IBM Full-System Simulator User's Guide

Modeling Systems based on the Cell Broadband Engine Processor

Version 3.1

IBM Full-System Simulator User’s Guide

© International Business Machines Corporation (2009). All Rights Reserved.

Printed in the United States of America May 2009.

May only be used pursuant to any IBM License Agreement, or any Addendum to IBM License Agreement. No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual, or otherwise, without prior written permission of IBM Corporation. IBM Corporation grants you limited permission to make hardcopy or other reproductions of any machine-readable documentation for your own use, provided that each such reproduction shall carry the IBM Corporation copyright notice. No other rights under copyright are granted without prior written permission of IBM Corporation.

All information contained in this document is subject to change without notice. The information contained in this document does not affect or change IBM product specifications or warranties. Nothing in this document shall operate as an express or implied license or indemnity under the intellectual property rights of IBM or third parties. All information contained in this document was obtained in specific environments, and is presented as an illustration. The results obtained in other operating environments may vary.

While the information contained herein is believed to be accurate, such information is preliminary, and should not be relied upon for accuracy or completeness, and no representations or warranties of accuracy or completeness are made. This document contains information on products in the design, sampling and/or initial production phases of development. This information is subject to change without notice, and is provided without warranty of any kind. The document is not intended for production.

THE INFORMATION CONTAINED IN THIS DOCUMENT IS PROVIDED ON AN “AS IS” BASIS. In no event will IBM be liable for damages arising directly or indirectly from any use of the information contained in this document.

U.S. Government Users Restricted Rights—Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corporation.

IBM is a registered trademark of International Business Machines Corporation in the United States, other countries, or both.

All S/T/I rights and copyrights apply. The IBM logo, PowerPC, PowerPC logo, and PowerPC architecture are trademarks of International Business Machines Corporation in the United States, or other countries, or both.

Linux is a registered trademark of Linus Torvalds. Linux is written and distributed under the GNU General Public License in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.

The IBM home page can be found at www.ibm.com.

SystemSimCELLUsersGuide, Version 3.1

Contents

Preface v

1 **Introduction to the IBM Full-System Simulator** 1
- Simulator Overview. 2
- Invoking the Simulator 2
- Simulator Basics 3
  - Interacting with the Simulator. 3
  - Operating-System Modes. 4
- Changing the Configuration of the Simulated System. 5
  - Starting a Simulation in SMP Mode 5
  - Specifying the Processor Architecture. 6
  - Changing the Size of System Memory in the Configuration 6
  - Modifying the root file system. 7

2 **The SystemSim Graphical User Interface** 9
- Graphical User Interface 10
  - The Simulation Panel. 11
  - GUI Buttons. 20

3 **SystemSim Command Syntax and Usage** 25
- Understanding and Using Simulator Commands 26
- Defining and Managing a Simulated Machine 28
  - A typical initial run script. 29
- Summary of Top-Level Simulator Commands. 30

4 **Debugging Features in SystemSim** 33
- Detecting SPU Stack Overflow 34
- Bus errors caused by DMA errors 35
- Kernel debugging 36

5 **Accessing the Host Environment**. 39
- The Callthru Utility 40
- Bogus Network Support 40
  - Bogusnet Tcl functions 40
  - Extended Description of Bogusnet support. 41
  - Setting up TUN/TAP on the host system 41
  - Setting up a TUN/TAP interface for a non-root user 41

© International Business Machines Corporation. All rights reserved.

Configuring SystemSim support for Bogus Net 42
The bogus net device driver 43
Connecting to the simulation host 43
Troubleshooting BogusNet 43
Simulation Library 43

© IBM Corporation. All rights reserved.

Preface

The IBM Full-System Simulator, internally referred to as “Mambo,” has been developed and refined by the IBM Austin Research Lab (ARL) in conjunction with several large system design projects built upon the IBM Power architecture. As an execution-driven, full-system simulator, the IBM Full-System Simulator has facilitated the experimentation and evaluation of a wide variety of system components for core IBM initiatives. The IBM Full-System Simulator for the Cell Broadband Engine (Cell/B.E.) Processor, available from the IBM alphaWorks Emerging Technologies web site, enables development teams both within IBM and externally to simulate a Cell/B.E. processor-based system in order to develop and enhance application support for this platform.

The *IBM Full System Simulator User’s Guide* describes the basic structure and operation of the IBM Full-System Simulator and its graphic user interface (GUI) and command line user interface.

Intended Audience

This document is intended for designers and programmers who are developing and testing applications that are designed to run on systems based on the Cell Broadband Engine Processor. Potential users include:

- System and software designers
- Hardware and software tool developers
- Application and product engineers
- Validation engineers

Using This Version of the Guide

This version of the *IBM Full System Simulator User’s Guide* applies to version 3.1 of the IBM Full-System Simulator for the Cell Broadband Engine Processor, available from IBM’s alphaWorks Emerging Technologies website located at http://www.alphaworks.ibm.com/tech/cellsystemsim.

The guide is organized into topics that cover concepts and procedures for execution and analysis of CBEA applications. This book includes the following chapters and appendices:

- *Chapter 1, Introduction to the IBM Full-System Simulator,* describes the IBM Full System Simulator developed by the IBM Austin Research Lab (ARL), and introduces the Cell Broadband Engine Architecture (CBEA) modeled by the IBM Full-System Simulator.
- *Chapter 2, The SystemSim Graphical User Interface*, provides an overview of the graphical user interface of the IBM Full System Simulator
- *Chapter 3, SystemSim Command Syntax and Usage*, describes the IBM Full System Simulator command framework and introduces the structure, format, and usage of simulator commands.

© International Business Machines Corporation. All rights reserved.

- *Chapter 4, Debugging Features in SystemSim* describes some of the simulator's debugging features that are specifically designed for SPU debugging.
- *Chapter 5, Accessing the Host Environment* describes several mechanisms that are provided to allow interactions between the host and simulated systems.

What's New in this Release

Release 3.1of the IBM Full-System Simulator for the Cell Broadband Engine provides the following enhancements:

- Support for systems with Tcl/Tk 8.5.

Conventions

This guide provides screen captures to illustrate example interface elements and uses code samples to represent example implementations. Your software interface or development environment may vary from these examples depending on your system and product environment.

The following typographical components are used for defining special terms and command syntax:

Table 2-1. Typographical Conventions

Convention	Description
Bold typeface	Represents literal information, such as: Information and controls displayed on screen, including menu options, application pages, windows, dialogs, and field names. Commands, file names, and directories. In-line programming elements, such as function names and XML elements when referenced in the main text.
Italics typeface	Italics font is used to emphasize new concepts and terms, and to stress important ideas. Additionally, book and chapter titles are displayed in italics.
`Sans serif` typeface	Represents example code output, such as XML output or C/C++ code examples. ***`Italic sans serif`*** words or characters in code and commands represent values for variables that you must supply, such as arguments to commands or path names for your particular system. For example: **`cd /users/your_name`**
... (Horizontal or Vertical ellipsis) . . .	In format and syntax descriptions, an ellipsis indicates that some material has been omitted to simplify a discussion.
{ } (Braces)	Encloses a list from which you must choose an item or information in syntax descriptions.
[ ] (Brackets)	Encloses optional items in format and syntax descriptions. For example, in the statement `SELECT [DISTINCT], DISTINCT` is an optional keyword.
| (Vertical rule)	Separates items in a list of choices enclosed in { } (braces) in format and syntax descriptions.
UPPERCASE	Indicates keys or key combinations that you can use. For example, press CTRL + ALT + DEL.

© IBM Corporation. All rights reserved.

Table 2-1. Typographical Conventions

Convention	Description
Hyperlink	Web-based URIs are displayed in blue text to denote a virtual link to an external document. For example: http://www.ibm.com
NOTE This is note text.	The note block denotes information that emphasizes a concept or provides peripheral information.

Related Guides and Recommended Reference

The IBM Full-System Simulator

Performance Analysis with the IBM Full-System Simulator describes facilities and techniques for application and system performance analysis using the IBM Full-System Simulator. This includes facilities to capture and process statistics on the performance of computation kernels, library routines, and full applications in the context of a full-system execution. Performance Analysis with the IBM *Full-System Simulator* is commonly distributed with alphaWorks releases in the docs directory as SystemSim.PerfAnalysis.pdf.

The simulator's command interface is implemented as an extension of the Tool Control Language (Tcl), and the graphical user interface is implemented in the Tk package. Information about Tcl and Tk syntax and features can be found in:

- *Practical Programming in Tcl and Tk* by Brent B. Welch. Prentice Hall, Inc.

The Cell/B.E. processor

Documentation for the Cell/B.E. processor is available in the IBM online technical library at:

http://www.ibm.com/chips/techlib/techlib.nsf/products/Cell_Broadband_Engine

Among the documents in this library, the following are particularly helpful in understanding the operation of the IBM Full- System Simulator:

- *Cell Broadband Engine Architecture*
- *Cell Broadband Engine Programming Handbook*
- *Cell Broadband Engine Registers*

PowerPC Base

The following documents can be found on the developerWorks Web site at:

http://www.ibm.com/developerworks/eserver/library

- *PowerPC Architecture Book, Version 2.02*
  - *Book I: PowerPC User Instruction Set Architecture*
  - *Book II: PowerPC Virtual Environment Architecture*
  - *Book III: PowerPC Operating Environment Architecture*

Contacting the IBM Full-System Simulator Development Team

The IBM Full-System Simulator development team at ARL is very interested in hearing from you about your experience with the IBM Full-System Simulator and its supporting information set. Should you have questions or encounter any

issues with the IBM Full-System Simulator, visit the project forum at
http://www.alphaworks.ibm.com/tech/cellsystemsim/forum.

© IBM Corporation. All rights reserved.

CHAPTER 1

Introduction to the IBM Full-System Simulator

This chapter provides an overview of the IBM Full System Simulator, SystemSim, developed by the IBM Austin Research Lab (ARL), and provides concepts and procedures for using the simulator for the Cell Broadband Engine Processor. It also describes configuration parameters for setting up and running the simulation environment in standalone and Linux mode. Topics in this chapter include:

- *Simulator Overview*
- *Invoking the Simulator*
- *Simulator Basics*
- *Changing the Configuration of the Simulated System*

© International Business Machines Corporation. All rights reserved.

Simulator Overview

The IBM Full-System Simulator for the Cell Broadband Engine is a generalized simulator that can be configured to simulate a broad range of full-system configurations. It supports *functional simulation* of complete systems based on the Cell Broadband Engine processor, including simulation of the PPE, SPUs, MFCs, memory, disk, network, and system console. The SDK, however, provides a ready-made configuration of the simulator for Cell Broadband Engine system development and analysis. The simulator also includes support for *performance simulation* (or timing simulation) of certain components to allow users to analyze performance of Cell Broadband Engine applications. It can simulate and capture many levels of operational details on instruction execution, cache and memory subsystems, interrupt subsystems, communications, and other important system functions.

Figure 1-1 shows the simulation stack. The simulator is part of the software development kit (SDK), which is available through IBM alphaWorks Emerging Technologies at http://www.alphaworks.ibm.com/tech/cellsystemsim.

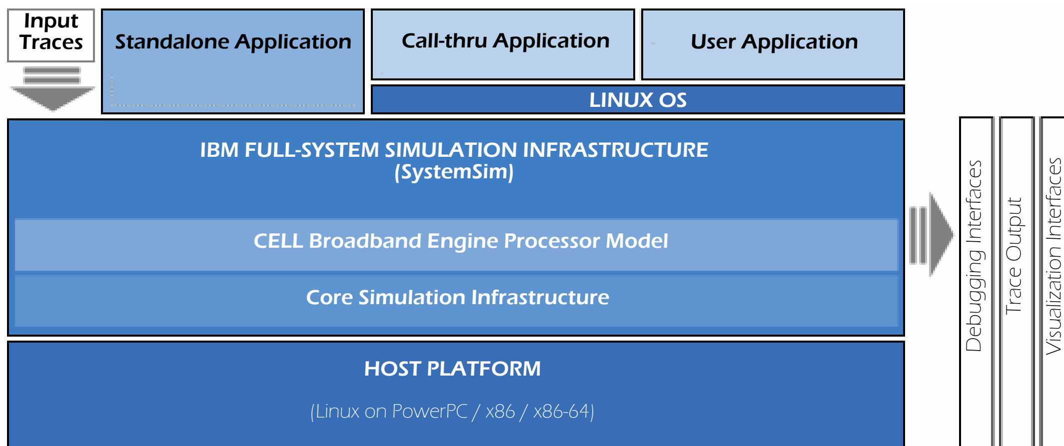


Figure 1-1. Simulator Stack for the Cell Broadband Engine

If accurate timing information and performance statistics are not required, the simulator can be used in its *functional-only mode*, simulating the architectural behavior of the system to test the functions and features of a program. For performance analysis, the simulator can be used in *performance simulation* mode. Simulator configurations are extensible and can be modified using Tool Command Language (Tcl) commands to produce the type and level of analysis required.

The simulator is a general tool that can be configured for a broad range of microprocessors and hardware simulations. The SDK, however, provides a ready-made configuration of the simulator for Cell Broadband Engine system development and analysis.

Invoking the Simulator

The simulator is invoked using the systemsim command. This command is in the bin directory of the systemsim-cell release, which should be added to the user's PATH before invoking systemsim.

NOTE By default, the simulator is installed in the /opt/ibm/systemsim-cell directory. This directory will be used in all the examples shown in this book.

When the simulator starts, it loads an initial run script which typically configures and initializes the simulated machine. The name of the initial run script can be passed to systemsim with the -f option. When not specified on the command line, the simulator will look in the current directory for the file .systemsim.tcl, and if present, will use this file as the initial run script. Otherwise, it will use the file lib/cell/systemsim.tcl provided with the systemsim-cell release. When specified

© IBM Corporation. All rights reserved.

using the -f option, the name of the initial run script can contain an absolute or relative path. The simulator searches for Initial run scripts with a relative path by first looking in the current directory, and then in the lib/cell directory of the systemsim-cell release, and finally in the lib directory of the systemsim-cell release. If the simulator fails to find the initial run script specified with the -f option, it issues an error message and exits.

It is generally the task of the initial run script to locate the operating system and filesystem images to be used by simulated machine. For the Cell simulator, the default initial run script searches for a Linux kernel image named vmlinux and a filesystem image named sysroot_disk. The script will look first in the current directory and then in the systemsim-cell/images/cell directory, and uses the first instance it finds for these images. If the script fails to find either of these images in one of these locations, it will print an error message and terminate the simulator.

The following examples illustrate various ways to invoke the simulator. These examples assume that the simulator was installed into /opt/ibm/systemsim-cell.

1. To start the simulator with the GUI window enabled, specify the "-g" option on the command line when invoking systemsim. For example, to run the simulator with the GUI using either the user's .systemsim.tcl or the simulator's lib/cell/systemsim.tcl as the initial run script, issue:

   ```
   PATH=/opt/ibm/systemsim-cell/bin:$PATH systemsim -g
   ```

2. To run the simulator without the GUI, issue:

   ```
   PATH=/opt/ibm/systemsim-cell/bin:$PATH systemsim
   ```

   If the user has created a run script named .systemsim.tcl in the current directory, the simulator will use this as the initial run script. Otherwise, the simulator uses systemsim.tcl in the lib/cell directory of the systemsim-cell release as the initial run script.

3. To run the simulator without the gui, without a console window (-n), in quiet mode (-q), using the initial run script myrun.tcl, issue:

   ```
   PATH=/opt/ibm/systemsim-cell/bin:$PATH systemsim -n -q -f myrun.tcl
   ```

When the simulator starts, the window in which it was started becomes the simulator command window where you can enter simulator commands. The simulator also creates the console window (unless this was disabled with -n) which is initially labeled *UART0* in the window's title bar, and a GUI window if this was requested with the -g option.

Simulator Basics

Interacting with the Simulator

There are two ways to interact with the simulator:

- Issuing commands to the *simulated system*
- Issuing commands to the *simulator*

The simulated system is the Linux environment on top of the simulated Cell Broadband Engine, where you run and debug programs. You interact with it by entering commands at the Linux command prompt, in the *console window*. The console window is a Linux shell of the simulated Linux operating system.

You can also control the simulator itself, configuring it to do such tasks as collect and display performance statistics on particular SPEs, or set breakpoints in code. These commands are entered at the simulator command line in the *simulator command window*, or using the equivalent actions in the graphical user interface (GUI). The GUI is a graphical means of interacting with the simulator. The GUI is described in "The SystemSim Graphical User Interface" on page 9.

Figure 1-2 shows the simulator windows, and the layers with which they communicate.

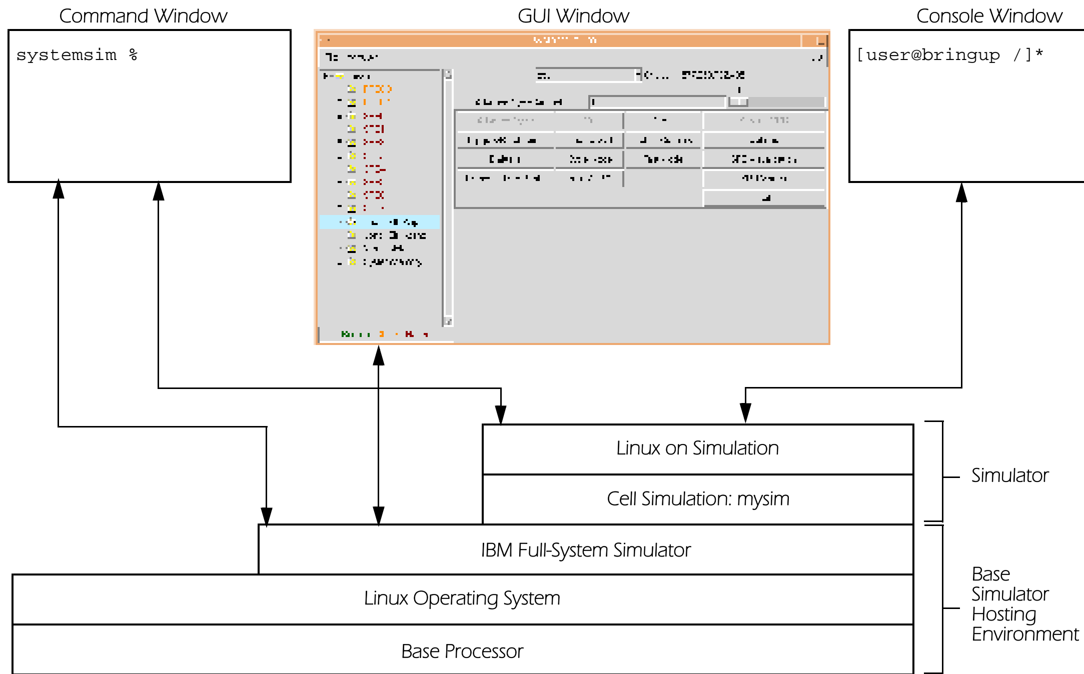


Figure 1-2. Simulator Structure and Screens

All simulator commands must be entered at the prompt in the command window (that is, the window in which the simulator was started). Some of the important commands are shown in Table 1-1:

Table 1-1. Top-level commands for the IBM Full-System Simulator for the Cell Broadband Engine

Simulator Command	*Description*
quit	Closes the simulation and exits the simulator.
help	Displays a list of the available simulator commands.
mysim go	Starts or continues the simulation. The first time it is issued, the simulator boots the Linux operating system on the simulation.
mysim spu n set model ***mode***	Sets SPEn into model ***mode***, where n is a value from 0 to 7 and ***mode*** is pipeline, instruction. or fast.
mysim spu n stats print	Displays to the simulator command window, the performance analysis statistics collected on SPU*n*, where n is a value from 0 to 7. Statistics are only collected when the SPU is executing in pipeline mode.

The simulator prompt is displayed in the command window when the simulation is stopped, or paused. When the simulation is running, the command window also displays a copy of the output to the console window and simulation-cycle information every few seconds, and the prompt is not available. To stop the simulation and get back the prompt—use the Ctrl-C key sequence. This will stop the simulation, and the prompt will reappear.

Operating-System Modes

A key attribute of the IBM Full-System Simulator is its ability to boot and run a complete PowerPC system. By booting an operating system, such as Linux, the IBM Full-System Simulator can execute many typical application programs that utilize standard operating system functionality. Alternatively, applications can be run in *standalone* mode, in which all operating system functions are supplied by the simulator and normal operating system effects do not occur, such as paging and scheduling. The IBM Full-System Simulator can also execute SPU programs in standalone mode

© IBM Corporation. All rights reserved.

on a given SPU. These two approaches to running applications on the simulator are referred to as Linux mode and standalone mode.

- *Linux Mode*. In Linux mode, after the simulator is configured and loaded, the simulator boots the Linux operating system on the simulated system. At runtime, the operating system is simulated along with the running programs. The simulated operating system takes care of all the system calls, just as it would in a nonsimulation (real) environment.
- *Standalone Mode*. In standalone mode, the application is loaded without an operating system. Standalone applications are user-mode applications that are normally run on an operating system. On a real system, these applications rely on the operating system to perform certain tasks, including loading the program, address translation, and system-call support. In standalone mode, the simulator provides some of this support, allowing applications to run without having to first boot an operating system on the simulator.

However, there are limitations that apply when building an application to be loaded and run by the simulator without an operating system. For example, applications should be linked statically with any libraries they require since the standard operating system shared libraries are not available in standalone mode. Another example is support for virtual memory address translation. Typically, the operating system provides address-translation support. Since an operating system is not present in standalone mode, the simulator loads executables without address translation, so that the effective address is the same as the real address. Therefore, all addresses referenced in the executable must be valid real addresses. If the simulator has been configured with 64 MB of memory, all addresses must fit in the range of x'0' to x'3FFFFFF'.

Changing the Configuration of the Simulated System

Starting a Simulation in SMP Mode

The default system configuration that is used when the simulator is started is a system with a single Cell Broadband Engine processor. Beginning in Version 1.1, the simulator provides a means to specify an alternate system configuration that uses two Cell Broadband Engine processors, referred to as SMP mode or Dual BE mode. There are a variety of ways to specify the SMP configuration, but for most users, the best approach is to simply use an alternate initial run script provided with the simulator. This script is called config_smp.tcl, and is located in the lib/cell directory of the systemsim-cell release. To use this run script to start the simulator with a configuration with two Cell Broadband Engine processors, issue:

```
PATH=/opt/ibm/systemsim-cell/bin:$PATH systemsim -f config_smp.tcl
```

The config_smp.tcl script can also be sourced, using the Tcl source command, from within a user's run script, which allows scripted simulator executions for SMP configurations.

There is also another approach to configuring the simulator to use an SMP mode configuration. This approach utilizes a special procedure, called config_hook, which is called from the lib/cell/systemsim.tcl file to modify the configuration before the machine is instantiated. In this approach, the config_dual_be command is used to modify the configuration to have two Cell Broadband Engine processors. The following example mySMPsim.tcl file illustrates a custom initial run script containing sample code that the config_hook procedure to issue the config_dual_be command to modify the system configuration.

```
0:  # top-level script to set SMP mode configuration
1:
2:  proc config_hook {conf} {
3:   config_dual_be $conf
4:  }
```

```
5:
6:  source .systemsim.tcl
```

The config_hook procedure is a general mechanism to modify the configuration for a simulated machine. In particular, line 3 configures the machine to run in SMP mode. You can add additional configuration settings to the config_hook procedure in the same manner.

1. To start the simulator with the GUI window enabled using the initial run script mySMPsim.tcl that contains newly defined SMP mode setting issue:

   ```
   PATH=/opt/ibm/systemsim-cell/bin:$PATH systemsim -g -f mySMPsim.tcl
   ```

   The simulator GUI is launched, in which each BE and its PPE and SPE components are displayed in the vertical panel. See Chapter 2, "The SystemSim Graphical User Interface" for more information about GUI elements and windows.

2. To run the simulator without the gui using the initial run script mySMPsim.tcl that contains newly defined SMP mode setting issue:

   ```
   PATH=/opt/ibm/systemsim-cell/bin:$PATH systemsim -f mySMPsim.tcl
   ```

Specifying the Processor Architecture

The simulator provides support for both the Cell/B.E. processor and the PowerXCell 8i processor. The PowerXCell 8i processor is a CBEA-compliant processor with a fully pipelined, enhanced double precision SPU. The processor also supports five optional instructions to the SPU Instruction Set Architecture. The default configuration uses the Cell/B.E. processor, but there are a variety of ways to create and run simulations using the PowerXCell 8i processor.

The simulator now accepts a new command line option, **`-mcelledp`**, which specifies that the PowerXCell 8i should be used as the default processor for the simulation. The following example shows how to use the command line option to start the simulator with a configuration using the PowerXCell 8i processor:

```
PATH=/opt/ibm/systemsim-cell/bin:$PATH systemsim -mcelledp
```

The simulator also provides an alternate initial run script, config_edp_smp.tcl, to create an SMP configuration with two PowerXCell 8i processors, similar to an IBM BladeCenter QS22. This run script is located in the lib/cell directory of the systemsim-cell release. To use this run script, issue the command:

```
PATH=/opt/ibm/systemsim-cell/bin:$PATH systemsim -f config_edp_smp.tcl
```

Changing the Size of System Memory in the Configuration

New configuration settings have been introduced to allow the simulator to be configured with memory sizes larger than 1GB. However, the proper use of these settings is somewhat complicated, so a new tcl procedure, config_memory_size, has been provided to allows users to specify the memory size per Cell BE. This procedure is used as follows

```
config_memory_size <config-object> <memory-size>
```

where `<config-object>` is the name of the configuration object to update, and `<memory-size>` is the amount of memory to be allocated for each BE in the configuration. The `<memory-size>` value can be specified with an 'MB' (megabytes) or 'GB' (gigabytes) suffix. When the memory size value is larger than 1GB, then it must be some multiple of 1GB. This tcl procedure must be used to modify the configuration before the simulated machine is created. This means that `config_memory_size` must be invoked within the config_hook procedure provided by systemsim.tcl or in a user-written startup tcl script.

 © IBM Corporation. All rights reserved.

Modifying the root file system

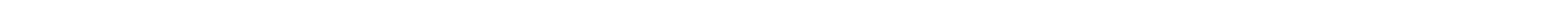
A root file system disk image (sysroot disk) for the Cell version of the simulator is provided as part of the SDK for Multicore Acceleration. For version 3.1 of the SDK, you can simply download and install the sysroot disk file image from:

http://www.bsc.es/projects/deepcomputing/linuxoncell/cellsimulator/sdk3.1/CellSDK-Open-Fedora/cbea/sysroot_image-3.1-1.noarch.rpm

The details of creating a sysroot disk file can be rather complicated. For those curious about how to create a sysroot disk file from scratch, you can download the sysroot_image source rpm and study the rpm spec and build scripts to learn how this sysroot disk file was created.

The sysroot disk provided with the IBM SDK for Multicore Acceleration contains a minimal set of build tools and libraries to limit the size of the disk image file. This should be sufficient for running most simple applications, but in some cases users will want additional packages installed into the sysroot disk. There are several ways to modify the contents of the sysroot disk:

1. Mount file system using '-o loop' and simply copy files to it. This is the simplest way, but it may require root privileges on some systems, and it is appropriate only for simply file copies. If you use this approach, make sure the simulator is not running while you are making your changes, and make sure to unmount the disk image before restarting the simulator.
2. If you want to install additional rpms into the sysroot disk, and you have access to a PowerPC system (such as a PS3), you can mount the file system as in option 1 and install the rpms using the rpm command, specifying mount point on the –root option and the –noscripts option. This approach only works on a PowerPC host system because the rpm database on the sysroot disk is for a PowerPC system, and its data format is not readable by x86 architecture systems. As in option 1, make sure the simulator is not running while you are making your changes, and make sure to unmount the disk image before restarting the simulator.
3. The third approach is to modify the sysroot disk within the simulator. By default, the simulator writes filesystem changes to a copy-on-write ("cow") file, which are then discarded at the end of the simulation, so that each simulation will start in a deterministic state. To make persistent changes to the sysroot disk image, you must start the simulator with an alternate startup script that specifies that changes should be written directly to the sysroot disk file. The easiest way to create this alternate startup script is to make a copy of the systemsim.tcl startup script (in lib/cell), and change the 'newcow' option on the bogus disk init command to 'rw'. Now you can modify the sysroot disk using the simulator as follows:
   - Start the simulator with the alternate startup script.
   - Wait for the shell prompt to appear in the console window.
   - To copy normal files to the sysroot, simply use the callthru source command from the console window to copy the files into place from the host filesystem.
   - To install rpms, first copy the rpm file to the simulator file system with callthru source. Then install the package with rpm -Uvh <package>.rpm. After the package is installed, you can delete the rpm file to preserve space in the simulated file system.
   - When all packages are installed, issue two consecutive sync commands ('sync;sync') to make sure all changes are written to the sysroot disk file.
   - Exit the simulator.



 © IBM Corporation. All rights reserved.

CHAPTER 2

The SystemSim Graphical User Interface

This chapter provides an overview of the graphical user interface of the IBM Full System Simulator. Topics in this chapter include:

- *Graphical User Interface*

© International Business Machines Corporation. All rights reserved.

Graphical User Interface

The simulator's GUI offers a visual display of the state of the simulated system, including the PPE and the eight SPEs. You can view the values of the registers, memory, and channels, as well as viewing performance statistics. The GUI also offers an alternate method of interacting with the simulator. Figure 2-1 shows the main GUI window that appears when the GUI is launched.

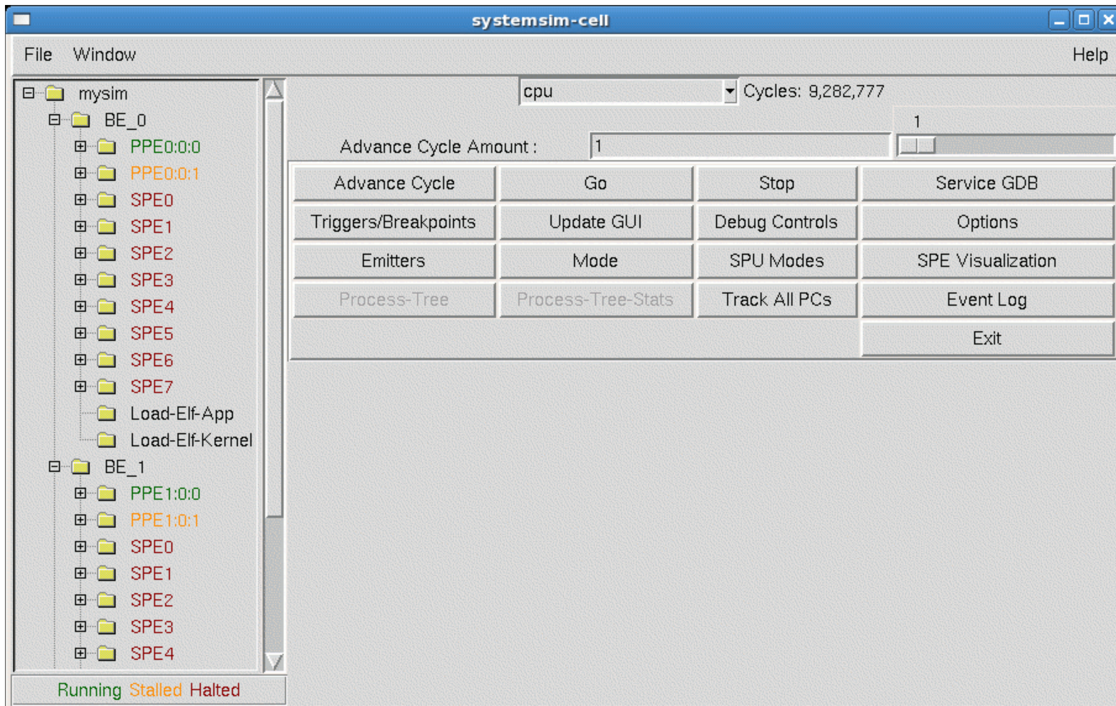


Figure 2-1. Graphical User Interface for the Simulator

The main GUI window has two basic areas: the vertical panel on the left, and the rows of buttons on the right. The vertical panel represents the simulated system and its components. The rows of buttons are used to control the simulator.

When the simulator is started it creates a simulated machine containing a Cell Broadband Engine processor and displays the main GUI window, labeled with the name of the simulator program. When the GUI window first appears, click the *Go* button to boot the Linux operating system.

If the simulator is launched in SMP, or dual Cell-based system, mode (see "Simulator Basics" on page 3), the vertical panel in the main window displays each BE with its components, as shown in Figure 2-2.

© IBM Corporation. All rights reserved.

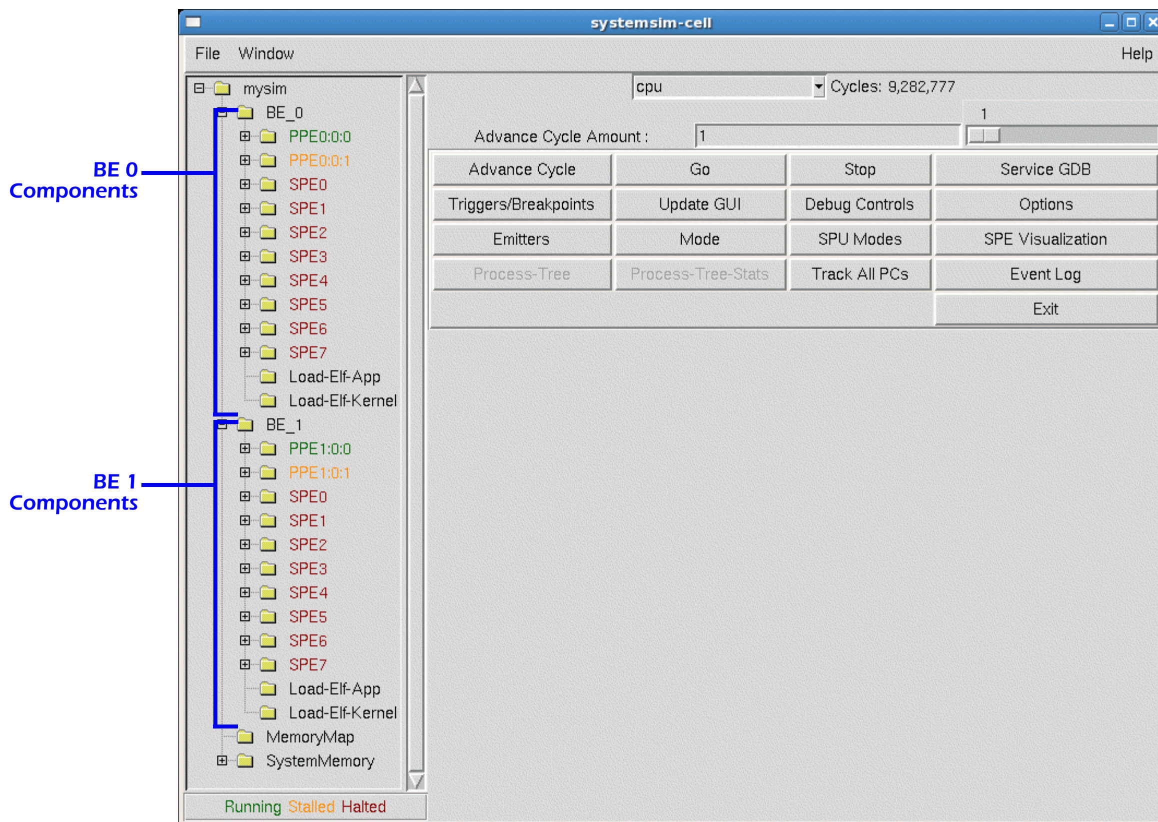


Figure 2-2. Simulator Graphical User Interface Started in SMP Mode

The Simulation Panel

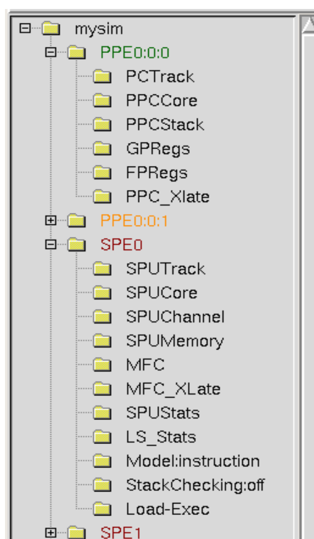


Figure 2-3. Project and Processor Folders

When the main GUI window first appears, the vertical panel contains a single folder labeled *mysim*. To see its contents, click on the plus sign (+) in front of the folder icon. When the folder is expanded, you can see its contents; these include a PPE (labelled *PPE0:0:0* and *PPE0:0:1*, the two threads of the PPE), and eight SPEs (*SPE0... SPE7*). Processor labels are color-coded to indicate the current state (running/stalled/halted) of the processor. The folders representing the processors can be further expanded to show the viewable objects and the options and actions available. Figure 2-3 on page 142 shows the vertical panel with several of the processor folders expanded.

NOTE When the simulator is launched in SMP mode, each BE (BE 0 and BE 1) contains the same contents described in this section.

PPE Components

There are six PPE components visible in the expanded PPE folder: PCTrack, PCCCore, PPCStack, GPRegs, FPRegs and PPC_Xlate. Double-clicking a folder icon brings up a window displaying the program-state data. Several of the available windows are shown in the following figures.

The PPE PC Tracker window is displayed when the user double-clicks on the PCTrack folder icon. An example of this window is shown in Figure 2-4. The window displays the region of storage containing the instructions that are currently being executed by the PPE. Each line of the window shows the effective and real address of a word in

storage and its contents in hexadecimal, ASCII, and as a disassembled PowerPC instruction. The highlighted line in the window indicates the current position of the program counter. The Step button at the bottom of the window can be used to advance execution of the system until the PPE completes one instruction. Double-clicking on a line will toggle a breakpoint on that line (indicated by the red B at the front of the line), and hovering over a register name in the disassembled instruction will display the contents of that register.

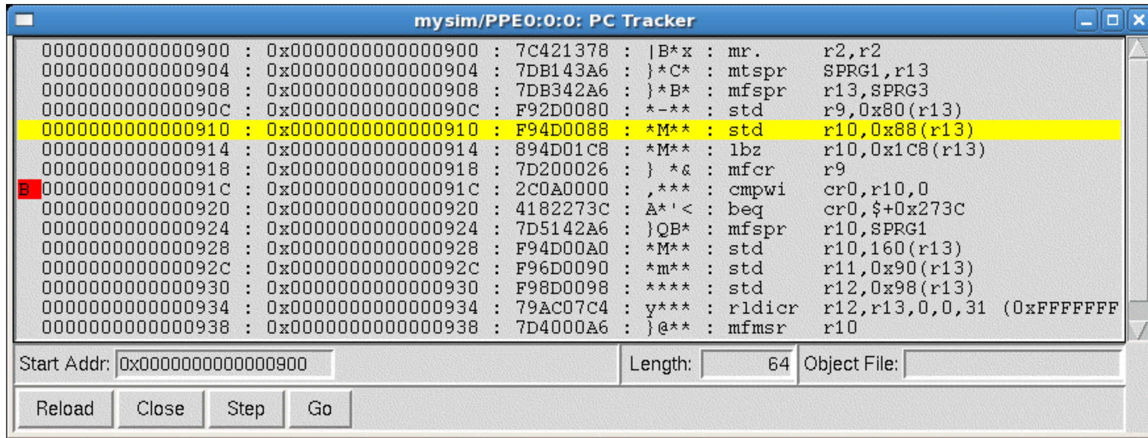


Figure 2-4. PPE PC Tracker Window

The PPE Core window (PPCCore) shows the contents of all the registers of the PPE, including the general purpose registers, floating point registers, Vector/SIMD Multimedia Extension registers, and special purpose registers. Figure 2-5 shows the PPE Core window.

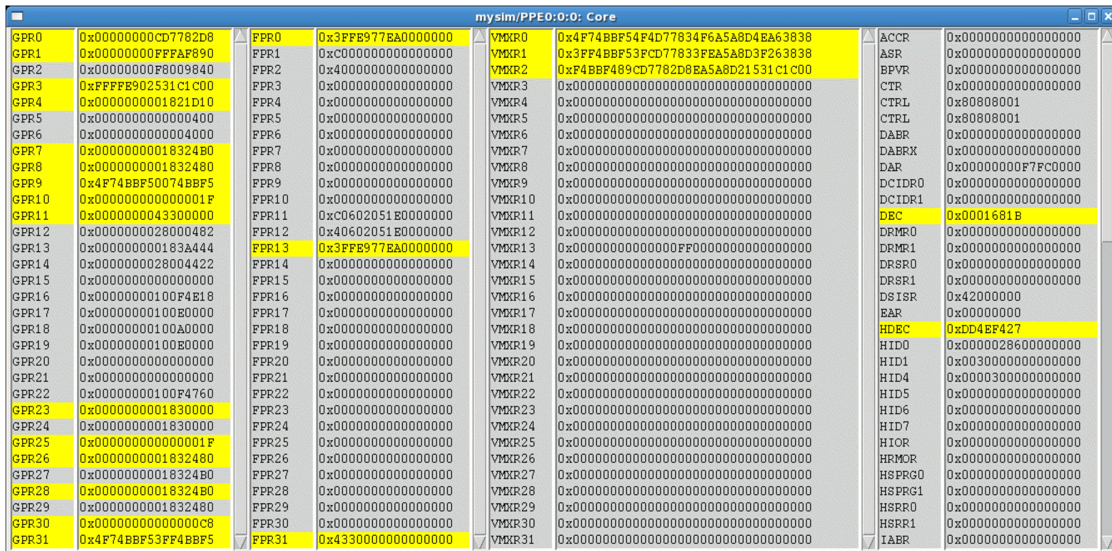


Figure 2-5. PPE Core Window

The PPE Stack dialog (PPCStack) shows a back trace of the program call stack of the application running on the selected PPE processor. Figure 2-6 shows the PPC Stack dialog for PPE 0:0:0. When the simulator can locate the object file that maps to this region of memory, it will display symbol and source file line information for the addresses in the call stack if these are available. Clicking on the address text within a trace entry will bring up the system memory dialog to display the contents of memory at this address.

© IBM Corporation. All rights reserved.

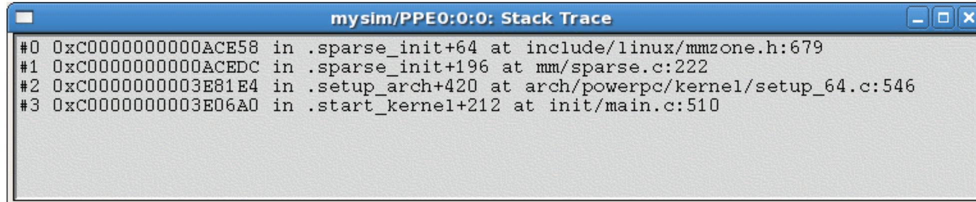


Figure 2-6. PPC Stack Trace Dialog Window

The general-purpose registers (GPRs) and the floating-point registers (FPRs) can be viewed separately by double-clicking on the GPRegs and the FPRegs folders respectively. Figure 2-7 shows the GPR window, and Figure 2-8 on page 13 shows the FPR window. As data changes in the simulated registers, the data in the windows is updated and registers that have changed state are highlighted.

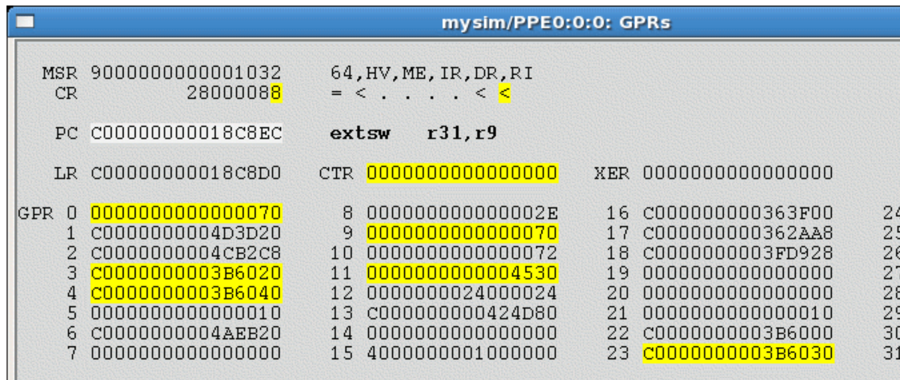


Figure 2-7. PPE General-Purpose Registers Window

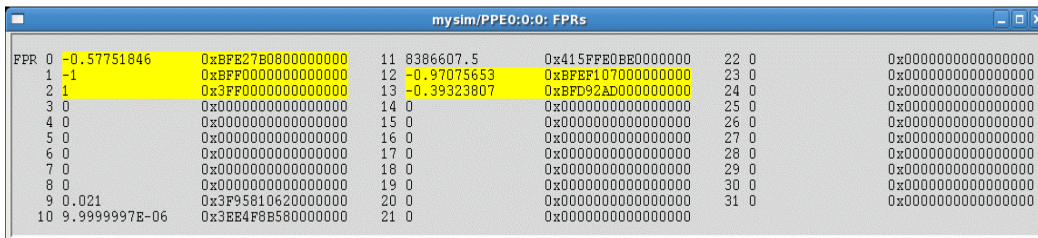


Figure 2-8. PPE Floating-Point Registers Window

The PPC Address Translation window (PPC_Xlate) shows the contents of the main address translation structures for the PPE, including the segment lookaside buffers (SLBs) and translation lookaside buffers (TLBs). Please refer to the *Cell Broadband Engine Programming Handbook, Chapter 4 Virtual Storage Environment* for information on the contents of these structures.

SPE Components

The SPE folders (*SPE0 ... SPE7*) each contain eleven elements. Four of the elements – SPUTrack, SPUCore, SPUChannel, and SPUMemory – present windows that show data in the registers, channels, and memory of the SPU. Two of the elements – MFC and MFC_XLate – present windows that show state information on the MFC. Two elements – SPUStats and LS_Stats – display dialogs containing statistics on the operation of the SPU core and local storage. The last three sub-items — Model, StackChecking, and Load-Exec — represent actions to perform on the SPE.

The SPU PC Tracker window is displayed when the user double-clicks on the SPUTrack folder icon for an SPE. An example of this window is shown in Figure 2-9. This window operates in a similar manner to the PPE PC Tracker

window described above. The window displays the region of the SPU's local storage containing the instructions that are currently being executed by the SPU. Each line of the window shows the address of a word in storage and its contents in hexadecimal, ASCII, and as a disassembled SPU instruction. The highlighted line in the window indicates the current position of the program counter. The Step button at the bottom of the window can be used to advance execution of the system until the SPU completes one instruction. Double-clicking on a line will toggle a breakpoint on that line (indicated by the red B at the front of the line). Like the PPE PC Tracker window, this dialog will display the contents of a register if the mouse is positioned over the register name in the disassembled instruction. In addition, hovering over a channel name will display the channel count and channel value, as shown in Figure 2-9.

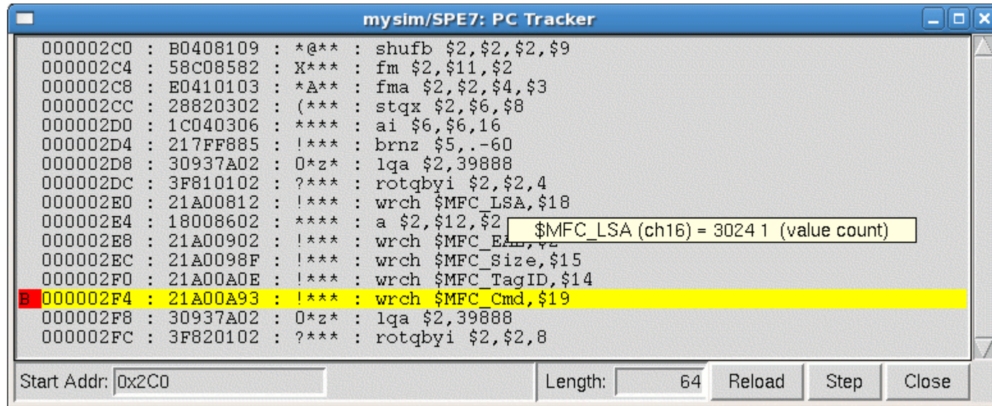


Figure 2-9. SPU PC Tracker Window

Figure 2-10 shows the SPU Core Window, which displays the contents of all the registers of the SPU in a pair of scrollable list boxes. Below each list box is a display that shows the contents of the selected register formatted as four 32-bit integers, four 32-bit single precision floats, and two 64-bit double precision floats. The window also displays the current value of the SPU's floating point status and control register (FPSCR) and status register.

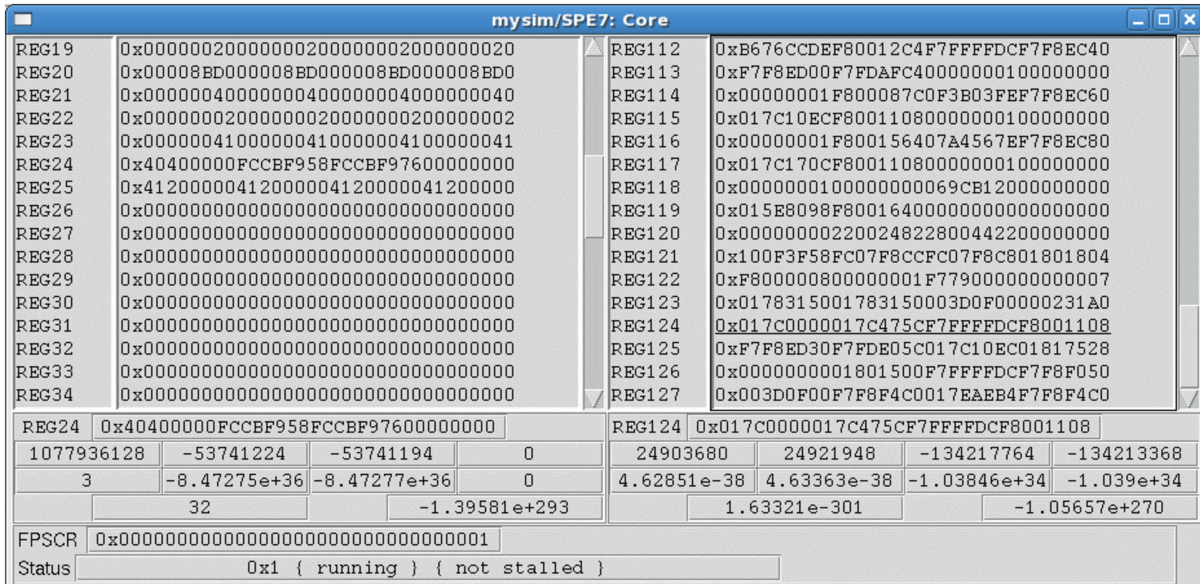


Figure 2-10. SPU Core Window

Figure 2-11 shows the SPU Channels window, which displays the channel contents and channel count for each of the SPU's channels. As in the simulated registers, the data in this window is updated as the simulation proceeds and

© IBM Corporation. All rights reserved.

values that have changed state are highlighted. The BP button next to each channel opens a dialog for setting a breakpoint on accesses to the channel, which can be conditional on the value read or written by the access.

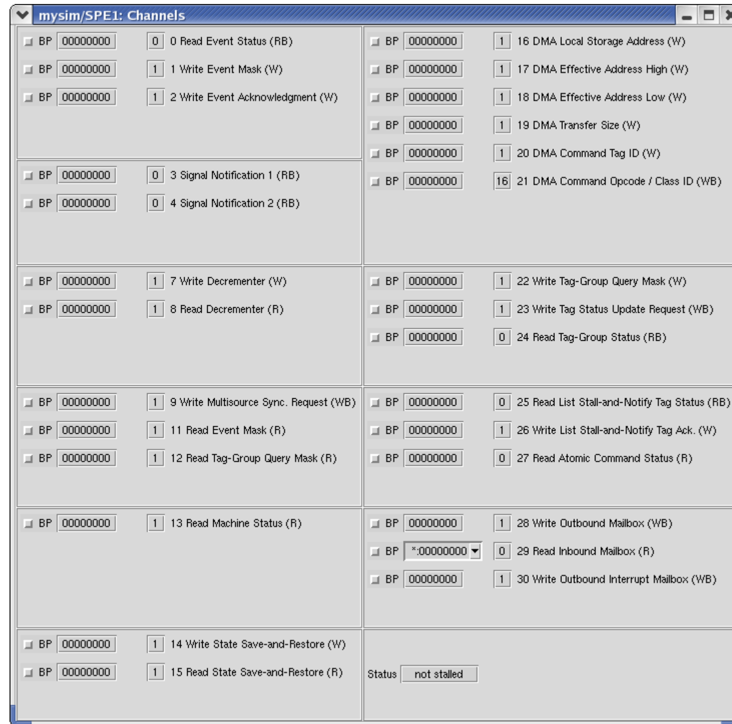


Figure 2-11. SPU Channels Window

The SPU Memory Window displays the contents of a region in the SPU's local memory. An example of this dialog window is shown in Figure 2-12. The starting address and size of the region are specified in the fields at the bottom of the dialog window. Memory contents can be displayed as hex and ASCII strings or as SPU instructions. The SPU Memory Window also allows users to set breakpoints on read or write accesses to a region of SPU memory.

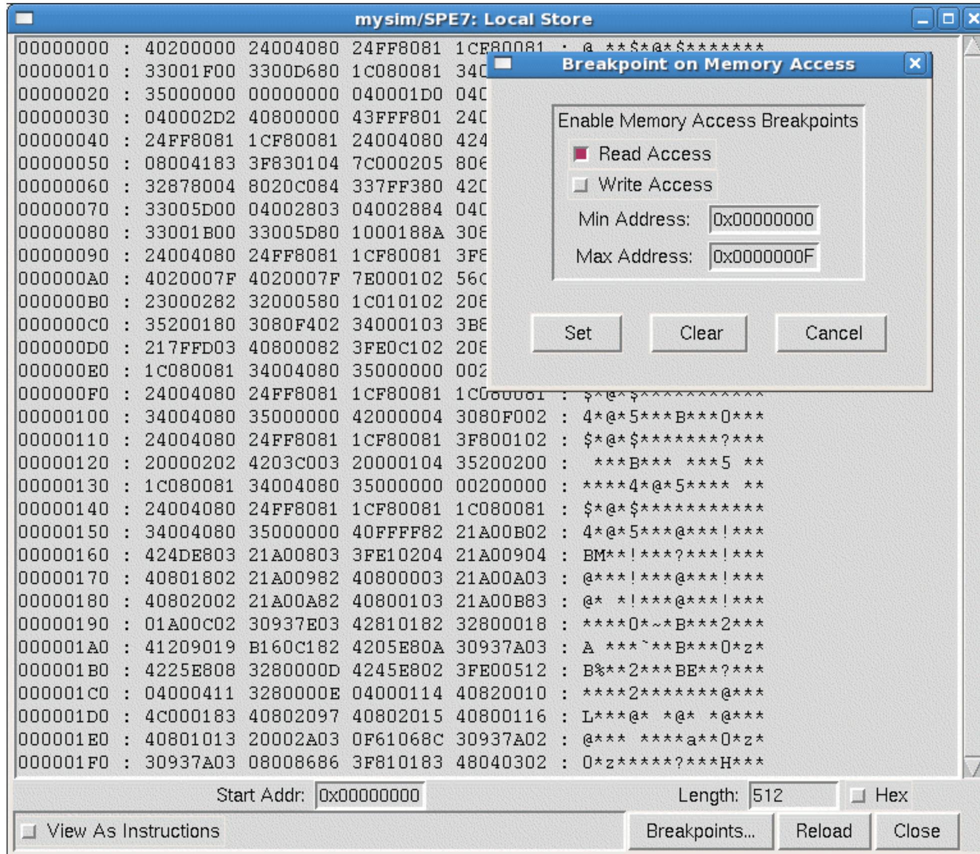


Figure 2-12. SPU Memory Window

Figure 2-13 shows the MFC window, which provides internal MFC state information. Figure 2-14 on page 17 shows the MFC_XLate window, which provides translation structure state information.

© IBM Corporation. All rights reserved.

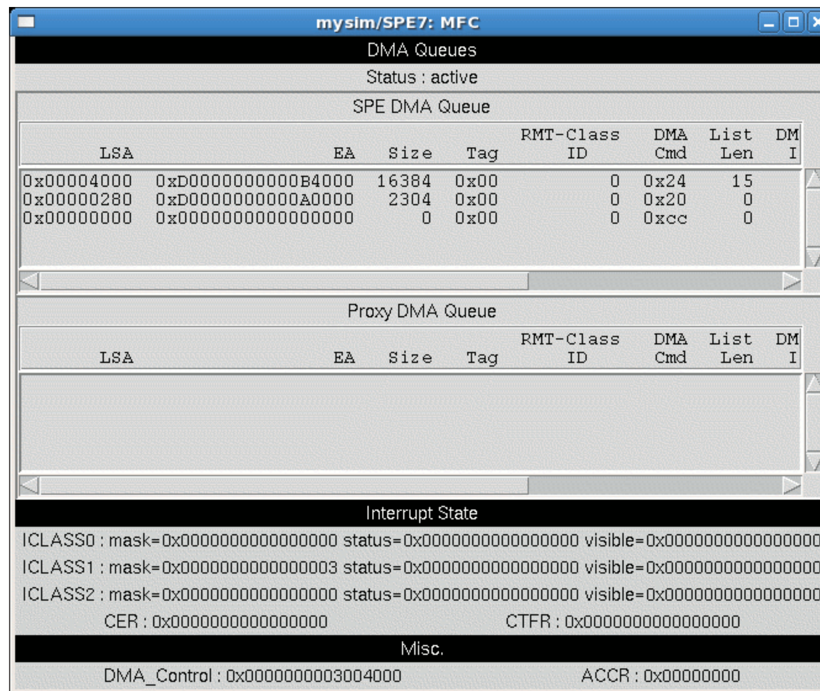


Figure 2-13. SPE MFC Window

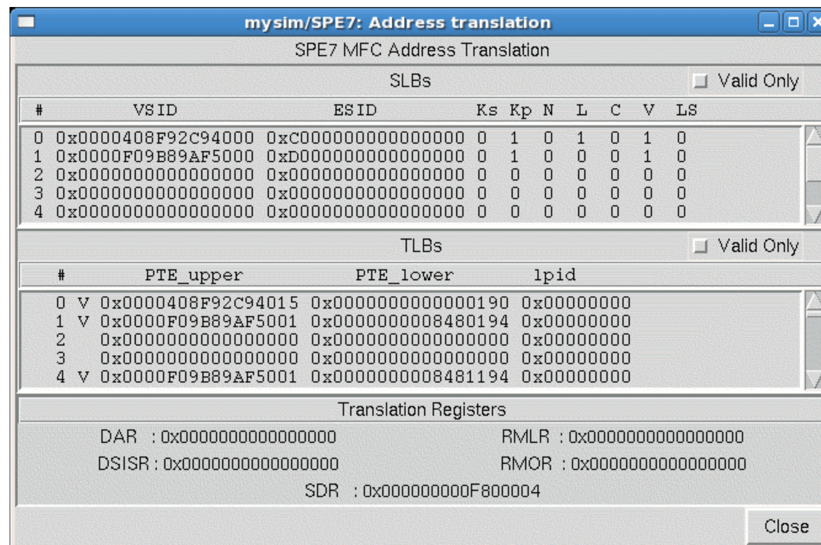
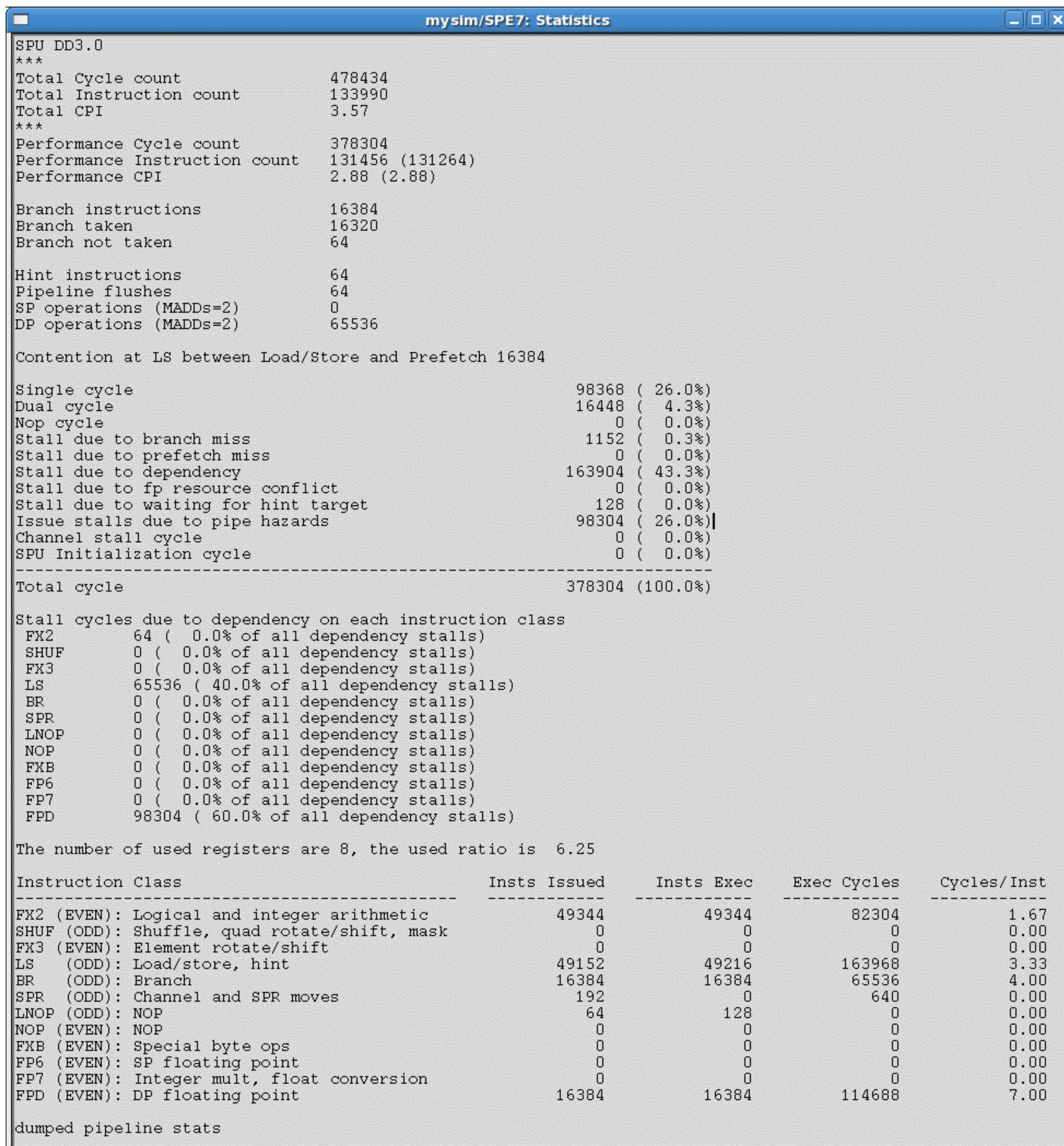


Figure 2-14. SPE MFC Address Translation Window

The next two SPE elements display statistics about the SPU core and local storage. Double-clicking on the SPUStats element will display the SPU Statistics window as shown in Figure 2-15. These statistics are only collected when the Model for the SPE is set to *pipeline*.



mysim/SPE7: Statistics

```
SPU DD3.0
***
Total Cycle count              478434
Total Instruction count        133990
Total CPI                      3.57
***
Performance Cycle count        378304
Performance Instruction count  131456 (131264)
Performance CPI                2.88 (2.88)

Branch instructions            16384
Branch taken                   16320
Branch not taken               64

Hint instructions              64
Pipeline flushes               64
SP operations (MADDs=2)        0
DP operations (MADDs=2)        65536

Contention at LS between Load/Store and Prefetch 16384

Single cycle                                              98368 ( 26.0%)
Dual cycle                                                16448 (  4.3%)
Nop cycle                                                     0 (  0.0%)
Stall due to branch miss                                   1152 (  0.3%)
Stall due to prefetch miss                                    0 (  0.0%)
Stall due to dependency                                  163904 ( 43.3%)
Stall due to fp resource conflict                             0 (  0.0%)
Stall due to waiting for hint target                        128 (  0.0%)
Issue stalls due to pipe hazards                          98304 ( 26.0%)
Channel stall cycle                                           0 (  0.0%)
SPU Initialization cycle                                      0 (  0.0%)
-----------------------------------------------------------------------
Total cycle                                              378304 (100.0%)

Stall cycles due to dependency on each instruction class
 FX2        64 (  0.0% of all dependency stalls)
 SHUF       0 (  0.0% of all dependency stalls)
 FX3        0 (  0.0% of all dependency stalls)
 LS         65536 ( 40.0% of all dependency stalls)
 BR         0 (  0.0% of all dependency stalls)
 SPR        0 (  0.0% of all dependency stalls)
 LNOP       0 (  0.0% of all dependency stalls)
 NOP        0 (  0.0% of all dependency stalls)
 FXB        0 (  0.0% of all dependency stalls)
 FP6        0 (  0.0% of all dependency stalls)
 FP7        0 (  0.0% of all dependency stalls)
 FPD        98304 ( 60.0% of all dependency stalls)

The number of used registers are 8, the used ratio is  6.25

Instruction Class                              Insts Issued    Insts Exec   Exec Cycles    Cycles/Inst
---------------------------------------------  ------------  ------------  ------------   ------------
FX2 (EVEN): Logical and integer arithmetic            49344         49344         82304           1.67
SHUF (ODD): Shuffle, quad rotate/shift, mask              0             0             0           0.00
FX3 (EVEN): Element rotate/shift                          0             0             0           0.00
LS   (ODD): Load/store, hint                          49152         49216        163968           3.33
BR   (ODD): Branch                                    16384         16384         65536           4.00
SPR  (ODD): Channel and SPR moves                       192             0           640           0.00
LNOP (ODD): NOP                                          64           128             0           0.00
NOP (EVEN): NOP                                           0             0             0           0.00
FXB (EVEN): Special byte ops                              0             0             0           0.00
FP6 (EVEN): SP floating point                             0             0             0           0.00
FP7 (EVEN): Integer mult, float conversion                0             0             0           0.00
FPD (EVEN): DP floating point                         16384         16384        114688           7.00

dumped pipeline stats
```

Figure 2-15. SPU Statistics

Figure 2-16 shows the LS_Stats window, which brings up the new local store display map. This dialog presents accesses to the SPEs local storage in a graphical manner, where access types are color-coded as shown in the legend and displayed for a range of addresses in the local store arranged across the x-axis of the plot. A trace of the accesses shown in the plot is also displayed in text form in the list boxes on the lower part of the window.

© IBM Corporation. All rights reserved.

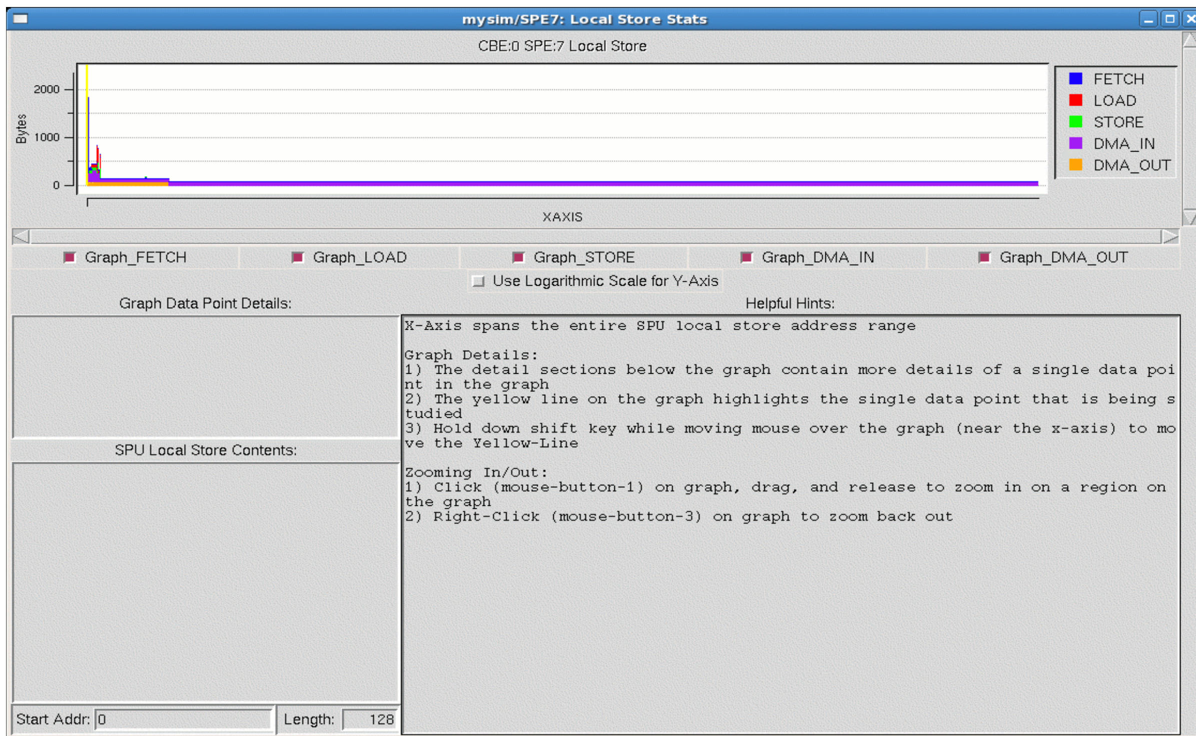


Figure 2-16. SPE Local Store Statistics Window

The last three items in an SPE folder represent actions to perform, with respect to the associated SPE. The first of these action elements is labelled either *Model: instruction, Model: pipeline, or Model: Fast*. The label indicates whether the simulation for this SPU is in *instruction mode*, for checking and debugging the functionality of a program, or *pipeline mode*, for collecting performance statistics on the program, or *fast mode* for quickly advancing the execution of a program to a region of interest. The mode can be toggled by double-clicking the item. The SPU Modes button in the button array of the main GUI window can also be used as a more efficient way to set the modes of all of the SPEs simultaneously.

The next item is labelled StackChecking:off or StackChecking:on. Double-clicking on this item will enable or disable special checks for application stack overflow. This feature checks the "available space" element of the stack pointer register (R1), as defined in the SPU Application Binary Interface specification, to detect stack overflow. The check is performed on every write access to local storage and thus can significantly impact simulation performance.

The last item in the SPE folder, *Load-Exec*, is used for loading an executable onto an SPE to be executed in standalone mode. When you double-click the item, a file-browsing window is displayed, allowing you to find and select the executable file to load.

Simulator and BE Components

There are additional simulator- and BE-specific components that are available from these levels in the simulation panel. These features include:

- At the BE level, the *Load-Elf-App* and *Load-Elf-Kernel* folders are available for loading an ELF binary or ELF kernel for execution on the BE. Double-clicking these items opens a file-browsing window from which you can find and select an application or kernel file to load.
- At the simulator level, the *MemoryMap* and *SystemMemory* components display information about system memory. The *MemoryMap* window displays the name of each region with the corresponding start and end memory addresses for each region. The *SystemMemory* window displays a region of memory, specified using

either a physical or effective address. Memory can be displayed as data in hex and ASCII strings or as PowerPC instructions. *SystemMemory* also allows users to set breakpoints on read or write accesses to a region of memory. Figure 2-17 and Figure 2-18 illustrate sample windows for MemoryMap and SystemMemory.

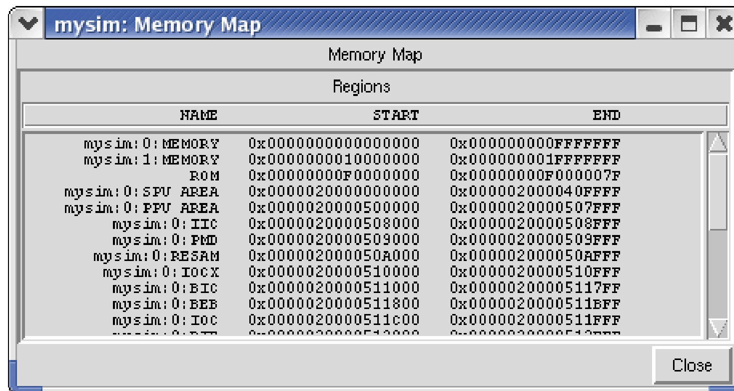


Figure 2-17. MemoryMap Window

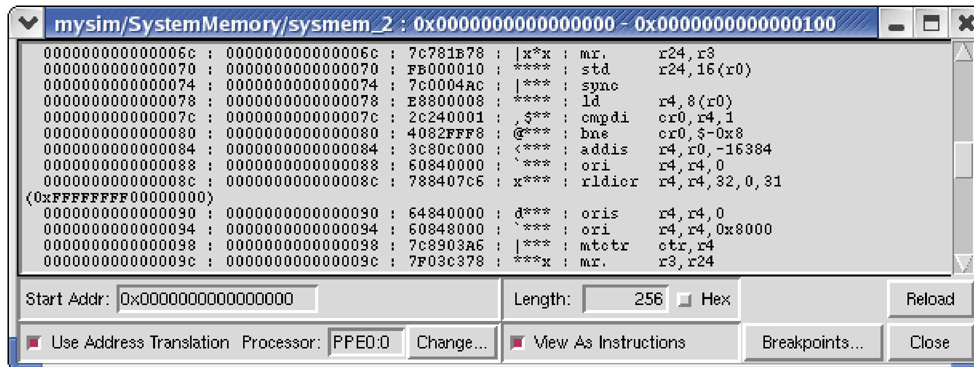


Figure 2-18. SystemMemory Window

GUI Buttons

On the right side of the GUI screen (Figure 2-1 on page 10) are five rows of buttons. These are used to manipulate the simulation process. The buttons do the following:

- *Advance Cycle*–Advances the simulation by a set number of cycles. The default value is 1 cycle, but it can be changed by entering an integer value in the textbox above the buttons, or by moving the slider next to the textbox. The drop-down menu at the top of the GUI allows the user to select the time domain for cycle stepping. The time units to use for cycles are expressed in terms of various system components. The simulation must be stopped for this button to work; if the simulation is not stopped, the button is inactive.
- *Go*–Starts or continues the simulation. In the SDK's simulator, the first time the Go button is clicked it initiates the Linux boot process. (In general, the action of the Go button is determined by the startup *tcl* file located in the directory from which the simulator is started.)
- *Stop*–Pauses the simulation.
- *Service GDB*–Allows the external gdb debugger to attach to the running program. This button is also inactive while the simulation is running.
- *Triggers/Breakpoints*–Displays a window showing the current triggers and breakpoints.
- *Update GUI*–Refreshes all of the GUI screens. By default, the GUI screens are updated automatically every four seconds. Click this button to force an update.

© IBM Corporation. All rights reserved.

- *Debug Controls*—Displays a window of the available debug controls and allows you to select which ones should be active.
- *Options*—Displays a window containing a Display tab from which you can set fonts for the GUI display, and a Simulator tab from which you can set the gdb debugger port.
- *Emitters*—Displays a window with the defined emitters, with separate tabs for writers and readers.
- *Mode*—Displays a window containing a message indicating the current simulation mode and three buttons which will change the simulation mode to fast, simple, or cycle mode. Figure 2-19 shows an example of this dialog window. This dialog provides a convenient way to set the simulation mode for components of the system in a consistent manner. The simulation mode can also be selected with the "mysim mode" command and displayed with the "mysim display mode" command.
- *SPU Modes*—Provides a convenient means to set each SPU's simulation mode to instruction mode, pipeline (cycle accurate) mode or fast mode. The same capabilities are available using the Model toggle menu sub-item under each SPE in the tree menu at the left of the main control panel. Figure 2-20 shows the SPU Modes window.
- *SPE Visualization*—Plots histograms of SPU and DMA event counts. The counts are sampled at user defined intervals, and are continuously displayed. Two modes of display are provided: a "scroll" view, which tracks only the most recent time segment, and a "compress" view, which accumulates samples to provide an overview of the event counts during the time elapsed. Users can view collected data in either detail or summary panels. The detailed, single-SPE panel tracks SPU pipeline phenomena (such as stalls, instructions executed by type, and issue events), and DMA transaction counts by type (gets, puts, atomics, and so forth). The summary panel tracks all eight SPEs for the Cell Broadband Engine, with each plot showing a subset of the detailed event count data available. Figure 2-21 on page 22 shows the SPE Visualization window.
- *Track All PCs*—Provides a limited version of the PC tracker that shows the current execution location of all SPUs and PPUs for each MCM in the system. If SMP mode is enabled, the Track All PCs window displays SPU and PPU state for both MCMs. Figure 2-22 shows the Track All PCs window.
- *Event Log*—Enables a set of pre-defines triggers to start collecting the log information. The window provides a set of buttons that can be used to set the marker cycle to a point the the process.
- *Exit*—Exits the simulator and closes the GUI window.

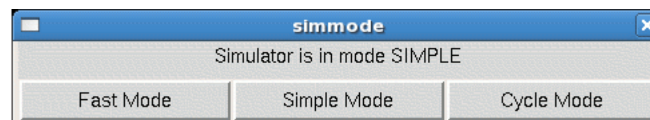


Figure 2-19. Simulation Mode Window

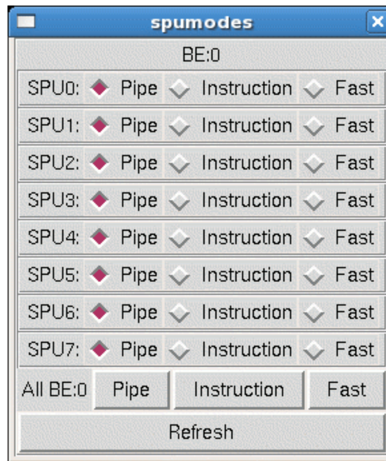


Figure 2-20. SPU Modes Windows

Figure 2-21. SPE Visualization Window

 © IBM Corporation. All rights reserved.

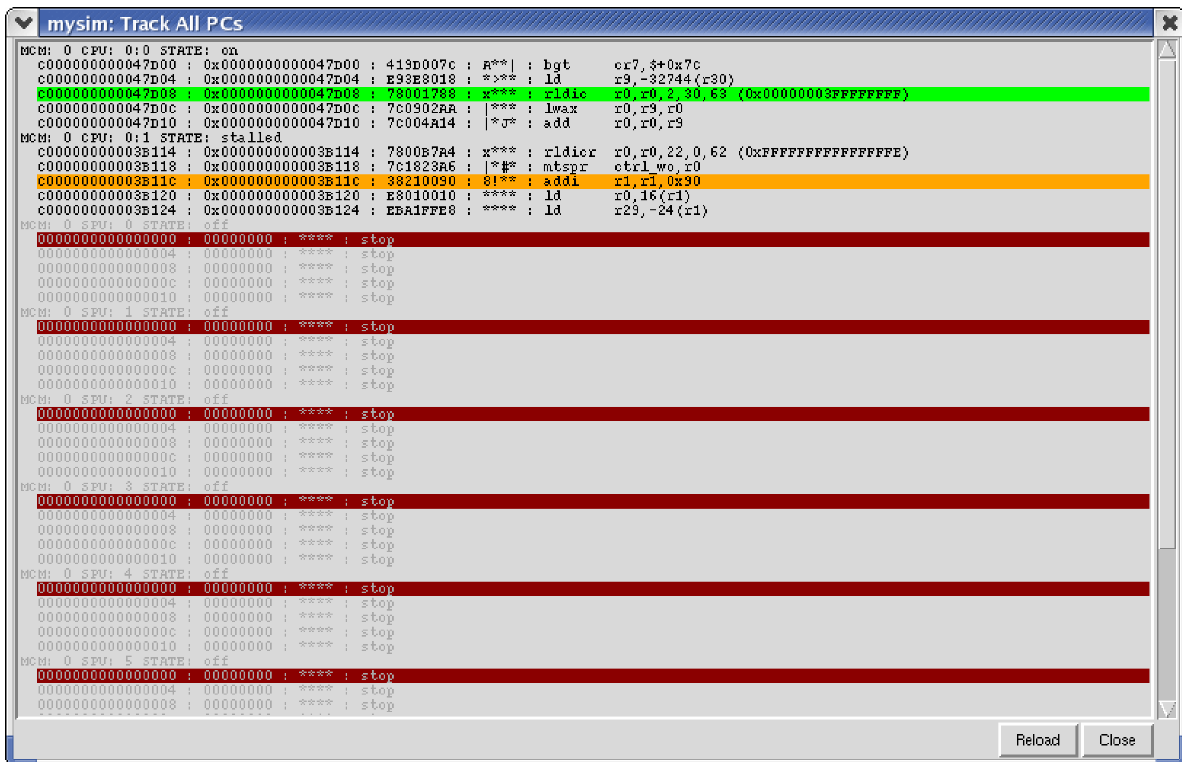


Figure 2-22. Track All PCs Window

© IBM Corporation. All rights reserved.

CHAPTER 3

SystemSim Command Syntax and Usage

This chapter describes the IBM Full-System Simulator command framework, and introduces the structure, format, and usage of simulator commands. Topics in this chapter include:

- *Understanding and Using Simulator Commands*
- *Defining and Managing a Simulated Machine*
- *Summary of Top-Level Simulator Commands*

© International Business Machines Corporation. All rights reserved.

Understanding and Using Simulator Commands

The IBM Full-System Simulator provides a unified, cross-platform application programming interface that enables users to easily set up the simulation environment, manage simulated architecture components, and write debugging and performance analysis routines. The IBM Full-System Simulator has harnessed the power of Tcl/Tk to develop a simple and programmable text-oriented syntax that is easily extended and minimizes the need for proprietary and difficult programming grammar and usage. By extending Tcl with exported functions, data types, and numerous predefined interfaces that are used for all interobject communication, the simulator provides a rapid, cross-platform development environment that enables users to quickly start working in the simulation environment.

The IBM Full-System Simulator command framework provides an extensive set of commands for modeling, simulating, and tuning microprocessor components in a system. Each component in a microprocessor system is configured via commands that not only define the component's run-time behavior and characteristics, but govern its relationships and interactions with surrounding components in the system. The *SystemSim Command Reference* provides syntax and usage information for Tcl/Tk commands that are used in the simulator environment.

Commands in the IBM Full-System Simulator are organized into a hierarchy of operations based on the command function. At the top level, commands perform general sets of operations in the simulation environment, such as defining and displaying machine properties and system configurations, modifying configurable parameters, performing 64-bit arithmetic operations that are not provided by default in Tcl, or managing the simulation environment and its data collection and analysis tools.

In addition to configuring system components, the simulator commands can be combined with programming logic and Tcl programming constructs to gather, analyze, and visualize simulation events, run workloads on the modeled microarchitecture, and generate performance metrics with new or revised configurations to forecast performance at future workloads. The command line interface also can be used to perform a number of operations on the simulator itself, such as to control a simulation, start data collection and visualization tools, and define and load virtual devices and disk images.

© IBM Corporation. All rights reserved.

Figure 3-1 illustrates how commands are processed in the simulation environment and describes the different categories of commands that are available:

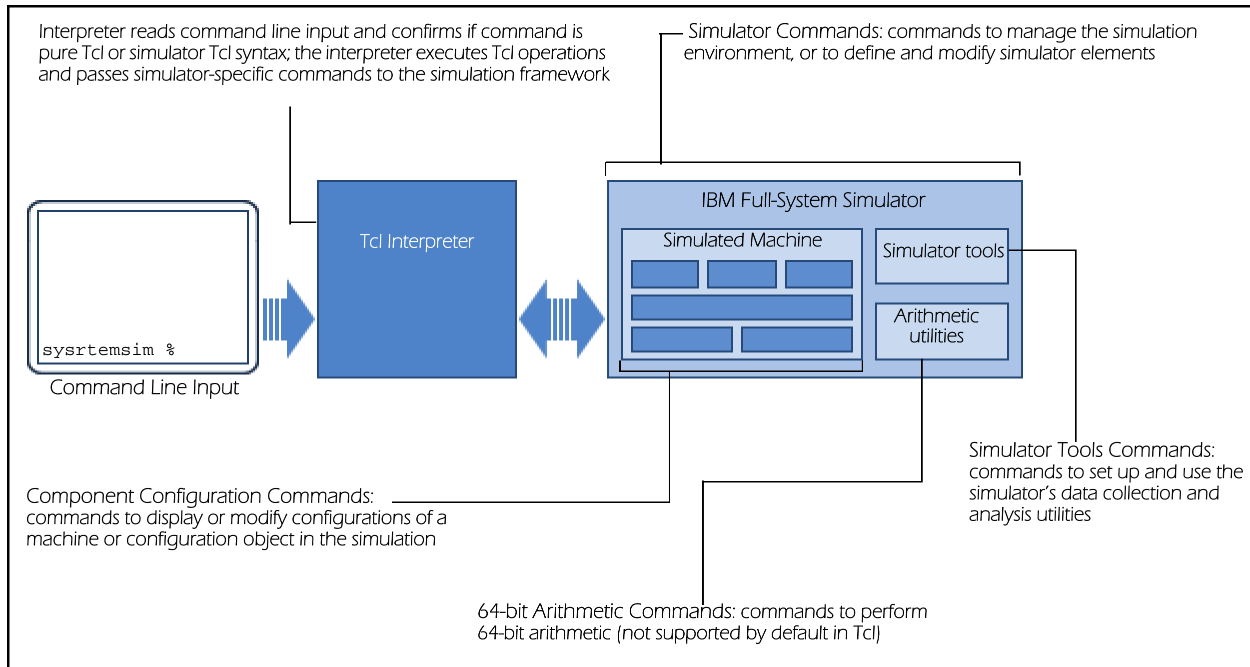


Figure 3-1. Categories of Simulator Commands

Once the simulator is started, commands may be entered at the simulator command line or via simulation Tcl scripts. Figure 3-2 illustrates the simulator command line at start-up and the simulated Linux console that is launched from the simulator command line with the **mysim go** command:

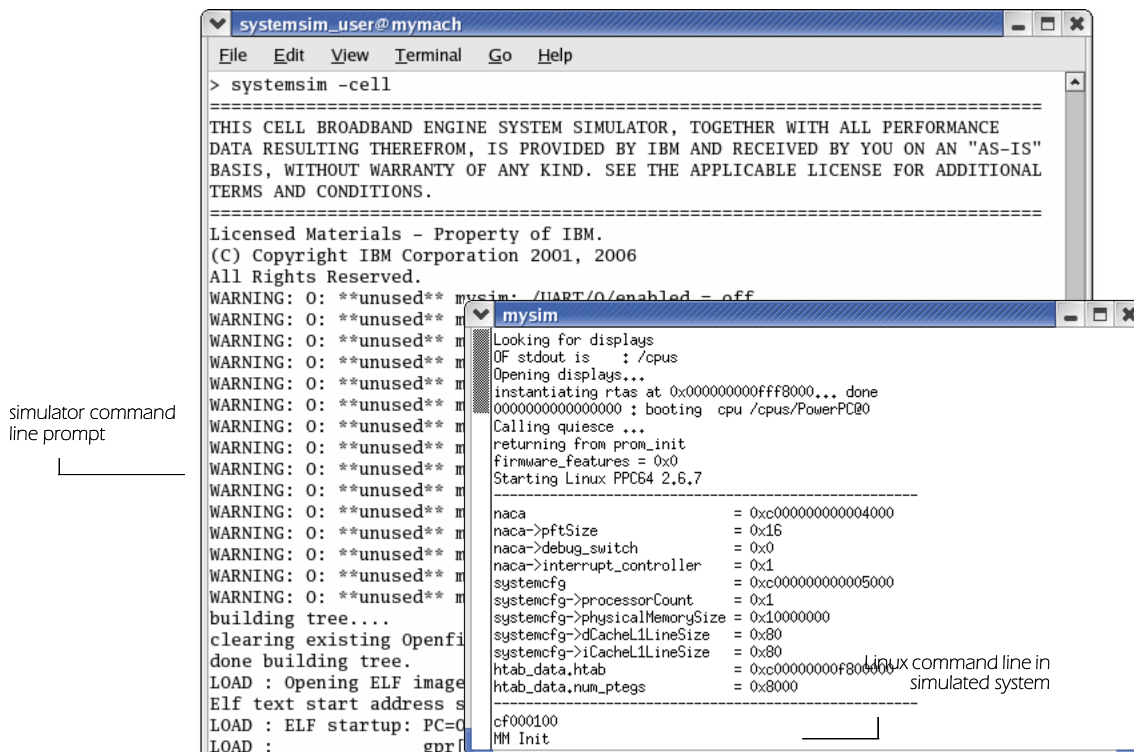


Figure 3-2. IBM Full-System Simulator Command Line

Defining and Managing a Simulated Machine

Each version of the IBM Full-System Simulator delivers a set of default configurations for the type of PowerPC processor it is modeling. Using these configurations, users can instantiate a simulated machine based on a default *configuration object* to examine functionality and performance of a baseline system. Alternatively, users can create a configuration object of the pre-defined machine, replace one or more default settings, and instantiate a custom machine to evaluate how an individual component or the entire processor architecture performs under customized conditions.

Figure 3-3 describes the general sequence of commands that are used to define a machine in the simulation environment:

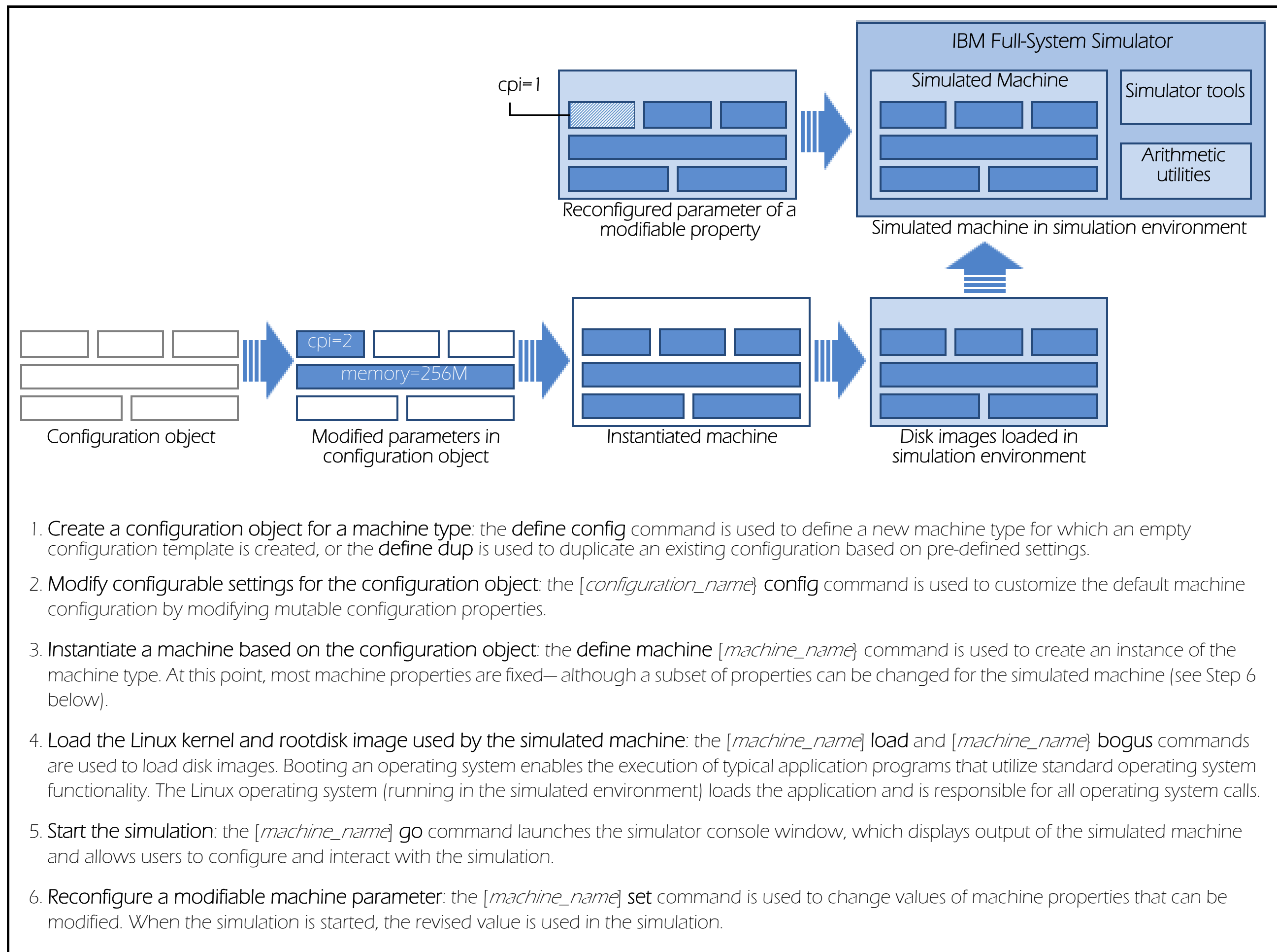


Figure 3-3. Defining, Creating, and Starting a Simulated Machine

Commands to configure and initialize a simulated machine are typically provided to the simulator with a Tcl configuration and start-up file called an initial run script that is loaded when the simulator starts. The initial run script specifies commands to create a machine configuration and machine instance using this configuration, locate and load the operating system and file system image files, and prepare the machine to begin execution. The name of the initial run script can be passed to **systemsim** with the **-f** option. If there is no initial run script specified when the simulator is started, the simulator will use either a user-defined initial run script, which by convention is named **.systemsim.tcl** and resides in the current directory, or the default initial run script **lib/cell/systemsim.tcl** provided with the systemsim-cell release.

.

 © IBM Corporation. All rights reserved.

A typical initial run script

The specific commands in an initial run script vary slightly for various machine configurations, but all follow the basic procedure described above. The following describes a command sequence that may be found in a typical initial run script.

1. Create a configuration object called myconf and initialize it with the configuration named cell. The cell configuration is a fixed configuration provided in the IBM Full-System Simulator as a baseline Cell BEA machine configuration:

   ```
   define dup cell myconf
   ```

2. At this point, other configuration settings can be specified or changed. An example of a configuration change that can be made here is to specify that the configuration should employ two Cell Broadband Engine processors, referred to as an SMP or Dual BE configuration. Since this requires changes to a number of configuration parameters, a special procedure, config_dual_be, has been provided to specify an SMP configuration. This procedure takes as a parameter the name of the configuration to be modified, for example:

   ```
   config_dual_be myconf
   ```

3. Create a simulator named mysim for a machine with the configuration myconf:

   ```
   define machine myconf mysim.
   ```

4. Load the operating system kernel into memory:

   ```
   mysim load vmlinux <path_to_vmlinux_file> 0x1000000
   ```

   In hardware this would generally be performed by the system firmware, but the simulator is typically configured without firmware installed and thus a simulator command is used to load the kernel into memory. In many cases, the initial run script uses a standard search order to locate the vmlinux file, starting with the current directory and then the images directory under the simulator root directory. The 0x1000000 parameter in this command specifies the address at which to load the kernel file.

5. Specify a file containing the root filesystem (sysroot) image:

   ```
   mysim bogus disk init 0 <path_to_sysroot_disk_file> newcow <cowfile> 1024
   ```

   As with the kernel image, the initial run script typically uses a standard search order to locate the sysroot_disk file starting with the current directory. The `newcow` parameter indicates that the disk image should be accessed copy-on-write, with changes stored in `<cowfile>`. This method treats the contents of the `sysroot_disk` file as read-only, so that subsequent simulations can be performed with repeatable results. Specifying an access type of `rw` (for read-write) instead of `newcow` indicates that modifications to the root filesystem during the simulation should be stored back into the `sysroot_disk` file. When the `sysroot_disk` is accessed read-write, the user should issue the `sync` command before exiting the simulator to ensure consistency of the filesystem image.

6. Start the simulation with the mysim go command:

   ```
   mysim go
   ```

   This will start the boot of the Linux operating system.

7. Once the operating system has completed boot, you can execute applications that run on the PPE, SPU, or both, by entering commands at the Linux console. To automate console input, use the mysim console create command. This command automates the interactions that are typically performed by manually typing commands in the simulator console window:

   ```
   mysim console create input in string <console_input>
   ```

where *console_input* specifies a string containing console commands to execute. The string contents are identical to any commands that are typed in the console window, including new lines (which can be entered with the escape sequence \n). Typically, the last command of the console input is callthru exit to return control to the simulation Tcl command script.

Summary of Top-Level Simulator Commands

Table 3-1 summarizes functionality of selected top-level commands that are used to define, modify, and use the simulator. The IBM Full-System Simulator Command Reference provides the complete command line syntax and usage of each command or class of commands.

Table 3-1. IBM Full-System Simulator Top-Level Commands

Command	Command Summary
alias	Assigns a user-specified personal shorthand for a command string. The alias command allows users to call a small, more familiar command or name to execute long or complex command strings.
define	Defines settings for a configuration object. The define command also provides that ability to duplicate configurations from a pre-defined machine type, instantiate a machine based on a configuration object, and enumerate a list of machines that are active in the simulation.
display	Displays system-wide information about configurations, machines, instruction settings, and warning levels. The display command is especially useful to determine properties that are configured for machines that are currently available in a simulation.
ereader	Controls emitter readers that are used in a simulation for performance data collection and measurement.
help or helprecursive	Displays a listing the IBM Full-System Simulator commands. The helprecursive command displays a comprehensive command tree that hierarchically lists syntax and input parameters for all available commands.
modify	Modifies configurable simulation settings or parameters. The modify command is useful for changing various run-time paramters, such as the warning level that is set for the simulation environment.
object	Provides the ability to interrogate information from one or more executable files to examine low-level execution details.
quit	Ends the current simulation and exits to the operating system command line.
simdebug	Provides low-level tracing capabilities that are useful for debugging functionality or performance issues in the simulated system.
simemit	Specifies event types to be written to the shared memory buffer that is recording emitted data.
simstop	Stops the simulation and waits for instruction at the simulator command line. The simstop command performs the same operation as typing CTRL+C to interrupt the simulation.
version	Displays the version number of simulation system components, the date and timestamp of the installed IBM Full-System Simulator build, and compile-time flags that are enabled in the build.

© IBM Corporation. All rights reserved.

Table 3-1. IBM Full-System Simulator Top-Level Commands

Command	Command Summary
64-bit arithmetic operations	By default, arithmetic operations in the Tcl scripting language do not support 64-bit arithmetic. The IBM Full-System Simulator provides the following operations to perform general calculations on 64-bit numbers: ■ add64 ■ and64 ■ compare64 ■ div64 ■ format64 ■ incr64 ■ invert64 ■ lessthan_u64 ■ lshift64 ■ mul64 ■ or64 ■ percent64 ■ rshift64 ■ sub64 ■ uint32_to_float

At any time, users can type the help command at the command line to retrieve a list of command choices that are available from that point in the syntax statement. In most cases, you can also just type a partial command sequence and hit return. For example, at the top level, help displays a of top-level commands. An arrow indicates that a subsequent level of command functionality is available for this command.

© IBM Corporation. All rights reserved.

CHAPTER 4

Debugging Features in SystemSim

The simulator has a vast array of debug facilities. This chapter describes some of the debugging features that are specifically designed for SPU debugging. Topics in this chapter include

- *Detecting SPU Stack Overflow*
- *Bus errors caused by DMA errors*
- *Kernel debugging*

© International Business Machines Corporation. All rights reserved.

Detecting SPU Stack Overflow

The SPU Local Store has no memory protection, and memory access wraps from the end of Local Store back to the beginning. An SPU program is free to write anywhere in Local Store including its own instruction space. A common problem in SPU programming is the corruption of the SPU program text or dynamically allocated data when the program's stack area overflows into the heap or program area. This problem typically does not become apparent until some later point in the program execution. If the overflow corrupted the program text, this typically results in an illegal instruction exception when the program attempts to execute code in area that was corrupted. For corruption of heap storage, the program might generate incorrect results or fail is some other manner. Even with a debugger it can be difficult to track down this type of problem because the cause and effect can occur far apart in the program execution. Adding printf's just moves the failure point around.

The simulator has a feature that checks for a stack overflow condition during the execution of an SPU program. This feature checks the "available space" element of the stack pointer register (R1), as defined in the SPU Application Binary Interface specification, for a negative value, which indicates stack overflow. The check is performed on every write access to local storage and thus can significantly impact simulation performance.

Two methods are available to enable SPU stack checking. The first method is accessed through the SPE folder in the tree view of the Graphical User Interface. Each SPE folder contains a StackChecking element that will toggle the status of stack checking for that SPE. SPU stack checking can also be controlled using Tcl procedures provided with the simulator. To enable stack checking for a specified SPU, use:

```
spu_stack_check::enable [spu_number]
```

Stack checking can be disabled with the Tcl statement

```
spu_stack_check::disable [spu_number]
```

When the simulator detects SPU stack overflow, it halts the simulation and displays a message indicating that overflow has occurred. Figure 4-1 shows the simulator console window and command window from a simulator run that employed enable_stack_checking to detect a stack overflow in a Cell/B.E. application. Note: Another approach for detecting stack overflows is to use the stack checking options provided by the compiler. The -fstack-check compile flag results in the insertion of runtime tests which will detect stack overflow by checking the amount of stack space available whenever the stack pointer is modified. The program halts in the event of overflow.

© IBM Corporation. All rights reserved.

```
root@(none):~
NET: Registered protocol family 2
IP route cache hash table entries: 2048 (order: 2, 16384 bytes)
TCP established hash table entries: 8192 (order: 5, 131072 bytes)
TCP bind hash table entries: 8192 (order: 5, 131072 bytes)
TCP: Hash tables configured (established 8192 bind 8192)
TCP reno registered
TCP bic registered
NET: Registered protocol family 1
Initializing disk 0 with devsz 1638400
VFS: Mounted root (ext2 filesystem).
Freeing unused kernel memory: 164k freed
INIT: version 2.86 booting
touch: setting times of `/etc/fstab': Function not implemented
                Welcome to Fedora Core
                Press 'I' to enter interactive startup.
INIT: Entering runlevel: 2
[root@(none) ~]# callthru source ppu/hpl_sp > hpl_sp
[root@(none) ~]# chmod +x hpl_sp
[root@(none) ~]# ./hpl_sp 256 1
Linpack(S.P.) 256 x 256, num_spes = 1, block_size=64
WARNING: unable to open file /huge/linpack_mem.bin (errno=2). Using malloc heap.
Matrix initialization complete ... starting SPEs
SPE thread invoked : speid[0] = 0x1834940
```

```
root@(none):~
File  Edit  View  Terminal  Tabs  Help
250981169: (65247894): TCP bind hash table entries: 8192 (order: 5, 131072 bytes)
251110731: (65377179): TCP: Hash tables configured (established 8192 bind 8192)
251114244: (65380686): TCP reno registered
251134898: (65401153): TCP bic registered
251149778: (65415394): NET: Registered protocol family 1
251904951: (66168439): Initializing disk 0 with devsz 1638400
252137826: (66389241): VFS: Mounted root (ext2 filesystem).
252335429: (66586481): Freeing unused kernel memory: 164k freed
259520582: (73643333): INIT: version 2.86 booting
269148178: (83083965): touch: setting times of `/etc/fstab': Function not implemented
274678278: (88489496):          Welcome to Fedora Core
279830766: (93561742):          Press 'I' to enter interactive startup.
396429817: (207639986): INIT: Entering runlevel: 2
450509924: (260745816): [root@(none) ~]# callthru source ppu/hpl_sp > hpl_sp
75318419257: (293185171): [root@(none) ~]# chmod +x hpl_sp
86478974975: (298463165): [root@(none) ~]# WARNING: 112022307222: Caught INTERRUPT Signal. Stopping S
imulation
112022307222: ** Execution stopped: user interrupt,  **
112022307222: ** finished running 309573161 instructions **

systemsim % enable_stack_checking 7 spu/solve
apu->breakpoint 16908
prog_start 7 0x00016908 0x0001cba8 0x0003db50 0x24004080 0x24FFC0D0 0x24FF80D1 0x24FF40D2
systemsim % mysim go
./hpl_sp 256 1
176080414605: (339744020): Linpack(S.P.) 256 x 256, num_spes = 1, block_size=64
176080430761: (339759603): WARNING: unable to open file /huge/linpack_mem.bin (errno=2). Using malloc
 heap.
176090911445: (349189741): Matrix initialization complete ... starting SPEs
176093406892: (351672347): SPE thread invoked : speid[0] = 0x1834940
new break count == 0x0
apu->breakpoint 1cba8
Stack overflow detetection enabled on SPU 7
SPU7: CP0, 0(0), 0
SPU7: CP30, 0(0), 0
STACK OVERFLOW DETECTED ON SPU 7!
176094700357: ** Execution stopped: user (tcl),  **
176094700357: ** finished running 352963886 instructions **
systemsim %
```

Figure 4-1. Example of SystemSim's stack overflow detection facility

Bus errors caused by DMA errors

The simulator also provides assistance with identifying the cause of bus errors that occur because of an error in processing a DMA request issued by an SPU. Common errors encountered in DMA processing are alignment errors and attempts to access unallocated or protected areas of system memory. The alignment rules for DMAs specify that transfers for less than 16 bytes must be "naturally aligned," meaning that the address must be divisible by the size of the transfer. Transfers of 16 bytes or more must be 16-byte aligned. The size can have a value of 1, 2, 4, 8, 16, or a

multiple of 16 bytes to a maximum of 16KB. In addition, the low-order four bits of the Local Store address must match the low-order four bits of effective address (in other words, they must have the same alignment within a quadword). Any DMA that violates one of these rules will generate an alignment exception which is presented to the user as a bus error. System memory addresses specified in DMA commands undergo address translation to ensure that the storage area is allocated an accessible in the manner requested (read or write) by the DMA request. Errors in address translation during DMA processing are also converted into bus errors.

The simulator checks the DMA alignment requirements and raises alignment exceptions as necessary to match the behavior of the hardware. But in addition to this, the simulator also generates warning messages to aid the programmer in finding and correcting these problems. Figure 4-2 illustrates a warning message, "WARNING: 441391050: GET command with illegal size (12) (< 16 and not 0, 1, 2, 4, or 8)," (highlighted in red in the figure) issued by the simulator for a DMA alignment exception.

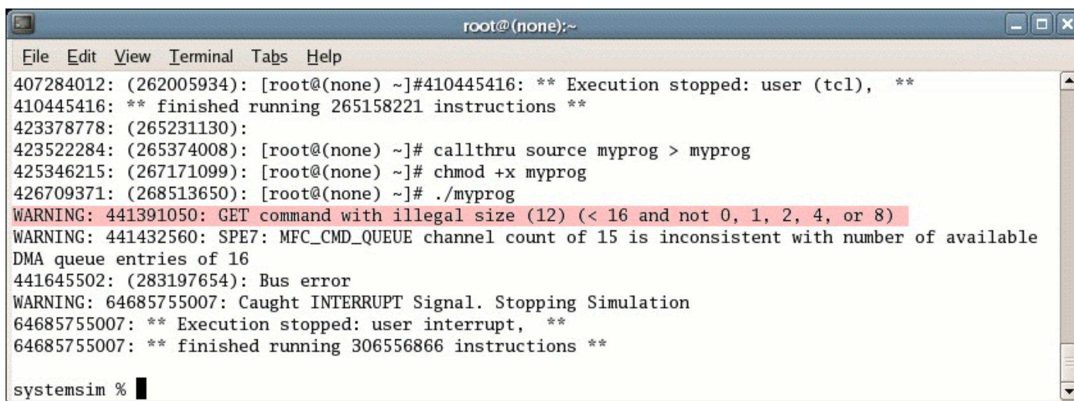


Figure 4-2. Warning message from simulator for DMA alignment exception

Kernel debugging

Debugging the Linux kernel can be a difficult task, in part because the kernel is a complex piece of software, but also because the debugger cannot rely on basic OS functions being available or working properly. On the Cell BE SDK, kernel debugging is simplified because the IBM Full-System Simulator, part of the Cell BE SDK, allows a debugger running on the host system to debug a Linux kernel running inside the simulator.

To exploit this feature, you must have a version of GDB that supports the 64-bit PowerPC® architecture. On 64-bit PowerPC host systems, this version of GDB might be available as part of the standard OS installation. Otherwise, download and build a version of GDB with the appropriate architecture support. (Note: We currently recommend using GDB version 6.3 to interface with the simulator. Later versions of GDB have introduced new interactions that cause problems for simulator GDB stub.) The following commands illustrate the steps needed to configure, compile, and install the correct version of GDB..

```
#
# Script to download and build gdb for ppc64.
#
mkdir -p base
mkdir -p obj
wget -c ftp://ftp.gnu.org/pub/gnu/gdb/gdb-6.3.tar.bz2 -P base
tar jxvf base/gdb-6.3.tar.bz2
pushd obj
../gdb-6.3/configure --target=powerpc64-linux
make all
make install
```

 © IBM Corporation. All rights reserved.

```
popd
```

Simply cut and paste this into a file and execute it as a shell script, sh file. If the wget of the GDB source fails, download it manually from one of the many mirror sites and comment out that line of the script. By default the install stage installs into /usr/local/; for those who do not have write access to /usr/local, specify the –prefix option on configure to specify a different installation directory (for example, configure –target=powerpc64- linux –prefix /home/sdkuser/ local).

Next you should build a version of the kernel that contains the debugging information. To do this, you need a version of the Linux kernel source that contains support for the Cell BE platform. The easiest way to do this is to download and install the kernel source RPM from the Linux on CBE-based Systems Web site at the Barcelona Supercomputing Center (BSC; see Resources). The process for building the kernel depends on the host system, installed tools, and other details, and is beyond the scope of this paper. This article only covers the necessary steps to enable the debugging information. The example commands shown illustrate these steps on a Linux x86 platform with Cell BE SDK 3.0 installed. To enable debugging information in the kernel, go to the directory where you will build the kernel and type:

```
ARCH=powerpc PLATFORM=cell CROSS_COMPILE=/opt/cell/bin/ppu- make xconfig
```

The make xconfig command brings up the configuration menu shown in Figure 4-3. Scroll down and click on the "Kernel hacking" in the left-hand set of options, then click on the "Compile the kernel with debug info" (DEBUG_INFO) on the right-hand side set of options. This option specifies that symbols and source information are retained in the generated binary to allow source-level debugging. In some cases, you might also choose to turn off certain compiler optimizations to make debugging easier. In particular, disabling the - fomit-frame-pointer optimization allows the debugger backtrack command to work reliably, and changing the optimization level from -Os to -O0 will make it easier for GDB to associate individual instructions with a line in the source code. After making all the desired changes, save the configuration, exit the configuration dialog, and then rebuild the kernel.

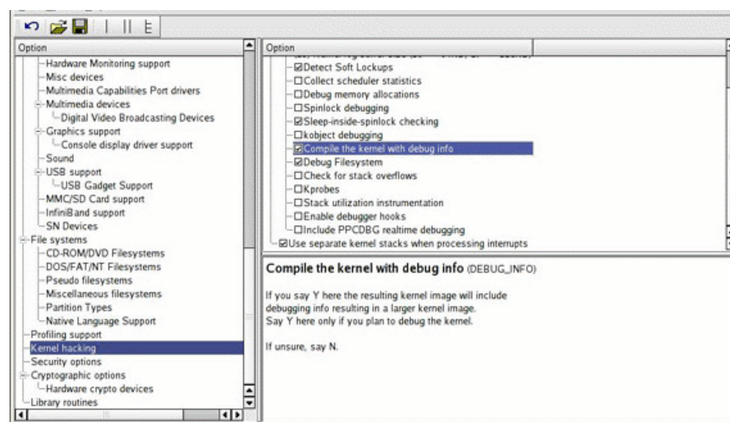


Figure 4-3. The make xconfig screen

Next, start the simulator with the newly built kernel. To ensure that the simulator is using the new kernel, create a symbolic link named vmlinux to the new kernel in the current directory before starting the simulator. To verify that the correct kernel is being used, check the name of the kernel file displayed by the simulator during start-up.

Now you are ready to start a debug session. First, start the simulator and click on the "Service GDB" button; notice that the text of the button changes to "Waiting for GDB... ." In another window, change directories to the location where you compiled vmlinux and start the GDB session with the command

```
/usr/local/bin/powerpc64-linux-gdbtui vmlinux
```

then at the (gdb) prompt type

```
break start_kernel
target remote :2345
continue
```

You should see something very similar to Figure 4-4.

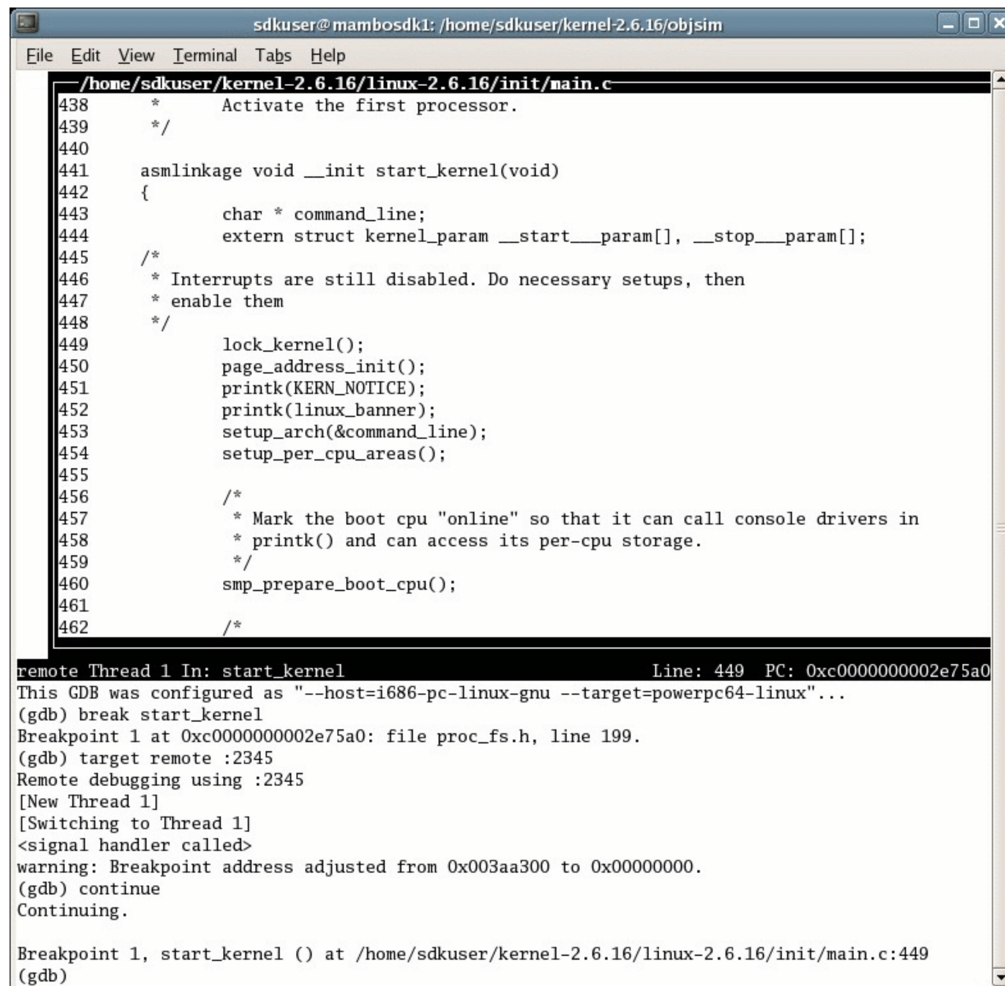


Figure 4-4. The kernel debug session

Now GDB is attached to the simulator and can monitor and control the execution of the Linux kernel. From here it is possible to set additional breakpoints, display variables by name, display processor registers, display a stack trace, single-step execution, and so on.

 © IBM Corporation. All rights reserved.

CHAPTER 5

Accessing the Host Environment

This chapter describes several mechanisms that are provided to allow interactions between the host and simulated systems. Topics in this chapter include:

- *The Callthru Utility*
- *Bogus Network Support*

© International Business Machines Corporation. All rights reserved.

The Callthru Utility

The callthru utility allows you to copy files between the host system and the simulated system while it is running. This utility runs within the simulated system and accesses files in the host system using special callthru functions of the simulator. The source code for this utility is provided with the simulator in the sample/callthru directory as a sample of the use of the simulator callthru functions. In the Cell SDK, the callthru utility is installed as a binary application in the simulator system root image in the /usr/bin directory. The callthru utility supports the following options:

- To write standard input into <filename> on the host system, issue

  ```
  callthru sink <filename>
  ```

- To write the contents of <filename> on the host system to standard output, issue

  ```
  callthru source <filename>
  ```

Redirecting appropriately lets you copy files between the host and simulated system. For example, to copy the /tmp/matrix_mul application from the host into the simulated system and then run it, issue the following commands in the console window of the simulated system:

```
callthru source /tmp/matrix_mul > matrix_mul
chmod +x matrix_mul
./matrix_mul
```

Another commonly used feature of the callthru utility is the exit option, which will stop the simulation, similar to the stop button of the GUI, but initiated by the callthru utility inside the simulator rather than through user interaction. This is especially useful for constructing “scripted” executions of the simulator that involve alternating steps in the simulator and the simulated system.

- To stop the simulator and return control back to currently active run script or the GUI / command line, issue

  ```
  callthru sink <filename>
  ```

Bogus Network Support

Bogus network support was developed to enable network communications with reasonable performance between the simulated system and other systems. This is accomplished using a special ethernet device that uses callthru functions of the simulator to send and receive network packets to the host system. To enable communication with other systems, the host system must be configured to relay packets from the simulated system out to the real network.

The bogus network facility can be configured and used in a variety of ways. A detailed description of the Linux and SystemSim commands to set up and manage bogus network communications is provided below. For user convenience, the most common approach to using the bogus network has been automated using Tcl procedures. These will be described first since most users should find these sufficient for simple network communication between the host and simulated system.

Bogusnet Tcl functions

To simplify the process of using the bogus network features of the simulator, two Tcl procedures are provided that automate most of the steps required to set up the network interface and then clean up once the simulation has finished. These procedures are:

```
% bogusnet::init <network>    ;# To initialize the network
% bogusnet::cleanup           ;# To free network resources
```

The <network> parameter specifies the IP network to be used by the bogusnet facility. Typically this will be a network that has been reserved for use in private networks, such as 172.20.0. To use these Tcl procedures, simply issue the

© IBM Corporation. All rights reserved.

bogusnet::init before initiating the Linux boot process, and issue the bogusnet::cleanup just before exiting the simulator.

Certain tasks for configuring the bogus network require root privileges. A special utility, snif (for Systemsim Network InterFace) is provided with the simulator to perform these privileged configuration tasks. The snif utility is designed to perform only those tasks required to access the bogus network support, and nothing else (details on the tasks performed by snif are provided below). This means that non-root users simply need to be able to execute the snif utility with root privileges to be able to use the bogus network support.

There are a variety of mechanisms that can be used to provide these privileges. The default approach used in the SDK is to install the snif utility as setuid root. This means that snif will run with root privileges when run by any user. If this arrangement is considered undesirable, other approaches can used. The simulator includes an alternate version of the bogusnet initialization procedure, bogusnet::sudo_init, that uses sudo as the means to provide snif with root privileges. The following entries should be added to the sudoers file to give the non-root user sdkuser the authorization to perform these tasks:

```
sdkuser     ALL=(ALL) NOPASSWD: /opt/ibm/systemsim-cell/bin/snif
```

Extended Description of Bogusnet support

There are three key components to Bogus net communications:

1. A facility on the host system that provides SystemSim with a path to the network. The TUN/TAP support available for Linux is a good choice for this component, and we assume TUN/TAP in the remainder of this how-to.
2. The systemsim support for bogus net. This support is not enabled by default. Simulator commands are used to enable the bogus net support.
3. An OS kernel with a bogus network driver. Patches for several versions of linux are available.

Setting up TUN/TAP on the host system

To start, we want to make sure that TUN/TAP support is available on the host system. To do this, check your kernel config file for CONFIG_TUN=m or CONFIG_TUN=y.

```
% grep CONFIG_TUN= /boot/config-`uname -r`
```

TUN/TAP is configured in the stock kernels of many newer distributions, including Fedora Core 6. If your kernel does not include TUN/TAP support, you will need to either install a new kernel or reconfigure/rebuild your kernel.

Also you need to make sure /dev/net/tun exists.

```
% ls -l /dev/net/tun
```

If /dev/net/tun does not exist, make it (as root) with:

```
% mknod /dev/net/tun c 10 200
```

At this point, the TUN/TAP device should be ready for use by the simulator bogusnet support if the simulator is run by the root user. However, to make Tun/Tap support available to non-root users, additional steps are required. These are covered in the following section.

Setting up a TUN/TAP interface for a non-root user

The procedure for making TUN/TAP available to non-root users depends on the version of Linux kernel being used on the system. Prior to Linux 2.6.18, a simple though perhaps not very secure means of doing this is to allow all users read and write access to the /dev/net/tun device. This can be accomplished with a simple chmod command (as root):

```
% chmod 666 /dev/net/tun
```

More secure approaches, such as restricting access by groupid, are also possible. Note: Many newer Linux distributions include support for udev, which is designed to dynamically manage device nodes along with their permissions and other attributes. For these systems, udev may be used to create /dev/net/tun with the desired permissions. For example, in Fedora Core 6 systems, this can be accomplished by creating a file (like systemsim.rules) in the directory /etc/udev/rules.d/ containing:

```
KERNEL=="tun", MODE="0666"
```

Note: Some distributions that include udev support explicitly disable udev support for /dev/net/tun. In particular, udev support for /dev/net/tun is disabled in Fedora Core 4 and Fedora Core 5.

For systems using 2.6.18 and later versions of Linux, the kernel imposes a further restriction on access to the TUN/TAP facility, effectively preventing non-root users from creating new TUN/TAP interfaces. For these systems, non-root users can only access existing TUN/TAP interfaces which were previously created by root and are designated as owned by that user. Unfortunately, there are no standard utilities for creating / assigning TUN/TAP interfaces, so the snif (for Systemsim Network InterFace) utility has been developed and provided with simulator for this purpose.

The snif utility is available in the simulator bin directory. To create a new TUN/TAP interface with IP address 172.20.0.1 for use by a user with uid 5555, issue the following command:

```
% snif -c -u 5555 172.20.0.1
```

To create a new TUN/TAP interface, snif opens the /dev/net/tun device file and issues a sequence of ioctl commands to set the name of the interface (to "tap*n*" where n is chosen as the first available tap interface name), set the owner of the interface to the specified uid, and make the interface persistent. snif then uses the ifconfig command to set the IP address for the interface. If these operations are successful, snif will display the name of the interface (typically "tap0") that was created. This interface name must be specified on the simulator command to initialize the simulator bogus net support.

snif will establish a private network on the host system to which the bogus network device on the simulated system can attach after it is booted. We use the IP address 172.20.0.1 since it belongs to one of the special ranges of addresses that have been reserved for use in private networks (see [Wikipedia: classful network]).

The TUN/TAP interface created by snif is persistent, meaning that it remains allocated until explicitly deleted (or until the host system is restarted). Therefore, it is important to release this interface when it is no longer required (such as at the end of the simulator session). To free a TUN/TAP interface previously allocated by snif, issue the following command:

```
% snif -d <interface name>
```

where the <interface name> parameter is the name of the interface to be freed, typically "tap0".

Configuring SystemSim support for Bogus Net

To enable bogus net support, you need to issue simulator commands that configure and initialize the bogus network. These commands must be issued before booting the Linux kernel on the simulator so that Linux recognizes the bogus network device during its boot process. The general form of the command to initialize bogus net is

```
mysim bogus net init 0 <mac address> <interface name> <irq>
```

The <mac address> parameter is the MAC Hardware Address you want the emulated ethernet to use. It should be unique on your network (i.e. not used by any other emulated hosts, or by any host network adapter). The <interface name> parameter is the name of the interface to be used, typically "tap0". The <irq> parameter specifies the interrupt request queue id to be used by bogus network device. Some simulated machines, such as Cell, do not include a

 © IBM Corporation. All rights reserved.

model for the external interrupt controller, and for these machines the bogus network device must be operated in polled mode, which is specified by a value of 0 for the <irq> parameter. The GPU/L simulator does include an external interrupt controller model, and for GPU/L the <irq> parameter should be set to 9.

The bogus net device driver

The final component required for bogus network communication is an OS kernel with the bogus net device driver. This driver can be either built into the kernel or packaged as a loadable module. Currently, the bogus net device driver is not included in the mainline kernel and must be added as a patch on top of the kernel. Patches for several versions of Linux are available. Depending on where you obtained the kernel you use on SystemSim, this step may have already be done for you.

Connecting to the simulation host

Once these components are in place, you are ready to use bogus net for network communications with the simulated system. Start the simulation and when you reach the linux prompt enter the following commands on the simulated console (UART):

```
% mount /proc
% ifconfig eth0 172.20.0.2 netmask 255.255.255.0
```

You should then be able to ping the host system from the simulated system (and vice-versa)

```
% ping -c 1 172.20.0.1
```

Troubleshooting BogusNet

ping of system host from simulated host results in ping icmp open socket: Operation not permitted:

This was seen in the Cell kernel. First thing, check and see if you can ping the simulated host from the system host. If this is working, you know that the bogus network traffic is flowing in both directions and the problem is most likely with your kernel or root disk – most likely the latter.

operations seems to hang :

Beware of firewalls/ipchains/iptables – most like you have a firewall of some sort blocking the port. Either disable your firewall or open up the specific port.

Simulation Library

The simulator provides sample source code and makefiles for a library that can be used by SPE programs to conveniently access certain simulator callthru features, such as file operations and printf. The sample source code can be found in /opt/ibm/systemsim-cell/sample/cell/libsim. The functions provided include:

- sim_open – open a file on the host system
- sim_close - close a host file
- sim_read - read data from a host file
- sim_write - write data to a host file
- sim_lseek - respostion the file pointer for a host file
- sim_start_timer - start simulator timer
- sim_stop_timer - stop the simulator timer and print elapsed time

- sim_printf - format and print data to simulator output file.
- sim_instructions - return the instruction count for the current SPU
- sim_cycles - return the cycle count for the current SPU

The header file, libsim.h, fully describes the functions available in the library.

© IBM Corporation. All rights reserved.

© IBM Corporation. All rights reserved.