
                        write-only;
                }
        }
}
```

The module class `processor` has one instance, called `processor1`. Its arguments specify that the instance should have seven register windows and that prefetching is enabled. The `processor1` instance has three interfaces.

The `ram` interface is of type `master` and is connected to the `slave` interface of the `ram1` module instance. The `ASI` argument controls the behavior of the `master` interface. The `control` interface is also of type `master`. The configuration file syntax allows multiple interfaces of the same type as long as they have different interface names (`ram` and `control` in this case).

The `interrupt` interface is not connected to another interface and does not receive messages. If `processor` sends it a message, a fatal error occurs.

The `cmd_done` interface is a write-only interface that must be connected to a read-only interface.

5.2 Language

This section describes the language of the configuration file, including its syntax and semantics.

5.2.1 Syntax

TABLE 5-1 explains the conventions for the syntax of the configuration file.

TABLE 5-1 Conventions for the Syntax of the Configuration File

Symbol	Meaning
`item`	Literal text
item	Metavariable
=	Definition for a metavariable
`(`*item item*`)`	Grouping
item`*`	0 or more of *item* (optional with repetition)
item`+`	1 or more of *item* (mandatory with repetition)
`|`	Alternation

Comments in a configuration file start with # or `//` and continue to the end of the line. You can use white space (spaces, tabs, and new lines) to separate tokens. A token is one of the following:

- A keyword:

`module`	`unconnected`	`write-only`
`instance`	`connected to`	`public`
`args`	`read-only`	`external`
`interface of type`		

- A punctuation mark:
  `; { } :`
- A name that consists of an alphabetic character or underscore, followed by any number of alphanumeric, underscore, or period characters. Alphabetic characters can be upper or lower case; the case is significant. If you choose to use a keyword as a name, you must enclose it in quotes (") to prevent its interpretation as a keyword.
- A string, which is any text (except newlines and quotes), enclosed in quotes (").

The tokenized file must satisfy this grammar:

config_file = *module**

module = `module` *name* `{` *instance** `}`

```
instance = instance name (; | { argument* interface* global*})

argument = args string ;

interface = interface name of type name { interface_param }

global = public name ; | external name from name:name ;

interface_param = argument* | connection | rw_flag

connection = unconnected ; | connected to name : name ;

rw_flag = read-only ; | write-only ;
```

5.2.2 Semantics

The *config_file* metavariable represents the entire contents of the configuration file.

The *module* metavariable represents a module class declaration. The *name* metavariable is the name of the module class. You can use only module classes that are in the simulator executable; use the `list classes` user-interface command in the simulator to display available module classes. A module class declaration can appear more than once for a specific module class.

The *instance* metavariable represents a module instance declaration. The *name* metavariable is the module instance name and is chosen by the configuration file author for expressing connectivity information in the configuration file and also in user-interface commands, such as `focus`. All instance names must be unique across all module classes.

The *argument* metavariable passes information to a module instance or an interface. See the corresponding module class section in Chapter 7, *Modules*, for the arguments that are used by the modules. The characters between the quotes are passed to the module, and the configuration file syntax enforces no formats. However, all sample module classes use the convention of uppercase characters for literals and lowercase characters for user-supplied values.

The *interface* metavariable represents an interface declaration of a module instance. The first *name* metavariable is the interface name and is chosen by the configuration file author for expressing connectivity information in the configuration file and also in user-interface commands. All interface names must be unique across their module instance.

The second *name* metavariable is the interface type name. See the corresponding module class section in Chapter 7, *Modules*, to see what interface types are supported by the modules.

The *global* metavariable makes a name public (`public` keyword) or gets a reference to a global name (`external` keyword). It is used by a module instance to share portions of its state with other modules. A `public` declaration associates a pointer with *name*. An `external` declaration gets the value of the external pointer (known as *name*) from *name*`:`*name*, where the first name is the module instance and the second name is the public name.

The *interface_param* metavariable specifies connectivity, interface arguments, and read or write flags. If no read-only or write-only flag is specified, interfaces provide a bidirectional, one-to-one connection. A unidirectional, many-to-one connection is allowed if the modules that own the "many" interfaces only send messages to their interface (write-only) and the module connected to the "one" interface only receives messages (read-only). This methodology is useful in situations when multiple modules communicate with another module, which does not reply.

The *connection* metavariable either specifies that an interface is unconnected or identifies the interface to which it is connected. If an interface is identified, it is specified as *name*`:`*name*, where the first name is the module instance and the second name is the interface name. Interfaces can be connected to themselves.

The *rw_flag* metavariable specifies whether an interface is read-only or write-only. If neither flag is specified, the interface is read-write. A read-only interface only receives messages. A write-only interface is only sent by a module. Multiple write-only interfaces can be connected to a single read-only interface.

You can enclose *name* that matches one of the language keywords, such as `module`, in quotes to prevent its interpretation as a keyword.

5.3 Preprocessing

A preprocessor can parse the configuration file before it is parsed by the framework. The default is to preprocess the configuration file by the C preprocessor (`cpp`). The framework obtains the `cpp` path name by examining the `UNIX CPP` environment variable. If `UNIX CPP` does not exist, the path name defaults to `/usr/lib/cpp`.

You can disable the preprocessing or specify the preprocessor path name by using the appropriate MPSAS command-line switches (see Appendix B, *Command Manual Pages*).

If you use `cpp` to preprocess the configuration file, do not use comments that start with a pound sign (#) because `cpp` may interpret them as preprocessor directives, such as #`define`. Use the C style comment symbol (`/* */`) or the C++ style comment symbol (`//`) instead.

CHAPTER 6

Message Types

Each message that is sent between module instances has an associated message type that specifies the format and meaning of the message data.

The modules that the simulator provides use eight message types:

- `coproc_pkt`
- `fpu_pkt`
- `gen_bus_pkt`
- `gen_int_pkt`
- `no_data`
- `sigio`
- `string`
- `trap_pkt`

The `pkt` suffix is an abbreviation for “packet.”

6.1 `coproc_pkt` and `fpu_pkt`

The `coproc_pkt` message type is used by the `cpu` module and a coprocessor module; MPSAS does not provide any coprocessor module examples. The `fpu_pkt` message type is used by the `cpu` module and the `fpu` module. Both message types have the same format and usage; the only difference is their field names.

The `fpu_pkt` format is as follows.

```
struct fpu_packet {
        Word type;
        Word status;
        union {
                Word fsr;
                Word fcc;
                struct {
                        Byte freg;
                        Word data;
                } modreg;
                struct {
                           Word fpop;
                        LWord pc;
                } modfpq;
         } data_u;
};
```

coproc_pkt has the same format, except that the leading f of the field names is replaced with a c.

The type field determines the type of request and the members of the data_u union that are valid, according to TABLE 6-1.

TABLE 6-1 type Field Values for coproc_pkt and fpu_pkt

Type	Interpretation	Valid data_u Members
0x10	Read register	freg data
0x11	Write register	freg data
0x12	Read status register	fsr
0x13	Write status register	fsr
0x14	Read queue	fpop pc
0x15	Write queue	fpop pc
0x16	Read condition codes	fcc

The status field specifies the success of a floating-point or coprocessor unit operation: 0x0 for success, 0x2 for failure.

The fsr field contains the contents of the status register to be read or written.

The fcc field contains the contents of the condition code register to be read or written.

The freg field contains the index of the register to access.

The data field contains the contents of the register to be read or written.

The fpop field contains the SPARC instruction with a floating-point or coprocessor unit operation.

The pc field contains the virtual address of the SPARC instruction in the fpop field.

6.2 gen_bus_pkt

The gen_bus_pkt (generic bus packet) message type is used by a master module to request a data transfer with a slave module, for example, processor reading memory. The request can pass through the bus and other types of intermediary modules. gen_bus_pkt contains the data that define the request as well as status information.

The format is as follows.

```
struct gen_bus_pkt {
        Word  type;
        Word  status;
        Word  extra;
        Word  routeflg;
        Word  asi;
        Word  size;
        LWord paddr;
        LWord vaddr;
        union {
                Byte bytes[1];
                HWord hwords[1];
                Word words[1];
                LWord lwords[1];
        } data;
        Byte other[1];
};
```

The type field specifies the type of transfer. TABLE 6-2 defines the values.

TABLE 6-2 type Field Values for gen_bus_pkt

Type	Interpretation	Symbol
0x0	Write	GEN_BUS_WR
0x1	Read	GEN_BUS_RD
0x2	Read and write	GEN_BUS_RW
0x3	Reference (no data transfer)	GEN_BUS_REF

The status field specifies the status of the request. TABLE 6-3 defines the values.

TABLE 6-3 status Field Values for gen_bus_pkt

Status	Interpretation	Symbol
0x0	Okay	GEN_BUS_OK
0x1	Fault	GEN_BUS_FAULT
0x2	Busy	GEN_BUS_BUSY

The extra field extends the gen_bus_pkt protocol (for example, bus signals). Its exact use is defined by the two interfaces that are involved in its transfer. Many interfaces do not use the extra field at all. If all of the extra information does not fit in the 32 bits that are available, a convention exists to place this information at the end of gen_bus_pkt and to set the extra field to the offset of this information from the start of gen_bus_pkt.

routeflg (route flag) also extends the gen_bus_pkt protocol. It provides another 32 bits of general-purpose space.

asi (address space identifier) optionally supplements the address fields. The least significant byte contains the SPARC ASI when the vaddr field is in use.

The size field specifies the size of the request in bytes. It is the size of the variable-length data array.

The paddr field specifies the physical address of the request and provides space for 64 bits. The validity of the paddr field is determined by the module that sends gen_bus_pkt.

The vaddr field specifies the virtual address of the request and provides space for 64 bits. The validity of the vaddr field is determined by the module that sends gen_bus_pkt.

The data field is a variable-length array of size bytes. It is interpreted as an array of bytes, halfwords, words, or longwords. The C language union notation used in the definition of the data field is slightly misleading in suggesting that the data field is a minimum of 8 bytes in size; its minimum size is 0 bytes.

The other field is a variable array of bytes that starts after the end of the data field and stops at the end of gen_bus_pkt. Again, the C language notation used is slightly misleading in suggesting that the location of the other field is fixed. The other field is present to allow users to observe any information that is located after the data field.

6.3 gen_int_pkt

The gen_int_pkt (generic interrupt packet) message type is used by devices to signal level-sensitive interrupts.

The format is as follows.

```
struct gen_int_pkt {
        Word  irl;
        Word  action;
        Word  irl_valid;
        Word  extra;
};
```

The irl field optionally contains the interrupt request level.

The action field specifies if the interrupt is being set (0x1) or cleared (0x0).

The irl_valid field is nonzero if the irl field contains a valid value; otherwise, it is zero.

The extra field extends the gen_int_pkt protocol.

6.4 no_data

The no_data message type has no associated data. It signals another module that an event has occurred, but no information about that event is required.

6.5 sigio

The sigio message type is used by the sigio module to transfer I/O data to and from other modules.

The format is as follows.

```
struct sigio_msg {
        int     fd;
        short   data_size;
        char    data[1];
};
```

The fd field contains the UNIX file descriptor that is being accessed. The data_size field contains the number of bytes in the data field (a variable length array).

6.6 string

The string message type sends a null-terminated string between modules. Its operation is not visible at the user level.

6.7 trap_pkt

The trap_pkt message type is used by the cpu module and the trap module for external cpu trap instructions.

The format is as follows.

```
struct trap_pkt {
        Word    trap_num;
        Word    out_reg[8];
        Bool    update_out[8];
        Word    local_reg[8];
        Bool    update_local[8];
        Word    global_reg[8];
        Bool    update_global[8];
        Bool    carry_flag;
        Bool    update_carry;
        Bool    super_flag;
        Bool    update_super;
};
```

The `trap_num` field contains the trap number (128 to 255).

The `out_reg`, `local_reg`, and `global_reg` fields contain the values of the `out`, `local`, and `global` registers of the `cpu` module.

The `update_out`, `update_local`, and `update_global` fields control the writing of the `out`, `local`, and `global` registers of the `cpu` module. If `1` (`true`), the corresponding register is written; otherwise, it is not.

The `carry_flag` field and `super_flag` field contain the value of the `carry` flag and `supervisor` flag (respectively) of the `cpu` module's PSR register.

The `update_carry` field and `update_super` field control the writing of the `carry` flag and `supervisor` flag (respectively) of the `cpu` module's PSR register. If `1` (`true`), the flag is modified; otherwise, it is not.

CHAPTER 7

Modules

This chapter describes the modules that MPSAS provides, including their purposes and behavior, as well as any variables that the module defines for accessing the state of a module instance and the module commands that it offers. Also included are details on configuration arguments and the module interfaces. For those interested in the implementation of modules, the source files for each module are listed.

All modules included with MPSAS simulate at the instruction level.

7.1 `cmu`: Cypress 604 / 605 Cache and MMU Module

The `cmu` (cache / MMU) module simulates the Cypress CY604 and CY605, as described in *SPARC RISC User's Guide*, second edition, February 1990, except for known errors in that document and the following features, which are not implemented in the simulation:

- The write buffer
- The multichip operation
- Watchdog, software internal, and software external reset

The write buffer is not implemented; hence, `cmu` does not generate asynchronous errors.

`cmu` contains a SPARC Reference MMU and a virtual cache.

7.1.1 Simulated Behavior

cmu accepts memory access requests from a processor module. Depending on the request and the state of cmu, cmu can interrogate its cache for the data, pass the request to the Mbus, or handle the request itself. cmu simulates the registers and functions of the CY604/CY605, except those mentioned in the previous section.

7.1.2 Variables

The following variables represent the internal state of instances of this module class and are available for use in expressions and in commands that require variables, such as print.

- afar — This Word variable contains the asynchronous fault address register.
- afsr — This Word variable contains the asynchronous fault status register. It is composed of the following fields: uc, to, be, afa, and afo. Modifying the asynchronous fault status register performs no functions.
- ctpr — This Word variable contains the context table pointer register. The least significant 10 bits must always be set to 0.
- ctxtr or context — This Word variable contains the context register. Valid values are from 0 to 4095.
- dptp — This Word variable contains the data access PTP register. It is composed of the addr and valid fields.
- iptp — This Word variable contains the instruction access PTP register. It is composed of the addr and valid fields.
- itr — This Word variable contains the index tag register. It is composed of the itag and dtag fields.
- mptag — This Word array variable contains the Mbus physical tags for each cache line (2,048 lines). It is composed of the following fields: tag, sh, m, and v.
- pvtag — This Word array variable contains the processor virtual tags for each cache line (2,048 lines). It is composed of the following fields: tag, context, v, sh, and sup.
- rpr — This Word variable contains the root pointer register. It is composed of the addr and valid fields.
- rr — These Word variables contain the reset register. It is composed of the following fields: wdr, sir, and ser. The reset functions of the CY604/CY605 are not simulated, so modifying the reset register performs no function.
- scr — This Word variable contains the system control register. For the CY604 mode, it is composed of the following fields: impl, ver, mca, mcm, mv, bm, c, cm, cl, ce, nf, and me. For the CY605 mode, it is composed of the following fields: impl, ver, mca, mcm, mv, mid, bm, c, mr, cm, ce, nf, and me.

- `sfar` — This `Word` variable contains the synchronous fault address register.
- `sfsr` — This `Word` variable contains the synchronous fault status register. It is composed of the following fields: `cbt`, `uc`, `to`, `be`, `lvl`, `at`, `ft`, `fav`, and `ow`.
- `tlb` — This `LWord` array variable contains the TLB contents. It is composed of the following fields: `va`, `ctxt`, `ppn`, `cacheable`, `modified`, `acc`, `st`, and `v`. There is one array element for each TLB entry.
- `trcr` — This `Word` variable contains the TLB replacement register. It is composed of the following fields: `rc` and `irc`. These fields are each 8 bits since the `cmu` supports up to 256 TLB entries.

7.1.3 Commands

In addition to the base set of commands, the following commands are available for dealing with instances of this module class. See *Universally Available Commands* on page 16 for details on invoking these commands.

- `alias_cnt` — This command displays the number of virtual address aliases that are detected.
- `clstat` — This command displays the cache statistics for each line.
- `contexts` — This command displays contexts that have any valid virtual-to-physical address translations.
- `cstat` — This command displays statistics that are related to the cache portion of `cmu`.
- `lines` — This command displays the context, virtual address, physical address, state, and data of all the valid cache lines. The state is one of `EC` (Exclusive Clean), `EM` (Exclusive Modified), `SC` (Shared Clean), or `SM` (Shared Modified).
- `mstat` — This command displays statistics related to the MMU portion of `cmu`.
- `ranges` [*context*] — This command displays the virtual-to-physical address mapping ranges for *context* or for the current context (as stored in the `context` register) if none is specified.
- `sstat` — This command displays the statistics related to the entire `cmu` module.
- `tables` [*context*] — This command displays the translation tables for *context* or for the current context (as stored in the `context` register) if none is specified.
- `xlate` *virtual address* [`user`] — This command displays the physical address corresponding to *virtual address* for the current context (as stored in the `context` register). It also sets the `cmd_result` variable of the `ui` module to the physical address. The virtual address is treated as a supervisor access unless `user` is specified. The translation tables and TLB are used to calculate the physical address regardless of the current `cmu` mode.

7.1.4 Configuration

There are nine interface types: `virtual`, `cache_flush`, `control_registers`, `local`, `bypass`, `diagnostic`, `mbus_master`, `mbus_snoop`, and `cmd_done`. The first six interface types are known as the processor request interfaces and are described as a group in *Interfaces* on page 58. The last three interface types are described individually.

There must be one interface of each of the nine types, except for `mbus_snoop`. If the `cmu` module instance is configured to behave as a CY604, then no `mbus_snoop` interfaces are allowed; otherwise (that is, the CY605 operation), one `mbus_snoop` interface is required.

Instance Arguments

Following are the instance arguments:

`CMU_TYPE` 604|605 — This mandatory argument specifies whether the `cmu` module instance should behave as a CY604 or a CY605.

`MID` *value from 0 to 15* — This mandatory argument specifies the Mbus module ID `cmu` uses when it makes requests on the Mbus. The `mid` field of the SCR is set to the `MID` value. The `MID` for the CY604 must be set to 15.

`NUM_TLB_ENTRIES` *value from 1 to 256* — This mandatory argument specifies the number of TLB entries in the MMU. The actual CY604 and CY605 have 64 entries each.

Note – The following information about this module is for advanced users.

7.1.5 Interfaces

This section describes the interfaces.

Processor Request Interfaces

The processor request interfaces all handle requests from a processor and support simulation and debug channel accesses, which use the gen_bus_pkt protocol. These interfaces use the type, status, asi, vaddr, size, and data fields of the gen_bus_pkt in the normal way and do not use the extra, routeflg, and paddr fields.

The type field values known by all processor request interfaces are GEN_BUS_RD, GEN_BUS_WR, and GEN_BUS_RW. Any other value is a fatal error for the simulation channel.

cmu can only handle one request at a time to its processor request interfaces. If it receives a request but is currently processing a previous one, a fatal error occurs for the simulation channel request.

Virtual Processor Request Interface

The virtual interface handles memory access requests from a processor.

The asi field must be 8, 9, 10, or 11.

A request received on the virtual interface can be handled locally by cmu if the cache is enabled and a cache hit occurs. Otherwise, one or more Mbus transactions occur for operations, such as cache line fills, cache flushes, and table walks.

If a request is completed successfully, it is returned to the virtual interface with status set to GEN_BUS_OK. Otherwise, status is set to GEN_BUS_FAULT, and the cmu synchronous fault status and address registers contain information about the fault.

cache_flush Processor Request Interface

The cache_flush interface handles requests to flush a cmu cache line (ASI 0x10 to 0x14 in CY604/CY605). The least significant 4 bits of asi specify a page (0x0), segment (0x1), region (0x2), context (0x3), or user (0x4) type flush. Other values cause a fatal error.

The gen_bus_pkt size and data fields are ignored.

When the flush completes, the request is returned to the cache_flush interface, with status set to GEN_BUS_OK.

control_registers Processor Request Interface

The control_registers interface handles requests to access the cmu control registers (ASI 0x4 in the CY604/CY605).

The control registers are words. If size is not 4 bytes, the request is sent back to the control_registers interface, an error message is displayed, and the simulation is stopped.

Bits 8 through 15 of vaddr specify the cmu register to access. If they specify an illegal register address, a message is displayed and the request is returned to the control_register interface. Otherwise, the specified register is accessed and the request is returned. In both cases, status is set to GEN_BUS_OK.

local Processor Request Interface

The local interface handles requests to access the Mbus with the Mbus MBL signal active (ASI 0x1 in CY604/CY605). The cache and MMU are bypassed.

If the Mbus request is completed successfully, the request is returned to the local interface with status set to GEN_BUS_OK. Otherwise, status is set to GEN_BUS_FAULT and the cmu synchronous fault status and address registers contain information about the fault.

bypass Processor Request Interface

The bypass interface handles requests to access the Mbus (ASI 0x20 to 0x2f in CY604/CY605). The least significant 32 bits of vaddr are copied to the least significant 32 bits of paddr. Bits 32 through 35 of paddr are set to bits 0 through 3 of asi. The cache and MMU are bypassed.

If the Mbus request is completed successfully, the request is returned to the bypass interface with status set to GEN_BUS_OK. Otherwise, status is set to GEN_BUS_FAULT and the cmu synchronous fault status and address registers contain information about the fault.

diagnostic Processor Request Interface

The diagnostic interface handles requests to access the cmu diagnostic facilities.

A write to ASI 0x3 causes a TLB flush (size and data fields ignored). A read to ASI 0x3 causes a TLB probe (size must be 4 bytes). An access to ASI 0x6 accesses the TLB entries (size must be 4 bytes). An access to ASI 0xe accesses the cache virtual and physical address tags (size must be 4 bytes). An access to ASI 0xf accesses the cache data (size must be from 1 to 8 bytes).

When the access is complete, the request is returned to the diagnostic interface with status set to GEN_BUS_OK.

mbus_master

The mbus_master interface issues requests to access Mbus slaves and supports simulation and debug channel accesses. The mbus_master interface uses the gen_bus_pkt protocol. It uses the type, status, asi, paddr, size, and data fields of gen_bus_pkt in the normal way and does not use the vaddr field. The routeflg contains the mbus multiplexed signals. The extra field contains the mbus physical signals (for responses only).

If the mbus physical signals indicate that a slave is busy (RETRY or RELENQUISH and RETRY), the request is returned to the mbus_master interface.

cmu sets the mid field of the multiplexed signals to the mid field of the SCR on each request that it initiates.

If cmu needs to lock the Mbus, it sets the keep_bus field in the multiplexed signals to 1. If it turns out that the cmu does not need to lock the Mbus, such as if the current transaction fails, it sends a request to the mbus_master interface with the gen_bus_pkt type set to GEN_BUS_REF and the keep_bus field set to 0.

mbus_snoop

The mbus_snoop interface handles requests to snoop the cmu cache and supports simulation and debug channel accesses. This interface uses the gen_bus_pkt protocol and the type, paddr, size, and data fields of the gen_bus_pkt in the normal way, but does not use the status, asi, and vaddr fields. The extra field contains the mbus physical signals. The routeflg contains the mbus multiplexed signals.

After cmu performs the requested snoop, it returns the request to the mbus_snoop interface. It sets the msh and mih fields of the physical signals appropriately. If mih is set, cmu also loads the cache line being snooped into the gen_bus_pkt data field.

cmd_done

The cmd_done interface notifies the user interface that a cmu user command is complete. It supports only debug channel accesses. cmu sends to this interface only and does not receive requests. The cmd_done interface uses the no_data protocol.

Source Files

The source files are:

- `mbus/cmu.c` — The source for the module, but not cache and MMU submodules
- `mbus/cmu.h` — Private declarations for the module
- `mbus/cmu_cache.c` — The source for the cache submodule
- `mbus/cmu_cache.h` — Private declarations for the cache submodule
- `mbus/cmu_mmu.c` — The source for the reference MMU submodule
- `mbus/cmu_mmu.h` — Private declarations for the MMU submodule
- `mbus/cmu_ui.c` — Commands for the `cmu` module (but not submodules)
- `mbus/srmmu.h` — Declarations for any reference MMU

7.2 `cpu`: SPARC Processor Module

The `cpu` module simulates a SPARC IU and supports SPARC architecture version 7 or version 8 operations. We recommend that you use the `fcpu` module instead of the `cpu` module because `fcpu` offers approximately two times higher performance than `cpu` and several enhancements. The `cpu` module source is in the MPSAS source tree but is not compiled.

7.2.1 Simulated Behavior

The `cpu` module has three pipeline stages: fetch, execute, and writeback.

- The fetch stage is used when prefetching is performed. It stores the extra instruction that is prefetched. When `cpu` attempts to access the next instruction, it firsts looks in the fetch stage to see the instruction has already been prefetched.
- The execute stage performs the read and execute stages of a typical SPARC processor. When an instruction is received from memory, it is put in the read stage.
- The writeback stage writes the result of an executed instruction into the register file.

The `cpu` module supports the following features:

- External interrupts
- Communication with an FPU and a coprocessor

The `cpu` module has no ancillary state registers (ASRs) and does not include a cache or MMU.

7.2.2 Variables

The following variables represent the internal state of instances of this module class and are available for use in expressions and in commands that require variables, such as `print`.

- `active` — This `Bool` variable indicates whether `cpu` is active, which means that it will fetch instructions for execution. It is `true` by default.
- `annulled_count` — This read-only `Word` variable keeps track of the number of instructions `cpu` annuls.
- `cycles` — This read-only `Word` variable keeps track of the number of cycles `cpu` executes.
- `executed_count` — This read-only `Word` variable keeps track of the number of instructions the `cpu` executes.
- `fetch`, `execute`, `write` — These `Word` variables contain the pipeline stage information of the corresponding name. Each pipeline stage contains a processor unit (`p-unit`), which has the following members:
  - `annulled` — The flag that indicates whether the instruction is annulled
  - `instr` — The instruction in the pipeline stage
  - `cwp` — The current window pointer
  - `pc` — The `PC` that corresponds to the instruction
  - `r1` — The `RS1` value
  - `r2` — The `RS2` or immediate value
  - `trap_num` — The trap number if trapped is set
  - `trapped` — The flag that indicates whether a trap is taken
  - `wr` — The value to write back

  If a stage is empty, the text `empty stage` is displayed.

- `flags` — This `Word` variable contains fields that indicate what type of instruction is being executed. The fields are:
  - `co_load` — Performing a coprocessor load instruction
  - `co_store` — Performing a coprocessor store instruction
  - `csr_cc` — Performing a CSR condition code instruction
  - `csr_rw` — Performing a CSR read or write instruction
  - `doing_coproc` — Performing a coprocessor operation
  - `doing_float` — Performing a floating-point operation
  - `fetch` — Fetching an instruction
  - `float_load` — Performing a floating-point load instruction
  - `float_store` — Performing a floating-point store instruction
  - `fsr_cc` — Performing an FSR condition code instruction
  - `fsr_rw` — Performing a Floating-point Status Register (`FSR`) read or write instruction
  - `load` — Any type of load instruction
  - `store` — Any type of store instruction

- Integer Unit (IU) registers — The following `Word` variables contain the value of their respective SPARC registers:

  `pc npc y i`<*0-7*> `o`<*0-7*> `l`<*0-7*> `g`<*0-7*> `psr tbr wim sp fp cc`

  `sp` (stack pointer) and `fp` (frame pointer) are equivalent to `l6` and `l7`, respectively.

  When the value of a window register is printed, the current window is assumed unless (*window number*) is appended to the variable name. For example, `i0(4)` displays `i0` for window 4.

  The fields for the `psr` and `tbr` registers can be accessed individually by name. The `psr` fields are `impl`, `ver`, `ec`, `ef`, `pil`, `s`, `ps`, `et`, and `cwp`. The variable `cc` is the `icc` field of the `PSR`. Each bit can be accessed individually as follows: `n`, `z`, `v`, and `c`. The `tbr` fields are `tba` and `tt`.

- `irl` — This `Byte` variable contains the current interrupt request level (`IRL`) (0 to 15).
- `latest_instr` — This read-only `Word` variable contains the latest instruction that entered the execute stage.
- `latest_instr_addr` — This read-only `Word` variable contains the address of the latest instruction that entered the execute stage.
- `latest_mem_addr` — If `latest_mem_addr_valid` is set, this read-only `Word` variable contains the address of the latest memory access or the CTI target address.
- `latest_mem_addr_valid` — This read-only `Bool` variable indicates whether the variable `latest_mem_addr` has a valid value. It becomes valid when a load, store, or CTI instruction finishes executing in the execute stage.
- `latest_mem_data` — If `latest_mem_data_size` is nonzero, this read-only `LWord` variable contains the data for the latest memory access.
- `latest_mem_data_size` — This read-only `Byte` variable contains the data size for the latest memory access.
- `latest_trap_instr` — This read-only `Word` variable contains the latest instruction that caused a trap.
- `latest_trap_num` — This read-only `Byte` variable contains the trap number of the latest trap.
- `latest_trap_pc` — This read-only `Word` variable contains the `PC` of the instruction that caused the latest trap.
- `ls_debug` — This `Bool` variable indicates whether to print debug statements regarding loads and stores the `cpu` module performs.
- `nwins` — This read-only `Word` variable contains the number of register windows and is set by the `NUM_REGISTER_WINDOWS` configuration file argument.

- `prefetch` — This `Bool` variable indicates whether prefetch is enabled. It is controlled by the `PREFETCH` configuration file argument.
- `sanitycheck` — If this `Bool` variable is `true`, then when the `save` instruction is executed in supervisor mode, a check ensures that a subsequent overflow does not write to a supervisor area that does not exist. A message is displayed if there is a conflict.
- `stop_on_reset` — This `Bool` variable controls whether the simulation should stop when a reset trap occurs. It is `true` by default.
- `trap_count` — This read-only `Word` variable keeps track of the number of traps taken.
- `trap_instr` — When a trap is taken, this read-only `Word` variable contains the instruction.
- `trap_npc` — When a trap is taken, this read-only `Word` variable contains the address that is written into `local2` (`%l2`) for the trap handler.
- `trap_pc` — When a trap is taken, this read-only `Word` variable contains the address of the instruction that is written into `local1` (`%l1`) for the trap handler.
- `trap_type` — This `Byte` variable contains the trap type when a trap is taken.
- `watchfetch`, `watchexecute`, `watchwrite` — These `Byte` variables, if nonzero, indicate whether or how to display the corresponding pipeline stage contents each cycle. They can be set to one of the following:
  - 0 — Does not display the stage
  - 1 — Prints the instruction in the stage
  - 2 — Displays the entire `p-unit` contents for the stage

  If the pipeline stage is empty, nothing is displayed.
- `watchallstages` — This `Byte` variable, if nonzero, indicates whether or how to print the contents of all the three pipeline stages (fetch, execute, and write) each cycle. They can be set to one of the following:
  - 0 — Does not display the stage
  - 1 — Prints the instruction in the stage
  - 2 — Displays the entire `p-unit` contents for the stage

7.2.3 Commands

In addition to the base set of commands, the following commands are available for dealing with instances of this module class. See *Universally Available Commands* on page 16 for details.

- `accesses` [*asi*]

  This command displays statistics about the `cpu` accesses over its `master` interfaces. If *asi* is specified, only statistics for that ASI are displayed; otherwise, all ASIs with a `master` interface are displayed with the number of reads, writes, read-modify-writes, and references.

- `allstages`

  This command displays the contents of the fetch, execute, and write stages of the pipeline.

- `breakpoint` [`add` *address* `|` `delete` *number*]

  This command manages the fast breakpoint facility, as follows:

  - `breakpoint` — Shows all current breakpoints
  - `breakpoint add` *address* — Creates a new breakpoint
  - `breakpoint delete` *number* — Deletes the specified breakpoint

  If a breakpoint occurs, the simulation is stopped before that instruction is executed. This command offers lower overhead than the `when` command as a breakpoint (for example, `when cpu1.pc` == *address* {`stop`}). `breakpoint` stops only if the instruction at *address* will be executed; `when` stops even if the instruction is annulled.

  If a breakpoint is set on the first instruction of a function, the stack backtrace displayed by the `where` command may list the wrong caller function. This error occurs because of pipeline effects and does not occur when the pipeline is disabled. To see the correct caller function, step one cycle and invoke the `where` command again.

  The `cmd_result` variable of the `ui` module is set to the breakpoint number that is created when the `add` option is specified.

- `globals`

  This command displays the global registers.

- `ins` [*window number*]

  This command displays the in registers. If *window number* is specified (for example, `ins 4`), the in registers for that window are displayed. Otherwise, the current window registers are displayed.

- `itrace` [[`-v`] [`–l` *count*] | `showsize` | *bufsize*]

  This command manages the instruction trace facility. After instructions (including annulled instructions and instructions that cause traps) are executed, they are placed in a circular buffer. With no arguments, `itrace` shows the contents of the buffer in the order the instructions were executed. If `-v` is specified, a more verbose display is produced. If `-l` is specified, the most recent *count* instruction is displayed. If `showsize` is specified, the size of the buffer is displayed. If *bufsize* is specified (a 32-bit integer expression), the instruction buffer is set to *bufsize* instructions. *bufsize* defaults to zero, in which case no instruction tracing is performed.

- `locals` [*window number*]

  This command displays the local registers. If *window number* is specified, (for example, `locals 4`), the local registers for that window are displayed. Otherwise, the current window registers are displayed.

- `outs` [*window number*]

  This command displays the out registers. If *window number* is specified (for example, `outs 4`), the out registers for that window are displayed. Otherwise, the current window registers are displayed.

- `profile` [`create` *base limit granularity* `| delete | enable | disable | clear | summary`]

  This command manages the profile facility. A profile keeps counts of load or store references to memory ranges. When a profile is created, *base* and *limit* (low and high) virtual addresses to examine are specified. *granularity* specifies the number of bytes each counter covers. The profiling is enabled when first created, and commands exist to enable and disable the gathering of the counts. The summary displays the value of all nonzero counters. If `profile` has no parameters, the current state of the profile is displayed.

- `read` [`-asi` *asi*] `inst|lword|word|hword|byte` *address count*

  This command causes `cpu` to perform a load (read memory), starting at *address* and displaying *count* of one of the following:

  - `inst` — Disassembled instruction
  - `lword` — A long (double) word
  - `word` — A word
  - `hword` — A halfword
  - `byte` — A byte

If *asi* is *not* specified, `read` sets it according to TABLE 7-1 to mimic the default `asi` generated by the SPARC processor for its memory references. The word `data` under the Type column represents all types, except `inst` (that is, `lword`, `word`, `hword`, and `byte`).

TABLE 7-1 Default `asi` Values

`PSR.S`	**Type**	`asi`
`0`	`data`	0xa (user data)
`0`	`inst`	0x8 (user instruction)
`1`	`data`	0xb (supervisor data)
`1`	`inst`	0x9 (supervisor instruction)

If *asi* is specified, `read` overrides the *asi* calculated in the table.

- `regs` [*window number*]

  This command displays all the processor registers, which are `pc`, `npc`, `ins`, `outs`, `locals`, `globals`, `psr`, `tbr`, `sp`, `fp`, `wim`, and `y`. If *window number* is specified (for example, `regs 4`), the registers for that window are displayed. Otherwise, the current window registers are displayed.

- `step` [*count*]

  This command causes the simulation to run for a number of instructions. If *count* is omitted, `step` defaults to one. The `cpu` pipeline must be disabled with the `PIPELINE` configuration file argument, and the `cpu active` variable must be `true`.

- `write [-asi` *asi*] `inst|lword|word|hword|byte` *address data*[`,` *data*]

  This command causes `cpu` to perform a store (write memory), starting at `address` and storing *data*. *data* can be in the following formats:

  - `inst` — A word
  - `lword` — A double word
  - `word` — A word
  - `hword` — A halfword
  - `byte` — A byte

  `write` handles *asi* in the same manner as `read` does.

- `where` [*max number of stackframes*]

  This command displays the entire stack backtrace or only the specified number of stack frames. It makes some assumptions about the register and register window usage of the program that runs on `cpu`. If the program does not support stack frames, `where` produces incorrect results.

7.2.4 Configuration

The cpu interfaces are master, null_master, trap, interrupt, fpu, coproc, and cmd_done. There must be one null_master, one cmd_done, and one interrupt interface. There can be 0 to 256 master interfaces (up to 1 for each ASI), 0 to 128 trap (for trap numbers 128–255) interfaces, 1 fpu interface, and 1 coproc interface.

Instance Arguments

Following are the instance arguments.

- NUM_REGISTER_WINDOWS *number of register windows*

  This mandatory argument specifies the number of register windows for the cpu instance. Valid values are 1 to 16.

- IMPLEMENTATION *processor implementation value*

  This mandatory argument specifies the architecture implementation. Its value is used in the impl field of the PSR. Valid values are 0 to 15.

- VERSION *processor version value*

  This mandatory argument specifies the processor version number. Its value is used in the ver field of the PSR. Valid values are 0 to 15.

- PREFETCH enabled | disabled

  This mandatory argument specifies whether to enable instruction prefetch in the cpu pipeline. The fetch stage is used only when prefetching is performed.

- INITIAL_REG_VALUE *32-bit value*

  This mandatory argument specifies the value to which all the cpu registers are initialized.

- PIPELINE enabled | disabled

  This optional argument specifies whether to enable the cpu pipeline. If the pipeline is disabled, cpu executes only one instruction to completion at a time. The pipeline must be disabled to enable the step command. The default value is disabled.

- DELAYED_WRITE_INSTRUCTION_COUNT *count*

  This optional argument specifies the number of instructions to wait before the writing of delayed-write registers. It is an integer from 0 to 3. By default, if the pipeline is enabled, the count is set to 3; if the pipeline is disabled, it is set to 0.

- NEG_RESULT_OVERFLOW_METHOD 0 | 1

  This optional argument is only used if the SDIV or SDIVcc instructions are valid. The value specified indicates which negative result overflow detection method to use, as follows:

  - 0 indicates use of the following negative result overflow detection method: result < (–2**31).
  - 1 indicates use of the following negative result overflow detection method: result < (–2**31 with a remainder of 0).

  See *The SPARC Architecture Manual/Version 8* for more details on these methods.

If you do not specify the following optional arguments in the configuration file, the default value for each is implemented. If you specify any of them as unimplemented and that instruction is then executed, an illegal_instruction trap occurs.

- SMUL_INSTR implemented | unimplemented
- SMULcc_INSTR implemented | unimplemented
- UMUL_INSTR implemented | unimplemented
- UMULcc_INSTR implemented | unimplemented
- SDIV_INSTR implemented | unimplemented
- SDIVcc_INSTR implemented | unimplemented
- UDIV_INSTR implemented | unimplemented
- UDIVcc_INSTR implemented | unimplemented

Master Interface Arguments

Only one master interface argument applies:

ASI *asi range* — This mandatory argument specifies the ASIs for which the master interface handles cpu requests. *asi range* can be a single value or a range of values from 0 to 255.

For example, ASI 0-3 7-8 specifies ASIs 0, 1, 2, 3, 7, and 8.

Trap Interface Arguments

Only one trap interface argument applies:

TRAP *trap range* — This mandatory argument specifies traps that are to be considered external and handled by the corresponding interface. *trap range* is a range of values from 128 to 255. When an external trap is taken, it is handled by the module to which the trap interface is connected; otherwise, the trap is executed normally.

For example, TRAP 200-202 specifies that when trap 200, 201, or 202 occurs, it is handled by the associated interface.

Note – The following information about this module is for advanced users.

7.2.5 Interfaces

This section describes the interfaces.

master

The master interface issues requests to the memory subsystem. There can be 0 to 256 master interfaces (up to 1 for each ASI). This interface supports simulation and debug channel accesses.

The master interface type uses gen_bus_pkt, the generic bus packet protocol, which is used by the cpu to perform fetches, loads, and stores.

Requests

Each gen_bus_pkt sent represents a single request to memory, where type is one of GEN_BUS_WR, GEN_BUS_RD, or GEN_BUS_RW. status is set to GEN_BUS_OK; the vaddr, size, and asi fields identify the address of the request to be copied to or read from the data field of the packet. paddr is set to the same value as the vaddr. The routeflg and extra fields are not used.

Responses

The master interface receives responses from the memory subsystem. If status is GEN_BUS_OK, the memory response was successful. If status is GEN_BUS_FAULT or GEN_BUS_BUSY, cpu takes a trap. The type of trap depends on the operation cpu was performing.

null_master

There must be one null_master interface, which supports simulation and debug channel accesses.

The null_master interface handles the requests for the ASIs that were not configured with the master interface. For example, if only master interfaces for ASIs 0–127 are configured, then any request to an ASI in the range 128–255 are sent to the null_master interface. null_master should be connected to itself. When null_master receives a request, it prints out a message to indicate that a request was sent to a null interface and the simulation stops.

trap

The trap interface handles external traps—traps that should be handled by other modules). It sends a trap packet (trap_pkt) when an external trap occurs and handles the response. There can be 0 to 128 trap interfaces. Only simulation channel accesses are supported.

Requests

The trap interface sends a trap_pkt for an external trap. It uses the fields trap_num, out_reg, local_reg, global_reg, and carry_flag in the normal way and sets the fields update_out, update_local, update_global, and update_carry to false.

Responses

The trap interface receives responses for external traps. If any of the update_out, update_local, and update_global flags are set, the appropriate register in the current window is overwritten with the field that corresponds to the flag. If the update_carry flag is set, the PSR carry bit is overwritten with the value in carry_flag.

interrupt

The interrupt interface handles external interrupts. There must be one interrupt interface, which supports only simulation channel accesses. It uses the gen_int_pkt protocol.

cpu does not send from this interface; instead, it receives external interrupt requests. Only the irl field is read into the cpu state to take the interrupt.

fpu

There can be zero or one fpu interface. The fpu interface handles requests to the Floating Point Unit (FPU) and its responses. It supports only simulation channel accesses and uses the FPU packet (fpu_pkt) protocol.

Requests

cpu sends a request to the fpu interface when it needs to handle FPU data. The request types are as follows:

- Read or write a floating-point register
- Read or write the floating-point status register
- Read or write the floating-point queue
- Read the floating-point condition codes

The type field of the fpu_pkt indicates the type of request and is set to one of the following values: RDFREG, RDFSR, RDFPQ, WRFREG, WRFSR, WRFPQ, or RDFCC.

Responses

The fpu interface receives the response from the FPU. The status field of the fpu_pkt indicates whether a floating-point exception occurred.

coproc

There can be one coproc interface, which handles requests and responses from the coprocessor and supports only simulation channel accesses. It uses the coprocessor packet (coproc_pkt) protocol.

The behavior is the same as that of the fpu interface, but with coproc instructions.

cmd_done

There must be one cmd_done interface, which cpu uses to notify the user interface that a cpu user command is complete. It only supports the debug channel and sends to this interface but never receives requests. The cmd_done interface uses the no_data protocol.

7.2.6 Source Files

All the files associated with this module are in the sparc/cpu directory, as follows:

- cpu.h — State and register declarations
- cpu_asr.c — ASR handling routines
- cpu_cache.c — Routine that handles the FLUSH instruction
- cpu_execute.c — Routines that execute most of the IU instructions
- cpu_float.c — Routines that execute coprocessor and FPU instructions

- `cpu_load_store.c` — Routines that execute loads and stores
- `cpu_module.c` — Routines that configure the `cpu` module, most `ui` commands, `fetch` stage routines, and `read` stage routines
- `cpu_trap.c` — Trap handling routines
- `cpu_window_reg.c` — Routines that handle the `cpu_window_reg` access class
- `cpu_window_register.h` — Structure that handles the `cpu_window_reg` access class
- `cpu_arch_trap.h` — Trap types definitions
- `cpu_globals.h` — Prototypes for all `cpu` global routines
- `cpu_register.h` — Macros that handle register operations
- `cpu_sparc.h` — Macros that decode SPARC instructions

7.3 `fcpu`: Fast SPARC Processor Module

The `fcpu` module simulates a fast SPARC IU. It supports the SPARC architecture version 7 or version 8 operation.

7.3.1 Simulated Behavior

The `fcpu` module has no pipeline stages, which is the basis for its performance advantage over the `cpu` module.

The `fcpu` module supports the following features:

- External interrupts
- Communication with an FPU
- One instruction prefetch

The `fcpu` module has no ASRs and does not include a cache or MMU. It does not support a coprocessor.

7.3.2 Variables

The following variables represent the internal state of instances of this module class and are available for use in expressions and in commands that require variables, such as `print`.

`lwords`	`bytes`
`words`	`chars`
`instructions`	`doubles`
`hwords`	`floats`

These memory variables provide access to the virtual address space as seen by the `fcpu` module. They are similar to the similarly named memory variables in the `ram` and `rom` modules, except that they accept an optional fourth parameter: the ASI. If the ASI is not specified, it is set according to TABLE 7-2.

TABLE 7-2 Default Memory Variable `asi` Values

PSR.S	Type	asi
0	data	0xa (user data)
0	inst	0x8 (user instruction)
1	data	0xb (supervisor data)
1	inst	0x9 (supervisor instruction)

When these memory variables are used in commands, such as `print`, the `fcpu` module sends debug channel messages to the memory on its `master` interfaces to read or write the required data.

- `annul`

  If this `Bool` variable is true, the next instruction to be executed is annulled.

- `annulled_count`

  This read-only `Word` variable keeps track of the number of instructions annulled by `fcpu`.

- `exec`

  This `Group` variable contains information on the instruction being executed. The `exec` members are:

  - `instr` — `Word` that contains the instruction
  - `pc` — `Word` that contains the address of the instruction
  - `cycle` — `Word` that contains the cycle instruction that completed the execution
  - `annulled` — `Bool` that is set to `true` if the instruction was annulled

- trapped — Bool that is set to true if the instruction caused a trap
- trap_num — Byte that contains the trap number if trapped true

- executed_count

  This read-only Word variable keeps track of the number of instructions executed by fcpu.

- ext_trap_pending

  This read-only signed Word variable contains the trap number if fcpu is waiting for an external trap to complete; –1 otherwise.

- external_count

  This read-only Word variable keeps track of the number of external traps taken.

- fpu_ea

  This Word variable contains the effective address for an FPU load or store operation.

- fpu_Wtmp

  This Word variable stores 32 bits of a 64-bit FPU register access.

- IU registers

  The following Word variables contain the value of their respective SPARC registers.

  pc npc y i<*0-7*> o<*0-7*> l<*0-7*> g<*0-7*> psr tbr wim sp fp cc

  sp (stack pointer) and fp (frame pointer) are equivalent to l6 and l7, respectively.

  When the value of a window register is printed, the current window is assumed unless (*window number*) is appended to the variable name. For example, i0(4) displays i0 for window 4.

  The fields for the psr and tbr registers can be accessed individually by name. The PSR fields are impl, ver, ec, ef, pil, s, ps, et, and cwp. The variable cc is the icc field of the PSR. Each of its bits can be accessed individually as follows: n, z, v, and c. The tbr fields are tba and tt.

- intr_count

  This read-only Word variable keeps track of the number of asynchronous traps (that is, interrupts) that are taken.

- irl

  This Byte variable contains the current interrupt request level (0 to 15).

- latest_instr

  This read-only Word variable contains the latest instruction that began execution.

- `latest_instr_addr`

  This read-only `Word` variable contains the address of the latest instruction that began execution.

- `latest_mem_addr`

  If `latest_mem_addr_valid` is set, this read-only `Word` variable contains the address of the latest memory access or CTI target address.

- `latest_mem_addr_valid`

  This read-only `Bool` variable indicates whether the variable `latest_mem_addr` has a valid value. This variable becomes valid when a load, store, or CTI instruction finishes executing.

- `latest_mem_data`

  If `latest_mem_data_size` is nonzero, this read-only `LWord` variable contains the data for the latest memory access.

- `latest_mem_data_size`

  This read-only `Byte` variable contains the data size for the latest memory access.

- `latest_trap_instr`

  This read-only `Word` variable contains the latest instruction that caused a trap.

- `latest_trap_num`

  This read-only `Byte` variable contains the trap number of the latest trap.

- `latest_trap_pc`

  This read-only `Word` variable contains the `PC` of the instruction that caused the latest trap.

- `master_rcv_routine`

  This read-only `Byte` variable specifies the type of memory response `fcpu` is expecting. The response types are:

  - 0 — No response expected
  - 1 — Expecting an instruction fetch response
  - 2 — Expecting an IU load or atomic load or store response
  - 3 — Expecting an FPU load response
  - 4 — Expecting a store response (IU or FPU)

- `nwins`

  This read-only `Word` variable contains the number of register windows and is set by the `NUM_REGISTER_WINDOWS` configuration file argument.

- `prefetch`

This `Bool` variable indicates whether the one instruction prefetch is enabled. It defaults to the value specified by the `PREFETCH` configuration file argument; if that argument is not present, to `true`.

- `prefetch_instr`

  This `Word` variable contains the prefetched instruction when `prefetch_valid` is `true`.

- `prefetch_instr_pc`

  This `Word` variable contains the address of the prefetched instruction when `prefetch_valid` is `true`.

- `prefetch_valid`

  This `Bool` variable is `true` if the prefetch buffer contains an instruction.

- `sanitycheck`

  If this `Bool` variable is `true`, then when the `save` instruction is executed in supervisor mode, `sanitycheck` checks to ensure that a subsequent overflow does not write to a supervisor area that does not exist. A message is displayed in case of conflict.

- `stop_on_reset`

  This `Bool` variable controls whether the simulation stops when a reset trap occurs. It is `true` by default.

- `trap_count`

  This read-only `Word` variable keeps track of the number of synchronous traps taken.

- `watchexec`

  If this `Bool` variable is `true`, instructions are displayed after execution.

- `watchexternal`

  If this `Bool` variable is `true`, a message is displayed when external traps are taken.

- `watchintr`

  If this `Bool` variable is `true`, a message is displayed when asynchronous traps (that is, interrupts) are taken.

- `watchtrap`

  If this `Bool` variable is `true`, a message is displayed when synchronous traps are taken.

7.3.3 Commands

In addition to the base set of commands, the following commands are available for dealing with instances of this module class. See *Universally Available Commands* on page 16 for details on invoking these commands.

- `accesses` [*asi*]

  This command displays statistics about the `fcpu` accesses over its `master` interfaces. If *asi* is specified, it displays only the statistics for that ASI; otherwise, it displays all ASIs with a `master` interface. It also shows the number of reads, writes, read-modify-writes, and references.

- `breakpoint` [`add` *address* `|` `delete` *number*]

  This command manages the fast breakpoint facility, as follows:

  - `breakpoint` — Shows all current breakpoints.
  - `breakpoint add` *address* — Creates a new breakpoint.
  - `breakpoint delete` *number* — Deletes the specified breakpoint.

  If a breakpoint occurs, the simulation is stopped before that instruction is executed. This command offers lower overhead than using the `when` command as a breakpoint (for example, `when fcpu1.pc` == *address* {`stop`}). `breakpoint` stops only if the instruction at *address* will be executed; `when` still stops even if the instruction is annulled.

  The `cmd_result` variable of the `ui` module is set to the breakpoint number that is created when the `add` option is specified.

- `globals`

  This command displays the global registers.

- `ins` [*window number*]

  This command displays the in registers. If *window number* is specified (for example, `ins 4`), `ins` displays the in registers for that window. Otherwise, it displays the current window registers.

- `itrace` [`–l` *count* `|` `showsize` `|` *bufsize*]

  This command manages the instruction trace facility. After instructions are executed, including annulled instructions and instructions that cause traps, they are placed in a circular buffer. With no arguments, `itrace` shows the contents of the buffer in the order the instructions were executed. If `–l` is specified, `itrace` displays the most recent *count* instructions. If `showsize` is specified, `itrace` displays the size of the buffer. If *bufsize* (a 32-bit integer expression) is specified, `itrace` sets the instruction buffer to *bufsize* instructions. *Bufsize* defaults to zero, that is, no instruction tracing is performed.

The commands in the `itrace` buffer are displayed in the following format:

(*cycle*): *label* : *instruction*

where:

- *cycle* is the cycle count when the instruction completed execution.
- *label* is a symbolic representation of the instruction address.
- *instruction* is the disassembled instruction.

- `locals` [*window number*]

  This command displays the local registers. If *window number* is specified (for example, `locals 4`), `locals` displays the local registers for that window. Otherwise, it displays the current window registers.

- `outs` [*window number*]

  This command displays the out registers. If *window number* is specified (for example, `outs 4`), `outs` displays the out registers for that window. Otherwise, it displays the current window registers.

- `profile [ create` *base limit granularity* `| delete | enable | disable | clear | summary ]`

  This command manages the profile facility. A profile keeps counts of load or store references to memory ranges. When a profile is created, the *base* and *limit* (low and high) virtual addresses to examine are specified. *granularity* specifies the number of bytes each counter covers. The profiling is enabled when first created, and commands exist to enable and disable the gathering of the counts. The summary displays the value of all nonzero counters. If `profile` is specified with no parameters, it displays the current state of the profile.

- `read [-asi` *asi*`] inst|lword|word|hword|byte` *address count*

  This command causes `fcpu` to perform a load (read memory), starting at `address` and displaying *count* of one of the following:

  - `inst` — A word
  - `lword` — A double word
  - `word` — A word
  - `hword` — A halfword
  - `byte` — A byte

  If *asi* is not specified, `read` sets it according to TABLE 7-1 on page 68 to mimic the default `asi` generated by the SPARC processor for its memory references.

  If *asi* is specified, it overrides the *asi* calculated in the table.

- `regs` [*window number*]

  This command displays all the processor registers, which are `pc`, `npc`, `ins`, `outs`, `locals`, `globals`, `psr`, `tbr`, `sp`, `fp`, `wim`, and `y`. If *window number* is specified (for example, `regs 4`), `regs` displays the registers for that window. Otherwise, it displays the current window registers.

- `start`

  This command starts the `fcpu` module. Use it only if the `MANUAL_START` keyword is specified in the module configuration file entry.

- `step` [*count*]

  This command causes the simulation to run for a number of instructions. If *count* is omitted, `step` defaults to 1.

- `where` [*max number of stackframes*]

  This command displays the entire stack backtrace or only the specified number of stack frames. It makes some assumptions about the register and register window usage of the program that runs on `fcpu`. If the program does not support stack frames, `where` produces incorrect results.

- `write` [`-asi` *asi*] `inst|lword|word|hword|byte` *address data*[`,` *data*]

  This command causes `fcpu` to perform a store (write memory) starting at `address` and storing *data* as specified. *data* can be in the following formats:

  - `inst` — A word
  - `lword` — A double word
  - `word` — A word
  - `hword` — A halfword
  - `byte` — A byte

  `write` handles *asi* in the same manner as `read`.

7.3.4 Configuration

The `fcpu` interfaces are `master`, `trap`, `master_access`, `interrupt`, `fpu`, and `cmd_done`. There must be one `cmd_done` and one `interrupt` interface. There can be 0 to 256 `master` interfaces (up to 1 for each ASI), 0 to 128 `trap` (for trap numbers 128–255) interfaces, any number of `master_access` interfaces, and 1 `fpu` interface.

Instance Arguments

Following are the instance arguments.

- `NUM_REGISTER_WINDOWS` *number of register windows*

  This mandatory argument specifies the number of register windows for this `fcpu` instance. Valid values are 1 to 16.

- `IMPLEMENTATION` *processor implementation value*

  This mandatory argument specifies the architecture implementation. *value* is used in the `impl` field of the PSR. Valid values are 0-15.

- `VERSION` *processor version value*

  This mandatory argument specifies the processor version number. *value* is used in the `ver` field of the PSR. Valid values are 0-15.

- `PREFETCH` `enabled | disabled`

  This mandatory argument specifies whether to enable instruction prefetch in the `fcpu` pipeline. Prefetching can be enabled or disabled during the simulation with the `fcpu` `prefetch` variable.

- `INITIAL_REG_VALUE` *32-bit value*

  This mandatory argument specifies the value to which all the `fcpu` registers are initialized.

- `MANUAL_START`

  If this argument is specified, `fcpu` does not start fetching instructions until you execute the `start` command. If this argument is omitted, `fcpu` starts fetching instructions when the simulation is started.

- `DELAYED_WRITE_INSTRUCTION_COUNT` *count*

  This optional argument specifies the number of instructions to wait before the writing of delayed-write registers. It is an integer from 0 to 3. By default, if the pipeline is enabled, the count is set to 3; if the pipeline is disabled, it is set to 0.

- `NEG_RESULT_OVERFLOW_METHOD` `0 | 1`

  This optional argument is used only if the `SDIV` or `SDIVcc` instructions are valid. The value specified indicates which negative result overflow detection method to use, as follows:

  - 0 indicates to use the following negative result overflow detection method: result < (–2**31).
  - 1 indicates to use the following negative result overflow detection method: result < (–2**31 with a remainder of 0).

  See *The SPARC Architecture Manual/Version 8* for more details on these methods.

If you specify any of the above arguments as `unimplemented` and that instruction is then executed, an `illegal_instruction` trap occurs. If you do not specify the following optional arguments in the configuration file, the default value for each is `implemented`. If you specify any of them as `unimplemented` and then execute that instruction, an `illegal_instruction` trap occurs.

- `SMUL_INSTR implemented | unimplemented`
- `SMULcc_INSTR implemented | unimplemented`
- `UMUL_INSTR implemented | unimplemented`
- `UMULcc_INSTR implemented | unimplemented`
- `SDIV_INSTR implemented | unimplemented`
- `SDIVcc_INSTR implemented | unimplemented`
- `UDIV_INSTR implemented | unimplemented`
- `UDIVcc_INSTR implemented | unimplemented`

Master Interface Arguments

Only one master interface argument applies:

`ASI` *asi range* — This mandatory argument specifies the ASIs for which the master interface that handles `cpu` requests. *asi range* can be a single value or a range of values from 0 to 255.

For example, `ASI 0-3 7-8` specifies ASIs 0, 1, 2, 3, 7, and 8.

Trap Interface Arguments

Only one trap interface argument applies:

`TRAP` *trap range* — This mandatory argument specifies traps that are to be considered external and handled by the corresponding interface. *trap range* is a range of values from 128 to 255. When an external trap is taken, it is handled by the module to which the `trap` interface is connected; otherwise, the trap is executed normally.

For example, `TRAP 200-202` specifies that when trap 200, 201, or 202 occurs, it is handled by the associated interface.

Note – The following information about this module is for advanced users.

7.3.5 Interfaces

This section describes the interfaces.

master

The master interface issues requests to the memory subsystem. There can be 0 to 256 master interfaces (up to 1 for each ASI). It supports simulation and debug channel accesses.

The master interface type uses gen_bus_pkt, the generic bus packet protocol, which is used by the fcpu to perform fetches, loads and stores.

Requests

Each gen_bus_pkt sent represents a single request to memory, where type is one of GEN_BUS_WR, GEN_BUS_RD, or GEN_BUS_RW. status is set to GEN_BUS_OK; the vaddr, size, and asi fields identify the address of the request to be copied to or read from the data field of the packet. paddr is set to the same value as the vaddr. The routeflg and extra fields are not used.

Responses

The master interface receives responses from the memory subsystem. If status is GEN_BUS_OK, the memory response was successful. If status is GEN_BUS_FAULT or GEN_BUS_BUSY, cpu takes a trap. The type of trap depends on the operation cpu was performing.

trap

The trap interface handles external traps—traps that should be handled by other modules. It sends a trap packet (trap_pkt) when an external trap occurs and handles the response. There can be 0 to 128 trap interfaces. The trap interface supports only simulation channel accesses.

Requests

The trap interface sends a trap_pkt for an external trap. It uses the fields trap_num, out_reg, local_reg, global_reg, and carry_flag in the normal way and sets the fields update_out, update_local, update_global, and update_carry to false.

Responses

The trap interface receives responses for external traps. If any of the update_out, update_local, and update_global flags are set, the appropriate register in the current window is overwritten with the field that corresponds to the flag. If the update_carry flag is set, the PSR carry bit is overwritten with the value in carry_flag.

master_access

The master_access interface handles requests by other modules to perform debug channel accesses to the memory subsystem as seen by the fcpu module. When the fcpu module receives a message on the master_access interface, it accesses its memory subsystem through its master interfaces and performs the required access. The message is returned to the sender with the request completed.

The gen_bus_pkt protocol is used. The vaddr field specifies the virtual address to access. The size field specifies the number of bytes to transfer. The type field specifies the type of access. The asi field specifies the ASI of the access. The data field transfers the data to or from the memory.

If an error occurs, the gen_bus_pkt status field is set to GEN_BUS_FAULT.

interrupt

The interrupt interface handles external interrupts. There must be one interrupt interface, which supports only simulation channel accesses. It uses the gen_int_pkt protocol.

fcpu does not send from this interface. It receives external interrupt requests. Only the irl field is read into the fcpu state, and it takes the interrupt.

fpu

There can be zero or one fpu interface, which handles requests to and responses from the FPU. It supports only simulation channel accesses and uses the FPU packet (fpu_pkt) protocol.

Requests

fcpu sends a request to the fpu interface when it needs to handle FPU data. The request types are as follows:

- Read or write a floating-point register
- Read or write the floating-point status register
- Read or write the floating-point queue
- Read the floating-point condition codes

The `type` field of the `fpu_pkt` indicates the type of request. It is set to one of the following values: `RDFREG`, `RDFSR`, `RDFPQ`, `WRFREG`, `WRFSR`, `WRFPQ`, or `RDFCC`.

Responses

The `fpu` interface receives the response from the FPU. The `status` field of the `fpu_pkt` indicates whether a floating-point exception occurred.

`cmd_done`

There must be one `cmd_done` interface. The `fcpu` uses this interface to notify the user interface that a `fcpu` user command is complete. It supports only the debug channel and sends to this interface but never receives requests. This interface uses the `no_data` protocol.

7.3.6 Source Files

All the files associated with this module are in the `sparc/fcpu` directory, as follows:

- `fcpu.h` — State and register declarations
- `fcpu_arch_trap.h` — Trap type definitions
- `fcpu_asr.c` — ASR handling routines
- `fcpu_cache.c` — Routine that handles the `FLUSH` instruction
- `fcpu_execute.c` — Routines that execute most of the IU instructions
- `fcpu32_fifo.c` — Routines that implement the delayed-write registers
- `fcpu32_float.c` — Routines that execute coprocessor and FPU instructions
- `fpu_globals.h` — Prototypes for all `fcpu` global routines
- `fcpu32_module.c` — Routines that configure the `fcpu` module
- `fcpu_register.h` — Macros that handle register operations
- `fcpu_sparc.h` — Macros that decode SPARC instructions
- `fcpu_trap.c` — Trap handling routines
- `fcpu32_ui.c` — Routines that implement user-interface commands

- `fcpu32_window_register.h` — Structure that handles the `fcpu_window_reg` access class

7.4 `fpu`: SPARC Floating-Point Unit Module

The `fpu` module simulates a generic SPARC version 7, 8, or 9 FPU.

7.4.1 Simulated Behavior

The `fpu` module connects directly to the `cpu` module's `fpu` interface. The communication between the `fpu` and `cpu` modules models the logical behavior of the hardware interface between the SPARC IU and its FPU. The `fpu` module maintains the following:

- Floating-point registers
- Floating-point status register, including condition codes
- Floating-point trap state
- Execution of floating-point instructions

The floating-point module runs in two different modes:

- **Normal mode** — When the FPU is running in normal mode, floating-point instructions finish executing and update the floating-point register file immediately.
- **Delayed-write mode** — When the delayed-write mode is active, the result of all `fpops` is queued and prevents the register file from being modified. The `FSR` is written immediately (for example, `fcmp` instructions). Later on, you must issue the `finish_fpop` command to write the results to the register file. In this mode, you can control the order that `fpop` operations complete.

The `fpu` module only acts as a slave to the `cpu` module and does not communicate with the `cpu` module.

7.4.2 Variables

The following variables represent the internal state of instances of the `fpu` module class and are available for use in expressions and in commands that require variables, such as `print`.

- `delayed_write_active`

  This read-only `Boolean` variable reflects the state of the delayed-write mode.

- Floating-point registers

  The floating-point registers can be accessed as single-precision values by use of the names `f0-f31`, or as double-precision values by use of the names `df0-df30` for version 7 and version 8 modes and `df0-df60` for version 9 mode. The register numbers for the double-precision values follow the conventions used in assembly language programming and are always even numbers.

- `fregs`, `dfregs`

  These two variable names correspond to the group of registers: `fregs` and `dfregs`. `fregs` and `dfregs` are useful with the `print` command to print out the set of floating-point registers as single-precision or double-precision values, respectively. If used in an expression, they evaluate to `0`.

- `fsr`

  This variable corresponds to the `FSR` and contains a number of bit fields that can be accessed individually, namely, `rd`, `rp`, `tem`, `ns`, `ver`, `ftt`, `qne`, `pr`, `fcc`, `aexc`, and `cexc`.

  For a complete description of the `FSR` and its fields, see the *The SPARC Architecture Manual*.

- `sparc_version`

  This read-only `Byte` variable contains the SPARC version that is being simulated (7, 8, or 9).

- `trap_state`

  This variable represents the state of the FPU with respect to traps. The allowed values are:

  - 0 — No exception
  - 1 — Exception mode
  - 2 — Pending exception

7.4.3 Commands

In addition to the base set of commands, the following commands are available for dealing with instances of the `fpu` module class. See *Universally Available Commands* on page 16 for details on invoking these commands.

- `finish_fpop` *vaddr*

  This command writes the result of an `fpop` to the register file when `fpu` is in delayed-write mode. It causes the FPU to finishing executing the oldest `fpop` (that is, the `fpop` closest to the head of the queue) that matches *vaddr*, a 64-bit integer value. If an error occurs, the user interface module's `cmd_result` variable is set to `1`; otherwise, it is set to `0`.

- `show_wrq`

  This command displays the contents of the delayed-write queue.

7.4.4 Configuration

The `fpu` module communicates over only one interface, which is usually connected to a `cpu` module's `fpu` interface.

Instance Arguments

Following are the instance arguments:

- `FSR_SPARC_VERSION` *7, 8, or 9*

  This argument specifies which version of the SPARC architecture the FPU should simulate. If the value associated with this argument is `8` or `9`, argument `FSR_FPU_VERSION` is required in the configuration file. If the value is `7`, use of `FSR_FPU_VERSION` is a syntax error in the configuration file.

- `FSR_FPU_VERSION` *value from 0 to 7*

  value is placed in the `FSR version` field. If the value associated with the `FSR_SPARC_VERSION` is `7`, this argument is a syntax error in the configuration file.

Note – The remaining information about this module is for advanced users.

7.4.5 Interfaces

The `cpu` interface, the only interface to the `fpu` module, receives messages from and responds to messages from the `cpu` module. The `fpu` module does not communicate with the `cpu` module; it only responds to its messages. The request types that are made to the `fpu` module on this interface are as follows:

- Read a floating-point register
- Write a floating-point register
- Read the floating-point status register
- Write the floating-point status register
- Read the floating-point queue
- Write the floating-point queue
- Read the floating-point condition codes

The `fpu_pkt` protocol is used to communicate with the `cpu` module, as described in Chapter 6, *Message Types*.

7.4.6 Source Files

The source files are:

- `sparc/fpu/fpu.c` — Source code for the floating-point module
- `sparc/fpu/`{`addsub`, `compare`, `div`, `fpu_simulator`, `mul`, `pack`, `unpack`, `utility`}`.c` — Code that emulates the floating-point operations
- `sparc/fpu/fpu.h` — Common macro and structure definitions
- `sparc/fpu/{fpu_simulator, globals, ieeefp, reg}.h` — Declarations the emulator uses
- `sparc/include/fpu_pkt.h` — Definition of message packet used to communicate to the `fpu` module

7.5 `gintr`: sun4m Interrupt Controller Module

The `gintr` module simulates a sun4m interrupt controller and has the following features:

- Twenty-three maskable interrupt devices priority encoded to a 4-bit level (undirected interrupts)
- Support for distribution of interrupts
- Support for one to four processors
- Support for processor-to-processor interrupts (directed interrupts)
- Support for generating interrupts for asynchronous errors (broadcast interrupts)

7.5.1 Simulated Behavior

There are no major differences between the sun4m interrupt controller and the simulated `gintr` module.

7.5.2 Variables

The following variables represent the internal state of instances of the `fpu` module class and are available for use in expressions and in commands that require variables, such as `print`.

The system register variables are as follows.

- `intr_tar_mask_reg`

  This `Word` variable contains the interrupt target mask register and can be written to or read from, using two other addresses: the interrupt target mask set address and the interrupt target mask clear address. The `set` and `clear` functions on the interrupt target mask register can be triggered with these addresses.

- `intr_target_reg`

  This `Word` variable contains the `intr_target_reg`. It contains a `target` field.

- `sys_intr_pend_reg`

  This `Word` variable contains the system interrupt pending register. It contains the following fields: `me`, `i`, `m`, `v`, `fl`, `mi`, `vi`, `t`, `sc`, `a`, `e`, `s`, `k`, `sbus`, and `vme`.

The processor register variable is `intr_pend_reg (`*index*`)`, which is a `Word` variable array that contains the interrupt pending register. *index* ranges from 0 to the number of processors less 1.

7.5.3 Configuration

There are four types of interfaces: `cpu, device, slave`, and `broadcast`. There can be up to 4 `cpu` interfaces, 23 `device` interfaces, 4 `broadcast` interfaces, and any number of `slave` interfaces.

`cpu` Interface Arguments

`INDEX` *value between 0 and 7* is a mandatory entry that specifies the processor index and associates each of the processor registers with its processor interrupt interface. The indices of all `cpu` interfaces must be contiguous.

`device` Interface Arguments

There are two `device` interface arguments:

- `IRL` *value between 1 and 15*

  This mandatory entry specifies the interrupt request level of the device.

- MASK *32-bit value*

  This mandatory entry specifies the device bit mask used in the sys_intr_pend_reg and the intr_tar_mask_reg registers. Normally, only one bit should be set in the value. The same bit cannot be set in more than one device mask, except for the processor timer device masks. Since there is a timer-counter for each processor, the mask for each processor timer is the same.

Note – The remaining information about this module is for advanced users.

7.5.4 Interfaces

The interfaces are as follows.

cpu

The cpu interface informs the processor of interrupt-level changes and supports only simulation channel accesses.

The cpu interface uses the gen_int_pkt protocol and does not use the irl_valid, action, and extra fields. The irl field is set to the greatest irl of all interrupting devices. If no device is interrupting the processor, irl is 0. A request is sent to the cpu interface only when the greatest irl of all interrupting devices changes. No response is expected or allowed.

device

The device interface monitors the status of a device interrupt line and supports only simulation channel accesses.

The device interface uses the gen_int_pkt protocol and does not use the irl, irl_valid, and extra fields. The action field is set to INTERRUPT_SET to signal an active interrupt and is set to INTERRUPT_CLEAR to signal an inactive interrupt. INTERRUPT_SET and INTERRUPT_CLEAR actions field values set and reset bits in the system interrupt pending register. The action field can be set to two other values: INTERRUPT_SET_PROC and INTERRUPT_CLEAR_PROC set and clear bits in the processor interrupt pending register.

slave

The slave interface accepts requests to read and write the addressable registers of gintr and supports only simulation channel accesses. It merely responds and does not initiate requests.

The slave interface uses the gen_bus_pkt protocol. It uses the type, status, paddr, size, and data fields in the normal way and does not use the extra, routeflg, asi, and vaddr fields.

paddr bits 12, 13, 14, and 15 specify the processor; bits 0 through 4 specify the registers. If paddr specifies an unknown processor or register or size is not 4 bytes, the request is returned with status set to GEN_BUS_FAULT. If a write access is done to the read-only register, or vice versa, then a warning message is printed and the request is returned with status set to GEN_BUS_FAULT. Otherwise, the required access is performed and the request is returned with status set to GEN_BUS_OK.

If intr_target_mask_set pseudo-register or intr_target_mask_clear, the pseudo-register is written, the intr_pend_reg register for the target processor is updated, and possibly a new irl is sent to the processor.

broadcast

The broadcast interface broadcasts level-15 interrupts to all processors and supports only simulation channel accesses.

The broadcast interface uses the gen_int_pkt protocol and does not use the irl, irl_valid, and extra fields. The action field is set to INTERRUPT_SET to signal an active asynchronous error interrupt and the INTERRUPT_CLEAR to signal an inactive asynchronous error interrupt.

7.5.5 Source Files

The source files are:

- mbus/gintr.c — Source for the module
- mbus/gintr.h — Private declarations for the module
- mbus/gal_int_stuff.h — Additional values for the action field in the gen_int_pkt (only used in the gint and gtimer modules)

7.6 `gtimer`: sun4m Timer Module

The `gtimer` module simulates the sun4m timer-counter chip. The timer-counters follow the structure of other sun4 architectures. The resolution of the timers is 500 ns.

7.6.1 Simulated Behavior

The `gtimer` module models the sun4m timer chip very closely.

7.6.2 Variables

The following variables represent the internal state of instances of this module class and are available for use in expressions and in commands that require variables, such as `print`.

- `counter`
  `counter(`*i*`)`

  This `Word` variable array contains the `counter` registers. Index *i* is the `counter` for timer *i*. Each element is composed of the bit fields `limit` and `value`.

- `limit`
  `limit(`*i*`)`

  This `Word` variable array contains the `limit` registers. Index *i* is the `limit` for timer *i*. Each element is composed of the bit fields `limit` and `value`.

- `start_stop`
  `start_stop(`*i*`)`

  This `Word` variable contains the `start_stop` register. Index *i* is the `start_stop` for timer *i*. Each element is composed of bit field `run`.

- `sys_counter`

  This `Word` variable contains the `sys_counter` register. It is composed of `limit` and `value` bit fields.

- `sys_limit`

  This `Word` variable contains the `sys_limit` register. It is composed of `limit` and `value` bit fields.

- `timer_config`

  This `Word` variable contains the `timer_config` register. It has four bit fields: `t0`, `t1`, `t2`, and `t3`. `t0` refers to timer 0; `t1` refers to timer 1, and so on.

- `user_timer`
  `user_timer(`*i*`)`

  This `LWord` variable array contains the `user_timer` registers. Index *i* is the `user_timer` for the timer *i*. Each element is composed of the bit fields `limit`, `value`, and `value1`. The number in the `value` bit field and the number `value1` bit field must be concatenated to form the actual value of `user_timer`.

7.6.3 Configuration

There are three types of interfaces: `slave`, `proc_interrupt`, and `sys_interrupt`. There can be any number of `slave` interfaces, but there must be only one `sys_interrupt` interface. There can be as many `proc_interrupt` interfaces as there are processors in the system.

Instance Arguments

The instance arguments are:

- `NUM_TIMERS` *count*

  This mandatory argument specifies the number of processor timers for the `timer` instance being declared. *count* can be 1, 2, or 4.

- `CYCLES_PER_TICK` *count*

  This mandatory argument specifies the number of simulator cycles per timer tick. It can be any nonzero 32-bit value.

Interrupt Interface Arguments

The interrupt interface argument is `TIMER` *value*, which specifies the timer to which the `interrupt` interface corresponds. *value* ranges from 0 to the number of timers less 1.

Note – The remaining information about this module is for advanced users.

7.6.4 Interfaces

This section describes the interfaces.

slave

The slave interface accepts requests to read and write the addressable registers of the timer. It supports simulation and debug channel accesses.

The slave interface uses the gen_bus_pkt protocol. It uses the type, status, paddr, size, and data fields in the normal way and does not use the extra, routeflg, asi, and vaddr fields.

The paddr field specifies which register to access. If size is not 4 bytes or 8 bytes or if the paddr specifies an empty address, the request is returned to the slave interface with status set to GEN_BUS_FAULT. Otherwise, the specified register is accessed according to the gen_bus_pkt type field, and the packet is returned to the slave interface with status set to GEN_BUS_OK.

proc_interrupt

The proc_interrupt interface causes an interrupt and supports only simulation channel accesses.

The interrupt interface uses the gen_int_pkt protocol and does not use the irl, irl_valid, and extra fields. The action field is set to INTERRUPT_SET_PROC to signal an active interrupt and is set to INTERRUPT_CLEAR_PROC to signal an inactive interrupt.

A response to the interrupt request is not required. If one is received, it is sent back to the interrupt interface with a delay of 1.

sys_interrupt

The sys_interrupt interface causes an interrupt and supports only simulation channel accesses.

The interrupt interface uses the gen_int_pkt protocol and does not use the irl, irl_valid, and extra fields. The action field is set to INTERRUPT_SET to signal an active interrupt and is set to INTERRUPT_CLEAR to signal an inactive interrupt.

A response to the interrupt request is not required. If one is received, it is sent back to the interrupt interface with a delay of 1.

7.6.5 Source Files

The source files are:

- `mbus/gtimer.c` — Source for the module
- `mbus/gtimer.h` — Private declarations for the module

7.7 `intr`: Interrupt Controller Module

The `intr` module simulates an interrupt controller and not any particular hardware. It has the following features:

- Thirty-two maskable interrupt devices priority encoded to a 4-bit level
- Support for one to eight processors
- Support for processor-to-processor interrupts (`softint`)

7.7.1 Simulated Behavior

Each interrupt device interrupts at one `IRL`. That device can set and clear its interrupt line. The `intr` module monitors the status of each line. It assigns each device a 32-bit mask (usually with only one bit set), called the device mask. If the bits of the device mask are set in the `device_intr_reg`, that device's interrupt line is active; otherwise, it is inactive. The `device_intr_reg` is not externally accessible.

There are three addressable registers for each processor: `masked`, `pending`, and `softint`. All registers are word size and can be accessed only as a word on a word-aligned address. Address bits 4, 5, and 6 specify the processor; bits 2 and 3 specify the register (`pending` = `00`, `masked` = `01`, and `softint` = `10`). For example, the address of the `masked` register for processor *i* is *i* * 16 + 4.

The `masked` register specifies which interrupt devices are allowed to interrupt the processor. Its format is the same as `device_intr_reg`. If the bits of a device mask are set in the `masked` register, that device can interrupt the processor.

The `pending` register specifies the interrupt status of each device that is not masked in the `masked` register. If the bits of a device mask are set in the `pending` register, that device is interrupting the processor.

The `softint` register allows any processor to interrupt the processor associated with the `softint` register. It is composed of 15 bits, one for each nonzero `IRL`. If bit *i* is set, then `IRL` is active. The `softint` register is similar to an interrupt device; it has a device mask of 0x1 (bit 0). It differs from all other devices since it can interrupt on more than one IRL. If any of the softint bits (bits 1 through 15) are set and bit 0 of the masked register is set, bit 0 of the pending register is set and softint attempts to interrupt the processor at the greatest IRL of the sofint.

The intr module selects the greatest IRL of all the pending interrupt devices and softint and updates the processor with any changes. For example, if no interrupts are active and a serial device of IRL 12 and a timer device of IRL 14 both activate their interrupt lines, the processor sees an IRL of 14. Eventually, it services the timer interrupt, the timer deactivates its interrupt line, and the processor sees an IRL of 12.

7.7.2 Variables

The following variables represent the internal state of instances of this module class and are available for use in expressions and in commands that require variables, such as print.

- device_intr_reg

  This Word variable contains the device interrupt register.

- masked(*index*)

  This Word variable array contains the masked register for each processor. *index* ranges from 0 to the number of processors less 1.

- pending(*index*)

  This Word variable array contains the pending register for each processor. *index* ranges from 0 to the number of processors less 1.

- softint(*index*)

  This Word variable array contains the softint register for each processor. *index* ranges from 0 to the number of processors less 1.

7.7.3 Configuration

There are three types of interfaces: cpu, device, and slave. There can be 0 to 8 cpu interfaces, 31 device interfaces, and any number of slave interfaces.

cpu Interface Arguments

The cpu interface argument is INDEX *value between 0 and 7*, which is a mandatory entry that specifies the processor index. It associates the per-processor registers with their processor interrupt interface. The indices of all cpu interfaces must be contiguous.

device Interface Arguments

The device interface arguments are as follows.

- IRL *value between 1 and 15*

  This mandatory entry specifies the interrupt request level of the device.

- MASK *32-bit value*

  This mandatory entry specifies the device bit mask used in the device_intr_reg, masked, and pending registers. Normally, only one bit should be set in the value. The same bit cannot be set in more than one device mask. softint has bit mask 0x1 reserved for it.

Note – The remaining information about this module is for advanced users.

7.7.4 Interfaces

This section describes the interfaces.

cpu

The cpu interface informs the processor of interrupt-level changes and supports only simulation channel accesses.

The cpu interface uses the gen_int_pkt protocol and does not use the irl_valid, action, and extra fields. The irl field is set to the greatest irl of all interrupting devices. If no devices are interrupting the processor, irl is 0. A request is sent to the cpu interface only when the greatest IRL of all interrupting devices changes. No response is expected or allowed.

device

The device interface monitors the status of a device interrupt line and supports only simulation channel accesses.

The device interface uses the gen_int_pkt protocol and does not use the irl, irl_valid, and extra fields. The action field is set to INTERRUPT_SET to signal an active interrupt and is set to INTERRUPT_CLEAR to signal an inactive interrupt.

`slave`

The `slave` interface accepts requests to read and write the addressable registers of `gintr` and supports only simulation channel accesses. It merely responds and does not initiate requests.

The `slave` interface uses the `gen_bus_pkt` protocol. It uses the `type`, `status`, `paddr`, `size` and `data` fields in the normal way and does not use the `extra`, `routeflg`, `asi`, and `vaddr` fields.

`paddr` bits 4, 5, and 6 specify the processor; bits 2 and 3 specify the register (`pending` = `00`, `masked` = `01`, and `softint` = `10`). If `paddr` specifies an unknown processor or register or `size` is not 4 bytes, the request is returned with `status` set to `GEN_BUS_FAULT`. Otherwise, the required access is performed and the request is returned with `status` set to `GEN_BUS_OK`.

If a `masked` register is written, the `pending` register for its processor is updated and possibly a new `IRL` is sent to the processor.

7.7.5 Source Files

The source files are:

- `mbus/intr.c` — Source for the module
- `mbus/intr.h` — Private declarations for the module

7.8 `mbus`: Mbus Module

The `mbus` module simulates the level 1 and level 2 Mbus as defined in the *SPARC Mbus Interface Specification*, Rev. 1.1. The differences between the specification and the `mbus` module are:

- `mbus` uses a fair, round-robin arbitration mechanism.
- `mbus` does not support reflective memory systems.
- Interrupts and AERR are external to `mbus`.
- `mbus` snoops coherent write invalidate transactions of any size.
- `mbus` supports a maximum of eight masters and eight slaves.

7.8.1 Simulated Behavior

The mbus module optionally supports a built-in or external arbitration enable register; only one can be present. The arbitration enable register specifies which mbus masters are enabled for arbitration. If the register is not present, all mbus masters are enabled for arbitration.

Each mbus master has 1 bit associated with it in the arbitration enable register. If that bit is 0, the master is not granted the mbus. Bit *i* of the register controls the mbus master in slot *i*; bit 0 is the least significant bit of a word. The register is 32 bits.

If the built-in register is activated by appropriate configuration file entries, all processors see it at the same physical address. It only supports word accesses on a word boundary.

If the external register is activated, it is not considered part of the mbus module and has no physical address associated with it by the mbus. The register is actually contained in another module, which provides user interface and processor access to its contents.

7.8.2 Variables

The following variables represent the internal state of instances of this module class and are available for use in expressions and in commands that require variables, such as print.

- enable_reg_mask

  This Word variable contains the value of the arbiter enable register. It only exists when the built-in arbiter enable register is in use.

- locking_master_interface

  If a master has the mbus locked, this read-only string variable is the name of the master interface. Otherwise, it is the string nil. It is updated each cycle.

- master_interface

  If a master has the mbus, this read-only string variable is the name of that master interface. Otherwise, it is the string nil. It is updated each cycle.

- master_mbus_request_mask

  If bit *i* is 1 in this Word variable, the master in slot *i* is waiting for the mbus. This variable is updated each cycle.

- mbus_cycle_type

  This Byte variable contains the 4-bit Mbus transaction type of the most recent mbus access.

- msh_snoop_signal

  The mbus cache consistency snoop code stores the msh value it receives from snoop interfaces in this Bool variable, which is updated each cycle.

- next_master_to_get_mbus

  The mbus arbitration code uses this Byte variable as an index into its array of masters to keep track of which master gets the mbus next. It is updated each cycle.

- num_masters_waiting

  This Byte variable contains the number of mbus masters that are waiting to get the mbus. It is updated each cycle.

- slave_interface

  If the mbus is waiting for a slave to respond to a master's request, this read-only string variable is the name of the slave interface. Otherwise, it is the string nil. It is updated each cycle.

7.8.3 Configuration

There are three types of interfaces: master, slave, and snoop. There can be 0 to 8 masters, 1 to 8 slaves, and 1 snoop interface per master interface.

One type of object shared is used: arb_enable_reg. If it is specified, the external arbiter enable register is activated. It cannot be specified if the built-in arbiter enable register is also activated.

Instance Arguments

The instance arguments are:

- ENABLE_REGISTER_ADDR *36-bit address*
- ENABLE_REGISTER_INITIAL_VALUE *32-bit address*
- ENABLE_REGISTER_STUCKAT_ONE *32-bit address*

These optional arguments specify information about the built-in arbitration enable register and also cause it to become active. They denote the address of the built-in arbiter enable register, its initial value, and a mask of the bits (if any) that are always 1. If bit *i* is 1, the master in slot *i* is enabled for arbitration; bit 0 is the least significant bit.

If ENABLE_REGISTER_INITIAL_VALUE or ENABLE_REGISTER_STUCKAT_ONE is specified, ENABLE_REGISTER_ADDR must be specified. If it is *not* specified, the built-in arbiter enable register is not present and all masters are enabled for arbitration.

master Interface Arguments

The master interface arguments are as follows.

- SLOT *value between 0 and* 31

  This mandatory argument specifies the slot associated with the master interface. The slot is used by the arbiter enable register and to match a master interface with its snoop interface.

- MID *value between 0 and 15*

  This optional argument specifies the mbus module ID of the master. If it is provided, the mbus module adds MID into all requests made for the master. If MID is not provided, the module connected to the master interface must load its MID into each of its requests.

- MAKE_MBUS_SIGNALS

  This optional argument specifies that the master does not generate mbus multiplexed signals; therefore, the mbus module must perform this task. This argument is valid for simple masters only.

- ALWAYS_ENABLED

  This optional argument specifies that the master is always enabled for arbitration independent of the contents of the arbitration enable register. It is significant if the built-in or external arbitration enable register is in use.

slave Interface Arguments

The slave interface arguments are as follows.

- ADDR_BASE *36-bit value*
  ADDR_MASK *36-bit value*

  These mandatory entries specify the ranges of addresses to which a slave responds. The range of each slave must be distinct. ADDR_BASE specifies the lowest address in the range. ADDR_MASK specifies which bits of the address are examined. If the address bitwise AND'd with the mask is equal to the base, then that address is inside the range.

- MEMORY

  One of the mbus slaves must specify this argument. Mbus coherent invalidate transactions are acknowledged by the memory slave.

- MAKE_MBUS_SIGNALS

  This optional argument specifies that slave does not generate mbus physical status signals; therefore, the mbus module must perform this task. This argument is valid for simple slaves only.

- DEBUG_SUPPORTED

  This optional argument specifies that slave supports debug channel accesses.

snoop Interface Arguments

The snoop interface argument is SLOT *value between 0 and 7*, which is a mandatory argument that specifies the slot associated with the snoop interface. The slot is used to match a snoop interface with its master interface.

Note – The remaining information about this module is for advanced users.

7.8.4 Interfaces

This section describes the interfaces.

master

The master interface handles requests to access mbus slaves and supports only simulation and debug channel accesses. It uses the gen_bus_pkt protocol.

Requests

mbus queues its requests until the master is granted the mbus. It supports only one outstanding request per master at a time.

mbus uses the type, status, paddr, size, and data fields of the gen_bus_pkt in the normal way and does not use the asi, vaddr, and extra fields.

The route field of the gen_bus_pkt contains some of the Mbus multiplexed signals. The format is as follows:

```
struct multiplexed_signals {
        u_int   pad    :11;     /* unused */
        u_int   mbl     :1;     /* boot mode/local mode */
        u_int   mid     :4;     /* Module Identifier */
        u_int   va      :8;     /* vaddr[19:12] */
        u_int   sup     :1;     /* Supervisor access */
        u_int   c       :1;     /* cacheable */
        u_int   lock    :1;     /* bus lock */
        u_int   keep_bus:1;     /* allows master to lock Mbus */
        u_int   type    :4;     /* Transaction type */
};
```

The `multiplexed_signals` members are used in a manner consistent with the Mbus specification, except that a value of `1` is always considered active and `0` is considered inactive.

If the `keep_bus` bit is set, the master locks the bus so that only it can access the bus next. It is similar to the `mbb` physical signal of the real Mbus; `mbb` is not in the simulation. The `mbb` signal is asserted until the master wants to free the bus, but the `keep_bus` signal is only asserted when the master wants the bus on the next cycle.

All 16 possible values of `type` are legal. The `mbus` module interprets only the values specified in TABLE 7-3; the others are passed to the slave, as specified by `paddr`.

TABLE 7-3 Defined Mbus Transaction Types

`mbus` **Transaction Type**	**Value**
`MBUS_WRITE`	0
`MBUS_READ`	1
`MBUS_COHERENT_INVALIDATE`	2
`MBUS_COHERENT_READ`	3
`MBUS_COHERENT_WRITE_INVALIDATE`	4
`MBUS_COHERENT_READ_INVALIDATE`	5
`MBUS_MEM_REF`	6

All of the above types, except `MBUS_MEM_REF`, are defined by the Mbus specification. `MBUS_MEM_REF` is used when the `gen_bus_pkt type` is set to `GEN_BUS_REF` (no data transfers, just a memory reference). It is useful to free a previously locked `mbus`.

The `mbus` module does not use the `c`, `lock`, `mbl`, `va`, `sup`, and `mid` bits, which are provided for the slave to observe.

The `MID` must be set by the master on each transaction or specified in the `master` interface declaration in the `mbus` declaration of the configuration file.

The master must set the gen_bus_pkt type to match the mbus transaction type according to TABLE 7-4.

TABLE 7-4 Corresponding Mbus and Gen_bus_pkt Types

mbus **Transaction Type**	gen_bus_pkt **Type**
MBUS_WRITE	GEN_BUS_WR
MBUS_READ	GEN_BUS_RD
MBUS_COHERENT_INVALIDATE	GEN_BUS_REF
MBUS_COHERENT_READ	GEN_BUS_RD
MBUS_COHERENT_WRITE_INVALIDATE	GEN_BUS_WR
MBUS_COHERENT_READ_INVALIDATE	GEN_BUS_RD
MBUS_MEM_REF	GEN_BUS_REF

If the master specifies an address that does not match any mbus slave, the mbus module sets the gen_bus_pkt status to GEN_BUS_FAULT, sets the mbus status to time-out, and returns the request to the master.

Messages

A mbus master receives a message only in response to a previous mbus request it made to access a slave. It uses the type, status, paddr, size, and data fields in the normal way and does not use the asi, vaddr, and routeflg fields.

The extra field contains some of the Mbus physical signals. The format is as follows:

```
struct physical_signals {
        u_int   pad    :27;     /* unused */
        u_int   msh     :1;     /* (level 2) shared block */
        u_int   mih     :1;     /* (level 2) Memory inhibit */
        u_int   status  :3;     /* slave status */
};
```

The physical_signals members are used in a manner consistent with the Mbus specification, except that a value of 1 is always considered active and 0 inactive.

The msh signal is 0 for unshared and 1 for shared. The mih signal is used only by the mbus module and the snoop interfaces and is not valid.

The mrdy, mrty, and merr signals encode the status of the slave access. TABLE 7-5 lists the status codes.

TABLE 7-5 Mbus Slave Access Status Codes

merr	mrdy	mrty	mbus **Status**
0	0	0	Idle
0	0	1	Relinquish and Retry
0	1	0	Valid (request successful)
0	1	1	Reserved
1	0	0	Bus error
1	0	1	Timeout
1	1	0	Fatal (uncorrectable error)
1	1	1	Retry (correctable error)

If the status is Retry or Relinquish and Retry, the master must immediately send the original request back to the mbus. If the original mbus transaction type was MBUS_COHERENT_INVALIDATE and status is Relinquish and Retry, the master must change the transaction type to MBUS_COHERENT_READ_INVALIDATE.

Support for Simple Masters

Masters that do not want to generate the mbus multiplexed signals when making an mbus request can specify the keyword MAKE_MBUS_SIGNALS in their mbus interface declaration in the configuration file.

The mbus looks at gen_bus_pkt and sets the mbus transaction type according to TABLE 7-6.

TABLE 7-6 Mbus Type Assignment for Simple Masters

gen_bus_pkt **Type**	mbus **Type**
GEN_BUS_RD	MBUS_READ
GEN_BUS_WR	MBUS_WRITE
GEN_BUS_RW	MBUS_WRITE
GEN_BUS_REF	MBUS_MEM_REF

The other fields of the multiplexed signals are all set to 0, except for the MID, which must be specified in the configuration file.

When a master receives a response, it examines the status field of gen_bus_pkt to determine the status of a request. The mapping between possible mbus status (mrdy, mrty, and merr) is specified in the next section on the slave interface.

This translation facility allows a simple master to be connected directly to mbus without knowledge of the special mbus use of the gen_bus_pkt.

slave

The slave interface connects to mbus slaves and supports simulation and debug channel accesses.

One slave must be identified as the memory slave since it acknowledges coherent invalidate transactions on mbus. Each slave responds to a unique range of the address space, as specified in the configuration file.

The slave interface uses the gen_bus_pkt protocol.

Messages

A slave receives a message from the mbus when a master attempts to access it. The type, paddr, size, and data fields of the gen_bus_pkt are set according to the standard gen_bus_pkt protocol; the slave interface does not use the extra field. The routeflg contains the mbus multiplexed signals described above and is set according to those conventions.

The mbus transaction information is specified by the mbus multiplexed signal type field (4 bits). Slaves can look at this field to determine the exact transaction. Many slaves need not know the exact transaction—for example, MBUS_READ and MBUS_COHERENT_READ are handled the same way by the slave. They can examine the gen_bus_pkt type field since the master sets this field as well as the mbus type.

Responses

After a slave has performed the transaction from a previous request, it sends a response back to the mbus. The slave must set the physical signals, described above, to indicate the result of the request.

If the request is successful, the physical signal status should be set to valid; the gen_bus_pkt status should be set to GEN_BUS_OK.

If the request is to an invalid address within the slave and the slave needs to inform the master of this, it sets the gen_bus_pkt status to GEN_BUS_FAULT and sets the physical signal status to timeout.

If the slave is busy, it can set the gen_bus_pkt status to GEN_BUS_BUSY and the physical signal status to relinquish and retry. Alternatively, the slave can queue the message internally and send a response to mbus when it is no longer busy. mbus is inactive until the slave responds. The first method has lower performance but is simpler and has less potential for deadlocks on mbus.

Support for Simple Slaves

Slaves that do not want to generate mbus physical signals when responding to a mbus request can specify the keyword MAKE_MBUS_SIGNALS in their mbus interface declarations in the configuration file.

mbus looks at the gen_bus_pkt status and converts it to the mbus status (the mrdy, mrty, and merr bits) according to TABLE 7-7.

TABLE 7-7 gen_bus_pkt Status Assignment for Simple Slaves

gen_bus_pkt **Status**	mbus **Status**
GEN_BUS_OK	Valid
GEN_BUS_FAULT	Bus error
GEN_BUS_BUSY	Relinquish and retry

The mapping of the mbus transaction types and mbus response status to standard gen_bus_pkt status allows a simple slave to be connected directly to the mbus without knowledge of the special mbus use of gen_bus_pkt.

snoop

Each master interface can have one snoop interface, which supports simulation and debug channel accesses. The snoop interface uses the gen_bus_pkt protocol and is connected to cache modules.

If a master acquires mbus and it is a MBUS_COHERENT_INVALIDATE, MBUS_COHERENT_READ, MBUS_COHERENT_READ_INVALIDATE, or MBUS_COHERENT_WRITE_INVALIDATE transaction, the transaction is snooped. Before the request is sent to the addressed slave, it is sent to each snoop interface in turn (except the snoop interface of the master that originated the request). Each cache examines the gen_bus_pkt, possibly modifies it, and then returns it to the snoop interface.

The type, paddr, size, and data fields of the gen_bus_pkt are set according to the standard gen_bus_pkt protocol. The extra field contains the mbus physical signals. The routeflg contains the mbus multiplexed signals as set by the master. The behavior of a cache module that receives a message on its snoop interface depends on the mbus transaction type encoded in the multiplexed signals.

Coherent Invalidates, Coherent Writes, and Invalidates

The cache must invalidate the cache line that contains `paddr`.

Coherent Reads

If the cache does not contain the line that contains `paddr`, it simply returns the message. If the cache does have the line but does not own it, it sets the `msh` bit to 1 and returns the message. If the cache owns the line, it sets `msh` and `mih` to 1, copies the line into the data field of the `gen_bus_pkt`, and returns the message.

Coherent Reads and Invalidates

Coherent reads and invalidates are handled in the same manner as `MBUS_COHERENT_READ`, except that the line is invalidated and `msh` does not need to be set.

7.8.5 Source Files

The source files are:

- `mbus/mbus.c` — Source for module
- `mbus/mbus.h` — Structures, macros, and definitions useful to modules that connect to `mbus`
- `mbus/mbus_private.h` — Declarations used only by the `mbus` module

7.9 `mmu`: sun4c/sun4e MMU Module

The `mmu` module simulates the MMU of the sun4c (SPARCstation 1 and 1+) and sun4e (SPARCengine 1e) systems. This two-level MMU is very simple and does not contain a cache or hardware table walk.

The context register of the sun4c/sun4e MMU is in the `sys` module, not the `mmu` module.

7.9.1 Simulated Behavior

The `mmu` module contains a segment map and a page map, which are consulted when a virtual-to-physical address translation is requested. Simulator programmers must maintain the maps to get the desired address translations.

The translation maps are accessible to the CPU via ASI 0x3 for the segment map and ASI 0x4 for the page map. The bits of the virtual address that select the segment entry and page entry during virtual-to-physical translations, along with the context register value, are used to select their respective entry for direct access to the translation maps.

7.9.2 Variables

The following variables represent the internal state of instances of this module class and are available for use in expressions and in commands that require variables, such as `print`.

- `pme`
  `pme (`*index*`)`

This `Word` variable array contains all the PMEs in the page table. *index* specifies the PME to access and ranges from 0 to 8,191 (the number of PMEs less 1). Each PME is composed of the following bit fields: `valid`, `write`, `super`, `uncacheable`, `type`, `acc`, `mod`, and `ppn`.

- `sme`
  `sme (`*index*`)`

  This `Byte` variable array contains all the SMEs in the segment table. *index* specifies the SME to access and ranges from 0 to 32,767 (the number of SMEs less 1).

7.9.3 Commands

In addition to the base set of commands, the following command is available for dealing with instances of this module class:

`xlate` *virtual address context* — This command displays the physical address corresponding to *the virtual address* and *context* expressions. It does not modify the state of `mmu`. The `cmd_result` variable of the `ui` module is set to the physical address if the translation succeeds.

See *Universally Available Commands* on page 16 for details.

7.9.4 Configuration

There are three types of interfaces: `virtual`, `physical`, and `dup_context0`. There must be one `virtual` interface, one `physical` interface, and any number of `dup_context0` interfaces.

Instance Arguments

The instance argument is:

mmu initialization filename

This mandatory argument specifies the path name of the `mmu init` file, which contains the values of the segment and page tables. The file is read during configuration and provides initial virtual-to-physical mapping for `mmu`.

The `mmu init` file contains the entire contents of the segment table (32 Kbytes), followed by the entire contents of the page table (32 Kbytes).

A utility program, called `mmugen`, creates an `mmu init` file with a default mapping of the first 4 Mbytes of virtual address space mapped to the first 4 Mbytes of physical address space in context 0.

Note – The remaining information about this module is for advanced users.

7.9.5 Interfaces

This section describes the interfaces.

`virtual`

The `virtual` interface handles requests to translate virtual addresses to physical addresses and to access the segment and page maps. It supports simulation and debug channel accesses and uses the `gen_bus_pkt` protocol. A request made to an unknown ASI is a fatal error for the simulation channel.

Address Translation Requests

When an address translation request is received, the virtual address of the request is converted to its corresponding physical address and the request sent to the mmu physical interface.

The interface uses the type and status fields of the gen_bus_pkt in the normal way and does not use the routeflg, size, and data fields.

The type field must be GEN_BUS_RD, GEN_BUS_WR, or GEN_BUS_RW. Any other type value is a fatal error for the simulation channel.

The extra field contains the context to which the translation applies. The asi field must contain 8, 9, 10, or 11.

The address to translate is extracted from the least significant 32 bits of the gen_bus_pkt vaddr field. If the conversion fails, the gen_bus_pkt status is set to GEN_BUS_FAULT, the reason for the failure is encoded in the extra field, and the request is returned to the virtual interface. The translation can fail because of an invalid PTE (extra = 2) or a protection violation (extra = 1).

If the translation succeeds, the least significant 32 bits of the gen_bus_pkt paddr field are filled in with the translated address, the extra field is set to the page type (from the PME), and the request is sent to the physical interface. No response to the request is expected or allowed on the physical interface.

Map Access Requests

When a map access request is received, the map entry is read or written and the response is returned to the virtual interface.

The interface uses the type and status fields of the gen_bus_pkt in the normal way and does not use the routeflg and paddr fields. The extra field contains the context to which the access applies. The asi field must contain 3 to access the segment map and 4 to access the page map.

The least significant 32 bits of the gen_bus_pkt vaddr field and the context are used to select the SME in the segment map or the PME in the page map according to the same algorithm used by address translation requests.

The selected SME or PME is read if the gen_bus_pkt type field is GEN_BUS_RD and is written if it is GEN_BUS_WR. If that field is any other value, it is a fatal error for the simulation channel.

If the gen_bus_pkt size field is not 1 or 2 for a segment access or is not 4 for a page access, the gen_bus_pkt status field is set to GEN_BUS_FAULT and the extra field is set to 3.

On a 2-byte segment access, only one SME is accessed. The data field is interpreted as a halfword; the most-significant byte is set to 0 on a read and is ignored on a write.

physical

The physical interface sends requests to access devices specified by their physical addresses. A successful translation request received on the virtual interface is sent to the physical interface to complete the request. The physical interface uses the gen_bus_pkt protocol and supports simulation and debug channel accesses.

The interface uses the type, status, and paddr fields of the gen_bus_pktin the normal way and does not use the routeflg, asi, vaddr, size, and data fields.

The extra field contains the page type from the PME of the translation. TABLE 7-8 lists the allowable page type values.

TABLE 7-8 MMU Page Types

Page Type	Description
0	Main memory
1	I/O
2	16-bit VME (sun4e only)
3	32-bit VME (sun4e only)

No response to the packets sent to the physical interface is expected or allowed.

dup_context0

If the dup_context0 interface receives a message, it duplicates the SMEs for context zero into the other seven contexts of the segment table. It supports simulation and debug channel accesses and requires no data with a received message. mmu never sends to the dup_context0 interface—not even to acknowledge a received message.

7.9.6 Source Files

Following are the source files:

- sun4e/mmu.c — Source for the module
- sun4e/mmu.h — Public declarations related to the module
- sun4e/mmu_private.h — Private declarations for the module
- sun4e/asi_arch.h — Definitions of the ASI values specific to the mmu module
- sun4c/mmugen.c — Stand-alone program that generates the mmu init file

The `sun4c` directory contains symbolic links to the `mmu.c`, `mmu.h` and `mmu_private.h` files in the `sun4e` directory. Also, the `sun4e` directory contains a symbolic link to the `mmugen.c` file in the `sun4c` directory.

7.10 `msi`: Mbus to SBus Module

The `msi` module simulates the MSI chip of the SPARCserver 600 and SPARCstation 3 systems. Following are the differences between the module and the real hardware:

- The module does not support diagnostic access (`paddr` 0xfe0000100 and 0xfe0000200).
- All M-S and S-M accesses are synchronous.
- The M-to-S Asynchronous Fault Status Register never has an error.
- SBus slot configuration registers are ignored.
- SBus master IOMMU bypass mode is not supported.
- SBus masters that use direct virtual memory accesses (DVMA) cannot access SBus slaves.
- Mbus access to SBus does not have a higher priority than SBus masters.
- Arbitration is not performed for the Mbus and SBus, although `msi` does contain the arbiter enabled register.
- The `MID` register functions only if the `mbus` module understands `MID`.

7.10.1 Variables

The following variables represent the internal state of instances of this module class and are available for use in expressions and in commands that require variables, such as `print`.

- `afar`

  This `Word` variable contains the asynchronous fault address register and is ignored by the `msi` module.

- `afsr`

  This `Word` variable contains the asynchronous fault status register and the following bit fields: `err`, `le`, `to`, `berr`, `siz`, `s`, `mid`, `me`, `rd`, `sa`, `ssiz`, `wm`, and `pa`. It is ignored by the `msi` module.

- arb_enable_reg

  This Word variable contains the arbiter enable register and the following bit fields: sbw, en_SBus_0xf, en_SBus_0x3, en_SBus_0x2, en_SBus_0x1, en_SBus_0x0, en_Mbus_0xb, en_Mbus_0xa, en_Mbus_0x9, en_Mbus_0x8.

- base_address_reg

  This Word variable contains the IOMMU base address register and the following bit fields: paddr_hi and paddr_lo.

- control_reg

  This Word variable contains the MSI control register and the following bit fields: impl, ver, range, de, and me.

- sbus_slot
  sbus_slot (*index*)

  This Word array variable contains the sbus configuration slot registers and the following bit fields: sega, cp, wma, ba64, ba32, ba16, ba8, and by. *index* specifies the slot to access. The msi module ignores the sbus_slot variable.

- show_translation_fault

  If this Bool variable is true, which is the default, and an IOMMU translation fault occurs, the msi module displays an error message.

- tlb
  tlb(*index*)

  This Word array variable contains the IOMMU TLB contents. It is composed of the following fields: va, lruq, ppn, cacheable, writeable, and valid. There is one array element for each TLB entry.

- translation_fault_count

  This Word variable contains a count of the number of IOMMU translation faults that have occurred.

7.10.2 Commands

In addition to the base set of commands, the following commands are available for dealing with instances of this module class. See *Universally Available Commands* on page 16 for details on invoking these commands.

- ranges — This command displays the virtual-to-physical address mapping ranges for the IOMMU.
- show_tlb — This command displays valid IOMMU TLB entries.
- stat — This command displays the statistics related to the msi module.

- `xlate` *virtual address* — This command displays the physical address that corresponds to *virtual address* according to the IOMMU translation tables. It also sets the `cmd_result` variable of the `ui` module to the physical address.

7.10.3 Configuration

There are six types of interfaces: `mbus_master`, `mbus_slave`, `register_slave`, `sbus_master`, `virtual`, and `cmd_done`. There must be one `mbus_master`, `sbus_master`, and `cmd_done` interfaces, as well as zero or one `virtual` interface. There can be any number of `mbus_slave` and `register_slave` interfaces.

Note – The remaining information about this module is for advanced users.

7.10.4 Interfaces

This section describes the interfaces.

`mbus_master`

The `mbus_master` interface accesses Mbus slaves for SBus masters and supports simulation and debug channel accesses. It uses the `gen_bus_pkt` protocol with `mbus` module extensions.

`mbus_slave`

The `mbus_slave` interface is used by Mbus masters to access SBus slaves (via `msi`, of course) and supports simulation and debug channel accesses. It uses the `gen_bus_pkt` protocol without `mbus` module extensions.

`register_slave`

The `register_slave` interface provides access to the `msi` registers and supports simulation and debug channel accesses. It uses the `gen_bus_pkt` protocol.

sbus_master

The sbus_master interface accesses SBus slaves for Mbus masters and supports simulation and debug channel accesses. It uses the gen_bus_pkt protocol with sbus module extensions.

virtual

The virtual interface handles SBus DVMA requests to translate virtual addresses to physical addresses and supports simulation and debug channel accesses. It uses the gen_bus_pkt protocol with sbus module extensions.

cmd_done

When ui executes a module command that may not complete immediately, it waits until it receives a message on this interface before it starts executing other commands. cmd_done supports only the debug channel and uses the no_data protocol. The ui module never sends to the cmd_done interface.

7.10.5 Source Files

The source files are:

- mbus/msi.c — Source for the module
- mbus/msi.h — Declarations that are related to the module

7.11 ram and rom: Memory Modules

The ram and rom modules each simulate a contiguous chunk of byte-addressable memory. There are only minor differences in their behavior.

7.11.1 Simulated Behavior

The ram module simulates memory: You write to a memory location and later on you can read it back. The starting address and size of the module instance memory are specified in the configuration file. The block of memory thus described responds to reads, writes, and read-modify-writes of arbitrary size, starting at arbitrary addresses. For example, no checking is done to prevent a 4-byte read from taking place at an odd address.

During initialization, a `ram` module instance attempts to read its initial contents from a file in the current directory. For details of this initialization file, see *Initial Contents* on page 122.

`rom` is like `ram`, except that the response to any write is an error. `rom` module contents can be initialized from an initialization file or set with various commands, but the simulated system itself cannot write to `rom`.

7.11.2 Variables

The following variables represent the internal state of instances of this module class and are available for use in expressions and in commands that require variables, such as `print`.

- *display mode*(*start addr*)
  display mode(*start addr, number*)
  display mode(*start addr, end addr*)

  You can examine and set the contents of memory with the above syntax formats. *display mode* is one of the following:

  - `lwords` (unsigned 64-bit integers)
  - `words` (unsigned 32-bit integers)
  - `instructions` (same as `words`, but disassembles when printed)
  - `hwords` (unsigned 16-bit integers)
  - `bytes` (unsigned 8-bit integers)
  - `chars` (same as `bytes`, but prints as ASCII characters)
  - `doubles` (64-bit IEEE double-precision floating-point values)
  - `floats` (32-bit IEEE single-precision floating-point values)

  Because of the minimum ambiguity recognizer for variable names, you need type only the first character of the mode, for example, `i` for `instructions`.

  If you inadvertently ask for a large memory array to be printed, you can stop the print with Control-C.

  The first format specifies only a starting address, which usually means "the element containing this address," such as the command. For example,

  ```
  print bytes(0x171)
  ```

  prints the byte at address 0x171.

Similarly, the command `print words(0x171)`prints the word at 0x170 because that is the word that contains the byte at address 0x171. In the `set` command, this syntax means that as many elements should be affected as there are values supplied. For example, the command

```
    set words(0x171) = 1, 2, 3
```

sets the word at 0x170 to `1`, the word at 0x174 to `2`, and the word at 0x178 to `3`.

In the second format, the number of elements is explicitly specified. For example, the command

```
    set words(0x171, 4) = 1, 2, 3
```

sets the four words starting at 0x170 to `1`, `2`, `3`, `3`. That is, the last value is repeated as many times as necessary.

In the third format, the first and third parameters are supplied, skipping the second parameter. The third parameter is the end address; all elements from the one that contains the start address through the one that contains the end address are affected. For example, the command

```
    set words(0x171,0x17d) = 1, 2, 3
```

sets the four words starting at 0x170 to `1`, `2`, `3`, `3`, just like the previous command.

When these formats are used in expressions, they evaluate to unsigned integers or floating-point numbers. If only one element is involved, the expression evaluates to the value of that element. If more than one element is involved, the expression evaluates to a special `checksum`, which should change if the contents of those elements changes. This function is very useful with the `changes` operator; for example, the command

```
    when bytes(my_table, 100) changes
```

arranges for the simulation to stop if any of the 100 bytes starting at the address in variable `my_table` changes. The `checksum` calculation is such that as long as only one element is affected each cycle, `checksum` always changes.

- `offset_for_disassembly`

  This `Word` variable can correct disassembly labels when code is mapped to a virtual address that differs from its physical address. Its value is initially `0`, but should be set to the offset between where instructions appear in the virtual address space and where they appear in the physical address space. For example, if you have linked a file at address 0 and loaded it into module instance `rom1` at 0xfe000000 and boot mode is mapping `rom1` into address 0, then by setting the `rom1 offset_for_disassembly` to 0xfe000000, you can derive the right symbols when you disassemble.

  This feature is intended for situations where a single offset applies to all instructions in the `ram`/`rom` module instance, not for situations in which pages are arbitrarily mapped with an MMU.

- `size`

  This read-only `Word` variable contains the size of the memory array in bytes, as configured.

- `start_addr`

  This read-only `LWord` variable contains the lowest address of the memory array in bytes, as configured.

7.11.3 Load Command

You can use `rom` and `ram` modules with the `load` and `load_section` commands. The `load` address is interpreted as the address at which the first byte should be placed.

7.11.4 Configuration

This section discusses the configurations.

Instance Arguments

Two instance arguments apply:

- `SIZE` *size*

  This mandatory argument specifies the number of bytes of memory to simulate. *size* must be a power of 2. It can end with the letter K, in which case it is multiplied by 1,024. Similarly, it can end with the letter M, in which case it is multiplied by 1,024 * 1,024. The range of valid sizes is 32 to 0x20000000 (0.5 Gbyte).

  Be aware that the module actually allocates a buffer of this size for the memory array, so adequate swap space must be available.

- `START_ADDR` *64-bit address*

  This mandatory argument specifies the physical memory address to which the lowest byte of the simulated memory corresponds. The start address must be a multiple of the memory size.

The `ram` and `rom` modules look only at those address bits, which determine the offset into the memory array, completely ignoring any higher bits. That way, the memory appears redundantly within the address space, which feeds into the memory module. For example, if a `ram` module is configured onto a bus in such a way that 16 Mbytes of address space are mapped into it but its size is only 4 Mbytes, then the first byte of the memory array appears at offsets 0, 4M, 8M, and 12M.

Initial Contents

When the module initializes, it sets all of memory to zeroes. Then, the module checks for an initialization file in the current directory. The file should have the name *module instance*`.init`, but its use is optional. If used, the file must contain hexadecimal digits that represent the initial data. You can arbitrarily separate those digits with white space (spaces, tabs, or newlines). The # symbol causes the rest of the line to be ignored for comments. An example is:

```
2   3        # release and version
fe002000 0100 # start address and length of table
```

By default, initialization begins with the first byte of memory. You can skip to an arbitrary address with @*addr*, which causes the hexadecimal digits that follow to be initialized, starting at address *addr*. If *addr* is a hexadecimal address, it must be prefixed with `0x`.

Note – The remaining information about this module is for advanced users.

7.11.5 Interfaces

One type of interface is supported by these modules: the `slave` interface.

The `slave` interface handles all requests to access memory and supports simulation and debug channel accesses. You can configure an arbitrary number of `slave` interfaces for `ram` or `rom` modules.

The `slave` interface uses the `gen_bus_pkt` protocol. Each packet received represents a single access to memory, where the type is `GEN_BUS_RD`, `GEN_BUS_WR`, `GEN_BUS_RW`, or `GEN_BUS_REF`. The `paddr` and `size` fields of the packet identify the region of memory to be copied to or from the `data` field of the packet. The upper bits of the `paddr` are masked off, leaving an index into the memory array; that is, the memory appears redundantly if mapped into an address space larger than the memory size.

After copying the data, the module sends `gen_bus_pkt` back with the status field set to `GEN_BUS_OK`. If the request is for a write to a `rom` module, no data are copied and the `gen_bus_pkt` is returned with status `GEN_BUS_FAULT`. In all cases, the module processes the request and replies immediately, with no associated delays.

7.11.6 Source Files

The files associated with these modules are in the `computer` directory:

- `ram.c`, `rom.c` — Source for the `ram` and `rom` modules
- `mem.c` — Source for the memory library routines
- `include/mem.h` — Public header file for memory library routines
- `ac_mem_{byte,char,hword,instr,word,lword}.c, mem.act, ac_mem.h` — Source for access classes

7.12 `s4vme`: sun4e SBus to VME Bus Controller Module

The s4vme module simulates the Sun S4-VME chip that exists in the SPARCengine 1E board. This chip connects the SPARCengine 1E SBus to the VME bus. `s4vme` simulates only the following S4-VME chip functions:

- 16-bit and 32-bit VME address spaces
- SBus slave
- VME master (16-bit and 32-bit address space)
- VME slave (16-bit address space)
- VME mailbox register

`s4vme` contains no arbitration or interrupt control logic.

7.12.1 Simulated Behavior

The `s4vme` SBus slave interface provides access to one register: the `mailbox` register, which is accessed as a byte. The format of the `mailbox` register matches the S4-VME chip specification.

The portion of the SBus address space occupied by the `s4vme` module is controlled by the architecture. The least significant 5 bits of the address seen by `s4vme` specify the address of the register to be accessed. The `mailbox` register is accessed with these bits set to 0x10. Any other values are rejected.

There are no registers in the `s4vme` to which the VME `slave` interface provides access. However, if the `mailbox` register is enabled and the VME `slave` interface receives a request that matches its `mailbox` address, `s4vme` generates an interrupt. The `mailbox` address is programmable.

7.12.2 Variables

The following variable represents the internal state of instances of this module class and is available for use in expressions and in commands that require variables, such as `print`.

`mbox` — This `Byte` variable contains the `mailbox` register and is composed of the bitfields `iflg` (interrupt pending flag), `en` (`mailbox` interrupt enabled), `comp1` (address bits 15 and 14), and `comp2` (address bits 3, 2, and 1).

7.12.3 Configuration

There are four types of interfaces: `SBus_slave`, `SBus_interrupt`, `vme_master`, and `vme_slave16`. There must be one interface of each type, except for `vme_slave16`, which can have any number of interfaces.

Note – The remaining information about this module is for advanced users.

7.12.4 Interfaces

This section describes the interfaces.

`SBus_slave`

The `SBus_slave` interface accepts requests to read and write the addressable registers of the `s4vme` module and also to initiate requests on the VME bus. It supports simulation and debug channel accesses.

The `SBus_slave` interface uses the `gen_bus_pkt` protocol. The `type`, `paddr`, `size`, and `data` fields of the `gen_bus_pkt` are set according to the standard `gen_bus_pkt` protocol. The `status`, `routeflg`, `asi`, and `vaddr` fields are not used.

The `extra` field contains the SPARCengine 1E address space identifier. Valid values are `1` (I/O), `2` (16-bit VME address), and `3` (32-bit VME address). Other values cause `s4vme` to cause a fatal error for simulation channel accesses.

If an I/O request is received (extra = 1) and the bottom 5 bits of paddr do not match the address of any s4vme register, or gen_bus_pkt size is not 1 byte, the request is returned with status set to GEN_BUS_FAULT. Otherwise, the specified register is accessed and the request is returned with status set to GEN_BUS_OK.

If a VME request is received (extra = 2 or 3), the request is forwarded to the vme_master interface. When that interface receives a response, the response is returned to the SBus_slave interface with the extra field set to contain the SPARCengine 1 address space identifier.

If the SBus_slave interface receives a request and it is busy with a previous request that was forwarded to the vme_master interface, the current request is returned with status set to GEN_BUS_BUSY.

SBus_interrupt

The SBus_interrupt interface is used by s4vme to cause an interrupt and supports only simulation channel accesses.

The SBus_interrupt interface uses the gen_int_pkt protocol and does not use the extra field. The irl field is set to 7, irl_valid is set to 1; action is set to INTERRUPT_SET to signal an active interrupt and is set to INTERRUPT_CLEAR to signal an inactive interrupt.

A response to an interrupt request is not required. If one is received, it is sent back to the SBus_interrupt interface with a delay of 1.

vme_master

The vme_master interface is used by s4vme to access VME slaves. It performs these requests in response to a request received on the SBus_slave interface with address space (in extra field) set to 2 or 3. It supports simulation and debug channel accesses.

The vme_master interface uses the gen_bus_pkt protocol. It uses the type, status, paddr, size, and data fields in the normal way and does not use the routeflg, asi, and vaddr fields. The extra field is set to the VME address space in bits (16 or 32).

If the response to a request sent to the vme_master interface has a status of GEN_BUS_BUSY, the response is sent to the vme_master interface again with a delay of one cycle. Otherwise, the request is returned to the SBus_slave interface since it initiated the request. The extra field is set back to the SPARCengine 1 address space of the request.

vme_slave16

The vme_slave16 interface accepts requests from VME masters in 16-bit VME address space to access the s4vme mailbox register. It does not initiate requests, but only responds. vme_slave16 supports only simulation channel accesses.

The vme_slave16 interface uses the gen_bus_pkt protocol. It uses the status, paddr, and size fields in the normal way and does not use the type, extra, routeflg, asi, vaddr, and data fields.

If the mailbox register is disabled, size is not 1 or 2 bytes. If bits 15, 14, 3, 2, and 1 of paddr do not match the values programmed into the mailbox register, the request is returned to the vme_slave16 interface with status set to GEN_BUS_FAULT. Otherwise, a mailbox hit has occurred and the response is sent back with status set to GEN_BUS_OK.

7.12.5 Source Files

Following are the source files:

- sun4e/s4vme.c — Source code for the module
- sun4e/s4vme.h — Private declarations for the module
- sun4e/mmu.h — Public declarations for the SPARCengine 1E memory types

7.13 sbus: SBus Module

The sbus module simulates a simplified SBus. Its characteristics are:

- The module uses a fair, round-robin arbitration mechanism.
- Interrupts are external to sbus.
- It supports a maximum of eight masters and eight slaves.
- It does not support SBus rerun or late errors.
- Slave addresses can be up to 36 bits in length.
- DVMA SBus masters cannot access SBus slaves.

7.13.1 Simulated Behavior

The sbus module optionally supports a built-in or external arbitration enable register; only one can be present. The arbitration enable register specifies which sbus masters are enabled for arbitration. If it is not present, all sbus masters are enabled for arbitration.

Each `sbus` master has 1 associated bit in the arbitration enable register. If that bit is 0, the master is not granted `sbus`. Bit *i* of the register controls the `sbus` master in offset *i*; bit 0 is the least significant bit of a word. The register is 32 bits.

If the built-in register is activated by appropriate configuration file entries, all processors see it at the same physical address. It only supports word accesses on a word boundary.

If the external register is activated, it is not considered part of the `sbus` module and has no physical address associated with it by `sbus`. The register is actually contained in another module, and that module provides user interface and processor access to its contents.

7.13.2 Variables

The following variables represent the internal state of instances of this module class and are available for use in expressions and in commands that require variables, such as `print`.

- `enable_reg_mask`

  This `Word` variable contains the value of the `arbiter enable` register. It exists only when the built-in arbiter enable register is in use.

- `next_master_to_get_sbus`

  The `sbus` arbitration code uses this `Byte` variable as an index into its array of masters to keep track of which master gets the `sbus` next. It is updated each cycle.

- `num_masters_waiting`

  This `Byte` variable contains the number of `sbus` masters that are waiting to get `sbus`. It is updated each cycle.

- `master_sbus_request_mask`

  If bit *i* is 1 in this `Word` variable, the master in offset *i* is waiting for `sbus`. This variable is updated each cycle.

- `master_interface`

  If a master has `sbus`, this read-only string variable is the name of that master interface. Otherwise, it is the string `nil`. It is updated each cycle.

- `locking_master_interface`

  If a master has `sbus` locked, this read-only string variable is the name of the master interface. Otherwise, it is the string `nil`. It is updated each cycle.

- `slave_interface`

  If `sbus` is waiting for a slave to respond to a master's request, this read-only string variable is the name of the slave interface. Otherwise, it is the string `nil`. It is updated each cycle.

7.13.3 Configuration

There are three types of interfaces: `master`, `slave`, and `iommu`. There can be zero to eight `master` interfaces, one to eight `slave` interfaces, and zero or one `iommu` interface.

There is one type of object shared used: `arb_enable_reg`. If it is specified, the external arbiter enable register is activated. It cannot be specified if the built-in arbiter enable register is also activated, however.

Instance Arguments

The optional instance arguments are:

- `ENABLE_REGISTER_ADDR` *36-bit address*
- `ENABLE_REGISTER_INITIAL_VALUE` *32-bit value*
- `ENABLE_REGISTER_STUCKAT_ONE` *32-bit value*

These arguments specify information about the built-in arbitration enable register and also activate the register. The information includes the address of the built-in arbiter enable register, its initial value, and a mask of the bits (if any) that are always 1. If bit *i* is 1, the master in offset *i* is enabled for arbitration; bit 0 is the least significant bit.

If `ENABLE_REGISTER_INITIAL_VALUE` or `ENABLE_REGISTER_STUCKAT_ONE` is specified, `ENABLE_REGISTER_ADDR` must also be specified. If *not*, the built-in arbiter enable register is not present and all masters are always enabled for arbitration.

Master Interface Arguments

The following master interface arguments apply.

- `USE_IOMMU`

  If this flag is specified, the `sbus` master uses DVMA.

- `SLOT` *value between 0 and* 15

  This argument specifies the SBus slot associated with the `master` interface. It is mandatory if `USE_IOMMU` is specified for the master. Only one master can use each slot value.

- `OFFSET` *value between 0 and* 31

  This argument specifies the bit offset into the arbiter enable register associated with the `master` interface. It is mandatory if the arbiter enable register is in use. Only one master can use each offset value.

- `MAKE_SBUS_SIGNALS`

  This optional argument specifies that the master does not generate `sbus` signals; hence, the `sbus` module must do so. This protocol works for simple masters only.

- `ALWAYS_ENABLED`

  This optional argument specifies that the master is always enabled for arbitration independently of the contents of the arbitration enable register. It is meaningful only if the built-in or external arbitration enable register is in use.

Slave Interface Arguments

The slave interface arguments are:

- `ADDR_BASE` *36-bit value*
  `ADDR_MASK` *36-bit value*

  These mandatory entries specify the ranges of addresses to which a slave responds. The range of each slave must be distinct. `ADDR_BASE` specifies the lowest address in the range. `ADDR_MASK` specifies which bits of the address are examined. If the address bitwise `AND`'d with the mask is equal to the base, then that address is inside the range.

- `DEBUG_SUPPORTED`

  This optional argument specifies that the slave supports debug cha`nnel` accesses.

Note – The remaining information about this module is for advanced users.

7.13.4 Interfaces

This section describes the interfaces.

`master`

The `master` interface handles requests to access `sbus` slaves and supports simulation and debug channel accesses.

The `master` interface uses the `gen_bus_pkt` protocol.

Requests

sbus queues its requests until the master is granted the sbus. Only one outstanding request per master is supported at a time.

The master interface uses the type, status, paddr, size, and data fields of the gen_bus_pkt in the normal way and does not use the asi, vaddr, and extra fields.

The route field of the gen_bus_pkt contains some SBus signals. The format is as follows:

```
struct multiplexed_signals {
        u_int   pad    :27;     /* unused */
        u_int   slot    :4;     /* SBus slot number */
        u_int   keep_bus:1;     /* allows master to lock SBus */
};
```

If the keep_bus bit is set, the master locks the bus so that only it can access the bus next. It is similar to the mbb physical signal of the Mbus; mbb is not in the simulation. mbb signal is asserted until the master wants to free the bus. However, the keep_bus signal is only asserted when the master wants the bus on the next cycle.

The slot field is set by the sbus module for a master if it specifies MAKE_SBUS_SIGNALS in the configuration file declaration for that interface. Otherwise, the master is expected to set the slot field itself.

If the master specifies an address that does not match any sbus slave, the sbus module sets gen_bus_pkt status to GEN_BUS_FAULT and returns the request to the master.

Responses

An sbus master receives a message only in response to a previous sbus request it made to access a slave.

In this case, the master interface uses the type, status, paddr, size, and data fields of the gen_bus_pkt in the normal way and does not use the asi, vaddr, extra, and routeflg fields.

slave

The slave interface connects to mbus slaves and supports simulation and debug channel accesses.

One slave must be identified as the memory slave since it acknowledges coherent invalidate transactions on mbus. Each slave responds to a unique range of the address space, which is specified in the configuration file.

The slave interface uses the gen_bus_pkt protocol.

Requests

A slave receives a message from the sbus when a master attempts to access it. It sets the type, paddr, size, and data fields of gen_bus_pkt according to the standard gen_bus_pkt protocol and does not use the extra field. The routeflg contains the sbus signals, described above, and is set according to those conventions.

Responses

After a slave has performed the transaction from a previous request, it sends a response back to sbus. The slave must set the gen_bus_pkt status field to indicate the result of the request.

If the request is successful, the physical signal status should be set to valid and the gen_bus_pkt status should be set to GEN_BUS_OK.

If the request is to an invalid address within the slave and the slave must inform the master of this error, it sets the gen_bus_pkt status to GEN_BUS_FAULT.

The slave cannot be busy since it does not support SBus rerun.

iommu

The iommu interface is present if the SBus supports DVMA. It supports simulation and debug channel accesses. The iommu interface uses the gen_bus_pkt protocol.

SBus DVMA master requests are routed to the iommu interface by the sbus module instead of to a sbus slave. The module connected to the iommu interface then processes the request. When the process is complete, it returns the request to the sbus module on the iommu interface, which then returns it to the original sbus master.

The module connected to the iommu interface cannot send a request it receives back to sbus on any sbus master interface. If it attempts to do so, a deadlock occurs. For this reason, sbus DVMA masters cannot access sbus slaves.

7.13.5 Source Files

The source files are:

- `mbus/sbus.c` — Source for module
- `mbus/sbus.h` — Structures, macros, and definitions useful to modules that connect to `sbus`
- `mbus/sbus_private.h` — Declarations used only by the `sbus` module

7.14 `serial`: Dual Serial Port Module

The `serial` module is modeled after the Z85C30 Serial Communications Controller (Z-SCC). For details on the Z85C30 SCC, see the *Z8030/Z8530 Serial Communications Controller Technical Manual, Advanced Micro Devices, 1988*.

The `serial` module features:

- Two channels (also called ports), A and B
- Asynchronous character transmissions
- Support for receive interrupts

7.14.1 Simulated Behavior

Four addresses access the serial ports. The actual physical address is controlled by the architecture. The bottom three bits of the address seen by `serial` specify the type of access. The ports are accessed according to TABLE 7-9.

TABLE 7-9 `serial` Registers

Address<2:0>	Description
0x0	Port B control access
0x2	Port B data access
0x4	Port A control access
0x6	Port A data access

Internal Registers

Each channel in the `serial` module contains 14 write registers and seven read registers, 1 byte each. The modes of communication, such as interrupts enabled or interrupts disabled, are established by the values of the write registers. As data are received or transmitted, the read register values may change to indicate the status of `serial`.

The registers are accessed in a two-step process via a register pointer to perform the addressing. For a particular register to be accessed, the pointer bits must be set by writing to `WR0` (write register 0). The next read or write operation then accesses the desired register. At the conclusion of this operation, the pointer bits are reset to `0` so that the next control operation accesses either read or write register 0.

All write registers can be written to. However, the `serial` module only uses the contents of `WR0`, `WR1`, and `WR9`.

Bits 5:0 of `WR0` indicate which register is accessed.

Bits 4:3 of WR1 indicate the Receive Interrupt Disable bits. The values of these bits indicate the following:

- 0x0 — Receive interrupts disabled
- 0x1 — Receive interrupts enabled on first character or special condition
- 0x2 — Receive interrupts enabled for all characters or special condition
- 0x3 — Receive interrupts on special condition only

The `serial` module functions if receive interrupts are enabled or disabled for all characters (values 0x2 and 0x0, respectively). Therefore, `serial` interprets the values or 0x1 and 0x3 as interrupts disabled.

Bit 3 of `WR9` is the Master Interrupt Enable (`MIE`) bit and is used to globally inhibit interrupts. If it is `1`, then interrupts for both channel A and B are enabled.

The read registers keep status information, which can be read. However, the `serial` module uses only `RR0` and `RR1` for receive interrupt conditions.

Bit 0 of `RR0` is the Rx Character Available bit. It is set to `1` when a character is available in the `serial` receive buffer and is reset to `0` when the buffer is empty. The size of the character buffer has no fixed limit.

Bit 2 and bit 5 of `RR3` indicate the Rx Interrupt Pending for each channel.

At simulation startup, all registers are cleared, the Master Interrupt Enable is set to `1`, and the Rx Interrupt Disable bits are set to `1` and `0`, respectively, to signal that receive interrupts are enabled.

7.14.2 Variables

The following variables represent the internal state of instances of this module class and are available for use in expressions and in commands that require variables, such as `print`.

- `a_enabled_rcv_int`

  If this `Bitfield` variable is set to `1`, the Rx Interrupt Disable bits are set to 0x2 for port A (receive interrupts enabled). If it is set to `0`, the Rx Interrupt Disable bits are set to 0x0 (receive interrupts disabled).

- `b_enabled_rcv_int`

  If this `Bitfield` variable is set to `1`, the Rx Interrupt Disable bits are set to 0x2 for port B (receive interrupts enabled). If it is set to `0`, the Rx Interrupt Disable bits are set to 0x0 (receive interrupts disabled).

- `a_pending_rcv_int`

  This `Bitfield` variable contains the value of the Rx Interrupt Pending bit for port A.

- `b_pending_rcv_int`

  This `Bitfield` variable contains the value of the Rx Interrupt Pending bit for port B.

- `int_enable`

  This `Bitfield` variable contains the value of the Master Interrupt Enable bit.

- `port_a_redirect_filename`

  This `String` variable indicates the name of the file to which to redirect the output of a write port A request. If this variable is not a null string, the `serial` module writes the character to the file instead of sending it to the `sigio` module.

- `port_b_redirect_filename`

  This `String` variable indicates the name of the file to which to redirect the output of a write port B request. If it is not a null string, the `serial` module writes the character to the file as opposed to sending it to the `sigio` module.

7.14.3 Configuration

There are three types of interfaces: `slave`, `sigio`, and `interrupt`. There can be any number of `slave`, one `sigio`, and zero or one `interrupt` interfaces.

During startup, a pseudo-`tty` is opened for each port and the information is displayed. For example:

```
serial1: Serial Port A is /dev/ttyp4
serial1: Serial Port B is /dev/ttyp5
```

The other end of the tty is intended to be connected to with the UNIX tip command in another window, where characters sent to serial are displayed. Characters typed in the window are sent to serial as input.

7.14.4 Interfaces

This section describes the interfaces.

slave

The slave interface accepts requests to read and write the control registers and the character buffer for the A and B ports. It only responds, does not initiate requests, and supports simulation channel accesses only.

The slave interface uses the gen_bus_pkt protocol. It uses the type, status, paddr, size, and data fields in the normal way and does not use the extra, routeflg, asi, and vaddr fields.

The paddr field of the gen_bus_pkt specifies what is to be accessed. If an invalid address is specified, the request is returned to the slave interface with status set to GEN_BUS_FAULT. Otherwise, the specified request (read or write) is performed according to the gen_bus_pkt type field.

The behavior of serial for the different types of requests is as follows.

- **Read data**
  - A character is read from the character buffer.
  - The Rx Character Available bit in RR0 is set to 0 if there are no characters left in the buffer.
  - If the Rx Interrupt Pending bit of RR3 is set, it is cleared. gen_int_pkt is sent to the interrupt interface. The action field of the packet is set to INTERRUPT_CLEAR to indicate that the interrupt has been cleared.
- **Write data**

  A character is sent to the sigio interface or to a file if redirected.
- **Read control**
  - The read register specified by the contents of WR0<5:0> is read.
  - WR0<5:0> is reset to 0.
- Write control
  - Data are written into the register specified by the contents of WR0<5:0>.
  - If the register written to is not WR0, WR0<5:0> is reset to 0.

At the end of the operation, the gen_bus_pkt is returned to the slave interface with status set to GEN_BUS_OK.

sigio

The sigio interface accepts characters as they arrive from the sigio module. It supports only debug channel accesses.

The character is placed into the character buffer and the Rx Character Available bit in RR0 is set. If receive interrupts are enabled and there is not an interrupt pending, Rx Interrupt Pending is set and gen_int_pkt is sent to the interrupt interface with the action set to INTERRUPT_SET.

interrupt

The interrupt interface is used by serial to handle interrupt requests and responses. It supports only simulation channel accesses.

When a character is received from the sigio interface and interrupts are enabled, gen_int_pkt is sent to the interrupt interface with the action set to INTERRUPT_SET.

A response to the interrupt request is not required. If one is received, the request is returned to the interrupt interface with a delay of 1.

7.14.5 Source Files

The source files are:

- computer/serial.c — Source for the module
- computer/serial.h — Private declarations for the module
- computer/include/serial_registers.h — Internal register formats

7.15 sigio: Simulator Input and Output Module

The sigio module provides file system input and output facilities to modules.

7.15.1 Configuration

Only one instance of the sigio module class can be created. One interface type applies: sigio. There can be any number of sigio instances.

Note – The remaining information about this module is for advanced users.

7.15.2 Interfaces

The sigio interface sends and receives data to modules and supports only the debug channel. It uses the sigio protocol.

Each sigio interface has an associated file descriptor. When input is available from that file descriptor, the sigio module reads it, places it in a message, and sends it to the sigio interface associated with the file descriptor. When the sigio module receives a message on a sigio interface, it writes the data in the message to the file descriptor that is associated with the interface.

7.15.3 Source Files

The source files are:

- fw/sigio.c — Source for module and some framework routines
- fw/sigio_private.h — Private declarations related to the module
- fw/include/sigio.h — Public declarations related to the module

7.16 simdisk: Simulated Disk Module

The simdisk module simulates a disk controller, but not any particular hardware.

The simdisk module features the following:

- Memory-mapped registers that control its operation
- Bus master capability that transfers disk blocks directly to memory
- Support for up to three disk partitions, stored as UNIX files
- Support for interrupts
- Support for 64-bit addresses

Besides the memory-mapped interface to `simdisk`, most architectures support a special CPU trap to access that module. See *trap: External Trap Module* on page 155 for more information.

7.16.1 Simulated Behavior

A program uses `simdisk` by loading its registers to specify a disk to memory transfer and setting the `start` bit in one of the registers to start the transfer. If interrupts are used, the program disk interrupt handler is invoked and the handler accesses a register to clear the interrupt.

The `simdisk` module responds by becoming a bus master and performing the transfer between memory and disk. If interrupts are enabled, `simdisk` posts an interrupt when the transfer is complete.

If an error occurs during the access of the disk file, a message is displayed and the simulation stops.

7.16.2 Registers

There are five addressable registers in `simdisk`. The portion of the address space they occupy is controlled by the architecture. Each of the registers is 4 bytes and can be read and written.The bottom 5 bits of the address seen by `simdisk` specify the address of the first register to be accessed according to TABLE 7-10.

TABLE 7-10 `simdisk` Registers

Address<2:0>	Description
0x0	Control
0x4	Memory start address (high 32 bits)
0x8	Memory start address (low 32 bits)
0xc	Partition
0x10	Disk block number

The registers must be accessed by a size that is a multiple of 4 bytes. The size divided by 4 specifies the number of registers to be accessed. For example, if the bottom 5 bits of the address are 0x8 and size is 8 bytes, the `simdisk` partition and disk block numbers are accessed.

Control and Status Register

The format of the control and status register is as follows:

<31:4> `transfer count`	Specifies the number of bytes to transfer between the disk and memory.
<3:2> `command`	If 1, reads a block from the disk; if 2, writes a block.
<1> `interrupt enabled`	If 1, `simdisk` generates an interrupt after a transfer is complete.
<0> `start transfer`	If 1, the transfer between disk and memory is started. All parameters required by the transfer must be loaded into the `simdisk` registers before this bit can be set. When the transfer is complete, the bit is set to 0.

Memory Start Address Registers

The memory start address registers are two 32-bit registers that logically create one 64-bit register, which specifies the starting memory address for a transfer between disk and memory.

Partition Register

The partition register specifies which of the three partitions (UNIX files) are in the next disk operation. Its value can be 0, 1, or 2, corresponding to the three partitions.

Disk Block Number Register

The disk block number register contains the starting block number for a transfer between disk and memory. Blocks are 512 bytes.

Number of Bytes Left to Transfer Register

Another register has the number of bytes that are left to be transferred by `simdisk`. After each transfer, this register is decremented by the number of bytes that have been transferred.

7.16.3 Variables

The following variables represent the internal state of instances of this module class and are available for use in expressions and in commands that require variables, such as `print`.

- `block_number`

  This `Word` variable contains the disk block number register.

- `busy`

  This `Bool` variable is `true` if `simdisk` is currently handling a disk-to-memory transfer; otherwise, it is `false`.

- `csr`

  This `Word` variable contains the control and status register. It is composed of the bit fields `count`, `command`, `interrupt_enabled`, and `start_transfer`.

- `interrupting`

  This `Bool` variable is `true` if the `simdisk` interrupt is active; otherwise, it is `false`.

- `partition`

  This `Word` variable contains the partition register.

- `start_addr`

  This `LWord` variable contains the memory start address register.

7.16.4 Configuration

There are three types of interfaces: `slave`, `master`, and `interrupt`. There can be any number of `slave`s, one `master`, and zero or one `interrupt` interfaces.

One mandatory instance argument applies: *partition information filename*, which is the name of a file that describes the three simulated disk partitions. The syntax of each line is *filename partition* where:

- *filename* is the name of the UNIX file that contains the partition data.
- *partition* is the maximum number of 512-byte blocks in the partition and can be any value.

There must be three of these lines, one for each partition. If a partition is not needed, use `nofile`.

Lines that start with a pound sign (#) are ignored.

Note – The remaining information about this module is for advanced users.

7.16.5 Interfaces

This section describes the interfaces.

slave

The slave interface accepts requests to read and write the addressable registers of the simdisk. It only responds, does not initiate requests, and supports simulation channel accesses only.

The slave interface uses the gen_bus_pkt protocol. It uses the type, status, paddr, size, and data fields in the normal way and does not use the extra, routeflg, asi, and vaddr fields.

The paddr and size fields specify which registers to access.

If simdisk is busy with a disk transfer when the slave request is received, the request is returned to the slave interface with status set to GEN_BUS_BUSY.

Any access to the slave interface when simdisk is not busy with a disk transfer clears the interrupt if it is active.

The paddr and size fields of gen_bus_pkt specify which registers to access. If an invalid address and size combination is specified, the request is returned to the slave interface with status set to GEN_BUS_FAULT. Otherwise, the specified registers are accessed according to the gen_bus_pkt type field, and the packet is returned to the slave interface with status set to GEN_BUS_OK.

master

The master interface is used by simdisk to transfer data between the disk and simulated memory. It supports simulation channel accesses only.

The master interface uses the gen_bus_pkt protocol. It uses the type, status, paddr, vaddr, size, and data fields in the normal way and does not use the extra, routeflg, and asi fields.

simdisk sends a message to the master interface when the start_transfer bit of the csr register is set to 1. The number of bytes, the disk block number (512-byte blocks), the starting memory address, and the UNIX file (simdisk partition) to access are taken from the appropriate simdisk registers. The start address is loaded into the vaddr and paddr fields of gen_bus_pkt.

At the beginning of a disk operation, gen_bus_pkt is allocated large enough to hold the specified number of bytes. A disk read operation reads the data from the partition into the packet's data area and then sends the packet to the master interface with type set to GEN_BUS_WR. A disk write operation does not yet access the disk (because it has no data to write) but sends the packet to the master interface with type set to GEN_BUS_RD.

The simdisk waits until the master interface receives a response to its request. If the response status is GEN_BUS_BUSY, status is set to GEN_BUS_OK and the request is sent to the master interface again.

If the status is GEN_BUS_OK or GEN_BUS_FAULT, the simdisk request is completed. If a disk write is being performed and the packet status is GEN_BUS_OK, the data are written to the partition (UNIX file).

interrupt

The interrupt interface is used by simdisk to cause an interrupt. It supports simulation channel accesses only.

The interrupt interface uses the gen_int_pkt protocol and does not use the irl, irl_valid, and extra fields. The action field is set to INTERRUPT_SET to signal an active interrupt and INTERRUPT_CLEAR to signal an inactive interrupt.

At the end of a simdisk disk transfer, an interrupt is asserted if interrupts are enabled in the simdisk control register.

A response to the interrupt request is not required. If one is received, it is returned to the interrupt interface with a delay of 1.

If simdisk is interrupting and an access is made to any of its slave interfaces, the interrupt condition is cleared.

7.16.6 Source Files

The source files are:

- computer/simdisk.c — Source for the module
- computer/simdisk.h — Private declarations for the module
- computer/include/simdisk_registers.h — Internal register formats

7.17 socket: Message to UNIX Socket Module

The socket module provides a mechanism to transfer MPSAS messages over a UNIX socket so that the modules in an MPSAS simulation can communicate with programs that are external to the simulation.

A program can connect to the socket as long as it follows the protocol expected by the MPSAS modules with which it communicates. For example, a socket module can connect to another MPSAS simulation that runs on a different machine or to another type of simulator, such as Verilog.

The socket module is a socket client. Therefore, you cannot use two socket modules to connect two MPSAS simulations.

7.17.1 Simulated Behavior

The socket module accepts messages from other MPSAS modules through interfaces and then multiplexes them over a UNIX socket.

Each interface on a socket module has a unique numeric identifier associated with it. When a message is received from a module on one of the socket interfaces, the socket module writes the data in the message to the socket prefixed with the identifier associated with that interface. When a message is available from the socket, the socket module first reads the identifier from the socket, followed by the message, and then forwards the message to the specified interface.

7.17.2 Configuration

There are two types of interfaces: pkt and sigio. There must be one sigio interface; there can be any number of pkt interfaces.

Instance Arguments

Two mandatory instance arguments apply:

- SOCKET_NUMBER *integer number* — Specifies the UNIX socket number to which to connect.
- MACHINE_NAME *namer* — Specifies the host machine to which to connect.

pkt Interface Arguments

The mandatory argument ID *number from 0 to 255* associates a unique numeric identifier with each pkt interface. Identifiers need only be unique for each socket module instance.

Note – The remaining information about this module is for advanced users.

7.17.3 Interfaces

This section describes the interfaces.

pkt

The pkt interface connects to other MPSAS modules. Any messages it receives are written to the socket. Any messages received by the socket are sent to a pkt interface.

The format of the messages read and written to the socket are described by the following C language data structure. Be sure to take into account alignment of fields if this data structure declaration is not used by the program connected to the socket.

```
struct socket_pkt {
        Byte    intf_id;
        caddr_t type;
        int     delay;
        int     size;
        Byte    data[1];
};
```

All messages sent by the socket module on its pkt interfaces are sent in the positive phase of the simulation.

The intf_id field is the interface identifier and is set by the socket module when it writes a message to the socket. The socket module requires this field to be set when it reads a message from a socket.

The type field is the MPSAS message type associated with the packet.

The delay field is the MPSAS cycle delay associated with the packet. If it is equal to –1, the packet is a debug channel packet.

The `size` field is the total size of the message data in bytes.

The `data` field is a variable-length array of the message data and contains `size` bytes.

`sigio`

The `sigio` interface connects to the `sigio` module and uses the `sigio` protocol.

7.17.4 Source Files

The source files are:

- `fw/socket.c` — Source code for the module
- `fw/socket.h` — Private declarations for the module
- `fw/ipp.c` — Interprocess package code (UNIX socket support routines)
- `fw/ipp.h` — Interprocess package declarations
- `fw/tcp_socket.c` — TCP/IP support code
- `fw/udp_socket.c` — UDP support code

7.18 `sys4c` and `sys4e`: sun4c and sun4e System Modules

The `sys4c` and`sys4e` modules are the central modules in the sun4c and sun4e architectures, respectively. They implement system registers and route memory requests to various devices, including the MMU. For the remainder of this chapter, the term `sys` module refers to both the `sys4e` and `sys4c` modules.

7.18.1 Simulated Behavior

The `sys` module routes memory and interrupt traffic between the CPU, MMU, memory, and devices in the system. The devices include simple slave devices, DVMA bus masters, and other buses.

The `sys` module also maintains several system registers that control the system behavior, as follows. For more details on the behavior of these registers, refer to the sun4c or sun4e architecture manuals.

- Asynchronous error register
- Asynchronous error virtual address register
- Counter-timer registers
- Context register
- System enable register
- Memory error register
- Synchronous error register
- Synchronous error virtual address register
- Interrupt register

7.18.2 Variables

The following variables represent the internal state of instances of this module class and are available for use in expressions and in commands that require variables, such as `print`. Many of them correspond directly to the registers in the sun4c and sun4e architectures.

- `async_err_reg`

  This `Word` variable corresponds to the asynchronous error register in the system. It is broken up into the following fields; `dvmaerr`, `invalid`, and `timeout`.

- `async_err_vaddr`

  This `Word` variable corresponds to the asynchronous error virtual address register in the system.

- `cntr0 limr0 cntr1 limr1`

  These `Word` variables correspond to the control registers for the counter-timer in the system. Each is divided into the `limit` and `value` bit fields.

- `context ctxt_reg`

  These `Byte` variables correspond to the context register in the system.

- `en_reg`

  This `Byte` variable corresponds to the system enable register. It is broken up into the following fields: `boot`, `cache`, `diag`, `reset`, and `sdvma`. When the simulation starts up and at reset, all of these fields are set to `0`, thereby enabling boot state and disabling all others.

Note – Boot state is enabled when the `boot` bit is set to `0`.

- `intr`

  This `Byte` variable corresponds to the interrupt register in the system. It is broken up into the following fields: `en_int`, `en_int1`, `en_int4`, `en_int6`, `en_int8`, `en_int10`, and `en_int14`.

- intr_pending

  This Word variable does not correspond to a hardware register in the system and is an informative register. It contains a field for each interrupt level, which is 1 if an interrupt is pending at that level. The bit fields are lvl1 to lvl15.

- merr

  This Word variable corresponds to the memory error register. In the hardware, it controls the parity error checking mechanism. In the simulation, parity errors are never generated. Modifying this register does not change the simulation. If a program writes to this register, the data are retained in the register but the requested action does not occur. The size of this register is one word. It is broken up into the following fields: mperr, pcheck, perr, perr00, perr08, perr16, perr24, and ptest.

- sync_err_reg

  This Word variable corresponds to the synchronous error register in the system. It is broken up into the following fields: invalid, memerr, proterr, rw, sberr, sizerr, time-out, and watchdog.

- sync_err_vaddr

  This Word variable corresponds to the synchronous error virtual address register in the system.

7.18.3 Configuration

The sys4c module supports 11 interface types: cpu, cpu_intr, vac_cpu, vac_mmu, mmu_vir, mmu_phy, type_0_device_space, type_1_device_space, system_space, dvma, and intr_set. The sys4e module supports the same interface types as the sys4c module as well as the vme and vme_intr interface types.

There must be only one interface of each of the following interface types configured into the system: cpu, cpu_intr, vac_cpu, vac_mmu, mmu_vir, mmu_phy, and type_0_device_space. There can be zero or more interfaces of the following interface types configured in the system: type_1_device_space, system_space, dvma, and intr_set. In the case of a sys4e module, there must be zero or one vme interfaces and there can be zero more vme_intr interfaces.

Arguments

The following arguments apply.

- DEBUG_SUPPORTED

  Both sys interfaces support this optional argument, which specifies that the slave supports debug channel accesses.

- ADDR_BASE *base addr* ADDR_MASK *mask*

  These mandatory type_1_device_space interface arguments determine when a memory request is directed to the device connected to this interface. The address of the request (A) is directed to this device when this expression is true: *base addr* == (A & *mask*).

- ADDR_BASE *base addr* ADDR_MASK *mask*

  These mandatory system_space interface arguments determine when a memory request is directed to the device connected to this interface. The address of the request (A) is directed to this device when this expression is true: *base addr* == (A & *mask*).

- IRL *level*

  This optional intr_set interface argument specifies the interrupt request level that is used for the device connected to this interface. The value of *level* must be greater than or equal to 0 and less than 16. If the value is 0, which is the default, then no interrupts are generated.

- ADDR_BASE *base addr* ADDR_MASK *mask*

  These mandatory VME arguments determine when a memory request is directed to the device connected to this interface. The address of the request (A) is directed to this device when this expression is true: *base addr* == (A & *mask*).

Note – The remaining information about this module is for advanced users.

7.18.4 Interfaces

This section describes the interfaces.

cpu

The cpu interface handles memory requests for the cpu module. It uses the gen_bus_pkt protocol and the type, asi, status, vaddr, size, and data fields of gen_bus_pkt in the normal way. It does not use the paddr, routeflg, and extra fields.

The cpu interface is where the sys module receives memory requests from the cpu module. Depending on the asi field of gen_bus_pkt, the requests can be sent to other modules via other interfaces. The eventual response to a memory request is returned to the cpu module via this interface.

cpu_intr

The cpu_intr interface sends interrupts to the cpu module. The sys module does not receive messages on this interface. Rather, the gen_int_pkt protocol communicates interrupts to the cpu module.

The sys module sends messages to the cpu module on this interface to change the interrupt request level at the processor. To set an interrupt, set the irl field to the interrupt level of the interrupt. To clear an interrupt, set the irl field to 0. The remainder of the fields in gen_int_pkt are not used.

vac_cpu and vac_mmu

The vac_cpu and vac_mmu interfaces are not in use and are included for possible future enhancements.

mmu_vir

The sys module uses the mmu_vir interface to pass memory requests to the mmu module, which can be for physical address translations or for direct access to the segment and page maps. It uses the gen_bus_pkt protocol and the type, asi, status, vaddr, size, and data fields of the gen_bus_pkt in the normal way and does not use the paddr field. It uses the extra field to pass the context number to the MMU and the routeflg field to mark this request as a CPU request (routeflg = 1) or DVMA request (routeflg = 2). The sys module expects the mmu module to leave routeflg field unchanged.

Messages received on this interface are the result of failed memory translation requests or responses to segment and page map accesses.

mmu_phy

The mmu_phy interface connects to the physical address side of an MMU. After gen_bus_pkt is sent to the MMU on the mmu_vir interface, the MMU places the physical address of gen_bus_pkt in the paddr field. If the address translation is successful, the MMU forwards gen_bus_pkt to the sys module on this interface, which then routes that packet to the correct device, according to the device space type (in the extra field) and paddr. When the response to the packet comes from the device, it is returned directly to the cpu module and is not sent to the MMU.

mmu_phy uses the type, status, paddr, size, and data fields of the gen_bus_pkt in the normal way and does not use the asi and vaddr fields. The extra field contains the device space type. The routeflg marks the request as a CPU request or DVMA request.

If routing the request fails, the status field is set to GEN_BUS_FAULT (1) and the message is sent to the cpu module through the cpu interface.

type_0_device_space

The type_0_device_space interface communicates to main memory (*type 0* space), as defined in the sun4e and sun4c architectures.

gen_bus_pkt communicates over this interface and uses the type, asi, status, vaddr, paddr, size, and data fields of the gen_bus_pkt in the normal way. The extra field contains the device space type. routeflg marks the request as a CPU request or DVMA request. Responses from the device connected to this interface are sent through the cpu interface to the requester and not back through either the mmu_vir or mmu_phy interfaces.

type_1_device_space

The type_1_device_space interface communicates to devices (*type 1* space), as defined in the sun4e and sun4c architectures.

gen_bus_pkt communicates over this interface. It uses the type, asi, status, vaddr, paddr, size, and data fields of gen_bus_pkt in the normal way. The extra field contains the device space type. routeflg marks the request as a CPU request or DVMA request. Responses from devices connected to this interface are sent through the cpu interface to the requester and not back through either mmu_vir or mmu_phy.

system_space

When memory requests arrive on the cpu interface with the system space ASI, they are sent out to the system_space interface, bypassing the MMU.

The gen_bus_pkt communicates on this interface. It uses the type, asi, status, vaddr, paddr, size, and data fields of the gen_bus_pkt in the normal way. The extra field contains the device space type. routeflg marks the request as a CPU request or DVMA request.

dvma

The dvma interface is connected to devices that perform DVMAs.

The device that requests a DVMA sends a gen_bus_pkt message to this interface. If there is already a DVMA in process, the message is returned to the requester on this interface. Otherwise, the DVMA request is sent to the mmu_vir interface. The memory request propagates through the system, just like a CPU memory request, with the final response being sent to the DVMA requester. During propagation, the routeflg field of the gen_bus_pkt contains the value GEN_BUS_DVMA_REQ (2).

intr_set

The intr_set interface connects to devices that generate interrupts. When this interface receives a request, it checks the action field of gen_int_pkt to determine if the request is to clear (action = 0) or set (action = 1) an interrupt. If there is already an interrupt pending at the requester's interrupt level, the message is immediately returned on the same interface.

vme

When a memory request is made with the device space type of 2 or 3 (extra field in gen_bus_pkt is 2 or 3), it is sent out on the vme interface. The response comes back on this interface and is directed back to the requester.

vme_intr

Similar to the intr_set interface, the vme_intr interface receives gen_int_pkt messages, which tell the sys module that a device wants to generate an interrupt. The difference between this interface and the intr_set interface is that the irl field of gen_int_pkt is interpreted differently. The irl field in this interface is different from the eventual interrupt level sent to the CPU. See TABLE 7-11 for the mapping information.

TABLE 7-11 Mapping of vme_intr Interrupt Levels to CPU Interrupt Levels 1

vme_intr IRL	**CPU** IRL
1	2
2	3
3	5
4	8

TABLE 7-11 Mapping of vme_intr Interrupt Levels to CPU Interrupt Levels 1 *(Continued)*

5	9
6	11
7	13

7.18.5 Source Files

The sys4c and sys4e modules share the same source code, as follows. The differences are incorporated by conditional compilation based on the constant sun4e.

- sun4c/sys.c — Symbolic link to sun4e/sys.c
- sun4c/sys.h — Symbolic link to sun4e/sys.h
- sun4e/sys.c — Source code for sys4e module
- sun4e/sys.h — Include file for sys4e module

7.19 timer: Simple Timer Module

The timer module simulates a timer chip with one, two, four, or eight timers. The timers are similar to those in the sun4c/sun4e architectures, except that they have 31 bits of resolution.

7.19.1 Simulated Behavior

Each timer has two associated registers: counter and limit. Their formats are the same. Bits 0 through 30 contain the count value; bit 31 contains the limit flag.

The counter register value is incremented with each timer tick and is architecture dependent. If the counter value reaches the limit register value, the counter is reset to 1, the limit flags of both registers are set, and the timer module generates an interrupt, the level for which is architecture dependent).

Reading the limit register clears the interrupt and the limit flags of each register. Writing the limit register provides a value for the counter register to “match” and resets the counter register to 1. If the limit flag of the limit register is cleared, the limit flag of the counter register is also cleared.

Reading the counter register reads the value of that register. Writing the counter register modifies the value of that register.

Setting the `limit` register to `0` causes the timer to generate an interrupt each time the counter overflows back to `0`.

The exact address of the `timer` module's registers is architecture dependent. The `timer` module examines only bits 0 through 5 of the address it sees. All registers are 32 bits. Only word-aligned accesses of word size are allowed. The counter register of timer *i* is at address *i* * 8. The limit register of timer *i* is at address *i* `* 8 + 4`.

Each timer generates interrupts independently.

7.19.2 Variables

The following variables represent the internal state of instances of this module class and are available for use in expressions and in commands that require variables, such as `print`.

- `counter`
  `counter(`*i*`)`

  This `Word` variable array contains the `counter` registers. Index *i* is the `counter` for timer *i*. Each element is composed of the bit fields `limit` and `value`.

- `limit`
  `limit(`*i*`)`

  This `Word` variable array contains the `limit` registers. Index *i* is the `limit` for timer *i*. Each element is composed of the bit fields `limit` and `value`.

7.19.3 Configuration

There are two types of interfaces: `slave` and `interrupt`. There can be any number of `slave` interfaces, but there must be one `interrupt` interface.

Instance Arguments

Two instance arguments apply.

- `NUM_TIMERS` *count*

  This mandatory argument specifies the number of timers for the `timer` instance being declared. *count* can be 1, 2, 4, or 8.

- `CYCLES_PER_TICK` *count*

  This mandatory argument specifies the number of simulator cycles per timer tick. It can be any nonzero 32-bit value.

Interrupt Interface Arguments

`TIMER` *value* is an interrupt interface argument that specifies the timer to which the `interrupt` interface corresponds. *value* ranges from 1 to the number of timers.

Note – The remaining information about this module is for advanced users.

7.19.4 Interfaces

This section describes the interfaces.

`slave`

The `slave` interface accepts requests to read and write the addressable registers of the `timer`. It supports both simulation and debug channel accesses.

The `slave` interface uses the `gen_bus_pkt` protocol and the `type`, `status`, `paddr`, `size`, and `data` fields in the normal way. It does not use the `extra`, `routeflg`, `asi`, and `vaddr` fields.

The `paddr` field specifies which register to access. If `size` is not 4 bytes or the `paddr` specifies an empty address, the request is returned to the `slave` interface with `status` set to `GEN_BUS_FAULT`. Otherwise, the specified register is accessed according to the `gen_bus_pkt` `type` field and the packet is returned to the `slave` interface with `status` set to `GEN_BUS_OK`.

`interrupt`

The `interrupt` interface causes an interrupt. It supports only simulation channel accesses.

The `interrupt` interface uses the `gen_int_pkt` protocol and does not use the `irl`, `irl_valid`, and `extra` fields. The `action` field is set to `INTERRUPT_SET` to signal an active interrupt and is set to `INTERRUPT_CLEAR` to signal an inactive interrupt.

A response to the interrupt request is not required. If one is received, it is returned to the `interrupt` interface with a delay of 1.

7.19.5 Source Files

The source files are:

- `mbus/timer.c` — Source for the module
- `mbus/timer.h` — Private declarations for the module

7.20 `trap`: External Trap Module

The `trap` module handles `trap` packets from the CPU, which generates them when it executes a trap instruction that it has designated as external. These external traps typically implement input or output operations in an architecturally independent manner.

7.20.1 Simulated Behavior

The operation of the `trap` module is only visible to the simulator programmer through SPARC trap instructions. The architecture configuration determines which traps are considered external.

Arguments to and results from external traps are passed between the `trap` module and the program that executes on the SPARC CPU via the CPU in, out, local, and global registers. The `carry` bit of the CPU `PSR` can also be modified by an external trap.

TABLE 7-12 lists the traps (SPARC V9 values) the `trap` module understands.

TABLE 7-12 External Traps

Trap Number	Function
0x148 (328)	Stop simulation
0x149 (329)	Simulator command
0x14a (330	Simulator string command
0x14b (331)	Halt simulation
0x14e (334)	Load file
0x151 (377)	Empty message
0x152 (338)	Disk operation
0x178 (376)	Get character waiting flag from port B
0x179 (377)	Read character from port B
0x17a (378)	Write character to port B

TABLE 7-12 External Traps *(Continued)*

0x17c (380)	Read character from port A
0x17d (381)	Write character from port A
0x17e (382)	Get character waiting flag from port A
0x3ff (1023)	Bad CPU trap

`stop simulation` Trap

The `stop simulation` trap stops the simulation (similar to the `stop` command). It takes no arguments and returns no results. To restart the simulation, type **`run`**.

`simulator command` Trap

The `simulator command` trap allows a command to be executed on the simulator as though you typed it. Thus, a program that runs on the simulator can control it.

The `trap` module builds the command string in a local buffer of 100 characters. Each time `simulator command trap` occurs, the least significant byte of CPU out register 0 (`%o0`) is added to the buffer. When a null character is added to the buffer or if the buffer fills up, the contents of the buffer are sent to the user interface, which executes the commands in the buffer as though you typed them.

Exercise caution on which commands are executed. For example, in the case of `quit`, the simulator stops and returns to the UNIX shell.

`simulator string command` Trap

The `simulator string command` trap allows a command to be executed on the simulator as though you typed it. With this trap, a pointer to the null-terminated command string is placed in CPU out register 0 (`%o0`) and the `trap` module reads the string itself. Including the null character, the string can be up to 1,024 characters in length.

`halt simulation` Trap

The `halt simulation` trap halts the simulation so that it cannot be restarted. It takes no arguments and returns no results.

load file Trap

The load file trap can read the contents of a file in the host file system into simulated memory. The parameters are as follows:

- %o0 — Pointer to file name in simulated address space
- %o1 — Offset into a file
- %o2 — Virtual address to place data
- %o3 — Number of bytes to read
- %o4 — ASI (least-significant byte)
- %o5 — 0 for loading instructions, 1 for loading data

This trap returns the number of bytes actually read in %o0. If an error occurs, –1 is returned and the carry bit in the PSR is set.

empty message Trap

The empty message trap sends a message that contains no data to the empty message interface. It takes no arguments and returns no results. The behavior of this trap depends on what the empty message interface, which is architecturally dependent, is connected to.

disk operation Trap

The disk operation trap supports transfers between simulated memory and a simulated disk. The memory and the disk in the transfer are architecture dependent. The memory is usually the system RAM; the disk is usually the simdisk module.

The disk operation is specified by the value of the CPU global register 1 (%g1). The C language header file /usr/include/sys/syscall.h contains definitions of the operating system calls. Four system calls are supported: open (SYS_open), write (SYS_write), read (SYS_read), and lseek (SYS_lseek).

The open operation is invoked before any of the other disk operations. The partition to open is passed in CPU out register 0 (%o0), which must be 0, 1, or 2 if the trap module is connected to the simdisk module. If the partition is valid, the CPU carry flag is cleared after the external trap instruction; otherwise, it is set. The open operation returns a file descriptor in the CPU out register 0 that must be used with subsequent disk operations. The open operation completes immediately.

The write operation initiates a data transfer from memory to the current disk block. CPU out register 0 contains the file descriptor from a previous disk open operation; out 1 contains the 32-bit memory start address; out 2 contains the transfer count in bytes. The interpretation of the memory start address (for example, virtual or physical) is architecture dependent.

The `read` operation initiates a data transfer from the current disk block to memory. CPU out register 0 contains the file descriptor from a previous `open` operation; out 1 contains the 32-bit memory start address; out 2 contains the transfer count in bytes. The interpretation of the memory start address (for example, virtual or physical) is architecture dependent.

The `lseek` operation changes the current disk block (for all partitions). CPU out register 0 contains the file descriptor from a previous `open` operation; out 1 contains the offset in bytes. The offset is rounded down to a 512-byte disk block.

If the disk module is busy with a previous transfer when the write, read, or `lseek` operations are attempted, the operations wait until the disk is free. If the operations cause an error, the `carry` flag is set on return from the CPU external trap; otherwise, it is cleared.

Interrupts are not used to signal that a disk read or write transfer has completed. In most architectures, some assumptions about when the operation is completed can be made. If this is not possible, an `lseek` operation to the current disk position (that is, a `nop` for the disk) can be started immediately after a trap read or write operation. The `lseek` trap then waits until the read or write is complete before returning to the CPU.

`get char waiting flag` Traps

The `get char waiting flag` traps determine if any characters are waiting to be read at a character device port (usually the `serial` module).

All of the port traps support two ports: A and B. The architecture determines which character devices are considered ports A and B.

The `get char waiting flag` traps have no arguments. They set CPU out register 0 (`%o0`) to `0` if no characters are waiting, and `1` if characters are waiting at their respective serial ports (A or B).

`read character from port` Traps

The `read character from port` traps cause a character to be read from the port associated with the trap. The character is placed in the least significant byte of CPU out register 0 (`%o0`). If no characters are available,`%o0` is set to –`1` (all ones).

`write character to port` Traps

The `write character from port` traps cause a character to be written to the port associated with the trap. The character is read from the least significant byte of CPU out register 0 (`%o0`).

bad cpu Trap

The bad cpu trap is executed in the trap handler for CPU traps that should not occur. CPU local register 3 (%l3) must contain the bad trap's trap number; local 1 must contain the PC at the time of the trap. The trap module displays a message with this information and then stops the simulation.

7.20.2 Configuration

There are five types of interfaces: cpu_trap, simdisk_slave, serial_slave, cmd_request, and empty_message. There must be one cpu_trap interface. It connects to the trap interface of the cpu module.

There can be zero or one simdisk_slave, serial_slave, cmd_request, and empty_message interfaces. The simdisk_slave interface connects to the simdisk module's slave interface. The serial_slave interface connects to the serial module's slave interface. The cmd_request interface connects to the user interface module's cmd_request interface. The empty_message interface connects to any interface that requires no data to be transferred for messages it receives, usually the dup_content0 interface of the sun4c/sun4e mmu module.

If the simdisk_slave, serial_slave, cmd_request, or empty_message interfaces are not present and a trap that uses the interface is encountered, the trap module displays a warning message and sets the CPU carry bit to 1 to indicate an error.

There are no configuration file arguments for the trap module instance or any of its interfaces.

Note – The remaining information about this module is for advanced users.

7.20.3 Interfaces

This section describes the interfaces.

cpu_trap

The cpu_trap interface accepts trap_pkt messages on the simulation channel. It receives a message each time the CPU executes a trap instruction that it considers external.

When the trap module has completed the trap, it sends a trap packet back to the CPU trap interface. Any registers in the trap packet that the trap module wrote have the corresponding update flag set to true.

simdisk_slave

The simdisk_slave interface sends requests to write the registers of the simdisk module on the simulation channel.

The simdisk_slave interface uses the gen_bus_pkt protocol. It always writes the simdisk registers, so the gen_bus_pkt type field is always set to GEN_BUS_WR. The least significant 5 bits of the vaddr and paddr fields are set to the first register to write, and size is set to the number of registers multiplied by 4. The trap module writes multiple and contiguous registers in simdisk in each packet that it sends to the simdisk module.

Each packet sent to the simdisk is later sent back to the trap module in acknowledgment. If gen_bus_pkt status of the response is GEN_BUS_BUSY, status is set to GEN_BUS_OK and the packet is sent back to the simdisk with a delay of 1. Otherwise, the trap is considered complete and trap_pkt is sent to the cpu_trap interface. If gen_bus_pkt status of the response is GEN_BUS_FAULT, the carry bit of the trap_pkt is set; otherwise, it is cleared.

serial_slave

The serial_slave interface sends requests to read and write the registers of the serial module on the simulation channel. All of the port traps (for example, read character from port) use the serial_slave interface.

This interface uses the gen_bus_pkt protocol. Multiple gen_bus_pkts are sent to the serial_slave interface to read and write its registers, as specified by the 8530 type bus interface protocol, which the serial module supports.

cmd_request

The cmd_request interface issues user-interface commands to the simulator on the debug channel. data of the message is the command string and the message type is String. No response is expected or allowed. This interface is used by the simulator command trap.

empty_message

The empty_message interface notifies another module of an event. When the empty message trap occurs, a message with no data and of type no_data is sent to the empty_message interface. No response is expected or allowed.

7.20.4 Source Files

The source files are:

- sparc/trap.c — Source for module
- sparc/trap.h — Private header file for module

7.21 ui: User Interface Module

The ui module provides a command-line user interface to the simulator. It parses user commands and executes them. It also allows modules to send it commands.

For a description of the user interface commands and features, see Chapter 3, *User Interface*.

7.21.1 Variables

The following variables represent the internal state of instances of this module class and are available for use in expressions and in commands that require variables, such as print.

- cmd_result

  This LWord variable is set by some commands to indicate their results. If a command displays a numerical result that fits in an LWord, this variable can contain that value. Some commands also use this variable to indicate success or failure (0 for success, nonzero for failure). This variable is useful mainly for command scripts invoked with the file command.

- cmd_result_double

  This Double variable is used in the same manner as cmd_result, except that it is used by commands that product floating-point results.

- `cyclecount`

  This `Word` variable contains the number of cycles the simulation has simulated. It measures the performance of the simulator (the `time` command).

- `history`

  This `Word` variable controls the maximum number of events that can be stored in the buffer that is used by the history mechanism. It defaults to `100` and cannot be set to 0.

- `instrcount`

  This `Word` variable contains the number of instructions executed by all processors in the simulation. It measures the performance of the simulator (the `time` command).

- `isconstprompt`

  If this `Bool` variable is true, the `ui` prompt is set to `sas:`. Otherwise, it is set to the name of the module instance on which the `ui` is focused, followed by a colon.

7.21.2 Commands

See Appendix B, *Command Manual Pages*, for a complete description of the user-interface commands. See *Processing of Command Lines* on page 22 for details on invoking the commands.

7.21.3 Configuration

Only one instance of the `ui` module class can be created.

There are four interface types: `cmd_done`, `sigio`, `cmd_request`, and `stop_simulation`. There must be one `cmd_done` and `sigio` interfaces and zero or one `cmd_request` and `stop_simulation` interfaces.

Note – The remaining information about this module is for advanced users.

7.21.4 Interfaces

This section describes the interfaces.

stop_simulation

When the stop_simulation interface receives a message, it stops the simulation. This interface supports only the simulation channel and uses the no_data protocol. The ui module never sends to the stop_simulation interface.

cmd_done

When ui executes a module command that may not complete immediately, it waits until it receives a message on this interface until it starts executing other commands. cmd_done supports only the simulation channel and uses the no_data protocol. The ui module never sends to the cmd_done interface.

sigio

ui receives user commands from the sigio module on this interface, which supports the debug channel only. It uses the sigio protocol. The ui module never sends to the sigio interface.

cmd_request

ui receives user commands from any module on this interface, which supports only the debug channel. It uses the string protocol. The ui module never sends to the cmd_request interface.

7.21.5 Source Files

The source files are:

- fw/ui.c — Source for module and some framework routines
- fw/include/ui.h — Public declarations related to the module
- fw/include/ui_cmds.h — Public declarations for layer commands
- fw/ui_fw_cmds.c — Source for framework layer ui commands
- fw/ui_fw_cmds.h — Declarations for framework layer ui commands
- sparc/ui_sparc_cmds.c — Source for SPARC layer ui commands
- sparc/ui_sparc_cmds.h — Declarations for SPARC layer ui commands
- fw/event_cmds.c — Source for when, snoop, and onstop commands
- fw/include/event_cmds.h — Declarations for event commands

7.22 vmebus: Primitive VMEbus Module

The vmebus module connects VME masters and VME slaves. It supports 16-bit and 32-bit VME address spaces. The address range to which a slave responds can be specified to vmebus. The vmebus arbitration is very simple: The first master to request the bus receives it until the slave is done.

7.22.1 Configuration

There are five types of interfaces: one master and four slave types. There can be between zero to eight master interfaces.

The four slave interface types are slave16_mapped, slave16_unmapped, slave32_mapped, and slave32_unmapped. 16 or 32 refers to the address space occupied by the slave. Mapped slaves have a constant address range to which they respond; unmapped slaves do not. There can be between zero to eight slave interfaces of each slave interface type.

slave16_mapped and slave32_mapped Interface Arguments

There are two slave16_mapped and slave32_mapped interface arguments.

- ADDR_MASK *mask*
  ADDR_BASE *base*

  These mandatory entries specify the range of addresses to which a slave responds. The range of each slave in its address space must be distinct. ADDR_BASE specifies the lowest address in the range; ADDR_MASK specifies which bits of the address are examined. If the address bitwise AND'd with *mask* is equal to *base*, then that address is inside the range. *mask* and *base* are 16-bit values for 16-bit address space and 32-bit values for 32-bit address space.

- DEBUG_SUPPORTED

  This optional entry specifies that the slave supports debug channel accesses.

slave16_unmapped and slave32_unmapped Interface Argument

slave16_unmapped and slave32_unmapped take one optional argument: DEBUG_SUPPORTED, which specifies that the slave supports debug channel accesses.

Note – The remaining information about this module is for advanced users.

7.22.2 Interfaces

The master interface handles requests to access vmebus slaves and supports simulation and debug channel accesses. It uses the gen_bus_pkt protocol.

vmebus uses the status and paddr fields of gen_bus_pkt in the normal way. It also uses all other fields, except extra. The extra field contains the address space of the slave in bits. Values other than 16 or 32 are a fatal error for the simulation channel.

When a request is received from a master, vmebus determines if the paddr lies in the range of one of the mapped slaves in the specified address space. It examines the least significant 16 or 32 bits of paddr to match the address space.

A master receives a message only in response to a previous vmebus request it made to access a slave.

vmebus uses only the status field of the gen_bus_pkt. If vmebus is busy with a request from another master, status is set to GEN_BUS_BUSY. If the request was not accepted by any slave, status is set to GEN_BUS_FAULT. Otherwise, status is set by the slave.

- slave16_mapped
- slave32_mapped
- slave16_unmapped
- slave32_unmapped

The slave interfaces connect to vmebus slaves and support simulation and debug channel accesses. They use the gen_bus_pkt protocol.

A slave receives a message from vmebus when the address space size of a master request matches that of the slave. If it is a mapped slave, the address of the request in the paddr field must lie in the range of addresses specified by the slave interface in the configuration file.

The type, paddr, size, and data fields of gen_bus_pkt are set according to the standard gen_bus_pkt protocol. The extra field contains the address space size (16 or 32). The routeflg, asi, and vaddr fields are not used.

For unmapped slaves, the status field is GEN_BUS_OK. For mapped slaves, vmebus does not use the status field.

A mapped slave performs the request, sets the gen_bus_pkt status to one of the standard values to indicate the state of the request, and returns the request to its slave interface.

An unmapped slave examines the request. If it accepts the request, the slave performs the request and returns gen_bus_pkt to its slave interface with status set to GEN_BUS_OK. If the slave does not accept the request, it returns the packet with status set to GEN_BUS_FAULT. Any other status values cause a fatal error for the simulation channel.

7.22.3 Source Files

The source files are:

- sun4e/vmebus.c — Source for module
- sun4e/vmebus.h — Declarations used only by the vmebus module

CHAPTER **8**

Trace Tools

The built-in simulator trace facilities (trace, dump, and group commands) allow module variables and messages to be traced. The util directory contains trace tools to simplify tracing of the cpu module as it executes instructions. These trace tools include a simulator command file that sets up the built-in simulator trace facilities for the cpu module and a UNIX utility to convert this trace output into SPARCsim version 5 trace records. You can display and analyze these trace records with the SPARCsim trace record tools (sta, std, and stdsm).

8.1 make_ss_trace

make_ss_trace is a UNIX program that reads simulator cpu module trace information from its standard input (stdin), converts it to SPARCsim trace records, and writes these to its standard input (stdout). The syntax is:

```
make_ss_trace [{-a | -e | -s | -t} name]* [object file]
```

The a, e, s, and t options used in conjunction with *name* specify the name of the annulled instruction, executed instruction, supervisor, and trap groups, respectively. The defaults are: annulled_record, executed_record, supervisor_record, and trap_record. These are the groups that ss_trace_cmds sets up, as described in the next section.

If *object file* is specified, make_ss_trace places its name into the SPARCsim trace file. If *object file* cannot be opened for read access, make_ss_trace displays an error message and exits. If *object file* is *not* specified, the default file name is the null string (“”). We strongly recommend that you include *object file* because many SPARCsim trace processing tools require that argument.

make_ss_trace outputs a SPARCsim trace record header that specifies a SPARCsim type 5 IU and a SPARCsim type 0 FPU. The type 5 IU is interpreted as a SPARC Version 8 processor.

8.2 ss_trace_cmds

The ss_trace_cmds command file sets up the simulator to trace cpu module activity and pipe that information to stdin of the make_ss_trace program. The command file must be invoked at the simulator prompt when the current working directory of the simulator is one of the architecture directories. The syntax that invokes the script from the simulator is:

```
file ../util/ss_trace_cmds cpu instance tracename [object-file]
```

where:

- *cpu instance* is the name of the processor module instance that is used as the source of the tracing information.
- *trace name* is the name of the trace within the simulator (as used by the trace command) and also the name of the SPARCsim trace record file created by make_ss_trace.
- The optional *object-file* is the name of a SPARC a.out or ELF file, which is used by SPARCsim trace analysis programs to convert processor addresses into symbols. If you omit *object file*, ss_trace_cmds uses the file name in the latest_file_loaded simulator environment (set by the load and load_section commands) instead. If you omit *object-file* and if the environment variable does not exist, the ss_trace_cmds script displays an error message and does not perform the trace.

ss_trace_cmds sets up the trace in the simulator, which involves three types of functions: opening the trace, creating groups that the make_ss_trace filter requires, and creating the group triggers.

The trace command opens a trace, called *tracename*. All output to the trace is piped to the make_ss_trace filter program, which produces a SPARCsim trace record file, also called *trace name*.

The four groups required by the make_ss_trace program are created, as follows:

- The executed_record group contains information about the latest instruction to be successfully executed.
- The annulled_record group contains information about the latest instruction annulled.
- The trap_record group contains information about the latest trap to be taken.
- The supervisor_record group contains the supervisor bit of the processor PSR.

Each of these groups has a trigger variable. A when event is created for each trigger so that when the trigger changes value, its corresponding group is dumped to the trace.

APPENDIX **A**

Error Messages

This appendix does not cover all the error messages that can be displayed while you are running MPSAS. It describes only the common errors in the following areas:

- Configuration file processing
- Expression handling
- Miscellaneous

A.1 Configuration File Processing

Following are some of the error messages that relate to the processing of configuration files.

`Configuration file error, line` *num* `at` *"keyword"* `syntax error`

There is a syntax error in the configuration file at line number *num*. *keyword* is either the invalid keyword that caused the error or the valid keyword closest to where the error occurred.

On occasion, other error messages are also displayed as a result of a syntax errors.

`Module instance` *"inst-name"*`, interface` *"intf-name"*`, connected to unknown module instance` *"inst-name2"*`.`

When declaring an interface, the `connected to` *inst-name2:intf-name2* statement specifies *inst-name2*, which is unknown to the simulator.

`Module instance` *"inst-name"*`, interface` *"intf-name"*`, connected to unknown interface` *"intf-name2"* `of module instance` *"inst-name2"*`.`

When declaring an interface, the `connected to` *inst-name2:intf-name2* statement specifies an interface *intf-name2*, which is unknown to the instance *inst-name2*.

`Module instance` *"inst-name"*`, interface` *"intf-name"*`, not connected to a read-only.`

When declaring an interface with the write-only flag, the `connected to` *inst-name2:intf-name2* statement specifies *intf-name2*, which is not a read-only interface. A write-only interface must be connected to a read-only interface.

`Module instance` *"inst-name"*`, interface` *"intf-name"*`, is not write-only but it is connected to a read-only.`

When declaring an interface, the `connected to` *inst-name2:intf-name2* statement specifies *intf-name2*, which is a read-only interface. Only a write-only interface can be connected to a read-only interface.

`Module instance` *"inst-name"*`, interface` *"intf-name"*`, connected to module instance` *"inst-name2"*`, interface` *"intf-name2"* `but it is not connected back.`

The interface *inst-name:intf-name* is connected to *inst-name2:intf-name2*; however, in the declaration of *intf-name2*, it is not connected to *inst-name:intf-name*.

`intf_create_interface: instance` *"inst-name"*`, interface` *"intf-name"*`: interface already declared in configuration file.`

The interface *intf-name* is declared multiple times. Each interface name within an instance must have a unique name.

`Instance Name` *"inst-name"* `already used`

The instance *inst-name* is declared multiple times. Each instance name must have a unique name.

`create_module_class: module` *"mod-name"*`: Module class already created.`

The module class *mod-name* is declared multiple times. Each module class must have a unique name.

The following error messages pertain to the modules. Individual modules may vary in their messages.

module-class`: module instance` *"inst-name"*`: interface type` *"intf-type"* `not present.`

An interface of type *intf-type* was not declared. This interface must be declared.

module-class`: module instance` *"inst-name"*`, interface` *"intf-name"*`: Only one` *"intf-type"* `interface type allowed.`

More than one instance of type *intf-type* was declared; only one interface of this type is allowed.

module-class`: module instance “`*inst-name*`”, interface “`*intf-name*`”: Unknown interface type “`*intf-type*`”.`

An interface was declared with an invalid interface type.

module-class`: module instance “`*inst-name*`”, interface “`*intf-name*`”: Interface type “`*intf-type*`” cannot be unconnected.`

Interface *intf-name* of type *intf-type* is not connected to another interface (via the `connect to` statement).

`Unknown or multiply defined keyword “`*keyword-argument*`” in args.`

The instance argument *keyword-argument* is either invalid or defined multiple times for an instance.

module-class`_config_intf: module instance “`*inst-name*`”, interface “`*intf-name*`”: Unknown keyword “`*keyword-argument*`” in config file args.`

An invalid interface argument *keyword-argument* was specified for interface *intf-name*.

instance-name`: error in config file`

If this message is displayed and no other information is provided, a required argument for *instance-name* is missing in the file.

instance-name`:` *keyword-argument* `value out of range`

A message of this type indicates that a particular instance or interface argument for *instance-name* was specified with an invalid value.

A.2 Expression Handling

This section discusses some of the error messages the expression parser produces.

`expression parser: bad symbol:` *symbol-name*

An unknown symbol name was specified in an expression. For example:

```
set pc = &main1
```

However, `main1` or `_main1` is not in the symbol table.

`expression parser: don’t know what “`*name*`” is`

An undefined name was specified as part of an expression. The common causes are:

- Syntax errors, such as

  ```
  set pc = x9999
  ```

  which should have been:

  ```
  set pc = 0x9999
  ```

- A symbol table name without the & character was specified. For example,

  ```
  set pc = main
  ```

  which should have been:

  ```
  set pc = &main
  ```

`expression parser: cannot use message types here expression parser: cannot use message fields here.`

Both of these messages result from a reference to a message when one is not being transmitted. Message types and message fields can only be evaluated within a `snoop` command list. For example:

```
expr gen_bus_pkt.asi
```

is invalid. Instead, it should read:

```
snoop (gen_bus_pkt.asi != 9) {print -v gen_bus_pkt.asi}
```

A.3 Miscellaneous

`ui: command queue too large -- flushing`

A command in the user interface command queue has caused the queue to overflow. All the commands in the queue will be flushed; therefore, none of these commands will be executed.

`Malloc failed`
`Out of memory`

Different kinds of these messages may appear during a simulation run—for example, if MPSAS does not have enough memory to continue execution. The simulation session sometimes terminates because of lack of memory. Other times, only the error message is displayed. Nonetheless, MPSAS cannot continue execution unless you allocate more memory for the process.

APPENDIX **B**

Command Manual Pages

This appendix describes in detail the universally available MPSAS commands in a format that is similar to that of the UNIX manual pages. The first manual page is for `mpsas` and shows how to invoke it from the shell.

Module commands are not described in this appendix. See Chapter 7, *Modules*, for details on module commands.

Name

mpsas — Multiprocessor SPARC architectural simulator

Synopsis

mpsas [-k *keyfile*] [-l *logfile*] [-o *name*=*value*] [-p *configdir*] [-s *cmdfile*] [-n]
[-m *preprocessor*] [*configfile*]

Description

mpsas is an architectural simulator for SPARC computers. It is equipped to simulate several architectures and can be extended to simulate others. It is a useful tool in evaluating hypothetical hardware designs and in porting software to hardware that has yet to be built.

mpsas requires a configuration file that describes the system to be simulated in terms of modules compiled into the simulator and their connections. See Chapter 5, *Configuration File*, for details on the contents of this file. By default, this file is ./config_file.

When mpsas starts, it reads the configuration file and sets up the described system. Some modules require additional initialization files. For details on the requirements of a module for initialization, see the relevant section in Chapter 7, *Modules*, for that module.

After reading the configuration file, mpsas looks for a startup command file called .mpsasrc in your home directory and, if it finds one, executes all its commands. Next, it looks for another startup file (./.mpsasrc by default) and executes its user-interface commands. (Typically, you would put architecture-independent commands in your home directory and architecture-dependent commands in each architecture directory you use.)

Finally, mpsas displays a prompt and accepts commands interactively. See Chapter 3, *User Interface*, for a description of this command-line interface.

mpsas records all keyboard input in a file (./mpsas.key by default). It also records the entire session (keyboard input and simulator output) in another file (by default, ./mpsas.log).

Options

-k *keyfile*
: Specifies the path of the file to which to store keyboard input (by default, ./mpsas.keyt).

`-l` *logfile*

Specifies the path of the file to which to store keyboard input and simulator output (by default,`./mpsas.log`).

`-o` *name*=*value*

Sets option *name* to *value*. Available options are described in the `option` command on page 201. No spaces are allowed before or after the equal sign.

`-p` *configdir*

Specifies the path of a directory in which to look for configuration information, which defaults to the current directory, including both the configuration file and any initialization files for modules.

`-s` *cmdfile*

Specifies the path of a file from which to read commands on startup (by default,`./.mpsasrc`).

`-n`

Does not preprocess configuration file.

`-m` *preprocessor*

Specifies the path name of the configuration file preprocessor, which defaults to the value of the `CPP UNIX` environment variable or `/usr/lib/cpp` if one does not exist.

Name

`alias`, `unalias` — Manage the command alias facility

Synopsis

`alias` [*name* [*value*]]
`unalias` *name* …

Description

The `alias` and `unalias` commands support a command alias facility similar to that of the C shell. To add a new alias or replace an existing alias, specify *name* and *value* to the `alias` command. *value* is terminated by an unescaped semicolon or the end-of-line. To display the value of an existing alias, type **`alias`** *name*. To display the names of all aliases currently defined, type the `alias` command by itself. To delete one or more aliases, specify *name* to the `unalias` command.

Invoke an alias on the simulator command line by specifying its name in place of the commands its represents. *value* can contain multiple commands, separated by semicolcons; be sure to escape each semicolon with a backslash.

Examples

```
alias w cpu1.where
```

Creates an alias for the `cpu1` module's `where` command.

```
alias ls sh ls
```

Creates an alias for the UNIX `ls` command.

```
alias start set cpu1.watchexecute = 1 \; run
```

Creates an alias that enables the display of executed commands in `cpu1` and then starts the simulation.

See Also

`setenv`

Name

`cycle` — Run the simulation for one or more cycles

Synopsis

`cycle` [*count*]

Description

The `cycle` command starts the simulation for *count* cycles or 1 cycle if *count* is not specified. *count* is an unsigned 32-bit integer expression.

If the simulation is halted by a module (typically when the module discovers an error), the simulation cannot be restarted.

The user-interface prompt is not displayed until the simulation stops running; however, user-interface commands are still accepted. If you type the `stop` command or Control-C, the simulation stops early.

If the output of the `cycle` command is redirected, all output produced by the simulator is redirected until the simulation stops.

Example

```
cycle 100 >log
```

Runs the simulation for 100 cycles. Any output generated by the simulator is sent to a file called `log`.

See Also

`run`
`stop`
`cpu` module `step` command

Name

`dasm` — Disassemble a SPARC instruction

Synopsis

`dasm` *address instruction*

Description

The `dasm` command prints the disassembly of a SPARC instruction. Both *address* and *instruction* are 32-bit integer expressions, unsigned or signed.

The `cpu` module's `read` command can disassemble instructions, as can the `ram` and `rom` modules' `instructions` variable. Certain other `cpu` variables provide disassembly as well. The `dasm` command is intended for situations other than those in which you have the numeric value of an instruction and want to disassemble it.

See Also

`cpu` module `read` command

Name

`dump` — Write contents of groups to traces

Synopsis

`dump` *tracename groupname*

Description

The `dump` command writes the contents of the members of *groupname* to *tracename*. Each time a group is dumped to a trace, each member of the group is dumped to the trace. The members that are dumped and their order match those that are displayed by the `group -v` command for the group. The group data are prefixed with a byte label assigned to that group for that trace.

Each group that is dumped to a trace is assigned a one-byte integer label to identify that group in the trace output. The first time that a group is dumped to a trace, it is assigned the next consecutive label for that trace, starting with 1. The group definition is dumped to the trace in the following `printf()` style format:

“`%c%c%s=%s%c`”,
`0` (special byte label reserved for group definitions),
byte label for *groupname*,
groupname,
members of *groupname* (as displayed by the `group` command),
`'\0'` (null character string terminator)

See Also

`group`
`trace`

Name

`echo` — Enable or disable command echo or display any text

Synopsis

`echo on` | `off` | [`-n`] *text*

Description

The `echo` command enables or disables echoing and displays messages. If echoing is enabled, commands from a `file` command script are displayed when executed.

`echo on` turns on echoing; `echo off` turns off echoing. `echo` followed by *text* displays *text* followed by a newline. *text* is terminated by the end of the line or the next command on the line.

`echo` accepts the following special characters:

- `\b` — Backspace
- `\c` — Print line without a new line
- `\f` — Form feed
- `\n` — Newline
- `\r` — Return
- `\t` — Tab
- `\v` — Vertical tab
- `\\` — Backslash
- `\`*n* — The 8-bit character whose ASCII code is the 1-digit, 2-digit, 3-digit, or 4-digit octal number *n*. The first digit must be 0.

The leading backslash of the special characters must be entered into MPSAS as `\\` so that the user interface does not interpret the special characters.

Options

`-n`	Suppress displaying a new line character after *text*.

See Also

`file`

Name

`enable`, `disable`, `delete`, `status` — Manage the `event` commands

Synopsis

`enable` *event#* [*event#* …] | `all` | `when` | `snoop` | `onstop` …
`disable` *event#* | `all` | `when` | `snoop` | `onstop` | `current` …
`delete` *event#* | `all` | `when` | `snoop` | `onstop` | `current` …
`status` [*event#* [*event#* …] | `all` | `when` | `snoop` | `onstop` …]

Description

These commands manage events created by the commands `snoop`, `when`, and `onstop`.

The arguments specify which events are managed, as follows:

- *event#* — The specific event number
- `all` — All events
- `when` — All `when` events
- `snoop` — All snoops
- `onstop` — All `onstop` events
- `current` — The current event

Each command can accept more than one keyword.

`enable` enables the specified events.

`disable` disables the specified event. If you specify `current`, the current event that caused the hit (if any) is disabled. You can use `current` within an event command list only; it refers to that event command. For example:

```
when (cpu1.pc == 0x1000) {echo pc == 0x1000; disable current}
```

When there is a hit (that is, `cpu1.pc` == `0x1000`), MPSAS executes `echo` and then `disable` causes the `when` event to be disabled.

`delete` deletes the specified events. The `current` option behaves in the same way as with the `disable` command.

`status` displays the status of the specified events. If no argument is specified, the status of all the events is displayed.

See Also

`when`
`snoop`
`onstop`

Name

`expr` — Display the values of expressions

Synopsis

`expr` [`-v`] *expression* [`,`*expression*] …

Description

The `expr` command evaluates arbitrary expressions and displays their values. The output is formatted according to the expression's data type: unsigned, signed, floating-point, or string. The value of each expression appears on a line by itself.

The `expr` command differs from the `print` command in two ways:

- `expr` can display any expression; `print` can only display variable expressions.
- `print` can often display a more user-friendly representation of a variable; `expr` displays an expression of a given data type the same way.

For unsigned and signed expressions, `expr` ordinarily displays the value in hexadecimal decimal if that value fits into 32 bits; character, if it is valid ASCII; and a symbolic representation, if the value is in the range of some symbol table and is larger than some symbol from that symbol table. However, if the option `simpleprint` is nonzero, `expr` only displays hexadecimal.

The `float_precision` option controls the number of mantissa digits displayed for floating-point expressions.

Options

`-v`	The expression itself is displayed, with parentheses as necessary (mostly around binary operators) to show how the expression is parsed.

See Also

`print`
`option`

Name

`file` — Execute simulator commands from a file

Synopsis

`file` *filename* [*arg*] …

Description

The `file` command causes simulator commands to be executed from *filename* as though you typed them.

You can pass up to 99 arguments to the commands in *filename*. The commands in the file reference the arguments as `$0`, `$1`, and so on. The file name is `$0`; the parameters are `$1` to `$99`. The number of arguments (*excluding* `$0`) is available as `$argc`.

If an argument is referred to but not specified in the `file` command, the name expands to the null string; for example, `$5` expands to “” if there are fewer than five arguments.

`file` supports nested file commands.

If echo is enabled (`echo on` command), commands are displayed as executed.

To exit from a command script early, use the `flush` command. All commands after `flush` are ignored.

Example

```
if ($argc != 1) { \
        echo Usage: $0 <cpu\\>; flush \
}

setenv old_focus $focused_instance
focus $1 >/dev/null
if (ui1.cmd_result) { \
        echo $0: Unknown module instance "$1"; flush \
}
focus $old_focus

print -v $1.pc, $1.npc, $1.psr
```

The preceding is the content of a text file designed to be executed by the `file` command. It prints the `pc`, `npc`, and `psr` variables of a `cpu` module instance. The `if` command in the first line echoes a usage message if the number of `file` command parameters (`$argc`) is not `1`. It then uses the `focus` command to ensure the parameter is a module instance. Finally, the specified variables of the module instance are printed.

See Also

`echo`
`flush`

Name

`flush` — Empty user interface command queue

Synopsis

```
flush
```

Description

The `flush` command empties the user-interface command queue. It is used mainly in `file` command scripts to exit from the script early.

See Also

`file`

Name

`focus` — Change the default user-interface module instance

Synopsis

`focus` *instance*

Description

The `focus` command changes the default user-interface module instance to *instance*. The uses of the default user-interface module instance (referred to as the focused instance) include:

- Instance variables of the focused instance are referred to by name; the instance name and period prefix are optional. For example, if the focused instance is `cpu1`, then the names `pc` and `cpu1.pc` are equivalent.
- Commands of the focused instance are referred to by name; the instance name and period prefix are optional. For example, if the focused instance is `cpu1`, then the commands `read` and `cpu1.read` are equivalent.
- The `help` command displays information on the commands of the focused instance.
- The environment variable `focused_instance` is set to *instance*.

The `ui` module's `cmd_result` variable is set to `0` on success and `1` on failure, for example, in case of a bad module instance name; this characteristic can be used to test for the existence of a module instance in `file` command scripts.

See Also

```
file
help
setenv
```

Name

fork — Fork a duplicate of the simulator process

Synopsis

```
fork
```

Description

The fork command performs a UNIX fork operation on the simulator. The parent process waits until the child dies. To return to the parent from the child, use the quit command.

The fork command and the state command both preserve the state of the simulator. state dumps the state to the file system; fork preserves the state in memory.

fork supports nested fork commands.

See Also

quit
state

Name

`group` — Manage the grouping of variable expressions

Synopsis

`group` [[`-v`] *name*]
`group` *name* `=` *member* [`,` *member*] …

Description

The `group` command creates groups. A group treats several variable expressions, message types, and other groups as members of a new variable.

Groups can be used in expressions. If a group is evaluated (for example, the `expr` command) and all of the members evaluate to a valid value, the group evaluates to a 64-bit `checksum` of the members' values. If any of the members evaluate to invalid, the entire group evaluates to invalid. If a group is displayed with the `print` command, each member of the group is displayed.

If `group` is invoked without arguments, the names of all groups defined are displayed. If `group` is invoked with *name*, the members of the group used to create `group` are displayed.

If `group` is invoked with *name*, an equal sign, and at least one member, a new group is created. *member* must be a variable expression.

You cannot change a group definition that has been dumped to an open trace.

Options

`-v`	Displays the members of the group as dumped to a trace. This is a superset of the list of members that contains not only the members themselves, but also their variable expression arguments (transitive closure on the set of all members).

Example

```
group foo = cpu1.psr, ram1.words(cpu1.pc)
```

Creates a group called `foo` that contains `cpu1.psr` and the word of `ram1` that matches the value of the `cpu1.pc` register. The output of the `group -v` command looks like this:

```
 group foo = cpu1.psr, cpu1.pc, ram1.words(cpu1.pc)
```

`cpu1.pc` is listed as a member of the group since `ram1.words` depends on it.

See Also

```
dump
trace
```

Name

`help` — Access online user-interface command help

Synopsis

`help` [*command*]

Description

The `help` command provides information about simulator user-interface commands. Help is available for all `ui` module commands and for the commands of the currently focused module instance.

If *command* is specified, detailed help on *command* is displayed. Otherwise, a brief description of all the available commands is displayed. This description may be quite long; you may wish to pipe the output to the UNIX `more` program by typing **`help | more`**.

See Also

`focus`

Name

history — Display the history event buffer

Synopsis

history [-h] [*count*]

Description

The history command displays the contents of the history buffer. If you specify *count*, history displays only the most recent *count* events.

The history mechanism is similar to the UNIX C shell history mechanism. For information about how to use this mechanism, see *Processing of Command Lines* on page 22.

Options

-h	Does not display the event numbers.

Name

`if` — Conditionally execute user-interface commands

Synopsis

`if` *expression* {*commands*}

Description

The `if` command evaluates integer *expression*. If it is valid and nonzero, *commands* are executed.

The `if` command is used mainly in `file` command scripts.

See Also

`file`

Name

`list` — Display information about the simulator and its modules

Synopsis

`list` [`-v`] [*instance-name*]
`list` *variable-name*
`list interfaces` [*instance-name*]
`list all` | `instances` | `classes` | `msgtypes`

Description

The `list` command displays information about the simulator and its modules. If no arguments are specified, the variable names of the currently focused module instance are displayed.

- `list` *instance-name* — Displays the names of *instance-name*'s variables
- `list` *variable-name* — Displays the names of the members of *variable-name* (usually bitfields)
- `list interfaces` [*instance-name*] — Displays the interface names for *instance-name* or for the currently focused instance
- `list all` — Displays all variable and interface names in the system
- `list instance` — Displays all module instance names configured
- `list classes` — Displays all module class names available
- `list msgtypes` — Displays all message type names

Options

`-v`	Verbose option. For the first form of the `list` command, `list` *instance-name*, the data type, class, and name of each variable are displayed. Variables that are normally hidden when `-v` is not specified are displayed, enclosed within parentheses.

Name

load — Load a binary file into a module instance

Synopsis

load *address instance file* [*symtable-start symtable-end*]

Description

The load command initializes module instances that contain memory (for example, RAM and ROM). It reads the contents of the SPARC statically linked binary *file* and loads it into *instance* at *address* (a 64-bit integer expression). The interpretation of the *address* value is determined by *instance*—it is a physical address for memory modules.

If *file* contains a symbol table, the symbol table is loaded into the framework. The symbols in this table can then be used in expressions (the & operator) and are used by some commands to associate symbolic names with program addresses.

The optional *symtable-start* and *symtable-end* arguments specify the range of virtual addresses for which to use the symbol table. They can be a symbol from the file's symbol table or a 64-bit integer constant. If omitted, *symtable-start* defaults to 0 and *symtable-end* defaults to the largest 64-bit virtual address.

The load command loads SPARC a.out and SPARC ELF binary files the same way as the SPARCstation boot PROM loads a kernel. If more precise control over the sections loaded is needed, use the load_section command. The text segment is loaded starting at *address*. For a.out OMAGIC files, the data segment is loaded at the first byte after the end of the text segment. For a.out ZMAGIC and NMAGIC files, the data segment is loaded at the first 8-Kbyte boundary after the end of the text segment. For ELF files, all loadable segments are loaded in the order they appear in *file*. Those that have execute permission but no write permissions are considered text segments; the rest are considered data segments.

The cmd_result variable of the ui module is set to the program entry point address, specified in *file*.

Several environment variables are set when a successful a.out load occurs, as follows:

- latest_file_loaded is set to *file*.
- text_start is set to the hexadecimal value of the text section start address in the module. This is equal to *address* for all binary files, except for a.out ZMAGIC files. For these types of files, text_start is equal to *address* plus the size of an a.out execution structure.

- `text_size` is set to the hexadecimal value of the text section size in bytes.
- `data_start` is set to the hexadecimal value of the data section start address in the module.
- `data_size` is set to the hexadecimal value of the data section size in bytes.
- `entrypoint` is set to the hexadecimal value of the file's entry point.

Several environment variables are set when a successful ELF `load` occurs, as follows:

- `latest_file_loaded` is set to *file*.
- For each text or data segment loaded, `text_`*n*`_start` or `data_`*n*`_start`, respectively, is set to the hexadecimal value of the segment's start address in the module.

n in the variable name is replaced with an integer value, which is incremented for each segment of that type loaded. For example, the first text segment loaded is `text_0_start`.

- For each segment loaded, `text_`*n*`_size` or `data_`*n*`_size` is set to the hexadecimal value of the segment's size in bytes.
- `entrypoint` is set to the hexadecimal value of the file's entry point.

See Also

`load_section`
`symtab`

Name

load_section — Load a section of a binary file into a module instance

Synopsis

load_section *address instance file section* [*symtable-start*]

Description

The load_section command initializes module instances that contain memory (for example, RAM and ROM). Its operation is similar to the load command, except that it loads *section* from the binary file instead of all sections to allow more precise control over what is loaded from a file and the address it is loaded into a module.

load_section reads the contents of *section* from the SPARC statically linked a.out or ELF binary *file* and loads it into *instance* at *address* (a 64-bit integer expression). *instance* determines the interpretation of the *address* value, which is a physical address for memory modules.

If *file* contains a symbol table, load_section loads the symbol table into the framework. The symbols in this table can then be used in expressions (the & operator) and are used by some commands to associate symbolic names with program addresses.

The optional *symtable-start* argument specifies the first virtual address for which the symbol table applies. It can be a symbol from the file's symbol table or a 64-bit numeric constant. If *symtable-start* is omitted, it defaults to 0. If *symtable-start* is specified, the last virtual address the symbol table is referenced for is equal to *symtable-start* plus the section size, less 1. Otherwise, it defaults to the largest virtual address.

a.out files only contain sections, called text and data. ELF files can have an arbitrary number of sections with arbitrary names, although .text and .data usually exist.

The cmd_result variable of the ui module is set to the program entry point address, specified in *file*.

Several environment variables are set when a successful a.out load_section occurs, as follows:

- latest_file_loaded is set to *file*.
- aout_section_start is set to the hexadecimal value of the *section* start address in the module. For the data section, the start address is equal to *address*. For the text section, it is equal to *address* for all a.out formats, except ZMAGIC, whose start address is equal to *address* plus the size of an a.out execution structure.

- `aout_section_size` is set to the hexadecimal value of the *section* size in bytes.
- `entrypoint` is set to the hexadecimal value of the file's entry point.

Several environment variables are set when a successful ELF `load_section` occurs, as follows:

- `latest_file_loaded` is set to *file*.
- `elf_section_start` is set to *address*.
- `elf_section_size` is set to the hexadecimal value of the *section* size in bytes.
- `entrypoint` is set to the hexadecimal value of the file's entry point.

See Also

`load`
`symtab`

Name

`msg` — Show the contents of a message or the format of a message type

Synopsis

`msg` [`-v`] [*msgtype*]

Description

If you include *msgtype*, the `msg` command prints the description of *msgtype*, showing the class and name of each variable that it contains. These variables generally correspond to fields within the actual C language structure for the message; however, multiple variables may describe the same field to present different ways of viewing or accessing it.

If you omit *msgtype*, `msg` prints information about the current message. This form of the command is only valid in the command list of a snoop event. The output is a summary line of this form:

```
cycle cyclenum: SIM|DBG src_instance:interface -> msgtype(size) ->
dst_instance:interface
```

which means: "On cycle *cyclenum*, module instance *src_instance* sent over interface *interface* a *size*-byte message of type *msgtype* to module instance *dst_instance*'s *interface* interface."

`SIM` and `DBG` specify whether the message was transmitted on the simulation or debug channel, respectively.

Options

`-v` When *msgtype* is omitted, this option causes the contents of the current message—that is, all of its variables—to be displayed after the summary line.

Examples

```
ui1: snoop {msg}

ui1: run
cycle 1: SIM cpu1:ram -> gen_bus_pkt(48) -> ram1:slave
cycle 1: SIM ram1:slave -> gen_bus_pkt(48) -> cpu1:ram
cycle 6: SIM cpu1:ram -> gen_bus_pkt(48) -> ram1:slave
...
```

See Also

`snoop`

Name

`onstop` — An event that executes a series of commands every time simulation stops

Synopsis

`onstop` [[`-p` *period*] [`-d`] {*command-list*}]

Description

The `onstop` command executes a sequence of commands (*command-list*) whenever the simulation stops.

To display the status of the `onstop` events, use the `onstop` command without any parameters or the `status onstop` command. To delete an `onstop` event, use the `delete` command. To disable or enable an `onstop` event, use the `disable` or `enable` command.

When the `onstop` event is created, the `ui` variable `cmd_result` is set to the event number.

Options

`-d` Adds the `onstop` event in a disabled state

`-p` *period*
Causes *command-list* to be executed every *period* hits. For example, the command `onstop -p 5` {`echo` "`5 more`"} causes the `echo` command to be executed once for every five times that the simulation stops. Without this option, `onstop` executes *command-list* every time simulation stops.

See Also

`events`
`when`
`snoop`

Name

`option` — Set or examine global simulator options

Synopsis

`option` [*name* [*value*]]

Description

The `option` command manages the option mechanism. Options are parameters that affect the entire simulation.

If *name* and *value* are specified, option *name* is set to *value*. If only *name* is specified, the value of option *name* is displayed. If *option* is specified with no arguments, the value of all options is displayed.

You can also set options on the command line when the simulator is first started (see the `mpsas` command on page 174). However, you can set those options that must not be changed during the simulation on the simulator command line only.

The options are:

- `block_signals` — If `block_signals` is nonzero, the `SIGINT` and `SIGTERM` signals are blocked by the simulator. You can set this option only on the simulator command line.
- `float_precision` — The `float_precision` option specifies the number of mantissa digits displayed for floating-point values.
- `simpleprint` — If `simpleprint` is nonzero, the display of many variables is simplified to ease parsing of the output by a program and the user-interface prompt remains constant.

See Also

`mpsas`

Name

`print` — Display the values of variables

Synopsis

`print` [`-v`] *variable* [`,` *variable*] …

Description

The `print` command displays the specified variables (more precisely, "variable expressions," which are variable names plus parameters, if any). For some variables, the output is the same as that produced by the `expr` command; for others, it is in a more human-readable form. For a group, `print` prints the members of the group. For an array, `print` prints all elements of the array.

Options

`-v`	Displays more labeling of the values that are printed. This option is often useful for groups and multiple variables.

See Also

`expr`

Name

`quit` — Exit the simulator

Synopsis

`quit` [*exit-value*]

Description

The `quit` command exits the simulator and returns to the operating system shell. *exit-value* is a 32-bit integer expression. If you specify *exit-value*, the simulator exits with that value; otherwise, it exits with a status of `0`.

Name

`rtimer`, `vtimer` — Set a real or virtual time limit on a simulation session

Synopsis

`rtimer` [*seconds*]
`vtimer` [*seconds*]

Description

The `rtimer` and `vtimer` commands set a limit on the amount of real time and virtual time, respectively, which the simulator is allowed to execute. When the timer expires, the simulator quits with an exit status of `1`.

seconds is a 32-bit integer expression. If you specify *seconds*, the time limit is in *seconds*. If *seconds* is `0`, the timer is cancelled. If you omit *seconds*, the number of seconds that remain before the timer expires is displayed and the `ui` module `cmd_result` variable is set to this value.

Use the `rtimer` and `vtimer` commands to ensure that the simulator will exit, for example, when the simulator is in an automated testing environment.

Name

`run` — Start the simulation

Synopsis

```
run
```

Description

The `run` command starts the simulation. It does not restart the simulation but continues from its current state.

If the simulation is halted by a module (typically when it discovers an error), you cannot restart the simulation.

The user-interface prompt is not displayed until the simulation stops running, but you can execute user-interface commands. If you type **`stop`** or Control-C, the simulation stops early. Other actions, such as breakpoints, can cause the simulation to stop.

If the output of the `run` command is redirected, all output produced by the simulator is redirected until the simulation stops.

See Also

`cycle`
`stop`

Name

`set` — Set variables

Synopsis

`set` *variable* `=` *expression* `[,`*expression*`]` …

Description

The `set` command changes the value of a variable (more precisely, a "variable expression," which is a variable name plus parameters, if any). How a variable is set is determined by that particular variable. For instance, you can assign a list of values to some variable expressions.

See Also

`print`

Name

`setenv` — Associate text with a name

Synopsis

`setenv` [*name* [*text*]]

Description

The `setenv` command associates arbitrary text with a name. In subsequent commands, the syntax `$`*name* is replaced with *text*, which includes everything from the next nonblank character after *name*, if there is one, to the end of the command. If you omit *text*, the syntax `$`*name* expands to nothing.

name must start with a letter or underscore and can be followed by any number of alphanumeric characters and underscores. If you omit *name*, the simulator displays all the names that have text associated with them.

When `$`*name* appears in a command, *name* must *not* be immediately followed by an alphanumeric character or an underscore because MPSAS considers such characters part of *name* as well.

The MPSAS `setenv` command is syntactically and semantically similar to the UNIX C shell command by the same name. However, the names that are defined with the MPSAS `setenv` command are not added to the environment, nor are the environment variables accessible with the `$`*name* syntax.

Name

`sh` — Execute UNIX commands

Synopsis

`sh` [*command*]

Description

The `sh` command executes UNIX commands as a child process of the simulator.

command specifies the UNIX command to execute. Any special symbols interpreted by the simulator user interface (such as `;` and `>`) that must be passed to *command* must be escaped by a backslash.

If you omit *command*, the shell program specified by the UNIX `SHELL` environment variable is invoked interactively.

Examples

```
sh ls
```

Executes the UNIX `ls` command.

```
sh ls *.c \| more
```

Executes the UNIX `ls` command and pipes the output to the UNIX `more` command.

Name

`snoop` — Create an event that is triggered by a message

Synopsis

`snoop` [`-p` *period*] [`-d`] [`-sim`] [`-dbg`] [`-src` *inst*[:*intf*]] [`-dst` *inst*[:*intf*]] [`-path` *from-inst*[:*intf*] *to-inst*[:*intf*]] [*expr*] [{*command-list*}]

Description

The `snoop` command causes *command-list* to be executed after a message that matches the specified description is received by a module and the expression *expr*, which typically refers to the fields within a message, is `true` (that is, a hit).

If you specify `-sim`, `snoop` performs snooping on simulation channel messages. If you specify `-dbg`, `snoop` performs snooping on debug channel messages. Without either of these options, `snoop` performs snooping on simulation channel messages only.

If you specify *expr*, `snoop` executes *command-list* every time a message that matches the option restrictions is received. If you do not specify *expr* or any of the options, `snoop` executes *command-list* every time any message is received by any module (that is, all messages are snooped and cause a hit). If *command-list* is not specified, `snoop` defaults to the `msg -v` command.

To display the status of the snoop events, use the `snoop` command without any parameters or `status snoop`. To delete a snoop event, use `delete`. To disable or enable a snoop event, use `disable` or `enable`, respectively.

Refer to *Expressions* on page 25 for details on expressions.

When `snoop` creates a `snoop` event, it sets the `ui` variable `cmd_result` to the event number.

Options

`-d` Creates the `snoop` event in a disabled state.

`-sim` Snoops the simulation channel.

`-dbg` Snoops the debug channel.

`-p` *period*
: Causes *command-list* to be executed every period hits. For example, the command `snoop -p 5 (gen_bus_pkt.asi == 9) {echo 5 more}` causes the

`echo` command to be executed once for every five messages of type `gen_bus_pkt`, in which the `asi` is 9. Without this option, *command-list* is executed on every hit.

`-src` *inst*[:*intf*]
: Snoops every message that is sent from instance *inst*. Snooping is performed when the message is received by any of the destination interfaces.

`-dst` *inst*[:*intf*]
: Snoops every message that is received by instance *inst*. Snooping is performed when the message is received by the destination interface.

`-path` *from-inst*[:*intf*] *to-inst*[:*intf*]
: Snoops every message that is sent from instance *from-inst* to instance *to-inst*. Snooping is performed when the message is received by the destination interface.

In the `-src`, `-dst`, and `-path` options, if the interface (:*intf*) is not specified, all the interfaces for the specified instance are snooped; otherwise, only the specified interface is snooped.

Examples

The following command line, in which *command-list* is not specified, causes the default command (`msg -v`) to be executed when a message of type `gen_bus_pkt`, in which the `asi` is 9, is received by a module.

```
snoop (gen_bus_pkt.asi == 9)
```

The following command line creates a snoop event that uses the default command (`msg -v`), but there is no trigger expression. The empty braces are required because `snoop` with no parameters lists the current snoop events.

```
snoop {}
```

The following `snoop` event causes information about the `gen_int_pkt` message to be displayed every time a message of type `gen_int_pkt` is received by a module.

```
snoop (gen_int_pkt) { msg }
```

The following command line causes the command `stop` to be executed whenever any message is received by any module.

```
snoop { stop }
```

The following command line modifies a message with the `set` command. You can then modify the information being sent between modules and use this command line to test a module or patch it (temporarily change its behavior). Here, any `gen_bus_pkt`s sent by the `cpu1` module with an `asi` of 8 are changed to 9.

```
snoop -src cpu1 (gen_bus_pkt && gen_bus_pkt.asi == 8) { set
    gen_bus_pkt.asi = 9 }
```

See Also

```
events
msg
expr
when
onstop
```

Name

`state` —Dump or restore a snapshot of the simulator state

Synopsis

`state` [(`dump` | `restore` | `delete`) *name* [*instance*]]

Description

The `state` command manages the state snapshot facility and is used mainly as a time-saving measure. If you would like to repeatedly examine some program behavior that occurs after many simulated cycles, you can dump the state of the simulation to the file system slightly before the behavior of interest and then restore it any number of times later (even if the simulator has been restarted).

If you specify *instance*, `state` manipulates only the state of that module instance. Otherwise, it manipulates the state of all module instances as well as the framework state. *name* is a user-specified name associated with the state; it must not start with a slash or a period.

The `state dump` command writes the state of *instance* or the entire simulator to the file system. If you specify *instance*, `state` creates a file, called *name*`.`*instance*`.state`. Otherwise, it creates a directory called *name* in the current working directory, writes the state of each module instance to its own file, writes the state of the framework to a file, and creates links to any files that contain symbol tables that are currently loaded (by the `load`, `load_section`, or `symtab` commands). If the state associated with *name* already exists in the file system, `state` overwrites it.

The `state restore` command replaces the current state of *instance* or the entire simulator with the state in *name*.

The `state delete` command removes state *name* from the file system.

If you do not specify arguments to the `state` command, `state` lists the names of all the previously dumped states.

The dumping or restoring of a state may take some time because some modules, such as RAM, may have very large states. You can use the `fork` command to save the state in memory, proceed with the simulation, and then return to the original state later.

See Also

`fork`

Name

`stop` — Stop the simulation from running

Synopsis

```
stop
```

Description

The `stop` command stops the simulation. You can use it after the `run` and `cycle` commands. As you do with all user-interface commands, you can type `stop` when the simulation is running.

Typing Control-C also stops the simulation, but it also empties the user-interface command queue. Typing Control-C is the only method to stop the simulation on demand after you have entered a `wait` command.

See Also

```
run
cycle
wait
```

Name

`symtab` — Manage symbol tables

Synopsis

`symtab`
`symtab add` *filename start-address end-address*
`symtab delete` *number*
`symtab dump` *number*
`symtab offset` *offset number* [*segment-name*]

Description

The `symtab` command manages the simulator's knowledge of symbols. With no arguments, `symtab` lists all the symbol tables known to the simulator, along with the address ranges assigned to those symbol tables.

`symtab add` reads the symbols from executable *filename* (which can be in either `a.out` or ELF format) and assigns them to the range of addresses, from *start-address* to *end-address*. When address ranges of symbol tables overlap, `symtab add` uses the most recently added symbol table. Each symbol table added is assigned a number, which is stored in the `ui` module's `cmd_result` variable and is shown when symbol tables are listed.

`symtab delete` deletes symbol table *number*.

`symtab dump` displays all the symbols in symbol table *number*, along with their values, in ascending order by value. Also shown is the name of the segment that contains the symbol. Global (and, for ELF executables, weak) symbols are flagged as such.

`symtab offset` adds *offset* (a 64-bit signed integer expression) to the value of the symbols of symbol table *number*. If you specify *segment-name*, `symtab offset` changes only the values of the symbols in that segment. You can use `symtab offset` to load symbol tables of relocatable files. Typically, you use `symtab add` to load the symbols first and then use `symtab offset` to add the base address of each segment to the symbols in that segment. This procedure assumes that the relocatable symbols initially have values relative to an address of zero.

The `load` and `load_section` commands also load symbol tables, just as though they had been added with `symtab add`.

`symtab` loads only relocatable symbols (labels); in particular, it does not load absolute symbols.

See Also

```
load
load_section
```

Name

`time` — Display time-related simulator statistics

Synopsis

`time [elapsed| relative]`

Description

The `time` command displays the real, system, and user time for the simulation. It also displays other performance statistics, such as cycles per second and instructions per second. The `ui` module variable `cmd_result_double` is set to the sum of the system and user times.

If you specify `elapsed`, `time` displays the values since the very first the `time` command was executed. If you specify `relative`, `time` displays the values since the previous `time` command invocation was executed.

Name

`trace` — Manage the trace facility

Synopsis

`trace`
`trace open` *tracename* `file` *filename*
`trace open` *tracename* `pipe` *UNIX-command*
`trace flush` *tracename*
`trace close` *tracename*

Description

The `trace` command manages the trace facility. A trace provides a destination for the output of the `dump` command. Multiple traces can be active at one time; the *tracename* parameter allows each trace to be manipulated individually.

`trace open` creates a new trace called *tracename*. If you specify `file`, `trace open` stores all output sent to the trace in *filename*. If *filename* already exists, `trace open` overwrites it. If you specify `pipe`, `trace open` runs *UNIX-command* as a child process of the simulator and sends all output sent to the trace to its standard input.

`trace flush` causes the internal buffer associated with *tracename* to be flushed to its associated file or pipe.

`trace close` deactivates *tracename* and closes the associated file or pipe.

`trace` with no arguments lists the names of the currently open traces.

See Also

`dump`
`group`

Name

`var` — Manage user-defined variables

Synopsis

`var` [*name* …]
`var add` *name datatype class* [*num-elements*]
`var delete` [*name* …]

Description

The `var` command lists, creates, or deletes variables of various types, which can be scalars or one-dimensional arrays. Once created, a user-defined variable can be used in any expression. The `set` command sets a user-defined variable just like other variables.

Note – Groups are also user-defined variables, but of a different kind. You manage groups with the `group` command rather than with the `var` command.

The first syntax lists user-defined variables. If you provide *name*s, `var` lists those variables; otherwise, `var` lists all variables along with their data types and classes.

A variable's data type is one of the following:

```
    unsigned     signed float string
```

The data type determines the interpretation of the bits that constitute the variable. A variable's class dictates its size (number of bits) and other details of its handling. See Chapter 3, *User Interface*, for more information.

User-defined variables can be of any of the classes that are defined in TABLE B-1.

TABLE B-1 Classes of User-Defined Variables

Class	**Description**	**Allowed Data Types**
`LWord`	64 bits	`unsigned`, `signed`, `float`
`Word`	32 bits	`unsigned`, `signed`, `float`
`HWord`	16 bits	`unsigned`, `signed`

TABLE B-1 Classes of User-Defined Variables *(Continued)*

Class	Description	Allowed Data Types
`Byte`	8 bits	`unsigned`, `signed`
`Bool`	8 bits, boolean	`unsigned`, `signed`
`String`	ASCII string	string

`var add` creates a new user-defined variable with the specified *name* that has the specified *datatype* and *class*. If you provide *num-elements* that are greater than 1, the resulting variable is an array.

An array can be referred to as a whole in an expression, in which case it evaluates to a simple `checksum` of its members. This reference is useful with the `changes` operator so that you can trigger an event whenever any element of an array changes.

Individual members of an array are selected by supplying the index as a parameter to the variable, in parentheses.

`var delete` deletes the specified user-defined variables.

Examples

To create a variable `zort` that holds a single-precision floating-point value, type:

```
var add zort float Word
```

To create an array, add an expression for the number of elements by typing:

```
var add zorts float Word 20
```

Then, you can set the tenth element, for example, by typing:

```
set zorts(9)=3.1415926535
```

See Also

`group`

Name

`version` — Print information that identifies the simulator

Synopsis

```
version
```

Description

The `version` command produces output similar to the following:

```
Simple mpsas Release 1.0; layers are:
-rw-rw-r--  1 ns           5914 Jul 19 18:37 ../simple/libsimple.a
-rw-rw-r--  1 ns        1075604 Jul 19 18:34 ../fw/libfw.a
-rw-rw-r--  1 ns        1156712 Jul 19 18:27 ../sparc/libsparc.a
-rw-rw-r--  1 ns         536980 Jul 19 18:10 ../computer/libcomputer.a
Made in NSE environment "sim1" by user "fred"
```

The first line identifies the architecture (in this case, `simple`) and the release of the simulator; this information comes directly from a string in file `vers.c` in the architecture directory.

The next few lines are an `ls -l` listing of the layers that constitute the simulator when the simulator was built. If you are using the Network Software Environment (NSE), the final line reports the user who made the simulator and the NSE in which it was done. If not, the line shows the directory of the `make` in place of the NSE.

Name

`wait` — Wait for simulation to stop before executing the next command

Synopsis

```
wait
```

Description

The `wait` command prevents the next command from being executed until the simulation has stopped. In the following example:

```
run; wait; print cpu1.pc
```

the `print cpu1.pc` command is not executed until simulation has stopped.

Examples

In the following example, when there is a hit (that is, `cpu1.pc` has the value 0x1000), `pc` is displayed. The `print cycle` command is not executed until simulation stops.

```
when (cpu1.pc == 0x1000) {print pc; wait; print cycle}
```

See Also

`when`

Name

`when` — An event that executes a series of commands when an expression is `true`

Synopsis

`when` [`-p` *period*] [`-d`] [*expr*] [{*command-list*}]

Description

The `when` command executes the sequence of commands, *command-list*, whenever the specified expression *expr* is `true` (a hit). The expression is evaluated every cycle while the simulation is running.

If you do not specify *expr*, the `when` event hits every cycle. If you do not specify *command-list*, `when` defaults to the `stop` command.

To display the status of the `when` events, use `when` without any parameters or the `status when` command. To delete a `when` event, use `delete`. To disable or enable a `when` event, use `disable` or `enable`, respectively.

Refer to *Expressions* on page 25 for details on expressions.

When the `when` event is created, the `ui` variable `cmd_result` is set to the event number.

Options

`-d` Adds the `when` event in a disabled state.

`-p` *period*
Causes the *command-list* to be executed every *period* hits. For example, the command `when –p 5 (cpu1.pc == 0x1000) {echo "5 more"}` causes `echo` to be executed once for every five hits. Without this option, *command-list* is executed every hit.

Examples

The following example, in which *command-list* is not specified, causes `stop` to be executed when the `pc` variable of `cpu1` has value 0x1000:

```
when (cpu1.pc == 0x1000)
```

`when {}`causes `stop` to be executed every cycle.

See Also

```
events
snoop
onstop
```

Name

`window` — Manage display windows

Synopsis

`window add` *window-name* [`-s`] [`-d` *display*] {*command-list*}
`window delete` [*window-name*]
`window update` [*window-name*]
`window` [`list` [*window-name*]]

Description

The `window` command manages the window facility. A window contains a list of commands that are executed every time the simulation stops. The output of the commands is displayed in the window.

`window add` creates a new window, called *window-name*. The program specified by the MPSAS `WINDOW_PROGRAM` environment variable is forked. If this variable does not exist, the program name defaults to `tty_tool`, located in the current directory or in the directory `../util`. The window program is passed the arguments specified by `WINDOW_ARGS`. If this variables does not exist, the window program is invoked with the following arguments:

`-WL` *window-name* `-Wl` *window-name* `-Wi`

`window` sends the output produced by *command-list* to the standard input of the window program every time the simulation stops.

The `tty_tool` program is an OpenWindows™ application for the `window` command.

`window delete` deletes *window-name* if one is specified; otherwise, it deletes all windows.

`window update` executes *command-list* associated with one or more windows so that their contents are updated. If you specify *window-name*, `window update` updates it; otherwise, it updates all windows. If you modify a variable being displayed in a window, for example, with the `set` command, the new value is not displayed until the simulation stops. You can use `update` to force the new value to be displayed.

`window list` displays the names of the current windows. If you specify *window-name*, `window list` displays information about that window only. With no parameters, `window list` displays the names of the current windows.

If scrolling is disabled (default), the window is refreshed by echoing the string specified by the `WINDOW_CLEAR` environment variable to the window before the command list associated with the window is executed. If this variable does not exist, it defaults to the string `\\012`.

The `echo` command echoes the clear string to the window and interprets the default clear string as `\012`; command processing replaces `\\` with `\`. Octal 012 is the ASCII code that clears the screen of the `tty_tool` program. To specify ASCII control characters in the `WINDOW_CLEAR` environment variable, precede each octal code with four backslashes.

Options

`-s` Causes the output to the window to scroll. By default, it is refreshed each time.

`-d` *display*
Displays the window on the specified OpenWindows display. This option causes the `-display` option to be passed to the window program when it is forked.

Examples

```
window add regs { cpu1.regs }
```

Displays the contents of the `cpu1` registers.

```
window add stack { cpu1.where }
```

Displays the stack backtrace for `cpu1`.

See Also

`when`
`onstop`

APPENDIX **C**

The `stand` Directory

The `stand` directory contains the source code for a number of small SPARC stand-alone programs. Included in this directory are trap tables, trap handlers, start-up code, utility routines, and program source. It gives you a working set of example SPARC programs and several utilities to facilitate the creation of simple stand-alone programs.

Note – This directory is not intended to provide a comprehensive general-purpose program development environment.

C.1 Overview

SPARC IUs and computer systems require a certain amount of setup before they become useful. Likewise, application programs need support before they can run. Typically, the operating system provides the setup and application program support. When writing a stand-alone program, you must link in all of this support with the program. The `stand` directory contains code that starts the SPARC processor and provides a minimal amount of support that is required by programs, as well as some example programs.

The files in the directory are divided into three categories:

- SPARC support
- Utility routines
- Example programs

TABLE C-1, TABLE C-2, and TABLE C-3 list the files in each of the categories.

TABLE C-1 SPARC Support Files in the `stand` Directory

File Name	Description
`crt.S`	Routines needed by the compiler
`srt0.S`	Start-up code for the processor
`trap.S`	Trap handlers
`trap_table.S`	Trap table

TABLE C-2 Utility Files in the `stand` Directory

File Name	Description
`ld_st_a.S`	C routines to access load alternate, store alternate, and atomic memory instructions
`mp.c`	Synchronization primitives for multiprocessor systems
`mpsas_trap.S`	C routines to access traps handled by the trap module
`printf.c`	A limited, small `printf` routine
`add_map.c`	A set of routines used by the `refmmu` and `mpmmu` programs to set up page tables

TABLE C-3 Example Programs in the `stand` Directory

File Name	Description
`delay.S`	A main program that determines the instruction delay for PSR writes
`fsqrt.s`	A main program for testing floating point
`inexact.c`	A main program that generates an inexact floating-point exception
`loop_test.c`	A main program that loops to keep the processor busy
`mpmmu.c`	A main program that starts a multiprocessor `mbus` system with MMU and caches on
`refmmu.c`	A main program that starts a single-processor system with MMU and cache on
`pad.s`	A page of initialized data used to force data to another page for the `mpmmu` and `refmmu` programs
`reg_read.c`	A main program that prints out the values of a few registers
`tutorial.c`	A main program for the tutorial in this book

C.2 SPARC Support

The stand directory includes code that is necessary to set up and run the SPARC processor: a trap table, trap handlers, start-up code, and routines that are required by the compiler.

The trap table is in trap_table.S. The majority of entries in the trap table stop the simulation with the trap number in register %l3 and the PSR in %l0. The overflow, underflow, and floating-point exception trap table entries branch to the handlers in trap.S. The reset trap table entry branches to the start-up code in srt.S. The start-up code sets up the processor's WIM, PSR, TBR, and stack pointer registers, as well as the FPU's FSR register, and then calls the function main. If main returns, the start-up code executes a special trap instruction that stops the simulation.

The C compiler generates code that depends on a number of routines. The stand directory supplies these routines in the crt.S file. Among them are routines to multiply, divide, and handle the returning of structures and call procedures through a pointer in a register.

C.3 Utility Routines

The stand directory includes a small set of utility routines to facilitate development of stand-alone programs. They provide the following functions:

- Access to load alternate, store alternate, and atomic instructions from C
- Multiprocessor synchronization primitives
- Access to traps handled by the trap module
- Formatted output

The following routines are described in greater detail in the manual pages at the end of this appendix, starting at page 233.

C.3.1 Access to Load Alternate, Store Alternate, and Atomic Instructions from C

The stand directory provides routines that give access to assembly load and store instructions that are not accessible directly from the C language. The loada and storea routines allow loads and stores to alternate ASIs.

Also, the functions ldstub and swap execute the ldstub and swap SPARC instructions.

C.3.2 Synchronization Primitives

The `stand` directory provides synchronization primitives. Blocking and nonblocking lock routines are named `lock_blk` and `lock_nblk`, respectively. The companion routine `unlock` releases the locks obtained by both lock routines.

Also available is a routine called `barrier`, which does not return on a processor until the specified number of processors have called `barrier`. It can ensure that all processors have reached a certain point in the code before any of them proceed.

C.3.3 Access to Traps Handled by the `trap` Module

MPSAS supports software traps that are signals to the simulator to provide a service. The processor does not jump into the trap table when these traps are executed. The simulator provides the service, and execution continues as if the trap has returned.

The services provided by this mechanism are:

- Termination of the simulation
- Issuance of commands to the simulator
- Input and output on the serial ports
- Access to a simulated disk

C.3.4 Formatted Output

The `stand` directory contains a `printf` routine, which can handle a maximum number of 10 arguments and supports only a few of the formats (`c`, `d`, `s`, `u`, `x`).

C.4 Example Programs

The procedure for building the example programs, as described below, is contained in `Makefile` in the `stand` directory. The `mpmmu`, `refmmu`, and `reg_read` programs only run on the `mbus` architecture. The remaining programs run on other architectures as long as RAM appears to start at address 0 for the processor (for example, by turning boot mode off before running the program).

C.4.1 delay

The delay program measures the number of instructions a write to the PSR is delayed before taking effect. It writes the PSR and then reads it into four different registers. Afterward, the program checks the registers to determine which of them contain the newly written value of the PSR. It then reports the number of instructions that are executed before the PSR changes to the new value.

C.4.2 fsqrt

The fsqrt program is a small program that exercises the FPU. This program executes three floating-point loads, three floating-point stores, and a square root operation in an infinite loop.

C.4.3 inexact

The inexact program, also a small program, generates an inexact IEEE floating-point exception. It investigates the behavior of the FPU while taking exceptions.

C.4.4 loop_test

The loop_test program keeps the processor busy long enough to test operations, such as interrupts. It maintains a loop, incrementing a global variable and printing out the latest value. A global variable is chosen to keep the compiler from optimizing the loop out if printf is taken out.

C.4.5 mpmmu

The mpmmu program tests the Mbus architecture and runs on a multiprocessor Mbus system. It sets up the processors with the MMUs and caches turned on and each processor-MMU combination in a different context. The contexts are very similar with only a difference in the stack's memory mapping.

This program prints out messages along the way to inform you of what is going on.

After each processor is set up, mpmmu calls the do_something function, which executes some atomic instructions, MMU probes, and cache flushes before returning. The main function waits for all processors to return from do_something before calling exit.

C.4.6 refmmu

The refmmu program can test the mbus architecture with the MMU and caches on and is similar to the mpmmu program. This program starts up a single processor mbus architecture machine by turning on the cache and the MMU. It also calls a routine called do_something when the MMU and cache are on, which performs a number of probes, atomic instructions, and cache flushes.

C.4.7 reg_read

The reg_read program is a simple program that prints out the values of the registers that are obtained via access to alternate ASIs.

C.4.8 tutorial

The tutorial program is the MPSAS tutorial. It does a calculation, including the factorial of two global variables.

Name

`loada`, `storea`, `swap`, `ldstub` — Access to alternate ASIs and atomic instructions

Synopsis

```
unsigned int loada(asi, address)
unsigned int asi, address;

void storea(asi, address, data)
unsigned int asi, address, data;

unsigned int swap(a, b)
unsigned int a, *b;

unsigned char ldstub(a)
unsigned char *a;
```

Description

The `loada` routine provides the ability to load from an alternate ASI. The parameter *asi* is the ASI from which to read; the parameter *address* is the address to access on that ASI. This routine only handles ASIs in the range 0–0x2f.

The `storea` routine provides the ability to store from an alternate ASI. The parameter *asi* is the ASI to which to store; the parameter *address* is the address to access on the ASI. The parameter *data* is the data to be written. This routine only handles ASIs in the range 0–0x2f.

The `swap` routine provides access to the atomic instruction `swap`. The two parameters are the same parameters as the instruction. This routine swaps the value `a` with the contents of the address `b`, using the `swap` instruction. The contents of `b` are returned.

The `ldstub` routine provides access to the atomic instruction `ldstub`. The argument *a* is a pointer to an address in memory, where the `ldstub` is done. The byte contents of the address `a` are returned, and the byte pointed to by `a` is set to 0xff atomically, using the `ldstub` instruction. The `ldstub` routine is used by the sychronization primitives.

See Also

Sync

Name

map_list — Build page table for a list of pages

Synopsis

```
#include "mapping.h"

map_list(ctxt, map_descrip, num_pages)
int ctxt;
struct map_page_descrip *map_descrip;
int num_pages;
```

Description

map_list builds a page table for a number of pages in a particular context for a standard reference MMU. The arguments specify the context number (*ctxt*), a description of the desired map (*map_descrip*), and the number of pages to map (*num_pages*).

The description of the pages to be mapped takes the form of an array of map_page_descrip structures. Each structure defines the desired map for a particular page. Following is the map_page_descrip structure:

```
struct map_page_descrip {
        unsigned phys_page;
        unsigned virt_page;
        unsigned access;
        unsigned cacheable;
};
```

The virt_page field is set to the virtual address to be mapped. The phys_page field specifies the physical address to be associated with the virtual address in virt_page. Because physical addresses for the standard reference MMU are 36 bits wide, the phys_page field is set to the top 32 bits of the physical address; the bottom 4 bits are assumed to be 0. The access field specifies the access to the page and should be set to a constant based on TABLE C-4.

The cacheable field specifies whether the page is cacheable. This field should be set to 1 for cacheable and 0 for noncacheable.

TABLE C-4 `access` Field Values

User Access	Supervisor Access	Constant	Value
Read-only	Read-only	`SR__UR__`	0
Read-write	Read-write	`SRX_URX_`	1
Read-execute	Read-execute	`SR_XUR_X`	2
Read-write-execute	Read-write-execute	`SRWXURWX`	3
Execute-only	Execute-only	`S__XU__X`	4
Read-only	Read-write	`SRW_UR__`	5
No access	Read-execute	`SR_XU___`	6
No access	Read-write-execute	`SRWXU___`	7

Name

`printf` — Formatted output

Synopsis

```
int printf( format [ ,arg ] ...)
char *format;
int arg;
```

```
int fprintf(port, format, [ ,arg ] ...)
int port;
char *s;
int arg;
```

Description

This `printf` utility is a limited version of the standard `printf`. It is limited to a maximum of 10 arguments besides the `format` string. The only formats supported are `c`, `d`, `s`, `u`, and `x`.

See Also

`printf` manual page

Name

exit, mpsas_cmd, char_out_a, char_out_b, char_in_a, char_in_b, char_waiting_a, char_waiting_b, s_open, s_read, s_write, s_lseek — Access to simulator supported traps

Synopsis

```
void exit(status)
int status;

void mpsas_cmd(str)
char *str;

void char_out_a(c)
char c;

void char_out_b(c)
char c;

int char_in_a()

int char_in_b()

int char_waiting_a()

int char_waiting_b()

int s_open(part, flag)
int part;
int flag;

int s_read(fd, buffer, size)
int fd;
char *buffer;
int size;

int s_write(fd, buffer, size)
int fd;
char *buffer;
int size;

int s_lseek(fd, offset)
int fd;
int offset;
```

Description

Each of the simulator trap routines corresponds to a trap supported by the simulator. The routines set up the proper arguments and execute the proper trap instructions to signal the simulator to perform a particular service.

`exit` executes a trap instruction that stops the simulation. The program calls this function to stop the simulation and return to an MPSAS user-interface prompt.

`mpsas_cmd` routine passes a string to MPSAS to be executed as a user-interface command. The maximum allowed length of the command is 99 characters. You can use the backslash character to create longer commands.

The simulator supports a number of traps that manipulate the serial ports. These are for character input, output, and polling. There is a trap for each of these operations for each of two serial ports, making six total traps. The detailed behavior is described in *trap: External Trap Module* on page 155.

`char_in_a` and `char_in_b` retrieve characters from the `a` and `b` serial ports. If no characters are available, these routines return –`1`. The routines return characters in the least significant byte of the integer.

`char_out_a` and `char_out_b` send characters to the `a` and `b` serial ports. The character should be placed in the least significant byte of the integer.

`char_waiting_a` and `char_waiting_b` check the serial ports to see if data is waiting to be read. They return a nonzero number if a character is waiting, `0` if no characters are waiting.

`s_open`, `s_read`, `s_write`, and `s_lseek` manipulate the simulated disk partitions. Their detailed behavior is described in *trap: External Trap Module* on page 155. These routines pass the arguments expected by the `trap` module.

`s_open` returns a handle to the partition you are interested in using. The *partition* argument specifies the partition number. Partition numbers are allotted by the `simdisk` module based on the order in the `simdisk` initialization file.

`s_read` and `s_write` read and write from the raw partition. The *fd* argument specifies the partition number and is obtained from the `s_open` call. The *buffer* and *size* arguments specify the data and size of the transaction.

`s_lseek` seeks to a different spot on the simulated partition and subsequent reads and writes start at that point in the partition. The *fd* argument specifies the partition on which you are to operate. The *offset* argument specifies the new position in the partition.

See Also

trap: External Trap Module on page 155
serial: Dual Serial Port Module on page 132
simdisk: Simulated Disk Module on page 137

Name

lock_blk, lock_nblk, unlock, barrier — Synchronization primitives

Synopsis

```
#include "mp.h"

LOCKDEC(x);

void lock_blk(lck)
lock_t *lck;

unsigned int lock_nblk(lck)
lock_t *lck;

void unlock(lck)
lock_t *lck;

BARDEC(x);

void barrier(bar, num_procs)
barrier_t *bar;
int num_procs;
```

Description

The lock_blk routine is a blocking lock primitive. The routine does not return until the lock pointed to by *lck* has been acquired.

The lock_nblk routine is a nonblocking lock primitive. The routine returns after attempting to acquire the lock pointed to by *lck*. If the caller did acquire the lock, the return value is GOT_LOCK (0). If the caller did not acquire the lock, DIDNT_GET_LOCK (0xff) is returned.

The unlock routine releases the locks obtained by lock_blk and lock_nblk.

Locks are declared and initialized by the LOCKDEC macro.

The barrier routine does not return on a processor until the specified number of processors have called barrier. It can be used to ensure that all processors have reached a certain point in the code before they proceed. The parameter *bar* is the barrier being operated on. The pointer passed to barrier in the *bar* parameter should be declared and initialized with the BARDEC macro. The parameter *num_procs* is the number of processors that call barrier before any of them return.

These synchronization routines use the ldstub routine.

See Also

Load and store

Index