Reconfigurable Application-Specific Computing User's Guide

007-4718-004

CONTRIBUTORS

Compiled and edited by Terry Schultz

Chapter 2, “Altix System Overview” written by Mark Schwenden

Engineering contributions by David Anderson, Kenneth Chan, Faye Kasemset, Matthias Fouquet-Lapar, Brian Larson, Chris Lindahl, Bruce Losure, Alan Mayer, Steve Miller, Amy Mitby, Rebecca Lipon, Dick Riegner, Kaustubh Sanghani, Jason Sylvain, Teruo Utsumi, and Amir Zeineddini

Illustrated by Chrystie Danzer

Production by Terry Schultz

COPYRIGHT

© 2004, 2005, 2006, Silicon Graphics, Inc. All rights reserved; provided portions may be copyright in third parties, as indicated elsewhere herein. No permission is granted to copy, distribute, or create derivative works from the contents of this electronic documentation in any manner, in whole or in part, without the prior written permission of Silicon Graphics, Inc.

LIMITED RIGHTS LEGEND

The software described in this document is “commercial computer software” provided with restricted rights (except as to included open/free source) as specified in the FAR 52.227-19 and/or the DFAR 227.7202, or successive sections. Use beyond license provisions is a violation of worldwide intellectual property laws, treaties and conventions. This document is provided with limited rights as defined in 52.227-14.

TRADEMARKS AND ATTRIBUTIONS

Silicon Graphics, SGI, the SGI logo, and Altix are registered trademarks and NUMAflex, NUMAlink, RASC, and SGI ProPack are trademarks of Silicon Graphics, Inc., in the United States and/or other countries worldwide.

Celoxica is trademark of Celoxica Limited. Intel and Itanium are registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Linux is a registered trademark of Linus Torvalds, used with permission by Silicon Graphics, Inc. Mitrionics is a trademark of Mitrionics, Inc. Red Hat and all Red Hat-based trademarks are trademarks or registered trademarks of Red Hat, Inc. in the United States and other countries. Amplify, Synplicity, and Synplify Pro are registered trademarks of Synplicity, Inc. UNIX is a registered trademarks of the Open Group in the United States and other countries. Verilog is a registered trademark of Cadence Design Systems, Inc. Viva is a registered trademark of Starbridge Systems, Inc. Windows is a registered trademark of Microsoft Corporation in the United States and/or other countries. Xilinx is a registered trademark of Xilinx, Inc.

All other trademarks mentioned herein are the property of their respective owners.

Record of Revision

Version	Description
001	September 2004 Original publication
002	November 2004 Updated to support latest version of RASC software.
003	July 2005 Updated to support a new version of the RASC-brick with increased SRAM memory and the RASC 1.0 software release.
004	March 2006 Updated to support the blade version of SGI RASC Technology and the RASC 2.0 software release

New Features in This Guide

This revision of the *Reconfigurable Application-Specific Computing User's Guide* supports the RASC 2.0 software release.

Major Documentation Changes

Changes in this guide for this release include the following:

- Added information about useful documentation supplied by Xilinx, Inc. in "About This Guide".
- Added a new section called "Getting Started with RASC Programming" on page 1.
- Updated RASC hardware description in "An Overview of Reconfigurable Computing" on page 6.
- Added information on the RASC blade implementation in Chapter 3, "RASC Algorithm FPGA Hardware Design Guide".
- Added a section on the support of a limited set of `rasclib` calls in the sample test bench in "Simulation of rasclib Functions" on page 71.
- Added information about the `rasclib_algo_callback` function in "rasclib_algorithm_commit Function" on page 83.
- Added information about the `rasclib_cop_callback` function in "rasclib_cop_commit Function" on page 91.
- Added information on the RASC blade implementation in Chapter 5, "RASC Algorithm FPGA Implementation Guide".
- Added information about environment variables that can be used to turn on or turn off server and client debugging in "Turn Debugging On or Off" on page 124.
- Added information about Device Manager logging in "Device Manager Logging Facility" on page 127.
- Updated information in Appendix B.

Contents

Figures

Tables

About This Guide

The SGI reconfigurable application-specific software computing (RASC) program delivers scalable, configurable computing elements for the SGI Altix family of servers and superclusters.

This guide provides information about RASC and covers the following topics:

- Chapter 1, “RASC Introduction”
- Chapter 2, “Altix System Overview”
- Chapter 3, “RASC Algorithm FPGA Hardware Design Guide”
- Chapter 4, “RASC Abstraction Layer”
- Chapter 5, “RASC Algorithm FPGA Implementation Guide”
- Chapter 6, “Running and Debugging Your Application”
- Chapter 7, “RASC Examples and Tutorials”
- Appendix A, “Device Driver”
- Appendix B, “SSP Stub User’s Guide”
- Appendix C, “How Extractor Works”

Related Publications

Documents listed in this section contain additional information that might be helpful, as follows:

- "SGI Documentation" on page xx
- "Additional Documentation Sites and Useful Reading" on page xxii

SGI Documentation

The following documentation is available for the SGI Altix family of servers and superclusters and is available from the online SGI Technical Publications Library:

- *SGI ProPack 4 for Linux Start Here*
  Provides information about the SGI ProPack for Linux release including information about major new features, software installation, and product support.
- *SGI ProPack 4 for Linux Service Pack 3 Release Notes*
  Provide the latest information about software and documentation in this release. The release notes are on the SGI ProPack for Linux Documentation CD in the `root` directory, in a file named `README.TXT`.
- *Linux Device Driver Programmer's Guide- Porting to SGI Altix Systems*
  Provides information on programming, integrating, and controlling drivers.
- *Porting IRIX Applications to SGI Altix Platforms: SGI ProPack for Linux*
  Provides information about porting an application to the SGI Altix platform.
- *Message Passing Toolkit (MPT) User's Guide*
  Describes industry-standard message passing protocol optimized for SGI computers.
- *Performance Co-Pilot for IA-64 Linux User's and Administrator's Guide*
  Describes the Performance Co-Pilot (PCP) software package of advanced performance tools for SGI systems running the Linux operating system.
- *Linux Configuration and Operations Guide*
  Provides information on how to perform system configuration and operations for SGI ProPack servers.
- *Linux Resource Administration Guide*

Provides a reference for people who manage the operation of SGI ProPack servers and contains information needed in the administration of various system resource management features such as Comprehensive System Accounting (CSA), Array Services, CPU memory sets and scheduling, and the Cpuset System.

- *SGI Altix 350 System User's Guide*
  Provides an overview of the Altix 350 system components, and it describes how to set up and operate this system.
- *SGI Altix 350 Quick Start Guide*
  Guides a knowledgeable user through the installation, setup, and simple configuration of most SGI Altix 350 systems.
- *SGI Altix 3700 Bx2 User's Guide*
  This guide provides an overview of the architecture and descriptions of the major components that compose the SGI Altix 3700 Bx2 family of servers. It also provides the standard procedures for powering on and powering off the system, basic troubleshooting information, and important safety and regulatory specifications.
- *SGI Altix 4700 System User's Guide*
  This guide provides an overview of the architecture and descriptions of the major components that compose the SGI Altix 4700 family of servers. It also provides the standard procedures for powering on and powering off the system, basic troubleshooting information, and important safety and regulatory specifications.
- *Silicon Graphics Prism Visualization System User's Guide*
  Provides an overview of the Silicon Graphics Prism Visualization System components and it describes how to set up and operate this system.
- *SGIconsole 2.1 Start Here*
  Provides an introduction to SGIconsole and information about setting up and configuring SGIconsole hardware and software.
- *Console Manager for SGIconsole Administrator's Guide*
  Provides information about the Console Manager software graphical interface allows you to control multiple SGI servers, SGI partitioned systems, and large single-system image servers.
- *SGI L1 and L2 Controller Software User's Guide*
  Describes how to use the L1 and L2 controller commands at your system console to monitor and manage the SGI Altix 3000 and SGI Altix 4000 family of servers and superclusters.
- *XFS for Linux Administration*

Describes XFS, an open-source, fast recovery, journaling filesystem that provides direct I/O support, space preallocation, access control lists, quotas, and other commercial file system features.

- *XVM Volume Manager Administrator's Guide*
  Describes the configuration and administration of XVM logical volumes using the XVM Volume Manager.
- *Event Manager User Guide*
  Provides information about the Event Manger application that collects event information from other applications. This document describes the Event Manager application, the application programming interface that you can use to access it, the procedures that you can use to communicate with it from another application, and the commands that you can use to control it.
- *Embedded Support Partner User Guide*
  Provides information about using the Embedded Support Partner (ESP) software suite to monitor events, set up proactive notification, and generate reports. This revision of the document describes ESP version 3.0, which is the first version of ESP that supports the Linux operating system.
- *Linux Application Tuning Guide*
  Provides information about tuning application programs on SGI Altix systems. Application programs include Fortran and C programs written on SGI Linux systems with the compilers provided by Intel.
- *SCSL User's Guide*
  Provides information about the scientific libraries on SGI Altix systems and SGI IRIX systems. Topics include discussions about BLAS, LAPACK, and FFT routines.

Additional Documentation Sites and Useful Reading

The following sites and books may be useful:

- *Xilinx Development System Reference Guide*

  This manual provides a bitstream generation workflow diagram and detailed description of all the files generated and used in the workflow and the tools that create and use these files. It is available at http://www.xilinx.com/. Click on the **Documentation** link. Under **Design Tools Documentation**, select **Software Manuals**. For the RASC 2.0 release, reference the **6.x Softare Manuals**.
- *Verilog Quick Start, A Practical Guide to Simulation and Synthesis in Verilog*, Third Edition, James M. Lee, Copyright 2002 by Kluwer Academic Publishers

- *Verilog HDL Synthesis, A Practical Primer*, J. Bhasker, Copyright 1998 by Lucent Technologies and published by Star Galaxy Publishing.
- *A Verilog HDL Primer*, J. Bhasker, Second Edition, Copyright 1997, 1999 by Lucent Technologies and published by Star Galaxy Publishing.

Obtaining SGI Publications

You can obtain SGI documentation in the following ways:

- See the SGI Technical Publications Library at http://docs.sgi.com. Various formats are available. This library contains the most recent and most comprehensive set of online books, release notes, man pages, and other information.
- You can view man pages by typing `man` <*title*> on a command line.

Conventions

The following conventions are used throughout this publication:

Convention	Meaning
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures. Angle brackets <> are sometimes used in tables to save space.
variable	Italic typeface denotes variable entries and words or concepts being defined.
`user input`	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. (Output is shown in nonbold, fixed-space font.)
`manpage`(*x*)	Man page section identifiers appear in parentheses after man page names.
GUI element	This font denotes the names of graphical user interface (GUI) elements such as windows, screens, dialog boxes, menus, toolbars, icons, buttons, boxes, fields, and lists.

Reader Comments

If you have comments about the technical accuracy, content, or organization of this document, contact SGI. Be sure to include the title and document number of the manual with your comments. (Online, the document number is located in the front matter of the manual. In printed manuals, the document number is located at the bottom of each page.)

You can contact SGI in any of the following ways:

- Send e-mail to the following address:

  techpubs@sgi.com

- Use the Feedback option on the Technical Publications Library Web page:

  http://docs.sgi.com

- Contact your customer service representative and ask that an incident be filed in the SGI incident tracking system.

- Send mail to the following address:

  Technical Publications
  SGI
  1500 Crittenden Lane, M/S 535
  Mountain View, CA 94043-1351

SGI values your comments and will respond to them promptly.

RASC Introduction

This chapter provides an introduction to Reconfigurable Computing (RC) including challenges in design and implementation of RC systems, an introduction to SGI's system platform and an overview of SGI's Reconfigurable Application Specific Computing (RASC). It covers the following topics:

Getting Started with RASC Programming

This section provides an overview of RASC programming and covers these topics:

Overview of FPGA Programming

Currently, FPGA programming is a fairly complex task. The main FPGA programming languages are VHDL and Verilog. They require an electrical engineering background and the understanding of timing constraints; that is, the time it takes for an electrical signal

to travel on the chips, delays introduced by buffers, and so on. The blank FPGA has physical connection from, for example, the memory pins to FPGA I/O pins, but a protocol like QDR-II has to be implemented that specifies memory transfer rates for dual in-line memory modules. These low-level abstractions can almost be considered as the "BIOS" of the RASC unit. They allow an application to read a memory location without understanding the underlying hardware. SGI RASC calls this functionality core services (for more information, see “Algorithm / Core Services Block Interface” on page 28).

Currently, FPGAs are running at clock speeds of about 200 MHz. This may seem slow compared to an Itanium processor, for example; however, the FPGA can be optimized for specific algorithms and potentially performs several hundreds or thousands of operations in parallel.

Programming high-performance computing applications in VHDL and/or Verilog is extremely time-consuming and resource-intensive and probably best left to very advanced users. However, high-level tools provided by vendors such as Celoxica Limited, Mitrionics, Inc., or Starbridge Systems, Inc. are available.

These tools produce VHDL or Verilog code (potentially thousands of lines for even a small code fragment). This code then has to be synthesized (compiled) into a netlist. This netlist then is used by a place and route program to implement the physical layout on the FPGAs.

SGI FPGA Programming Approach

A summary of the SGI approach to FPGA programming is, as follows: (see Figure 1-1 on page 4 and Figure 1-2 on page 5)

- Write an application in C programming language for system microprocessor
- Identify computation intense routine(s)
- Generate a bitstream using core services and language of choice
- Create module in Mitrion-C, Handel-C, or Viva
- Verify algorithm with the appropriate toolkit
- Synthesize using tools, such as, XST, Synplify Pro, or Amplify
- Create bitstream using Xilinx ISE place and route tools
- Replace routines with RASC abstraction layer (`rasclib`) calls

- Run your application and debug with GDB (see "Helpful SGI Tools" on page 6)

The RASC tutorial (see "Tutorial Overview" on page 134) steps you through the entire RASC design flow: integrating the algorithm with core service; simulating behavior on the algorithm interfaces; synthesizing the algorithm code; generating a bitstream; transferring that bitstream and metadata to the Altix platform; executing an application; and using GDB to debug an application on the Altix system and FPGA simultaneously.

Bitstream Development Overview

In this guide, *implementation flow* (bitstream development) refers to the comprehensive run of the extractor, synthesis, and Xilinx ISE tools that turn the Verilog or VHDL source into a binary bitstream and configuration file that can be downloaded into the RASC Algorithm FPGA (for more information, see "Summary of the Implementation Flow" on page 100).

Figure 1-1 shows bitstream development on an IA-32 Linux platform for an Altix RASC hardware. The RASC Abstraction Layer (`rasclib`) provides an application programming interface (API) for the kernel device driver and the RASC hardware. It is intended to provide a similar level of support for application development as the standard open/close/read/write/ioctl calls for IO peripheral. For more on the RASC Abstraction Layer, see Chapter 4, "RASC Abstraction Layer".

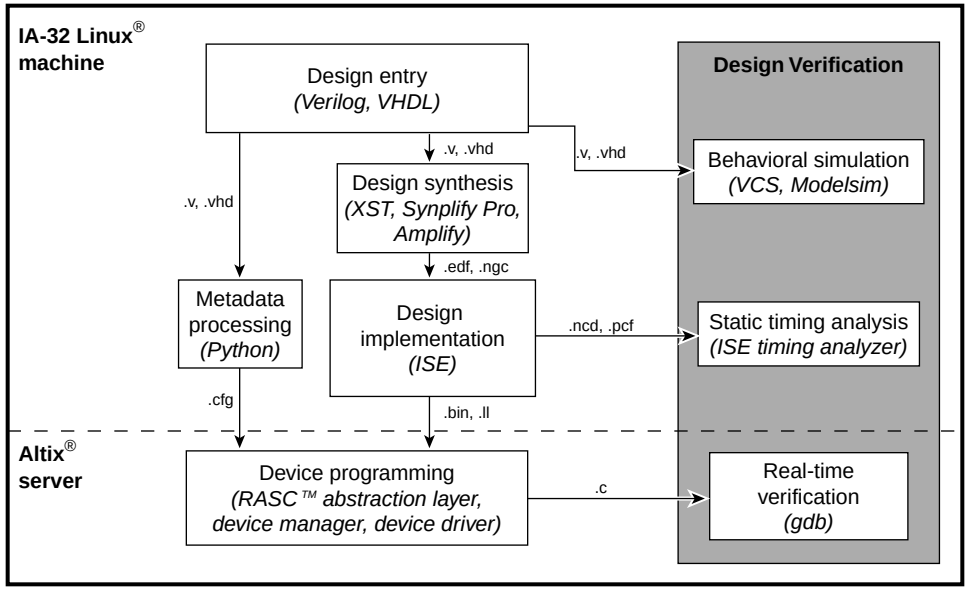

Figure 1-1 Bitstream Development

Figure 1-2 shows bitstream development using third-party development tools. (see "SGI FPGA Programming Approach" on page 2)

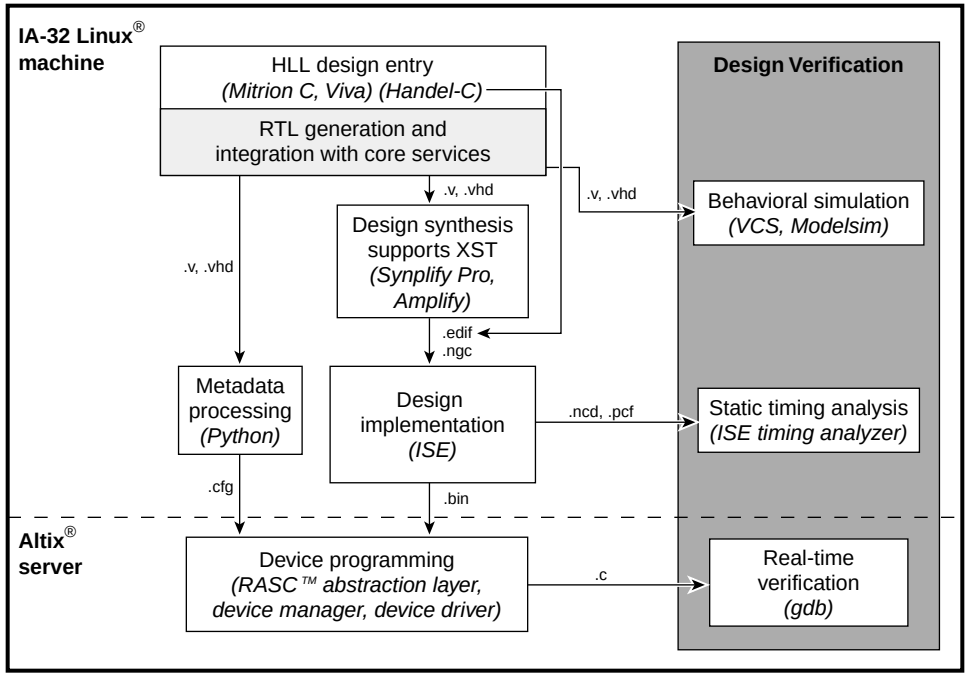

Figure 1-2 Bitstream Development with High-level Tools

Amplify and Synplify Pro are synthesis products developed by Synplicity, Inc. For information on these products, click on the **Literature** link on the left sidebar of the homepage at www.synplicity.com.

Xilinx Synthesis Technology (XST) is a synthesis product developed by Xilinx, Inc. Information on XST is available at http://www.xilinx.com/.

The *Xilinx Development System Reference Guide* provides a bitstream generation workflow diagram and detailed description of all the files generated and used in the workflow and the tools that create and use these files. From the Xilinix, Inc. homepage, Click on the **Documentation** link. Under **Design Tools Documentation**, select **Software Manuals**. For the RASC 2.0 release, reference the **6.x Software Manuals**.

For additional documentation that you may find helpful, see "Additional Documentation Sites and Useful Reading" on page xxii.

Helpful SGI Tools

SGI provides a Device Manager for loading bitstreams. It maintains a registry of algorithm bitstreams that can be loaded and executed using the RASC abstraction layer (`rasclib`). The `devmgr` user command is used to add, delete, and query algorithms in the registry. For more information on the Device Manager, see “RASC Device Manager” on page 118.

The SGI RASC environment has a product called `gdbfpga` that provides extensions to the GNU Debugger (GDB) command set to handle debugging of one or more FPGAs. An overview of GDB is, as follows:

- Based on Open Source GNU Debugger (GDB)
- Uses extensions to current command set
- Can debug host application and FPGA
- Provides notification when FPGA starts or stops
- Supplies information on FPGA characteristics
- Can "single-step" or "run N steps" of the algorithm
- Dumps data regarding the set of "registers" that are visible when the FPGA is active

For more information on debugging in the RASC environment, see “Using the GNU Project Debugger (GDB)” on page 128.

An Overview of Reconfigurable Computing

Reconfigurable computing is defined as a computer having hardware that can be reconfigured to implement application-specific functions (see Figure 1-3). The basic concept of a reconfigurable computer (RC) was proposed in the 1960s, but only in the last decade has an RC been feasible. RC systems combine microprocessors and programmable logic devices, typically field programmable gate arrays (FPGAs), into a single system. Reconfigurable computing allows applications to map computationally dense code to hardware. This mapping often provides orders of magnitude improvements in speed while decreasing power and space requirements.

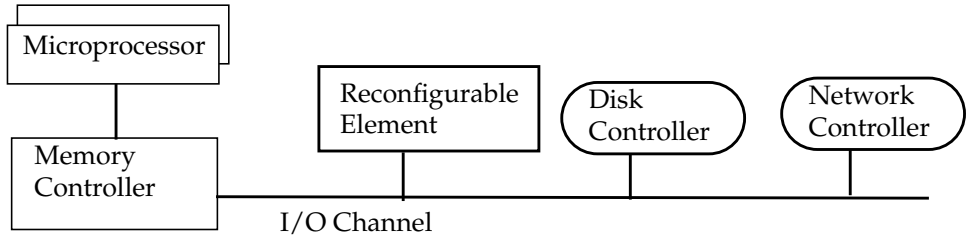

Figure 1-3 Reconfigurable Computer

As defined above, RC uses programmable FPGAs. Current FPGA technology provides more than 10 million logic elements, internal clock rates over 200MHz, and pin toggle rates approaching 10Gb/s. With these large devices, the scope of applications that can target an FPGA has dramatically increased. The challenges to using FPGAs effectively fall into two categories: ease of use and performance. Ease of use issues include the following:

- Methodology of generating the "program" or bitstream for the FPGA
- Ability to debug an application running on both the microprocessor and FPGA
- Interface between the application and the system or Application Programming Interface (API)

Performance issues include the following:

- Data movement (bandwidth) between microprocessors and FPGAs
- Latency of communication between microprocessors and FPGAs
- Scalability of the system topology

Programming a FPGA with current technology requires a hardware design engineer. Typically an algorithm is hand translated into a Hardware Description Language (HDL), verified by human-generated HDL tests, synthesized into logic elements, physically placed in the FPGA and then analyzed for speed of operation. If errors occur, these design and verification steps are repeated. This iterative process is conducive for

semiconductor chip design, but it impedes the rapid proto-typing and solution oriented goals of application-specific programming.

Debugging FPGAs requires specialized logic to be inserted into register transfer level (RTL) code. Then FPGA-specific tools in conjuction with typical microprocessor debug methods are used to analyze anomalies in application behavior. Typically, the FPGA tools and microprocessor system tools do not communicate, or worse they create roadblocks for each other.

Lastly, users are hampered by the lack of standardized application interfaces. This deficiency forces recoding when hardware or platform upgrades become available--a time-consuming and error-inducing process.

Performance is the fundamental reason for using RC systems. By mapping algorithms to hardware, designers can tailor not only the computational components, but also perform data flow optimization to match the algorithm. Today's FPGAs provide over a terabyte per second of memory bandwidth from small on chip memories as well as tens of billions of operations per second. Transferring data to the FPGA and receiving the results poses a difficult challenge to RC system designers. In addition to bandwidth, efficient use of FPGA resources requires low latency communication between the host microprocessor and the FPGAs. Achieving low latency in the presence of high bandwidth communication is one of the most difficult obstacles facing a system designer. When low latency is achieved, scaling and optimization across multiple computational elements can occur, often called load balancing.

An architectural paradigm shift is required to meet the complex problems facing high-performance computing (HPC). Although challenges abound, RC systems allow us to explore solutions that are not viable in today's limited computing environments. The benefits in size, speed and power alone make RC systems a necessity.

Silicon Graphics ccNUMA Systems

SGI was founded in 1982 based on Stanford University's research in accelerating a specific application, three dimensional graphics. SGI pioneered acceleration of graphics through hardware setting records and providing technological capabilities that were impossible without specialized computational elements. Tackling difficult problems required a supercomputer with capabilities that were not available from other computer vendors. SGI chose to develop its own large scale supercomputer with the features needed to drive graphics. The development of large scale single system image (SSI)

machines was also pioneered at SGI. From the early days systems such as Power Series, Challenge, and Power Challenge defined large shared bus SSI systems, but the focus continued to be on providing the high bandwidth and scaling that was needed to drive graphics applications. To transcend the 36 microprocessor Challenge Series systems, conventional single backplane architectures were not sufficient. SGI returned again to its roots at Stanford University and the Distributed Architecture for Shared (DASH) memory project. The original concepts for cache coherent non-uniform memory access (ccNUMA) architecture are based on the DASH memory project. The ccNUMA architecture allows for distributed memory through the use of a directory-based coherency scheme, removing the need for large common busses like those in the Challenge systems. Without the restrictions of a single bus, bandwidth for the system increased by orders of magnitude, while latency was reduced. This architecture has allowed SGI to set new records for system scalability including 1024 CPU SSI, 1TB/s Streams benchmark performance, and many others.

The SGI Altix system is the only fourth generation Distributed Shared Memory (DSM) machine using a NUMA architecture that is connected by a high–bandwidth, low–latency interconnect. In keeping with ever increasing demands, Altix allows independent scaling for CPUs, memory, Graphics Processing Units (GPU), I/O interfaces, and specialized processors. The NUMAlink interconnect allows Altix to scale to thousands of CPUs, terabytes of memory, hundreds of I/O channels, hundreds of graphics processors and thousands of application-specific devices.

SGI uses NUMAlink on all of its ccNUMA systems. NUMAlink 4 is a third generation fabric that supports topologies starting at basic rings. With the addition of routers, meshes, hypercubes, modified hypercubes and full fat tree topologies can be built. The protocol and fabric allows the topology to be matched to the workload as needed. By using the high bandwidth, low latency interconnect, SGI has a flexible and powerful platform to deliver optimized solutions for HPC.

Silicon Graphics Reconfigurable Application-Specific Computing (RASC)

The RASC program leverages more than 20 years of SGI experience accelerating algorithms in hardware. Rather than using relatively fixed implementations, such as graphics processing units (GPUs), RASC uses FPGA technology to develop a full-featured reconfigurable computer. The RASC program also addresses the ease of use and performance issues present in typical RC environments.

To address performance issues, RASC connects FPGAs into the NUMAlink fabric making them a peer to the microprocessor and providing both high bandwidth and low latency. By attaching the FPGA devices to the NUMAlink interconnect, RASC places the FPGA resources inside the coherency domain of the computer system. This placement allows the FPGAs extremely high bandwidth (up to 6.4GB/s/FPGA), low latency, and hardware barriers. These features enable both extreme performance and scalability. The RASC product also provides system infrastructure to manage and reprogram the contents of the FPGA quickly for reuse of resources.

RASC defines a set of APIs through the RASC Abstraction Layer (RASCAL). The abstraction layer can abstract the hardware to provide deep and wide scaling or direct and specific control over each hardware element in the system. In addition RASC provides a FPGA-aware version of GNU Debugger (GDB) that is based on the standard Linux version of GDB with extensions for managing multiple FPGAs. The RASC debug environment does not require learning new tool sets to quickly debug an accelerated application.

RASC supports the common hardware description languages (HDLs) for generating algorithms. RASC provides templates for Verilog- and VHDL-based algorithms. Several 3rd-party high-level language tool vendors are developing RASC interfaces and templates to use in place of traditional hardware design languages.

RASC Hardware Overview

The initial RASC hardware implementation used SGI's first generation peer attached I/O brick for the base hardware. The RASC hardware module is based on an application-specific integrated circuit (ASIC) called TIO. TIO attaches to the Altix system NUMAlink interconnect directly instead of being driven from a compute node using the XIO channel. TIO supports two PCI-X busses, an AGP-8X bus, and the Scalable System Port (SSP) port that is used to connect the Field Programmable Gate Array (FPGA) to the rest of the Altix system for the RASC program. The RASC module contains a card with the co-processor (COP) FPGA device as shown Figure 1-4.

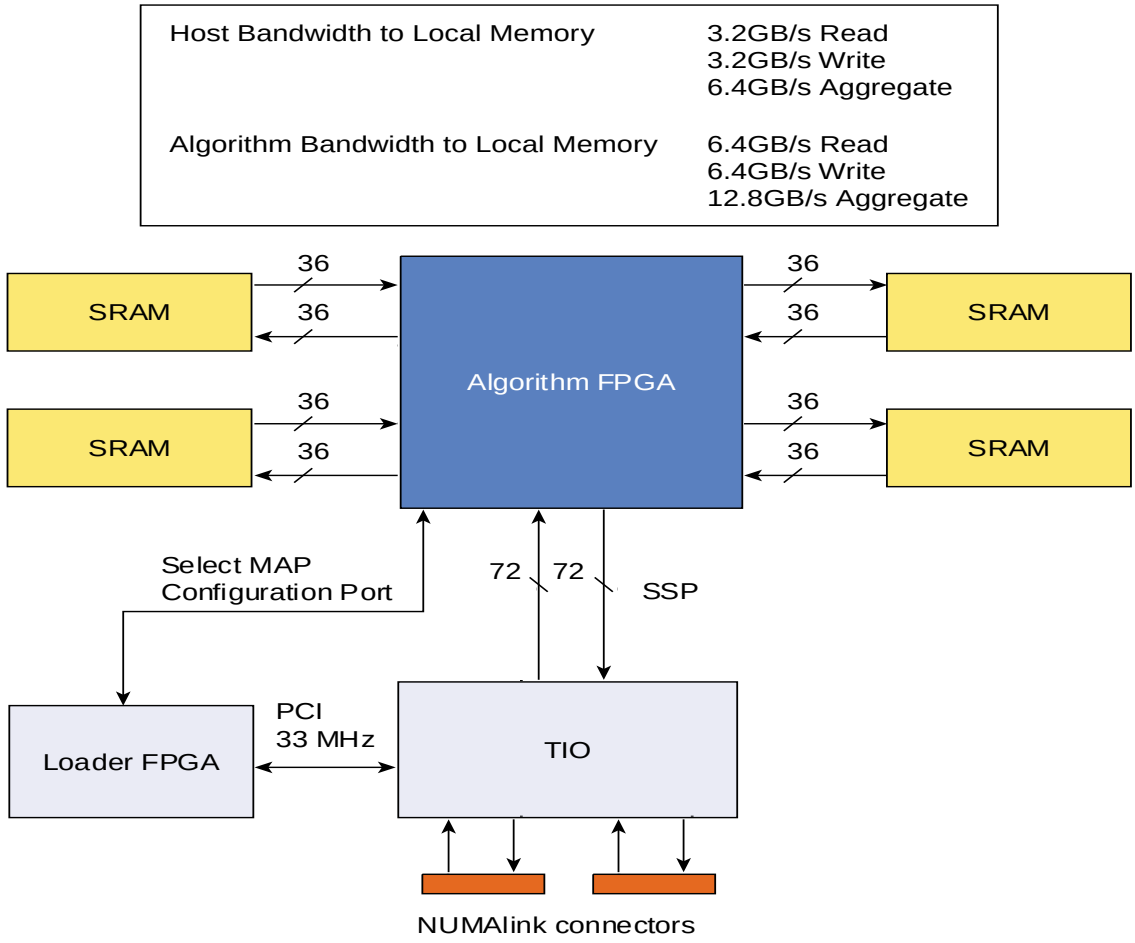

Figure 1-4 RASC FPGA Functional Block Diagram

Figure 1-5 shows a block diagram of the RASC FPGA with Core Services and the Algorithm Block. The FPGA is connected to an SGI Altix system via the SSP port on the TIO ASIC. It is loaded with a bitstream that contains two major functional blocks:

- The reprogrammable algorithm
- The Core Services that facilitate running the algorithm

For more information on the RASC FPGA, see Chapter 3, “RASC Algorithm FPGA Hardware Design Guide”.

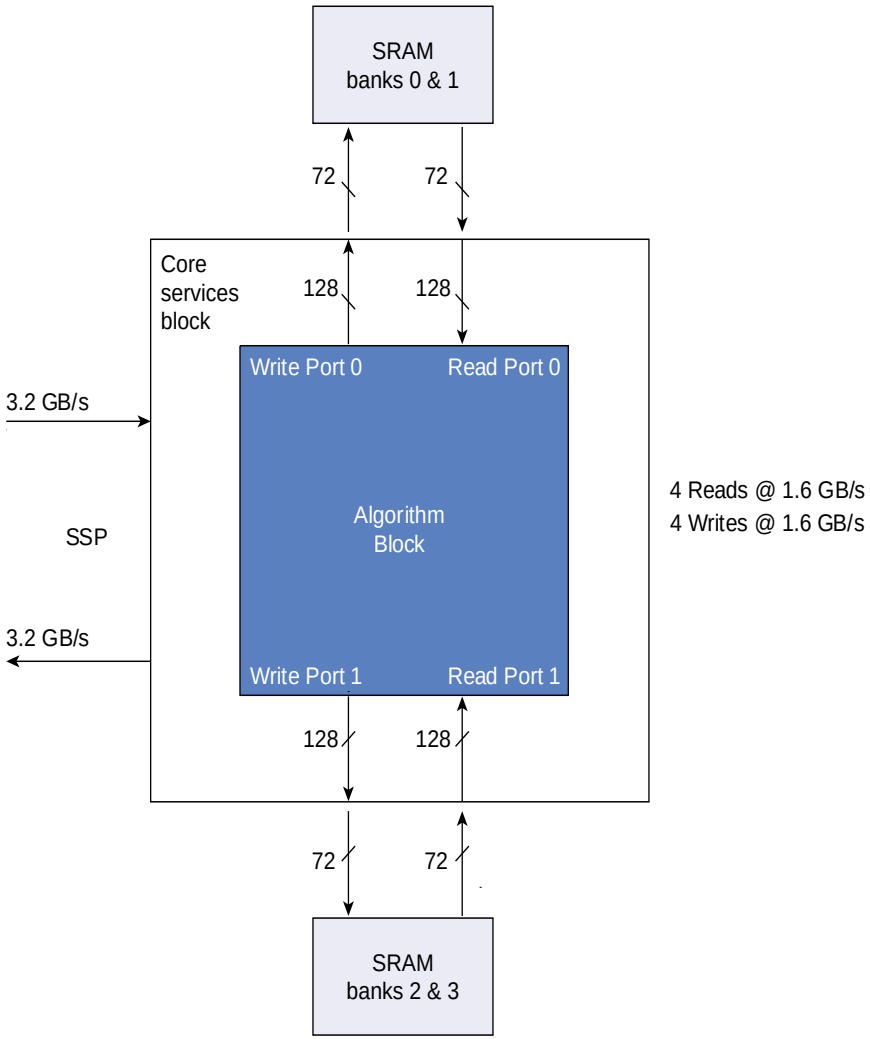

Figure 1-5 FPGA Architecture

For more information on the Altix system topology and a general overview of the Altix 350 system architecture, see Chapter 2, “Altix System Overview”.

RASC hardware implementation for SGI Altix 4700 systems is based on blade packaging as shown in Figure 1-6. For an overview of the Altix 4700 system, see “SGI Altix 4700 System Overview” on page 19.

The RASC hardware blade contains two computational FPGAs, two TIO ASICs, and a loader FPGA for loading bitstreams into the computational FPGAs. The computational FPGAs connect directly into the NUMAlink fabric via SSP ports on the TIO ASICS. The new RASC blade has high-performance FPGAs with 160K logic cells and increased

memory resources with 10 synchronous static RAM dual in-line memory modules (SSRAM DIMMs).

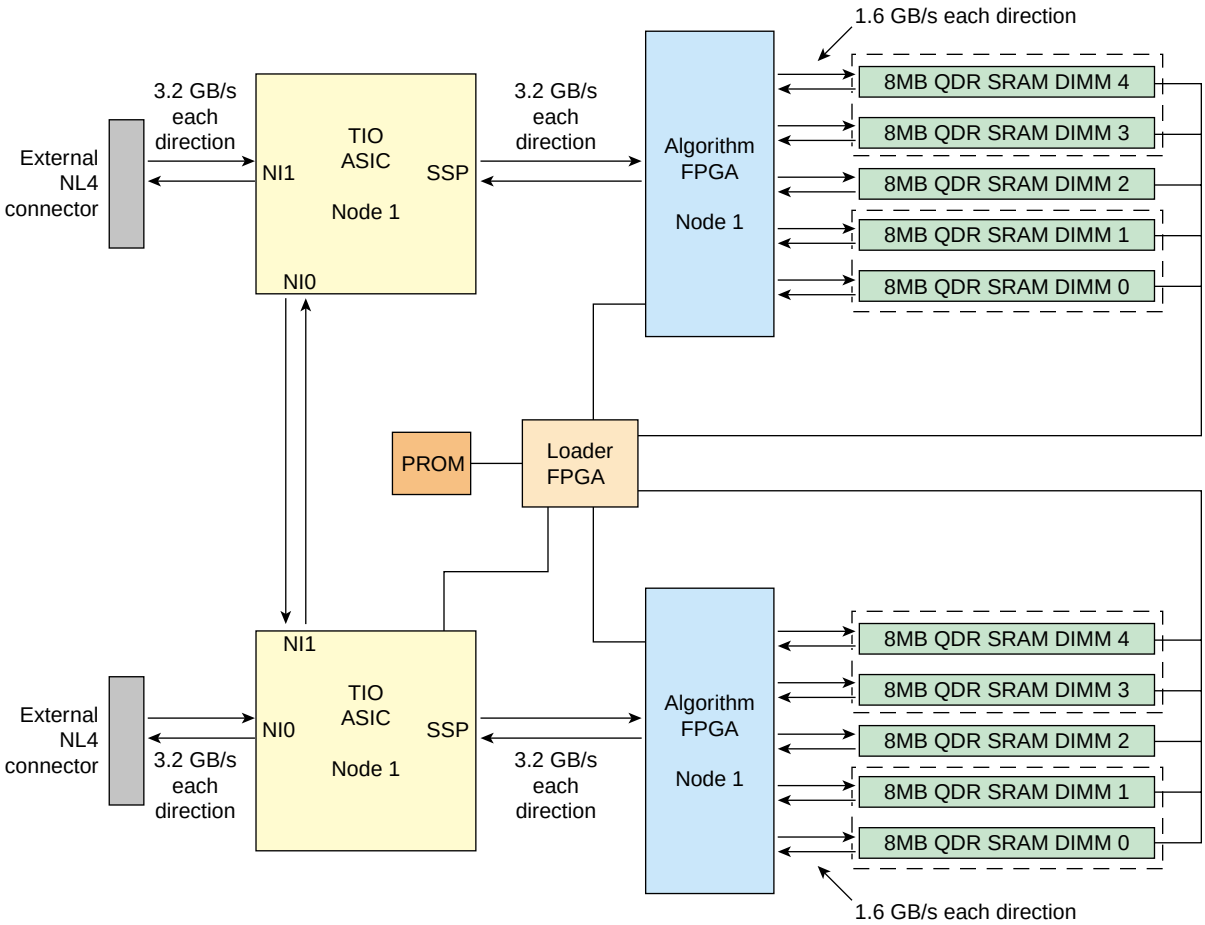

Figure 1-6 RASC Blade Hardware

For legacy systems, optional brick packaging is available for the latest RASC hardware.

RASC Software Overview

Figure 1-7 shows an overview of the RASC software.

Major software components are, as follows:

- Standard Linux GNU debugger with FPGA extensions
- Download utilities
- Abstraction layer library
- Device Manager
- Algorithm device driver
- Download driver
- Linux kernel
- COP (TIO, Algorithm FPGA, memory, download FPGA)

This software is described in detail in this manual in the chapters that follow.

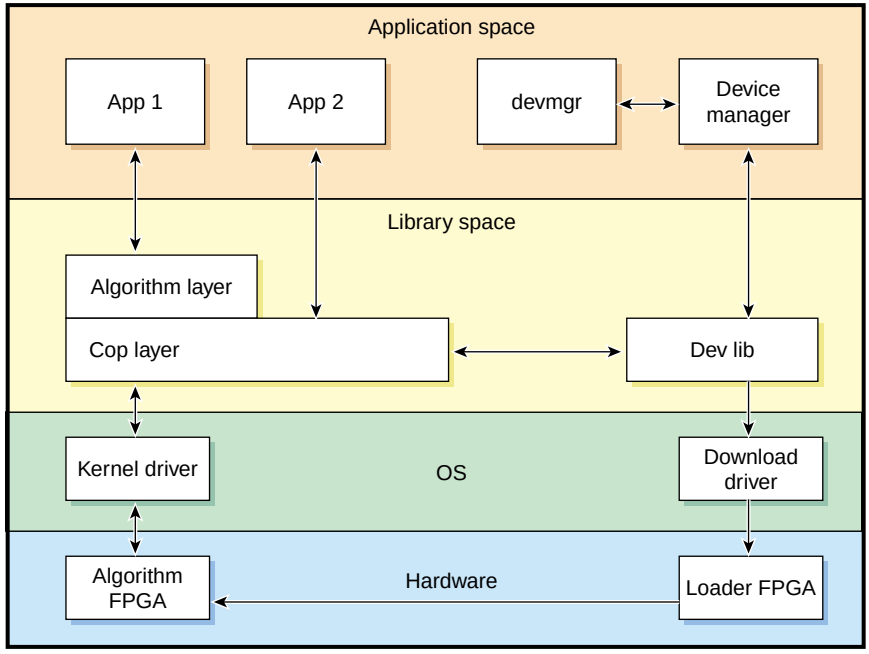

Figure 1-7 RASC Software Overview

Altix System Overview

This chapter provides an overview of the physical and architectural aspects of your SGI Altix 350 system or SGI Altix 4700 system. This chapter includes the following sections:

Note: This chapter provides an overview of the SGI Altix 350 system. For more information on this system, see the *SGI Altix 350 System User's Guide* available on the SGI Technical Publications Library. It provides a detailed overview of the SGI Altix 350 system components and it describes how to set up and operate the system. You can find detailed information about the Silicon Graphics Prism Visualization System based on the Altix platform, in the *Silicon Graphics Prism Visualization System User's Guide* or the *Silicon Graphics Prism Deskside Visualization System User's Guide*. For an overview of SGI ProPack software and installation and upgrade information, see the *SGI ProPack 4 for Linux Start Here*.

SGI Altix 350 System Overview

The Altix 350 system advances the SGI NUMAflex approach to mid-range modular computing. It is designed to deliver maximum sustained performance in a compact system footprint. Independent scaling of computational power, I/O bandwidth, and in-rack storage lets you configure a system to meet your unique computational needs. The small footprint and highly modular design of the Altix 350 system makes it ideal for computational throughput, media streaming, or complex data management.

The Altix 350 system can be expanded from a standalone single-module system with 2GB of memory and 4 PCI/PCI-X slots to a high-performance system that contains 32 processors, one or two routers, up to 192 GB of memory, and 64 PCI/PCI-X slots. For most configurations, the Altix 350 system is housed in one 17U rack or one 39U rack as

shown in Figure 2-1; however, for small system configurations, the Altix 350 system can be placed on a table top.

Systems that are housed in 17U racks have a maximum weight of approximately 610 lb (277 kg). The maximum weight of systems that are housed in 39U racks is approximately 1,366 lb (620 kg). The racks have casters that enable you to remove the system from the shipping container and roll it to its placement at your site.

See Chapter 1, “Installation and Operation,” in the *SGI Altix 350 System User’s Guide* for more information about installing your system. Check with your SGI service representative for additional physical planning documentation that may be available.

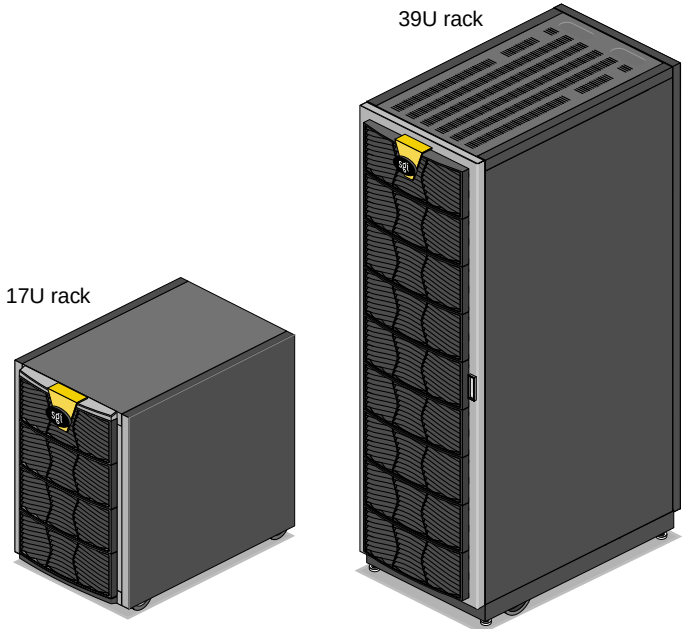

Figure 2-1 Example of SGI Altix 350 Rack Systems

The Altix 350 system is based on the SGI NUMAflex architecture, which is a shared-memory system architecture that is the basis of SGI HPC servers and supercomputers. The NUMAflex architecture is specifically engineered to provide technical professionals with superior performance and scalability in a design that is easy to deploy, program, and manage. It has the following features:

Shared access of processors, memory, and I/O. The Super Hub (SHub) ASICs and the NUMAlink-4 interconnect functions of the NUMAflex architecture enable applications to share processors, memory, and I/O devices.

- Each SHub ASIC in the system acts as a memory controller between processors and memory for both local and remote memory references.
- The NUMAlink interconnect channels information between all the modules in the system to create a single contiguous memory in the system of up to 384 GB and enables every processor in a system direct access to every I/O slot in the system.

Together, the SHub ASICs and the NUMAlink interconnect enable efficient access to processors, local and remote memory, and I/O devices without the bottlenecks associated with switches, backplanes, and other commodity interconnect technologies.

System scalability. The NUMAflex architecture incorporates a low-latency, high-bandwidth interconnect that is designed to maintain performance as you scale system computing, I/O, and storage functions. For example, the computing dimension in some system configurations can range from 1 to 32 processors in a single system image (SSI).

Efficient resource management. The NUMAflex architecture is designed to run complex models and, because the entire memory space is shared, large models can fit into memory with no programming restrictions. Rather than waiting for all of the processors to complete their assigned tasks, the system dynamically reallocates memory, resulting in faster time to solution.

SGI Altix 4700 System Overview

Note: This chapter provides a brief overview of the SGI Altix 4700 series system. For more information on this system, see the *SGI Altix 4700 System User's Guide* available on the SGI Technical Publications Library. It provides a detailed overview of the SGI Altix 4700 system components and it describes how to set up and operate the system.

The Altix 4700 series is a family of multiprocessor distributed shared memory (DSM) computer systems that initially scale from 16 to 512 Intel 64-bit processors as a cache-coherent single system image (SSI). Future releases will scale to larger processor

counts for single system image (SSI) applications. Contact your SGI sales or service representative for the most current information on this topic.

In a DSM system, each processor board contains memory that it shares with the other processors in the system. Because the DSM system is modular, it combines the advantages of low entry-level cost with global scalability in processors, memory, and I/O. You can install and operate the Altix 4700 series system in a rack in your lab or server room. Each 42U SGI rack holds from one to four 10U high enclosures that support up to ten processor and I/O sub modules known as "blades." These blades are single printed circuit boards (PCBs) with ASICS, processors, and memory components mounted on a mechanical carrier. The blades slide directly in and out of the Altix 4700 1RU enclosures. Each individual rack unit (IRU) is 10U in height (see Figure 2-2).

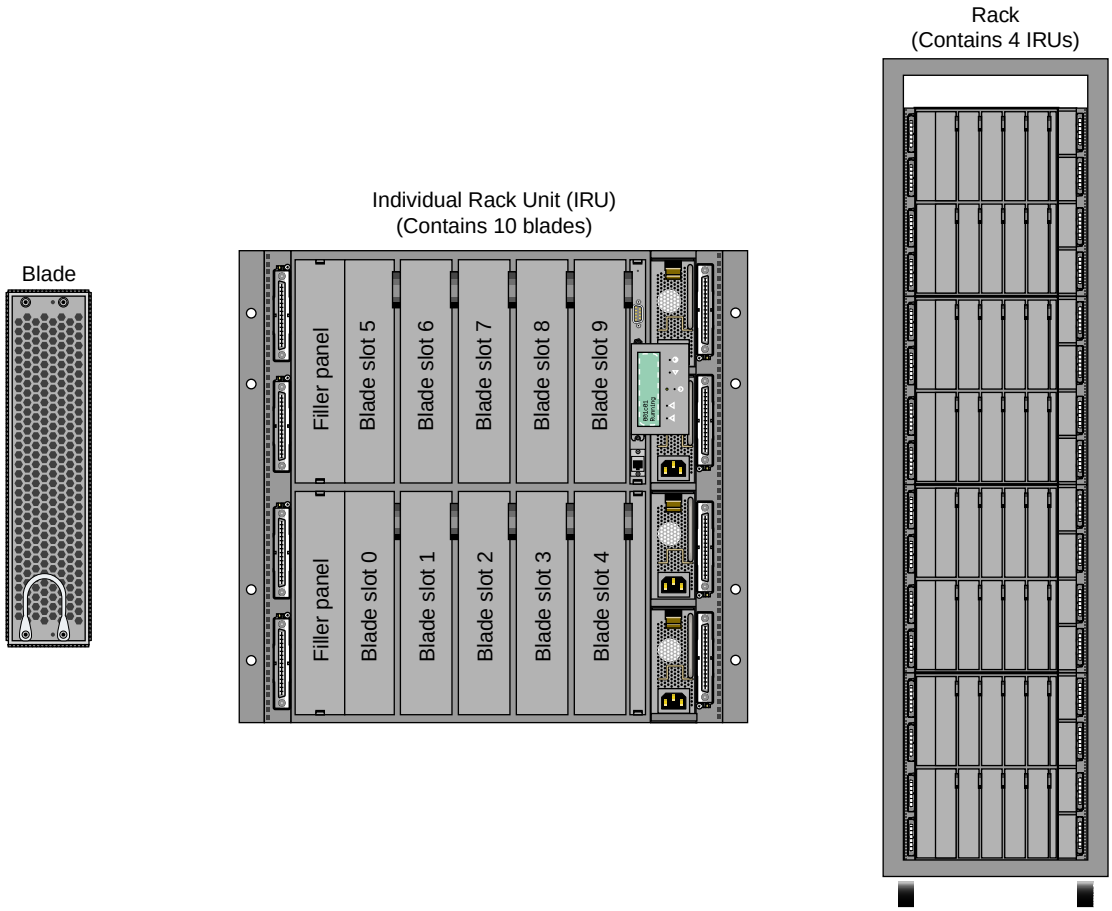

Figure 2-2 Altix 4700 Blade, Individual Rack Unit, and Rack

Note: An Altix 4700 system can support up to eight RASC blades per single system image. Current configuration rules require two compute blades for every RASC blade in your system.

The Altix 4700 computer system is based on a distributed shared memory (DSM) architecture. The system uses a global-address-space, cache-coherent multiprocessor that

scales up to sixty four Intel 64-bit processors in a single rack. Because it is modular, the DSM combines the advantages of lower entry cost with the ability to scale processors, memory, and I/O independently to a maximum of 512 processors on a single-system image (SSI). Larger SSI configurations may be offered in the future, contact your SGI sales or service representative for information.

The system architecture for the Altix 4700 system is a fourth-generation NUMAflex DSM architecture known as NUMAlink-4. In the NUMAlink-4 architecture, all processors and memory are tied together into a single logical system with special crossbar switches (routers). This combination of processors, memory, and crossbar switches constitute the interconnect fabric called NUMAlink. There are four router switches in each 10U IRU enclosure.

The basic expansion building block for the NUMAlink interconnect is the processor node; each processor node consists of a Super-Hub (SHub) ASIC and one or two 64-bit processors with three levels of on-chip secondary caches. The Intel 64-bit processors are connected to the SHub ASIC via a single high-speed front side bus.

The SHub ASIC is the heart of the processor and memory node blade technology. This specialized ASIC acts as a crossbar between the processors, local SDRAM memory, the network interface, and the I/O interface. The SHub ASIC memory interface enables any processor in the system to access the memory of all processors in the system. Its I/O interface connects processors to system I/O, which allows every processor in a system direct access to every I/O slot in the system.

Another component of the NUMAlink-4 architecture is the router ASIC. The router ASIC is a custom designed 8-port crossbar ASIC. Using the router ASICs with a highly specialized backplane or NUMAlink-4 cables provides a high-bandwidth, extremely low-latency interconnect between all processor, I/O, and other option blades within the system.

RASC Algorithm FPGA Hardware Design Guide

This chapter describes how to implement the algorithm that you have identified as a candidate for acceleration and provides a hardware reference for the RASC Algorithm Field Programmable Gate Array (FPGA) hardware. It covers the following topics:

RASC FPGA Overview

This section provides an overview of the RASC Algorithm FPGA hardware. It covers the following topics:

RASC FPGA Block Diagram

Figure 3-1 shows a block diagram of RASC Scalable System Port (SSP) Field Programmable Gate Array (FPGA) with Core Services and Algorithm Block.

The RASC SSP FPGA is a Xilinx Virtex 4 LX200 part (XC4VLX200-FF1513-10). It is connected to an SGI Altix system via the SSP port on the TIO ASIC and loaded with a bitstream that contains two major functional blocks:

- The reprogrammable Algorithm Block
- The Core Services Block that facilitates running the algorithm.

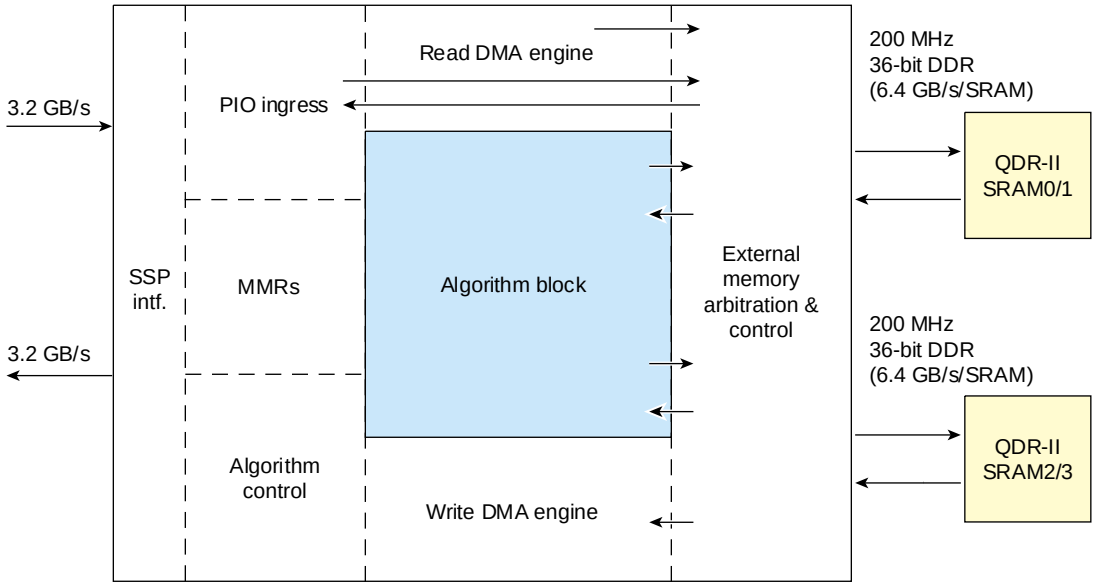

Figure 3-1 Block Diagram of the RASC Algorithm FPGA

Algorithm Block View

Figure 3-2 shows the algorithm view. The Algorithm Block sees two memory resources, each with independent read and write ports: up to 1M words deep and 128-bits wide per bank (up to 16 MB total per port per bank). It also sees a set of control and status flags on its Core Services Interface. Not shown in the diagram are the debug register outputs and algorithm defined memory-mapped register (MMR) inputs. Details are provided in the "Algorithm Design Details" on page 35.

\

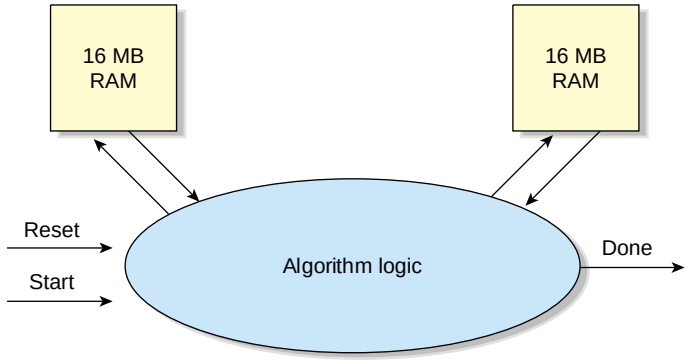

Figure 3-2 Simple Algorithm View

Core Services Features

This section highlights the services provided by the Core Services Block of the RASC FPGA. These features include:

- Scalable System Port (SSP) Implementation: physical interface and protocol
- Global Clock Generation and Control
- Two sets of independent read and write ports to each of the two random access memories (SRAMs)
- Single-step and multi-step control of the Algorithm Logic
- Independent direct memory access (DMA) engines for read and write data
- Peer Input/Output (PIO) access to algorithm's Debug port and algorithm defined registers
- Control and Status registers
- Host and FPGA process synchronization capabilities, including interrupts and atomic memory operations (AMOs).

Algorithm Run Modes

The Core Services Logic provides the mechanism for application and debugger software to run the Algorithm Block in one of two modes: ***Normal Mode*** or ***Debug Mode***. In Normal Mode, the algorithm is enabled to run and allowed to run to completion uninterrupted. In Debug Mode, the algorithm is enabled to run but forced to execute in blocks of one or more steps, allowing the user to stall the algorithm at intermediate points in the execution and query intermediate internal values. A step could be as small as one Algorithm Block clock cycle, or an adhoc size defined by Algorithm Block logic. For a description of how the Algorithm Block can use the debug mode hooks, see the "Algorithm / Core Services Block Interface" on page 28.

Algorithm / Core Services Block Interface

This section defines the interface signals between the Algorithm Block and the Core Services Block. It covers the following topics:

The Algorithm Block (top level is `alg_block_top.v/.vhd`) has three groups of interface signals to the Core Services Logic.

- General Algorithm Control Interface
- External Memory Interface
- Debug Port and Optional Algorithm Defined Registers (Algorithm-defined memory mapped space)

The associated interface signals for all three interfaces are shown in Figure 3-3, and are discussed in detail in the following sections. Not all external memory port signals are shown; only the signals for SRAM Bank 0 are included. The signals for SRAM Bank 1 are identical to those for Bank 0.

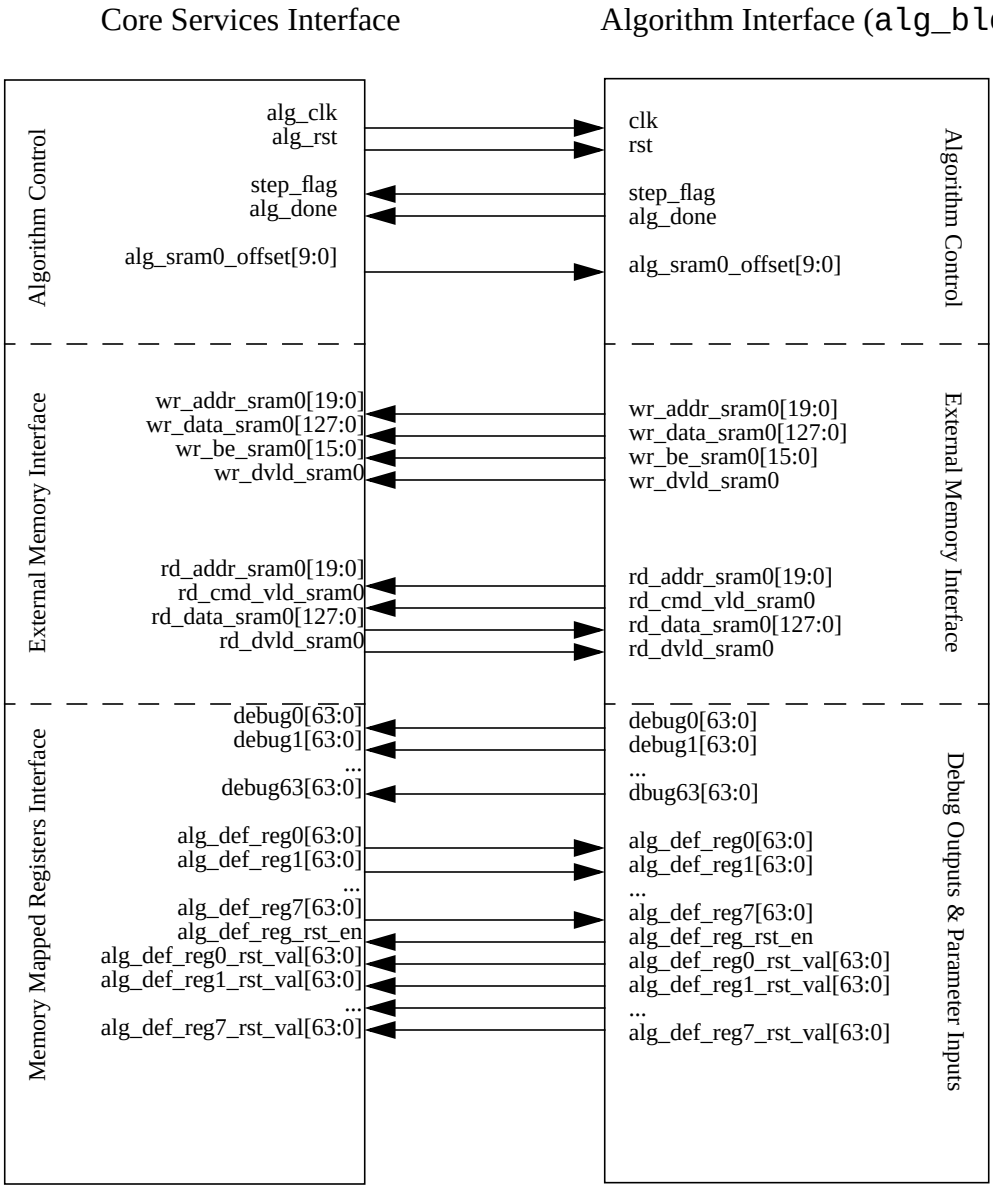

Figure 3-3 Algorithm / Core Services Interface Diagram

General Algorithm Control Interface

The General Algorithm Control Interface provides the algorithm with a clock, and provides reset, triggering, and stepping control. The signals on this interface are listed below. In/out is with respect to the Algorithm Block.

Table 3-1 General Algorithm Control Interface

Signal Name	in/out	Functional Description
clk	input	50/66/100/200 MHz gated clock, synchronous and phase-aligned to the core_clk used in the Algorithm Block. The clock frequency is determined by a macro defined at synthesis of the FPGA bitstream. The clock is driven on global clock buffers for low skew routing.
alg_rst	input	Reset the algorithm logic. The algorithm logic is always put into reset for 4 clock cycles before triggering it to execute. Active high.
step_flag	output	Step boundary indicator flag. For every clock cycle that step_flag is asserted, it signals to Core Services that one step has been completed. The Core Services Logic ignores this signal when the algorithm is not run in debug mode.
alg_done	output	Set when the algorithm has finished processing. This can either be set and held until the algorithm is reset or pulsed. When alg_done is asserted, **clk** will become inactive the following clock cycle. The signal alg_done must clear when alg_rst is asserted.
alg_sram0_offset[9:0]	input	Used in streaming-capable algorithms; sets the base of the segment address (upper 10 bits of 20-bit quad-word address) for input or output data residing on SRAM Bank 0. For details, see “Designing an Algorithm for Streaming” on page 43.
alg_sram1_offset[9:0]	input	Used in streaming-capable algorithms; sets the base of the segment address (upper 10 bits of 20-bit quad-word address) for input or output data residing on SRAM Bank 1. For details, see “Designing an Algorithm for Streaming” on page 43.

External Memory Interface

The External Memory Interface provides the algorithm with a simple interface to the read and write ports of two banks of SRAMs connected to the FPGA. Each SRAM bank provides read and write access of 16 bytes per clock cycle. The External Memory Controller encapsulates the physical layer to the SRAMs and arbitration between the other Core Services processes that can access the SRAM (DMA engines, PIO engine).

Each SRAM bank is independent from the other, and each direction (read/write) is independent, so a total of two writes and two reads can be performed simultaneously to different addresses. It is possible for the Algorithm Block to be operating on one SRAM while a DMA engine operates on another. It is also possible to have the Algorithm Logic write to a SRAM while a DMA engine (or PIO controller) reads from the same SRAM.

Access to the SRAM ports by the Algorithm Block is specified by macros defined at the synthesis of the FPGA bitstream.

The two write ports and two read ports accessible to the Algorithm Block are detailed in the tables below. In/out is with respect to the Core Services Logic. Only the signals for SRAM Bank 0 are shown. Those for SRAM Bank 1 are identical.

Table 3-2 External SRAM Bank 0 Write Interface

Signal Name	In/Out	Functional Description
wr_addr_sram0[19:0]	output	SRAM address for an Algorithm write. This address selects a quad-word (128-bit) value. Each SRAM bank is up to 16MB, or 1M quad-words.
wr_data_sram0[127:0]	output	128-bit write data.
wr_be_sram0[15:0]	output	Active high byte enables for 128-bit write data.
wr_dvld_sram0	output	Command to generate a write request to SRAM Bank 0, with the address, data, and byte enables specified.

Table 3-3 External SRAM Bank 0 Read Interface

Signal Name	In/Out	Functional Description
rd_addr_sram0[19:0]	output	SRAM address for an Algorithm Block read. This address selects a quad-word (128-bit) value. Each SRAM bank is up to 16MB, or 1M quad-words
rd_cmd_vld_sram0	output	Read Request Valid - command to generate a read request to SRAM Bank 0 with the address specified.
rd_data_sram0[127:0]	input	128-bit read data.
rd_dvld_sram0	input	Signals that the read data is valid.

Debug Port Interface

The Algorithm Block makes its internal signals visible to the GNU debugger software by connecting them to one or more debug output signals. The Algorithm Block can tie any internal signal or group of signals up to sixty-four, 64-bit debug outputs.

Table 3-4 Debug Port Interface

Signal Name	In/Out	Functional Description
debug0[63:0]	output	Debug signal 0, drives host memory-mapped debug register address 0.
debug1[63:0]	output	Debug signal 1, drives host memory-mapped debug register address 1.
debugN[63:0]	output	Debug signal N, drives host memory-mapped debug register address N (N = {0-63}).
debug63[63:0]	output	Debug signal 63, drives host memory-mapped debug register address 63.

In all provided examples, the algorithm designer is required to dedicate debug register 0 to indicate the algorithm identification number and revision number.

Table 3-5 Debug Register 0 Fields Used in Sample Algorithms

Bits	Access	Reset Value	Field Name
31:0	RO	Tied	Algorithm Revision Number
63:32	RO	Tied	Algorithm Identification Number

Optional Algorithm Defined Registers

There are eight 64-bit wide software-write / hardware-read control registers with a signal indicating whether reset should be used on these registers or not. When not in use, the Algorithm Block should tie all outputs to either one or zero.

The signals on this interface include the current register values from the Core Services Block, the desired reset values from the Algorithm The use of these additional control registers is determined by algorithm needs. One suggested use for them is to pass small parameters to the algorithm that can change for each run.

Table 3-6 Optional Algorithm Defined Registers

Signal Name	In/Out	Functional Description
alg_def_reg0 [63:0]	input	Current value of Algorithm Defined Register 0
alg_def_reg1 [63:0]	input	Current value of Algorithm Defined Register 1
alg_def_reg2 [63:0]	input	Current value of Algorithm Defined Register 2
alg_def_reg3 [63:0]	input	Current value of Algorithm Defined Register 3
alg_def_reg4 [63:0]	input	Current value of Algorithm Defined Register 4
alg_def_reg5 [63:0]	input	Current value of Algorithm Defined Register 5
alg_def_reg6 [63:0]	input	Current value of Algorithm Defined Register 6
alg_def_reg7 [63:0]	input	Current value of Algorithm Defined Register 7

Table 3-6 Optional Algorithm Defined Registers **(continued)**

Signal Name	In/Out	Functional Description
alg_def_reg_rst_en	output	Enables / disables reset function of the Algorithm Defined Registers (active high). This signal should be tied high or low. When alg_def_reg_rst_en=1, all Algorithm Defined Registers will be reset to the specified reset values on **core_rst** (not **alg_rst**). When alg_def_reg_rst_en=0, reset will not affect the values of the Algorithm Defined Registers.
alg_def_reg0_rst_val[63:0]	output	Reset value to use for Algorithm Defined Register 0
alg_def_reg1_rst_val[63:0]	output	Reset value to use for Algorithm Defined Register 1
alg_def_reg2_rst_val[63:0]	output	Reset value to use for Algorithm Defined Register 2
alg_def_reg3_rst_val[63:0]	output	Reset value to use for Algorithm Defined Register 3
alg_def_reg4_rst_val[63:0]	output	Reset value to use for Algorithm Defined Register 4
alg_def_reg5_rst_val[63:0]	output	Reset value to use for Algorithm Defined Register 5
alg_def_reg6_rst_val[63:0]	output	Reset value to use for Algorithm Defined Register 6
alg_def_reg7_rst_val[63:0]	output	Reset value to use for Algorithm Defined Register 7

Algorithm Design Details

This section provides information that the algorithm designer needs to implement the hardware accelerated algorithm within the FPGA and the Core Services Block framework. It covers the following topics:

Basic Algorithm Control

This section covers the general algorithm control sequence during a normal algorithm run (a run without breakpoints). Figure 3-4 illustrates such a sequence. The algorithm clock begins to toggle and the algorithm is put into reset for 4 algorithm clock cycles before triggering a new iteration of the algorithm.

When the algorithm is done, it should pulse its **alg_done** output. Once asserted, one more algorithm clock cycle will be generated. At this point, the algorithm is done and no further algorithm clock pulses will be generated for this iteration. The user can then probe the final internal Algorithm Block signals. The next time the algorithm is triggered by software, the reset sequence will start all over again. When there are no breakpoints, the activity of the **step_flag** signal is ignored. This particular example holds **step_flag** high, the method for clock based stepping.

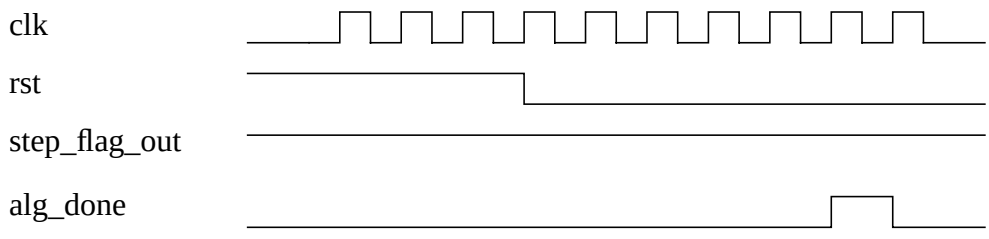

Figure 3-4 Example of a Continuous, Normal Mode Algorithm Run

An overview of the hardware algorithm design steps is presented in Figure 3-5. The shaded steps are covered in this section.

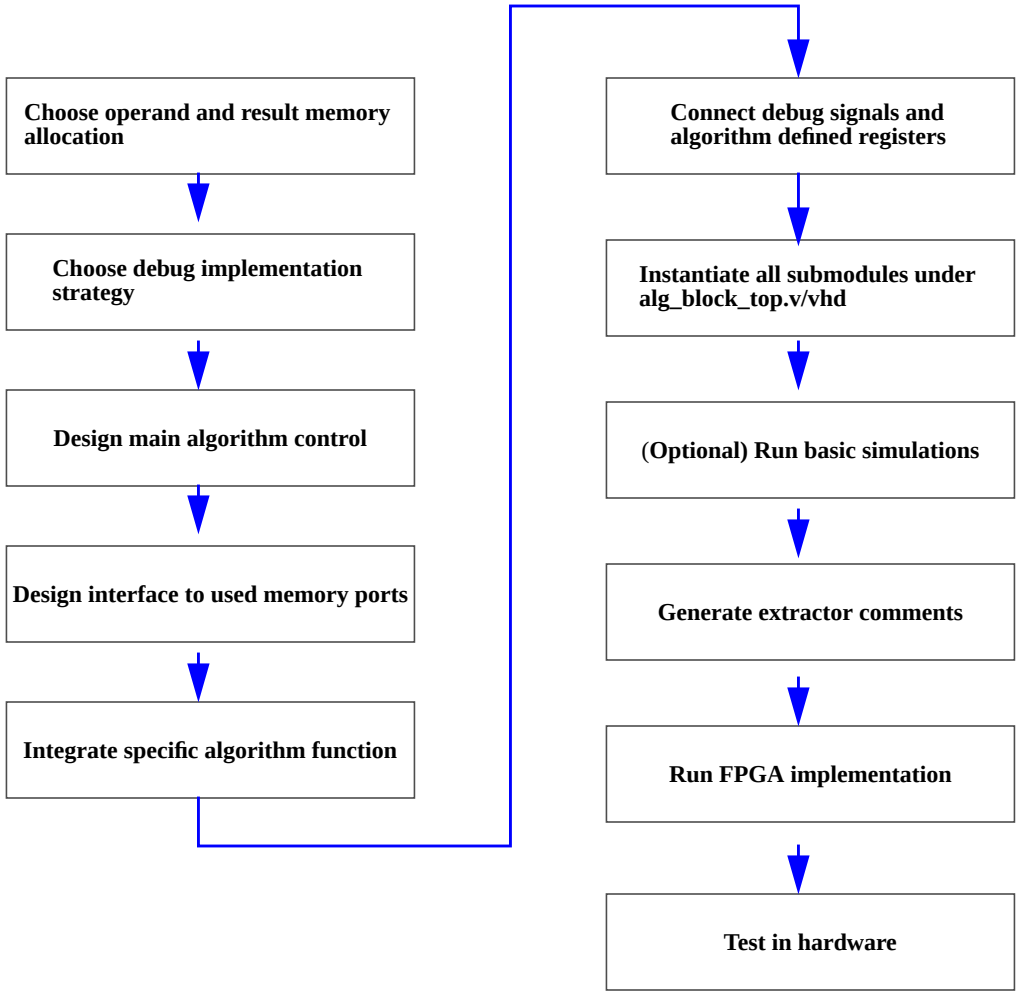

Figure 3-5 Hardware Accelerated Algorithm Design Flow

Recommendations for Memory Distribution

The RASC FPGA gives the algorithm access to two banks of up to 16MB SRAM. This section discusses the considerations for algorithm designers when deciding how to distribute input operands and parameters and output results among the available SRAM banks.

Input and Output Placement

The primary recommendation for data distribution is to organize algorithm inputs and outputs on separate SRAM banks. In other words, if bank 0 is used for input data, it should not also be used for output data and vice versa (by splitting the SRAM into two logical halves, for example). The motivation for this guideline comes from the fact that when an algorithm accesses a particular bank's read or write port, it blocks access to the DMA engine that wants to unload finished results or load new operands.

To avoid multiple arbitration cycles that add to read and write latency, the algorithm is automatically given access to the ports while the algorithm is active.

In order for the hardware accelerated algorithm to run efficiently on large data sets, it is recommended to overlap data loading and unloading with algorithm execution. To do this successfully, the algorithm designer needs to start with an SRAM layout of algorithm operands and results that allows each bank to be simultaneously accessed by the Algorithm for one direction (read or write) and a DMA engine for the other direction (write or read).

Implementation Options for Debug Mode

The Algorithm can implement two different forms of Debug Mode, based on convenience or the desired granularity of step size: clock cycle based stepping or variable (ad hoc) stepping. The differences between the two determine the step size, or how long the algorithm will run when triggered to step once. There are also implementation differences for the step size variants. Currently, only one type of debug mode is supported at a time.

Clock Cycle Step Size Mode

Clock cycle based stepping means that the step size is one **clk** cycle. This method is easily implemented in RTL-based algorithms by tying the **step_flag** output to one (logic high). The step counter used by the debugger is 16 bits, so the maximum number of clock cycles that can be stepped in one debugger command is 2^16-1 = 65,535.

Note that since the Algorithm Block cannot detect when **clk** has stopped, the effect of stepping is transparent to the Algorithm Block.

An example of this mode is shown in Figure 3-6.

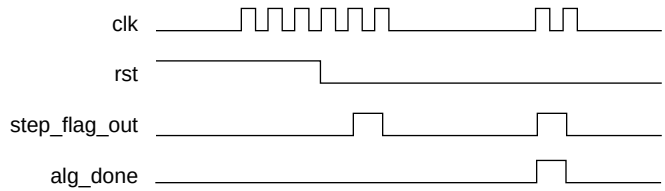

Figure 3-6 Clock Cycle Stepping Mode Example

Variable Step Size Mode

Another approach for implementing debugging is to assert **step_flag** at points of interest rather than every clock cycle, which makes the step size a variable number of clock cycles. One example of this method would be to use **step_flag** as an output of the last state of the FSM. Another example would be for the user to put in a "trigger" for when an internal counter or state machine reaches a specific value (with an indeterminate number of clock cycle steps in between). In this case, **step_flag** is tied to the trigger so that the algorithm can break at a designated point.

The ad hoc nature of this approach requires the Algorithm to define and notify Core Services of step boundaries with the **step_flag** signal. The **clk** signal will not stop toggling during the same clock cycle that the **step_flag** signal is asserted; it will turn off on the following clock cycle. See the timing diagram in Figure 3-7.

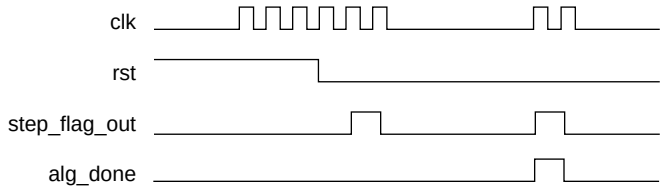

Figure 3-7 Variable Step Size Mode Example

External Memory Write Transaction Control

The process of using a write port involves the following step (example given for SRAM0 alone):

When the address, data and byte enables are valid for a write, assert **wr_dvld_sram0** (reoccurring phase)

Example Write Transaction Timing Diagram

Figure 3-8 shows single and back-to-back write commands.

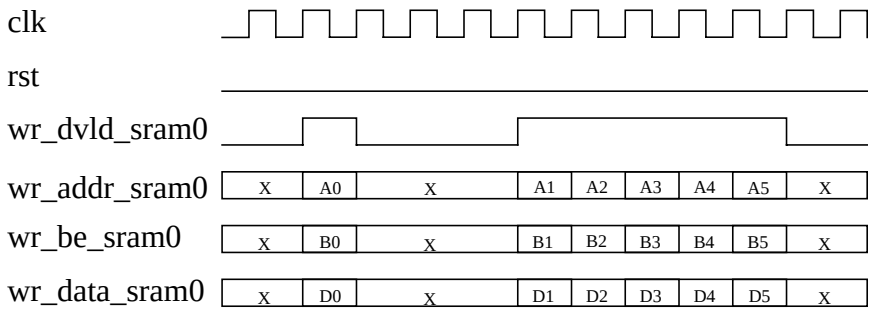

Figure 3-8 Single, and Multiple Write Commands

External Memory Read Transaction Control

The process of using a read port involves the following steps (example given for SRAM0 alone):

1. When the address is valid for a read, assert **rd_cmd_vld_sram0** (reoccurring phase). This step can be repeated while waiting for data (back-to-back burst reads). The Algorithm can issue one quad-word (16 byte) read command every **clk** cycle per bank.
2. The read data will return on the bus **rd_data_sram0[127:0]** several clock cycles later in the order it was requested (**rd_dvld_sram0** indicates that the read data is valid). The observed read latency from the Algorithm's perspective will vary based the

clock period ratio between **alg_clk** and **core_clk**. Read latency is nominally 10 core_clk cycles; burst read commands are recommended for optimal read performance.

The algorithm should use the provided read data valid signal to determine when read data is valid and not attempt to count data cycles for expected latency.

Example Read Transaction Timing Diagram

Figure 3-9 and Figure 3-10 show single and back-to-back read commands.

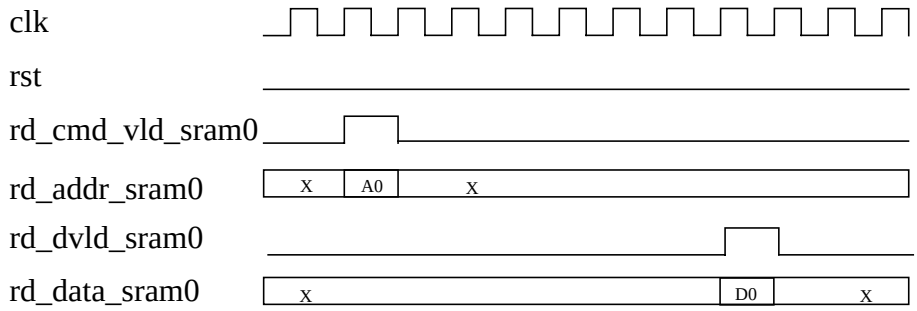

Figure 3-9 Single Read Transaction

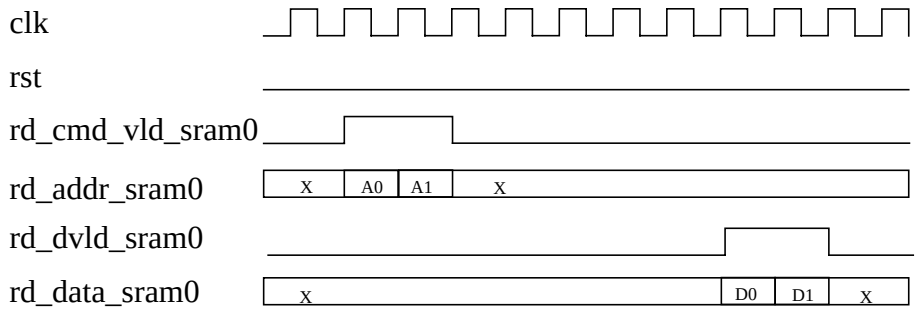

Figure 3-10 Multiple Read Transaction

Designing an Algorithm for Streaming

Purpose

Many applications targeted for hardware acceleration have large input and output data sets that will need to be segmented to fit subsets into the RASC-brick's external SRAM banks at any given time (a total of 32MB are available). Certain applications have large input data sets that have the same processing performed on subsets and only require that new input data be available in order to maintain continuous processing. These requirements bring up the notion of streaming a large data set through an algorithm. Streaming data suggests a continuous and parallel flow of data to and from the algorithm. Streaming is to sequentially load data, execute, and unload a block of data in a loop one at a time.

In order for the hardware accelerated algorithm to run efficiently on large data sets, it is recommended to overlap data loading and unloading with algorithm execution. To do this successfully, start with an SRAM layout of algorithm operands and results that complies with the recommendations for memory distribution. The input and output data needs to be segmented into at least two segments per SRAM bank so that the Algorithm Block can execute on one segment while the next data segment is loaded by the Read DMA Engine and the previous output data segment is unloaded by the Write DMA Engine (ping-pong buffer effect). The SRAM bank can be segmented into any number of segments, as the algorithm and application designers best see fit.

Note: The term *streaming* as used in this section differs from the conventional definition of streaming.

Definitions

- **algorithm iteration**

  One run of the algorithm; the operation that the algorithm performs between the time alg_rst is deasserted until the alg_done flag is asserted. Successive iterations require software to retrigger the algorithm.

- **segment / segment size**

The amount of memory needed on a particular SRAM bank for an algorithm iteration, round up to the nearest power of 2. One segment could include multiple input operands or multiple output operands, with spaces of unused memory within the segment if desired. A segment does not include fixed parameters that are applied to multiple algorithm iterations.

Streaming a segment must be at most 1/2 of the SRAM bank's size, or 8MB. The minimum segment size is 16KB (but not all 16KB need to be used). Different SRAM banks can have different segment sizes according to the sizes and number of operands / results that reside on a particular SRAM bank.

alg_sram0_offset[9:0] and alg_sram1_offset[9:0]

Inputs to the Algorithm Block that specify for each SRAM the starting address of the current data segment. When used, these inputs get mapped to the Algorithm Block's SRAM address bits [19:10] (128-bit word aligned address). The actual number of bits that are used for the mapping is determined by the segment size. If the segment size is 16KB, all 10 bits are mapped to SRAM address bits [19:10]. If the segment size is 512 KB, only bits [9:5] are mapped to SRAM address bits [19:15]. If the segment size is 16MB, only bit [9] is mapped to SRAM address bit [19].

Hardware Support

In order to support streaming, an algorithm should allow the upper bit(s) of its read and write SRAM addresses be programmable variable via the `sram*_offset` inputs. The offset inputs come from an internal FPGA register within the Core Services Block, accessible by the software layer. The offset inputs are 10 bits each, and can map to bits [19:10] of the corresponding SRAM address. Only the bits that correspond to the segment offset are used for a particular algorithm/application. For example, if the segment size is 32 KB, which leads to 512 segments in the 16MB bank, only the upper 9 of the 10 offset bits are used. Example Register Transfer Level (RTL) code for this configuration is shown below:

```
reg [8:0] rd_segment_addr_sram0;
reg [10:0] rd_laddr_sram0;
wire [19:0] rd_addr_sram0;

// Read pointer
always @(posedge clk)
begin
  // Fixed upper address bits [19:11] per iteration
 if (rst) rd_segment_addr_sram0 <= alg_sram0_offset[9:1];

// Counter for lower address bits [10:0]
```

```
 if (rst)
   rd_laddr_sram0 <= 11'h000;
else if (rd_advance_en)
   rd_laddr_sram0 <= rd_addr_sram0 + 1'b1;
end

assign rd_addr_sram0 = {rd_segment_addr_sram0, rd_laddr_sram0};
```

The algorithm must define a legal segment size between the minimum and maximum allowable sizes, and only operate on and generate one segment worth of data per iteration.

Special extractor directives are required to pass information about the algorithm's data layout and streaming capabilities to the software layer. The software layer requires a declaration of the input and output data arrays on each SRAM bank, with attributes size, type, and buffer-ability defined. The declarations are provided as extractor directives, or comments in the Verilog or VHDL source code. The following example comments are used to declare two 16KB input data arrays located on SRAM 0, and one non-streaming input parameter array, also located on SRAM 0:

```
##Array name
##   # of elements in array
##         Bit width of element
##                   SRAM location
##                        Byte offset (within given SRAM)
##                           Direction
##                                Type
##                                     Stream flag
// extractor SRAM:input_a 2048 64 sram[0] 0x000000 in unsigned stream
// extractor SRAM:input_b 2048 64 sram[0] 0x004000 in unsigned stream
// extractor SRAM:param_i 512  64 sram[0] 0xffc000 in unsigned fixed
```

For arrays that are defined as buffered, the byte offset provided in the extractor comments is used to establish the data placement within a particular segment. In the declaration, the byte offset is given at the lowest segment address. For fixed arrays, the byte offset is the absolute placement of the data within the SRAM.

Further details on extractor comments can be found in Chapter 5, "RASC Algorithm FPGA Implementation Guide"

Software Responsibilities

Software uses configuration information derived from extractor directives to move the current active segment on each run of the algorithm. Software changes the values of **sram0_offset[9:0]** and **sram1_offset[9:0]** and for each run of the algorithm and similarly moves the DMA engines' input and output memory regions based on the current active segment.

Passing Parameters to Algorithm Block

This section describes the ways that variable parameters can get passed to the algorithm. For the purposes of this document, a parameter is distinguished from the general input data in that it can be fixed over multiple runs of the algorithm and does not require reloading. It is assumed that input data changes more often than parameters.

Small Parameters

The method used to pass variable parameters depends on the size and number of the required parameters. For a small number of 1-8 byte-sized parameters, the Algorithm Block can associate parameters with up to eight Algorithm Defined Registers. The Algorithm Defined Registers are eight optional registers that exist within the SSP memory mapped register region whose current values are inputs to the Algorithm Block (**alg_def_reg0[63:0]** - a**lg_def_reg7[63:0]**). The Algorithm Block can assign reset (default) values for the parameters by tying the output signals **alg_def_reg0_rst_val[63:0]** - **alg_def_reg7_rst_val[63:0]**, and allow the host application to change them. The algorithm can also use the **alg_def_reg_rst_en** signal to select whether there should be a reset value, or if the parameter should be unaffected by reset.

One example is shown here is, as follows:

```
always @(posedge clk)
  begin
    match_val1 <= alg_def_reg0[15:0];
    match_val2 <= alg_def_reg0[31:16];
  end
```

The algorithm passes information about the mapping of parameters to Algorithm Defined Registers to the software Abstraction Layer by way of the algorithm configuration file. The configuration file and its creation is discussed in detail in Chapter 5, “RASC Algorithm FPGA Implementation Guide”

To summarize here, comments about the parameters are added to the algorithm source code, similar to synthesis directives. Software needs to know what the parameter’s descriptor is, how many bits it comprises, the data type of the parameter, and which Algorithm Defined Register it is associated with. An extractor tool parses these comments and builds up a table of algorithm specific information for use by the Abstraction Layer. Therefore, in order for these parameters to be recognized by the Abstraction Layer and used by the host application, extractor directives need to be added to the source code to indicate parameter mapping.

A simple example is provided below. In this example, the parameter **match_val** is 32-bits wide, unsigned, and mapped to **alg_def_reg0**, bits [63:32].

```
// extractor REG_IN:<parameter_name> <bit width> <type:signed/unsigned>
alg_def_reg[<reg_num>][<bit range>]

// extractor REG_IN:match_val 32 u alg_def_reg[0][63:32]
```

Parameter Arrays

When an algorithm requires larger fixed parameters, portions of the SRAM banks can be used to hold the parameter data. This portion of the SRAM needs to be reserved for parameter data and kept unused by input data, so parameters need to be considered in the initial memory allocation decisions. Just as with small parameters, the mapping of parameter data to SRAM addresses is specified with extractor comments. The template and an example is provided below; further details are in the "Adding Extractor Directives to the Source Code" on page 106. In the provided example, a 1024-element parameter matrix (8KB) is mapped to the upper 8KB of SRAM0, which starts at address 0xFFE000. The type is unsigned and the array is fixed, which denotes that it is a parameter array and not as variable as an input data array (the other option is "stream").

```
// extractor SRAM:<parameter array name> <number of elements in array>
<bit width of elements> <sram bank> <offset byte address into sram> in
<data type of array> fixed

// extractor SRAM:param_matrix0 1024 64 sram[0] 0xFFE000 in u fixed
```

Another use of declaring a fixed array in one of the SRAMs could be for a dedicated scratch pad space. The only drawback to using SRAM memory for scratch pad space is that an algorithm normally writes and then reads back scratch pad data. This usage model violates the streaming rule requiring an algorithm to dedicating each SRAM bank for either inputs or outputs. If you have a free SRAM bank that you do not need for inputs or outputs, this violation can be avoided and the streaming model can be maintained. If you have a free SRAM that is not being used for anything else, then you do not even have to add an extractor directive. An extractor directive is necessary if the SRAM bank is being used for other purposes so that software does not overwrite your scratch pad space. An extractor directive is also necessary to be able to access the scratch pad space from the debugger (reads and writes), so in general, an extractor directive is recommended.

Note that if you violate the streaming model for SRAM direction allocation, data will not be corrupted but the benefit of streaming will not occur because data transfer and

algorithm execution cannot be overlapped. A template and an example is provided below for writing an extractor comment for a scratch pad space.

```
// extractor SRAM:<scratch_pad_array_name>
                  <number of elements in array>
                  <bit width of elements>
                  <sram bank>
                  <offset byte address into sram>
                  inout
                  <data type of array>
                  <signed / unsigned>
                  fixed

// extractor SRAM:scratch1 1024 64 sram[2] 0x000000 inout u fixed
```

Recommended Coding Guidelines for Meeting Internal Timing Requirements

These guidelines are suggestions for achieving 200 MHz timing, when possible (not including floating point operations or use of multipliers).

1. Flop all outputs of the algorithm block, especially critical outputs, such as **step_flag**.
2. Flop and replicate the **rst** input if needed to distribute it as a high fanout signal.
3. Flop the inputs **rd_dvld_sram0** and **rd_data_sram0** before performing combinatorial logic on the data or data valid signals.
4. The general rule to abide by when trying to code a design that passes timing at 200 MHz is this: do not give PAR (the place and route tool) any tough decisions on placement where it would be difficult to find a good location. If a critical signal loads logic in multiple blocks, replicate it so that PAR does not have to try to optimize placement of the driving flop relative to the various loading blocks. You may have to add synthesis directives to prevent the synthesis tool from "optimizing out" your manually replicated flops. As far as possible, do not have a flop drive combinational logic in one block that then loads additional combinational logic in another block (such as Core Services), unless they can be physically grouped to adjacent locations, or in the worst case, minimize the total number of logic levels.

Connecting Internal Signals to the Debugger

This section shows how to make signals internal to the Algorithm Block viewable by the FPGA-enabled GNU Debugger (GDB) software. The Algorithm Block has 64 debugger output ports, each 64-bits wide. In order to make internal signals visible, the algorithm code should connect signals of interest to these outputs ports. To ease the timing issues on paths coming from the Algorithm Block, it is suggested to feed reregistered outputs to the debug outputs. Several examples are shown below:

```
assign debug0 = 64'h0000_000c_0000_0003; //[63:32] alg#, [31:0] rev#
```

In the above example, the outputs are tied, so it is not important to register the outputs.

```
always @(posedge clk)
   debug1 <= {32'h0, running_pop_count};
```

Since the intermediate value `running_pop_count` is also loaded by internal Algorithm Block logic, it is recommended to flop debug register 1 rather than use a wire connection. This helps isolate the loads of `running_pop_count` and reduce the number of constraints on the place and route program.

Besides connections to the debug port, the algorithm has to contain extractor comments that will pass the debug information to the software layer. Debug outputs use the `REG_OUT` type of extractor comment. The extractor comment tells the software layer what the mapping will be for internal signals to the corresponding debug address location. Examples are as follows:

```
// extractor REG_OUT:rev_id 32 u debug_port[0][31:0]
```

and

```
// extractor REG_OUT:running_pop_count 32 u debug_port[1][31:0]
```

The general format is:

```
REG_OUT:<signal name> <signal bit width> <type:unsigned/signed> <debug
port connection>[<bit range>]
```

RASC FPGA Design Integration

This section discusses additional details, including locations of the Algorithm Block in the design hierarchy, and global FPGA logic, such as clocks and resets. It covers the following topics:

Design Hierarchy

Figure 3-11 shows the instance hierarchy of the RASC FPGA design. The top-of-chip is a wrapper module called `acs_top`. The instances in the top level include I/O buffer wrappers, clock resources wrappers, and the two major subdivisions of the logic design: `acs_core`, the pre-synthesized Core Services logic, and the `user_space_wrapper`, the top-level wrapper for the user/algorithm logic. As the algorithm application writer, you should begin the algorithm design using `alg_block_top` as the top level of the algorithm. The other instances within `user_space_wrapper` are small parts of Core Services resources that are left to be resynthesized based on their use and fanout within the algorithm logic. These include reset fanout logic and the debug port multiplexor.

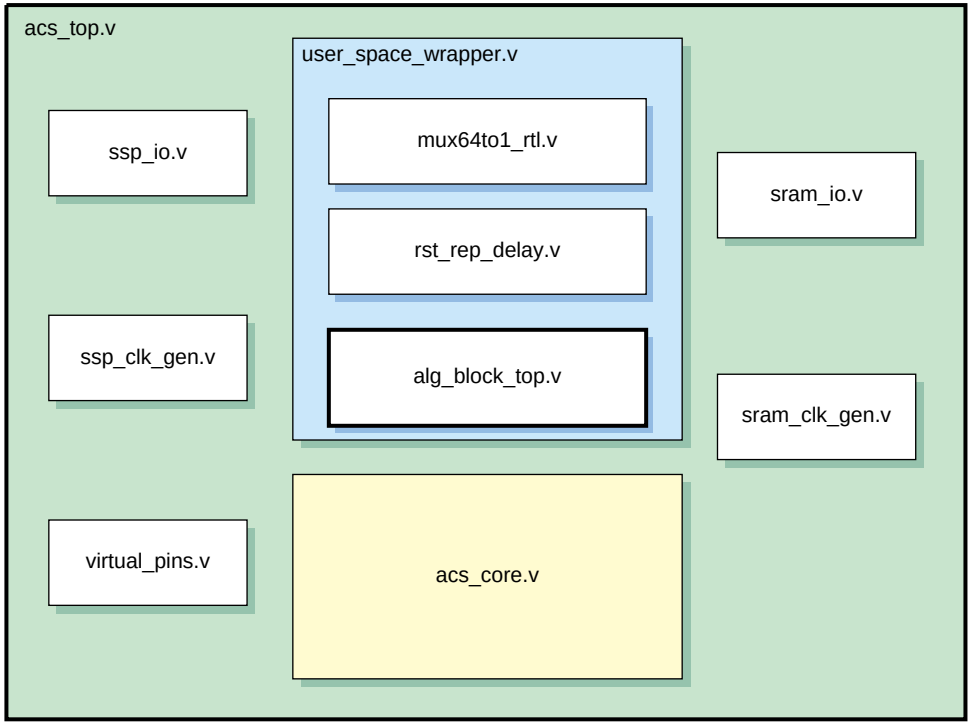

Figure 3-11 Instance Hierarchy of the RASC FPGA Design

The Algorithm / Core Services interface as defined in the section entitled "Algorithm / Core Services Block Interface" on page 28, consist of the input and output signals defined for the module `alg_block_top`.

FPGA Clock Domains

This section describes the clock domains within the RASC Field Programmable Gate Array (FPGA), with a focus on the algorithm clock domain used by the Algorithm Block logic. There are two major domains: core clock and algorithm clock. However, the two domains are not completely asynchronous. They may either both be 200 MHz and phase aligned, or the algorithm clock can have a 50, 66, or 100 MHz and the clocks will be phase / edge-aligned (that is, a rising edge of the algorithm clock will correspond to a rising edge of the core clock).

This section covers the following topics:

Core Clock Domain

The core clock has a fixed clock rate of 200 MHz. It is the main clock used within the Core Services Block. It is derived from the input control clock on the Scalable System Port (SSP) interface. The input clock is used as the clock input to the core clock digital clock manager (DCM) module. The clk0 output of the DCM is driven onto a low-skew global clock buffer (BUFG) and from there is routed to **core_clk** domain registers as well as the feedback of the DCM for fine phase adjustments. In the place and route report, this clock is labelled **core_clk**.

Algorithm Clock Domain

The algorithm clock rate is selectable through the use of macro definitions. Speeds of 50, 66, 100, and 200 MHz can be selected. For speeds slower than 200 MHz, the DCM clock divider in the Xilinx Virtex 4 FPGA is used to create the specified clock signal. Figure 3-12 shows a block diagram of the logic used to create the clock signals.

Macro definitions are used to select between the **clk0** and **clkdv** outputs of the DCM to drive the BUFGCE clock buffer, which in turn drives the **clk** signal in the Algorithm Block via the **alg_clk** signal.

The signal **alg_clk_enable** (not shown) generated by Core Services gates **alg_clk**. By gating the clock, the signal toggles only when the Algorithm Block is active. In the place and route clock report, this clock is labelled **alg_clk**.

When **clkdv** is used to drive the algorithm clock, the phase relationship between the core clock and the algorithm clock is determined by the Virtex 4 parameter `CLKOUT_PHASE`, which specifies the phase relationship between the DCM outputs of the same DCM. For Virtex 4 FPGAs, this parameters is specified as +/- 140 ps. Although the Xilinx timing tools do not take the `CLKOUT_PHASE` into account directly during analysis, an additional 140 ps has been added as input clock jitter to force the tools to correctly check paths crossing the **core_clk** and **alg_clk** domains. Any phase difference that is derived from the

varying routes between the DCM outputs and the BUFG / BUFGCE elements as well as the clock tree fanouts are automatically considered by the Xilinx timing tools.

Core clock domain signals in the Core Services Block that communicate with the Algorithm Block, which is entirely in the **alg_clk** domain, have been designed to transition on the rising edge of **alg_clk**, even when **alg_clk** is run slower than the rate of **core_clk**.

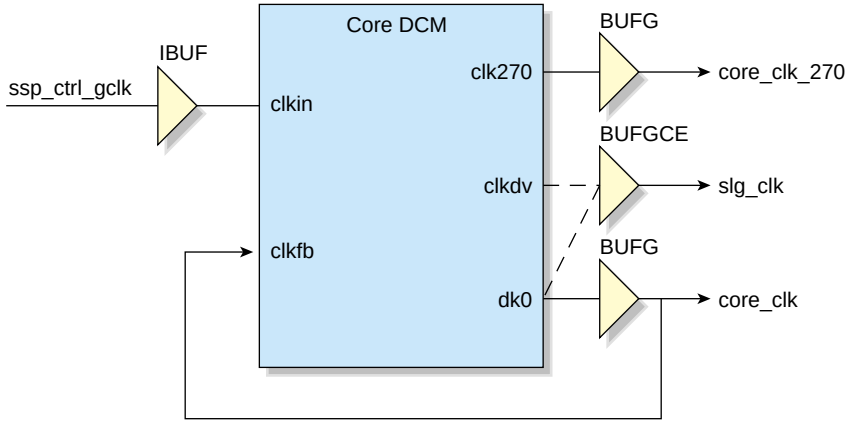

Figure 3-12 Core Clock and Algorithm Clock Source

SSP Clock Domain

Besides the core clock domain, which is equivalent to the Scalable System Port (SSP) control group domain, there are four data group clock domains within the SSP source synchronous receiver and transmitter logic. These four groups have a determined phase relationship between one another: each group is separated by a nominal 250 ps shift to reduce simultaneous switching noise on the SSP interface. In the place and route clock report, these clocks are labelled `int_grp1_clk` - `int_grp4_clk`.

QDR-II SRAM Clock Domains

The QDR-II SRAM module interfaces, a part of Core Services, uses five additional clock domains: one receive clock for each external SRAM (for a total of four, one for each physical SRAM component), and one common data transmit clock, which is a quarter clock cycle early relative to the core_clk. In the place and route clock report, these clocks

are labelled `bank0_sram_rcv_clk`, `bank1_sram_rcv_clk`, `bank2_sram_rcv_clk`, `bank3_sram_rcv_clk`, and `core_clk270`.

Resets

This section describes the hardware resets in the RASC Field Programmable Gate Array. The primary reset input to the FPGA is the `CM_RESET` signal on the SSP port. It is used as both a clock reset and as the basis of a full-chip control logic reset. When used as a clock reset, it is used as an asynchronous reset to the DCMs in the design. A full-chip logic reset is generated based on the assertion of `CM_RESET` or the de-assertion of any of the DCM locked signals. There is a circuit in Core Services that generates a synchronous `core_rst` signal, synchronous to the core clock domain. This reset is used throughout the control logic in Core Services.

The Algorithm Block receives a different reset generated by the Core Services' Algorithm Controller sub-module. The Algorithm Block receives a reset that is synchronous to the algorithm clock. After a full-chip reset, the Algorithm Block's reset input will remain asserted even after `core_rst` is removed. When the algorithm is started by software for the first time, the algorithm reset is removed. This is to allow for debug testing of internal signals after the algorithm completes. When software restarts the algorithm for another run (by setting the `ALG_GO` bit in the `CM_CONTROL` Register), the Algorithm Block will be held in reset for a total of 4 algorithm clock cycles, and then the algorithm reset will be released to allow the Algorithm Block to execute.

To summarize: the Algorithm Block is held in reset until its first use. Each time the algorithm is triggered to execute, the Algorithm Block will be held in reset for 4 clock cycles.

Algorithm Synthesis-time Parameters

This section describes the parameters to be specified by the algorithm designer in order to set the clock rate of the algorithm clock and to enable access to specific SRAM banks.

These synthesis-time parameters are specified in a Verilog include file called `alg.h` and are used by the top-level of design hierarchy. An example of this header file can be found in `$RASC/example/alg_simple_v/alg.h`.

Algorithm Clock Speed

The algorithm clock rate can be set at 50, 66, 100, or 200 MHz.

The following is a portion from the above example include file that selects the clock speed. This shows the four macros used to specify an algorithm clock speed.

```
//////////////////////////////////////////
//    Specify clock speed of algorithm    //
//////////////////////////////////////////
// Only one of the below four should be uncommented

// For 200 MHz
  ‘define alg_clk_5ns

// For 100 MHz
//  ‘define alg_clk_10ns

// For  66 MHz
//  ‘define alg_clk_15ns

// For  50 MHz
//  ‘define alg_clk_20ns
```

One and only one of the above four macros must be defined. Here the macro to set the algorithm clock rate to 200 MHz is defined and uncommented whereas the other macros are left undefined.

SRAM Port Usage

In order for the Algorithm Block to have access to a given SRAM port, the corresponding macro must be defined in `alg.h`.

The following portion from the include file enables access to SRAM ports.

```
//////////////////////////////////////////
//  Specify SRAM ports for algorithm use  //
//////////////////////////////////////////

‘define alg_uses_sram0_rd
//‘define alg_uses_sram0_wr
//‘define alg_uses_sram1_rd
‘define alg_uses_sram1_wr
```

All, some, or none of these macros can be defined. In this example, the read port for SRAM bank 0 and the write port for SRAM bank 1 are enabled, while the write port for SRAM bank 0 and the read port for SRAM bank 1 cannot be used by the Algorithm Block.

If the algorithm is to buffer a large data set, the macros for the read port of one SRAM bank and the write port for the other must be undefined. Otherwise, ports will be inaccessible to the DMA engines while the Algorithm Block is active, and therefore, DMA and the Algorithm Block will not be able to operate concurrently.

Simulating the Design

This section provides a reference on how to simulate the Algorithm using the provided SSP Stub, sample test bench, and VCS simulator. It covers the following topics:

Intent of the Sample Test Bench

The Sample Test Bench (also called sample_tb) is a basic simulation environment for users to do sandbox testing of their algorithm code. The Sample Test Bench is provided as an optional intermediate step between writing an algorithm and loading the algorithm RASC hardware. It is intended to help insure that the algorithm will function on a basic level (e.g. a single algorithm iteration) prior to debugging a bitstream in hardware.

Sample Test Bench Setup

The sample test bench is designed for use with VCS. For use with other simulators, the user should modify the sample test bench along with associated scripts and makefiles.

A primary component of the sample test bench, the SSP Stub, consists predominantly of Verilog modules, although it also includes PLI calls to functions written in C code. The stub is instantiated in a sample Verilog test bench along with the Algorithm FPGA. The files for this test bench are in the directory, `$RASC/dv/sample_tb/`. In this directory you will find the following Verilog modules and other files:

- `top.v`: The top level of the sample test bench containing the Algorithm FPGA design (Core Services and the user's algorithm), SSP Stub, SRAM simulation models, and clock generator.
- `ssp_stub.v`: Top level Verilog of the SSP Stub which passes signals to and from conversion modules. More information on submodules, PLI calls, and C functions that comprise the SSP Stub can be found in the "SSP Stub User's Guide" section of this document.
- `init_sram0.dat`, `init_sram1.dat`, `init_sram2.dat`, `init_sram3.dat`: These SRAM initialization files contain data which is automatically loaded into the respective SRAM simulation models at the beginning of simulation. The data is in a format which the SRAM simulation model uses (one bit of parity per byte of data is shifted in with the data). These default files can be overridden by the user on the command line at runtime.
- `final_sram0.dat`, `final_sram1.dat`, `final_sram2.dat`, `final_sram3.dat`: These files contain data extracted from the respective SRAM simulation models at the end of simulation. These default files can be overridden by the user on the command line at runtime.
- `timescale.v`: This file contains the Verilog timescale of each of the components of the SSP Stub, as well as the algorithm FPGA design files. It is required that the algorithm being simulated makes use of the same timescale as the rest of the design.

In order to use the sample test bench, your VCS environment variables should be set up as follows:

```
### Environment Variables for VCS ###
setenv  VCS_HOME <your_vcs_install_directory>
setenv  VCSPLIDIR $VCS_HOME/<your_vcs_pli_directory>
setenv  PATH        $PATH\:$VCS_HOME/bin
```

Compiling the Sample Test Bench

Compiling the sample test bench is done using the Makefile provided. In order to compile the sample testbench including the SSP Stub and the algorithm Core Services logic, an algorithm must be specified (See the following note).

Note: The `Makefile` in the `sample_tb` directory uses the `$ALG_DIR` environment variable. This defaults to `$RASC/examples` though it can be modified by the user. The design files of the algorithm you specify must be in a directory under the `$ALG_DIR` path.

The algorithm you are building is specified on the command line. To compile the design with your algorithm, change directory to `$RASC/dv/sample_tb` and enter:

```
% make ALG=<your_algorithm>
```

where <*your_algorithm*> is the directory name where the algorithm design files are. When no algorithm is specified, the default is `ALG=alg_simple_v.`

To remove older compiled copies of the testbench, type:

```
% make clean
```

Running a Diagnostic

To run a diagnostic on your algorithm, call the Makefile in the `sample_tb` directory using the "run" target and specifying which diag to run. The following is the usage and options of the "run" target:

```
% make run DIAG=diag_filename ALG=your_agorithm
SRAM0_IN=sram0_input_file  SRAM1_IN=sram1_input_file
SRAM2_IN=sram2_input_file  SRAM3_IN=sram3_input_file
SRAM0_OUT=sram0_output_file  SRAM1_OUT=sram1_output_file
SRAM2_OUT=sram2_output_file  SRAM3_OUT=sram3_output_file
```

The `diag_file` specifies the diagnostic to be run and should be relative to the current directory. Again, the algorithm must be specified using the ALG=*your_algorithm* command line option. If none is specified, the runtime command uses same default as above (`ALG=alg_simple_v`). Specifying `ALG` this way allows the user to reuse the same diagnostic for multiple algorithms. The contents of each SRAM at the end of simulation will be dumped into `.dat` files that can be user-specified. If they are not specified, they default to:

```
init_sram0_good_parity.dat
init_sram1_good_parity.dat
init_sram2_good_parity.dat
init_sram3_good_parity.dat
final_sram0.dat
```

```
final_sram1.dat
final_sram2.dat
final_sram3.dat
```

Note that there are four input and four output SRAM data files while the design is implemented for two logical SRAMs. Each of the logical SRAMs is implemented as two separate physical SRAMs in the sample testbench. The sram0* and sram1* files correspond to the first logical SRAM while sram2* and sram3* correspond to the second logical SRAM.

By specifying the SRAM input and output files the user can skip the DMA process for quick verification of the algorithm. This shortens the diagnostic run time, makes for less complex diagnostics, and allows the user to ignore core services as it has already been verified by SGI. The option of utilizing the DMA engines in simulation is included for completeness but should not be necessary for typical algorithm verification.

As the diagnostic runs, it will output status to the screen and to an output file named <*diag_filename*>.<*your_algorithm*>`.run.log`. When the stub receives an unexpected packet, it will output the following information in order: the command for the next expected packet, SSP fields of the expected packet, the command translation (if one exists) for the received packet, and the SSP fields of the received packet. This log file will appear in the same directory in which that the diagnostic is located.

Table 3-7 shows a summary of the algorithms, diagnostics, and commands provided with the sample testbench.

Table 3-7 Sample Testbench Algorithms and Commands

Algorithm Name	Diagnostic	Compile and Run Commands
`alg_simple_v`	`diags/alg_simple_v`	`make ALG=alg_simple_v` `make run DIAG=diags/alg_simple_v ALG=alg_simple_v`
`alg_data_flow_v`	`diags/alg_data_flow_v`	`make ALG=alg_data_flow_v` `make run DIAG=diags/alg_data_flow_v ALG=alg_data_flow_v`

Viewing Waveform Results

Each time a diagnostic is run, a file named `vcdplus.vpd` is generated in the `sample_tb` directory. This file can be input to Virsim for viewing the waveform. Since this file is generally large, it is overwritten for each diagnostic run. To save the waveform for a given diagnostic, copy the corresponding `vcdplus.vpd` file to a new name.

To view the waveform saved in the `vcdplus.vpd` file, use the following command:

```
% vcs -RPP vcdplus.vpd
```

A sample configuration file `sample_tb/basic.cfg` is provided in the for use when viewing waveforms in Virsim. It contains a limited number of relevant signals on the SSP interface, SRAM interfaces, and inside the design. Figure 3-13 shows a sample `vcdplus.vpd` waveform in Virsim.

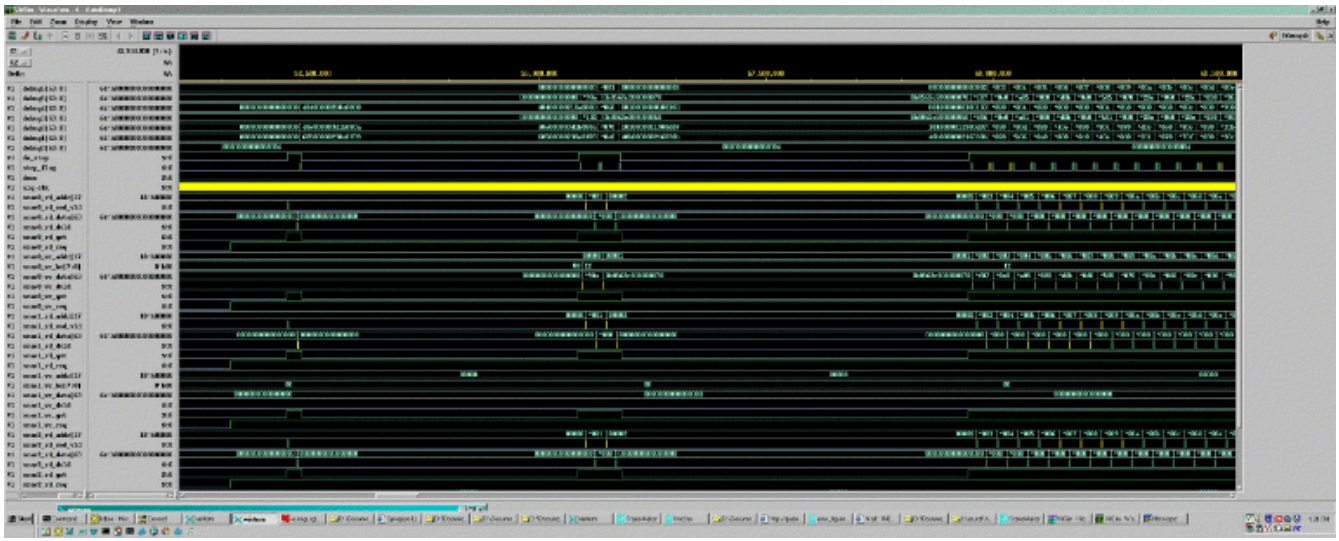

Figure 3-13 Sample `vcdplus.vpd` Waveform in Virsim

Writing a Diagnostic

The SSP Stub retrieves instructions through a text input file, the diagnostic. The SSP Stub parses this file at each semicolon to extract commands that the Stub executes. Many of the allowed commands in a diagnostic correspond to SSP packet types. There are other commands that the SSP Stub supports for diagnostic writing and debugging. The primary components of the diagnostic file are: packet commands, debugging commands, and comments.

It is important to note that most SSP packets come in pairs: a request and a response. For these types of packets, the request command and response command must be listed sequentially in a diagnostic. This method of keeping requests and response paired is used by the stub to associate request and response packets with the corresponding transaction number (TNUM). For more information on SSP packet types, see the *Scalable System Port Specification*. Also, when running the DMA engines, all transactions related to that sequence of events should be grouped together. See Appendix B, “SSP Stub User’s Guide” for more details on diagnostic writing and using the SSP stub.

The code listed below comprises a diagnostic that exercises the basic functionality of the algorithm FPGA outlined in the following steps:

- Initializes the algorithm FPGA Core Services (primarily MMR Writes)
- Executes DMA Reads to send data to the FPGA (stored in SRAM)
- Starts the Algorithm (d = a & b | c) and polls the memory mapped registers (MMRs) to see when the Algorithm is done
- Executes DMA Writes to retrieve the Algorithm's results
- Checks the error status in the MMRs to verify that no errors were flagged.

The example diagnostic provided below is intended as a template that may be edited to match the user's algorithm.

```
#######  Initialization packets.  #######
# Arm regs by setting the REARM_STAT_REGS bit in the CM_CONTROL reg
snd_wr_req ( PIO, DW, ANY, 0x00000000000020, 0x0000000600f00003 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );

# Clear the CM_ERROR_STATUS register by writing all zeroes.
snd_wr_req ( PIO, DW, 3, 0x00000000000060, 0x0000000000000000 );
rcv_wr_rsp ( PIO, DW, 3, 0);

# Enable CM_ERROR_DETAIL_* regs by writing all zeroes to CM_ERROR_DETAIL_1.
snd_wr_req ( PIO, DW, ANY, 0x00000000000010, 0x0000000000000000 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );

# Enable desired interrupt notification in the CM_ERROR_INTERRUPT_ENABLE
register.
snd_wr_req ( PIO, DW, 4, 0x00000000000070, 0xFFFFFFFFFFFFFFFF );
rcv_wr_rsp ( PIO, DW, 4, 0 );

# Set up the Interrupt Destination Register.
snd_wr_req ( PIO, DW, ANY, 0x00000000000038, 0x0000000000000000 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );

print "\n\n*******Initialization finished\n\n";


#######  Configure DMA Engines and Algorithm.  #######

#######  Configure the Read DMA Engine Registers.  #######
print "\n\n*******Configure Read DMA Engine.  Tell it to fill 32 cache lines
of data.\n\n";

# RD_DMA_CTRL register.
snd_wr_req ( PIO, DW, ANY, 0x00000000000110, 0x0000000000100020 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );

# RD DMA addresses.
snd_wr_req ( PIO, DW, ANY, 0x00000000000100, 0x0000000000100000 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );
snd_wr_req ( PIO, DW, ANY, 0x00000000000108, 0x0000000000000000 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );

# RD AMO address.
```

```
snd_wr_req ( PIO, DW, ANY, 0x00000000000118, 0x0000000000000000 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );

# RD_DMA_DEST_INT
snd_wr_req ( PIO, DW, ANY, 0x00000000000120, 0x0000000200002000 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );


#######  Configure the Write DMA Engine Registers.  #######
print "\n\n*******Configure Write DMA Engine.\n\n";

# Write to the WR_DMA_CTRL register.
snd_wr_req ( PIO, DW, ANY, 0x00000000000210, 0x0000000000100020 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );

# WR_DMA_SYS_ADDR
snd_wr_req ( PIO, DW, ANY, 0x00000000000200, 0x0000000000100000 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );

# WR_DMA_LOC_ADDR
snd_wr_req ( PIO, DW, ANY, 0x00000000000208, 0x0000000000000000 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );

# WR_DMA_AMO_DEST
snd_wr_req ( PIO, DW, ANY, 0x00000000000218, 0x0000000000000000 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );

# WR_DMA_INT_DEST
snd_wr_req ( PIO, DW, ANY, 0x00000000000220, 0x0000000400004000 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );

#######  Configure the Algorithm Registers.  #######
print "\n\n*******Configure Algorithm Registers\n\n";

snd_wr_req ( PIO, DW, ANY, 0x00000000000300, 0x0000000000000000 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );

snd_wr_req ( PIO, DW, ANY, 0x00000000000308, 0x0000000600006000 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );


#######  Start Read DMA Engine for Read DMA 1  #######
print "\n\n*******Start Read DMA Engine for SRAM0\n\n";

# Set Bit 36 of the CM_CONTROL Reg to 1.
snd_wr_req ( PIO, DW, ANY, 0x00000000000020, 0x0000001400f00003 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );

# 1 of 32
rcv_rd_req ( MEM, FCL, ANY, 0x00000000100000 );
snd_rd_rsp ( MEM, FCL, ANY, 0, 0xDEADBEEFDEADBEEF, 0xDEADBEEFDEADBEEF,
                               0xDEADBEEFDEADBEEF, 0xDEADBEEFDEADBEEF,
                               0xDEADBEEFDEADBEEF, 0xDEADBEEFDEADBEEF,
                               0xDEADBEEFDEADBEEF, 0xDEADBEEFDEADBEEF,
                               0xDEADBEEFDEADBEEF, 0xDEADBEEFDEADBEEF,
                               0xDEADBEEFDEADBEEF, 0xDEADBEEFDEADBEEF,
                               0xDEADBEEFDEADBEEF, 0xDEADBEEFDEADBEEF );

# Other Read DMA Transactions omitted here

# 32 of 32
```

```
rcv_rd_req ( MEM, FCL, ANY, 0x00000000100F80 );
snd_rd_rsp ( MEM, FCL, ANY, 0, 0xDEADBEEFDEADBEEF, 0xDEADBEEFDEADBEEF,
                               0xDEADBEEFDEADBEEF, 0xDEADBEEFDEADBEEF,
                               0xDEADBEEFDEADBEEF, 0xDEADBEEFDEADBEEF,
                               0xDEADBEEFDEADBEEF, 0xDEADBEEFDEADBEEF,
                               0xDEADBEEFDEADBEEF, 0xDEADBEEFDEADBEEF,
                               0xDEADBEEFDEADBEEF, 0xDEADBEEFDEADBEEF,
                               0xDEADBEEFDEADBEEF, 0xDEADBEEFDEADBEEF );

print "\n\n*******Polling for DMA RD-SRAM0 done (bit 42 of CM_STATUS).\n\n";
poll (0x8, 42, 20);
print "\n\n*******Done storing data in SRAM 0.\n\n";


#######  Reconfigure DMA Engine for Read DMA 2  #######

# RD_DMA_SYS_ADDR
snd_wr_req ( PIO, DW, ANY, 0x00000000000100, 0x0000000000100000 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );

# RD_DMA_LOC_ADDR
snd_wr_req ( PIO, DW, ANY, 0x00000000000108, 0x0000000000200000 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );


#######  Start Read DMA Engine for Read DMA 2  #######
print "\n\n*******Start Read DMA Engine for SRAM1\n\n";

# Set Bit 36 of the CM_CONTROL Reg to 1.
snd_wr_req ( PIO, DW, ANY, 0x00000000000020, 0x0000001400f00003 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );

# 1 of 32
rcv_rd_req ( MEM, FCL, ANY, 0x00000000100000 );
snd_rd_rsp ( MEM, FCL, ANY, 0, 0xF0F0F0F0F0F0F0F0, 0xF0F0F0F0F0F0F0F0,
                               0xF0F0F0F0F0F0F0F0, 0xF0F0F0F0F0F0F0F0,
                               0xF0F0F0F0F0F0F0F0, 0xF0F0F0F0F0F0F0F0,
                               0xF0F0F0F0F0F0F0F0, 0xF0F0F0F0F0F0F0F0,
                               0xF0F0F0F0F0F0F0F0, 0xF0F0F0F0F0F0F0F0,
                               0xF0F0F0F0F0F0F0F0, 0xF0F0F0F0F0F0F0F0,
                               0xF0F0F0F0F0F0F0F0, 0xF0F0F0F0F0F0F0F0 );

# Other Read DMA Transactions omitted here

# 32 of 32
rcv_rd_req ( MEM, FCL, ANY, 0x00000000100F80 );
snd_rd_rsp ( MEM, FCL, ANY, 0, 0xF0F0F0F0F0F0F0F0, 0xF0F0F0F0F0F0F0F0,
                               0xF0F0F0F0F0F0F0F0, 0xF0F0F0F0F0F0F0F0,
                               0xF0F0F0F0F0F0F0F0, 0xF0F0F0F0F0F0F0F0,
                               0xF0F0F0F0F0F0F0F0, 0xF0F0F0F0F0F0F0F0,
                               0xF0F0F0F0F0F0F0F0, 0xF0F0F0F0F0F0F0F0,
                               0xF0F0F0F0F0F0F0F0, 0xF0F0F0F0F0F0F0F0,
                               0xF0F0F0F0F0F0F0F0, 0xF0F0F0F0F0F0F0F0 );

print "\n\n*******Polling for DMA RD-SRAM1 done (bit 42 of CM_STATUS).\n\n";
poll (0x8, 42, 200);
print "\n\n*******Done storing data in SRAM 1.\n\n";
```

```
#######  Reconfigure DMA Engine for Read DMA 3  #######

# RD DMA addresses.
snd_wr_req ( PIO, DW, ANY, 0x00000000000100, 0x0000000000100000 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );
snd_wr_req ( PIO, DW, ANY, 0x00000000000108, 0x0000000000400000 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );


#######  Start Read DMA Engine for Read DMA 3  #######
print "\n\n*******Start Read DMA Engine for SRAM2\n\n";

# Set Bit 36 of the CM_CONTROL Reg to 1.
snd_wr_req ( PIO, DW, ANY, 0x00000000000020, 0x0000001400f00003 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );

# 1 of 32
rcv_rd_req ( MEM, FCL, ANY, 0x00000000100000 );
snd_rd_rsp ( MEM, FCL, ANY, 0, 0x0C0C0C0C0C0C0C0C, 0x0C0C0C0C0C0C0C0C,
                               0x0C0C0C0C0C0C0C0C, 0x0C0C0C0C0C0C0C0C,
                               0x0C0C0C0C0C0C0C0C, 0x0C0C0C0C0C0C0C0C,
                               0x0C0C0C0C0C0C0C0C, 0x0C0C0C0C0C0C0C0C,
                               0x0C0C0C0C0C0C0C0C, 0x0C0C0C0C0C0C0C0C,
                               0x0C0C0C0C0C0C0C0C, 0x0C0C0C0C0C0C0C0C,
                               0x0C0C0C0C0C0C0C0C, 0x0C0C0C0C0C0C0C0C );

# Other Read DMA Transactions omitted here

# 32 of 32
rcv_rd_req ( MEM, FCL, ANY, 0x00000000100F80 );
snd_rd_rsp ( MEM, FCL, ANY, 0, 0x0C0C0C0C0C0C0C0C, 0x0C0C0C0C0C0C0C0C,
                               0x0C0C0C0C0C0C0C0C, 0x0C0C0C0C0C0C0C0C,
                               0x0C0C0C0C0C0C0C0C, 0x0C0C0C0C0C0C0C0C,
                               0x0C0C0C0C0C0C0C0C, 0x0C0C0C0C0C0C0C0C,
                               0x0C0C0C0C0C0C0C0C, 0x0C0C0C0C0C0C0C0C,
                               0x0C0C0C0C0C0C0C0C, 0x0C0C0C0C0C0C0C0C,
                               0x0C0C0C0C0C0C0C0C, 0x0C0C0C0C0C0C0C0C );

print "\n\n*******Polling for DMA RD-SRAM2 done (bit 42 of CM_STATUS).\n\n";
poll (0x8, 42, 200);
print "\n\n*******Done storing data in SRAM 2.\n\n";

#######  Start the Algorithm  #######

# Set bit 38 of CM Control Register to 1 to start algorithm.
snd_wr_req ( PIO, DW, ANY, 0x00000000000020, 0x0000004400f00003 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );

print "\n\n*******Started Algorithm.\n\n";

# Poll for ALG_DONE bit in CM_STATUS.
poll ( 0x8, 48, 2000);
print "\n\n*******Algorithm Finished.\n\n";

#######  Start Write DMA Engine.  #######

# Set bit 37 of CM Control Register to 1 to start Write DMA Engine.
snd_wr_req ( PIO, DW, ANY, 0x00000000000020, 0x0000002400f00003 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );
```

```
print "\n\n*******Started Write DMA Engine.\n\n";

# 1 of 32
rcv_wr_req ( MEM, FCL, ANY, 0x00000000100000, 0xDCACBCECDCACBCEC,0xDCACBCECDCACBCEC,
                                               0xDCACBCECDCACBCEC,0xDCACBCECDCACBCEC,
                                               0xDCACBCECDCACBCEC,0xDCACBCECDCACBCEC,
                                               0xDCACBCECDCACBCEC,0xDCACBCECDCACBCEC,
                                               0xDCACBCECDCACBCEC,0xDCACBCECDCACBCEC,
                                               0xDCACBCECDCACBCEC,0xDCACBCECDCACBCEC,
                                               0xDCACBCECDCACBCEC,0xDCACBCECDCACBCEC,
                                               0xDCACBCECDCACBCEC,0xDCACBCECDCACBCEC );
snd_wr_rsp ( MEM, FCL, ANY, 0 );

# Other Write DMA Transactions omitted here

# 32 of 32
rcv_wr_req ( MEM, FCL, ANY, 0x00000000100F80, 0xDCACBCECDCACBCEC,0xDCACBCECDCACBCEC,
                                               0xDCACBCECDCACBCEC,0xDCACBCECDCACBCEC,
                                               0xDCACBCECDCACBCEC,0xDCACBCECDCACBCEC,
                                               0xDCACBCECDCACBCEC,0xDCACBCECDCACBCEC,
                                               0xDCACBCECDCACBCEC,0xDCACBCECDCACBCEC,
                                               0xDCACBCECDCACBCEC,0xDCACBCECDCACBCEC,
                                               0xDCACBCECDCACBCEC,0xDCACBCECDCACBCEC,
                                               0xDCACBCECDCACBCEC,0xDCACBCECDCACBCEC );
snd_wr_rsp ( MEM, FCL, ANY, 0 );

print "\n\n*******Polling for DMA WR-SRAM0 done (bit 45 of CM_STATUS).\n\n";
poll (0x8, 45, 200);
print "\n\n*******Done retrieving data from SRAM 0.\n\n";

#######  Finish Up  ######

# dma_clear().  Set bits 39, 40, and 41 to 1 in CM_CONTROL.
snd_wr_req ( PIO, DW, ANY, 0x00000000000020, 0x0000038400f00003 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );

# finalcheck_ccc() Check CACHE_RD_DMA_FSM.
snd_rd_req ( PIO, DW, ANY, 0x00000000000130 );
rcv_rd_rsp ( PIO, DW, ANY, 0, 0x0000000000400000 );

print "Reading the Error Status Register to insure no errors were
logged.\n";
snd_rd_req ( PIO, DW, ANY, 0x00000000000060 );
rcv_rd_rsp ( PIO, DW, ANY, 0, 0x0000000000000000 );
```

Sample Test Bench Constants and Dependencies

Various constants and definitions for the sample test bench are contained within the following files:

- `ssp_defines.h` (internal stub variables)
- `user_const.h` (user modifiable)

Table 3-8 lists the files in the `sample_tb` directory with their function and calls dependencies.

Table 3-8 Files in the `sample_tb` directory

File	Functions	Dependent On
start_ssp.c	start_ssp()	queue_pkt.h, setup_pkt.h, send_rcv_flits.h
send_rcv_flits.h	send_rcv_flits(), send_flit(), rcv_flit(),snd_poll(), rcv_poll(), finish_ssp()	setup_pkt.h, process_pkt.h, get_fields.h, snd_rcv_fns.h
queue_pkt.h	queue_pkt(string), q_string_it(token, pkt_string),strtok_checked(s1, s2)	--
setup_pkt.h	setup_pkt(snd_rcv)	snd_rcv_fns.h
snd_rcv_fns.h	snd_wr_req(pio_mem_n, size, tnum, addr, data, pkt), snd_rd_req(pio_mem_n, size, tnum, addr, pkt), snd_wr_rsp(pio_mem_n, size, tnum, error, pkt), snd_rd_rsp(pio_mem_n, size, tnum, error, data), snd_amo_rsp(tnum, error, pkt), inv_flush(tnum, pkt), rcv_wr_rsp(pio_mem_n, size, tnum, error, pkt), rcv_rd_rsp(pio_mem_n, size, tnum, error, data, pkt), rcv_wr_req(pio_mem_n, size, tnum, addr, data, pkt), rcv_rd_req(pio_mem_n, size, tnum, addr, pkt),rcv_amo_req(tnum, addr, data, pkt)	construct_pkt.h
construct_pkt.h	construct_pkt(type, tnum, address, data, error, pkt, to_from_n), pkt_size(type)	make_command.h
make_command.h	make_command(type, tnum, error, to_from_n)	--
get_fields.h	get_fields(type), f_string_it(token)	--
process_pkt.h	process_pkt(type)	--

Sample Test Bench Utilities

The sample test bench includes utilities that help in generating and interpreting diagnostic data. To compile these files into executables, run the following command:

```
% gcc -c file_name -o executable name
```

The utilities provided include the following:

- `convert_sram_to_dw.c`

  This program takes a standard SRAM input/output file (e.g. `final_sram0.dat`), and converts it to a more readable version consisting of one SGI double word (64-bits) of data per line. It assumes that the input file is made up of 36-bit words containing parity bits. This utility is helpful when trying to interpret results from the stub output files.

  Use: `convert_sram_to_dw` *input_file* [*output_file*]

  Default output file: `convert_sram_to_dw_output.dat`

- `convert_dw_to_sram_good_parity.c`

  This program takes a file containing one SGI double word (64-bits) of data per line, calculates parity and outputs a file that can be loaded into SRAM for simulation (36-bits of data with parity per line). It assumes the input file contains the correct number of lines to fill the SRAM. This utility is useful when you want to input specific data to an SRAM and skip the DMA process in simulation.

  Use: `convert_dw_to_sram_good_parity` *input_file* [*output_file*]

  Default output file: `convert_dw_to_sram_good_parity_output.dat`.

- `command_fields.c`

  This program takes an SSP command word, splits it into its SSP fields and outputs the SSP field information to the screen. The utility provides this data in the same format as the `get_fields.h` function in the SSP stub. This feature is potentially useful in debugging from the Virsim viewer.

  Use: `command_fields` *32-bit_hex_value*

- `check_alg_data_flow.c`

  This program uses the SRAM output file to check data against input data. It takes the data in `init_sram0_good_parity.dat`, removes parity, and performs a byte-sort on this data (byte-sorts each 8 byte quantity as done by `alg_data_flow_v`). The program compares the result to the data in the file `final_sram1.dat` (with parity removed). As the comparisons are done, the program prints the byte-sorted input data on the left and the results from the final data on the right. If there are differences in the data, it prints "ERROR" on the corresponding line. The program exits after it finding a finite number of errors (adjustable in the source code).

- `check_alg_simple.c`

This file is an example of a program that uses the SRAM output data file to check data against input data. It takes the SRAM0 and SRAM1 input data in `init_sram0_good_parity.dat` and `init_sram1_good_parity.dat` and calculates the results of a A & B | C. It then compares its expected data to the data in the files `final_sram2.dat` and `final_sram3.dat` (the default output for SRAM2 and SRAM3, respectively). As it proceeds, it prints out the results of each double word result D: on the left, what it expects based on the input data, on the right, what it is seeing in the final results. If there are discrepancies in the data, it prints "ERROR" on the line in question.

The code is set to exit after it finds 64 errors, but this number can be raised or lowered easily (line 60).

Simulation of `rasclib` Functions

A recently implemented feature of the sample test bench is the support of a limited set of `rasclib` calls. The intent is to allow you to compile your application code with simple simulation definitions of `rasclib` commands such as `rasclib_algorithm_send`(), `rasclib_algorithm_go`(), `rasclib_algorithm_receive`(), and so on. Using simulation, you can specify input data, start your algorithm, and verify output for a single iteration of the algorithm.

Using `rasclib` in simulation is suggested for testing and debugging an algorithm early in development. Simulation using `rasclib` has limitations that may require you to limit the complexity of your implementation while simulating. A couple of guidelines are, as follows:

- Limit simulation to a single iteration of the algorithm
- Limit the amount of data consumed or produced by the algorithm to around 4KB (for simulation only)

  Allowing the algorithm to work on several megabytes of data is useful for implementation in real hardware but is typically unreasonable for simulation. Limiting data will help insure that simulation completes in a reasonable amount of wall clock time.

Simulating with `rasclib` requires the following:

- The RTL of the algorithm block (`alg_block_top.v`) and any submodules. For simulation, this RTL should be located in the `$RASC/examples/<`*alg_name*`>/` directory which should be created by the user.

- User application code which calls the user's algorithm. For simulation, this is application code should be located in the `$RASC/dv/sample_tb/tests/` directory which is part of the sample test bench. It is recommended, though not required that the application code be named <*alg_name*>`.cpp` as has been done for the examples provided.
- The configuration file contains a description of where inputs and outputs should be located in local memory. For simulations, this configuration file should be located in the existing `$RASC/dv/sample_tb/tests/` directory. For simulation, this configuration file should be named <*alg_name*>`.cfg`. Unlike the other files above, this file is not written by the user. Rather, it is generated as part of algorithm implementation. The configuration file is output from the Extractor script which is described in Appendix C, "How Extractor Works".

Working examples of each of these files can be found in their respective locations. To simulate using `rasclib`, the application code must be compiled with the simulation version of the `rasclib` calls. A Makefile has been provided to help compile the application code correctly. To compile, ensure that the RTL, application code, and configuration file are in their respective locations. Then go to the `$RASC/dv/sample_tb/tests/` directory and enter `make` <*alg_name*>. For example:

```
make alg_simple_v
```

Or:

```
make alg_data_flow_v
```

This compiles the application code found in this directory. The application code will compile with the simulation version of the `rasclib` calls that are responsible for running the simulator. The executable that is generated is named the same as the application code without the dot suffix file extension. Once this compiles cleanly, run the executable. For example:

```
./alg_simple_v
```

Or:

```
./alg_data_flow_v
```

This executes the application. The appropriate `rasclib` calls send data from the application to the simulator, compile and run the simulator, and provide the results from the simulator back to the application. For more details on simulating `rasclib`, SGI recommends that you try the sample algorithms and look through the sample

application code provided. To compile application code from a clean state, start by executing this command:

```
make clean
```

The final application code that runs the algorithm in hardware (non-simulation) may likely have significant functionality that differs from the application code used for debugging with the simulation version of `rasclib`. Below are some differences to keep in mind for porting applications written for `rasclib` on the 64-bit Altix platform to the simulation `rasclib` environment (typically a 32-bit programming environment). Note that these differences have already been incorporated into the samples provided.

- Change variables of type unsigned long to type unsigned long long for simulation.
- Limit the array buffer size to one algorithm run for simulation. It should match the array size in the configuration file (e.g. `alg_simple_v.cfg`).

RASC Abstraction Layer

This section describes the Reconfigurable Application-Specific Computing (RASC) Abstraction Layer and covers the following topics:

RASC Abstraction Layer Overview

The RASC Abstraction Layer provides an application programming interface (API) for the kernel device driver and the RASC hardware. It is intended to provide a similar level of support for application development as the standard open/close/read/write/ioctl calls for IO peripherals.

The Abstraction Layer is actually implemented as two layers. The lowest level is the COP (Co-Processor) level. This provides calls to function individual devices. The upper level, which is built on top of the COP level, is the algorithm level. Here, the application treats a collection of devices as a single, logical algorithm implementation. The Abstraction Layer manages data movement to and from the devices, spreading work across multiple devices to implement scaling.

As an application develops, you must decide which level you are programming to. They are mutually exclusive. You can use one or the other, but **never** both.

Figure 4-1 illustrates the interactions between the pieces and layers of the system.

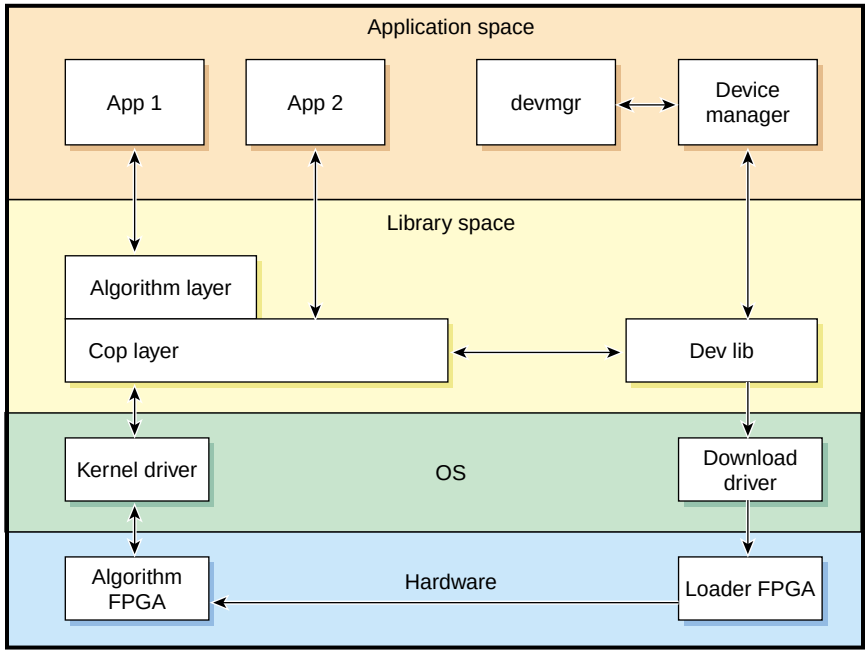

Figure 4-1 Abstraction Layer Software Block Diagram

RASC Abstraction Layer Calls

Table 4-1 contains a summary of the abstraction layer function definitions.

All function prototypes, data structures and constants are defined in `rasclib.h`.

Table 4-1 Abstraction Layer Function Definitions - Summary

Function	Description
`rasclib_resource_alloc`	Sends a request to the device manager to allocate/reserve devices for the application.
`rasclib_resource_free`	Returns devices to the device manager's free pool.
`rasclib_algorithm_open`	Notifies the `rasclib` library that the user wants to use all of the reserved devices loaded with the bitstream associated with `al_id` as a single logical device.
`rasclib_algorithm_send`	Queues up a command that will move data from host memory to the device input buffer identified by `algo_array_id.`
`rasclib_algorithm_get_num_cops`	Returns the number of physical devices that are participating in the algorithm.
`rasclib_algorithm_receive`	Queues up a command that will move data from the device output buffer to host memory.
`rasclib_algorithm_go`	Queues up a command to tell the bitstream compute engine to begin computation.
`rasclib_algorithm_commit`	Causes the `rasclib` library to send all queued commands to the kernel driver for execution on the device.
`rasclib_algorithm_wait`	Blocks until all commands that have been sent to the driver have complete execution.
`rasclib_algorithm_close`	Releases all host resources associated with the algorithm.
`rasclib_algorithm_reg_multi_cast`	Write the same data word to all FPGAs that consititute the algorithm.
`rasclib_algorithm_reg_read`	Read data from the registers of the FPGAs.

Table 4-1 Abstraction Layer Function Definitions - Summary **(continued)**

Function	Description
`rasclib_algorithm_reg_write`	Write data to the registers of the FPGAs.
`rasclib_cop_open`	Notifies `rasclib` that the user wants to use one of the reserved devices loaded with the bitstream associated with `al_id`.
`rasclib_cop_send`	`Queue`s up a command that will move data from host memory to the device input buffer identified by `algo_array_id`.
`rasclib_cop_receive`	Queues up a command that will move data from the device output buffer to host memory.
`rasclib_cop_go`	Queues up a command to tell the bitstream compute engine to begin computation.
`rasclib_cop_commit`	Causes `rasclib` to send all queued commands to the kernel driver for execution on the device
`rasclib_cop_wait`	Blocks until all commands that have been sent to the kernel driver have complete execution.
`rasclib_cop_close`	Releases all host resources associated with the algorithm.
`rasclib_cop_read`	Read data from the FPGA register.
`rasclib_cop_write`	Write data to the FPGA register.
`rasclib_perror`	Print error message.
`rasclib_error_dump`	Print all error messages.

`rasclib_resource_alloc` Function

The `rasclib_resource_alloc` function is, as follows:

SYNOPSIS

```
int rasclib_resource_alloc(rasclib_algorithm_request_t *al_request, int
num_cops);
```

DESCRIPTION

`rasclib_resource_alloc`() sends a request to the device manager to allocate/reserve devices for the application. The application must make only a single call to `rasclib_resource_alloc`() (though this is not enforced) to avoid deadlock conditions.

`al_request` is a pointer to an array of `rasclib_algorithm_request_t` requests. A `rasclib_algorithm_request_t` is defined as:

```
typedef struct rasclib_algorithm_request_s {
rasclib_algorithm_id_t  algorithm_id;
                int num_devices;
        }rasclib_algorithm_request_t;
```

The `algorithm_id` identifies which bitstream should be loaded. It is given to the device manager at bitstream registration and can be obtained from the device manager user interface. The `num_devices` argument tells the device manager how many devices should be loaded with the bitstream.

`num_cops` is the number of elements in the array of requests.

DIAGNOSTICS

Returns `RASCLIB_SUCCESS` when all goes well.

Returns `RASCLIB_NODEVICE_REQUESTED` if no devices are requested.

Returns `RASCLIB_RESERVE_FAILED` if the device reservaation fails.

Returns `RASCLIB_ALLOCATION_FAILED` if the device allocation fails.

`rasclib_resource_free` Function

The `rasclib_resource_free` function is, as follows:

SYNOPSIS

```
int rasclib_resource_free(rasclib_algorithm_request_t *al_request, int
num_cops);
```

DESCRIPTION

`rasclib_resource_free`() returns devices to the device manager's free pool. The arguments and their meaning are the same as the `rasclib_resource_alloc`() function.

DIAGNOSTICS

Returns `RASCLIB_SUCCESS` when all goes well.

`rasclib_algorithm_open` Function

The `rasclib_algorithm_open` function is, as follows:

SYNOPSIS

```
int rasclib_algorithm_open(rasclib_algorithm_id_t al_id);
```

DESCRIPTION

`rasclib_algorithm_open`() notifies `rasclib` that the user wants to use all of the reserved devices loaded with the bitstream associated with `al_id` as a single logical device. The `rasclib` library initializes all necessary data structures and returns a small integer referred to as an algorithm descriptor. This is similar in concept to a UNIX file descriptor. It is used internally in the library to identify a particular algorithm.

`al_id` is the same as the algorithm id in the allocate and free routines.

DIAGNOSTICS

Returns `RASCLIB_FAIL` on failure, an algorithm descriptor (which is greater than or equal to zero) on success.

`rasclib_algorithm_send` Function

The `rasclib_algorithm_send` function is, as follows:

SYNOPSIS

```
int rasclib_algorithm_send(int algo_desc, rasclib_algo_array_id_t
algo_array_id, void *buf, int count);
```

DESCRIPTION

`rasclib_algorithm_send`() queues up a command that will move data from host memory to the device input buffer identified by `algo_array_id`.

`algo_desc` is the value returned by the `rasclib_algorithm_open`() function (see "rasclib_algorithm_open Function" on page 80).

`algo_array_id` identifies the input data area and is specified in the configuration file associated with the bitstream.

`buf` is a pointer to the data in host memory.

`count` is the number of bytes in the data. Count must be an integral multiple of the size of the data area on the device.

DIAGNOSTICS

Returns `RASCLIB_SUCCESS` when all goes well.

Returns `RASCLIB_ARRAY_NOTFOUND` if the input array name cannot be found in the configuration data.

Returns `RASCLIB_READONLY_SRAM` when it is asked to send data to a read-only SRAM.

Returns `RASCLIB_ASYMETRIC_DATA` when the data size of the send and receive do not match up.

`rasclib_algorithm_get_num_cops` Function

The `rasclib_algorithm_get_num_cops` function is, as follows:

SYNOPSIS

```
int rasclib_algorithm_get_num_cops(int al_desc);
```

DESCRIPTION

`rasclib_algorithm_get_num_cops` gets the number of COPs that are participating in the algorithm referred to by `al_desc`.

DIAGNOSTICS

Returns the number of COPs involved in the algorithm or `RASCLIB_FAIL` on failure.

`rasclib_algorithm_receive` Function

The `rasclib_algorithm_receive` function is, as follows:

SYNOPSIS

```
int rasclib_algorithm_receive(int algo_desc, rasclib_algo_array_id_t
algo_array_id, void *buf, int count);
```

DESCRIPTION

`rasclib_algorithm_receive`() queues up a command that will move data from the device output buffer to host memory.

`algo_desc` is the value returned by the `rasclib_algorithm_open`() function (see "rasclib_algorithm_open Function" on page 80).

`algo_array_id` identifies the output data area and is specified in the configuration file associated with the bitstream.

`buf` is a pointer to the data buffer in host memory that will receive the data.

`count` is the number of bytes in the data. Count must be an integral multiple of the size of the data area on the device.

DIAGNOSTICS

Returns `RASCLIB_SUCCESS` when all goes well.

Returns `RASCLIB_ARRAY_NOTFOUND` if the input array name cannot be found in the configuration data.

Returns `RASCLIB_WRITEONLY_SRAM` when it is asked to receive data from a write-only SRAM.

Returns `RASCLIB_ASYMETRIC_DATA` when the data size of the send and receive do not match up.

rasclib_algorithm_go Function

The rasclib_algorithm_go function is, as follows:

SYNOPSIS

```
void rasclib_algorithm_go(int al_desc);
```

DESCRIPTION

rasclib_algorithm_go() issues a command to tell the bitstream compute engine to begin computation.

al_desc is the value returned by rasclib_algorithm_open().

DIAGNOSTICS

Returns RASCLIB_FAIL on failure, RASCLIB_SUCCESS on success.

rasclib_algorithm_commit Function

The rasclib_algorithm_commit function is, as follows:

SYNOPSIS

```
void rasclib_algorithm_commit(int al_desc,
void(*rasclib_algo_callback)(int, char*, unsigned long));
```

DESCRIPTION

rasclib_algorithm_commit() causes rasclib to send all queued commands to the kernel driver for execution on the device. The commands are sent as a single command list. Until a commit is done, all user buffers are still in use and may not be reused without the (almost certain) risk of data corruption.

al_desc is the value returned by rasclib_algorithm_open() (see "rasclib_algorithm_open Function" on page 80).

rasclib_algo_callback is a function provided by the user to handle situations such as data overruns or data underruns. It is called by rasclib when user intervention is required. Currently, it is not called, therefore, this parameter should always be NULL.

DIAGNOSTICS

Returns `RASCLIB_FAIL` on failure, `RASCLIB_SUCCESS` on success.

`rasclib_algorithm_wait` Function

The `rasclib_algorithm_wait` function is, as follows:

SYNOPSIS

```
void rasclib_algorithm_wait(int al_desc);
```

DESCRIPTION

`rasclib_algorithm_wait`() blocks until all commands that have been sent to the driver have complete execution. There is an implied commit associated with the wait. That is, before the thread of execution waits, it sends all queued commands to the kernel driver for execution (see `rasclib_algorithm_commit`). After the wait call returns, all outstanding data transfers have occurred and the application may do any buffer management that it may require.

`al_desc` is the value returned by the `rasclib_algorithm_open`() function (see "rasclib_algorithm_open Function" on page 80).

DIAGNOSTICS

Returns `RASCLIB_FAIL` on failure, `RASCLIB_SUCCESS` on success.

`rasclib_algorithm_close` Function

The `rasclib_algorithm_close` function is, as follows:

SYNOPSIS

```
void rasclib_algorithm_close(int al_desc);
```

DESCRIPTION

`rasclib_algorithm_close`() releases all host resources associated with the algorithm. It does not release any allocated devices (see `rasclib_resource_free`).

`al_desc` is the value returned by the `rasclib_algorithm_open`() function (see "rasclib_algorithm_open Function" on page 80).

`rasclib_algorithm_reg_multi_cast` Function

The `rasclib_algorithm_reg_multi_cast` function is, as follows:

SYNOPSIS

```
int rasclib_algorithm__reg_multi_cast(int al_desc, char *alg_reg,
unsigned long *data, unsigned flags);
```

DESCRIPTION

`rasclib_algorithm_reg_multi_cast sends the same data to all of the FPGAs that implement the algorithm.`

`al_desc` is the algorithm descriptor returned from the `rasclib_algorithm_open` function (see "rasclib_algorithm_open Function" on page 80).

`alg_reg` is the name of the register.

`data` is a pointer to the data to be written to the register.

`flags` can be either `RASCLIB_QCMD` or `RASCLIB_IMMED_CMD`.

- If `flags` is `RASCLIB_QCMD`, the command is queued in rasclib to be sent to the driver to be executed in order within the command list.
- If `flags` is `RASCLIB_IMMED_CMD`, the register writes are done before `rasclib_algorithm_reg_multi_cast` returns to the user.

The `rasclib_algorithm_reg_multi_cast` function supports fat registers. A *fat register* is register that spans multiple FPGA registers. Their length in bits must be an integral multiple of 64 and they must begin at the beginning of a physical register. Therefore, a fat register exactly spans an integral number of physical registers. It is assumed that the data array supplied is large enough to hold all the data for fat register. The `rasclib` library does not support partial reads or writes to fat registers.

DIAGNOSTICS

Returns RASCLIB_SUCCESS. Exits with exit status RASCLIB_MALLOC_FAILED if rasclib cannot allocate memory.

rasclib_algorithm__reg_read Function

The rasclib_algorithm_reg_read function is, as follows:

SYNOPSIS

```
int rasclib_algorithm_reg_read(int al_desc, char *alg_reg, unsigned
long *data, unsigned flags);
```

DESCRIPTION

rasclib_algorithm_reg_read reads a register from each of the FPGAs that implement an algorithm.

It is assumed that there is enough space in the array pointed to by data to handle all of the data. See "rasclib_algorithm_get_num_cops Function" on page 81.

al_desc is the algorithm descriptor returned from the rasclib_algorithm_open function (see "rasclib_algorithm_open Function" on page 80).

alg_reg is the name of the register.

data is a pointer to the data where the data read from the register will be placed.

flags can be either RASCLIB_QCMD or RASCLIB_IMMED_CMD.

- If flags is RASCLIB_QCMD, the command is queued in rasclib to be sent to the driver to be executed in order within the command list.
- If flags is RASCLIB_IMMED_CMD, the register reads are done before rasclib_algorithm_reg_read returns to the user.

The rasclib_algorithm_reg_read function supports fat registers. A fat register is register that spans multiple FPGA registers. Their length in bits must be an integral multiple of 64 and they must begin at the beginning of a physical register. Therefore, a fat register exactly spans an integral number of physical registers. It is assumed that the data array supplied is large enough to hold all the data for fat register. The rasclib library does not support partial reads or writes to fat registers

DIAGNOSTICS

Always returns `RASCLIB_SUCCESS`.

`rasclib_algorithm__reg_write` Function

The `rasclib_algorithm_reg_write` function is, as follows:

SYNOPSIS

```
int rasclib_algorithm_reg_write(int al_desc, char *alg_reg, unsigned
long *data, unsigned flags);
```

DESCRIPTION

`rasclib_algorithm_reg_write` writes the supplied data to the FPGAs that implement the algorithm. It is assumed that there is enough data supplied to satisfy one register write for each fpga in the algorithm (see "rasclib_algorithm_get_num_cops Function" on page 81).

`al_desc` is the algorithm descriptor returned from the `rasclib_algorithm_open` function (see "rasclib_algorithm_open Function" on page 80).

`alg_reg` is the name of the register.

`data` is a pointer to the data to be written to the register.

`flags` can be either `RASCLIB_QCMD` or `RASCLIB_IMMED_CMD`.

- If `flags` is `RASCLIB_QCMD`, the command is queued in `rasclib` to be sent to the driver to be executed in order within the command list.
- If `flags` is `RASCLIB_IMMED_CMD`, the register writes are done before `rasclib_algorithm_reg_write` returns to the user.

The `rasclib_algorithm_reg_write` function supports fat registers. A fat register is register that spans multiple FPGA registers. Their length in bits must be an integral multiple of 64 and they must begin at the beginning of a physical register. Therefore, a fat register exactly spans an integral number of physical registers. It is assumed that the data array supplied is large enough to hold all the data for fat register. The `rasclib` library does not support partial reads or writes to fat registers

DIAGNOSTICS

Always returns `RASCLIB_SUCCESS`.

`rasclib_cop_open` Function

The `rasclib_cop_open` function is, as follows:

SYNOPSIS

```
int rasclib_cop_open(rasclib_algorithm_id_t as_id);
```

DESCRIPTION

`rasclib_cop_open`() notifies `rasclib` that the user wants to use one of the reserved devices loaded with the bitstream associated with `al_id`. The `rasclib` library initializes all necessary data structures and returns a small integer referred to as an COP descriptor. This is similar in concept to a UNIX file descriptor. It is used internally in the library to identify a particular COP.

`al_id` is the same as the algorithm ID in the allocate and free routines.

DIAGNOSTICS

Returns a `cop_desc` > 0 on success.

Returns `RASCLIB_NO_HANDLE` if no cop handle can be found for `alg_id`.

Returns `RASCLIB_NO_DESC_NUM` if no descriptor can be found.

Returns `RASCLIB_DEV_OPEN_FAILED` if an open system call failed.

Returns `RASCLIB_MALLOC_FAILED` if a `malloc` call failed.

`rasclib_cop_send` Function

The `rasclib_cop_send` function is, as follows:

SYNOPSIS

```
int rasclib_cop_send(int cop_desc, rasclib_algo_array_id_t
algo_array_id, void *buf, int count);
```

DESCRIPTION

`rasclib_cop_send`() queues up a command that will move data from host memory to the device input buffer identified by `algo_array_id`.

`cop_desc` is the value returned by rasclib_cop_open().

`algo_array_id` identifies the input data area and is specified in the configuration file associated with the bitstream.

`buf` is a pointer to the data in host memory.

`count` is the number of bytes in the data. Count must be equal to the size of the data area on the device.

DIAGNOSTICS

Returns `RASCLIB_SUCCESS` when all goes well.

Returns `RASCLIB_NO_DESC` if cop_desc cannot be tranlated into a descriptor pointer.

Returns `RASCLIB_MALLOC_FAILED` if a `malloc` call failed.

Returns `RASCLIB_NO_ARRAY` if the input array name cannot be found in the configuration data.

Returns `RASCLIB_READONLY_SRAM` when it is asked to send data to a read-only SRAM.

Returns `RASCLIB_SIZE_MISMATCH` if the input size does not match the SRAM buffer size.

`rasclib_cop_receive` Function

The `rasclib_cop_receive` function is, as follows:

SYNOPSIS

```
int rasclib_cop_receive(int cop_desc, rasclib_algo_array_id_t
algo_array_id, void *buf, int count);
```

DESCRIPTION

`rasclib_cop_receive`() queues up a command that will move data from the device output buffer to host memory.

`cop_desc` is the value returned by `rasclib_cop_open`().

`algo_array_id` identifies the output data area and is specified in the configuration file associated with the bitstream.

`buf` is a pointer to the data buffer in host memory that will receive the data.

`count` is the number of bytes in the data. Count must be equal to the size of the data area on the device.

DIAGNOSTICS

Returns `RASCLIB_SUCCESS` when all goes well.

Returns `RASCLIB_NO_DESC` if `cop_desc` cannot be tranlated into a descriptor pointer.

Returns `RASCLIB_MALLOC_FAILED` if a `malloc` call failed.

Returns `RASCLIB_NO_ARRAY` if the input array name cannot be found in the configuration data.

Returns `RASCLIB_WRITEONLY_SRAM` when it is asked to receive data from a write-only SRAM.

Returns `RASCLIB_SIZE_MISMATCH` if the input size does not match the SRAM buffer size.

`rasclib_cop_go` Function

The `rasclib_cop_go` function is, as follows:

SYNOPSIS

```
int rasclib_cop_go(int cop_desc);
```

DESCRIPTION

`rasclib_cop_go`() issues a command to tell the bitstream compute engine to begin computation.

`cop_desc` is the value returned by `rasclib_cop_open`().

DIAGNOSTICS

Returns `RASCLIB_SUCCESS` when all goes well.

Returns `RASCLIB_MALLOC_FAILED` if a `malloc` call failed.

`rasclib_cop_commit` Function

The `rasclib_cop_commit` function is, as follows:

SYNOPSIS

```
int rasclib_cop_commit(int cop_desc, void(*rasclib_cop_callback(int,
char*, unsigned long);
```

DESCRIPTION

`rasclib_cop_commit`() causes the `rasclib` library to send all queued commands to the kernel driver for execution on the device. The commands are sent as a single command list. Until a commit is done, all user buffers are still in use and may not be reused without the (almost certain) risk of data corruption.

`cop_desc` is the value returned by `rasclib_cop_open`().

`rasclib_cop_callback` is a function provided by the user to handle situations such as data overruns or data underruns. It is called by `rasclib` when user intervention is required. Currently, it is not called, therefore, this parameter should always be NULL.

DIAGNOSTICS

Returns `RASCLIB_SUCCESS` when all goes well.

Returns RASCLIB_NO_DESC if cop_desc cannot be tranlated into a descriptor pointer.

Returns RASCLIB_MALLOC_FAILED if a malloc call failed.

rasclib_cop_wait Function

The rasclib_cop_wait function is, as follows:

SYNOPSIS

```
int rasclib_cop_wait(int cop_desc);
```

DESCRIPTION

rasclib_cop_wait() blocks until all commands that have been sent to the kernel driver have complete execution. There is an implied commit associated with the wait. That is, before the thread of execution waits, it sends all queued commands to the kernel driver for execution (see rasclib_cop_commit). After the wait call returns, all outstanding data transfers have occurred and the application may do any buffer management that it may require.

cop_desc is the value returned by rasclib_cop_open().

DIAGNOSTICS

Returns RASCLIB_SUCCESS when all goes well.

Returns RASCLIB_NO_DESC if cop_desc cannot be tranlated into a descriptor pointer.

Returns RASCLIB_MALLOC_FAILED if a malloc call failed.

rasclib_cop_close Function

The rasclib_cop_close function is, as follows:

SYNOPSIS

```
void rasclib_cop_close(int cop_desc);
```

DESCRIPTION

`rasclib_cop_close`() releases all host resources associated with the algorithm. It does not release any allocated devices (see `rasclib_resource_free`).

DIAGNOSTICS

`cop_desc` is the value returned by `rasclib_cop_open`().

`rasclib_cop_reg_read` Function

The `rasclib_cop_reg_read` function is, as follows:

SYNOPSIS

```
int rasclib_cop_reg_read(int cop_desc, char *alg_reg, unsigned long
*data, unsigned flags);
```

DESCRIPTION

`rasclib_cop__reg_read`() reads a register from a single FPGA, as specified by `cop_desc`.

It is assumed that there is enough space in the array pointed to by `data` to handle all of the data.

`cop_desc` is the cop descriptor returned from the `rasclib_cop_open` function (see "rasclib_cop_open Function" on page 88).

`alg_reg` is the name of the register.

`data` is a pointer to the data where the data read from the register will be placed.

`flags` can be either `RASCLIB_QCMD` or `RASCLIB_IMMED_CMD`.

- If `flags` is `RASCLIB_QCMD`, the command is queued in `rasclib` to be sent to the driver to be executed in order within the command list.
- If `flags` is `RASCLIB_IMMED_CMD`, the register reads are done before `rasclib_cop_reg_read` returns to the user.

The `rasclib_cop_reg_read` function supports fat registers. A fat register is register that spans multiple FPGA registers. Their length in bits must be an integral multiple of 64 and they must begin at the beginning of a physical register. Therefore, a fat register

exactly spans an integral number of physical registers. It is assumed that the data array supplied is large enough to hold all the data for fat register. The `rasclib` library does not support partial reads or writes to fat registers

DIAGNOSTICS

Returns `RASCLIB_SUCCESS` when all goes well.

Returns `RASCLIB_INVAL_FLAG` if the flag value is invalid.

`rasclib_cop_reg_write` Function

The `rasclib_cop_reg_write` function is, as follows:

SYNOPSIS

```
int rasclib_cop_reg_write(int cop_desc, char *alg_reg, unsigned long
*data, unsigned flags);
```

DESCRIPTION

`rasclib_cop__reg_write`() writes to a register of a single FPGA, as specified by `cop_desc`.

It is assumed that there is enough space in the array pointed to by `data` to fill the register.

`cop_desc` is the cop descriptor returned from the `rasclib_cop_open` function (see "rasclib_cop_open Function" on page 88).

`alg_reg` is the name of the register.

`data` is a pointer to the data area where the data to be written resides.

`flags` can be either `RASCLIB_QCMD` or `RASCLIB_IMMED_CMD`.

- If `flags` is `RASCLIB_QCMD`, the command is queued in `rasclib` to be sent to the driver to be executed in order within the command list.
- If `flags` is `RASCLIB_IMMED_CMD`, the register reads are done before `rasclib_cop_reg_write` returns to the user.

The `rasclib_cop_reg_write` function supports fat registers. A fat register is register that spans multiple FPGA registers. Their length in bits must be an integral multiple of 64 and they must begin at the beginning of a physical register. Therefore, a fat register exactly spans an integral number of physical registers. It is assumed that the data array supplied is large enough to hold all the data for fat register. The `rasclib` library does not support partial reads or writes to fat registers

DIAGNOSTICS

Returns `RASCLIB_SUCCESS` when all goes well.

Returns `RASCLIB_INVAL_FLAG` if the flag value is invalid.

`rasclib_perror` Function

The `rasclib_perror` function is, as follows:

SYNOPSIS

```
void rasclib_perror(char *string, int ecode);
```

DESCRIPTION

`rasclib_perror` prints an error string and a description of the error code.

`string` is a descriptive character string supplied by the caller.

`ecode` is an error code returned by a `rasclib` routine.

`rasclib_error_dump` Function

The `rasclib_error_dump` function is, as follows:

SYNOPSIS

```
void rasclib_error_dump(void);
```

DESCRIPTION

`rasclib_error_dump` prints the last three errors detected by rasclib. The `rasclib` library maintains a number of three-deep circular buffer of errors encountered by `rasclib`. One is a global buffer, one is associated with each algorithm, and one is associated with each co-processor (cop). `rasclib_error_dump` first prints the global errors, then the algorithm errors, and then the cop errors.

How the RASC Abstraction Layer Works

The `rasclib` library provides an asynchronous interface to the FPGA devices. What this means is that at each call into the interface to perform some action on the device (get, put, go), the command is just queued within the RASC abstraction layer. Commands are not actually sent to the driver (and, thus, to the device) until a "commit" is done. At commit time, all queued commands are sent to the driver as a list of commands. The driver sends the commands to the device. It is done this way to take advantage of any parallelism that may be available in the DMA engines or compute section. The user application cannot make any assumptions as to when the committed commands have finished executing. The user application must execute a wait call to make sure all committed commands have finished. Wait blocks until all outstanding commands have completed. When the wait call returns, all user data buffers have either been completely read from or completely filled, depending on their use.

The algorithm layer exists for two purposes. The secondary purpose is for ease of use. The application can reserve as many devices as it can get with the allocate call, then allow the abstraction layer to use them as efficiently as it can, spreading the data across as many devices as there are available and looping over the devices until all the data is consumed. The primary purpose is related to the above. By spreading the data across all the devices, the abstraction layer implements wide scaling. That is, multiple devices participate in the calculation in parallel. Deep scaling, or chaining the output of one device to the input of another device without having to stage the data through host memory, is left for later.

The input/output data buffers are assumed to be at least 16 KBs in size. In this initial release, user data requests to the algorithm layer are assumed to be an integral multiple of the data buffer in size and user data requests to the COP layer are assumed to be exactly the size of the data buffer.

So, given the above, here is how it all works. Assume a C-like program:

```
unsigned long A[SIZE],B[SIZE],C[SIZE];
main() {
        comput(A,B,C);
```

```
}
```

Where A and B are inputs to some function and C is the output. If the developer were to implement the function “compute” in an FPGA, then he would transform the above to:

```
unsigned long A[SIZE],B[SIZE],C[SIZE];
main() {
        rasclib_algorithm_request_t     al_request;
        int                             al_desc;

        al_request.algorithm_id = al_id;
        al_request.num_devices = N; /* N == number of physical devices
with al_id loaded into them */

        if (rasclib_resource_alloc(&al_request, 1) == RASCLIB_FAIL) {
                exit(1);
        }

        al_desc = rasclib_algorithm_open(al_id);
        if (al_desc == RASCLIB_FAIL) {
                exit(1);
        }

        if (rasclib_algorithm_get(al_desc, "input_one", A, SIZE *
sizeof(unsigned long)) == RASCLIB_FAIL) {
                exit(1);
        }

        if (rasclib_algorithm_get(al_desc, "input_two", B, SIZE *
sizeof(unsigned long)) == RASCLIB_FAIL) {
                exit(1);
        }

        if (rasclib_algorithm_go(al_desc) == RASCLIB_FAIL) {
                exit(1);
        }

        if (rasclib_algorithm_receive(al_desc, "output", C, SIZE *
sizeof(unsigned long)) == RASCLIB_FAIL) {
                exit(1);
        }

        if (rasclib_algorithm_commit(al_desc) == RASCLIB_FAIL) {
                exit(1);
        }
```

```
        if (rasclib_algorithm_wait(al_desc) == RASCLIB_FAIL) {
                exit(1);
        }

        rasclib_resource_free(&al_request, 1);
}
```

The `rasclib_resource_alloc` reserves devices through the device manager. The application must request all of the devices it needs for the life of the application in one call or deadlocks may result. The `rasclib_algorithm_open` tells the abstraction layer do allocate all necessary internal data structures for a logical algorithm. The two `rasclib_algorithm_send` calls pull data down to the input data areas on the COP. The `rasclib_algorithm_go` starts execution of the COP. The `rasclib_algorithm_receive` call pushes the result back out to host memory. The `rasclib_algorithm_commit` causes all of the commands that have been queued up by the previous four calls to be sent to the devices. The data is broken up on data area size boundaries and spread out over all available COPs. The `rasclib_algorithm_wait` blocks until all the command that were sent to all of the devices are complete, then returns. The `rasclib_resource_free` call tells the device manager that the application is done with the devices and that they can be allocated to other applications.

RASC Algorithm FPGA Implementation Guide

This chapter describes how to generate a bitstream and the associated configuration files for the RASC Algorithm FPGA once the Algorithm Block design has been completed. Instructions for installation, setup, and implementation are included along with examples to demonstrate different implementation flows. The implementation flow begins with a completed HDL design and ends with generating the <*alg_name*>`.bin` file, which is used by the Loader FPGA to configure the Algorithm FPGA. Other implementation flow outputs are the files `user_space.cfg` and `core_services.cfg`, which are used by the software abstraction layer to recognize the algorithm.

This chapter contains the following sections:

Implementation Overview

This section provides an overview of the RASC Algorithm FPGA implementation process. It covers the following topics:

Summary of the Implementation Flow

In this document, *implementation flow* refers to the comprehensive run of the extractor, synthesis, and Xilinx ISE tools that turn the Verilog or VHDL source into a binary bitstream and configuration file that can be downloaded into the RASC Algorithm FPGA. There are several stages to this process, but the implementation flow has been encapsulated in a Makefile flow for ease of use.

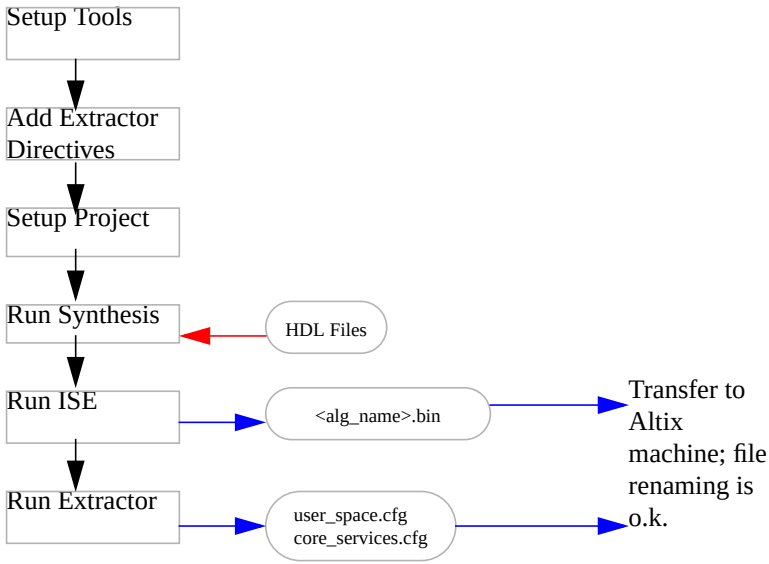

Figure 5-1 RASC FPGA Implementation Flow

There are two supported ways to create a bitstream. The first method, and recommended for most cases, is to implement the Algorithm FPGA using our pre-synthesized Core

Services Block EDIF netlist. This allows faster synthesis time and easy trials of multiple algorithms.

The second method, which allows more optimization between the Core Services and the Algorithm Block, is to synthesize from the top-down, including re-synthesis of the Core Services Block source codes.

Supported Tools and OS Versions

The supported flow has been tested using the tool versions and operating system shown in Table 5-1.

Table 5-1 Supported Implementation Tools

Tool	Version
OS	Red Hat Enterprise Linux 3.0
Shell	Bash or tcsh
Python	Python 2.0 or higher
Synthesis	Synplicity - Synplify Pro 8.4
ISE	Xilinx ISE 8.1.02i

SGI suggests that you consult the reference documentation associated with the supported tools of your choice for the details on tool setup.

- ISE Software Manuals

  http://www.xilinx.com/support/sw_manuals/xilinx8/index.htm
- Synplify Pro Documentation

  http://www.synplicity.com/literature/index.html

Installation and Setup

SGI Altix System Installation

RASC software is distributed either on a CD-ROM or through file transfer protocol (FTP). If an International Standards Organization (ISO) file is received through FTP, put it in a convenient directory in the SGI Altix system. If the directory `/mnt/cdrom` does not exist, create that directory with the following command:

```
mkdir /mnt/cdrom
```

At the directory where the ISO file is saved, execute the following command:

`mount -o loop` *iso_file_name* `/mnt/cdrom`

The CD-ROM image will be visible in the directory `/mnt/cdrom`.

On the CD-ROM image, there will be one or more files with the extension `.rpm` and a `readme.txt` file. Read the `readme.txt` file for further instructions for installing the software.

PC Installation

This section describes the installation and setup of the RASC Algorithm FPGA Developer's Bundle. Please note that the Xilinx ISE tools and the synthesis tool of choice must be installed on an IA-32 Linux system and each user must have the associated environment variables defined for their use.

The RASC Algorithm FPGA Developer's Bundle comes as a `tar.gz` file. This `tar.gz` file is available in the Altix system after the rasc-xxx.ia64.rpm is installed. It is in the directory `/usr/share/rasc/ia32_dev_env`. Run the executable `ia32_dev_env_install` and read the license agreement. Agree to the agreement to extract the file `ia_32_dev_env`. Transfer this file to a directory in the PC. If there is an older version of the bundle software, remove it. Unzip the `tar.gz` file using the following command:

`tar -zxvf` *file_name*

The Developer's Bundle requires setting one additional environment variable, RASC, to identify the Bundle installation directory to the supporting Makefiles (see Table 5-2).

Table 5-2 Environment Variables Required for Bundle Environment

Name	Description
RASC	RASC Algorithm FPGA Developer's Bundle Installation Directory
XILINX	Xilinx tools installation directory
SYNPLIFY	Synplify Installation directory
LIB	Required Synplify variable

The directory hierarchy within the RASC Algorithm FPGA Developer's Bundle is shown in the README file at the top level. A description of the top level directories included in the Bundle is shown in Table 5-3.

Table 5-3 Top Level Directory Descriptions

Directory	Description
examples	Contains subdirectores with example algorithm source code and implementation files.
design	Contains Verilog source codes for the top-level FPGA wrapper and the Core Services Block modules.
pd	Additional files needed for physical design / implementation: synthesis and ISE constraint files, the pre-synthesized Core Services Block netlist and the macros needed for re-synthesis of the Core Services Block.
implementations	Contains template files for running the bitstream implementation flow (Makefiles and project files).
dv	Functional simulation components: testbench with SSP stub and QDR2-SRAM models, example diagnostic stimulus code, and scripts for VCS simulator.

When the Developer's Bundle is installed and the $RASC variable is set, the user can proceed to experiment with the example implementations.

Implementation Constraint Files

Synthesis Using Synplify Pro

A synthesis constraint file (`.sdc`) is provided for use with the Synplify Pro tool. It is used to provide additional inputs to the synthesis tool on desired mapping, netlist formats, and post-synthesis timing estimates.

The synthesis constraint file is specified in the synthesis project file `<alg_name>.prj`. If a user desires to add constraints for portions of the Algorithm Block, this can be done by editing the `.sdc` file manually or through the associated synthesis GUI.

The synthesis constraint files are located under the install directory at `$RASC/pd/constraints/synthesis`.

Synthesis Using XST

A script file (`.scr`) is provided for use with the XST tool. An XST script is a set of various options used to run synthesis. It is utilized to control various aspects of the synthesis process to meet your design goals or obtain the best implementation of your design.

The synthesis script is used along with the synthesis project file `<alg_name>_xst.prj`. If a user desires to add constraints for portions of the Algorithm Block, this can be done by creating an `.xcf` file. To specify the XCF file in the script, use the `-uc` switch with `run` command.

The XST script file should be located under the synthesis project directory of each algorithm.

Table 5-4 Synthesis Constraint Files Provided

File	Synthesis Tool	Description
`acs_top.sdc`	Synplify Pro	Basic synthesis constraint file
`<alg_name>_xst.scr`	XST	General (global) constraints

ISE (User Constraint File)

A top level user constraint file (ASCII) needs to be used with the ISE tools to guide placement and routing and to achieve timing-driven placement results. The UCF file is

generated from separate constraint files (see Table 5-5) located in the directory `$RASC/pd/constraints/ise`.

Table 5-5 Implementation Constraint Files Provided

File	Description
`ssp.ucf`	Timing constraints and I/O assignments
`virtual_gnd.ucf`	Virtual ground I/O assignments
`prohibit.ucf`	Prohibit constraints for specific sites
`qdr_bank0.ucf, qdr_bank1.ucf, ..., qdr_bank4.ucf, and qdr_misc.ucf`	Location and specific constraints for QDR banks

The constraint file should always be included in the design, and the provided constraints should not be modified. However, a user can add to the constraint file to put in constraints related to the Algorithm Block. If a user desires to rename the constraint file, the file name change also needs to be applied to the `Makefile.local` by adding a new variable *CURR_UCF_FILE*. An alternate directory location can be modified by adding the *UCF_DIR* variable in `Makefile.local`.

The constraint file provided in the Bundle provides the following types of constraints:

- Period specifications for all internal clock signals, both for the Core Services Block and the Algorithm Block
- I/O Pad placement constraints and I/O standard settings
- Setup and hold constraints for input pads
- DCM attributes settings
- Cross clock domain path constraints
- BUFG and DCM placement constraints

All constraints are written in the Xilinx-specified User Constraint File format.

Adding Extractor Directives to the Source Code

Symbol table information is required for GDB to correctly display internal FPGA values and for the RASC Abstraction Layer (RASCAL) to communicate parameters and data that should be written to and read from the SRAM banks and internal registers. A python script called *extractor* parses all the Verilog, VHDL, and header files in an arbitrary algorithm directory that is passed as an argument. The user specifies the top level algorithm directory by setting the *ALG_DIR* variable in the `Makefile.local` file, to be discussed later. When the extractor script is called, it generates two configuration files that communicate the necessary information for the RASCAL and GNU source-level debugger (GDB) to communicate with the hardware correctly.

The extractor script parses comment lines that contain special extractor directives from within source and header files in the algorithm directory. There is a template in the `examples` directory, in addition to full examples for each of the example algorithms. The comment fields can be located in one or more files below the directory specified in the second argument to the extractor call.

The command to run the extractor has been automated in the Makefile flow discussed in detail later. The user does not need to run extractor at the command line, but can issue the `make extractor` command after the Makefiles have been set up.

Inserting Extractor Comments

The comment fields that need to be inserted within the algorithm source or header code are: *core services version*, *algorithm version*, *sram* (denoting where data will be read from and written to on the sram interface), *register in* (for parameters set through an application writer's code), and *register out* (for a section of code that needs to be mapped to a debug register).

The comments can be located anywhere within a Verilog, VHDL or header file. They can have Verilog or VHDL syntax ("//" or "--" respectively). Extractor comments begin with the tag "//extractor" in the source file. The general format is shown below.

//extractor <type>:[<value/specifier1> ... <value/specifierN>] (Verilog)

- - extractor <type>:[<value/specifier1> ... <value/specifierN>] (VHDL)

For details on each extractor comment, see Table 5-6.

Example of Comments in a Verilog, VHDL, or header File

Table 5-6 Extractor Comment Fields

Type	Description	Value	Example
CS	Core Services Version: Tag detailing which iteration of core services was used in this bitstream.	<version>.<revision>	//extractor CS:0.7
VERSION	Algorithm Version: Tag for the type and version number of the algorithm in this bitstream. This is user defined but it should match the first register in the debug space (see Chapter 3, "RASC Algorithm FPGA Hardware Design Guide"for more details.)	<version>.<revision>	//extractor VERSION:0.12
SRAM	Array: Tag for the SRAM inputs and outputs to the algorithms.	<array name> <number of elements in array> <bit-width of each element> <SRAM bank for array> <starting byte offset> <array direction: in | out |internal> <data type for array elements: u=unsigned> <array use attribute: stream | fixed>	//extractor SRAM:a_in 512 64 sram[0] 0x0000 in u stream

Table 5-6 Extractor Comment Fields **(continued)**

Type	Description	Value	Example
REG_IN	register in: Tag for the registers that are PIO written as specified by the application and where the value will be mapped in the address space.	<register name> <bit width of register> <data type for element> <alg_def_reg mapping> <bit range mapping of alg_def_reg>	//extractor REG_IN:match 32 u alg_def_reg[0][31:0]
REG_OUT	register out: Tag for the registers that are PIO written or read by the application or the debugger and where the value will be mapped in the address space.	<register name> <bit width of register> <data type for element> <debug port mapping> <bit range mapping of debug_port>	//extractor REG_OUT: a 16 u debug_port[1] debug_port[1] [15:0]

A sample of the comments that might be included within a source file is:

```
// extractor VERSION: 6.3
// extractor CS: 1.0
// extractor SRAM:a_in  2048  64 sram[0] 0x0000 in  u stream
// extractor SRAM:b_in  2048  64 sram[0] 0x4000 in  u stream
// extractor SRAM:c_in  2048  64 sram[0] 0x8000 in  u stream
// extractor SRAM:d_out 2048  64 sram[1] 0x0000 out u stream
// extractor REG_OUT:alg_id 32 u debug_port[0][63:32]
// extractor REG_OUT:alg_rev 32 u debug_port[0][31:0]
// extractor REG_OUT:rd_addr 64 u debug_port[1]
// extractor REG_OUT:rd_data_sram0_lo 64 u debug_port[2]
// extractor REG_OUT:rd_data_sram0_hi 64 u debug_port[3]
// extractor REG_OUT:wr_addr 64 u debug_port[4]
// extractor REG_OUT:wr_data_sram1_lo 64 u debug_port[5]
// extractor REG_OUT:wr_data_sram1_hi 64 u debug_port[6]
// extractor REG_OUT:cntl_sigs 64 u debug_port[7]
```

Implementation with Pre-synthesized Core

The first recommended implementation flow uses the pre-synthesized version of the Core Services Block. The bulk of the Core Services Block (excluding clocking, resets, and I/O) has been pre-synthesized and has soft-fixed source code (we do not support customer modifications). The implementation time could be significantly reduced by using the pre-synthesized core, but results vary depending on the loading of signals between the Algorithm Block and the Core Services Block. We suggest that you first try this method, and if timing is not met due to loading issues on the Algorithm Block / Core Services interface, re-synthesize the full design from scratch (discussed later).

The pre-synthesized Core Services Block is located under the install directory at `$RASC/pd/acs_core/acs_core.edf`. When you synthesize the full FPGA, the synthesis project views the acs_core.v file as a black box, and the NGDBUILD tool will stitch in `acs_core.edf` in its place.

Each algorithm project will have its own associated implementation project directory, which can be located anywhere in the directory space of the user. The user customizes template Makefiles and synthesis project files to run the desired bitstream generation flow. Each project directory must contain the following three files, the first two provided in the `$RASC/implementations/templates` directory:

- **`Makefile`** (linked or copied from the templates directory).

  This Makefile contains general definitions and targets necessary to run the synthesis and ISE tools. It includes the Makefile.local file. This file is meant to be linked into each new algorithm project directory. Customizations should be added to Makefile.local; it is recommended to keep this file common between all projects.

- **`Makefile.local`**

  This included Makefile allows custom definitions and targets for each algorithm project. Here you can define the synthesis project name, specify different constraint files, and override tool options.

- **Synthesis Project File**

  – **Synplify Pro** <*alg_name*>`.prj`.

  This file defines the algorithm source files and synthesis tool options. ***This is where you need to add / modify the paths to the algorithm code.*** The Synplify Pro project file is a TCL script, which can also be opened and run by the GUI version of the synthesis tools. Read the synthesis tool documentation for more information on the TCL commands available. You will notice that the template

synthesis project file does not contain any source files for the Core Services Block. All Core Services files are stored in another file in the templates directory, `acs.files`. The Makefile flow combines `acs.files` with the synthesis project file of the user to create a complete synthesis project (renamed as `full_<`*alg_name*`>.prj`). If you are creating a synthesis project from scratch or not using the Makefile flow, you will need to copy `acs`.files into your synthesis project.

– **XST** <*alg_name*>`_xst.prj`.

This file only defines the algorithm source files. ***This is where you need to add / modify the pointers to the algorithm code.*** The synthesis options are included in a separate script file <*alg_name*>`_xst.scr` which can be opened and run by the synthesis tools.

Examples of each of these required files are located in the templates directory, as well as the included example algorithm implementation project directories:

```
$RASC/examples/<alg_name>/impl
```

An example of the steps for setting up a new implementation directory is, as follows:

```
% cd $RASC/examples
% mkdir my_alg
% mkdir my_alg/impl
% cd my_alg/impl
% ln -s $RASC/implementations/templates/Makefile Makefile
% cp $RASC/implementations/templates/Makefile.local Makefile.local
% cp $RASC/implementations/templates/synplify_template.prj
      my_alg.prj
% ls -lat
   2748 Aug 27 11:42 my_alg.prj
   1433 Aug 27 11:42 Makefile.local
   21 Aug 27 11:42 Makefile ->
   ../../../implementations/templates/Makefile
```

Edit `my_alg.prj` to set the variable *ALG_DIR* to `$RASC/examples/my_alg/impl.`

Makefile.local Customizations

Any Makefile variables you want to redefine can be changed using the `Makefile.local` file. This allows each project you create to use the same base Makefile, but have project-specific customizations in a separate file. The key Makefile variables

that are commonly redefined for each project are listed here. Default settings are provided with the `Makefile.local` version in the templates directory.

Synthesis Project Customization

Table 5-7 shows common `Makefile.local` variable settings.

Table 5-7 Common Makefile.local Variable Settings

Variable	Example	Definition
`ALG_DIR`	`$RASC/examples/alg_simple`	Pointer to top level algorithm directory. This is a full path, not a relative path.
`SYNTHESIS_PROJ`	<any name>, e.g., `alg_simple_hc`	Name of the synthesis project; the base filename to Synplify's *.prj file.
`SYNTHESIS_DIR`	<any relative path directory>, e.g. `rev_1`	Output directory for synthesis and implementation flow, relative to project's implementation directory.
`DEFAULT_SYNTHESIS_TOOL`	`synplify_pro`, `ise_xst`	Specify synthesis tool to use for default make flow
`SYN_RESULT_EXT`	`edf`, `ngc`	Specify the synthesis tool's output file extension
`LOCAL_SOURCE_DIR`	<any full path list of directories>	List of directories of additional edf/sources related to the algorithm that need to be read by ngdbuild (default: none).
`MAP_OPTIONS`, `PAR_OPTIONS`	Any ISE allowed options	Set ISE tool options for a specific run.
`UCF_DIR`, `CURR_UCF_FILE`, etc.		Additional variables to override the Makefile settings

Synplify Pro

The Synplify Pro tool uses a <*alg_name*>`.prj` file to specify the HDL files for synthesis and other compiler, mapping, and output options. When creating a new project for a new

algorithm, the algorithm related files need to be added to the project file so that the synthesis tool knows which algorithm to compile. The user should specify the location of the files `alg_block_top.v` and alg.h, along with any submodules. For example, if you are compiling an algorithm whose two source files `alg_block_top.v` and `sort_byte.v` are located under `$RASC/examples/alg_simple_v/`, you would need to add the following lines to the project file:

```
add_file -verilog "$RASC/examples/alg_simple_v/alg_block_top.v"
```

Please note that for some projects it is useful to read the EDIF netlist of Core Services Block during the synthesis. To do that add the following line to the project file:

```
add_file -edif "$RASC/pd/acs_core/acs_core.edf"
```

XST

The XST tool uses a <*alg_name*>`_xst.prj` file to specify the HDL files for synthesis and creates Xilinx specific netlist files called `ngc` file. All other compiler (for example, verilog include directories), synthesis, and output options are included in a separate script file `<alg_name>_xst.prj`. When creating a new project for a new algorithm, the algorithm related files need to be added to the project file so that the synthesis tool knows which algorithm to compile. The user should specify the location of the files `alg_block_top.v` and alg.h, along with any submodules. For example, if you are compiling an algorithm whose source file `alg_block_top.v` is located under `$RASC/examples/alg_simple_v/`, you would need to add its full or relative path to the project file:

```
verilog work "../../example/alg_simple_v/alg_block_top.v"
```

Makefile Targets

After the Makefile, `Makefile.local`, and synthesis project file are set up and customized, the next step is to run the `make` command to generate the bitstream and configuration files for download to the RASC hardware.

FPGA implementation comprises many sub-steps such as synthesis, ngdbuild, map, place-and-route, bitgen, and timing analysis. The user can choose to run the entire flow or individual subsections based on the `Makefile` target. If no target is specified, the full implementation flow is executed:

```
% cd $RASC/implementations/alg_simple_v
% make all
```

Other available targets are described in Table 5-8. The default command is `make all`.

Table 5-8 Makefile Targets

Command Line	Description
`make all`	Runs extractor, synthesis and ISE tools, results in the `acs_top_ssmap.bin` and related `.cfg` files, ready to download to FPGA. This is the default make target.
`make extractor`	Run the Extractor script to generate the configuration files based on the source code comments,
`make synplify_pro`	Run Synplify Pro tool alone. Results in EDF file.
`make amplify`	Run Amplify tool alone. Results in EDF file.
`make ise_xst`	Run XST tool alone. Results in NGC file.
`make oneloop`	Run ISE tools, does not rerun synthesis. Performs 1 PAR iteration. Results in NCD output
`make ngdbuild`	Run NGDBUILD alone.
`make map`	Run MAP alone.
`make par`	Run PAR alone.
`make trce`	Run TRACE (static timing analysis) alone.
`make bitgen`	Run BITGEN alone.
`make clean`	Remove all intermediate and output files.

Full-chip Implementation

A full-chip implementation refers to synthesis of the entire FPGA design, including a re-synthesis of the Core Services Block. The implementation steps for this flow are similar to those for the flow using the pre-synthesized Core Services Block netlist, with differences specified in the `Makefile.local` and the synthesis project file.

Makefile.local Customizations

When synthesizing the full FPGA, including a re-synthesis of the Core Services Block, we only support the use of Synplify Pro. The Core Services Block design has been optimized for use with these tools and results from using another synthesis tool will not likely achieve the same quality-of-results and timing closure. Therefore, the `Makefile.local` for the full chip project must use the following Makefile variable settings (see Table 5-9).

Table 5-9 Required Full Chip Makefile Variable Settings

Variable	Possibilities
DEFAULT_SYNTHESIS_TOOL	synplify_pro
SYN_RESULT_EXT	edf

Synthesis Project Customization

A synthesis option must be set to allow the Core Services Block to be re-synthesized. This option is in the form of a Verilog compiler directive, which can be added to the synthesis project file with the following TCL command:

```
set_option -hdl_define -set RESYNTHESIZE_CS=1
```

This option is commented in the `<alg_name>.prj` and must be uncommented by the user desiring to re-synthesize the Core Services Block. The rest of the synthesis project should be the same as described earlier.

Implementation File Descriptions

Table 5-10 includes the commonly referenced files. The synthesis and ISE tools use a lot of intermediate files. To find out more information about files not discussed here, please consult the on-line reference guides mentioned above.

Table 5-10 Commonly Referenced Files

Name	Type	Produced By	Description
BIN	Binary	Bitgen	Contains only configuration data for the FPGA. The .bin has no header like the .bit file. Meant for input to non-Xilinx tools such as the FPGA Loader Device Driver.
BIT	Binary	Bitgen	Contains proprietary header information as well as configuration data. Meant for input to other Xilinx tools such as PROMGen and Impact.
CFG	ASCII	Extractor	Configuration file used by the Abstraction Layer software to recognize features of the Algorithm and Core Services.
EDIF	ASCII	Synplify Pro Synthesis Tool	EDIF Netlist.
NGC	ASCII	XST Synthesis Tool	NGC Netlist.
NCD	Data	Map, PAR	Flat physical design database. After PAR, this is the database file with the full placed-and-routed design.
NCF	ASCII	Synplify Pro Synthesis Tool	Constraint file produced by the Synplify Pro synthesis tool to be associated with the EDIF file of the same name.
PRJ	ASCII	User or Synthesis Tool	Synthesis Project definition file.
SDC	ASCII	User (text editor) / Synthesis tools GUI (Synplify Pro or Amplify)	Provides directives and attributes for the synthesis tool to use while it is compiling the design
TWR	ASCII	Trace	Timing report file. Provides details on failing timing paths.
TWX	XML	Trace	Timing report file, with the same information as TWR but can be opened by the Timing Analyzer tool.
UCF	ASCII	User (text editor) / ISE Constraints Editor or PACE	User-specified logic constraints file. Constrains pinout, placement and timing constraints.

Running and Debugging Your Application

This chapter describes how to run and debug your application and covers the following topics:

Loading the Bitstream

The Device Manager maintains a bitstream registry of algorithm bitstreams that can be loaded and executed using the RASC abstraction layer. The `devmgr` user command is used to add, delete, and query algorithms in the bitstream registry. An algorithm is identified by its user supplied name and consists of a set of files, as follows:

- Algorithm bitstream
- Configuration file for the algorithm bitstream
- Core services configuration file

All of these files must be accessible by the `devmgr` user command when the algorithm is added to the bitstream registry.

The `devmgr` user command executes on the target Altix system, but the algorithm bitstream and configuration files are built on a PC using one of several FPGA development tools.

Note: For detailed information using the `devmgr` user command, see "RASC Device Manager" on page 118.

To make the algorithm files available to the `devmgr` command, perform the following:

1. FTP the three files from the PC to the Altix system. The files can reside anywhere on the Altix system that can be accessed by the `devmgr` command. A convenient location is, as follows:

   ```
   /usr/src/RASC/bitstream/your_algorithm_directory
   ```

2. Create the bitstream and optionally a user specific subdirectory in /usr/src/RASC, if they do not already exist on the Altix system.

3. Execute the FTP command from the PC to transfer the algorithm bitstream, its configuration file, and the core services configuration file to the user generated subdirectory.

4. Login to the Altix system and execute the `devmgr` user command to add the algorithm to the bitstream registry giving the algorithm a user supplied name. All references to the bitstream in the bitstream registry is made using this name. Note that the bitstream file suffix must be `*.bin` and the configuration files suffix must be `*.cfg`.

   For example, assume that an algorithm bitstream and configuration files have been FTP'd to the Altix system in subdirectory `dog`. A list of the files in `dog` could look like the following:

   ```
   /usr/src/RASC/bitstream/dog/spot.bin
   /usr/src/RASC/bitstream/dog/spot.cfg
   /usr/src/RASC/bitstream/dog/core_services.cfg
   ```

5. This `devmgr` command could be executed on the Altix system to add the algorithm to the bitstream registry with the name `bark`, as follows:

   ```
   devmgr -a -n bark -b /usr/src/RASC/bitstream/dog/spot.bin
   ```

   Notice that the full path name of the bitstream must be specified. If the algorithm bitstream configuration file is not called `spot.cfg`, the `devmgr -c` option must also specify the full path name of the algorithm configuration file. Similarly, if the core services file is not `core_services.cfg`, the `-s` option must specify its full path name.

RASC Device Manager

This section provides detailed information on using the device manager software and covers the following topics:

RASC Device Manager Overview

The RASC device manager maintains a bitstream registry and a FPGA hardware inventory. You can add, delete, and list bitstreams in the bitstream registry using the `devmgr` command and can allocate available FPGAs by making requests using the RASC abstraction layer software. You can also use the `devmgr` command to list the FPGAs in the inventory.

You can manually load an algorithm into an FPGA on demand using the `devmgr` user command `-l` option.

You can use the `devmgr` -r option to direct the Device Manager to always reload an algorithm into an FPGA, even when that FPGA is already loaded with the current version of the requested algorithm.

When an FPGA is allocated to a user, the requested bitstream is loaded into those FPGAs. Allocation requests must be made using abstraction software (see Chapter 4, "RASC Abstraction Layer" for specific API calls) and are passed to the device manager.

Each entry in the bitstream registry is identified by the name, which is limited to 63 characters, that is assigned by the user when the bitstream is added to the bitstream registry. The entry also includes copies of the bitstream binary file, the bitstream configuration file, and the cores services configuration file. The location of these files on the local machine is also specified by the user when the entry is added.

You can use the `devmgr -m unavail` option to mark an FPGA device node as unavailable for application allocation. You can use the `devmgr -m avail` option to mark an FPGA device node as available for application allocation.

The `devmgr -v` option shows the `devmgr` command build date and time. This information can be helpful when diagnosing Device Manager software problems.

RASC Device Manager Structure

The device manager is structured as client and server components that communicate over a TCP/IP connection. Both the `devmgr` user command and the services requested by the abstraction layer interface with the client side of the device manager. The bitstream registry management and FPGA allocation and loader services occur on the server side. Before a bitstream can be added to the bitstream registry, all of the bitstream files must exist on the machine where the server is running.

Using the Device Manager Command (`devmgr`)

The `devmgr` user command requires one action argument that may require additional arguments, and accepts an optional client debugging argument. The command actions are, as follows:

- "Add a Bitstream To the Bitstream registry" on page 121
- "Delete a Bitstream From the Bitstream registry" on page 121
- "List the Contents of a Bitstream registry" on page 121
- "Update an Algorithm in the Bitstream registry" on page 122
- "List the FPGAs in the Inventory" on page 122
- "Mark an FPGA as Available or Unavailable" on page 123
- "Turn Debugging On or Off" on page 124
- "Device Manager Load FPGA Command" on page 124
- "Device Manager Reload FPGA Command" on page 125
- "Device Manager Version Information" on page 126

Guidelines for using the `devmgr` command are, as follows:

- The add and delete actions require a bitstream registry entry name; the add action also requires the binary bitstream file name, which must have the "`.bin`" suffix.
- If the bitstream configuration file name is not specified, it defaults to the bitstream file basename with a "`.cfg`" suffix in the same directory as the binary bitstream file.

- If the cores services file name is not specified, it defaults to "`core_services.cfg`" in the same directory as the binary bitstream file.

Add a Bitstream To the Bitstream registry

The devmgr command syntax to add a bitstream to the bitstream registry is, as follows:

`devmgr -a -n` *algorithm name* `-b` *bitstream file name* `[-c` *bitstream config file* `name] [-s` *core services config file name*`] [-y on | off]`

The `devmgr` command accepts the following options:

`-a`	Adds an entry to the bitstream registry.
`-n` *algorithm name*	Names an entry in the bitstream registry.
-b bitstream file name	Indicates the location of the bitstream file.
-c bitstream config file name	Indicates the location of the configuration file.
`-s` *core services config file name*	Indicates the location of the core services config file.
`-y on | off`	Turn client debugging on or off.

Delete a Bitstream From the Bitstream registry

The `devmgr` command syntax to delete a bitstream from the bitstream registry is, as follows:

`devmgr -d -n` *algorithm name* `[-y on | off]`

The `devmgr` command accepts the following options:

`-d`	Deletes an entry from the bitstream registry.
`-n` *algorithm name*	Identifies the name of an entry in the bitstream registry.
`-y on | off`	Turns client degugging on or off.

List the Contents of a Bitstream registry

The devmgr command syntax to list a specific entry or all entries in the bitstream registry is, as follows:

`devmgr -q [-n` *algorithm name*`][-y on | off]`

The `devmgr` command accepts the following options:

-q	Lists all entries in the bitstream registry.
`-n` *algorithm name*	List a specific algorithm in the bitstream registry.
-y on | off	Turns client debugging on or off.

Update an Algorithm in the Bitstream registry

You can use the `-u` option to update an existing algorithm in the Device Manager bitstream registry with a new version of the bitstream file, bitstream configuration file, and core services configuration file. All three files are replaced in the bitstream registry with the new files specified with the command. The `-u` option is logically equivalent to a `-d` option followed by the `-a` option, where both options specify the same algorithm name.

After a successful update of an algorithm in the bitstream registry, any FPGAs currently loaded with the same algorithm name will be reloaded with the new version of the algorithm the next time that FPGA is allocated.

The `devmgr` command syntax to update an algorithm in the bitstream registry is, as follows:

`devmgr -u -n` *algorithm name* `-b` *bitstream file name* `[-c` *bitstream config file* `name] [-s` *core services config file name*`] [-y on | off]`

The `devmgr` command accepts the following options:

`-u`	Updates the entry in the bitstream registry.
`-n` *algorithm name*	Names an entry in the bitstream registry.
`-b` *bitstream file name*	Indicates the location of the bitstream file.
`-c` *bitstream config file name*	Indicates the location of the configuration file.
`-s` *core services config file name*	Indicates the location of the core services config file.
`-y on | off`	Turn client debugging on or off.

List the FPGAs in the Inventory

You can use the `devmgr` command to list the FPGAs in the inventory. The list indicates whether the FPGA is available or in use, the system node where the FPGA resides (its device node ID), part number, and manufacture (mfg) number. FPGAs that are in use also

list the Linux process ID (pid) and Pthread thread ID (thread id) of the current FPGA user, and the name of the algorithm currently loaded in to the FPGA.

The `devmgr` command syntax to list a specific entry or all entries in the FPGA inventory is, as follows:

```
devmgr -i [-N odd integer][-y on | off]
```

The `devmgr` command accepts the following options:

`-i`	Lists all entries in the inventory.
`-N` *odd integer*	List the specific FPGA device node ID.
`-y on | off`	Turns client debugging on or off.

Mark an FPGA as Available or Unavailable

You can mark an FPGA device node as unavailable for use by an application with the `devmgr -m unavail` option. If the FPGA is currently in use, the mark action will remain pending until the FPGA is freed; then the FPGA is marked as unavailable.

The `devmgr -m avail` option marks an FPGA device node as available for allocation by an application.

You must specify the target FPGA device node ID using the `-N` option whenever you use the `devmgr -m` option.

You can use the `devmgr -i` option to display the availability of an FPGA device (see "List the FPGAs in the Inventory" on page 122).

The `devmgr` command syntax to to mark an FPGA device as available or unavailable is, as follows:

```
devmgr -m avail | unavail -N odd integer [-y on | off]
```

The `devmgr` command accepts the following options:

`-m avail | unavail`	Marks an FPGA device node as available or unavailable for use.
`-N odd integer`	Specifies the target FPGA device node ID.
`-y on | off`	Turns client debugging on or off.

Turn Debugging On or Off

The `devmgr` command syntax to turn server or client debugging on or off is, as follows:

```
devmgr -x on | off [-y on | off]
```

The `devmgr` command accepts the following options:

`-x on | off` Turns server debugging on or off.

`-y on | off` Turns client debugging on or off.

Device Manager server and client debugging can also be turned on or off using environment variables. This is the only way to control Device Manager client debugging for RASC applications. The `devmgr_server` command `-x` and `-y` options override any environment variable settings.

To turn client debugging on or off, set the `RASC_DEVMGR_CLIENT_DEBUG` variable, as follows (for the Korn shell):

```
export RASC_DEVMGR_CLIENT_DEBUG=on
export RASC_DEVMGR_CLIENT_DEBUG=ON

export RASC_DEVMGR_CLIENT_DEBUG=off
export RASC_DEVMGR_CLIENT_DEBUG=OFF
```

To turn server debugging on or off set the `RASC_DEVMGR_SERVER_DEBUG` variable, as follows (for the Korn shell):

```
export RASC_DEVMGR_SERVER_DEBUG=on
export RASC_DEVMGR_SERVER_DEBUG=ON

export RASC_DEVMGR_SERVER_DEBUG=off
export RASC_DEVMGR_SERVER_DEBUG=OFF
```

Device Manager Load FPGA Command

You can manually load an algorithm into an FPGA on demand using the `devmgr` user command `-l` option. Usually the algorithm is automatically loaded into the FPGA by the RASC abstraction layer as part of the co–processor (COP) allocation request. FPGAs loaded by the abstraction layer are marked as in use until the RASC abstraction layer frees the FPGA. FPGAs manually loaded using the `devmgr` command are not marked in use and their algorithm can be immediately changed by another user command or the RASC abstraction layer after the `devmgr` command terminates.

The `-l` option requires an algorithm name specified with the `-n` option and a system node ID identifying the FPGA location with the `-N` option.

Note: Note that I/O nodes, including FPGA nodes, are always an odd number. Use the `devmgr` command with the `-i` option to view the FPGA inventory and their node IDs.

The `devmgr` command syntax to manually load an FPGA with an algorithm is, as follows:

`devmgr -l -N` *FPGA odd numbered node ID* `-n` *algorithm name* `[-y on | off]`

The `devmgr` command accepts the following options:

`-l`	Loads this FPGA with this algorithm.
`-N` *FPGA odd numbered node ID*	FPGA device odd numbered node ID.
`-y on | off`	Turns client debugging on or off.

Device Manager Reload FPGA Command

By default, the Device Manager only loads an algorithm into an FPGA under the following conditions:

- The FPGA is **not** already loaded with the requested algorithm, that is, the name of the algorithm loaded in the FPGA is not the same as the requested algorithm name.
- A newer version of the algorithm is available in the bitstream registry.

Otherwise, the FPGA is not loaded, eliminating the latency needed to load the FPGA.

You can use the -r option to direct the Device Manager to always reload an algorithm into an FPGA, even when that FPGA is already loaded with the current version of the requested algorithm. Always reloading an FPGA can be useful when debugging a suspected hardware of software FPGA load problem. Specify `on` to enable always reload an FPGA and `off` to only reload an FPGA when needed.

To determine whether the always reload option is on or off, use the `devmgr -i` command.

The `devmgr` command syntax to always reload an FPGA or only reload an FPGA when it is needed is, as follows:

```
devmgr -r on | off [-y on | off]
```

The `devmgr` command accepts the following options:

`-r on | off`	Turns the always reload an FPGA action on or off.
`-y on | off`	Turns client debugging on or off.

Device Manager Version Information

You can use the `devmgr -v` option to show the build date and time of the `devmgr` command. This information can help you diagnose Device Manager software problems.

The `devmgr` command syntax to show version information is, as follows:

```
devmgr -v [-y on | off]
```

The `devmgr` command accepts the following options:

`-v`	Shows `devmgr` version information.
`-y on | off`	Turns client debugging on or off.

Device Manager Server Command

The device manager includes a server component that manages the algorithm bitstream registry, allocates and frees FPGA devices from its inventory, and loads algorithm bitstreams into the FPGAs. Both `devmgr` user command and abstraction layer requests are sent over a TCP connection to the server to be processed, and the response is sent back over the same connection to the requester.

By default, the server listens for incoming connection requests on port number 9999. The port number can be changed by the system administrator by adding a port definition to the TCP/IP services file. Usually this file is located, as follows:

```
/etc/services
```

An example of how to change the listening port to 9998 is, as follows:

```
rasc_devmgr      9998/tcp          # RASC Device Manager Server
```

Note: After you make a port number change, you must restart the server (see the "Using the Device Manager (devmgr_server) Command").

Using the Device Manager (devmgr_server) Command

After changing a port number as described in "Device Manager Server Command", the server must be started before any requests can be processed. The `devmgr_server` command is used to start the server component of the device manager. The debug option enables server initialization debugging, which cannot be done using the `devmgr` command with the `-x` on action. By the time a `devmgr` command can request that server debugging be turned on, the server initialization has already completed.

The `devmgr_server` command syntax is, as follows:

```
devmgr_server [-x on | off]
```

where the `-x` option turns server debugging on or off. The default setting is off.

Device Manager Logging Facility

The Device Manager client and server provides an optional logging facility that tracks Device Manager requests and responses. Client or server logging is independently enabled by specifying the log file name using an environment variable. Log entries are time stamped and marked with the owning process and thread identifiers. The server log entries also include the process and thread identifiers of the client.

A summary of client application Device Manager activity can be obtained by enabling client logging for the application. Each client should use its own log file by specifying a unique file name for its log. This makes finding and browsing the log for a particular application easier.

The current working directory of the device manager's server is always root. Unless a path is specified, the server log file is created in the root directory.

To enable client logging set the `RASC_DEVMGR_CLIENT_LOG` variable to the name of the file that is to contain the client log entries, as shown below for the Korn shell:

```
export RASC_DEVMGR_CLIENT_LOG=client_log_file_name
```

To disable client logging, unset the RASC_DEVMGR_CLIENT_LOG variable, as shown below for the Korn shell:

```
unset RASC_DEVMGR_CLIENT_LOG
```

To enable server logging set the RASC_DEVMGR_SERVER_LOG variable to the name of the file that is to contain the server log entries, as shown below for the Korn shell:

```
export RASC_DEVMGR_SERVER_LOG=server_log_file_name
```

To disable server logging, unset the RASC_DEVMGR_SERVER_LOG variable, as shown below for the Korn shell:

```
unset RASC_DEVMGR_SERVER_LOG
```

Using the GNU Project Debugger (GDB)

This document describes extensions to the GNU Debugger (GDB) command set to handle debugging of one or more FPGAs. Normal GDB commands and facilities are unchanged.

Brackets [ ] are used in this chapter to indicate that the value is optional. The [ ] here is not something a GDB user types; it is a syntactic convention used in this document to express an optional command field.

We use N here to indicate a number such as 0 or 1. The FPGA number is assigned by GDB when the FPGA is opened (loaded into the GDB command session). The first FPGA opened is assigned number zero, the next is assigned one, and so on. Numbers are not reused. GDB interacts with the RASC abstraction layer to implement several of the GDB features documented below. The N FPGA number is not the same as the 'cop_desc' integer assigned in a user's application (that cop_desc integer is printed in the 'info fpga' output).

Some FPGA data is treated as if it were in registers. Such things use GDB normal register syntax, so one types a prefix $ (and GDB shows a prefix $) to the data name. Other FPGA data is treated variables or arrays and normal c-like syntax is used.

GDB Commands

The commands added to GDB unique to RASC are, as follows:

- `fpgaactive [on/off]`
- `set fpga fpganum = N`
- `info fpgaregisters [regname]`
- `info fr (alias for 'info fpgregisters')`
- `info fpga`
- `fpgastep`
- `fpgacont`
- `fpgatrace [on/off]`

Examples of standard commands and expressions with special meanings for FPGAs are, as follows:

```
print $a_0
print a_in[12]
```

where the RASC configuration has specified `a_0` as being in the Debug Port Interfaceof the Algorithm Block, and `a_in` is an array in an SRAM (possibly N buffered).

These are FPGA-specific and visible only when actively stepping an FPGA. For more information, see "Adding Extractor Directives to the Source Code" on page 106.

fpgaactive

The syntax for the `fpgaactive` command is, as follows:

```
fpgaactive [on,off]
```

With no operands, shows the activity-state (on or off) of the current FPGA. Defaults to `off`, meaning FPGA is inactive and registers are invisible. The RASC Abstraction layer normally maintains the off/on state when you load GDB FPGA with an executable that allocates an FPGA. The state can be set on or off manually (in GDB, not the device) with an optional operand of `on` or `off`, however, doing so is not useful, and could break an application.

set fpga fpganum

The syntax for the `fpganum` command is, as follows:

```
set fpga fpganum = N
```

This sets the `current fpga number`. This value provides an identity so when there are multiple FPGAs one knows which one is being acted/reported on. The `N` is the FPGA number assigned by GDB, not the `cop_desc` integer set in the application executable. The FPGA number is not 'recycled' in a GDB session.

fpgaregisters

The syntax for the `fpgaregisters` command is, as follows:

```
info fpgaregisters [regname]
```

Prints the list of register names and values for the current FPGA. If `regname` is given prints just that register name and value. It ignores normal processor registers.

info fpga

The syntax for the `info fpga` command is, as follows:

```
info fpga [full] [N]
```

This reports information about the FPGA set in use. That is, it displays overview information about each FPGA. The `full` option expands the information shown. The `N` option restricts output to a specfic FPGA (N being the GDB-assigned FPGA number).

fpgastep

The syntax for the `fpgastep` command is, as follows:

```
fpgastep [N]
```

This command does a single step by default. The N option tells GDB to step the FPGA `N` steps. Because hardware does the `N` steps, `N` steps go as fast as a single step.

fpgacont

The syntax for the `fpgacont` command is, as follows:

```
fpgacont
```

Sets the current FPGA running to completion.

fpgatrace

The syntax for the `fpgatrace` command is, as follows:

```
fpgatrace [on/off
```

You can use the `fpgatrace` to dynamically report FPGA use.

Turn `fpga start/stop` tracking (reporting) on or off. With no operands this reports the current state of tracking. With tracking on, each start/stop of an FPGA prints a short line giving the FPGA configuration file name, GDB number, and the ID assigned by the application. It has not escaped our attention that the mechanism could be elaborated to provide detailed information live or to a postprocessing tool to aid in understanding when multiple FPGAs are in use or the FPGA use is otherwise complicated.

Registers

FPGA values that have been mapped to debug ports and labelled by extractor directives are visible as if in the processor register set. If the user space configuration file says `a` is a register (using a `REG_OUT` extractor directive) then (when the FPGA is active) the command '`print $a`' will print that register.

The requirements for this operation are that the register name is unique per FPGA register and across the entire set of register names in the processor. The FPGA registers are an extended set appended to the processor registers. It is important to note that these registers are only visible when the FPGA is active. This is a normal GDB construct with an FPGA aware extension and it adheres to all typical register syntax.

It may be meaningful at times to change the value of such a 'register' and the normal `GDB` syntax of '`set $a = 2`' has that effect.

Values and Stepping

When an FPGA is running one may want to see the internal values if one is stepping the FPGA. Internal values that are exposed depends on the following:

- FPGA programming
- What the FPGA version of core-services supports
- What the device can do

- Linux device driver support

As with normal language programming, there is 'stepping'as known at the hardware level, and 'stepping' at a higher level. This is particularly relevant for a device like an FPGA because whether a given value is even 'current' or 'valid' depends on details of the FPGA state. The device program exposes 'step points' that are meaningful at a software level. Programming to the interface ensures values are meaningful at those step points, and those step points are what is iterated `fpgastep`. Some device programs may not expose such logical-step points and for those device programs values printed via GDB will be more difficult to interpret sensibly (since it is not possible to print the entire FPGA internal state).

FPGA Run Status

When the GDB break command is used on an application executable that is enabled with RASC API calls, `rasclib_brkpt_start`() and `rasclib_brkpt_done`() functions, the GDB inserts break points when an FPGA is started or done, respectively. An alternative method of determining FPGA run status is the `fpgatrace on` command that reports details of FPGA starts and stops with more detail than the `break` command.

Changes To GDB Commands

The gdb `cont` command has a new side effect in `gdbfpga` of implying `fpgacont` on all FPGAs on which you can apply the `fpgastep` or `fpgacont` commands. The side effect makes `cont` get the entire application running as you would expect no matter what state FPGAs are in.

RASC Examples and Tutorials

The chapter contains Reconfigurable Application-Specific Computing Software (RASC) examples and tutorials and covers these topics:

System Requirements

For design, synthesis and bitstream generation you need the following

- PC with 1 GHz or greater clock speed
- At least 512 MB random access memory (RAM)
- Red Hat Linux Enterprise version 3.0
- Xilinx ISE development tools (version 8.1.02i or higher)
- Optional: High-level language compiler.
- Optional: 3rd party FPGA synthesis software supporting Xilinx FPGAs.
- Optional: 3rd party HDL simulation software:

For bitstream download, algorithm acceleration, and real-time verification you need the following:

- One Altix system
- One or more RASC bricks or blades
- SGI ProPack 4 for Linux Service Pack 3

- RASC software module
- A network connection to the PC detailed earlier

Prerequisites

The information and tutorials in this Examples and Tutorials section of the User Guide assume that you have previously installed and familiarized yourself with the Xilinx ISE tools and all optional software. It is also assumed that you have read the Chapter 5, "RASC Algorithm FPGA Implementation Guide" and Chapter 6, "Running and Debugging Your Application" and that you have some experience with Verilog and/or VHDL.

Additional background information (not from SGI) is available in the following:

- *Xilinx ISE Software Manuals and Help*

  http://toolbox.xilinx.com/docsan/xilinx6/books/manuals.pdf
- *Synplicity's Synplify Pro User Guide and Tutorial*

  http://www.synplicity.com/literature/pdf/synpro_ug_1001.pdf

Tutorial Overview

The following tutorials illustrate the implementation details of the algorithm programming interface using two different algorithms. During the following sections you will learn how to integrate algorithms into the RASC brick or RASC blade that are written in hardware description languages (HDLs). You will also see a subset of the optimizations that can be made for RASC implementations.

For both algorithms we will step through the entire RASC design flow: integrating the algorithm with core service; simulating behavior on the algorithm interfaces; synthesizing the algorithm code; generating a bitstream; transferring that bitstream and metadata to the Altix platform; executing an application; and using GDB to debug an application on the Altix system and FPGA simultaneously.

These tutorials only illustrate a subset of the options available for implementing an algorithm on RASC. For more details, see Chapter 3, "RASC Algorithm FPGA Hardware Design Guide" and Chapter 5, "RASC Algorithm FPGA Implementation Guide".

Simple Algorithm Tutorial

Overview

The first algorithm we will use to describe the interfaces and various programming templates for RASC is (d = a & b | c). This simple algorithm allows you to compare coding options and analyze optimization techniques. This section steps through integrating an algorithm written in Verilog and VHDL. Then it will demonstrate simulation, synthesis, bitstream generation, platform transfer, running and debugging the application. These steps are the same for this algorithm regardless of the coding technique used.

Figure 7-1 contains a diagram of the algorithm and its memory patterns.

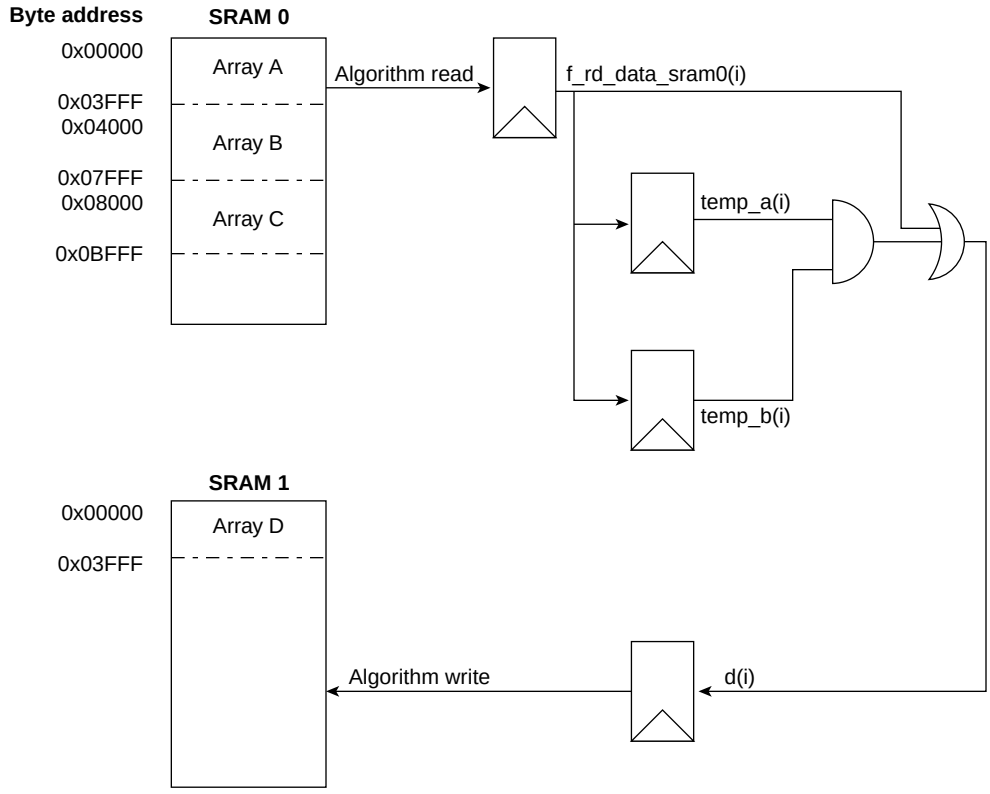

Figure 7-1 Simple Algorithm for Verilog

Application

The application that runs (d = a & b | c) on the Altix platform is fairly simple. The following code demonstrates the RASC Abstraction Layer calls that are required to utilize the RASC-brick as a Co-Processing (COP) unit. The application also runs the algorithm on the Altix box to verify its results. The application C code is on your Altix system at the following location:

```
/usr/share/rasc/examples/alg6.c
```

This simple application runs quickly on an Altix machine, although it is not optimized C code. Please note that this application is not chosen to emphasize the optimizations

available from acceleration in hardware, but rather to compare and contrast the various programming methods for RASC. As you work through the tutorials for the different languages, there will be similarities and differences that highlight advantages of one methodology versus another. For a more computationally intensive example, please see the Data Flow Algorithm in VHDL and Verilog later in this chapter.

Coding Techniques: Verilog

Overview

First we will analyze how to write a Verilog version of (d = a & b | c) for RASC. It is important to note that the source code for this example allows for multi-buffering.

Integrating with Core Services

Begin by loading the hardware description file for the Verilog algorithm, `alg_block_top.v`, into your text editor. Change directory to `$RASC/examples/alg_simple_v` and select `alg_block_top.v`.

This file is the top level module for the algorithm. The other file that is required for this and all other implementation is the file alg.h. This Verilog version of (d= a & b | c) reads a from the first address in SRAM 0, b from the 16384th address of SRAM 0, c from the 32768th address in SRAM 0, and then it writes out the resulting value, d, to the first address of SRAM 1. Arrays a, b, c, and d are 2048 elements long where each element is a 64-bit unsigned integer, and all the arrays are enabled for multi-buffering by the RASC Abstraction Layer. The version of the algorithm, read data, write data, read address, write address, and control signals are all brought out to debug mux registers.

Figure 7-2 contains a diagram of the algorithm and its memory access patterns.

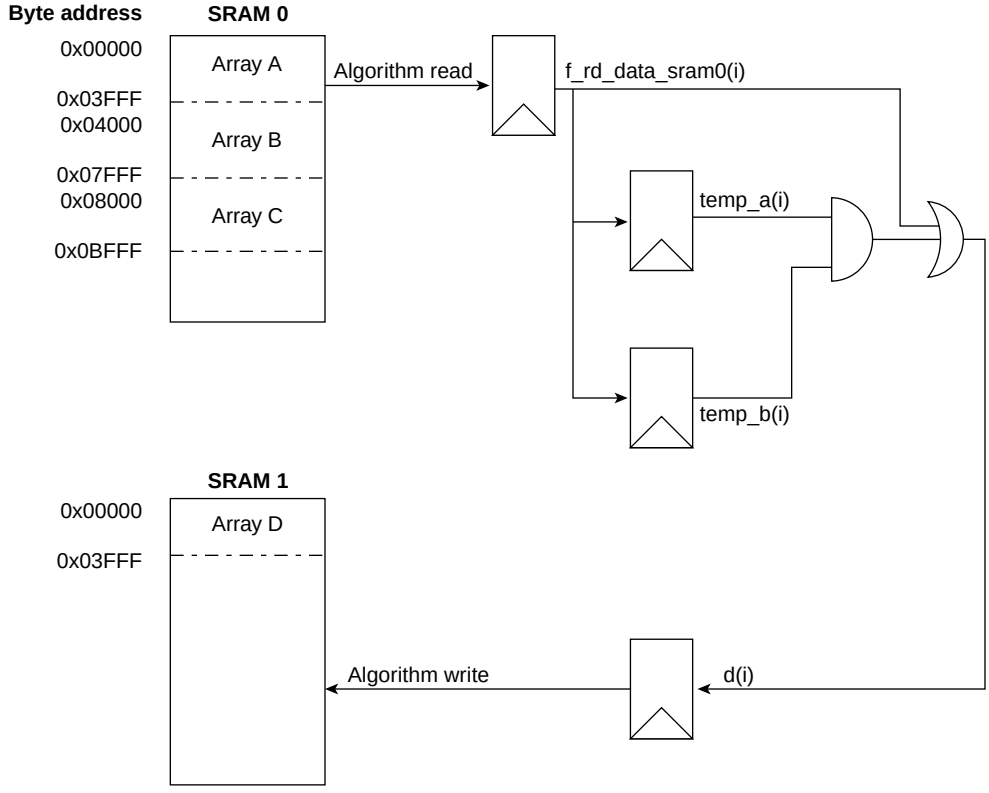

Figure 7-2 Simple Algorithm for Verilog andVHDL

The source code is available on your Altix system at the following location:

`$RASC/examples/alg_simple_v/alg_block_top.v`

Extractor Comments

Other important source code considerations include adding the extractor comments that are required for accurate data movement and debugger control by RASClib. A python script called extractor parses all the Verilog, VHDL, and header files in your algorithm directory to generate the symbol tables required by GDB and to communicate to the abstraction layer the data that should be written and read from the sram banks.

Comment fields to generate the configuration files for the algorithm are inserted in these examples. There is a template in the `alg_core` directory, and several examples. The comment fields can be located in any file in or below the directory specified as the second argument to the extractor call (see Chapter 5, "RASC Algorithm FPGA Implementation Guide" for more detail on how to specify the makefile target). The fields are core services version, algorithm version, sram denoting where data will be read from and written to on the sram interface, register in for parameters set through an application writer's code, or register out for a section of code that needs to be mapped to a debug register.

The debug comments for metadata parsing in this file are:

```
// extractor VERSION: 6.3
// extractor CS: 1.0
// extractor SRAM:a_in 2048 64 sram[0] 0x0000 in u stream
// extractor SRAM:b_in 2048 64 sram[0] 0x4000 in u stream
// extractor SRAM:c_in 2048 64 sram[0] 0x8000 in u stream
// extractor SRAM:d_out 2048 64 sram[1] 0x0000 out u stream
// extractor REG_IN:op_length1 10 u alg_def_reg[0][9:0]
// extractor REG_OUT:alg_id 32 u debug_port[0][63:32]
// extractor REG_OUT:alg_rev 32 u debug_port[0][31:0]
// extractor REG_OUT:rd_addr 64 u debug_port[1]
// extractor REG_OUT:rd_data_sram0_lo 64 u debug_port[2]
// extractor REG_OUT:rd_data_sram0_hi 64 u debug_port[3]
// extractor REG_OUT:wr_addr 64 u debug_port[4]
// extractor REG_OUT:wr_data_sram1_lo 64 u debug_port[5]
// extractor REG_OUT:wr_data_sram1_hi 64 u debug_port[6]
// extractor REG_OUT:cntl_sigs 64 u debug_port[7]
```

These comments are located within `alg_block_top.v` in this case, but they can be anywhere within the algorithm hierarchy as a header or source file. The core services tag helps describe which version of core services was used in generating a bitstream, this is useful with debugging. The version tag allows the user to understand from their GDB session which algorithm and revision he or she has loaded. The register out tag specifies registers that are pulled out to the debug mux. The sram tag is to describe arrays that are written to or read from the sram banks by the algorithm. For more information, see Chapter 5, "RASC Algorithm FPGA Implementation Guide".

Coding Techniques: VHDL Algorithm

Overview

Now we will analyze how to write a VHDL version of (d = a & b | c) for RASC. This source code also allows for multi-buffering.

Figure 7-3 contains a diagram of the algorithm and its memory patterns.

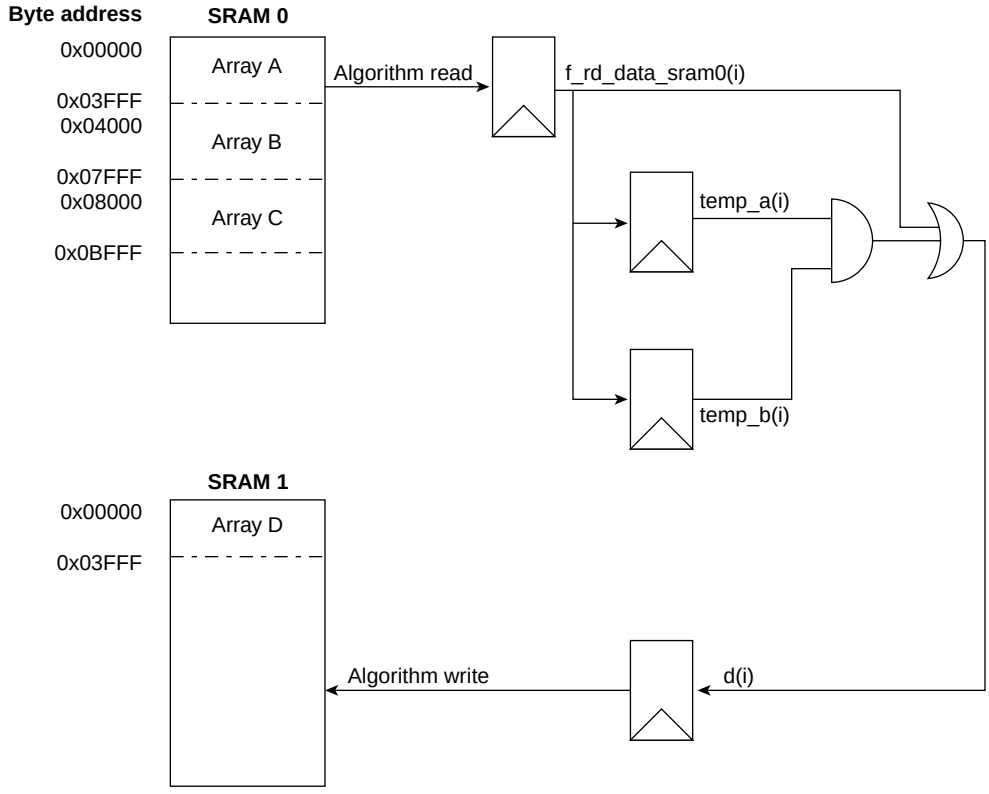

Figure 7-3 Simple Algorithm for Verilog and VHDL

Integrating with Core Services

Begin by loading the hardware description file for the VHDL algorithm, `alg_block_top.vhd`, into your text editor. Change directory to `$RASC/examples/alg_simple_vhd` where you will see `alg_block_top.v and alg_block.vhd.`

These files are the top level module and computation block, respectively. This VHDL version of (d = a & b | c) reads a from the first address in SRAM 0, b from the 16384th address of SRAM 0, c from the 32768th address in SRAM 0, and then it writes out the resulting value, d, to the first address of SRAM 1. Arrays a, b, c, and d are 2048 elements long where each element is a 64-bit unsigned integer, and all the arrays are enabled for

multi-buffering by the RASC Abstraction Layer. The version of the algorithm, read data, write data, read address, and write address for the algorithm are all brought out to debug mux registers.

The source for the `alg_block_top.v` file is similar in this instance to the verilog version of (d = a & b | c), except that it performs no calculation. Instead, it wraps the `alg_block.vhd` so it can speak to the `user_space_wrapper.v` that instantiates it. There is no reason why a computation block needs to be wrapped in a Verilog module. In this case it was done for convenience.

Extractor Comments

Other important source code considerations include adding the extractor comments that are required for accurate data movement and debugger control. A python script called extractor parses all the Verilog, VHDL, and header files in your algorithm directory to generate the symbol tables required by GDB and to communicate to the abstraction layer the data that should be written and read from the sram banks.

Comment fields to generate the configuration files for the algorithm are provided in this example for `alg_block_top.v`. There is a template in the `alg_core` directory, and several examples. The comment fields that can be located anywhere below the directory specified in the second argument to the extractor call (see Chapter 5, "RASC Algorithm FPGA Implementation Guide" for more detail on how to specify the makefile target). The fields are core services version, algorithm version, sram denoting where data will be read from and written to on the sram interface, register in for parameters set through an application writer's code, or register out for a section of code that needs to be mapped to a debug register.

The debug comments for metadata parsing in this file are embedded in the VHDL code. They appear as:

```
-- extractor VERSION: 9.1
-- extractor CS: 0.7
-- extractor SRAM:a_in  2048 64 sram[0] 0x0000 in  u stream
-- extractor SRAM:b_in  2048 64 sram[0] 0x4000 in  u stream
-- extractor SRAM:c_in  2048 64 sram[0] 0x8000 in  u stream
-- extractor SRAM:d_out 2048 64 sram[1] 0x0000 out u stream
-- extractor REG_OUT:version 64 u debug_port[0]
-- extractor REG_OUT:rd_addr 64 u debug_port[1]
-- extractor REG_OUT:rd_data_sram0_lo 64 u debug_port[2]
-- extractor REG_OUT:rd_data_sram0_hi 64 u debug_port[3]
-- extractor REG_OUT:wr_addr 64 u debug_port[4]
-- extractor REG_OUT:wr_data_sram1_lo 64 u debug_port[5]
-- extractor REG_OUT:wr_data_sram1_hi 64 u debug_port[6]
```

These comments are located within `alg_simple.vhd` in this case, but they can be anywhere within the algorithm hierarchy as a header or source file. The core services tag helps describe what version of core services was used in generating a bitstream. This information is useful when debugging. The version tag allows the user to understand from his GDB session which algorithm and revision he has loaded. The register out tag specifies registers that are mapped to the debug mux. The sram tag is to describe arrays that are written to or read from the sram banks by the algorithm.

Compiling for Simulation

To build the Generic SSP stub test bench for the Verilog version of the simple algorithm, change to the `$RASC/dv/sample_tb` directory. At the prompt, enter the following:

```
% make ALG=alg_simple_v
```

If you do not enter the `ALG` tag, the makefile will default to compiling for `alg_simple_v`.

To run the diagnostic `alg_simple_v`, at the prompt, enter the following:

```
% make run DIAG=diags/alg_simple_v ALG=alg_simple_v
```

As with the case in building the test bench, the ALG tag default is `alg_simple` and it needs to be overwritten for the data flow algorithm. The `SRAM0_IN`, `SRAM1_IN`, `SRAM2_IN`, and `SRAM3_IN` defaults are `sram0_input_file, sram1_input_file, sram2_input_file and sram3_input_file`, respectively. The `SRAM0_OUT`, `SRAM1_OUT`, `SRAM2_OUT` and `SRAM3_OUT` defaults are `sram0_output_file`, `sram1_output_file`, `sram2_output_file`, `and sram3_output_file` respectively. These can all be overwritten on the command line.

By specifying the SRAM input files, the user can skip the DMA process for the purposes of testing the algorithm, providing a fast check for the algorithm without verifying the DMA engines of core services.

You can also use a simple C program called `check_alg_simple.c` to verify the test results; build and run it to analyze the initial and final SRAM simulation contents.

Running a diag through this test bench produces results in four formats:

- *vcdplus.vpd* - this file contains the simulation results for the run.

- *terminal output* - the status of the test is output to the screen as the it runs, notifying the user of packets sent/received.
- *log file* - the output to the screen is also stored in the logfile:<*diag_name*>.<*alg_name*>.run.log (for example, dma_alg_simple_v.alg_simple_v.run.log)
- *sram output files* - at the end of simulation (when the diag finishes because it has been successful, an incorrect packet has been received, or time-out has occurred), the contents of the 2 SRAMs are dumped to .dat files (the defaults or user-specified files).

Building an Implementation

When the algorithm has been integrated and verified, it is time to build an implementation.

Change directories to $RASC/implementations/alg_simple_*/

To synthesize the design type make synplify, or make amplify. This is set up utilize the black-boxed version of core services and should synthesize faster.

To generate the required metadata information for the abstraction layer and the debugger, you need to run the extractor script on your file. The physical design makefile includes a make extractor target for this purpose. When it is executed, it will generate two configuration files--one describing core services, and one describing the algorithm behavior.

To execute the ISE foundation tools and run the extractor script on the file type make all. This will take approximate one to two hours due to the complex mapping and place and route algorithms executed by the ISE tools. Please note that the details of setting up your own project are described in Chapter 5, “RASC Algorithm FPGA Implementation Guide”.

Transferring to the Altix Platform

To transfer to the Altix platform, you must add your RASC design implementation into the Device Manager Registry. This transfer must occur regardless of the algorithm generation method.

1. Use FTP to move the algorithm files from the PC to the /usr/share/rasc/bitstreams directory on the Altix machine:

   `$RASC/implementations/alg_simple_*/rev_1/acs_top_ssmap.bin`

   `$RASC/implementations/alg_simple_*/<`*core_services*`>.cfg`

   `$RASC/implementations/alg_simple_*/<`*user_space*`>.cfg`

2. Log into the Altix machine and execute the Device Manager user command `devmgr`

   `devmgr -a -n alg_simple_v -b acs_top_ssmap.bin -c <`*user_space***`>.cfg -s <`***core_services***`>.cfg`**

The script will default the bitstream and configuration files to these names, although the device manager can add files of any name to the registry, so users should feel free to rename project files as convenient.

Verification using GDB

To run a debug session on this bitstream, you must start the application from a GDB session window. GDB is enabled with all versions of this algorithm. To run an application using RASClib, you must execute the extended GDB on the application detailed at the beginning of this example.

```
% gdbfpga /usr/share/rasc/examples/alg6
(gdb) break rasclib_brkpt_start
Make breakpoint pending on future shared library load? (y or [n]) y

Breakpoint 1 (rasclib_brkpt_start) pending.
(gdb) handle SIGUSR1 nostop pass noprint
(gdb) run
Starting program: /usr/share/rasc/examples/alg6
Breakpoint 2 at 0x2000000000063920: file rasclib_debug.c, line 87.
Pending breakpoint "rasclib_brkpt_start" resolved
rasclib_cop_alloc is a deprecated function.  Use rasclib_resource_alloc() instead
client_task_init: can not find TCP service rasc_devmgr in /etc/services -
                  using default server port number 9999
Breakpoint 2, rasclib_brkpt_start (cop_desc=0x0) at rasclib_debug.c:87
87      rasclib_debug.c: No such file or directory.
        in rasclib_debug.c
(gdb) info fpga
fpga 0
  Active       : on
  Singlestep   : off
  Algorithm id : alg6
  Core svc ver : 0.800000
  Algorithm ver: 6.300000
  Algorithm src: v
  Alg. dev     :
  Alg. config  : /var/rasc/rasc_registry/alg6/bitstream.cfg
  CS version   : 0.800000
  CS config    : /var/rasc/rasc_registry/alg6/core_services.cfg
```

```
  prev step ct : 0
  step ct      : 0
(gdb) info fpgaregisters
alg_id         0x6      0x6
alg_rev 0x3     0x3
rd_addr        0x0      0x0
rd_data_sram0  0x0      0x0
rd_data_sram1  0x0      0x0
wr_addr        0x0      0x0
wr_data_sram2  0x0      0x0
cntl_sigs      0x0      0x0
dummy_param0_out 0x0    0x0
dummy_param1_out 0x1111 0x1111
dummy_param2_out 0x2222 0x2222
dummy_param3_out 0x3333 0x3333
op_length1     0x7ff    0x7ff
dummy_param0_in0x0      0x0
dummy_param1_in0x1111   0x1111
dummy_param2_in0x2222   0x2222
dummy_param3_in0x3333   0x3333
(gdb) fpgastep 55
(gdb) info fpgaregisters
alg_id         0x6      0x6
alg_rev        0x3      0x3
rd_addr        0x19     0x19
rd_data_sram0  0x7574737271706f6e       0x7574737271706f6e
rd_data_sram1  0x767574737271706f       0x767574737271706f
wr_addr        0xd      0xd
wr_data_sram2  0x6f6c6f6a6b686f66       0x6f6c6f6a6b686f66
cntl_sigs      0x22     0x22
dummy_param0_out 0x0    0x0
dummy_param1_out 0x1111 0x1111
dummy_param2_out 0x2222 0x2222
dummy_param3_out 0x3333 0x3333
op_length1     0x7ff    0x7ff
dummy_param0_in0x0      0x0
dummy_param1_in0x1111   0x1111
dummy_param2_in0x2222   0x2222
dummy_param3_in0x3333   0x3333
(gdb) fpgastep 3
(gdb) print $rd_data_sram0
$1 = 0x8786858483828180
(gdb) print a_in[8]
$2 = 0x4746454443424140
(gdb) fpgastep 6
(gdb) print $a_in0
$3 = 0x9f9e9d9c9b9a9998
(gdb) print a_in[14]
$4 = 0x7776757473727170
(gdb) delete
Delete all breakpoints? (y or n) y
(gdb) fpgacont
(gdb) cont
Continuing.
success

Program exited with code 01.
```

Many other commands are available. For more information on these commands, see “Using the GNU Project Debugger (GDB)” on page 128.

Data Flow Algorithm Tutorial

This example algorithm illustrates the optimization considerations of multi-buffering a complex algorithm on the RASC platform. This section steps you through the design process for this algorithm with source code in Verilog that steps by clocks.

Application

The application for the Data Flow algorithm is slightly more complex. This example creates a 16 KB array, sorts it from most-significant byte to least-significant byte, runs a string search on the sorted data against a match tag, and then performs a pop count. The location of application code to perform this operation on both the Altix system and the FPGA is provided below. The results are compared to verify the algorithm implementation. The application C code is on your Altix system at the following location:

```
/usr/share/rasc/examples/alg10.c
```

Loading the Tutorial

Begin by loading the hardware description files into your text editor. Change directory to `$RASC/examples/alg_data_flow_v/`

and you will see several files:

```
alg_block_top.v, sort_byte.v, string_search.v and pop_cnt.v.
```

If you look through the files you will see that he data flow algorithm reads 16K bytes of data from SRAM 0. Then it sorts the bytes of each double-word of the input data from most significant to least significant byte order. The algorithm writes those results out to SRAM 1, and then it performs a string search on the sorted data with a 16-bit match string that is provided by the application writer. The match tags resulting from the string search are written out to SRAM 1 and a population count is then run on the data. The resulting population count is written to debug register 1.

Figure 7-4 contains a diagram of the major computational blocks and the memory access patterns for the data flow algorithm.

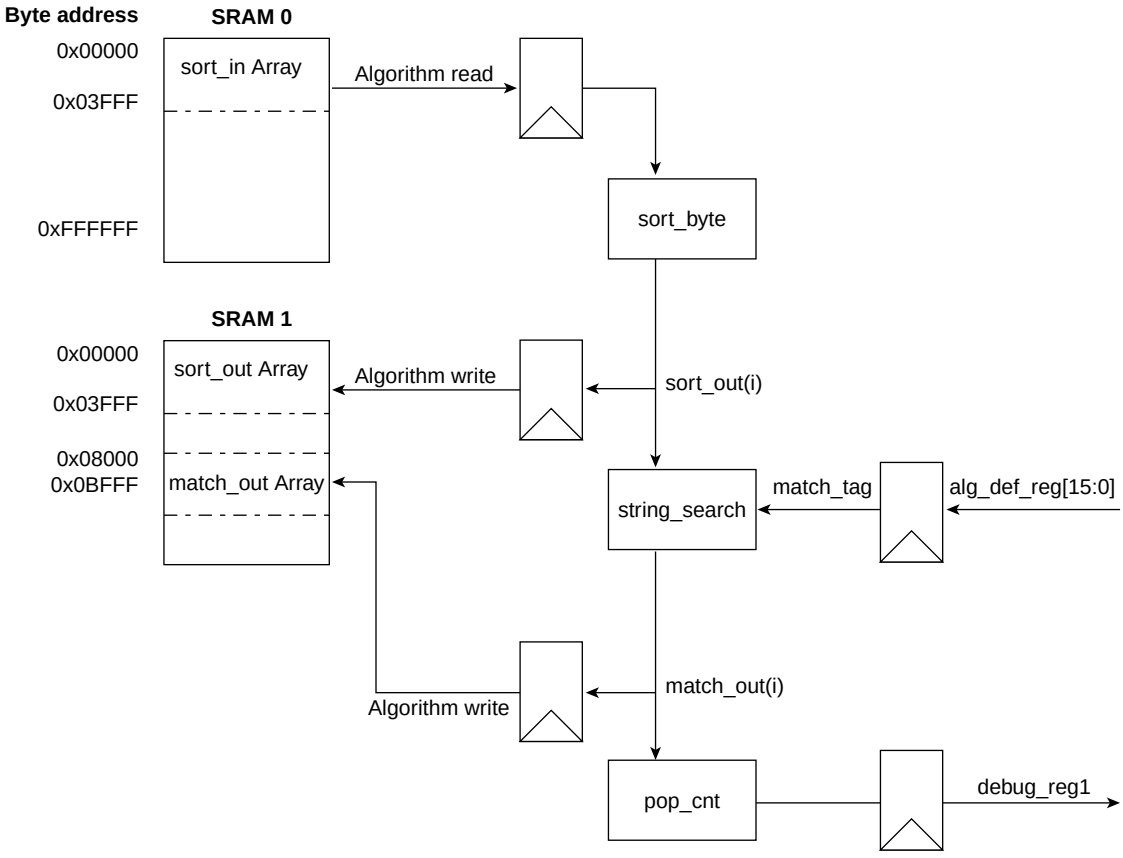

Figure 7-4 Data Flow Algorithm

Integrating with Core Services

Extractor Comments

Extractor comments are inserted in the hierarchy to describe the algorithm. In the `alg_block_top.v` file for the data flow algorithm the following comments exist:

```
// extractor CS:1.0
// extractor VERSION:10.1
// extractor SRAM:data_in               2048 64 sram[0] 0x0000  in u stream
// extractor SRAM:sort_output           2048 64 sram[1] 0x0000 out u stream
// extractor SRAM:bitsearch_match_vector 2048 64 sram[2] 0x8000 out u stream
// extractor REG_IN:match_string 16 u alg_def_reg[0][15:0]
// extractor REG_IN:multi_iter_rst 64 u alg_def_reg[2]
// extractor REG_IN:op_length1 11 u alg_def_reg[1][10:0]
// extractor REG_OUT:version 64 u debug_port[0]
// extractor REG_OUT:running_pop_count 64 u debug_port[1]
// extractor REG_OUT:sort_input 64 u debug_port[2]
// extractor REG_OUT:sort_output 128 u debug_port[3]
// extractor REG_OUT:match_vector 64 u debug_port[5]
// extractor REG_OUT:pipe_vld 64 u debug_port[6]
// extractor REG_OUT:dw_pcnt 64 u debug_port[8]
// extractor REG_OUT:f_pcnt_63_0 64 u debug_port[9]
// extractor REG_OUT:test_cnt 64 u debug_port[10]
```

The core services tag helps describe what version of core services was used in generating a bitstream which is useful with debugging. The version tag allows the user to understand from his GDB session which algorithm and revision he has loaded. The register out tag specifies registers that are pulled out to the debug mux. The sram tag is to describe arrays that are written to or read from the sram banks by the algorithm.

Compiling for Simulation

To build the Generic SSP stub test bench for the data flow algorithm change to the `$RASC/dv/sample_tb` directory. At the prompt, enter the following:

```
% make ALG=alg_data_flow_v
```

If you do not enter the `ALG` tag, the makefile will default to compiling for `alg_simple_v`.

To run the diagnostic `alg_data_flow_v`, at the prompt, enter the following:

```
% make run DIAG=diags/alg_data_flow_v ALG=alg_data_flow_v
```

As with the case in building the test bench, the ALG tag default is alg_simple and it needs to be overwritten for the data flow algorithm. The `SRAM0_IN`, `SRAM1_IN`, `SRAM2_IN`,

and SRAM3_IN defaults are `sram0_input_file, sram1_input_file, sram2_input_file and sram3_input_file`, respectively. The SRAM0_OUT, SRAM1_OUT, SRAM2_OUT and SRAM3_OUT defaults are `sram0_output_file`, `sram1_output_file`, `sram2_output_file`, `and sram3_output_file` respectively. These can all be overwritten on the command line.

By specifying the SRAM input files, the user can skip the DMA process for the purposes of testing the algorithm, providing a fast check for the algorithm without verifying the DMA engines of core services. You can also use a simple C program called `check_alg_data_flow.c` to verify the test results.

Running a diag through this test bench produces results in four formats:

- *`vcdplus.vpd`* - this file contains the simulation results for the run.
- *terminal output* - the status of the test is output to the screen as the it runs, notifying the user of packets sent/received.
- *log file* - the output to the screen is also stored in the logfile:<*diag_name*>.<*alg_name*>`.run.log` (for example, `dma_alg_data_flow_v.alg_data_flow_v.run.log`)
- *sram output files* - at the end of simulation (when the diag finishes because it has been successful, an incorrect packet has been received, or time-out has occurred), the contents of the 4 SRAMs are dumped to .dat files (the defaults or user-specified files).

Building an Implementation

When the algorithm has been integrated and verified, it is time to build an implementation.

Change directories to `$RASC/implementations/alg_data_flow_v/`

To synthesize the design type make synplify, make amplify, or make xst. This is set up utilize the black-boxed version of core services and should synthesize faster.

To generate the required metadata information for the abstraction layer and the debugger, you need to run the extractor script on your file. The physical design makefile includes a make extractor target for this purpose. When it is executed, it will generate two configuration files--one describing core services, and one describing the algorithm behavior.

To execute the ISE foundation tools and run the extractor script on the file type make all. This will take approximate one to two hours due to the complex mapping and place and route algorithms executed by the ISE tools. Please note that the details of setting up your own project are described in Physical Implementation chapter.

Transferring to the Altix Platform

To transfer to the Altix platform, you must add your RASC design implementation into the Device Manager Registry by performing following steps:

1. Use FTP to move the algorithm files from the PC to the /usr/share/rasc/bitstreams/ directory on the Altix machine:

   `$RASC/implementations/alg_data_flow_v/rev_1/acs_top_ssmap.bin`

   `$RASC/implementations/alg_data_flow_v/<`*core_service*`s>.cfg`

   `$RASC/inplementations/alg_data_flow_v/<`*user_space*`>.cfg`

2. Log into the Altix machine and execute the Device Manager user command `devmgr`

   `devmgr -a -n alg_data_flow_v -b acs_top_ssmap.bin -c <`*user_space***`>.cfg -s <`***core_services***`>.cfg`**

The script will default the bitstream and configuration files to these names, although the device manager can add files of any name to the registry, so users should feel free to rename project files as convenient.

Verification Using GDB

To run a debug session on this bitstream, you must start the application from a GDB session window. To do that, you must execute the extended GDB on the application detailed at the beginning of this example.

```
% gdbfpga /usr/share/rasc/examples/alg10
(gdb) break rasclib_brkpt_start
Function "rasclib_brkpt_start" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y

Breakpoint 1 (rasclib_brkpt_start) pending.
(gdb) handle SIGUSR1 nostop pass noprint
(gdb) run
Starting program: /usr/share/rasc/examples/alg10
Breakpoint 2 at 0x2000000000063920: file rasclib_debug.c, line 87.
Pending breakpoint "rasclib_brkpt_start" resolved
client_task_init: can not find TCP service rasc_devmgr in /etc/services -
                  using default server port number 9999
```

```
Breakpoint 2, rasclib_brkpt_start (cop_desc=0x0) at rasclib_debug.c:87
87      rasclib_debug.c: No such file or directory.
        in rasclib_debug.c
(gdb) info fpga
fpga 0
  Active       : on
  Singlestep   : off
  Algorithm id : alg10
  Core svc ver : 0.800000
  Algorithm ver: 10.400000
  Algorithm src: v
  Alg. dev     :
  Alg. config  : /var/rasc/rasc_registry/alg10/bitstream.cfg
  CS version   : 0.800000
  CS config    : /var/rasc/rasc_registry/alg10/core_services.cfg
  prev step ct : 0
  step ct      : 0
(gdb) info fpgaregisters
version        0xa00000004      0xa00000004
running_pop_count [Thread 2305843009226520784 (LWP 465) exited]
0x0     0x0
sort_input     0x0      0x0
sort_output    0x0      0x0
match_vector   0x0      0x0
wr_data_sram2  0x0      0x0
pipe_vld       0x0      0x0
dw_pcnt        0x0      0x0
f_pcnt_63_0    0x0      0x0
test_cnt       0x0      0x0
match_string   0x2cea   0x2cea
multi_iter_rst 0x0      0x0
op_length1     0x7ff    0x7ff
(gdb) fpgastep 5
(gdb) info fpgaregisters
version        0xa00000004      0xa00000004
running_pop_count 0x0   0x0
sort_input     0x0      0x0
sort_output    0x0      0x0
match_vector   0x0      0x0
wr_data_sram2  0x0      0x0
pipe_vld       0x0      0x0
dw_pcnt        0x0      0x0
f_pcnt_63_0    0x0      0x0
test_cnt       0x500000005      0x500000005
match_string   0x2cea   0x2cea
multi_iter_rst 0x0      0x0
op_length1     0x7ff    0x7ff

(gdb) fpgastep 3
(gdb) print $test_cnt
$1 = 0x800000008
(gdb) print data_in[8]
$2 = 0x67fce141a13ee970
(gdb) fpgastep 6
(gdb) print $test_cnt
$3 = 0xe0000000e
(gdb) print data_in[14]
$4 = 0xbb5cf98961bed875
(gdb) delete
Delete all breakpoints? (y or n) y
(gdb) fpgacont
(gdb) cont
```

```
Continuing.
success sorted
success match_list
popcnts = 10c
HW POP COUNT = 268
SW POP COUNT = 268

Program exited normally.
```

The above commands would execute the application, and then hit the breakpoint inserted by the `rasclib_breakpoint_start` function call. At that stage you would be able to query generic data about the FPGA that is configured in the system. Turning stepping on, you can view internal registers and arrays within the session at different steps. Many other commands are available. For more information, see “GDB Commands” on page 128.

Device Driver

This section describes the Field Programmable Gate Arrays (FPGA) core services device driver and covers the following topics:

FPGA Core Services

All FPGAs connected to a TIO in a RASC-brick will contain a fixed set of services or core logic from SGI that control data movement, function initialization and initiation. These services are spelled out in detail in Chapter 3, "RASC Algorithm FPGA Hardware Design Guide".

The FPGA core services device driver is implemented as a character special device driver. It provides open, close, read, write, ioctl and mmap entry points that allow access to the core services direct memory access (DMA) engines and registers.

Control and Status Registers

Specific knowledge of the core services memory-mapped registers is not required to use the core services device driver. The layout of those registers is provide in Chapter 3, "RASC Algorithm FPGA Hardware Design Guide".

Interrupts

The device driver uses interrupts for DMA and Algorithm completion. The atomic memory operation (AMO) features of the RASC-brick will be implemented in a later revision. All system calls block until complete.

Driver Application Programming Interface (API)

The device driver API implements these system calls (see Table A-1):

Table A-1 Device Driver API System Calls

System Call	Description
`open`()	Opens the character-special file. Only one application may have the file opened at any one time.
`close`()	Closes the character-special file.
`read`()	Reads from the FPGA SDRAM to host memory. Note that RASC-brick has three SDRAM's of 2MB each. The device driver treats them as one 6MB SDRAM. `lseek`() may be used if a given RAM is required.
`write`()	Writes to the FPGA SDRAM to host memory.
`lseek`()	Seeks to a specific location in FPGA SDRAM.
`mmap`()	Maps the core services registers or SDRAM into user space.

The character special files used with the device driver are, as follows:

- `/dev/RASC/acs/<nasid>/gscr`

  Used for memory mapping core services memory mapped registers.

- `/dev/RASC/acs/<nasid>/sram`

  Used for all other system calls, for example. reading, writing, and memory mapping SRAM.

Direct Memory Access (DMA)

The two RASC-brick DMA engines may operate independently and are driven via the read and write system calls to the core services device driver. The read system call drives the write DMA engine and the write system call drives the read DMA engine. Each system call will block until the associated dma operation is complete or an error occurs. Upon successful completion ehe number of bytes transferred is return. Otherwise -1 is returned and `errno` is set.

Function Control

The `ioctl` command `COP_IOCTL_ALGO_START` is used to start the algorithm in the FPGA. Use `COP_IOCTL_ALGO_STEP` instead of `COP_IOCTL_ALGO_START` to start the algorithm but only run a given number of clocks. Successive `COP_IOCTL_ALGO_STEP` `ioctl` calls will move the clock. The `COP_IOCTL_ALGO_CONT` call is used to drive the algorithm to completion after stepping.

Upon successful completion, zero is return. Otherwise -1 is returned and `errno` is set.

Example Use of Device Driver

This section provides an example of using the FPGA core services device driver.

```
#include "sys/types.h"
#include "stdio.h"
#include "stdlib.h"
#include "unistd.h"
#include "string.h"
#include "fcntl.h"
#include "sys/ioctl.h"

#include "acs.h"

/*
 * FPGA algorithm info.
 *
 * (This is a simple test-only algorithm built largely for software
testing.)
*

 *      FPGA algorithm is 'a = a & b | c'.
 *      Max number of each operands is 512 64-bit words.
 *      Operand 'a' input starts at word zero of sram0
 *      Operand 'b' input starts at word zero of sram1
 *      Operand 'c' input starts at word zero of sram2
 *      The results are placed at word zero of sram0 (overwriting
operand a)
 */

typedef unsigned long long uint64_t;
```

```
#define RAM_SIZE (2*1024*1024)          // individual SRAM size
#define SRAM0 0
#define SRAM1 RAM_SIZE
#define SRAM2 (RAM_SIZE*2)


#define OPERAND_COUNT 512

/* 64-bit words to bytes */
#define W2B(wc) (wc*sizeof(uint64_t))

uint64_t operand_a[OPERAND_COUNT];
uint64_t operand_b[OPERAND_COUNT];
uint64_t operand_c[OPERAND_COUNT];
uint64_t hard_results[OPERAND_COUNT];
uint64_t soft_results[OPERAND_COUNT];

/*
 * host processor version of algorithm
 */
void
soft_algo(uint64_t *r, uint64_t *a, uint64_t *b, uint64_t *c, int
count)
{
        int i;

        for (i=0; i < count; i++)
                r[i] = (a[i] & b[i]) | c[i];
}

/*
 * FPGA version of algorithm
 */
int
hard_algo(uint64_t *r, uint64_t *a, uint64_t *b, uint64_t *c, int
count)
{
        int fd;
        int bufsize = W2B(count);
        int n, rv = 0;
        char *path = "/dev/RASC/acs/1/sram";

        /*
         * open core services
         */
```

```
fd = open(path, O_RDWR);
if (fd < 0) {
        perror("open");
        return -1;
}

/*
 * move operand 'a'
 */
lseek(fd, SRAM0, SEEK_SET);
n = write(fd, a, bufsize);
if (n != bufsize) {
        fprintf(stderr, "wrote %d of %d bytes\n", n, bufsize);
        perror("write");
        rv = -2;
        goto exit;
}

/*
 * move operand 'b'
 */
lseek(fd, SRAM1, SEEK_SET);
n = write(fd, b, bufsize);
if (n != bufsize) {
        fprintf(stderr, "wrote %d of %d bytes\n", n, bufsize);
        perror("write");
        rv = -3;
        goto exit;
}

/*
 * move operand 'c'
 */
lseek(fd, SRAM2, SEEK_SET);
n = write(fd, c, bufsize);
if (n != bufsize) {

        fprintf(stderr, "wrote %d of %d bytes\n", n, bufsize);
        perror("write");
        rv = -4;

        goto exit;
}

/*
```

```
         * start the FPGA algorithm and wait for it to complete
         */
        if (ioctl(fd, COP_IOCTL_ALGO_START, 0) < 0) {
                perror("ioctl");
                rv = -5;
                goto exit;
        }

        /*
         * get the results
         */
        lseek(fd, SRAM0, SEEK_SET);
        n = read(fd, r, bufsize);
        if (n != bufsize) {
                fprintf(stderr, "read %d of %d bytes\n", n, bufsize);
                perror("read");
                rv = -6;
                goto exit;
        }

exit:
        close(fd);

        return rv;
}

void
compare_results(uint64_t *hard_results, uint64_t *soft_results, int
count)
{
        int i;
        int misses = 0;

        for (i=0; i < count; i++)
                if (hard_results[i] != soft_results[i]) {
                        if (!misses)
                        printf("miscompare at %d: got: 0x%llx expected:
0x%llx\n",
                                i, hard_results[i], soft_results[i]);
                        misses++;
                }

        printf("%d result miscompares between algorithms\n", misses);
}
```

```
void
operand_set(uint64_t *operand, int operand_count, uint64_t value)
{
        int i;

        for (i=0; i < operand_count; i++)
                *operand++ = value;
}

int
main(int argc, char **argv)

{
        int operand_count, bufsize;
        int n, c;

        uint64_t val_a = 'a';
        uint64_t val_b = 'b';
        uint64_t val_c = 'c';
        int verify = 0;


        operand_count = OPERAND_COUNT;

        while ((c = getopt(argc, argv, "a:b:c:s:v")) != EOF) {
                switch(c) {
                case 'a':
                        val_a = strtoull(optarg, 0, 0);
                        break;
                case 'b':
                        val_b = strtoull(optarg, 0, 0);
                        break;
                case 'c':
                        val_c = strtoull(optarg, 0, 0);
                        break;
                case 's':
                        operand_count = strtoull(optarg, 0, 0);
                        break;
                case 'v':
                        verify++;
                        break;
                }
        }
```

```
        if (operand_count > OPERAND_COUNT || operand_count <= 0) {
                printf("invalid operand_count. ");
                printf("must be: 0 > operand_count <=
OPERAND_COUNT\n");
                return -1;
        }

        /* syscalls & lib functions usually require a byte count */
        bufsize = W2B(operand_count);

        /*
         * initialize input and output buffers
         */
        operand_set(operand_b, operand_count, val_b);
        operand_set(operand_c, operand_count, val_c);
        operand_set(operand_a, operand_count, val_a);
        memset(hard_results, 0x0, bufsize);

        /*
         * optionally generate results on host processor
         */
        if (verify) {
                printf("Starting the software algorithm\n");
                soft_algo(soft_results, operand_a, operand_b,
operand_c,
                                                        operand_count);
        }

        /*
         * run the FPGA algorithm
         */
        printf("Starting the hardware algorithm\n");
        n = hard_algo(hard_results, operand_a, operand_b, operand_c,
                                                        operand_count);
        if (n < 0) {
                printf("hard_algo failed: %d\n", n);
                return -1;
        }

        printf("example results:\n    0x%llx & 0x%llx | 0x%llx =
0x%llx\n",
                operand_a[0], operand_b[0], operand_c[0],
hard_results[0]);

        if (verify) {
```

```
                compare_results(hard_results, soft_results,
operand_count);
        }

        printf("done\n");

        return 0;
}
```

SSP Stub User's Guide

Introduction to SSP Stub

The Scalable System Port (SSP) Stub is a verification tool intended to help simulate and verify an algorithm designed for a RASC Field Programmable Gate Array (FPGA). This tool has been created to assist in initial testing and debugging of an algorithm in simulation prior to loading the algorithm onto the FPGA. Used with VCS/Virsim, the Stub allows the user to simulate sending/receiving SSP packets to/from the RASC FPGA to transfer data, start and stop the algorithm, and check status. This appendix covers the following topics:

Recommended Reading

Related and recommended documents include the following:

- *VCS User Guide*
- *GNU Make Manual* (or equivalent text on the Make utility)

The *VCS User Guide* and *GNU Make Manual* are available through the world-wide web.

Verification Environment and Testbench

Provided with the SSP Stub is a verification environment and sample testbench. This consists of the files which make up the SSP Stub as well as supporting files located in a directory tree. The directory tree, testbench, and sample tests are intended to help you quickly gain proficiency in creating tests for a specific algorithm. This section covers the following topics:

Verification Environment

The SSP Stub is intended for use on Verilog designs and is customized for use with the VCS/Virsim simulator. To insure that SSP Stub and sample testbench function properly, the following environment variables must be set to the correct directories:

- `VCS_HOME (`*your_vcs_install_directory)*
- `VCSPLIDIR $VCS_HOME/gui/virsim/linux/vcdplus/vcs` (*your_vcs_version*)
- `PATH  $PATH\:$VCS_HOME/bin`

Sample Test Bench

The SSP Stub consists of Verilog modules as well as PLI calls to functions written in C code. The stub is instantiated in a sample Verilog testbench along with the RASC FPGA. The C-code and top-level stub modules are located in the *your_root*`/dv/sample_tb` directory. Here, *your_root* is the installation directory of the RASC Core Services design and verification tree. The RASC Core Services modules are located in the design branch of the tree under the *your_root*`/design` directory. The algorithm being tested must be created in the design branch of the tree, located in the directory *your_root*`/design/alg_core/alg_name`. See existing sample algorithms for exact file locations. The sample test bench relies on the algorithm to be in this location for compiling.

The primary Verilog modules in the SSP Stub and other files are listed below. They can be found in the `sample_tb` directory.

- `top.v:` Top level of the Sample Test Bench containing the SSP Stub, Algorithm FPGA, clock generator, and SRAM instances. Note that there are four SRAM instances in top.v while there are only two logical SRAMs. The SRAMs are used in pairs to form a single logical SRAM. This structure mirrors the real hardware.
- `ssp_stub.v:` Top level Verilog of the SSP Stub which passes signals to and from conversion modules.
- `init_sram0.dat`, `init_sram1.dat`, `init_sram2.dat`, `init_sram3.dat:` These SRAM initialization files contain data which is automatically loaded into the respective SRAM simulation models at the beginning of simulation. The data is in a format which the SRAM simulation model uses (one bit of parity per byte of data is shifted in with the data). These default files can be overridden by the user on the command line at runtime.
- `final_sram0.dat, final_sram1.dat, final_sram2.dat, final_sram3.dat:` These files contain data extracted from the respective SRAM simulation models at the end of simulation. These default files can be overridden by the user on the command line at runtime.
- `timescale.v:` This file contains the Verilog timescale of each of the components of the SSP Stub, as well as the algorithm FPGA design files. It is required that the algorithm being simulated makes use of the same timescale as the rest of the design.

For simulation, the Verilog timescale of each of the components of the SSP Stub, as well as the RASC FPGA design files, is set by the `timescale.v` file in the `sample_tb` directory. It is required that the algorithm being simulated makes use of the same timescale as the rest of the design.

SSP Stub File Descriptions

The SSP Stub consists of the following files:

- `ssp_stub.v:` Top level Verilog of the SSP Stub which passes signals to and from conversion modules.
- `cx_stub_ssr.v:` A component of the SSP Stub which converts the raw 64-bit Double Data Rate (DDR) flits from the SSP link into 128-bit Single Data Rate (SDR) flits for use in the stub.
- `ssp_rcv_recap.v:` A component of the SSP Stub which captures data (from cx_stub_ssr.v) on rising edge.
- `cx_stub_ssd.v:` A component of the SSP Stub which converts the 128-bit SDR data from the stub to 64-bit DDR data for transmitting over the SSP link.

- `ssp_sdr_stub.v:` A component of the SSP Stub which passes 128-bit data to and from C-code.
- `tio_shrd_ecc_generate.v:` A component of the SSP Stubh which generates ECC bits in accordance with SSP protocol.
- `pli.tab:` This PLI file links the `$start_ssp` call from the Verilog to the `start_ssp()` routine in C code.
- `start_ssp.c:` This file directs the initialization of the stub. It opens the diag file, parses the commands, sets up the packet queues, initializes counters, and calls `send_rcv_flits`() to enter the main loop.
- `user_const.h:` A header file containing user specified constants (e.g. simulation timeout, maximum size of the input file, maximum number of commands). Additional constants may be added to this file as needed by the user.
- `ssp_defines.h:` This header file contains macros for the program, such as the names of packet types.
- `send_rcv_flits.h:` This header file contains the main loop function, `send_rcv_flits`(), which calls itself once every clock cycle through the `tf_setdelay`(5) PLI call [`tf_setdelay(time`) waits for the specified amount of time, then calls `misctf` with input "reason_reactivate"; in the `pli.tab` file, the misc function is specified to be sent to `"send_rcv_flits"]. send_rcv_flits`() runs through the various conditions for sending, receiving, timing out, printing, delaying, and polling. It uses the function `process_pkt` to get information about error packets (in the syntax used for the input diag file) and the function get_fields to get a breakdown of the command word for logging purposes. It uses the sub-functions `send_flit`(), `rcv_flit`(), `snd_poll`(), and `rcv_poll`() to place data on the outgoing lines and read data off of the incoming lines. It calls `finish_ssp`() to print success and exit. `send_flit`() puts data for the next flit on the outgoing lines. `rcv_flit`() takes data from the incoming lines and stores it after comparing it with expected data. `snd_poll`() sends out a PIO 8-Byte Read Request packet for a given address. `rcv_poll`() takes the read response to a poll request, and checks the bit in question to see if it has been set.
- `queue_pkt.h:` This file contains the mechanism for reading an input command line from the diag and converting it into data values representing the packet that will need to be sent/received OR the information for a `print/delay/poll. queue_pkt(string`) goes to tokenizes the input string and extracts data from each `field.q_string_it(token, pkt_string`) appends the latest token to the stored packet string (for logging purposes). `strtok_checked(s1, s2)` calls

strtok on the inputs and throws an error if the resulting token is NULL when it should not be.

- `setup_pkt.h:` This file contains the function `setup_pkt(snd_rcv)`, which takes care of setting up packets for regular sends and receives (not poll packets), based on the information stored by `queue_pkt(string)`. It uses the functions in `snd_rcv_fns.h` to get the data for each flit, and sets up the remaining fields individually (head, tail, error, `req_dval`, `rsp_dval`).
- `snd_rcv_fns.h:` This file contains the equivalent of the eleven packet functions available to the diag, but with an extra field for the pkt[] array which will contain their results. Each function passes the appropriate type and fields to the `construct_pkt(type, tnum, addr, data, error, pkt, snd_rcv_n)` function.
- `construct_pkt.h:` This file takes the packet information from `snd_rcv_fns.h` and turns it into 64-bit segments of data. construct_pkt(type, tnum, address, data, error, pkt, to_from_n) takes the input information and creates appropriate data, storing this data in pkt. pkt_size(type) takes the type and returns a value for the number of flits in that packet (1, 2, 9, or 10).
- `make_command.h:` This file takes the type, transaction number, error bit, and direction of a packet and constructs an SSP command word, which it returns as an int.
- `get_fields.h:` This file takes input of the type of packet to deconstruct (i.e., send, receive, incoming, poll request, poll response, poll expected) and returns the command word with its decomposed fields as well as any address or data fields. It also contains a helper function, f_string_it(token), which appends information to its destination string (data_fields[]) as it proceeds.
- `process_pkt.h:` This file takes the type of packet (either incoming or poll response) and turns the data from that receive packet into a string following the syntax of the diag file. It has a helper function p_string_it(token) that behaves the same as f_string_it(token), but stores its data in processed_string[].

Compiling and Running a Test

Compiling the sample testbench is done using the provided `Makefile`. In order to compile the sample testbench including the SSP Stub and the RASC Core Services logic, an algorithm must be specified. This algorithm is specified to the `make` utility on the command line, (change directory to the `sample_tb` directory and enter the following command) as follows:

```
% cd your_root/dv/sample_tb
% make ( ALG=your_algorithm )
```

The Makefile uses *your_algorithm* to find the directory in the tree where you algorithm design files exist (for example, `ALG=alg0`). For the case where no `ALG=`*your_algorithm* is specified, the default algorithm is used from *your_root*`/examples/alg_simple_v`. As mentioned ealier, it is required that user generated algorithms are saved in a corresponding directory (for example,. y*our_root*`/examples/my_alg/*.v`).

To run a test, remain in the `sample_tb` directory and enter the following command:

```
% make run DIAG=diag_filename ( ALG=your_algorithm
SRAM0_IN=sram0_input_filename SRAM1_IN=sram1_input_filename
SRAM2_IN=sram2_input_filename SRAM3_IN=sram3_input_filename
SRAM0_OUT=sram0_output_filename SRAM1_OUT=sram1_output_filename
SRAM2_OUT=sram2_output_filename SRAM3_OUT=sram3_output_filename )
```

The *diag_filename* specifies the diag to be run and should be relative to the current directory. Again, the algorithm must be specified using the `ALG=`*your_algorithm* command line option. If none is specified, the runtime command uses same default as above . The ALG option allows the user to reuse the same diag for multiple algorithms. The input and output data files for the three SRAMs may be specified; if they are not specified, they default to:

```
SRAM0_IN=init_sram0_good_parity.dat
SRAM1_IN=init_sram1_good_parity.dat
SRAM2_IN=init_sram2_good_parity.dat
SRAM3_IN=init_sram3_good_parity.dat
SRAM0_OUT=final_sram0.dat
SRAM1_OUT=final_sram1.dat
SRAM2_OUT=final_sram2.dat
SRAM3_OUT=final_sram3.dat
```

As the test runs, it will output status to the screen and to an output file named *diag_filename*`.`*your_algorithm*`.run.log`. This log file will appear in the same directory that the diagnostic is located in. The contents of each SRAM at the end of simulation will be dumped into .dat files which can be either the default filenames or user-specified. Each time a diagnostic is run, the file vcdplus.vpd is generated in the sample_tb directory. This file can be input to Virsim for viewing the waveform. Since the file vcdplus.vpd is generally large, it is overwritten for each diagnostic run. To save the waveform for a given diagnostic, copy corresponding vcdplus.vpd file to a new name.

To view the waveform saved in the vcdplus.vpd file, use the following command:

```
% vcs -RPP vcdplus.vpd
```

When the stub receives an incorrect packet, it will output, in order, the following information: the command for the next expected packet, SSP fields of the expected packet, the command translation (if one exists) for the received packet, and the SSP fields of the received packet.

SSP Stub Commands

The Stub retrieves instructions from a test through a text input file, the diagnostic. The SSP Stub parses this file at each semicolon to extract commands which the Stub executes. Many of the allowed commands in a diagnostic correspond to SSP packets. There are other commands that the SSP Stub supports other commands that are used for diagnostic writing and debugging. The primary components of the diagnostic file are, packet commands, other commands, and comments.

It is important to note that most SSP packets come in pairs, a request and a response. For these types of packets, the request command and response command **must** be listed sequentially in a diagnostic. This method of keeping requests and response paired is used by the stub to associate request and response packets with the corresponding tnums. Also, when running the DMA engines, all transactions related to that sequence of events should be grouped together. See the sample diagnostic included later in this document for an example of how this is done.

Packet Commands

RASC makes use of a subset of the available SSP packet types. The packet types used by the SSP Stub are included in the table below.

Table B-1 Packet Types used by SSP

Type	Transaction	Valid Transactions Stub to FPGA	Valid Transactions FPGA to Stub
Reads	PIO 8-Byte Read	Yes	No
	PIO Full 128-Byte Read	Yes	No

Table B-1 Packet Types used by SSP

	Memory Full 128-Byte Read	No	Yes
Writes	PIO 8-Byte Write	Yes	Yes
	PIO Partial 128-Byte Write	Yes	No
	Memory Full 128-Byte Write	No	Yes
Store AMOs	Store AMO	No	Yes
Invalidate-Cache Flush	Invalidate-Cache Flush	Yes	No

Of the standard SSP packet types listed in the SSP Specification, this stub does **NOT** support FPGA-to-Stub Processor Input/Output (PIO) 8-Byte Reads, FPGA-to-Stub Memory Partial 128-Byte Writes, FPGA-to-Stub PIO Partial 128-Byte Reads or Writes, Fetch atomic memory operations (AMOs), or Graphics Writes. It does, however, support all SSP packets that are used or accepted by the RASC Core Services.

Packet commands begin with the name "`snd_`" or "`rcv_`", contain fields within parentheses separated by commas, and end with a semicolon, for example:

```
snd_wr_req(PIO, DW, ANY, 0x0000000000000000, 0x000000000000FFFF);
```

There are two categories of stub commands: sends and receives. The diagnostic should specify both the requests and responses it wishes to send as well as the requests and responses it expects to receive from the FPGA.

Command Fields

Below is a list of the various fields used by packet commands. A list of allowable values and their corresponding meanings is also included.

- Field: PIO/MEM

  PIO (Processor Input/Output): Use for read/write requests initiated by the stub, also used for interrupt packets from the FPGA.

  MEM (Memory): Use for read/write requests initiated by the FPGA (except interrupts).

- Field: Size

  DW (Double Word): Use for 8-Byte transactions (MMRs and Interrupts). Note that Double Word refers to a 32-bit quantity in this document.

FCL (Full Cache-Line): Use for Full 128-Byte transactions (usually DMA's).

- Field: Tnum

  0x0 - 0xff: Correspond to the tnum of a packet.

  ANY: Used when any tnum is allowable or when responding with the tnum of the corresponding request.

  For outgoing requests, the tnum may be specified (though overlapping tnums for the same type of transactions should not be used per the SSP Specifiation). If "ANY" is specified for an outgoing request, the Stub will assign a tnum automatically. For incoming responses, the tnum may be specified if the tnum of the corresponding request was specified; in all other cases, specifying "ANY" will allow the stub to accept any tnum for the incoming packet.

  Also, note that the RASC FPGA will send out PIO Write Requests or AMO Requests to the stub with tnums always in the range 0xf0 - 0xff. The RASC FPGA should not use tnums in this range for any other purpose.

- Field: Address

  Any 64-bit value: Used to specifiy the intended or expected address of a packet (note that actual addresses are 56-bit values). For all requests, this field specifies the SRAM, MMR, or Memory address location.

- Field: Data

  A list of 64-bit values, dependent on the size specified. Each 64-bit data segment (from least-significant to most-significant) is separated by a comma. For DWs, there is just one data segment. For FCLs, there are 16 segments. The sample diagnostic included later in this document contains examples.

- Field: Error

  0: No error is reported in the header of the SSP packet.

  1: An error is reported/expected in the corresponding packet.

  The error field is required for all responses, whether received or sent. The error bit is one when there has been some error in the transaction; for example, the type of request may not be valid or the address may be invalid.

Send Commands

snd_wr_req

The syntax of the `snd_wr_req` function is, as follows:

```
snd_wr_req(pio_mem_n, size, tnum, addr, data):
```

The `snd_wr_req` function is used to send PIO writes to the memory-mapped registers (MMRs) (8-Byte).

- `pio_mem_n` = PIO
- `size` = DW
- `tnum` = ANY or user-specified
- `addr` = address of MMR or in SRAM
- `data` = data to be written

snd_rd_req

The syntax of the `snd_rd_req` function is, as follows:

```
snd_rd_req(pio_mem_n, size, tnum, addr):
```

The `snd_rd_req` function is used to request PIO reads from the MMRs (8-Byte) and SRAMs (Full 128-Byte).

- `pio_mem_n` = PIO
- `size` = DW or FCL
- `tnum` = ANY or user-specified
- `addr` = address of MMR or in SRAM

snd_wr_rsp

The syntax of the `snd_wr_rsp` function is, as follows:

```
snd_wr_rsp(pio_mem_n, size, tnum, error):
```

The `snd_wr_rsp` function is used to respond to write requests from the FPGA during DMAs (Full 128-Byte) and interrupts (8-Byte).

- `pio_mem_n` = PIO (interrupts) or MEM (DMAs)
- `size` = DW (interrupts) or FCL (DMAs)
- `tnum` = ANY or user-specified

- `error` = 0 or 1

snd_rd_rsp

The syntax of the `snd_rd_rsp` function is, as follows:

```
snd_rd_rsp(pio_mem_n, size, tnum, error, data):
```

The `snd_rd_rsp` function is used to respond to read requests from the FPGA: DMAs (Full 128-Byte).

- `pio_mem_n` = MEM
- `size` = FCL
- `tnum` = ANY or user-specified
- `error` = 0 or 1
- `data` = 0 if this is an error response, otherwise a list of 16 64-bit data values, separated by commas.

snd_amo_rsp

The syntax of the `snd_amo_rsp` function is, as follows:

```
snd_amo_rsp(tnum):
```

- `tnum` = ANY or user-specified

inv_flush

The syntax of the `inv_flush` function is, as follows:

```
inv_flush(tnum):
```

- `tnum` = ANY or user-specified

Receive Commands

rcv_wr_rsp

The syntax of the `rcv_wr_rsp` function is, as follows:

```
rcv_wr_rsp(pio_mem_n, size, tnum, error):
```

The `rcv_wr_rsp` function is used to receive write responses from the FPGA (after writes to MMRs).

- `pio_mem_n` = PIO
- `size` = DW
- `tnum` = ANY or user-specified
- `error` = 0 or 1

rcv_rd_rsp

The syntax of the `rcv_rd_rsp` function is, as follows:

```
rcv_rd_rsp(pio_mem_n, size, tnum, error, data):
```

The `rcv_rd_rsp` function is used to receive read responses from the FPGA (after read requests to the MMRs).

- `pio_mem_n` = PIO
- `size` = DW
- `tnum` = ANY or user-specified
- `error` = 0 or 1
- `data` = 0 if an error response, one 64-bit value otherwise

rcv_wr_req

The syntax of the `rcv_wr_req` function is, as follows:

```
rcv_wr_req(pio_mem_n, size, tnum, addr, data):
```

The `rcv_wr_req` function is used to receive write request from the FPGA (interrupts and DMA writes).

- `pio_mem_n` = PIO (interrupts) or MEM (DMAs)
- `size` = DW (interrupts) or FCL (DMAs)
- `tnum` = ANY or user-specified
- `addr` = address specified for interrupts or system memory address (64-bits)
- `data` = one 64-bit value for interrupts; 16 64-bit values for DMAs

rcv_rd_req

The syntax of the `rcv_rd_req` function is, as follows:

```
rcv_rd_req(pio_mem_n, size, tnum, addr):
```

The rcv_rd_req function is used to receive read requests from the FPGA (DMA reads).

- pio_mem_n = MEM
- size = FCL
- tnum = ANY or user-specified
- addr = system memory address

rcv_amo_req

The syntax of the `rcv_amo_req` function is, as follows:

```
rcv_amo_req(tnum, addr, data):
```

The `rcv_amo_req` function is used to accept Store AMO requests from the FPGA.

- `num` = ANY or user-specified
- `addr` = address specified for AMOs
- `data` = one 64-bit value

Other Commands

`print "`*`text`*`";`

The user can output information from the diagnostic through the print command. These are indicated by the keyword "`print`" followed by the output text enclosed in quotation marks and followed by a semicolon:

```
print "Done initializing registers.\n";
```

The stub will output the characters in between the quotation marks verbatim, with three exceptions: '\n', '\t', and '\v' are treated as "newline," "horizontal tab," and "vertical tab," respectively. The print command will execute immediately after the command above it is executed. The SSP Stub will not wait for a print command to be executed before proceeding with the rest of the diagnostic.

delay(cycles);

The user can specify a specific delay between sending packets. The value specified in the delay command is the number of 5 ns clock cycles to wait before continuing to execute commands in the diagnostic. This can be used to wait for the algorithm to execute steps or finish executing entirely.

poll(*address***,** *bit***,** *interval***);**

The user can specify a bit to read at a specified interval to see if a process is finished (for example, if the algorithm, DMA read engine, or DMA write engine is complete). The stub will read the MMR of *address* on the specified *interval* until the *bit* is set to 1

Note: NOTE: If the interval chosen is smaller than the time necessary to complete the read transaction (approximately 44 clock cycles), the stub will wait until it receives a response to its last request before proceeding with the next read request (that is, the actual time between reads may be greater than the specified interval).

- *address* = address of MMR to be read
- *bit* = bit in register to be checked
- *interval* = number of clock cyles in between register reads

Command Summary

The table below provides a summary of packet commands and their possible usages.

Command	Packet Type	PIO/ MEM	Size	Tnum	Error	Address	Data
snd_wr_req	PIO 8-Byte Write Request	PIO	DW	User-specified or ANY	-	MMR or SRAM Address	1 64-bit value
snd_rd_req	PIO 8-Byte Read Request	PIO	DW	User-specified or ANY	-	MMR Address	-
	Memory Full 128-Byte Read Request	MEM	FCL	User-specified or ANY	-	SRAM Address	-

snd_wr_rsp	PIO 8-Byte Write Response	PIO	DW	User-specified or ANY	0 or 1	-	-
	Memory Full 128-Byte Write Response	MEM	FCL	User-specified or ANY	0 or 1	-	-
snd_rd_rsp	Memory Full 128-Byte Read Response	MEM	FCL	User-specified or ANY	0 or 1	-	16 64-bit values
snd_amo_rsp	Store AMO Response	-	-	User-specified \or ANY	0 or 1	-	-
inv_flush	Invalidate Cache-Flush Request	-	-	User-specified or ANY	-	-	-
rcv_wr_rsp	PIO 8-Byte Write Response	PIO	DW	User-specified or ANY	0 or 1	-	-
rcv_rd_rsp	PIO 8-Byte Read Response	PIO	DW	User-specified or ANY	0 or 1	-	1 64-bit value
rcv_wr_req	PIO 8-Byte Write Request	PIO	DW	User-specified or ANY	-	Interrupt Destination Address	1 64-bit value
	Memory 128-Byte Write Request	MEM	FCL	User-specified or ANY	-	System Memory Address	16 64-bit values
rcv_rd_req	Memory 128-Byte Read Request	MEM	FCL	User-specified or ANY	-	System Memory Address	-
rcv_amo_req	Store AMO Request	-	-	User-specified or ANY	-	AMO Destination Address	1 64-bit value

Comments

Comments are indicated by a '#' sign. The stub will ignore sections of the diagnostic from the '#' indicator to the end of the line (excluding '#'s contained in print statements).

Sample Diagnostic

The code listed below comprises a diagnostic which exercises the basic functionality of the RASC FPGA outlined in the following steps:

1. Initializes the RASC FPGA Core Services (primarily MMR Writes)
2. Executes DMA Reads to send data to the FPGA
3. Starts the Algorithm (A & B | C) and polls the MMR's to see when the Algorithm is done
4. Executes DMA Writes to retrieve the Algorithm's results
5. Checks the error status in the MMR's to verify that no errors were flagged.

```
#######  Initialization packets.  #######
# Arm reigsters by setting the REARM_STAT_REGS bit in the CM_CONTROL register.
snd_wr_req ( PIO, DW, ANY, 0x00000000000020, 0x0000000600f00003 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );

# Clear the CM_ERROR_STATUS register by writing all zeroes.
snd_wr_req ( PIO, DW, 3, 0x00000000000060, 0x0000000000000000 );
rcv_wr_rsp ( PIO, DW, 3, 0);

# Enable CM_ERROR_DETAIL_* regs by writing all zeroes to CM_ERROR_DETAIL_1.
snd_wr_req ( PIO, DW, ANY, 0x00000000000010, 0x0000000000000000 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );

# Enable desired interrupt notification in the CM_ERROR_INTERRUPT_ENABLE
register.
snd_wr_req ( PIO, DW, 4, 0x00000000000070, 0xFFFFFFFFFFFFFFFF );
rcv_wr_rsp ( PIO, DW, 4, 0 );

# Set up the Interrupt Destination Register.
snd_wr_req ( PIO, DW, ANY, 0x00000000000038, 0x0000000000000000 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );

print "\n\n*******Initialization finished\n\n";


#######  Configure DMA Engines and Algorithm.  #######

#######  Configure the Read DMA Engine Registers.  #######
print "\n\n*******Configure Read DMA Engine.  Tell it to fill 32 cache lines of
data.\n\n";

# RD_DMA_CTRL register.
snd_wr_req ( PIO, DW, ANY, 0x00000000000110, 0x0000000000100020 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );

# RD DMA System Address.
snd_wr_req ( PIO, DW, ANY, 0x00000000000100, 0x0000000000100000 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );
```

```
# RD DMA Local Address.
snd_wr_req ( PIO, DW, ANY, 0x00000000000108, 0x0000000000000000 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );

# RD AMO address.
snd_wr_req ( PIO, DW, ANY, 0x00000000000118, 0x0000000000000000 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );

# RD_DMA_DEST_INT
snd_wr_req ( PIO, DW, ANY, 0x00000000000120, 0x0000000200002000 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );


#######  Configure the Write DMA Engine Registers.  #######
print "\n\n*******Configure Write DMA Engine.\n\n";

# Write to the WR_DMA_CTRL register.
snd_wr_req ( PIO, DW, ANY, 0x00000000000210, 0x0000000000100020 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );

# WR_DMA_SYS_ADDR
snd_wr_req ( PIO, DW, ANY, 0x00000000000200, 0x0000000000100000 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );

# WR_DMA_LOC_ADDR
snd_wr_req ( PIO, DW, ANY, 0x00000000000208, 0x0000000000000000 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );

# WR_DMA_AMO_DEST
snd_wr_req ( PIO, DW, ANY, 0x00000000000218, 0x0000000000000000 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );

# WR_DMA_INT_DEST
snd_wr_req ( PIO, DW, ANY, 0x00000000000220, 0x0000000400004000 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );

#######  Configure the Algorithm Registers.  #######
print "\n\n*******Configure Algorithm Registers\n\n";

snd_wr_req ( PIO, DW, ANY, 0x00000000000300, 0x0000000000000000 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );

snd_wr_req ( PIO, DW, ANY, 0x00000000000308, 0x0000000600006000 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );


#######  Start Read DMA Engine for Read DMA 1  #######
print "\n\n*******Start Read DMA Engine for SRAM0\n\n";
```

```
# Set Bit 36 of the CM_CONTROL Reg to 1.
snd_wr_req ( PIO, DW, ANY, 0x00000000000020, 0x0000001400f00003 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );

# 1 of 32
rcv_rd_req ( MEM, FCL, ANY, 0x00000000100000 );
snd_rd_rsp ( MEM, FCL, ANY, 0, 0xDEADBEEFDEADBEEF, 0xDEADBEEFDEADBEEF,
       0xDEADBEEFDEADBEEF, 0xDEADBEEFDEADBEEF,
       0xDEADBEEFDEADBEEF, 0xDEADBEEFDEADBEEF,
       0xDEADBEEFDEADBEEF, 0xDEADBEEFDEADBEEF,
       0xDEADBEEFDEADBEEF, 0xDEADBEEFDEADBEEF,
       0xDEADBEEFDEADBEEF, 0xDEADBEEFDEADBEEF,
       0xDEADBEEFDEADBEEF, 0xDEADBEEFDEADBEEF );

# Other Read DMA Transactions omitted here

# 32 of 32
rcv_rd_req ( MEM, FCL, ANY, 0x00000000100F80 );
snd_rd_rsp ( MEM, FCL, ANY, 0, 0xDEADBEEFDEADBEEF, 0xDEADBEEFDEADBEEF,
       0xDEADBEEFDEADBEEF, 0xDEADBEEFDEADBEEF,
       0xDEADBEEFDEADBEEF, 0xDEADBEEFDEADBEEF,
       0xDEADBEEFDEADBEEF, 0xDEADBEEFDEADBEEF,
       0xDEADBEEFDEADBEEF, 0xDEADBEEFDEADBEEF,
       0xDEADBEEFDEADBEEF, 0xDEADBEEFDEADBEEF,
       0xDEADBEEFDEADBEEF, 0xDEADBEEFDEADBEEF );

print "\n\n*******Done storing 32 cache lines of data in SRAM0.\n\n";

print "\n\n*******Polling for DMA RD-SRAM0 done (bit 42 of CM_STATUS).\n\n";
poll (0x8, 42, 20);


#######  Reconfigure DMA Engine for Read DMA 2  #######

# RD_DMA_SYS_ADDR
snd_wr_req ( PIO, DW, ANY, 0x00000000000100, 0x0000000000100000 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );

# RD_DMA_LOC_ADDR
snd_wr_req ( PIO, DW, ANY, 0x00000000000108, 0x0000000000200000 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );


#######  Start Read DMA Engine for Read DMA 2  #######
print "\n\n*******Start Read DMA Engine for SRAM1\n\n";
```

```
# Set Bit 36 of the CM_CONTROL Reg to 1.
snd_wr_req ( PIO, DW, ANY, 0x00000000000020, 0x0000001400f00003 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );

# 1 of 32
rcv_rd_req ( MEM, FCL, ANY, 0x00000000100000 );
snd_rd_rsp ( MEM, FCL, ANY, 0, 0xF0F0F0F0F0F0F0F0, 0xF0F0F0F0F0F0F0F0,
       0xF0F0F0F0F0F0F0F0, 0xF0F0F0F0F0F0F0F0,
       0xF0F0F0F0F0F0F0F0, 0xF0F0F0F0F0F0F0F0,
       0xF0F0F0F0F0F0F0F0, 0xF0F0F0F0F0F0F0F0,
       0xF0F0F0F0F0F0F0F0, 0xF0F0F0F0F0F0F0F0,
       0xF0F0F0F0F0F0F0F0, 0xF0F0F0F0F0F0F0F0,
       0xF0F0F0F0F0F0F0F0, 0xF0F0F0F0F0F0F0F0 );

# Other Read DMA Transactions omitted here

# 32 of 32
rcv_rd_req ( MEM, FCL, ANY, 0x00000000100F80 );
snd_rd_rsp ( MEM, FCL, ANY, 0, 0xF0F0F0F0F0F0F0F0, 0xF0F0F0F0F0F0F0F0,
       0xF0F0F0F0F0F0F0F0, 0xF0F0F0F0F0F0F0F0,
       0xF0F0F0F0F0F0F0F0, 0xF0F0F0F0F0F0F0F0,
       0xF0F0F0F0F0F0F0F0, 0xF0F0F0F0F0F0F0F0,
       0xF0F0F0F0F0F0F0F0, 0xF0F0F0F0F0F0F0F0,
       0xF0F0F0F0F0F0F0F0, 0xF0F0F0F0F0F0F0F0,
       0xF0F0F0F0F0F0F0F0, 0xF0F0F0F0F0F0F0F0 );

print "\n\n*******Done storing 32 cache lines of data in SRAM1.\n\n";

print "\n\n*******Polling for DMA RD-SRAM1 done (bit 42 of CM_STATUS).\n\n";
poll (0x8, 42, 200);

#######  Reconfigure DMA Engine for Read DMA 3  #######

# RD DMA addresses.
snd_wr_req ( PIO, DW, ANY, 0x00000000000100, 0x0000000000100000 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );
snd_wr_req ( PIO, DW, ANY, 0x00000000000108, 0x0000000000400000 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );


#######  Start Read DMA Engine for Read DMA 3  #######
print "\n\n*******Start Read DMA Engine for SRAM2\n\n";

# Set Bit 36 of the CM_CONTROL Reg to 1.
snd_wr_req ( PIO, DW, ANY, 0x00000000000020, 0x0000001400f00003 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );
```

```
# 1 of 32
rcv_rd_req ( MEM, FCL, ANY, 0x00000000100000 );
snd_rd_rsp ( MEM, FCL, ANY, 0, 0x0C0C0C0C0C0C0C0C, 0x0C0C0C0C0C0C0C0C,
       0x0C0C0C0C0C0C0C0C, 0x0C0C0C0C0C0C0C0C,
       0x0C0C0C0C0C0C0C0C, 0x0C0C0C0C0C0C0C0C,
       0x0C0C0C0C0C0C0C0C, 0x0C0C0C0C0C0C0C0C,
       0x0C0C0C0C0C0C0C0C, 0x0C0C0C0C0C0C0C0C,
       0x0C0C0C0C0C0C0C0C, 0x0C0C0C0C0C0C0C0C,
       0x0C0C0C0C0C0C0C0C, 0x0C0C0C0C0C0C0C0C,
       0x0C0C0C0C0C0C0C0C, 0x0C0C0C0C0C0C0C0C );

# Other Read DMA Transactions omitted here

# 32 of 32
rcv_rd_req ( MEM, FCL, ANY, 0x00000000100F80 );
snd_rd_rsp ( MEM, FCL, ANY, 0, 0x0C0C0C0C0C0C0C0C, 0x0C0C0C0C0C0C0C0C,
       0x0C0C0C0C0C0C0C0C, 0x0C0C0C0C0C0C0C0C,
       0x0C0C0C0C0C0C0C0C, 0x0C0C0C0C0C0C0C0C,
       0x0C0C0C0C0C0C0C0C, 0x0C0C0C0C0C0C0C0C,
       0x0C0C0C0C0C0C0C0C, 0x0C0C0C0C0C0C0C0C,
       0x0C0C0C0C0C0C0C0C, 0x0C0C0C0C0C0C0C0C,
       0x0C0C0C0C0C0C0C0C, 0x0C0C0C0C0C0C0C0C );

print "\n\n*******Done storing 32 cache lines of data in SRAM 3.\n\n";

print "\n\n*******Polling for DMA RD-SRAM2 done (bit 42 of CM_STATUS).\n\n";
poll (0x8, 42, 200);

#######  Start the Algorithm  #######

# Set bit 38 of CM Control Register to 1 to start algorithm.
snd_wr_req ( PIO, DW, ANY, 0x00000000000020, 0x0000004400f00003 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );

print "\n\n*******Started Algorithm.\n\n";

# Poll for ALG_DONE bit in CM_STATUS.
poll ( 0x8, 48, 200 );

#######  Start Write DMA Engine.  #######

# Set bit 37 of CM Control Register to 1 to start Write DMA Engine.
snd_wr_req ( PIO, DW, ANY, 0x00000000000020, 0x0000002400f00003 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );

print "\n\n*******Started Write DMA Engine.\n\n";

# 1 of 32
```

```
rcv_wr_req ( MEM, FCL, ANY, 0x00000000100000, 0xDCACBCECDCACBCEC,
0xDCACBCECDCACBCEC,
      0xDCACBCECDCACBCEC, 0xDCACBCECDCACBCEC,
      0xDCACBCECDCACBCEC, 0xDCACBCECDCACBCEC,
      0xDCACBCECDCACBCEC, 0xDCACBCECDCACBCEC,
      0xDCACBCECDCACBCEC, 0xDCACBCECDCACBCEC,
      0xDCACBCECDCACBCEC, 0xDCACBCECDCACBCEC,
      0xDCACBCECDCACBCEC, 0xDCACBCECDCACBCEC,
      0xDCACBCECDCACBCEC, 0xDCACBCECDCACBCEC );
snd_wr_rsp ( MEM, FCL, ANY, 0 );

# Other Write DMA Transactions omitted here

# 32 of 32
rcv_wr_req ( MEM, FCL, ANY, 0x00000000100F80, 0xDCACBCECDCACBCEC,
0xDCACBCECDCACBCEC,
      0xDCACBCECDCACBCEC, 0xDCACBCECDCACBCEC,
      0xDCACBCECDCACBCEC, 0xDCACBCECDCACBCEC,
      0xDCACBCECDCACBCEC, 0xDCACBCECDCACBCEC,
      0xDCACBCECDCACBCEC, 0xDCACBCECDCACBCEC,
      0xDCACBCECDCACBCEC, 0xDCACBCECDCACBCEC,
      0xDCACBCECDCACBCEC, 0xDCACBCECDCACBCEC );
snd_wr_rsp ( MEM, FCL, ANY, 0 );

print "\n\n*******Polling for DMA WR-SRAM0 done (bit 45 of CM_STATUS).\n\n";
poll (0x8, 45, 200);

#######  Finish Up  ######

# dma_clear().  Set bits 39, 40, and 41 to 1 in CM_CONTROL.
snd_wr_req ( PIO, DW, ANY, 0x00000000000020, 0x0000038400f00003 );
rcv_wr_rsp ( PIO, DW, ANY, 0 );

# finalcheck_ccc() Check CACHE_RD_DMA_FSM.
snd_rd_req ( PIO, DW, ANY, 0x00000000000130 );
rcv_rd_rsp ( PIO, DW, ANY, 0, 0x0000000000400000 );

print "Reading the Error Status Register to insure no errors were logged.\n";
snd_rd_req ( PIO, DW, ANY, 0x00000000000060 );
rcv_rd_rsp ( PIO, DW, ANY, 0, 0x0000000000000000 );
```

Using the Stub

Because the SSP Stub is a very simple verification tool intended only for sandbox testing of new algorithm designs, capabilities and support for this stub are minimal. The main

intent is to provide the user with a means to simulate loading data, running their algorithm, and reading the results. More comprehensive testing should be done on real hardware. Due to limitations of the stub, below are some guidelines/rules for writing a diagnostic which will help the user quickly create functional tests.

- Commands in the stub **MUST** alternate between request and responses. This allows the stub to associate transaction pairs and calculate transaction numbers accordingly.
- Comments operate as follows: when the stub encounters a '#' character, it ignores all text from that character until it reaches the end of that line. Print statements are an exception to this rule; if the stub encounters a '#' in between the quotation marks of a print statement, it does **NOT** recognize this as a comment.
- A command should **ALWAYS** be followed by a semicolon.
- The parser checks for the correct number of inputs to a function, but it does NOT check the validity of these inputs. Misplaced semicolons may also lead to unpredictable behavior.
- Because the input file is in text format, no mathmatical operations, functions, or variables are available. The diagnostic writer must know the full data value to be written to a register.
- Diagnostic files must **ALWAYS** end with a newline character.
- Always follow a long sequence of DMA transactions a `poll`() command of the `WR_DMA_DONE` bit or `RD_DMA_DONE` bit. Enabling Interrupts or AMO's will help test for the completion of a long set of transactions.
- Other typical errors include the following:
  - diagnostic is too long. If you are running into memory-related errors, try increasing the `FILE_MAX` or `COMMAND_MAX` constants in `user_const.h` (number of characters allowed in diagnostic file).
  - Timed out while waiting for an operation to complete. If you are timing out before you receive your algorithm AMO or interrupt, increase `TIME_OUT` in `user_const.h`. If you are timing out during a poll, increase `POLL_MAX` in `user_const.h`.
- Read the Error Status Register at the end of a diagnostic to insure that no unreported errors were logged during the diagnostic.

How Extractor Works

This section describes how metadata is extracted from source files and how to interpret the generated configuration files. It covers the following topics:

Extractor Script

The extractor script extracts necessary configuration information from a user's register transfer level (RTL) code.

To use the extractor script, enter the following:

```
python $RASC/pd/shrd/extractor $RASC/design/tc_backend/wrapper/cs_rev_id_v.h $ALG_DIR
```

where `$RASC` is the installation path to the top of the RASC Core Services tree, and `$ALG_DIR` is the top of the algorithm design file tree. These variables can be set in the physical design makefile. Please see the physical design specifications for further details.

When called from a directory, extractor creates two configuration files: core_services.cfg and user_space.cfg.

The `Core_services.cfg` file is created from the file that is passed in as the first command line argument. In the file that is passed as the first argument, extractor searches the header for the glob that begins with '`core_services_versio`n' and ends with '`end:`'.

The `user_space.cfg` file is created from the files in the directory that is passed as the second argument to extractor. The search includes all `*.v`, `*.h`, or `*.vhd` files in and below the directory that is passed as the second argument. In those files, extractor searches for any comments that start with: extractor. Then it searches for the particular

tag, either `CS`, `VERSION`, `REG_IN`, `REG_OUT`, or `SRAM`. The line following these tags is then stored, manipulated in some cases, and written out to the `user_space.cfg` configuration file.

Both configuration files are written to the directory in which the script is called.

Please note that the system extractor is run upon requires a Python 2 interpreter. You can obtain the interpreter at no cost from www.python.org.

Core Services Configuration File

The format for entries in the core services configuration file is:

```
<type>:<value>
```

The various fields for the configuration file are described in detail in Table C-1.

Table C-1 Core Services Configuration File Fields

Type	Description	Value	Example
core_services_version	Tag for the version and revision of core services.	<version>.<revision>	core_services_version:0.7
part_num	Tag for the part number that corresponds to the CM_ID register in the hardware.	<hexidecimal part number>	part_num:0xF001
mfg_id	Tag for the manufacturer's identification number that corresponds to the CM_ID register in the hardware.	<hexidecimal manufacturer's ID>	mfg_id: 0x1
dma	Tag for the number and direction of the DMA engines.	<number> <direction[0] ... direction[n-1]>.	For two read and one write DMA engine, dma:3 rd rd wr
sram[n]	Tag for the SRAMs on the board The address field is from the DMA perspective; it is not the PIO space for that SRAM.	<DMA address for sram 'n'> <size of sram 'n' in MB>	sram[0]:0x00000000000000 2MB
amo	Tag for whether or not this core services is AMO capable	1 is yes, 0 is no	amo:1

Table C-1 Core Services Configuration File Fields **(continued)**

Type	Description	Value	Example
interrupt	Tag for whether or not this core services is interrupt capable	1 is yes, 0 is no	interrupt:1
semaphore	Tag for whether or not this core services is internal semaphore capable	1 is yes, 0 is no	semaphore:0
step	Tag for whether or not this core services is algorithm step capable.	1 is yes, 0 is no	step:1
mmr	Tag for the MMR space.	<base register for PIO> <length of space> <alignment>	mmr:0x00000000000000 2KB 64
debug	Tag for the Debug space.	<base register for PIO> <length of space> <alignment>	debug:0x00000000100000 1MB 64
debug_port[n]	Tag for the PIO address of the debug register that is associated with debug port 'n'.	<PIO address for the register> <bit width of the register>	debug_port[0]:0x0000 64
alg_def_reg[n]	Tag for the PIO address for a register 'n' that the RASC application wants to write to the algorithm	<PIO address for the register> <bit width of the register>	alg_def_reg[0]:0x0320 64
end	Tag for the end of the file	It has no value	end:

The core services configuration file should have the name `core_services.cfg`. This file does not use the ability for the application to write an internal register, so the file should appear as:

```
part_num:0xF004
mfg_id:0x1
dma:2 rd wr
buffer:2 16MB 0x00000000000000 0x00000001000000
sram[0]: 0x00000000000000 16MB
sram[1]: 0x00000001000000 16MB
amo:1
interrupt:1
```

```
semaphore:0
step:1
mmr:0x00000000000000 2KB 64
debug:0x00000000100000 1MB 64
debug_port[0]: 0x00000000000000 64
debug_port[1]: 0x00000000000008 64
debug_port[2]: 0x00000000000010 64
debug_port[3]: 0x00000000000018 64
debug_port[4]: 0x00000000000020 64
debug_port[5]: 0x00000000000028 64
debug_port[6]: 0x00000000000030 64
debug_port[7]: 0x00000000000038 64
debug_port[8]: 0x00000000000040 64
debug_port[9]: 0x00000000000048 64
debug_port[10]: 0x00000000000050 64
debug_port[11]: 0x00000000000058 64
debug_port[12]: 0x00000000000060 64
debug_port[13]: 0x00000000000068 64
debug_port[14]: 0x00000000000070 64
debug_port[15]: 0x00000000000078 64
debug_port[16]: 0x00000000000080 64
debug_port[17]: 0x00000000000088 64
debug_port[18]: 0x00000000000090 64
debug_port[19]: 0x00000000000098 64
debug_port[20]: 0x0000000000000A0 64
debug_port[21]: 0x0000000000000A8 64
debug_port[22]: 0x0000000000000B0 64
debug_port[23]: 0x0000000000000B8 64
debug_port[24]: 0x0000000000000C0 64
debug_port[25]: 0x0000000000000C8 64
debug_port[26]: 0x0000000000000D0 64
debug_port[27]: 0x0000000000000D8 64
debug_port[28]: 0x0000000000000E0 64
debug_port[29]: 0x0000000000000E8 64
debug_port[30]: 0x0000000000000F0 64
debug_port[31]: 0x0000000000000F8 64
debug_port[32]: 0x00000000000100 64
debug_port[33]: 0x00000000000108 64
debug_port[34]: 0x00000000000110 64
debug_port[35]: 0x00000000000118 64
debug_port[36]: 0x00000000000120 64
debug_port[37]: 0x00000000000128 64
debug_port[38]: 0x00000000000130 64
debug_port[39]: 0x00000000000138 64
debug_port[40]: 0x00000000000140 64
```

```
debug_port[41]: 0x00000000000148 64
debug_port[42]: 0x00000000000150 64
debug_port[43]: 0x00000000000158 64
debug_port[44]: 0x00000000000160 64
debug_port[45]: 0x00000000000168 64
debug_port[46]: 0x00000000000170 64
debug_port[47]: 0x00000000000178 64
debug_port[48]: 0x00000000000180 64
debug_port[49]: 0x00000000000188 64
debug_port[50]: 0x00000000000190 64
debug_port[51]: 0x00000000000198 64
debug_port[52]: 0x000000000001A0 64
debug_port[53]: 0x000000000001A8 64
debug_port[54]: 0x000000000001B0 64
debug_port[55]: 0x000000000001B8 64
debug_port[56]: 0x000000000001C0 64
debug_port[57]: 0x000000000001C8 64
debug_port[58]: 0x000000000001D0 64
debug_port[59]: 0x000000000001D8 64
debug_port[60]: 0x000000000001E0 64
debug_port[61]: 0x000000000001E8 64
debug_port[62]: 0x000000000001F0 64
debug_port[63]: 0x000000000001F8 64
alg_def_reg[0]: 0x00000000000320 64
alg_def_reg[1]: 0x00000000000328 64
alg_def_reg[2]: 0x00000000000330 64
alg_def_reg[3]: 0x00000000000338 64
alg_def_reg[4]: 0x00000000000340 64
alg_def_reg[5]: 0x00000000000348 64
alg_def_reg[6]: 0x00000000000350 64
alg_def_reg[7]: 0x00000000000358 64
end:
```

Algorithm Configuration File

The format for entries in the algorithm configuration file is:

```
<type>:<value>
```

The various fields for the configuration file are described in detail in Table C-2..

Table C-2 Algorithm Configuration File Fields

Type	Description	Value	Example
core_services_version	Tag detailing which iteration of core services was used in this bitstream.	<version> <revision>	core_services_version:0.7
algorithm_version	Tag for the type and version number of the algorithm in this bitstream. This is user defined but it should match the first register in the debug space (see Hardware Reference for more details.)	<version> <revision>	algorithm_version:0.12
src	Tag for source type that generated the algorithm.	Verilog by default and VHDL if ANY .vhd file exists in the hierarchy	src:v
reg	Tag for the registers that are PIO written or read by the application or the debugger and where the value will be mapped in the address space.	<register name> <bit width of register> <data type of the element mapped> <port mapping to the application or the debug relevant debug register>	reg:a 64 u debug_port[1]
array	Tag for the SRAM inputs and outputs to the algorithm.	<array name> <number of elements in array> <width of an element in bits> <associated SRAM bank for array> <byte offset into the SRAM bank> <direction of array: in, out, or internal> <data type for the elements in array> <whether the array is streaming or fixed>	array:a_in 512 64 sram[0] 0x0000 in u stream
end	Tag for the end of the file.	It has no value field.	end:

The title of the algorithm configuration file that is generated by the extractor script will be `user_space.cfg`. This name can be changed after the script is run at the user's discretion. The algorithm configuration file should appear as follows:

```
core_services_version:0.7
algorithm_version:0.6
src:v
reg:a_0 64 u debug_port[1]
reg:b_0 64 u debug_port[2]
reg:c_0 64 u debug_port[3]
reg:tmp 64 u debug_port[4]
reg:d_0 64 u debug_port[5]
reg:i_0 64 u debug_port[6]
array:a 512 64 sram[0] 0x00 in u stream
array:b 512 64 sram[1] 0x00 in u stream
array:c 512 64 sram[2] 0x00 in u stream
array:d 512 64 sram[0] 0x00 out u stream
end:
```

Index

S

U

V

W