Contents

Preface ix

1. Quick Start 1–1
   1.1 System Requirements 1–1
   1.2 EDA Tool Requirements 1–2
   1.3 Running Simulations and Synthesis 1–3

2. OpenSPARC T1 Design Implementation 2–1
   2.1 OpenSPARC T1 Design Hierarchy 2–1
   2.2 Module Directory Structure 2–3
   2.3 Megacells 2–6
   2.4 External Interfaces 2–7

3. OpenSPARC T1 Verification Environment 3–1
   3.1 OpenSPARC T1 Verification Environment 3–1
   3.2 Running a Regression 3–3
   3.3 Verification Code 3–6
   3.4 PLI Code Used For the Test Bench 3–8
   3.5 Verification Test File Locations 3–9
   3.6 Compiling Source Code for Tools 3–10
   3.7 Gate-Level Verification 3–10
4. OpenSPARC T1 Synthesis 4–1
   4.1 Synthesis Flow for the OpenSPARC T1 Processor 4–1
   4.2 Synthesis Output 4–3

5. OpenSPARC T1 FPGA Synthesis 5–1
   5.1 Synplicity FPGA Synthesis Flow for the OpenSPARC T1 Processor 5–1
   5.2 Synplicity FPGA Synthesis Output 5–4
   5.3 XST Synthesis Flow for the OpenSPARC T1 Processor 5–4
   5.4 XST Synthesis Output 5–6
   5.5 Selecting OpenSPARC T1 Options for Reduced Size 5–6

6. OpenSPARC T1 EDK Project 6–1
   6.1 System Description of the EDK project 6–1
   6.2 Generation of a bit file for a Xilinx FPGA 6–3
   6.3 Running OpenSPARC T1 diagnostics on an evaluation board 6–5
   6.4 Running a standalone program on the OpenSPARC T1 core on an evaluation board 6–8
   6.5 Running system-level simulation with Modelsim 6–12

A. Design and Verification Manual Pages A–1
   A.1 sims A–1
   A.2 midas A–15
   A.3 goldfinger A–24
   A.4 regreport A–32
   A.5 vlog A–33
# Figures

| FIGURE 2-1 | OpenSPARC T1 Block Diagram   | 2–2 |
| FIGURE 2-2 | OpenSPARC T1 External Interfaces | 2–7 |
| FIGURE 3-1 | OpenSPARC T1 Verification Test Bench Overview | 3–3 |
| FIGURE 6-1 | EDK OpenSPARC T1 system block diagram | 6–2 |
| FIGURE 6-2 | Xilinx Platform Studio (XPS) window with OpenSPARC project open. | 6–4 |
| FIGURE 6-3 | Allocation of the ML410 64-MB DRAM with the hypervisor firmware set-up | 6–8 |
| FIGURE 6-4 | Memory addressing of the MicroBlaze memory controller | 6–13 |
Tables

TABLE 1-1 Disk Space Requirements 1–1
TABLE 1-2 Contents of the OpenSPARC T1 Directory 1–3
TABLE 1-3 Environment Variables in .cshrc File 1–4
TABLE 2-1 OpenSPARC T1 Top-Level Modules 2–3
TABLE 3-1 Source Code Types in the Verification Environment 3–2
TABLE 3-2 Details of Regression Groups 3–3
TABLE 3-3 OpenSPARC T1 Verification Test Bench Modules 3–6
TABLE 3-4 PLI Source Code and Object Libraries 3–8
TABLE 3-5 Options for the mkplilib Script 3–9
TABLE 3-6 Verification Test File Directories 3–10
TABLE 3-7 Gate Netlist Files 3–12
TABLE 4-1 Synthesis Script Details 4–2
TABLE 4-2 Synthesis Output 4–3
TABLE 5-1 Synplicity FPGA Synthesis Script Details 5–2
TABLE 5-2 Synplicity FPGA Synthesis Output 5–4
TABLE 5-3 XST Synthesis Script Details 5–5
TABLE 5-4 FPGA Synthesis Output 5–6
TABLE A-1 Environment Variables A–13
Preface

The OpenSPARC™ T1 Processor Design and Verification User’s Guide gives an overview of the design hierarchy on the OpenSPARC T1 processor. It also describes the files, procedures, and tools needed for running simulations and synthesis on the OpenSPARC T1 processor.

This book covers the following topics:
- Design and Verification implementation overview
- Design and Verification directory and files structure
- System and Electronic Design Automation (EDA) tools required to run simulations and synthesis
- Tools and scripts required to run simulation or complete regressions, including simulation flow
- Synthesis flow and scripts

How This Document Is Organized

Chapter 1 describes quick steps to run simulations after you download the design and verification files from the web site. It also includes system requirements and EDA tools requirements to run simulations and synthesis.

Chapter 2 gives an overview of the OpenSPARC T1 design hierarchy and directory structure.

Chapter 3 gives an overview of the OpenSPARC T1 verification environment implementation and directory structure. The verification environment includes test benches, tests, scripts, and Verilog Programming Language Interface (PLI).

Chapter 4 describes the synthesis flow and synthesis scripts.
Chapter 5 describes the Synplicity Pro software scripts and the XST software scripts for synthesizing field programmable gate arrays (FPGA).

Chapter 6 describes the included EDK project, which enables the user to download the synthesized OpenSPARC T1 core to a Xilinx FPGA, run diagnostic tests on it, and boot hypervisor.

Appendix A has manual pages for regression commands.

Using UNIX Commands

This document might not contain information about basic UNIX® commands and procedures such as shutting down the system, booting the system, and configuring devices. Refer to the following for this information:

- Software documentation that you received with your system
- Solaris™ Operating System documentation, which is at:
  
  http://docs.sun.com

Shell Prompts

<table>
<thead>
<tr>
<th>Shell</th>
<th>Prompt</th>
</tr>
</thead>
<tbody>
<tr>
<td>C shell</td>
<td>machine-name%</td>
</tr>
<tr>
<td>C shell superuser</td>
<td>machine-name#</td>
</tr>
<tr>
<td>Bourne shell and Korn shell</td>
<td>$</td>
</tr>
<tr>
<td>Bourne shell and Korn shell superuser</td>
<td>#</td>
</tr>
</tbody>
</table>
Typographic Conventions

<table>
<thead>
<tr>
<th>Typeface</th>
<th>Meaning</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>AaBbCc123</td>
<td>The names of commands, files, and directories; on-screen computer output</td>
<td>Edit your .login file. Use ls -a to list all files. % You have mail.</td>
</tr>
<tr>
<td>AaBbCc123</td>
<td>What you type, when contrasted with on-screen computer output</td>
<td>% su Password:</td>
</tr>
<tr>
<td>AaBhCc123</td>
<td>Book titles, new words or terms, words to be emphasized. Replace command-line variables with real names or values.</td>
<td>Read Chapter 6 in the User’s Guide. These are called <em>class</em> options. You must be superuser to do this. To delete a file, type rm filename.</td>
</tr>
</tbody>
</table>

* The settings on your browser might differ from these settings.

Related Documentation

The documents listed as online or download are available at:

http://www.opensparc.net/

<table>
<thead>
<tr>
<th>Application</th>
<th>Title</th>
<th>Part Number</th>
<th>Format</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenSPARC T1 instruction set</td>
<td>UltraSPARC Architecture 2005 Specification</td>
<td>950-4895-03</td>
<td>PDF</td>
<td>Online</td>
</tr>
<tr>
<td>OpenSPARC T1 processor’s internal registers</td>
<td>UltraSPARC T1 Supplement to the UltraSPARC Architecture 2005</td>
<td>819-3404-02</td>
<td>PDF</td>
<td>Online</td>
</tr>
<tr>
<td>OpenSPARC T1 megacells</td>
<td>OpenSPARC T1 Processor Megacell Specification</td>
<td>819-5016-10</td>
<td>PDF</td>
<td>Download</td>
</tr>
<tr>
<td>OpenSPARC T1 signal pin list</td>
<td>OpenSPARC T1 Processor Datasheet</td>
<td>819-5015-10</td>
<td>PDF</td>
<td>Download</td>
</tr>
<tr>
<td>OpenSPARC T1 processor J-Bus and SSI interfaces</td>
<td>OpenSPARC T1 Processor External Interface Specification</td>
<td>819-5014-10</td>
<td>PDF</td>
<td>Download</td>
</tr>
</tbody>
</table>
Documentation, Support, and Training

<table>
<thead>
<tr>
<th>Sun Function</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Documentation</td>
<td><a href="http://www.sun.com/documentation/">http://www.sun.com/documentation/</a></td>
</tr>
<tr>
<td>Support</td>
<td><a href="http://www.sun.com/support/">http://www.sun.com/support/</a></td>
</tr>
<tr>
<td>Training</td>
<td><a href="http://www.sun.com/training/">http://www.sun.com/training/</a></td>
</tr>
</tbody>
</table>

Third-Party Web Sites

Sun is not responsible for the availability of third-party web sites mentioned in this document. Sun does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Sun will not be responsible or liable for any actual or alleged damage or loss caused by or in connection with the use of or reliance on any such content, goods, or services that are available on or through such sites or resources.

Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. You can submit your comments by going to:

http://www.sun.com/hwdocs/feedback

Please include the title and part number of your document with your feedback:

_OpenSPARC T1 Processor Design and Verification User’s Guide,
part number 819-5019-12_
Quick Start

This chapter covers the following topics:

- System Requirements
- EDA Tool Requirements
- Running Simulations and Synthesis

Before you start running simulations or synthesis, make sure you meet system requirements and that you have the required Electronic Design Automation (EDA) tools. Once you download the OpenSPARC T1 tar file from the http://www.opensparc.net web site, follow the steps in this chapter to get started and run your first regression on the OpenSPARC T1 design.

1.1 System Requirements

OpenSPARC T1 regressions are currently supported to run only on SPARC® systems running the Solaris 9 or Solaris 10 Operating System.

Disk space requirements are listed in Table 1-1.

**Table 1-1** Disk Space Requirements

<table>
<thead>
<tr>
<th>Disk Space required</th>
<th>Required for:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.5 Gbyte</td>
<td>Download, unzip or uncompress, and extract from the tar file</td>
</tr>
<tr>
<td>1.6 Gbyte</td>
<td>Run a mini-regression</td>
</tr>
<tr>
<td>67 Gbyte</td>
<td>Run a full regression</td>
</tr>
<tr>
<td>0.3 Gbyte</td>
<td>Run synthesis</td>
</tr>
<tr>
<td>70.4 Gbyte</td>
<td>Total</td>
</tr>
</tbody>
</table>
1.2 EDA Tool Requirements

This section describes the commercial EDA tools required for running simulations for the OpenSPARC T1 processor and synthesizing OpenSPARC T1 Verilog Register Transfer Level (RTL) code.

1.2.1 EDA Simulation Tools

The following EDA tools are required to run Verilog simulations: Verilog Simulator, either VCS or NCVerilog.

- VCS from Synopsys, version 7.1.1R21 or later
- NCVerilog from Cadence, version 5.3.s2 or later

It is permissible to use a Verilog Simulator other than VCS or NCVerilog. See details in Section 3.2.3, “Running Regression With Other Simulators” on page 3-5.

The following EDA tools are optional for running Verilog simulations:

- Vera from Synopsys, version 6.2.8 or later
- Debussy from Novas, version 5.3v19 or later

1.2.2 EDA Synthesis Tools

The following EDA tool is required to perform Verilog RTL synthesis:

- Design Compiler from Synopsys, version X-2005.09 or later

One of the following EDA tool is required to perform Verilog RTL synthesis for field programmable gate arrays (FPGA):

- Synplicity Pro from Synplicity, version 8.5 or later or
- Xilinx Synthesis Technology (XST) from Xilinx, version 9.1i or later

1.2.3 FPGA Tools

The following EDA tools are required to place and route a design on a Xilinx FPGA, download the design to the Xilinx FPGA, and run tests on the FPGA system:

- Embedded Development Kit (EDK) from Xilinx, version 9.1i or later
- Integrated Synthesis Environment (ISE) from Xilinx, version 9.1i or later
- Modelsim from Mentor Graphics, version 6.1e
1.3 Running Simulations and Synthesis

This section outlines the steps needed to obtain the simulation tools, set up the simulation environment, run the simulation, and read its log file.

1.3.1 Get the Simulation Files

1. Download the file.

   Download the OpenSPARCT1.tar.bz2 file from the http://www.opensparc.net web site. For this procedure’s examples, the destination directory is:

   /home/johndoe/OpenSPARCT1

2. Change directories to the directory where you downloaded the file. For example:

   ```
   % cd /home/johndoe/OpenSPARCT1
   ```

3. Use the `bunzip2` command to unzip the file.

   ```
   % bunzip2 OpenSPARC_1.tar.bz2
   ```

4. Extract the tar file using the `tar` command.

   ```
   % tar -xvf OpenSPARC_1.tar
   ```

   This step creates the files and subdirectories listed in Table 1-2 in your current directory.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenSPARCT1.cshrc</td>
<td>File</td>
<td>File to set up environment variables and paths</td>
</tr>
<tr>
<td>README</td>
<td>File</td>
<td>Instructions to set up and run simulations</td>
</tr>
<tr>
<td>lib</td>
<td>Directory</td>
<td>Verilog libraries</td>
</tr>
<tr>
<td>verif</td>
<td>Directory</td>
<td>Verification directories and files</td>
</tr>
</tbody>
</table>
**1.3.2 Set Up Environment Variables**

Edit the `OpenSPARCT1.cshrc` file to set the required environment variables as shown in **TABLE 1-3**:

**TABLE 1-3 Environment Variables in .cshrc File**

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Usage</th>
<th>Example value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DV_ROOT</td>
<td>Running simulations and synthesis</td>
<td><code>/home/johndoe/OpenSPARCT1</code> (Directory where you ran the <code>tar</code> command above)</td>
</tr>
<tr>
<td>MODEL_DIR</td>
<td>Running simulations</td>
<td><code>/home/johndoe/OpenSPARCT1_model</code> (Directory where you want to run your simulations)</td>
</tr>
<tr>
<td>VERA_HOME</td>
<td>Running simulations</td>
<td><code>/import/EDAtools/vera/vera,v6.2.10/5.x</code> (Directory where Vera is installed)</td>
</tr>
<tr>
<td>NOVAS_HOME</td>
<td>Running simulations</td>
<td><code>/import/EDAtools/debussy/debussy,v5.3v19/5.x</code> (Directory where Debussy is installed)</td>
</tr>
<tr>
<td>VCS_HOME</td>
<td>Running VCS simulations</td>
<td><code>/import/EDAtools/vcs7.1.1R21</code> (Directory where VCS is installed)</td>
</tr>
<tr>
<td>NCV_HOME</td>
<td>Running NCVerilog simulations</td>
<td><code>/import/EDAtools/ncverilog/ncverilog.v5.3.55/5.x</code> (Directory where NCVerilog is installed)</td>
</tr>
<tr>
<td>SYN_HOME</td>
<td>Running synthesis</td>
<td><code>/import/EDAtools/synopsys/synopsys.vX-2005.09</code> (Directory where Synopsys is installed)</td>
</tr>
<tr>
<td>CC_BIN</td>
<td>Compiling PLI code</td>
<td><code>/usr/dist/pkgs/devpro,v4.2.5.x-sparc/bin</code> (Directory where C++ Compiler binaries are installed)</td>
</tr>
<tr>
<td>LM_LICENSE_FILE</td>
<td>Running simulations and synthesis</td>
<td><code>/import/EDAtools/licenses/synopsys_key:/import/EDAtools/licenses/ncverilog_key</code> (EDA tool license files)</td>
</tr>
<tr>
<td>SYNP_HOME</td>
<td>Running Synplicity for FPGA synthesis</td>
<td><code>/import/EDAtools/synplicity/synplify.v8.5/fpga_85</code> (Directory where Synplicity is installed)</td>
</tr>
<tr>
<td>MODEL_HOME</td>
<td>Running Modelsim</td>
<td><code>/import/EDAtools/modelsim.v6.1e/modeltech</code></td>
</tr>
</tbody>
</table>
Once you set the environment variables from TABLE 1-3, the OpenSPARCT1.cshrc file sets the following environment variables:

- TRE_ENTRY
- TRE_LOG
- TRE_SEARCH
- ENVDIR
- PERL_MODULE_BASE

The OpenSPARCT1.cshrc script also adds the following directories to your PATH and path variables:

- $DV_ROOT/tools/bin
- $NCV_HOME/tools/bin
- $VCS_HOME/bin
- $VERA_HOME/bin
- $SYN_HOME/sparcOS5/syn/bin
- $CC_BIN

After completing your OpenSPARCT1.cshrc file edits, source it by using the source command:

```
% source /home/johndoe/OpenSPARCT1/OpenSPARCT1.cshrc
```

You might want to include the above command in your ~/.cshrc file so that the above environment variables are set every time you log in.

Finally, create the following symbolic link to set up the correct platform files for the verification environment. For example, if you are running the verification on a x86_64 Linux cluster, you would create the symbolic link as follows:

```
% cd $DV_ROOT/tools/env
% ln -s Makefile.Linux.x86_64 Makefile.system
```
1.3.3 Run Your First Regression

**Note** – OpenSPARC T1 Release 1.4 includes one more environment for single-core, single-thread implementation of OpenSPARC T1. Tests included in the thread1 environment are a subset of the core1 environment. The thread1 environment does not include tests that verify multi-threaded functionality and Stream Processing Unit (SPU) related functionality.

The OpenSPARC T1 Design/Verification package comes with two test bench environments: core1 and chip8.

The core1 environment consists of:
- One SPARC CPU core
- Cache
- Memory
- Crossbar

The core1 environment does not have an I/O subsystem.

The chip8 environment consists of:
- A full OpenSPARC T1 chip, including all eight cores
- Cache
- Memory
- Crossbar
- I/O subsystem

Each environment can perform either a mini-regression or a full regression.

To run a regression, use the sims command as described in Section 1.3.3.1, “To Run a Regression” on page 1-7. The important parameters for the sims command are:
- `-sim_type`: Simulator type
  - Set this to vcs or ncv. For example: `-sim_type=vcs`
- `-group`: Regression group name
  - The choices for -group are: core1_mini, core1_full, chip8_mini, chip8_full, thread1_mini and thread1_full.
  
  For example: `-group=core1_mini`
1.3.3.1 To Run a Regression

1. Create the $MODEL_DIR directory.

```bash
% mkdir $MODEL_DIR
```

2. Change directory to $MODEL_DIR.

```bash
% cd $MODEL_DIR
```

This is where the simulations are run.

3. Run a mini-regression for the core1 environment using the VCS simulator.

```bash
% sims -sim_type=vcs -group=core1_mini
```

This command creates two directories:

- A directory called core1 under $MODEL_DIR. The regression compiles Vera and Verilog code under the core1 directory. This is the Vera and Verilog “build” directory.
- A directory named with today’s date and a serial number, such as 2006_01_07_0 (the format is YYYY_MM_DD_ID) under the current directory where simulations will run. This is the Verilog simulation’s “run” directory. There is one subdirectory under this directory for each diagnostics test.

By default, the simulations are run with Vera. If you do not want to use Vera, add following option to the sims command:

```
-novera_build -novera_run
```

4. Once simulations are completed, run the regreport command to generate a regression report.

```bash
% cd run-directory
% regreport $PWD > report.log
```

Where run-directory is the “run” directory created in the above step, such as 2006_01_07_0.

The core1_mini regression has 68 tests. An example of its report.log output is shown in CODE EXAMPLE 1-1.
If your `report.log` file displays a similar status, you have successfully completed running a mini-regression for the OpenSPARC T1 processor.

### 1.3.4 Run Your First Synthesis

The command to run a synthesis is `rsyn`. For example, to run a synthesis for one of the modules called `efc`, type:

```
% rsyn efc
```

This command runs a synthesis for the `efc` block and creates gate level netlists under the `$DV_ROOT/design/sys/iop/efc/synthesis/gate` directory.

The synthesis flow and scripts are described in more detail in Chapter 4.
1.3.5 Gate-Level Verification

OpenSPARC T1 depends heavily on Cross-Module References (XMRs) within the verification environment. Therefore, dropping in a netlist in place of the RTL core will produce a high number of XMR errors. Because of this, a simple playback support is now added. The details of the suggested methodology are described in Chapter 3.
OpenSPARC T1 Design Implementation

This chapter gives details on the following topics:
- OpenSPARC T1 Design Hierarchy
- Module Directory Structure
- Megacells
- External Interfaces

2.1 OpenSPARC T1 Design Hierarchy

The top-level Verilog module for the OpenSPARC T1 processor is called OpenSPARCT1. There are various types of design blocks at the top level:
- **Cluster** – A hierarchical block with one or more instances of this block in the design. For example, a SPARC CPU core is called sparc and has eight instances at the top level.
- **Pads** – Input, Output and Bi-directional pins on the OpenSPARC T1 processor, including input buffer, output driver, etc. For example, pad_ddr0 contains pads for DDR bank 0.
- **Repeaters** – Many buffers and repeaters at the top level for signals going to blocks or signals with long traces in the physical implementation, such as dram0_ddr0_rptr.
- **Clock** – This includes global clock distribution, including buffers, drivers, repeaters, and so on.
A block diagram of the OpenSPARC T1 processor is shown in FIGURE 2-1.

FIGURE 2-1  OpenSPARC T1 Block Diagram
### 2.2 Module Directory Structure

The Verilog RTL for the OpenSPARC T1 processor is in the `$DV_ROOT/design/sys/iop` directory. All the top-level modules that make up that RTL, and their locations, are listed in TABLE 2-1. You can also browse the Verilog source code on the OpenSPARC web site at [http://www.opensparc.net](http://www.opensparc.net).

**TABLE 2-1**  OpenSPARC T1 Top-Level Modules

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Type</th>
<th>Number of Instances</th>
<th>Instance Names</th>
<th>Directory Location under $DV_ROOT/design/sys/iop</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ccx</td>
<td>Cluster</td>
<td>1</td>
<td>ccx</td>
<td>ccx</td>
<td>CPU-Cache Crossbar</td>
</tr>
<tr>
<td>ctu</td>
<td>Cluster</td>
<td>1</td>
<td>ctu</td>
<td>ctu</td>
<td>Clock and Test Unit</td>
</tr>
<tr>
<td>dram</td>
<td>Cluster</td>
<td>2</td>
<td>dram02, dram13</td>
<td>dram</td>
<td>DRAM controller</td>
</tr>
<tr>
<td>efc</td>
<td>Cluster</td>
<td>1</td>
<td>efc</td>
<td>efc</td>
<td>e-Fuse Cluster</td>
</tr>
<tr>
<td>fpu</td>
<td>Cluster</td>
<td>1</td>
<td>fpu</td>
<td>fpu</td>
<td>Floating Point Unit</td>
</tr>
<tr>
<td>iobdg</td>
<td>Cluster</td>
<td>1</td>
<td>iobdg</td>
<td>iobdg</td>
<td>I/O bridge</td>
</tr>
<tr>
<td>jbi</td>
<td>Cluster</td>
<td>1</td>
<td>jbi</td>
<td>jbi</td>
<td>J-Bus Interface</td>
</tr>
<tr>
<td>scbuf</td>
<td>Cluster</td>
<td>4</td>
<td>scbuf[0-3]</td>
<td>scbuf</td>
<td>L2 $ buffer</td>
</tr>
<tr>
<td>scdata</td>
<td>Cluster</td>
<td>4</td>
<td>scdata[0-3]</td>
<td>scdata</td>
<td>L2 $ data</td>
</tr>
<tr>
<td>sctag</td>
<td>Cluster</td>
<td>4</td>
<td>sctag[0-3]</td>
<td>sctag</td>
<td>L2 $ tag</td>
</tr>
<tr>
<td>sparc</td>
<td>Cluster</td>
<td>8</td>
<td>sparc[0-7]</td>
<td>sparc</td>
<td>SPARC CPU core</td>
</tr>
<tr>
<td>pad_ddr0</td>
<td>Pad</td>
<td>1</td>
<td>pad_ddr0</td>
<td>pads</td>
<td>DDR0 pads</td>
</tr>
<tr>
<td>pad_ddr1</td>
<td>Pad</td>
<td>1</td>
<td>pad_ddr1</td>
<td>pads</td>
<td>DDR1 pads</td>
</tr>
<tr>
<td>pad_ddr2</td>
<td>Pad</td>
<td>1</td>
<td>pad_ddr2</td>
<td>pads</td>
<td>DDR2 pads</td>
</tr>
<tr>
<td>pad_ddr3</td>
<td>Pad</td>
<td>1</td>
<td>pad_ddr3</td>
<td>pads</td>
<td>DDR3 pads</td>
</tr>
<tr>
<td>pad_efc</td>
<td>Pad</td>
<td>1</td>
<td>pad_efc</td>
<td>pads</td>
<td>efc pads</td>
</tr>
<tr>
<td>pad_jbusr</td>
<td>Pad</td>
<td>1</td>
<td>pad_jbusr</td>
<td>pads</td>
<td>J-Bus pads</td>
</tr>
<tr>
<td>pad_jbusl</td>
<td>Pad</td>
<td>2</td>
<td>pad_jbusl, pad_dbg</td>
<td>pads</td>
<td>J-Bus pads</td>
</tr>
<tr>
<td>Module Name</td>
<td>Type</td>
<td>Number of Instances</td>
<td>Instance Names</td>
<td>Directory Location under $DV_ROOT/design/syss/iop</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>--------------</td>
<td>---------------------</td>
<td>---------------------------------</td>
<td>---------------------------------------------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>pad_misc</td>
<td>Pad</td>
<td>1</td>
<td>pad_misc</td>
<td>pads</td>
<td>Miscellaneous pads</td>
</tr>
<tr>
<td>bw_temp_diode</td>
<td>Pad</td>
<td>2</td>
<td>pad_diode0, pad_diode1</td>
<td>analog</td>
<td>Temperature diode pads</td>
</tr>
<tr>
<td>bw_ctu_pad_cluster</td>
<td>Pad</td>
<td>1</td>
<td>pad_ctu</td>
<td>analog</td>
<td>CTU pads</td>
</tr>
<tr>
<td>ccx_iob_rptr</td>
<td>Repeater</td>
<td>1</td>
<td>ccx_iob_rptr</td>
<td>cmp</td>
<td>ccx repeater</td>
</tr>
<tr>
<td>ccx_spc_rpt</td>
<td>Repeater</td>
<td>8</td>
<td>ccx_spc_rpt[0-7]</td>
<td>cmp</td>
<td>ccx repeater</td>
</tr>
<tr>
<td>ctu_top_rptr</td>
<td>Repeater</td>
<td>1</td>
<td>ctu_top_rptr</td>
<td>cmp</td>
<td>ctu repeater</td>
</tr>
<tr>
<td>ctu_top_rptr2</td>
<td>Repeater</td>
<td>1</td>
<td>ctu_top_rptr2</td>
<td>cmp</td>
<td>ctu repeater</td>
</tr>
<tr>
<td>ctu_bottom_rptr</td>
<td>Repeater</td>
<td>1</td>
<td>ctu_bottom_rptr</td>
<td>cmp</td>
<td>ctu repeater</td>
</tr>
<tr>
<td>dram0_ddr0_rptr</td>
<td>Repeater</td>
<td>1</td>
<td>dram0_ddr0_rptr0</td>
<td>cmp</td>
<td>dram repeater</td>
</tr>
<tr>
<td>dram1_ddr1_rptr</td>
<td>Repeater</td>
<td>1</td>
<td>dram1_ddr1_rptr0</td>
<td>cmp</td>
<td>dram repeater</td>
</tr>
<tr>
<td>dram2_ddr2_rptr</td>
<td>Repeater</td>
<td>1</td>
<td>dram2_ddr2_rptr0</td>
<td>cmp</td>
<td>dram repeater</td>
</tr>
<tr>
<td>dram3_ddr3_rptr</td>
<td>Repeater</td>
<td>1</td>
<td>dram3_ddr3_rptr0</td>
<td>cmp</td>
<td>dram repeater</td>
</tr>
<tr>
<td>dram_ddr_pad_rptr</td>
<td>Repeater</td>
<td>2</td>
<td>dram0_ddr0_rptr2, dram2_ddr2_rptr2</td>
<td>cmp</td>
<td>dram repeater</td>
</tr>
<tr>
<td>dram_ddr_pad_rptr_south</td>
<td>Repeater</td>
<td>2</td>
<td>dram1_ddr1_rptr2, dram3_ddr3_rptr2</td>
<td>cmp</td>
<td>dram repeater</td>
</tr>
<tr>
<td>dram_ddr_rptr</td>
<td>Repeater</td>
<td>2</td>
<td>dram0_ddr0_rptr1, dram2_ddr2_rptr1</td>
<td>cmp</td>
<td>dram repeater</td>
</tr>
<tr>
<td>dram_ddr_rptr_south</td>
<td>Repeater</td>
<td>2</td>
<td>dram1_ddr1_rptr1, dram3_ddr3_rptr1</td>
<td>cmp</td>
<td>dram repeater</td>
</tr>
<tr>
<td>dram_12_buf2</td>
<td>Repeater</td>
<td>8</td>
<td>dram_sc_[0-3]<em>rep1, dram_sc</em>[0-3]_rep3</td>
<td>cmp</td>
<td>dram repeater</td>
</tr>
<tr>
<td>dram_sc_0_rep2</td>
<td>Repeater</td>
<td>1</td>
<td>dram_sc_0_rep2</td>
<td>cmp</td>
<td>dram repeater</td>
</tr>
<tr>
<td>dram_sc_1_rep2</td>
<td>Repeater</td>
<td>1</td>
<td>dram_sc_1_rep2</td>
<td>cmp</td>
<td>dram repeater</td>
</tr>
<tr>
<td>dram_sc_2_rep2</td>
<td>Repeater</td>
<td>1</td>
<td>dram_sc_2_rep2</td>
<td>cmp</td>
<td>dram repeater</td>
</tr>
<tr>
<td>dram_sc_3_rep2</td>
<td>Repeater</td>
<td>1</td>
<td>dram_sc_3_rep2</td>
<td>cmp</td>
<td>dram repeater</td>
</tr>
<tr>
<td>ff_dram_sc_bank0</td>
<td>Repeater</td>
<td>1</td>
<td>ff_dram_sc_bank0</td>
<td>cmp</td>
<td>dram repeater</td>
</tr>
</tbody>
</table>
### TABLE 2-1  OpenSPARC T1 Top-Level Modules (Continued)

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Type</th>
<th>Number of Instances</th>
<th>Instance Names</th>
<th>Directory Location under $DV_ROOT/design/sy/s/iop</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ff_dram_sc_bank1</td>
<td>Repeater</td>
<td>1</td>
<td>ff_dram_sc_bank1</td>
<td>cmp</td>
<td>dram repeater</td>
</tr>
<tr>
<td>ff_dram_sc_bank2</td>
<td>Repeater</td>
<td>1</td>
<td>ff_dram_sc_bank2</td>
<td>cmp</td>
<td>dram repeater</td>
</tr>
<tr>
<td>ff_dram_sc_bank3</td>
<td>Repeater</td>
<td>1</td>
<td>ff_dram_sc_bank3</td>
<td>cmp</td>
<td>dram repeater</td>
</tr>
<tr>
<td>ff_jbi_sc0_1</td>
<td>Repeater</td>
<td>1</td>
<td>ff_jbi_sc0_1</td>
<td>cmp</td>
<td>jbi repeater</td>
</tr>
<tr>
<td>ff_jbi_sc0_2</td>
<td>Repeater</td>
<td>1</td>
<td>ff_jbi_sc0_2</td>
<td>cmp</td>
<td>jbi repeater</td>
</tr>
<tr>
<td>ff_jbi_sc1_1</td>
<td>Repeater</td>
<td>1</td>
<td>ff_jbi_sc1_1</td>
<td>cmp</td>
<td>jbi repeater</td>
</tr>
<tr>
<td>ff_jbi_sc1_2</td>
<td>Repeater</td>
<td>1</td>
<td>ff_jbi_sc1_2</td>
<td>cmp</td>
<td>jbi repeater</td>
</tr>
<tr>
<td>ff_jbi_sc2_1</td>
<td>Repeater</td>
<td>1</td>
<td>ff_jbi_sc2_1</td>
<td>cmp</td>
<td>jbi repeater</td>
</tr>
<tr>
<td>ff_jbi_sc2_2</td>
<td>Repeater</td>
<td>1</td>
<td>ff_jbi_sc2_2</td>
<td>cmp</td>
<td>jbi repeater</td>
</tr>
<tr>
<td>ff_jbi_sc3_1</td>
<td>Repeater</td>
<td>1</td>
<td>ff_jbi_sc3_1</td>
<td>cmp</td>
<td>jbi repeater</td>
</tr>
<tr>
<td>ff_jbi_sc3_2</td>
<td>Repeater</td>
<td>1</td>
<td>ff_jbi_sc3_2</td>
<td>cmp</td>
<td>jbi repeater</td>
</tr>
<tr>
<td>iob_ccx_rptr</td>
<td>Repeater</td>
<td>1</td>
<td>iob_ccx_rptr</td>
<td>cmp</td>
<td>iob repeater</td>
</tr>
<tr>
<td>iob_jbi_rptr_0</td>
<td>Repeater</td>
<td>1</td>
<td>iob_jbi_rptr_0</td>
<td>cmp</td>
<td>iob repeater</td>
</tr>
<tr>
<td>iob_jbi_rptr_1</td>
<td>Repeater</td>
<td>1</td>
<td>iob_jbi_rptr_1</td>
<td>cmp</td>
<td>iob repeater</td>
</tr>
<tr>
<td>jbi_l2_buf2</td>
<td>Repeater</td>
<td>4</td>
<td>rep_jbi_sc[0-3]_1</td>
<td>cmp</td>
<td>jbi repeater</td>
</tr>
<tr>
<td>rep_jbi_sc0_2</td>
<td>Repeater</td>
<td>1</td>
<td>rep_jbi_sc0_2</td>
<td>cmp</td>
<td>jbi repeater</td>
</tr>
<tr>
<td>rep_jbi_sc1_2</td>
<td>Repeater</td>
<td>1</td>
<td>rep_jbi_sc1_2</td>
<td>cmp</td>
<td>jbi repeater</td>
</tr>
<tr>
<td>rep_jbi_sc2_2</td>
<td>Repeater</td>
<td>1</td>
<td>rep_jbi_sc2_2</td>
<td>cmp</td>
<td>jbi repeater</td>
</tr>
<tr>
<td>rep_jbi_sc3_2</td>
<td>Repeater</td>
<td>1</td>
<td>rep_jbi_sc3_2</td>
<td>cmp</td>
<td>jbi repeater</td>
</tr>
<tr>
<td>sc_0_1_dbg_rptr</td>
<td>Repeater</td>
<td>1</td>
<td>sc_0_1_dbg_rptr</td>
<td>cmp</td>
<td>L2 repeater</td>
</tr>
<tr>
<td>sc_2_3_dbg_rptr</td>
<td>Repeater</td>
<td>1</td>
<td>sc_2_3_dbg_rptr</td>
<td>cmp</td>
<td>L2 repeater</td>
</tr>
<tr>
<td>sctag_cpx_rptr_0</td>
<td>Repeater</td>
<td>1</td>
<td>sctag_cpx_rptr_0</td>
<td>cmp</td>
<td>L2 repeater</td>
</tr>
<tr>
<td>sctag_cpx_rptr_1</td>
<td>Repeater</td>
<td>1</td>
<td>sctag_cpx_rptr_1</td>
<td>cmp</td>
<td>L2 repeater</td>
</tr>
<tr>
<td>sctag_cpx_rptr_2</td>
<td>Repeater</td>
<td>1</td>
<td>sctag_cpx_rptr_2</td>
<td>cmp</td>
<td>L2 repeater</td>
</tr>
<tr>
<td>sctag_cpx_rptr_3</td>
<td>Repeater</td>
<td>1</td>
<td>sctag_cpx_rptr_3</td>
<td>cmp</td>
<td>L2 repeater</td>
</tr>
<tr>
<td>sctag_pcm_rptr_0</td>
<td>Repeater</td>
<td>1</td>
<td>sctag_pcm_rptr_0</td>
<td>cmp</td>
<td>L2 repeater</td>
</tr>
<tr>
<td>sctag_pcm_rptr_1</td>
<td>Repeater</td>
<td>1</td>
<td>sctag_pcm_rptr_1</td>
<td>cmp</td>
<td>L2 repeater</td>
</tr>
<tr>
<td>sctag_pcm_rptr_2</td>
<td>Repeater</td>
<td>1</td>
<td>sctag_pcm_rptr_2</td>
<td>cmp</td>
<td>L2 repeater</td>
</tr>
</tbody>
</table>
2.3 Megacells

The OpenSPARC T1 design contains many megacells, which are custom blocks for static random access memory (SRAMs), translation lookaside buffer (TLB), TAGs, Level 2 Cache, and so on. These megacells are instantiated in the top-level clusters. The detailed descriptions of all megacells, including their function descriptions, I/O lists, block diagrams, and timing diagrams, are in the OpenSPARC T1 Megacell Specification.
2.4 **External Interfaces**

The OpenSPARC T1 processor has the following external interfaces:

- Four DDR-II interfaces
- J-Bus
- SSI - Serial System Interface
- JTAG - IEEE 1149.1 interface
- System control
- Test and debug
- Debug port

The block diagram of external interfaces is shown in **FIGURE 2-2**.

**FIGURE 2-2**  OpenSPARC T1 External Interfaces

- **System control**: 26 pins
- **SSI**: 4 pins
- **Test and debug**: 13 pins
- **Debug port**: 46 pins
- **JTAG signals**: 5 pins
- **J-Bus interface**: 120-200 MHz
- **DDR-II 400 [2]** SDRAM interface 400 MT/s 214 pins
- **DDR-II 400 [3]** SDRAM interface 400 MT/s 214 pins
- **DDR-II 400 [0]** SDRAM interface 400 MT/s 214 pins
- **DDR-II 400 [1]** SDRAM interface 400 MT/s 214 pins
- **J- Bus interface**: 120-200 MHz 161 pins

**Total: 1111 Signal Pins**
CHAPTER 3

OpenSPARC T1 Verification Environment

This chapter describes the following topics:
- OpenSPARC T1 Verification Environment
- Running a Regression
- Verification Code
- PLI Code Used For the Test Bench
- Verification Test File Locations
- Compiling Source Code for Tools
- Gate-Level Verification

3.1 OpenSPARC T1 Verification Environment

The OpenSPARC T1 verification environment is a highly automated environment. With a simple command, you can run the entire regression suite for the OpenSPARC T1 processor, containing thousands of tests. With a second command, you can check the results of the regression.

The OpenSPARC T1 Design and Verification package comes with two test bench environments: core1 and chip8.

The core1 environment consists of:
- One SPARC CPU core
- Cache
- Memory
- Crossbar

The core1 environment does not have an I/O subsystem.
The chip8 environment consists of:

- A full OpenSPARC T1 chip, including all eight cores
- Cache
- Memory
- Crossbar
- I/O subsystem

OpenSPARC T1 Release 1.4 includes a third regression environment for single-thread implementation of the OpenSPARC T1 core. This regression environment has all the components present in core1 except that it only supports one hardware thread and removes the Stream Processing Unit (SPU). This implementation is primarily developed to create a core with a foot-print amenable for the FPGA map. You can add back SPU into the design by disabling FPGA_SYN_NO_SPU flag during design compile time.

The verification environment uses source code in various languages. TABLE 3-1 shows a summary of the types of source code and their uses.

**TABLE 3-1** Source Code Types in the Verification Environment

<table>
<thead>
<tr>
<th>Source Code Language</th>
<th>Used for:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verilog</td>
<td>Chip design, test bench drivers, and monitors.</td>
</tr>
<tr>
<td>Vera</td>
<td>Test bench drivers, monitors, and coverage objects. Use of Vera is optional.</td>
</tr>
<tr>
<td>PERL</td>
<td>Scripts for running simulations and regressions.</td>
</tr>
<tr>
<td>C and C++</td>
<td>PLI (Programming Language Interface) for Verilog.</td>
</tr>
<tr>
<td>SPARC Assembly</td>
<td>Verification tests.</td>
</tr>
</tbody>
</table>
The block diagram for the verification test bench is in FIGURE 3-1.

FIGURE 3-1  OpenSPARC T1 Verification Test Bench Overview

The top-level module for the test bench is called cmp_top. The same test bench is used for both the core1 and chip8 environments with compile-time options.

3.2 Running a Regression

For each environment, there is a mini-regression and a full regression. TABLE 3-2 describes the regression groups.

TABLE 3-2  Details of Regression Groups

<table>
<thead>
<tr>
<th>Regression Group name</th>
<th>Environment</th>
<th>No. of Tests</th>
<th>Disk space needed to run (Mbyte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread1_mini</td>
<td>thread1</td>
<td>42</td>
<td>25</td>
</tr>
<tr>
<td>thread1_full</td>
<td>thread1</td>
<td>605</td>
<td>900</td>
</tr>
<tr>
<td>core1_mini</td>
<td>core1</td>
<td>68</td>
<td>41</td>
</tr>
</tbody>
</table>
3.2.1 To Run a Regression

1. Run the sims command with your chosen parameters.

   For instance, to run a mini-regression for the core1 environment using the VCS simulator, set up the sims command as follows:

   ```
   % sims -sim_type=vcs -group=core1_mini
   ```

   To run regressions on multiple groups at the same time, specify multiple -group= parameters at the same time. For a complete list of command-line options for the sims command, see Appendix A.

2. Run the regreport command to get a summary of the regression.

   ```
   % regreport $PWD/2006_01_25_0 > report.log
   ```

3.2.2 What the sims Command Does

When running a simulation, the sims command performs the following steps:

1. Compiles the design into the $MODEL_DIR/core1 or $MODEL_DIR/chip8 directory, depending on which environment is being used.

2. Creates a directory for regression called $PWD/DATE_ID, where $PWD is your current directory, DATE is in YYYY_MM_DD format, and ID is a serial number starting with 0. For example, for the first regression on Jan 25, 2006, a directory called $PWD/2006_01_26_0 is created. For the second regression run on the same day, the last ID is incremented to become $PWD/2006_01_26_1.

3. Creates a master_diaglist.regression_group file under the above directory, such as master_diaglist.core1_mini for the core1_mini regression group. This file is created based on diaglists under the $DV_ROOT/verif/diag directory.

### TABLE 3-2 Details of Regression Groups

<table>
<thead>
<tr>
<th>Regression Group name</th>
<th>Environment</th>
<th>No. of Tests</th>
<th>Disk space needed to run (Mbyte)</th>
</tr>
</thead>
<tbody>
<tr>
<td>core1_full</td>
<td>core1</td>
<td>900</td>
<td>1,680</td>
</tr>
<tr>
<td>chip8_mini</td>
<td>chip8</td>
<td>492</td>
<td>1,517</td>
</tr>
<tr>
<td>chip8_full</td>
<td>chip8</td>
<td>3789</td>
<td>29,000</td>
</tr>
</tbody>
</table>
4. Creates a subdirectory with the test name under the regression directory created in step 2 above.

5. Creates a sim_command file for the test based on the parameters in the diaglist file for the group.

6. Executes sim_command to run a Verilog simulation for the test. If the -sas option is specified for the test, it also runs the SPARC Architecture Simulator (SAS) in parallel with the Verilog simulator. The results of the Verilog simulation are compared with the SAS results after each instruction.

The sim_command command creates many files in the test directory. Following are the sample files in the test directory:

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>diag.ev</td>
<td></td>
</tr>
<tr>
<td>diag.s</td>
<td></td>
</tr>
<tr>
<td>l2way.log</td>
<td></td>
</tr>
<tr>
<td>perf.log</td>
<td></td>
</tr>
<tr>
<td>sas.log.gz</td>
<td></td>
</tr>
<tr>
<td>sims.log</td>
<td></td>
</tr>
<tr>
<td>symbol.tbl</td>
<td></td>
</tr>
<tr>
<td>sim.perf.log</td>
<td></td>
</tr>
<tr>
<td>diag.exe.gz</td>
<td></td>
</tr>
<tr>
<td>efuse.img</td>
<td></td>
</tr>
<tr>
<td>midas.log</td>
<td></td>
</tr>
<tr>
<td>sim_command</td>
<td></td>
</tr>
<tr>
<td>status.log</td>
<td></td>
</tr>
<tr>
<td>sim.log.gz</td>
<td></td>
</tr>
</tbody>
</table>

The status.log file has a summary of the status, where the first line contains the name of the test and its status (PASS/FAIL).

7. Repeats steps 4 to 6 for each test in the regression group.

### 3.2.3 Running Regression With Other Simulators

To use a Verilog simulator other than VCS or NCVerilog, use following options for the sims command:

- `-sim_type="Your simulator name"`
- `-sim_build_cmd="Your simulator command to build/compile RTL"
- `-sim_run_cmd="Your simulator command to run simulations"
- `-sim_build_args="Arguments to build/compile"
- `-sim_run_args="Arguments to run simulations"

You only need to specify the sim_type, sim_build_cmd, and sim_run_cmd options once. You can specify sim_build_args and sim_run_args multiple times to specify multiple argument options.
3.3 Verification Code

This section outlines Verilog and Vera code structures and locations.

3.3.1 Verilog Code Used for Verification

There are various test bench drivers and monitors written in Verilog. A list of all Verilog modules, the location of the source code, and descriptions is in TABLE 3-3. All verification Verilog files are in the $DV_ROOT/verif/env directory.

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Type</th>
<th>Number of instances</th>
<th>Instance Names</th>
<th>Directory Location under $DV_ROOT/verif/env</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenSPARCT1</td>
<td>Chip</td>
<td>1</td>
<td>iop</td>
<td>$DV_ROOT/design/sys/iop/rtl</td>
<td>OpenSPARC T1 top level</td>
</tr>
<tr>
<td>bw_sys</td>
<td>Driver</td>
<td>1</td>
<td>bw_sys</td>
<td>cmp</td>
<td>SSI bus driver</td>
</tr>
<tr>
<td>cmp_clk</td>
<td>Driver</td>
<td>1</td>
<td>cmp_clk</td>
<td>cmp</td>
<td>Clock driver</td>
</tr>
<tr>
<td>cmp_dram</td>
<td>Model</td>
<td>1</td>
<td>cmp_dram</td>
<td>cmp</td>
<td>DRAM modules</td>
</tr>
<tr>
<td>cmp_mem</td>
<td>Driver</td>
<td>1</td>
<td>cmp_mem</td>
<td>cmp</td>
<td>Memory tasks</td>
</tr>
<tr>
<td>cpx_stall</td>
<td>Driver</td>
<td>1</td>
<td>cpx_stall</td>
<td>cmp</td>
<td>CPX stall</td>
</tr>
<tr>
<td>dbg_port_chk</td>
<td>Monitor</td>
<td>1</td>
<td>dbg_port_chk</td>
<td>cmp</td>
<td>Debug port checker</td>
</tr>
<tr>
<td>dffrl_async</td>
<td>Driver</td>
<td>4</td>
<td>flop_ddr[0-3]_oe</td>
<td>$DV_ROOT/design/sys/iop/common/rtl</td>
<td>Flip-flop</td>
</tr>
<tr>
<td>err_inject</td>
<td>Driver</td>
<td>1</td>
<td>err_inject</td>
<td>cmp</td>
<td>Error Injector</td>
</tr>
<tr>
<td>jbus_monitor</td>
<td>Monitor</td>
<td>1</td>
<td>jbus_monitor</td>
<td>iss/pli/jbus_mon/rtl</td>
<td>J-Bus Monitor</td>
</tr>
<tr>
<td>jp_sjm</td>
<td>Driver</td>
<td>2</td>
<td>j_sjm_4, j_sjm_5</td>
<td>iss/pli/sjm/rtl</td>
<td>J-Bus Driver</td>
</tr>
<tr>
<td>monitor</td>
<td>Monitor</td>
<td>1</td>
<td>monitor</td>
<td>cmp</td>
<td>Various monitors</td>
</tr>
<tr>
<td>one_hot_mux_mon</td>
<td>Monitor</td>
<td>1</td>
<td>one_hot_mux_mon</td>
<td>cmp</td>
<td>Hot mux monitor</td>
</tr>
<tr>
<td>pcx_stall</td>
<td>Driver</td>
<td>1</td>
<td>pcx_stall</td>
<td>cmp</td>
<td>PCX stall</td>
</tr>
<tr>
<td>sas_intf</td>
<td>SAS</td>
<td>1</td>
<td>sas_intf</td>
<td>cmp</td>
<td>SAS interface</td>
</tr>
<tr>
<td>sas_tasks</td>
<td>SAS</td>
<td>1</td>
<td>sas_tasks</td>
<td>cmp</td>
<td>SAS tasks</td>
</tr>
</tbody>
</table>
3.3.2 Vera Code Used for Verification

Two types of Vera code are included in the OpenSPARC T1 verification environment:
- Test bench driver and Monitor Vera code
- Vera Object coverage Vera code

Vera code is in the $DV_ROOT/verif/env/cmp/vera directory. Each object coverage module has a corresponding subdirectory. Following is a list of Vera object coverage modules:

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Type</th>
<th>Number of instances</th>
<th>Instance Names</th>
<th>Directory Location under $DV_ROOT/verif/env</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>slam_init</td>
<td>Driver</td>
<td>1</td>
<td>slam_init</td>
<td>cmp</td>
<td>Initialization tasks</td>
</tr>
<tr>
<td>sparc_pipe_flow</td>
<td>Monitor</td>
<td>1</td>
<td>sparc_pipe_flow</td>
<td>cmp</td>
<td>SPARC pipe flow monitor</td>
</tr>
<tr>
<td>tap_stub</td>
<td>Driver</td>
<td>1</td>
<td>tap_stub</td>
<td>cmp</td>
<td>JTAG driver</td>
</tr>
</tbody>
</table>

TABLE 3-3  OpenSPARC T1 Verification Test Bench Modules (Continued)

Object coverage Vera code for jbi is in the
$DV_ROOT/verif/env/iss/vera/jbi_coverage directory. Object coverage Vera code is only used for the chip8_cov regression groups.
3.4 PLI Code Used For the Test Bench

Verilog’s PLI (Programming Language Interface) is used to drive and monitor the simulations of the OpenSPARC T1 design. There are eight different directories for PLI source code. Some PLI code is in C language, and some is in C++ language. The object libraries for the VCS simulator and NC-Verilog simulator are included for the PLI code in the $DV_ROOT/tools/SunOS/sparc/lib directory. TABLE 3-4 gives the details of PLI code directories, VCS libraries, and NC-Verilog libraries.

### TABLE 3-4 PLI Source Code and Object Libraries

<table>
<thead>
<tr>
<th>PLI name</th>
<th>Source code location under $DV_ROOT</th>
<th>VCS object library name</th>
<th>NC-Verilog object library name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iop</td>
<td>tools/pli/iop</td>
<td>libiob.a</td>
<td>libiob_ncv.so</td>
<td>Monitors and drivers</td>
</tr>
<tr>
<td>mem</td>
<td>tools/pli/mem</td>
<td>libmem_pli.a</td>
<td>libmem_pli_ncv.so</td>
<td>Memory read/write</td>
</tr>
<tr>
<td>socket</td>
<td>tools/pli/socket</td>
<td>libsocket_pli.a</td>
<td>libsocket_pli_ncv.so</td>
<td>Sockets to SAS</td>
</tr>
<tr>
<td>utility</td>
<td>tools/pli/utility</td>
<td>libutility_pli.a</td>
<td>libutility_ncv.so</td>
<td>Utility functions</td>
</tr>
<tr>
<td>common</td>
<td>verif/env/iss/pli/common/c</td>
<td>libjpcommon.a</td>
<td>libjpcommon_ncv.so</td>
<td>Common PLI functions</td>
</tr>
<tr>
<td>jbus_mon</td>
<td>verif/env/iss/pli/jbus_mon/c</td>
<td>libjbus_mon.a</td>
<td>libjbus_mon_ncv.so</td>
<td>J-Bus Monitor</td>
</tr>
<tr>
<td>monitor</td>
<td>verif/env/iss/pli/monitor/c</td>
<td>libmonitor.a</td>
<td>libmonitor_ncv.so</td>
<td>Various</td>
</tr>
<tr>
<td>sjm</td>
<td>verif/env/iss/pli/sjm/c</td>
<td>libsjm.a</td>
<td>libsjm_ncv.so</td>
<td>J-Bus Driver</td>
</tr>
</tbody>
</table>

VCS object libraries are statically linked libraries (.a files) which are linked when VCS compiles the Verilog code to generate a simv executable. NC-Verilog object libraries are dynamically loadable libraries (.so files) which are linked dynamically while running the simulations.

Makefiles are provided to compile PLI code. There is a makefile file under each PLI directory to create a static object library (.a file). There is a makefile.ncv file under each PLI directory to create a dynamic object library.
3.4.1 To Compile All PLI Libraries

To compile all PLI libraries, run the mkplilib script. This script has three options as listed in **TABLE 3-5**.

**TABLE 3-5** Options for the mkplilib Script

<table>
<thead>
<tr>
<th>Option</th>
<th>Used for</th>
</tr>
</thead>
<tbody>
<tr>
<td>vcs</td>
<td>Compiling PLI libraries for VCS</td>
</tr>
<tr>
<td>ncverilog</td>
<td>Compiling PLI libraries for NC-Verilog</td>
</tr>
<tr>
<td>clean</td>
<td>Deleting all PLI libraries</td>
</tr>
</tbody>
</table>

- Compile PLI libraries with your chosen option.
  
  For example, to compile PLI libraries in VCS, type the following:
  
  ```
  % mkplilib vcs
  ```

  Either version of this procedure, VCS or NC_Verilog, compiles C/C++ code, creates static or dynamic libraries, and copies them to the
  `$DV_ROOT/tools/SunOS/sparc/lib` directory.

3.5 Verification Test File Locations

The verification or diagnostics tests (diags) for the OpenSPARC T1 processor are written in SPARC assembly language (the file names have a `.s` extension). Some diags require command files for a J-Bus Driver. Those command files are named sjm_4.cmd and sjm_5.cmd. Some diagnostics test cases in SPARC assembly are automatically generated by Perl scripts.
The main diaglist for core1 is core1.diaglist. The main diaglist for chip8 is chip8.diaglist. These main diaglists for each environment also include many other diaglists. The locations of various verification test files are listed in TABLE 3-6.

<table>
<thead>
<tr>
<th>Directory</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>$DV_ROOT/verif/diag</td>
<td>All diagnostics, various diagnostic list files with the extension diaglist.</td>
</tr>
<tr>
<td>$DV_ROOT/verif/diag/efuse</td>
<td>EFuse cluster default memory load files.</td>
</tr>
</tbody>
</table>

### 3.6 Compiling Source Code for Tools

To compile source code for some Sun tools used for the OpenSPARC T1 processor, use the mktools script. The tools source code is located in the $DV_ROOT/tools/src directory.

The mktools script compiles the source code and copies the binaries to $DV_ROOT/tools/<Operating System>/<Processor Type> directory, where:

- <Operating System> is defined by the `uname -s` command
- <Processor Type> is defined by the `uname -p` command

### 3.7 Gate-Level Verification

OpenSPARCT1 depends heavily on Cross-Module References (XMRs) within the verification environment. Therefore, dropping in a `netlist` in place of the RTL core will produce a high number of XMR errors. In order to overcome this difficulty, a simple playback support is now added.

Although we anticipate this method to be useful primarily to verify FPGA synthesized netlists, it could be potentially used with netlists generated by semi-custom synthesis flows as well (for example, Synopsys).
**Caution** – Running this vector playback mechanism on RTL, although feasible, is not recommended due to some array initialization issues. In the gate playback mode, all arrays are explicitly initialized to zero while in RTL and some arrays are initialized to random values. This may result in mismatch in playback simulation. If RTL arrays are initialized correctly (zeros) then this mechanism can be used to verify RTL netlist as well.

To verify a netlist, do the following:

1. **Run RTL mini or full regression to generate stimuli files for netlist verification.**

   To do this, add the `-vcs_build_args=` $DV_ROOT/verif/env/cmp/playback_dump.v option to the regression command.

   For example, thread1_mini regression command for the SPARC level driver (stimuli) generation would require the following:

   ```
   % sims -sim_type=vcs -group=thread1_mini -debussy \ 
   -vcs_build_args=$DV_ROOT/verif/env/cmp/playback_dump.v
   %
   $DV_ROOT/verif/gatesim/build_gates <flist>
   ```

   The above regression will generate stimuli.txt file under the run directory of each diagnostic. Sample stimuli.txt files are included under $DV_ROOT/verif/gatesim for the thread1_mini regression (file thread1_mini_stim.tar.gz). These files are generated with VCS build args (-vcs_build_args) FPGA_SYN, FPGA_SYN_1THREAD, FPGA_SYN_8TLB and FPGA_SYN_NO_SPU flags.

2. **Create a verilog file list which includes the following files:**

   $DV_ROOT/verif/env/cmp/playback_driver.v
   <SPARC level gate netlist>.v
   <library used for synthesis>.v

   Sample flist is provided under $DV_ROOT/verif/gatesim for reference (file flist.xilinx_unisims)

3. **Compile the design to build the gate level model.**

   A sample compile script is provided under the $DV_ROOT/verif/gatesim directory. The following shows usage of the compile script:

   ```
   % $DV_ROOT/verif/gatesim/build_gates <flist>
   ```
4. Run the simulation by including `+stim_file=<path to stim file>/stimuli.txt`

   If playback fails, the simulation will return with “Playback FAILED with # mismatches!” If it passes, it will return with “Playback PASSED!”. Use fsdb generation options in the compile script to debug failing runs.

   A simple run_gates script is also included for reference under the `$DV_ROOT/verif/gatesim` directory. The following shows usage of the run script:

   ```bash
   % $DV_ROOT/verif/gatesim/run_gates <path to stim file>
   ```

### TABLE 3-7 Gate Netlist Files

<table>
<thead>
<tr>
<th>Directory/File</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>$DV_ROOT/verif/gatesim/build_gates</code></td>
<td>Compile script to create gate level model of SPARC netlist</td>
</tr>
<tr>
<td><code>$DV_ROOT/verif/gatesim/run_gates</code></td>
<td>Run script to execute playback of vectors on gate netlist</td>
</tr>
<tr>
<td><code>$DV_ROOT/verif/gatesim/flist.xilinx.unisims</code></td>
<td>Sample verilog file list with Xilinx synthesis library</td>
</tr>
<tr>
<td><code>$DV_ROOT/verif/gatesim/thread1_mini_stim.tar.gz</code></td>
<td>FOR REFERENCE ONLY: Collection pre-packaged stimulus files for thread1_mini regression suit.</td>
</tr>
</tbody>
</table>
OpenSPARC T1 Synthesis

This chapter describes the following topics:

- Synthesis Flow for the OpenSPARC T1 Processor
- Synthesis Output

The scripts provided in the source code are for the Synopsys Design Compiler.

4.1 Synthesis Flow for the OpenSPARC T1 Processor

There are two types of synthesis scripts:

- One set to run the Synopsys Design Compiler (rsyn and syn_command)
- One set used as input for the Design Compiler

The main script used to run Synopsys Design Compiler is called rsyn. This is a PERL script that calls a second script, syn_command, once for each module you are synthesizing. The command-line options for the rsyn script are described in CODE EXAMPLE 4-1.
Synthesis scripts for most of the modules are provided in the $DV_ROOT/design sub-directories. There are no synthesis scripts for the following types of modules:

- Megacell modules (SRAMS, TLB, TAG, Cache, etc.)
- Top-level hierarchical modules

Synopsys scripts, their locations, and their descriptions are listed in TABLE 4-1.

### CODE EXAMPLE 4-1  Command-Line Options for rsyn Script

```bash
rsyn : Run Synthesis for OpenSPARC T1

- all
  to run synthesis for all blocks
- h / -help
  to print help
- syn_q_command='Your job Queue command'
  to specify Job queue command. e.g. specify submit command for LSF or GRID
block_list :
  specify list of blocks to synthesize

Examples:
rsyn -all
rsyn fpu_add
```

### TABLE 4-1  Synthesis Script Details

<table>
<thead>
<tr>
<th>Script name</th>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>run.scr</td>
<td>$DV_ROOT/design/sys/synopsys/script</td>
<td></td>
</tr>
<tr>
<td>project_sparc_cfg.scr</td>
<td>$DV_ROOT/design/sys/synopsys/script</td>
<td>SPARC module-specific synthesis script</td>
</tr>
<tr>
<td>project_io_cfg.scr</td>
<td>$DV_ROOT/design/sys/synopsys/script</td>
<td>I/O module-specific synthesis script</td>
</tr>
<tr>
<td>target_lib.scr</td>
<td>$DV_ROOT/design/sys/synopsys/script</td>
<td>Target library-specific script</td>
</tr>
<tr>
<td>user_cfg.scr</td>
<td>Module directory/synopsys/script</td>
<td>Module-specific synthesis script</td>
</tr>
</tbody>
</table>

The top-level Synopsys script, run.scr, calls the module-specific script named user_cfg.scr. The user_cfg.scr script calls the project_sparc_cfg.scr script or the project_io_cfg.scr script, depending on whether the module belongs to sparc or io.
The list of all modules with synthesis scripts is in the $DV_ROOT/design/sys/synopsys/block.list file. Each module has:

- A synopsys directory under the module directory
- A script directory under each synopsys directory
- The user_cfg.scr file under the script directory

For example, the efc module-specific synthesis script has the following directory path:

```
$DV_ROOT/design/sys/iop/efc/synopsys/script/user_cfg.scr
```

The target library is set to a generic library called lsi_10k.db in the target_lib.scr script. Modify this file to set your own target library and its required variables.

### 4.2 Synthesis Output

Running synthesis for a module creates files and directories under the Module name/synopsys directory, described in TABLE 4-2.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dc_shell.log</td>
<td>File</td>
<td>Log file from running Design Compiler</td>
</tr>
<tr>
<td>command.log</td>
<td>File</td>
<td>Command log from running Design Compiler</td>
</tr>
<tr>
<td>log</td>
<td>Directory</td>
<td>Area report files from Design Compiler</td>
</tr>
<tr>
<td>gate</td>
<td>Directory</td>
<td>Gate netlist generated by Design Compiler</td>
</tr>
<tr>
<td>.template</td>
<td>Directory</td>
<td>Template directory used by Design Compiler</td>
</tr>
</tbody>
</table>
OpenSPARC T1 FPGA Synthesis

This chapter describes the following topics:

- Synplicity FPGA Synthesis Flow for the OpenSPARC T1 Processor
- Synplicity FPGA Synthesis Output
- XST Synthesis Flow for the OpenSPARC T1 Processor
- XST Synthesis Output
- Selecting OpenSPARC T1 Options for Reduced Size

The scripts provided in the OpenSPARC T1 source code are for the Synplicity Pro software, version 8.5, and for the Xilinx Synthesis Technology (XST) Software, version 9.1.

5.1 Synplicity FPGA Synthesis Flow for the OpenSPARC T1 Processor

Several scripts are required to run FPGA synthesis with Synplicity

- Shell scripts to run the Synplicity Pro software (rsynp and synp_command)
- Synplicity scripts which are read by the Synplicity Pro software

The main script used to run the Synplicity software is a PERL script called rsynp. rsynp calls a second script, synp_command, once for each module you are synthesizing into an FPGA. CODE EXAMPLE 5-1 describes the command-line options of the rsynp script.
The OpenSPARC T1 source code provides FPGA synthesis scripts for the following modules: sparc, fpu, and ccx. The rsynp script first creates a module-specific script proj.prj in the <module-directory>/synplicity directory, and then it calls the Synplicity software in batch mode using the proj.prj script as input. TABLE 5-1 lists the FPGA Synplicity scripts, their locations, and their descriptions. The proj.prj script uses all the scripts listed in this table.

**TABLE 5-1 Synplicity FPGA Synthesis Script Details**

<table>
<thead>
<tr>
<th>Script name</th>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>block.list</td>
<td>$DV_ROOT/design/sys/synplicity</td>
<td>List of blocks with synthesis scripts</td>
</tr>
</tbody>
</table>

**CODE EXAMPLE 5-1 Command-Line Options for the rsynp Script**

```
rsynp : Run Synplicity for OpenSPARC T1

Options are :
 -all
to run Synplicity for all blocks
-h / -help
to print help
-syn_q_command='Your job Queue command'
to specify Job queue command
-device='Target Device'
to specify Target FPGA device
-flat
To run synthesis flat, must use this for Altera parts.
-clean
To remove all unneeded files and/or directories.
Need to specify target device when not using default device

block_list :
specify list of blocks to synthesize

Examples:
rsynp -all
rsynp -device=XC4VLX200 sparc
rsynp -flat -device=EP2S180 sparc
```
The `$DV_ROOT/design/sys/synplicity/block.list` file lists all of the modules that can be synthesized using the Synplicity software. Each module has:

- A `synplicity` directory under the module (`<module>`) directory.
- A `<module>.flist` file under each `<module>/synplicity` directory. (This file lists the Verilog files for the module.)
- Optional `<module>.mlist` file under `<module>/synplicity` directory, this is the list of SRAM modules for the module. SRAM modules are synthesized hierarchically.
- Optional `<module>.fmlist` file under `<module>/synplicity` directory, this is the flat list of SRAM Verilog files.
- Optional `<module>.prj` file under `<module>/synplicity` directory, this is the file for module specific Synplicity settings.
- Optional `<module>.sed` file to change names of the modules to make them unique module names in the Synplicity-generated output design `.vm` file.

### Table 5-1 Synplicity FPGA Synthesis Script Details (Continued)

<table>
<thead>
<tr>
<th>Script name</th>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre_syn_settings.prj</td>
<td><code>$DV_ROOT/design/sys/synplicity</code></td>
<td>Synplicity software settings</td>
</tr>
<tr>
<td>env.prj</td>
<td><code>$DV_ROOT/design/sys/synplicity</code></td>
<td>OpenSPARC environment-related settings</td>
</tr>
<tr>
<td><code>&lt;device&gt;.prj</code></td>
<td><code>$DV_ROOT/design/sys/synplicity</code></td>
<td>Target device-specific settings (for example, the XC4VLX200.prj script would be for the Xilinx XC4VLX200 device)</td>
</tr>
</tbody>
</table>
5.2 Synplicity FPGA Synthesis Output

While running a FPGA synthesis for a module, the Synplicity software will create files and directories under the \texttt{<module>/synplicity} directory. TABLE 5-2 describes these files and directories.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{&lt;device&gt;}</td>
<td>Directory</td>
<td>Target device-specific directory (for example, \texttt{XC4VLX200} would be the directory name for the Xilinx XC4VLX200 device). The Synplicity software will create a number of files and subdirectories under this \texttt{&lt;device&gt;} directory.</td>
</tr>
<tr>
<td>\texttt{&lt;device&gt;}/\texttt{&lt;module&gt;}.srr</td>
<td>File</td>
<td>Synplicity software output file for a \texttt{&lt;module&gt;} using the \texttt{&lt;device&gt;} as a target FPGA. This file is a log of the synthesis process, and contains information about the estimated timing, area, and so on.</td>
</tr>
<tr>
<td>\texttt{&lt;device&gt;}/\texttt{&lt;module&gt;}.edf</td>
<td>File</td>
<td>EDIF netlist for the synthesized block.</td>
</tr>
</tbody>
</table>

5.3 XST Synthesis Flow for the OpenSPARC T1 Processor

Scripts are now available to allow automated synthesis of the OpenSPARC T1 using Xilinx Synthesis Technology (XST).

The main script used to run the XST software is a PERL script called \texttt{rxil}.

The \texttt{rxil} script calls XST once for each module that is being synthesized into an FPGA. CODE EXAMPLE 5-2 describes the command-line options of the \texttt{rxil} script.
CODE EXAMPLE 5-2  Command-Line Options for the rxil Script

```latex
\begin{verbatim}
rxil : Run XST for OpenSPARC T1
Options are :
  -all
to run XST for all blocks
  -h / -help
to print help
  -device='Target Device'
to specify Target FPGA device
block_list :
specify list of blocks to synthesize
Examples:
  rxil -all
  rxil -device=XC4VLX200 sparc
  rxil -device=XC4VLX200 fpu ccx
\end{verbatim}
```

The OpenSPARC T1 source code provides XST synthesis scripts for the following modules: sparc, fpu, and ccx. The rxil script first copies a device file for the target device into the `<module-directory>/xst` directory, creates a `<device>` subdirectory, and then it calls the XST software in batch mode using the device file and the `<block>.flist` file as input.

<table>
<thead>
<tr>
<th>Table 5-3</th>
<th>XST Synthesis Script Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Script Name</td>
<td>Type</td>
</tr>
<tr>
<td>block.list</td>
<td>$DV_ROOT/design/sys/xst</td>
</tr>
<tr>
<td>XC4VFX100.xst</td>
<td>$DV_ROOT/design/sys/xst</td>
</tr>
<tr>
<td>xst_defines.h</td>
<td>$DV_ROOT/design/sys/iop/include</td>
</tr>
</tbody>
</table>

The `$DV_ROOT/design/sys/xst/block.list` file lists all of the modules that can be synthesized using the XST software. Each module has:

- A `xst` directory under the module `<module>` directory.
- A `<module>.flist` file under each `<module>/xst` directory. (This file lists the Verilog files for the module.)
5.4 XST Synthesis Output

While running a FPGA synthesis for a module, the XST software will create files and directories under the `<module>/xst` directory. TABLE 5-4 describes these files and directories.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;device&gt;</code></td>
<td>Directory</td>
<td>Target device specific directory (for example, XC4VLX200 would be the directory name for the Xilinx XC4VLX200 device). The XST software will create a number of files and subdirectories under this <code>&lt;device&gt;</code> directory.</td>
</tr>
<tr>
<td><code>&lt;device&gt;</code>/&lt;block&gt;.ngc</td>
<td>File</td>
<td>Synthesized block netlist in Xilinx NGC format.</td>
</tr>
<tr>
<td><code>&lt;device&gt;</code>/&lt;block&gt;.v</td>
<td>File</td>
<td>Synthesized block netlist in Verilog format.</td>
</tr>
</tbody>
</table>

5.5 Selecting OpenSPARC T1 Options for Reduced Size

OpenSPARC T1 RTL now includes four conditional compile options in the design. Each of these options create a different variant of the OpenSPARC T1 core. Although these four options are orthogonal to each other in terms of what change it brings to the design, when combined together, they produce a significantly smaller design and a very compelling solution for FPGA mapping. The following describes these options:

- **FPGA_SYN**: This option enables different implementation for some of the megacells and SRAM arrays to make it more effectively utilize FPGA resources like Block RAMs and multipliers. This option also removes all asynchronous logic (for example, latches) from the design to make it amiable for FPGA synthesis. FPGA synthesis will not complete without this option.

- **FPGA_SYN_1THREAD**: This option reduces multi-threaded overhead of logic and creates a small and clean single thread implementation of OpenSPARC T1 core. The basic functionality of the SPARC core here is the same.
FPGA_SYN_NO_SPU: The Stream Processing Unit (SPU) in OpenSPARC T1 provides hardware acceleration for cryptographic functions. In designing general purpose FPGA based processor, this unit can be safely removed without affecting the base functionality. This option removes SPU from the design and provides further reduction in the design size.

FPGA_SYN_8TLB: The OpenSPARC T1 TLB normally has 64 entries. To save space, a reduced-size TLB of 8 entries may be selected. The function of the TLB remains the same. It just has fewer entries.

Based on our experiments, combining all four options create a design that would consume about 40,000 4-input Look-Up Tables (LUT) in the Xilinx Virtex-4 family.

To run the FPGA synthesis with Synplicity Simplify Pro,

1. Add the following line in the file `$DV_ROOT/design/sys/synplicity/env.prj`

   ```
   # Set +defines
   set_option -hdl_define -set "FPGA_SYN FPGA_SYN_NO_SPU FPGA_SYN_1THREAD FPGA_SYN_8TLB"
   ```

2. Then run rsynp.

   For rsynp script details, see Section 5.1, “Synplicity FPGA Synthesis Flow for the OpenSPARC T1 Processor” on page 5-1.

To run the FPGA synthesis with Xilinx XST,

1. Add the following lines in the file `$DV_ROOT/design/sys/iop/include/xst_defines.h`

   ```
   `define FPGA_SYN
   `define FPGA_SYN_NO_SPU
   `define FPGA_SYN_1THREAD
   `define FPGA_SYN_8TLB
   ```

2. Then run rxil. For rxil script details, see Section 5.3, “XST Synthesis Flow for the OpenSPARC T1 Processor” on page 5-4

Caution – The four options described above should be used only with the thread1_mini and thread1_full regression environments. Attempting to include these options in core1 regressions will cause the diagnostics to fail. core1 regression includes tests that verify multi-threaded and SPU related functionality which does not exist when the model is built with the above options.
OpenSPARC T1 EDK Project

This chapter describes the following topics:

- System Description of the EDK project
- Generation of a bit file for a Xilinx FPGA
- Running OpenSPARC T1 diagnostics on an evaluation board
- Running a standalone program on the OpenSPARC T1 core on an evaluation board
- Running system-level simulation with Modelsim

6.1 System Description of the EDK project

A Xilinx Embedded Development Kit (EDK) project is provided to provide a platform for running the OpenSPARC T1 core on a Xilinx ML410 or ML411 evaluation board. The Xilinx EDK environment allows the user to quickly put a new system together, place and route the design on a target FPGA, then download the design to the target FPGA to test and debug it.

A block diagram of the system provided in the EDK project is shown in FIGURE 6-1.

The system contains the following components.

- OpenSPARC T1 core
- An Adapter block which adapts the cache crossbar interface of the OpenSPARC T1 core to the MicroBlaze FSL interface.
- A MicroBlaze controller which will run software to service all memory and I/O requests coming from the core. The MicroBlaze core contains:
  - 32k I-cache
  - 32k D-cache
6.1.1 Hardware Operation

The EDK system was designed for minimal overhead on top of the core. This makes it possible to fit the core into a fairly small FPGA. The system operates as follows:

Memory accesses from the core (instruction fetches, loads, and stores) are sent over the 124-bit processor-to-cache crossbar (PCX) interface. Interrupts are also communicated through this interface. The ccx2mb adapter block breaks these 124-bit requests into four 32-bit words and places those words into an FSL FIFO. Firmware running on the MicroBlaze core reads the requests out of the FSL FIFO, interprets them, and services them. Once the firmware has completed the request, it generates a 144-bit response packet, which is placed in a second FSL FIFO as 5 32-bit words. The ccx2mb block reads these words from the FIFO, reconstructs the 144-bit response, and forwards it on to the OpenSPARC T1 core.

6.1.2 MicroBlaze Firmware Operation

The firmware must perform the following functions:

- Interpret and service the memory requests coming from the OpenSPARC T1 core.
- Maintain a memory map which maps OpenSPARC T1 addresses to addresses in the MicroBlaze memory system.
- Maintain directories of the level 1 instruction and data caches of the OpenSPARC T1. This is required in order to keep these caches coherent.
- Send invalidations to the level 1 caches to keep them coherent.
- Send the initial wake-up interrupt to the OpenSPARC T1 after reset completes.
Communicate device interrupts to the OpenSPARC T1 core.

The firmware operates by continuously polling the FSL FIFO connected to the OpenSPARC T1 PCX interface. When there is a valid packet, the firmware decodes it and services the request. For any kind of memory transaction, the firmware must translate the OpenSPARC T1 address to the corresponding board address. Once the correct board address is obtained, the firmware can get data from or store data to memory. Then it generates the appropriate packets to send back to the OpenSPARC T1 core.

There are two variations of the firmware: One to run OpenSPARC T1 diagnostic tests, and one to run standalone programs under hypervisor. The only difference between these variations is the memory mapping of OpenSPARC T1 addresses to board addresses.

6.2 Generation of a bit file for a Xilinx FPGA

The OpenSPARC T1 EDK project should be ready to go. To generate a bitstream for a Xilinx FPGA, there is a simple two-step process:

1. Copy the correct OpenSPARC T1 netlist into the pcores directory if necessary.

   By default, there is a Synplicity netlist of a one-thread core in the netlist directory for the OpenSPARCT1. The provided netlist was synthesized with the following features:
   - Single Thread
   - 8-entry TLB
   - No stream processing unit (SPU)

   To use an XST netlist instead, the Synplicity netlist must be removed and the XST netlist placed in the netlist directory in its place. There can only be one netlist in the netlist directory. The text box shows how to replace the Synplicity netlist with the XST netlist:

   ```text
   % cd design/sys/edk/pcores/iop_fpga_v1_00_a
   % mv netlist/sparc.edf .
   % cp path-to-netlist/sparc.ngc netlist
   ```

   After the new netlist has been put in place, the file `data/iop_fpga_v2_1_0.bbd` must be edited to point to the new netlist.
2. Start Xilinx Platform Studio (XPS)
Xilinx Platform Studio will come up as shown in FIGURE 6-2.

FIGURE 6-2  Xilinx Platform Studio (XPS) window with OpenSPARC project open.

3. Select the menu option to generate a bitstream
Select the menu item:
Hardware --> Generate Bitstream
6.3 Running OpenSPARC T1 diagnostics on an evaluation board

Once a bitstream has been generated, it can be downloaded to the FPGA on the evaluation board, and SPARC diagnostic tests (called diags at Sun) can be run on the hardware.

The version of the firmware to run stand-alone diagnostics requires one source file, mbfw_diag_memimage.c, which is a C structure representation of the memory image for the test. To run any diag, the assembly language program must be assembled into an executable. Then the executable is converted to a binary file, and a perl script is run to convert the binary file into the C structure representation that is needed by the firmware. Then the firmware is re-compiled with the new C structure. The EDK project relies on the sims program to generate the executable programs, since it calls the assembler with all the proper options for each test.

The mb-firmware software project is set up to run the win_restore0 test. To run this test, perform the following procedure:

6.3.1 Running the default diag: win_restore0

1. Select the Applications tab in the Project Information Area (the left-hand window). A list of software applications will be shown.

2. Right-click on the microblaze_0_bootloop application and select “Mark to initialize BRAMs.”

   This will select the microblaze_0_bootloop program to be automatically downloaded to BRAMs with the bitstream. This will keep the MicroBlaze processor safely looping at address 0 until the mb-firmware program is downloaded to the DRAM. Make sure no other program is marked to initialize BRAMs.

3. Select the following menu item to update the bitstream.

   Select menu Device Configuration --> Update Bitstream

4. Download the bitstream to the FPGA.

   Select menu Device Configuration --> Download Bitstream
5. Start any terminal window, such as Hyperterminal, and connect to the serial port that is connected to the FPGA board.

The communication settings should be set to 9600 baud, data 8 bits, parity none, stop bits 1, and flow control hardware.

6. Launch XMD.

Select menu Debug --> Launch XMD or click the button to launch XMD from the XPS GUI.

7. Download the firmware executable file.

   XMD% dow mb-firmware/executable.elf

8. Run the diagnostic test.

   XMD% run

The output from the test will appear in the terminal window.

6.3.2 Running other diags on the FPGA board

To run other diags on the FPGA board, the assembly language program for the desired diag must be assembled, and the resulting executable must be converted into a C structure which is compiled into the mb-firmware application.

1. Run sims to simulate the desired test.

   This calls the assembler to create an executable from assembly language source of the test. The resulting executable contains some simple reset code, trap tables, as well as the main diagnostic code. All addresses in this executable are the physical addresses. The sims program then takes this executable and creates a memory image file. All the code and data sections from the executable are placed into the memory image file. Then a few page table entries and translation storage buffers for virtual memory are added.

2. Run the genmemimage.pl script to create a C source file representation of the executable.

   % genmemimage.pl -single memory-image-file

   This program takes the mem.image file created by sims and converts it into a C structure that is compiled with the MicroBlaze firmware.
3. Copy the output C file to the firmware source directory.

   % cp mbfw_diag_memimage.c mb-firmware/src

4. Recompile the mb-firmware project.
   Right-click on the mb-firmware software project and select “Build Project.”

5. Follow the steps in the previous section to run the diag on the FPGA board.

6.3.3 Running an entire regression on the FPGA board

Scripts are included to allow the user to run an entire regression on the FPGA board. These scripts automatically copy the C structure representation of each executable to the mb-firmware source directory, re-compile the mb-firmware code, download the bitstream to the FPGA, and run the test. The results of each test are recorded, and a summary is generated when the regression is completed. Here is the procedure to run a complete regression:

1. Run a sims regression on either the thread1_mini group or the thread1_full group.
   The sims program will run all the tests included in the regression. At the end each test will have a directory which includes the executable for that test.

   % sims -sim_type=vcs -group=thread1_mini -copyall

   NOTE: For a four-thread core which contains the SPU, more complete regressions are core1_mini and core1_full.
   NOTE: Use the -copyall option to save the mem.image file.

2. Run the genmemimage.pl script to generate C structure representations of all the tests in the regression.

   % genmemimage.pl -d regression-dir -mode {core1:thread1_full:0
   core1:thread1_mini:0}

   This will create a directory called diags which contains all the output files. There will be one file for every test in the regression.

3. Edit the file diag_full.list or diag_mini.list in the EDK project.
   These files are lists of tests which will be run as a set.
4. Run the regression.

The regression must be run from the Chipscope install directory in order to pick up the proper shared library file.

```
% cd chipscope-install-dir/bin/operating-system
% xtclsh edk-project-dir/chipscope_tcl/rundiags.tcl -edk edk-project-dir -list edk-project-dir/scripts/diag_mini.list -mode (corel:thread1_full:0 corel:thread1_mini:0)
```

6.4 Running a standalone program on the OpenSPARC T1 core on an evaluation board

A second firmware set-up is provided to allow the running of a stand-alone program under hypervisor. In this set-up, the OpenSPARC T1 core will boot hypervisor, which will then branch to a stand-alone program. A very simple “Hello World” program is included with the project to demonstrate this. This set-up may also be used to boot an operating system under hypervisor.

The memory map of the system is shown in FIGURE 6-3. One megabyte of the 64 MB on-board memory is allocated for the MicroBlaze firmware code. Another megabyte is allocated for the OpenSPARC boot PROM image. The remaining 62 MB is available for the memory space of the OpenSPARC T1 core. The firmware translates the OpenSPARC T1 addresses to board addresses as shown in FIGURE 6-3.

**FIGURE 6-3** Allocation of the ML410 64-MB DRAM with the hypervisor firmware set-up

<table>
<thead>
<tr>
<th>MicroBlaze Address</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x2400_0000</td>
<td>MicroBlaze Firmware Code (1MB)</td>
</tr>
<tr>
<td>0x2410_0000</td>
<td>OpenSPARC Memory Space: (62 MB)</td>
</tr>
<tr>
<td>-</td>
<td>Addresses: 0x00_0000_0000 - 0x00_03df_ffff</td>
</tr>
<tr>
<td>0x27f0_0000</td>
<td>OpenSPARC boot PROM Image: 0xff_f000_0000 (1MB)</td>
</tr>
</tbody>
</table>
Note that hypervisor requires a core with all 64 TLB entries. For instructions on how to synthesize a netlist with the required TLB size, see Chapter 5.

6.4.1 Running the included “Hello World” Program

An example “Hello World” program is included in the EDK project. The program may be run under hypervisor by performing the following procedure:

1. Compile the hypervisor program.
   The hypervisor program is included in the software package, which is downloaded separately from the hardware package. The software package does include a memory image of the hypervisor program for the convenience of users not on Sun systems. However, if the hypervisor program needs to be re-compiled, the procedure to do it is shown here:

   ```bash
   % setenv SUN_STUDIO path-to-SunStudio-Compilers
   % cd sw-install-dir/softsim/hypervisor/src
   % make
   ```

2. Compile the application program.
   A pre-built memory image of the “Hello World” program is included in the EDK project. In addition, make scripts are included to make it easy to re-compile the example program. The make must be run on a SPARC machine with SunStudio compilers. To compile the code, execute the following commands:

   ```bash
   % setenv SUN_STUDIO path-to-SunStudio-Compilers
   % setenv QTOOLS sw-install-dir/softsim/hypervisor/src/hypervisor-tools
   % cd design/sys/edk/examples/src
   % make install
   ```

3. Compile the MicroBlaze firmware for standalone programs.
   In the XPS user interface right-click on the `mb-firmware-hv` project in the Applications window and select Build.

4. Make sure that the `microblaze_0_bootloop` software project is set to initialize BRAMs.
   Right-click on the `microblaze_0_bootloop` project in the Applications window and select “Mark to initialize BRAMs.”
5. Select the following menu item to update the bitstream.
   Device Configuration --> Update Bitstream

6. Download the bitstream to the FPGA.
   Device Configuration --> Download Bitstream

7. Start any terminal window, such as Hyperterminal, and connect to the serial port
   that is connected to the FPGA board.

8. Start XMD.
   Select menu Debug --> Launch XMD or click the button on the XPS GUI.

9. Download the firmware program to the FPGA.

```
XMD% dow mb-firmware-hv/executable.elf
```

10. Download the binary image of the code for the OpenSPARC T1 to DRAM.

```
XMD% dow -data examples/bin/hello_world.mem.image 0x24900000
```

11. Download the reset and hypervisor code for the OpenSPARC T1 to DRAM.

```
XMD% dow -data sw-install-dir/hypervisor/src/greatlakes/ontario/t1_fpga/prom.bin 0x27f00000
```

12. Run the program.

```
XMD% run
```

The output of the program will appear in the terminal window. CODE EXAMPLE 6-1
shows the hypervisor output, followed by the output of the “Hello World” program.

### 6.4.2 Creating your own stand-alone program

Other programs can be run under hypervisor using this set-up, but there are some
limitations:

- The program must be in a binary image format, because ELF format is not
  understood by hypervisor.
There is no operating system, so system and library calls that are normally handled by the operating system are not available.

Hypervisor does not dynamically link programs, so the program must be statically linked.

Solaris executable programs are in ELF format. When an executable program is run on Solaris, a memory image of the process is created from the executable program by Solaris run-time loader. Hypervisor doesn’t understand ELF format and it doesn’t provide a run-time loader for ELF executable programs. The memory image of the application program is generated off-line by an ELF to memory image utility. It is this memory image that is preloaded in memory. Hypervisor doesn’t support shared libraries and the executable program must be statically linked.

The stand-alone program must be statically linked with libos.a library.

A minimalist Operating System for running the stand-alone program is provided by the library libos.a. All stand-alone programs must be linked with this OS library. The libos.a library provides functionality to initialize hypervisor interface, initialize privileged registers, set up a trap table, set up stack for the stand-alone program and write to console. Spill/Fill trap handlers are provided by the trap table so that the stand-alone program can execute nested and/or recursive function calls.

**CODE EXAMPLE 6-1**  Output of the “Hello World” program running under hypervisor

```
Entering hypervisor
Scrubbing initial RAM
Copying from ROM to RAM
Entering hypervisor
Running from RAM
Hypervisor version: Hypervisor 0.1 2007/08/03 11:23
[greatlakes:ontario:t1_fpga]
Not scrubbing remaining hypervisor RAM. Using pre-initialized memory.
Not scrubbing guest memory. Using pre-initialized guest memory.
Using pre-initialized Guest stand-alone program memory image.

Hello World

Guest stand-alone program has terminated.
Entering infinite loop.
MB-Firmware Warning: No PCX packet received from T1 core
Retrying...
```
6.5 Running system-level simulation with Modelsim

Environment files are provided to enable the user to run a system-level simulation of an OpenSPARC T1 diagnostic test using Xilinx Platform Studio (XPS) and the Modelsim simulator from Mentor Graphics. The simulation environment includes the following items:

- A board model which instantiates the FPGA and a DDR DRAM.
- Scripts to generate the full FPGA model, compile the firmware programs and set up the simulation.

To run the system simulation using the default win_restore0 test, follow the following procedure:

1. **Compile the Xilinx libraries with Modelsim or link the project with already-compiled libraries.**
   
   From the XPS user interface, select the following menu item and follow the instructions from the wizard that pops up:
   
   Simulation --> Compile Simulation Libraries

2. **Ensure that the software project mb-firmware is set to mark BRAMs.**
   
   This should be set by default in the project. If it is not, then right-click on the mb-firmware software project and select "Mark to Initialize BRAMs."

3. **Change the compile options to include the -DSIMULATION option.**
   
   The -DSIMULATION compile option suppresses all print statements from the firmware. This is necessary to avoid simulating the printing of messages, which are sent by the UART at 9600 baud. To set this option, right-click on the mb-firmware software project and select Set Simulation Options. Then click on the Paths and Options tab. The best options for simulation are shown in the box below:

   ```
   -w -DREGRESSION_MODE -DSIMULATION
   ```

4. **Next call a TCL script to generate the system model.**

   ```
   % xps -nw -scr boardsim/setupsim.tcl system.xmp
   ```

   This script generates a simulation model for the entire FPGA, including the OpenSPARC T1 core, MicroBlaze core, memory controllers, and all the connecting logic. It also compiles the firmware code.
5. Download a DDR SDRAM model and modify it to allow pre-initialization using the verilog function `readmemh`.

The system-level simulation requires a DDR SDRAM model, which must be downloaded from a DRAM vendor. The model must allow for pre-initialization so that the firmware code can be placed in memory at the start of the test. The second thing item to be careful about is the ordering of memory addresses within the model. The DRAM memory space is broken up into banks, rows and columns, and the code will be scrambled up if the model partitions its memory differently than the memory controller does. **FIGURE 6-4** shows how the address of the DRAM is split between row, column and band addresses.

**FIGURE 6-4**  Memory addressing of the MicroBlaze memory controller

A DDR SDRAM model from Micron Technology, Inc. was used in preparing the system simulation. This model did not have the capability for pre-initialization of the model, and it used a different bits of the address for row, column and bank addresses. A script is provided to hack this memory model to work with the system simulation set-up. To run this script do the following:

**a. Download the Micron DDR memory model from the following web site:**

http://download.micron.com/downloads/models/verilog/sdram/ddr/256meg/256Mb_ddr.zip

**b. Copy the models to the edk directory and run the patch script.**

```
% cp ddr.v ddr_parameters.v design/sys/edk/boardsim
% cd design/sys/edk/boardsim
% ./micronddr_patch
```

If different DDR SDRAM models are used, the procedure to get them working in the system simulation will be different.
6. Run the simulation.

Change to the behavioral directory and run the simulation.

```
% cd simulation/behavioral
% vsim -do ../../do_sim_mb.do
```

The simulation should start. The first 3 ms of the simulation are required for the MicroBlaze processor to initialize its instruction and data caches. When this is complete, the OpenSPARC T1 core is started and begins to execute code. A monitor model prints diagnostic messages to the log every time the OpenSPARC T1 core fetches instructions from memory. Sample output from the code monitor is shown in CODE EXAMPLE 6-2.

**CODE EXAMPLE 6-2**  Output from the PCX Monitor

```
# PCX: 3082235.000 ns : I-fetch from address 0xfff0000020
# PCX: 3143915.000 ns : I-fetch from address 0xfff0000024
# PCX: 3166855.000 ns : I-fetch from address 0xfff0000028
# PCX: 3188215.000 ns : I-fetch from address 0xfff000002c
# PCX: 3209655.000 ns : I-fetch from address 0xfff0000030

...  

# PCX: 6191955.000 ns : I-fetch from address 0x1130000140
# PCX: 6223335.000 ns : I-fetch from address 0x1130000160
# PCX: 6255515.000 ns : I-fetch from address 0x1000122000
# PCX: 6255515.000 ns : Reached good trap: Diag Passed
```

At the end of the test, the code will branch to one of two locations. The *good trap* location indicates a successful test, while the *bad trap* location indicates a problem. The PCX monitor will stop the simulation shortly after the *good trap* or *bad trap* location is reached.

The default diagnostic test that is simulated is *win_restore0*. Other diagnostic tests may be simulated by following the same procedure used to run these tests on the FPGA board. This is discussed in Section 6.3.2 “Running other diags on the FPGA board” on page 6-6.
Design and Verification Manual Pages

This appendix provides the manual pages for commands used for OpenSPARC T1 design and verification.

A.1 sims

NAME
sims - Verilog rtl simulation environment and regression script

SYNOPSIS
sims [args ...]

NOTE: Use "=" instead of "space" to separate args and their options.

where args are:

SIMULATION ENV

-sys=NAME
sys is a pointer to a specific testbench configuration to be built and run. A config file is used to associate the sys with a set of default options to build the testbench and run diagnostics on it. The arguments in the config file are the same as the arguments passed on the command line.

-group=NAME
  group name identifies a set of diags to run in a
regression. The presence of this argument indicates that this is a regression run. The group must be found in the diaglist. Multiple groups may be specified to be run within the same regression.

-group=NAME -alias=ALIAS
This combination of options gets the diag run time options from the diaglist based on the given group and alias. The group must be found in the diaglist. The alias is made up of diag_alias:name_tag. Only one group should be specified when using this command format.

**VERILOG COMPIILATION RELATED**

- **-sim_type=vcs/ncv**
  Defines which simulator to use, vcs or ncverilog, Defaults to vcs.

- **-sim_q_command="command"**
  Defines which job queue manager command to use to launch jobs. Defaults to /bin/sh and runs simulation jobs on the local machine.

- **-ncv_build/-noncv_build**
  Builds a ncverilog model and the vera testbench. Defaults to off.

- **-ncv_build_args=OPTION**
  ncverilog compile options. Multiple options can be specified using multiple such arguments.

- **-ncv_use_vera/-noncv_use_vera**
  Compiles in the vera libraries. Defaults to off.

- **-vcs_build/-novcs_build**
  Builds a vcs model and the vera testbench. Defaults to off.

- **-vcs_build_args=OPTION**
  vcs compile options. Multiple options can be specified using multiple such arguments.

- **-vcs_clean/-novcs_clean**
  Wipes out the model directory and rebuilds it from scratch. Defaults to off.

- **-vcs_use_2state/-novcs_use_2state**
  Builds a two-state model instead of the default four-state model. This defaults to off.

- **-vcs_use_initreg/-novcs_use_initreg**
  Initializes all registers to a valid state (1/0). This feature works with -tg_seed to set the seed of the random
initialization. Defaults to off.

-vcs_use_fsdb/-novcs_use_fsdb
Uses the debussy fsdb pli and include the dump calls in the
testbench. this defaults to on.

-vcs_use_vcsd/-novcs_use_vcsd
uses the vcs direct kernel interface to dump out debussy files.
Defaults to on.

-vcs_use_vera/-novcs_use_vera
Compiles in the vera libraries. If -vcs_use_ntb and -vcs_use_vera are
used, -vcs_use_ntb wins. Defaults to off.

-vcs_use_ntb/-novcs_use_ntb
Enables the use of NTB when building model (simv) and running simv.
If -vcs_use_ntb and -vcs_use_vera are used, -vcs_use_ntb wins.
Defaults to off.

-vcs_use_rad/-novcs_use_rad
Uses the +rad option when building a vcs model (simv).
Defaults to off.

-vcs_use_sdf/-novcs_use_sdf
Builds vcs model (simv) with an sdf file. Defaults to off.

-vcs_use_cli/-novcs_use_cli
Uses the +cli -line options when building a vcs model (simv).
Defaults to off.

-flist=FLIST
Full path to flist to be appended together to generate the
final verilog flist. Multiple such arguments may be used and
each flist will be concatenated into the final verilog flist
used to build the model.

-graft_flist=GRAFTFILE
GRAFTFILE is the full path to a file that lists each verilog
file that will be grafted into the design. The full path to
the verilog files must also be given in the GRAFTFILE.

-vfile=FILE
Verilog file to be included into the flist

-config_rtl=DEFINE
Places each such parameter as a 'define' in config.v to
configure the model being built properly. This allows
each testbench to select only the rtl code that it needs
from the top-level rtl file.
-model=NAME
   The name of a model to be built. The full path to a model is $MODEL_DIR/$model/$vcs_rel_name.

-vcs_rel_name=NAME
   Specifies the release of the model to be built. The full path to a model is $MODEL_DIR/$model/$vcs_rel_name.

**VERA COMPILATION RELATED**

VERA and NTB share all of the vera options except a few. See NTB RELATED.

-verbose_build/-noverbose_build
   Builds the vera/ntb testbench. Defaults to on.

-verbose_clean/-noverbose_clean
   Performs a make clean on the vera/ntb testbench before building the model. Defaults to off.

-vera_build_args=OPTION
   Vera testbench compile time options. Multiple options can be specified using multiple such commands. These are passed as arguments to the gmake call when building the vera testbench.

-vera_diag_args=OPTION
   Vera/ntb diag compile-time options. Multiple options can be specified using multiple such arguments.

-vera_dummy_diag=NAME
   Provides a dummy vera diag name that will be overridden if a vera diag is specified, else used for vera diag compilation.

-vera_pal_diag_args=OPTION
   Vera/ntb pal diag expansion options. (i.e., "pal OPTIONS -o diag.vr diag.vrpal") Multiple options can be specified using multiple such arguments.

-vera_proj_args=OPTION
   Vera proj file-generation options. Multiple options can be specified using multiple such arguments.

-vera_vcon_file=ARG
   Name of the vera vcon file that is used when running the simulation.

-vera_cov_obj=OBJ
   This argument is passed to the vera Makefile as a OBJ=1 and to vera as -DOBJ to enable a given vera coverage object. Multiple
such arguments can be specified for multiple coverage objects.

NTB RELATED

NTB and VERA share all of the vera options except these:

-vcs_use_ntb/-novcs_use_ntb
   Enables the use of NTB when building model (simv).
   If -vcs_use_ntb and -vcs_use_vera are used, -vcs_use_ntb wins.
   Defaults to off.

-ntb_lib/-nontb_lib
   Enables the NTB two-part compile where the Vera/NTB files get
   compiled first into a libtb.so file which is dynamically
   loaded by vcs at runtime. The libtb.so file is built by
   the Vera Makefile, not sims. Use the Makefile to affect the
   build. If not using -ntb_lib, sims will build VCS and NTB
   together in one pass (use Makefile to affect that build as
   well). Defaults to off.

VERILOG RUNTIME RELATED

-vera_run/-novera_run
   Runs the vcs simulation and loads in the vera proj file
   or the ntb libtb.so file. Defaults to on.

-vcd/-novcd
   Signals the bench to dump in VCD format.

-vcdfile=filename
   The name of the vcd dump file. If the file name starts with
   a "/", that is the file dumped to; otherwise, the actual file is
   created under $tmp_dir/$vcdfile and copied back to the current
   directory when the simulation ends. Use "-vcdfile='pwd'/filename"
   to force the file to be written in the current directory directly
   (not efficient since dumping is done over network instead of to
   a local disk).

-vcs_run/-novcs_run
   Runs the vcs simulation (simv). Defaults to off.

-vcs_run_args=OPTION
   vcs (simv) runtime options. Multiple options can be specified
   using multiple such arguments.

-vcs_finish=TIMESTAMP
   Forces vcs to finish and exit at the specified timestamp.
-fast_boot/-nofast_boot
    Speeds up booting when using the ciop model. Passes the
    +fast_boot switch to the simv run and the -sas_run_args=-DFAST_BOOT
    and -midas_args=-DFAST_BOOT to sas and midas. Also sends
    -DFAST_BOOT to the diaglist and config file preprocessors.

-debussy/-nodebussy
    Enables debussy dump. This must be implemented in the testbench
    to work properly. Defaults to off.

-start_dump=START
    Starts dumping out a waveform after START number of units.

-stop_dump=STOP
    Stops dumping out a waveform after STOP number of units.

-fsdb2vcd
    Runs fsdb2vcd after the simulation has completed to generate a vcd file.

-fsdbfile=filename
    The name of the debussy dump file.
    If the file name starts with a "/", that is the file dumped to.
    Otherwise, the actual file is created under $tmp_dir/$fsdbfile
    and copied back to the current directory when the simulation ends.
    Use "-fsdbfile='pwd'/filename" to force the file to be
    written in the current directory directly (not efficient since
    dumping is done over network instead of to a local disk).

-fsdbDumplimit=SIZE_IN_MB
    Max size of Debussy dump file. Minimum value is 32MB.
    Latest values of signal values making up that size is saved.

-fsdb_glitch
    Turns on glitch and sequence dumping in fsdb file. This will collect
    glitches and sequence of events within time in the fsdb waveform.
    beware that this will cause the fsdb file size to grow significantly.
    This option effectively does this:
        setenv FSDB_ENV_DUMP_SEQ_NUM 1
        setenv FSDB_ENV_MAX_GLITCH_NUM 0
    Defaults to off.

-rerun
    Reruns the simulation from an existing regression run directory.

-post_process_cmd=COMMAND
    Post-processing command to be run after vcs (simv) run completes.

-pre_process_cmd=COMMAND
Pre-processing command to be run before vcs (simv) run starts.

-use_denalirc=FILE
   Uses FILE as the .denalirc in the run area. Default copies $env_base/.denalirc

ZEROIN RELATED

-zeroIn_checklist
   Runs 0in checklist

-zeroIn_build
   Builds 0In pli for simulation into vcs model.

-zeroInSearch_build
   Builds 0In search pli for simulation into vcs model.

-zeroIn_build_args
   Additional arguments to be passed to the 0in command.

-zeroIn_dbg_args
   Additional debug arguments to be passed to the 0in shell.

SAS RELATED

-sas/-nosas
   Runs architecture-simulator. If vcs_run option is OFF, simulation is sas-only. If vcs_run option is ON, sas runs in lock-step with rtl. Defaults to off.

-sas_run_args=DARGS
   Defines arguments for sas.

MIDAS RELATED

midas is the diag assembler.

-midas_args=DARGS
   Arguments for midas. midas creates memory image and user-event files from the assembly diag.

-midas_only
   Compiles the diag using midas and exit without running it.

-midas_use_tgseed
   Adds -DTG_SEED=tg_seed to midas command line. Use -tg_seed to set the value passed to midas or use a random value from /dev/random.

SJM RELATED
sjm is the J-Bus bus functional model

-**sjm_args**
  Arguments to be passed in to sjm_tstgen.pl for generation of an sjm random diagnostic.

-**sjm/-nosjm**
  Generates a random sjm diagnostic using the -tg_seed if provided. Defaults to off.

-**tg_seed**
  Random generator seed for sjm random test generators.
  Also the value passed to +initreg+ to randomly initialize registers when -vcs_use_initreg is used.

**VCS COVERMETER RELATED**

-**vcs_use_cm/-novcs_use_cmd**
  Passes in the -cm switch to vcs at build time and simv at runtime Defaults to off.

-**vcs_cm_args=ARGS**
  Argument to be given to the -cm switch.

-**vcs_cm_cond=ARGS**
  Argument to be given to the -cm_cond switch.

-**vcs_cm_config=ARGS**
  Argument to be given to the -cm_hier switch.

-**vcs_cm_fsmcfg=ARGS**
  Argument to be given to the -cm_fsmcfg switch.
  Specifies an FSM coverage configuration file.

-**vcs_cm_name=ARGS**
  Argument to be given to the -cm_name switch. defaults to cm_data.

**MISC**

-**nobuild**
  Master switch to disable all building options.
  There is no such thing as -build to enable all build options.

-**copyall/-nocopyall**
  Copies back all files to launch directory after passing regression run. Normally, only failing runs cause a copy back of files. Defaults to off.
-copydump/-nocopydump
Copies back dump file to launch directory after passing regression run. Normally, only failing runs cause a copy back of non-log files. The file copied back is sim.fsdb, or sim.vcd if -fsdb2vcd option is set. Default is off.

-tarcopy/-notarcopy
Copies back files using 'tar'. This only works in copyall or in the case the simulations 'fails' (per sims' determination). Default is to use 'cp'.

-diag_pl_args=ARGS
If the assembly diag has a Perl portion at the end, it is put into diag.pl and is run as a Perl script. This allows you to give arguments to that Perl script. The arguments accumulate, if the option is used multiple times.

-pal_use_tgseed
Sends '-seed=<tg_seed_value>' to pal diags. Adds -pal_diag_args=-seed=tg_seed to midas command line, and -seed=tg_seed to pal options (vrpal diags). Use -tg_seed to set the value passed to midas or use a random value from /dev/random.

-parallel
When specifying multiple groups for regressions, this switch will submit each group to Job Q manager to be executed as a separate regression. This has the effect of speeding up regression submissions. NOTE: This switch must not be used with -injobq

-reg_count=COUNT
Runs the specified group multiple times in regression mode. This is useful when we want to run the same diag multiple times using a different random generator seed each time or some such.

-regress_id=ID
Specifies the name of the regression.

-report
Used to produce a report of a an old or running regression. With -group options, sims produces the report after the regression run. Report for the previous regression run can be produced using -regress_id=ID option along with this option.

-finish_mask=MASK
Masks for vcs simulation termination. Simulation terminates when it hits 'good_trap' or 'bad_trap'. For multithread
simulation, simulation terminates when any of the thread
hits bad_trap, or all the threads specified by the finish_mask
hits the good_trap.
example: \texttt{-finish\_mask}=0xe
Simulation will be terminated by good\_trap, if threads 1, 2 and
3 hit the good\_trap.

\texttt{-stub\_mask}=MASK
Mask for vcs simulation termination. Simulation ends when the
stub driving the relevant bit in the mask is asserted. This
is a hexadecimal value similar to \texttt{-finish\_mask}.

\texttt{-wait\_cycle\_to\_kill}=VAL
Passes a \texttt{+wait\_cycle\_to\_kill} to the \texttt{simv} run. A testbench
may chose to implement this plusarg to delay killing a
simulation by a number of clock cycles to allow collection
of some more data before exiting (e.g. waveform).

\texttt{-rtl\_timeout}
Passes a \texttt{+TIMEOUT} to the \texttt{simv} run.
Sets the number of clock cycles after all threads have become
inactive for the diag to exit with an error. If all threads hit
good trap on their own the diag exits right away. If any of the
threads is inactive without hitting good trap/bad trap the
rtl\_timeout will be reached and the diag fails. Defaults to 1000.
This is only implemented in the cmp based testbenches.

\texttt{-max\_cycle}
Passes a \texttt{+max\_cycle} to the \texttt{simv} run.
Sets the maximum number of clock cycle that the diag will take
to complete. Defaults to 30000. If \texttt{max\_cycle} is hit the diag
exits with a failure. Not all testbenches implement this
feature.

\texttt{-norun\_diag\_pl}
Does not run diag.pl (if it exists) after simv (vcs) run.
Use this option if, for some reason, you want to run an
existing assembly diag without the Perl part that is in
the original diag.

\texttt{-nosaslog}
Turns off redirection of sas stdout to the sas.log file.
Use this option when doing interactive runs with sas.

\texttt{-nosimslog}
Turns off redirection of stdout and stderr to the sims.log file.
Use this option to get to the cli prompt when using vcs or to
see a truncated sim.log file that exited with an error. This
must be used if you want control-c to work while vcs is running.
Appendix A Design and Verification Manual Pages

-nogzip
   Turns off compression of log files before they are copied over
during regressions.

-version
   Print version number.

-help
   Prints this man page.

IT SYSTEM RELATED

-use_iver=FILE
   Full path to iver file for frozen tools.

-use_sims_iver
   For reruns of regression tests only, use sims.iver to choose
   TRE tool versions saved during original regression run.

-dv_root=PATH
   Absolute path to design root directory. This overrides $DV_ROOT.

-model_dir=PATH
   Absolute path to model root directory. This overrides $MODEL_DIR.

-tmp_dir=PATH
   Path where temporary files such as debussy dumps will be created.

-sims_config=FILE
   Full path to sims config file.

-env_base=PATH
   Specifies the root directory for the bench environment.
   It is typically defined in the bench config file. It has no
default.

-config_cpp_args=OPTION
   Allows the user to provide CPP arguments (defines/undefines)
   that will be used when the testbench configuration file is
   processed through cpp. Multiple options are concatenated
   together.

-result_dir=PATH
   Allows the regression run to be launched from a different
directory than the one sims was launched from. Defaults to
   $ENV(PWD).

-diaglist=FILE
Full path to diaglist file.

-diaglist_cpp_args=OPTION
   Allows the user to provide CPP arguments (defines/undefines) that will be used when the diaglist file is processed through cpp. Multiple options are concatenated together.

-name of the diagnostic to be run.

-asm_diag_name=NAME
-asm_diag_root=PATH
-asm_diag_path=PATH

-tpt_diag_name=NAME
-tpt_diag_root=PATH
-tpt_diag_path=PATH

-tap_diag_name=NAME
-tap_diag_root=PATH
-tap_diag_path=PATH

-vera_diag_name=NAME
-vera_diag_root=PATH
-vera_diag_path=PATH

-vera_config_name=NAME
-vera_config_root=PATH
-vera_config_path=PATH

-efuse_image_name=NAME
-efuse_image_root=PATH
-efuse_image_path=PATH

-image_diag_name=NAME
-image_diag_root=PATH
-image_diag_path=PATH

-sjm_diag_name=NAME
-sjm_diag_root=PATH
-sjm_diag_path=PATH

-pci_diag_name=NAME
-pci_diag_root=PATH
-pci_diag_path=PATH

Absolute path to diag root directory. sims will perform a find from here to find the specified type of diag. If more than one instance of the diag name is found under root, sims exits with an error. this option can be specified multiple times to allow multiple roots to be searched for the diag.

Absolute path to diag directory. sims expects the specified diag to be in this directory. The last value of this option is the one used as the path.

**ENV VARIABLES**

sims sets or uses the following ENV variables that may be used with pre/post
processing scripts, and other internal tools:

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIMS_LAUNCH_DIR</td>
<td>Path to launch directory where sims is running the job</td>
</tr>
<tr>
<td>ASM_DIAG_NAME</td>
<td>Contains the assembly diag name</td>
</tr>
<tr>
<td>VERA_LIBDIR</td>
<td>Dir where Vera files are compiled</td>
</tr>
<tr>
<td>DV_ROOT</td>
<td>Overwrite by -dv_root if specified</td>
</tr>
<tr>
<td>MODEL_DIR</td>
<td>Overwrite by -model_dir if specified</td>
</tr>
<tr>
<td>TRE_SEARCH</td>
<td>Based on -use_iver, -use_sims_iver</td>
</tr>
<tr>
<td>DENALI</td>
<td>User defined</td>
</tr>
<tr>
<td>VCS_HOME</td>
<td>User defined</td>
</tr>
<tr>
<td>VERA_HOME</td>
<td>User defined</td>
</tr>
<tr>
<td>NCV_HOME</td>
<td>User defined</td>
</tr>
</tbody>
</table>

PLUSARGS

+args are not implemented in sims. They are passed directly to simulator at compile time and at runtime.

DESCRIPTION

sims is the frontend for vcs to run single simulations and regressions.

How To Build Models

Build a vcs model using $DV_ROOT as design root:

```bash
sims -sys=cmp -vcs_build
```

Build a ncverilog model using $DV_ROOT as design root:

```bash
sims -sys=cmp -ncv_build
```

Build the vera testbench only using $DV_ROOT as design root:

```bash
sims -sys=cmp -vera_build
```

Build a model from any design root:

```bash
sims -sys=cmp -vcs_build -dv_root=/home/regress/2002_06_03
```
Build a graft model from any design root:

    sims -sys=cmp -vcs_build -dv_root=/model/2002_06_03 \
    -graft_flist=/regress/graftfile

Build a model and re-build the vera:

    sims -sys=cmp -vcs_build -vera_clean

Build a model and turn off incremental compile:

    sims -sys=cmp -vcs_build -vcs_clean

Build a model with a given name:

    sims -sys=cmp -vcs_build -vcs_rel_name=mymodel

**How to Run Models**

Run a diag with default model:

    sims -sys=cmp -vcs_run diag.s

Run a diag with a specified model:

    sims -sys=cmp -vcs_rel_name=mymodel -vcs_run diag.s

Run a diag with debussy dump with default model:

    sims -sys=cmp -debussy +dump=cmp_top:0 -vcs_run diag.s

=head2 Run regressions

Run a regression using $DV_ROOT as design root:

    sims -group=mini

Run a regression using $DV_ROOT as design root and specify the diaglist:

    sims -group=mini -diaglist=/home/user/my_dialist

Run a regression using any design root:

    sims -group=mini -dv_root=/afara/design/regress/model/2002_06_03
A.2 midas

NAME
midas - assembles diags (Midas Is a Diag Assembler)

SYNOPSIS
midas [options] <diag_name>

DESCRIPTION
This program builds assembly diags. It is substantially more involved than simply assembling the diag because it also has to link the diag, program the MMU, and generate several output files.

The diag specified on the command line will be built. Pretty much everything else is configurable.

Options
The following are the options you need to get started:

- h  Display man page.
-verbose [level] / -noverbose (abbreviated -v / -nov)
  Sets verbosity level (default=2). -noverbose (or -nov)
  is a synonym for -verbose 0, which means to generate no output in the absence of errors. The highest level of verbosity currently defined is 3.

-version
  Returns version information and exit.

-format
  Displays help on the diag format and exit.

-config <file>
  Uses this file as the config file instead of the one that is distributed with Midas.

-project <project>
  Uses this project for project-specific configuration. Default is the environment variable $PROJECT. Legal value is OpenSPARCT1.
Common Options

The following are the commonly used options:

-diag_root <path>
  Uses the specified path as a base for finding standard
  include files. Default is $DV_ROOT.

-build_dir <path>
  Path (absolute or relative to where command is invoked)
  to directory where temporary files are generated and the
  build is done. Default is './build'.

-dest_dir <path>
  Path (absolute or relative to where command is invoked)
  of where to store output files. Default is './'.

-find_root <dir>
  Interprets the diag on the command line as the name of a
diag to search for. It does a breadth-first search
under the specified directory. The default behavior is
not to do any search, but to assume that the specified
diag is a full or relative path to the file.

-find
  This is a shortcut for "-find_root
  <diag_root>/verif/diag".

-mmu <mmu_type>
  Generates programming for the specified MMU. Recognized
  options are "ultra2", "OpenSPARCT1".

-ttefmt <tte_format>
  Specifies TTE format for those MMUs that require it.
  May be "sun4u" or "sun4v". Default is project specific:
  "sun4v" for OpenSPARC T1.

-tsbtagfmt <tsbtagfmt>
  Specifies the format of the TSB tag. Legal values are
  'tagaccess' and 'tagtarget'. Default is
  project-specific: 'tagaccess' OpenSPARC T1.

-force_build or -f
  Builds the diag, even if it looks like it has the same
  input as before and the same args as before.

-copy_products / -nocopy_products
  By default, the product files generated in the build
directory are hard linked to the destination directory.
The reason they are hard linked and not copied is for speed. If the hard link fails, it will fall back to a copy in case the directories are on different physical disks. If -copy_products is given, however, it will always do a copy, not a hard link. Default is project specific: -nocopy_products for OpenSPARC T1.

-E Stops after the preprocessing stage.

-clean up / -nocleanup
   If -clean up is enabled, then after a successful build, the build directory is erased if and only if the build directory was created by this invocation of midas. Default is project specific: -clean up for OpenSPARC T1.

-force_clean up / -noforce_clean up
   If -clean up is enabled, but this invocation of midas did not create the build directory, -force_clean up will remove the build directory anyway. Default is project specific: -noforce_clean up for OpenSPARC T1.

-D<symbol> or -D<symbol>=<value>
   Adds a define to the preprocessing line. Option may be repeated.

-stddef / -nostddef
   Includes standard preprocessor definitions on command line. -nostddef disables these. Default is -stddef, but no standard symbols are currently defined.

-I<dir>
   Adds a directory to the include path used by cpp and m4. Path should be absolute or relative to the directory where midas was invoked. Option may be repeated.

-stdinc / -nostdinc
   With -stdinc, the standard include paths are used during preprocessing (both cpp and m4). -nostdinc disables these. The standard include directories are the directory where midas was invoked, the build directory and <diag_root>/verif/diag/assembly/include (keep in mind that <diag_root> defaults to $DV_ROOT). Default is -stdinc.

-include_build / -noinclude_build
   This option is only meaningful with -nostdinc. If standard includes are switched off, -include_build will add the build directory back to the include path. Default is -noinclude_build.
-include_start / -noinclude_start
This option is only meaningful with -nostdinc. If
standard includes are switched off, -include_start will
add the start directory (the directory where midas was
invoked) back to the include path. Default is
-noinclude_start.

-L<dir>
Adds a directory to the search path when looking for
object files in a MIDAS_OBJ directive. Option may be
repeated.

-C<dir>
Adds a directory to the search path when looking for C
source files in a MIDAS_CC directive. Option may be
repeated.

-pal_diag_args <args>
If the diag is run through pal, gives these arguments to
the pal diag. Option may be repeated. Note that these
arguments are given to the diag, not pal itself. For
instance, "midas -pal_args -abc mydiag.pal
-pal_diag_args def -pal_diag_args ghi" will run the pal
command sline "pal -abc mydiag.pal def ghi".

-build_threads <num_threads>
When doing work that can be done in parallel (such as
assembling a bunch of files), use <num_threads> to do
it. Default is project specific: 3 for OpenSPARC T1.

-print_errors / -noprint_errors
If -noprint_errors is defined, then generation of error
messages is turned off. When used with -verbose 0,
midas is completely silent. This is probably only useful
for the test harness (which is why the switch is there).

-copy_products / -nocopy_products
If this is set, then copies files from the build directory
to the starting directory. With -nocopy_products, the
files are hard linked instead. If it tries to create a
hard link and fails, it will fall back to a copy.
Default is -nocopy_products.

-compress_image / -nocompress_image
If -compress_image is enabled (as it is by default),
then allows compressed mem.images to be generated. By
default, all MMU-generated blocks are compressed when
written to mem.image, meaning that instead of
initializing unused sections to zero, they are simply
uninitialized. The -nocompress_image is equivalent to explicitly putting a ‘compressimage=0’ in all attr_text/attr_data blocks.

-env_zero / -noenv_zero
When compressing blocks, if -env_zero is enabled the blocks will contain ‘// zero_image’ directives to the environment. These directives are supported only by OpenSPARC T1, and they are used to backdoor initialize large tracts of memory to zero. If -noenv_zero is used, then compression will simply leave the data uninitialized.

-default_radix <decimal|hex>
Radix to assume for all parameters that do not explicitly start with ‘0x’. Default is ‘decimal’.

-gen_all_tsbs / -nogen_all_tsbs
If -gen_all_tsbs is given, then all TSBs that are defined are written to the memory image. If -nogen_all_tsbs, then generate only the TSBs that are used. Default is project specific: -nogen_all_tsbs for OpenSPARC T1.

-allow_tsb_conflicts / -noallow_tsb_conflicts
If -allow_tsb_conflicts is enabled, then it is legal to have multiple virtual addresses map to the same entry in a TSB. A linked list will be created to hold all entries. With -noallow_tsb_conflicts (which is the default for N1), collisions in the TSB can only happen with the same VA but different contexts. Default is project specific.

-allow_empty_sections / -noallow_empty_sections
If TEXT_VA is specified, then at least one attr_text block for the section has to be specified, and the same is true for DATA_VA and attr_data blocks. If -allow_empty_sections is specified, then midas will allow you to specify a TEXT_VA(DATA_VA) for the section, even if the section has no attr_text(attr_data) blocks. Of course, any text(data) in such a section will be ignored. Default is project specific: -noallow_empty_sections for OpenSPARC T1.

-allow_duplicate_tags / -noallow_duplicate_tags
When adding to a TSB link list, it is an error to add the same tag twice. -allow_duplicate_tags suspends the error check. Default is project specific: -noallow_duplicate_tags for OpenSPARC T1.
-allowIllegalPageSizes / -noallowIllegalPageSizes
If -allowIllegalPageSizes, then tte_size attributes are not checked for valid values, though they are still checked against the width of the field. For instance, in the OpenSPARC T1 MMU, there are 3 page bits, so values can be specified 0-7. However, the only legal values for OpenSPARC T1 are 0, 1, 3, and 5, and unless -allowIllegalPageSizes is in effect, setting page bits of 2, 4, 6, or 7 will cause an error. The default is project specific: -noallowIllegalPageSizes for OpenSPARC T1.

-allowMisalignedTsbBase / -noallowMisalignedTsbBase
If -allowMisalignedTsbBase is set, then a TSB base address need not be aligned with the TSB size. If an unaligned address is specified as the base and -allowMisalignedTsbBase is specified, then midas will forcibly align the address. Default should be -noallowMisalignedTsbBase for all projects.

-errcode <error_code>
Prints a one-line description for the midas error code, then exits with status 0.

Configuring Commands

midas runs several commands in the course of its operation. Several of these can be configured. The configurable commands are: pal, cpp, m4, gcc, as, and ld. Each configurable command has 3 associated options:

-std<command>args / -nostd<command>args
When -std<command>args is enabled, the standard set of arguments for <command> is used. Default is -std<command>args

-<command>args <args>
Adds <args> to the argument list for the specified <command>.

-<command>cmd <custom_command>
Uses <custom_command> to run the specified <command> instead of the standard version.

Example

For instance, to add -foo to the link line, use my_cpp to preprocess, and not use any standard assembler options, use:
midas -ld_args -foo -cpp_cmd my_cpp -nostd_as_args mydiag.s

Configuring Filenames

There are several generated files, and they all have default names. You can configure the names of many of the files with the following option:

-file <tag>=<name>

Causes midas to name the file whose tag is <tag> to be named <name> instead of the default. <name> is treated as the name of a file in the build directory.

Valid tags for the -file option are:

src Local version of the original source code for the diag. Default is diag.src.
s  Assembly portion of diag before any preprocessing. Default is diag.s.
pl  Perl portion of the diag. Default is diag.pl.
cpp Output of the C preprocessor. Default is diag.cpp.
m4  Output of the m4 preprocessor. Default is diag.m4.
ldscr Linker script. Default is diag.ls_scr.
exe Linked executable. Default is diag*.exe where * is application name.
image Verilog memory image. Default is mem.image.
events Events file Default is diag.ev.
symtab Symbol table. Default is symbol.tbl.
goldfinger Specification to goldfinger on how to create memory image. Default is diag.goldfinger.
directives File to contain midas directives after section splitting. Default is diag.midas.
cmdfile
File to stash the midas command-line. Default is .midas_args.

oldcmdfile
File to move old command-line options. Default is .midas_args.old.

oldm4
File to stash m4 output of previous run. Default is .midas.diag.m4.old.

Running Specific Phases

The build process is broken into phases: setup, preprocess, sectioning, assemble, link, postprocess, copydest, cleanup. The default behavior is to run all phases. You can, however, restrict operation to a selected set of phases.

-start_phase <phase_name>
Starts with the named phase and run all subsequent phases.

-phase <phase_name>
Runs the specified phase. If any -phase or -start_phase option exists, then by default all phases are off (except for the ones that -phase and -start_phase switch on). You can have multiple -phase options.

-E This option (mentioned above, which runs the preprocessor only) is just a shortcut for "-phase setup -phase preprocess").

Keep in mind that running selected phases is caveat emptor. There are cases where phases expect data or files from previous phases.

Errors

When midas is unable to run correctly it will exit with one of the following error codes.

M_NOERROR (#0): No error.
M_MISC (#1): Miscellaneous error
M_CODE (#2): Error in midas code.
M_DIR (#3): Directory error.
M_FILE (#4): File error.
M_CMDFAIL (#5): Command failed.
M_SECSYNTAX (#6): Error in section syntax.
M_ATTRSYNTAX (#7): Error in attr syntax.
M_MISSINGPARAM (#8): Missing parameter.
M_ILLEGALPARAM (#9): Illegal parameter.
M_OUTOFRANGE (#10): Out of range.
M_NOTNUM (#11): Not a number.
M_VACOLLIDE (#12): VA collision.
M_PACOLLIDE (#13): PA collision.
M_ASMFAIL (#16): Assembler failed.
M_CCFAIL (#17): C compiler failed.
M_LINKFAIL (#18): Linker failed.
M_CPPFAIL (#19): CPP failed.
M_M4FAIL (#20): M4 preprocessor failed.
M_BADCONFIG (#21): Bad configuration.
M_EVENTERR (#22): Event parsing error.
M_ARGERR (#23): Argument error.
M_NOSEC (#24): Undefined section.
M_BADTSB (#25): Bad TSB.
M_BADALIGN (#26): Bad Alignment.
M_EMPTYSECTION (#27): Empty section.
M_TSBSYNTAX (#28): Error in tsb syntax.
M_APPSYNTAX (#29): Error in app syntax.
M_MEMORY (#30): Memory error.
M_GOLDFINGERPARSE (#31): Goldfinger parse error.
M_GOLDFINGERARG (#32): Goldfinger arg error.
M_ELF (#33): ELF error.
M_BADLABEL (#34): Bad label.
M_GOLDFINGERMISC (#35): Uncategorized goldfinger error.
M_GOLDFINGERVERSION (#36): Bad version of goldfinger
M_DUPLICATETAG (#37): Duplicate tags in TSB
M_BLOCKSYNTAX (#38): Error defining goldfinger BLOCK
A.3 goldfinger

NAME

goldfinger - Midas’ partner for building diags

SYNOPSIS

goldfinger [options]

DESCRIPTION

Goldfinger is midas’ partner. Goldfinger is implemented in C and uses libelf for efficient analysis of ELF files. In the new regime, midas builds a linked executable and a command file (i.e., a .goldfinger file), which are then processed by goldfinger. The final output files are produced by goldfinger. It is the intention that end users never invoke goldfinger directly, but only through midas. Nevertheless, users may find a case why they need to build a diag in a very non-standard way, and goldfinger provides a lower-level interface.

Goldfinger is typically used twice in a normal build process:

Section splitting
"goldfinger -splitsec <diag_file>" is used to split a diag into multiple assembly files, one per section. All embedded midas directives are written to a separate file.

Extracting from executable file
After the executable file is linked, midas needs to extract a memory image and a symbol table. The options "goldfinger -in <cmd_file> -genimage -gentsbs -gensymtab" will generate these files based on the directives in <cmd_file>.

Options

The options recognized by goldfinger are:

- h  Show usage.

- version  Print version number and exit.
-v or -verbose
   Make it more chatty.

-d or -debug
   Make it very chatty.

-silent
   Say nothing unless there’s an error.

-n or -nooutput
   Do not write any output files (for debugging only).

-noprint_errors
   Don’t print any error messages (usually used with
   -silent). You can still tell there was an error by the
   exit status.

-prefix <string>
   Prepend <string> to each line of normal output.

-destdir <dir>
   All created files go in this directory (or a relative
   path from it). The directory specified can be absolute
   or relative from where goldfinger is invoked.

-srcdir <dir>
   If any of the command files specify filenames with
   relative paths, start searching from this directory.
   Note that the command files themselves are always
   specified absolutely or relative to where goldfinger is
   run.

Section splitting options

The following functions are meaningful when splitting
sections.

-splitsec <file>
   Splits the specified file into sections and writes an
   assembly file for each. Writes all midas directives
   into a file that must be specified by the -midasfile
   option.

-midasfile <file>
   When doing section splitting, write all midas directives
   into this file.
Linked executable options

The following options are meaningful when analyzing linked executables.

-in <command_file>
   Analyzes linked executables based in the directives in
   <command_file> (also referred to as a .goldfinger file).

-genimage
   Generates a memory image based on the linked executable.
   Goes to stdout unless -imagefile is also specified.

-imagefile <file>
   If -genimage is also specified, then redirects output here
   instead of stdout.

-gensymtab
   Generates a symbol table from the linked executable.
   Goes to stdout unless -symtabfile is also specified.

-symtabfile <file>
   If -gensymtab is also specified, then writes the symbol
   table here instead of stdout.

-gentsbs
   Generates TSB programming based on the object files. It
   is in mem.image format. It will go to stdout unless
   -imagefile is also specified.

-allow_tsb_conflicts
   If -tsbgen is also provided, then doesn’t cause a fatal
   error if there is a collision in the TSB. Adds to the
   TSB_LINK area instead.

-allow_duplicate_tags
   If -allow_tsb_conflicts is enabled, you are adding
   elements to a TSB_LINK area, and you try to add the same
   tag more than once, it is normally an error. This
   option disables the error check. This option is not
   recommended, since the duplicate tag defines a
   translation that can never be used.

-nocompress
   Does not do compression of mem.image sections for any
   sections, regardless of what is in the imagespec file.

-noenvzero
   Does not use the backdoor environment initialization to
zero during image compression.

COMMAND FILE SYNTAX

In the command file (i.e., the .goldfinger file), all keywords can be either all uppercase or all lowercase, but not mixed. All numbers are 64-bit numbers. They can be written as decimal (first digit is 1-9), octal (first digit is 0), or hex (begins with 0x). A boolean option can be set to a nonzero number (true) or 0 (false). If a boolean option is named, but not assigned to (e.g., "COMPRESS;" instead of "COMPRESS = 1;"), then it is assigned to 1.

The attrs file is a list of four types of objects at the top level. They can appear on any order:

```
PA_SIZE = num;
APP <name>
    app_lines
END APP
TSB <name>
    tsb_lines
END TSB
TSB_LINK <name>
    tsb_link_lines
END TSB_LINK
```

The PA_SIZE field is the only top-level attribute. It defines the size of a physical address in bits. The default is 40.

All types of block contain two attributes:

```
SRC_FILE = "file";
    File name where this block is originally defined. Used for error and debugging output.
SRC_LINE = num;
    Line number in SRC_FILE where this block is originally defined. Used for error and debugging output.
```

APP

An APP object contains a few parameters and a list of block objects. An APP names one linked executable (see ELF_FILE) and a list of blocks that describe what to do with that
file. The APP syntax is:

APP <appname>

SRC_FILE = "source file";
SRC_LINE = <num>;
ELF_FILE = "executable file";

BLOCK <name>
    block_attrs
END BLOCK

BLOCK <another_name>
    block_attrs
    BLOCK_TSB <name>
        block_tsb_attrs
    END BLOCK_TSB
END BLOCK

...

END APP

ELF_FILE = "executable file";
Names the linked executable file (relative to srcdir) that will be processed by this APP object.

BLOCK

A BLOCK defines a section of a linked executable that should be treated the same way. It can take the following parameters:

SECTION_NAME = "name";
Name of the section (e.g., ".MAIN") where this block is defined (used for debugging and error reporting). Used only for error reporting.

SEGMENT_NAME = "name";
Name of segment within the section (e.g., "text") for which this block is defined. Used only for error reporting.

LINK_SECTION = "name";
ELF section name where this block should look in the executable.

VA = <num>;}
START_LABEL = "label";
  Optionally specifies that the block should start at a
  particular address or label. You can specify one or the
  other, but not both. If neither is specified, then the
  starting VA for the elf section is used. The starting
  address, however it is specified, must be page aligned
  if it is to be added to a TSB.

END_VA = <num>;
END_LABEL = "label";
  Optionally specifies that the block should end at a
  particular address or label. You can specify one or the
  other, but not both. If neither is specified, the the
  ending VA for the elf section is used.

COMPRESS = num;
  Boolean. If set, then compresses the output of this block
  in the image. Compression means that if an entire line
  (i.e., aligned 32 bytes) is zero, the line is skipped.
  If -noenvzero is enabled, the 32 bytes are simply
  uninitialized. Otherwise, the backdoor ‘// zero_bytes’
  syntax is used to initialize the memory in the
  environment. The backdoor syntax is specific to
  Niagara, so other projects should either adopt it or use
  -noenvzero.

IN_IMAGE = <boolean>;
  If this is defined and num is zero, then doesn’t write
  this block to the memory image. It is still included in
  the symbol table.

PA = <num>;
  Physical address to write to the image file. Also used
  in symbol table.

RA = <num>;
  Real address. Used to write into the TSB and for the
  symbol table. Written as ’X’ in the symbol table if
  this is not specified.

RA_EQ_VA = <boolean>;
  Boolean. If set, then sets RA to VA (perhaps after VA is
  computed from a label). It is illegal to set both RA
  and RA_EQ_VA.

PA_EQ_VA = <boolean>;
  Boolean. If set, then sets PA to VA (perhaps after VA is
  computed from a label). It is illegal to set both RA
  and PA_EQ_VA.
NO_END_RANGE_CHECK = <boolean>

If this is set to a nonzero value, then does not do an error check to make sure that end_in is not off the end of the segment.

BLOCK_TSB <name>
A BLOCK may contain one or more BLOCK_TSB blocks delimited by "BLOCK_TSB <name>" and "END BLOCK_TSB". A BLOCK_TSB definition names a TSB (see TSB objects below) that the block should add itself to. It also defines parameters about how the block should add itself.

BLOCK_TSB
A BLOCK_TSB object defines how to add its containing block to a TSB. The name of the BLOCK_TSB object is the name of the TSB object (see below) that the block should add itself to.

TAG_BASE = num;
Number to use as the basis for TSB tags in this attr block. The virtual address is OR'd into the proper bit range in this number (using TAG_ADDR_BITS) to form the TSB tags.

DATA_BASE = num;
Basis for the TSB data entries in this attr block. The real address is OR'd into the proper bit range in this number (using DATA_ADDR_BITS) to form the TSB data entries.

START_RA = num;
Starting real address for this attr block. Must be page aligned.

PAGE_SIZE = num;
Page size used for computing number of TSB entries and for alignment checks.

VA_INDEX_BITS = hi : lo;
Bits of the virtual address used to index a TSB. This is independent of the TSB size. If the TSBs being used have non-zero size_bits, they will add the size_bits to the 'hi' value specified.

TAG_ADDR_BITS = hi : lo;
Bits of the TSB tag that should contain a portion of the virtual address.
TTE_TAG_ADDR_BITS = hi : lo;
   Bits in the TSB tag that contain the VA.

DATA_ADDR_BITS = hi : lo;
   Bits of the TSB data that should contain a
   portion of the real address.

TSB
   Defines a TSB. The ATTR blocks define which TSBs they
   want to write to use, and this holds the address
   translations for them.

START_ADDR = num;
   Physical address of where this TSB should live in
   memory.

NUM_ENTRIES = num;
   Number of entries in the TSB. This can be computed from
   SIZE_BITS for a particular MMU, but goldfinger doesn’t
   want to get in the business of interpreting processor-
   specific bit fields.

SIZE_BITS = num;
   Sizes bits from the config register. It is used in the
   index calculation.

SPLIT = binary;
   If set and num is non-zero, then makes this a split TSB.

LINKAREA = name;
   If there is a collision at any entry in the TSB, it can
   create a linked list. This parameter names a TSB_LINK
   object (see below) that should contain the linked list.
   If a collision occurs when -allow_tsb_conflicts is not
   set, however, a collision is a fatal error.

TSB_LINK
   This defines a link area. If there is a collision in the TSB, a
   linked list can be used to track the multiple entries. This is an
   object for containing that linked list.

START_ADDR = num;
   Physical address where the linked list should live.

EXIT STATUS
   The exit status will be 0 if the command succeeds. If it
   fails, it will exit with a positive exit status. The error
   codes are identical between goldfinger and midas. See
   “midas -h” for the most up to date description of the
A.4  regreport

NAME
regreport - regression report generator

SYNOPSIS
regreport <options> [directory] [list]

DESCRIPTION
regreport examines all regression *.log files for diags under regression directory
and prints report. It is called by sims for each diag. User typically calls
regreport to generate summary of regression by typing following:

regreport <regression_directory>

<options>: [default]

-1 [diag_dir]:
  Prints report for the specified or current-directory diag

-regress <output_file> <directory>:
  In regression mode, regreport writes summary status for finished
diags to a file until all diags are finished. NOTE: if
some diag doesn’t produce status, regreport,1.73 will wait forever.

-sas_only
  Verilog simulator will not run, sas only.

-regenerate
  Regenerates the status.log files in the diag directories.
  Call it from the parent dir of all diag runs e.g. 2004_04_04/

<directory> [list]
  Prints report for all diags under <directory>. <list> is 0 or more
  of simulation system (testbench) names, such as ‘core1’, ‘chip8’, etc.
  When nothing specified, all systems are included.

-simline
  Typically only 1000 last lines of sim.log will be examined.
  -simline=NNN can increase or decrease this number

-full
The whole file will be processed in regreport -l mode

-[-no]printpassed
  Does not print passed diags in detailed summary

-[-no]vlog
  Disables vlog run on failing diags. Enabled by default
  If a diag fails we run vlog on it. This is good for automation.

-debug
  Runs with debug on.

-summary
  Prints only summary

-emailaddr=<e-mail address>
  Gives an email address where regression status will be sent.

A.5  vlog

NAME
  vlog - post process verilog log file

SYNOPSIS


DESCRIPTION

vlog is called by regreport and user does not need call it directly. Supported
command options are:

-ccx
  Prints ccx related messages

-l2
  Prints l2 related messages

-dram
  Prints dram related messages

-h
  Prints out this screen
-debug
  Script debug.

-cycles
  Prints the cycles and not the time

-sort
  Sorts sim.log according to time stamps first [default is on]

-perf
  Prints all kinds of performance data - I, D miss e.t.c.

Examples:
  vlog -ccx -l2 -dram >! vlog.log
  vlog <my_path>/sim.log >! vlog.log