

SPUパイプライン化アセンブラ (SPA)

ユーザガイド

目次

このドキュメントについて	4
目的	4
対象読者と前提条件	4
関連ドキュメント	4
SPAツールのバージョン履歴	5
ドキュメントのバージョン履歴	6
表記形式	7
フィードバック	7
1 SPUパイプライン化アセンブラ（SPA）の概要	8
アプリケーションの概要	8
現時点での制限	8
マニュアルで最適化されたコードとの比較	8
実行可能ファイル	9
2 ソフトウェアパイプライン化	10
ソフトウェアパイプライン化とは	10
ソフトウェアパイプライン化を行う理由	10
パイプライン化可能なループ	13
最小トリップ数	13
スケジューラ	14
アセンブリオプティマイザの出力の理解	15
3 SPAの使用	18
SPAの起動	18
アセンブリ命令	19
チャンネルニーモニック	19
制御フロー	19
コメント	20
仮想レジスタ	20
アセンブリ関数	21
入力、出力、およびABIに関する注意点	22
擬似命令	25
シャッフルマスク	27
データの宣言	27
エイリアシング	28
外部参照	29
ソフトウェアパイプライン化アルゴリズムの有効化/無効化	29
リンク	30
最適化レベル	30
4 プロプロセス	31

プリプロセッサシンボルの定義	31
条件コンパイル	31
プリプロセス後の出力の保存	32
ファイルのインクルード	32
文字列の連結	33
マクロ	33
ファイルの依存関係	34
5 デバッグ	35
トラブルシューティング	36
6 FAQ（よくある質問）	37
Cコンパイラによって生成されたアセンブラをSPAで最適化できますか？	37
SPAによって生成される「.s」ファイルと「.o」ファイルの違いは何ですか？	37
どうしてSPA関数がリンクできないのでしょうか？	37
レジスタが不足したらどうしたらよいですか？	37
ループ統計がループカーネルに一致しないのはなぜですか？	37
どのようにすれば反復制約を取り除くことができますか？	38
7 ヒントとテクニック	40
バランスの維持	40
アラインされていないメモリからの読み取り	40
簡単なループ展開	41

このドキュメントについて

目的

このドキュメントは、アセンブリ最適化ツールである SPU パイプライン化アセンブラ（SPA）のユーザガイドです。

対象読者と前提条件

このドキュメントは、PlayStation®3 アプリケーション用の効率的で管理しやすい SPU アセンブリコードを記述しようとしている PlayStation®3 ディベロッパーのために書かれたものです。

このドキュメントでは、ディベロッパーが次の内容に習熟していることを前提にしています。

- アセンブリ言語
- SPUアセンブリ言語仕様（以下の「[関連ドキュメント](#)」を参照）
- SPU の基本動作
- C と C++

関連ドキュメント

SPU アセンブリ言語と ABI

以下のドキュメントは、PlayStation®3 Developer Networkウェブサイト（<https://ps3.scedev.net>）の PlayStation®3 SDK Documentationパッケージから取得できます。

- SPU アセンブリ言語の概要については、「Cell OS Lv-2 SPU アセンブリ言語仕様書」を参照してください。
- SPU ABI インタフェースについては、「Cell OS Lv-2 Cell Broadband Engine™および SPU ABI 仕様書」を参照してください。

Edge ライブラリ

PlayStation®Edge Animation、DXT、Post の中核となる SPU プロセスは、高いスループットを実現できるように SPA を使って記述されています。付属のソースコードは、SPA のコーディングサンプルとして参考にしてください。

以下のドキュメントは、Edge ライブラリの使用法やリファレンスについての情報を提供しています。

- 「PlayStation®Edge ライブラリ 概要」
- 「PlayStation®Edge ジオメトリライブラリ クイックスタート」
- 「PlayStation®Edge ジオメトリライブラリ リファレンス」
- 「PlayStation®Edge オフラインツール用ジオメトリライブラリ リファレンス」
- 「PlayStation®Edge アニメーションライブラリ リファレンス」
- 「PlayStation®Edge オフラインツール用アニメーションライブラリ リファレンス」
- 「PlayStation®Edge Zlib ライブラリ リファレンス」
- 「PlayStation®Edge DXT ライブラリ リファレンス」

- 「PlayStation®Edge LZMA ライブラリ リファレンス」
- 「PlayStation®Edge LZ0 ライブラリ リファレンス」
- 「PlayStation®Edge Post ライブラリ リファレンス」

SPAツールのバージョン履歴

以下に、SPA 実行可能ファイルのバージョン履歴をまとめておきます。

バージョン番号/日付	変更内容
v 1.4.6 build 1 2010 年 03 月 01 日	「readoptional」レジスタ注釈を追加
v 1.4.5 build 4 2010 年 01 月 28 日	著作権表示の更新
v 1.4.5 build 3 2010 年 01 月 19 日	.setoptim という新しいディレクティブを使うことにより、ファイル内での最適化レベル設定が可能になりました
v 1.4.5 build 2 2009 年 12 月 14 日	デッドコード除去の改善
v 1.4.5 build 1 2009 年 11 月 20 日	未初期化のレジスタからの読み込みや、冗長なレジスタへの書き込みに対する SPA による警告を追加
v 1.4.4 build 1 2009 年 11 月 16 日	コード生成の改善
v 1.4.3 build 1 2009 年 11 月 09 日	関数セクションのサポートを追加
v 1.4.2 build 3 2009 年 09 月 18 日	32 ビット整数イミディエイトをロードするための il32 疑似命令を追加。 明示的な Even パイプ移動疑似命令 (mve) を追加 明示的な Odd パイプ移動疑似命令 (mvo) を追加
v 1.4.2 build 2 2009 年 09 月 08 日	ILA でラベルからのイミディエイトオフセットが可能になりました
v 1.4.2 build 1 2009 年 09 月 03 日	--align パラメータがテキスト出力 (-s) に反映されないバグを修正
v 1.4.1 build 7 2009 年 5 月 8 日	無効なコマンドラインパラメータのフィードバックを改善 マイナーなバグの修正
v 1.4.1 build 3 2009 年 4 月 30 日	互換性を高めるために SSE2 のサポートを無効化
v 1.4.1 build 1 2009 年 4 月 22 日	O1 でローカルスケジューリングが無効化されるようになりました（大規模/複雑なループでのレジスタのプレッシャーが高すぎる場合に便利）。 レジスタのスピル (spill) 時にスタックチェック用コードを挿入する新しいオプション
v 1.4.0 build 1 2009 年 4 月 2 日	データセクションのラベルがデフォルトでローカルになりました（これらのラベルにリンカーからアクセスできるようにするには、新しい.global ディレクティブを使用する）。 警告をエラーとして扱えるようになりました。 lqd のオフセットにラベルを指定できるようになりました。 パイプライン化可能なループが見つからない場合に警告が表示されるようになりました。 関数やデータが一切含まれていないファイルを処理しようとすると、エラーが発生するようになりました。
v 1.3.4 build 1 2009 年 1 月 28 日	レジスタ割り当ての失敗後にスケジューリングが再開された場合の速度改善
v 1.3.3 build 1 2009 年 1 月 26 日	バックエンドのさまざまな速度改善 SMS スケジューラがデフォルトで無効化されるようになりました。コマンド

バージョン番号/日付	変更内容
	ラインから再度有効にできます。
v 1.3.2 build 1 2008 年 11 月 20 日	マクロ名をマクロの引数として渡せるようになりました。 新しい.localreg プリプロセッサディレクティブを使えば、マクロローカルな仮想レジスタを宣言できます。
v 1.3.1 build 5 2008 年 8 月 8 日	.short データ宣言ディレクティブを追加
v 1.3.1 build 4 2008 年 7 月 9 日	ilf32 擬似命令で 1 個または 4 個の即値を指定できるようになりました。
v 1.3.1 build 3 2008 年 6 月 25 日	強力な反復制約を含む大規模ループでの速度改善
v 1.3.1 build 2 2008 年 6 月 18 日	文字列連結演算子とマクロ UID 演算子が、マクロ呼び出し時のパラメータリストでも使用できるようになりました。
v 1.3.1 build 1 2008 年 6 月 4 日	プリプロセッサがマクロをサポートするようになりました。
v 1.3.0 build 3 2008 年 5 月 8 日	プリプロセッサによる行番号のカウントミスの原因となっていたバグを修正
v 1.3.0 build 1 2008 年 5 月 6 日	.include ディレクティブを使ってファイルをインクルードできます。
v 1.2.0 build 2 2008 年 4 月 10 日	関数スコープ外の任意の場所でデータを宣言できるようになりました。 .float または.word で宣言された値を式で初期化できるようになりました。
v 1.2.0 build 1 2008 年 4 月 3 日	最初のリリース

ドキュメントのバージョン履歴

以下に、このドキュメントのバージョン履歴をまとめておきます。

バージョン番号/日付	変更内容
v 1.17 2010 年 06 月 28 日	ファイル依存関係に関するサブセクションを追加 SPA ツールのバージョン履歴を更新
v 1.16 2010 年 03 月 01 日	レジスタ注釈に関するサブセクションを追加 SPA ツールのバージョン履歴を更新
v 1.15 2010 年 01 月 28 日	最適化ディレクティブに関するセクションを追加 「ヒントとテクニック」のセクションに新しいヒントを追加 SPA ツールのバージョン履歴を更新
v 1.14 2009 年 09 月 08 日	SPA ツールのバージョン履歴を更新
v 1.13 2009 年 9 月 3 日	SPA ツールのバージョン履歴を更新。
v 1.12 2009 年 6 月 24 日	序章を追加し、第 1 章をリライト 誤記の修正と他の編集上の改善
v 1.11 2009 年 5 月 8 日	SPA ツールのバージョン履歴を更新
v 1.10 2009 年 4 月 2 日	データラベルをグローバルにする方法についての情報を追加 SPA ツールのバージョン履歴を更新
v 1.9 2009 年 1 月 28 日	SPA ツールのバージョン履歴を更新
v 1.8 2008 年 11 月 20 日	マクロ引数に関するドキュメントを拡張 新しい.localreg ディレクティブに関するドキュメントを追加

バージョン番号/日付	変更内容
	SPA ツールのバージョン履歴を更新
v 1.7 2008 年 8 月 8 日	新しい .short ディレクティブに関するドキュメントを追加 SPA ツールのバージョン履歴を更新
v 1.6 2008 年 7 月 9 日	「デバッグ」の章に「トラブルシューティング」のセクションを追加 ilf32 擬似命令に関するセクションを更新
v 1.5 2008 年 6 月 25 日	SPA ツールのバージョン履歴を更新
v 1.4 2008 年 6 月 18 日	SPA ツールのバージョン履歴を更新
v 1.3 2008 年 6 月 4 日	「マクロ」のセクションを追加
v 1.2 2008 年 5 月 6 日	「最小トリップ数」のセクションを追加 「ファイルのインクルード」のセクションを追加
v 1.1 2008 年 4 月 3 日	「変更一覧」の章を追加 「ヒント/テクニック」の章を追加
v 1.0 2008 年 4 月 3 日	このドキュメントの最初のリリース

表記形式

このドキュメントでは、以下のような表記形式を利用しています。

記法	意味
等幅フォント	プログラミングコード、処理命令、レジスタ名、データ型、イベント、ファイル名などのリテラルを表します。また、関数、構造体、マクロ名なども表します。
<u>青色+下線のテキスト</u>	ハイパーリンクを表します（青く表示されるのは、カラープリンタもしくはオンラインの場合だけです）。

フィードバック

SPAについて、ご意見をお寄せください。弊社は常にツールの改善に努めており、皆様からのフィードバックをお待ちしています。PlayStation®3 Developer Network (<https://ps3.scedev.net>) で、テクニカルサポートかパブリックフォーラムに以下のような情報のフィードバックを送信することができます。

- 生成されたコードのサイズ
- 生成されたコードの速度
- brsl/brasl 命令のサポート
- 新機能

1 SPUパイプライン化アセンブラ（SPA）の概要

アプリケーションの概要

アセンブリ最適化ツールである SPA を使用すると、SPU 低レベルコードを記述するプログラマは、次のことが行えます。

- 重要なループの最適化（メインコードは依然として C/C++ であると想定）
- 最適化されたアセンブリコードの記述と管理
- アセンブリオプティマイザのフィードバックに従った効率的なコードの最適化

SPA が提供する機能は次のとおりです。

- スケジューリング
 - (1) ソフトウェアパイプライン化（単純なカウンタを用いたループの場合だけ）
 - (2) 「古典的」なスケジューリング
- レジスタの割り当て
 - (3) グローバルレジスタ割り当て
 - (4) 移動合併/コピー伝播
- デバッグ
 - (5) DWARF2 デバッグのサポート（デバッガ内でのソース/逆アセンブリの混在表示が可能）

SPA は、C/C++ から呼び出し可能なリンク可能オブジェクトファイルを出力します。SPA は入力として、「Cell OS Lv-2 SPU アセンブリ言語仕様書」で説明されているアセンブリ構文をいくつか制限付きで受け入れるほか、擬似命令やプリプロセスのオプションもいくつか受け入れます。詳しくは、第 3 章「[SPA の使用](#)」を参照してください。

現時点での制限

- 単純なループ（デクリメントカウンタループ）で、「最高」とは言えませんが、「非常に良い」動作を示します。
- どのような形式の間接分岐もサポートしていません（リターンは除く）。
- 生成されるコードのサイズが、かなり大きくなります。

マニュアルで最適化されたコードとの比較

マニュアルで最適化されたループの最適化されていないバージョンに対し、SPA をテスト目的で実行しました。これらのループの大部分（95%）は単純なカウンタを用いたループであったため、最小限の変更を加えただけで、このツールで処理できました。

速度（SPA での最適化と手作業での最適化との対比）：

- 平均 = SPA の方が 0～5% 低速でした。
- 50 サイクル/イテレーションを超えるループの場合（もっとも重要な場合） = SPA の方が 0～1% 低速でした。
- 最悪の場合 = 9 サイクルのループが 10 サイクルでしかスケジューリングできませんでした。

サイズ：

- SPAの方が平均で、手作業で最適化されたリファレンスコードよりも30～50%サイズが大きくなりました。
- コードサイズを減らすための作業をいくつか実施しました（エピローグの除去など）。

ループの約5%は、良い結果が得られなかったか、あるいはSPAで処理できませんでした。以下に詳細を示します。

- 「Do」ループは、SPAで処理することはできますが、ソフトウェアパイプライン化の対象にはなりません（投機実行の問題のため）。したがって、それらは、通常のコンパイラを使用した場合と同じく、「古典的な」方法でしかスケジューリングできません。
- いくつかの他のループは、コードの速度とサイズのバランスを取るように調整された構成要素（そのほとんどはBIベースのもの）を使用しているため、SPAで処理できません。

実行可能ファイル

SPAを使用するのに必要なファイルは、次のとおりです。

ファイル名	解説
spa.exe	Win32 実行可能ファイル

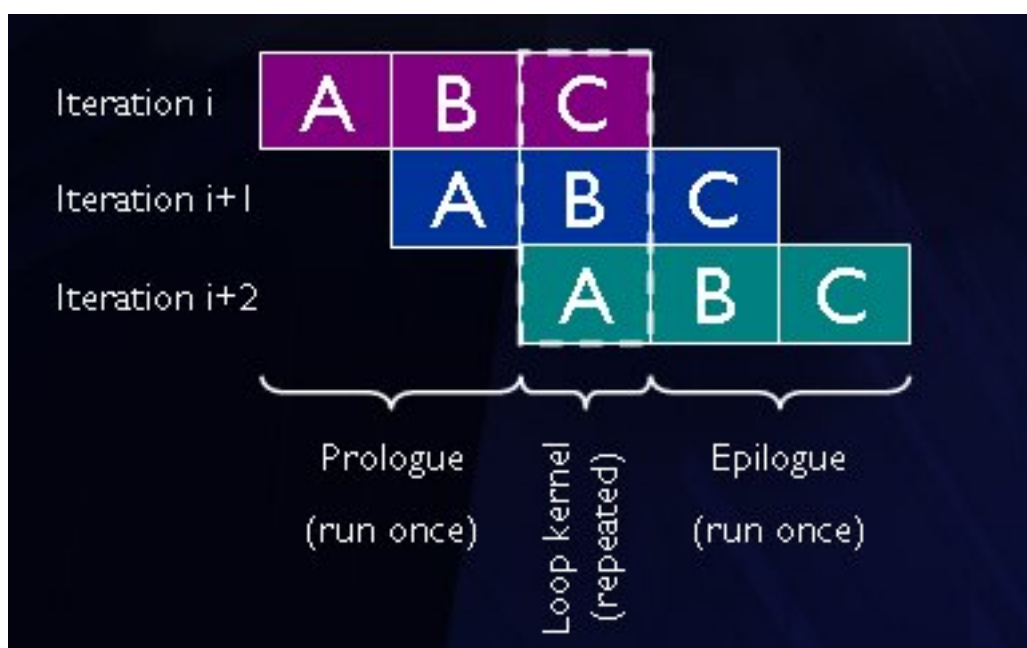
2 ソフトウェアパイプライン化

ソフトウェアパイプライン化とは

これは、「できるだけ最適化された」ループカーネルを作成できるように、ループの現在のイテレーションの結果がまだ利用できない場合に他のイテレーションに含まれる命令を使用する、というアイデアです。これが必要になるのは SPU 命令の待ち時間が長いからなのですが、これはまた、手作業で最適化された良質の SPU アセンブリを記述する人々が使用するテクニックでもあります。

実際のコンパイラでもっともよく使用されているソフトウェアパイプライン化アルゴリズムである「モジュロスケジューリング」を使用します。

図1 モジュロスケジューリングとソフトウェアパイプライン化



ソフトウェアパイプライン化を行う理由

以下のような非常に単純なループ(スケジューリングされていないコード – 意味のある処理は行っていない)を考えます。

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

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

    fa      a0, b0, c0       // even 6 サイクル
    fa      a1, b1, c1       // even 6 サイクル
```

```

fa      a2, b2, c2      // even 6 サイクル
fa      a3, b3, c3      // even 6 サイクル

stq     a0, -0x40(pA)    // odd 6 サイクル
stq     a1, -0x30(pA)    // odd 6 サイクル
stq     a2, -0x20(pA)    // odd 6 サイクル
stq     a3, -0x10(pA)    // odd 6 サイクル

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

```

このループには、6 個の Even 命令と 9 個の Odd 命令があります。これは、理想的には 9 サイクル/イテレーションでスケジューリングを行いたいことを意味します。

残念ながら、待ち時間が存在するため、これは不可能であり、最初のロードから 12 サイクル経過した後で stq をスケジューリングすることしかできません。古典的（リニア）なスケジュールで見つけることのできる最良のものは、実際には 19 サイクル/イテレーションです。SPA はさまざまな工夫を凝らして、このループのソフトウェアパイプライン化されたバージョン（9 サイクル/イテレーション）を生成します。

図 2 「myloop」サンプルのソフトウェアパイプライン化されたバージョン - 9 サイクル/イテレーション（維持）*

```

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]

```

* ループカーネルを白色で表示しています。

図3 「myloop」 サンプルのパイプライン化されていないバージョン - 19 サイクル/イテレーション*

```

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>]

```

* ループカーネルを白色で表示しています。

スケジューラが行った処理を理解するために、デバッグ出力に含まれるループカーネルを次に示します。

```

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

```

- 9 サイクル/イテレーションを実現できるように、3つのイテレーションが同時に実行されています。
- コメント内の数字は、その命令が実際に属しているイテレーションを表します。
- また、スケジューラが長すぎるライブ範囲のいくつかを自動的に分割するため、スケジュールのパフォーマンスに影響を与えないようにいくつかの移動命令（「ori x,y,0」）を挿入していることも読み取れます。

プロログやエピログ（必要な場合）など、必要な補償コード（ここでは示していません）はすべて自動生成されています。

パイプライン化可能なループ

以下に、理解すべき重要な点をいくつか列挙します。

- パイプライン化を行えるのは、実行されるイテレーション数が予測可能な do-while ループだけです。
- 標準的な「while」ループはパイプライン化できません。
- ネストされたループでパイプライン化されるのは、もっとも内側のループだけです。
- ループカーネルでは複数のイテレーションが同時に実行されるため、ループを開始するための特別のコード（**プロローグ**）とループを終了するためのコード（**エピローグ**）が通常は必要になります。したがってコードのサイズが増えます。（ループの種類によっては、SPAがプロローグやエピローグを「除去する」ことでコードサイズの増大を回避できる場合があります。後述する「[プロローグとエピローグの除去](#)」を参照してください）。
- この構造では、ある特定の最小イテレーション数が必要になります。このイテレーション数よりも少ない回数しかループが実行されない可能性がある場合には、ソフトウェアパイプライン化されていないループのコピーを追加で用意する必要があります。これも、コードサイズ増大の要因の 1 つになります。

通常、現在の実装でアセンブリオプティマイザがパイプライン化を試みる唯一のループタイプは、次のようなデクリメントループです。

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

重要

- アセンブリオプティマイザはループの変換を一切行いません。ソフトウェアパイプライン化が可能なようにループを変換するのは、ユーザの責任です。
- ソフトウェアパイプライン化のフィードバックがアセンブリオプティマイザから一切出力されない場合、そのループがパイプライン化不可能とみなされたことを意味しています。

最小トリップ数

すでに述べたように、少ないイテレーション数にも対応できるように、パイプライン化されていないループのコピーを生成しなければならない場合があります。たとえば、ループの 3 つのイテレーションが同時に実行される場合、おそらく、イテレーション数が 1 回または 2 回の実行を処理できるように、SPA はパイプライン化されていない通常のループを生成する必要があります。それが、次のフィードバックで SPA が表示している内容です。

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

ここで、4 回よりも少ないイテレーション数でこのループが実行されることは決してないと指定すれば、このパイプライン化されていないコピーが作成されなくなり、コードのサイズを大幅に減らすことができます。次のように \$mintrip を使って、分岐命令に注釈を加えます。

```
// ループが少なくとも 3 回は実行されることを保証します
brnz count, cffLoop : $mintrip<3>
```

すると、次のようなフィードバックが表示されます。

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

スケジューラ

繰り返し検索

モジュロスケジューラは、ある特定の開始間隔「ii」（1 イテレーション当たりのサイクル数）に対するスケジュールの検索を試みながら動作します。

このプロセスの概要は、次のとおりです。

- min_ii（詳細は後述）を決定します
- max_ii（パイプライン化を行わない古典的なスケジューラの場合）を決定します
- (ii < max_ii) の間、次の処理を繰り返します。
 - (6) この ii に対する有効なスケジュールの検索を試みます
 - (7) 失敗した場合は ii をインクリメントします（指数的な検索）

ループのスケジューリングが困難であるほど、スケジューラの試行回数も多くなります。

モジュロスケジューラ

スケジューリングは NP 困難な問題であるため、同じループに対して複数のモジュロスケジューリングアルゴリズムが実行され、最良の結果が選択されます。これらのアルゴリズムを以下に示します。

- Iterative Modulo Scheduling (IMS) の一種。これは低速（バックトラックが激しい）ですが、現時点で最良の結果を出力するスケジューラです。
- Swing Modulo Scheduler (SMS) の改善版。2 回実行されます（トップダウン優先度とボトムアップ優先度で 1 回ずつ）。これは非常に低品質のスケジューラですが、時々（テストでは約 5% の確率で）、良い結果を出力します。

長すぎる存続期間

ループカーネルのスケジューリングを行う際にはすべての反依存が無視されるため、最終的に 1 つのイテレーションよりも長く存続するレジスタが発生してしまふことがあります。これは、アセンブリ最適マイザが「長すぎる存続時間」と呼ぶものです。SPA は、この問題を解決するため、新しいレジスタに「移動」する命令をスケジュールの空スロットに挿入し、変数の存続期間を「カット」しようとしています。これらの移動命令を空スロット内にスケジューリングできなかった場合、この開始間隔に対するスケジューリングが中断されます。

アセンブリオプティマイザの出力の理解

SPA の出力

以下に、ループに対する典型的な出力を示します。ここでは、`--verboseswp` フラグを使って冗長ソフトウェアパイプライン化フィードバックをオンにします。

最初の部分には、次のように、パイプライン化するループに関する一般的な統計情報がいくつか表示されます。

```
~$ 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)
(...)
```

その後の部分には、スケジューラの実際の結果が表示されます。

```
(...)
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)
(...)
```

反復制約とリソース制約

まず、アセンブリオプティマイザは**最小開始間隔**の決定を試みます。これは、ループの連続する2つのイテレーションの開始位置の間の最小サイクル数です。

- ユーザがコードの最適化やバランス調整を行ううえで非常に重要な情報です。
- さらに、(高速化のために) 実現不可能なスケジュールの試行を避けるために使用される下限値でもあります

制約には次の2種類があります。

- リソース：1 イテレーション当たりの命令数に関係します。
- 反復：次のイテレーションで再利用される値の依存サイクルに関係します。たとえば、「FMA」を使って結果を蓄積するようなループを6 サイクル/イテレーション未満で実行することは不可能です。なぜなら、6 サイクルがFMA のレイテンシだからです。

典型的な出力を以下に示します。

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

通常は、反復制約ではなくリソース制限が考慮されます。

リソースの使用率

アセンブリオプティマイザは各パイプラインの命令数を表示し、どちらのパイプラインが制限要因になっているかを示します。

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

重要なポイント

このループに含まれる命令は次のとおりです。

- 1 イテレーション当たり 17 個の even 命令
- 1 イテレーション当たり 27 個の odd 命令

したがって、27 サイクル/イテレーション未満で実行することは**決して**できません。これは odd パイプラインによって制約されています。

- このコードのバランスを調整すると、パフォーマンスを改善しやすくなる可能性があります
- すぐには気づかない最適化も役立つ可能性があります。たとえば、上の例で 3 個の odd 命令を 7 個の even 命令で置き換えることができれば、even 命令と odd 命令がともに 24 個となり、最小開始間隔が 27 から 24 に減ります！

リニアスケジューラ

パイプライン化を行わない古典的なスケジューラも用意されています。これはコードの残りの部分で使用されますが、パイプライン化対象のループに対しても、次の目的のために実行されます。

- パイプライン化するだけの価値があるかどうかを判断するほか、試行を停止する上限を決定する
- このループをパイプライン化するメリットを判断するための興味深い統計情報を得る（たとえば、コードサイズに関して何らかのトレードオフを行う必要がある場合に役立つ可能性がある）

この例では、SPA が見つけた（パイプライン化されていない）最良のリニアスケジューラは、38 サイクル/イテレーションで実行されます。これは、このループのパイプライン化によって実現可能な理論上の最高パフォーマンスよりも、1.4 倍低速です。

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

この例の場合、アセンブリオプティマイザは、37 サイクル/イテレーション未満でパイプライン化されたスケジューラを見つけられなかった場合にはパイプライン化をあきらめ、代わりに通常のリニアスケジューラを使用します。

スケジューラの出力

前述したように、スケジューラはさまざまな工夫をして、最小イテレーション間隔に対する有効なスケジューラを見つけようとします。


```
(...)
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)
(...)
```

この例では実際に、次のような処理が行われました。

- IMS が、27 サイクル/イテレーションに対するスケジュール（3つのイテレーションを同時実行）を見つけました
- ボトムアップ SMS 実装は有効なスケジュールを見つけられませんでした
- トップダウン SMS 実装が、27 サイクル/イテレーションに対するスケジュール（3つのイテレーションを同時実行）を見つけました（IMS 実装と同じ）

プロローグとエピローグの除去

アセンブリオプティマイザはコードサイズを減らす試みの1つとして、ループエピローグの除去を自動的に試みます（これは、当初の予想よりも多くループイテレーションを実行することを意味します）。

- 余分な命令の投機実行が安全に行えない場合（ストア命令など）、エピローグのパイプライン化は行えません
- アセンブリオプティマイザがループカウンタを調整できない場合（たとえば、シフトベースの変わったループの場合）、エピローグの除去は失敗します

```
(...)
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)
(...)
```

以下の点に注意してください。

- エピローグを除去すると、別の興味深いコードサイズ低減メリットが得られます。通常、パイプライン化されていないループのコピーが不要になります
- プロローグの除去は、現時点では我々のループのほとんどでまだ成功していません。実装が非常に単純で、SPU コードの大部分にはストアの投機実行が関わっているからです
- ほとんどの典型的なループではエピローグを除去できます（ただし、値が返される場合は除く）

3 SPAの使用

SPAの起動

SPA は、C/C++から呼び出し可能なリンク可能オブジェクトファイルを生成します。次に、典型的なコマンドラインを示します。

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

- `-o` は、出力（ELF 形式）リンク可能ファイル（.o）です。
- ソースファイル名がコマンドラインの最後の要素になっている（すべてのオプションの後に存在している）必要があります。

SPAでサポートされるすべてのオプションを表示するには、引数を何も指定せずに実行可能ファイルを呼び出します。表 1 に示すような情報が表示されます。

表 1 SPA のオプション

引数	解説
<filename>	入力ファイル名。
必須フラグ	
<code>-o <filename></code>	出力オブジェクトファイルを設定します。 <filename> 出力オブジェクトファイルの名前。
オプションフラグ	
<code>-O<unsigned int></code>	最適化レベルを設定します。 <unsigned int> 最適化レベル（デフォルト：2）
<code>--ims <bool></code>	Iterative Modulo Scheduling を有効にします。 <bool> IMS スケジューラを有効にするかどうか（最適化レベルが 2 の場合のデフォルト：true）
<code>--sms <bool></code>	Swing Modulo Scheduling を有効にします。 <bool> SMS スケジューラを有効にするかどうか（最適化レベルが 2 の場合のデフォルト：true）
<code>--asmout (-s) <filename></code>	出力アセンブリファイルを設定します（デバッグ情報が失われるので非推奨）。 <filename> 出力アセンブリファイルの名前
<code>--header <filename></code>	出力アセンブリファイルのヘッダを含むテキストファイルを指定します。 <filename> ヘッダテキストファイルの名前
<code>--singlecol <bool></code>	アセンブリファイルの命令を単一の列に出力します。 <bool> 単一の列を使用するかどうか（デフォルト：false）
<code>--swpindent <bool></code>	SWP ループのイテレーションをインデントを使って示します。 <bool> インデントを使用するかどうか（デフォルト：false）
<code>--preout (-p) <filename></code>	プリプロセス後のアセンブリをファイルに保存します。 <filename> プリプロセッサの出力を保存するファイルの名前。
<code>--align <uint></code>	オブジェクトファイル内の .text セクションのアラインメント。 <uint> アラインメント（バイト）（デフォルト：16）
<code>--dwarf2 <bool></code>	DWARF2 デバッグ情報を生成します。 <bool> DWARF を有効にするかどうか（デフォルト：true）
<code>--fetch <bool></code>	試験的なフェッチを有効にします/制御コードを発行します。

引数	解説
	<bool> 有効にするかどうか（デフォルト：false）
--define (-D) <string>	整数シンボルの事前定義を行います。 <string> symbolname または symbolname=value（デフォルト値：1）
--definesingle (-Df) <string>	単精度 float シンボルの事前定義を行います。 <string> symbolname または symbolname=value（デフォルト値：1.0）
--definedouble (-Dd) <string>	倍精度 float シンボルの事前定義を行います。 <string> symbolname または symbolname=value（デフォルト値：1.0）
--quiet	静的モードを有効にします。
--nopicwarn	PIC に関する警告を無効にします。
--noswp	ソフトウェアパイプライン化を無効にします。
--verboseswp	ソフトウェアパイプライン化の冗長なフィードバックを有効にします。

アセンブリ命令

SPAでは、「Cell OS Lv-2 SPUアセンブリ言語仕様書」（「[関連ドキュメント](#)」セクションに記載）で説明されているアセンブリ構文が使用できるほか、プログラマを支援するために追加されたいくつかの擬似命令（i1128、movなど）やプロプロセスのオプションも使用できます。ただし、いくつか制限もあります。

命令の引数に関する制限

標準 SPU アセンブリでは、命令の引数の即値を次の方法でエンコードできます。

- 即値としての定数または式
- PC 相対アドレス（現在のプログラムカウンタはドット記号「.」で表現される）
- シンボリックラベルのアドレス

SPA では、上記のエンコーディングに次の制限が課せられます。

- 分岐命令では、定数や式を即値として使用できません。分岐命令はシンボリックラベルを参照する必要があります。
- PC 相対アドレスはどの命令でも使用できません。

チャンネルニーモニック

SPA では、「Cell OS Lv-2 SPU アセンブリ言語仕様書」で説明されているすべてのチャンネルニーモニックがサポートされます。

制御フロー

現時点では、「単純な」制御フロー命令（同一アセンブリ関数内のシンボルに分岐するもの）しか処理されません。

表 2 サポートされている/されていない分岐命令

命令	サポートされているかどうか
BR	サポートされています
BRA	サポートされています
BRSL/BRASL	現時点ではサポートされていません
BISL/BISLED	サポートされていません
BI	サポートされていません（ただし、関数の末尾には SPA によって自動的に「BI \$0」（リターン）が追加される）
BIZ/BIHZ/BINZ/BIHNZ	サポートされていません
IRET	サポートされていません

コメント

C スタイルと C++スタイルの両方のコメントが使用できます。

仮想レジスタ

SPA では、バックエンドで割り当てられる仮想レジスタを使用できます。これらのレジスタは、使用する前に .reg ディレクティブを使って宣言する必要があります。仮想レジスタと物理レジスタは、次のように混在させることができます。

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

制限

仮想レジスタへの書き込み時に、対応するレジスタ名は自動的に変更されません。仮想レジスタは、使用されるたびに同じ物理レジスタにマッピングされます。

- バックエンドは、プログラマが自分が何をしているか理解しているものと仮定します。
- ユーザは、この点に注意する必要があります。別の場所で使用されたレジスタへの書き込みを行うと、依存関係/スケジューラ制約を導入してしまうからです。典型的な悪い例は、同じ tmp 仮想レジスタを複数の異なる状況で繰り返し使用することです。

スケジューラに良いコードを生成させるには、仮想レジスタを使用する必要があることに注意してください。すでにレジスタが割り当てられたコードを使用すると、**非常に問題のあるスケジューラ**が生成されます。物理レジスタによって課せられる不要な反依存 (anti-dependency) をスケジューラが考慮しなければならないからです。spu-gcc の出力を SPA に入力するのは非常に問題があります。非常に問題のあるスケジューラが生成されるからです。

レジスタのエイリアス

.reg ディレクティブでは、次のようにエイリアスを使って特定の物理レジスタを参照することもできます。

```
.reg myResult=$5    // これで、myResult はレジスタ$5 のエイリアスになります
ai    myResult, $0, 1    // $5=$0+1
```

先に述べたレジスタの手動割り当てに関する注意を忘れないでください。レジスタの割り当ては極力 SPA に任せるべきです。この構文を使ってレジスタの手動割り当てを行うと、不要な反依存を導入することになるので注意してください。

レジスタ宣言の注釈

仮想レジスタの宣言時には、「readoptional」注釈を追加することができます。この注釈があると、レジスタに書き込まれた値が読み込まれない場合でも、SPA は警告を発行「しません」。(マクロ中に標準定数が多数存在しているがその定数のすべてを使わない時に、未使用定数の除去を SPA のデッドコード除去に依存している場合、この注釈は便利です)。「readoptional」の構文は以下の通りです。

```
.reg myreg1: readoptional // myreg1 に書き込まれた値が読み込まれない場合に警告を発行
                             しません
.reg myreg2, myreg3: readoptional // myreg2 および myreg3 に対して警告を発行しませ
                             ん
```

アセンブリ関数

アセンブリ関数を宣言するには、.func と.cfunc のどちらかのディレクティブを使用します。どちらの場合も、リターン命令（つまり「bi \$0」）が関数の末尾に自動的に追加されます。この命令をコード内に含めないでください。

.cfunc ディレクティブ

C または C++ から SPA 関数を呼び出す場合には、.cfunc ディレクティブを使って関数宣言を行うことをお勧めします。

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

    // .cfunc を使って関数を宣言すると、SPA は自動的に、
    // 「pInput1」、「count」という名前の仮想レジスタを作成し、
    // それらをパラメータの値で初期化します。

    // 何らかの処理を行い、有用な値を workRegister に格納します。
    .reg workRegister
    ...

    // この関数の戻り値は void ではないため、SPA は、
    // 「@result」という名前の仮想レジスタを自動的に作成します。
    // このレジスタに戻り値を格納する必要があります
    mov @result, workRegister

.endfunc
```

.cfunc を使って関数を宣言すると、その関数では標準の SPU ABI（「Cell OS Lv-2 Cell Broadband Engine™ および SPU ABI 仕様書」を参照）が使用され、SPA は自動的に各パラメータを同じ名前の仮想レジスタ内に格納します。関数の戻り値の型が「void」でない場合、SPA は「@result」という名前の仮想レジスタの宣言も行います。このレジスタに関数の戻り値を格納する必要があります。

.cfunc 関数に関する制限

SPA 関数のパラメータとして渡せるのは、1つのレジスタに収まるデータ型だけです。SPA では、.cfunc 宣言で指定された型の解釈や検証が試みられることはなく、各パラメータはいずれかのレジスタ内に存在するものとみなされます。1つのレジスタに収まらない変数を SPA 関数に渡したい場合は、代わりにそのポインタを渡すようにすべきです。

.func ディレクティブ

別の方法として、.func ディレクティブを使用することもできます。この場合は、次のように単純に .func と関数名を宣言します。

```
.func MyFunction2

// .func ディレクティブを使って関数を宣言した場合、
// ABI の処理はすべてユーザの責任になります。

// アセンブリ関数のコードをここに記述します。

.endfunc
```

.func 関数に関する制限

.func を使って関数を宣言した場合、ABI の処理はすべてユーザの責任になります。つまり、パラメータの受け渡しや返り値/ライブ入出力レジスタの処理は、プログラマが明示的に行う必要があります。これは、ユーザが独自の ABI を定義したい場合に役立ちます。詳細については、次のセクションを参照してください。

入力、出力、およびABIに関する注意点

標準の SPU ABI に従う場合（つまり、C または C++ とのインタフェースを定義する場合）には、.cfunc ディレクティブを使って関数を宣言することをお勧めします。SPA がユーザに代わって、ABI の詳細をすべて処理してくれるからです。一方、.func ディレクティブを使って関数を宣言する場合には、以下の説明に従って ABI の処理を行う必要があります。

表 3 に、SPU ABI におけるレジスタの使用法の概要を示します。

表 3 SPU ABI におけるレジスタの使用法

命令	解説
\$0	リターンアドレス
\$1	初期スタックポインタ
\$3-\$79	パラメータ
\$3	返り値

次に、入力/出力を宣言する場合に使用できるディレクティブを示します。

- .input – 関数の入力
- .output – 関数の出力

入力/出力に指定できるのは物理レジスタだけです（仮想レジスタは指定不可）。引数は、次のいずれかになります。

- 単一のレジスタ。例： `.input $1`
- レジスタの範囲。例： `.input $3-$6`
- 同一行の複数の引数。例： `.input $1, $3-$6`

アセンブラからの呼び出し用として `.func` を使って関数を宣言した場合、`.input/.output` ディレクティブを使って独自の ABI を記述できます。それ以外の場合は `.cfunc` を使用できます。`.cfunc` を使って関数が宣言された場合には、次の処理が発生します。

- レジスタ `$1` と `$80~$127` が自動的に入力/出力として宣言されます。
- 各ライブパラメータレジスタが自動的に入力として宣言されます。
- SPA でのレジスタ割り当て処理中にレジスタが不足すると、そのレジスタ割り当て処理を完了させるのに必要なだけの不揮発性 ABI レジスタに対するセーブ/リストアコードが自動生成されます。これは繰り返し行われるため、レジスタ割り当て処理が数回行われないと適切なスケジュールが見つからない可能性があります。
- 関数が値を返す場合、SPA は自動的に「`@result`」という名前の仮想レジスタを宣言します。ユーザはこのレジスタに返り値を格納する必要があります。
- `.cfunc` で SPA が自動的に 80 番以上のレジスタのセーブ/リストアを行うのは、レジスタアロケータからの要求があった場合だけです。これらのレジスタのいずれかをコード内で明示的に使用したい場合は、そのセーブ/リストアを手動で行う必要があります。

以上の処理は、`.func` を使って宣言された関数では一切行われません。

`.func`、`.cfunc` のいずれが指定された場合でも、SPA は常に `$0` がリターンアドレスを含む暗黙的な入力であると仮定します。

ABI の例

次のような C 関数を考えます。

```
float TestFunction(qword a, qword b, int c, int d)
{
    // この時点で、
    // - a は$3 に格納されています。
    // - b は$4 に格納されています。
    // - c は$5 のデフォルトスロットに格納されています。
    // - d は$6 のデフォルトスロットに格納されています。

    // 関数の処理をここで行います。
    // ...

    return someFloat; // someFloat が$3 のデフォルトスロットに返り値として格納されます
}
```

これと同等の SPU アセンブリ関数を宣言するには、次のようにします（標準の SPU ABI を手動で指定）。

```
.func TestFunction

    // パラメータ
    .input $3, $4, $5, $6

    // 返り値
    .output $3
```

```
// SPU ABI :
// これらの物理レジスタは入力/出力として宣言されていますが、
// コード内では明示的に使用されていないため、レジスタアロケータによって仮想レジスタが
// これらのいずれかにマップされることは決してありません。
// ライブ範囲の衝突が発生するからです。

// スタックポインタを確保します
.input $1
.output $1

// 80 番以上のレジスタを確保します
// (これらは不揮発性として標準 SPU ABI で指定されている)
.input $80-$127
.output $80-$127

.reg someFloat
// 関数の処理をここで行います...
// ...

// 戻り値をレジスタ$3 に格納すべきです
mov $3, someFloat

.endfunc
```

.func を使って宣言された関数では、その出力と入力を指定する必要があります。そうしなかった場合には、**レジスタの割り当てが失敗します**。また、何らかの値を返す関数で出力レジスタを定義しなかった場合にも、「デッドコード除去」パスが過剰に積極的な動作を示します。

あるいは、この関数は標準 SPU ABI を使用しているので、次のように .cfunc を使って宣言することもできます。

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

// この関数は.cfunc で宣言されているため、標準の SPU ABI が使用されることを
// SPA は理解しています。したがって、次の処理が行われます。
// - レジスタ$1 と$80～$127 が自動的に
//   入力/出力として宣言されます。レジスタアロケータで
//   レジスタ不足が発生すると、$80～$127 の必要な数のレジスタに対し、
//   セーブ/リストアが自動的に行われます
// - パラメータレジスタ (この場合は$3、$4、$5、$6) は、
//   入力として自動的に宣言されます
// - 「a」、「b」、「c」、「d」という名前の仮想レジスタが
//   自動的に宣言され、$3、$4、$5、$6
//   の値で初期化されます
// - 関数の戻り値の型が void ではないため、@result
//   という名前の仮想レジスタが自動的に宣言されます。
//   このレジスタに戻り値を格納する必要があります

.reg someFloat
// 関数の処理をここで行います...
// ...

// レジスタ@result に戻り値を格納すべきです
mov @result, someFloat
```



```
.endfunc
```

ご覧のように、`.cfunc` を使って関数宣言を行えばプログラマの作業が大きく軽減され、レジスタアロケータで領域不足が発生した場合にも、不揮発性レジスタが自動的にセーブ/リストアされるというメリットが得られます。

擬似命令

mov

擬似命令 `mov` を使えば、あるレジスタの値を別のレジスタにコピーできます。次に例を示します。

```
mov $3, $4 // レジスタ$4 の内容をレジスタ$3 にコピーします
```

SPA は、`mov` を即値ゼロの `ori` に翻訳します。

gpo

擬似命令 `gpo` は現在の PIC オフセットを取得し、それをターゲットレジスタのデフォルトスロットに格納します。このため、位置に依存しないコードを記述することができます。次に例を示します。

```
// データセクションで2つのクワッドワードを宣言します
.data
MYDATA:
.quad 0xAAAAAAAA_BBBBBBBB_CCCCCCCC_DDDDDDDD
.quad 0xEEEEEEEE_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 // この時点でデフォルトスロットにPIC オフセットが含まれています
lqx val, addr, picoffset

// 関数は値を処理します...
(...)
.endfunc
```

mfpc

擬似命令 `mfpc` は、指定されたレジスタのデフォルトスロットに、次の命令のアドレスを格納します。次に例を示します。

```
mfpc $3 // $3 のデフォルトスロットに次の命令のアドレスを格納します
```

ilf32

擬似命令 `ilf32` を使えば、(単精度の) 浮動小数点の即値をレジスタにロードできます。次に例を示します。

```
ilf32    $5, 3.141592f // 3.141592f をレジスタ$5 にロードします
```

アセンブリオプティマイザは `ilf32` を単一の `ilhu`、`ilhu` と `iohl`、のいずれかに翻訳します。即値の IEEE754 表現の下位 2 バイトがゼロの場合は `ilhu` のみが生成され、それ以外の場合は `iohl` も生成されます。

たとえば、SPA が次のコードを処理するとします。

```
ilf32    $5, 1.0f // IEEE754 表現は 0x3F800000 です
```

これは次の命令に翻訳されます。

```
// 下位 2 バイトはゼロなので、ロードする必要があるのは上位 2 バイトだけです
// (下位のハーフワードは ilhu によって自動的にゼロクリアされる)
ilhu     $5, 0x3F80
```

次に、以下のような異なる定数のロードを試みます。

```
ilf32    $5, 0.63661977236758f // IEEE754 表現は 0x3F22F983 です
```

SPA はこの即値ロード命令を単一の命令に置き換えることができないため、次の 2 つの命令を自動生成します。

```
ilhu     $5, 0x3F22 // 上位のハーフワードをロードします
iohl     $5, 0xF983 // 下位のハーフワードをロードします
```

あるいは、4 つの浮動小数点即値を指定して `ilf32` を呼び出すこともできます。これらの値は、ターゲットレジスタの各ワード要素にロードされます。次に例を示します。

```
ilf32    $5, 1.0f, 2.0f, 3.0f, 4.0f // 各 float をレジスタの各ワードにロードします
```

この場合、指定された 4 つの値は、レジスタ \$5 の対応するワードスロット内にロードされます。このように `ilf32` を使用すると、メモリ内に格納される定数と、それをロードするための `lqr` 命令が生成されます。

il128

擬似命令 `il128` を使えば、クワッドワードの即値をレジスタにロードできます。これは、定数（シャッフルマスクなど）を外部で宣言する必要がないことを意味します。ユーザは `il128` 命令を使用できます。たとえば、次のように記述できます。

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

この定数は明らかに 32 ビット SPU 命令に収まりません。そこで、アセンブリオプティマイザは次のようにします。

- 定数をメモリ内に格納します（定数のコードの直後で）
- それを `LQR` 命令でロードします

2 番目の例から、数値を指定する際には、読みやすいように下線 (`_`) を区切り文字として使えることがわかります。

また、`il128` ではシャッフルマスクの作成/ロードも容易に行えます。詳しくは、次のセクションを参照してください。

シャッフルマスク

il128 擬似命令は、次のような通常のシャッフルマスク構文もサポートしています。

- ワードシャッフル用の 4 文字から成る文字列 (“xxxx”)
- ハーフワードシャッフル用の 8 文字から成る文字列 (“xxxxxxxx”)
- バイトシャッフル用の 16 文字から成る文字列 (“xxxxxxxxxxxxxxxxxxxx”)
- A... は最初のクワッドワードの要素です
- a... は 2 番目のクワッドワードの要素です

次に例を示します。

```
// ワードシャッフル。0x000102031415161708090A0B1C1D1E1F になります
il128 shuf2, "AbCd"
// ハーフワードシャッフル。0x000112130405161708091A1B0C0D1E1F になります
il128 shuf3, "AbCdEfGh"
// バイトシャッフル。0x001102130415061708091A0B1C0D1E0F になります
il128 shuf4, "AbCdEfGhIjKlMnOp"
```

SPU固有のシャッフル拡張をサポートするために、構文が拡張されています（表 4 を参照）。

表 4 シャッフルマスク拡張

文字	マスク値	バイト
0	0x80	0x00
1/X/x	0xC0	0xFF
8	0xE0	0x80

データの宣言

生成されるオブジェクトファイルに追加すべきデータを宣言することができます。これは、.data および .rodata ディレクティブを使って関数のスコープ外で行う必要があります。データセクションで宣言されたラベルは、デフォルトではローカルになります。これらのラベルにリンカーからアクセスできるようにするには、.global キーワードを使用します。

表 5 に示すように、いくつかのデータ型ディレクティブがサポートされています。

表 5 サポートされるデータ型ディレクティブ

ディレクティブ	解説
.quad <quadword list>	クワッドワードのコンマ区切りリストを宣言します。クワッドワードを指定する構文は、il128 の場合と同じです。
.byte <byte list>	バイトのコンマ区切りリストを宣言します。
.short <short list>	short (16 ビット) のコンマ区切りリストを宣言します。
.word <word list>	ワードのコンマ区切りリストを宣言します。
.float <float list>	単精度浮動小数点値のコンマ区切りリストを宣言します。
.align <power of two bytes>	現在のデータセクションのパディングを行い、指定されたバイト境界にアラインされるようにします。値は 2 の累乗バイトを表します（たとえば、.align 3 を指定した場合、8 バイトのアラインメントが実現される）。

次の例は、これらのディレクティブの使用方法を示したものです。

```
.rodata // 以下のデータは、読み取り専用データセクションに配置します
```

```

SHUFFLEMASK1:
.quad "AbCdEfGhIjKlMnOp"
MYDATA1:
.word 0xAABBCCDD, 0xEEFF0011
.global MYDATA1 // ラベル MYDATA1 をグローバルにします (リンカーからアクセスできるように)

.data // 以下のデータは、データセクションに配置します
MYDATA2:
.byte 0xAB, 0xCB
.align 3 // データセクションをパディングして 8 バイトアラインメントを実現します
MYDATA3:
.float 1.0f, 3.14159265f

```

エイリアシング

SPA のバックエンドは、エイリアシングは発生しないという前提で動作しており、エイリアシングが発生する可能性のあるすべてのメモリアクセスに注釈を加える作業は、プログラマに任されています。命令に注釈を加えるには、それらの命令を「メモリグループ」に割り当てます。同じグループ内のすべての命令でエイリアシングが発生するものとみなされます。以下のコードはその構文を示したものです。

```

// エイリアシングを指定するための「メモリグループ」を宣言します
.memgroup  mg_row0, mg_row1, mg_row2, mg_row3
loop:

// 行列 1 をロードします
.reg  mat1a_row0, mat1a_row1, mat1a_row2, mat1a_row3
// 行 0 の STQD とのエイリアシング
lqx   [mg_row0] mat1a_row0, pWorldBase_row0, pWorldOffset
// 行 1 の STQD とのエイリアシング
lqx   [mg_row1] mat1a_row1, pWorldBase_row1, pWorldOffset
// 行 2 の STQD とのエイリアシング
lqx   [mg_row2] mat1a_row2, pWorldBase_row2, pWorldOffset
// 行 3 の STQD とのエイリアシング
lqx   [mg_row3] mat1a_row3, pWorldBase_row3, pWorldOffset

// 関数の処理をここで行います...
(...)

// 行 0 の LQX とのエイリアシング
stqd  [mg_row0] matR_row0, -0x40(pOutputWorld)
// 行 1 の LQX とのエイリアシング
stqd  [mg_row1] matR_row1, -0x30(pOutputWorld)
// 行 2 の LQX とのエイリアシング
stqd  [mg_row2] matR_row2, -0x20(pOutputWorld)
// 行 3 の LQX とのエイリアシング
stqd  [mg_row3] matR_row3, -0x10(pOutputWorld)

// ループ
brnz  count, loop

```

ここで、pWorldBase_row0+pWorldOffset からのロードと -0x40(pOutputWorld) への書き込みでは、同じメモリ場所へのアクセスが発生する可能性があります。他の行でも同様の依存関係が存在しています。これらの依存関係を明示するには、同じメモリ領域にアクセスする可能性のあるすべての命令を同じメモ

リグループに含めます。そうすれば、バックエンドは、最適化を行う際にそれらの依存関係を考慮する必要がありますことを知り、データハザードの発生を回避できます。

制限

ストア命令は、最大でも1つのメモリグループにしか割り当てることができません。ロード命令は、最大でも2つのメモリグループにしか割り当てることができません（[memgroup1, memgroup2]の形式で割り当てます）。

外部参照

別のモジュール（C やアセンブラなど）で定義されたグローバルデータシンボルにアクセスすることが可能です。その解決はリンク時に行われます。.spa ファイル内で次のように記述します。

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

同様に、外部で宣言されたテーブルにアクセスすることもできます（ILA を使ってラベルを参照すると、コードが位置非依存でなくなることに注意）。

```
.func SomeFunction
    (...)
    .extern MyTable
    (...)
    .reg tableBase
    (...)
    ila    tableBase, MyTable // 位置非依存でない
    (...)
    lqx val, someOffset, tableBase
    (...)
.endfunc
```

ソフトウェアパイプライン化アルゴリズムの有効化/無効化

SPA で現在サポートされているソフトウェアパイプライン化アルゴリズムには、Swing Modulo Scheduling と Iterative Modulo Scheduling の2つがあります。これらの有効/無効を切り替えるには、通常はコマンドライン引数を使用します。ただし、個々のファイルやファイル内の関数ごとに異なる設定を使用したい、という状況も考えられます。これを実現するには、次のように.setswp ディレクティブを使用します。

```
.setswp sms 1    // Swing Modulo Scheduling をオンにします
.setswp ims 0    // Iterative Modulo Scheduling をオフにします
```

関数スコープ内で.setswp を指定した場合、その設定は、その関数にしか適用されません（1つの関数内で同じ設定を複数回変更した場合には警告が発行され、最初の設定のみが適用される）。ファイルスコープ

内で`.setswp`を指定した場合、その設定は、そのファイル内のそこから先で見つかったすべての関数（別の`.setswp`ディレクティブによって設定が再度変更された場合はそこまでの関数）に適用されます。

リンク

SPA 関数が次の形式で宣言されているとします。

```
.func MyFunction

    // ...

    .reg myParam0, myParam1
    mov myParam0, $3 // パラメータ 0 を myParam0 に移動
    mov myParam1, $4 // パラメータ 1 を myParam1 に移動

    // ...
.endfunc
```

標準 ABI が使用されると仮定した場合、対応する C++ プロトタイプは次のようになります。

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

（ここでは標準ABIを使用しているため、同じ関数をSPAの`.cfunc`ディレクティブを使って宣言することも可能です。詳しくは「[アセンブリ命令](#)」と「[入力、出力、およびABIに関する注意点](#)」を参照してください）。

最適化レベル

最適化レベルをファイルに設定する通常の方法は、コマンドラインを使用する方法ですが、`.setoptim`ディレクティブを使うと、ファイルの中で設定することもできます。`.setoptim`によって設定された最適化レベルは、コマンドラインで設定されたいかなる値よりも優先されます。ファイルスコープの`.setoptim`ディレクティブは、ファイル中のその位置以後のあらゆる関数に影響します。関数スコープの`.setoptim`ディレクティブは、その関数だけに影響します。

構文は次のとおりです。

```
.setoptim 0 // 最適化を無効にします
```

注：最適化レベルを設定すると、複数の設定（有効なスケジューリングアルゴリズムなど）が変更されます。したがって、`.setoptim`は、`.setswp`のようなディレクティブによって既に行われた設定を変更する可能性があることに注意してください。

4 プリプロセッサ

SPA に含まれるアセンブリプリプロセッサは、条件に応じてコードを削除/インクルードしたり、コメントを除去したり、基本的なシンボル置換を実行したりします。

プリプロセッサシンボルの定義

プリプロセッサのシンボルを定義するには、`.set` キーワードを使用します。たとえば、次のようなコードがあるとします。

```
// プリプロセッサシンボルを作成します
.set MEMORY_OFFSET, 0x40

// プリプロセッサはこのシンボルをその整数値で置き換えます
lqd rt, MEMORY_OFFSET(ra)
```

プリプロセッサは上記を次のコードで置き換えます。

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

コメントやプリプロセッサディレクティブは削除されますが行番号は維持されることに注意してください。プリプロセッサのシンボルは次のように、それまでに定義されたシンボルと算術演算子を使って定義できます。

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

シンボルの再定義は次のように、必要なだけ何度でも行えます。

```
.set MEMORY_OFFSET, 0x40 // シンボルを定義します

lqd rt, MEMORY_OFFSET(ra) // ここでは 0x40 に置き換えられます

.set MEMORY_OFFSET, 0x20 // 以前に定義したシンボルを再定義します -正しい使い方です

lqd rt, MEMORY_OFFSET(ra) // ここでは 0x20 に置き換えられます
```

制限

混在モードの算術は行えません。

条件コンパイル

表 6 に示す条件ディレクティブがサポートされています。

表 6 サポートされているプリプロセッサディレクティブ

プリプロセッサディレクティブ
<code>.set <symbol>, <expression></code>
<code>.if <expression></code>
<code>.ifdef <symbol></code>
<code>.else</code>
<code>.elif <expression></code>
<code>.endif</code>

式は、以前に定義されたプリプロセッサシンボル、整数、および算術/論理/関係演算子で構成されます。ディレクティブは任意の深さまでネスト可能であり、標準的な方法で動作します。以下に例を示します。

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

// .ifdef (定義済みのシンボルが指定されている)
#ifdef SYMBOL0
// 次のディレクティブはアセンブリオプティマイザに渡されます
.reg regA
#endif
    ai regA, regA, 0x4

// .ifdef (未定義のシンボルが指定されている)
#ifdef UNDEFINED_SYMBOL
// 次の命令はアセンブリオプティマイザに渡されません
    ai regA, regA, 0x4
#endif

// .if (評価結果が true になる式が指定されている)
.if 3*(5-14)
// 次のディレクティブはアセンブリオプティマイザに渡されます
.reg regD
#endif
    ai regD, regD, 0x4

// .if (評価結果が true になる式が指定されている)
.if SYMBOL1+3
// 次のディレクティブはアセンブリオプティマイザに渡されます
.reg regE
#endif
    ai regE, regE, 0x4
```

プロプロセス後の出力の保存

ソースファイルのプロプロセス後のバージョンを表示できると、デバッグに役立つ場合があります。プロプロセス後の出力を保存するファイルを指定するには、次のように`--preout` フラグを使用します。

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

次のように簡易版の`-p` も使用できます。

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

行番号は、ソースファイルとプロプロセス後のファイルとの間で維持されます。

ファイルのインクルード

このプリプロセッサでは SPA ソースに他のファイルをインクルードできますが、それには`.include` ディレクティブを使用します。構文は次のとおりです。

```
.include "path_relative_to_dir_of_current/file.spa"
```



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

インクルードファイル名が引用符で囲まれている場合、SPA はそのファイルパスを、現在解析中のファイルのパスからの相対パスとみなします。ファイル名が角括弧で囲まれている場合、SPA はそのファイルパスを、コマンドラインで `-I` パラメータを使って指定されたインクルードパスからの相対パスとみなします。インクルードパスの検索は、コマンドラインで指定された順番で行われます。

絶対パス (Windows では `c:\path\to\file.spa` の形式) を使ってファイルを指定することもできます。その場合、ファイル名を引用符と角括弧のどちらで囲んでも同じになります。

文字列の連結

このプリプロセッサでは、`##` 演算子による文字列連結がサポートされています。オペランドはリテラル文字列、プリプロセッサシンボルのいずれかになります。

マクロ

マクロを使うと、コードの管理がもっと容易になります。マクロを宣言するには、次のように `.macro/.endmacro` ディレクティブを使用します。

```
// 仮想レジスタを宣言し、それを PI で初期化するマクロ
.macro LoadPi( regName )
    .reg regName
    ilf32 regName, 3.141592653589f
.endmacro
```

宣言済みのマクロは、他のマクロ宣言の本体部分も含め、ほとんどすべての場所で使用できます。上のマクロの呼び出しは次のようになります。

```
.func MyFunction

    // ...

    // PiReg という名前の仮想レジスタを宣言し、それを PI で初期化します
    LoadPi (PiReg)

    // ...
.endfunc
```

プリプロセッサはマクロ呼び出しを検出するたびに、その呼び出しを、指定されたパラメータが適用されたマクロ本体で置き換えます。

マクロには 0 個以上のパラメータを指定できます。パラメータを持たないマクロは次のように宣言されます。

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

マクロのパラメータとしてのマクロ

あるマクロの名前を別のマクロに引数として渡すことができます (C で関数ポインタを引数として渡すのに似ています)。したがって、マクロの動作を、引数として渡された他のマクロに基づいて変えることができます。

マクロのローカルレジスタ

マクロを宣言する際に、一時的なローカルレジスタを使用したい場合が考えられます。これを実現するには、`.localreg` プリプロセッサディレクティブを使用します。`.localreg` を使って宣言された各レジスタには、そのマクロの本体内でしかアクセスできません。プリプロセッサはマクロをインスタンス化するたびに、`.localreg` を使って宣言された各レジスタの一意名を生成します。このため、不要な反依存の発生が避けられ、マクロを複数回呼び出しても名前の衝突が発生しないことになります。

```
.macro MyMacro
    .localreg tmpRegA // 一意のローカルレジスタ名を生成します
    .localreg tmpRegB // 一意のローカルレジスタ名を生成します
    // ...
.endmacro
```

マクロ一意 ID 演算子

SPA は、マクロの各インスタンスに一意の数値 ID を割り当てます。マクロの宣言時にこの ID を参照するには、`@@` 演算子を使用します。この演算子と文字列連結演算子を組み合わせると、呼び出されるたびに一意の変数名を生成するマクロを宣言することができます。たとえば、マクロ UID 演算子と文字列連結演算子を使って、`.localreg` ディレクティブの動作を模倣することができます（ただし、`.localreg` をできるだけ使用することをお勧めします。マクロの可読性や管理のしやすさが高まるからです）。

ファイルの依存関係

SPA は、生成したオブジェクトファイルの依存情報を出力することができます。この機能を有効にするには、`-emitdeps` というコマンドライン引数を使います。依存関係はオブジェクトファイルと同じディレクトリのファイルに出力されます。ファイル名は、オブジェクトファイルと同じですが、ファイル拡張子は `.o` でなく `.d` です。（オブジェクトファイルに拡張子 `.o` がない場合、依存関係ファイル名はオブジェクトファイル名に `.d` を付加したものになります）。

依存関係ファイルのパスは、すべて、現在の作業ディレクトリからの相対パスです。依存関係ファイルのフォーマットは、`make` ユーティリティの要求するフォーマットに基づいています。

また、（一部のコンパイラの `-MP` オプションと同じように）それぞれの依存関係のために「擬似ターゲット」を追加したい場合には、代わりに `-emitdeps` オプションを使います。コマンドラインから SPA に渡されたソースファイルには、擬似ターゲットが作成されないことに注意してください。

5 デバッグ

SPAはDWARF2 デバッグ形式をサポートしているため、デバッガで関数を実行している際に、そのソースとアセンブリを混在して表示させることができます。また、「Locals」ビューを使えば、仮想レジスタにその時点で割り当てられているハードウェアレジスタや値を確認することもできます。デバッグを有効にするには--dwarf2 フラグを使用します。図 4 に、デバッガ内に表示されたSPA関数を示します。

図 4 デバッガ内に表示された SPA 関数

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_c0db000000010002
0088A4 217FF49A brnz     r026, local_c0de000000010002      ?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
chiftUi	\$r040	vector uint32_t	\$r040

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

重要： ソースコード内のある仮想レジスタがデバッガの「Locals」ビューに表示されない場合、その仮想レジスタを使用するすべての命令が SPA の最適化によって削除されたことを意味します。これは、「@result」のような、SPA によって自動的に宣言された仮想レジスタでも起こる可能性があります。

トラブルシューティング

SPA コードのステップ実行を行っている際にデバッグ情報が表示されない場合には、次の点を確認してください。

- SPA でコードをビルドする際にデバッグ情報を有効にしたかどうか（コマンドラインで「--dwarf2 true」を指定します。現時点ではデフォルトで有効になります）。
- SPU ELF ファイルの検索場所をデバッガに知らせているかどうか（ProDG Debugger で Tools → Options → Projects → Search Directories を選択し、SPU ELF files セクションに SPU ELF ファイルへのパスを追加します）。
- デバッガがソースコードを表示するように設定されているかどうか（ProDG Debugger の SPU ソースウィンドウで右クリックし、Disassembly サブウィンドウの設定を確認してください）。

6 FAQ（よくある質問）

Cコンパイラによって生成されたアセンブラをSPAで最適化できますか？

これに対する簡単な答えは、「おそらく可能ですが、**極力**そうしないようにすべきです」となります。SPAでは仮想レジスタと物理レジスタを混在できるため、コード内の構成要素がSPAでサポートされているものである（`bi` や `brsl` などでない）限り、技術的には正常に動作するはずです。しかしながら、その結果は非常に悪いものになります。すでに割り当てられた物理レジスタは変更されないため、不要な依存関係を多数抱え込むことになるからです。

ユーザが自身の行っていることを本当に理解しているのでない限り、決してお勧めできません。

SPAによって生成される「.s」ファイルと「.o」ファイルの違いは何ですか？

SPAは独自の内部アセンブラを使って、バイナリコードを直接生成することができます。さらに、次のような追加デバッグ情報を含む「.s」出力モードも存在しています。

- ペアリング/実行単位
- パイプラインからのフィードバック
- パイプライン化されたループイテレーションを示す追加のコメント

「.s」ファイルはgccでもアセンブル可能な「はず」であり、同じオブジェクトファイルが生成されます（ただし、DWARF デバッグ情報は除く）。

どうしてSPA関数がリンクできないのでしょうか？

関連するオブジェクトファイルがすべて正しくリンクされていると仮定した場合、おそらく宣言内で「`extern "C"`」が記述されていない可能性があります。

SPAはC++の装飾名を生成せず、関数のオーバーロードは行えません。

レジスタが不足したらどうしたらよいですか？

`.func` ディレクティブを使って関数を宣言した場合、SPAは最小限のことしか行いません。レジスタが不足した場合には処理が失敗します。典型的なメッセージを以下に示します。

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

C ABIのデフォルト定義を使用している場合には、`.cfunc`ディレクティブを使って関数を宣言すれば、48個の追加レジスタを非常に容易に扱えるようになります。詳しくは「[入力、出力、およびABIに関する注意点](#)」を参照してください。

ループ統計がループカーネルに一致しないのはなぜですか？

おそらく、以下のような余分な命令のことを言っているのだと思います。

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

これらの命令は、スケジューラによってスケジュールの空スロットに追加された「移動命令」ですが、その目的は、1つのイテレーションよりも長いレジスタ存続期間を分割し、モジュロ変数展開を回避することです。

ただし、スケジューラはこれらの移動命令のために余分なサイクルを挿入することは決してありません。別の方法がとられます。スケジューラが「i」サイクルの処理に失敗すると、次に「i+1」が試みられますが、当然ながら、移動命令に利用可能なスロットが2個増えることになります。

どのようにすれば反復制約を取り除くことができますか？

これが正常であるような何の問題もない状況（IIRフィルタを使用する場合など）も考えられます。しかしながら、通常、同じポインタが非常に長い計算の入力/出力の両方として再利用されたことを意味します。こうした状況で行うべきことは、「誘導変数の反転（induction variable reversal）」と呼ばれますが、これは基本的に、ポインタ更新と処理コード間のカップリングを断ち切ることを意味します。これについては、「[反復制約とリソース制約](#)」（第2章「[ソフトウェアパイプライン化](#)」）を参照してください。

通常、反復制約はループ誘導反転を行うことで解決できます。現時点では、オプティマイザによって自動的に行われません。たとえば、以下の例を考えます。

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

    // 若干の待ち時間を追加するだけの処理
    fma rOut, rIn, rKA, rKB
    fma gOut, gIn, gKA, gKB
    fma bOut, bIn, bKA, bKB

    // ストア（最初のロードから 12 サイクル以上経過していますが、
    // ポインタはまだ有効であるはずです）
    stqd rOut, 0x00(pColours)
    stqd gOut, 0x10(pColours)
    stqd bOut, 0x20(pColours)

    // ここでポインタを更新しますが、最初のロードから少なくとも、
    // 6 サイクル (LQD) + 6 サイクル (FMA) = 12 サイクル経過しています。
    ai pColours, pColours, 0x30

    // ループ管理
    ai count, count, -1
    brnz count, Loop :
```

これは反復によって大きく制限されています。これはユーザの望むところではありません。

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

誘導反転を行うと、次のように反復制約から解放されます。

```
Loop:
    // *** ポインタ更新のカップリングが取り除かれています ***
    ai pColours, pColours, 0x30
```

```
// 読み取り (オフセットが変更されていることに注意)
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

// 書き込み (オフセットが変更されていることに注意)
stqd rOut, -0x30(pColours)
stqd gOut, -0x20(pColours)
stqd bOut, -0x10(pColours)

ai count, count, -1

brnz count, Loop
```

この時点で反復に含まれるのは、ポインタ更新 (2 サイクルの AI – 大幅な単純化) だけになっています。

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

スケジューラは存続期間の長すぎるレジスタの分割/名前変更を自動的行います (ただし、それらが反復パス上に存在している場合は除く)。したがって、2 番目の状況では、pColours が少なくとも 15 サイクル存続しなければならないように見えても、それは問題ではありません。実際には本物のレジスタではなく、pColours の 1 つ (または複数) のコピーだからです。

7 ヒントとテクニック

バランスの維持

リソースによって制約されたループで odd パイプラインと even パイプラインの使用率にアンバランスが生じている場合には、even 命令を機能的に同等な odd 命令で置き換える（あるいはその逆を行う）ことでパイプラインのバランスを回復できることが少なくありません。これには、最小開始間隔が短くなる（つまりループが高速になる）、というメリットがあります。

この例としては、shufb 命令（odd パイプ）と selb 命令（even パイプ）が挙げられます。特定の処理（すべてではない）では、どちらかの命令から同等の機能が得られます。

アラインされていないメモリからの読み取り

バイトアラインされているけれども、かならずしもクワッドワードアラインされていない float 配列へのポインタが存在する場合、それらの配列がまるでクワッドワードアラインされているかのようにレジスタへのロードを行う方法を示したのが、以下のコードです（このサンプルコードは単なる例であり、さらに最適化することが可能である点に注意してください。たとえば、1つのループイテレーションに2つのロード命令を含め、「mov」を削除することもできます）。

```
// 対象の float は「pInput」によって指されており、
// 「count」という名前のループカウンタが存在しているとします。

// 入力シャッフルマスクを計算します
.reg IN_SHUFFLE, DDDDDDDDDDDDDDDDD, tmp1
// 最初のクワッドワードのすべてのワードを選択します。
il128 IN_SHUFFLE, "ABCD"
// 最初のクワッドワードの4番目のバイトを複製します。
il128 DDDDDDDDDDDDDDDDD, "DDDDDDDDDDDDDDDD"

// 入力ポインタの、クワッドワード境界からの相対オフセットを検索します。
andi tmp1, pInput, 0xF
shufb tmp1, tmp1, tmp1, DDDDDDDDDDDDDDDDD // オフセットバイトを複製します。
// シャッフルマスクによって選択されるバイトを調整します。
// たとえば、tmp1==4の場合、マスクは「BCDa」になります。
a IN_SHUFFLE, IN_SHUFFLE, tmp1

.reg INA, INB, IN

// pInput がクワッドワードアラインされていなくても
// ロードは常にクワッドワードアラインされることに注意してください。
lqd INB, 0(pInput)

loop:
// 隣り合うクワッドワードを INA と INB にロードします。
mov INA, INB
ai pInput, pInput, 0x10
lqd INB, 0(pInput)

// INA と INB から、適切にアラインされたクワッドワードを選択します。
```



```
shufb IN,INA,INB,IN_SHUFFLE
```

```
// この時点で 4 つの float が IN に格納されています。  
... IN を使って何らかの処理を行います....
```

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

簡単なループ展開

ソフトウェアのパイプライン処理では除去できない命令レイテンシを隠すために利用できるループ展開の準備として、ループ本体をマクロとして書くことをお勧めします。こうすれば、展開後にループ本体のコピーを 1 つ保持するだけで済むからです。