

SPU Pipelining Assembler (SPA) User's Guide

Table of Contents

About This Document	4
Purpose	4
Audience and Prerequisites	4
Related Documentation	4
SPA Tool Version History	5
Document Version History	6
Typographic Conventions	7
Your Feedback	7
1 Introducing SPU Pipelining Assembler (SPA)	8
Application Overview	8
Current Limitations	8
Comparison to Hand-Optimized Code	8
Executable File	9
2 Software Pipelining	10
What Is Software Pipelining?	10
Why Software Pipelining?	10
Pipelunable Loops	13
Minimum Trip Count	13
Schedulers	14
Iterative Search	14
Modulo Schedulers	14
Live Too Long	14
Understanding Assembly Optimizer Output	14
SPA Output	14
Recurrence vs. Resource Constraints	15
Resource Usage	15
Important To Understand	15
Linear Schedule	16
Scheduler Output	16
Prolog and Epilog Collapse	16
3 Using SPA	18
Invoking SPA	18
Assembly Instructions	19
Restrictions on Instruction Arguments	19
Channel Mnemonics	19
Control Flow	19
Comments	20
Virtual Registers	20
Restrictions	20
Register Aliases	20
Annotating Register Declaration	20

Assembly Functions	20
.cfunc Directive	21
Restrictions on .cfunc Functions	21
.func Directive	21
Restrictions on .func Functions	21
Inputs, Outputs, and ABI Considerations	22
Pseudo-instructions	24
mov	24
gpo	24
mfpc	25
ilf32	25
il128	25
Shuffle Masks	26
Declaring Data	26
Aliasing	27
Restrictions	28
Externs	28
Enabling/Disabling Software Pipelining Algorithms	28
Linking	29
Optimization Level	29
4 Preprocessing	30
Defining Preprocessor Symbols	30
Restrictions	30
Conditional Compilation	30
Saving Preprocessed Output	31
Including Files	31
String Concatenation	32
Macros	32
Macros as Parameters to Macros	32
Macro Local Registers	33
Unique Identifier Operator for Macros	33
File Dependencies	33
5 Debugging	34
Troubleshooting	35
6 Frequently Asked Questions	36
Can I use SPA to optimize assembler generated by the C compiler?	36
What's the difference between ".s" and ".o" files generated by SPA?	36
Why can't I link my SPA function?	36
What should I do if I run out of registers?	36
Why don't my loop statistics match the loop kernel?	36
How can I eliminate my recurrence constraints?	37
7 Tips and Tricks	39
Maintaining Balance	39
Reading from Unaligned Memory	39
Easy Loop Unrolling	40

About This Document

Purpose

This document provides a user's guide for the SPU Pipelining Assembler (SPA), an assembly optimization tool.

Audience and Prerequisites

This document was written for PlayStation®3 developers who want to write more efficient and maintainable SPU assembly code for PlayStation®3 applications.

It is assumed that such developers have familiarity with the following:

- Assembly language
- The SPU assembly language specification (see "[Related Documentation](#)" below)
- General SPU operation
- C and C++

Related Documentation

SPU Assembly Language and ABI

You can obtain the following documents from the PlayStation®3 SDK Documentation package on the PlayStation®3 Developer Network website (<https://ps3.scedev.net>).

- For an overview of SPU Assembly Language, see *SPU Assembly Language Specifications for Cell OS Lv-2*.
- For information on the SPU ABI interface, see *Cell Broadband Engine™ and SPU ABI Specifications for Cell OS Lv-2*.

Edge Library

Core SPU processes in PlayStation®Edge Animation, DXT, and Post are written using SPA to achieve high throughput. Source code is provided and could be referenced as coding samples for SPA.

The following documents provide complete usage and reference information about the Edge library:

- *PlayStation®Edge Library Overview*
- *PlayStation®Edge Geometry Library: Quick Start*
- *PlayStation®Edge Geometry Library Reference*
- *PlayStation®Edge Geometry Library for Offline Tool: Reference*
- *PlayStation®Edge Animation Library Reference*
- *PlayStation®Edge Animation Library for Offline Tool: Reference*
- *PlayStation®Edge Zlib Library Reference*
- *PlayStation®Edge DXT Reference*
- *PlayStation®Edge LZMA Library Reference*
- *PlayStation®Edge LZO Library Reference*
- *PlayStation®Edge Post Library Reference*

SPA Tool Version History

This section provides the version history of the SPA executable.

Version Number & Date	Changes
v 1.4.6 build 1 March 1, 2010	Added “readoptional” register annotation.
v 1.4.5 build 4 January 28, 2010	Updated copyright notice.
v 1.4.5 build 3 January 19, 2010	Optimization level can now be set within files using new <code>.setoptim</code> directive.
v 1.4.5 build 2 December 14, 2009	Improved dead code elimination.
v 1.4.5 build 1 November 20, 2009	SPA now warns on reading from uninitialized registers and redundant writes to registers.
v 1.4.4 build 1 November 16, 2009	Improved code generation.
v 1.4.3 build 1 November 9, 2009	Function sections are now supported.
v 1.4.2 build 3 September 18, 2009	Added <code>il32</code> pseudo-instruction for loading 32-bit integer immediates. Added explicit even pipe <code>mov</code> pseudo-instruction (<code>mve</code>). Added explicit odd pipe <code>mov</code> pseudo-instruction (<code>mvo</code>).
v 1.4.2 build 2 September 8, 2009	Immediate offsets from labels are now possible with ILA.
v 1.4.2 build 1 September 3, 2009	Fixed bug where <code>--align</code> parameter was not reflected in text (<code>-s</code>) output.
v. 1.4.1 build 7 May 8, 2009	Improved feedback for invalid command-line parameters. Minor bug fixes.
v. 1.4.1 build 3 April 30, 2009	Turned off SSE2 support for wider compatibility.
v. 1.4.1 build 1 April 22, 2009	<code>O1</code> now disables local scheduling (useful if register pressure is too high on large/complex loops). New option to insert stack checking code upon register spills.
v. 1.4.0 build 1 April 2, 2009	Labels in data sections are now local by default (make them visible to the linker using the new <code>.global</code> directive). Warnings can now be treated as errors. <code>lqd</code> now accepts labels for the offset. A warning is now emitted if no pipelinable loops are found. Trying to process a file with no functions and no data now causes an error.
v. 1.3.4 build 1 January 28, 2009	Speed improvement if scheduling restarts after failed register allocation.
v. 1.3.3 build 1 January 26, 2009	Various backend speed improvements. The SMS scheduler is now disabled by default. It can be re-enabled at the command line.
v. 1.3.2 build 1 November 20, 2008	Macro names can now be passed as macro arguments. New <code>.localreg</code> preprocessor directive allows macro-local virtual registers to be declared.
v. 1.3.1 build 5 August 8, 2008	Added <code>.short</code> data declaration directive.
v. 1.3.1 build 4 July 9, 2008	<code>ilf32</code> pseudo-instruction can now accept either 1 or 4 immediates.
v. 1.3.1 build 3 June 25, 2008	Speed improvement for large loops with strong recurrence constraints.

Version Number & Date	Changes
v. 1.3.1 build 2 June 18, 2008	String concatenation and macro UID operators can now also be used in parameter lists when calling macros.
v. 1.3.1 build 1 June 4, 2008	The pre-processor now supports macros.
v. 1.3.0 build 3 May 8, 2008	Fixed bug that caused the preprocessor to count line numbers incorrectly.
v. 1.3.0 build 1 May 6, 2008	Files can be included using the <code>.include</code> directive.
v. 1.2.0 build 2 April 10, 2008	Data can now be declared anywhere outside function scope. Values declared with <code>.float</code> or <code>.word</code> can now be initialised with expressions.
v. 1.2.0 build 1 April 3, 2008	Initial release.

Document Version History

This section provides the version history of this document.

Version Number & Date	Changes
v 1.17 June 28, 2010	Added subsection about file dependencies. Updated SPA Tool version history.
v 1.16 March 1, 2010	Added subsection about register annotations. Updated SPA Tool version history.
v1.15 January 28, 2010	Added section about optimization directives. Added new tip to “Tips and Tricks” section. Updated SPA Tool version history.
v1.14 September 8, 2009	Updated SPA Tool version history.
v1.13 September 3, 2009	Updated SPA Tool version history.
v.1.12 June 24, 2009	Added a preface and rewrote chapter 1. Corrected typos and made other editorial improvements.
v.1.11 May 8, 2009	Updated SPA Tool version history.
v.1.10 April 2, 2009	Added information about making data labels global. Updated SPA Tool version history.
v. 1.9 January 28, 2009	Updated SPA Tool version history.
v. 1.8 November 20, 2008	Extended documentation for macro arguments. Added documentation for new <code>.localreg</code> directive Updated SPA Tool version history.
v. 1.7 August 8, 2008	Added documentation for new <code>.short</code> directive. Updated SPA Tool version history.
v. 1.6 July 9, 2008	Added “Troubleshooting” section of Debugging chapter. Updated section on <code>ilf32</code> pseudo-instruction.
v. 1.5 June 25, 2008	Updated SPA Tool version history.
v. 1.4 June 18, 2008	Updated SPA Tool version history.
v. 1.3 June 4, 2008	Added “Macros” section.
v. 1.2 May 6, 2008	Added “Minimum Trip Count” section. Added “Including Files” section.

Version Number & Date	Changes
v. 1.1 April 3, 2008	Added "Change List" chapter. Added "Tips and Tricks" chapter.
v. 1.0 April 3, 2008	Initial release of this document.

Typographic Conventions

This document uses the following typographic conventions:

Convention	Meaning
<code>fixed-width font</code>	Indicates programming code and literals, such as processing instructions, register names, data types, events, and file names. Also indicates function, structure, and macro names.
blue + underlined text	Indicates a hyperlink (blue displays in color printers or online only).

Your Feedback

Let us know what you think of SPA. We are always trying to improve the tool, and value your feedback. You can submit feedback to technical support or the public forum on the PlayStation®3 Developer Network website (<https://ps3.scedev.net>).

Areas of particular interest include:

- Generated code size
- Generated code speed
- Support for `brsl`/`brasl` instructions
- New features

1 Introducing SPU Pipelining Assembler (SPA)

Application Overview

The SPA is an assembly optimization tool that allows SPU low-level programmers to:

- Optimize critical loops (main code is expected to still be in C/C++)
- Easily write and maintain optimized assembly code
- Efficiently optimize code according to the assembly optimizer's feedback.

The SPA provides the following features:

- Scheduling:
 - Software pipelining (simple counted loops only)
 - "Classical" scheduling
- Register allocation:
 - Global register allocation
 - Move coalescing/copy propagation
- Debugging:
 - DWARF2 debugging support (allows mixed source/disassembly view in the debugger)

SPA outputs linkable object files that can be called from C/C++. As input, with a few restrictions SPA accepts the assembly syntax described in *SPU Assembly Language Specifications for Cell OS Lv-2*, in addition to some pseudo-instructions and preprocessing options. See Chapter 3, "[Using SPA](#)", for details.

Current Limitations

- Less than excellent (only "very good") at doing simple loops (decrementing counted loops).
- Does not support any form of indirect branching (except return).
- Generated code size is quite large.

Comparison to Hand-Optimized Code

For testing purposes, SPA was run on unoptimized versions of hand-optimized loops. Most (95%) of these loops were simple counted loops, and therefore could be handled by the tool with minimal changes.

Speed (SPA-optimized versus hand-optimized):

- Average = SPA was 0 to 5% slower.
- On loops above 50 cycles/iteration (the most critical cases) = SPA was 0 to 1% slower.
- Worst case = One 9 cycles loop could only be scheduled in 10 cycles.

Size:

- SPA was on average between 30% and 50% larger than hand-optimized reference code.
- We have already done some work to reduce code size (epilog collapse, and so forth).

Approximately 5% of the loops did not provide good results or could not be handled by SPA. Details:

- "Do" loops are handled by SPA but are not software-pipelined (because of speculative execution issues). Therefore they are only scheduled in a "classical" way as a regular compiler would do.
- Some other loops use constructs tuned for balance between code speed and size (mostly BI-based) and cannot be handled by SPA.

Executable File

The file required to use SPA is as follows.

File Name	Description
spa.exe	Win32 executable

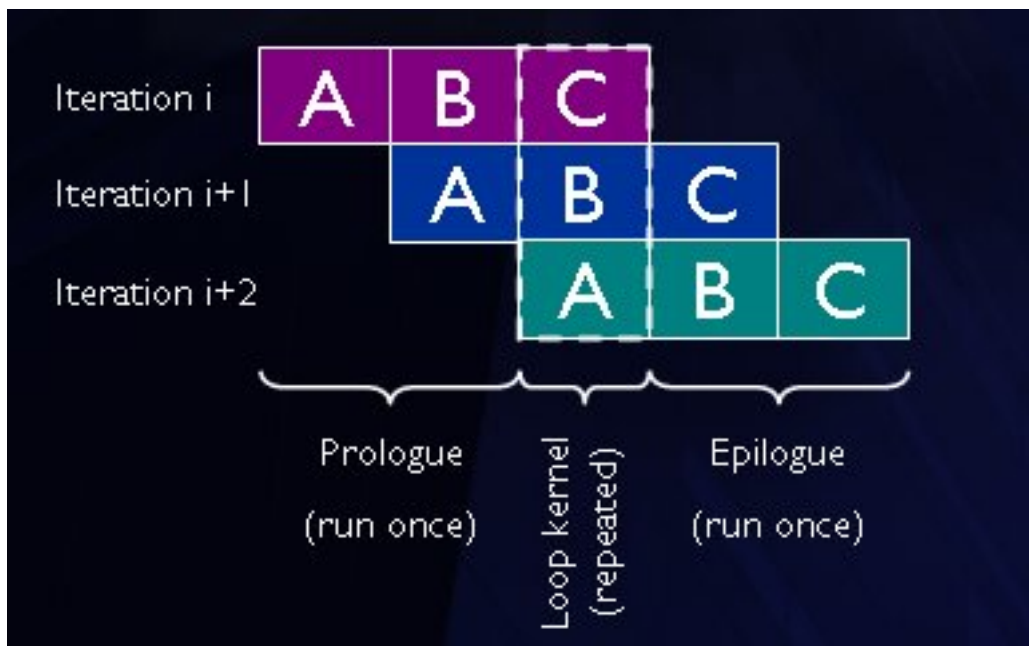
2 Software Pipelining

What Is Software Pipelining?

The idea is to create an “as optimal as possible” loop kernel by using instructions from other loop iterations when results from current are not yet available. It is necessary because of the long latencies of SPU instructions and is also the technique used by people writing good hand-optimized SPU assembly.

We are using the family of software pipelining algorithms that is most frequently used in production compilers: modulo scheduling.

Figure 1 Modulo Scheduling and Software Pipelining



Why Software Pipelining?

Consider the following very simple loop (unscheduled code – not supposed to do anything interesting):

```
(...)
myloop:
    ai      pA, pA, 0x40

    lqd     c0, -0x40(pA)    // odd 6 cycles
    lqd     c1, -0x30(pA)    // odd 6 cycles
    lqd     c2, -0x20(pA)    // odd 6 cycles
    lqd     c3, -0x10(pA)    // odd 6 cycles

    fa      a0, b0, c0       // even 6 cycles
    fa      a1, b1, c1       // even 6 cycles
    fa      a2, b2, c2       // even 6 cycles
    fa      a3, b3, c3       // even 6 cycles

    stqd    a0, -0x40(pA)    // odd 6 cycles
    stqd    a1, -0x30(pA)    // odd 6 cycles
```

```

    stqd    a2, -0x20(pA)    // odd 6 cycles
    stqd    a3, -0x10(pA)    // odd 6 cycles

    ai      $6, $6, -1
    brnz    $6, myloop
    (...)

```

This loop has 6 even instructions and 9 odd instructions. This means that ideally we want it to be scheduled in 9 cycles/iteration.

Unfortunately latencies do not allow this – at best one `stqd` can be scheduled 12 cycles after the initial load. The best classical (linear) schedule we can find is actually 19 cycles/iteration. SPA manages to generate a software pipelined version of this loop in 9 cycles/iteration.

Figure 2 Software Pipelined Version of “myloop” Example – 9 cycles/iteration (sustained)*

D Cycle	52:	PC:000b0	[P0:fa \$12,\$8,\$15]	PC:000b4	[P1:lqd \$17,0x3fc(<\$2>)]
D Cycle	53:	PC:000b8	[P0:fa \$11,\$7,\$18]	PC:000bc	[P1:lqd \$16,0x3fd(<\$2>)]
S Cycle	54:	PC:000c0	[P0:nopi ,0x086329]	PC:000c4	[P1:lqd \$15,0x3fe(<\$2>)]
D Cycle	55:	PC:000c8	[P0:ori \$3,\$4,0x000]	PC:000cc	[P1:stqd \$14,0x3fc(<\$4>)]
D Cycle	56:	PC:000d0	[P0:ori \$4,\$2,0x000]	PC:000d4	[P1:lqd \$18,0x3ff(<\$2>)]
D Cycle	57:	PC:000d8	[P0:ai \$6,\$6,0x3ff]	PC:000dc	[P1:stqd \$13,0x3fd(<\$3>)]
D Cycle	58:	PC:000e0	[P0:fa \$14,\$10,\$17]	PC:000e4	[P1:stqd \$12,0x3fe(<\$3>)]
D Cycle	59:	PC:000e8	[P0:ai \$2,\$2,0x040]	PC:000ec	[P1:stqd \$11,0x3ff(<\$3>)]
D Cycle	60:	PC:000f0	[P0:fa \$13,\$9,\$16]	PC:000f4	[P1:brnz \$6,0x3ffbc]
D Cycle	61:	PC:000b0	[P0:fa \$12,\$8,\$15]	PC:000b4	[P1:lqd \$17,0x3fc(<\$2>)]
D Cycle	62:	PC:000b8	[P0:fa \$11,\$7,\$18]	PC:000bc	[P1:lqd \$16,0x3fd(<\$2>)]
S Cycle	63:	PC:000c0	[P0:nopi ,0x086329]	PC:000c4	[P1:lqd \$15,0x3fe(<\$2>)]
D Cycle	64:	PC:000c8	[P0:ori \$3,\$4,0x000]	PC:000cc	[P1:stqd \$14,0x3fc(<\$4>)]
D Cycle	65:	PC:000d0	[P0:ori \$4,\$2,0x000]	PC:000d4	[P1:lqd \$18,0x3ff(<\$2>)]
D Cycle	66:	PC:000d8	[P0:ai \$6,\$6,0x3ff]	PC:000dc	[P1:stqd \$13,0x3fd(<\$3>)]
D Cycle	67:	PC:000e0	[P0:fa \$14,\$10,\$17]	PC:000e4	[P1:stqd \$12,0x3fe(<\$3>)]
D Cycle	68:	PC:000e8	[P0:ai \$2,\$2,0x040]	PC:000ec	[P1:stqd \$11,0x3ff(<\$3>)]
D Cycle	69:	PC:000f0	[P0:fa \$13,\$9,\$16]	PC:000f4	[P1:brnz \$6,0x3ffbc]
D Cycle	70:	PC:000b0	[P0:fa \$12,\$8,\$15]	PC:000b4	[P1:lqd \$17,0x3fc(<\$2>)]
D Cycle	71:	PC:000b8	[P0:fa \$11,\$7,\$18]	PC:000bc	[P1:lqd \$16,0x3fd(<\$2>)]
S Cycle	72:	PC:000c0	[P0:nopi ,0x086329]	PC:000c4	[P1:lqd \$15,0x3fe(<\$2>)]
D Cycle	73:	PC:000c8	[P0:ori \$3,\$4,0x000]	PC:000cc	[P1:stqd \$14,0x3fc(<\$4>)]
D Cycle	74:	PC:000d0	[P0:ori \$4,\$2,0x000]	PC:000d4	[P1:lqd \$18,0x3ff(<\$2>)]
D Cycle	75:	PC:000d8	[P0:ai \$6,\$6,0x3ff]	PC:000dc	[P1:stqd \$13,0x3fd(<\$3>)]
D Cycle	76:	PC:000e0	[P0:fa \$14,\$10,\$17]	PC:000e4	[P1:stqd \$12,0x3fe(<\$3>)]
D Cycle	77:	PC:000e8	[P0:ai \$2,\$2,0x040]	PC:000ec	[P1:stqd \$11,0x3ff(<\$3>)]
D Cycle	78:	PC:000f0	[P0:fa \$13,\$9,\$16]	PC:000f4	[P1:brnz \$6,0x3ffbc]

* Loop kernel shown in white.

Figure 3 Non-pipelined Version of “myloop” Example – 19 cycles/iteration*

S Cycle	58:	PC:000ac	[P0:fa \$15,\$10,\$13]
S Cycle	59:	PC:000b0	[P0:fa \$16,\$9,\$12]
S Cycle	60:	PC:000b4	[P0:fa \$17,\$8,\$11]
S Cycle	61:	PC:000b8	[P0:fa \$18,\$7,\$14]
R Cycle	62:		
R Cycle	63:		
S Cycle	64:	PC:000bc	[P1:stqd \$15,0x3fc<\$2>]
S Cycle	65:	PC:000c0	[P1:stqd \$16,0x3fd<\$2>]
S Cycle	66:	PC:000c4	[P1:stqd \$17,0x3fe<\$2>]
S Cycle	67:	PC:000c8	[P1:stqd \$18,0x3ff<\$2>]
S Cycle	68:	PC:000cc	[P1:brnz \$6,0x3ffc4]
S Cycle	69:	PC:00090	[P0:ai \$2,\$2,0x040] PC:00094 [P1:lnopi ,0x086329]
S Cycle	70:	PC:00098	[P0:ai \$6,\$6,0x3ff]
S Cycle	71:	PC:0009c	[P1:lqd \$13,0x3fc<\$2>]
S Cycle	72:	PC:000a0	[P1:lqd \$12,0x3fd<\$2>]
S Cycle	73:	PC:000a4	[P1:lqd \$11,0x3fe<\$2>]
S Cycle	74:	PC:000a8	[P1:lqd \$14,0x3ff<\$2>]
R Cycle	75:		
R Cycle	76:		
S Cycle	77:	PC:000ac	[P0:fa \$15,\$10,\$13]
S Cycle	78:	PC:000b0	[P0:fa \$16,\$9,\$12]
S Cycle	79:	PC:000b4	[P0:fa \$17,\$8,\$11]
S Cycle	80:	PC:000b8	[P0:fa \$18,\$7,\$14]
R Cycle	81:		
R Cycle	82:		
S Cycle	83:	PC:000bc	[P1:stqd \$15,0x3fc<\$2>]
S Cycle	84:	PC:000c0	[P1:stqd \$16,0x3fd<\$2>]
S Cycle	85:	PC:000c4	[P1:stqd \$17,0x3fe<\$2>]
S Cycle	86:	PC:000c8	[P1:stqd \$18,0x3ff<\$2>]
S Cycle	87:	PC:000cc	[P1:brnz \$6,0x3ffc4]
S Cycle	88:	PC:00090	[P0:ai \$2,\$2,0x040] PC:00094 [P1:lnopi ,0x086329]
S Cycle	89:	PC:00098	[P0:ai \$6,\$6,0x3ff]
S Cycle	90:	PC:0009c	[P1:lqd \$13,0x3fc<\$2>]

* Loop kernel shown in white.

To understand what the scheduler did, let's look at the loop kernel, in the debug output:

```
(...)
Local_c0de0001:
    fa      $12,$8,$15;    /* +2 */    lqd      $17,-64($2) /* +1 */
    fa      $11,$7,$18;    /* +2 */    lqd      $16,-48($2) /* +1 */
    nop     ;              /* +1 */    lqd      $15,-32($2) /* +1 */
    ori     $3,$4,0;        /* +2 */    stqd     $14,-64($4) /* +2 */
    ori     $4,$2,0;        /* +1 */    lqd      $18,-16($2) /* +1 */
    ai      $6,$6,-1;       /* +2 */    stqd     $13,-48($3) /* +2 */
    fa      $14,$10,$17;    /* +1 */    stqd     $12,-32($3) /* +2 */
    ai      $2,$2,64;       /* +2 */    stqd     $11,-16($3) /* +2 */
    fa      $13,$9,$16     /* +1 */
Local_c0db0001:                brnz     $6,Local_c0de0001 /* +2 */
(...)
```

- We have 3 iterations in parallel, to be able to achieve 9 cycles/iteration.
- The numbers in comments refer to which iteration this instruction actually belongs to.
- You can also see that the scheduler automatically splits some live ranges that would have been too long by inserting some moves (“ori x,y,0”) without affecting the schedule performance.

All the necessary compensation code (not shown here), like prolog and epilog (if necessary), has been automatically generated.

Pipelizable Loops

There are a few important things to understand:

- Only do-while loops whose number of iterations can be predicted are pipelizable.
- Canonical “while” loops are not pipelizable!
- In case of nested loops, only the innermost loop will be pipelined.
- Because the loop kernel runs multiple iterations at the same time, it may require some specific code to start the loop (called **prolog**) and to end the loop (called **epilog**). Therefore the code size will be increased. (For certain loops, SPA may be able to “collapse” the prolog or epilog, thereby avoiding the increase in code size. See [“Prolog and Epilog Collapse”](#) later in this chapter).
- This structure enforces a certain minimum number of iterations; if the loop may potentially be run for less than this number of iterations, an additional non-software pipelined copy of the loop must be provided. This will also cause the code size to increase.

Typically, the only kind of loop that the assembly optimizer will try to pipeline in its current implementation is a decrementing loop like this:

```
(...)  
label:  
    (...)  
    ai count, count, -kSomeConstant  
    (...)  
    brnz count, label  
    (...)
```

Important

- The assembly optimizer does not perform loop transformation; it is the user’s responsibility to transform the loop so that it can be software pipelined.
- If the assembly optimizer prints no software pipelining feedback, it means that that the loop is considered to be non-pipelizable.

Minimum Trip Count

As discussed above, a non-pipelined copy of the loop will sometimes need to be generated to cope with a small number of iterations. For example, if a loop has three iterations in parallel, SPA may need to generate a regular non-pipelined loop to handle cases where it should run for 2 or 1 iterations. That is what SPA displays in this feedback:

```
(...)  
generating non-pipelined loop for 1<=trip_count<3  
(...)
```

Now, if we specify that we know that this loop will never run for less than 4 iterations, we can make the code much smaller by avoiding creating this non-pipelined copy. We annotate the branch instruction with \$mintrip as follows:

```
// we guarantee that the loop will be at least executed 3 times  
brnz count, cffLoop : $mintrip<3>
```

And now the feedback displays the following:

```
(...)  
not generating non-pipelined loop since trip count >=3  
(...)
```

Schedulers

Iterative Search

Modulo schedulers operate by trying to find a schedule for a given initiation interval (“*ii*”), which means for a given number of cycles per iteration.

Roughly the process is:

- Determine *min_ii* (more detail below)
- Determine *max_ii* (classical non-pipelining scheduler)
- While (*ii* < *max_ii*)
 - Try to find a valid schedule for this *ii*
 - If failed, increment *ii* (exponential search)

The more difficult the loop is to schedule, the more attempts the scheduler will make.

Modulo Schedulers

Because scheduling is an NP-hard problem, we run multiple modulo scheduling algorithms on the same loop, and select the best result. These algorithms are:

- A derivative of Iterative Modulo Scheduling (IMS). The scheduler is slow (heavily backtracking) but, by far, produces the best results.
- An improved Swing Modulo Scheduler (SMS), run twice, with alternatively top-down and bottom-up priorities. It is a very poor scheduler, but sometimes (approximately 5% of our test cases) it does produce a better result.

Live Too Long

When we schedule the loop kernel, we ignore all anti-dependencies and as a result sometimes registers live for longer than one iteration. This is what the assembly optimizer refers to as “live too long”. SPA addresses this problem by trying to insert “moves” to new registers in empty slots in the schedule to “cut” the lifetime of variables. If we cannot schedule these moves in an empty slot, scheduling is aborted for this initiation interval.

Understanding Assembly Optimizer Output

SPA Output

Here is a typical output on a loop. We turn on verbose software pipelining feedback using the `--verboseswp` flag.

The first part shows some general statistics about the loop you want to pipeline:

```
~$ spa --verboseswp -o temp.o test.spa
loop stats:
    resmii : 27 (*)          (resource constrained)
    recmii : 2              (recurrence constrained)
resource usage:
    even pipe : 17 inst. (63% use)
                  FX[15] WS[2]
    odd pipe  : 27 inst. (100% use) (*)
                  LS[18] SH[8] BR[1]
misc:
    linear schedule = 38 cycles (for information only)
(...)
```

The second part provides the actual results of the scheduler:

```
(...)  
software pipelining:  
    valid schedule for ims@27 (pipelined, 3 iterations in parallel)  
    schedule failed for sms-bu@27  
    valid schedule for sms-td@27 (pipelined, 3 iterations in parallel)  
    best pipelined schedule = 27 cycles (pipelined, 3 iterations in parallel)  
(...)
```

Recurrence vs. Resource Constraints

The assembly optimizer first tries to determine the **minimum initiation interval**. This is the minimal number of cycles between the start of two consecutive iterations of the loop.

- This information is very important for the user to optimize and balance their code.
- It is also a lower bound used to avoid attempting impossible schedules (to increase speed).

There are two types of constraints:

- Resources: related to the number of instructions per iteration.
- Recurrence: related to dependency cycles on values that are reused in the next iteration; for example if a loop is accumulating results with “FMA”, you will never be able to run in less than 6 cycles/iteration due to the latency of an FMA.

Typical output:

```
(...)  
loop stats:  
    resmii : 27 (*)          (resource constrained)  
    recmii : 2              (recurrence constrained)  
(...)
```

You **usually** want to be resource limited and not recurrence-constrained.

Resource Usage

The assembly optimizer lists the instructions for each pipeline, and shows which one is the limiting factor.

```
(...)  
resource usage:  
    even pipe : 17 inst. (63% use)  
                FX[15] WS[2]  
    odd pipe  : 27 inst. (100% use) (*)  
                LS[18] SH[8] BR[1]  
(...)
```

Important To Understand

This loop has:

- 17 even instructions/iteration
- 27 odd instructions/iteration

Therefore, it can **never** run in less than 27 cycles/iterations; it is odd-pipe resource constrained.

- Balancing this code can help you achieve better performance
- Even non-obvious optimizations can help; for example if we could replace 3 odd instructions by 7 even instructions in that example, we would have 24 even/24 odd -> minimum initiation interval would be 24 instead of 27!

Linear Schedule

We also have a more classical non-pipelining scheduler; it is used for the rest of the code, but we also run it for pipelined loops:

- To determine if they are worth pipelining and how many cycles/iterations to make before using the non-pipelining (linear) scheduler instead
- As an interesting statistic to see the benefit of pipelining this loop (can be useful when you have to do some trade-off for code size for example)

In our example the best linear (non-pipelined) schedule found by SPA runs in 38 cycles/iteration. It is 1.4 times slower than the theoretical ideal performance that we can achieve for this loop by pipelining it.

```
(...)  
misc:  
    - linear schedule = 38 cycles  
(...)
```

In this example, if the assembly optimizer cannot find a pipelined schedule at 37 or less cycles/iterations, it will give up and use a regular linear scheduler instead.

Scheduler Output

As explained earlier, the schedulers manage to find a valid schedule for the minimal iteration interval:

```
(...)  
software pipelining:  
    valid schedule for ims@27 (pipelined, 3 iterations in parallel)  
    schedule failed for sms-bu@27  
    valid schedule for sms-td@27 (pipelined, 3 iterations in parallel)  
    best pipelined schedule = 27 cycles (pipelined, 3 iterations in parallel)  
(...)
```

Actually, in this example:

- IMS found a schedule for 27 cycles/iteration, with 3 iterations in parallel.
- The bottom-up SMS implementation failed to find a valid schedule.
- The top-down SMS implementation found a schedule for 27 cycles/iteration, with 3 iterations in parallel (the same as the IMS implementation).

Prolog and Epilog Collapse

As an attempt to reduce code size, the assembly optimizer will automatically try to eliminate the loop epilog (that means executing the loop iteration more than originally expected).

- If the extra instructions are not safely speculatively executable (for example, a store), the epilog cannot be pipelined.
- If the assembly optimizer cannot tweak the loop counter (for example for some shift-based odd loops) epilog collapse will fail.

```
(...)  
software pipelining adjustments:  
    collapsed epilog stage 1/2 (removed 42 instructions)  
    collapsed epilog stage 2/2 (removed 35 instructions)  
    couldn't collapse prolog stage 1/2 (1 instructions)  
    couldn't collapse prolog stage 2/2 (8 instructions)  
    not generating non-pipelined loop since trip count >=1 (1)  
(...)
```


Note that:

- Epilog collapsing also brings another interesting code-size reduction benefit: you usually do not need the non-pipelined copy of the loop anymore
- Prolog collapsing does not currently succeed with most of our loops, as the implementation is quite simple, and in most of our SPU code, it would involve speculative execution of stores
- Most typical loops can have the epilog collapsed, unless they return a value

3 Using SPA

Invoking SPA

SPA generates linkable object files that can be called from C/C++. A typical command line is:

```
~$ spa -o Debug/test.o test.spa
```

- -o is the output (ELF format) linkable file (.o).
- The source filename must be the last element of the command line (after all options).

To see all of the options that SPA supports, call the executable without any arguments. You will receive information similar to what is shown in Table 1.

Table 1 SPA Options

Arguments	Description
<filename>	Input filename.
Required Flags	
-o <filename>	Set output object file. <filename> Name of the output object file.
Optional Flags	
-O<unsigned int>	Set optimization level. <unsigned int> Optimization level (default: 2)
--ims <bool>	Enable Iterative Modulo Scheduling. <bool> Enable IMS scheduler? (default at optimization level 2: true)
--sms <bool>	Enable Swing Modulo Scheduling. <bool> Enable SMS scheduler? (default at optimization level 2: true)
--asmout (-s) <filename>	Set output assembly file (not recommended because you will lose debug information). <filename> Name of the output asm file
--header <filename>	Specify a text file containing a header for the output assembly file. <filename> Name of header text file.
--singlecol <bool>	Output assembly file instructions in a single column. <bool> Use single column? (default: false)
--swpindent <bool>	Indicate SWP loop iterations using indents. <bool> Use indenting? (default: false)
--preout (-p) <filename>	Save preprocessed assembly to a file. <filename> Filename to save preprocessor output to.
--align <uint>	Alignment of .text section in object file. <uint> Alignment in bytes (default: 16)
--dwarf2 <bool>	Generate DWARF2 debug information. <bool> Enable DWARF? (default: true)
--fetch <bool>	Enable experimental fetch/issue control code. <bool> Enable? (default: false)
--define (-D) <string>	Predefine an integer symbol. <string> symbolname, or symbolname=value (default value: 1)

Arguments	Description
--definesingle (-Df) <string>	Predefine a single-precision float symbol. <string> symbolname, or symbolname=value (default value: 1.0)
--definedouble (-Dd) <string>	Predefine a double-precision float symbol. <string> symbolname, or symbolname=value (default value: 1.0)
--quiet	Enable quiet mode.
--nopicwarn	Disable warnings relating to PIC.
--noswp	Disable software pipelining.
--verboseswp	Enable verbose software pipelining feedback.

Assembly Instructions

SPA accepts the assembly syntax described in *SPU Assembly Language Specifications for Cell OS Lv-2* (listed in the “[About This Document](#)” section), with some added pseudo-instructions (for example, `il128` and `mov`) and preprocessing options to assist programmers. There are also some restrictions.

Restrictions on Instruction Arguments

Standard SPU Assembly allows immediate values for instruction arguments to be encoded in the following ways:

- An immediate constant value or expression
- A PC relative address (the current program counter is expressed by a dot (“.”) symbol)
- A symbolic label address

SPA places the following restrictions on these encodings:

- Immediate constant values or expressions may not be used in branch instructions. Branches **MUST** reference a symbolic label
- PC relative addresses cannot be used for any instruction

Channel Mnemonics

SPA supports all of the channel mnemonics described in *SPU Assembly Language Specifications for Cell OS Lv-2*.

Control Flow

Only “simple” control flow instructions, branching to symbols inside the same assembly function, are currently handled.

Table 2 Supported and Unsupported Branch Instructions

Instruction	Supported/Unsupported
BR	Supported
BRA	Supported
BRSL/BRASL	Not currently supported
BISL/BISLED	Not supported
BI	Not supported (but SPA will automatically add a “BI \$0” (return) at the end of your function)
BIZ/BIHZ/BINZ/BIHNZ	Not supported
IRET	Not supported

Comments

Both C and C++ style comments are accepted.

Virtual Registers

SPA allows you to use virtual registers that will be allocated by the back end; they must be declared with the `.reg` directive prior to their use; virtual registers and physical registers can be mixed:

```
.reg myResult
ai    myResult, $0, 1    // myResult=$0+1
```

Restrictions

Virtual registers are NOT automatically renamed on write; a virtual register will map to the same physical register every time it is used:

- The backend assumes the programmer knows what they are doing.
- You must be careful with this as you will add dependencies/scheduler constraints if you write to a register used elsewhere. A typical example of bad practice is to recycle the same tmp virtual register in multiple different situations.

Note that if you want the scheduler to produce good code, you have to use virtual registers. Using code that is already register-allocated will result in **very** poor schedules since the scheduler will have to consider unnecessary anti-dependencies imposed by physical registers. Feeding spu-gcc output to SPA is a very bad idea as it will produce very bad schedules.

Register Aliases

The `.reg` directive also allows you to refer to a specific physical register using an alias:

```
.reg myResult=$5          // myResult is now an alias for register $5
ai    myResult, $0, 1      // $5=$0+1
```

Keep in mind the previous warning about manually allocating registers – register allocation should be left up to SPA if at all possible. Be careful not to create unnecessary anti-dependencies by manually allocating registers using this syntax.

Annotating Register Declaration

When declaring a virtual register, you can add a “readoptional” annotation. Such an annotation will cause SPA to *not* issue warnings if a value written to the register is never read. (This can be useful if you have a macro that declares many standard constants, but you know that not all of the constants will be used and you are relying on SPA’s dead-code elimination to remove unused constants). The “readoptional” syntax is as follows:

```
.reg myreg1: readoptional // don't warn if a value written to myreg1 is not read
.reg myreg2, myreg3: readoptional // no warnings for myreg2 or myreg3
```

Assembly Functions

Assembly functions are declared using either the `.func` or `.cfunc` directives. In either case, a return instruction (i.e., `‘bi $0’`) is automatically added to the end of the function; you must not have it in your code.

.cfunc Directive

If you are calling your SPA function from C or C++, the recommended method of declaring your function is the `.cfunc` directive:

```
.cfunc int MyFunction1(qword* pInput1, int count)

    // When you declare a function using .cfunc, SPA will automatically
    // create virtual registers called "pInput1" and "count" and
    // initialize them with the parameter values

    // Do some work and put a useful value in workRegister
    .reg workRegister
    ...

    // As this function returns non-void, SPA will automatically create
    // a virtual register called "@result". We must put the return
    // value in this register
    mov @result, workRegister

.endfunc
```

When you declare a function using `.cfunc`, the function will use the standard SPU ABI (described in *Cell Broadband Engine™ and SPU ABI Specifications for Cell OS Lv-2*), and SPA will automatically put the parameters into virtual registers of the same name. If the return type of your function is not “void”, SPA will also declare a virtual register called “@result”. You must put the return value of your function in this register.

Restrictions on .cfunc Functions

Only data types that fit into a register may be passed as parameters to SPA functions. SPA does not try to interpret or validate the types specified in a `.cfunc` declaration – it assumes that each parameter will be in a register. If you want to pass a variable that will not fit into a register to an SPA function, you should pass a pointer to it instead.

.func Directive

Alternatively, you can use the `.func` directive. In this case you simply declare `.func` and the function name:

```
.func MyFunction2

    // When you declare a function using the .func directive, you
    // are responsible for all ABI handling!

    // Assembly function code goes here...

.endfunc
```

Restrictions on .func Functions

When you declare a function using `.func`, you are responsible for all handling the ABI. In other words, parameter passing, return values and live input/output registers must be explicitly dealt with by the programmer. This is useful if you wish to define your own ABI. See the following section for details.

Inputs, Outputs, and ABI Considerations

If you want to follow the standard SPU ABI (in other words, if you are interfacing with C or C++), it is recommended that you declare your functions using the `.cfunc` directive because SPA will handle all of the ABI details for you. Alternatively, if you declare your function using the `.func` directive, you must handle the ABI as described in this section.

A brief reminder of the SPU ABI's register usage is provided in Table 3.

Table 3 SPU ABI Register Usage

Instruction	Description
\$0	Return address
\$1	Initial stack pointer
\$3-\$79	Parameters
\$3	Return value

The following directives are available to declare inputs and outputs:

- `.input` – inputs of the function
- `.output` – outputs of the function

Inputs/outputs can only be physical registers, not virtual registers. Arguments can be:

- Single register. For example: `.input $1`
- Register range. For example: `.input $3-$6`
- Multiple arguments on the same line. For example: `.input $1, $3-$6`

If you are calling from assembler and you declare your function using `.func`, you can describe your own ABI with the `.input/.output` directives; otherwise you can use `.cfunc`. When a function is declared using `.cfunc`, the following occurs:

- Registers \$1 and \$80 to \$127 are automatically declared as inputs and outputs.
- Each of the live parameter registers are automatically declared as inputs.
- If SPA runs out of registers during register allocation, it will automatically generate save/restore code for as many non-volatile ABI registers as required to complete register allocation. It does this iteratively, so register allocation may take several attempts before a suitable schedule is found.
- If your function returns a value, SPA will automatically declare a virtual register called “@result” where you must put the return value.
- `.cfunc` will only cause SPA to automatically save/restore registers ≥ 80 if they are needed by the register allocator. If you wish to use any of these registers **explicitly** in your code, you must still save and restore them manually.

None of this occurs for functions declared using `.func`.

Regardless of whether `.func` or `.cfunc` is specified, SPA always assumes that \$0 is an implicit input that contains the return address.

ABI Examples

Consider the following C function:

```
float TestFunction(qword a, qword b, int c, int d)
{
    // at this point :
    // - a is in $3
    // - b is in $4
    // - c is in the preferred slot of $5
    // - d is in the preferred slot of $6
```

```
// The function does its work here...
// ...

return someFloat; // someFloat is returned in the preferred slot of $3
}
```

An equivalent SPU assembly function could be declared like this (specifying the standard SPU ABI manually):

```
.func TestFunction

// Parameters
.input $3, $4, $5, $6

// Return value
.output $3

// SPU ABI:
// because these physical registers are declared as inputs and outputs,
// and are not used explicitly in the code, the register allocator will
// never map any virtual register to one them since it will result in
// a live-range conflict.

// Preserve stack pointer:
.input $1
.output $1

// Preserve registers above 80 (these are specified as non-volatile
// by the standard SPU ABI):
.input $80-$127
.output $80-$127

.reg someFloat
// The function does its work here...
// ...

// Return value should be put in register $3
mov $3, someFloat

.endfunc
```

A function declared with `.func` **must** specify its outputs and its inputs; **not doing this will result in broken register allocation**. Failing to define the output registers, if the function is returning any value, will also result in the “dead code removal” pass being too aggressive.

Alternatively, because this function is using the standard SPU ABI, we could declare it using `.cfunc`:

```
.cfunc float TestFunction(qword a, qword b, int c, int d)

// Because the function was declared with .cfunc, SPA knows
// that it uses the standard SPU ABI, so the following will happen:
// - Registers $1 and $80 to $127 will automatically be
//   declared as inputs and outputs. If the register allocator
//   runs out of registers, it will automatically save and
//   restore as many of $80-$127 as required
// - The parameter registers (in this case $3, $4, $5, $6) are
//   automatically declared as inputs
// - Virtual registers called "a", "b", "c" and "d" will
//   automatically be declared and initialized with the values from
```

```
// $3, $4, $5, $6
// - The return type of the function is non-void so a virtual
// register called @result will automatically be declared. We
// must put the return value in this register

.reg someFloat
// The function does its work here...
// ...

// Return value should be put in register @result
mov @result, someFloat

.endfunc
```

As you can see, there is much less work for the programmer when functions are declared using `.cfunc`, and you get the benefit of automatic save/restore on non-volatile registers if the register allocator runs out of space.

Pseudo-instructions

mov

The pseudo-instruction `mov` allows you to copy values from one register to another. For example:

```
mov $3, $4 // Copy contents of register $4 to register $3
```

SPA translates `mov` into an `ori` with a zero immediate.

gpo

The pseudo-instruction `gpo` retrieves the current PIC offset and puts it in the preferred slot of the target register. This allows position-independent code to be written. For example:

```
// Declare two quadwords in the data section
.data
MYDATA:
.quad 0xAAAAAAAA_BBBBBBBB_CCCCCCCC_DDDDDDDD
.quad 0xEEEEEEEEE_FFFFFFFF_00000000_11111111

.func PicExample
// ABI
.input $1,$3,$80-$127
.output $1,$3,$80-$127

.reg val, addr, pc

.reg picoffset

ila addr, MYDATA
gpo picoffset // Preferred slot now contains PIC offset
lqx val, addr, picoffset

// Function does work on value...
(...)
.endfunc
```


mfpc

The pseudo-instruction `mfpc` puts the address on the next instruction in the preferred slot of the specified register. For example:

```
mfpc $3 // Put the address of the next instruction in the preferred slot of $3
```

ilf32

The pseudo-instruction `ilf32` allows you to load a (single precision) floating-point immediate into a register. For example:

```
ilf32    $5, 3.141592f // Load 3.141592f into register $5
```

The assembly optimizer will translate `ilf32` into either a single `ilhu` or an `ilhu` and an `iohl`. If the lower 2 bytes of the IEEE754 representation of the immediate are zero, then only an `ilhu` will be generated, otherwise an `iohl` will be generated too.

For example, if SPA sees the following code:

```
ilf32    $5, 1.0f // IEEE754 representation is 0x3F800000
```

It will translate this into the following instruction:

```
// We only need to load the upper 2 bytes as the lower two bytes are zero
// (ilhu automatically zeros the lower halfword)
ilhu     $5, 0x3F80
```

However, if we try to load a different constant:

```
ilf32    $5, 0.63661977236758f // IEEE754 representation is 0x3F22F983
```

SPA cannot fit this immediate load into a single instruction, so it will automatically generate two instructions:

```
ilhu     $5, 0x3F22 // Load upper halfword
iohl     $5, 0xF983 // Load lower halfword
```

Alternatively, `ilf32` can be called with four floating-point immediates. These will be loaded into each of the word elements of the target register. For example:

```
ilf32    $5, 1.0f, 2.0f, 3.0f, 4.0f // Load floats into each word of register
```

causes the four supplied values to be loaded into the respective word slots of the register \$5. This usage of `ilf32` will generate a constant which is stored in memory and an `lqr` instruction to load it.

il128

The pseudo-instruction `il128` allows you to load a quadword immediate into a register. This means that you do not need to declare your constants (such as shuffle masks) externally; you can use the `il128` instruction. For example, you can write:

```
il128    shuf0, 0x80800001808002038080040580800607
il128    shuf1, 0x80800809_80800a0b_80800c0d_80800e0f
```

Obviously, this constant does not fit in a 32-bit SPU instruction; what the assembly optimizer will do is:

- Store a constant in memory (just after the code of your constant)
- Load it with an `LQR` instruction

The second example demonstrates that numbers can have underscores (_) in them as separators, to aid readability.

`il128` can also be used to easily create and load shuffle masks. See the following section for details.

Shuffle Masks

The `il128` pseudo-instruction also supports the usual shuffle-mask syntax:

- 4 character strings ("xxxx") for word shuffles
- 8 character strings ("xxxxxxxx") for half-word shuffles
- 16 character strings ("xxxxxxxxxxxxxxxxxxxx") for byte shuffles
- A... being elements of the first quadword
- a... being elements of the second quadword

For example:

```
// Word shuffle, will be: 0x000102031415161708090A0B1C1D1E1F
il128 shuf2, "AbCd"
// Half word shuffle, will be: 0x000112130405161708091A1B0C0D1E1F
il128 shuf3, "AbCdEfGh"
// Byte shuffle, will be: 0x001102130415061708091A0B1C0D1E0F
il128 shuf4, "AbCdEfGhIjKlMnOp"
```

The syntax has been extended to support SPU-specific shuffle extensions, as described in Table 4.

Table 4 Shuffle Mask Extensions

Char	Mask Value	Byte
0	0x80	0x00
1/X/x	0xC0	0xFF
8	0xE0	0x80

Declaring Data

You may declare to be added to the generated object file. This must be done outside of function scope using the `.data` and `.rodata` directives. Labels declared in data sections are local by default. If you wish these labels to be visible to the linker, use the `.global` keyword.

As shown in Table 5, several data type directives are supported.

Table 5 Supported Data Type Directives

Directive	Description
<code>.quad <quadword list></code>	Declares a comma-separated list of quadwords. The syntax for specifying the quadword is the same as for <code>il128</code> .
<code>.byte <byte list></code>	Declares a comma-separated list of bytes.
<code>.short <short list></code>	Declares a comma-separated list of (16-bit) shorts.
<code>.word <word list></code>	Declares a comma-separated list of words.
<code>.float <float list></code>	Declares a comma separated list of single-precision floating-point values.
<code>.align <power of two bytes></code>	Pads the current data section until it is aligned to the specified byte alignment. The value specifies the power of two bytes (for example, specifying <code>.align 3</code> will achieve 8-byte alignment).

The following example shows these directives in use:

```
.rodata // Put the following data in the read-only data section
SHUFFLEMASK1:
.quad "AbCdEfGhIjKlMnOp"
MYDATA1:
.word 0xAABBCCDD, 0xEEFF0011
.global MYDATA1 // Make the label MYDATA1 global (so it is visible to the linker)

.data // Put the following data in the data section
MYDATA2:
.byte 0xAB, 0xCB
.align 3 // Pad the data section to 8 byte alignment
MYDATA3:
.float 1.0f, 3.14159265f
```

Aliasing

The SPA backend assumes that no aliasing occurs and it is up to the programmer to annotate any memory accesses that may alias. Instructions are annotated by assigning them to a “memory group”. Any instructions that are in the same group are assumed to alias. The following code demonstrates the syntax:

```
// Declare "memory groups" for specifying aliasing
.memgroup  mg_row0, mg_row1, mg_row2, mg_row3
loop:

// Load matrix 1
.reg  matla_row0, matla_row1, matla_row2, matla_row3
// aliasing with the STQD for row 0
lqx   [mg_row0] matla_row0, pWorldBase_row0, pWorldOffset
// aliasing with the STQD for row 1
lqx   [mg_row1] matla_row1, pWorldBase_row1, pWorldOffset
// aliasing with the STQD for row 2
lqx   [mg_row2] matla_row2, pWorldBase_row2, pWorldOffset
// aliasing with the STQD for row 3
lqx   [mg_row3] matla_row3, pWorldBase_row3, pWorldOffset

// The function does some work here...
(...)

// aliasing with the LQX for row 0
stqd  [mg_row0] matR_row0, -0x40(pOutputWorld)
// aliasing with the LQX for row 1
stqd  [mg_row1] matR_row1, -0x30(pOutputWorld)
// aliasing with the LQX for row 2
stqd  [mg_row2] matR_row2, -0x20(pOutputWorld)
// aliasing with the LQX for row 3
stqd  [mg_row3] matR_row3, -0x10(pOutputWorld)

// Loop
brnz  count, loop
```

Here the load from `pWorldBase_row0+pWorldOffset` and the write to `-0x40(pOutputWorld)` may be accessing the same memory location. There are similar dependencies for the other rows. We make these dependencies explicit by putting all instructions that may access the same area of memory into the same memory group. The backend then knows to respect these dependencies when performing optimization and can avoid data hazards.

Restrictions

Store instructions can be assigned to up to one memory group. Load instructions can be assigned to up to two memory groups (in the format [memgroup1, memgroup2]).

Externs

You can interface with global data symbols that are defined in another module (such as C or Assembler). They will be resolved at link time. In the .spa file:

```
.func SomeFunction
    (...)
    .extern SomeGlobal
    (...)
    .reg val
    (...)
    lqr val, SomeGlobal
    (...)
.endfunc
```

You can also access a table declared externally the same way. (It is important to understand that using ILA to reference a label will make your code not position-independent):

```
.func SomeFunction
    (...)
    .extern MyTable
    (...)
    .reg tableBase
    (...)
    ila    tableBase, MyTable // not position independent
    (...)
    lqx val, someOffset, tableBase
    (...)
.endfunc
```

Enabling/Disabling Software Pipelining Algorithms

SPA currently supports two software pipelining algorithms: Swing Modulo Scheduling and Iterative Modulo Scheduling. The normal way to enable or disable them is via command-line arguments. However, there are situations where you may want to use different settings for individual files or functions within a file. You can do this using the .setswp directive:

```
.setswp sms 1    // Turn on Swing Modulo Scheduling
.setswp ims 0    // Turn off Iterative Modulo Scheduling
```

If .setswp is specified within function scope, the setting only applies to that function (changing the same setting multiple times within one function will generate a warning and only the first setting will be applied). If .setswp is specified at file scope, the setting will persist for all functions encountered in the file from that point onward (or until the setting is changed again using another .setswp directive)

Linking

For an SPA function declaration of the form:

```
.func MyFunction

    // ...

    .reg myParam0, myParam1
    mov myParam0, $3 // Move parameter 0 into myParam0
    mov myParam1, $4 // Move parameter 1 into myParam1

    // ...
.endfunc
```

Assuming that the standard ABI is used, the corresponding C++ prototype would be:

```
extern "C" void MyFunction(qword myParam0, qword myParam1);
```

(Because we are using the standard ABI, the same function could alternatively be declared in SPA using the `.cfunc` directive. See the “[Assembly Instructions](#)” and “[Inputs, Outputs, and ABI Considerations](#)” sections for more details).

Optimization Level

The usual method of setting the optimization level for a file is from the command line, but it can also be set within a file using the `.setoptim` directive. The optimization level set with `.setoptim` overrides any value set at the command line. A `.setoptim` directive at file scope affects all functions encountered from that point in the file onward. A `.setoptim` directive at function scope affects only that function.

The syntax is as follows:

```
.setoptim 0 // turn off optimisation
```

Note: Setting the optimization level changes multiple settings (such as which scheduling algorithms are enabled). Therefore, be aware that `.setoptim` may change settings already made by directives such as `.setswp`.

4 Preprocessing

SPA has an assembly preprocessor that conditionally removes/includes code, strips out comments, and performs basic symbol substitution.

Defining Preprocessor Symbols

Preprocessor symbols can be defined using the `.set` keyword. For example the preprocessor will turn the following code:

```
// Create preprocessor symbol
.set MEMORY_OFFSET, 0x40

// The preprocessor will substitute this symbol for its integer value
lqd rt, MEMORY_OFFSET(ra)
```

into this:

```
lqd rt, 0x40(ra)
```

Note that comments and preprocessor directives have also been removed, but line numbers have been preserved. Preprocessor symbols can be defined in terms of previously defined symbols and arithmetic operators:

```
.set SYMBOL1, 0xFF
.set SYMBOL2, SYMBOL1+0xFF00
.set SYMBOL3, (SYMBOL1+SYMBOL2) * 2
```

Symbols may be redefined as many times as you wish:

```
.set MEMORY_OFFSET, 0x40 // Define symbol

lqd rt, MEMORY_OFFSET(ra) // 0x40 will be substituted here

.set MEMORY_OFFSET, 0x20 // Redefining previously declared symbol - this is legal

lqd rt, MEMORY_OFFSET(ra) // 0x20 will be substituted here
```

Restrictions

Mixed-mode arithmetic is not permitted.

Conditional Compilation

The conditional directives shown in Table 6 are supported.

Table 6 Supported Preprocessor Directives

Preprocessor Directives
<code>.set <symbol>, <expression></code>
<code>.if <expression></code>
<code>.ifndef <symbol></code>
<code>.else</code>
<code>.elif <expression></code>
<code>.endif</code>

Expressions may be composed of previously defined preprocessor symbols, integers and arithmetic, logical and relational operators. The directives can be nested to any depth, and work in the standard way, as shown by the following example:

```
.set SYMBOL0, 0
.set SYMBOL1, 1
.set SYMBOL2, 0x2

    // .ifdef (with a defined symbol)
.ifdef SYMBOL0
    // The following directive will get passed to the assembly optimizer
    .reg regA
.endif
    ai regA, regA, 0x4

    // .ifdef (with an undefined symbol)
.ifdef UNDEFINED_SYMBOL
    // The following instruction will NOT get passed to the assembly optimizer
    ai regA, regA, 0x4
.endif

    // .if (with an expression that evaluates to true)
.if 3*(5-14)
    // The following directive will get passed to the assembly optimizer
    .reg regD
.endif
    ai regD, regD, 0x4

    // .if (with an expression that evaluates to true)
.if SYMBOL1+3
    // The following directive will get passed to the assembly optimizer
    .reg regE
.endif
    ai regE, regE, 0x4
```

Saving Preprocessed Output

It can sometimes help with debugging to view the preprocessed version of your source file. To specify which file to save the preprocessed output to, use the `--preout` flag:

```
D:\Tests\SPA>spa -o MyTest.spa.o --preout MyTest.spa.pre MyTest.spa
```

The shorthand `-p` is also valid:

```
D:\Tests\SPA>spa -o MyTest.spa.o -p MyTest.spa.pre MyTest.spa
```

Line numbers are preserved between the source and preprocessed files.

Including Files

The preprocessor allows you to include other files in your SPA source by using the `.include` directive. The syntax is as follows:

```
.include "path_relative_to_dir_of_current/file.spa"
.include <path_relative_to_command_line_specified_include_dirs/file.spa>
```

If the include filename is surrounded by quotes, SPA will interpret the file path as relative to the path of the file currently being parsed. If the filename is surrounded by angled brackets, SPA will interpret the path of the file relative to the include paths that are specified in the command line using the `-I` parameter. The include paths will be searched in the order they are specified at the command line.

Files may also be specified using an absolute path (such as on Windows the format would be `c:\path\to\file.spa`). In this case, surrounding the filename with quotes or angled brackets is equivalent.

String Concatenation

The preprocessor supports string concatenation using the `##` operator. The operands may be literal strings or preprocessor symbols.

Macros

Macros can make your code easier to maintain. They are declared using the `.macro/.endmacro` directives as follows:

```
// Macro to declare a virtual register and initialize it with PI
.macro LoadPi( regName )
    .reg regName
    ilf32regName, 3.141592653589f
.endmacro
```

Once a macro has been declared it can be used almost anywhere, including the body of other macro declarations. The above macro would be called like so:

```
.func MyFunction

    // ...

    // Declare a virtual reg called PiReg and initialize it with PI
    LoadPi(PiReg)

    // ...
.endfunc
```

The preprocessor will detect any macro calls and replace the call with the macro body using the supplied parameters.

Macros can have zero or more parameters. A macro that does not have parameters is declared as follows:

```
.macro DeclareStdAbiRegs
    .input $1, $80-$127
    .output $1, $80-$127
.endmacro
```

Macros as Parameters to Macros

You may pass the name of a macro to another macro as an argument (in a similar manner to passing function pointers as arguments in C). This enables the behavior of macros to change according to which other macros they are passed as arguments.

Macro Local Registers

When declaring a macro, you may wish to have temporary local registers. This can be achieved using the `.localreg` preprocessor directive. Each register declared using `.localreg` will only be accessible within the body of that macro. Each time the preprocessor instantiates a macro, it generates a unique name for each register declared using `.localreg`, which avoids unnecessary anti-dependencies and means macros can be called multiple times without name collisions.

```
.macro MyMacro
    .localreg tmpRegA // Generate unique, local register name
    .localreg tmpRegB // Generate unique, local register name
    // ...
.endmacro
```

Unique Identifier Operator for Macros

SPA assigns a unique numerical identifier to each instantiation of a macro. When declaring a macro, you may refer to this identifier using the `@@` operator. Combining this operator with the string concatenation operator allows you to declare macros that generate unique variable names each time they are called. For example, you could mimic the behavior of the `.localreg` directive using the macro `uid` and string concatenation operators (although it is recommended that you use `.localreg` if possible, because it makes macros easier to read and maintain).

File Dependencies

SPA can emit dependency information for the object file that it is generating. This is enabled with the `-emitdeps` command-line argument. The dependencies are output to a file in the same directory as the object file; the file name is the same as the object file, but with a `.d` file extension rather than `.o`. (If the object file does not have a `.o` extension, the dependency file name is just the object file name with `.d` appended).

All of the paths in the dependency file are relative to the current working directory. The format of the dependency file is based on the format expected by the `make` utility.

If you would like to also include a “phony target” for each of the dependencies (similar to the `-MP` option on some compilers), use the `-emitdeps` option instead. Note that a phony target is not created for the source file that is passed to SPA at the command line.

5 Debugging

SPA supports the DWARF2 debugging format. This enables you to use mixed source and assembly view when your function is running in a debugger. You can also see which hardware registers your virtual registers are currently assigned to and their value by using the “Locals” view. To enable debugging, use the `--dwarf2` flag. Figure 4 shows an SPA function viewed in a debugger:

Figure 4 SPA Function Viewed in a Debugger

The screenshot displays a debugger interface with two main panels. The top panel shows the source code of an SPA function, with line numbers 258 to 284. The code includes comments and assembly instructions. A yellow arrow points to line 274, which is highlighted in green. The bottom panel shows the "Locals" window, which lists variables and their values. The variables are: pScim, pScre, pWins, re, scim, scimtmp, and scre. The values are shown in hexadecimal and decimal. The types are listed as vector uint32_t. The addresses are also shown.

Source [Disassembly] [SPU [0x4000100:0x100]]

```

258
259 // Load
260 .reg scre, scim, re, im
261 .reg pScreTmp, pScimTmp
262 lqx re, pRe, offset
008870 388703A1 lqx r033,r007,r028 ODD
263 lqx im, pIm, offset
008874 38870420 lqx r032,r008,r028
264 lqx scre, pScre, offset
008878 38870497 lqx r023,r009,r028 ODD
265 lqx scim, pScim, offset
00887C 38870516 lqx r022,r010,r028
266
267 // Scale
268 .reg scretmp, scimtmp
269 fma scretmp, re, gain, scre
008880 E2A7D097 fma r021,r033,r031,r023 04 (00008878) REG
270 fma scimtmp, im, gain, scim
008884 E287D016 fma r020,r032,r031,r022 EVN
271
272 // Select valid words to store
273 .reg validmask_max, validmask_min, validmask
274 cgt validmask_max, index_max, indices
008888 4804881D cgt r029,r018,r018
275 cgt validmask_min, indices, index_min
00888C 48044913 cgt r019,r018,r017 EVN
276 and validmask, validmask_max, validmask_min
008890 1824CE9E and r030,r029,r019 01 (0000888C) REG
277 selb scretmp, scre, scretmp, validmask
008894 84454B9E selb r034,r023,r021,r030 01 (00008890) REG EVN
278 selb scimtmp, scim, scimtmp, validmask
008898 84650B1E selb r035,r022,r020,r030
279
280 // Store
281 stqx scretmp, pScre, offset
00889C 288704A2 stqx r034,r009,r028 01 (00008894) REG
282 stqx scimtmp, pScim, offset
0088A0 28870523 stqx r035,r010,r028 ODD
283
284 brnz count4, winloop
_local_c0db0000000010002
0088A4 217FF49A brnz r026, local_c0de0000000010002 ?HINT

```

Locals [SPU [0x4000100:0x100]]

Name	Value	Type	Address
pScim	\$r010	vector uint32_t	\$r010
[0]	0x00022000	uint32_t	\$r010.x
[1]	2054	uint32_t	\$r010.y
[2]	2304	uint32_t	\$r010.z
[3]	50048	uint32_t	\$r010.w
pScre	\$r009	vector uint32_t	\$r009
pWins	\$r005	vector uint32_t	\$r005
re	\$r033	vector uint32_t	\$r033
scim	\$r022	vector uint32_t	\$r022
scimtmp	\$r020	vector uint32_t	\$r020
[0]	2.10261E-04	uint32_t	\$r020.x
[1]	1.38375E-04	uint32_t	\$r020.y
[2]	9.89503E-06	uint32_t	\$r020.z
[3]	-3.14842E-07	uint32_t	\$r020.w
scre	\$r023	vector uint32_t	\$r023
scretmp	\$r021	vector uint32_t	\$r021
chiffUi	\$r040	vector uint32_t	\$r040

Processes Locals [S... Callstack [... Watch [SP...]

Important: If a virtual register in your source code does not appear in the “Locals” view of the debugger, it means that SPA has optimized out all instructions that use the virtual register. Note that this may even happen for virtual registers automatically declared by SPA, such as “@result”.

Troubleshooting

If you find that the debugging information does not appear when you step through your SPA code, check the following:

- Have you enabled debugging information when building your code with SPA? (Specify “`--dwarf2 true`” at the command line. It is currently enabled by default.)
- Does the debugger know where to find the SPU ELF file? (In the ProDG Debugger, go to “Tools -> Options -> Projects -> Search Directories” and add the path to your SPU ELF file in the “SPU ELF files” section.)
- Is the debugger set to display the source code? (In the ProDG Debugger, right-click in the SPU source window and check the settings in the “Disassembly” sub-window.)

6 Frequently Asked Questions

Can I use SPA to optimize assembler generated by the C compiler?

The short answer is “maybe, but you *really* should not”. SPA can mix virtual and physical registers, so technically, as long as the code constructs are supported by SPA (no `bi`, `brsl`, and so forth.), it may work. But the result will be very bad since the already-allocated physical registers will not be changed, and therefore, you’ll end up having lots of unnecessary dependencies.

It is definitely not recommended unless you really know what you are doing.

What’s the difference between “.s” and “.o” files generated by SPA?

SPA is able to directly generate binary code using its own internal assembler. We also have a “.s” output mode with additional debugging information:

- Pairing/execution units
- Pipeliner feedback
- Additional commenting to show pipelined loop iterations

The “.s” file *should* be assemblable by gcc and generate the same object file (apart from DWARF debugging information).

Why can’t I link my SPA function?

Assuming you are correctly linking all relevant object files, it is probably that you have forgotten `extern "C"` in your declaration.

SPA does not generate C++ decorated names. Overloading functions is not possible.

What should I do if I run out of registers?

If you have declared your function using the `.func` directive, SPA does not spill. If it runs out of registers, it fails. A typical message would be:

```
Error: global register allocation failed: Insufficient registers (16 more
required)
```

If you are using the default definitions for the C ABI, you can get around 48 more registers very easily by declaring your function using the `.cfunc` directive. For details, see the [“Inputs, Outputs, and ABI Considerations”](#) section.

Why don’t my loop statistics match the loop kernel?

You are probably referring to extra instructions like:

- `ori $x, $y, 0`
- `rotqbii $x, $y, 0`

These instructions are “moves” added by the scheduler in empty slots of the schedule, to split register lifetimes that last longer than one iteration and avoid modulo variable expansion.

Note that the scheduler will never insert an extra cycle for these moves. It goes the other way. If scheduler fails for “i” cycles, then the “i+1” attempts will naturally have two more available slots for moves.

How can I eliminate my recurrence constraints?

There may be perfectly legitimate situations where this is normal (IIR filters, for example). But often, it means that you have reused the same pointer both as input and output of a very long calculation. What you need to do in these situations is called “induction variable reversal”, which basically means decoupling the pointer update from your processing code. It is mentioned in the [“Recurrence vs. Resource Constraints”](#) section of Chapter 2, [“Software Pipelining”](#).

If you are recurrence-constrained, this can usually be solved by doing loop induction reversal. This is not currently done automatically by the optimizer. Consider this example:

```
Loop:
    // Load
    lqd rIn, 0x00(pColours)
    lqd gIn, 0x10(pColours)
    lqd bIn, 0x20(pColours)

    // Stupid operations just to add a few latencies
    fma rOut, rIn, rKA, rKB
    fma gOut, gIn, gKA, gKB
    fma bOut, bIn, bKA, bKB

    // Stores (we're at least 12 cycles after the first
    // load and the pointer should still be alive)
    stqd rOut, 0x00(pColours)
    stqd gOut, 0x10(pColours)
    stqd bOut, 0x20(pColours)

    // Now update pointer, we're at least
    // 6 cycles (LQD) + 6 cycles (FMA) = 12 cycles after the first load..
    ai pColours, pColours, 0x30

    // Loop management
    ai count, count, -1
    brnz count, Loop :
```

It is heavily recurrence-limited; that is not what you want!

```
loop stats:
    resmii : 7          (resource constrained)
    recmii : 15 (*)     (recurrence constrained)
```

After induction reversal, it is not recurrence-constrained anymore:

```
Loop:
    // *** Pointer update decoupled ***
    ai pColours, pColours, 0x30

    // Reads (notice the offset has been changed)
    lqd rIn, -0x30(pColours)
    lqd gIn, -0x20(pColours)
    lqd bIn, -0x10(pColours)

    fma rOut, rIn, rKA, rKB
    fma gOut, gIn, gKA, gKB
    fma bOut, bIn, bKA, bKB

    // Writes (notice the offset has been changed)
    stqd rOut, -0x30(pColours)
    stqd gOut, -0x20(pColours)
```

```
    stqd bOut, -0x10(pColours)

    ai count, count, -1

    brnz    count, Loop
```

The recurrence now only contains the pointer update (2 cycles AI – much simpler!)

```
loop stats:
    resmii : 7 (*)          (resource constrained)
    recmii : 2              (recurrence constrained)
```

Notice that the scheduler will automatically split/rename registers that live too long, if they are not on the recurrence path. So, in the second situation, even if `pColours` seems to need to be alive for at least 15 cycles, it is not a problem, because it will actually be one or more copies of `pColours`, not the real register.

7 Tips and Tricks

Maintaining Balance

If your loop is resource-constrained and the odd and even pipeline usage is imbalanced, you can often rebalance the pipelines by replacing even instructions with functionally equivalent odd instructions or vice versa. This has the advantage of lowering your minimum initiation interval, which means your loop will be faster.

An example of this is the `shufb` (odd pipe) and `selb` (even pipe) instructions. For certain operations (but not all) you can obtain equivalent functionality from either instruction.

Reading from Unaligned Memory

If you have a pointer to an array of floats which is byte-aligned, but not necessarily quadword-aligned, the following code shows how to load them into registers as if they were quadword-aligned. (Note that the example code is just for illustration and could be optimized further, such as we could have two loads per loop iteration and remove the “`mov`”.)

```
// Assume our floats are pointed to by "pInput", and we have a loop
// counter called "count"

// Calculate input shuffle masks
.reg IN_SHUFFLE, DDDDDDDDDDDDDDDDD, tmp1
// Selects all words from first quadword
il128 IN_SHUFFLE, "ABCD"
// Replicates the fourth byte from the first quadword
il128 DDDDDDDDDDDDDDDDD, "DDDDDDDDDDDDDDDD"

// Find input pointer offset relative to quadword boundary
andi tmp1, pInput, 0xF
shufb tmp1, tmp1, tmp1, DDDDDDDDDDDDDDDDD // Replicate offset byte
// Adjust bytes selected by shuffle mask.
// for example, if tmp1==4 then our mask becomes "BCDa"
a IN_SHUFFLE, IN_SHUFFLE, tmp1

.reg INA, INB, IN

// Remember, loads will always be qw aligned, even if pInput is not
lqd INB, 0(pInput)

loop:
// Load adjacent quadwords into INA and INB
mov INA, INB
ai pInput, pInput, 0x10
lqd INB, 0(pInput)

// Select an appropriately aligned quadword from INA and INB
shufb IN, INA, INB, IN_SHUFFLE
```

```
// We now have our four floats in IN
... Do something with IN ....

ai count, count, -1
brnz count, loop
```

Easy Loop Unrolling

To prepare for loop unrolling, which can be used to hide instruction latencies that software pipelining cannot take care of, it is recommended that you write the loop body as a macro. This means that you only have to maintain a single copy of the loop body after unrolling.