



Stream User's Guide

This User's Guide gives an introduction to Stream programming and to the use of Stream tools. It describes how to install the Stream toolset. It presents an overview of SPI stream processors, the Stream programming model, and the software development tools used to compile, simulate, run, and debug Stream programs. It gives a detailed tutorial introduction to the design and implementation of an application program using a concrete programming example.

SWUG-00001-007

This document contains confidential and proprietary information of Stream Processors, Inc. Possession of this document or any part thereof in any form constitutes full acceptance of the terms and conditions of the mutual Non-Disclosure Agreement in effect between the recipient and Stream Processors, Inc. The contents of this document are preliminary and subject to change without notice. The stream processing technology and other technologies described in this document are subject to issued patents and pending patent applications in the United States and other countries. This document confers upon recipient no right or license to make, have made, use, sell, or practice any of the technology or inventions described herein.

Stream Processors, Inc.

455 DeGuigne Drive
Sunnyvale, CA 94085-3890 USA
Telephone: +1.408.616.3338
Fax: +1.408.616.3337
Email: info@streamprocessors.com
Web: www.streamprocessors.com

© 2005-2009 by Stream Processors, Inc. All rights reserved. This document contains advance information on SPI products, some of which are in development, sampling or initial production phases. The information and specifications contained herein are preliminary and are subject to change at the discretion of Stream Processors, Inc.





Table of Contents

1	Introduction.....	7
1.1	Typographical conventions	7
1.2	Document revision history	7
2	Installation	8
2.1	Install the toolset	8
2.2	Distribution contents	9
3	Stream Programming	10
3.1	Stream programming model	11
3.2	Stream language extensions	11
3.2.1	Added keywords	11
3.2.2	Predefined macros	12
3.2.3	Types	12
4	Component API.....	13
4.1	Basics.....	13
4.1.1	Components	13
4.1.2	Buffers	14
4.1.3	Ports	15
4.1.4	Connections	16
4.1.5	Commands and responses	16
4.2	Execution.....	18
4.2.1	Component instance states	18
4.2.2	Execution requirements	18
4.2.3	Scheduling priorities	19
4.2.4	Buffer lifecycle and ownership	21
4.2.5	Framebuffers.....	22
4.2.6	Processing elements.....	22
4.2.7	Resources	22
4.2.8	Providers	22
4.3	Runtime reporting.....	23
4.3.1	Logs	23
4.3.2	Timers	23
4.3.3	Tracing.....	24
4.4	Initialization files.....	24
4.4.1	Syntax	25
4.4.2	Example	26
5	Pipeline API.....	28
5.1	Streams	28
5.1.1	Restrictions	29
5.1.2	Stream and scalar parameter attributes	31
5.1.3	Example	31



5.2	Stream functions	32
5.2.1	Count	32
5.2.2	Block loads and stores	33
5.2.3	Strided loads and stores	33
5.2.4	Indexed loads and stores	33
5.2.5	Scalar output	33
6	Kernel API	34
6.1	Kernels	34
6.1.1	Limitations	35
6.2	DPU basic types	36
6.2.1	Type conversions	36
6.2.2	DPU booleans	37
6.3	Scalar and vector variables	37
6.4	Arrays	37
6.5	Operators	38
6.6	Control flow constructs	39
6.7	Stream access functions	40
6.7.1	Sequential streams	41
6.7.2	Conditional streams	41
6.7.3	Array streams	42
6.8	Intrinsic operations	42
6.8.1	Saturation arithmetic	43
6.8.2	Fractional arithmetic	43
6.8.3	Multiplication intrinsics	44
6.9	__repeat__	45
6.10	#pragma pipeline	45
6.11	#pragma local_array_size	45
7	Demo Application spm_demo	47
7.1	Testbench main	47
7.2	Data representation	48
7.3	Implementation alternatives	48
7.4	Buffer allocation	49
7.5	Streams	49
7.6	Kernels	51
7.7	File input component	53
7.7.1	Component definition	54
7.7.2	Properties function	54
7.7.3	Instance initialization function	55
7.7.4	Command handler function	56
7.7.5	Execute function	57
7.7.6	Destroy function	58



7.8	File output component.....	58
7.9	Green screen removal component	59
7.10	Component main.....	60
7.10.1	Initialization file.....	62
8	Command line tools.....	64
8.1	Functional mode: Run on host.....	64
8.2	Simulate with spsim	65
8.3	Run on hardware	65
8.3.1	Run from web page.....	67
8.4	Run application on host or on DSP MIPS	70
8.5	Run application on hardware.....	70
8.5.1	Initialization file.....	71
8.6	Logs.....	71
8.7	Timers.....	72
8.8	Performance.....	73
9	Stream Program Development.....	74
9.1	Invoke spide	75
9.2	Create a project.....	77
9.2.1	Create Stream project.....	77
9.2.2	Import source files	80
9.2.3	Create testbench module.....	81
9.3	Functional mode	83
9.3.1	Build	83
9.3.2	Run on host.....	84
9.3.3	Debug.....	86
9.3.4	Fast functional mode.....	88
9.4	Profile mode	89
9.4.1	Build	89
9.4.2	Run under simulator.....	89
9.4.3	View profile data	90
9.4.4	Run on hardware.....	91
9.5	Release mode.....	94
9.6	Complete application.....	95
9.6.1	Create System MIPS module.....	95
9.6.2	Create DSP MIPS Module.....	97
9.6.3	Run application.....	98
9.7	Import a project	100
9.8	Use Makefile from command line.....	100
10	Performance optimization	102
10.1	Pipelines.....	103
10.2	Visualization	105



10.3	Components	107
10.4	Tables.....	108
10.5	Stream operations	112
10.5.1	Dependency delays	112
10.5.2	Dispatch delays	117
10.6	Kernels	118
10.6.1	Tune	118
10.6.2	Reduce critical path	119
10.6.3	Remove control flow	120
10.6.4	Software pipeline	120
10.6.5	Unroll	120
11	Glossary.....	122
12	Index.....	125



1 Introduction

A *stream processor* is a high performance programmable processor for digital image processing and digital signal processing (DSP) applications. The stream processors of Stream Processors, Inc. (SPI) are programmable in an extended version of the C programming language, using the *Stream programming model* (SPM). The Stream programming model exposes the parallelism and locality inherent in an application program, and the SPI processor design and software development tools exploit this parallelism and locality in hardware.

This document gives a tutorial introduction to Stream programming and to the use of SPI Stream tools. It describes how to install the SPI Stream tools. It presents the essential concepts of SPI stream processors that you must understand to write efficient Stream programs. It describes the Stream extensions to the C language and the application programming interface (API) to the Stream programming model. It uses a demo program as a detailed introductory Stream programming example. It describes stream program development flow under an integrated development environment (IDE).

A companion volume, *Stream Reference Manual*, contains detailed reference information on Stream programming and on the tools in the Stream toolset. *Stream Release Notes* gives specific information related to the current release of the Stream tools.

1.1 *Typographical conventions*

This manual indicates a definition by setting the defined word in *italic* type. Italic type also indicates a placeholder that may take on different values; for example, an *n*-bit object might contain 8, 16, or 32 bits. **Bold** type indicates filenames and programming language literals; for example, **int** is a C data type. Monospace typeface `Courier` is used for command line input text and for C and Stream program fragments.

1.2 *Document revision history*

Document number	Date	Description	Release Version
SWUG-00001-001	December 2007	Initial release	RapiDev 1.0
SWUG-00001-002	January 2008	Revision	RapiDev 1.0.1
SWUG-00001-003	April 2008	Revision	RapiDev 1.0.2
SWUG-00001-004	June 2008	Major revision	Stream 2.0
SWUG-00001-005	August 2008	Revision	Stream 2.1
SWUG-00001-006	December 2008	Revision	Stream 2.2
SWUG-00001-007	March 2009	Revision	Stream 2.3



2 Installation

This chapter describes the installation of SPI Stream tools for Linux. Shell script **install.sh** installs the toolset from compressed **tar** archive files on the SPI [customer support website](#) or on a distribution CD.

The Stream development environment runs on a Linux host system and compiles Stream programs using a **gcc**-based MIPS cross compiler. The toolset is extensively tested running on FedoraCore 8.0, but it should also run successfully on most other x86 Linux distributions.

For users who wish to run the SPI Stream tools under Windows, SPI provides a Linux virtual machine that runs under the free VMware player (see www.vmware.com/products/player/). A separate document *VMware Player Installation and Setup Guide* provides instructions for the installation and use of the SPI VMware distribution.

2.1 Install the toolset

You should normally run as the superuser **root** to install the SPI Stream tools under Linux. If you need to install the tools on a machine on which you do not have root privileges, first install the SPI virtual machine distribution, and then perform the installation as **root** under the VMware player. Alternatively, you can specify the **-no_root** option to the installation script **install.sh**; in this case, the installation will not include an NFS-mountable filesystem for use with Linux running on System MIPS on an SPI development board.

Shell script **install.sh** installs the Stream distribution from a source (for example, the SPI customer support website or a distribution CD) to an arbitrary destination. By default, it downloads packages required for installation. Its usage is:

Usage: **install.sh** [*option* ...]

Options:

- d** *dest* Install to given directory *dest* [default: **/opt/spi/Stream_nnn**]
- no_root** Install without root permissions [does not produce NFS-mountable filesystem]
- r** *rep* Use directory *rep* as package download repository [default: **/opt/spi/download**]
- s** *src* Find packages locally in repository directory *src*
- url** *url* Download and install packages from *url* [default: SPI website]

To install the Stream distribution from the SPI customer support website, type:

```
$ ./install.sh [ -d dest ]
```

where *dest* gives an optional destination (default: **/opt/spi/Stream_nnn**). To install from a CD distribution instead, use the **-s** option:

```
$ src/install.sh -s src [ -d dest ]
```

where *src* gives the Stream distribution location (e.g., **/media/Stream_nnn** for a mounted CD).

After the installation is complete, you must add the Stream tools **bin** and **lib** directories to the settings of environment variables **PATH** and **LD_LIBRARY_PATH**, respectively. For example, for the **bash** shell, type:

```
export LD_LIBRARY_PATH=dest/lib:$LD_LIBRARY_PATH
export PATH=dest/bin:$PATH
```

You may want to add these lines to your **\$HOME/.bashrc** or to the global **/etc/bashrc**.



The **bsp** directory in the Stream distribution contains firmware for an SPI development board. It may be used to update board firmware using the on-board web pages. The **bsp** directory also includes compressed **tar** files containing sources. These sources are supplied in compliance with the [GNU Public License](#).

2.2 *Distribution contents*

This section gives a quick overview of the directory structure of the Stream distribution.

benchmark/	benchmark programs
bin/	binaries
bsp/	hardware board support package
demos/	demo programs
demos/spm_demo	Stream programming model demo application
doc/	documentation
include/	header files
installed_pkgs/	installed package repository
internal/	toolset internals
lib/	libraries
linux/	System MIPS Linux distribution
linux/target/	System MIPS Linux root filesystem

Later chapters of this manual use **spm_demo** to illustrate the Stream programming model and the use of the Stream tools. The **demos/** directory includes a video demo application **video_demo** in addition to **spm_demo**.

The **doc/** directory includes *Stream User's Guide*, *Stream Reference Manual*, and *Storm-1 Benchmarks* in PDF format.



3 Stream Programming

A stream processor contains a *general purpose unit* (GPU) for data handling and control, a *data parallel unit* (DPU) for compute-intensive inner loop computations, and *peripheral units* for device i/o. The Storm-1 processor GPU contains two MIPS processors: *System MIPS* (running Linux) handles user interface and device i/o, while *DSP MIPS* handles communication with the DPU.

A stream processor application program begins execution on System MIPS. The System MIPS application is a C source program (extension `.c`), compiled to a MIPS executable object that runs under Linux on System MIPS. The application may load and execute a DSP MIPS image, compiled by the Stream compiler `spc` from a Stream source program (extension `.sc`). The DSP MIPS image may in turn load and execute *kernel functions* on the stream processor DPU; the Stream source program defines both the DSP MIPS portion and the DPU portion of the program. The execution of System MIPS, DSP MIPS and DPU is asynchronous, with the Stream programming model handling any required synchronization.

A *stream* represents a sequence of structured data elements called *records*, each of the same type, stored in the lane register file (LRF) of a stream processor. A *kernel function* (or simply *kernel*) performs a computationally intensive operation on one or more input streams and produces one or more output streams. The DPU can access memory in the LRF, in a scalar operand register file (SORF), and in an operand register file (ORF), but it cannot access arbitrary memory. As a result, a Stream program running on DSP MIPS communicates with a kernel function running on the DPU only by means of streams (stored in the LRF) and scalar variables (stored in the SORF) that are the kernel function's arguments.

A kernel function is like a C function, but with some limitations on the types of statements that it can use; kernels are designed for high performance, which restricts the language features available in kernel code. A Stream program defines streams and passes streams as arguments to or from kernels, and stream processor hardware allows a kernel to access stream data efficiently. When kernel execution terminates, the Stream program can process the kernel's output streams and read the values returned from the kernel by scalar output variables.

The DPU design is a *VLIW* (very large instruction word) *SIMD* (single instruction, multiple data) architecture. The VLIW design allows the DPU to issue simultaneous instructions to multiple arithmetic-logical units (ALUs) in each hardware cycle. The SIMD design executes each instruction ("single instruction") simultaneously in multiple independent arithmetic processors called *lanes* (8 in SP8, 16 in SP16), with each lane operating on different data ("multiple data"). A kernel can perform multiple operations on multiple records in a data stream concurrently, resulting in very high efficiency.



3.1 *Stream programming model*

The SPI Stream programming model (SPM) is a parallel programming and execution model for stream processors. It allows the programmer to create Stream programs that use the powerful hardware features of a stream processor efficiently. It covers all levels of embedded system programming, from low-level data-parallel programming to efficient high-level multi-core parallelism. It consists of three application programming interfaces (APIs), each described in detail in a later chapter of this document:

- The *Component API* captures multi-core parallelism in a high-level modular program design framework.
- The *Pipeline API* uses on-chip memory management to communicate data efficiently between parts of a stream processor.
- The *Kernel API* captures data-level parallelism with direct access to efficient kernel operations.

The Stream programming model uses the C language with simple extensions to support data-parallel programming.

The Stream execution model is based on a set of connected components operating in a data-flow manner. An application calls **spi_spm_start** to start the Stream programming model runtime and calls **spi_spm_stop** to stop it. Alternatively, a program compiled with **spc** option **-m testbench** starts the SPM runtime automatically before it calls the user-supplied **spi_main** function.

Later chapters introduce the essential concepts of each Stream programming model API. Chapter [Demo Application spm_demo](#) uses a demo program to demonstrate the use of the APIs. *Stream Reference Manual* gives a detailed description of each SPM data type and function.

3.2 *Stream language extensions*

This section describes the language used for Stream programs, which is just standard C with a few extensions. A program can define structured record types and streams. It can define kernels that take streams and scalar variables as arguments. It can invoke kernels and execute special functions to control kernels and streams.

Many features of the Stream language are taken directly from standard C and are therefore not described here; see e.g. the C Standard (*American National Standard for Programming Languages – C*, ANSI/ISO 9899-1990, ISO/IEC 14882) for details. Lexical elements of the language are the same as C, except that several new keywords are added, as detailed in the [Added keywords](#) section below. Stream code outside of kernel functions is compliant with the C Standard, but kernel code supports a restricted subset of C, as described in the [Kernel API](#) chapter. The DSP MIPS runtime does not fully support the standard C library; see the *DSP MIPS Standard Library Functions* chapter of the *Stream Reference Manual* for details.

The Stream compiler **spc** compiles Stream programs. **spc** requires definitions from header file **spi_spm.h**, so all Stream programs must `#include "spi_spm.h"`.

3.2.1 *Added keywords*

The *Added keywords* section of the *Stream Reference Manual* gives a complete list of the keywords reserved for use by a Stream program in addition to the usual C keywords. Type modifiers **kernel** and **stream** identify kernels and streams. Type modifier **vec** in kernel code identifies a vector variable (i.e., a variable with a different value in each lane of the DPU). Types **int32x1**, **int16x2**, **int8x4**, **uint32x1**, **uint16x2**, and **uint8x4** represent DPU data types (one



32-bit signed integer, two 16-bit signed integers packed into one 32-bit word, and four 8-bit signed integers packed into one 32-bit word, plus their unsigned counterparts). The **__repeat__** keyword repeats a program block.

3.2.2 Predefined macros

The *Predefined macros* section of the *Stream Reference Manual* gives a complete list of preprocessor macros defined during compilation by the Stream compiler **spc**. Most of the macros depend on compilation options. Macro **SPI_LANES** defines the number of lanes in the DPU (16 on SP16, 8 on SP8).

3.2.3 Types

This section describes Stream types. In addition to the usual C data types, Stream programs can use DPU basic types (described in the [DPU basic types](#) section below), user-defined structured record types, and stream types.

3.2.3.1 Standard C types

Stream programs can use standard C data types:

- **char** and **unsigned char** are represented by an 8-bit byte.
- **short** and **unsigned short** are represented by a 16-bit halfword (two bytes).
- **int**, **unsigned int**, **long** and **unsigned long** are represented by a 32-bit word (four bytes).
- Pointers are represented by a 32-bit word (four bytes).
- **float** is represented by a 32-bit word (four bytes).
- **double** and **long double** are represented by a 64-bit dword (eight bytes).
- C9X types **long long** and **unsigned long long** are represented by a 64-bit dword (eight bytes).

Signed integers use 2's complement representation. Floating point types use IEEE format. Stream stores multibyte data in littleendian format. If **unsigned integer i** contains 0x03020100, Stream stores its bytes to successive increasing memory locations as 0x00, 0x01, 0x02, 0x03. Similarly, if **unsigned short s** contains 0x0100, Stream stores its bytes to successive increasing memory locations as 0x00, 0x01.

Kernels defined in Stream programs can use only special DPU basic types; see the [DPU basic types](#) section below for details.

3.2.3.2 Structured types

Stream functions and kernel functions use user-defined structured data types to represent stream data conveniently and concisely. A structure represents a fixed-length data *record* that forms a single element of a stream. It contains one or more members, where each member is a DPU basic type or a previously defined structured record type. For example,

```
typedef struct {  
    int32x1 x, y, z;  
} xyz;
```

defines type **xyz** that consists of three **int32x1** (32-bit signed integer) values. The structure name can be used as a new type. As in standard C usage, the member operator “.” provides access to a member of a record.

Stream does not permit bit-field structure members. Stream currently does not permit nested structures; only single-level **struct** is allowed. Structure members currently must be basic Stream types, not user-defined types.



4 Component API

This section introduces the basic elements of the Component API: components, buffers, ports, connections, commands and responses, instance states, execution requirements, the Stream execution model, logging, tracing, and timers.

4.1 Basics

4.1.1 Components

The *component* is the central concept of the Stream programming model. A component is a high-level data-driven computational module that typically reads input data from one or more input ports and writes output data to one or more output ports (though a source component only has output ports and a sink component only has input ports). A program may define components and may use components from supplied component libraries. The abstract modular nature of component definition encourages the interoperability and reuse of component libraries.

An application can create multiple *instances* of a component. For example, an application might invoke two instances of the same multiplexing component to produce two streams of output data from four streams of input data.

Within a component, program execution follows the familiar C programming model of single-threaded sequential execution. The Stream programming model frees the programmer from the burden of dealing with deadlock, race conditions, mutual exclusion, and data coherence (cache) issues.

Data type **spi_component_t** represents a component and **spi_instance_t** represents a component instance. An instance-specific context of type **spi_instance_context_t** identifies each instance. The Stream programming model defines the component functions listed below; see *Stream Reference Manual* for details.

- | | |
|--|--|
| • spi_component_find | Find a component with a given name and provider |
| • spi_component_get_desc | Get the description of a component |
| • spi_component_get_name | Get the name of a component |
| • spi_component_get_provider | Get the provider (e.g., SPI) of a component |
| • spi_component_get_version | Get the version of a component |
| • SPI_COMPONENT_NEW | Define a component |
| • spi_component_set_flags | Set the flags for a component |
| • spi_component_set_resource_requirements | Set the resource requirements for a component |
| • spi_get_component | Get the name of the component for the current component instance |
| • spi_get_name | Get the name of the current component instance |
| • spi_instance_new | Create a new component instance |
| • spi_schedgroup_component_find | Find a component in a scheduling group |

Macro **SPI_COMPONENT_NEW** defines a component. It takes as arguments five functions that specify the behavior of a component:

- the *properties function* defines properties of the component,
- the *instance initialization function* initializes a component instance,
- the *destroy function* destroys a component instance,
- the *execute function* executes a component instance when given conditions are satisfied, and



- the *command handler function* handles commands to a component instance.

A component properties function executes once, when the Stream programming model runtime begins execution; it can set component properties and resource requirements and register commands, ports, and execution requirements that apply to all instances of the component.. A component instance initialization function executes when **spl_instance_new** creates a new component instance. A component execution function executes when the component is running and specified execution properties are met; for example, a component might begin execution when input data is available on its input port and space is available on its output port. A component command handler handles component-specific commands.

The [Component definition](#) section of the [Demo Application spl_demo](#) chapter below gives an example of the use of **SPL_COMPONENT_NEW**.

4.1.2 Buffers

A *buffer* is a region of shared memory with a fixed size and alignment used to communicate data efficiently (i.e., without copying) between component instances. A Stream program must use a connection to pass a buffer between component instances; any other method results in undefined behavior. The use of buffers allows the programmer to write Stream code without explicit cache or processor synchronization code; the Stream programming model handles caching and synchronization issues automatically.

A Stream program uses a buffer as a data source for a Pipeline API **spl_load_*** function or as a data destination for a **spl_store_*** function. A kernel uses a Kernel API function **spl_*read** to read from a buffer and **spl_*write** to write to a buffer.

spl_buffer_new creates a new buffer with a given size, alignment, and flags. **spl_buffer_open** returns a pointer to the contents of a buffer (i.e., to the shared memory that the buffer represents). Buffer flags specify whether the buffer contents are readonly or reside in cached memory. **spl_buffer_close** closes a buffer and **spl_buffer_free** returns a buffer to a buffer pool.

spl_connection_pop pops a buffer from an input port and **spl_connection_push** pushes a buffer to an output port.

Data type **spl_buffer_t** represents a buffer. The Stream programming model defines the buffer functions listed below; see *Stream Reference Manual* for details.

- **spl_buffer_clone** Clone a buffer
- **spl_buffer_close** Close a buffer
- **spl_buffer_free** Free a buffer
- **spl_buffer_get_info** Get buffer information
- **spl_buffer_get_info_size** Get the buffer information size
- **spl_buffer_get_size** Get buffer size
- **spl_buffer_merge** Merge cloned buffers
- **spl_buffer_new** Create a new buffer
- **spl_buffer_open** Open a buffer (to allow access to its contents)
- **spl_buffer_set_info** Set buffer information
- **spl_connection_pop** Pop a buffer from a connection
- **spl_connection_push** Push a buffer to a connection
- **spl_get_buffer_heap_highwater** Get the shared memory heap highwater mark
- **spl_get_buffer_heap_size**



- **spi_load_*** Get the current shared memory heap size
- **spi_pool_get_buffer** Load data from a buffer to LRF
- **spi_store_*** Get a buffer from a buffer pool
- **spi_store_*** Store data from LRF to a buffer

4.1.2.1 Buffer pools

A *buffer pool* is a set of identically sized and aligned buffers. To avoid memory fragmentation, the Stream program model reuses buffers in a pool as they become available. Components and stream applications on System MIPS or on DSP MIPS can use buffer pools.

spi_pool_new creates a buffer pool with buffers of a given size and alignment. If the requested initial buffer count is non-zero, **spi_pool_new** allocates memory for the requested number of buffers.

spi_pool_get_buffer gets a buffer from a buffer pool. If the pool does not have any available buffers but was created with the **SPI_POOL_FLAG_GROW** flag, **spi_pool_get_buffer** allocates memory for a new buffer. A Stream program can allocate and free memory with the standard C library memory allocation functions **malloc**, **realloc**, **calloc**, and **free**, but memory allocated with these functions cannot be used as a buffer and cannot be shared between instances.

The Stream programming model defines the buffer pool functions listed below; see *Stream Reference Manual* for details.

- **spi_get_pool** Get the pool with a given name
- **spi_pool_free** Free a buffer pool
- **spi_pool_get_avail_buffer_count** Get the number of buffers available from a pool
- **spi_pool_get_buffer** Get a buffer from a buffer pool
- **spi_pool_get_desc** Get the description of a buffer pool
- **spi_pool_get_name** Get the name of a buffer pool
- **spi_pool_new** Create a buffer pool

4.1.2.2 Buffer information

An application can optionally attach additional *buffer information* to a buffer. Buffer information typically specifies properties of the buffer data (for example, how much of the buffer data is valid).

spi_buffer_set_info sets the information associated with a buffer, attaching a copy of the buffer information to the buffer. Thus, changing the contents of the specified object after this call does not change the information associated with the passed buffer.

spi_buffer_get_info returns a pointer to the information associated with a buffer. Buffer information becomes invalid when ownership of the buffer is released. Before ownership of the buffer is released, the information associated with the buffer can be modified using the pointer returned by **spi_buffer_get_info**.

4.1.3 Ports

A *port* provides the data interface between a component and the outside world. A port is either an input port or an output port. A program creates a connection to a port to move data to it or from it. A component may allow multiple connections to a single port.



Data type **spi_portdir_t** defines the direction of a port (input or output). The Stream programming model defines the port functions listed below; see *Stream Reference Manual* for details.

- | | |
|--|---|
| • spi_port_export | Export a port on a contained instance |
| • spi_port_get_connection | Get a connection attached to a port |
| • spi_port_get_connection_count | Get the number of connections of a port |
| • spi_port_get_desc | Get the description of a port |
| • spi_port_get_dir | Get the direction of a port |
| • spi_port_get_max_connection_count | Get the maximum number of connections allowed on a port |
| • spi_port_get_name | Get the name of a port |
| • spi_port_register | Define a port |

4.1.4 Connections

A Stream programming model application uses a *connection* to move data between component instances. A connection represents a single-writer single-reader FIFO that can contain a fixed number of buffers (the *depth* of the connection). **spi_connect** creates a connection between ports of two existing component instances, while **spi_connection_new** creates a connection from an application to a port on a contained component instance. **spi_connection_push** and **spi_connection_pop** push/pop a buffer to/from a connection.

Data type **spi_connection_t** represents a connection. The Stream programming model defines the connection functions listed below; see *Stream Reference Manual* for details.

- | | |
|-----------------------------------|---|
| • spi_connect | Create a connection between instances |
| • spi_connection_get_depth | Get the FIFO depth of a connection |
| • spi_connection_get_name | Get the name of a connection |
| • spi_connection_is_empty | Determine if a connection is empty |
| • spi_connection_is_full | Determine if a connection is full |
| • spi_connection_new | Create a connection to a contained instance |
| • spi_connection_pop | Pop a buffer from a connection |
| • spi_connection_push | Push a buffer to a connection |
| • spi_port_get_connection | Get a connection on a port |

4.1.5 Commands and responses

A Stream application or a component instance can send a *command* to a component instance, and the instance that receives the command can send back a *response* to indicate the success or failure of the command. Components and Stream applications on System MIPS or on DSP MIPS can send commands and responses.

A component defines the set of commands that it recognizes; the component's properties function calls **spi_cmd_register** to register each recognized command. For each command, the **spi_cmd_register** call also defines the format of the command *payload* (if any) and the format of the command *response payload* (if any). The **SPI_COMPONENT_NEW** macro that defines a component specifies a *command handler* function, invoked when an instance of the component receives a command. An instance may also call **spi_response_set_handler** to register response handler functions. The SPM runtime calls a response handler function when an instance receives a response to a previously sent command.



spi_instance_new creates a component instance, returning an instance handle. As there is no other way to obtain an instance handle, **spi_cmd_send** can only send commands to instances in the hierarchy of instances created under an instance, not to arbitrary instances.

When **spi_cmd_send** sends a command to a component instance, the receiving instance returns a **spi_response_t** response handle. Eventually, when the command handler of the receiving instance (specified by the **spi_component_instance_cmdhandler_fn_t** function in the **SPI_COMPONENT_NEW** definition of the receiving component) finishes processing the command, it calls **spi_cmd_send_response** to send the command response. The response handler of the sending instance (specified by **spi_response_set_handler**) handles the response, using the **spi_response_t** handle returned by **spi_cmd_send** to identify the command. The command response may include data in the form of a response payload.

Data type **spi_cmd_t** represents a command. The Stream programming model defines the command functions listed below; see *Stream Reference Manual* for details.

- **spi_cmd_free** Free a command
- **spi_cmd_get_desc** Get the command description
- **spi_cmd_get_id** Get the command id
- **spi_cmd_get_name** Get the command name
- **spi_cmd_get_payload** Get the command payload
- **spi_cmd_get_payload_size** Get the size of the command payload
- **spi_cmd_get_payload_type** Get the type of the command payload
- **spi_cmd_get_response_payload_type** Get the type of the command response payload
- **spi_cmd_register** Define a command
- **spi_cmd_send** Send a command
- **spi_cmd_send_response** Send a command response

Data type **spi_response_t** represents a response; **spi_send_command** returns a response. The Stream programming model defines the response functions listed below; see *Stream Reference Manual* for details.

- **spi_response_free** Free a response
- **spi_response_get_errno** Get the response error code
- **spi_response_get_payload** Get the response payload
- **spi_response_get_payload_size** Get the size of the response payload
- **spi_response_get_payload_type** Get the type of the response payload
- **spi_response_set_handler** Set a response handler
- **spi_response_strerror** Get a string describing a response error code

4.1.5.1 Command/response lifecycle

spi_cmd_send sends a command with a given ID to a component instance. The Stream runtime creates a **spi_cmd_t** command handle that represents the command and passes it to the receiving instance's command handler. The receiving instance then owns the **spi_cmd_t** object (including the optional command payload), which it should free with **spi_cmd_free** when it is no longer needed.

If the command handler does not recognize a command, it should free the **spi_cmd_t** handle and return 1; the Stream runtime then sends a response indicating that the command was not recognized. If the command handler recognizes the command, the receiving instance eventually should call **spi_cmd_send_response** to send a command response; it can send the response immediately or at some future time. When the response has been sent and the **spi_cmd_t** object is no longer needed, the receiving instance should free it with **spi_cmd_free**.



The behavior of **spi_cmd_send** differs depending on whether it is called from a Stream application or from a component instance. A **spi_cmd_send** call from a Stream application returns only when the receiving instance returns a response; that is, the **spi_cmd_send** blocks while awaiting a response. The application should free the **spi_response_t** object returned by **spi_cmd_send** with **spi_response_free** when it is no longer needed.

In contrast, a **spi_cmd_send** call from a component instance always returns a **spi_response_t** response token immediately, before the receiving instance returns the actual response. If the sending instance does not need to be notified of the actual response, it should free the returned **spi_response_t** with **spi_response_free**. If the sending instance does need to be notified of the actual response, it should call **spi_response_set_handler** to register a response handler. The sending instance will execute the registered response handler when it receives the actual response from the receiving instance. The sending instance should free the **spi_response_t** response once it is no longer needed.

4.2 Execution

4.2.1 Component instance states

An instance is always in one of three states: stopped (**SPI_INSTANCE_STATE_STOPPED**), paused (**SPI_INSTANCE_STATE_PAUSED**), or running (**SPI_INSTANCE_STATE_RUNNING**). The Stream execution model places a newly created instance in the paused state, so the instance's execute function will never be called, even if its execution requirements are satisfied. An instance may change its own state with **spi_set_state** or may have its state changed by receiving a **SPI_CMD_START**, **SPI_CMD_PAUSE**, or **SPI_CMD_STOP** built-in command. For example, to have new instances of a component start in the running state, add the following command to the component's initialization function:

```
spi_set_state(SPI_INSTANCE_STATE_RUNNING);
```

Typically, the application or component that creates an instance controls the state of the created instance. The following command changes instance **i0** to the running state:

```
spi_response_t response;  
response = spi_cmd_send(i0, SPI_CMD_START, NULL, 0);
```

Data type **spi_instance_state_t** represents a component instance state. The Stream programming model defines the instance state functions listed below; see *Stream Reference Manual* for details.

- **spi_get_state** Get the state of an instance
- **spi_set_state** Set the state of an instance

4.2.2 Execution requirements

An *execution requirement* is a condition that must be satisfied before the Stream scheduler invokes the execute function of a component instance. The Stream programming model provides several types of execution requirements that can be combined to create complex conditions. Execution requirement functions may be invoked only within a component, either on System MIPS or on DSP MIPS; that is, a program may not invoke an execution requirement function directly.

Data type **spi_execution_requirement_t** represents an execution requirement type. The Stream programming model defines the execution requirement functions listed below; see *Stream Reference Manual* for details.



- **spi_exec_req_activate** Make an execution requirement active
- **spi_exec_req_delete** Delete an execution requirement
- **spi_exec_req_is_satisfied** True if an execution requirement is satisfied
- **spi_exec_req_register** Register an execution requirement

4.2.2.1 Execution requirement types

Stream supports the following execution requirement types:

- **SPI_EXEC_ALLOF** Satisfied if all of a set of execution requirements are satisfied; used to compose a set of execution requirements into a more complex execution requirement.
- **SPI_EXEC_ALWAYS** Always satisfied.
- **SPI_EXEC_ANYOF** Satisfied if any of a set of execution requirements is satisfied; used to compose a set of other execution requirements into a more complex execution requirement.
- **SPI_EXEC_FD_READ** Satisfied if all of a set of file descriptors are ready for reading.
- **SPI_EXEC_FD_WRITE** Satisfied if all of a set of file descriptors are ready for writing.
- **SPI_EXEC_NEVER** Never satisfied.
- **SPI_EXEC_POOL** Satisfied if all of a set of buffer pools are ready. A buffer pool is ready if it contains at least one free buffer (that is, if the next call to **spi_pool_get_buffer** will return a buffer).
- **SPI_EXEC_PORT_ALLOF** Satisfied if all connections on a set of ports are ready. An incoming connection is ready if its FIFO is not empty and an outgoing connection is ready if its FIFO is not full.
- **SPI_EXEC_PORT_ANYOF** Satisfied if any connection on a set of ports is ready. An incoming connection is ready if its FIFO is not empty and an outgoing connection is ready if its FIFO is not full.

4.2.2.2 Execution requirement lifecycle

spi_exec_req_register creates an execution requirement of a given type with a given id. If the properties function of a component creates execution requirements, the requirements apply to all instances of the component. The initialization, execute, or command handler functions of a component may also call **spi_exec_req_register** to add additional execution requirements for a component instance.

By default, all execution requirements for a component instance must be satisfied before the instance's execute function is invoked. If an instance has no registered execution requirements, its execute function is always ready to be invoked. As an alternative to the default behavior, **spi_exec_req_activate** specifies a single execution requirement for an instance. **spi_exec_req_activate** may be called as often as desired to change the active execution requirement.

spi_exec_req_delete removes an execution requirement for an instance. If all execution requirements of an instance are removed, the instance's execute function is assumed to always be ready to be invoked.

4.2.3 Scheduling priorities

Each instance has a scheduling priority, with priority level 0 being the highest priority and priority level 15 the lowest. By default, all new instances are initially at priority level 8. **spi_set_priority** can change the priority of an instance. Sending built-in command **SPI_CMD_SET_PRIORITY** to an instance also can change its priority level.



The scheduling priority is an integer. The Stream programming model defines the scheduling priority functions listed below; see *Stream Reference Manual* for details.

- **spi_get_priority** Get the scheduling priority of an instance
- **spi_set_priority** Set the scheduling priority of an instance

4.2.3.1 Scheduling groups

Component instances within a group of instances called a *scheduling group* compete to have their execute functions invoked. A single image may contain any number of scheduling groups. By default, all components in an image are in the same default scheduling group. If a component is explicitly assigned to one or more scheduling groups in an image with **spi_schedgroup_register_component**, it is not placed in the default scheduling group of the image.

A typical Storm-1 application consists of two images: a System MIPS image that contains a **main** function plus zero or more components that execute on System MIPS, and a DSP MIPS image that contains one or more components that execute on DSP MIPS (including all components that use the DPU). Thus, a typical application has two scheduling groups: one runs on System MIPS and one runs on DSP MIPS. Macro **SPI_SCHEDGROUP_NEW** creates a new scheduling group explicitly.

Each scheduling group controls all component instances created from components in the group. All component instances in a scheduling group compete for scheduling based on their priority, state, and execution requirements. Once an instance's initialization, execute, command handler, or response handler function is invoked, that function is guaranteed to complete before the scheduler invokes any other function of an instance from the scheduling group; the instance's functions are never preempted.

Each scheduling group maintains 16 priority queues, one for each scheduling priority level. Within a priority queue, ready instances are scheduled in round-robin order. The scheduler searches the queues in priority order to find a ready instance: if queue 0 (the highest priority) contains no ready instance, the scheduler searches for a ready instance in queue 1, and so on.

Each scheduling group uses the following processing loop:

- Command and response processing:
 - Check each instance for incoming commands.
 - If any, invoke the instance's command handler function for the incoming command.
 - Check each instance for incoming responses.
 - If any, invoke the response handler associated with the response.
- Schedule execution:
 - Search the priority queues for the highest-priority ready instance
 - Invoke the execute function for the instance.
 - After execution, move the instance to the end of its priority queue.

The Stream programming model defines the scheduling group functions listed below; see *Stream Reference Manual* for details.

- **spi_schedgroup_component_find** Find a component in a scheduling group
- **SPI_SCHEDGROUP_NEW** Define a new scheduling group
- **spi_schedgroup_register_component** Register a scheduling group component
- **spi_schedgroup_set_controlled_resources** Set the resources controlled by a scheduling group
- **spi_schedgroup_set_min_stacksize** Set the minimum stacksize for a scheduling group
- **spi_schedgroup_set_processing_elements** Set the processing elements required for a scheduling group



4.2.4 Buffer lifecycle and ownership

4.2.4.1 Buffer lifecycle

The component instance or Stream application that creates a buffer pool with **spi_pool_new** owns the pool. Only the owning instance or application can call functions that use the pool; a pool cannot be shared or communicated to other instances or applications. A component properties function cannot create a pool; instead, its instance initialization function can create a pool, so that each instance of the component gets its own pool. The command handler function or the execute function of a component instance also can create a buffer pool.

spi_buffer_clone creates a new buffer that represents the same memory region as an existing buffer. Cloning a buffer allows multiple component instances to access the same buffer data. Because different instances can execute in arbitrary order or even concurrently, the use of buffer clones potentially can lead to non-deterministic behavior if a buffer clone writes to a memory location accessed by another buffer clone. To avoid this non-determinism, buffer clones should only access non-overlapping memory locations (though multiple clones can read from the same location without introducing non-deterministic behavior). Future Stream implementations will provide debugging support to verify that buffer clones do not access overlapping memory.

If a Stream application writes to two or more buffers that represent the same memory region, it must use **spi_buffer_merge** to unify the buffers into a new buffer that consolidates the writes. **spi_buffer_merge** can only merge buffers that represent the same memory region.

When a buffer is no longer needed, a Stream application or component may free it with **spi_buffer_free** or **spi_buffer_merge**. When all buffers that represent a memory region have been freed, the memory region returns to the buffer pool and becomes available for reuse.

spi_load_* loads a stream with the contents of a buffer so that the DPU can read the buffer's data from a stream. Similarly, **spi_store_*** stores a stream to a buffer so that DSP MIPS can access the buffer's data. A program can use these pipeline API functions to modify buffer data. If the program instead wishes to access buffer data directly (e.g., through a pointer to the buffer data), it must first call **spi_buffer_open** to obtain a pointer to the memory region the buffer represents. The program then can read or write data within the region through the pointer. When the program is finished with its direct access to the buffer data, it should call **spi_buffer_close** to invalidate the pointer returned by **spi_buffer_open**, disallowing further accesses to the buffer's memory region using that pointer. A buffer cannot be opened if it is already open.

If a program only needs to read the contents of a buffer, it should call **spi_buffer_open** with flag **SPI_BUFFER_FLAG_READONLY**. By default, a buffer is in uncached memory, but flag **SPI_BUFFER_FLAG_CACHED** can be used to obtain a buffer in cached memory instead. **spi_buffer_close** flushes cached buffers to propagate all buffer modifications to memory.

4.2.4.2 Buffer ownership

A buffer has at most one owner at any time, and buffer ownership changes as a buffer is transferred between component instances and Stream applications. An instance may open, close, or free a buffer, or use the buffer as an argument to a **spi_load_*** or **spi_store_*** function, only if the instance owns the buffer.

Initially, the instance or application that gets a buffer with **spi_buffer_new** or **spi_pool_get_buffer** owns the buffer. The owning instance or application releases buffer ownership when it frees the buffer with **spi_buffer_free**, when it



merges the buffer with **spi_buffer_merge**, or when it pushes the buffer onto a connection with **spi_connection_push**. An instance or application takes ownership of a new buffer created with **spi_buffer_clone** or **spi_buffer_merge**, and also takes ownership of a buffer popped from a connection with **spi_connection_pop**.

4.2.5 Framebuffers

A *framebuffer* is part of a Linux graphical abstraction layer, as described in the Wikipedia article [Linux framebuffer](#) and in Linux documentation. A System MIPS application can initialize framebuffer use, for example with command **fbset** or through **/dev/fb***. The following Stream programming model functions provide framebuffer support:

- **spi_fb_get_line_length** Get the line length of a framebuffer in bytes
- **spi_fb_get_pixel_type** Get the pixel type of a framebuffer
- **spi_fb_get_xres** Get the horizontal (X) resolution of a framebuffer in pixels
- **spi_fb_get_yres** Get the vertical (Y) resolution of a framebuffer in pixels
- **spi_fb_is_fb_available** Check whether a framebuffer is available
- **spi_fb_pool_new** Create a new framebuffer buffer pool

4.2.6 Processing elements

A Stream programming model *processing element* represents a hardware processor (for example, System MIPS or DSP MIPS) on which a scheduling group can execute. Some components might be coded to run on either System MIPS or DSP MIPS. Other components might be tied to a specific processor: a device i/o component might require System MIPS resources, while a component that uses the DPU must run on DSP MIPS to communicate with the DPU.

Data type **spi_pels_t** represents a set of processing elements. Function **spi_load_image** loads a program image on a processing element.

4.2.7 Resources

A Stream programming model *resource* represents a hardware or software resource (for example, the DPU).

Data type **spi_resources_t** represents a set of resources. The Stream programming model defines the timer functions listed below; see *Stream Reference Manual* for details.

- **spi_component_set_resource_requirements** Set the resource requirements for a component
- **spi_schedgroup_set_controlled_resources** Set the resource resources for a scheduling group

4.2.8 Providers

A *provider* is an organization that provides Stream programming model components. For example, Stream Processors, Inc. is provider **SPI_PROVIDER_SPI**.

Data type **spi_provider_t** identifies a provider. The Stream programming model defines the provider functions listed below; see *Stream Reference Manual* for details.



- | | |
|--|---|
| • spi_component_get_provider | Get the provider of a component |
| • SPI_COMPONENT_NEW | Define a component, including its provider |
| • spi_provider_get_name | Get a provider name |
| • SPI_SCHEDGROUP_NEW | Define a scheduling group, including its provider |
| • spi_schedgroup_register_component | Register a component with a scheduling group |

4.3 Runtime reporting

4.3.1 Logs

The Stream programming model provides *logs* for runtime messages. Every component generates a debug log with log name **SPI_LOG_DEBUG** and an error log with log name **SPI_LOG_ERROR**. A component may define additional logs with **spi_log_new**.

A logging level controls the amount of logged information. The logging level is a 32-bit bitmask, called the *enable mask* of the log, so a program can control up to 32 independent logging levels for each log. By default, the SPM runtime disables all debug log levels, enables all error log levels, and intermixes timestamped output from all logs on **stdout**. The user can control log behavior with special SPM command-line options:

- | | |
|-----------------------------------|---|
| --spi_log_dir=dir | specifies a log file directory, |
| --spi_log_mask=log,mask | specifies an enable mask for a log, and |
| --spi_log_timestamps=[0 1] | disables or enables log entry timestamps. |

The Stream programming model defines the logging functions listed below; see *Stream Reference Manual* for details.

- | | |
|----------------------------------|-------------------------------|
| • spi_get_log | Get the log with a given name |
| • spi_log | Write a message to a log |
| • spi_log_get_desc | Get the description of a log |
| • spi_log_get_enable_mask | Get the enable mask of a log |
| • spi_log_get_name | Get the name of a log |
| • spi_log_new | Define a log |
| • spi_log_set_enable_mask | Set the enable mask of a log |

4.3.2 Timers

The Stream programming model provides built-in *timers* to measure program performance. A component can also define additional timers with **spi_timer_new**. Timers measure execution time on stream processor hardware or on simulation with **sprun**. Timer measurements under the simulator are very accurate for DSP MIPS code and for long-running kernels, but can differ from hardware execution times for stream operations.

The Stream programming model includes several built-in timers:

- **SPI_TIMER_CMDHANDLER** measures the time spent in the command handler function of a component.
- **SPI_TIMER_EXECUTE** measures the time spent in the execute function of a component.
- **SPI_TIMER_KERNEL** measures the time spent in the most recently invoked kernel.
- **SPI_TIMER_LOAD_DSP** measures the time required to load a DSP MIPS image.



- **SPI_TIMER_SPM** starts when the stream programming model runtime starts. The stream programming model runtime never stops this timer, so a program can use it to measure elapsed time since the runtime started.
- **SPI_TIMER_STARTUP** measures the startup time of the stream programming model runtime.

To reduce execution overhead, the SPM runtime updates **SPI_TIMER_CMDHANDLER** and **SPI_TIMER_EXECUTE** timers only in debug mode or in profile mode, not in release mode. It updates the other three built-in timers in all modes.

The Stream programming model defines the timer functions listed below; see *Stream Reference Manual* for details.

- | | |
|--|--|
| • spi_get_time | Get the system time |
| • spi_get_timer | Get the timer with a given name |
| • spi_timer_get_desc | Get the description of a timer |
| • spi_timer_get_name | Get the name of a timer |
| • spi_timer_get_nanoseconds | Get the elapsed time since a timer started |
| • spi_timer_get_start_count | Get the number of times a timer has been started |
| • spi_timer_get_total_nanoseconds | Get the total elapsed time of a timer |
| • spi_timer_new | Define a timer |
| • spi_timer_start | Start a timer |
| • spi_timer_stop | Stop a timer |

4.3.3 Tracing

Simulation of a program compiled in profile mode produces trace information, allowing the user to evaluate program performance with **spperf** or **spide**. The Stream programming model defines the program tracing functions listed below; see *Stream Reference Manual* for details.

- | | |
|-------------------------------|-----------------------------|
| • spi_trace_is_enabled | Check if tracing is enabled |
| • spi_trace_start | Start tracing |
| • spi_trace_stop | Stop tracing |

4.4 Initialization files

Instead of providing explicit code to create component instances, create connections between instances, and execute instance initialization commands for an application, a programmer can provide a high-level description of component instances, connections, and instance initialization commands in an *initialization file*. Initialization files can simplify the coding of SPM applications.

The user may specify initialization files at runtime by passing one or more **--spi_init_file=file** options to **spi_spm_start**. **spi_spm_start** processes initialization files in the order of the **--spi_init_file** options. It returns a failure status if it encounters any error while processing an initialization file. To see the cause of the failure, build a debug version of the executable (or of both System MIPS and DSP MIPS executables) with **spc -g**. Then pass option **--spi_log_mask=debug,1** to **spi_spm_start** to enable log level **SPI_LOG_LEVEL_DEBUG** in the **SPI_LOG_DEBUG** log and examine the debug log to diagnose the cause of the failure.

spi_spm_start creates instances and connections described in the initialization files and executes initialization commands in the files in the given order. If any command receives a failing response error code (that is, any error code other than **SPI_RESPONSE_ERRNO_OK**), **spi_spm_start** aborts initialization file processing and returns



failure status. The application can use **spi_get_instance** and **spi_get_connection** to get access to created instances and connections, as shown in the [Example](#) below.

4.4.1 Syntax

An initialization file is a sequence of statements using XML syntax. Each statement consists of a tag followed by one or more key/value pairs:

```
<tag key=value ... />
```

Here *tag* is one of **image**, **instance**, **connection**, or **command**; each is described below. Each *key* is a tag-specific name, and *value* gives the value for *key*. In keeping with XML syntax, each *value* should be quoted.

The initialization file may also include XML-style comments:

```
<!--This is a comment -->
```

Comments must be on a single line, but other XML statements in an initialization file may span multiple lines.

4.4.1.1 *image*

The **image** statement loads an image onto the DSP MIPS processor. Its format is:

```
<image target=pel file=pathname [ argv=arglist ] />
```

Here *pel* is **SPI_PEL_DSP_MIPS** to specify the DSP MIPS processor and *pathname* gives the pathname of the executable image to be loaded to DSP MIPS. The optional **argv** key gives the argument list *arglist* for the target image. The *arglist* consists of whitespace-separated arguments, with single quotes to enclose an argument containing whitespace. For example,

```
<image target="SPI_PEL_DSP_MIPS"
      file="prog.dsp.out"
      argv="foo bar 'foo and bar'" />
```

specifies **argv[0]** = "foo", **argv[1]** = "bar", and **argv[2]** = "foo and bar".

4.4.1.2 *instance*

The **instance** statement creates a new instance of a component and sets it to the running state. Its format is:

```
<instance name=name
      component=component
      provider=provider
      [ schedgroup=schedgroup ]
      [ min_version=min ]
      [ max_version=max ]
      [ initial_state=initial_state ] />
```



Here *name* gives the name of the new instance; the application can subsequently call **spi_get_instance(name)** to get the **spi_instance_t** handle of the instance. *component* gives the name of the component from which the instance is created. *provider* gives the component provider (for example, **SPI_PROVIDER_SPI**).

The remaining **instance** keys are optional. *schedgroup* specifies a scheduling group name to search for the component; **spi_spm_start** searches for *component* with **spi_schedgroup_component_find** if this key is given, or with **spi_component_find** otherwise. *min* and *max* specify the required component version. *initial_state* specifies the initial state of the component, with possible values "**paused**" and "**running**"; a new instance normally starts in the paused state.

4.4.1.3 connection

The **connection** statement creates a connection between two previously created instances or between a previously created instance and the application. Its format is:

```
<connection name=name
    depth=depth
    [ from=instance:port ]
    [ to=instance:port ] />
```

Here *name* gives the name of the new connection; the application can subsequently call **spi_get_connection(name)** to get the **spi_connection_t** handle of the connection. *depth* is the maximum number of buffers allowed in the connection at any one time. *instance* and *port* specify an instance name and port name to connect. If the **connection** statement specifies both **from** and **to** keys, **spi_spm_start** creates a connection in the same manner as **spi_connect**. If the statement specifies only a **from** or a **to** key, **spi_spm_start** creates a connection between the application and an instance in the same manner as **spi_connection_new**.

4.4.1.4 command

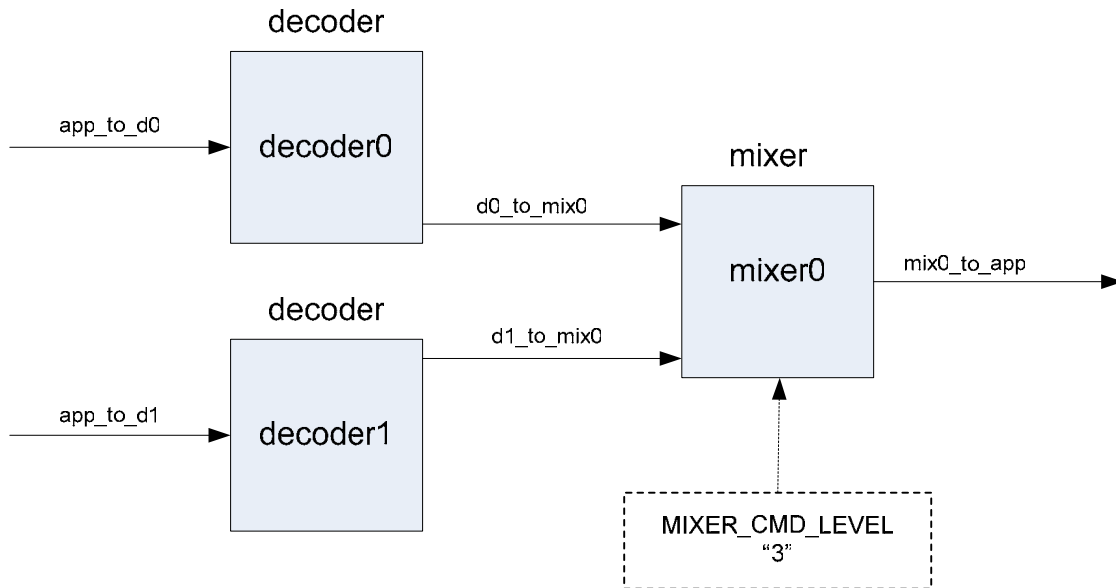
The **command** statement sends one or more commands to an instance. Its format is:

```
<command instance=name cmd=payload ... />
```

Here *name* gives the name of the instance to which the commands are sent. *cmd* is the name of a command: either a built-in command (for example, **SPI_CMD_START**), or the name of the command created by **spi_cmd_register** (for example, **FOO_CMD_DOIT**). *payload* is the payload associated with the *cmd*; for a command with no payload, *payload* must be "**null**" or "**NULL**".

4.4.2 Example

In this example, DSP MIPS image **dsp.out** contains two components, a decoder and a mixer. Component **decoder** has one input port **DECODER_PORT_IN** and one output port **DECODER_PORT_OUT**. Component **mixer** has one input port **MIXER_PORT_IN** and one output port **MIXER_PORT_OUT**. The mixer component defines command **MIXER_CMD_LEVEL** with an integer payload. The initialization file below creates two instances of **decoder** and one instance of **mixer**, connects the instances and the application, and sends a command to the mixer:



```

<image target="SPI_PEL_DSP_MIPS" file="dsp.out" />

<instance name="decoder0" component="decoder" provider="SPI_PROVIDER_SPI" />
<instance name="decoder1" component="decoder" provider="SPI_PROVIDER_SPI" />
<instance name="mixer0" component="mixer" provider="SPI_PROVIDER_SPI" />

<connection name="app_to_d0" depth="2" to="decoder0:DECODER_PORT_IN" />
<connection name="app_to_d1" depth="2" to="decoder1:DECODER_PORT_IN" />
<connection name="d0_to_mix0" depth="2"
  from="decoder0:DECODER_PORT_OUT"
  to="mixer0:MIXER_PORT_IN" />
<connection name="d1_to_mix0" depth="2"
  from="decoder1:DECODER_PORT_OUT"
  to="mixer0:MIXER_PORT_IN" />
<connection name="mix0_to_app" depth="2" from="mix0:MIXER_PORT_OUT" />

<command instance="mixer0" MIXER_CMD_LEVEL="3" />

```

The application can use **spi_get_connection** to get a handle to the **app_to_d0** connection and then use that connection to send a buffer to instance **decoder0**:

```

spi_connection_t app_to_d0_connection = spi_get_connection("app_to_d0");
spi_connection_push(app_to_d0_connection, buffer, -1);

```

Similarly, the application can use **spi_get_instance** to get a handle to the **mixer0** instance and then use that handle to send a command to the mixer:

```

spi_instance_t mixer0_inst = spi_get_instance("mixer0");
spi_response_t response = spi_cmd_send(mixer_inst, MIXER_CMD_LEVEL, 5, 0);

```

Section [Initialization file](#) of chapter [Demo Application spm_demo](#) below provides a concrete example of the use of an initialization file in demo program **spm_demo**.



5 Pipeline API

A kernel function running on the DPU cannot access Stream program data in DSP MIPS memory directly. The Stream programming model Pipeline API defines stream functions to load data from DSP MIPS memory to the lane register file (LRF) and to store data from the LRF to DSP MIPS memory, using efficient stream processor hardware instructions. These functions allow the Stream programming model to handle DSP MIPS / DPU data coherency issues (cache) automatically. The *Stream Reference Manual* chapter *Pipeline API* describes each Pipeline API function in more detail.

5.1 Streams

The DPU of a stream processor cannot access memory directly. Instead, it accesses data in the lane register file (LRF) of the processor. Stream programs represent LRF data as *streams* and use streams to pass data to and from kernel functions. A stream represents a fixed-length sequence of records of a given type in the LRF.

The [Pipeline API](#) chapter below describes DSP MIPS stream functions, including `spi_load_*` and `spi_store_*` functions that load stream data to the LRF and store stream data from the LRF. The [Kernel API](#) chapter below describes kernel stream functions, including `spi_*read` and `spi_*write` functions that read stream data from the LRF and write stream data to the LRF.

A Stream program may declare a stream only within a function (that is, as a local declaration); global stream declarations are not allowed. A stream declaration uses standard C syntax with one extension: the *size* of the stream in the LRF is specified in parentheses after the stream name:

```
stream int    chicken(16);           // a stream of 16 ints (one per lane on SP16)
```

The stream size indicates the number of records allocated in the LRF for this stream; it must be a compile time constant. The size gives the total number of data records for which LRF space is allocated, so each lane is allocated space for `size / SPI_LANES` data records. Because of DPU hardware restrictions, the specified stream size must always be a multiple of `SPI_LANES`.

A function that declares and uses streams is called a pipeline function. `spc` currently performs LRF allocation on a per-pipeline function basis, so a pipeline function may not call another pipeline function.

A stream declaration can specify an explicit LRF address (byte offset) in addition to a size:

```
stream int    turkey(256, 1024);    // a stream of 256 ints at LRF address 1024
```

This declares a stream of 256 words which begins at byte offset 1024 in the LRF. The offset must be a compile-time constant and a multiple of `4 * SPI_LANES`. A program should not declare streams with explicit offsets that result in overlapping streams, as `spc` will not handle the aliasing of the streams correctly. In general, SPI discourages the use of stream declarations with explicit LRF address specifications.

The LRF is of limited size: it contains `SPI_LRF_SIZE` words per lane. On SP16 and SP8, `SPI_LRF_SIZE` is 4,096, so the LRF contains 256 Kbytes on SP16, 128 Kbytes on SP8. The total LRF space allocated by all streams “live” at any one time cannot exceed the size of the LRF. `spc` determines the “live” range of a stream in a program through analysis of stream use in the code. By default, `spc` tries to preserve parallelism between kernels and stream loads and stream stores. It searches backwards from each `spi_load_*` to find the first preceding kernel, and then it allocates the LRF so that the load and the kernel can proceed in parallel if they are not data-dependent. Similarly, it searches forward from each `spi_store_*` to find the first subsequent kernel, and then it allocates the LRF so that the store and the kernel can proceed in parallel if they are not data-dependent. If this algorithm results in over-allocation of the LRF, `spc` issues a warning and attempts to allocate streams by reducing program parallelism. It



reports a compile time error if the LRF remains over-allocated. In this case, the programmer must reduce LRF use by reducing stream sizes.

By default, **spc** allocates 1 Kbyte per lane to hold local arrays for a kernel. Use the **local_array_size** pragma described [below](#) to change the default value for a kernel.

Stream stores records sequentially in memory, just like an array. For example, consider the following code:

```
typedef struct { int32x1 x, y, z; } xyz;
stream xyz    my_stream(96);
spi_buffer_t buf;
...
spi_load_block(my_stream, buf, 0, 96);
...
```

Here **spi_load_block** loads 96 3-word records (288 words) of stream data from buffer **buf** into the LRF. If the data stored in **buf** is record **r[0]** through record **r[95]**, then the records are stored in **my_stream** in the LRF as follows:

Word:	0	1	2	3	4	5	...	285	286	287
Member:	r[0].x	r[0].y	r[0].z	r[1].x	r[1].y	r[1].z	...	r[95].x	r[95].y	r[95].z
Record:	r[0]			r[1]			...	r[95]		

Stream stores multibyte data in littleendian format; the diagram above does not show individual bytes.

5.1.1 Restrictions

Because streams are used for transferring data to a kernel function running on the DPU, stream data record types must be constructed from DPU basic types. User-defined structured stream data types may only contain DPU basic types. Stream code cannot assign to streams, use streams in expressions, use pointers to streams, or use arrays of streams.

For example:

```
stream int          a(16), b(16); // Legal
stream int32x1      *d, e(32);    // Illegal: cannot have pointers to streams
stream int32x1      f[10];        // Illegal: cannot have array of streams
...
a = b;               // Illegal: cannot assign streams
d = &e;              // Illegal: cannot have pointers to streams
```



The table below provides additional detail on the use of various Stream types.

Type	Example(s+)	Declare in contexts				Derived types				
		C	Kernel argument	inline kernel argument	Inside a kernel	Struct field	Vector of	Array of	Pointer to	Stream of
DPU basic type	<code>int8x4</code>	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Other basic type	<code>char</code>	Yes	-	-	-	Yes	-	Yes	Yes	-
Struct of <i>only</i> DPU basic types	<pre>struct { int8x4 x; }</pre>	Yes	-	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Struct of other	<pre>struct { struct { int8x4 x; } } struct { char x; }</pre>	Yes	-	-	-	Yes	-	Yes	Yes	-
Vector	<code>vec int8x4</code>	-	-	Yes	Yes	-	-	Yes	-	-
Array of vector type	<code>vec int8x4 [...]</code>	-	-	Yes	Yes	-	-	-	-	-
Array of other	<code>int8x4 [...]</code>	Yes	-	-	-	Yes	-	Yes	Yes	-
Pointer	<code>int8x4*</code>	Yes	-	-	-	Yes	-	Yes	Yes	-
Stream	<code>stream int8x4 (...)</code>	Yes	Yes	Yes	-	-	-	-	-	-



5.1.2 Stream and scalar parameter attributes

Stream and scalar parameters to kernels may have *attributes* that modify the behavior of a specific use of a stream or scalar. Stream code specifies attributes in parentheses directly after a stream or scalar variable name; this syntax is an extension to standard C syntax. Four attributes can be applied to streams or scalars.

Attribute	Description	Name	Value	Where Valid	Example
Size	Size of stream in records	size	Integer. Must be a compile-time constant and a multiple of SPI_LANES .	Required in stream declaration	<code>stream int foo(size=32);</code>
LRF address	LRF address (byte offset) of stream	lrf_address	Integer. Must be a compile-time constant and a multiple of 4 * SPI_LANES .	Optional in stream declaration	<code>stream int turkey(size=256, lrf_address=1024);</code>
I/O type	Direction and type of stream or scalar argument to kernel function	type	For a stream: one of in , out , seq_in , seq_out , cond_in , cond_out , array_in , array_out , array_io . For a scalar: one of in , out .	Required in kernel function declaration	<code>kernel void k1(stream int in_s(type=seq_in), stream int out_s(type=seq_out), int count(type=in));</code>
Substream	Selects subset of stream; used to efficiently process a subset of the LRF space allocated for a stream	offset , size	Unsigned integers. size is the substream size in records and offset is an offset in records; each must be a multiple of SPI_LANES , and offset + size must not be greater than the size specified in the stream declaration.	Optional in parameter to spl_load_* , spl_store_* , or a kernel function call	<code>k1(in_s(offset=16, size=32), out_s);</code>

The programmer can specify attributes by name or by position. For example:

```
stream int foo(64);           // equivalent to: stream int foo(size=64);
k1(in_s(16, 32), out_s);     // equivalent to: k1(in_s(size=32, offset=16), out_s);
```

The following code further demonstrates the use of attributes.

```
#define IN_LENGTH      256
#define OUT_LENGTH     (IN_LENGTH / 4)
...
    stream int32x1      in_str(size=IN_LENGTH);           // LRF size attribute
    stream int32x1      out_str(size=OUT_LENGTH);
...
    // Load a big buffer into in_str
    spl_load_block(in_str, in_buffer, 0, IN_LENGTH);

    for (i = 0; i < IN_LENGTH; i = i + IN_LENGTH / 4)
    {
        // Use substream to "slide" a window along in_str,
        // processing only 1/4 of the input data at a time.
        k1(in_str(i, IN_LENGTH / 4), out_str);
        spl_store_block(out_str, out_buffer, 0);
        ...
    }
```

5.1.3 Example

A typical sequence of stream operations is as follows:



- Declare streams with constant sizes.
- Load kernel input data from memory into the LRF using a Pipeline API **spi_load_*** function.
- Execute a kernel function. Within the kernel:
 - Read data from an input stream (in the LRF) using a Kernel API **spi_*read** function.
 - Write data to an output stream (in the LRF) using a Kernel API **spi_*write** function.
- Store kernel output data from the LRF to memory using a Pipeline API **spi_store_*** function.

For example:

```
stream int    chicken(16);           // Declare a stream of 16 ints
stream int    meat(16);              // Temporary stream (only exists in LRF)
stream int    nuggets(32);           // Output of kernel sanders
spi_buffer_t   farm, stomach;        // Buffers
int           wallet;               // Decrement by kernel sanders
...
spi_load_block(chicken, farm, 0, 16); // Load buffer farm into stream chicken
colonel(chicken, meat);              // Kernel function - puts result in stream meat
sanders(meat, nuggets, wallet);      // Kernel function - reads data from stream meat
spi_store_block(nuggets, stomach, 0); // Store data from stream nuggets to buffer stomach
```

When program input data is too large to fit into the LRF at one time, a pipeline typically repeats the load/kernel/store sequence within a loop, processing the input in successive portions called *strips*. The program designer must analyse the program's data flow to determine how to map the input efficiently.

The stream size in a stream declaration must be a compile-time constant. The LRF contains **SPI_LRFSIZE** words per lane (4096 on Storm-1). If a pipeline calls a kernel that requires one input stream and one output stream of the same size and requires double buffering for performance (see chapter [Performance optimization](#)), then it needs to declare four streams. Leaving 256 words per lane for local arrays, it has a maximum stream size of $(\text{SPI_LRFSIZE} - 256) / 4$ words per lane (960 on Storm-1), so it can declare four streams of up to size $((\text{SPI_LRFSIZE} - 256) / 4) * \text{SPI_LANES}$ (15360 on Storm-1).

Of course, a program does not need to use all a stream; it can determine the size of stream loads and stores at runtime. Stream arguments to kernel functions or to **spi_load_*** or **spi_store_*** also may use substream attributes to indicate that only a portion of stream should be used; see the [Stream and scalar parameter attributes](#) table above.

5.2 Stream functions

Stream function **spi_count** returns the number of valid data records in a stream. **spi_out** returns the value of a kernel scalar output parameter. Stream functions **spi_load_block**, **spi_load_index**, and **spi_load_stride** load data from a buffer to a stream. Similarly, **spi_store_block**, **spi_store_index**, and **spi_store_stride** store data from a stream to a buffer. Arguments allow the user to specify an access pattern controlling the layout of the data in the LRF; for example, a **spi_load_stride** argument specifies a stride between each group of loaded data records. Subsections below describe block, strided and indexed load/store functions.

5.2.1 Count

spi_count returns the number of valid data records currently in a stream. A stream's record count is undefined when the stream is declared. Writing to an output stream sets the count to the number of records written to the stream. Reading or updating a stream does not change its count. Using a substream (including writing to a substream) does not change the count of the stream.



5.2.2 Block loads and stores

spi_load_block transfers a block of contiguous data records of a given length from a given offset in a data buffer to the LRF. This allows a Stream program running on DSP MIPS to pass input data to a kernel running on the DPU as an input stream. Successive records from the input land in successive lanes in the LRF; the input data is *striped* across the lanes.

Similarly, **spi_store_block** transfers data from the local register file LRF to a contiguous block at a given offset in a data buffer. This allows a Stream program running on DSP MIPS to access data written by a kernel running on the DPU to an output stream. **spi_store_block** uses the current stream count (**spi_count**(*str*) for an ordinary stream *str*, or the substream length for a substream) to determine the number of records to store.

5.2.3 Strided loads and stores

spi_load_stride and **spi_store_stride** are similar to **spi_load_block** and **spi_store_block**, but allow the programmer to specify a more complicated data access pattern for the load or store. Additional arguments supply a number of records per lane, a number of lanes per group, and a stride between successive groups. Rather than loading the LRF with successive records from a contiguous block of memory like **spi_load_block**, **spi_load_stride** can load multiple records to a single lane of the LRF and then skip (stride) to a different block of records.

5.2.4 Indexed loads and stores

spi_load_index and **spi_store_index** are similar to **spi_load_block** and **spi_store_block**, but allow the programmer to specify an index stream that defines the data access pattern for the load or the store. The demo example in the [Demo Application spm_demo](#) chapter below uses an indexed load to allow a kernel to access a block of adjacent pixels in an image, even though the block's pixel data are not adjacent in the input buffer.

5.2.5 Scalar output

A kernel can produce a scalar output as a result. Pipeline API function **spi_out** returns the value of a scalar output variable produced by a kernel. A variable that a Stream program uses as a scalar **out** parameter in a kernel call may only be used as an argument to **spi_out** or as an argument to another kernel call.



6 Kernel API

A kernel function (also called simply a *kernel*) is a function that runs on the stream processor DPU in parallel with Stream code that runs on DSP MIPS. The Stream programming model Kernel API defines kernel functions and kernel intrinsic operations that may be used only within kernel functions. The *Stream Reference Manual* chapter *Kernel API* describes each Kernel API function and intrinsic operation in detail.

6.1 Kernels

A Stream program declares a kernel function with the keyword **kernel** at the start of a function declaration. The syntax of a kernel function declaration is:

```
[ inline ] kernel type name(type name(io_type), ...);
```

Similarly, the syntax of a kernel function definition is:

```
[ inline ] kernel type name(type name(io_type), ...) { block }
```

The *type* of a non-inlined kernel must be **void**; top-level kernels do not return a value. However, an **inline** kernel may return a value with **return**; its *type* may be any DPU basic type (described below), user-defined structure, or vector of basic type or structure.

Kernel functions may call **inline kernel** functions, but may not call non-kernel functions. Kernel functions may use DPU intrinsic operations, described in the [Intrinsic operations](#) section of this chapter. *Stream Reference Manual* gives a complete list of intrinsic operations.

A kernel function called from another kernel function must be declared with the **inline** keyword, and its code is actually inlined: the Stream compiler **spc** inserts a copy of the inlined function code at every site where the function is called. The [Demo Application spm_demo](#) chapter below provides an example of an **inline** kernel.

The table below shows the arguments allowed in a kernel function declaration.

Type	I/O type	Example	Permitted in top-level kernel function?
stream	in, out, seq_in, seq_out, cond_in, cond_out, array_in, array_out, array_io	stream int data(cond_in)	Yes
scalar	in, out	int16x2 pivot(out)	Yes
vector	in, out	vec int8x4 pixel(in)	No
vector array	in, out	vec int8x4 pixels[32](in)	No

If a kernel declaration specifies a scalar **out** parameter, the corresponding actual parameter in the kernel call must be a local scalar variable, not a scalar expression. Outside of the kernel definition, the program may use the scalar variable only as an argument to another kernel call or as the argument to a **spi_out** call.

For example:

```
kernel void sort(int pivot(in),
                 stream int in_str(seq_in),
                 stream int out_str(seq_out));
inline kernel void read_array(stream int16x2 in_str(seq_in),
```



```
vec int16x2 va_out[32](out));
```

read_array can only be called from within another kernel function, because it has a vector array as an argument.

6.1.1 Limitations

Kernel functions have the following limitations:

- *No access to global variables.* The only way to communicate data to a kernel function is through its parameters. A kernel can reference only local (automatic) variables, not globals.
- *No recursion.* A kernel function cannot call itself recursively in any manner.
- *No pointers.* No “address of” operator ‘&’ or indirection operator ‘*’.
- Kernel code can call **inline** kernels, but not non-**inline** kernels. Other Stream code can call non-**inline** kernels, but not **inline** kernels.
- Kernel code can call **inline** kernel functions, kernel library functions, and kernel intrinsics. It cannot call other functions, including standard C functions.
- Kernel code can use only DPU basic types **int**, **int32x1**, **int16x2**, **int8x**, and their unsigned counterparts. Qualified versions of DPU basic types are not allowed. Structures of DPU basic types are permitted, but not in kernel function parameters.
- Only one-dimensional arrays of vectors are permitted. Arrays of scalars and arrays of streams may not be used. One-dimensional arrays of vectors with explicit size declarations may be used as kernel function parameters.
- Supported assignments: *vec = vec*, *vec = scalar*, and *scalar = scalar*, but not *scalar = vec*. Assigning a vector to a scalar is not permitted, as the compiler does not know from which lane to take the value. Instead, use intrinsic **spi_perm** to select a scalar value from a vector; `s = spi_perm32(i, v, 0);` assigns the scalar value from lane **i** of vector **v** to scalar **s**.
- No more than 8 sequential or conditional streams may be passed to a kernel function. Kernel arguments may contain a maximum of 24 array streams or scalar parameters and a maximum of 8 output parameters (sequential output streams, conditional output streams, or scalar outputs).

Some examples:

```
int32x1 i;
vec int32x1 r, v, av[4];
...
i = r;           // Illegal - can't assign vector to scalar
r = i;           // Legal - assigns scalar to vector, same value in every lane
r = av[i];       // Legal - indexing array of vec by scalar
r = av[v];       // Legal - indexing array of vec by vec
```

Vector subscript **v** in the latter example may take on a different value in each lane.



6.2 DPU basic types

Due to DPU hardware restrictions, kernels may use only a limited set of data types. All DPU basic types are 32-bit types, as the DPU operates only on 32-bit words. Standard C types **char**, **unsigned char**, **short**, **unsigned short**, **float**, **double** and pointer types are not DPU basic types; they may not be used in kernels.

Types **int32x1** and **uint32x1** represent a single 32-bit signed or unsigned integer; these types are synonyms for **int** and **unsigned int**.

Packed data types allow the DPU to perform multiple operations simultaneously on a single packed data item. Types **int16x2** and **uint16x2** represent two signed or unsigned 16-bit integers packed into a single 32-bit word. Types **int8x4** and **uint8x4** represent four signed or unsigned 8-bit integers packed into a single 32-bit word. An operation on a packed type performs multiple subword operations simultaneously in a single arithmetic-logical unit (ALU) of the DPU. For example, the following code executes four 8-bit additions at the same time on the DPU.

```
int8x4  a, b, c;
c = a + b; // performs four 8-bit adds simultaneously on one DPU ALU
```

New packed constant types allow for correct constant arithmetic. Suffix **p2** or **p4** appended to a valid C signed or unsigned integer constant of any radix indicates a packed constant type; suffix **p1** appended to a valid C signed or unsigned integer constant of any type is ignored. The Stream compiler **spc** warns about the use of incorrect constant types.

```
int8x4      bar = 0xDEF23008;           // Warning - not p4 constant
int16x2     bar = -27p2;                // Low order 16 bits = -27, hi = -1
unsigned int8x4  foo = 0xFE008023p4 + 0x018A8621p4; // foo = 0xFF8A0644
```

6.2.1 Type conversions

Any *explicit conversion* (cast) from one basic type to another basic type produces the same bit pattern; this is possible because all DPU basic types contain 32 bits. An explicit cast allows any 32-bit object to be used as if it were of any basic type. For example:

```
int32x1 a;
int16x2 b;
a = (int32x1)b; // bit pattern of packed 16x2 b assigned to 32x1 a
```

General rules called *implicit conversions* apply implicitly to operator and function call arguments.

Kernel basic data types fall into signed and unsigned categories. Each category contains three basic types (width variants), interpreting an object of the type as a single 32-bit value, a pair of 16-bit values, or a quad of 8-bit values; the *width* of the type is accordingly said to be 32, 16 or 8 bits. Stream performs implicit conversion from signed to unsigned (as in C). The implicit conversion is to the integral type with the same width as the original type (e.g., from **int16x2** to **uint16x2**). Conversion leaves the bit pattern unchanged; no range checking occurs.

Stream does not perform implicit conversions between objects with different type widths. Operators cannot mix argument widths:

```
int32x1 a, x;
int16x2 b;
x = a + b; // Illegal because + has mixed argument widths
```



6.2.2 DPU booleans

The boolean result of a C relational operator such as ‘==’ is 0 (false) or 1 (true). The boolean result of a DPU intrinsic operation such as **spi_veq*** instead is 0 (false) or all 1 bits (true: 0xFF, 0xFFFF, or 0xFFFFFFFF, depending on the width of the type). Because of this difference, relational operators in kernel functions generate multiple DPU operations, first obtaining the DPU boolean result and then converting it to a C boolean. The programmer can avoid unneeded operations by using DPU intrinsics directly instead of using C operators, although in many cases the Stream compiler **spc** removes unnecessary operations during optimization.

Similarly, conditional operators such as ‘a ? b : c’ in C check whether the control condition is 0 (false) or nonzero (true), while DPU conditional operators such as **spi_vselect*** check only the low bit of the control condition. As a result, ‘?:’ expressions generate multiple DPU operations, first converting the condition from a C boolean to a DPU boolean and then applying the DPU select operation. As with relations, the programmer often can avoid unneeded operations by using DPU intrinsics instead of C operators.

6.3 Scalar and vector variables

A kernel may use two types of variables: scalar variables and vector variables. Like an ordinary C variable, a scalar variable has a single value. A vector variable, declared in a kernel function in a Stream program with the storage class modifier **vec**, has a different value in each lane of the DPU. It may be thought of as an array of size **SPI_LANES**. Vector variables may be declared and used only within kernel functions. A DPU operation on a vector variable operates on all values in the “array” simultaneously, performing the same operation on the data in each lane (in SIMD: single instruction, multiple data). Any kernel function type or structure may be used as the type of a vector variable declaration.

```
typedef struct {
    int16x2      x, y, z;
} xyz;

int32x1      i;      // scalar variable
vec int32x1  v;      // vector variable
xyz          s;      // scalar record
vec xyz      v_s;    // vector record
```

Stream does not support vector variable initializers.

The following table shows examples of vector variables and their use.

Declaration	Use	Type	Description
<code>vec int16x2 e[10];</code>	<code>e[2]</code>	<code>vec int16x2</code>	third value in array e
<code>xyz c;</code>	<code>c.x</code>	<code>int16x2</code>	value of x member of struct c
<code>vec xyz f;</code>	<code>f.x</code>	<code>vec int16x2</code>	value of f.x in each lane
<code>vec xyz g[3];</code>	<code>g[1]</code>	<code>vec xyz</code>	second value in array g
	<code>g[1].y</code>	<code>vec int16x2</code>	value of y member of second value in array g

6.4 Arrays

A kernel may use one-dimensional arrays of vectors, declared as follows:

```
vec int      a[4];
```



If an array is used only with constant indices, then the Stream compiler **spc** may store the array in the operand register file (ORF) within the lane. Otherwise, **spc** will store the array in the LRF.

6.5 Operators

Expressions within a kernel function can use many standard C *operators*. Some operators map to a single DPU intrinsic operation; the [Intrinsic operations](#) section below gives an overview of intrinsics, and the *Stream Reference Manual* gives a detailed description of each intrinsic. For example:

```
vec int16x2 a, b, x;  
x = a + b;           // equivalent to x = spi_vadd16i(a, b);
```

This does just what the programmer expects: it performs two signed 16-bit additions in each lane (one in each halfword of each lane). Because of its SIMD architecture, the DPU performs computations in each lane simultaneously, and therefore the value of a vector expression differs in each lane. This operation requires only a single intrinsic.

Some operators map to multiple intrinsics. For example, the DPU does not support 32-bit by 32-bit multiplication:

```
vec int32x1 a, b, x;  
x = a * b;           // requires multiple DPU operations
```

This generates multiple operations to perform the required multiplication. Similarly, DPU intrinsics for relational operators return all 0 bits or all 1 bits rather than the C relational values of 0 or 1, so

```
vec int16x2 a, b, x;  
x = (a != b);         // requires multiple DPU operations
```

generates multiple operations rather than simply calling intrinsic **spi_vne16**. See section [DPU booleans](#) above for additional information about relational operators and DPU boolean intrinsics.

Kernel functions may not use all C operators. The *Kernel intrinsic functions* section of the *Stream Reference Manual* contains additional details about each operator, including argument type information. Supported operators in kernel expressions include:

Operator	Description	Related intrinsics
.	member extraction	<none>
[]	array subscripting	<none>
+ (unary)	unary plus	<none>
- (unary)	negation	spi_vsub
+ (binary)	addition	spi_vadd
- (binary)	subtraction	spi_vsub
* (binary)	multiplication	spi_vmul*
/	division	spi_vdivstep
%	remainder (modulus)	spi_vdivstep
<<, >>	bitwise shifts	spi_vshift, spi_vshifta



++, --	increment/decrement (prefix and postfix)	<none>
==, !=, <, >, <=, >=	relations	<none>
~	bitwise complement	spi_vnot
& (binary), , ^	bitwise operators	spi_vand, spi_vor, spi_vxor
&&, [N.B.: modified behavior, see below!]	logical operators	<none>
!	logical negation	<none>
? : [N.B.: modified behavior, see below!]	conditional	spi_vselect
(type)	casts	<none>
=	assignment	<none>

In kernel expressions, the ‘&&’ and ‘||’ operators do *not* use the C evaluation “short circuit” rules; the second argument is always evaluated, regardless of the value of the first argument. Similarly, ternary operator ‘?:’ does not use the C evaluation “short circuit” rule; both the second and third arguments are always evaluated, regardless of the value of the first argument. If ‘?:’ arguments contain side effects (e.g., assignments), the result is undefined.

Some C operators are *not* supported within kernel functions. These include:

sizeof	sizeof
->	pointer dereference
& (unary)	address of
* (unary)	indirection

6.6 Control flow constructs

Kernel functions can use most C control flow constructs. In general, a conditional control flow statement must use a scalar control expression to ensure that all lanes follow the same execution path; this is a limitation of SIMD machine architecture.

- if (<scalar_expression>) { ... } is converted to a simple branch with the same control flow in every lane; vector control expressions are not allowed, as control flow must be the same for every lane. If the given block executes only sequential read (**spi_read**) or sequential write (**spi_write**) operations, **spc** generates special code to execute the operations without a branch. This allows the use of code such as:

```
if (<scalar_expression>) { spi_write(s, vi); }
```

within a software pipelined inner loop.

- Looping constructs must only use scalar control expressions:

```
int      i;
vec int  v_i, d[10];
...
for (i = 0; i < 10; ++i) spi_read(in_str, d[i]); // Legal
```



```
while (v_i > 0) { ... } // Illegal - vector expression
```

- A **switch** statement may only have a scalar expression as the switch value.

```
int8x4      value;
vec int16x2 data;

switch (value)
{
case 0:
    spi_read(in_str, data);
    break;

case 1:
    if (data > 12) data = data + 14;
    break;

default:
    break;
}
```

- **goto**, **break**, **continue** and **return** are supported, provided they exist outside of any **if**-statement using a vector expression. **return** with a value is allowed only within an **inline** kernel

```
if (i > 10) return; // Legal
if (v_i != 16) v_i += 16;
else return; // Illegal - if with vector expression
```

6.7 Stream access functions

Kernel functions use Kernel API stream access functions to access stream data. Stream processor hardware supports three different types of stream access from kernel functions: sequential, conditional and array. Sequential access is the most efficient access method, conditional access permits a kernel function to read or write data only to or from selected lanes, and array access permits random stream access.

- **spi_array_read** Read data from an array stream
- **spi_array_write** Write data to an array stream
- **spi_cond_read** Read data from a conditional stream
- **spi_cond_write** Write data to a conditional stream
- **spi_eos** Check for end of stream
- **spi_read** Read data from a sequential stream
- **spi_write** Write data to a sequential stream

To use stream data inside a kernel function, you must pass the stream as a parameter and use stream access functions: you cannot access a data buffer directly. This allows for very high performance execution of kernel functions, in keeping with the architecture of the DPU.

Stream access functions read or write data records. The number of records that can be read from an input stream is determined either from the length of a substream attribute in the kernel function call or from the count of the stream (that is, the number of records written by **spi_load_*** or by a previous call to a kernel function that used the stream as an output).



The kernel function declaration specifies the type and direction of each stream parameter. There are limitations on which combination of stream access functions can be used within a single kernel function. The allowed combinations of stream access functions are shown in the table below.

Type	Stream Modifier	Stream access Functions						
		spi_read	spi_write	spi_cond_read	spi_cond_write	spi_array_read	spi_array_write	spi_eos
Input sequential	in seq_in	✓						✓
Output sequential	out seq_out		✓					
Input conditional	cond_in			✓				✓
Output conditional	cond_out				✓			
Input array	array_in					✓		
Output array	array_out						✓	
I/O array	array_io					✓	✓	

6.7.1 Sequential streams

Sequential streams have the fastest memory performance. **spi_read** and **spi_write** read and write data to and from *all* lanes in a sequential manner. Reading beyond the end of a stream returns zero.

On SP16, three calls to **spi_read** would read 48 records from the LRF, 16 at a time. The records are *striped* across the lanes:

	Lane 0	Lane 1	...	Lane 14	Lane 15
first spi_read call	record 0	record 1	...	record 14	record 15
second spi_read call	record 16	record 17	...	record 30	record 31
third spi_read call	record 32	record 33	...	record 46	record 47

It is possible to conserve space in the LRF by both reading and writing to the same sequential stream in a kernel function. To do this, pass the same stream to the kernel function twice, as both an input stream and an output stream. It is the programmer's responsibility to make sure that the number of reads exceeds the number of writes at any time, otherwise input data may be overwritten, resulting in undefined behavior.

6.7.2 Conditional streams

spi_cond_read reads *conditional input stream* data into a subset of the lanes, based on the value in each lane of a vector flag variable. Similarly, **spi_cond_write** writes *conditional output stream* data from a subset of the lanes, based on the value in each lane of a vector flag variable. As with sequential streams, reading beyond the end of a stream returns zero.

Due to the SIMD structure of the DPU, **spi_cond_read** overwrites the value of the destination variable in all lanes, regardless of the value of the conditional flag variable in the lane. If the conditional read flag is false for a lane, then the value will be a repeat of the last record read from the stream by the conditional read; if no data has been read,



then the value will be zero. It is the programmer's responsibility to ignore data returned by **spi_cond_read** in lanes where the read flag is false.

On SP8, three calls to **spi_cond_read** load 0 to 24 records, depending on the condition flags. For example:

	Lane 0	Lane 1	Lane 2	Lane 3	Lane 4	Lane 5	Lane 6	Lane 7
read flag	true	true	false	true	false	false	true	false
first spi_cond_read call	r0	r1	r1	r2	r2	r2	r3	r3
read flag	false	true	true	false	false	true	false	true
second spi_cond_read call	r3	r4	r5	r5	r5	r6	r6	r7
read flag	true	true	true	false	true	true	true	false
third spi_cond_read call	r8	r9	r10	r10	r11	r12	r13	r13

It is possible to conserve space in the LRF by using the same conditional stream as both an input argument and an output argument in a kernel function. It is the programmer's responsibility to make sure that the number of reads exceeds the writes at any time or input data may be overwritten; otherwise, undefined behavior will result.

6.7.3 Array streams

Array streams have the slowest memory performance. **spi_array_read** and **spi_array_write** read and write data to and from *all* lanes in a random access manner. Stream data can be reread as many times as desired. Note that even though the stream is accessed in an arbitrary manner, multiple values are still read sequentially from the stream into each lane for each call to **spi_array_read**.

	Lane 0	Lane 1	...	Lane 14	Lane 15
spi_array_read (str, dest, 0)	record 0	record 1	...	record 14	record 15
spi_array_read (str, dest, 1)	record 16	record 17	...	record 30	record 31
spi_array_read (str, dest, 2)	record 32	record 33	...	record 46	record 47

Reading or writing beyond the end of the stream results in undefined behavior.

6.8 Intrinsic operations

Kernel intrinsic operations (or simply *intrinsics*) represent Stream processor DPU hardware operations. Stream programs can use intrinsic operations *only* within kernel functions. The programmer can write highly efficient data-parallel DPU programs using intrinsic operations. The *Kernel API Intrinsic Functions* section of *Stream Reference Manual* provides a detailed description of each kernel intrinsic function.

A Stream program uses C function call syntax in kernel code to invoke an intrinsic operation. For example,

```
vec int32x1 va, vb, vx;  
vx = spi_vadd32i(va, vb);
```

adds two vectors of **int32x1** values to produce a vector of **int32x1** results. That is, in each lane of the processor, it adds two **int32x1** values to produce an **int32x1** result. Prefix **spi_** identifies the intrinsic as an SPI-specific



operation; the **v** indicates that the arguments are vectors (not scalars); **add** identifies the operation; and **32i** identifies the **int32x1** signed word variant of the operation.

Some intrinsic operations may also be represented using standard C operators. Binary operator '+' represents addition, as one might expect, so the above example can be rewritten as:

```
vec int32x1 va, vb, vx;
vx = va + vb;           // alternative using binary + operator
```

Many arithmetic operations are available in several width-specific or signedness-specific variants. For example, the addition operation **spi_vadd32i** adds two vectors of signed **int32x1** values, **spi_vadd32u** adds two vectors of unsigned **uint32x1** values, **spi_vadd16i** adds two vectors of packed **int16x2** values, and so on. Some operations are also available in both vector and scalar forms in DPU hardware, for example 32-bit signed addition:

```
int32x1 a, b, x;
x = spi_add32i(a, b);    // scalar intrinsic, not vector
x = a + b;               // alternative using binary + operator
```

Packed data types represent pairs of 16-bit values or quads of 8-bit values. Operations on packed data types perform the same operation on each half-word or byte component of the input in each lane and store the result in the corresponding half-word or byte of the generated output. For example,

```
vec int8x4 va, vb, vx;
vx = spi_vadd8i(va, vb);
```

performs four separate signed 8-bit additions in each lane of the processor using the bytes of **va** and **vb** as arguments and stores a packed word containing four 8-bit results into **vx**. Most operations on packed data perform the same operation on each halfword (for **int16x2** or **uint16x2**) or byte (for **int8x4** or **uint8x4**); intrinsic operation descriptions in the *Stream Reference Manual* apply to each component of a packed object unless otherwise noted.

Some DPU hardware operations return two values; for example, hardware operation **ADDC32** returns a 32-bit sum and a 32-bit carry. These operations have two corresponding intrinsic functions (e.g., **spi_vaddc32**, which returns a sum, and **spi_vaddc32_c**, which returns a carry); the Stream compiler **spc** merges paired calls to these intrinsics into a single hardware operation for efficiency.

All DPU basic types are 32 bits wide and all DPU hardware operations take 32-bit arguments. Arguments to intrinsics should be type compatible with the intrinsic prototype.

6.8.1 Saturation arithmetic

Standard integer arithmetic operations (both signed and unsigned) use standard 2's complement arithmetic, sometimes called *modulo* arithmetic. Some kernel intrinsic operations use *saturation* arithmetic; the page in *Stream Reference Manual* that describes an intrinsic notes whether it uses saturation arithmetic. If a result underflows or overflows the range of representable values for the result data type, saturation arithmetic operations return the minimum or maximum representable value for the type. For example, in one half-word of the 16-bit unsigned integer data type **uint16x2**, 0xFFFE plus 3 overflows the maximum representable 16-bit unsigned integer value 0xFFFF; it returns 1 in normal modulo arithmetic but 0xFFFF in saturation arithmetic.

6.8.2 Fractional arithmetic

The stream processor DPU does not include *floating point* arithmetic intrinsic operations, but it does include *fractional* arithmetic operations. DSP programmers often use fractional arithmetic instead of floating point.



In n -bit fractional arithmetic, a bit pattern that normally represents integer x instead represents fractional value $x / 2^m$, by shifting the implicit binary point (normally to the right of the low-order bit) left by m bits; this is called a $(n.m)$ fractional representation. Since the range of an n -bit signed integer is $[2^{n-1}, 2^{n-1})$, the range of a $(1.(n-1))$ signed fractional is $[-1, 1)$. Similarly, the range of a $(0.n)$ unsigned fractional is $[0, 1)$.

For example, the 16-bit quantity 0x4000 represents $2^{14} = 16384$ as a 16-bit signed integer. Moving the implicit binary point left 15 places (i.e., dividing by 2^{15}), the same bit pattern (binary 0.100 0000 0000 0000) represents $2^{14} / 2^{15} = .5$ in $(1,15)$ signed fractional representation. The same bit pattern also represents $2^{14} / 2^{16} = .25$ in $(0,16)$ unsigned fractional representation.

Because $a / 2^n + b / 2^n = (a + b) / 2^n$ and $a / 2^n - b / 2^n = (a - b) / 2^n$, ordinary 2's complement arithmetic operations can be used to perform fractional addition and subtraction. However, $a / 2^n * b / 2^n = ((a * b) / 2^n) / 2^n$, so ordinary 2's complement multiplication does not work for fractionals; the 2's complement product must be adjusted by an n -bit right shift (multiplication by 2^n) to obtain the correct fractional result.

To avoid loss of precision, a full-precision $2n$ -bit fractional product may be *rounded* to a final n -bit result. For example, in a 16-bit (1.15) signed fractional representation, let x be 0x0180, representing 384/32768 (decimal .1171875). The product $x * x$ (decimal .000137...) is not precisely representable in (1.15) . Shifting the full-precision 32-bit product 0x00024000 right 15 binary places to obtain a (1.15) fractional result produces binary 0000 0000 0100.1, which may be truncated to 0x0004 (decimal .000122...) or rounded up to 0x0005 (decimal .000152...).

Multiplication of fractional times integer to integer is similar to the fractional times fractional to fractional case above: $a / 2^n * b = (a * b) / 2^n$, so the $2n$ -bit product must be adjusted by an n -bit right shift (multiplication by 2^n) to obtain the correct integer result.

The stream processor DPU includes intrinsic operations that support fractional multiplication directly, with multiplication, shifting and rounding in a single operation. The [Multiplication intrinsics](#) section below summarizes the available multiplication intrinsic operations.

6.8.3 Multiplication intrinsics

The DPU hardware supports 27 different multiplication intrinsic operations. These operations fall into 8 separate groups; each group is described in detail on a separate page in the *Stream Reference Manual*, based on the intrinsic name. The following table gives an overview of all multiplication intrinsics, ordered by width.

Width	Ops	Variants	Accumulate	Saturate	Shift/Round
16 * 32 → 48 → 64	spi_vmulha32* spi_vmula32*	i i, ui	add	no	no
16 * 16 → 32 → 32	spi_vmuld16*	i, u, ui	no	no	no
16 * 16 → 32 → 16	spi_vmulha16*	i, u, ui	no	yes	yes
	spi_vmula16*	i, u	add		no
	spi_vmulra16*	i, u, ui	add		yes
8 * 8 → 16 → 16	spi_vmuld8*	i, u, ui	no	no	no
8 * 8 → 16 → 8	spi_vmula8*	i, u	add	yes	no

The Width column shows the width in bits of the product arguments, of the computed product, and of the result of the operation. The number in the operator name always indicates the width of the second argument.



The Variants column shows the supported signedness variants of the operation; **i** for signed times signed, **u** for unsigned times unsigned, **ui** for unsigned times signed. The suffix of the operator name indicates the signedness of its arguments.

The Accumulate column indicates whether the operation is a multiply/add or multiply/subtract operation. Multiply/accumulate operations have **a** or **s** in the operator name.

The Saturate column indicates whether saturation is applied to the result.

The Shift/Round column indicates whether the product is shifted and rounded. Rounding multiplications have '**r**' in the operator name. Shifting and rounding are used in multiplication for fractional arithmetic, as described above.

6.9 `__repeat__`

The `__repeat__` keyword indicates that a block of code should be repeated; its usage is:

```
__repeat__ ( [ varname ] ; count ) { block }
```

Here *count* must be an integer constant expression and the optional *varname* must be a scalar variable name. Each instance of *varname* in *block* is replaced by a current block number between 0 and *count* - 1 in the expanded code.

`__repeat__` may be used in any Stream code, including in kernels. It is particularly useful for coding manually unrolled loops within kernel code.

6.10 `#pragma pipeline`

Software pipelining (SWP) is a VLIW instruction scheduling technique in which a single iteration of a pipelined loop may execute operations from several different iterations of the original loop. Software pipelining can improve the efficiency of scheduled code.

The **pipeline** pragma instructs the VLIW scheduler to attempt to apply software pipelining to an inner loop; it should not be used on non-inner loops. The user should insert the pragma after the opening brace of an inner loop, as follows:

```
    for (i = 0; i < count; i++) {
#pragma pipeline
        ...
    }
```

Software pipelining degrades gracefully: if the scheduler cannot apply software pipelining to the loop, it simply schedules it without pipelining. The use of software pipelining can result in a substantial increase in the amount of time required for **spc** to compile a Stream program.

6.11 `#pragma local_array_size`

By default, **spc** allocates 256 words (1 Kbyte) per lane of LRF to hold the local arrays for a kernel. Pragma **local_array_size** preceding a kernel declaration changes the default value for the kernel. Because **spc** allocates LRF on a per-pipeline basis, the **local_array_size** pragma must be visible during compilation of the Stream pipeline; if kernels and pipelines are compiled from separate sources, it could be in the header that declares the kernel.



For example:

In **foo.h**:

```
#pragma local_array_size(k, 1000 * sizeof(int)); // allocate 4Kb per lane for k
extern void kernel k( ... );
```

In **foo.sc**:

```
void kernel k( ... ) {
    vec int x[1000];
    ...
}
```



7 Demo Application `spm_demo`

This chapter uses a concrete programming example to illustrate the basic concepts of Stream programming. Directory **`demos/spm_demo`** of the Stream distribution contains source code for the demo example. Code fragments in this chapter may differ from the distribution source.

The demo application removes a background color (“green screen”, though the background color need not be green) from an image. It performs the following steps:

- Read a bitmap file (**`.bmp`**) containing an image.
- Find the background color of the image:
 - Subdivide the image into blocks.
 - Compute the average color of each block.
 - Find the most common average block color; this is the background color.
- Replace the background color with a different color.
- Write a bitmap file (**`.bmp`**) containing an image.

The Stream programming model Component API allows the programmer to define components representing modular pieces of the program. Structuring a program to use the Component API encourages abstraction, modularity and encapsulation, as well as allowing the use of vendor-provided application libraries to perform standard tasks. The component version of **`spm_demo`** defines three components, corresponding in obvious fashion to the steps listed above:

- File input component **`file_in`** reads a bitmap (**`.bmp`**) input file containing an image and produces an output buffer containing image data.
- Green screen removal component **`gsr`** takes an image data buffer as input, performs green screen removal, and produces an output buffer containing modified image data.
- File output component **`file_out`** reads an image data input buffer and writes bitmap (**`.bmp`**) file output.

Alternatively, **`spm_demo`** could define four components instead of three, separating background color detection and background color replacement into separate components.

This chapter describes the **`spm_demo`** code in some detail, with an emphasis on its use of stream processor resources and the coding of its components. The following chapters describe how to build and run the demo application from the command line and under the Stream integrated development environment **`spide`**.

7.1 *Testbench main*

For program development purposes, it is often helpful to separate the essential work of a program from the stream programming model component framework. This allows you to build a functional version of a program that runs on a host processor and then a version that runs purely on DSP MIPS (either in simulation or on a hardware device) before you build the full component-based application that runs on System MIPS and DSP MIPS. The **`spm_demo`** source is structured accordingly.

Source file **`testbench/spimain.c`** defines a simple **`spi_main`** for a testbench version of the demo program. The testbench version of **`spm_demo`** does not use components. Instead, its **`spi_main`** function calls functions directly to perform the essential work of the program: read the input file, do the green screen removal, and write the output file. With error checking elided, it just consists of the following steps:

```
spi_buffer_t buffer;
...
buffer = read_bmp_file(argv[1]);    // read from .bmp input file into buffer
```



```
buffer = gsr_pipeline(buffer);      // process input buffer, return output buffer
write_bmp_file(argv[2], buffer);    // write from buffer to .bmp output file
```

Source file **file_io.c** defines the functions **read_bmp_file** and **write_bmp_file** that read and write bitmap files. Stream source file **gsr_pipeline.sc** defines the function **gsr_pipeline** that performs the green screen removal. These functions perform all the work for the testbench version of **spm_demo**.

You might expect these functions to manipulate data in memory (an array). Instead, they use a Stream programming model buffer (type **spi_buffer_t**). The green screen removal Stream code uses kernels that perform data-parallel computations efficiently on the DPU, and using a buffer allows the Stream runtime to handle DSP MIPS cache coherency and DSP MIPS / DPU synchronization issues without requiring explicit user code. In the non-testbench version of **spm_demo**, the file input and file output components run on System MIPS while the **gsr** component runs on DSP MIPS, so passing data between them using memory allocated directly (e.g., statically allocated or allocated using **malloc**) would not work.

Once you have debugged the basic functionality of a program, you can create an application that runs on the target device. For **spm_demo**, you can build the green screen removal component as a program that runs on DSP MIPS and uses the power of the DPU. You also build a System MIPS application that contains the file input component, the file output component, and the application **main** from **components/main.c**, as described in subsequent sections of this chapter. The functions described above perform the critical work of each component, greatly simplifying the port from debugged testbench version to complete component-based application running on stream processor device hardware.

7.2 Data representation

Header **components/bmp.h** defines the format of a bitmap (**.bmp**) file, which consists of a header followed by image data. The image data in the file is in row-major order starting at the bottom left of the image, so the pixel data for an image of size w by h is laid out as follows:

row $h - 1$:	pixel $(h - 1) * w$	pixel $(h - 1) * w - 1$...	pixel $h * w - 1$
...
row 1:	pixel w	pixel $w + 1$...	pixel $2 * w - 1$
row 0:	pixel 0	pixel 1	...	pixel $w - 1$

spm_demo assumes that the bitmap file contains 24-bit RGB color data (that is, 8 bits each of red, green, and blue color data for each pixel).

The background detection algorithm in **gsr_pipeline** subdivides the input image into blocks and computes the average color of each block. It implements the average color computation as a kernel that runs on the DPU. It processes blocks of size **BLOCK_WIDTH** x **BLOCK_HEIGHT**. For efficient DPU implementation, it defines both **BLOCK_WIDTH** and **BLOCK_HEIGHT** to be **SPI_LANES**; therefore it processes 16 x 16 blocks on SP16, 8 x 8 blocks on SP8. Because the DPU only operates on 32-bit words, **spm_demo** pads the 24-bit color data for each input pixel to a 32-bit word (DPU data type **uint8x4**) in the data buffer, adding an unused byte for each pixel. Later, it removes the padding before it writes the bitmap output file.

7.3 Implementation alternatives

The System MIPS, DSP MIPS and DPU of a stream processor run in parallel. This presents many implementation alternatives to the programmer. Device i/o operations must use System MIPS and heavily data-parallel computations should use the DPU for efficiency, but other parts of an application might be implemented on any of



the three processors. The Stream programming model Component API makes it easy for the programmer to experiment with different configurations simply by recompiling with different **spc** options, without source code changes. Performance analysis of different configurations then provides important programmer feedback to guide the implementation.

The data buffer padding in **spm_demo** provides a simple example. The code in **read_bmp_file** reads an input file, allocates a buffer for it, and pads the data from 24-bit RGB data to 32-bit word data in the buffer. Alternatively, it could execute a kernel on the DPU to do the padding, but the code required for the padding is very simple and would not benefit greatly by implementation on the DPU. Another alternative would be to recode the kernels that process the data subsequently to handle unpadded data rather than padded data, but this would result in much less efficient coding of the compute-intensive data-parallel kernels. **spm_demo** does the padding directly in **read_bmp_file** instead.

7.4 Buffer allocation

Function **read_bmp_file** allocates a data buffer and stores image data into the buffer. The size of the allocated buffer is determined by **gsr_pipeline**'s subsequent needs when processing the data. While it is ideologically impure for **read_bmp_file** to know buffer size requirements for later processing, allocating a large enough buffer in **read_bmp_file** ensures that **gsr_pipeline** can process the image without the inefficiency of allocating a larger new buffer and then copying image data to it.

The data buffer includes an associated **bmp_binfo_t** structure with buffer information: a magic number (to identify the buffer as a bitmap image data buffer) and the width and height of the bitmap image. **read_bmp_data** calls **spi_buffer_set_info** to set the buffer information.

Function **gsr_pipeline** reads from the input buffer allocated by **read_bmp_data** and writes to an output buffer. It could use the same buffer and simply update its contents, but instead it allocates a separate output buffer. Using separate buffers allows it to run much faster, as discussed in the [Optimization](#) chapter below.

7.5 Streams

The size of a data buffer is limited only by the amount of shared memory available on the processor, but the LRF of a stream processor is of limited size; on SP16, it contains 4,096 32-bit words in each of the 16 lanes, or $4K \times 16 \times 4 = 256$ Kb total. Application data is often too large to fit in the LRF at one time, so programs often process data in successive pieces.

For example, **spm_demo** processes an image of dimensions *width* x *height* pixels. It pads the 24-bit color data for each pixel to 32 bits for processing by the DPU because the DPU only operates on 32-bit types, so even a small 256×256 image would occupy the entire SP16 LRF (256×256 pixels = 64K pixels = 256 Kb). Therefore, **spm_demo** processes an image of any size by processing it in successive *strips*. All strips in use at one time must fit in the LRF. The strip width need not match the image width; a single strip might contain data from one or many rows of image blocks, depending on the image size. Header **gsr_pipeline.h** defines the size of a strip:

```
/*
 * gsr_pipeline processes the image one strip at a time.
 * A strip must be small enough to fit in the LRF.
 */
#define STRIP_WIDTH          512                /* strip width in pixels */
#define NPIXELS_PER_STRIP    (STRIP_WIDTH * BLOCK_HEIGHT)
                                /* pixels in stream at one time, must fit in LRF */
#define NBLOCKS_PER_STRIP    (STRIP_WIDTH / BLOCK_WIDTH)
```



Here **STRIP_WIDTH** is defined with an arbitrary value, subject to the constraints that it must be a multiple of **BLOCK_WIDTH** and that all streams in use at any one time must fit into the LRF. Function **gsr_pipeline**, defined in **gsr_pipeline.sc**, defines streams:

```
stream uint8x4      in_str(NPIXELS_PER_STRIP);
stream uint8x4      out_str(NPIXELS_PER_STRIP);
stream uint8x4      avg_str(AVG_STR_SIZE);
stream unsigned int idx_str(IDX_STR_SIZE);
```

Input stream **in_str** and output stream **out_str** contain image pixel values, with the 24-bit RGB pixel color data padded to a 32-bit packed word (DPU data type **uint8x4**). **avg_str** is an output stream of block color averages and **idx_str** is an index stream used to load color data in blocks. **gsr_pipeline** uses these fixed-size streams to process an input image of any size, using the streams repeatedly to process successive strips of the image.

gsr_pipeline contains a loop that reads successive strips of blocks in the image, computes the average color of each block in the strip, and stores the block averages. It looks like this:

```
for (i = 0; i < nstrips; i++) {
    /*
     * Load a strip of pixels NPIXELS_PER_STRIP wide into the stream.
     * The index stream makes each BLOCK_WIDTH wide row of an image block
     * fall in a successive lane.
     * The final strip may include unused data at the end.
     * This loop could be double buffered for better performance.
     */
    offset = (((i * STRIP_WIDTH) % width)
              + ((i * STRIP_WIDTH) / width) * width * BLOCK_HEIGHT);
    spi_load_index(in_str,
                  buffer,
                  offset * sizeof(uint8x4),           // offset
                  idx_str,                             // index stream
                  BLOCK_WIDTH,                         // recs_per_lane
                  1,                                   // lanes_per_group
                  NPIXELS_PER_STRIP);                  // count

    /* Find the average color in each block of the strip. */
    gsr_compute_average(in_str, avg_str);

    /* Store the computed block average stream. */
    spi_store_block(avg_str,
                   avg_buffer,
                   i * NBLOCKS_PER_STRIP * sizeof(uint8x4)); // offset
}
```

The actual code in **gsr_pipeline** is like the loop above, but is double buffered for better performance. The [Optimization](#) chapter explains double buffering.

A single block contains data from multiple rows of the image; for example, the block at the lower left of an image contains data from the start of each of rows 0 to **BLOCK_HEIGHT** - 1. The program reorders the image data for the kernel that performs the background color computation with indexed load function **spi_load_index**. The **spi_load_index** call *count* parameter tells it to load an entire strip containing **NPIXELS_PER_STRIP** pixels to the LRF for each call. The *recs_per_lane* parameter **BLOCK_WIDTH** tells it to load data from a complete row of a block (**BLOCK_WIDTH** pixels wide) into each lane of the LRF. The index stream parameter **idx_str** is defined so that data from successive rows of a block, although separated by *width* pixels in the input buffer, loads in successive lanes of the LRF. The *offset* parameter specifies the starting location of each strip.

Next, kernel function **gsr_compute_average** in **gsr_pipeline.sc** loads an entire **BLOCK_WIDTH** x **BLOCK_HEIGHT** block with calls to **spi_read_block**, so it can perform the block average color computation very efficiently. Then **spi_store_block** stores the block averages for the strip to the average buffer.



After **gsr_pipeline** computes the most common block average color (the background color), it replaces the background color with a new color. The replacement can process each pixel independently of neighboring pixels, so it uses **spi_load_block** to load successive strips of pixels (as opposed to strips of blocks); no index stream is needed. The code essentially does the following:

```
for (i = 0; i < nstrips; i++) {
    /*
     * Load the next strip of pixels into the stream.
     * The image pixels may be processed sequentially,
     * so there is no need for an indexed load here.
     * The last strip may include unused data at the end.
     */
    spi_load_block(in_str,
                  buffer,
                  i * NPIXELS_PER_STRIP * sizeof(uint8x4), // offset
                  NPIXELS_PER_STRIP);                      // count

    /* Remove the background. */
    gsr_remove_background(eps_sq,
                          bg_color,
                          NEW_COLOR,
                          in_str,
                          out_str);

    /* Store the updated strip. */
    spi_store_block(out_str,
                   buffer,
                   i * NPIXELS_PER_STRIP * sizeof(uint8x4)); // offset
}
```

The actual code in **gsr_pipeline** is double buffered for better performance.

7.6 Kernels

A kernel performs highly data-parallel operations very efficiently on the DPU of a stream processor. The programmer must decide which parts of an application to implement as kernels. **spm_demo** defines two top-level kernels and one **inline** kernel in **gsr_pipeline.sc**. Kernel **gsr_remove_background** replaces the background color with a new color:

```
/* Replace pixels that have color within eps_sq of bg_color with new_color. */
kernel void gsr_remove_background(unsigned int  eps_sq(in),
                                   uint8x4      bg_color(in),
                                   uint8x4      new_color(in),
                                   stream uint8x4 in_str(seq_in),
                                   stream uint8x4 out_str(seq_out))
{
    vec uint8x4 color;

    while (!spi_eos(in_str)) {
        #pragma pipeline
        spi_read(in_str, color);
        color = (gsr_color_dist_sq(bg_color, color) < eps_sq) ? new_color : color;
        spi_write(out_str, color);
    }
}
```

Scalar unsigned integer input parameter **eps_sq** (“epsilon squared”) gives the square of the tolerated color distance (“epsilon”) between two colors in RGB color space. Scalar packed unsigned byte parameters **bg_color** and **new_color** give the background color and the replacement color. Sequential streams **in_str** and **out_str** are the input and output streams of pixel color data. The code is very simple, as **inline kernel gsr_color_dist** (discussed



below) does most of the hard computational work. It reads a vector of color data (that is, one pixel's color data in each lane) from its input stream, replaces the color with the new color if it is close enough to the background color, and writes the vector of color data to its output stream. The **pipeline** pragma tells the compiler **spc** to apply software pipelining to the loop for efficiency.

Inline kernel **gsr_color_dist_sq** provides an instructive example of the power of data-parallel DPU operations. It computes the square of the distance between two colors in RGB color space:

```
/* Compute the square of the Cartesian distance between colors a and b. */
inline kernel vec int gsr_color_dist_sq( vec uint8x4 a(in),
                                         vec uint8x4 b(in) )
{
    vec uint8x4 d;
    vec uint16x2 phi, plo;

    d = spi_vabd8u(a, b);          /* absolute difference | a - b | in each byte */
    phi = spi_vmuld8u_hi(d, d);
    plo = spi_vmuld8u_lo(d, d);    /* d * d in four 16-bit results */
    return spi_vshuffieu(0x0B0A0100, phi, plo)
        + spi_vshuffieu(0x0F0E0706, phi, plo)
        + spi_vshuffieu(0x0D0C0504, phi, plo); /* 32-bit sum of 16-bit squares */
}
```

Each pixel in the image is a 24-bit RGB color, padded to 32 bits to fit into a packed **uint8x4** word containing four unsigned byte values. **spi_vabd8u** computes the absolute difference $d = |a - b|$ of vector arguments **a** and **b** in each byte of each lane. The two **spi_vmuld8u*** intrinsics represent a single hardware operation that computes $d * d$ as four 16-bit products of 8-byte operands. Three **spi_vshuffieu** operations zero-extend the meaningful 16-bit products to 32 bits; the fourth product contains meaningless padding. Finally, two 32-bit additions add the squares, and the inline kernel then returns the square of the Cartesian distance between the colors.

Kernel **gsr_compute_average** computes the average color of each block in its input stream very efficiently. It reads a sequential input stream that was loaded by a **spi_load_index** call and writes a block average to a conditional output stream.

```
/*
 * Compute the average color of each block in the input stream,
 * spi_load_index puts a row (BLOCK_WIDTH pixels) of an image block in each lane
 * so each while-loop iteration below processes one image block
 * and produces one average on the output stream.
 */
kernel void gsr_compute_average(stream uint8x4 in_str(seq_in),
                               stream uint8x4 avg_str(cond_out))
{
    vec unsigned int r, g, b;
    vec unsigned int color;
    vec int cond;
    unsigned int i;

    cond = (spi_laneid() == 0); /* for conditional write of average from lane 0 */

    while (!spi_eos(in_str)) { /* process one block on each iteration */

        r = 0;
        g = 0;
        b = 0;

        /*
         * Read a block of pixels.
         * Each spi_read call gets data from one column of an image block.
         * Successive calls get data from adjoining columns;
         * the data in each lane is from a single row of the block.
         * Accumulate 32-bit sums of the RGB components in each lane (image row).
         */
    }
```



```

    */
    for (i = 0; i < BLOCK_WIDTH; i += UNROLL) {
        repeat_(; UNROLL) {
            spi_read(in_str, color);
            r += spi_vshuffleu(0x0A0A0A02, color, 0);
            g += spi_vshuffleu(0x09090901, color, 0);
            b += spi_vshuffleu(0x08080800, color, 0);
        }
    }

    /* Sum the RGB components across the lanes (rows of the block). */
    r = spi_vadd32u(r, spi_vperm32(spi_laneid() ^ 1, r, 0));
    r = spi_vadd32u(r, spi_vperm32(spi_laneid() ^ 2, r, 0));
    r = spi_vadd32u(r, spi_vperm32(spi_laneid() ^ 4, r, 0));
#ifdef SPI_DEVICE_SP8
    r = spi_vadd32u(r, spi_vperm32(spi_laneid() ^ 8, r, 0));
#endif

    g = spi_vadd32u(g, spi_vperm32(spi_laneid() ^ 1, g, 0));
    g = spi_vadd32u(g, spi_vperm32(spi_laneid() ^ 2, g, 0));
    g = spi_vadd32u(g, spi_vperm32(spi_laneid() ^ 4, g, 0));
#ifdef SPI_DEVICE_SP8
    g = spi_vadd32u(g, spi_vperm32(spi_laneid() ^ 8, g, 0));
#endif

    b = spi_vadd32u(b, spi_vperm32(spi_laneid() ^ 1, b, 0));
    b = spi_vadd32u(b, spi_vperm32(spi_laneid() ^ 2, b, 0));
    b = spi_vadd32u(b, spi_vperm32(spi_laneid() ^ 4, b, 0));
#ifdef SPI_DEVICE_SP8
    b = spi_vadd32u(b, spi_vperm32(spi_laneid() ^ 8, b, 0));
#endif

    /*
     * rgb now contain 32-bit sums of RGB values over the entire block.
     * Divide by the number of elements (BLOCK_WIDTH * BLOCK_HEIGHT)
     * to compute the average RGB value for the block.
     * Since BLOCK_WIDTH and BLOCK_HEIGHT are always powers of 2,
     * the divide is optimized to a shift.
     */
    r = (r >> (LOG2_BLOCK_WIDTH + LOG2_BLOCK_HEIGHT)) & 0xFF;
    g = (g >> (LOG2_BLOCK_WIDTH + LOG2_BLOCK_HEIGHT)) & 0xFF;
    b = (b >> (LOG2_BLOCK_WIDTH + LOG2_BLOCK_HEIGHT)) & 0xFF;

    /* Pack up the RGB result and write the value from lane 0. */
    spi_cond_write(avg_str, (vec uint8x4)((r << 16) | (g << 8) | b), cond);
}

```

Each iteration of the **while** loop processes a block of the image. Each **spi_read** call reads **SPI_LANES** (equal to **BLOCK_HEIGHT**) pixels of color data from the input stream; because of the **spi_load_index** command that loaded the LRF, the data read into adjacent lanes corresponds to vertically adjacent pixels in the image; that is, each **spi_read** call reads a column of a block. The **BLOCK_WIDTH** successive calls to **spi_read** within the **for** loop reads data from horizontally adjacent pixels in the image; together, the **spi_read** calls in the **for**-loop read one entire block. The loop body can be unrolled for efficiency, as explained in the [Optimization](#) chapter. The three **spi_vshuffleu** calls extract the R, G, and B components from the color data and accumulate sums in each lane, then subsequent **spi_vadd32u** operations sum the R, G, and B sums across the lanes. Shifts convert the sums to averages, and finally a conditional write operation **spi_cond_write** writes the average color of the block to the output stream.

7.7 File input component

Source files **component/file_in.c** and **file_io.c** define the **spm_demo** application file input component, using definitions from header file **component/file_in.h** and **file_io.h**. This section walks through the file input component



source in detail to explain its implementation. The [System MIPS main](#) section below shows the invocation of the file input component by the demo application.

The **file_in** component defines one execution requirement, one command, and one port. The port is the output port where the component pushes the buffer containing its output. The execute function specifies that the component should execute when space is available on the FIFO of its output port connection. The command gives the name of the desired input file.

When an application creates a **file_in** component instance with **spi_instance_new**, the instance starts in the paused state. When the **file_in** component command handler receives a **FILE_IN_CMD_FILENAME** command with the name of the desired input file, it sets the instance to the running state, and the execute function then reads the input file and writes its contents to an output port.

7.7.1 Component definition

Macro **SPI_COMPONENT_NEW** defines a component. It provides a component id, component name, provider name, version number, and five functions (properties, initialization, destroy, execute, and command handler functions). The **SPI_COMPONENT_NEW** call must be at top level, not within a function. In **components/file_in.c**:

```
/* Define the file_in component. */
SPI_COMPONENT_NEW(
    FILE_IN_COMPONENT,          /* Component identifier */
    FILE_IN_NAME,               /* Component name */
    SPI_PROVIDER_SPI,          /* Component provider */
    FILE_IN_DESC,               /* Component description */
    FILE_IN_VERSION,           /* Component version */
    &file_in_properties,         /* Component properties function */
    &file_in_instance_init,     /* Instance initialization function */
    &file_in_instance_destroy,  /* Instance destroy function */
    &file_in_instance_execute,  /* Instance execute function */
    &file_in_instance_cmdhandler /* Instance command-handler function */
)
```

All the functions and structures defined in **components/file_in.c** are local; a programmer accesses the **file_in** component only through the Component API.

The following subsections describe each of the functions referenced in the component definition.

7.7.2 Properties function

Properties function **file_in_properties** sets the properties of the file input component. It defines an output port, a command, and an execution requirement for the component.

```
/* Set component properties for the file input component. */
static
void
file_in_properties()
{
    /* Register ports. */
    spi_port_register("out", "Output port", FILE_IN_PORT_OUT, SPI_PORT_OUT, 1);

    /* Register commands. */
    spi_cmd_register("FILENAME", NULL, FILE_IN_CMD_FILENAME,
        SPI_PAYLOAD_STRING, SPI_PAYLOAD_NULL);
}
```



```

/*
 * Register execution requirements.
 * Instance executes when there is free space on the output port.
 */
spi_exec_req_register("filein_exec_req",
                     "file_in component execution requirement",
                     0, /* ID */
                     SPI_EXEC_PORT_ALLOF,
                     1, /* One port in this requirement */
                     FILE_IN_PORT_OUT);
}

```

The properties function first calls **spi_port_register** to define a single port with ID **FILE_IN_PORT_OUT**. Argument **SPI_PORT_OUT** identifies it as an output port and the final argument indicates that the port allows only a single connection.

The properties function then calls **spi_cmd_register** to define the command **FILE_IN_CMD_FILENAME**. An application passes the name of the input file to the **file_in** component with a **FILE_IN_CMD_FILENAME** command; its payload, of type **SPI_PAYLOAD_STRING**, specifies the name of the input file.

Finally, the properties function calls **spi_exec_req_register** to define the file input component execution requirements. Its arguments specify that the component execute function will execute when a buffer is available on its output port. Because a newly created instance starts in the paused state, the execute function will not execute until the instance receives a **FILE_IN_CMD_FILENAME** command, as explained below.

7.7.3 Instance initialization function

Instance initialization function **file_in_instance_init** initializes an instance of the file input component. It returns a context of type **file_in_context_t**.

```

/* Per-instance context for the file input component. */
typedef struct {
    char *filename;
} file_in_context_t;

/* Initialize a file input component instance. */
static
spi_instance_context_t
file_in_instance_init(void)
{
    file_in_context_t *context;

    /* Create and initialize the context for an instance. */
    if ((context = (file_in_context_t *)malloc(sizeof(file_in_context_t)))
        == NULL) {
        spi_log(SPI_LOG_ERROR, SPI_LOG_LEVEL_ERROR_FATAL,
               "Unable to create context for instance \"%s\"\n",
               spi_get_name());
        spi_set_state(SPI_INSTANCE_STATE_STOPPED);
        return (spi_instance_context_t) NULL;
    }

    context->filename = NULL;

    return (spi_instance_context_t) context;
}

```

If **file_in_instance_init** fails to create the context, it logs a fatal error, stops the component instance, and returns **NULL**. Otherwise, it returns the context with its **filename** set to **NULL**. When the instance subsequently receives a



FILE_IN_CMD_FILENAME command, the command handler stores the name of the input file in the **filename** field of the context.

7.7.4 Command handler function

Command handler function **file_in_cmdhandler** handles a file input component instance command. It processes the **FILE_IN_CMD_FILENAME** command, which is the only command the file input component recognizes.

```
/* file_in component instance command handler function. */
static
void
file_in_instance_cmdhandler(spi_instance_context_t context, spi_cmd_t cmd)
{
    unsigned int    id;
    char *          bp;
    file_in_context_t * p_context;
    const char *     filename;

    id = spi_cmd_get_id(cmd);
    p_context = (file_in_context_t *) context;

    if (id == FILE_IN_CMD_FILENAME) {
        /* Command FILE_IN_CMD_FILENAME. */
        filename = (const char *) spi_cmd_get_payload(cmd);

        /* Fail if there is already an active file. */
        if (p_context->filename != NULL) {
            spi_cmd_send_response(cmd, SPI_RESPONSE_ERRNO_FAIL, NULL, 0);
            spi_cmd_free(cmd);
            return;
        }

        /* Make a copy of the filename in the context. */
        if ((p_context->filename = strdup(filename)) == NULL) {
            spi_cmd_send_response(cmd, SPI_RESPONSE_ERRNO_FAIL, NULL, 0);
            spi_cmd_free(cmd);
            return;
        }

        /* Send an OK response. */
        spi_cmd_send_response(cmd, SPI_RESPONSE_ERRNO_OK, NULL, 0);

        /*
         * An instance starts out in the paused state (SPI_INSTANCE_STATE_PAUSED),
         * so it responds to commands but does not invoke its execute function.
         * Start the instance running so that its execute function will be invoked
         * when its execution requirements are satisfied.
         */
        spi_set_state(SPI_INSTANCE_STATE_RUNNING);
        spi_cmd_free(cmd);
        return;
    }

    /* Unrecognized command. */
    spi_cmd_send_response(cmd, SPI_RESPONSE_ERRNO_UNKNOWN_CMD, NULL, 0);
    spi_cmd_free(cmd);
}
```

If the component instance is already processing an input file, the **filename** of the context is non-**NULL**, so the command handler sends the failure response **SPI_RESPONSE_ERRNO_FAIL** and returns. Otherwise, it copies the input filename passed in the command payload to the **filename** of its context. If the copy fails, it sends failure



response **SPI_RESPONSE_ERRNO_FAIL**. If it succeeds, it sends the success response **SPI_RESPONSE_ERROR_OK**.

All instances start in the paused state (**SPI_INSTANCE_STATE_PAUSED**). When a **file_in** component instance receives a **FILE_IN_CMD_FILENAME** command, it calls **spi_set_state** to set its state to running before it returns. This allows the component execute function to do the work of the component one space for its output buffer is available on the FIFO of its output port.

7.7.5 Execute function

Execute function **file_in_instance_execute** executes a file input component instance.

```
/*
 * Execution function for a file input component instance.
 * This is called when the instance's execution requirements are satisfied,
 * i.e., when space is available on the output port.
 * Read image data from the file into a buffer,
 * then push the buffer to the output port.
 */
static
void
file_in_instance_execute(spi_instance_context_t context)
{
    file_in_context_t *p_context;
    spi_buffer_t      buffer;

    p_context = (file_in_context_t *)context;

    /*
     * If no file is opened, pause the instance so it will not
     * execute until a file is opened.
     */
    if (p_context->filename == NULL) {
        spi_set_state(SPI_INSTANCE_STATE_PAUSED);
        return;
    }

    /* Read .bmp file data into the buffer. */
    if ((buffer = read_bmp_file(p_context->filename)) == NULL) {
        spi_log(spi_get_log(SPI_LOG_ERROR), SPI_LOG_LEVEL_ERROR_FATAL,
            "%s: cannot read bitmap file\n", spi_get_name());
        spi_set_state(SPI_INSTANCE_STATE_STOPPED);
        return;
    }

    /* Push the buffer. */
    if (spi_connection_push(spi_port_get_connection(FILE_IN_PORT_OUT, 0),
        buffer, 0)) {
        spi_log(spi_get_log(SPI_LOG_ERROR), SPI_LOG_LEVEL_ERROR_FATAL,
            "%s: cannot push buffer\n", spi_get_name());
        spi_set_state(SPI_INSTANCE_STATE_STOPPED);
        return;
    }
    spi_log(spi_get_log(SPI_LOG_DEBUG), SPI_LOG_LEVEL_DEBUG, "pushed buffer...\n");

    free(p_context->filename);
    p_context->filename = NULL;
    ...
    spi_set_state(SPI_INSTANCE_STATE_PAUSED);
}
```



If the instance has not received a **FILE_IN_CMD_FILENAME** function, the **filename** in its context will be **NULL**, so the instance calls **spi_set_state** to pause itself. Otherwise, it calls **read_bmp_file** to read file data into a buffer, pushes the buffer to its output port, and pauses itself. Source file **file_io.c** defines the function **read_bmp_file** that reads the bitmap input file and allocates the buffer for input file data; it contains straightforward C code and is not explained here.

7.7.6 Destroy function

Destroy function **file_in_instance_destroy** destroys a file input component instance.

```
/* Destroy a file input component instance. */
static
void
file_in_instance_destroy(spi_instance_context_t context)
{
    file_in_context_t *p_context;

    if ((p_context = (file_in_context_t *)context) != NULL) {
        free(p_context->filename);
        free(p_context);
    }
}
```

If the context argument is **NULL**, **file_in_instance_destroy** does nothing. Otherwise, it frees the context's **filename** and the context.

7.8 File output component

Source file **component/file_out.c** defines the **spm_demo** application file output component, using definitions from header file **component/file_out.h**. The file output component is similar to the file input component, so this section only describes a few important differences between them.

The **file_out** component recognizes two commands: **FILE_OUT_CMD_FILENAME** and **FILE_OUT_CMD_REPORT_WRITTEN**. Command **FILE_OUT_CMD_FILENAME** is like file input component command **FILE_IN_CMD_FILENAME**; it gives the name of the output file. Command **FILE_OUT_CMD_REPORT_WRITTEN** reports when file output is finished. File output component command handler function **file_out_instance_cmdhandler** handles the command. If no previous **FILE_OUT_CMD_REPORT_WRITTEN** command is pending and there is an active output file, it sets the **report_written_cmd** field of its context; the component execute function **file_out_instance_execute** will issue the command response on completion.

```
/* FILE_OUT_CMD_REPORT_WRITTEN */
if (id == FILE_OUT_CMD_REPORT_WRITTEN) {
    /* Fail if there is already an REPORT WRITTEN command pending. */
    if (ocontext->report_written_cmd != NULL) {
        spi_cmd_send_response(cmd, SPI_RESPONSE_ERRNO_FAIL, NULL, 0);
        spi_cmd_free(cmd);
        return;
    }

    /*
     * If no active file, then response immediately that it is
     * written. Otherwise record the command so the response can be
     * sent later by the execute function.
     */
    if (ocontext->fp == NULL) {
```



```

        spi_cmd_send_response(cmd, SPI_RESPONSE_ERRNO_OK, NULL, 0);
        ocontext->report_written_cmd = NULL;
        spi_cmd_free(cmd);
    } else {
        ocontext->report_written_cmd = cmd;
    }
    return;
}

```

The end of the execute function sends the response after it writes the output file.

```

/*
 * If a REPORT WRITTEN command has been received,
 * then reply that the file has been written.
 */
if (ocontext->report_written_cmd != NULL) {
    spi_cmd_send_response(ocontext->report_written_cmd,
        SPI_RESPONSE_ERRNO_OK, NULL, 0);
    spi_log(SPI_LOG_DEBUG, SPI_LOG_LEVEL_DEBUG, "reporting written file\n");
    ...
}

```

7.9 Green screen removal component

File **components/gsr.c** defines the green screen removal component. Its execute function **gsr_instance_execute** calls function **gsr_pipeline**, defined in Stream source **gsr_pipeline.sc**, to perform the work of green screen removal. The execute function is straightforward: it pops a buffer off its input port, processes the buffer with **gsr_pipeline**, pushes the processed buffer to its output port, and pauses itself:

```

/*
 * GSR component instance execute function.
 * Called when execution requirement is satisfied,
 * i.e., when a buffer is available on the input port
 * and space is available on the output port.
 */
static
void
gsr_instance_execute(spi_instance_context_t context)
{
    spi_connection_t in_conn, out_conn;
    spi_buffer_t buffer;

    in_conn = spi_port_get_connection(GSR_PORT_IN, 0);
    if ((buffer = spi_connection_pop(in_conn, 0)) == NULL) {
        spi_log(spi_get_log(SPI_LOG_ERROR), SPI_LOG_LEVEL_ERROR_ASSERT,
            "gsr: pop failed\n");
        spi_set_state(SPI_INSTANCE_STATE_STOPPED);
        return;
    }

    if (gsr_pipeline(buffer) != 0) {
        spi_log(spi_get_log(SPI_LOG_ERROR), SPI_LOG_LEVEL_ERROR_FATAL,
            "%s: buffer processing failed\n",
            spi_get_name());
        spi_set_state(SPI_INSTANCE_STATE_STOPPED);
        return;
    }

    out_conn = spi_port_get_connection(GSR_PORT_OUT, 0);
    if (spi_connection_push(out_conn, buffer, 0)) {
        spi_log(spi_get_log(SPI_LOG_ERROR), SPI_LOG_LEVEL_ERROR_ASSERT,
            "gsr: push failed\n");
        spi_set_state(SPI_INSTANCE_STATE_STOPPED);
    }
}

```



```
        return;  
    }  
    ...  
    spi_set_state(SPI_INSTANCE_STATE_PAUSED);  
}
```

file_in and **file_out** component instances start out in the paused state (**SPI_INSTANCE_STATE_PAUSED**). Their command handlers switch to the running state (**SPI_INSTANCE_STATE_RUNNING**) when they receive a command that specifies an input filename or an output filename. Since the **gsr** component does not respond to any commands, it instead switches to the running state as soon as a new instance is created, by calling **spi_set_state** from initialization function **gsr_instance_init**.

This remainder of this section gives an overview of the green screen removal algorithm. Section [Streams](#) above describes the division of the input image into blocks and strips and the use of **spi_load_index** and **spi_load_block** during image processing. Section [Kernels](#) above describes the kernels that perform data-parallel operations efficiently.

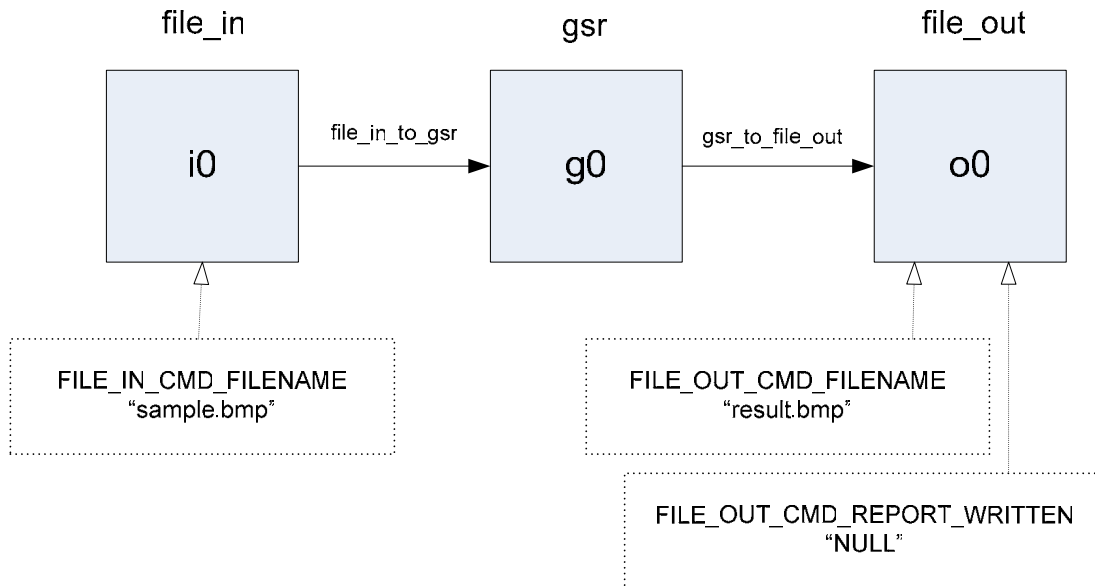
spi_load_index loads pixel data from **buffer** into **in_stream**, using the index stream **idx_stream** generated previously. Kernel **gsr_compute_average** does the computationally intensive data-parallel work of computing the average color in each block, reading input stream **in_stream** and writing output stream **avg_stream** containing block average data. Then **spi_store_block** stores the updated block average data back to **avg_buffer**.

Next, **gsr_pipeline** builds a histogram of the average color information and finds the mode, which gives the background color of the input image.

Finally, **gsr_pipeline** makes another pass over the pixel data, this time calling **gsr_remove_background** to replace any color within a given color distance of the background color with a replacement color.

7.10 Component main

Source file **components/main.c** defines the **main** of the component version of the demo application. This section walks through the source to explain its use of the Stream programming model; error handling code is elided here. **main** loads a DSP MIPS image, creates three component instances, creates two connections between them, and then sends three commands, as shown below.



main first starts the Stream programming model runtime:

```
spi_spm_start("spm_demo", &argc, argv, SPI_SPM_FLAG_NONE);
```

main processes its **argc/argv** after this call, not before, since **spi_spm_start** may adjust **argc/argv**, removing special SPM runtime options from the program's argument list. Then **main** loads a DSP MIPS image, passing it an argument to set the debug log enable mask to **0xf**.

```
char *dsp_argv[3] = { "spm_demo", "--spi_log_mask=debug,0xf", NULL };
if (spi_load_image(SPI_PEL_DSP_MIPS, image,
                  dsp_argv, NULL, SPI_IMAGE_FLAG_NONE) != 0) {
    ...
}
```

It then finds each of the three components required by the application (file input, green screen removal, and file output) and creates an instance of each component.

```
/* Find the file-in, gsr, and file-out components. */
file_in = spi_component_find("spi_example_filein",
                             SPI_PROVIDER_SPI, NULL, NULL);
file_out = spi_component_find("spi_example_fileout",
                              SPI_PROVIDER_SPI, NULL, NULL);
gsr = spi_component_find("spi_example_gsr",
                         SPI_PROVIDER_SPI, NULL, NULL);
...

/* Create one instance of each component. */
i0 = spi_instance_new("in0", file_in);
o0 = spi_instance_new("out0", file_out);
g0 = spi_instance_new("gsr0", gsr);
...
```

Next, it defines the plumbing to connect the components: the file input component output port connects to the green screen reduction component input port, and the green screen reduction component output port connects to the file output component input.

```
/* Connect file in instance to gsr inst. and gsr inst. to file out inst. */
spi_connection_t in_to_gsr = spi_connect("file_in_to_gsr",
    i0, FILE_IN_PORT_OUT,
    g0, SPI_GSR_PORT_IN,
    1 /* depth */);
```



```
spi_connection_t gsr_to_out = spi_connect("gsr_to_file_out",
                                           g0, SPI_GSR_PORT_OUT,
                                           o0, FILE_OUT_PORT_IN,
                                           1 /* depth */);
...
```

To start the ball rolling, the application issues a **FILE_IN_CMD_FILENAME** command to the file input component, passing the input filename in the command payload. If the response to the command is not **SPI_RESPONSE_ERRNO_OK**, the application fails.

```
/*
 * Send the input filename to the file in instance.
 * The FILE_IN_CMD_PAYLOAD is the filename string including the
 * '\0' terminator.
 */
response = spi_cmd_send(i0, FILE_IN_CMD_FILENAME,
                        (void *)infile, strlen(infile) + 1);
if (spi_response_get_errno(response) != SPI_RESPONSE_ERRNO_OK) {
    fatal("setting input filename: %s",
          spi_response_strerror(spi_response_get_errno(response)));
}
spi_response_free(response);
```

Similarly, the application issues a **FILE_OUT_CMD_FILENAME** command to the file output component, passing output filename in the command payload. If the response to the command is not **SPI_RESPONSE_ERRNO_OK**, the application fails.

```
/* Send the output filename to the file out instance. */
response = spi_cmd_send(o0, FILE_OUT_CMD_FILENAME,
                        (void *)outfile, strlen(outfile) + 1);
if (spi_response_get_errno(response) != SPI_RESPONSE_ERRNO_OK) {
    fatal("setting output filename: %s",
          spi_response_strerror(spi_response_get_errno(response)));
}
spi_response_free(response);
```

The application will be done once the file output component has written the output file. The application sends the **FILE_OUT_CMD_REPORT_WRITTEN** command to wait for completion.

```
/* Wait for the file out instance to write its buffer to the output file. */
response = spi_cmd_send(o0, FILE_OUT_CMD_REPORT_WRITTEN, NULL, 0);
if (spi_response_get_errno(response) != SPI_RESPONSE_ERRNO_OK) {
    fatal("waiting for file write: %s",
          spi_response_strerror(spi_response_get_errno(response)));
}
spi_response_free(response);
```

Finally, the application stops the Stream programming model; all the work is done.

```
spi_spm_stop();
```

The following subsection demonstrates the use of an initialization file to greatly simplify the coding of **components/main.c**.

7.10.1 Initialization file

Instead of writing explicit initialization code as described in the previous section, including tedious error checking, the programmer can provide a simple initialization file as described in [Initialization files](#) above. The initialization code in **components/main.c** is conditionalized **#if !defined(INIT_FILE)**. If it is compiled with **spc -D**



INIT_FILE, the source does not use the explicit initialization code, but rather assumes that the user will invoke the System MIPS application with a command line initialization file option:

```
$ ./spm_demo.init.sys.out --spi_init_file=init.xml
```

Initialization file **init.xml** contains statements that load the target image on DSP MIPS, create instances, create connections between instances, and issue initialization commands to instances; see the diagram in the preceding [Component main](#) section.

```
<image target="SPI_PEL_DSP_MIPS" file="spm_demo.dsp.out">
<instance name="i0" component="file_in" provider="SPI_PROVIDER_SPI">
<instance name="g0" component="gsr" provider="SPI_PROVIDER_SPI">
<instance name="o0" component="file_out" provider="SPI_PROVIDER_SPI">
<connection name="file_in_to_gsr" depth="1"
    from="i0:FILE_IN_PORT_OUT" to="g0:GSR_PORT_IN">
<connection name="gsr_to_file_out" depth="1"
    from="g0:GSR_PORT_OUT" to="o0:FILE_OUT_PORT_IN">
<command instance="i0" FILE_IN_CMD_FILENAME="sample.bmp">
<command instance="o0" FILE_OUT_CMD_FILENAME="result.bmp">
<command instance="o0" FILE_OUT_CMD_REPORT_WRITTEN="NULL" />
```

The remaining source in **main** in **components/main.c** is extremely simple: it just calls **spi_spm_start** and **spi_spm_stop**, letting the initialization file processing by **spi_spm_start** do all the work.

The next chapter explains how to build and run the complete **spm_demo** application from the command line.



8 Command line tools

This section introduces the use of the Stream compiler **spc** from a command line to compile programs. It shows how to run the resulting programs on the host PC, under the simulator **spsim**, or on stream processor hardware. Directory **demos/spm_demo/src** in the installed Stream distribution contains the **spm_demo** application sources. *Stream Reference Manual* describes the usage of each Stream tool.

The next chapter describes the use of the Stream integrated development environment **spide** to build programs rather than using the command-line tools. The executables built from the command line in this chapter are in the same locations as the executables **spide** builds.

8.1 Functional mode: Run on host

Compiling a Stream program with **spc** option **-m testbench** builds a *testbench* version of the program. The entry point of a testbench version is **spi_main**, not **main**. A testbench executable starts the Stream programming model runtime before it calls **spi_main**, so the program source does not need to call **spi_spm_start** explicitly.

Compiling a Stream program with **spc** option **-z** builds a *functional* version of the program. A functional executable runs directly on the host PC, not on a stream processor device. Functional mode programs provide quick turnaround for debugging. Programmers typically use functional mode to debug basic program functionality (for example, kernel correctness).

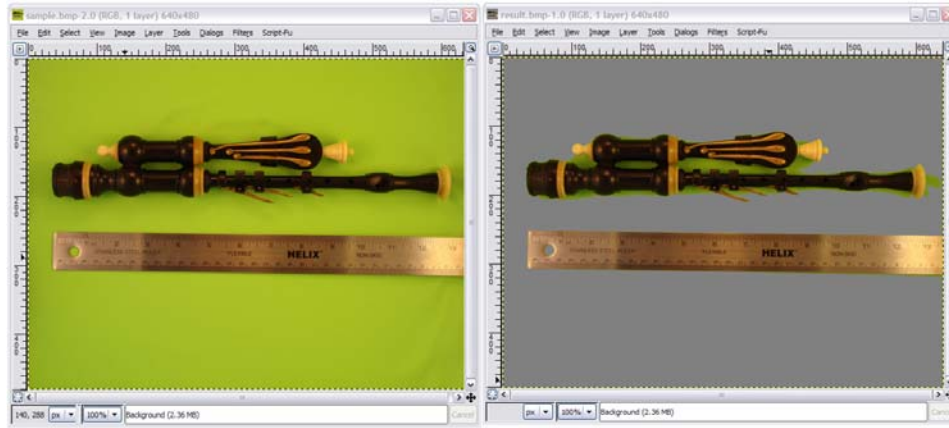
The following command builds an SP16 functional testbench version **testbench** of **spm_demo** from sources **file_io.c**, **gsr_pipeline.sc**, and **testbench/spimain.c**. From the **spm_demo** source directory, type:

```
$ mkdir -p ../build/sp16_functional/bin
$ spc -o ../build/sp16_functional/bin/testbench -m testbench -z \
    file_io.c gsr_pipeline.sc testbench/spimain.c
```

Subdirectory **data** contains a sample bitmap image **sample.bmp**. **spm_demo** takes command line arguments to specify an input file and an output file. To run the functional version of the program:

```
$ ../build/sp16_functional/bin/testbench data/sample.bmp data/result.bmp
```


You can use any image viewer to view the bitmap files. Below, the original image **sample.bmp** is on the left and the image after green screen removal **result.bmp** is on the right. (If you are viewing this page in black and white, it may be difficult to see the difference.)



You can also compile and run a functional version of **spm_demo** for SP8 rather than SP16. Add **-m sp8** to the **spc** command line:

```
$ mkdir -p ../build/sp8_functional/bin
$ spc -o ../build/sp8_functional/bin/testbench -m sp8 -m testbench -z \
    file_io.c gsr_pipeline.sc testbench/spimain.c
$ ../build/sp8_functional/bin/testbench data/sample.bmp data/result.bmp
```

8.2 Simulate with spsim

To build a testbench version of **spm_demo** for execution under the simulator **spsim** or for execution on stream processor hardware, type:

```
$ mkdir -p ../build/sp16_release/bin
$ spc -o ../build/sp16_release/bin/testbench -m testbench \
    file_io.c gsr_pipeline.sc testbench/spimain.c
```

spsim simulates a stream program that runs on DSP MIPS. To simulate the DSP MIPS executable with **spsim**, type:

```
$ spsim ../build/sp16_release/bin/testbench data/sample.bmp data/result.bmp
```

This simulates the file i/o required to read and write a file of roughly 1Mb size, so it takes several minutes to run.

You can compile a testbench version of **spm_demo** for SP8 by using **spc** option **-m sp8**. The **spsim** command to simulate the SP8 program does not require any special flags; it automatically detects that the program runs on SP8.

```
$ mkdir -p ../build/sp8_release/bin
$ spc -o ../build/sp8_release/bin/testbench -m testbench -m sp8 \
    file_io.c gsr_pipeline.sc testbench/spimain.c
$ spsim ../build/sp8_release/bin/testbench data/sample.bmp data/result.bmp
```

8.3 Run on hardware

The testbench executable image built in the preceding section runs on the DSP MIPS of a stream processor hardware device as well as under the simulator **spsim**. Transfer executable image **testbench** and sample bitmap image



sample.bmp to the device filesystem. For example, using **scp** and assuming the device is at IP address **172.18.18.88**:

```
$ scp -p ../build/sp16_release/bin/testbench data/sample.bmp \  
root@172.18.18.88:/tmp
```

Log in to System MIPS Linux running on the device (for example, by using **telnet 172.18.18.88**) and **cd** to the appropriate directory. Then use the **sprun** command to load and execute the DSP MIPS image:

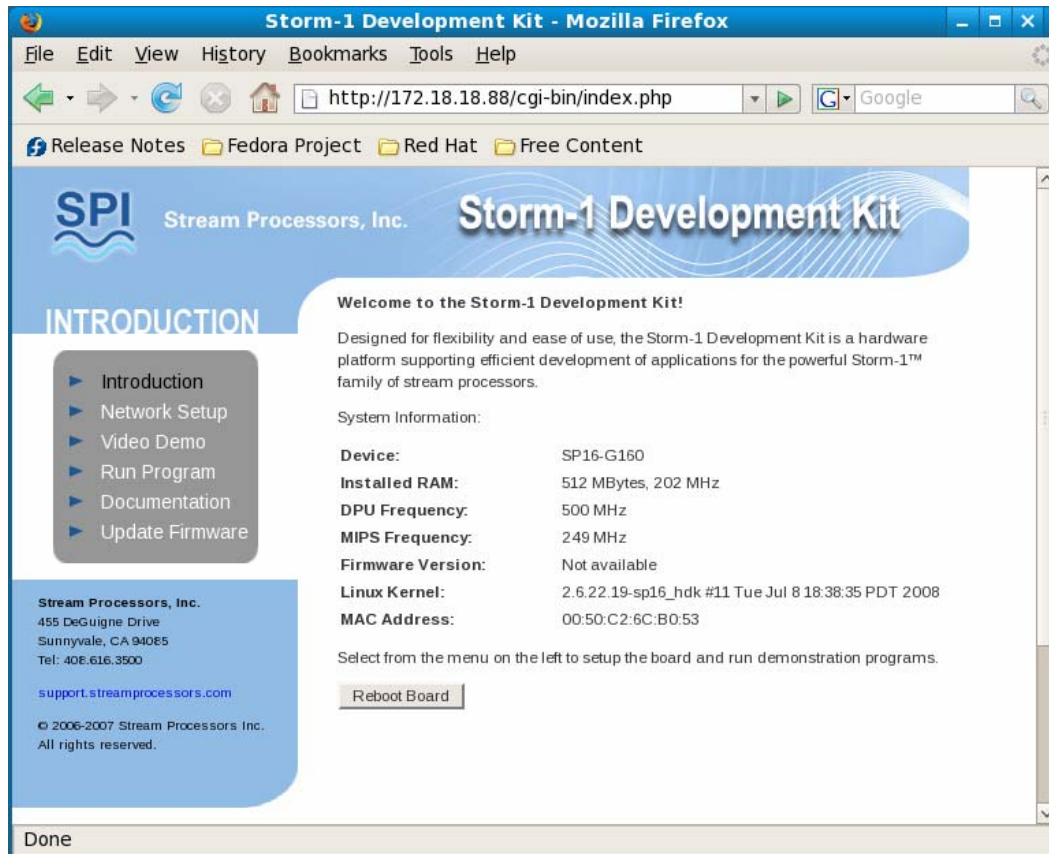
```
$ sprun testbench sample.bmp result.bmp
```

If **sprun** reports an error such as “failed to open...”, reset DSP MIPS with **sprun -s**, then try again.

Similarly, to run the testbench SP8 image on hardware, transfer the SP8 version of **testbench** and **sample.bmp** to the device filesystem, then type the same **sprun** command as above. No special options are necessary; **sprun** automatically detects that the executable is for SP8. An SP16 board can run both SP16 and SP8 executables, but an SP8 board can only run SP8 executables.

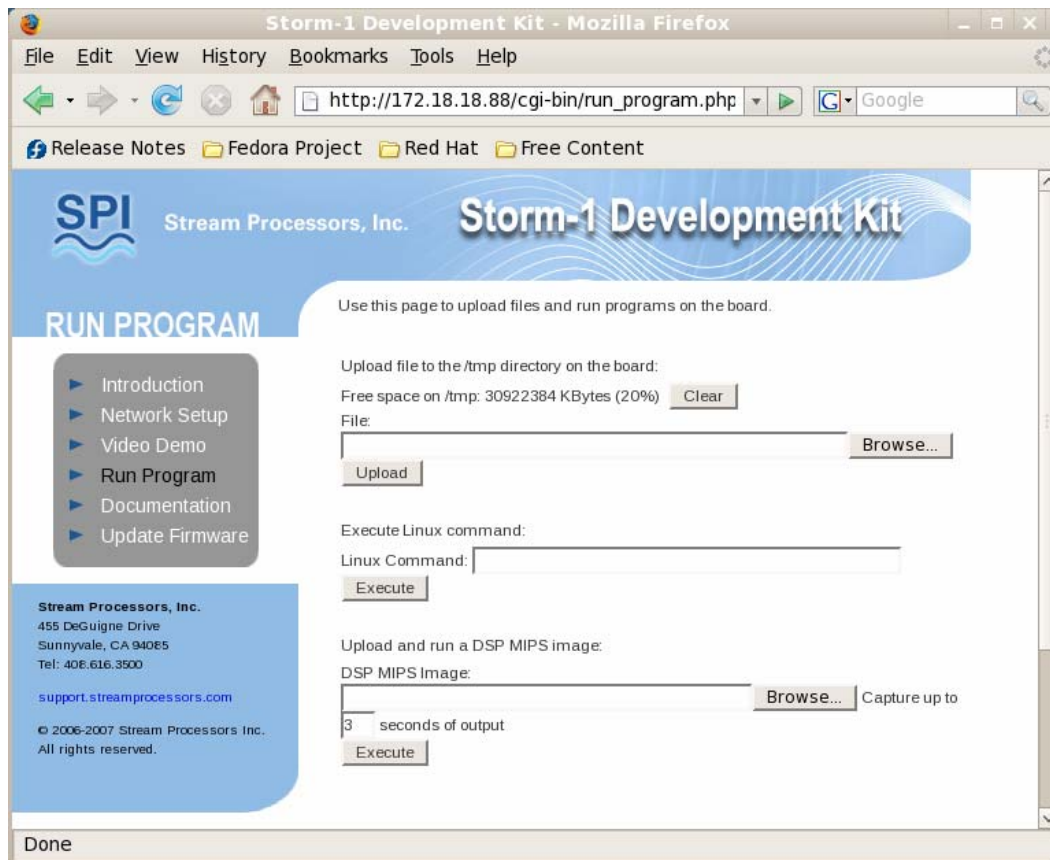
8.3.1 Run from web page

You can use the Stream Processors Storm-1 hardware development kit (HDK) on-board web interface to download and execute a program image. Enter the URL of the development board (for example, **http://w.x.y.z**, where w.x.y.z is the IP address of the board) in the browser address bar to bring up the main on-board web page:



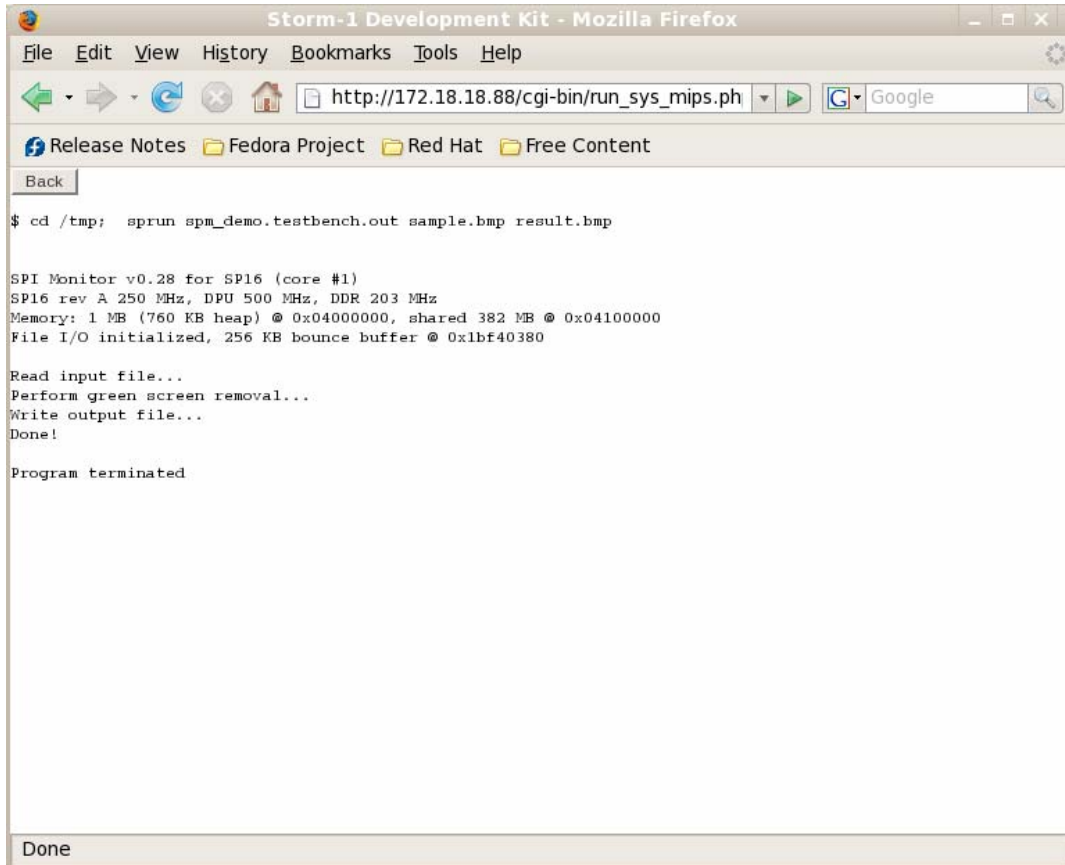


Click on **Run Program** to bring up the Run Program page:



The **Upload and run a DSP MIPS image** option uploads and runs a DSP MIPS program that does not require command line arguments. Since **spm_demo** requires command-line arguments, you should use the **Upload file to the /tmp directory on the board** and **Execute Linux command** options instead.

1. Hit the **Browse...** button for the **Upload file to the /tmp directory on the board** option.
2. Browse to the bitmap input file **sample.bmp** and hit **Upload** to upload it.
3. Browse to the SP16 release DSP MIPS executable **testbench** and hit **Upload** to upload it.
4. Enter the Linux command:
`cd /tmp; sprun testbench sample.bmp result.bmp`
5. Hit **Execute**. The program runs and the browser displays the output captured from **sprun**:

A screenshot of a Mozilla Firefox browser window titled "Storm-1 Development Kit - Mozilla Firefox". The address bar shows the URL "http://172.18.18.88/cgi-bin/run_sys_mips.ph". The browser's content area displays the output of a command executed on a Storm-1 Development Kit. The output starts with a "Back" button, followed by the command "\$ cd /tmp; sprun spm_demo.testbench.out sample.bmp result.bmp". The output then shows a banner from the SPI Monitor, followed by system information, and finally a message from sprun indicating that the program terminated. The browser's status bar at the bottom shows "Done".

```
Storm-1 Development Kit - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://172.18.18.88/cgi-bin/run_sys_mips.ph
Release Notes Fedora Project Red Hat Free Content
Back
$ cd /tmp; sprun spm_demo.testbench.out sample.bmp result.bmp

SPI Monitor v0.28 for SP16 (core #1)
SP16 rev A 250 MHz, DPU 500 MHz, DDR 203 MHz
Memory: 1 MB (760 KB heap) @ 0x04000000, shared 382 MB @ 0x04100000
File I/O initialized, 256 KB bounce buffer @ 0x1bf40380

Read input file...
Perform green screen removal...
Write output file...
Done!

Program terminated

Done
```

The **sprun** output starts with a banner from the SPI Monitor; the monitor performs communication between System MIPS and DSP MIPS. Then it shows the output from **spm_demo**. Finally, it shows a message from **sprun** indicating that the DSP MIPS program terminated.



8.4 Run application on host or on DSP MIPS

The testbench executables in the previous sections did not define or use SPM components. This section shows how to build component-based versions of **spm_demo** and run them on the host or on DSP MIPS. It builds a single executable that contains the **file_in**, **gsr** and **file_out** components.

To build a functional mode version of the component-based application and run it on the host, type:

```
$ spc -o ../build/sp16_functional/bin/app_host -z \
      file_io.c gsr_pipeline.sc components/gsr.c \
      components/file_in.c components/file_out.c \
      components/main.c
$ ../build/sp16_functional/bin/app_host data/sample.bmp data/result.bmp
```

To build a DSP MIPS executable version of the component-based application and simulate it, type:

```
$ spc -o ../build/sp16_release/bin/app_dsp_only \
      file_io.c gsr_pipeline.sc components/gsr.c \
      components/file_in.c components/file_out.c \
      components/main.c
$ spsim ../build/sp16_release/bin/app_dsp_only data/sample.bmp data/result.bmp
```

To run the same DSP MIPS executable on stream processor hardware, transfer **app_dsp_only** and **sample.bmp** to the device filesystem, then type:

```
$ sprun app_dsp_only sample.bmp result.bmp
```

8.5 Run application on hardware

The complete **spm_demo** application consists of a System MIPS executable and a DSP MIPS executable. The System MIPS executable contains the **file_in** and **file_out** components of the application, compiled from sources **file_io.c**, **components/file_in.c**, and **components/file_out.c**. It also contains the **main** function that starts SPM, creates component instances, creates connections, and issues initialization commands to the instances, compiled from source **components/main.c**. **spc** option **-m sp16_sys** builds an SP16 System MIPS executable **app_sys**:

```
$ spc -o ../build/sp16_release/bin/app_sys -m sp16_sys \
      file_io.c components/file_in.c components/file_out.c \
      components/main.c
```

The DSP MIPS executable contains the **gsr** component of the **spm_demo** application. The **gsr** component must run on DSP MIPS, as it contains Stream code that must run on DSP MIPS and kernels that must run on the DPU. The **gsr** component source is in files **gsr_pipeline.sc** and **components/gsr.c**. To build the DSP MIPS executable **app_dsp**:

```
$ spc -o ../build/sp16_release/bin/app_dsp gsr_pipeline.sc components/gsr.c
```

To run the complete application on stream processor hardware, download bitmap file **sample.bmp** and executable files **app_sys** and **app_dsp** to the filesystem of a stream processor device, then log into the board. Make sure the downloaded **app_sys** is executable and type:

```
$ ./app_sys sample.bmp result.bmp
```



Here the System MIPS program loads DSP MIPS image file **app_dsp** by default.

To build the SP8 version of the complete **spm_demo** application, use option **-m sp8_sys** to compile the System MIPS program and option **-m sp8** to compile the DSP MIPS program:

```
$ spc -o ../build/sp8_release/bin/app_sys -m sp8_sys \
      file_io.c components/file_in.c components/file_out.c \
      components/main.c
$ spc -o ../build/sp8_release/bin/app_dsp -m sp8 \
      gsr_pipeline.sc components/gsr.c
```

Then download **sample.bmp**, **app_sys**, and **app_dsp** to the filesystem of a stream processor device, log into the board, make sure **app_sys** is executable, and type:

```
$ ./app_sys sample.bmp result.bmp
```

8.5.1 Initialization file

The System MIPS application **main** in **components/main.c** includes code to load a DSP MIPS image, to create component instances, to create connections, and to issue initialization commands. As described in section [Initialization file](#) above, this code is conditionalized **#if !defined(INIT_FILE)**. If **components/main.c** is compiled with **spc -D INIT_FILE**, the resulting executable expects special SPM option **--spm_init_file=file** to supply an initialization file that specifies the required initialization instead. For example:

```
$ spc -o ../build/spi16_release/bin/app_sys_init -m spi16_sys -D INIT_FILE \
      file_io.c components/file_in.c components/file_out.c \
      components/main.c
$ spc -o ../build/spi16_release/bin/app_dsp gsr_pipeline.sc components/gsr.c
```

To run the application on stream processor hardware using an initialization file, download bitmap file **sample.bmp**, executable files **app_sys_init** and **app_dsp**, and initialization file **components/init.xml** to the filesystem of a stream processor device, then log into the board. Make sure the downloaded **app_sys_init** is executable and type:

```
$ ./app_sys_init --spi_init_file=init.xml
```

The filenames of the DSP MIPS image file, input file, and output file are hardcoded in initialization file **init.xml**, so no other command line arguments to **app_sys_init** are needed.

8.6 Logs

The **spm_demo** application source includes **spi_log** calls that write runtime messages to the built-in **debug** and **error** logs. In addition, the **gsr** component calls **spi_log_new** to create a log named **gsr**. By default, SPM runs programs with all error log levels enabled (**error** log enable mask 0xFFFFFFFF) and all debug log levels disabled (**debug** log enable mask 0). It also disables all user-defined logs (**gsr** log enable mask 0).

The following examples use the functional version of **spm_demo** to demonstrate logging. When you run the functional mode executable **testbench**, it prints the following output:

```
$ ../build/spi16_functional/bin/testbench data/sample.bmp data/result.bmp
Read input file...
Perform green screen removal...
Write output file...
Done!
```




The messages here are from **printf** calls in **testbench/spimain.c**. If you run **testbench** with **gsr** logging enabled, it prints additional information during **gsr** execution:

```
$ ../build/sp16_functional/bin/testbench --spi_log_mask=gsr,1 \
  data/sample.bmp data/result.bmp
Read input file...
Perform green screen removal...
gsr::__spi_testbench__ (1222899576285671936 ns): gsr_pipeline:
gsr::__spi_testbench__ (1222899576292269056 ns): Find background color...
gsr::__spi_testbench__ (1222899577647095808 ns): Background color: abbe20
gsr::__spi_testbench__ (1222899577647770880 ns): Replace background color...
Write output file...
Done!
```

Within the **spm_demo** components, some conditions generate output to the **debug** log rather than to the **error** log; this allows a component to report an error but continue execution. For example, suppose the user specifies the input **.bmp** file with a nonexistent filename:

```
$ ../build/sp16_functional/bin/testbench foo.bmp data/result.bmp
Read input file...
read bitmap file "foo.bmp" failed
```

These messages are from **printf** statements in **testbench/spimain.c**. For additional detail, enable **debug** level 1:

```
$ ../build/sp16_functional/bin/testbench --spi_log_mask=debug,1 \
  foo.bmp data/result.bmp
Read input file...
debug::__spi_testbench__ (1222900070227181056 ns): cannot open "foo.bmp" for
  reading
read bitmap file "foo.bmp" failed
```

8.7 Timers

The [Timers](#) section of the [Component API](#) chapter above describes built-in timers. **spm_demo** includes calls to print timer data from built-in timers and from a timer defined in **gsr_pipeline**. Run the app version of **spm_demo** on stream processor hardware to see some built-in timer data:

```
$ ./app_sys sample.bmp result.bmp
...
Startup:                8501397 ns
DSP MIPS load:          40701437 ns
Total:                  863282282 ns
Done!
```

The times shown here are from built-in timers **SPI_TIMER_STARTUP**, **SPI_TIMER_LOAD_DSP**, and **SPI_TIMER_SPM**, each printed at the end of **components/main.c/main**. To reduce execution overhead in release versions, SPM updates timers **SPI_TIMER_CMDHANDLER** and **SPI_TIMER_EXECUTE** for each component only in debug mode or profile mode. To see additional timer data, recompile **app_sys** and **app_dsp** in profile mode (with **spc** option **-p**), then run the profiling version on hardware:

```
$ ./app_sys sample.bmp result.bmp
...
file_in cmd handler:    223015 ns
file_in execute:        204558486 ns
gsr cmd handler:        0 ns
gsr execute:            16611011 ns
file_out cmd handler:   257153 ns
file_out execute:       680860124 ns
Startup:                26330698 ns
```




```
DSP MIPS load:      56076100 ns
Total:             993550214 ns
Done!
```

The **gsr** command handler time is 0 here because the **gsr** component does not handle any commands.

gsr_pipeline also defines a **gsr** timer, used to measure GSR performance. It prints timer data to the **gsr** log if the log's enable mask is set accordingly. To see this GSR performance data, add option **--spi_log_mask=gsr,2** to the program's argument list.

8.8 Performance

You can use **spperf** to evaluate the performance of a Stream program simulated by **spsim**. The *Stream Reference Manual* describes how to run **spperf**. To generate performance information using **spperf**:

- Add **spi_trace_start** and **spi_trace_stop** calls around regions of interest in the source.
- Compile with **spc -p** (profile mode).
- Simulate with **spsim**, using option **--spi_trace_file** after the DSP MIPS image argument.
- Run **spperf** to generate performance information in an HTML file from the profile information.

Source **gsr_pipeline.sc** already contains **spi_trace_start** and **spi_trace_stop** calls in **gsr_pipeline**. Type:

```
$ mkdir -p ../build/sp16_profile/bin
$ spc -o ../build/sp16_profile/bin/testbench -p -m testbench \
  file_io.c gsr_pipeline.sc testbench/spimain.c      # compile
$ spsim ../build/sp16_profile/bin/testbench --spi_trace_file=spm_demo.sbt \
  data/sample.bmp data/result.bmp                    # simulate
$ spperf ../build/sp16_profile/bin/testbench \
  spm_demo.sbt spm_demo_tcs.sbt                       \
  -o spm_demo.html                                   # analyze
```

You can open the generated HTML file in any browser. The [Optimization](#) chapter gives much more detailed information about performance, including a description of the tables in the HTML file.



9 Stream Program Development

The preceding chapters introduced basic notions of Stream programming and showed how to use command line tools to compile and run a Stream program. This chapter describes Stream program development using the Stream Processors integrated development environment (IDE), called **spide**. You can use **spide** to edit, build, execute, and debug Stream programs. **spide** is based on the Eclipse extensible open development platform. www.eclipse.org gives general information about Eclipse.

Stream program development under **spide** typically steps through a series of development modes:

- *Functional* mode simulates a program functionally on the host PC. This provides quick feedback as the developer debugs basic program correctness, but does not accurately simulate program performance.
 - uses **spc -z**
- *Profile* mode allows the developer to monitor and improve program performance through the use of stream command traces.
 - uses **spc -p**
- *Release* mode creates a release version of the debugged and optimized program.
 - uses **spc** with no **-z**, **-g** or **-p** option

The programmer uses functional mode to create a functionally correct version of an application, uses profile mode to evaluate its performance, modifies the program based on the performance data, and then repeats this cycle as needed. Once the programmer is satisfied with the result, release mode produces a final version of the program.

spide provides two additional modes that are used less frequently:

- *Fast functional* mode simulates an optimized version of the program functionally on the host. This gives better performance than functional mode, but the optimized program is harder to debug. Program development often skips this step.
 - uses **spc -z -On**
- *Debug* mode simulates the device executable in the IDE, allowing debugging of device-specific issues. Program development can skip this step unless functional mode and execution on the device produce different results.
 - uses **spc -g**

This chapter uses the **spm_demo** demo program in directory **demos/spm_demo** of the Stream distribution as a concrete example. It describes the use of **spide** to build and run **spm_demo** in each of the modes described above.

9.1 Invoke *spide*

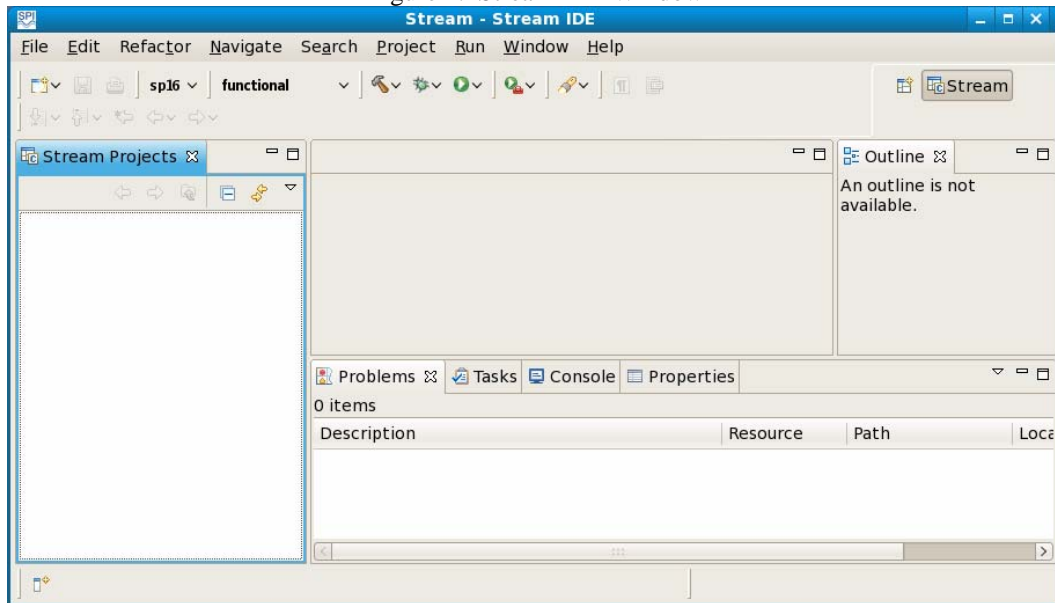
Make sure your **PATH** includes the Stream distribution **bin** directory before you invoke **spide**. Then invoke **spide**:

```
$ spide &
```




spide keeps information in a directory called a *workspace*. **spide** uses the workspace as the default location for any project files you create. You can have multiple workspaces and switch workspaces within the IDE. Each workspace contains one or more *projects*, where a project is a group of related files (sources, binaries, and data). Each project defines one or more *modules*; a module represents an executable or library built by **spide** from the source files in a project. When you build a project with multiple modules, the IDE builds each module in the project.

The first time you invoke **spide**, it displays a banner and then asks you to select a location for your workspace. After you specify a workspace directory, **spide** displays an empty IDE window (Figure 1):

Figure 1: Stream IDE Window



This window shows the Stream perspective, as indicated near the upper right corner. A *perspective* is an editor and a group of views that together provide a development environment. You can use the Stream perspective to edit, build, and run Stream applications. A *view* is a pane within a perspective. The Stream perspective above contains a Stream Projects view on the left, an Outline view on the right, and several additional views at the bottom. You can right-click on the header of a view to maximize, minimize, or detach it. The *editor* is the area of a window not occupied by a view.

Hovering over any icon on the IDE toolbar produces pop-up help information. The IDE toolbar contains pull-down selections for architecture (**sp16** or **sp8**) and mode (**functional**, **functional_fast**, **debug**, **profile**, or **release**), as well as pull-down icons for build (the hammer ) , debug (the bug ) , and run (the green right arrow ) . Other toolbar icons allow you to create a new project, to save the current project, and to print.



The Stream Projects view in Figure 1 above is initially empty, as the workspace does not yet contain any projects. After you import or create a Stream project, the Stream Projects view shows its structure. The project's structure generally corresponds to the **spide** workspace directory structure, but it may contain additional “virtual” folders that do not exist in the workspace. A project contains the following top-level folders:

- **Archives** is a virtual folder that contains a shortcut to every archive in the project. It does not exist if the project contains no archives.
- **Binaries** is a virtual folder that contains a shortcut to every executable image in the project. It does not exist if the project contains no binaries.
- **Includes** is a virtual folder that contains subfolders representing each of the system include paths configured for the project. You can use these subfolders to access system header files.
- **build** contains the build artifacts (objects, executables, static libraries, and trace information for **profile** mode execution) for the modules in the project. At the top level, it contains a subfolder for each type of build done for the project; the build type is a combination of architecture and mode, such as **sp16_functional**. Each build subfolder contains additional subfolders:
 - **bin** contains binaries for modules that generate executables.
 - **lib** contains libraries for modules that generate libraries.
 - *module_name* contains objects and dependency files.
 - **profile** contains Stream trace files for **profile** mode simulations.
- **include** contains external header files for the project, i.e., header files that describe the exported interface of a static library. If a module in project A depends on a module in project B, compilations of project A will include the project B **include** folder automatically.
- **modules** contains information about the modules in the project.
- **src** contains source files for the project. Within the **src** folder, sources and headers can be arranged in any folder hierarchy.

If you delete a project, be sure to delete the project directory from your IDE workspace. Deleting a project from the Stream Projects view does not delete the associated project files.

9.2 Create a project

This section uses demo program **spm_demo** as an example of how to create a new Stream project with **spide**. Directory **demos/spm_demo/src** of the Stream distribution contains the demo program sources.

The **spm_demo** directory in the Stream distribution also contains a pre-built **spide** project. If you wish to experiment with **spide** without repeating the steps below to create a new project, you can use the instructions in the [Import existing project](#) section to import the existing **spm_demo** project instead; then you can build and run the **spm_demo** project immediately.

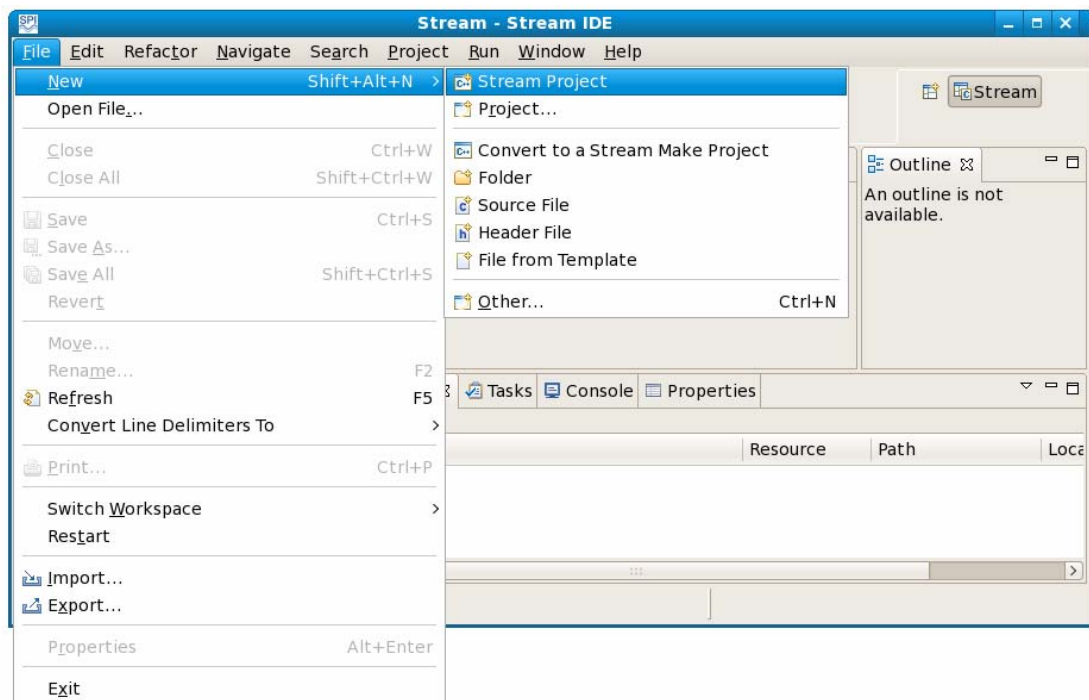
9.2.1 Create Stream project

First, you need to create a new Stream project. Make sure there are no spaces in your project name, as spaces in pathnames can cause problems for some of the tools **spide** invokes.

To create a new Stream project called **spm_demo** in the IDE:

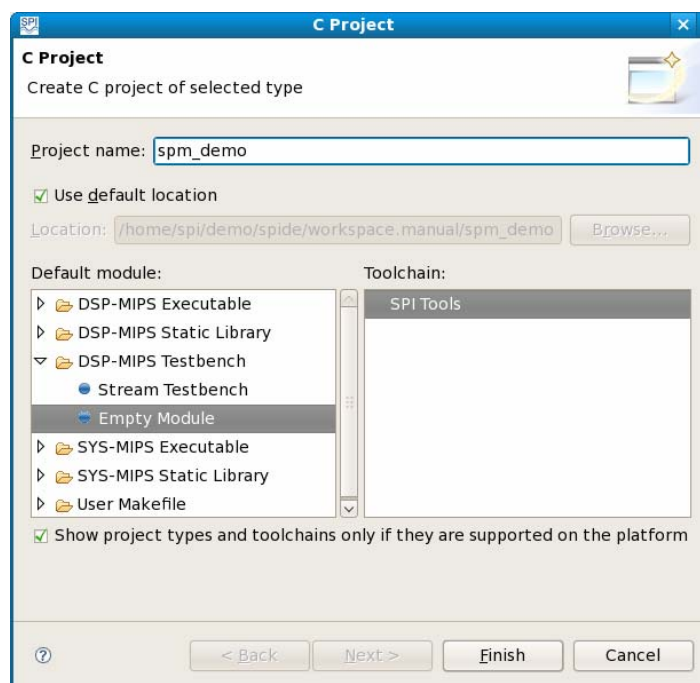
1. Select **File >> New >> Stream Project** (Figure 2). [Alternatively, pull down the **New** icon on the toolbar and select **Stream Project**.]

Figure 2: Create Stream Project



2. Enter **spm_demo** as the **Project name**.
3. Expand **DSP-MIPS Testbench** and select **Empty Module** (Figure 3).
4. Hit **Finish**.

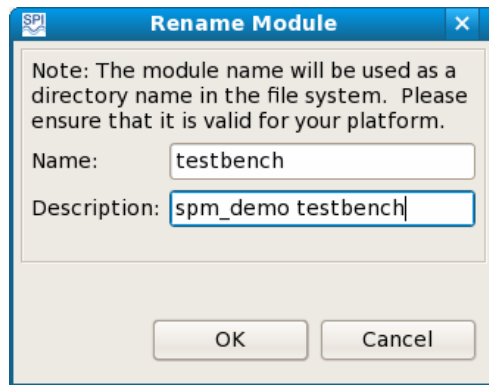
Figure 3: Create Empty DSP MIPS Testbench Module



By default, the new empty module inherits project name **spm_demo** as its module name. This would be fine for a project with a single module. But this chapter will later add additional modules to the project, so to avoid confusion you should rename this module as **testbench**:

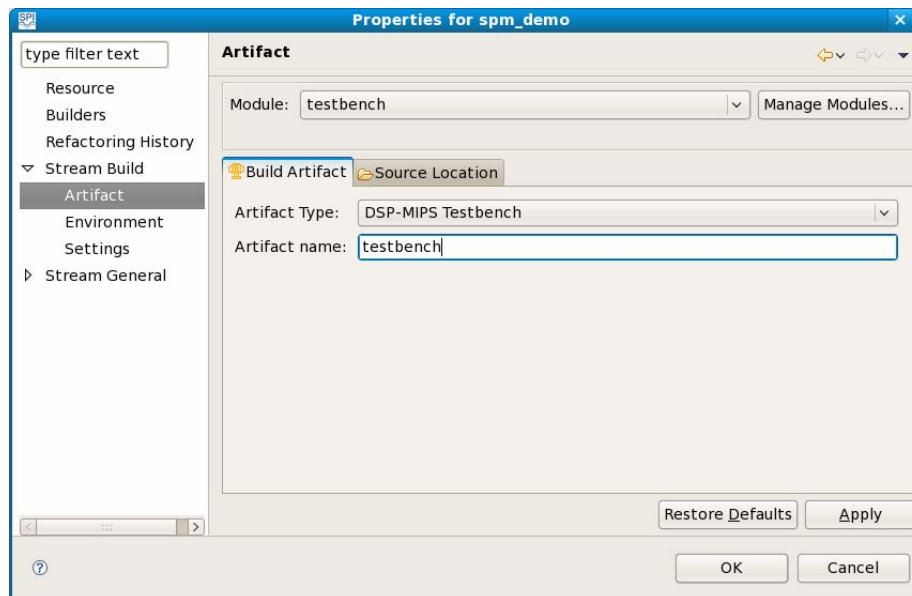
1. Select **Project >> Properties**.
2. Expand **Stream Build** and select **Artifact**.
3. Click **Manage Modules...**
4. Click **Rename** for module **spm_demo**.
5. Enter **testbench** as the new **Name** (Figure 4).
6. Enter a description.
7. Hit **OK** to close the **Rename module** window.
8. Hit **OK** again to close the **spm_demo: Manage modules** window.

Figure 4: Rename Module



9. In the Properties window **Build Artifact** tab, change the **Artifact name** to **testbench** (Figure 5). When **spide** builds the project, the result of building this module will be named **testbench**.
10. Hit **OK**.

Figure 5: **spm_demo** Properties

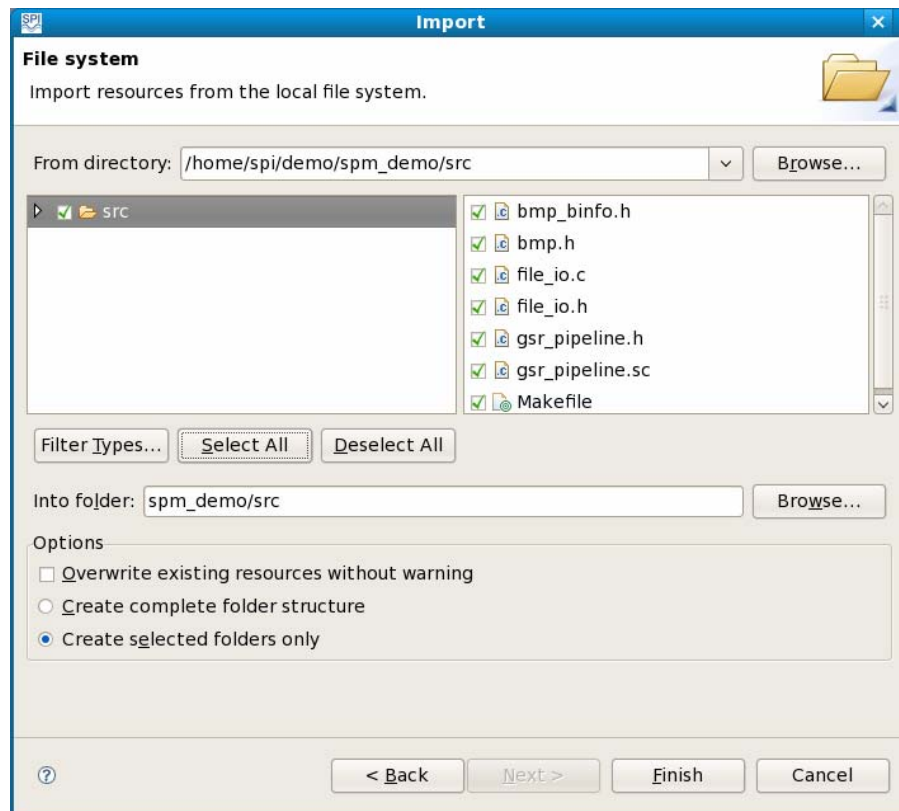


9.2.2 Import source files

The previous steps created a **testbench** module for project **spm_demo**, but the module project does not yet contain any source files. Next, you need to import **spm_demo** source files into the **src** directory of the project.

1. Expand **spm_demo** in the Stream projects view, right-click on **src**, and click **Import**.
2. In the **Import** pop-up window, expand **General** and double-click **File System**.
3. Specify the directory containing the **spm_demo** source files: either click **Browse...** and navigate to the source directory or type the source path in the **From directory** box. For example, browse to or type in **/opt/spi/Stream220/demos/spm_demo/src** for an installation in **/opt/spi/Stream220**.
4. Click **Select all** (Figure 6).
5. Click **Finish** to import the source files.

Figure 6: Import Files



The IDE copies imported files into its workspace. If you subsequently use the IDE to change a file, the copy in the workspace changes, but the file at the original location remains unchanged.

9.2.3 Create testbench module

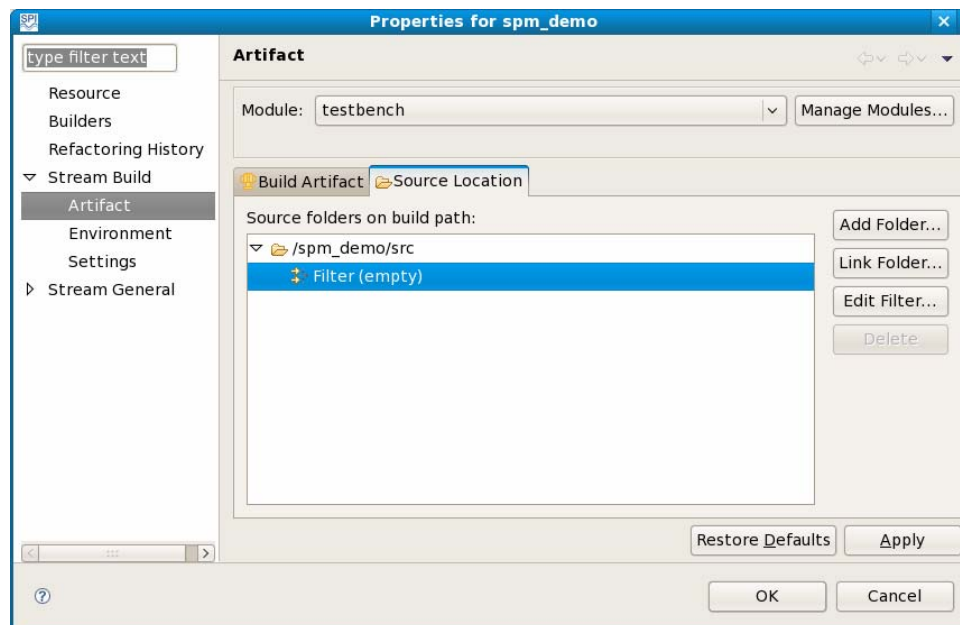
After the IDE imports the source files, the **spm_demo** project includes all the source files from the **spm_demo** directory. The following table shows which **spm_demo** source files are included in the testbench version and in the System MIPS and DSP MIPS images of the complete application. This table does not show other files in the **spm_demo** directory, such as header files and bitmap data file **data/sample.bmp**.

Source file	Description	Testbench	System MIPS	DSP MIPS
file_io.c	bitmap file i/o	x	x	
gsr_pipeline.sc	GSR	x		x
components/file_in.c	file input component		x	
components/file_out.c	file output component		x	
components/gsr.c	GSR component			x
components/main.c	component-based main		x	
testbench/spimain.c	testbench spi_main	x		

Module **testbench** builds a DSP MIPS testbench executable from a subset of the **spm_demo** source files. Some sources are not needed when building the **testbench** module, but will be used later to build modules for the complete application. This section configures the **testbench** module to build the DSP MIPS executable: it defines a *filter* to eliminate source files in the source directory that are not used to build the **testbench** module. Configure the **testbench** module as follows:

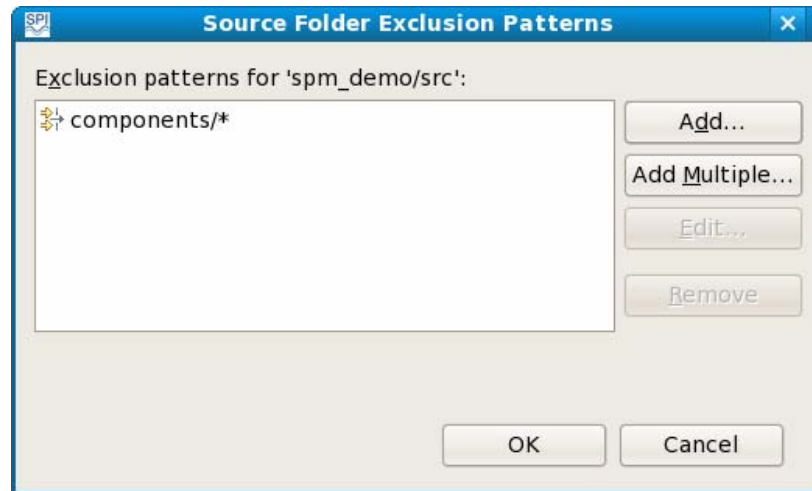
1. In the Stream Projects view, right-click on **spm_demo** and select **Properties**. [Alternatively, select **Project >> Properties**.]
2. Expand **Stream Build** and select **Artifact**.
3. Click the **Source Location** tab.
4. Expand **/spm_demo/src**.
5. Select **Filter (empty)** (Figure 7).

Figure 7: Properties for **spm_demo**



6. Click **Edit filter...** This opens a **Source Folder Exclusion Patterns** window.
7. Add all the files in subdirectory **components** as an exclusion pattern: click **Add...**, type **components/***, and then hit **OK** (Figure 8). [Alternatively, you could click **Add Multiple...** and then select specific source files to exclude from the **testbench** module.]

Figure 8: Source Folder Exclusion Patterns



8. Hit **OK** to close the Source Folder Exclusion Patterns window.
9. Hit **OK** again to close the Properties window.

You only need to exclude source files that should not be included in the build. The **spm_demo** directory contains subdirectory **data** with bitmap file **sample.bmp**, but since it is not a C or Stream source file, you do not need to exclude it explicitly.

9.3 Functional mode

This section describes how to build, run, and debug the **spm_demo** project **testbench** module in functional mode.

9.3.1 Build

To build the **testbench** module in functional mode:


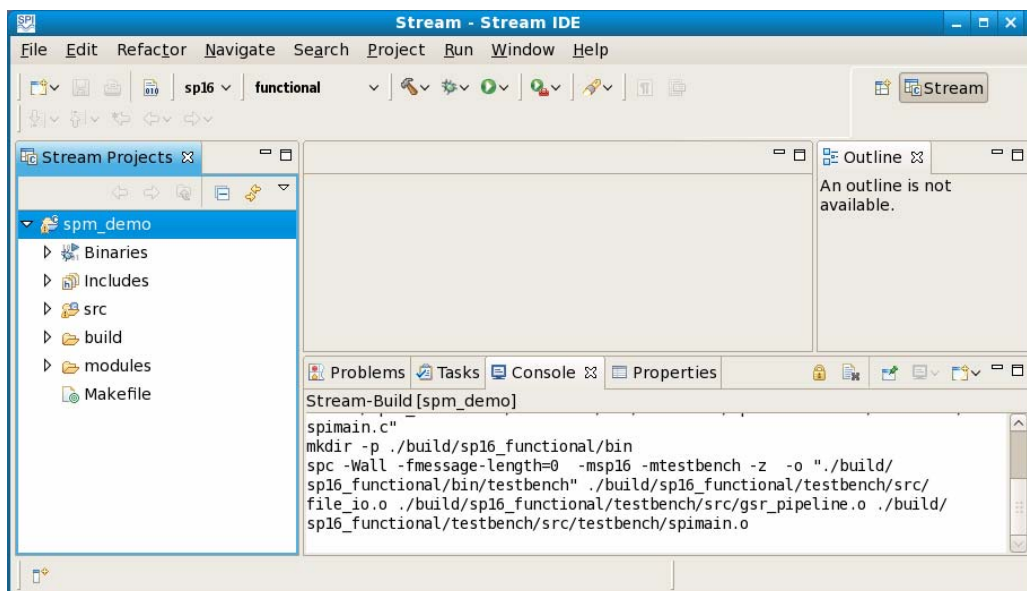
1. Select architecture **sp16** and mode **Functional** on the toolbar.
2. Click the arrow to the right of the build icon (the hammer)  and select **spm_demo** from the drop down options. The IDE displays build commands in the Console view as it runs the build (Figure 9). New folders **Binaries**, **Includes**, and **build** appear in the Stream Projects view.

Figure 9: Functional Mode Build



9.3.2 Run on host

After successfully building the module, you must create a run configuration to allow you to run it. For the **spm_demo testbench** module, the run configuration specifies the locations of the bitmap input and output files.


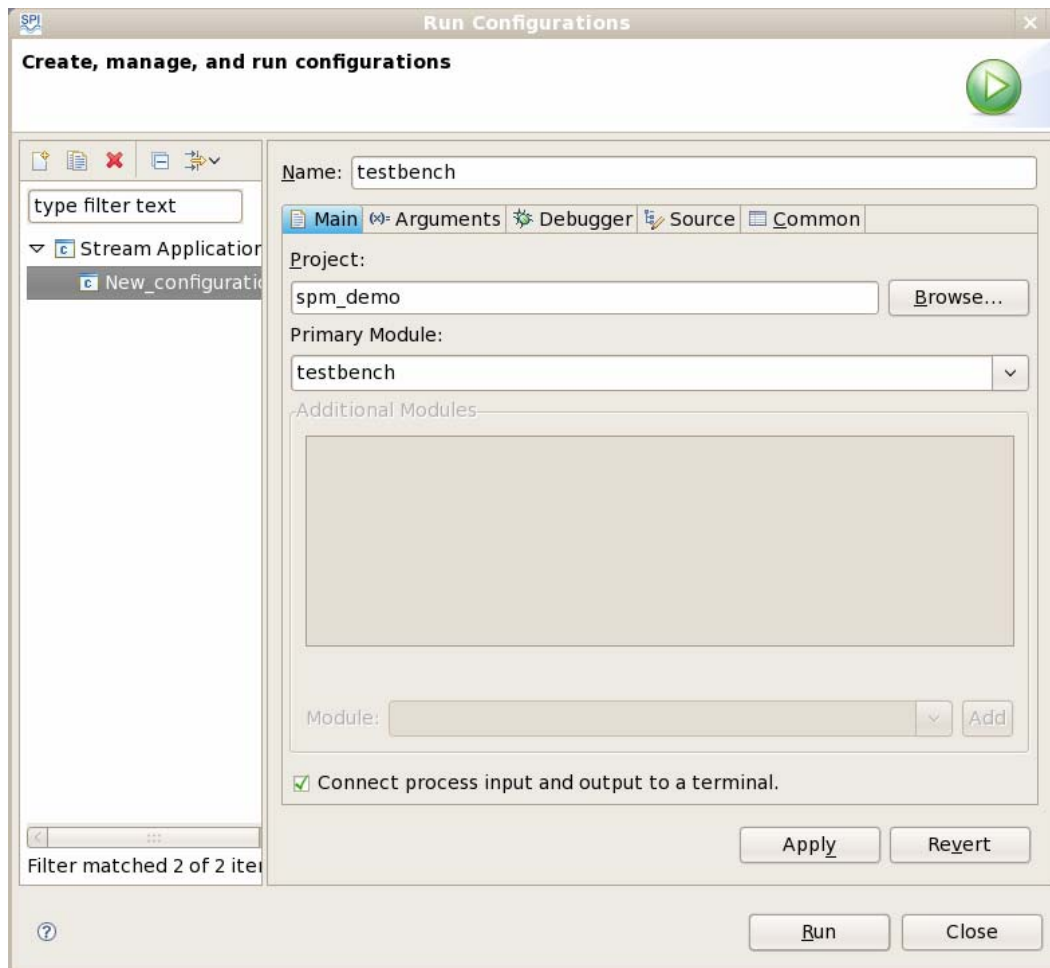
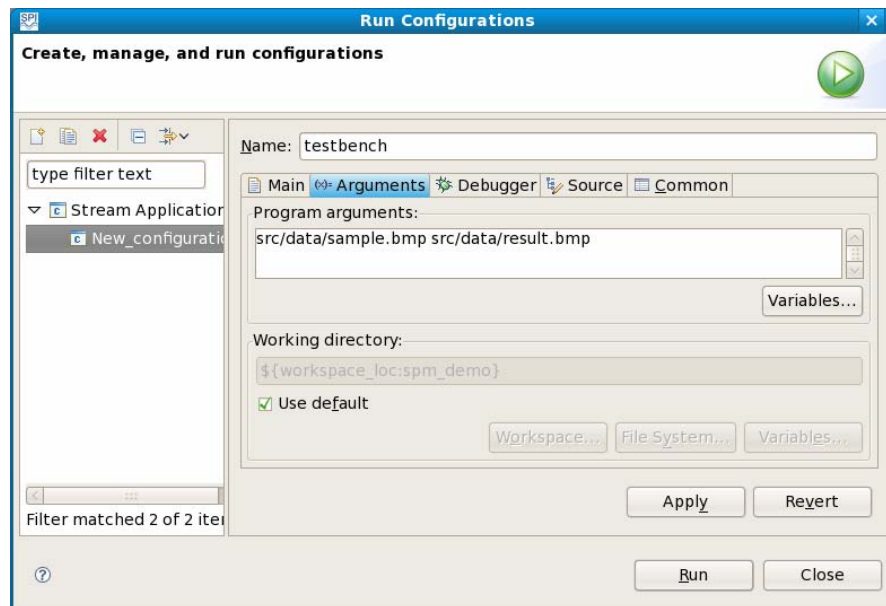
1. Click the arrow next to the Run  button on the toolbar and select **Run Configurations...**
2. In the **Run Configurations** window, right-click on **Stream Application** and select **New**. [Alternatively, click on **Stream Application**, then click on the **New** icon on the toolbar.]
3. Enter **testbench** as the **Name** of the run configuration.
4. Pull down **testbench** as the **Primary module** (Figure 10). Do not hit **Run** yet, as you still must enter the arguments for the run configuration.

Figure 10: Run Configuration



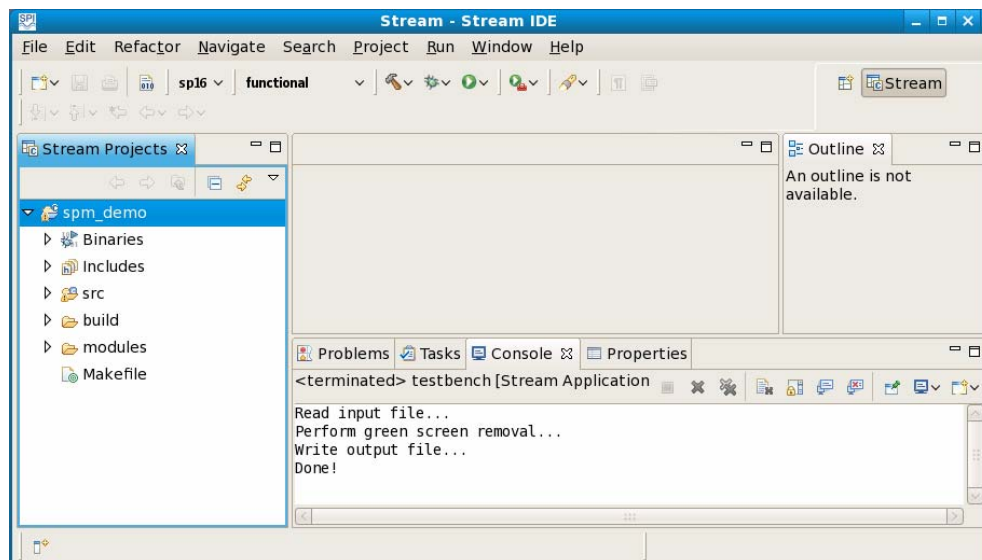
5. Click on the **Arguments** tab.
6. Enter the program arguments. The **testbench** arguments are the input file and the output file: **src/data/sample.bmp src/data/result.bmp** (Figure 11). Pathnames must be relative to the project directory. When you run an executable under the IDE, the IDE's current working directory is the project's directory within the IDE workspace, not the directory from which you invoked the IDE.

Figure 11: Run Configuration Arguments



7. Hit **Apply**, then hit **Run**. The IDE runs the **testbench** functional mode executable on the host and displays the program output in the Console view (Figure 12).

Figure 12: Run





9.3.3 Debug


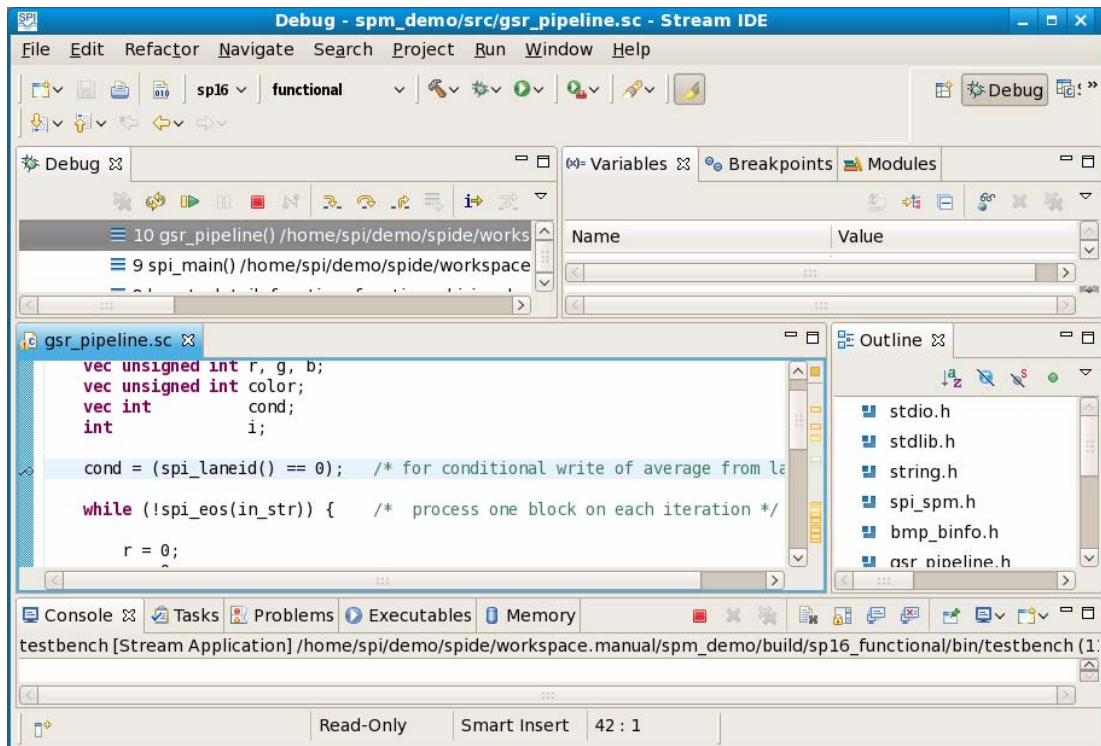
To begin debugging the functional mode program, pull down the arrow next to the debug  icon and select **testbench**. The IDE builds a debug executable and prompts you to confirm a switch to the Debug perspective. It starts program execution, stopping at the beginning of **spi_main** in source file **spimain.c**, as displayed in the **spimain.c** editor view (Figure 13).

Figure 13: Debug Perspective

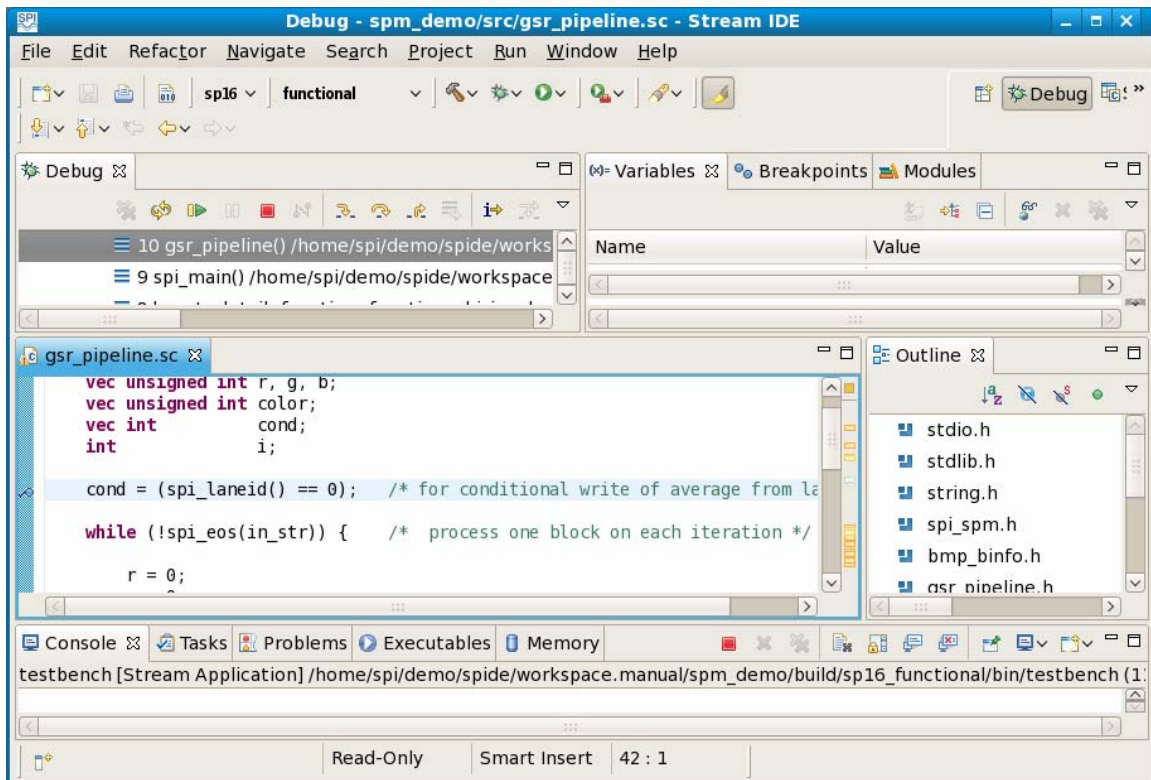


The debug perspective contains several additional views, including Variables (program variables), Breakpoints (debugging breakpoints), Streams (stream contents), Vectors (kernel vector variables), and Modules (program modules).

The **Run** menu shows the available program control options, including function keys **F5** (Step Into), **F6** (Step Over), **F7** (Step Return), and **F8** (Resume).

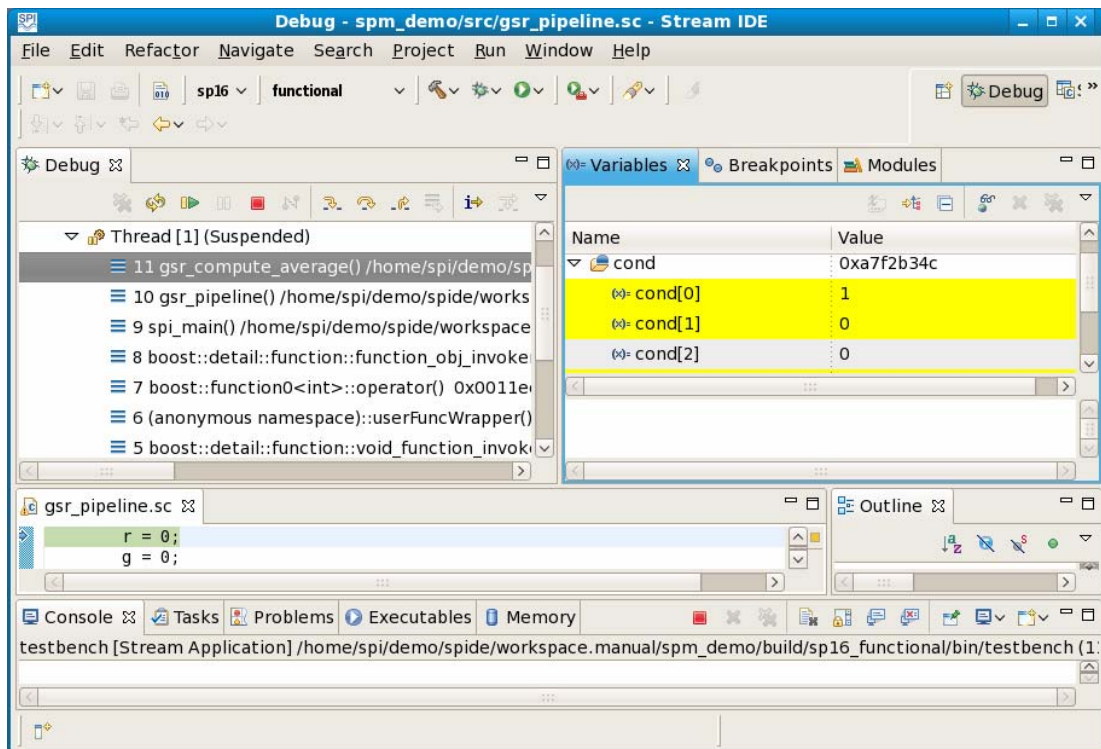
1. Hit **F6** four times and watch the **spimain.c** source view highlighting change as the debugger steps through the program.
2. Hit **F5** to step into **gsr_pipeline**; the source view switches to Stream source file **gsr_pipeline.sc**.
3. Click in the source window, then scroll up and set a breakpoint at the beginning of kernel **gsr_compute_average** by double-clicking on the left of the source view (Figure 14).


Figure 14: Set Breakpoint



4. Hit **F8** to run to the breakpoint.
5. Click in the Variables pane and scroll to find variable **cond**. Since **cond** is a vector variable, it has a different value in each lane of the stream processor.
6. Expand **cond** to see its uninitialized value in each lane.
7. Hit **F6** to step past the assignment to variable **cond**:
`cond = (spi_laneid() == 0);`
 This sets the value of **cond** to 1 in lane 0 and 0 in all other lanes. The Variables pane shows the new values, with the changed values highlighted (Figure 15).

Figure 15: Variables View



You should continue to step through the program and set breakpoints to become familiar with the operation of the IDE. You can use the stop button  on the console toolbar to stop a running program.

9.3.4 Fast functional mode

Like functional mode, fast functional mode simulates a program functionally on the host; it does not accurately simulate stream processor performance. It gives better performance than functional mode, but the generated program may be more difficult to debug because of optimization. For example, stepping through a program may jump to an unexpected location in the source, or the values of variables may change in unexpected ways.

To build and run a program in fast functional mode, pull down arrow next to the mode on the IDE toolbar and select **functional_fast**, then follow the instructions for functional mode above.

9.4 Profile mode

Functional mode and fast functional mode simulate a program functionally on the host. Debug mode, profile mode and release mode instead build programs that run under the Stream simulator or on stream processor hardware. For the **testbench** module, the generated program runs on DSP MIPS, either in simulation or on hardware. This section describes how to build a program in profile mode and then run it in simulation or on hardware. Debug mode is similar but is not described here. You can set breakpoints and control debug mode program execution in the IDE, just as described for functional mode above.

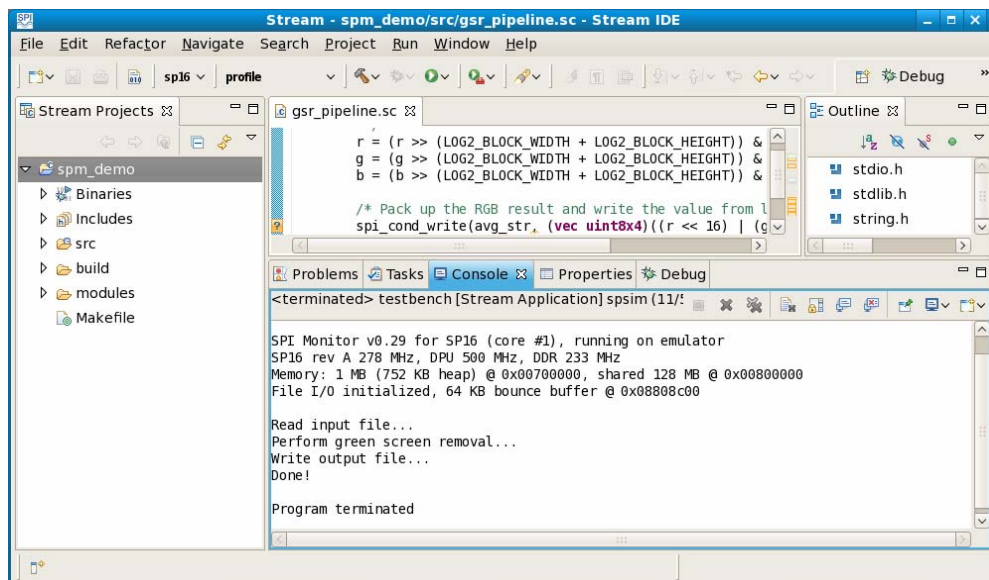
9.4.1 Build

Select the **Stream** perspective; you may need to drag the separator to the left of the **Debug** perspective icon near the upper right to expose the **Stream** perspective button. To build **spm_demo testbench** in profile mode, pull down the arrow next to the mode on the toolbar and select **profile**, then pull down the arrow next to the build icon and select **spm_demo**. The resulting DSP MIPS executable runs either under the simulator or on stream processor hardware.

9.4.2 Run under simulator

To run the **testbench** module under the simulator, pull down the arrow next to the run icon and select **testbench**. The IDE simulates program execution on DSP MIPS. Since **spm_demo** reads and writes a large file, execution under the simulator is rather slow; be patient. The console view shows the banner from the simulator and then the output of the simulated program (Figure 16). Wait for the “Program terminated” message from the monitor before you examine the generated profile.

Figure 16: Simulation



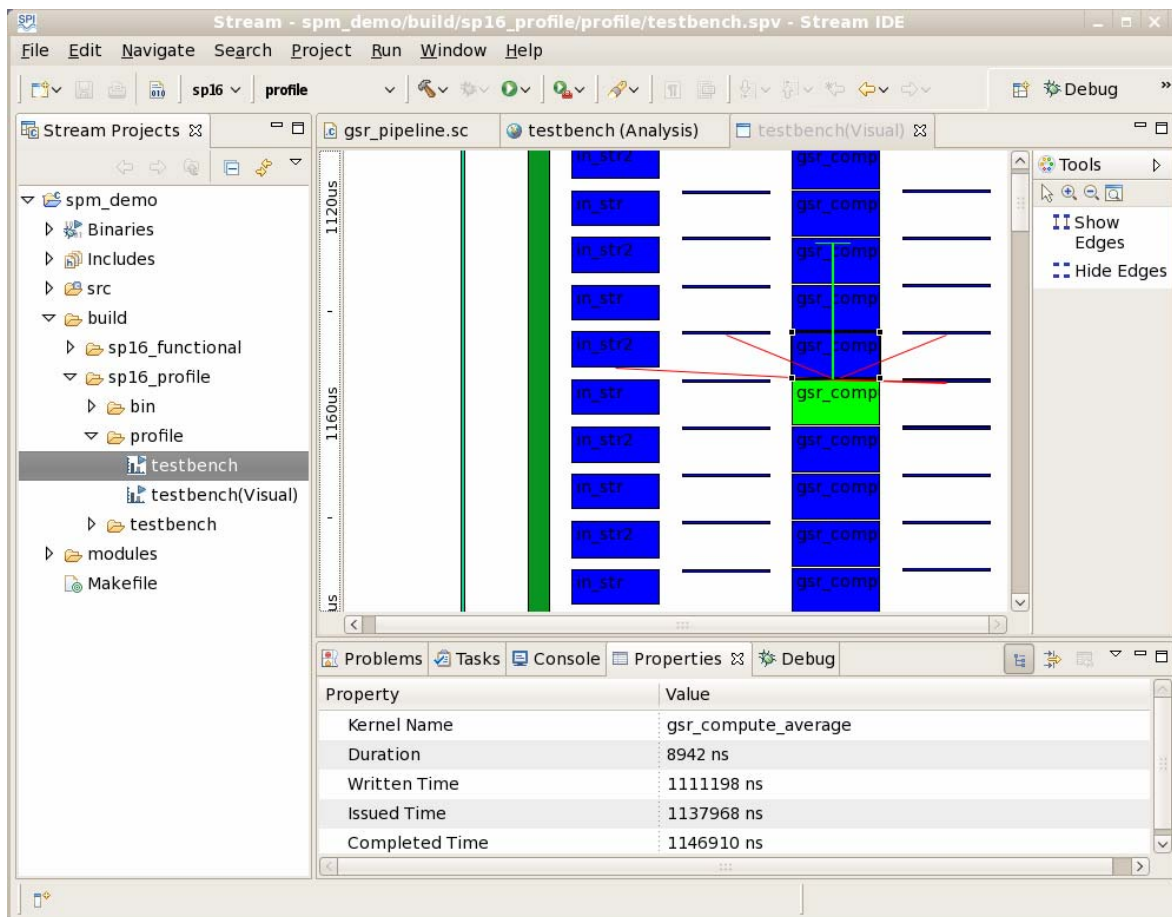
You can set simulator flags in the **Target** tab of the run configuration.

9.4.3 View profile data

Profile mode allows the developer to monitor and improve program performance through the use of stream command traces. Simulation or hardware execution of a program built in profile mode produces a stream trace. After simulation or hardware execution of the program:

1. Expand **build** under **spm_demo** in the Stream Projects view.
2. Expand **sp16_profile** and expand **profile**.
3. Double-click on **testbench** (the profile file named **testbench** in the **profile** subdirectory, not the identically named **testbench** subdirectory in the **sp16_profile** directory). The IDE displays program execution information in its testbench (Analysis) view and in its testbench (Visual) view.
4. You can use the scroll bar and the zoom buttons in the **Tools** menu to view the image (Figure 17).

Figure 17: Stream Trace View



The testbench (Analysis) view displays performance information in tabular form, as generated by **sppperf**. The testbench (Visual) view displays performance information visually. The vertical axis represents time, with a time scale ruler on the left edge of the window. The horizontal axis represents resources. In Figure 17, the mouse hovers over a call to kernel **gsr_compute_average**; the IDE displays the properties of the kernel call in the Properties view. The [Optimization](#) chapter below gives much more detail about how to use the performance data that **spide** displays to improve program performance.

9.4.4 Run on hardware

You need to define a new run configuration to run the **testbench** program on a stream processor device. The run configuration specifies the target hardware and program arguments.


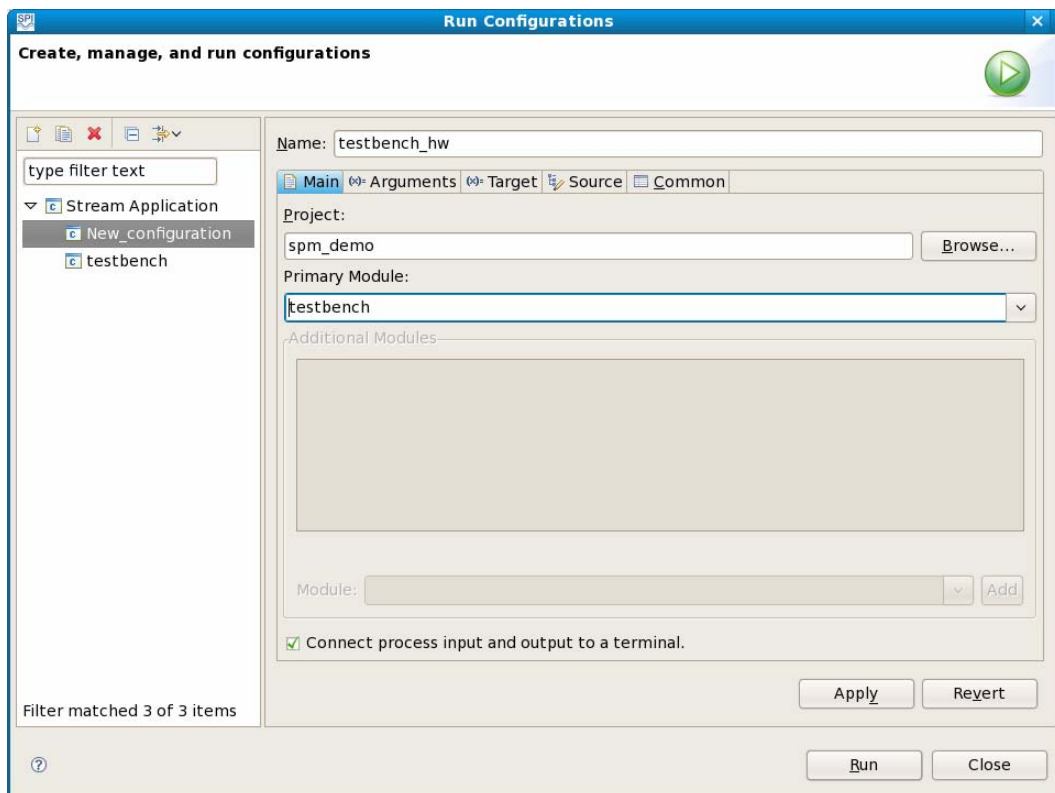
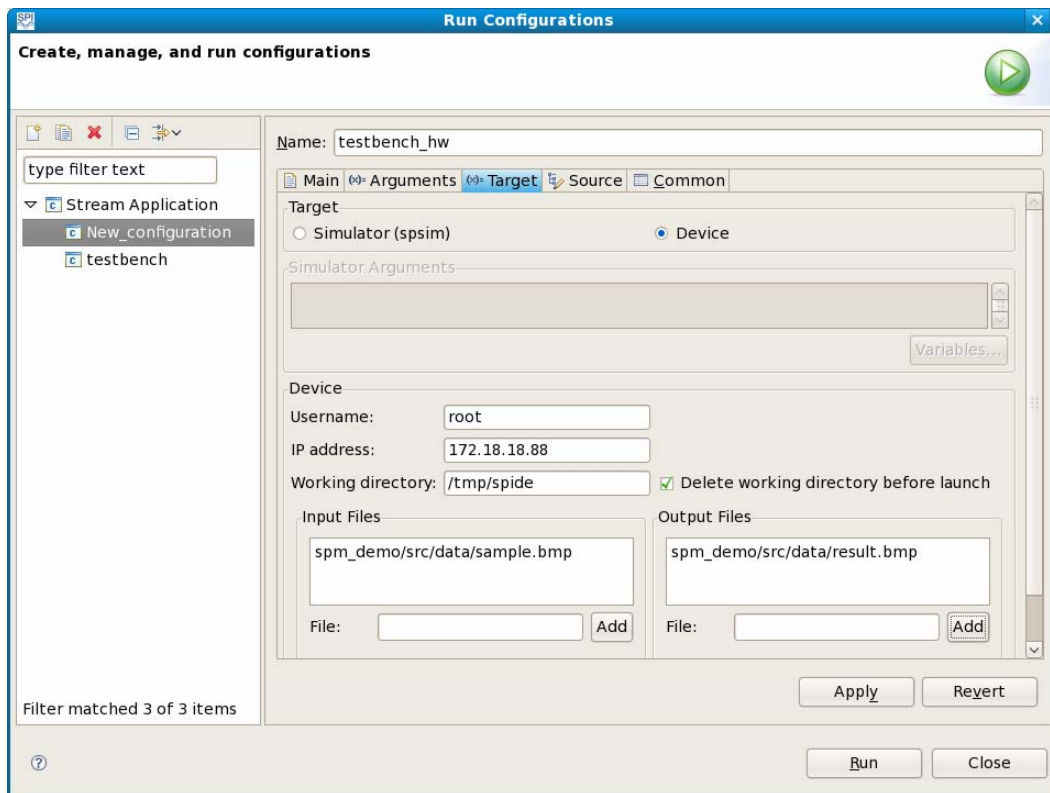
1. Click the arrow next to the Run  button on the toolbar and select **Run Configurations...**
2. In the **Run Configurations** window, right-click on **Stream Application** and select **New**. [Alternatively, click on **Stream Application**, then click on the **New** icon on the toolbar.]
3. Enter **testbench_hw** as the **Name** of the run configuration.
4. Hit **Browse...** and select **spm_demo** as the **Project**.
5. Pull down **testbench** as the **Primary module** (Figure 18). Do not hit **Run** yet, as you still must enter the target and argument information for the run configuration.

Figure 18: Hardware Run Configuration



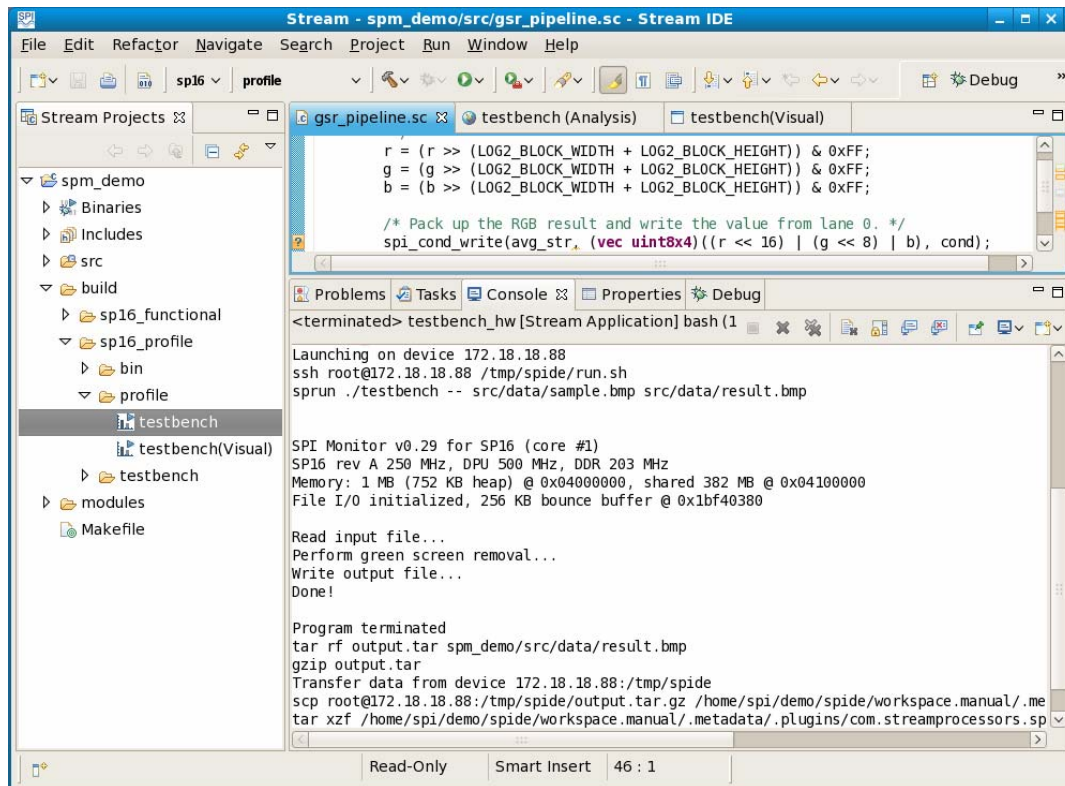
6. Click on the **Target** tab.
7. Click on the **Device** radio button to indicate execution on a stream processor hardware device.
8. Enter a user name, IP address and working directory.
9. Enter input filename **spm_demo/src/data/sample.bmp** and hit **Add**. Target input and output filenames are workspace-relative, not project-relative, as the directory structure on the device mirrors the directory structure in the **spide** workspace.
10. Enter output filename **spm_demo/src/data/result.bmp** and hit **Add** (Figure 19).

Figure 19: Target Run Configuration



11. Click on the **Arguments** tab.
12. Enter the program arguments: **src/data/sample.bmp src/data/result.bmp**. The program arguments are project-relative, not workspace-relative, so they do not include the **spm_demo** project component of the input file and output file pathnames specified in the device run configuration above. If you want to change the default SPM log mask settings, you can add one or more **--spi_log_mask=mask,value** options as additional arguments.
13. Hit **Run** (Figure 20).

Figure 20: Execution on Device



The Console view shows the work done by the IDE. It creates the specified working directory on the device, downloads the input file and the executable to the working directory, runs the program with the specified arguments, and uploads the output file to the desired location. The directory structure on the device mirrors the directory structure in the **spide** workspace.



9.5 *Release mode*

Release mode creates a release version of a program, without the overhead of debugging or profiling code. To build and run a program in release mode, pull down the arrow next to the mode button on the IDE toolbar and select **release**, then follow the instructions in the preceding sections.

9.6 Complete application

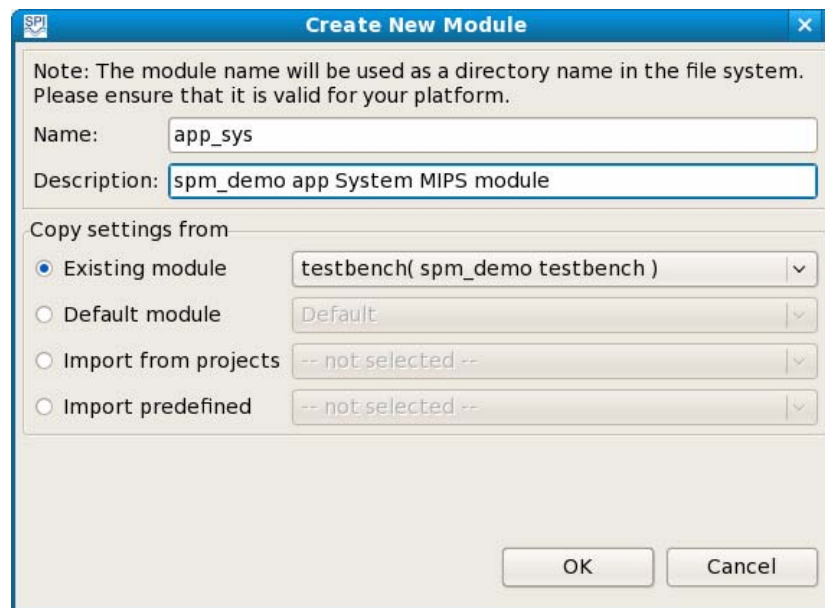
The preceding sections described how to build, run, and debug a **testbench** module. This section describes how to build the complete component-based **spm_demo** application and run it on stream processor hardware. The complete component-based application consists of a System MIPS image and a DSP MIPS image.

9.6.1 Create System MIPS module

This section describes how to configure a module to build the System MIPS image of the application. The next section describes how to configure a module to build the DSP MIPS image of the application. Together, these two images comprise the complete **spm_demo** application. The final section of this chapter shows how to create a run configuration to run the complete application on stream processor hardware.

1. In the **Stream projects** view, right-click **spm_demo** and select **Properties**. [Alternatively, select **Project >> Properties**.]
2. Click on **Stream Build**.
3. Click **Manage Modules...**
4. Hit **New** to create a new module.
5. Enter **app_sys** as the module name.
6. Enter a description (Figure 21).

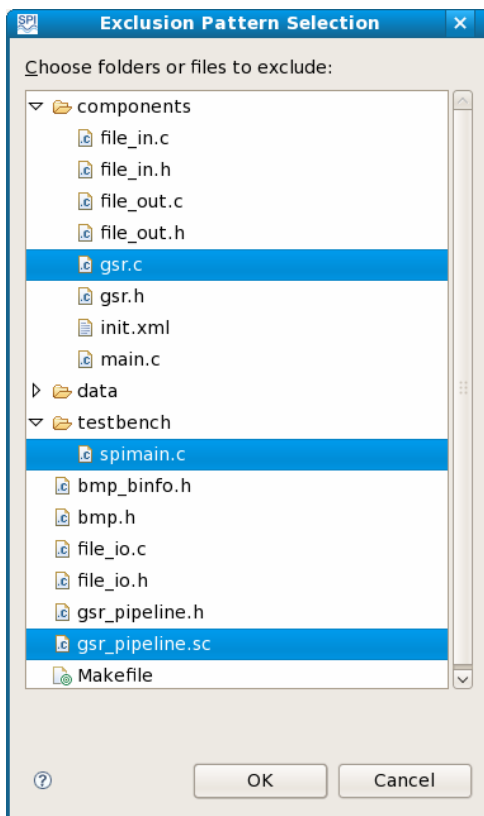
Figure 21: Create System MIPS Module



7. Hit **OK** to close the **Create New Module** window.
8. Hit **OK** again to close the **spm_demo: Manage Modules** window and return to the **spm_demo** Properties window.

9. Select **app_sys** from the **Module** drop-down menu.
10. Expand **Stream Build** and select **Artifact**.
11. Click the **Build artifact** tab.
12. Set the **Artifact Type** to **SYS-MIPS Executable**.
13. Click the **Source Location** tab.
14. Click **/spm_demo/src**.
15. Hit **Edit filter**. The **app_sys** module inherited the filter from the **testbench** module, so now you need to edit the filter for the System MIPS application.
16. Remove the existing pattern from the source file exclusion filter: click on **components/*** and hit **Remove**.
17. Click on **Add multiple** to select source files to exclude.
18. Expand the **components** and **testbench** directories, then hold down <Ctrl> and click to exclude files **components/gsr.c**, **testbench/spimain.c**, and **gsr_pipeline.sc** (Figure 22). These files are not part of the System MIPS executable.
19. Hit **OK** to close the **Exclusion Pattern Selection** window.
20. Hit **OK** again to close the **Source Folder Exclusion Patterns** window.
21. Hit **OK** again to finish adding the module and close the **Properties** window.

Figure 22: System MIPS Exclusion Pattern

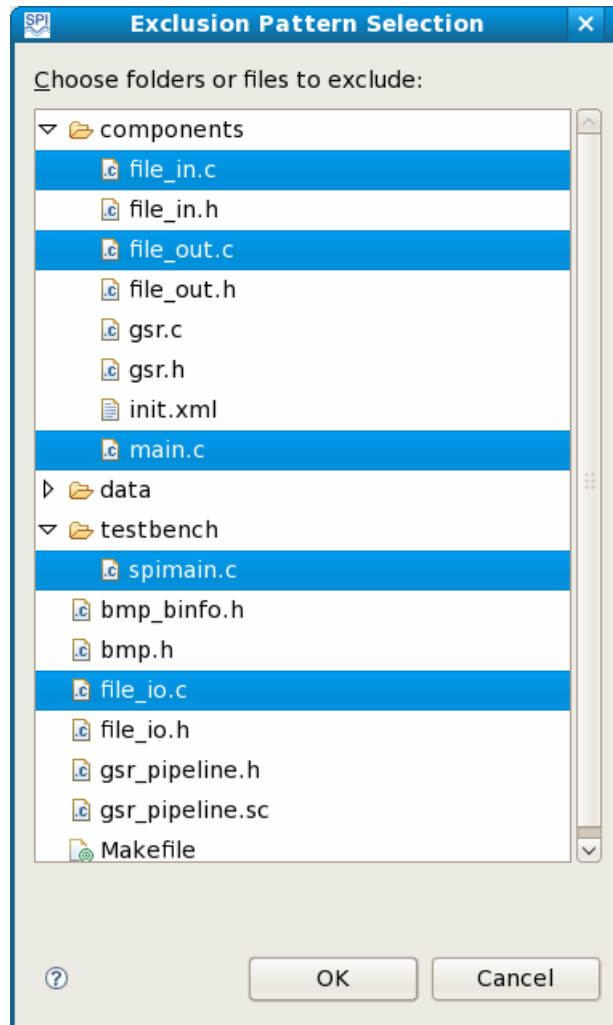


To build the module, select **release** mode on the toolbar and hit the build icon. When the build completes, you can expand **spm_demo/build/sp16_release/bin** in the Stream Projects view to see the **app_sys** executable. You still need to build the DSP MIPS executable as described in the next section before you can run the System MIPS executable.

9.6.2 Create DSP MIPS Module

Repeat the procedure of the preceding section to create a DSP MIPS module for the **spm_demo** application. Use the module name **app_dsp** and module type **DSP-MIPS executable**. The DSP MIPS executable uses only sources **component/gsr.c** and **gsr_pipeline.sc**, so the filter should exclude all other sources (Figure 23).

Figure 23: DSP MIPS Exclusion Pattern



Select **release** mode on the toolbar and hit the build icon. When the build completes, you can expand **spm_demo/build/sp16_release/bin** in the Stream Projects view to see the **app_dsp** executable. The next section describes how to run the complete application on stream processor hardware.

9.6.3 Run application

Finally, you need to define a run configuration to run the complete component-based **spm_demo** application on stream processor hardware.


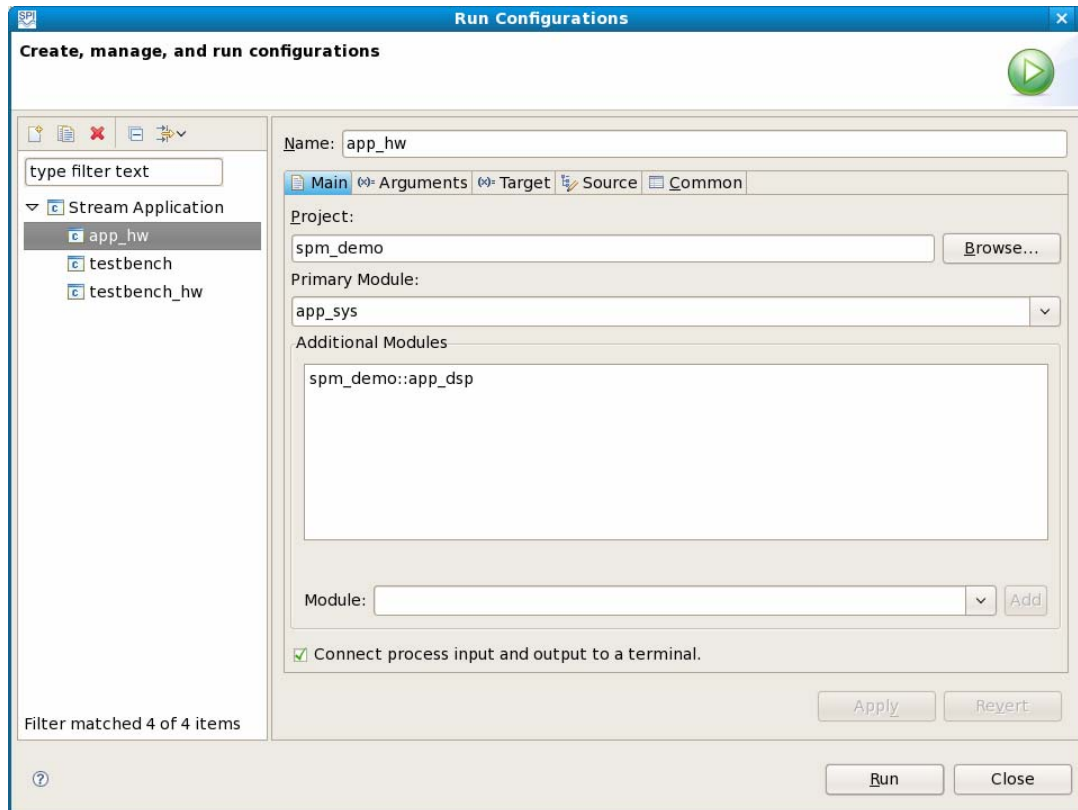
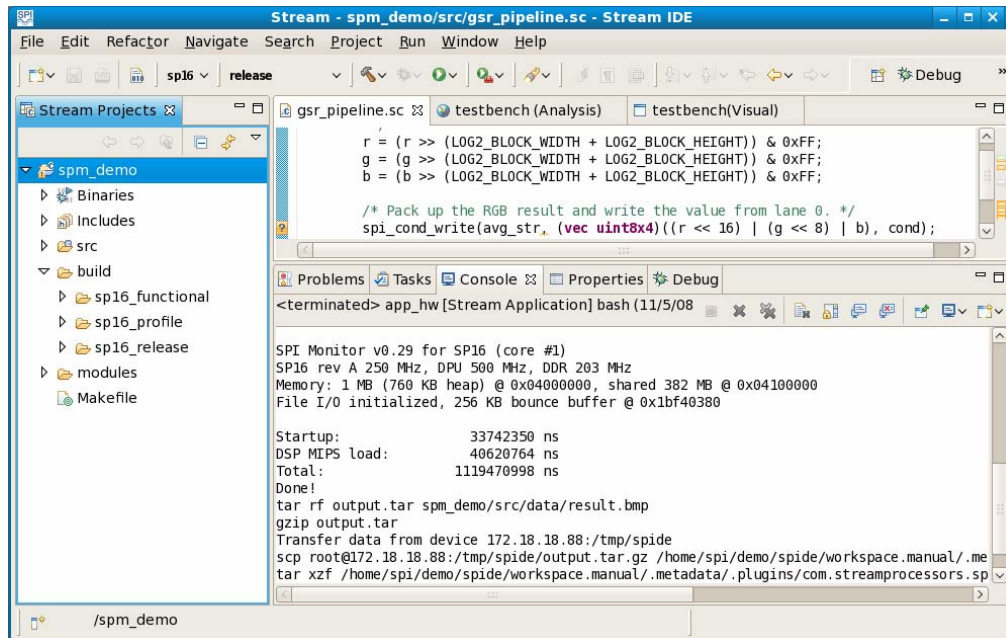
1. Click the arrow next to the Run  button on the toolbar and select **Run Configurations...**
2. In the **Run Configurations** window, right-click on **Stream Application** and select **New**.
3. Enter **app_hw** as the **Name** of the run configuration.
4. Pull down **app_sys** as the **Primary module**.
5. Pull down the **Module** list to **spm_demo::app_dsp** and hit **Add** to add the DSP MIPS module. The run configuration will run **app_sys**, but it also requires module **app_dsp**, so you must specify **app_dsp** as an additional module.
6. Hit **Apply** (Figure 24).

Figure 24: Application Run Configuration



7. Click on the **Target** tab.
8. Select the **Device** radio button.
9. Enter the user name, IP address, and working directory.
10. Enter workspace-relative input filename **spm_demo/src/data/sample.bmp** and hit **Add**.
11. Enter workspace-relative output filename **spm_demo/src/data/result.bmp** and hit **Add**.
12. Hit **Apply**.
13. Select the **Arguments** tab and enter program arguments. The **app_dsp** module builds a DSP MIPS image named **app_dsp**. By default, the System MIPS program tries to load a DSP MIPS image named **spm_demo.dsp.out** to DSP MIPS, so in addition to the project-relative input and output file arguments, its argument list must include a **-i** option giving the name of the image:
-i app_dsp src/data/sample.bmp src/data/result.bmp
14. Hit **Run**. The result of running the program appears in the Console view (Figure 25).

Figure 25: Run Application





9.7 *Import a project*

To import an existing **spide** project, such as the pre-built **spm_demo** project:

1. Select **File >> Import**.
2. Expand **General** in the Import window, then select **Existing Projects into Workspace**.
3. Browse to or type in the pathname of the existing project (e.g., `/opt/spi/Stream_nnn/demos/spm_demo`).
4. If you wish to work with copies of the imported project files in your workspace, check the **Copy projects into workspace** checkbox (see the paragraph below concerning readonly files). If you leave the checkbox unchecked, **spide** works with the project files in the original location instead.
5. Hit **Finish**.

After you import the **spm_demo** project from a Stream distribution, you will need to modify the hardware run configurations **testbench_hw** and **app_hw**, as they contain hard-wired user name, IP address and directory pathname specifications for the target hardware.

If you import a project with the **Copy projects into workspace** checkbox selected, **spide** preserves the permissions from the original project. If the original project contains readonly files, you must modify the permissions of any files you wish to change in the new workspace. In particular, run configurations (**.launch** files) should be writable for **spide** to work as expected.

9.8 *Use Makefile from command line*

spide creates a **Makefile** in the root directory of a project and then uses **make** to build the project. In some circumstances, you might find it convenient to build the project from the command line rather than under **spide**. This section gives a brief introduction to the **spide**-generated **Makefile**, using the **spm_demo** project as an example. It assumes that you are familiar with **make** and with Makefiles.

The project **Makefile** defines the following primary targets:

- **build** Build project artifacts
- **clean** Clean project artifacts
- **clobber** Clobber the entire build tree
- **package** Package the project into compressed tarballs
- **printvar-var** Display the value of **Makefile** variable *var*.

Useful variables used in the **Makefile** include:

ARCH	Architecture
MODE	Mode
MODULES	Modules: all (default), includes , libs , exes , or a module list
PROJECT_ARCHS	Architectures
PROJECT_DEPS	Dependencies
PROJECT_EXES	Executables
PROJECT_INCLUDES	Includes
PROJECT_LIBS	Libraries
PROJECT_MODES	Modes



PROJECT_MODULES	Modules
PROJECT_NAME	Name
PROJECT_TYPES	Types
PROJECT_VERSION	Version
TYPE	all (default), {exe,lib}_{artifacts,sources} , common
VARIANT_CFLAGS_COMMON	C compilation flags common to all modes
VARIANT_CFLAGS_mode	Mode-specific C compilation flags for <i>mode</i>
VERBOSE	If nonempty, print command lines generated by make

The **Makefile** references additional module-specific files **modules/module.mk**.

For example, to verbosely build an SP-16 release complete application version of **spm_demo**, type:

```
$ make build ARCH=sp16 MODE=release "MODULE=app_dsp app_sys" VERBOSE=1
```

To remove existing build artifacts and build an SP-16 functional testbench version of **spm_demo**, type:

```
$ make clean
$ make build ARCH=sp16 MODE=functional MODULE=testbench
```

To package the sources for **spm_demo** into a tarball, type:

```
$ make package TYPE=exe_sources
```

This builds tarball **build/pkg/spm_demo_1.0.0.0_exe_sources.tgz**.



10 Performance optimization

This chapter discusses Stream program performance analysis and optimization. The Stream programming model allows the programmer to create Stream programs that use the powerful hardware features of a stream processor efficiently, and the Stream tools produce performance data that guide the programmer toward improved program performance.

A programmer can obtain performance data for a Stream program from profile data generated during program simulation, as well as from timers coded explicitly in the program source. The [Performance](#) section of the [Command line tools](#) chapter above describes how to generate tabular program performance information with **spperf**, and the [Stream Program Development](#) chapter gives information on how to generate and view tabular and visual performance information using **spide**. This chapter introduces some basic optimization concepts and shows how to interpret the performance data generated by **spperf** and **spide**.

Production analysis breaks down a process into a set of tasks. Each task has a known set of resource requirements, possible dependencies on other tasks, and a required time to completion. Scheduling attempts find an optimal task schedule: a schedule that uses available resources to complete the process as quickly as possible. An optimal schedule has a *critical path* to completion; increasing the time required for a task on the critical path increases the total time required for the project.

A stream processor presents optimization opportunities at the component level, at the pipeline level, and at the kernel level. The following sections discuss optimization issues and techniques for each level.

Storm-1 Benchmarks describes the Storm-1 benchmark program in distribution directory **benchmarks/benchmark/**. The program and document provide detailed examples of Stream program optimization.

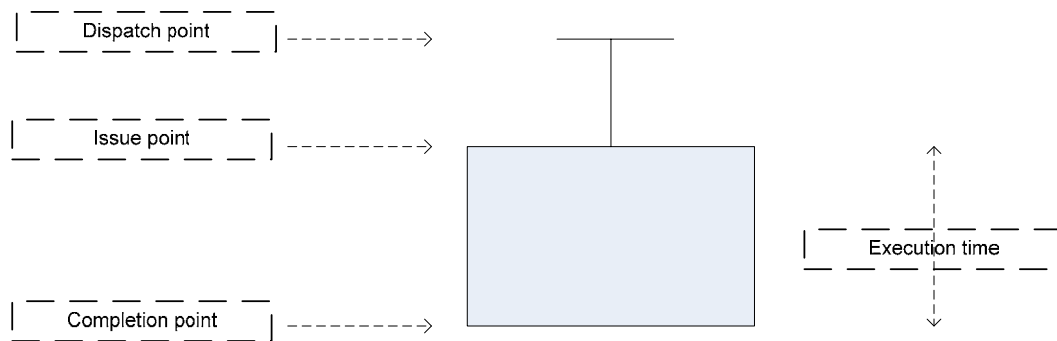


10.1 Pipelines

The Stream programming model frees the programmer from the burden of needing to specify the details of synchronization between parts of a stream processor. However, in many cases the programmer does need to understand the dependencies and resource requirements in Stream code in order to write an efficient program. The System MIPS program, the DSP MIPS program, the stream controller, and a kernel on the DPU may all be running simultaneously, and fully utilizing the stream processor's power involves careful programming. This section describes some common pipeline optimization issues.

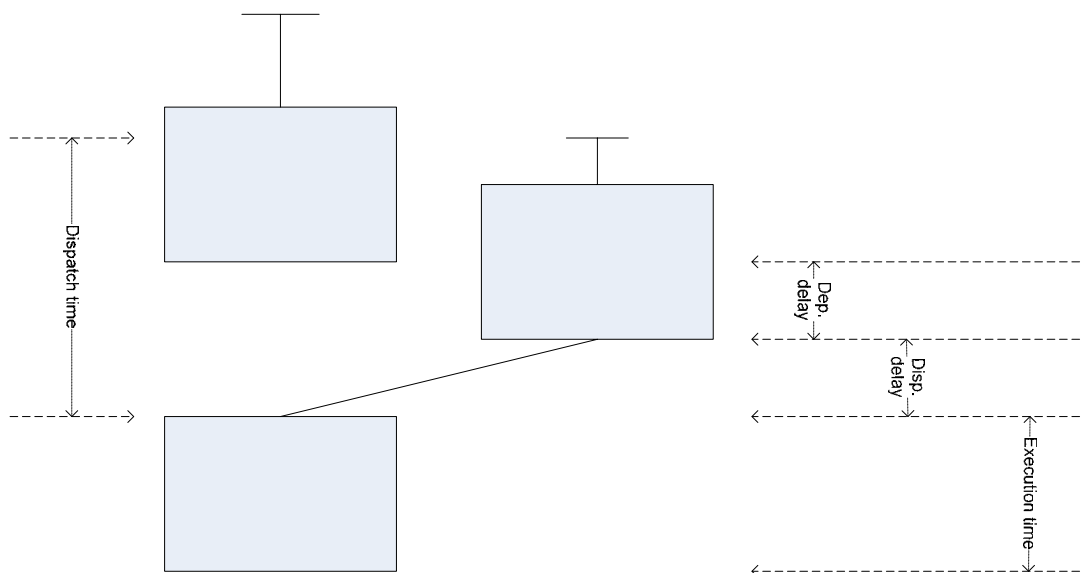
Stream processor hardware includes a *stream controller* that loads kernels to the DPU, runs kernels, and performs direct memory access (DMA) data transfers between memory and the LRF. When a pipeline function in a Stream program running on DSP MIPS executes a **spl_load_*** function, for example, it writes a stream command to the stream controller to initiate the transfer, and then it continues to execute subsequent code from the Stream program while the stream controller performs the data transfer. Stream commands have implicit dependencies: a **spl_load_*** command must wait for the completion of a previous **spl_store_*** command to the same buffer, a kernel may not begin execution until its argument streams are loaded, and so on. Stream commands also have resource requirements: the stream controller can only execute a single kernel at one time, for example. A stream controller command issues (begins execution) once all its dependencies and resource requirements are satisfied, and at some later time the command completes (finishes execution).

The point in time when DSP MIPS dispatches a stream operation to the stream controller is the operation's *dispatch point*, the point when the operation begins execution is its *issue point*, and the point when the operation completes is its *completion point*. The interval between its issue point and its completion point is its *execution time*.



Here and in **spide** visualizations below, the vertical axis represents time, while the horizontal axis represents resources; for example, the diagram above might represent a stream load operation. Since the vertical axis represents time, the height of the rectangle indicates its execution time..

The interval between the dispatch point of a stream operation and the dispatch point of the previous stream operation is its *dispatch time* (see the diagram below). The interval between when its resources are available and when its dependencies are satisfied is its *dependency delay*. The time between when its resources are available and its dependencies are satisfied and its dispatch point is its *dispatch delay*.



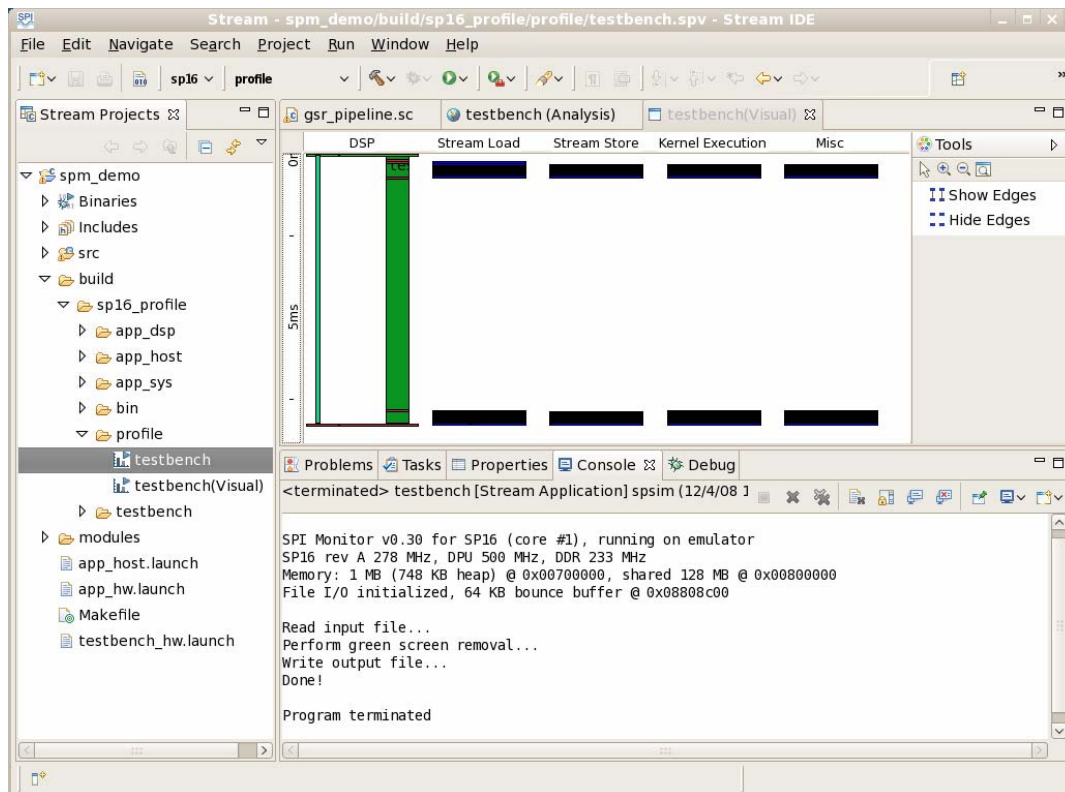
For each type of resource (e.g., the resource on the left in the diagram), the sum of execution times, dependency delays, and dispatch delays over the entire program equals the total program execution time. To achieve optimal performance, a program should try to fully utilize the performance-limiting resource of the processor; in other words, the performance-limiting resource should be kept busy all the time. If it is not busy, either it must be waiting for a command to be dispatched to it (dispatch delay) or it must be waiting for a dependency to be satisfied so that a command may begin execution (dependency delay). To improve performance, the programmer should pack operations to reduce dispatch delays and dependency delays, and then tune operations to reduce execution time.

The total dispatch delay time of a pipeline divided by its total execution time is its *dispatch-limited* time. Section [Dispatch delays](#) below describes how to reduce dispatch delays. The total dependency delay time of a pipeline divided by its total execution time is its *dependence-limited* time. Section [Dependency delays](#) below describes how to reduce dependency delays.

Simulation of a profile mode program generates a profile that contains performance information. The remainder of this chapter describes the use of Stream tools to evaluate performance and suggests how to use performance data to optimize performance.

10.2 Visualization

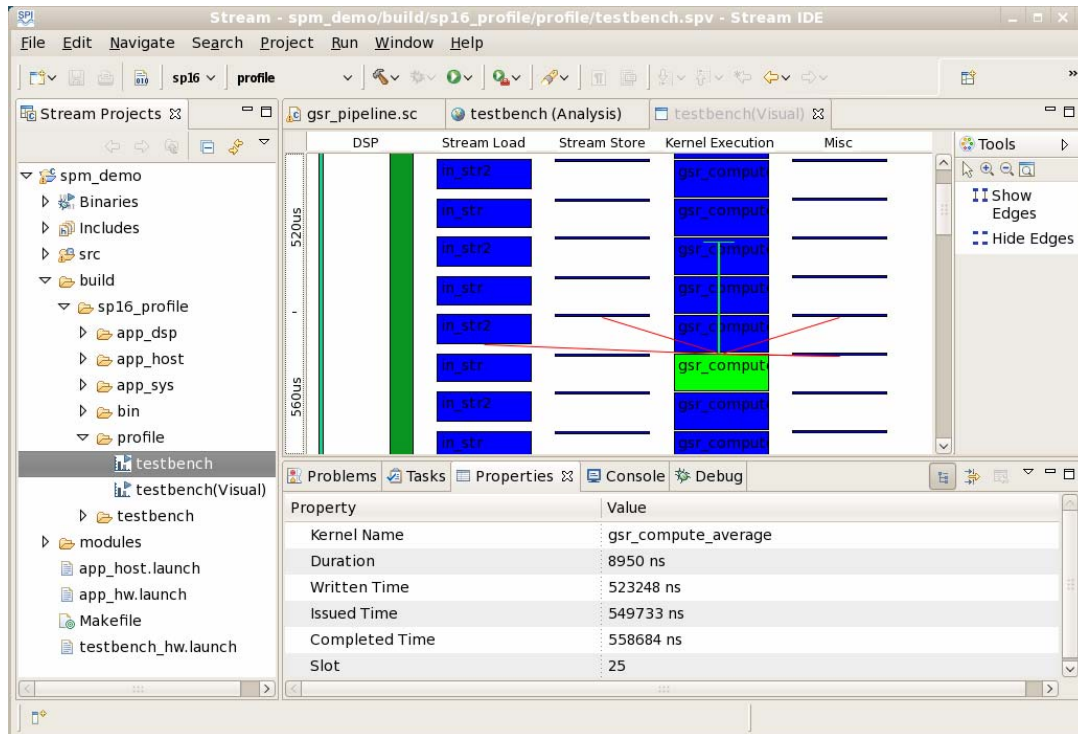
In **spide**, clicking on a profile file generated by profile mode simulation of a program opens a visualization of the program's execution. Build **spm_demo** in **sp16_profile** mode, then run the **testbench** version. After it terminates, click on profile file **testbench** under **build/sp16_profile/profile**; the IDE opens **testbench (Analysis)** and **testbench (Visual)** views. Hit the **Zoom to Fit** button to the right of the visual view to see the entire profile:



In **spide** visualizations, the vertical axis represents time, with a time ruler along the left edge. The horizontal access represents resources: DSP MIPS execution, stream loads, stream stores, kernel executions, and miscellaneous operations (kernel microcode VLIW loads and loads/stores for array, scalar, and conditional stream kernel arguments). Since the vertical axis represents time, the height of a rectangle represents its duration. After **spm_demo** starts DSP MIPS execution, it executes kernel **gsr_compute_average** repeatedly, shown by the very tightly-packed rectangles near the top of the visualization. (What appear to be single rectangles above are actually stacks of many very thin rectangles, as zooming in shows.) Then the program takes a relatively long time to sort the block averages and find the mode (background color); this code runs only on DSP MIPS, with no stream or kernel operations. Finally, it repeatedly calls **gsr_remove_background**, shown by the tightly-packed rectangles at the bottom of the visualization. Hovering over any item in the visualization brings up a pop-up description.



Zoom buttons to the right of the visual view let you zoom in or out. Hit the '+' zoom button several times to zoom in, then scroll to the group of loads, stores and kernel calls near the top of the visualization.



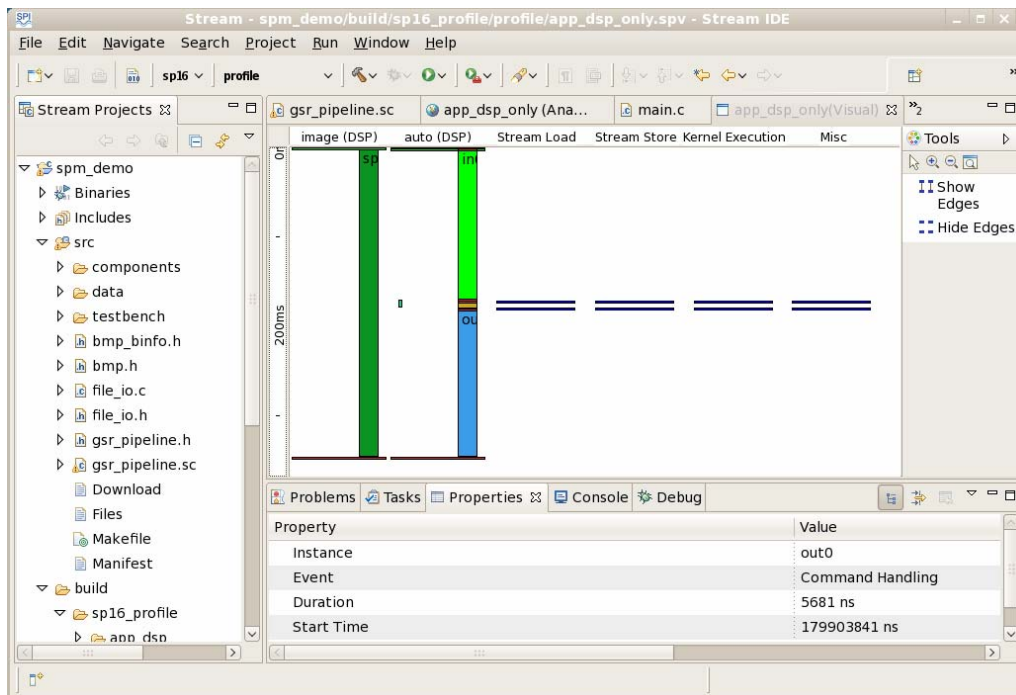
Clicking on any item produces information in the Properties view. The example above gives properties of one call of kernel **gsr_compute_average**: its total duration, when it was written, issued and completed, and the stream controller slot used by the operation.

Hovering over an item produces red lines that show its dependencies on other items. In the example above, the highlighted **gsr_compute_average** kernel execution depends on a stream load and a stream store, as well as on additional items. The top of the green 'T'-shaped line above the highlighted **gsr_compute_average** kernel execution rectangle indicates when the DSP MIPS program wrote the kernel execution request to the stream controller, the top of the highlighted rectangle indicates when the stream controller issued the operation, and the bottom of the highlighted rectangle indicates when the operation completed.

The **testbench (Analysis)** view gives tables with information about program performance, identical to the tables produced by **spperf**. The next section describes the tables. The remainder of this chapter shows how to use the information from **spide** visualizations and tables to improve Stream program performance.

10.3 Components

You can examine the performance of a component version of the **spm_demo** application with visualization. At present, you can only profile using simulated program execution under **spsim**, not execution on stream processor device hardware. If you build a component version of **spm_demo** that runs on DSP MIPS only (not using System MIPS) in profile mode and run it, the profile visualization (zoomed out all the way) looks like this:



The second column from the left in this visualization shows three component instances running successively: first file input component instance **in0** (green), then the background replacement component instance **gsr0** (brown), then the file output component instance **out0** (blue). Because the file input and file output components run on DSP MIPS rather than System MIPS here, they are slow. Only the **gsr0** instance uses streams and kernels. You can zoom and hover over operations for a more detailed view of **spm_demo** operation, including opening and closing of buffers with **spi_buffer_open** and **spi_buffer_close**, barriers **spi_barrier**, timers, component command handling and component execution, and so on. The visualization shows buffer operations, but not dependencies between them, as when one component waits for availability of a buffer provided by another component before it begins execution.



10.4 Tables

This section describes the performance tables in a **spide** profile Analysis view, which are identical to the tables generated by **spperf**. The data in the tables below may differ from the data in tables generated from a current Stream distribution.

The Simulation Configuration table gives basic information about the simulation: toolset, device, MIPS clock frequency, DPU clock frequency, DDR frequency and width, and the time units used in later tables. The default time unit is microseconds (**us**).

Tools	Device	MIPS Freq	DPU Freq	DDR Freq	DDR Width	Time Units
2.2.0	sp16	278.44 Mhz	499.50 Mhz	NA (233 Mhz is default)	NA (128 Bits is default)	us

The pipeline summary table gives information about each pipeline in the program. Since **spm_demo** contains a single pipeline, its pipeline summary table contains only one line. It gives the pipeline's total execution time, its percentage of the total application execution time, and the percentage of VLIW instruction memory it uses. If instruction memory usage exceeds 100%, the program must reload kernels during execution, so you should consider restructuring the pipeline.

ID	Function	Execution Time	Application Weight	I-Mem Usage	Pack				Tune	
					Dispatch Limited	Dependence Limited	DMA Utilization	DPU Utilization	DRAM Utilization	VLIW Utilization
1	gsr_pipeline.sc	8,030.36	100.0%	18.8%	90.5%	1.0%	8.5%	6.8%	100.0%	24.6%

The remaining information in the pipeline summary table is divided into two general categories, *pack* and *tune*. To optimize a stream program, you should first pack operations as densely as possible to assure full resource utilization. Once operations are well packed, you should tune operations to further improve performance.

Packing data tells you the *dispatch limited* and *dependence limited* percentages of pipeline execution time; section [Pipelines](#) above defines these terms. DMA utilization is the percentage of time the pipeline uses the stream controller DMA engine. DPU utilization is the percentage of time the pipeline uses the DPU. Ideally, a well-tuned program should fully utilize either the DMA engine (high DMA utilization, so the program is DMA-limited) or the DPU (high DPU utilization, so the program is DPU-limited). In the **spm_demo** example, much of the program execution time is spent waiting for DSP MIPS, so the program is neither DMA-limited nor DPU-limited.

Tuning data tells you how well your program uses DRAM and how well it uses DPU ALUs.

The packing data in the Pipeline Summary table immediately confirms the **spm_demo** performance issue noted in the visualization discussion above: it spends much of its execution time running DSP MIPS code, so it is largely dispatch limited.



The DPU kernels summary table shows the time used by each kernel in the program in nanoseconds and as a percentage of total program execution time. It also shows VLIW instruction memory use and percentage of ALU utilization for each kernel.

Kernel Name	Execution Time		% Instn Mem	% ALU Util	Stall Count		Filename (line)
gsr_compute_average	340.14	4.24%	8.1%	17.4%	31	0.7%	gsr_pipeline.sc(335, 337)
gsr_remove_background	202.98	2.53%	1.3%	31.8%	0	0.0%	gsr_pipeline.sc(392, 398)

Kernel **gsr_compute_average** executes in about 0.34 milliseconds and kernel **gsr_remove_background** executes in about 0.20 milliseconds. Performance numbers may vary in different Stream releases.

The remaining tables describe per-pipeline performance. Since **spm_demo** contains a single pipeline, all the remaining data applies to that pipeline. The execution breakdown table contains the same data as the pipeline summary table, but with per-pipeline percentages rather than per-program percentages.

Execution Time		8,030.36
Application Weight		100.0%
Instruction Memory Usage		18.8%
P A C K	Dispatch Limited	90.5%
	Dependence Limited	1.0%
	DMA Utilization	8.5%
	DPU Utilization	6.8%
T U N E	DRAM Utilization	100.0%
	VLIW Utilization	24.6%

Application weight indicates the relative effect of the pipeline on overall application performance. Instruction memory usage indicates how much of available VLIW instruction memory the pipeline uses; if it exceeds 100%, the program must reload kernels during execution, so you should consider restructuring the pipeline.



The stream operations table describes each stream operation (stream loads, stream stores, and kernel executions) in the pipeline. For each stream operation, it gives minimum, average, and maximum values of the operation's dispatch time, execute time, dispatch delay, and dependence delay. This detailed view lets you optimize packing.

Stream Operations			Calls	Dispatch Time			Execute Time			Dispatch Delay			Dependence Delay		
ID	Name	Type		Min	Median	Max	Min	Median	Max	Min	Median	Max	Min	Median	Max
1	idx_str	Load	1	0.00	0.00	0.00	0.70	0.70	0.70	0.00	0.00	0.00	0.00	0.00	0.00
2	in_str	Load	19	0.64	0.64	63.44	7.66	7.69	8.82	0.00	0.00	62.73	0.00	0.00	0.00
3	in_str2	Load	19	0.80	0.80	5.60	7.72	7.88	8.94	0.00	0.00	0.00	0.00	0.00	0.00
4	gsr_compute_average	Kernel	19	1.36	5.78	19.23	8.95	8.95	8.95	0.00	0.00	15.52	0.00	0.00	6.87
5	gsr_compute_average	Kernel	19	1.23	9.39	13.85	8.95	8.95	8.95	0.00	0.00	0.00	0.00	0.00	0.00
6	avg_str	Store	19	0.85	0.85	5.82	0.19	0.19	0.21	0.00	0.00	3.37	0.54	1.23	15.53
7	avg_str2	Store	19	0.78	0.79	2.81	0.17	0.19	0.21	0.00	0.00	0.00	0.12	1.24	8.75
8	in_str	Load	19	0.75	0.87	7,230.44	6.23	10.16	11.89	0.00	0.00	7,196.47	0.00	0.00	0.00
9	in_str2	Load	19	0.65	0.77	2.71	6.27	9.20	10.12	0.00	0.00	0.00	0.00	0.00	0.00
10	gsr_remove_background	Kernel	19	1.23	6.92	12.40	5.34	5.34	5.34	0.00	0.00	8.88	3.87	5.16	7,202.90
11	gsr_remove_background	Kernel	19	1.47	9.32	10.38	5.34	5.34	5.34	0.00	0.00	0.00	0.00	3.71	4.63
12	out_str	Store	19	0.85	1.06	2.08	7.08	8.74	9.38	0.00	0.00	1.65	0.00	0.00	7.95
13	out_str2	Store	19	0.58	0.58	3.10	5.48	9.04	9.48	0.00	0.00	0.00	0.00	0.00	2.54

This table shows the cause for the large dispatch delay limiting program performance: for id 8, the program must wait for DSP MIPS to compute the background color on the first iteration, resulting in a very large dispatch time and dispatch delay.

The remaining tables give pipeline performance tuning information. The DPU kernels table (broken into two sections below for readability) gives more detailed information about each kernel, including its usage of instruction memory (VLIW memory) and ALUs. If a pipeline uses more instruction memory than is available, pipeline execution will be slowed by reloading kernels as needed. If a pipeline uses less than the available instruction memory, performance may improve if the pipeline is combined with other pipelines. Improving ALU utilization is a key to tuning kernel performance.

DPU Kernels	DPU Time		% Instn Mem	% VLIW Util	Stall Cycles	
gsr_compute_average	340.14	4.24%	8.1%	17.4%	31	0.7%
gsr_remove_background	202.98	2.52%	1.3%	31.8%	0	0.0%
Total Instruction Memory			9.4%			



The table also gives data about each basic block that is an inner loop in a kernel. For each inner loop block, it gives the percentage of time the kernel spends in the block, the estimated number of iterations of the block, the number of ALU operations in the block, block cycle counts, and software pipelining information. In the example below, **gsr_compute_average** does not contain a pipelined inner loop, while **gsr_remove_background** does. Some kernels contain no inner loop blocks.

						Inner Loops						
ID	Est % Kernel Time	Est Iterations			Num Ops	Cycles				Software Pipeline Stages	Filename (line)	
		Min	Avg	Max		Limits			Achieved			
						Critical Resource	Critical Path	Reoccur II				
1	95.9%	148	148	148	76	-	27	-	29	-	gsr_pipeline.sc(335, 337)	
1	97.4%	514	514	514	23	5	27	5	5	6	gsr_pipeline.sc(392, 398)	

The DMA loads/stores table gives information about memory transfers, including minimum, average and maximum size of transfers and the percentage of DDR burst utilization.

DMA Load/Stores		% DMA Time	Actual DMA Tx (MB/s)	Useful DMA Tx (MB/s)	Size (bytes)			% DDR Burst Utilization			Filename (line)
Stream	Type				Min	Avg	Max	Min	Avg	Max	
in_str	Load	4.28%	1.188	1.188	32,768	32,768	32,768	100.0%	100.0%	100.0%	gsr_pipeline.sc(299, 354)
in_str2	Load	4.03%	1.188	1.188	32,768	32,768	32,768	100.0%	100.0%	100.0%	gsr_pipeline.sc(307, 360)
out_str2	Store	2.10%	0.594	0.594	32,768	32,768	32,768	100.0%	100.0%	100.0%	gsr_pipeline.sc(382)
out_str	Store	2.05%	0.594	0.594	32,768	32,768	32,768	100.0%	100.0%	100.0%	gsr_pipeline.sc(378)
avg_str2	Store	0.05%	0.002	0.002	128	128	128	100.0%	100.0%	100.0%	gsr_pipeline.sc(321)
idx_str	Load	0.05%	0.002	0.002	2,048	2,048	2,048	100.0%	100.0%	100.0%	gsr_pipeline.sc(270)
avg_str	Store	0.01%	0.002	0.002	128	128	128	100.0%	100.0%	100.0%	gsr_pipeline.sc(317)

Here the sum of the DMA time percentages (around 12%) exceeds the DMA utilization time in the execution breakdown table (8.5% of total execution time) because double buffering allows multiple stream operations to occur simultaneously.



10.5 Stream operations

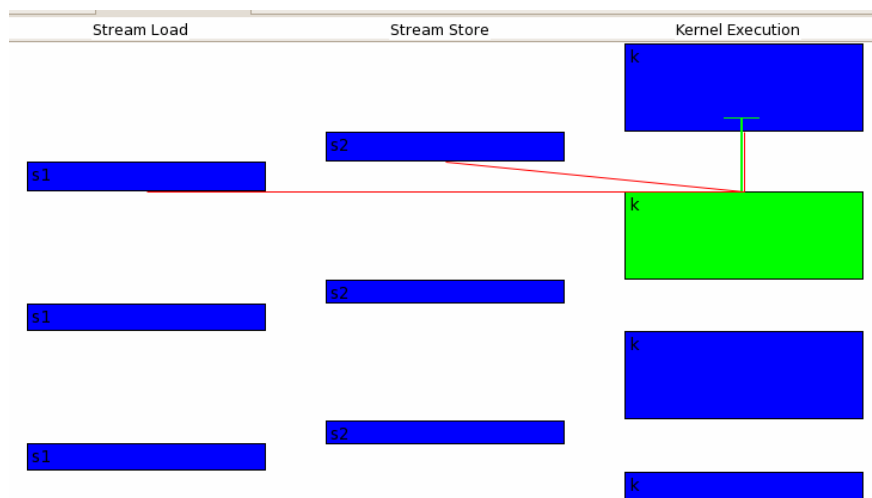
10.5.1 Dependency delays

Packing stream operations efficiently by removing dependency delays is an essential part of obtaining maximum performance from a stream processor. This section describes how to evaluate stream operation packing. It shows how a simple technique called *double buffering* can improve performance.

The following simple loop runs kernel **k** on successive strips of a buffer. **spi_load_block** loads input stream **s1** and **spi_store_block** stores output stream **s2**, and here both use the same buffer **buf1**:

```
/* Case 1: load and store to same buffer. */
for (i = 0; i < NSTRIPS; i++) {
    spi_load_block(s1, buf1, i * NBYTES, NRECS);
    k(s1, s2);
    spi_store_block(s2, buf1, i * NBYTES);
}
```

The execution trace looks like this:



Here kernel **k** depends on the preceding load: **k** uses **s1** as an input stream, so **k** cannot begin until the **s1** load completes. It also depends on the preceding store: **k** uses **s2** as an output stream, so **k** cannot begin until the **s2** store completes. Each store depends on the preceding kernel; it stores stream **s2**, an output stream of **k**, so it cannot begin until **k** completes. The load in the next loop iteration depends on the preceding store; because the store and the following load both use buffer **buf1**, the load cannot begin until the store completes. Each operation depends on its predecessor, so the loop operations are fully serialized; none of them may run in parallel.

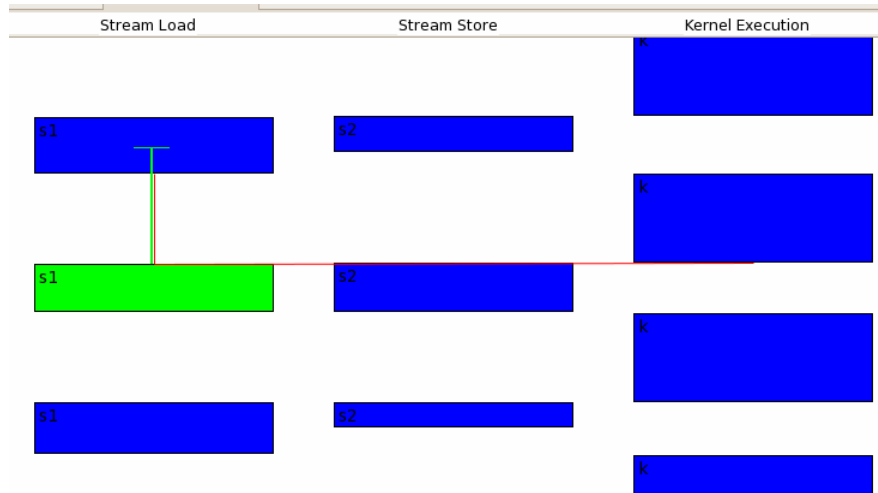
Each load, kernel, and store here is serialized, not overlapping with other operations. As a result, the stream operations are not densely packed; each resource goes unused at some times. To improve performance, the loop must be rewritten to remove dependencies.



A simple modification to the source to use separate buffers for input and output improves the loop's performance somewhat:

```
/* Case 2: load and store to separate buffers. */
for (i = 0; i < NSTRIPS; i++) {
    spi_load_block(s1, buf1, i * NBYTES, NRECS);
    k(s1, s2);
    spi_store_block(s2, buf2, i * NBYTES);
}
```

Execution now looks like this:



Each store still must depend on the preceding kernel, but the load in the next loop iteration no longer depends on the preceding store, so a store and a subsequent load now can occur simultaneously. However, the load in the next iteration depends on kernel **k**, because it cannot overwrite the kernel's input stream **s1** while **s1** is still in use by **k**. The packing is better than in the preceding case, but there is still room for improvement; each resource still goes unused at some times.

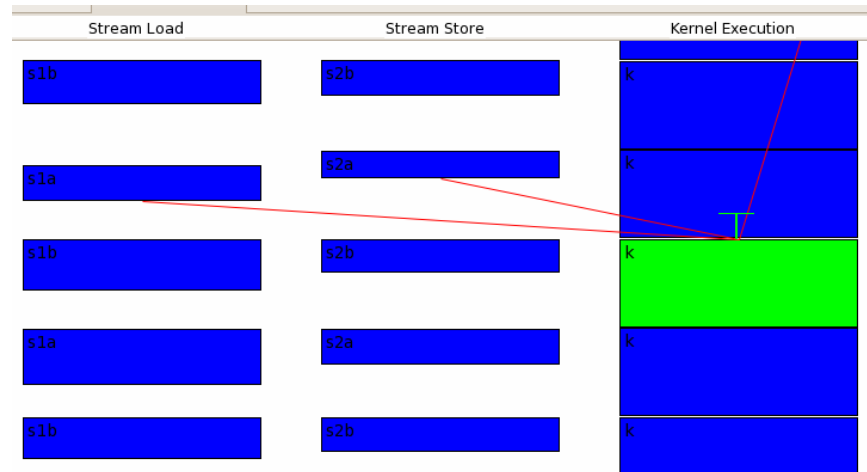
Note also that the load blocks here are somewhat longer than in the previous case. A load and a store run simultaneously, but now they contend for available DMA bandwidth, and the loads run slower than before as a result.



To further improve performance, use *double buffering* to avoid the delay caused by the dependency of each load on the preceding kernel execution. Double buffering essentially unrolls the loop once and performs successive loads and stores to separate streams rather than to the same stream, with loop operations rearranged. For clarity, the example below assumes the original loop count to be even; alternatively, the second load, second kernel call, and second store could each be conditionalized with `if (i + 1 < NSTRIPS)`.

```
/* Case 3: Double buffer. DPU-limited. */
for (i = 0; i < NSTRIPS; i += 2) {
    offset1 = i * NBYTES;
    offset2 = offset1 + NBYTES;
    spi_load_block(s1a, buf1, offset1, NRECS);
    spi_load_block(s1b, buf1, offset2, NRECS);
    k(s1a, s2a);
    k(s1b, s2b);
    spi_store_block(s2a, buf2, offset1);
    spi_store_block(s2b, buf2, offset2);
}
```

Execution of this loop looks like this after its early stages:



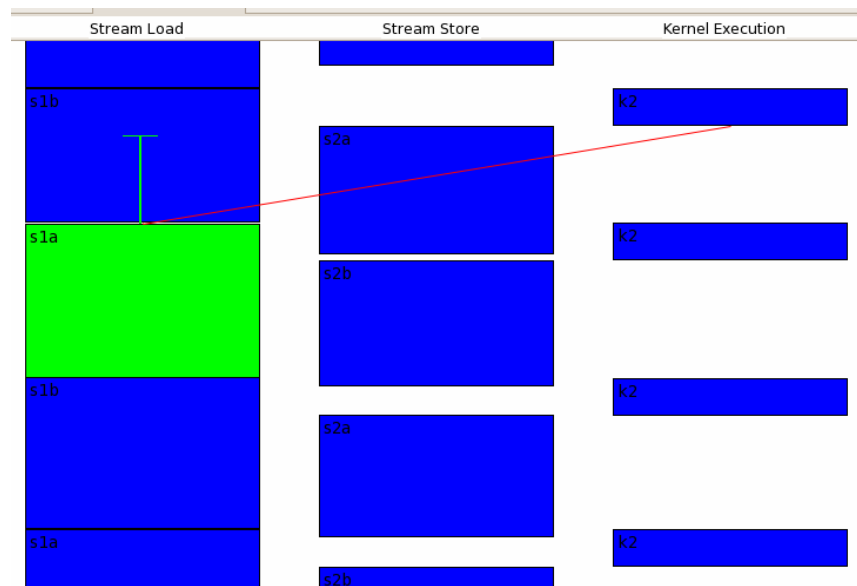
The kernel execution resource is fully packed here. The loop is now DPU-limited: it keeps the DPU fully occupied while the required loads and stores occur in parallel with kernel execution. Further performance optimization requires improving the performance of the kernel.



If kernel execution is faster than the required loads and stores, performance is DMA-limited rather than DPU-limited. Use the same code as above, but with larger loads and stores and a faster kernel:

```
/* Case 4: Double buffer. Same code but faster kernel; DMA-limited. */
for (i = 0; i < NSTRIPS; i += 2) {
    offset1 = i * NBYTES2;
    offset2 = offset1 + NBYTES2;
    spi_load_block(s1a, buf1, offset1, NRECS2);
    spi_load_block(s1b, buf1, offset2, NRECS2);
    k2(s1a, s2a);
    k2(s1b, s2b);
    spi_store_block(s2a, buf2, offset1);
    spi_store_block(s2b, buf2, offset2);
}
```

Execution performs continuous loads, so performance of the loop is now DMA-limited. Further performance improvement requires tuning the loads and stores, for example by combining pipelines or utilizing DRAM burst width fully.

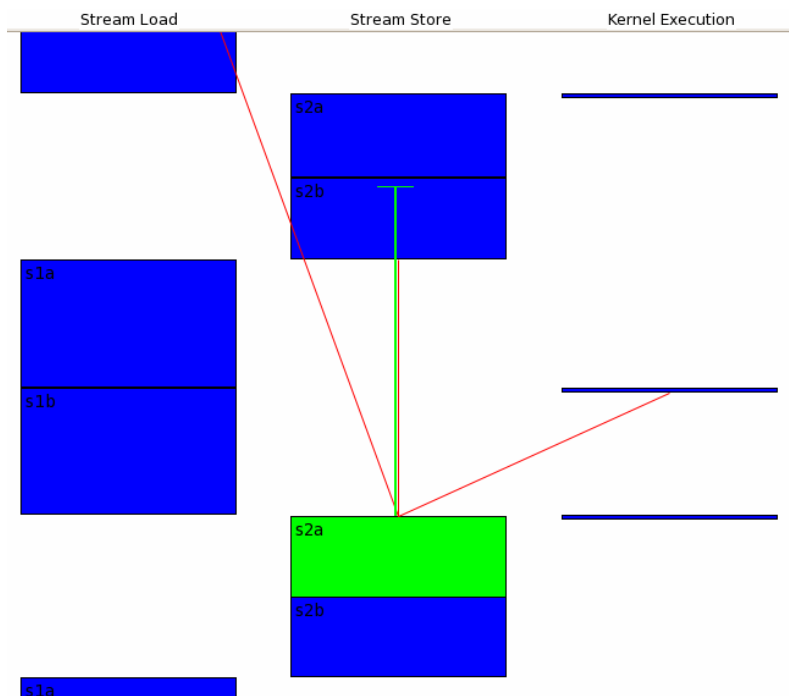




The stream controller can perform only a single indexed load or store at a time. In this case, double buffering is of limited benefit; it allows kernel execution and loads or stores to be parallelized, but the indexed loads and indexed stores cannot be parallelized. Consider code similar to the above example, but with **spi_load_index** and **spi_store_index** rather than **spi_load_block** and **spi_store_block**:

```
/* Case 5: Indexed loads and stores. */
for (i = 0; i < NSTRIPS; i += 2) {
    offset1 = i * NBYTES2;
    offset2 = offset1 + NBYTES2;
    spi_load_index(s1a, buf1, offset1, idx, 1, 1, NRECS2);
    spi_load_index(s1b, buf1, offset2, idx, 1, 1, NRECS2);
    k2(s1a, s2a);
    k2(s1b, s2b);
    spi_store_index(s2a, buf2, offset1, idx, 1, 1);
    spi_store_index(s2b, buf2, offset2, idx, 1, 1);
}
```

Because the stream controller cannot perform an indexed load and an indexed store simultaneously, the loads and stores are serialized in spite of the double buffering:



If the index stream is fairly simple (for example, selecting every other line of a rectangular array), it might be worth using block loads and stores and performing the work of the index stream in the kernel rather than using indexed loads and stores, because the block loads and stores can be parallelized but the indexed loads and stores cannot.



10.5.2 Dispatch delays

The time required for DSP MIPS to send an operation to the stream controller is typically about 500 nanoseconds for a load or store or about 1,000 nanoseconds for a kernel. Only one DSP MIPS sends stream operations but multiple resources can execute stream operations in parallel, so execution of each stream operation should take substantially longer than the time required to send it; if it takes longer to send each stream operation than to execute it, a pipeline will necessarily be dispatch limited. As a rule of thumb, stream operations should on average take at least twice as long to execute as to send: loads and stores should take at least $500 * 2 = 1,000$ nanoseconds to execute, and kernels should take at least $1,000 * 2 = 2,000$ nanoseconds. DSP MIPS can send stream operations ahead of execution, so a few stream operations can take less time to send than to execute, but then other stream operations should take longer to execute.

A stream command may have dependencies on other stream commands and resource requirements. If its dependencies and resource requirements are satisfied before a command is written, the delay between when its dependencies and requirements are satisfied and when it is written is called its dispatch delay. Dispatch delay has several components:

- *user code* time required by DSP MIPS code before the stream operation,
- *result wait* time required to wait for a kernel result,
- *dispatch wait* time for an available stream controller dispatch slot, and
- *send* time to send the operation.

spi_count and **spi_out** operations can introduce result wait time, as DSP MIPS must wait for a stream count or kernel scalar output. Wait time can sometimes be avoided by code rearrangement, moving **spi_count** or **spi_out** calls forward in the code to follow kernel calls or stream operations. This allows DPU execution to continue without waiting for the previous result.

Before:	After:
<pre>kernel1(..., x_out); x = spi_out(x_out); kernel2(...);</pre>	<pre>kernel1(..., x_out); kernel2(...); x = spi_out(x_out);</pre>

Dispatch waits are caused by stream controller hardware limits: the stream controller has queues of twelve load/store operation slots and four kernel slots, so DSP MIPS must wait to write a stream controller command if the required slots are unavailable. Reordering of loads, stores and kernel invocations can eliminate dispatch slot waits in some cases.



10.6 *Kernels*

A stream processor's DPU contains multiple arithmetic-logical units (ALUs). Its VLIW design allows it to execute operations on multiple ALUs in each clock cycle. The Stream compiler **spc** VLIW scheduler maps kernel source code to ALU instructions, scheduling on a basic block basis. Increasing the size of a basic block often provides the scheduler with opportunities to schedule code more efficiently.

A programmer should think carefully about algorithms when designing kernels. For example, different data layouts can lead to dramatically different performance results. A kernel that computes a digital filter, where each output is the sum of n filter coefficients times n inputs, might place successive input elements in successive lanes or alternatively might place groups of successive elements in each lane, with very different performance impact.

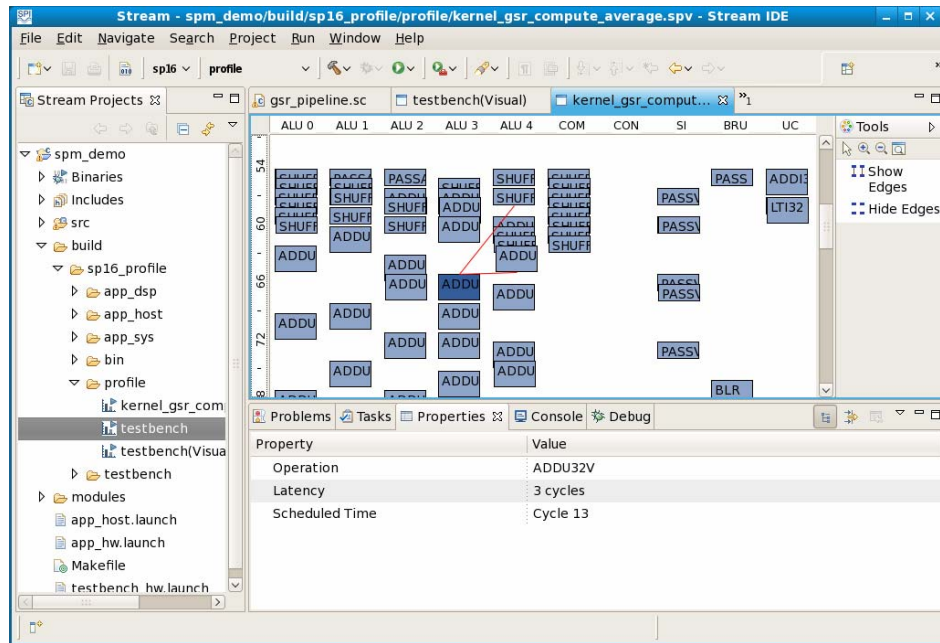
The performance of a kernel's inner loops typically determines its overall performance. This section presents some techniques to improve the performance of kernel inner loops.

10.6.1 Tune

A kernel performs with maximum efficiency if it uses all of the available ALUs to do useful work in all lanes. A kernel's VLIW utilization is the number of operations it executed divided by the maximum number of operations that could have been executed in the same number of cycles. SIMD utilization is the average percentage of lanes doing useful work.

Unlike sequential programming, where increasing the number of operations required by a computation degrades performance, in VLIW programming sometimes excess ALUs are available at no performance cost. To tune a kernel, improve its VLIW utilization and its SIMD utilization.

Double-clicking on a kernel execution rectangle in a profile visualization brings up a kernel visualization. For `gsr_compute_average`:



The vertical axis represents time in DPU cycles and the horizontal axis represents DPU resources (functional units, such as the five arithmetic-logical units ALU0 through ALU4). Clicking on any operation displays its properties in the Properties view; the Scheduled Time in the Properties view is basic-block-relative, so above the ADDU32V operation at cycle 66 is scheduled at cycle 13 of a basic block that starts at cycle 53; solid black horizontal lines separate basic blocks in the visualization. Hovering over any operation displays its dependencies on preceding operations. **Show Edges** and **Hide Edges** allow you to show or hide all dependency information. In a pipelined kernel, an operation may depend on an operation that occurs below it.

The VLIW scheduler in the compiler `spc` controls VLIW code generation, so the user does not have direct control over the generated kernel code, but the kernel visualization can provide a general picture of how efficiently the kernel uses DPU resources. Block unrolling or increasing the size of a basic block can provide the VLIW scheduler with more flexibility, leading to more efficient schedules.

10.6.2 Reduce critical path

The critical path of the inner block of a kernel loop, reported in the performance analysis [Tables](#) section and shown as a set of red arcs in a kernel visualization, gives a lower bound on how long the block must take to execute. Program performance can be dramatically improved by shortening the critical path, allowing better VLIW scheduling. Suppose a block computes `x`, `y`, and `z` that depend on `a`, `b`, and `c`:

```
x = (flag) ? a : b;
y = x + c;
z = 7 * y;
```

As coded above, `z` depends on `y`, which in turn depends on `x`; the program cannot compute `z` until the computation of `y` is complete. Rewrite the code as follows:

```
x = (flag) ? a : b;
```



```
y = ((flag) ? a : b) + c;  
z = ((flag) ? 7 * a : 7 * b) + 7 * c;
```

Here **y** and **z** depend only on **a**, **b**, and **c**, shortening the critical path for this code.

10.6.3 Remove control flow

Rewriting simple conditionals using kernel intrinsic operation **spi_vselect*** generates simpler code with predicated execution (no break of control flow):

Conditional:	Predicated:
<pre>for (i = 0; i < MAX; i++) { ... if (x < y) { min = x; } else { min = y; } ... if (x < y) { a = b; } ... }</pre>	<pre>for (i = 0; i < MAX; i++) { ... min = spi_vselect32(x < y, x, y); ... a = spi_vselect32(x < y, b, a); ... }</pre>

10.6.4 Software pipeline

Modulo software pipelining (SWP) rearranges the operations of a loop to construct a semantically equivalent loop with the shortest possible schedule length (in cycles), called the *minimum iteration interval* (MinII) of the loop. A single iteration of a pipelined loop may execute operations from several different iterations of the original loop. Execution of a pipelined loop suppresses some operations on some loop iterations to preserve the original loop semantics. Although the actual schedule length achieved by the scheduler is usually higher than MinII, reducing MinII usually reduces the achieved schedule length.

Software pipelining is the single most effective optimization for many inner loops. However, it can greatly increase compile time, and it does not always work on large loops. To software pipeline a loop, just add **#pragma pipeline** before the loop's opening brace.

10.6.5 Unroll

Loop unrolling makes a loop larger by replicating the loop body, thus providing the VLIW scheduler with opportunities to schedule the loop more efficiently. The **__repeat__** keyword described in section [__repeat__](#) above performs loop unrolling.

Kernel **gsr_compute_average** in **spm_demo/gsr_pipeline.sc** has a small loop that demonstrates the benefits of loop unrolling. It reads **BLOCK_WIDTH** (16 on SP16) RGB pixel color values from an input stream and accumulates sums of the RGB components:

```
for (i = 0; i < BLOCK_WIDTH; i += UNROLL) {  
    __repeat__(; UNROLL) {  
        spi_read(in_str, color);  
    }  
}
```




```
    r += spi_vshuffleu(0x0A0A0A02, color, 0);  
    g += spi_vshuffleu(0x09090901, color, 0);  
    b += spi_vshuffleu(0x08080800, color, 0);  
  }  
}
```

If compiled without unrolling (i.e., with **UNROLL=1**), this loop contains only 13 operations scheduled into 25 cycles, resulting in a dismal 5.4% ALU utilization; the function requires over 1.2 milliseconds to process file **sample.bmp**. If the same source is compiled with the loop unrolled eight times using **spc -D UNROLL=8**, the loop instead contains 62 operations scheduled into 32 cycles, for 16.8% ALU utilization; the function requires only about 0.37 milliseconds to process **sample.bmp**, a dramatic 3x performance improvement.

Loop unrolling has some of the benefits of software pipelining, and the two optimizations can be combined. Loop unrolling can increase VLIW instruction memory usage, so the programmer should pay attention to total VLIW instruction memory usage when unrolling. In general, the programmer can realize most of the benefits of unrolling by unrolling a loop two or four times. The programmer should try different unroll values and compare the resulting performance.



11 Glossary

The table below gives brief definitions of some common terms and acronyms. The second column identifies terms that are SPI-specific. For additional details on industry-standard terminology, see e.g. [Wikipedia](http://en.wikipedia.org)'s excellent explanations.

ADC		analog to digital converter
AHB		AMBA High-Performance Bus
ALSA		Advanced Linux Sound Architecture
ALU		arithmetic-logical unit
AMBA		Advanced Microcontroller Bus Architecture
API		application programming interface
application		the top-level software that implements a Stream program
AVC		Advanced Video Coding
basetype	SPI	the type of each data record in a stream
BOA		a high performance web server
CIF		common intermediate format; a 352x288 video format
CODEC		<u>code</u> r / <u>de</u> coder (or <u>comp</u> ressor / <u>de</u> compressor): program that manipulates stream data
component	SPI	a high-level data-driven computation module
CPB		coded picture buffer
CRAMFS		compressed ROM filesystem
D1		digital video format (PAL 720x576 MPEG-2, NTSC 720x480 MPEG-2)
DAC		digital to analog converter
DHCP		dynamic host configuration protocol
DLL		dynamically loadable library
DMA		direct memory access
DPU	SPI	data parallel unit: the part of a stream processor that executes kernels
DSP		digital signal processor or digital signal processing
DSP MIPS	SPI	one of two MIPS processors (System MIPS and DSP MIPS) on a stream processor
Eclipse		extensible open development platform
FIFO		first in / first out queue
GPU		general purpose processing unit: part of a stream processor that executes Stream code
GUI		graphical user interface
H.264		video compression standard (a.k.a. MPEG-4 Part 10, a.k.a. AVC)
HD		high definition video
HDK	SPI	hardware development kit
HDMI		high-definition multimedia interface: a digital audio/video interface
IC		integrated circuit
IDE		integrated development environment
I frame		intra frame (coded without reference to other frames)
in-lane	SPI	per lane; each lane can only access in-lane LRF data directly, not data from other lanes
I/O		input/output
IPC		interprocess communication
ISA		instruction set architecture
JFFS		journaling flash filesystem
JTAG		Joint Test Action Group: IEEE 1149.1 standard for debugging ICs and embedded systems
kernel	SPI	a DPU function to perform a computationally intensive operation on streams
lane	SPI	one of multiple identical arithmetic processors in a stream processor (8 on SP8, 16 on SP16)



LGPL		Gnu Lesser General Public License
Linux		operating system (the operating system on System MIPS)
LRF	SPI	lane register file: local storage for communicating stream data to/from a kernel
mb		macroblock
me		motion estimation
MinII		minimum iteration interval (of software pipelined loop)
MIPS		microprocessor without interlocked pipeline stages: a computer microprocessor architecture
MIPSSim		a simulator for MIPS programs
MPEG		Moving Picture Experts Group
MTD		memory technology device; a Linux subsystem for memory technology devices (e.g., flash)
module	SPI	an executable or library built from a spide project
NFS		network file system
NTSC		National Television Standards Committee; the television format used in the US and Japan
ORF	SPI	operand register file: local storage for each lane in a stream processor
OS		operating system
PAL		Phase Alternating Line; a television format used in Europe
PC		personal computer; also, program counter
PCM		Pulse Code Modulation: an encoding for digital audio data
P frame		predictively coded frame (coded with reference to other frames)
pipeline	SPI	top-level Stream function that performs stream operations
PPS		picture parameter set
project	SPI	a group of related files in a spide workspace
PSNR		peak signal-to-noise ratio
QCIF		quarter common intermediate format; a 176x144 video format
QP		quantization parameter
RAM		random access memory
RC		rate control
record	SPI	a structured data item that forms an element of a stream
RGB		an additive color model (red + green + blue)
RPC		remote procedure call
RTL		runtime library
RTP		real time transport protocol
RTSP		real time streaming protocol
SD		standard definition video
SDE		software development environment
SIMD		single instruction, multiple data; a variety of VLIW architecture design
SOC		system-on-a-chip
SORF	SPI	scalar operand register file: for communicating non-stream shared data to/from a kernel
SP16	SPI	a 16-lane stream processor from Stream Processors, Inc.
SP8	SPI	an 8-lane stream processor from Stream Processors, Inc.
spc	SPI	Stream Processors compiler
SPI	SPI	Stream Processors, Inc.; also Serial Peripheral Interface
spide	SPI	Stream Processors integrated development environment
SPM	SPI	Stream programming model
SPS		sequence parameter set
SRAM		static random access memory
Storm-1	SPI	a family of SPI processors, including SP16 and SP8
stream	SPI	a sequence of data records, each of identical type
striped	SPI	distributed across lanes, as with stream records
SUS		Single Unix Specification
SWP		software pipelining: rearranging loop operations to minimize iteration interval
System MIPS	SPI	one of two MIPS processors (System MIPS and DSP MIPS) on a stream processor



TCP		transfer control protocol
UART		universal asynchronous receiver/transmitter
V4L2		Video for Linux Two: a specification for Linux video
VBV		video buffering verifier (a.k.a. CPB)
vector	SPI	a variable with a separate value in each lane
VLIW		very large instruction word architecture
width	SPI	size in bits of each component of a basic data type (32, 16 or 8)
workspace	SPI	a directory containing spide metadata
YUV		a color model with one luma and two chrominance components.
YUV422		a YUV data format



12 Index

.bashrc, 8
/etc/bashrc, 8
__repeat__, 45, 121
2's complement representation, 12
access pattern, 32, 33
ADC, 123
address operator, 39
AHB, 123
aliasing, 28
ALSA, 123
ALU, 10, 36, 119, 123
ALU utilization, 110
AMBA, 123
API, 7, 11, 123
application, 10, 71, 123
application programming interface, 7, 11, 123
Application weight, 110
argc, 62
argc/argv, 25
arguments, 86, 92, 94
argv, 62
arithmetic, 43, 44
arithmetic-logical unit, 119, 123
array, 37, 40, 42
array_in, 31, 34
array_io, 31, 34
array_out, 31, 34
attributes, 31
AVC, 123
basetype, 123
bitmap file, 47, 48
bmp_bininfo_t, 49
BOA, 123
boolean, 37
buffer, 14, 48, 49
buffer allocation, 49
buffer clone, 21
buffer information, 15
buffer ownership, 21
buffer pool, 15
build, 84
build icon, 76, 84
built-in timer, 23, 73
byte ordering, 29
C boolean, 37
C type, 12
cache, 14, 28
cache coherency, 48
calloc, 15
cast, 39
CIF, 123
CODEC, 123
command, 16, 55
command handler, 14, 16, 56
command response, 16, 17
command statement, 25, 26
compiler, 10, 11, 12, 65
completion point, 104
component, 13, 123
Component API, 11
component instance, 13
compressed ROM filesystem, 123
cond_in, 31, 34
cond_out, 31, 34
conditional, 121
conditional operator, 39
conditional stream, 40, 41, 52
configure, 82
connection, 16
connection statement, 25, 26
Console view, 86
constant suffix, 36
constant type, 36
control flow, 39
conversion, 36
count, 33
CPB, 123
CRAMFS, 123
critical path, 103
customer support website, 8
D1, 123
DAC, 123
data access pattern, 33
data coherency, 28
data parallel unit, 10, 123
data type, 12, 36
data-parallel computations, 48, 49
debug icon, 76, 87
debug log, 23, 72
debug mode, 75, 90
Debug perspective, 87
debugging, 65, 87
dependence limited, 105, 109
dependency, 103, 104
dependency delay, 105
depth, 16
destroy function, 13, 58
development board, 9
development environment, 7, 123, 124
device i/o, 10, 48



- DHCP, 123
- digital signal processing, 7, 123
- direct memory access, 104, 123
- direction, 16
- directories, 9
- dispatch delay, 105
- dispatch limited, 105, 109
- dispatch point, 104
- dispatch time, 105
- division, 38
- DLL, 123
- DMA, 104, 123
- DMA bandwidth, 114
- DMA utilization, 109
- DMA-limited, 109, 116
- double buffering, 115
- DPU, 10, 22, 48, 49, 51, 123
- DPU basic type, 11, 12, 36
- DPU boolean, 37
- DPU intrinsic operation, 38
- DPU kernels summary, 110
- DPU kernels table, 111
- DPU utilization, 109
- DPU-limited, 109, 115
- DSP, 7, 123
- DSP MIPS, 10, 48, 123
- DSP MIPS / DPU synchronization, 48
- dynamic host configuration protocol, 123
- Eclipse, 123
- editor, 76
- enable mask, 23, 72
- endianness, 12, 29
- error log, 23, 72
- execute function, 13, 57
- execution breakdown table, 110
- execution model, 13
- execution requirement, 18, 55
- explicit conversion, 36
- fast functional mode, 75, 89
- FedoraCore 8.0, 8
- FIFO, 123
- file output component, 59
- file_in** component, 47, 71
- file_out** component, 47, 71
- filter*, 82
- firmware, 9
- flash filesystem, 123
- floating point, 12, 44
- fractional arithmetic, 44
- framebuffer, 22
- free**, 15
- functional mode, 47, 65, 75, 84
- gcc**, 8

- general purpose unit, 10, 123
- GPL, 9
- GPU, 10, 123
- graphical user interface, 123
- green screen removal component, 60
- gsr** component, 47, 71
- gsr** log, 72
- GUI, 123
- H.264, 123
- hardware development kit, 68
- HD, 123
- HDK, 68, 123
- HDMI, 123
- host configuration, 123
- host PC, 65
- host system, 8
- I frame, 123
- I/O, 123
- IC, 123
- icon, 76
- IDE, 7, 123, 124
- IEEE floating point format, 12
- image** statement, 25
- implementation alternatives, 48
- implicit conversion, 36
- import, 81
- in**, 31, 34
- index stream, 33
- indirection, 39
- initialization file, 24, 63, 72
- initialization function, 13
- in-lane, 123
- inline**, 34, 51
- inline kernel**, 34
- inner loop, 119
- input port, 16
- input/output, 123
- install.sh**, 8
- installation, 8
- instance, 13
- instance initialization function, 13, 55
- instance state, 18
- instance** statement, 25
- instruction memory usage, 110
- instruction memory use, 110
- instruction scheduling, 45
- instruction set architecture, 123
- int16x2**, 11, 36
- int32x1**, 11, 36
- int8x4**, 11, 36
- integrated circuit, 123
- integrated development environment, 7, 123, 124
- intra frame, 123



intrinsic operation, 34, 38, 42
IPC, 123
ISA, 123
issue point, 104
iteration interval, 124
JFFS, 123
journaling flash filesystem, 123
JTAG, 123
kernel, 10, 11, 34, 51, 123
Kernel API, 11, 28, 34
kernel basic types, 12
kernel function, 34
kernel intrinsic operation, 34, 42
keywords, 11
lane, 10, 123
lane register file, 10, 28, 124
LD_LIBRARY_PATH, 8
LGPL, 123
Linux, 8, 10, 124
littleendian, 12, 29
local_array_size, 29, 46
log, 23
logging level, 23
loop unrolling, 45, 121
LRF, 10, 28, 38, 49, 124
LRF address, 28, 31
LRF size, 28
macroblock, 124
macros, 12
main, 20, 61, 65, 71
make, 101
Makefile, 101
malloc, 15, 48
mb, 124
me, 124
member operator, 12
memory allocation, 15
memory technology device, 124
MinII, 121, 124
minimum iteration interval, 121, 124
MIPS, 8, 10, 124
MIPSim, 66
module, 76, 124
modulo arithmetic, 43
modulo software pipelining, 121
modulus, 39
motion estimation, 124
MPEG, 124
MTD, 124
multiplication, 44
NFS, 124
NTSC, 124
offset, 31
operand register file, 10, 38, 124
operating system, 124
operation packing, 113
operator, 38, 43
optimization, 103
ORF, 10, 38, 124
OS, 124
out, 31, 34
output port, 16
overflow, 43
owning instance, 21
P frame, 124
pack, 109
packed data types, 36
packing, 113
PAL, 124
PATH, 8
payload, 16
PC, 124
PCM, 124
performance, 74
performance analysis, 103
performance data, 103
performance optimization, 103
performance tables, 109
peripheral unit, 10
perspective, 76
picture parameter set, 124
pipeline, 124
Pipeline API, 11, 28
pipeline function, 28
pipeline summary, 109
pipelining, 121, 124
pointer dereference, 39
port, 15, 55
port direction, 16
PPS, 124
pragma, 45
pragma pipeline, 121
predefined macros, 12
predication, 121
predictively coded frame, 124
preprocessor macros, 12
priority, 19
priority level, 19
priority queue, 20
processing element, 22
processor synchronization, 14
profile, 105
profile data, 103
profile mode, 24, 75, 90, 91
program arguments, 86
program counter, 124
program development, 75
program trace, 24



programming model, 124
project, 76, 78, 124
properties function, 13, 55
provider, 22
PSNR, 124
QCIF, 124
QP, 124
quantization parameter, 124
RAM, 124
random access stream, 42
rate control, 124
RC, 124
realloc, 15
record, 10, 12, 124
record type, 12
release mode, 75, 95
remainder, 39
resource, 22
resource requirement, 103, 104
response, 16, 17
return, 34
RGB, 48, 124
rounding, 44
RPC, 124
RTL, 124
RTSP, 124
run configuration, 85, 92, 99
run icon, 76
saturation arithmetic, 43
scalar, 10, 33
scalar operand register file, 10, 124
scalar output variable, 33
scalar variable, 37
scp, 124
scheduler, 119
scheduling, 45
scheduling groups, 20
scheduling priority, 19
scp, 67
SD, 124
SDE, 124
seq_in, 31, 34
seq_out, 31, 34
sequence parameter set, 124
sequential stream, 40, 41, 52
Serial Peripheral Interface, 124
serialized operations, 113
shared memory, 14
signal-to-noise ratio, 124
SIMD, 10, 37, 124
simulation configuration, 109
simulator, 65, 66
size attribute, 31

sizeof, 39
SOC, 124
software development environment, 124
software pipelining, 45, 121, 124
SORF, 10, 124
SP16, 124
SP8, 124
spc, 10, 11, 12, 28, 65
SPI, 7, 124
spi_activate_exec_req, 19
spi_array_read, 40, 42
spi_array_write, 40, 42
spi_buffer_clone, 14, 21, 22
spi_buffer_close, 14, 21
SPI_BUFFER_FLAG_CACHED, 21
SPI_BUFFER_FLAG_READONLY, 21
spi_buffer_free, 14, 21
spi_buffer_get_info, 14, 15
spi_buffer_get_info_size, 14
spi_buffer_get_size, 14
spi_buffer_merge, 14, 21, 22
spi_buffer_new, 14
spi_buffer_open, 14, 21
spi_buffer_set_info, 14, 15, 49
spi_buffer_t, 14, 48
spi_cmd_free, 17
spi_cmd_get_desc, 17
spi_cmd_get_id, 17
spi_cmd_get_name, 17
spi_cmd_get_payload, 17
spi_cmd_get_payload_size, 17
spi_cmd_get_payload_type, 17
spi_cmd_get_response_payload_type, 17
SPI_CMD_PAUSE, 18
spi_cmd_send_response, 17
SPI_CMD_SET_PRIORITY, 19
SPI_CMD_START, 18
SPI_CMD_STOP, 18
spi_cmd_t, 17
spi_component_find, 13
spi_component_get_desc, 13
spi_component_get_name, 13
spi_component_get_provider, 23
spi_component_get_provider, 13
spi_component_get_version, 13
SPI_COMPONENT_NEW, 13, 16, 23, 54
spi_component_set_flags, 13
spi_component_set_resource_requirements, 22
spi_component_set_resource_requirements, 13
spi_component_t, 13
spi_cond_read, 40, 42
spi_cond_write, 40
spi_connect, 16



`spi_connection_get_depth`, 16
`spi_connection_get_name`, 16
`spi_connection_is_empty`, 16
`spi_connection_is_full`, 16
`spi_connection_pop`, 14, 16, 22
`spi_connection_push`, 14, 16, 22
`spi_connection_t`, 16
`spi_count`, 32, 33
`spi_delete_exec_req`, 19
`spi_eos`, 40
`SPI_EXEC_ALLOF`, 19
`SPI_EXEC_ALWAYS`, 19
`SPI_EXEC_ANYOF`, 19
`SPI_EXEC_FD_READ`, 19
`SPI_EXEC_FD_WRITE`, 19
`SPI_EXEC_NEVER`, 19
`SPI_EXEC_POOL`, 19
`SPI_EXEC_PORT_ALLOF`, 19
`SPI_EXEC_PORT_ANYOF`, 19
`spi_execution_requirement_t`, 18
`spi_export_port`, 16
`spi_fb_get_line_length`, 22
`spi_fb_get_pixel_type`, 22
`spi_fb_get_xres`, 22
`spi_fb_get_yres`, 22
`spi_fb_is_fb_available`, 22
`spi_fb_pool_new`, 22
`spi_get_buffer_heap_highwater`, 14
`spi_get_buffer_heap_size`, 14
`spi_get_component`, 13
`spi_get_log`, 23
`spi_get_name`, 13
`spi_get_pool`, 15
`spi_get_priority`, 20
`spi_get_state`, 18
`spi_get_time`, 24
`spi_get_timer`, 24
`spi_init_file`, 24
`spi_instance_context_t`, 13
`SPI_INSTANCE_STATE_PAUSED`, 18, 57, 60
`SPI_INSTANCE_STATE_RUNNING`, 18, 60
`SPI_INSTANCE_STATE_STOPPED`, 18
`spi_instance_state_t`, 18
`spi_instance_t`, 13
`SPI_LANES`, 12, 28, 48
`spi_load_*`, 14, 15, 21
`spi_load_block`, 32, 33, 61
`spi_load_index`, 32, 33, 52, 61
`spi_load_stride`, 32, 33
`spi_log`, 23, 72
`SPI_LOG_DEBUG`, 23, 24
`spi_log_dir`, 23
`SPI_LOG_ERROR`, 23
`spi_log_get_desc`, 23
`spi_log_get_enable_mask`, 23
`spi_log_get_name`, 23
`SPI_LOG_LEVEL_DEBUG`, 24
`spi_log_mask`, 23
`spi_log_new`, 23, 72
`spi_log_set_enable_mask`, 23
`spi_log_timestamps`, 23
`SPI_LRF_SIZE`, 28
`SPI_LRFSIZE`, 32
`spi_main`, 11, 65
`spi_new_connection`, 16
`spi_new_instance`, 13
`spi_out`, 32, 33
`SPI_PAYLOAD_STRING`, 55
`SPI_PEL_DSP_MIPS`, 25
`spi_pels_t`, 22
`spi_perm`, 35
`SPI_POOL_FLAG_GROW`, 15
`spi_pool_free`, 15
`spi_pool_get_avail_buffer_count`, 15
`spi_pool_get_buffer`, 15, 21
`spi_pool_get_desc`, 15
`spi_pool_get_name`, 15
`spi_pool_new`, 15, 21
`spi_port_get_connection`, 16
`spi_port_get_connection_coun`, 16
`spi_port_get_desc`, 16
`spi_port_get_dir`, 16
`spi_port_get_max_connection_count`, 16
`spi_port_get_name`, 16
`spi_portdir_t`, 16
`spi_provider_get_name`, 23
`SPI_PROVIDER_SPI`, 26
`spi_read`, 40, 41
`spi_register_cmd`, 16, 17, 55
`spi_register_exec_req`, 19, 55
`spi_register_port`, 16, 55
`spi_resources_t`, 22
`SPI_RESPONSE_ERRNO_FAIL`, 57
`SPI_RESPONSE_ERRNO_OK`, 24
`SPI_RESPONSE_ERROR_OK`, 57
`spi_response_free`, 17, 18
`spi_response_get_errno`, 17
`spi_response_get_payload`, 17
`spi_response_get_payload_size`, 17
`spi_response_get_payload_type`, 17
`spi_response_set_handler`, 16, 17, 18
`spi_response_strerror`, 17
`spi_response_t`, 18
`spi_schedgroup_component_find`, 13, 20
`SPI_SCHEDGROUP_NEW`, 20, 23
`spi_schedgroup_register_component`, 20, 23
`spi_schedgroup_set_controlled_resources`, 22
`spi_schedgroup_set_controlled_resources`, 20



spi_schedgroup_set_min_stacksize, 20
spi_schedgroup_set_processing_elements, 20
spi_send_cmd, 17
spi_set_priority, 19, 20
spi_set_state, 18, 57
spi_spm.h, 11
spi_spm_start, 11, 24, 62, 65
spi_spm_stop, 11
spi_store_*, 14, 15, 21
spi_store_block, 32, 33
spi_store_index, 32, 33
spi_store_stride, 33
SPI_TIMER_CMDHANDLER, 23, 73
SPI_TIMER_EXECUTE, 23, 73
spi_timer_get_desc, 24
spi_timer_get_name, 24
spi_timer_get_nanoseconds, 24
spi_timer_get_start_count, 24
spi_timer_get_total_nanoseconds, 24
SPI_TIMER_KERNEL, 23
SPI_TIMER_LOAD_DSP, 23, 73
spi_timer_new, 24
SPI_TIMER_SPM, 24, 73
spi_timer_start, 24
SPI_TIMER_STARTUP, 24, 73
spi_timer_stop, 24
spi_trace_is_enabled, 24
spi_trace_start, 24
spi_trace_stop, 24
spi_vabd8u, 52
spi_vshuffleu, 52
spi_write, 40, 41
spide, 24, 75, 76, 103, 124
SPM, 7, 11, 124
spm_demo, 47, 65
spperf, 24, 74, 103
sprun, 67
SPS, 124
spsim, 65, 66
SRAM, 124
Storm-1, 10, 68, 124
stream, 10, 11, 28, 49, 124
stream access function, 40
stream command, 104
stream command trace, 91
Stream compiler, 10, 11, 12, 65
stream controller, 104
stream count, 33
stream function, 28
Stream language, 11
stream operations, 113
stream operations table, 111
Stream perspective, 76
stream processor, 7, 10, 123
Stream programming model, 7, 11, 47, 124
stream size, 28, 32
stream type, 12, 40
stride, 33
striped, 41, 124
structured type, 12
substream, 31, 40
SUS, 124
SWP, 45, 121, 124
synchronization, 14
System MIPS, 10, 48, 123, 124
system-on-a-chip, 124
tables, 109
target, 92
TCP, 125
testbench, 47, 65
thread, 13
timers, 23, 73
toolbar, 76
toolset, 8
tracing, 24
tune, 109
two's complement arithmetic, 43
type, 12, 36
type attribute, 31
type conversion, 36
type width, 36
UART, 125
uint16x2, 11, 36
uint32x1, 11, 36
uint8x4, 11, 36
uncached buffer, 21
underflow, 43
unrolling, 121
user-defined type, 12
V4L2, 125
Variables pane, 89
VBV, 125
vec, 11, 37
vector, 125
vector variable, 37
view, 76
virtual machine, 8
visualization, 106
VLIW, 119, 125
VLIW scheduler, 119
VMware player, 8
web interface, 68
web server, 123
website, 8
width, 125
Wikipedia, 123



workspace, 76, 125
XML, 25

YUV, 125
YUV422, 125

© 2005-2009 by Stream Processors, Inc. All rights reserved.

For additional information or product support, please contact:

Stream Processors, Inc., 455 DeGuigne Drive, Sunnyvale, CA 94085-3890, USA

Telephone: +1.408.616.3338 • FAX: +1.408.616.3337 • Email: info@streamprocessors.com • Web: www.streamprocessors.com

This document contains advance information on SPI products, some of which are in development, sampling or initial production phases.
The information and specifications contained herein are preliminary and are subject to change at the discretion of Stream Processors, Inc.