

AI エンジン カーネル コーディング

ベスト プラクティス ガイド

UG1079 (v2020.2) 2021 年 2 月 4 日

この資料は表記のバージョンの英語版を翻訳したもので、内容に相違が生じる場合には原文を優先します。資料によっては英語版の更新に対応していないものがあります。日本語版は参考用としてご使用の上、最新情報につきましては、必ず最新英語版をご参照ください。



改訂履歴

次の表に、この文書の改訂履歴を示します。

セクション	改訂内容
2021 年 2 月 4 日 バージョン 2020.2	
初版。	なし

目次

改訂履歴.....	2
第 1 章: 概要.....	5
この資料に関連する設計プロセス.....	7
AI エンジン アーキテクチャ 概要.....	7
スカラー プロセッシング ユニット.....	9
ベクター プロセッシング ユニット.....	11
AI エンジンのメモリ.....	12
AI エンジン タイルのインターフェイス.....	13
ツール.....	13
第 2 章: シングル カーネルのプログラミング.....	15
組み込み関数.....	15
カーネル プラグマ.....	17
カーネル コンパイラ.....	17
カーネルのシミュレーション.....	17
カーネルの入力および出力.....	18
スカラーおよびベクター プログラミングの概要.....	18
AI エンジンのデータ型.....	19
ベクター レジスタ.....	20
アキュムレータ レジスタ.....	22
データ型変換.....	23
ベクターの初期化、読み込み、および保存.....	24
ベクター レジスタ レーンの並べ替え.....	26
ループ.....	39
ループのソフトウェア パイプライン処理.....	41
restrict キーワード.....	44
浮動小数点演算.....	44
Vitis IDE およびレポートの使用.....	44
第 3 章: インターフェイスに関する考慮事項.....	48
AI エンジン間のデータ移動.....	48
データ通信におけるウィンドウとストリームの違い.....	50
フリーランニング AI エンジン カーネル.....	51
ランタイム パラメーターの指定.....	52
AI エンジンと PL カーネル間のデータ移動.....	54
GMIO を介した DDR メモリ アクセス.....	54
第 4 章: デザイン解析およびプログラミング.....	56

AI エンジンへのアルゴリズムのマップ	56
シングル カーネルのコーディング例	58
複数カーネルのコーディング例: FIR フィルター	68
 付録 A: その他のリソースおよび法的通知	 76
ザイリンクス リソース	76
Documentation Navigator およびデザイン ハブ	76
参考資料	76
お読みください: 重要な法的通知	77

概要

Versal™ AI コア シリーズは、AI エンジンを使用した飛躍的な AI 推論アクセラレーションを提供し、現在のサーバークラス CPU の 100 倍を超える計算パフォーマンスを達成可能です。このシリーズは、非常に広い帯域幅のダイナミック ワークロードおよびネットワーク用のクラウドを含む広範囲のアプリケーション用に設計されており、高度な安全性およびセキュリティの機能も提供します。AI およびデータサイエンティスト、ソフトウェア開発者、ハードウェア開発者すべてが、高計算密度を活用してアプリケーションのパフォーマンスをアクセラレーションできます。AI エンジンは高度な信号処理計算機能を備えているので、無線、5G、バックホール、その他の高パフォーマンス DSP アプリケーションなど、高度に最適化されたワイヤレス アプリケーションに適しています。

AI エンジンは単一命令複数データ (SIMD) ベクター ユニットを含む超長命令語 (VLIW) プロセッサのアレイであり、計算負荷の高いアプリケーション (特にデジタル信号処理 (DSP))、5G ワイヤレス アプリケーション、および機械学習 (ML) などの AI テクノロジー用に高度に最適化されています。

AI エンジン アレイでは、次の 3 つのレベルの並列処理がサポートされています。

- 命令レベルの並列処理 (ILP): VLIW アーキテクチャにより、1 クロック サイクルで複数の演算を実行できます。
- SIMD: ベクター レジスタにより、複数の要素 (8 個など) を並列計算できます。
- マルチコア: AI エンジン アレイにより、400 個までの AI エンジンを並列実行できます。

命令レベルの並列処理には、スカラー演算、最大 2 つの移動、2 つのベクター読み出し (ロード)、1 つのベクター書き込み (ストア)、および実行可能な 1 つのベクター命令が含まれ、合計で 1 クロック サイクルごとに 7 ウェイ VLIW 命令を実行できます。データ レベル並列処理は、1 クロック サイクルごとに複数のデータセットに対してベクター レベル命令を実行することにより達成します。

各 AI エンジンには、ベクター プロセッサとスカラー プロセッサ、専用プログラム メモリ、ローカル 32 KB データ メモリ、それ自身および行によって異なる方向に隣接する 3 つの AI エンジンに含まれるローカル メモリへのアクセスが含まれます。また、DMA エンジンおよび AXI4 インターコネクト スイッチへのアクセスも含まれており、ストリームを使用してほかの AI エンジン、プログラマブル ロジック (PL)、または DMA とも通信できます。AI エンジン アレイおよびインターフェイスの詳細は、『Versal ACAP AI エンジン アーキテクチャ マニュアル』 (AM009: [英語版](#)、[日本語版](#)) を参照してください。

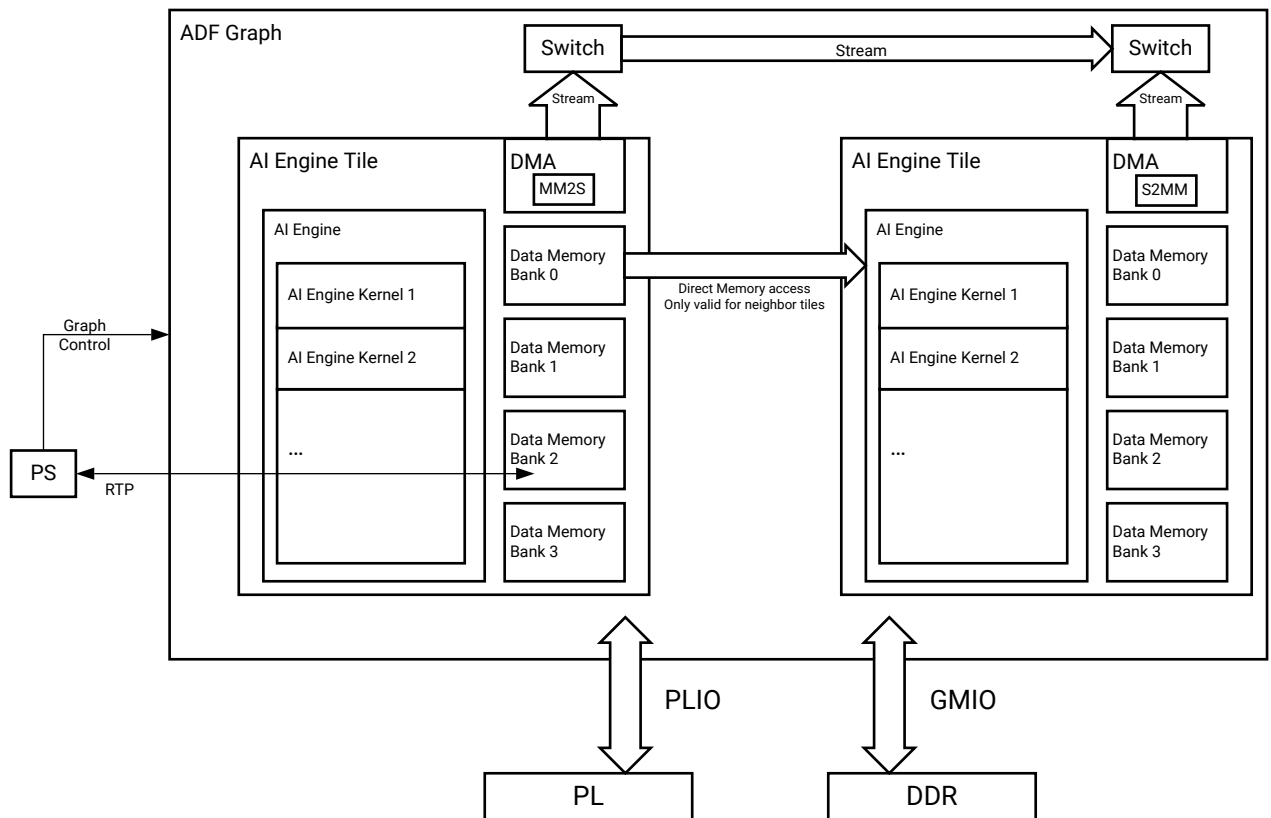
ほとんどの標準 C コードは AI エンジン用にコンパイルできますが、ハードウェアで提供される並列処理を最大限に活用するには、記述し直すことが必要な場合があります。AI エンジンの最大の利点は、各クロック サイクルで 2 つのベクターを使用して積和 (MAC) 演算を実行、次の演算用に 2 つのベクターをロード、前の演算からのベクターを格納、およびポインターをインクリメントまたは別のスカラー演算を実行できることです。組み込み関数と呼ばれる特殊な関数を使用すると、AI エンジン ベクター プロセッサおよびスカラー プロセッサをターゲットにし、複数の一般的なベクター関数およびスカラー関数をインプリメントできるので、ターゲット アルゴリズムに集中できます。AI エンジンには、ベクター ユニットに加え、非線形関数およびデータ型変換に使用可能なスカラー ユニットも含まれます。

AI エンジン プログラムは、C++ で記述されたデータフロー グラフ (適応型データフロー グラフ) 仕様で構成されています。この仕様は、AI エンジン コンパイラを使用してコンパイルおよび実行できます。適応型データフロー (ADF) グラフ アプリケーションは、計算カーネル関数を表すノードとデータ接続を表すエッジで構成されます。アプリケーションのカーネルは、ADF グラフ仕様の基本的な構築ブロックであり、AI エンジン上で実行されるようにコンパイルできます。ADF グラフは、並列動作する AI エンジン カーネルの**カーン プロセスネットワーク**です。AI エンジン カーネルは、データ ストリームに対して処理を実行します。これらのカーネルは、入力データ ブロックを消費し、出力データ ブロックを生成します。カーネルには、非同期または同期のスタティック データまたはランタイム パラメーター (RTP) 引数を含めることもできます。

次の図に、プロセッシング システム (PS)、プログラマブル ロジック (PL)、および DDR メモリを含む ADF グラフの概念図を示します。次のものが含まれます。

- AI エンジン: 各 AI エンジンは、スカラー ユニット 1 つ、ベクター ユニット 1 つ、ロード ユニット 2 つ、およびストア ユニット 1 つを含む VLIW プロセッサです。
- AI エンジン カーネル: カーネルは、AI エンジンで実行する C/C++ で記述されます。
- ADF グラフ: ADF グラフは、1 つの AI エンジン カーネル、またはデータ ストリームで接続された複数の AI エンジン カーネルを含むネットワークです。PL、グローバル メモリ、PLIO (プログラマブル ロジックとのストリーム接続を作成するのに使用されるグラフ プログラミングのポート属性) など特定のコンストラクトを使用した PS、GMIO (グローバル メモリとの外部メモリ マップ接続を作成するために使用されるグラフ プログラミングのポート属性)、および RTP とデータをやり取りします。

図 1: ADF グラフの概念図



X25022-011521

この資料では、AI エンジン カーネルのプログラミングに焦点を置き、シングル カーネル プログラミングだけでなく、アプリケーションを複数のカーネルに分割してシステム全体のパフォーマンスを達成するために必要な概念であるカーネル間のデータ通信など、複数カーネル プログラミングのいくつかの側面についても説明します。

グラフの構築、コンパイル、シミュレーション、およびハードウェア フローの詳細は、『Versal ACAP AI エンジン プログラミング環境ユーザー ガイド』(UG1076)を参照してください。

この資料に関連する設計プロセス

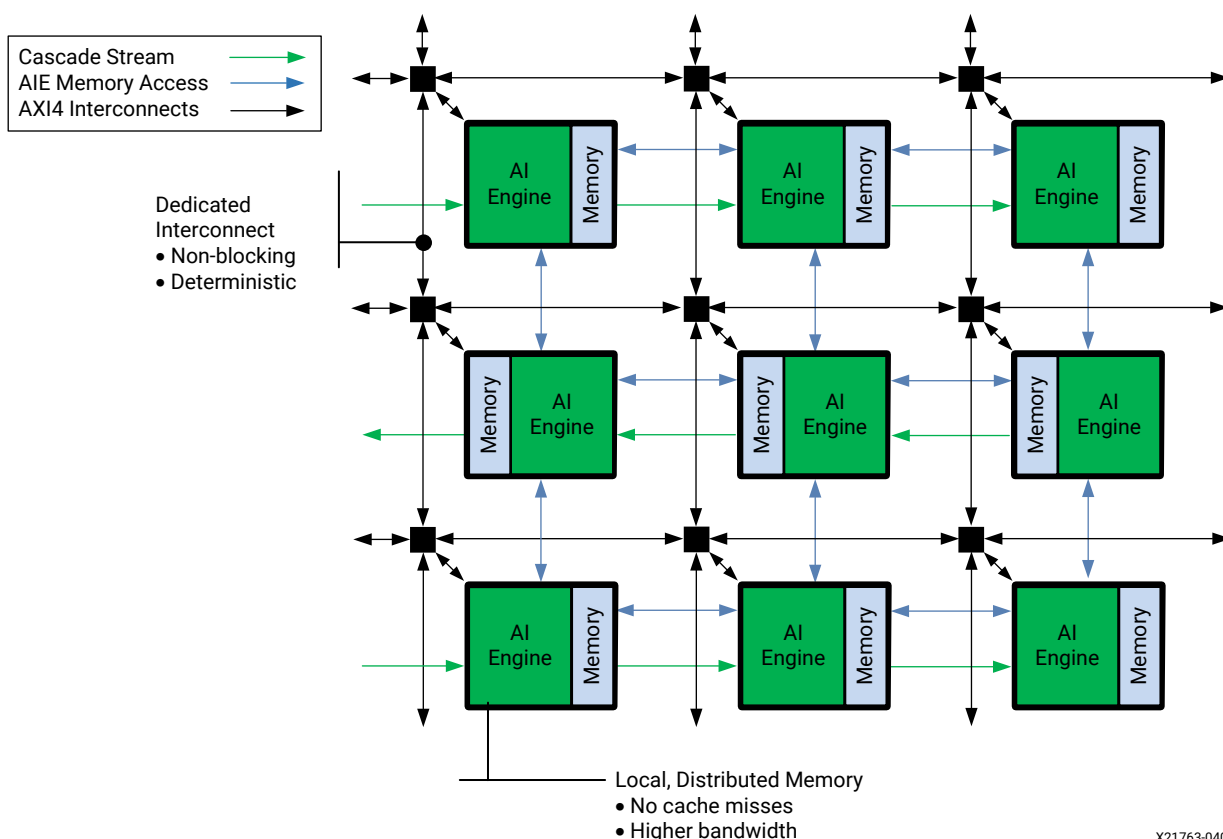
ザイリンクスの資料は、開発タスクに関連する内容を見つけやすいように、標準設計プロセスに基づいて構成されています。Versal™ ACAP デザイン プロセスの[デザイン ハブ](#)は、ザイリンクス ウェブサイトからアクセスできます。この資料では、次の設計プロセスについて説明します。

- AI エンジン開発: AI エンジン グラフおよびカーネルの作成、ライブラリの使用、シミュレーションのデバッグおよびプロファイリング、アルゴリズムの開発を実行します。PL と AI エンジン カーネルの統合も含まれます。

AI エンジン アーキテクチャ概要

AI エンジン アレイは、AI エンジン タイルの 2D アレイで構成されており、各 AI エンジン タイルには、AI エンジン、メモリ モジュール、およびタイル インターコネクト モジュールが含まれます。次の図に、AI エンジンの 2D アレイの概要を示します。

図 2: AI エンジン アレイ

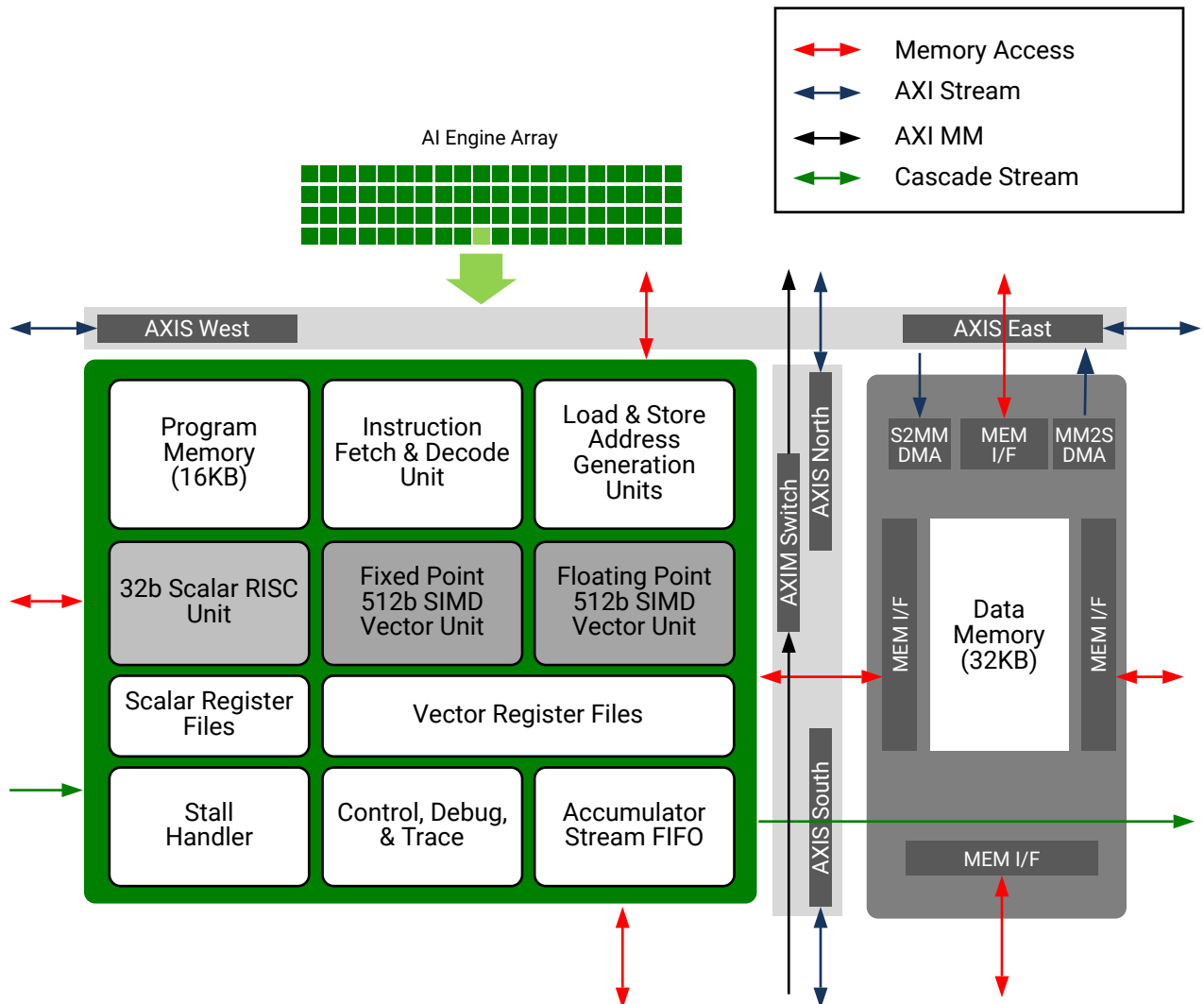


X21763-040519

メモリ モジュールは、アレイ内のタイルの位置によって、その上、下、右、または左に隣接する AI エンジンと共有されます。AI エンジンは、その上、下、右、左、および独自のメモリ モジュールにアクセスできます。AI エンジンは、専用のメモリ アクセス インターフェイスを介してこれらの隣接するメモリ モジュールにアクセスします。各アクセスは、最大 256 ビット幅にできます。AI エンジンは隣接する AI エンジンからカスケード ストリーミング データを送受信することもできます。カスケード ストリームは、左から右、または右から左への一方方向のストリームで、次の行に移動するときに折り返されます。AXI4 インターコネクト モジュールは、AI エンジン タイル間にストリーミング接続を提供し、ストリーミング インターフェイスとメモリ モジュール間に stream-to-memory (s2mm) または memory-to-stream (mm2s) 接続を提供します。また、インターコネクト モジュールは隣接するインターコネクト モジュールにも接続されており、グリッドのような柔軟な配線機能を提供します。

次の図に、1 つの AI エンジン タイルのアーキテクチャを示します。

図 3: AI エンジン タイルの詳細



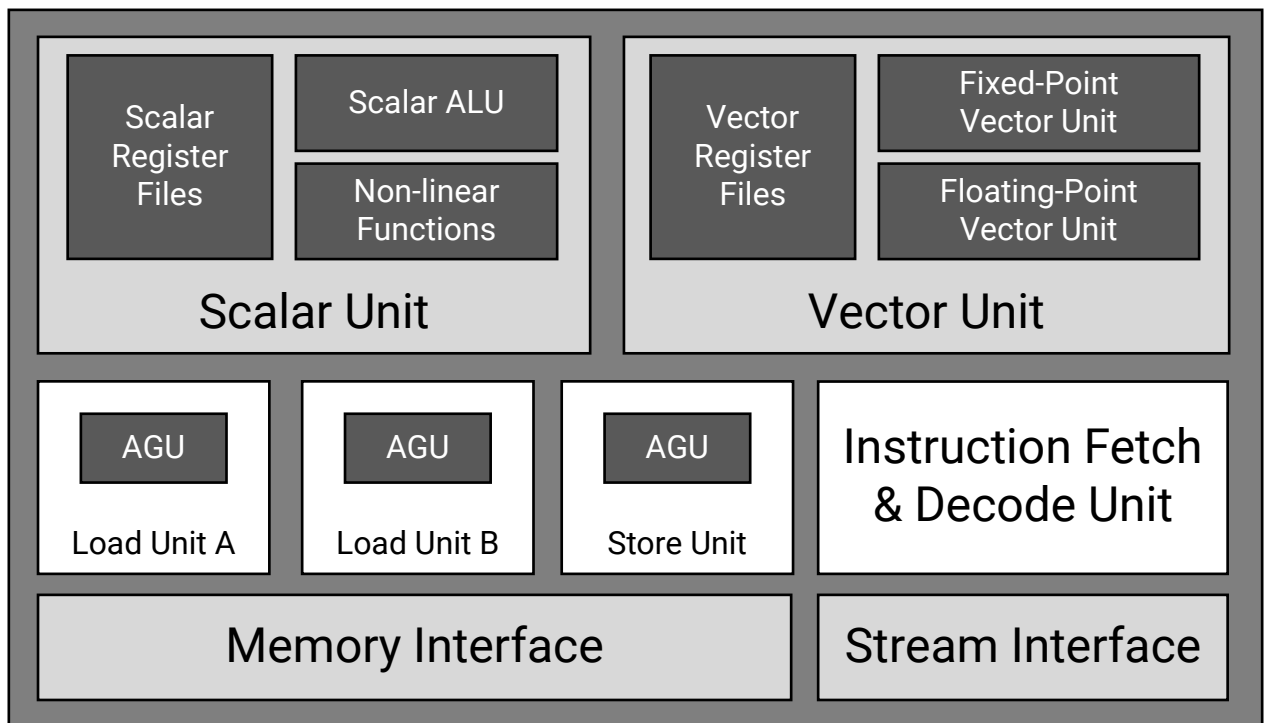
X24805-111120

各 AI エンジン タイルには、完全にプログラム可能な 32 ビットの AXI4-Stream クロスバーの AXI4-Stream スイッチがあり、バックプレッシャー付きの回線交換ストリームとパケット交換ストリームの両方がサポートされます。AXI4-Stream スイッチは、MM2S DMA および S2MM DMA を介して AI エンジン データ メモリとの間にストリームアクセスを提供します。また、このスイッチには、2 つの 16 深度 33 ビット (32 ビット データ + 1 ビット TLAST) 幅の FIFO も含まれます。これらの FIFO は、片方の FIFO の出力をもう片方の FIFO の入力に回路交換してチェーン接続し、32 深度の FIFO を形成できます。

次の図に示すように、AI エンジンは高度に最適化されたプロセッサであり、単一命令複数データ (SIMD) と超長命令語 (VLIW) プロセッサを搭載され、スカラー ユニット、ベクター ユニット、2 つのロード ユニット、1 つのストア ユニット、および命令フェッチ/デコード ユニットが含まれます。1 つの VLIW 命令では、最大 2 つのロード、1 つのストア、1 つのスカラー演算、1 つの固定小数点または浮動小数点のベクター演算、および 2 つの移動命令をサポートできます。

AI エンジンには、複数のアドレス指定モードをサポートする 3 つのアドレス生成ユニット (AGU) も含まれます。2 つの AGU は 2 つのロード ユニット専用で、1 つの AGU はストア ユニット専用です。

図 4: AI エンジン



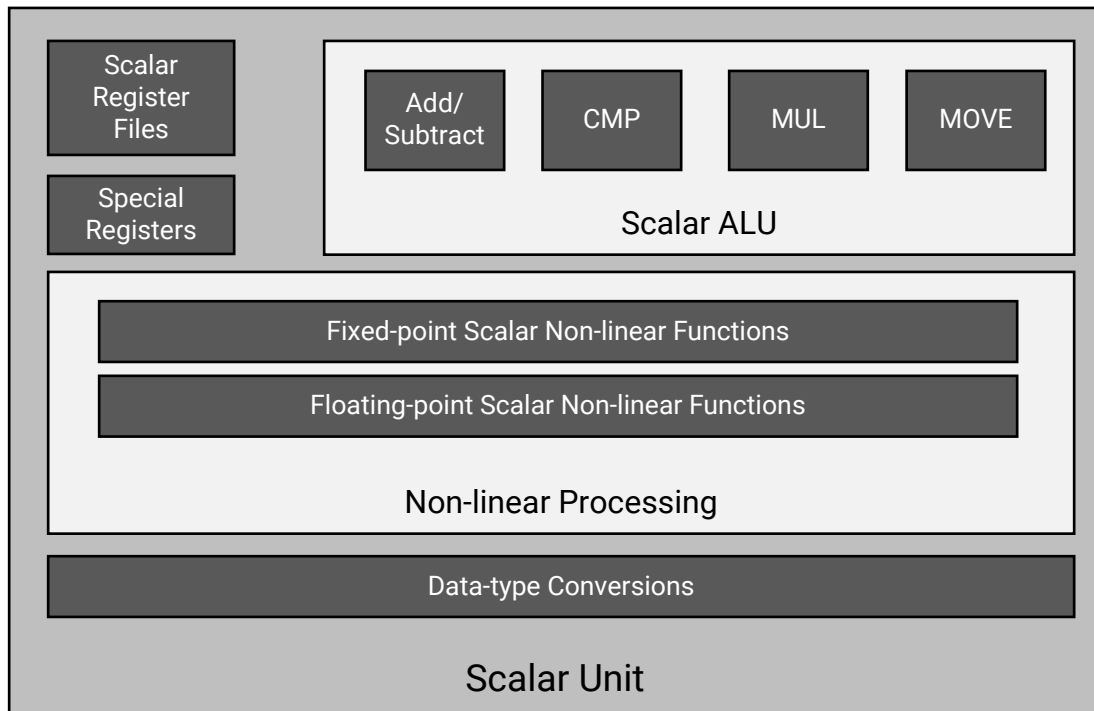
X25020-011321

ベクター処理ユニット、AI エンジン メモリ、および AI エンジン タイル インターフェイスの詳細は、次のセクションを参照してください。

スカラー プロセッシング ユニット

次の図に、スカラー ユニットのサブコンポーネントを示します。スカラー ユニットは、汎用プロセッサと同様、プログラム制御 (分岐、比較)、スカラー数学演算、非線形関数、およびデータ型変換に使用されます。汎用プロセッサと同様、汎用 C/C++ コードを使用できます。

図 5: スカラー プロセッシング ユニット



X25019-011521

入力および出力の格納には、レジスタ ファイルが使用されます。ポインター演算と、汎用およびコンフィギュレーション用に専用のレジスタがあります。特殊なレジスタには、スタック ポインター、循環バッファ、およびゼロ オーバーヘッド ループが含まれます。AI エンジンでは、固定小数点および浮動小数点精度の 2 つのタイプのスカラー初等非線形関数がサポートされます。

固定小数点の非線形関数には、次が含まれます。

- サインおよびコサイン
- 絶対値 (ABS)
- 先行ゼロのカウント (CLZ)
- 最小値または最大値を求めるための比較 (小なり (LG)/大なり (GT))
- 平方根
- 逆平方根および逆数

浮動小数点の非線形関数には、次が含まれます。

- 平方根
- 逆平方根
- 逆数
- 絶対値 (ABS)
- 最小値または最大値を求めるための比較 (小なり (LG)/大なり (GT))

AI エンジンの論理演算ユニット (ALU) は、1 サイクルごとに 1 命令の発行レートで次の演算を実行します。

- 32 ビットの整数の加算および減算。この演算のレイテンシは 1 サイクルです。
- 32 ビット整数に対するビット論理演算 (BAND、BOR、および BXOR)。この演算のレイテンシは 1 サイクルです。
- 整数の乗算: 32 x 32 ビット。32 ビットの結果を R レジスタ ファイルに格納します。この演算のレイテンシは 3 サイクルです。
- シフト演算: 左シフトと右シフトの両方がサポートされます。この演算のレイテンシは 1 サイクルです。

データ型変換は、`float2fix` および `fix2float` を使用して実行できます。この変換では、`sqrt`、`inv`、`inv_sqrt` 固定点演算もサポートされます。

スカラー プログラミング

コンパイラとスカラー ユニットにより、標準 C データ型を使用できます。次の表に、標準 C データ型とその精度を示します。float と double を除くすべてのデータ型は、符号付きおよび符号なし接頭辞をサポートします。

表 1: スカラー データ型

データ型	精度	注記
char	8 ビット符号付き	
short	16 ビット符号付き	
int	32 ビット符号付き	ネイティブ サポート
long	64 ビット符号付き	
float	32 ビット	
double	64 ビット	ソフトウェア ライブラリを使用してエミュレートされます。スカラー プロシージャには FPU は含まれません。

分岐などの制御フロー文は、ベクター命令がある場合でもスカラー ユニットで処理されることに注意してください。この概念は、AI エンジンのパフォーマンスを最大限にするために重要です。

ベクター プロセッシング ユニット

ベクター ユニットには、128 個の 8 ビット固定小数点乗算器で構成される固定小数点ユニットと、8 個の単精度浮動小数点乗算器で構成される浮動小数点ユニットが含まれています。ベクター レジスタおよび並べ替えネットワークは、固定小数点乗算器と浮動小数点乗算器の間で共有されます。最大パフォーマンスは、オペランドで使用するデータ型のサイズによって異なります。次の表に、ベクター プロセッサで命令ごとに実行可能な MAC 演算の数を示します。

表 2: AI エンジン ベクターの精度サポート

X オペランド	Z オペランド	出力	1 クロックごとの MAC 演算回数
8 実数	8 実数	48 実数	128
16 実数	8 実数	48 実数	64
16 実数	16 実数	48 実数	32
16 実数	16 複素数	48 複素数	16
16 複素数	16 実数	48 複素数	16

表 2: AI エンジン ベクターの精度サポート (続き)

X オペランド	Z オペランド	出力	1 クロックごとの MAC 演算回数
16 複素数	16 複素数	48 複素数	8
16 実数	32 実数	48/80 実数	16
16 実数	32 複素数	48/80 複素数	8
16 複素数	32 実数	48/80 複素数	8
16 複素数	32 複素数	48/80 複素数	4
32 実数	16 実数	48/80 実数	16
32 実数	16 複素数	48/80 複素数	8
32 複素数	16 実数	48/80 複素数	8
32 複素数	16 複素数	48/80 複素数	4
32 実数	32 実数	80 実数	8
32 実数	32 複素数	80 複素数	4
32 複素数	32 実数	80 複素数	4
32 複素数	32 複素数	80 複素数	2
32 単精度浮動小数点	32 単精度浮動小数点	32 単精度浮動小数点	8

X オペランドは 1024 ビット幅で、Z オペランドは 256 ビット幅です。コンポーネントの使用に関しては、上記の表の最初の行を考慮してください。乗算器オペランドは同じ 1024 ビットおよび 256 ビット入力レジスタから供給されますが、一部の値は複数の乗算器に送信されます。128 個の 8 ビット シングル乗算器があり、結果は後置加算および累算されて、16 個または 8 個のアクキュムレータ レーン (各 48 ビット) に入力されます。

データパスの最大パフォーマンスを計算するには、命令ごとの MAC の数と AI エンジン カーネルのクロック周波数を乗算します。たとえば、16 ビット入力ベクター X および Z では、ベクター プロセッサで命令ごとに 32 個の MAC を達成できます。最低速のスピード グレード デバイスのクロック周波数を使用すると、次のようになります。

$$32 \text{ 個の MAC} \times 1 \text{ GHz クロック周波数} = 32 \text{ ギガ MAC 演算/秒}$$

AI エンジンのメモリ

各 AI エンジンには 16 KB のプログラム メモリがあり、128 ビットの命令を 1024 個格納できます。AI エンジンの命令は最大 128 ビット幅で、複数の命令フォーマットおよび可変長命令をサポートしており、プログラム メモリ サイズを削減できます。最適化された内側ループの外にある多くの命令には、短いフォーマットを使用できます。

各 AI エンジン タイルには 8 つのメモリ バンクがあり、各メモリ バンクは 256 ワード x 128 ビットのシングルポート メモリです (合計 32 KB)。各 AI エンジンは、隣接する上下左右のタイルにあるメモリにアクセスでき、それ自体のデータ メモリを含め合計 128 KB のデータ メモリを使用できます。スタックはデータ メモリのサブセットです。スタック サイズおよびヒープ サイズのデフォルト値は 1 KB です。この値は、コンパイラ オプションを使用して変更できます。

論理表現では、128 KB メモリを 1 つの連続した 128 KB ブロックまたは 4 つの 32 KB ブロックと見ることができ、各ブロックは 4 つの奇数バンクと 4 つの偶数バンクに分割できます。1 つの偶数バンクと 1 つの奇数バンクは、ダブルバンクを構成します。AI エンジン アレイの端にある AI エンジンには、隣接するタイルが少ないので、使用可能なメモリも少なくなります。

各メモリポートは、256 ビット/128 ビット ベクター レジスタ モードまたは 32 ビット/16 ビット/8 ビット スカラー レジスタ モードで動作できます。256 ビット ポートは、偶数と基数のメモリ バンクをペアにして作成されます。8 ビットおよび 16 ビットのストアは、Read-Modify-Write 命令としてインプリメントされます。各ポートが異なるバンクにアクセスしている場合、3 つのポートすべての同時動作がサポートされます。

メモリに格納されるデータは、リトル エンディアン形式です。



推奨: データ メモリには、ベクター演算を使用して 128 ビット境界でアクセスすることをお勧めします。

各 AI エンジンには DMA コントローラーが含まれており、ストリーム データをメモリに格納する S2MM (32 ビット データ) とメモリの内容をストリームに書き込む MM2S (32 ビット データ) の 2 つのモジュールで構成されています。S2MM および MM2S には、どちらにも 2 つの独立したデータ チャンネルがあります。

AI エンジン タイルのインターフェイス

データ メモリ インターフェイス、ストリーム インターフェイス、およびカスケード ストリーム インターフェイスは、計算のために AI エンジンに対してデータの読み出しおよび書き込みを実行するプライマリ I/O インターフェイスです。

- データ メモリ インタフェースは、4 方向すべてにあるデータ メモリ モジュールを 1 つの連続したメモリ (合計容量 128 KB) として認識します。AI エンジンには 2 つの 256 ビット幅ロード ユニットと 1 つの 256 ビット幅ストア ユニットがあります。各ロードまたはストアで、128 ビット アライメントを使用して 128 ビットまたは 256 ビット幅でデータにアクセスできます。
- AI エンジンには 2 つの 32 ビット入力 AXI4-Stream インターフェイスと 2 つの 32 ビット出力 AXI4-Stream インターフェイスがあります。各ストリームは入力側と出力側の両方で FIFO に接続されており、AI エンジンは 1 つのストリームで 4 クロック サイクルごとに 128 ビット、1 サイクルごとに 32 ビット アクセスできます。
- 複数の専用カスケード ストリーム インターフェイスを使用してチェーンを構成することにより、1 つの AI エンジンの 384 ビット アキュムレータ データを別の AI エンジンに転送できます。入力ストリームと出力ストリームの両方に 384 ビット幅の 2 段 FIFO があり、AI エンジン間で最大 4 つの値を格納できます。チェーン接続された AI エンジンにより、各サイクルで 384 ビットを送受信できます。カスケード ストリーム チェーンは、同じスループットで動作する複数のカーネルの相対的な密結合を指定します。

AI エンジンプログラミングする際、各 AI エンジンが 2 つの 32 ビット AXI4-Stream 入力、2 つの 32 ビット AXI4-Stream 出力、1 つの 384 ビット カスケード ストリーム入力、1 つの 384 ビット カスケード ストリーム出力、2 つの 256 ビット データ ロード、および 1 つの 256 ビット データ ストアにアクセスできることに注意してください。ただし、命令の長さによっては、同じサイクルでこれらの操作すべて実行できるとは限りません。

ツール

Vitis 統合設計環境

Vitis™ 統合設計環境 (IDE) は、Versal デバイスを含むザイリンクス デバイスのターゲット システム プログラムに使用できます。1 つおよび複数の AI エンジン カーネル アプリケーションの開発をサポートします。このツールには、次の機能があります。

- 最適化された C/C++ コンパイラ: カーネルおよびグラフ コードをコンパイルし、必要な接続、配置、チェックを実行して、デバイス上で適切に機能するようにします。

- サイクル精度シミュレータ: アクセラレーション関数のシミュレータおよびプロファイリング ツールです。
- 高度なデバッグ環境: シミュレーション環境およびハードウェア環境の両方で使用できます。[Variables] ビュー、[Disassembly] ビュー、[Memory] ビュー、[Registers] ビュー、[Pipeline] ビューなどのビューがあります。

Vitis コマンド ライン ツール

ビルド、シミュレーション、出力ファイルおよびレポートの生成を実行するコマンド ライン ツールがあります。

- AI エンジン コンパイラは、カーネルおよびグラフ コードをコンパイルし、AI エンジン プロセッサで実行する ELF ファイルを生成します。
- AI エンジン シミュレータおよび x86simulator は、それぞれサイクル近似シミュレーションおよび論理シミュレーション用のツールです。
- Arm[®] コアのクロス コンパイラは、PS コードのコンパイル用提供されています。
- Vitis コンパイラは、システム全体を統合するためのシステム コンパイルおよびリンク ツールです。
- Vitis アナライザー IDE は、コマンド ライン ツールで生成された出力ファイルおよびレポートを表示して解析するために使用できます。

デザイン フローおよびツールの使用方法については、『Versal ACAP AI エンジン プログラミング環境ユーザー ガイド』 ([UG1076](#)) を参照してください。

シングル カーネルのプログラミング

AI エンジン カーネルは、ネイティブ C/C++ 言語と VLIW スカラーおよびベクター プロセッサをターゲットとする特別な組み込み関数を使用してつまり記述された C/C++ プログラムです。AI エンジン カーネル コードは、Vitis™ コア 開発キットに含まれる AI エンジン コンパイラ (aiecompiler) を使用してコンパイルされます。AI エンジン コンパイラは、カーネルをコンパイルし、AI エンジンで実行する ELF ファイルを生成します。

組み込み関数の詳細は、『Versal ACAP AI エンジン イントリンスクス資料』 ([UG1078](#)) を参照してください。AI エンジン コンパイラおよびシミュレータについては、この章の最初のいくつかのセクションで説明します。

AI エンジンは、ベクター プログラミング用の特殊なデータ型および組み込み関数をサポートしています。必要に応じて、これらの組み込み関数とベクター データ型を使用してスカラー アプリケーション コードを再構築することで、ベクター化されたアプリケーション コードをインプリメントできます。AI エンジン コンパイラは、組み込み関数の演算への割り当て、ベクターまたはスカラー レジスタの割り当てとデータ移動、自動スケジューリング、および VLIW 命令に効率的にパックされたマイクロコードの生成を実行します。

次のセクションで、AI エンジン カーネルでサポートされるデータ型と使用可能なレジスタについて説明します。また、初期化、ロード、ストア、適切なデータ型を使用したベクター レジスタの操作を実行するベクター組み込み関数についても説明します。

AI エンジンで最高のパフォーマンスを達成するためのシングル カーネル プログラミングでの主な目的は、ベクター プロセッサの使用が論理的な最大値に近づくようにすることです。アルゴリズムのベクター化は重要ですが、ベクター レジスタ、メモリ アクセス、およびソフトウェアのパイプライン処理を管理することも必要です。ベクター プロセッサはクロック サイクルごとに 1 つの演算を実行できるので、現在の演算の実行中に次の演算のデータを準備するようにします。ループでのソフトウェア パイプライン処理を使用した最適化は、プラグマを使用して適用できます。たとえば、内側のループにシーケンシャルまたはループ運搬依存がある場合、外側のループを展開して複数の値を並列に計算できる場合があります。次のセクションでは、これらの概念も説明します。

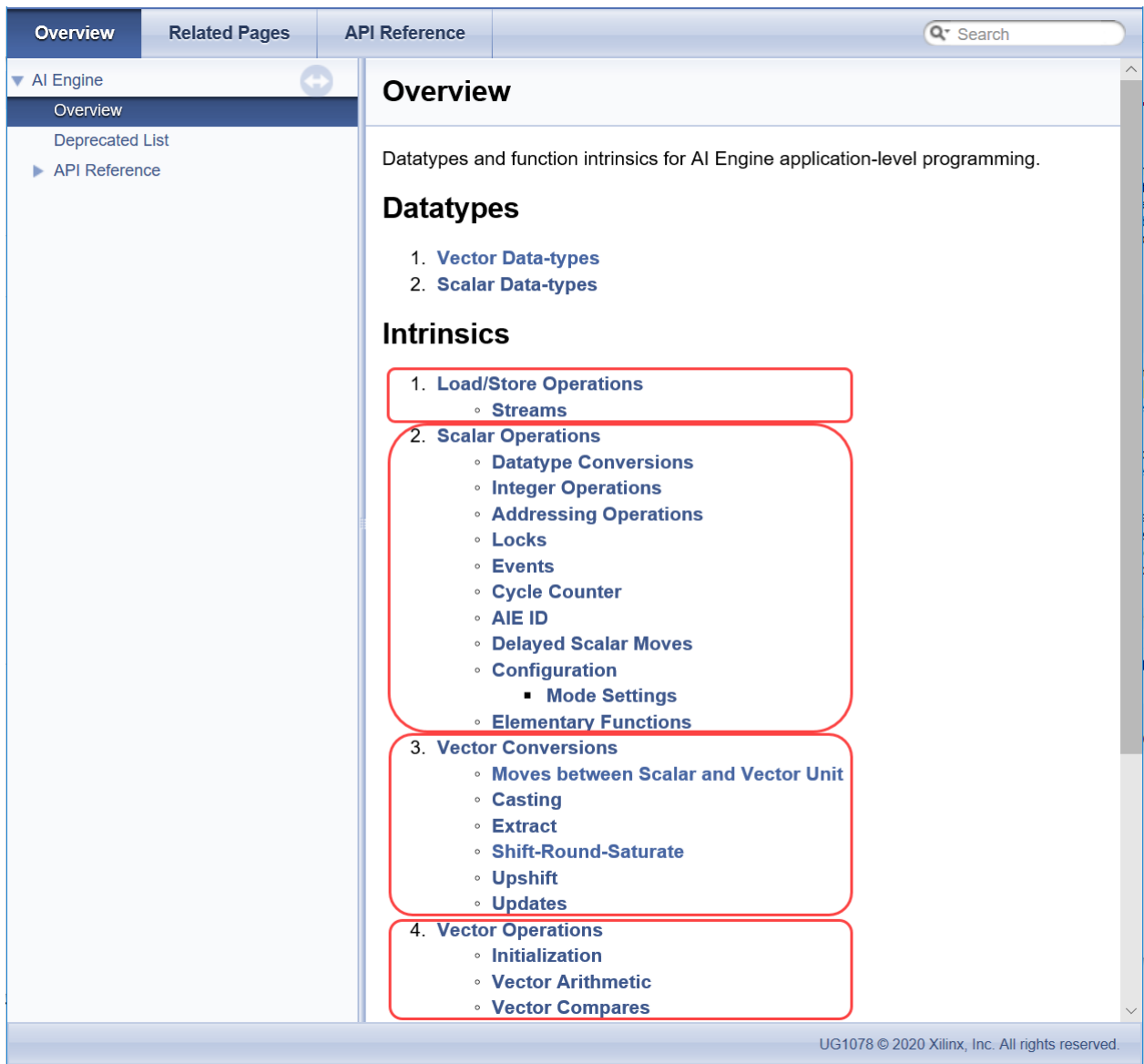
組み込み関数

組み込み関数の詳細は、『Versal ACAP AI エンジン イントリンスクス資料』 ([UG1078](#)) を参照してください。

この組み込み関数の資料は、演算およびデータ型に基づいて構成されています。上位レベルの関数呼び出しによって、次のセクションに分類されています。

- ロード/ストア演算: ポインター逆参照およびポインター演算、およびストリームに対する演算について。
- スカラー演算: 整数および浮動小数点スカラー値、設定、変換、アドレス指定、ロック、およびイベントの演算。
- ベクター変換: 異なるサイズおよび精度のベクターの処理。
- ベクター演算: ベクターに対して実行する算術演算。
- アプリケーション特定の組み込み関数: 特定のアプリケーションのインプリメンテーションに役立つ組み込み関数。

図 6: 組み込み関数の資料



Overview

Datatypes and function intrinsics for AI Engine application-level programming.

Datatypes

1. [Vector Data-types](#)
2. [Scalar Data-types](#)

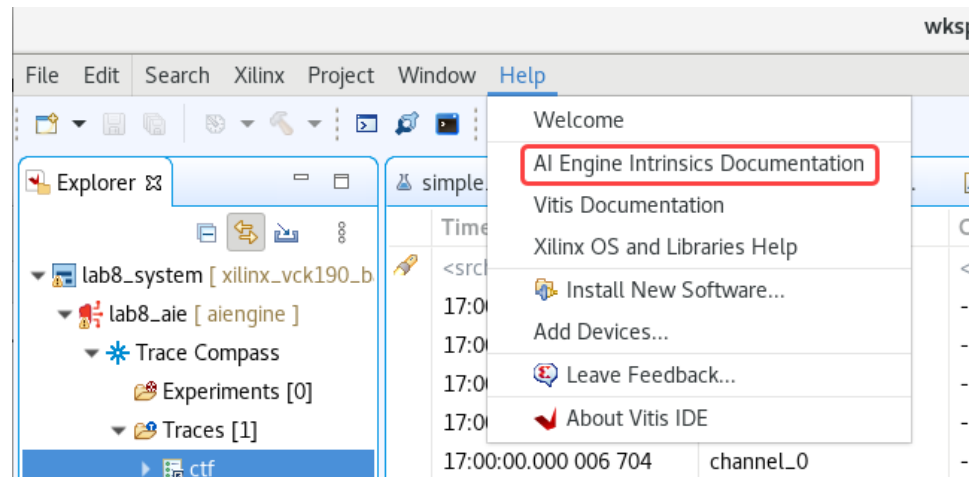
Intrinsics

1. [Load/Store Operations](#)
 - [Streams](#)
2. [Scalar Operations](#)
 - [Datatype Conversions](#)
 - [Integer Operations](#)
 - [Addressing Operations](#)
 - [Locks](#)
 - [Events](#)
 - [Cycle Counter](#)
 - [AIE ID](#)
 - [Delayed Scalar Moves](#)
 - [Configuration](#)
 - [Mode Settings](#)
 - [Elementary Functions](#)
3. [Vector Conversions](#)
 - [Moves between Scalar and Vector Unit](#)
 - [Casting](#)
 - [Extract](#)
 - [Shift-Round-Saturate](#)
 - [Upshift](#)
 - [Updates](#)
4. [Vector Operations](#)
 - [Initialization](#)
 - [Vector Arithmetic](#)
 - [Vector Compares](#)

UG1078 © 2020 Xilinx, Inc. All rights reserved.

この資料には、Vitis™ IDE からアクセスできます。

図 7: VitisIDE から組み込み関数の資料にアクセス



カーネル プラグマ

AI エンジン コンパイラは、効率的なループ スケジューリングのための専用指示子をサポートします。メモリの依存関係を削減し、関数階層を削除して、カーネルのパフォーマンスを最適化する追加のプラグマも含まれています。これらのプラグマの使用例が、この資料に含まれています。

Chess コンパイラ ユーザー マニュアルに、カーネルのコンフィギュレーションに使用されるプラグマおよび関数のリストが含まれています。このマニュアルは、AI エンジン ラウンジにあります。

カーネル コンパイラ

AI エンジン コンパイラは、AI エンジン カーネル コードのコンパイルに使用されます。AI エンジン コンパイラの使用法、使用可能なオプション、渡すことが可能な入力ファイル、および出力については、『Vitis 統合ソフトウェア プラットフォームの資料』(UG1416)の AI エンジン フローの [AI エンジン グラフ アプリケーションのコンパイル](#) を参照してください。

カーネルのシミュレーション

AI エンジン カーネルをシミュレーションするには、C++ で記述されたデータフロー グラフ仕様を含む AI エンジン グラフ アプリケーションを記述する必要があります。このグラフには、AI エンジン カーネルのみが含まれ、カーネルへの入力にはテストベンチのデータが使用されます。カーネルからのデータ出力はシミュレーション出力としてキャプチャし、ゴールデン データと比較できます。この仕様は、AI エンジン コンパイラを使用してコンパイルおよび実行できます。アプリケーションは AI エンジン シミュレータを使用してシミュレーションできます。シミュレータの詳細は、『Vitis 統合ソフトウェア プラットフォームの資料』(UG1416)の AI エンジン フローの [AI エンジン グラフ アプリケーションのシミュレーション](#) を参照してください。

カーネルの入力および出力

AI エンジン カーネルは、データのストリームまたはブロックに対して処理を実行します。AI エンジン カーネルは、int32、cint32 などの特定のデータ型を処理します。カーネルで使用されるデータのブロックは、データのウィンドウと呼ばれます。カーネルは、入力データ ストリームまたはウィンドウを消費し、出力データ ストリームまたはウィンドウを生成します。カーネルは、データ ストリームにサンプルごとにアクセスします。ウィンドウおよびストリーム API の詳細は、『Vitis 統合ソフトウェア プラットフォームの資料』 (UG1416) の AI エンジン フローの [ウィンドウおよびストリーミング データ API](#) を参照してください。

AI エンジン カーネルの RTP ポートは、PS からアップデートまたは読み出すこともできます。RTP の詳細は、『Vitis 統合ソフトウェア プラットフォームの資料』 (UG1416) の AI エンジン フローの [ランタイム時間グラフ制御 API](#) を参照してください。

スカラーおよびベクター プログラミングの概要

このセクションでは、スカラーおよびベクター プロセッシング要素のカーネル プログラミングの重要な要素について、概要を説明します。各要素および最適化手法の詳細を次に示します。

次の例は、スカラー エンジンのみを使用しています。512 個の int32 要素を反復する for ループを示します。各ループ反復は、int32 a と int32 b を乗算を 1 回実行し、結果を c に格納して、出力ウィンドウに書き込みます。scalar_mul カーネルは、データ input_window_int32 の 2 つの入力ブロック (ウィンドウ) に演算を実行し、データ output_window_int32 の出力ウィンドウを生成します。

カーネル外の循環バッファに対する読み出しと書き込みには、API window_readincr および window_writeincr を使用します。ウィンドウ API の詳細は、『Vitis 統合ソフトウェア プラットフォームの資料』 (UG1416) の AI エンジン フローの [ウィンドウおよびストリーミング データ API](#) を参照してください。

```
void scalar_mul(input_window_int32* data1,
               input_window_int32* data2,
               output_window_int32* out){
    for(int i=0;i<512;i++){
        {
            int32 a=window_readincr(data1);
            int32 b=window_readincr(data2);
            int32 c=a*b;
            window_writeincr(out,c);
        }
    }
}
```

次の例は、同じカーネルのベクター バージョンです。

```
void vect_mul(input_window_int32* __restrict data1,
             input_window_int32* __restrict data2,
             output_window_int32* __restrict out){
    for(int i=0;i<64;i++)
        chess_prepare_for_pipelining
    {
        v8int32 va=window_readincr_v8(data1);
        v8int32 vb=window_readincr_v8(data2);
        v8acc80 vt=mul(va,vb);
    }
}
```

```
v8int32 vc=srs(vt,0);

window_writeincr(out,vc);
}
}
```

上記のコードでは、データ型 `v8int32` および `v8acc80` が使用されています。ウィンドウ API `window_readincr_v8` は 8 個の `int32` ベクターを返し、変数 `va` および `vb` に格納します。これらの 2 つの変数はベクター型変数であり、`v8acc80` 型である組み込み関数 `mul` および `vt` に渡されます。`v8acc80` 型は、`v8int32` 型変数 `vc` を返して出力ウィンドウに書き込むシフト、丸め、飽和関数 `srs` によってリダクションされます。AI エンジンでサポートされるデータ型の追加の詳細は、この後のセクションで説明します。

`vect_mul` 関数の入力パラメーターと出力パラメーターに `--restrict` キーワードを使用すると、データの独立性が明示的に示され、コンパイラで最適化をより積極的に実行できます。

`chess_prepare_for_pipelining` は、カーネル コンパイラでループの最適化パイプラインが実現されるよう指示するコンパイラ プラグマです。

この関数例のスカラー バージョンは 1055 サイクルかかりますが、ベクター バージョンは 99 サイクルしかかかりません。カーネルのベクター バージョンを使用すると、10 倍以上高速になります。ベクター プロセッシング自体は、`int32` 乗算の 8 倍のスループットを達成しますが、レイテンシが高く、全体的なスループットは 8 倍になりません。ループ最適化を使用すると、10 倍近くになります。この後のセクションでは、使用可能なさまざまなデータ型、使用可能なレジスタ、ループでのソフトウェアのパイプライン処理の概念をや `--restrict` などのキーワード使用して AI エンジンで実現可能な最適化について詳しく説明します。

関連情報

[ループのソフトウェア パイプライン処理](#)
[restrict キーワード](#)

AI エンジンのデータ型

AI エンジン スカラー ユニットは、8、16、および 32 ビット幅の符号付きおよび符号なし整数と、特定の演算用に単精度浮動小数点をサポートします。

AI エンジン ベクター ユニットは、8、16、および 32 ビット幅の整数および複素整数と、実数および複素単精度浮動小数点値をサポートします。また、48 および 80 ビット幅の要素を持つアキュムレータ ベクター データ型もサポートされます。絶対値、加算、減算、比較、乗算、MAC などの組み込み関数は、これらのベクター データ型を使用して演算を実行します。ベクター データ型の名前は、要素数、実数または複素数、ベクター型またはアキュムレータ型、およびビット幅を含む次の命名規則に基づいています。

```
v{NumLanes}[c]{[u]int|float|acc}{SizeofElement}
```

オプションの仕様は次のとおりです。

- `NumLanes`: ベクターの要素数を示します。有効な値は、2、4、8、16、32、64、および 128 です。
- `c`: 実数部と虚数部を持つ複素データを示します。
- `int`: 整数ベクター データ値を示します。
- `float`: 単精度浮動小数点値を示します。

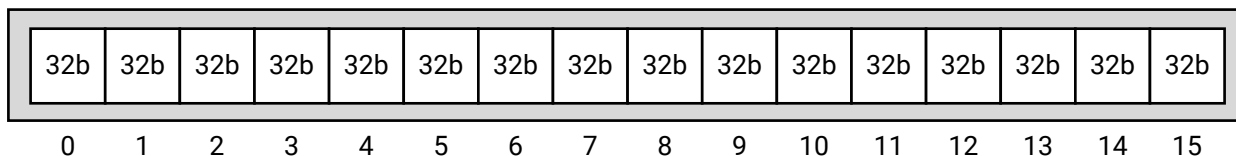
注記: 浮動小数点ベクターには、アキュムレータ レジスタはありません。

- `acc`: アキュムレータ ベクター データ値を示します。
- `u`: 符号なしを示します。符号なしは、`int8` ベクターにのみ存在します。
- `SizeofElement`: ベクター データ型要素のサイズを示します。
 - 1024 ビット整数ベクター型は、8 ビット、16 ビット、または 32 ビット ベクター要素のベクターです。これらのベクターには、16、32、64、または 128 個のレーンがあります。
 - 512 ビット整数ベクター型は、8 ビット、16 ビット、32 ビット、または 64 ビット ベクター要素のベクターです。これらのベクターには、4、8、16、32、または 64 個のレーンがあります。
 - 256 ビット整数ベクター型は、8 ビット、16 ビット、32 ビット、64 ビット、または 128 ビット ベクター要素のベクターです。これらのベクターには、1、2、4、8、または 32 個のレーンがあります。
 - 128 ビット整数ベクター型は、8 ビット、16 ビット、または 32 ビット ベクター要素のベクターです。これらのベクターには、2、4、8、または 16 個のレーンがあります。
 - アキュムレータ データ型は、80 ビットまたは 48 ビット要素のベクターです。これらのベクターには、2、4、8、または 16 個のレーンがあります。

ベクター データ型の合計データ幅は、128 ビット、256 ビット、512 ビット、または 1024 ビットです。アキュムレータ データ型の合計データ幅は、320/384 ビットまたは 640/768 ビットです。

たとえば `v16int32` は、32 ビット整数の 16 要素の整数ベクターです。ベクターの各要素は、レーンと呼ばれます。必要最小限のビット幅を使用すると、レジスタを効率的に使用でき、パフォーマンスを向上できます。

図 8: `v16int32`



X25021-011321

ベクター レジスタ

すべてのベクター 組み込み関数には、AI エンジン ベクター レジスタにオペランドが存在している必要があります。次の表に、ベクター レジスタのセットと、小さなレジスタを組み合わせる大きなレジスタを形成する方法を示します。

表 3: ベクター レジスタ

128 ビット	256 ビット	512 ビット	1024 ビット	
vrl0	wr0	xa	ya	なし
vrh0				
vrl1	wr1			
vrh1				
vrl2	wr2	xb		yd (msbs)
vrh2				
vrl3	wr3			
vrh3				
vcl0	wc0	xc	N/A	N/A
vch0				
vcl1	wc1			
vch1				
vdI0	wd0	xd	N/A	yd (lsbs)
vdh0				
vdI1	wd1			
vdh1				

基本となる最下層のハードウェア レジスタは 128 ビット幅で、接頭辞 v が付いています。v レジスタを 2 つ組み合わせて 256 ビット レジスタを構成できます (接頭辞 w)。wr、wc、および wd レジスタを 2 つ組み合わせて 512 ビット レジスタ (xa、xb、xc、および xd) を構成します。xa と xb で 1024 ビット幅の ya レジスタを構成し、xd と xb で 1024 ビット幅の yd レジスタを構成します。つまり、xb レジスタは ya レジスタと yd レジスタ間で共有されます。xb には、ya レジスタと yd レジスタ両方の最上位ビット (MSB) が含まれます。

ベクター レジスタ名を `chess_storage` 指示子と共に使用すると、ベクター データを特定のベクター レジスタに格納できます。次に例を示します。

```
v8int32 chess_storage(wr0) bufA;
v8int32 chess_storage(WR) bufB;
```

`chess_storage` 指示子に大文字が使用されている場合はレジスタ ファイル (4 つの wr レジスタのいずれかなど) を意味し、小文字が使用されている場合は特定のレジスタ (上記のコード例では wr0) のみが選択されることを意味します。

ベクター レジスタは有益なリソースです。コード生成中にコンパイラで使用可能なベクター レジスタを足りなくなった場合、収まらなかったレジスタの内容はローカル メモリに格納され、必要に応じて読み出されます。これにより、クロック サイクル数が増加します。

カーネルの実行中に使用されるベクター レジスタの名前は、カーネルのマイクロコード内にベクター ロード/ストア およびその他のベクター ベース命令に対して表示されます。このマイクロコードは、Vitis IDE の [Disassembly] ビューに表示されます。Vitis IDE の使用に関する詳細は、[Vitis IDE およびレポートの使用](#) を参照してください。

多くの組み込み関数には特定のベクター データ型のみを使用可能ですが、ベクターからのすべての値が必要とは限りません。たとえば、一部の組み込み関数には 512 ビット ベクターのみを使用可能です。カーネル コードのデータ サイズが小さい場合、`concat()` 組み込み関数を使用して、この小さいサイズのデータを未定義のベクター (データ型が定義されているが初期化されていないベクター) と連結すると役立つことがあります。

たとえば、`lmul8` 組み込み関数には、`xbuff` パラメーターに `v16int32` または `v32int32` ベクターのみを使用可能です。組み込み関数のプロトタイプは次のとおりです。

```
v8acc80 lmul8 (      v16int32      xbuff,
                int      xstart,
                unsigned int  xoffsets,
                v8int32      zbuff,
                int      zstart,
                unsigned int  zoffsets
            )
```

`xbuff` パラメーターには、16 要素ベクター (`v16int32`) を使用する必要があります。次の例では、8 要素ベクター (`v8int32`) `rva` があります。`concat()` 組み込み関数は、16 要素ベクターにアップグレードするのにも使用されます。連結後、16 要素ベクターの下位半分には `rva` の内容が含まれます。未定義の `v8int32` ベクターと連結しているため、16 要素ベクターの上位半分は初期化されません。

```
int32 a[8] = {1, 2, 3, 4, 5, 6, 7, 8};
v8int32 rva = *((v8int32*)a);
acc = lmul8(concat(rva, undef_v8int32()), 0, 0x76543210, rva, 0, 0x76543210);
```

ベクター ベースの組み込み関数の詳細は、[ベクター レジスタ レーンの並べ替え](#) を参照してください。

アキュムレータ レジスタ

アキュムレータ レジスタは 384 ビット幅で、48 ビットのベクター レーンが 8 個含まれます。これにより、32 ビットの乗算結果をビット オーバーフローなしに累算できるようになります。16 ビットのガード ビットにより、最大 2^{16} 回の累算が可能です。固定小数点ベクター MAC および MUL 組み込み関数の出力は、アキュムレータ レジスタに格納されます。次の表に、アキュムレータ レジスタのセットと、小さなレジスタを組み合わせる大きなレジスタを形成する方法を示します。

表 4: アキュムレータ レジスタ

384 ビット	768 ビット
aml0	bm0
amh0	
aml1	bm1
amh1	
aml2	bm2
amh2	
aml3	bm3
amh3	

アキュムレータ レジスタには、名前の先頭に `am` が付きます。これらのレジスタを 2 つ組み合わせて、名前の先頭に `bm` が付いた 768 ビット レジスタが形成されます。

シフト-丸め-飽和を実行する `srs()` 組み込み関数を使用すると、アキュムレータ レジスタから必要なシフトおよび丸め機能を備えたベクター レジスタに値を移動できます。

```
v8int32 res = srs(acc, 8); // shift right 8 bits, from accumulator register
                             to vector register
```

アップシフトを実行する `ups()` 組み込み関数を使用すると、ベクター レジスタからアキュムレータ レジスタに値を移動できます。

```
v8acc48 acc = ups(v, 8); //shift left 8 bits, from vector register to
                             accumulator register
```

`set_rnd()` および `set_sat()` 組み込み関数を使用すると、累積結果の丸めおよび飽和モードを設定できます。また、`clr_rnd()` および `clr_sat()` 組み込み関数を使用すると、丸めおよび飽和モードをクリアにして、累算結果を切り捨てます。

演算がシフト-丸め-飽和のデータパスを通過する場合にのみ、シフト、丸め、または飽和モードが有効になります。組み込み関数の中にはベクターの前置加算器演算のみが使用され、シフト、丸め、または飽和モードがコンフィギュレーションに使用されないものもあります。このような演算には、加算、減算、絶対値、ベクター比較、またはベクター選択/シャッフルがあります。シフト、丸め、または飽和モードのコンフィギュレーションを使用した減算の代わりに、MAC 組み込み関数を選択できます。次のコードは、`sub` 組み込み関数の代わりに、`mul` を使用して `va` と `vb` 間で減算を実行します。

```
v16cint16 va, vb;
int32 zbuff[8]={1,1,1,1,1,1,1,1};
v8int32 coeff=*(v8int32*)zbuff;
v8acc48 acc = mul8_antisym(va, 0, 0x76543210, vb, 0, false, coeff, 0 ,
0x76543210);
v8int32 res = srs(acc,0);
```

浮動小数点組み込み関数には、個別のアキュムレータ レジスタはないので、結果はベクター レジスタに返されます。

データ型変換

データ型変換組み込み関数 (`as_[Type]()`) を使用すると、同じサイズのベクター型またはスカラー型間の変換ができます。型変換は、アキュムレータ ベクター型でも実行できます。通常は、できるだけ最小のデータ型を使用することで、レジスタの流出を減らし、パフォーマンスを向上させることができます。たとえば、48 ビット アキュムレータ (`acc48`) がデザイン要件を満たす場合、より大きな 80 ビット アキュムレータ (`acc80`) よりもそちらを使用することをお勧めします。

注記: `acc80` ベクター データ型は、隣接する 2 つの 48 ビット レーンを占有します。

標準 C 型変換も使用でき、次の例に示すように、ほとんどすべての場合で同じように動作します。

```
v8int16 iv;
v4cint16 cv=as_v4cint16(iv);
v4cint16 cv2=*(v4cint16*)&iv;
v8acc80 cas_iv;
v8cacc48 cas_cv=as_v8cacc48(cas_iv);
```


浮動小数点から固定小数点 (`float2fix()`) および固定小数点から浮動小数点 (`fix2float()`) への変換用のハードウェア サポートが組み込まれています。たとえば、固定小数点平方根、逆平方根、および反転は、浮動小数点精度でインプリメントされ、`fix2float()` および `float2fix()` 変換は関数の前後で使用されます。この例では、平方根と反転関数はベクター化できないので、スカラー エンジンが使用されています。これは、関数プロトタイプの入力データ型からわかります。

```
float _sqrtf(float a) //scalar float operation
int sqrt(int a,...) //scalar integer operation
```

入力データ型はスカラー型 (`int`) で、ベクター型 (`vint`) ではなくになっています。

変換関数 (`fix2float`、`float2fix`) は、呼び出された関数によって、ベクター エンジンまたはスカラー エンジンのいずれかで処理できます。データ戻り値の型とデータ引数の型の違いに注意してください。

```
float fix2float(int n,...) //Uses the scalar engine
v8float fix2float(v8int32 ivec,...) //Uses the vector engine
```

注記: `float2fix` の場合、`float2fix_safe` (デフォルト) と `float2fix_fast` という 2 つのインプリメンテーションタイプがあり、`float2fix_safe` インプリメンテーションには、より厳密なデータ型チェックが含まれます。`FLOAT2FIX_FAST` マクロを定義すると、`float2fix` で `float2fix_fast` インプリメンテーションが選択されるようになります。

ベクターの初期化、読み込み、および保存

ベクター レジスタは、さまざまな方法で初期化、読み込み、および保存できます。最適なパフォーマンスにするには、ベクター レジスタの読み込みまたは保存に使用されるローカル メモリを 16 バイト境界に揃えることが重要です。

アライメント

`alignas` 規格の C 指定子を使用すると、ローカル メモリの適切なアライメントが正しくなります。次の例では、`reals` が 16 バイト境界にアライメントされています。

```
alignas(16) const int32 reals[8] =
    {32767, 23170, 0, -23170, -32768, -23170, 0, 23170};
//align to 16 bytes boundary, equivalent to "alignas(v4int32)"
```

初期化

次の関数を使用すると、ベクター レジスタを未定義、すべてを 0、ローカル メモリからのデータで初期化するか、または値の一部を別のレジスタから初期化し、残りの部分を未定義にすることができます。`undef_type()` 初期化子を使用して初期化すると、値の未定義の部分にかかわらず、コンパイラで最適化されるようになります。

```
v8int32 v;
v8int32 uv = undef_v8int32(); //undefined
v8int32 nv = null_v8int32(); //all 0's
v8int32 iv = *(v8int32 *) reals; //re-interpret "reals" as "v8int32"
pointer and load value from it
v16int32 sv = xset_w(0, iv); //create a new 512-bit vector with lower 256-bit set with "iv"
```


上記の例では、ベクター セット組み込み関数 `[T]set_[R]` により、1 部分のみを初期化し、ほかの部分は未定義のままにするベクターを作成できます。`[T]` は設定するターゲット ベクター レジスタを示しており、`W` レジスタ (256 ビット) の場合は `w`、`X` レジスタ (512 ビット) の場合は `x`、`Y` レジスタ (1024 ビット) の場合は `y` を指定します。`[R]` はソース値の供給元を示しており、`V` レジスタ (128 ビット) の場合は `v`、`W` レジスタ (256 ビット) の場合は `w`、`X` レジスタ (512 ビット) の場合は `x` を指定します。`[R]` の幅は `[T]` の幅よりも狭くなっています。有効なベクター セット組み込み関数は、`wset_v`、`xset_v`、`xset_w`、`yset_v`、`yset_w`、および `yset_x` です。

ロードおよびストア

ベクター レジスタからのロードおよびストア

コンパイラでは、ベクターの標準的なポインター逆参照およびポインター演算がサポートされています。ポインターのポスト インクリメントは、スケジューリングに最も効率的です。ベクター レジスタをロードするのに、特別な組み込み関数は必要ありません。

```
v8int32 * ptr_coeff_buffer = (v8int32 *)ptr_kernel_coeff;
v8int32 kernel_vec0 = *ptr_coeff_buffer++; // 1st 8 values (0 .. 7)
v8int32 kernel_vec1 = *ptr_coeff_buffer;    // 2nd 8 values (8 .. 15)
```

メモリからのロードおよびストア

AI エンジン API を使用すると、AI エンジン カーネルで使用可能なデータ メモリ、ストリーミング データ ポート、およびカスケード ストリーミング ポートからデータを読み書きできます。ウィンドウおよびストリーム API の詳細は、『Vitis 統合ソフトウェア プラットフォームの資料』 (UG1416) の AI エンジン フローの[ウィンドウおよびストリーミング データ API](#) を参照してください。次の例では、ウィンドウ `readincr (window_readincr_v8(din))` API を使用して、複素 `int16` データをデータ ベクターに読み出します。同様に、`cin` ストリームから `int16` データのサンプルを読み出すには、`readincr_v8(cin)` を使用します。カスケード ストリーム出力にデータを書き込むには、`writeincr_v4 (cas_out, v)` を使用します。

```
void func(input_window_cint16 *din,
          input_stream_int16 *cin,
          output_stream_cacc48 *cas_out){
    v8cint16 data=window_readincr_v8(din);
    v8int16 coef=readincr_v8(cin);
    v4cacc48 v;
    ...
    writeincr_v4(cas_out, v);
}
```

ポインターを使用したロードおよびストア

カーネル関数プロトタイプでは、入力および出力としてウィンドウ API を使用する必要があります。カーネル コードでは、データの読み出し/書き込みに直接ポインター参照を使用できます。

```
void func(input_window_int16 *w_input,
          output_window_cint16 *w_output){
    .....
    v16int16 *ptr_in = (v16int16 *)w_input->ptr;
    v8cint16 *ptr_out = (v8cint16 *)w_output->ptr;
    .....
}
```

ウィンドウ構造によりバッファ ロック トラッキング バッファ タイプ (ping/pong) が管理され、これがサイクル数が増加する可能性があります。これは特に、ロード/ストアが順不同 (スキッター ギャザー) の場合に言えます。ポインターを使用すると、ロードとストアに必要なサイクル数を削減できる可能性があります。

注記: ポインターを使用してデータをロードおよびストアする場合は、範囲外のメモリ アクセスを回避するのは、設計者の責任です。

ストリームを使用したロードおよびストア

次の例に示すように、ベクター データは、ストリームからロードまたはストリームにストアできます。

```
void func(input_stream_int32 *s0, input_stream_int32 *s1, ...){
    for(...){
        data0=readincr(s0);
        data1=readincr(s1);
        ...
    }
}
```

ウィンドウおよびストリーミング データ API の使用法の詳細は、『Vitis 統合ソフトウェア プラットフォームの資料』(UG1416) の AI エンジン フローの [ウィンドウおよびストリーミング データ API](#) を参照してください。

アップデート、抽出、およびシフト

ベクター レジスタの一部をアップデートするため、`upd_v()` (128 ビット)、`upd_w()` (256 ビット)、および `upd_x()` (512 ビット) 組み込み関数が提供されています。

注記: アップデートを使用すると、ベクターのその他の部分はアライブ状態に保ちながら、ベクターの一部が新しいデータでアップデートされます。ベクターは、複数のアップデートを実行してもアライブ状態に保持されます。不必要に多くのベクターがアライブ状態に保持されると、レジスタの漏出が発生し、パフォーマンスに影響することがあります。

ベクターの一部を抽出するためには、`ext_v()`、`ext_w()`、および `ext_x()` および組み込み関数が提供されています。

個々の要素をアップデートまたは抽出するには、`upd_elem()` および `ext_elem()` 組み込み関数を使用します。これらの組み込み関数は、ロードまたはストアする値が非連続のメモリ位置にない場合に使用します。1 つのベクターのロードまたはストアには、複数サイクルかかります。次の例は、ベクター `v1` の 0 番目の要素を `a` の値である 100 でアップデートします。

```
int a = 100;
v4int32 v1 = upd_elem(undef_v4int32(), 0, a);
```

もう 1 つの重要な使用法は、データをスカラー ユニットに移動し、反転または `sqrt` を実行することです。次の例は、ベクター `vf` の 0 番目の要素を抽出し、スカラー変数 `f` に格納します。

```
v4float vf;
float f=ext_elem(vf,0);
float i_f=invsqrt(f);
```

`shft_elem()` 組み込み関数を使用すると、ベクターの冒頭に新しい要素を挿入し、ほかの要素を 1 ずつシフトできます。

ベクター レジスタ レーンの並べ替え

AI エンジンの固定小数点ベクター ユニットのデータパスは、次の 3 つの個別に使用可能なパスで構成されます。

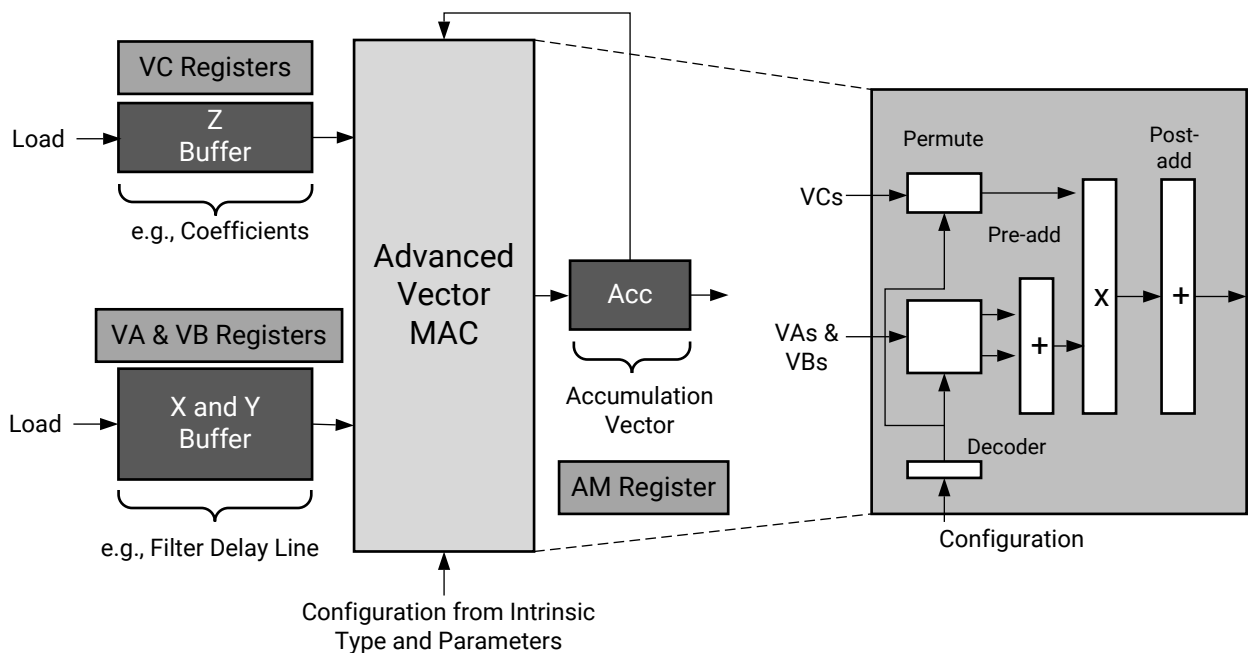
- メイン MAC データパス
- シフト-丸め-飽和パス
- アップシフト パス

メインの乗算パスはベクター レジスタから値を読み出し、ユーザー制御可能な方法でこれらを並べ替え、前置加算 (オプション) の後に乗算を実行し、後置加算した値をアキュムレータ レジスタの直前の値に加算します。

メイン データパスはアキュムレータに格納されますが、シフト-丸め-飽和パスはアキュムレータ レジスタから読み出され、ベクター レジスタまたはデータ メモリに格納されます。メイン データパスと並行して、アップシフト パスが実行されます。乗算は実行されませんが、単にベクターを読み出し、シフトアップして、結果をアキュムレータに送ります。固定点および浮動点データパスの詳細は、『Versal ACAP AI エンジン アーキテクチャ マニュアル』 (AM009: [英語版](#)、[日本語版](#)) を参照してください。これらのデータパスの実行に使用できる組み込み関数の詳細は、『Versal ACAP AI エンジン イントリンシクス資料』 ([UG1078](#)) を参照してください。

次の図に示すように、MAC データパスの基本的な機能には、X バッファと Z バッファからのデータ間のベクター積和演算があります。その他のパラメーターとオプションを使用すると、ベクター内での柔軟なデータ選択ができ、出力レーン数を指定できるほか、オプションの機能により、さまざまな入力データ サイズを使用したり、前置加算ができるようになります。入力バッファには、Y バッファというバッファもあります。このバッファの値は、乗算前に X バッファの値と一緒に前置加算できます。組み込み関数の結果は、アキュムレータに追加されます。

図 9: MAC データパスの機能の概要



X25023-011521

この演算は、「レーン」および「列」を使用して説明できます。レーンの数は、組み込み関数の呼び出しから生成される出力値の数に該当します。列数は、出力レーンごとに実行される乗算の数であり、それぞれの乗算結果は加算されます。次に例を示します。

```
acc0 += z00*(x00+y00) + z01*(x01+y01) + z02*(x02+y02) + z03*(x03+y03)
acc1 += z10*(x10+y10) + z11*(x11+y11) + z12*(x12+y12) + z13*(x13+y13)
acc2 += z20*(x20+y20) + z21*(x21+y21) + z22*(x22+y22) + z23*(x23+y23)
acc3 += z30*(x30+y30) + z31*(x31+y31) + z32*(x32+y32) + z33*(x33+y33)
```

この場合、4 つの出力が生成されるため、各出力 (X および Y バッファからの前置加算も含む) に 4 つのレーンと 4 つの列があります。

組み込み関数のパラメーターを使用すると、各レーンおよび列の異なる入力バッファから柔軟にデータを選択できるようになり、すべて同じパターンのパラメーターでもそのようになります。次のセクションでは、`shuffle` および `select` 組み込み関数を含む詳細な例を使用したデータ選択 (またはデータ並べ替え) 方法について説明します。`mac` 組み込み関数とそのバリエーションの詳細は、次のセクションで説明します。

データ選択

AI エンジン組み込み関数では、さまざまなタイプのデータ選択がサポートされます。次に、シャフル組み込み関数と選択組み込み関数の詳細を説明します。

データ シャッフル

AI エンジン シャッフル組み込み関数は、開始パラメーターとオフセット パラメーターに従って、1 つの入力データ バッファからデータを選択します。これにより、値を配置し直すことなく、入力ベクター値を柔軟に並べ替えることができます。`xbuff` は入力データ バッファで、`xstart` は `xbuff` データ バッファ内の各レーンの開始位置のオフセットを示し、`xoffset` はデータ バッファに適用される位置のオフセットを示します。シャッフル組み込み関数は、8、16、および 32 レーンのバリエーション (`shuffle8`、`shuffle16`、および `shuffle32`) で使用できます。データ (`xoffsets`) の主な並べ替えは 32 ビット粒度で、`xsquare` は主な並べ替え後にさらに 16 ビット粒度の並べ替えを可能にします。このため、8 ビットおよび 16 ビットのベクター組み込み関数には、より複雑な並べ替え用に `square` パラメーターをさらに設定できます。

たとえば、`shuffle16` 組み込み関数には次の関数プロトタイプがあります。

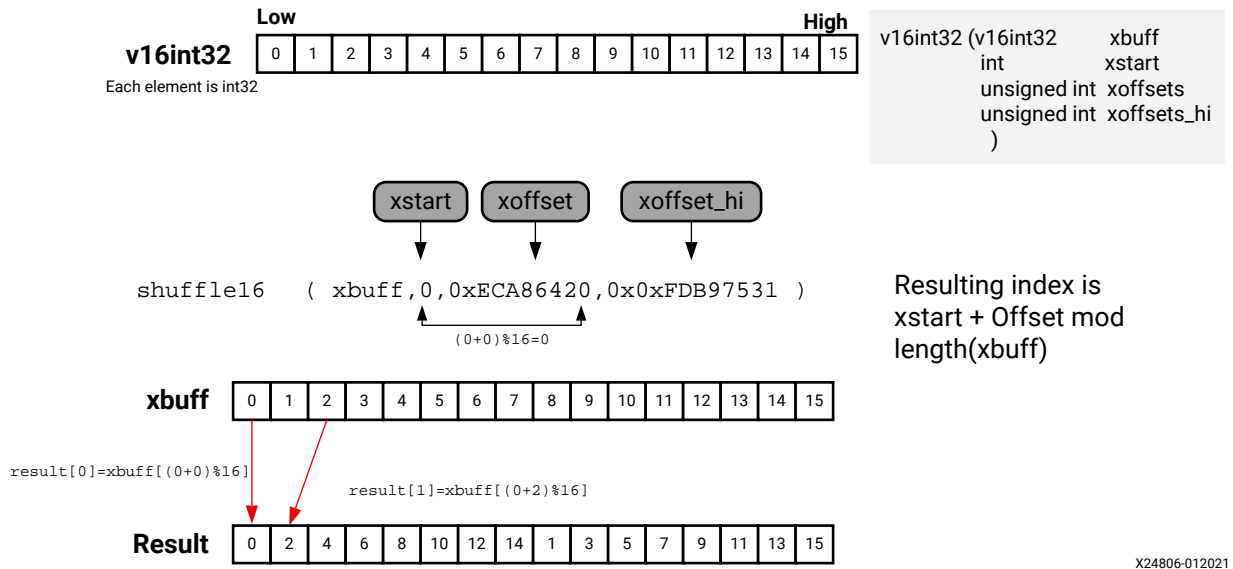
```
v16int32 shuffle16 ( v16int32 xbuff,
    int xstart,
    unsigned int xoffsets,
    unsigned int xoffsets_hi
)
```

データ並べ替えは 32 ビット粒度で実行されます。データ サイズが 32 ビットまたは 64 ビットの場合、開始およびオフセットはフル データ幅 (32 ビットまたは 64 ビット) を基準にします。レーン選択は、通常のレーン選択方式に従って実行されます。

```
f: result [lane number] = (xstart + xbuff [lane number]) Mod input_samples
```

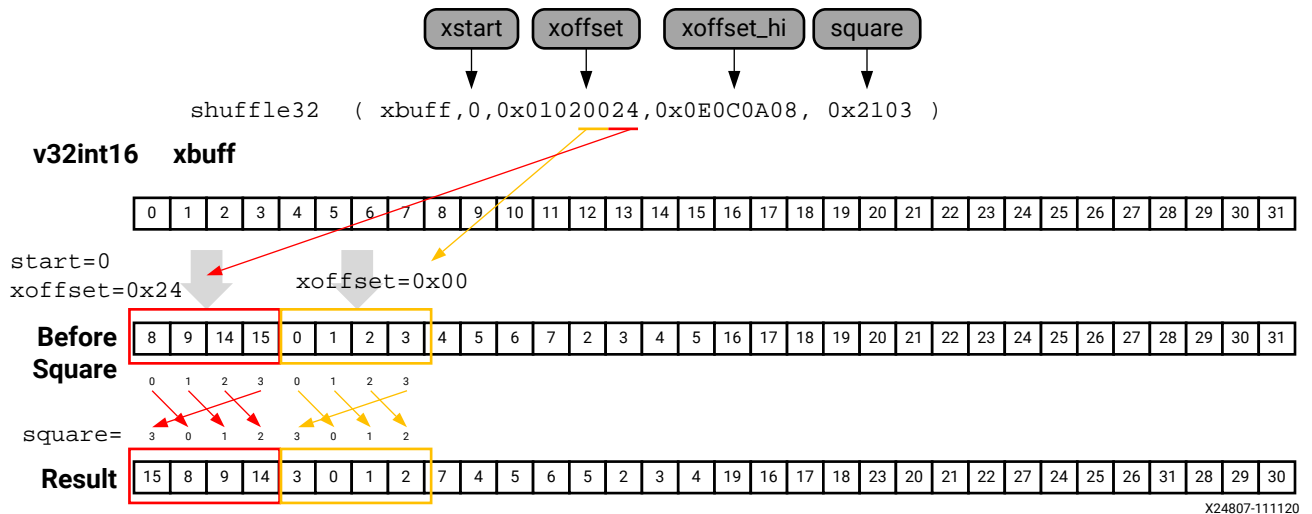
次に、`v16int32` ベクターのシャッフルの例を示します。`xoffset` および `xoffset_hi` の各レーンは 4 ビットです。この例では、バッファの偶数要素と奇数要素をバッファの下位部分と上位部分に移動します。

図 10: int32 型のデータ シャッフル



データ並べ替えが 16 ビット データで実行される場合、組み込み関数には別の `xsquare` パラメーターが含まれるので、データの 4 x 16 ビット ブロックごとに柔軟にデータ選択ができるようになります。`xoffset` は対になります。最初の 16 進数値は絶対 32 ビット オフセットで、2 x 16 ビット 値 (インデックス、インデックス + 1) を取得します。2 つ目の 16 進値は、最初の値 + 1 (32 ビット オフセット) からオフセットされ、2 x 16 ビット 値を取得します。たとえば、`0x00` はインデックス 0、1 とインデックス 2、3 を選択します。`0x24` はインデックス 8、9 とインデックス 14、15 を選択します。次に、`v32int16` ベクターのシャッフルの例を示します。

図 11: int16 型のデータ シャッフル



データ選択

select 組み込み関数は、select パラメーターの値に応じて、レーンの最初のセットまたは 2 番目のセットを選択します。select のレーンに対応するビットが 0 の場合、レーンの最初のセットの値が返されます。対応するビットが 1 の場合、レーンの 2 番目のセットの値が返されます。たとえば、select16 組み込み関数には次の関数プロトタイプがあります。

```
v16int32 select16 (    unsigned int    select,
    v16int32    xbuff,
    int    xstart,
    unsigned int    xoffsets,
    unsigned int    xoffsets_hi,
    v16int32    ybuff,
    int    ystart,
    unsigned int    yoffsets,
    unsigned int    yoffsets_hi
)
```

select の各ビット (下位から上位) に対して、レーンが xbuff から (select パラメーター ビットが 0 の場合) または ybuff から (select パラメーター ビットが 1 の場合) から選択されます。選択された xbuff または ybuff レーン上のデータ並べ替えは、shuffle で xoffsets または yoffsets の対応するビットを使用して達成されます。次に、select の疑似 C スタイルのコードを示します。

```
for (int i = 0; i < 16; i++){
    idx = f( xstart, xoffsets[i]); //i'th 4 bits of offsets
    idy = f( ystart, yoffsets[i]);
    o[i] = select[i] ? y[idy]:x[idx];
}
```

上記のコードで f がどのように機能するかに関する情報は、このセクションの最初に示されている通常のレーン選択スキームの式を参照してください。

int16 型を使用する場合、select 組み込み関数に追加の xsquare パラメーターを使用すると、メインの並べ替えの後にさらに 16 ビット粒度の並べ替えを実行できます。たとえば、select32 組み込み関数には次の関数プロトタイプがあります。

```
v32int16 select32 (    unsigned int    select,
    v64int16    xbuff,
    int    xstart,
    unsigned int    xoffsets,
    unsigned int    xoffsets_hi,
    unsigned int    xsquare,
    int    ystart,
    unsigned int    yoffsets,
    unsigned int    yoffsets_hi,
    unsigned int    ysquare
)
```

次に、select の疑似 C スタイルのコードを示します。

```
for (int i = 0; i < 32; i++){
    idx = f( xstart, xoffsets[i], xsquare);
    idy = f( ystart, yoffsets[i], ysquare);
    o[i] = select[i] ? y[idy]:x[idx];
}
```

次の例では、select32 を使用して、A と B の最初の 16 要素をインターリーブしています (A が先)。

```
int16 A[32]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
             16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31
};
int16 B[32]={32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,
             48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63
};
v32int16 *pA=(v32int16*)A;
v32int16 *pB=(v32int16*)B;
v32int16 C = select32(0xAAAAAAAA, concat(*pA,*pB),
                     0, 0x03020100, 0x07060504, 0x1100,
                     32, 0x03020100, 0x07060504, 0x1100);
```

上記のコードの出力 C は、次のとおりです。

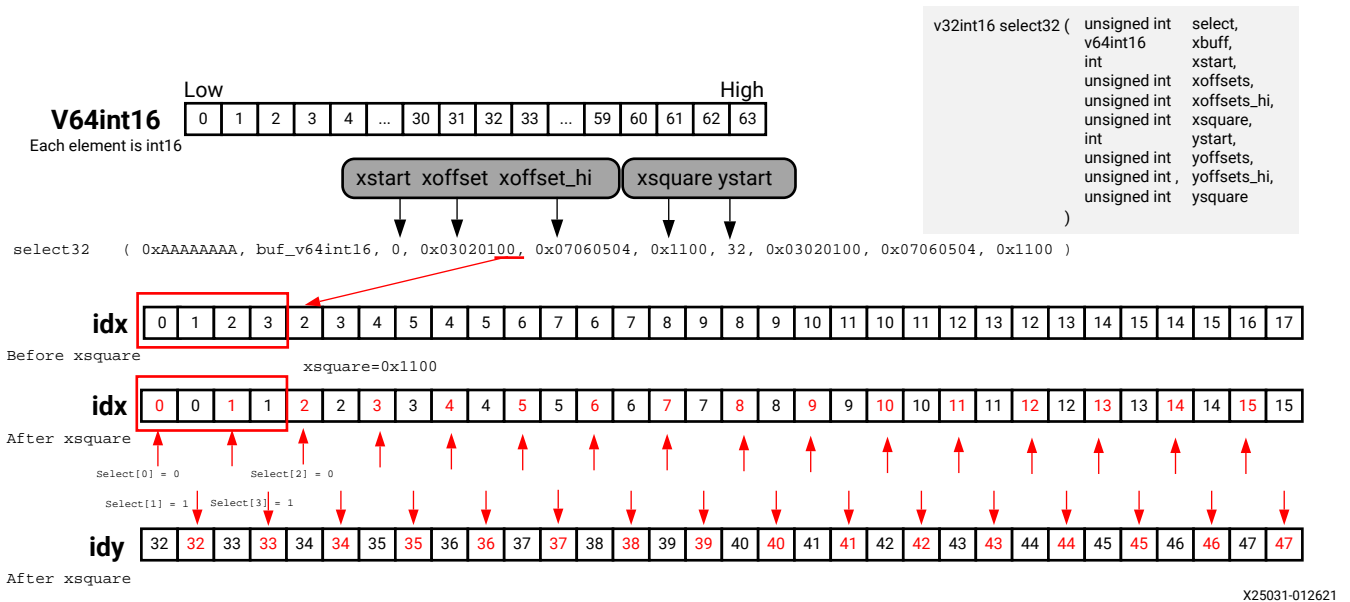
```
{0,32,1,33,2,34,3,35,4,36,5,37,6,38,7,39,8,40,9,41,10,42,11,43,12,44,13,45,14,46,15,47
}
```

これは、shuffle32 組み込み関数を使用しても達成できます。

```
v32int16 C = shuffle32(concat(*pA,*pB),
                      0, 0xF3F2F1F0, 0xF7F6F5F4, 0x3120);
```

次の図に、上記の select32 組み込み関数がどのように機能するかを示します。

図 12: int16 型のデータ選択



MAC 組み込み関数

MAC 組み込み関数は、2 つのバッファ (X および Z バッファ) からデータ間でベクトル積和演算を実行します。その他のパラメータおよびオプションを指定すると、柔軟性が増し (ベクトル内のデータ選択、出力レーン数)、オプションの機能 (異なる入力データサイズ、前置加算など) も追加できます。入力バッファには、Y バッファというバッファもあります。このバッファの値は、乗算前に X バッファの値と一緒に前置加算できます。組み込み関数の結果は、アキュムレータに追加されます。

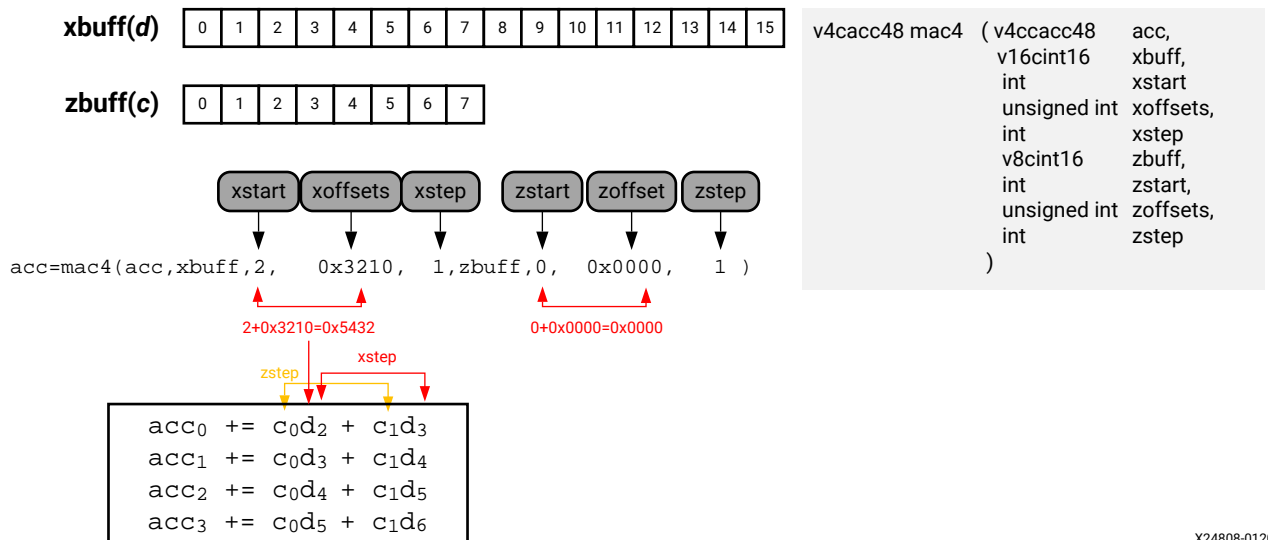
組み込み関数のパラメータを使用すると、各レーンおよび列の異なる入力バッファから柔軟にデータを選択できるようになり、すべて同じパターンのパラメータでもそのようになります。バッファ内の開始点は、最初の行と最初の列に最初の要素を選択する (x/y/z) start パラメータで指定します。各レーンに柔軟性を持たせるには、(x/y/z) オフセットで開始点に追加される各レーンのオフセット値を指定します。最後に、(x/y/z) step パラメータで、前の位置に基づいて各列間のデータ選択のステップを定義します。組み込み関数で ystep が指定されない場合は、xstep の対称になることに注意してください。

x/y バッファおよび z バッファの主な並べ替え粒度は、それぞれ 32 ビットと 16 ビットです。複素数は、並べ替えでは 1 つのエンティティとみなされます (たとえば、cint16 は並べ替えでは 32 ビットとみなされます)。zstart パラメータは、コンパイル時間定数にする必要があります。x/y の 8 ビットおよび 16 ビットの並べ替え粒度と z の 8 ビットの粒度には、このセクションの最後に示すように特定の制限があります。次のセクションでは、さまざまなデータ幅について説明し、これらのデータ幅での MAC 組み込み関数の結果について説明します。

32x32 ビットに対する MAC

次の図に、start、offsets、および step で cint16 型がどのように処理されるかを示します。

図 13: cint16 x cint16 型に対する MAC4



X24808-012021

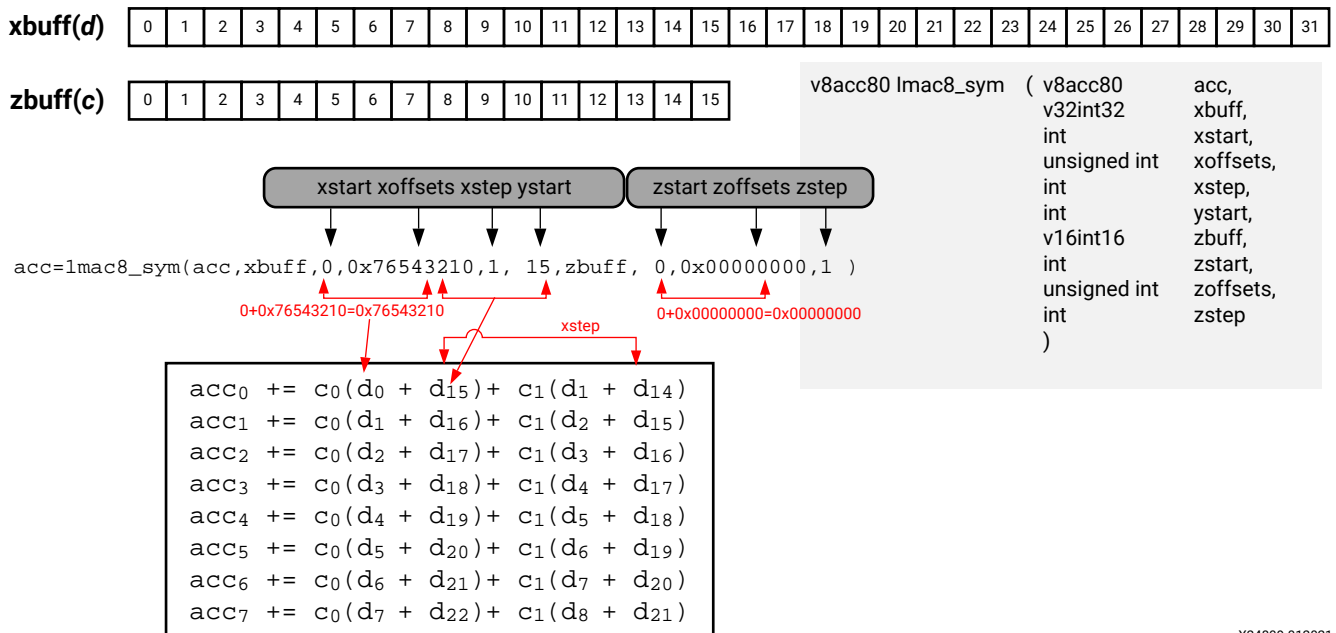
mac4 には、4 つの出力レーンがあります。データの最初の列は、xoffsets の 4 ビットごとに xstart を追加して選択します。データの次の列は、前の列に xstep を追加して選択します。表 2 から、cint16 x cint16 演算には 1 サイクルごとに 8 つの MAC があることがわかります。つまり、mac4 には 2 列の乗算があるということです。

mac4 の係数も同様に、zstart、zoffset、および zstep で選択されます。

32x16 ビットに対する MAC

次に、前置加算を含む MAC の例を示します。前置加算があると、X バッファからのデータをそれだけ追加するか、X バッファからのデータと Y バッファからのデータを追加できます。start、offsets、および step パラメータは、前の例と同様に機能します。Y バッファの ystart パラメータまたは X バッファからの別のデータがあります。step パラメータは、Y または X からの別のデータで逆に機能します。

図 14: int32 x int16 型に対する LMAC8_SYM



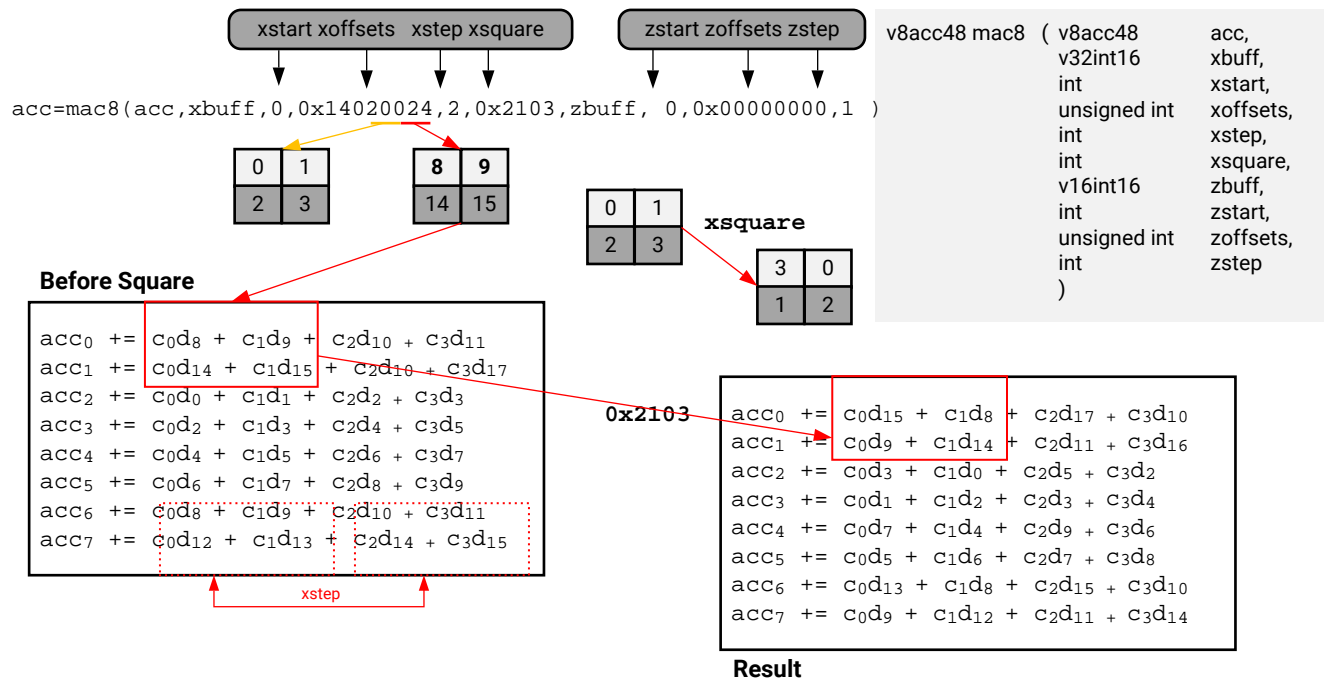
X24809-012021

16x16 ビットの MAC

次は、int16 の X バッファおよび int16 の Z バッファを使用した MAC の例です。X バッファの並べ替え粒度は 32 ビットです。start および step パラメータは、常にデータ型の粒度になります。このため、16 ビットのデータの場合、値 2 は 2 * 16 ビット先を選択します。xoffsets パラメータは対になります。最初の 16 進数値は絶対 32 ビット オフセットで、偶数行で 2 x 16 ビット値 (インデックス、インデックス +1) を取得します。2 番目の 16 進数値は、最初の値 +1 (32 ビット オフセット) からオフセットされ、奇数行で 2 x 16 ビット値を取得します。そのため、xoffsets の 16 進値 0x24 は偶数行にはインデックス 8、9 を、xbuff からの奇数行にはインデックス 14、15 を選択し、xoffsets の 16 進数値 0x00 は偶数行にはインデックス 0、1 を、xbuff からの奇数行にはインデックス 2、3 を選択します。

ほかに、xsquare というメインの並べ替え後に 16 ビット粒度の回転をするパラメータもあります。たとえば、xsquare の値 0x2103 (小さい 16 進値から大きい 16 進値の順) は、偶数行にインデックス 3、0、奇数行にインデックス 1、2 を入力します。xsquare パラメータの役割は、次の図の中央に表示されています。

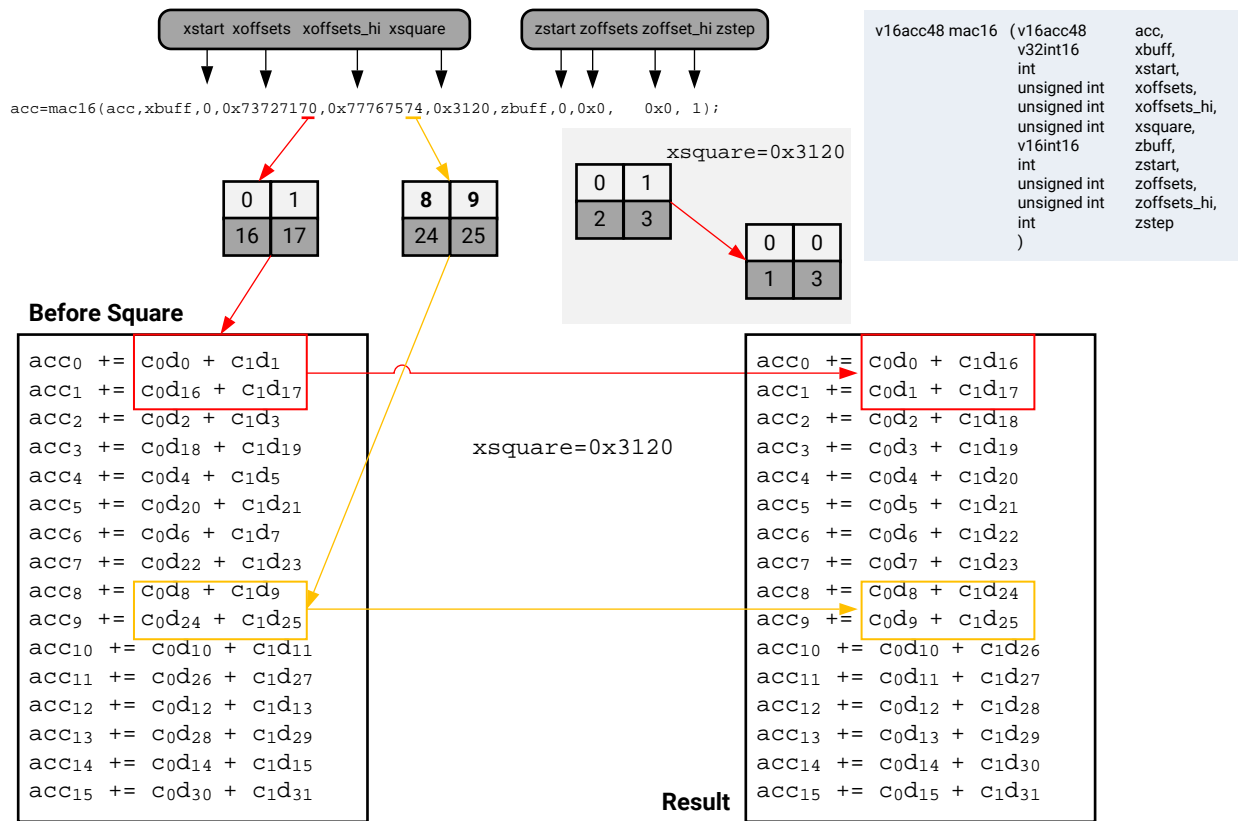
図 15: Iint16 x int16 型の MAC8



XZ4810-012021

次の図に、int16 x int16 の `mac16` 組み込み関数の例を示します。これは、[シングル カーネルのコーディング例](#) の行列ベクトル乗算および行列乗算のデザイン例で使用されています。

図 16: Iint16 x int16 型の MAC16



X25070-012821

8x8 ビットの MAC

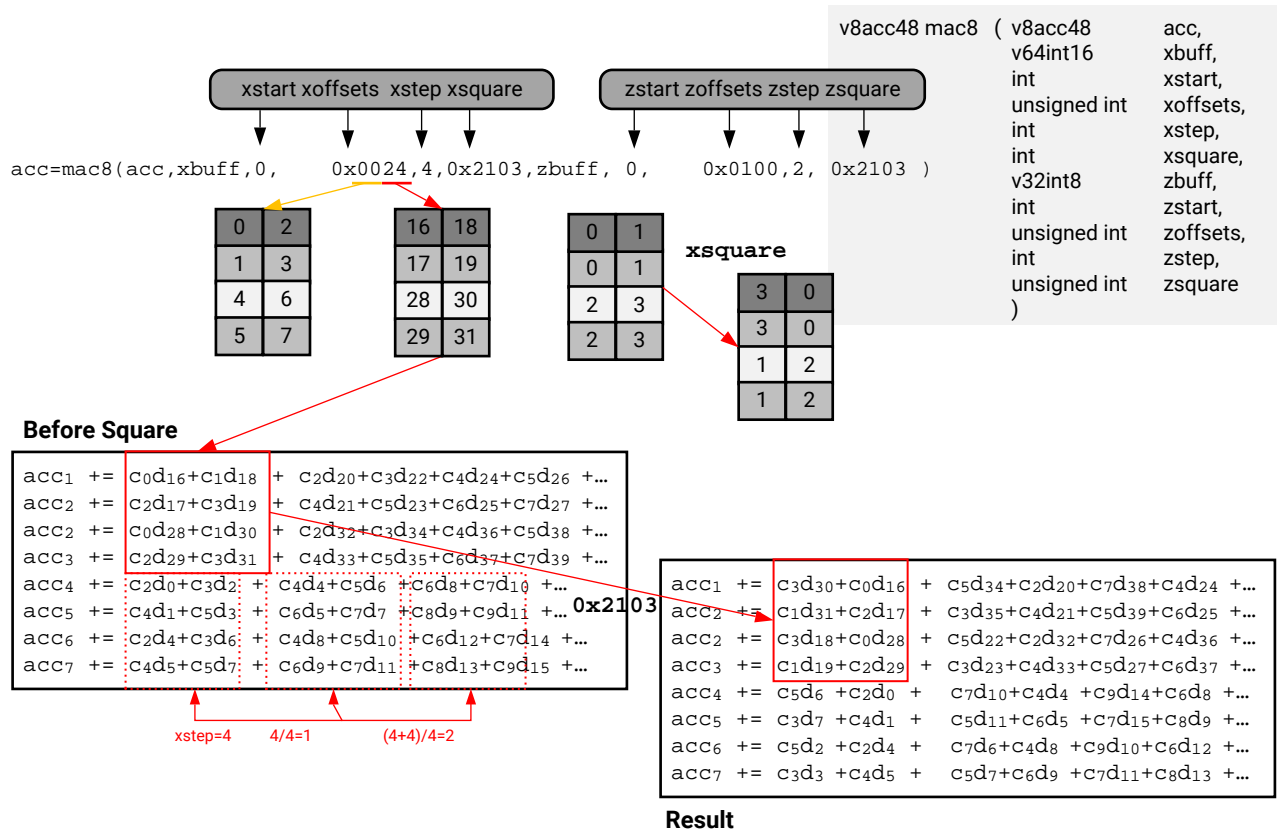
次の図に、`int8` の `X` バッファーと `int8` の `Z` バッファーを使用する MAC を示します。1 つ目の図はデータがどのように並べ替えられるかを示し、2 つ目の図は係数がどのように並べ替えられるかを示します。`X` バッファーと `Z` バッファーの並べ替え粒度は、それぞれ 32 ビットと 16 ビットです。`xoffsets` パラメーターは対になります。最初の 16 進数値は絶対 32 ビット オフセットで、4 x 8 ビット値 (インデックス、インデックス + 1、インデックス + 2、インデックス + 3) を取得します。2 つ目の 16 進数値は、最初の値 + 1 (32 ビット オフセット) からオフセットされ、4 x 8 ビット値を取得します。たとえば、`0x00` はインデックス 0、1、2、3 と 4、5、6、7 を選択し、`0x24` はインデックス 16、17、18、19 と 28、29、30、31 を選択します。

ほかに、`xsquare` という主な並べ替え後に 8 ビット粒度の回転をするパラメーターもあります。この例での `xsquare` パラメーターの役割は、次の図の中央に表示されています。

`start (xstart, zstart)` および `step (xstep, zstep)` パラメーターは常にデータ型の粒度になります。このため、16 ビットの場合、値 2 は 2 * 16 ビット先、8 ビット場合、値 2 は 2 * 8 先になります。`step` パラメーターは、選択したデータの次のブロックに適用されます。このため、`offset` パラメーターのペアが 2 * 2 ブロックを選択した場合、`step` は次の 2 * 2 ブロックに適用されます。インデックス値に追加された `step` は、並べ替え粒度 (データの場合は 32 ビット、係数の場合は 16 ビット) に揃える必要があります。たとえば、8 ビットのデータの場合、`xstep` は 4 の倍数にする必要があります。8 ビットの係数の場合、`zstep` は 2 の倍数である必要があります。次の 2 つの図は、`step` がデータと係数に使用されるところを示しています。

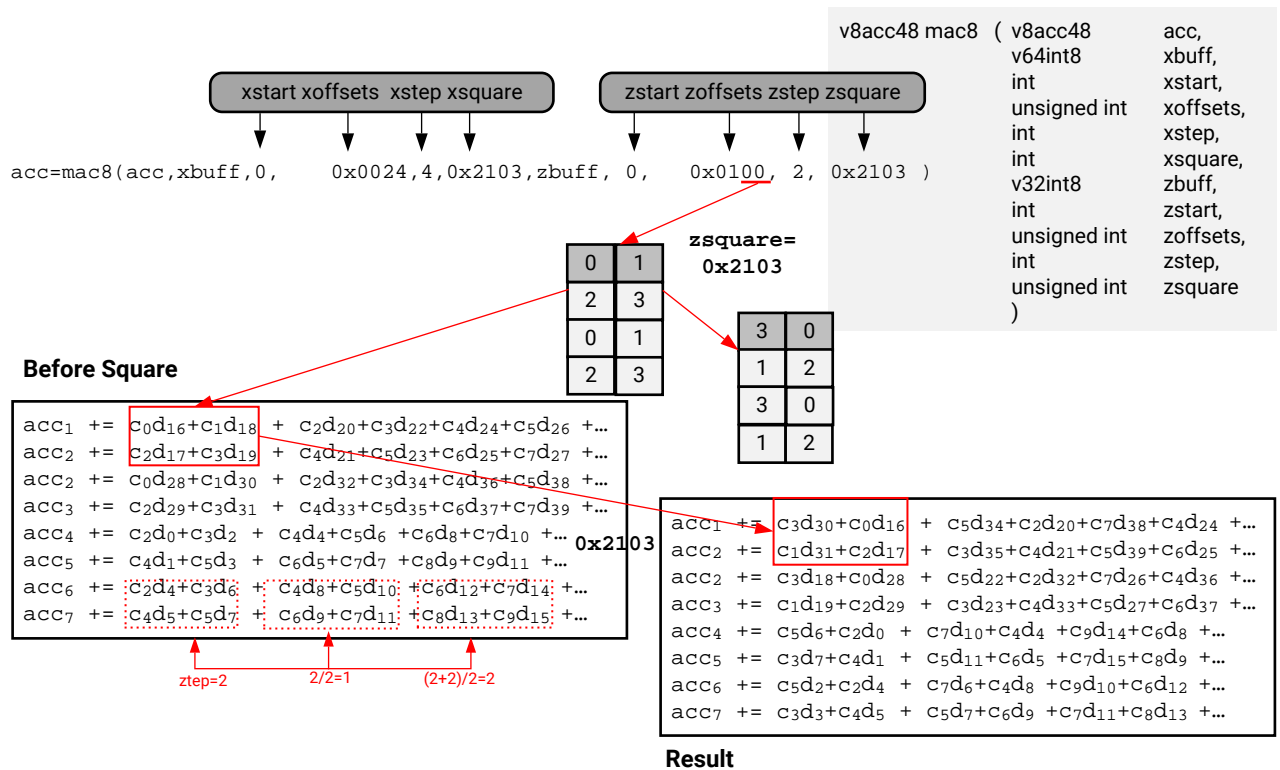
`int8 * int8` 型の係数の場合、では、2 * 2 インデックス ブロックが複製されて、4 * 2 ブロックが作成されます。図 18 のインデックス 0、1、2、3 の複製を参照してください。

図 17: int8 x int8 型の MAC8 (X 部分)



X24811-012021

図 18: int8 x int8 型の MAC8 (Z 部分)



X24812-012021

オプション

MAC 組み込み関数には、前置加算、前置減算、連結などの追加演算を含む多数のセットがあります。ベクター MAC 組み込み演算の命名規則は、次のとおりです。オプションの特性は [], 必須の特性は {} 内に示しています。

```
[1]{mac|msc|mul|negmul}{2|4|8|16}[_abs|_max|_min|_maxdiff][_conj][{_sym|_antisym}][_ct|_uct]][_c|_cc|_cn|_nc]
```

各演算は、乗算、アキュムレータの初期化、または 2、4、8、または 16 レーンの実行中のアキュムレータに累積される MAC 演算のいずれかです。

- 1: 演算に 80 ビット レーンのアキュムレータが使用されることを示します。
- sym および antisym: それぞれ前置加算および前置減算が使用されることを示します。
- max、min、および maxdiff: xbuff のレーンの事前選択が最大、最小、最大差異値に基づくことを示します。
- abs: xbuff の絶対値の事前計算を示します。
- ct: 部分的な前置加算および前置減算が使用されることを示します (最終列の X からのデータ入力に対する別の選択)。
- uct: 一部のタイプの FIR フィルターにユニット センター最適化に使用されます。詳細は、『Versal ACAP AI エンジン イントリックス資料』(UG1078)を参照してください。
- n および c: 複素値を含む入力バッファの 1 つに複素共役が使用されることを示します。

- `c`: 複素入力バッファのみが共役されます。
- `cn`: X (前置加算が使用される場合は XY) バッファの複素共役。
- `nc`: Z バッファの複素共役。
- `cc`: X (前置加算が使用される場合は XY) および Z バッファ両方の複素共役。
- `conj`: Y からのデータ入力を乗算する際に Z の複合共役が使用されることを示します。

データ並べ替えおよび MAC の例

次の例は、実数を `rva`、虚数を `rvb` (`v8int32` 型) に含む 2 つのベクターを取得し、オフセットを使用して必要に応じて値をインターリーブする新しい複素ベクターを作成します。

```
v8cint32 cv = as_v8cint32(select16(0xaaaa, concat(rva, rvb),
0, 0x03020100, 0x07060504, 8, 0x30201000, 0x70605040));
```

次の例は、`v8cint32` 型の `cv` ベクターの実数部と虚数部を抽出します。

```
v16int32 re_im = shuffle16(as_v16int32(cv), 0, 0xECA86420, 0xFDB97531);
v8int32 re = ext_w(re_im, 0);
v8int32 im = ext_w(re_im, 1);
```

シャッフル組み込み関数を使用すると、ベクター内の要素の順序を変更したり、すべての要素を同じ値に設定したりできます。組み込み関数の中には、大きめのレジスタでのみ動作するものもありますが、小さめのレジスタでも簡単に使用できます。次の例は、ベクター内の 4 つの要素すべてを定数値に設定する関数のインプリメント方法を示します。

```
v4int32 v2 = ext_v(shuffle16(xset_v(0, v1), 0, 0, 0), 0);
```

次の例は、`rva` の各要素と `rvb` の最初の要素を乗算する方法を示します。これは、ベクターと定数値を乗算するのに効率的な方法です。

```
v8acc80 acc = lmul8(concat(rva, undef_v8int32()), 0, 0x76543210, rvb, 0, 0x00);
```

次の例は、`rva` の各要素と `rvb` の対応する要素を乗算します。

```
acc = lmul8(concat(rva, undef_v8int32()), 0, 0x76543210, rvb, 0, 0x76543210);
acc = lmul8(upd_w(undef_v16int32(), 0, rva), 0, 0x76543210, rvb, 0, 0x76543210);
```

次の例は、データ ストレージが行ベースである場合に、`mul` 組み込み関数を使用して `int8 x int8` データ型の行列乗算を実行します。

```
//Z_{2x8} * X_{8x8} = A_{2x8}
mul16(Xbuff, 0, 0x11101110, 16, 0x3120, Zbuff, 0, 0x44440000, 2, 0x3210);
//Z_{4x8} * X_{8x4} = A_{4x4}
mul16(Xbuff, 0, 0x00000000, 8, 0x3120, Zbuff, 0, 0xCC884400, 2, 0x3210);
```

カーネルに複数の `mul` または `mac` 組み込み関数がある場合は、`xoffsets` パラメーターと `zoffsets` パラメーターを使用中は一定に保ち、`xtsart` パラメーターと `zstart` パラメーターを変更してください。これにより、スタック上でコンフィギュレーション レジスタがあふれ出ないようにできます。

ベクター レーンの並べ替えの詳細は、『Versal ACAP AI エンジン イントリニクス資料』 ([UG1078](#)) を参照してください。

ループ

AI エンジンには、比較および分岐のための分岐制御オーバーヘッドを発生させない 0 オーバーヘッド ループ構造があり、内部ループ サイクル カウントを削減できます。パイプライン処理により、コンパイラでプリアンブルおよびポストアンブルが追加され、ループ実行中に命令パイプラインが常にフルに保たれます。パイプライン ループを使用すると、前の反復が終了する前に新しい反復を開始して、より高い命令レベルの並列処理を実現できます。

次の図に、0 オーバーヘッド ループのアセンブリ コードを示します。2つのベクター ロード、1つのベクター ストア、1つのスカラー命令、2つのデータ移動、および1つのベクター命令が順に異なるスロットに表示されています。

図 19: 0 オーバーヘッド ループのアセンブリ コード

```

#####
VLDL w0, [p0], m1, c1,c2;      NOP:
00000000000005d8: VLDL w1, [p0], m1, c1,c2;      NOP:
00000000000005e0: VLDL w0, [p0], m1, c1,c2;      NOP:
00000000000005ec: VLDL w1, [p0], m1, c1,c2;      NOP:
00000000000005f4: VLDL w0, [p0], m1, c1,c2;      NOP:
0000000000000600: VLDL w1, [p0], m1, c1,c2;      NOP:
0000000000000608: VLDL w0, [p0], m1, c1,c2;      NOP:
0000000000000614: VLDL w1, [p0], m2, c1,c2;NOP: NO
zero-overhead loop start Label_Z15_F_Z12matmul_vec1IP12input_window$E51_P13output_window$E_336:
0000000000000620: VLDL w0, [p0], m1, c1,c2;      NOP:
000000000000062c: VLDL w1, [p0], m1, c1,c2;      NOP:
0000000000000634: VLDL w0, [p0], m1, c1,c2;      NOP:
0000000000000640: VLDL w1, [p0], m1, c1,c2;      NOP:
0000000000000648: VLDL w0, [p0], m1, c1,c2;      NOP:
0000000000000654: VLDL w1, [p0], m1, c1,c2;      NOP:
000000000000065c: VLDL w0, [p0], m1, c1,c2;      NOP:
0000000000000668: VLDL w1, [p0], m1, c1,c2;      NOP:
0000000000000670: VLDL w0, [p0], m1, c1,c2;      NOP:
000000000000067c: VLDL w1, [p0], m1, c1,c2;      NOP:
0000000000000684: VLDL w0, [p0], m1, c1,c2;      load store
0000000000000690: VLDL w1, [p0], m3, c1,c2;      NOP:
0000000000000698: VLDL w0, [p0], m1, c1,c2;      VST-48.SRSS b0, s1, [p2], m0, c1,c1;NOP:
00000000000006a4: VLDL w1, [p0], m1, c1,c2;NOP:
00000000000006b0: VLDL w0, [p0], m1, c1,c2;NOP: NO
zero-overhead loop end Label_Z16_F_Z12matmul_vec1IP12input_window$E51_P13output_window$E_496:
00000000000006c0: VLDL w0, [p0], m2, c1,c2;NOP: NO
00000000000006d0: MOV u20 p5, #173920;      NOP:
00000000000006e0: NO Scalar Operation move
00000000000006ea: NO Postamble
00000000000006ec: NO
00000000000006f4: NO
00000000000006fe: VST wr0, [p3];

```

次のプラグマは、コンパイラにループをパイプライン処理するように指示し、ループが常に 3 回以上実行されることを知らせます。

```
for (int i=0; i<N; i+=2)
    chess_prepare_for_pipelining
    chess_loop_range(3,)
```

`chess_loop_range(<minimum>, <maximum>)` は、対応するループが少なくとも `<minimum>` 回、多くとも `<maximum>` 回実行されることをコンパイラに知らせます (`<minimum>` およ `<maximum>` は負でない定数式、または省略可能)。省略すると、`<minimum>` はデフォルトで 0 に、`<maximum>` はコンパイラの最大プリセットになります。`<maximum>` はパイプラインのインプリメンテーションには関係ありませんが、`<minimum>` はパイプラインのインプリメンテーションをガイドする値になります。

<minimum> の数値は、ループが実行されるたびに、最小で実行されるループ反復の数を定義します。ソフトウェアパイプラインは、可能であれば、少なくともその数の反復を並行して実行できるように調整されます。また、反復が <minimum> 回繰り返される前にループの境界をチェックする必要がないことも指定します。

ループ範囲がコンパイル時間定数の場合、ループ範囲プラグマは必要ありません。通常、AI エンジン コンパイラはアルゴリズムのパイプラインが最適になるように理論上最適な数値をレポートします。範囲仕様が最適でない場合、コンパイラは警告メッセージを表示し、最適な範囲を提案します。この場合、最初に `<minimum>` を 1 つの `[chess_loop_range(1,)]` に設定して、コンパイラでレポートされる理論的に最適な `<minimum>` を観察することにしても問題ありません。

```
Warning in "matmul_vec16.cc", line 10: (loop #39)
further loop software pipelining (to 4 cycles) is feasible with
`chess_prepare_for_pipelining'
but requires a minimum loop count of 3
... consider annotating the loop with `chess_loop_range(3,)' if applicable,
... or remove the current `chess_loop_range(1,)' annotation
```


この時点で、レポートされた最適な値に <minimum> をアップデートすることもできます。

この2つ目のパイプライン インプリメンテーション部分により、実際の反復回数 <minimum> 回に到達しない場合に AI エンジン カーネルでデッドロックが発生することもあります。このため、反復回数は常に `chess_loop_range` 指示子で指定された数以上にする必要があります。

ループの依存関係は、コードのベクター化に影響します。内部ループの依存関係を削除できない場合は、レベルをステップアウトし、内部ループの複数のコピーが並列で実行される箇所を手動で展開します。

データをループする場合に、特定のオフセットで増減するには、循環バッファに `cyclic_add` 組み込み関数を使用します。`fft_data_incr` 組み込み関数は、バタフライ演算の現在のターゲットであるポインターの反復を有効にします。これらの関数を使用すると、標準の C で同等の機能をコーディングする際に複数のクロック サイクルを節約できます。データ型によっては、パラメーターと戻り値の型を変換する必要があります。

次の例では、実数の行列での演算に `fft_data_incr` 組み込み関数を使用しています。

```
pC = (v8float*)fft_data_incr( (v4cfloat*)pC, colB_tiled, pTarget);
```

使用する前に、ベクター レジスタを完全に満たす連続的なロードがないようにしてください。現在の MAC と次のロードを同じサイクルで実行できる MAC 組み込み関数を使用してロードをインターリーブすることをお勧めします。

```
acc = mul4_sym(lbuff, 4, 0x3210, 1, rbuff, 11, coeff, 0, 0x0000, 1);
lbuff = upd_w(lbuff, 0, *left);
acc = mac4_sym(acc, lbuff, 8, 0x3210, 1, rbuff, 7, coeff, 4, 0x0000, 1);
```

ユース ケースによっては、ループ内の命令を回転させるループ回転が役立つことがあります。ループの開始時にデータをベクターにロードするのではなく、ループの前に最初の反復のデータ ブロックをロードしてから、ループの終了時に近い次の反復のデータ ブロックをロードすることを検討してください。これにより、命令が追加されますが、ループの依存関係の長さが短くなり、ループ範囲が小さくなる可能性がある理想的なループを実現できます。

```
// Load starting data for first iteration
sbuff = upd_w(sbuff, 0, window_readincr_v8(cb_input)); // 0..7

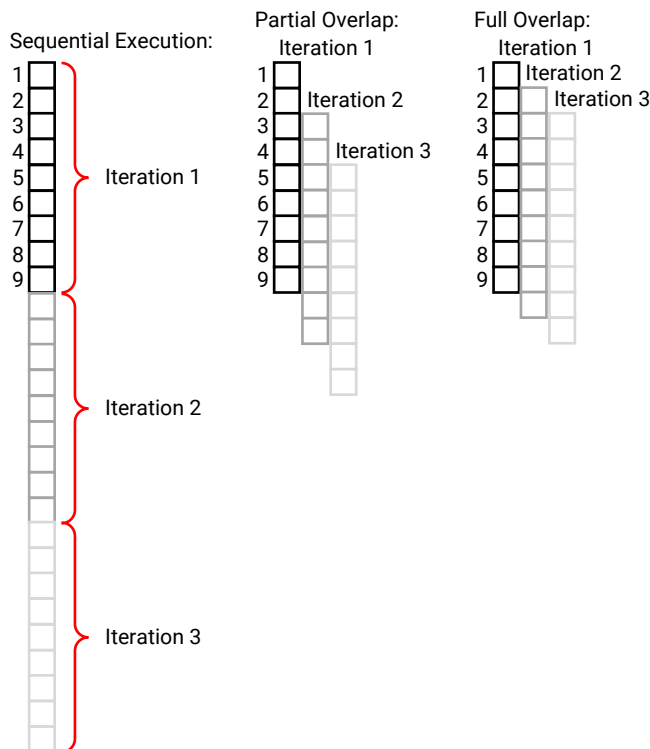
for ( int l=0; l<LSIZE; ++l )
chess_loop_range(5,)
chess_prepare_for_pipelining
{
    sbuff = upd_w(sbuff, 1, window_readincr_v8(cb_input)); // 8..15
    acc0 = mul4_sym(      sbuff,5 ,0x3210,1 ,12 ,coe,4,0x0000,1);

    sbuff = upd_w(sbuff, 2, window_readdecr_v8(cb_input)); // 16..23
    acc0 = mac4_sym(acc0,sbuff,1 ,0x3210,1 ,16,coe,0,0x0000,1);
    acc1 = mul4_sym(      sbuff,5 ,0x3210,1 ,20,coe,0,0x0000,1);
    window_writeincr(cb_output, srs(acc0, shift));
    // Load data for next iteration
    sbuff = upd_w(sbuff, 0, window_readincr_v8(cb_input)); // 0..7
    acc1 = mac4_sym(acc1,sbuff,9,0x3210,1,16,coe,4,0x0000,1);
    window_writeincr(cb_output, srs(acc1, shift));
}
```


ループのソフトウェア パイプライン処理

このセクションでは、ループのソフトウェア パイプライン処理について詳細に説明します。これは、AI エンジンを用意することでプログラムの異なる部分を同時実行するために重要な概念です。例として、次の図に 1 回の反復に合計 9 サイクルかかるループを示します。すべてを順次実行する場合からパイプライン処理して完全にオーバーラップさせる場合までを示しています。

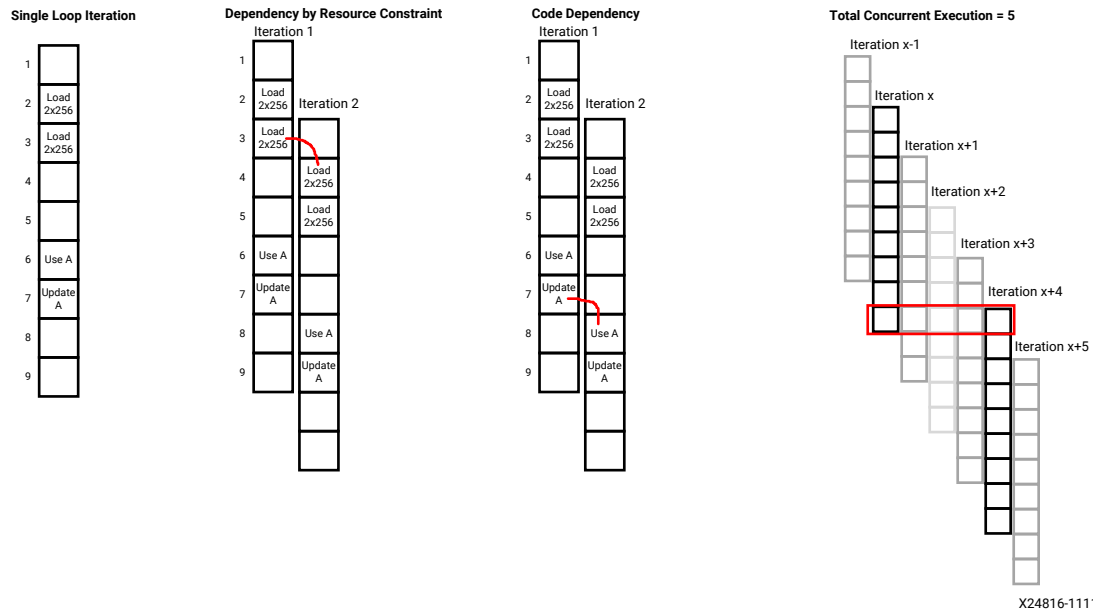
図 20: パイプライン処理の例



X23294-092619

これらの各例のサイクルをカウントすると、順次実行の場合は 3 つのループ反復を完全に実行するのに 27 サイクル必要ですが、部分的にオーバーラップさせたパイプラインでは 13 サイクル、完全にパイプライン処理されたループでは 11 サイクルしかかかりません。そのため、パフォーマンスの面からは完全にオーバーラップさせるパイプライン処理が最適です、ただし、リソース制約および反復間ループの依存関係により、完全にオーバーラップできない場合もあります (次の図を参照)。

図 21: パイプライン処理での依存関係



この例のプログラムは、サイクル 2 でロード A (2 x 256 ビット)、サイクル 3 でロード B (2 x 256 ビット)、サイクル 6 および 7 でループ変数 A に対する演算を実行します。この反復の残りの命令は、ループのパフォーマンス解析には重要ではありません。

このループ反復のサイクル 2 と 3 では、4 x 256 ビットのロード操作が実行されます。AI エンジンではサイクルごとに 2 つのロードしか実行できないので、必要な 4 つのロードが 2 サイクルで実行されます。これは、リソース制約と呼ばれます。この反復を含むループをパイプライン処理する場合、この制約によりオーバーラップが 2 サイクル以上に制限されます。同様に、サイクル 6 とサイクル 7 の反復間のコード依存関係により、追加のオーバーラップが阻止される場合があります。この例では、ループの次の反復を実行する前に A の値がアップデートされている必要であるため、オーバーラップが制限されます。

AI エンジン コンパイラは、各ループに関して次の形式でレポートします。

注記: コアのコンパイル レポートは `Work/aie/core_ID/core_ID.log` に生成されます。詳細レポートを生成するには、`-v` オプションを使用します。

```
HW do-loop #397 in "testbench.cc", line 132: (loop #16) :
Critical cycle of length 2 : b67 -> b68 -> b67
Minimum length due to resources: 2
Scheduling HW do-loop #397
(algo 1a)          -> # cycles: 9
(modulo)           -> # cycles: 2 ok (required budget ratio: 1)
(resume algo)      -> after folding: 2 (folded over 4 iterations)
-> HW do-loop #397 in "testbench.cc", line 132: (loop #16) : 2 cycles
NOTICE: loop #397 contains folded negative edges
NOTICE: postamble created
Removing chess_separator blocks (all)
```

上記の AI エンジン コンパイラ レポートでは、[Critical cycle of length] にコードの依存関係、[Minimum length due to resources] にリソース制約による最小オーバーラップ要件が示されています。algo 1a 行には、1 反復にかかる合計サイクル数が示されます。これらの数値から、最大 5 反復が同時にアクティブになり、パイプラインが形成されています。

AI エンジン コンパイラでは、これら 5 つのオーバーラップ反復 (現在の反復とオーバーラップされた 4 つの反復) が `resume algo` 行にレポートされます。さらに、開始間隔 (II)、次の反復を開始する前に 1 つの反復で実行する必要があるサイクル数 (この例では 2) を示します。

通常は、`chess_prepare_for_pipelining` 指示子を使用してコンパイラでソフトウェア パイプライン処理が実行されるようにするだけで充分です。ループの反復数がコンパイル時定数である場合、chess コンパイラにより最適なソフトウェア パイプラインが作成されます。

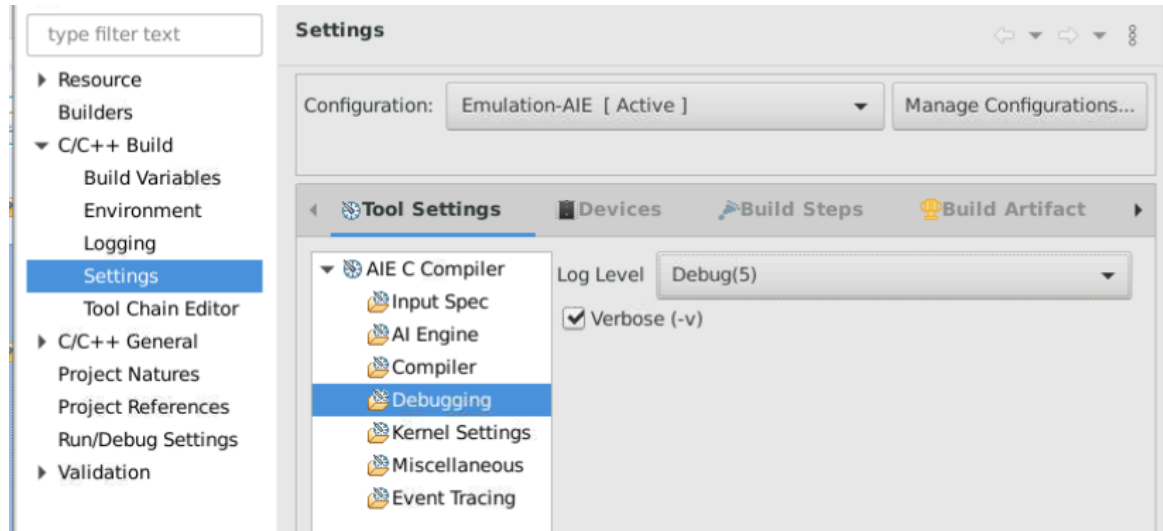
可変ループ範囲 (変数 `start/end` で定義) の場合、効率的なパイプライン ループ構造を作成するため、コンパイラに追加情報を供給する必要があります。これには、`chess_loop_range(<minimum>, <maximum>)` 指示子を使用します。`chess_loop_range(<minimum>, <maximum>)` 指示子の詳細は、[ループ](#) を参照してください。

注記: ループのサイクル数が 64 サイクルを超える場合、コンパイラがそのループのパイプライン処理をディスエーブルにすることがあります。この場合、AI エンジン コンパイラから次のようなメッセージが表示されます。

```
(algo 1a)          -> # cycles: 167 (exceeds -k 64) -> no folding: 167
                  -> HW do-loop #511 in "xxxx", line 794: (loop #8): 167 cycles
```



重要: 詳細レポートを生成するには、Vitis IDE で次の図に示すように [Verbose] をオンにするか、コマンドラインで AI エンジン コンパイラに `-v` オプションを使用します。



AI エンジン コンパイラに `-Xchess=main:backend.mist2.xargs--ggraph` オプションを指定すると、モジュール スケジューリングされたループにモジュール スケジューリング レポートを生成できます。モジュール スケジューリング レポートは、ソフトウェア パイプライン ループに対して、`*_modulo.rpt` in `Work/aie/core_ID/Release/chesswork/<mangled_function_name>/*.rpt` (* はブロック名) という名前で作成されます。モジュール スケジューリング レポートには、レジスタ ファイルのレジスタ ライブ範囲に関する情報も含まれます。これは、効率的でないレジスタの割り当てを見つけるのに役立ち、`chess_storage` を使用することにより改善できます。

コンパイルおよびリンクが完了したら、Vitis アナライザーを使用して各カーネルのコンパイル ログを開くことができます。Vitis アナライザーの詳細は、『Versal ACAP AI エンジン プログラミング環境ユーザー ガイド』([UG1076](#)) を参照してください。

restrict キーワード

C 規格には、データ間の独立性を明示的に示すことにより積極的なコンパイラの最適化ができるように、`__restrict` という特定のポインター修飾子があります。コンパイラは、デフォルトでは、同じアレイの異なるアクセスを区別しません。このため、アレイがパイプラインでアクセスされると、保守的な想定でループ間の間隔を長くできなくなることがあります。これにより、`__restrict` キーワードを使用してツールのパフォーマンスを向上させることが不可欠になる場合があります。`__restrict` キーワードを同じスコープ内のポインターに割り当てると、そのポインターが使用されたときに定義されていない動作になる可能性があるため、`__restrict` キーワードを使用するには注意が必要です。`__restrict` キーワードの概念に関する詳細は、『Vitis 統合ソフトウェア プラットフォームの資料』(UG1416) の AI エンジン フローの [AI エンジンでの restrict キーワードの使用](#) を参照してください。

浮動小数点演算

スカラー ユニットの浮動小数点ハードウェアでは、平方根、逆平方根、逆関数、絶対値、最小値、および最大値がサポートされます。エミュレーションによるほかの浮動小数点演算もサポートされます。エミュレーションを使用するテストベンチとカーネル コード用に `softfloat` ライブラリがリンクされている必要があります。数学ライブラリ関数の場合は、単精度浮動小数点バージョンを使用する必要があります (たとえば、`exp()` の代わりに `expf()` を使用します)。

AI エンジン ベクター ユニットには 8 レーンの単精度浮動小数点積和演算があります。このユニットは、固定小数点データパスと同じベクター レジスタ ファイルおよび並べ替えネットワークを使用します。通常、固定小数点または浮動小数点では、サイクルごとに実行できるベクターは 1 つだけです。

浮動小数点 MAC のレイテンシは 2 サイクルなので、2 つのアクキュムレータをピンポン方式で使用すると、コンパイラが各クロック サイクルで MAC をスケジュールできるため、パフォーマンスが向上します。

```
acc0 = fpmac( acc0, abuff, 1, 0x0, bbuff, 0, 0x76543210 );
acc1 = fpmac( acc1, abuff, 9, 0x0, bbuff, 0, 0x76543210 );
```

現時点では、除算スカラーやベクター組み込み関数はありませんが、ベクター除算は、次の例に示すように反転および乗算することでインプリメントできます。

```
invpi = upd_elem(invpi, 0, inv(pi));
acc = fpmul(concat(acc, undef_v8float()), 0, 0x76543210, invpi, 0, 0);
```

`sqrt`、`invsqrt`、および `sincos` ベクターも同様にインプリメントできます。

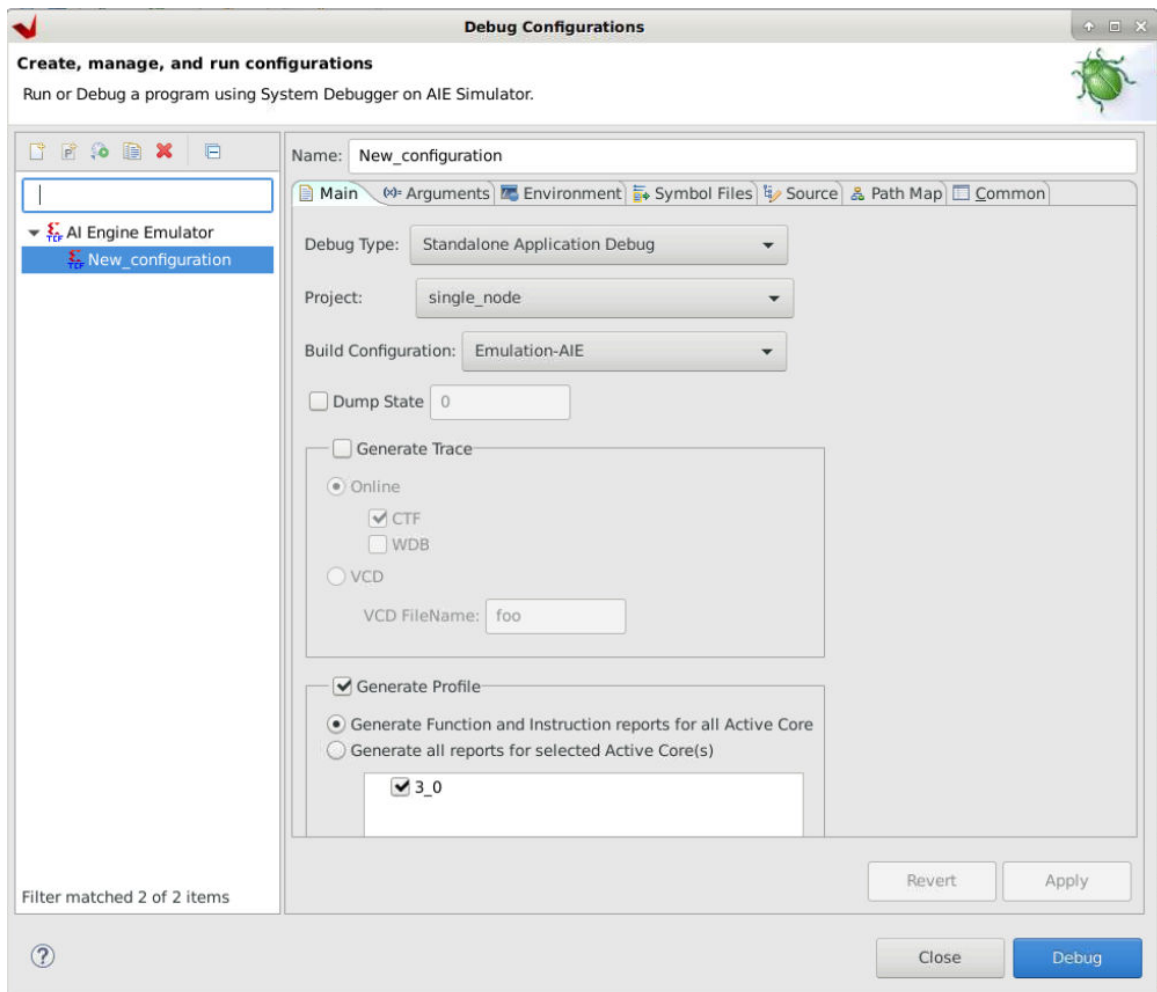
Vitis IDE およびレポートの使用

Vitis IDE では、AI エンジン グラフ、PL カーネル、および PS アプリケーションを使用してシステム プロジェクトを管理します。Vitis IDE には、AI エンジン カーネル開発用の表示ビューがあり、これらはカーネルおよび PS アプリケーションのデバッグに不可欠です。

Vitis IDE には、シングル カーネル開発の開始点として使用可能なシングル ノード グラフの例が含まれます。Vitis IDE には、レジスタ、変数、使用可能なブレークポイント、レジスタ/メモリ マップへの変数、内部/外部メモリの内容を表示するデバッグ ビューと、命令用の [Disassembly] ビューと、シングル AI エンジン カーネルの命令パイプラインを表示する [Pipeline] ビューがあります。

デバッグ設定で [Generate Profile] が選択されている場合は、デバッグ パースペクティブを起動するとコンソールに `printf` 出力が表示され、[Runtime Statistics] ウィンドウには、命令をステップ実行するときのリアルタイム サイクル カウントが表示されます。デバッグ設定の [Generate Trace] チェック ボックスを使用すると、イベント トレース データが生成され、メモリの停止やストリームの停止などのイベントがどのタイミングでどのように発生するかがより理解しやすくなります。イベント トレースは、パフォーマンスの調整に役立ちます。

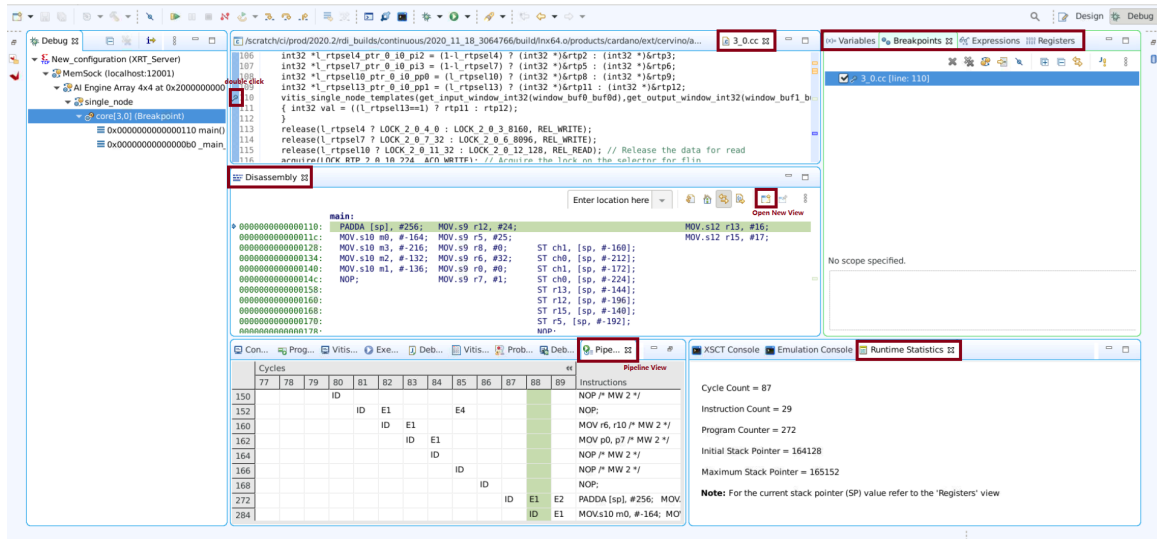
図 22: デバッグ設定



デバッグ パースペクティブでは、デバッグ コマンドの再開、ステップイン、ステップオーバーが使用できます。AI エンジン ソース コードが表示され、行をダブルクリックすると、ブレークポイントを設定できます。[Variables]、[Breakpoints]、[Registers] ウィンドウを使用すると、データ メモリまたはレジスタ ステータスを確認できます。[Disassembly] ビューは、組み込み関数の使用方法、特にパイプラインでのスケジュールを理解するのに役立ちます。[Disassembly] ビューの [Open New View] ボタンをクリックすると、新しいアクティブ ウィンドウが開きます。[Pipeline] ビューでは、特定のクロック サイクルで実行される命令と、マイクロコード/[Disassembly] ビューのラベルを関連付けることができます。

注記: [Pipeline] ビューは単一の AI エンジン デザインでのみ使用でき、デバッグの開始時に [Generate Profile] チェックボックスがオンになっている場合にのみ有効になります。

図 23: デバッグ コード

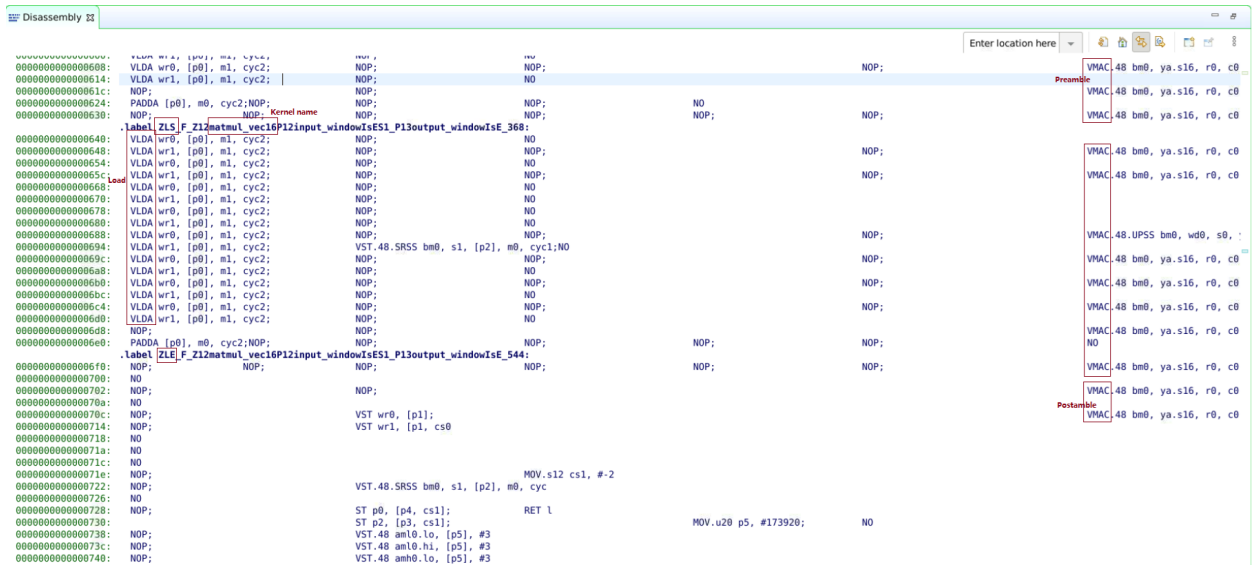


AI エンジン用に生成されるコード (Col_Row.cc) には、コアの AI エンジン カーネルとラッパー コードが含まれます。AI エンジン ラッパー コードでステップイン ボタンを何度かクリックすると、AI エンジン カーネル コードにステップインできます。AI エンジン カーネル ソース ファイルはデザイン パースペクティブから開いて、ブレークポイントを設定することもできます。デバッグやパフォーマンスの調整には、[Disassembly] ビュー、[Pipeline] ビュー、[Memory] ビュー、[Register] ビュー、[Variables] ビューなどの複数のビューを使用できます。

注記: ブレークポイントは、タイルごとに 4 つまでしか設定できません。この数よりも多くのブレークポイントを新たに設定するには、既存のブレークポイントを削除する必要があります。これを超える数のブレークポイントを設定しようとすると、エラー メッセージが表示されます。

[Disassembly] ビューには、ハードウェアをターゲットにしたコンパイラ生成のマイクロコードが表示されます。C/C++ コードは、ソース コードの参照用に行間に埋め込むこともできます。マイクロコードは、コンパイルされた結果、特にループ パイプラインの結果を理解するのに役立ちます。次の図に、パイプライン ループ用に生成されたマイクロコードを示します。[Disassembly] ビューでスクロールまたはステップインすると、カーネルのループを見つけることができます。ループは、0 オーバーヘッド ループの開始 (ZLS) から 0 オーバーヘッド ループの終了 (ZLE) までを繰り返します。ロード命令および MAC 命令の配置とパイプラインも確認できます。プリアンブル命令とポストアンブル命令は 0 オーバーヘッド ループ本体の前後に配置され、パイプライン段を満たしてフラッシュします。

図 24: ループ パイプラインの [Disassembly] ビュー



AI エンジン コアのリンカー メモリ マップ レポートは、Work/aie/core_ID/Release/core_id.map に含まれます。このレポートには、機能、スタティック変数、およびソフトウェア スタックごとにプログラムとデータ メモリの位置がリストされます。これらのレポートから、スタック サイズ、プログラム メモリ サイズ、グローバル バッファ、およびそれらのサイズが抽出できます。リンカー レポートの XML 版 (core_id.map.xml) は、AI エンジン コンパイラに -Xchess=\ "main:bridge.xargs=-fB\ " オプションを指定すると生成できます。

Work/<name>.aiecompile_summary は、Vitis アナライザーで開くことのできるコンパイル サマリです。Work 内のレポートには、カーネルやバッファのマッピング結果など、グラフのコンパイル結果に関する複数のレポートが含まれます。AI エンジン コンパイラ出力に関するその他の情報は、『Versal ACAP AI エンジン プログラミング環境 ユーザー ガイド』 (UG1076) を参照してください。

インターフェイスに関する考慮事項

シングル カーネル プログラミングでは、アルゴリズムを 1 つの AI エンジンにベクター化しますが、複数カーネル プログラミングでは、複数の AI エンジン カーネル間のデータフローを考慮します。

ADF グラフには、PS、PL、およびグローバル メモリとデータをやり取りする 1 つまたは複数のカーネルが含まれます。各 AI エンジン カーネルには、ランタイム比があります。この値は、カーネルの 1 回の呼び出し (1 つのブロック データを処理) にかかるサイクル数のサイクル バジェットに対する比率として計算されます。アプリケーションのサイクル バジェットは、予測されるデータ スループットおよび処理されるブロック サイズによって、通常は固定されています。ランタイム比は、ADF グラフの各 AI エンジン カーネルの制約として指定できます。

合計ランタイム比が 1 未満で複数のカーネルが AI エンジン プログラムメモリに収まる場合は、AI エンジン コンパイラで複数のカーネルが 1 つの AI エンジンに割り当てられます。複数の AI エンジンに割り当てすることも可能です。

ハードウェア リソースを最適に使用するには、ADF グラフと PS、PL、およびグローバル メモリ間のデータ転送、カーネル間のデータ転送に使用可能な異なる方法を理解し、データ移動のバランス取ってメモリまたはストリームのツールを可能な限り最小限に抑えます。これについては、次のセクションで説明します。

AI エンジン間のデータ移動

カーネル間でデータを転送するには、ウィンドウとストリームの 2 つの方法があります。ウィンドウを使用する場合、データ転送はピンポン バッファおよびオプションで 1 つのバッファを使用して達成できます。AI エンジン ツールにより、カーネル間のバッファ同期が処理されます。設計者は、アプリケーションのパーティションを使用して、カーネル間のウィンドウ サイズとバッファの位置を決定する必要があります。データの異なるウィンドウをオーバーラップさせる必要がある場合、AI エンジン ツールでは、ウィンドウのマージンを設定するオプションがあり、AI エンジン ツールによりデータのオーバーラップが自動的にコピーされます。

ストリームを使用する場合、データ移動には、2 つの入力ポートと 2 つの出力ストリーム ポートに加え、1 つの専用カスケード ストリーム入力ポートと出力ポートが使用されます。ストリーム ポートは、各ポートで、1 サイクルごとに 32 ビットまたは 4 サイクルごとに 128 ビットを転送できます。ストリーム インターフェイスは双方向であり、ストリーム ポートで隣接する AI エンジンまたは隣接しない AI エンジンを読み書きできます。ただし、カスケード ストリーム ポートは単方向で、隣接する AI エンジン との一方方向アクセスのみ可能です。

共有メモリを使用したデータ移動

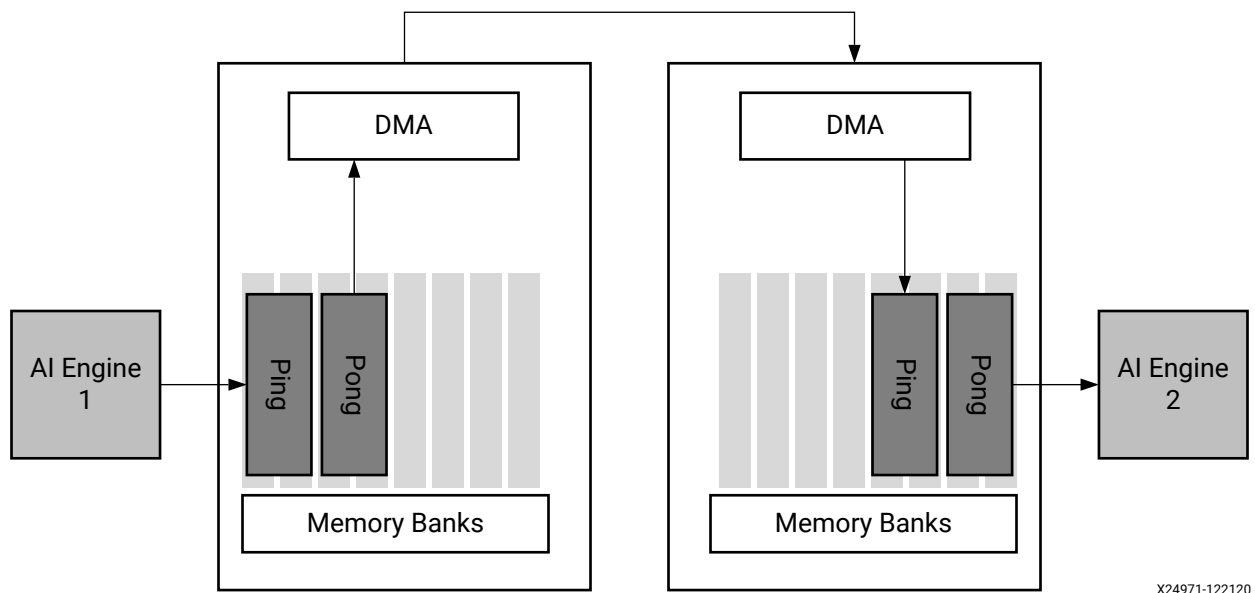
複数のカーネルが 1 つの AI エンジンに収まる場合、2 つ以上の連続するカーネル間のデータは、共有メモリ内の共通バッファを使用して移動できます。この場合、カーネルは時間多重化されるため、必要なバッファは 1 つだけです。

カーネルが隣接する別々の AI エンジン 内にある場合は、ピンポン バッファを使用する共有メモリ モジュールを介して移動できます。これらのバッファは別々のメモリ バンクにあるので、アクセスの競合を回避できます。同期はロックを使用して実行されます。AI エンジン カーネルの入力および出力バッファは、そのバッファに関連付けられたロックによって準備されます。このタイプのデータ移動では、DMA と AXI4-Stream インターコネクトが不要なため、配線リソースが節約されて、データ転送レイテンシがなくなります。

メモリおよび DMA を介したデータ通信

AI エンジンが隣接していない場合は、各 AI エンジンに関連付けられたメモリ モジュール内の DMA を使用して同様のデータフローを確立できます。各メモリ モジュール内のピンポン バッファが使用され、ロックを使用して同期が実行されます。共有メモリ通信と比較すると、通信レイテンシとメモリ リソースが増加します。

図 25: メモリおよび DMA を介したデータ通信



X24971-122120

AXI4-Stream インターコネクトを使用したデータ通信

AI エンジン どちら、DMA やメモリを使用せず AXI4-Stream インターコネクトを介して直接データをやり取りできます。データは、ストリーミング インターフェイスを介して 1 つの AI エンジンから別の AI エンジンに送信またはブロードキャストできます。ストリーミング接続のデータ帯域幅は、1 サイクルごとに 32 ビットで、ビルトイン ハンドシェイクおよびバック プレッシュャー メカニズムを使用できます。

ストリーミング入力および出力インターフェイスでパフォーマンスがストリーム数によって制限される場合は、AI エンジンは 1 つのストリーミング入力または 1 つのストリーミング出力の代わりに、2 つのストリーミング入力または 2 つのストリーミング出力を並列に使用できます。2 つの並列ストリームを使用するには、次の 2 つのマクロをペアで使用することをお勧めします。ここで、idx1 および idx2 は 2 つのストリームです。ストリーム ポートに `--restrict` キーワードを追加すると、並列処理用に最適化されるようになります。

```
READINCR(SS_rsrc1, idx1) and READINCR(SS_rsrc2, idx2)
READINCRW(WSS_rsrc1, idx1) and READINCRW(WSS_rsrc2, idx2)
WRITEINCR(MS_rsrc1, idx1, val) and WRITEINCR(MS_rsrc2, idx2, val)
WRITEINCRW(WMS_rsrc1, idx1, val) and WRITEINCRW(WMS_rsrc2, idx2, val)
```

次に、2つの並列入力ストリームを使用して間隔1でパイプライン処理を達成するサンプルコードを示します。間隔1は、1サイクルごとに2つの読み出し、1つの書き込み、および1つの加算が実行されることを意味します。

```
void simple(    input_stream_int32 * __restrict data0,
               input_stream_int32 * __restrict data1,
               output_stream_int32 * __restrict out) {
    for(int i=0; i<1024; i++)
        chess_prepare_for_pipelining
        {
            int32_t d = READINCR(SS_rsrc1, data0) ;
            int32_t e = READINCR(SS_rsrc2, data1) ;
            WRITEINCR(MS_rsrc1,out,d+e);
        }
}
```

組み込み関数はストリーム操作を直接実行するために使用されますが、AIエンジンで検出されたマップに基づいて2つのストリームを入れ替えないようにすることが重要です。

```
v16float off = *(v16float*)offset;
v8float v8in = undef_v8float();
v8float v8out = undef_v8float();
for(int i=0;i<128/4;i++)
    chess_prepare_for_pipelining
    {
        v8in = concat(getf_wss(0),getf_wss(1)); // reads 8 float values
        v8out = fpadd(v8in,off,0,0);
        put_wms(0,ext_v(v8out,0));
        put_wms(1,ext_v(v8out,1));
    }
```

ストリーム接続は、ユニキャストまたはマルチキャストにできます。マルチキャスト通信の場合、データはすべてのデスティネーションポートに同時に送信されるので、すべてのデスティネーションがデータを受信する準備ができていない場合にのみ送信されることに注意してください。

データ通信におけるウィンドウとストリームの違い

データフローグラフのAIエンジンカーネルは、データストリーム(型付き値の無限長シーケンス)に対して処理を行います。これらのデータストリームは、ウィンドウと呼ばれるブロックに分けることができ、カーネルで処理できます。カーネルは、入力データブロックを消費し、出力データブロックを生成します。初期化関数は、カーネルが入力データを処理し始める前に実行するように指定できます。カーネルはメモリからスカラーまたはベクターを読み出すことができますが、各読み出しおよび書き込み操作の有効なベクター長は128ビットまたは256ビットにする必要があります。入力データと出力バッファのウィンドウは、実行される前にカーネル用にロックされます。入力データウィンドウにはカーネルの開始前に入力データを挿入する必要があるため、ストリームインターフェイスと比較してレイテンシが増加します。カーネルはデータのウィンドウ内でランダムアクセスを実行でき、前のブロックからのバイトをいくつか必要とするアルゴリズムのウィンドウマージンを指定できます。

また、カーネルはサンプル単位でデータストリームにアクセスできます。ストリームは連続データに使用され、読み出しおよび書き込みにブロッキング呼び出しを使用します。カスケードストリームは、ブロッキングアクセスのみをサポートします。AIエンジンでは、32ビットのストリーム入力ポート2つと、32ビットのストリーム出力ポート2つがサポートされます。データストリームの読み出しまたは書き込みに有効なベクター長は、32ビットまたは128ビットです。パケットストリームは、プログラム内の独立したデータストリームの数が、使用可能なハードウェアストリームチャネルまたはポートの数を超える場合に便利です。AIエンジンインターコネクトとPLとの相互接続は、ストリームを介して実行されます。

次の表に、カーネル間でのウィンドウ接続とストリーム接続の違いを示します。

表 5: ウィンドウ接続とストリーム接続の違い

接続	マージン	パケットスイッチング	バックプレッシャー	ロック	VLIW による最大スループット (サイクルごと)	ソースとしてマルチキャスト
ウィンドウ	あり	あり ¹	なし	あり	2 × 256 ビット ロード + 1 × 256 ビット ストア	なし
ストリーム	なし	あり ¹	あり	なし	2 × 32 ビット読み出し + 1 × 32 ビット書き込み、または 1 × 32 ビット読み出し + 2 × 32 ビット書き込み	あり

注記:

1. パケット スwitching は、AI エンジン カーネルと PL カーネルの間でのみサポートされます。

グラフ コードは C++ で、カーネル ソース ファイルとは別のファイルです。コンパイラは、メモリ要件を考慮し、データフローに必要なすべての接続をしつつ、AI エンジン カーネルを AI エンジン アレイに配置します。コアの使用量が少ない複数のカーネルは、1 つのタイルに配置できます。

AI エンジン ツールでのグラフ プログラミングの概要は、『Versal ACAP AI エンジン プログラミング環境ユーザー ガイド』 ([UG1076](#)) を参照してください。

フリーランニング AI エンジン カーネル

AI エンジン カーネルは常に `graph::run(-1)` を使用すると実行できます。これを使用すると、カーネルが最後の反復が完了した後に自動的に再起動します。

注記: `graph::run()` の引数を指定しない場合、AI エンジン カーネルは以前に指定した回数の反復を実行します (グラフを引数なしで実行した場合、この回数はデフォルトで無限)。グラフが有限の反復回数で実行される場合 (たとえば `mygraph.run(3); mygraph.run();`)、2 回目の実行呼び出しも 3 回の反復で実行されます。

ただし、開始する前に入力バッファと出力バッファの準備が完了している必要があります。このため、カーネル実行の反復間には、わずかにオーバーヘッドがあります。このセクションでは、オーバーヘッドが 0 で永久に動作するタイプのカーネルを構築する方法について説明します。これは、フリーランニング AI エンジン カーネルと呼ばれます。

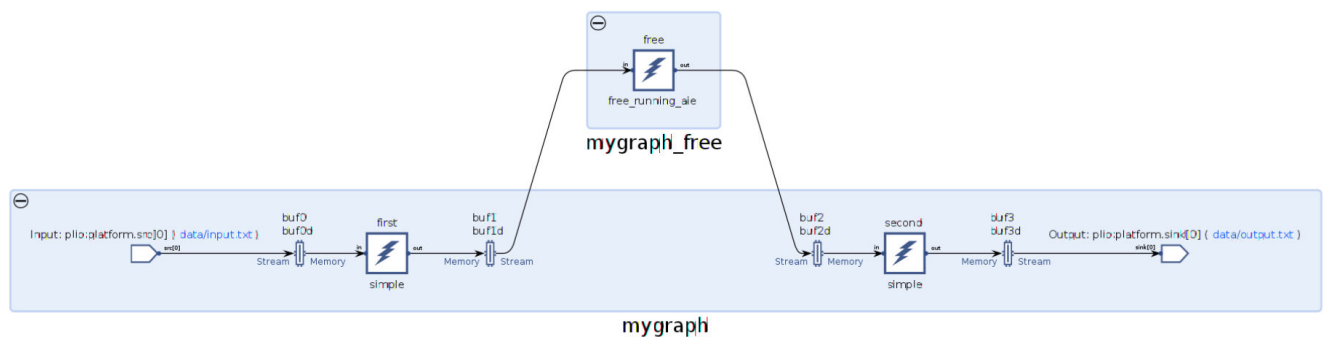
フリーランニング カーネルには、ストリーミング インターフェイスしか含まれません。無限に反復するループは、カーネル内に含めることができます。次に例を示します。

```
void free_running_aie(input_stream_cint16 * in,
    output_stream_cint16 * out) {
    while(true){ //This can be syntax supported by C++, for example: for(;;)
        writeincr(out, readincr(in));
    }
}
```

フリーランニング カーネルには、独自のグラフが定義されている必要があります。このグラフには、ほかのフリーランニングではないカーネルを含めないようにする必要があります。これは、グラフが停止することがないので、フリーランニング以外のカーネルは起動後に制御を失うからです。フリーランニング カーネルを含むグラフは、ほかのグラフに接続できる最上位グラフである必要があります。または、PLIO または GMIO に接続することもできます。フリーランニング グラフとほかのグラフの接続例を次に示します。

```
simpleGraph mygraph; //normal graph
freeGraph mygraph_free; //graph with free running kernel
simulation::platform<1,1> platform("data/input.txt", "data/output.txt");
connect<> net0(platform.src[0], mygraph.in);
connect<> net1(mygraph.sout,mygraph_free.in);
connect<> net2(mygraph_free.out,mygraph.sin);
connect<> net3(mygraph.out, platform.sink[0]);
```

図 26: フリーランニング グラフの接続



フリーランニング グラフは、`mygraph_free.run(-1)` を使用して開始することも、ロード後に自動的に開始することもできます。

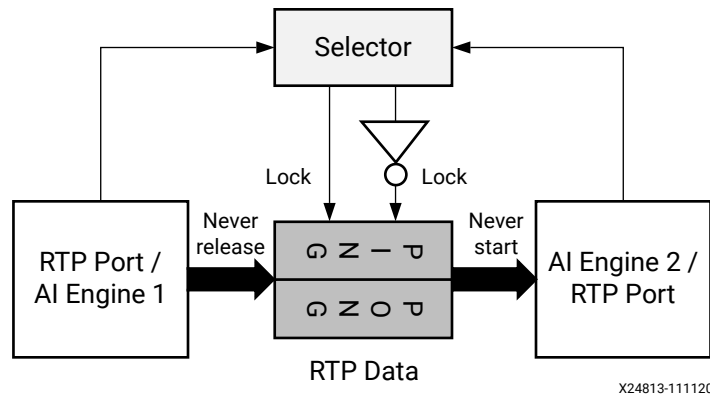
ランタイム パラメーターの指定

ランタイム パラメーター (RTP) は、カーネルにデータを渡す別の方法です。ランタイム パラメーターでは、2 つのタイプの実行モデルがサポートされています。

1. 非同期パラメーターは、Arm® などの制御プロセッサまたは別の AI エンジンにより、いつでも変更可能です。これらは、カーネルが起動されるたびに読み出されます。つまり、パラメーターのアップデートは、カーネルの異なる実行間で実行されますが、アップデートが特定のパターンで実行される必要はありません。たとえば、これらのタイプのパラメーターは、頻繁に変更されないフィルター係数として使用されます。
2. 同期パラメーター (トリガー パラメーター) は、Arm または別の AI エンジンなどの制御プロセッサでこれらのパラメーターが書き込まれるまで、カーネルが実行されないようにします。書き込まれると、カーネルが新しいアップデート値を読み出し、1 回実行します。これが完了すると、パラメーターが再びアップデートされるまで実行されません。これにより、通常のストリーミング モデルから異なるタイプの実行モデルを使用できるようになり、同期化をブロックすることが重要な特定のアップデート操作に役立つことがあります。

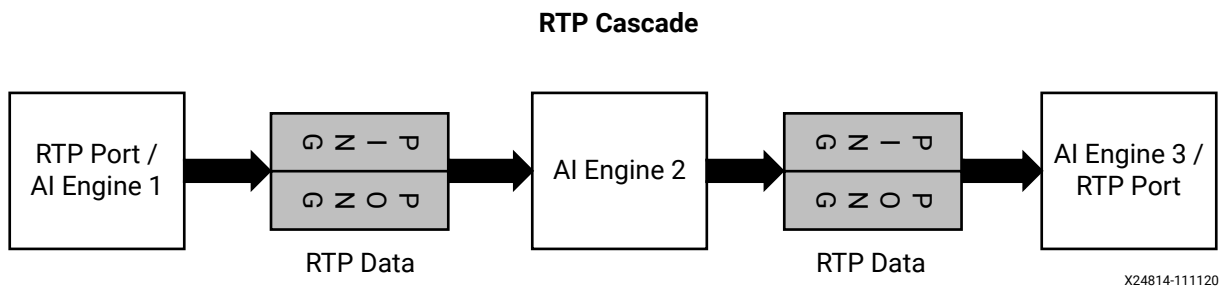
次の図に、AI エンジン RTP がハードウェアでどのように実現されるかを示します。RTP のソースは、制御プロセッサによって書き込まれるポート、または AI エンジン カーネルによって出力される RTP ポートにすることができます。RTP のデスティネーションは、プロセッサを制御して読み出される出力、または AI エンジン カーネルの RTP 入力にすることができます。ソースおよびデスティネーションは、RTP データ転送にピンポン バッファを使用します。ソースおよびデスティネーションはどちらも、セレクター値を読み出して、RTP 値の ping または pong を書き込むのか読み出すのかを判断します。書き込みまたは読み出しの前に、ソースとデスティネーションはカーネルの実行を開始する前にバッファのロックを試みます。

図 27: AI エンジン RTP



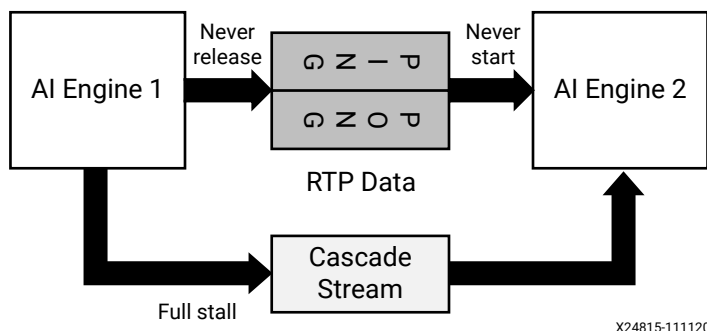
次の図に示すように、RTP のカスケード接続がサポートされます。RTP 接続では、同期から同期モードまたは非同期から非同期モードのみを使用可能です。非同期から同期モードまたは同期から非同期モードは使用できません。複数のデスティネーションにブロードキャストするのに RTP ポートを使用することはできません。

図 28: RTP のカスケード接続



AI エンジン カーネル間の RTP の通信は、カーネル実行境界でのみ発生するということを理解しておくことが重要です。つまり、ソース カーネルの RTP 出力は、ソース カーネルが現在の反復を完了したときに、デスティネーション カーネルでのみ読み出すことが可能です。ソースおよびデスティネーションがカーネル実行を完了または開始するのにお互いに依存している場合、デッドロードが発生する可能性があります。たとえば、ソースとデスティネーションが RTP 接続だけでなくカスケード ストリームで接続されている場合、カスケード ストリームはフルになるとソース AI エンジンをストールします。ただし、ソース カーネルは実行を完了していないため、RTP データは解放されません。そのため、デスティネーション AI エンジンは RTP データを取得せず、開始することはありません。

図 29: RTP カスケード デッドロックの例



注記: AI エンジン カーネルの RTP ポートは、カーネル実行の前後にロックおよびロック解除する必要があります。これにより、各カーネル反復に少しのオーバーヘッドが発生します。データをフレームに分割する場合は、システムレベルのパフォーマンス要件に応じてオーバーヘッドを考慮する必要があります。

ランタイム パラメーターの使用に関する詳細は、『Versal ACAP AI エンジン プログラミング環境ユーザー ガイド』(UG1076) を参照してください。

AI エンジンと PL カーネル間のデータ移動

AI エンジン アレイ インタフェースには、AXI4-Stream 接続を使用して AI エンジンと PL カーネル間で通信するモジュールが含まれます。通常、PL インタフェースはストリーム インタフェースを介してデータを生成または消費します。これらは AI エンジン ストリームを介して AI エンジン カーネルと接続されます。ウィンドウまたはストリームのデータが AI エンジン カーネルによって通信されるかどうかによって、DMA とピンポン バッファが使用される可能性があります。

PL カーネルは AI エンジン カーネルよりも低い周波数で動作することに注意してください。データは AI エンジン クロックと PL クロックの間のクロック ドメインをまたぐ (CDC) 必要があります。CDC パスは Vitis™ 環境で自動的に処理されます。可能であれば、PL カーネルの周波数を AI エンジンの周波数の整数係数として実行することをお勧めします。たとえば、AI エンジン クロック周波数の $\frac{1}{2}$ または $\frac{1}{4}$ で実行します。

AI エンジンと PL 間のレート マッチングの注意点は、『Versal ACAP AI エンジン プログラミング環境ユーザー ガイド』(UG1076) を参照してください。

GMIO を介した DDR メモリ アクセス

AI エンジンの主なデータ ストリームには、AI エンジン-to-PL ストリーミング インタフェースと GMIO があり、GMIO はグローバル メモリとの外部メモリ マップ接続に使用されます。PS と AI エンジン間のインタフェースは、コンフィギュレーションなどのスループットの低い目的をターゲットにできます。GMIO には、AI エンジン-GMIO と PL-GMIO の 2 種類があります。AI エンジン-GMIO は、AI エンジン-NOC マスター ユニット (NMU) を介して DDR メモリに直接接続します。PL-GMIO は PL-NOC NMU を使用しますが、AI エンジンは PL カーネルを介して DDR メモリに接続し、PL カーネルは AI エンジン-to-PL ストリーミング インタフェースを介して AI エンジンと接続します。PL-GMIO を使用する PL カーネルはグラフに含まれ、AI エンジン カーネルと一緒に ADF API で制御できます。

AI エンジン GMIO の帯域幅は、プラットフォームで使用される NMU および DDR メモリ コントローラーの数の影響を受けます。PL-GMIO は、プラットフォームで使用される DDR メモリ コントローラーの数や PL カーネルから DDR メモリへのインターフェイスに加えて、PL カーネルのクロック周波数と AI エンジン-to-PL インターフェイスの影響も受けます。

AI エンジン GMIO の利点には、DDR メモリに直接アクセスできることのほか、AI エンジン シミュレータ用の仮想プラットフォームであるだけでなく、PL カーネルなしのハードウェアでも動作できることが挙げられます。GMIO プログラミング モデルの詳細は、『Versal ACAP AI エンジン プログラミング環境ユーザー ガイド』 ([UG1076](#)) を参照してください。

デザイン解析およびプログラミング

AI エンジン、大量の VLIW および SIMD 計算ユニットを革新的なメモリおよび AXI4-Stream ネットワークを介して相互接続することにより、高い計算密度を実現します。アプリケーションを AI エンジンにインプリメントする場合、AI エンジンおよびデータの計算ニーズとスループット要件を評価することが重要です。たとえば、AI エンジンが PL カーネルおよび外部 DDR メモリとどのようにデータをやり取りするかなどを評価します。AI エンジンの計算およびデータのスループット要件を満たすことができれば、次の手順では、アルゴリズムを AI エンジン アレイにマップするため分割統治法を使用します。分割統治法を使用するには、ベクター プロセッサ アーキテクチャ、メモリ構造、AXI4-Stream、およびカスケード ストリーム インターフェイスについて理解する必要があります。この手順は通常複数回繰り返します。同時に、各 AI エンジン カーネルを最適化し、グラフを構築して反復的に最適化します。AI エンジン カーネルおよびグラフをシミュレーションおよびデバッグするには、AI エンジン ツールを使用します。その後、グラフを PL カーネル、GMIO、および PS と統合し、システム レベルの検証とパフォーマンスの調整を実行します。

この章では、分割統治法を使用してアルゴリズムをデータフロー図 (DFD) にマップする方法について簡単に説明します。シングル カーネル プログラミングと複数カーネル プログラミングの例を示し、計算バウンドおよびメモリ バウンド、シングル カーネルのベクター化と最適化、異なるカーネル間のストリーミングのバランス調整により、カーネルを分割する方法を示します。

AI エンジンへのアルゴリズムのマップ

純粋なソフトウェア モデルから開始する場合は、まずアプリケーション境界を特定する必要があります。アプリケーション境界を特定したら、アプリケーションの入力と出力を定義できます。これで、アプリケーションを特定の演算を実行するコンポーネント (プロセッシング ユニット) に分割できます。データは、入力から 1 つまたは複数のコンポーネントを介して出力に送られます。これは、データフロー図 (DFD) と呼ばれます。

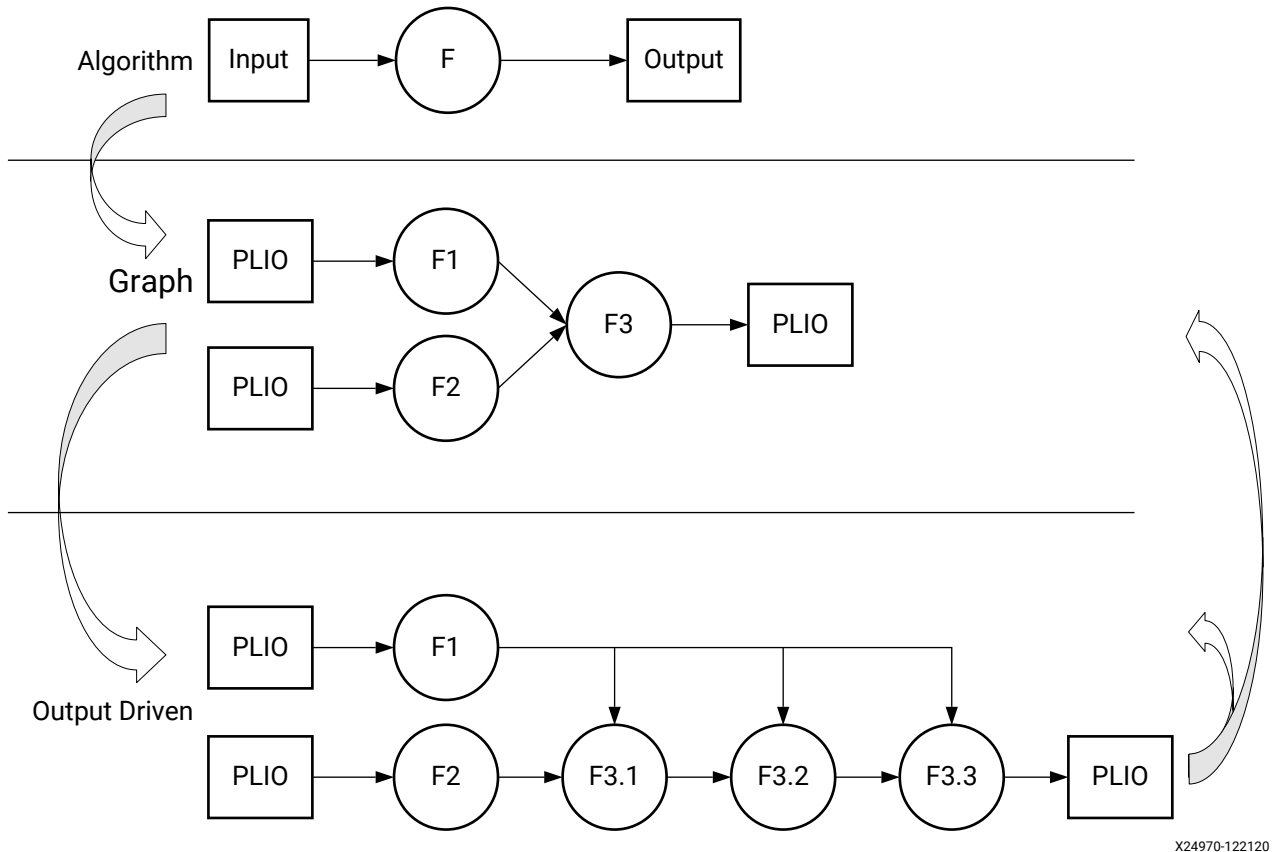
DFD は、グラフ (AI エンジンで実行される実際のデザイン) にマップできます。マップを実行する際は、システムのパフォーマンス要件に従って、入力ソースと出力ソース、およびそのスループットを考慮する必要があります。理想的には、高スループットのデータは PLIO または GMIO を通過する必要があります。PS は、グラフの RTP ポートからの帯域幅の狭いコンフィギュレーション データを処理できます。PL カーネルが提供できる帯域幅によって、AI と PL 間のインターフェイスを決定できます。

グラフ インタフェースを定義したら、グラフ内の要素をさらに別のカーネルに分割できます。このカーネル分割プロセスは、通常スループット重視です。1 つの AI エンジン カーネルで、AI エンジンの計算能力、データ メモリのサイズ、プログラム メモリのサイズの上限を超えないようにする必要があります。また、カーネル間のデータ転送を測定し、バランスを取る必要があります。AI エンジン カーネル間のデータ転送には、ピンポン メモリ、ストリーム、カスケード ストリームなど複数の方法があります。これには、使用可能なストリーム インターフェイス、使用可能なメモリ、およびバッファの位置を考慮する必要があります。これらは、カーネル間のデータ通信とデータ取得スループットに直接影響します。カーネルをカスケード ストリームでチェーン接続された複数のカーネルに分割する場合、これらのカーネルはカスケード ストリームのデータを生成または消費することにより厳密に統合および同期化されます。

カーネルを分割した後、カーネルの I/O インターフェイスとスループット 要件を明確にする必要があります。計算、メモリ、およびインターフェイスが理論上要件を満たすことができる場合は、シングル カーネルのベクター化を考慮することをお勧めします。

アプリケーションの分割とカーネル プログラミングは、シミュレーションで高速に反復できます。これらのプロセスをさらに進めていくと、カーネル間でデータフローを再分割したり、再構築したりすることが必要となる場合があります。次の図に、アルゴリズムのグラフへのマップ、アプリケーションの複数 AI エンジン カーネルへの分割、スループット重視のデータフロー調整を反復するこのプロセスを示します。

図 30: データフロー図のグラフへのマップ



X24970-122120

カーネルのランタイム比によって、1 つまたは複数のカーネルを 1 つの AI エンジンにマップできます。カーネルのランタイム比は、次の式で計算できます。

$$\text{ランタイム比} = (\text{カーネルの 1 回の実行にかかるサイクル数}) / (\text{サイクル バジレット})$$

サイクル バジレットは、カーネルの 1 回の実行に許容されるサイクル数で、システムのスループット要件によって決まります。カーネルの 1 回の実行にかかるサイクル数は、設計の初期段階で見積もり、ベクター化 コードが使用可能になったときに AI エンジン シミュレータでプロファイリングできます。

ランタイム比の設定は、グラフ仕様に必要であり、カーネル間でのデータ通信方法に影響します。データ通信の詳細は、[AI エンジン間のデータ移動](#) を参照してください。

グラフを構築し、カーネルを AI エンジン タイルで実現できる場合、各 AI エンジン カーネルがベクター化されます。概念を実証するため、初めは一部のカーネルをベクター化せずにおくことができます。

ベクター化のプロセスは、ベクター データ型およびベクター組み込み関数に基づきます。これには、レジスタと計算用にロード可能なデータを考慮する必要があります。通常、最適化の目的は、カーネルのメイン ループの開始間隔 (II) を最小限にすることです。これには、データ ロード、計算とストアをインターリーブし、ループのデータ依存と計算依存を解決します。階層ループを最適化する際は、外側ループをパイプライン処理し、内側ループのサイズがそれほど大きくない場合に内側ループを展開すると有益です。[第 2 章: シングル カーネルのプログラミング](#) で説明した手法は、シングル カーネルのベクター化と最適化に適用されます。

シングル カーネルのコーディング例

次のセクションでは、アルゴリズムを 1 つの AI エンジンにマップすることについて説明します。アルゴリズムの計算バウンドおよびメモリ バウンドと、AI エンジンの機能を考慮します。コードのスカラー バージョンとベクター バージョンを示し、ベクター化を説明します。

行列ベクター乗算

次の行列ベクター乗算の例は、1 つの AI エンジン カーネル ベクター化を示しており、次の行列ベクター乗算式をインプリメントします。

$$C(64 \times 1) = A(64 \times 16) * B(16 \times 1)$$

例では、行列のデータが列ベースの形式で格納され、行列 A と B のデータ型は int16 です。

```
c0 = a0*b0 + a64*b1 + a128*b2 + a192*b3 + a256*b4 + a320*b5 + a384*b6 +
a448*b7 + ...
c1 = a1*b0 + a65*b1 + a129*b2 + a193*b3 + a257*b4 + a321*b5 + a385*b6 +
a449*b7 + ...
c2 = a2*b0 + a66*b1 + a130*b2 + a194*b3 + a258*b4 + a322*b5 + a386*b6 +
a450*b7 + ...
c3 = a3*b0 + a67*b1 + a131*b2 + a195*b3 + a259*b4 + a323*b5 + a387*b6 +
a451*b7 + ...
...
c60 = a60*b0 + a124*b1 + a188*b2 + a252*b3 + a316*b4 + a380*b5 + a444*b6 +
a508*b7 + ...
c61 = a61*b0 + a125*b1 + a189*b2 + a253*b3 + a317*b4 + a381*b5 + a445*b6 +
a509*b7 + ...
c62 = a62*b0 + a126*b1 + a190*b2 + a254*b3 + a318*b4 + a382*b5 + a446*b6 +
a510*b7 + ...
c63 = a63*b0 + a127*b1 + a191*b2 + a255*b3 + a319*b4 + a383*b5 + a447*b6 +
a511*b7 + ...
```

カーネル コーディングの制限

この例では、1 つの出力値ごとに、合計 16 個の int16 x int16 乗算が必要です。行列 C は 64 個の値で構成されているため、1 つの行列乗算を完了するのに合計で $16 \times 64 = 1024$ 個の乗算が必要です。1 つの AI エンジンで 1 サイクルごとに 32 個の 16 ビット乗算を実行できるので、行列乗算に必要な最小サイクル数は $1024/32 = 32$ です。個々の項の加算は、MAC 演算の乗算と共に実行できるので、追加のサイクル要件はありません。これにより、カーネルの計算バウンドは次のようになります。

$$\text{計算バウンド} = 32 \text{ サイクル/呼び出し}$$

次に、カーネルのメモリ アクセス バウンドを解析します。ベクター ユニットの MAC パフォーマンスが最大限に使用された場合、1 サイクルごとに 32 個の 16 ビット乗算が実行されます。ベクター b は 16×16 ビット = 256 ビットなので、ベクター レジスタに格納できます。各 MAC 演算ごとに AI エンジン データ メモリまたはタイル インターフェイスからフェッチする必要はありません。計算に必要なデータ a には、1 サイクルごとに 32×16 ビット = 512 ビットが必要です。ストリーム インターフェイスでは、1 サイクルごとに 2×32 ビットのみがサポートされるので、メモリからのデータのフェッチを考慮できます。1 サイクルごとに 2 つの 256 ビット ロードが可能で、これは MAC のパフォーマンスに一致します。各サイクルで 2 つの 256 ビット ロードが実行された場合、カーネルのメモリ バウンドは次のようになります。

メモリ バウンド = 32 サイクル/呼び出し

計算バウンドおよびメモリ バウンドは、カーネル実現の論理的な制限です。メインの計算ループ外の関数オーバーヘッドは考慮されていません。カーネルがグラフの一部である場合、ほかのカーネルの帯域幅制限または低いシステムパフォーマンス要件により、カーネルが解放されることがあります。

ベクター化

複雑なベクター処理アルゴリズムでは、スカラー バージョンから開始することをお勧めします。これは、精度を検証するためのゴールデン リファレンスとしても利用できます。次に、行列乗算のスカラー バージョンを示します。

```
void matmul_scalar(input_window_int16* matA,
    input_window_int16* matB,
    output_window_int16* matC){ //A[M,N], B[N,1], C[M,1]. M=64, N=16
    for(int i=0; i<M; i++){
        int temp = 0 ;
        for(int j=0; j<N; j++){
            temp += window_read(matA)*window_readincr(matB) ;
            window_incr(matA,64); //Jump of 64 elements to access the next
            element of the same row
        }
        window_writeincr(matC,(int16_t)(temp>>15)) ;
        window_incr(matA,1); //Jump of one element for moving to the next
        row.
    }
}
```

上記のコードでは、matA は列ベースに格納され、matB はカーネルへの循環バッファです。これは、出力の異なる行を計算するため、window_readincr (バッファの最初にループ バック) で連続読み出しできます。

合計 64 個の出力 ($M = 64$) があり、各出力には 16 回 ($N = 16$) の乗算が必要です。データ型 $\text{Int16} \times \text{Int16}$ に対してベクター処理を実行する MAC 組み込み関数を選択する場合、レーン 4、8、16 を選択して式を実行できます。これを次の図に示します。

図 31: レーンの選択

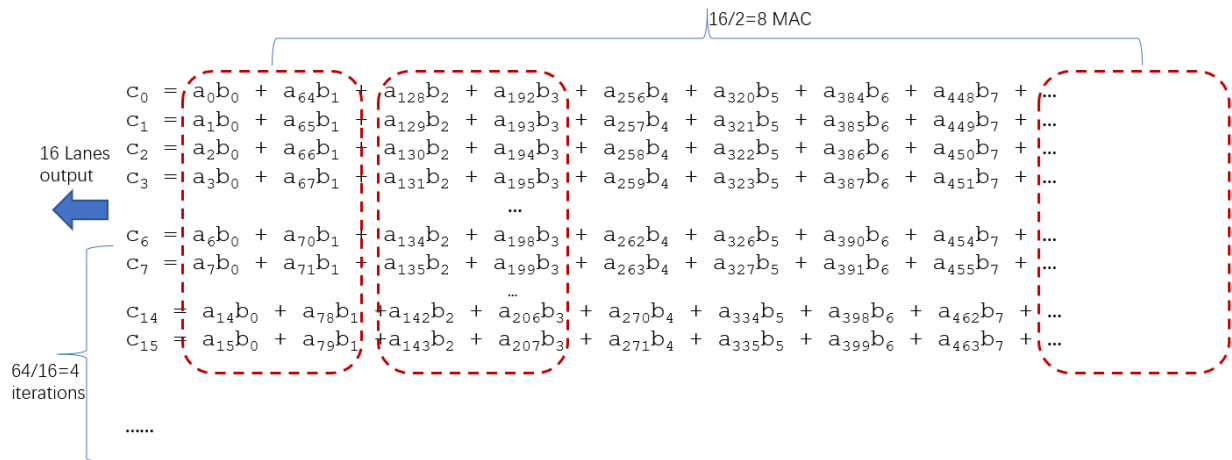
$$\begin{aligned}
 c_0 &= a_0b_0 + a_{64}b_1 + a_{128}b_2 + a_{192}b_3 + a_{256}b_4 + a_{320}b_5 + a_{384}b_6 + a_{448}b_7 + \dots \\
 c_1 &= a_1b_0 + a_{65}b_1 + a_{129}b_2 + a_{193}b_3 + a_{257}b_4 + a_{321}b_5 + a_{385}b_6 + a_{449}b_7 + \dots \\
 c_2 &= a_2b_0 + a_{66}b_1 + a_{130}b_2 + a_{194}b_3 + a_{258}b_4 + a_{322}b_5 + a_{386}b_6 + a_{450}b_7 + \dots \\
 c_3 &= a_3b_0 + a_{67}b_1 + a_{131}b_2 + a_{195}b_3 + a_{259}b_4 + a_{323}b_5 + a_{387}b_6 + a_{451}b_7 + \dots \\
 &\dots \\
 c_6 &= a_6b_0 + a_{70}b_1 + a_{134}b_2 + a_{198}b_3 + a_{262}b_4 + a_{326}b_5 + a_{390}b_6 + a_{454}b_7 + \dots \\
 c_7 &= a_7b_0 + a_{71}b_1 + a_{135}b_2 + a_{199}b_3 + a_{263}b_4 + a_{327}b_5 + a_{391}b_6 + a_{455}b_7 + \dots \\
 &\dots \\
 c_{14} &= a_{14}b_0 + a_{78}b_1 + a_{142}b_2 + a_{206}b_3 + a_{270}b_4 + a_{334}b_5 + a_{398}b_6 + a_{462}b_7 + \dots \\
 c_{15} &= a_{15}b_0 + a_{79}b_1 + a_{143}b_2 + a_{207}b_3 + a_{271}b_4 + a_{335}b_5 + a_{399}b_6 + a_{463}b_7 + \dots
 \end{aligned}$$

4、8、16 レーン MAC の主な違いは、データの消費方法です。データが列ごとに格納される場合、MAC 演算用にロードする必要がある連続データは、 $a_0 \sim a_{15}$ と $a_{64} \sim a_{79}$ の 2 つだけなので、16 レーンの MAC が最適な選択肢となります。 $a_0 \sim a_{15}$ は 256 ビットで、1 つのロードでベクター レジスタに値をロードできます。

同じサイクルで 2 つのロードが発生するようにするには、 $a_0 \sim a_{15}$ と $a_{64} \sim a_{79}$ を異なるデータ バンクに配置する必要があります。データは、カーネルへの 2 つの異なるバッファに列ごとに分割する必要があります。つまり、 $a_0 \sim a_{63}$ は最初のバッファに、 $a_{64} \sim a_{127}$ は 2 番目のバッファに、 $a_{128} \sim a_{191}$ は最初のバッファに、というように配置します。

ベクター化により、行列乗算は $64/16 = 4$ 個の反復のループを含めることができ、ループの各反復には 8 つの MAC 演算が含まれます。ループの反復ごとに 16 個の出力データが生成されます。これを次の図に示します。

図 32: ベクター化



使用される mac16() 組み込み関数には、次のインターフェイスがあります。

```

v16acc48 mac16( v16acc48 acc,
                v32int16 xbuff,
                int xstart,
                unsigned int xoffsets,
                unsigned int xoffsets_hi,
                unsigned int xsquare,
                v16int16 zbuff,

```

```
int zstart,
unsigned int zoffsets,
unsigned int zoffsets_hi,
int zstep
)
```

バッファには、バッファ (ベクター レジスタ) へのインデックスを計算するためのパラメーター (start、offsets、square、step) が含まれます。これらのパラメーターを使用したレーン アドレス指定方法の詳細は、[MAC 組み込み関数](#) を参照してください。

MAC 組み込み関数を使用したコード記述方法については、次のセクションを参照してください。

組み込み関数を使用したコード記述

関数が AI エンジン ベクター プロセッサにどのようにマップされるかを解析しました。ベクター化されたコードの最初のバージョンを見てみます。

```
inline void mac16_sub(input_window_int16* matA, v16int16 &buf_matB,
v16acc48 &acc, int i){
    v32int16 buf_matA = undef_v32int16(); // holds 32 elements of matA
    buf_matA=upd_w(buf_matA, 0, window_read_v16(matA));
    window_incr(matA,64);
    buf_matA = upd_w(buf_matA, 1, window_read_v16(matA));
    window_incr(matA,64);
    acc =
mac16(acc,buf_matA,0,0x73727170,0x77767574,0x3120,buf_matB,i,0x0,0x0,1);
}

void matmul_vec16(input_window_int16* matA,
input_window_int16* matB,
output_window_int16* matC){

    v16int16 buf_matB = window_read_v16(matB); // holds 16 elements of matB
    v16acc48 acc = null_v16acc48(); // holds acc value of Row * column dot
    product

    for (unsigned int i=0;i<M/16;i++) //M=64, Each iteration computes 16
    outputs
    {
        acc=null_v16acc48();
        for(int j=0;j<16;j+=2){
            mac16_sub(matA,buf_matB,acc,j);
        }
        window_writeincr(matC,srs(acc,15));
        window_incr(matA,16);
    }
}
```

メイン関数 `matmul_vec16` では、ループの反復ごとに 16 個の出力データが生成されます。外側ループ本体には、8 反復の内側ループが含まれています。内側ループの各反復で、インライン関数 `mac16_sub` が呼び出されます。インライン関数には、MAC 演算に 2 つのデータ ロードを使用する `mac16` 演算があります。

`mac16_sub()` 内では `buf_matA` がローカル変数として宣言され、メイン関数では `buf_matB` と `acc` がローカル変数として宣言されています。これらは、関数間で参照 (またはポインタ) を使用して渡されます。これにより、各変数に存在する同一ベクターは 1 つのみになります。関数には、`mac16()` 組み込み関数で次のように使用される 1 つのパラメーターがあります。この組み込み関数 (`i=0`) は、[MAC 組み込み関数](#) で説明されています。

```
acc =
mac16(acc,buf_matA,0,0x73727170,0x77767574,0x3120,buf_matB,i,0x0,0x0,1);
```

ループの各反復の最後に、データのウィンドウポインターが 16 ずつ (行列の 16 行) インクリメントされます。

注記: この例では、関数の境界を強制的に削除するために `inline` が使用されていますが、`__attribute__((noinline))` `void func(...)` を使用して関数の境界を保持すると便利な場合があります。

コンパイルされたカーネルのコードは、Vitis™ IDE の [Debug] パースペクティブの [Disassembly] ビューに表示されます。AI エンジン ツールを使用してカーネルをコンパイルするには、グラフが必要です。[Disassembly] ビューでのアセンブリコードの詳細は、[Vitis IDE およびレポートの使用](#) を参照してください。グラフのコーディングおよび Vitis IDE の使用方法の詳細は、『Versal ACAP AI エンジン プログラミング環境ユーザーガイド』 ([UG1076](#)) を参照してください。

図 33: ループのアセンブリコード



コンパイラは、内側ループを自動的に展開し、外側ループをパイプライン処理します。上記のループのアセンブリコードから、各反復に 19 サイクル必要なことがわかります。ただし、1 つのデータ (matA) のウィンドウインターフェイスでは、8 つの MAC に必要な最小サイクル数は 16 (MAC ごとに 2 つのデータロード) のはずです。このパフォーマンスの低下は、ループの最後のウィンドウポインターインクリメントのバランスが取れていないことが原因です。最後のインクリメントを最後の MAC 演算とペアにすると、これを解決できます。最適化されたコードは、次のとおりです。

```
inline void mac16_sub(input_window_int16* matA, v16int16 &buf_matB,
v16acc48 &acc, int i,int incr_num){
    v32int16 buf_matA = undef_v32int16(); // holds 32 elements of matA
    buf_matA=upd_w(buf_matA, 0, window_read_v16(matA));
    window_incr(matA,64);
    buf_matA = upd_w(buf_matA, 1, window_read_v16(matA));
    window_incr(matA,incr_num);
    acc =
mac16(acc,buf_matA,0,0x73727170,0x77767574,0x3120,buf_matB,i,0x0,0x0,1);
}

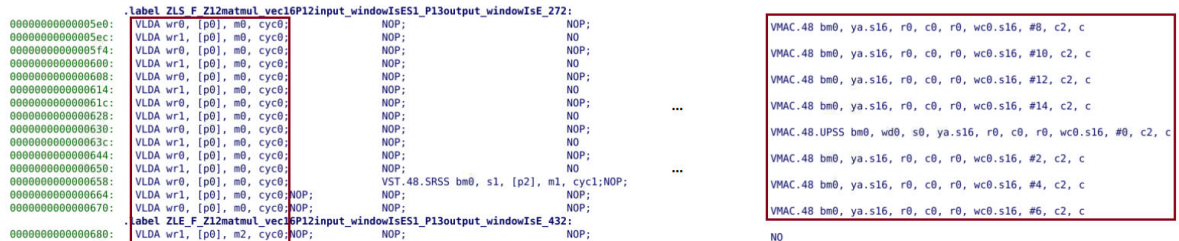
void matmul_vec16(input_window_int16* matA,
    input_window_int16* matB,
    output_window_int16* matC){

    v16int16 buf_matB = window_read_v16(matB); // holds 16 elements of matB
    v16acc48 acc = null_v16acc48(); // holds acc value of Row * column dot
    product

    for (unsigned int i=0;i<M/16;i++) //M=64, Each iteration computes 16
    outputs
    {
        acc=null_v16acc48();
        for(int j=0;j<16;j+=2){
            int incr_num=(j==14)?80:64;
            mac16_sub(matA,buf_matB,acc,j,incr_num);
        }
        window_writeincr(matC,srs(acc,15));
    }
}
```


関数 `mac16_sub` に新しいパラメーター `incr_num` があることに注意してください。このパラメーターはポインターインクリメントのためのもので、内側ループの最後の関数呼び出しのものとは異なります。最後の関数呼び出しのこのインクリメント値 80 は、外側ループの次の反復で次の 16 行のデータが選択されるようにするためのものです。次の図に、ループのアセンブリ コードを示します。

図 34: 最適化されたループのアセンブリ コード



ループの反復には、16 サイクル必要です。つまり、このカーネルの計算バウンドは、呼び出しごとに $16 \times 4 = 64$ サイクルだということです。前のセクションで示したように、呼び出しごとの理論的制限は 32 サイクルです。ループの反復は 8 サイクルなので、8 つの MAC 演算は 8 サイクルに削減されます。これは、システムのパフォーマンス要件によって、データ入力列を列ごとに 2 つのウィンドウ バッファ `matA_0` および `matA_1` に分割することにより達成できます。2 つのウィンドウのデータは、まず 2 つの `v16int16` ベクターに読み出され、1 つの `v32int16` ベクターに連結されて `mac16` 組み込み関数で使用されます。カーネルのコードは次のとおりです。

```
inline void mac16_sub_loads(input_window_int16* matA_0, input_window_int16*
matA_1, v16int16 &buf_matB, v16acc48 &acc, int i, int incr_num){
    v16int16 buf_matA0 = window_read_v16(matA_0);
    window_incr(matA_0, incr_num);
    v16int16 buf_matA1 = window_read_v16(matA_1);
    window_incr(matA_1, incr_num);
    acc =
mac16(acc, concat(buf_matA0, buf_matA1), 0, 0x73727170, 0x77767574, 0x3120, buf_mat
B, i, 0x0, 0x0, 1);
}

void matmul_vec16(input_window_int16* __restrict matA_0,
input_window_int16* __restrict matA_1,
input_window_int16* __restrict matB,
output_window_int16* __restrict matC){
    v16int16 buf_matB = window_read_v16(matB);
    for (unsigned int i=0; i<M/16; i++) //M=64, Each iteration computes 16
outputs
    chess_prepare_for_pipelining
    {
        v16acc48 acc=null_v16acc48();
        for(int j=0; j<16; j+=2){
            int incr_num=(j==14)?80:64;
            mac16_sub_loads(matA_0, matA_1, buf_matB, acc, j, incr_num);
        }
        window_writeincr(matC, srs(acc, 15));
    }
}
```

2 つの `v16int16` ベクター `buf_matA0` および `buf_matA1` は、`mac16` 組み込み関数用に定義され、連結されます。また、ループがパイプライン処理されてウィンドウ演算が最適化されたものになるようにするため、ループに `chess_prepare_for_pipelining`、ウィンドウ インターフェイスに `__restrict` キーワードが追加されています。



重要: `--restrict` キーワードの使用には注意が必要です。使用前に『[Vitis 統合ソフトウェア プラットフォームの資料](#)』(UG1416)のAIエンジンフローの[AIエンジンでの restrict キーワードの使用](#)を参照してください。

1 サイクルごとに 2 つのウィンドウ ロードのバージョンのアセンブリ コードは、次のとおりです。

図 35: 1 サイクルごとに 2 つのウィンドウ ロードのアセンブリ コード

Address	Disassembly	Comment
0000000000000072	label ZL5_F_Zl2matmul_veci6P12input_windowESI_S1_P13tputut_windowESI_384:	
0000000000000073	VLDIA w.r1, [p1], m0, c0;c0:VLDIB w.r0, [p0], m0, c0;c2 NOP;	NOP; VMAC-48 bmb, ya.s16, r0, c0, r0
0000000000000074	VLDIA w.r1, [p1], m0, c0;c0:VLDIB w.r0, [p0], m0, c0;c2 NOP;	NOP; VMAC-48 bmb, ya.s16, r0, c0, r0
0000000000000075	VLDIA w.r1, [p1], m0, c0;c0:VLDIB w.r0, [p0], m0, c0;c2 NOP;	NOP; VMAC-48 bmb, ya.s16, r0, c0, r0
0000000000000076	VLDIA w.r1, [p1], m0, c0;c0:VLDIB w.r0, [p0], m0, c0;c2 NOP;	NOP; VMAC-48 bmb, ya.s16, r0, c0, r0
0000000000000077	VLDIA w.r1, [p1], m0, c0;c0:VLDIB w.r0, [p0], m0, c0;c2 NOP;	NOP; VMAC-48 bmb, ya.s16, r0, c0, r0
0000000000000078	VLDIA w.r1, [p1], m0, c0;c0:VLDIB w.r0, [p0], m0, c0;c2 NOP;	NOP; VMAC-48 bmb, ya.s16, r0, c0, r0
0000000000000079	label ZL6_F_Zl2matmul_veci6P12input_windowESI_S1_P13tputut_windowESI_496:	
0000000000000080	VLDIA w.r1, [p1], m1, c0;c0:VLDIB w.r0, [p0], m1, c0;c2 NOP;	NOP; VMAC-48 UPSS bmb, w.d0, s0, ya.s16

行列乗算

ここで示す行列乗算の例は、次の式をインプリメントします。

$$C (64 \times 2) = A (64 \times 8) * B(8 \times 2)$$

この例では、行列のデータは列優先形式で格納されると想定され、行列 A および B のデータ型は int16 です。

最初の出力列は、次のように計算されます。

```

c0 = a0*b0 + a64*b1 + a128*b2 + a192*b3 + a256*b4 + a320*b5 + a384*b6 +
a448*b7
c1 = a1*b0 + a65*b1 + a129*b2 + a193*b3 + a257*b4 + a321*b5 + a385*b6 +
a449*b7
c2 = a2*b0 + a66*b1 + a130*b2 + a194*b3 + a258*b4 + a322*b5 + a386*b6 +
a450*b7
c3 = a3*b0 + a67*b1 + a131*b2 + a195*b3 + a259*b4 + a323*b5 + a387*b6 +
a451*b7
...
c60 = a60*b0 + a124*b1 + a188*b2 + a252*b3 + a316*b4 + a380*b5 + a444*b6 +
a508*b7
c61 = a61*b0 + a125*b1 + a189*b2 + a253*b3 + a317*b4 + a381*b5 + a445*b6 +
a509*b7
c62 = a62*b0 + a126*b1 + a190*b2 + a254*b3 + a318*b4 + a382*b5 + a446*b6 +
a510*b7
c63 = a63*b0 + a127*b1 + a191*b2 + a255*b3 + a319*b4 + a383*b5 + a447*b6 +
a511*b7

```

2 番目の出力列は、次のように計算されます。

```

c64 = a0*b8 + a64*b9 + a128*b10 + a192*b11 + a256*b12 + a320*b13 + a384*b14
+ a448*b15
c65 = a1*b8 + a65*b9 + a129*b10 + a193*b11 + a257*b12 + a321*b13 + a385*b14
+ a449*b15
c66 = a2*b8 + a66*b9 + a130*b10 + a194*b11 + a258*b12 + a322*b13 + a386*b14
+ a450*b15
c67 = a3*b8 + a67*b9 + a131*b10 + a195*b11 + a259*b12 + a323*b13 + a387*b14
+ a451*b15
...
c124 = a60*b8 + a124*b9 + a188*b10 + a252*b11 + a316*b12 + a380*b13 +
a444*b14 + a508*b15
c125 = a61*b8 + a125*b9 + a189*b10 + a253*b11 + a317*b12 + a381*b13 +

```



```
a445*b14 + a509*b15
c126 = a62*b8 + a126*b9 + a190*b10 + a254*b11 + a318*b12 + a382*b13 +
a446*b14 + a510*b15
c127 = a63*b8 + a127*b9 + a191*b10 + a255*b11 + a319*b12 + a383*b13 +
a447*b14 + a511*b15
```

カーネル コーディングの限界

この例では、128 の出力値を計算するために合計 1024 の int16 x int16 の乗算が必要です。AI エンジンでは、32 個の 16 ビット乗算をサイクルごとに実行できるため、カーネルの計算限界は次のようになります。

```
Compute bound = 32 cycles / invocation
```

行列 B は 16*16 ビット = 256 ビットなので、ベクター レジスタに格納できます。MAC 演算ごとに AI エンジンのデータ メモリまたはタイル インターフェイスからフェッチする必要はありません。計算にデータ a が必要だとすると、メモリからフェッチされる合計は 64*8*2=1024 バイトです。AI エンジンでは、サイクルごとに 2 つの 256 ビット (32 バイト) のロードができるので、カーネルのメモリ限界は次のようになります。

```
Memory bound = 1024 / (2*32) = 16 cycles / invocation
```

計算限界は、メモリ限界よりも大きいことがわかります。このため、ベクター プロセッサでの MAC 演算の理論上の制限に到達することをベクター化の目的とすることができます。

ベクター化

この行列乗算の例のスカラー参照コードは、次のようになります。データは列に格納されます。

```
void matmul_mat8_scalar(input_window_int16* matA,
    input_window_int16* matB,
    output_window_int16* matC){

    for(int i=0; i<M; i++){//M=64
        for(int j=0; j<L; j++){//L=2
            int temp = 0 ;
            for(int k=0; k<N; k++){//N=8
                temp += window_read(matA)*window_readincr(matB);//B is
circular buffer, size N*L
                window_incr(matA,64); //Jump of 64 elements to access the
next element of the same row
            }
            window_write(matC,(int16_t)(temp>>15)) ;
            window_incr(matC,64); //Jump to the next column
        }
        window_incr(matA,1); //Jump of one element for moving to the next
row.
        window_incr(matC,1); //Jump to the next row
    }
}
```

前の例 (行列ベクター乗算) で解析したように、列から 16 int16 を一度にロードできるので、16 レーンを同時に計算するには `mac16` 組み込み関数が最適です。1 つの列で 16 の出力データを計算するには、`mac16` 演算が 4 回必要です。ベクター `a` 内の同じデータは、2 つの出力列のデータを計算するために 2 回使用されます。このため、2 つのデータ列をロードして、2 つの `mac16` を 2 つの出力列に累積するために使用できます。これら 2 つのロードと 2 つの MAC が 4 回繰り返されて、2 つの出力列になります。この方法を次の疑似コードに示します。

```
C_[0:15,0] = A_[0:15,0:1]*B_[0:1,0]
C_[0:15,1] = A_[0:15,0:1]*B_[0:1,1]

C_[0:15,0] += A_[0:15,2:3]*B_[2:3,0]
C_[0:15,1] += A_[0:15,2:3]*B_[2:3,1]

C_[0:15,0] += A_[0:15,4:5]*B_[4:5,0]
C_[0:15,1] += A_[0:15,4:5]*B_[4:5,1]

C_[0:15,0] += A_[0:15,6:7]*B_[6:7,0]
C_[0:15,1] += A_[0:15,6:7]*B_[6:7,1]
```

前のコードでは、`*` がそれぞれ MAC 演算を示します。`C_[0:15,0]` および `C_[0:15,1]` は、別々累算される 2 つの出力列を示します。`A_[0:15,0:1]` は列 0 と 1 を示し、各列には 16 の要素があります。`B_[0:1,0]` は、2 つの要素を持つ列 0 を示します。64 行の出力があるため、実際にベクター化されたコードにはループがあります。使用される `mac16` 組み込み関数には、次のインターフェイスがあります。

```
v16acc48 mac16 ( v16acc48 acc,
v64int16 xbuff,
int xstart,
unsigned int xoffsets,
unsigned int xoffsets_hi,
unsigned int xsquare,
v16int16 zbuff,
int zstart,
unsigned int zoffsets,
unsigned int zoffsets_hi,
int zstep
)
```

バッファには、バッファ (ベクター レジスタ) へのインデックスを計算するためのパラメーター (`start`, `offsets`, `square`, `step`) が含まれます。これらのパラメーターを使用したレーン アドレス指定方法の詳細は、[MAC 組み込み関数](#) を参照してください。

`mac16` 組み込み関数のプロトタイプは、前の行列ベクター乗算の例で紹介したプロトタイプとは異なることに注意してください。この場合、`xbuff` は `v64int16` で、2 セットのデータをインターリーブ方式で格納および使用できます。

MAC 組み込み関数を使用したコード記述方法については、次のセクションを参照してください。

組み込み関数を使用したコード記述

関数が AI エンジン ベクター プロセッサにどのようにマップされるかを解析しました。ベクター化されたコードを見てください。

```
void matmul_mat8(input_window_int16* matA,
input_window_int16* matB,
output_window_int16* matC){

v16int16 buf_matB = window_read_v16(matB);

v64int16 buf_matA = undef_v64int16();
```

```

    buf_matA=upd_w(buf_matA,0>window_read_v16(matA));
    window_incr(matA,64);
    buf_matA=upd_w(buf_matA,1>window_read_v16(matA));
    window_incr(matA,64);

    for (unsigned int i=0;i<M/16;i++) //M=64, Each iteration computes 16
outputs
    chess_prepare_for_pipelining
    chess_loop_range(4,)
    {
        v16acc48 acc0=null_v16acc48();//For first output column
        v16acc48 acc1=null_v16acc48();//For second output column

        acc0 =
mac16(acc0,buf_matA,0,0x73727170,0x77767574,0x3120,buf_matB,0,0x0,0x0,1);
        buf_matA=upd_w(buf_matA,2>window_read_v16(matA));
        window_incr(matA,64);
        acc1 =
mac16(acc1,buf_matA,0,0x73727170,0x77767574,0x3120,buf_matB,8,0x0,0x0,1);
        buf_matA=upd_w(buf_matA,3>window_read_v16(matA));
        window_incr(matA,64);

        acc0 =
mac16(acc0,buf_matA,32,0x73727170,0x77767574,0x3120,buf_matB,2,0x0,0x0,1);
        buf_matA=upd_w(buf_matA,0>window_read_v16(matA));
        window_incr(matA,64);
        acc1 =
mac16(acc1,buf_matA,32,0x73727170,0x77767574,0x3120,buf_matB,10,0x0,0x0,1);
        buf_matA=upd_w(buf_matA,1>window_read_v16(matA));
        window_incr(matA,64);

        acc0 =
mac16(acc0,buf_matA,0,0x73727170,0x77767574,0x3120,buf_matB,4,0x0,0x0,1);
        buf_matA=upd_w(buf_matA,2>window_read_v16(matA));
        window_incr(matA,64);
        acc1 =
mac16(acc1,buf_matA,0,0x73727170,0x77767574,0x3120,buf_matB,12,0x0,0x0,1);
        buf_matA=upd_w(buf_matA,3>window_read_v16(matA));
        window_incr(matA,80);//point to next 16 rows

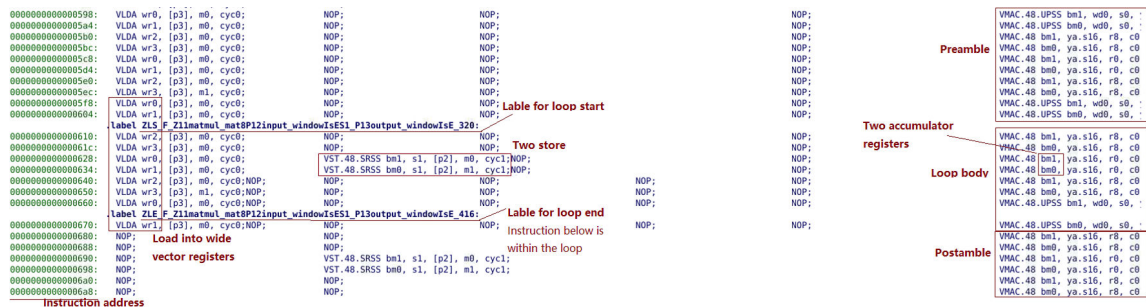
        acc0 =
mac16(acc0,buf_matA,32,0x73727170,0x77767574,0x3120,buf_matB,6,0x0,0x0,1);
        window_write(matC,srs(acc0,15));
        window_incr(matC,64);
        buf_matA=upd_w(buf_matA,0>window_read_v16(matA));
        window_incr(matA,64);
        acc1 =
mac16(acc1,buf_matA,32,0x73727170,0x77767574,0x3120,buf_matB,14,0x0,0x0,1);
        window_write(matC,srs(acc1,15));
        window_incr(matC,80);//point to next 16 rows
        buf_matA=upd_w(buf_matA,1>window_read_v16(matA));
        window_incr(matA,64);
    }
}

```

上記のコードでは、buf_matB は行列 B 用で、ループの外でロードされます。buf_matA は行列 A 用で、下位部分と上位部分に 2 つのセットが格納されます。mac16 の xstart の値が 0 の場合、buf_matA の下位部分が使用されます。mac16 の xstart の値が 32 の場合、buf_matA の上位部分が使用されます。acc0 と acc1 は、2 つの出力列の累積値です。

buf_matA は、ループの前にロードされることに注意してください。ループでは、ウィンドウ バッファ ポインターのインクリメントのロード、MAC 演算、ストアが順次実行されます。mac16() 組み込み関数の動作を理解するには、[MAC 組み込み関数](#) を参照してください。次の図に、ループのアセンブリ コードを示します。

図 36: ループのアセンブリ コード



上記のアセンブリ コードから、ループの各サイクルに MAC 演算とロード操作があるように見えます。幅の広いレジスタ wr0、wr1、wr2、および wr3 は、buf_matA に使用されます。アキュムレータ レジスタ bm0 および bm1 は、2 つの累積結果に使用されます。

ループが最大限にパイプライン処理されるようにするための推奨事項は、次のとおりです。

- ループの開始前にデータをベクター レジスタにロードします。
- ループ本体でデータ ロード、MAC 演算、データ ストアを順次実行します。
- 幅の広い入力データ ベクター レジスタ (例: v64int16) を使用し、データ ロードと MAC 演算をベクター レジスタの異なる部分で実行します。
- 複数のアキュムレータ レジスタを使用し、入力データを複数の出力に再利用します。
- データ ロードとバッファ ポインター インクリメントをペアで供給します。これは、データ ストアおよびバッファ ポインター インクリメントにも適用されます。

複数カーネルのコーディング例: FIR フィルター

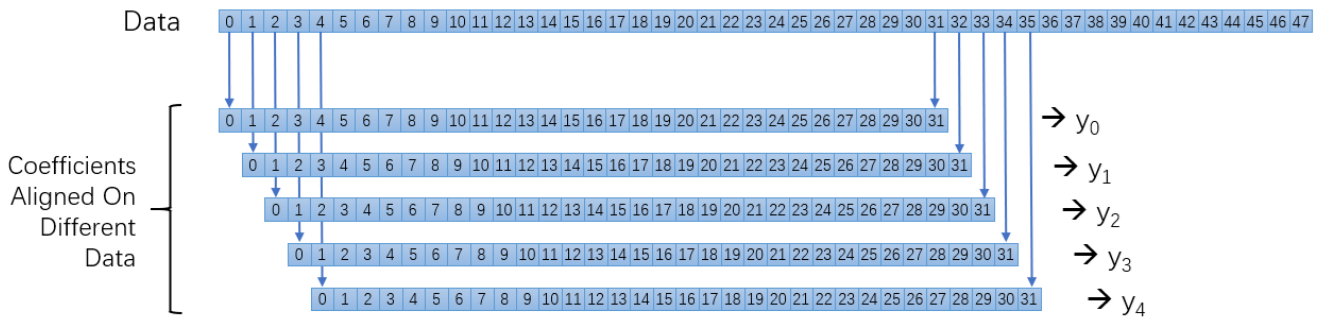
このセクションでは、フィルタ デザインを使用して、アプリケーションが 1 つの AI エンジンの計算能力を超えた場合にアプリケーションを複数の AI エンジンに分割する方法を説明します。有限インパルス応答 (FIR) フィルターは、インパルス応答 (または有限長入力に対する応答) が有限期間のフィルターです。

$$y_n = \sum_{k=0}^{N-1} C_k x_{n+k}$$

注記: 数学的には、この方程式はたたみ込みではなく相関式です。AI エンジン内での計算はこのように分類されます。これをたたみ込み (FIR フィルタリング) にするには、係数を逆の順序で C_k ベクターに格納する必要があります。これは、対称フィルタの場合は問題ではありません。

前の式では、N は各出力の計算に使用されるタップを表します。次の図に、32 タップ フィルターを例として使用した場合の計算プロセスを示します。データと係数には、例として int16 複素数型を使用しています。

図 37: FIR フィルター



カスケード ストリームを使用した 1 Gsps インプリメンテーション

AI エンジン ベクター ユニットでは、cint16 積和 cint16 型で 1 サイクルごとに 8 つの MAC がサポートされます。mul4/mac4 組み込み関数の 4 レーン インプリメンテーションを使用する場合、各レーンに 2 つの複素演算があります。

$$\begin{cases} y_0 = c_0 \cdot d_0 + c_1 \cdot d_1 \\ y_1 = c_0 \cdot d_1 + c_1 \cdot d_2 \\ y_2 = c_0 \cdot d_2 + c_1 \cdot d_3 \\ y_3 = c_0 \cdot d_3 + c_1 \cdot d_4 \end{cases}$$

各出力に 32 個の複素 MAC が必要であるため、4 つの出力を計算するため 16 mac4() が必要です。これは、4 つの出力を計算するのに、AI エンジンを使用して 16 サイクル必要であるということです。AI エンジンが 1 GHz で動作する場合、サンプル レートは次のようになります。

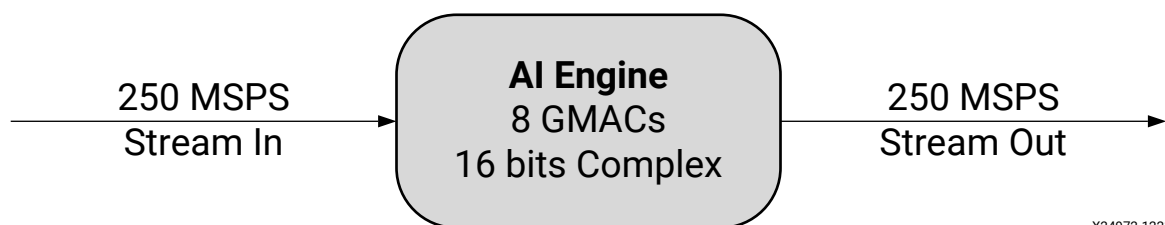
$$4 \text{ Gsps} / 16 = 0.25 \text{ Gsps} = 250 \text{ Msps}$$

これにより、AI エンジンの計算バウンドが計算されます。サンプル レートが満たされるかどうかを判断するには、メモリ バウンドも考慮する必要があります。データ転送に 1 つのストリーム入力と 1 つのストリーム出力のみが使用され、係数が AI エンジンの内部メモリに格納されているとします。AI エンジンのストリーム インターフェイスでは、1 サイクルごとに 32 ビットがサポートされます。各サイクルでデータの 1 サンプルを転送できます。データ転送の面から見ると、サンプル データは次のようになります。

$$1 \text{ サンプル/サイクル} \times 1 \text{ GHz} = 1 \text{ Gsps}$$

これは、250 Msps の計算バインドよりも大きいです。そのため、AI エンジン インプリメンテーションは 250 Msps で動作します。

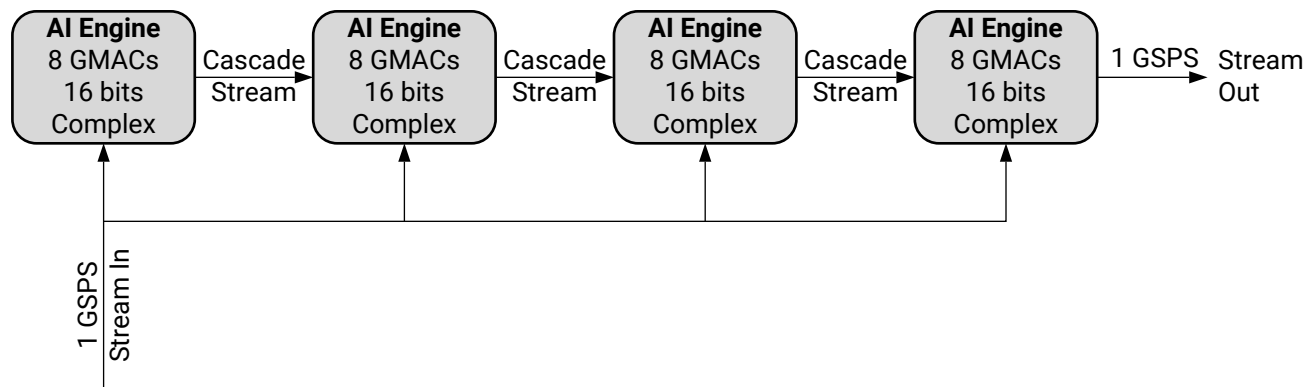
図 38: 1 つの AI エンジン FIR フィルターの実現



X24972-122120

計算に基づけば、ストリーム入力と出力ストリーム インターフェイスで 1 Gsps を達成可能です。シングル カーネル インプリメンテーションの MAC 演算を 4 つのカーネルに分割すると、 $4 \times 250 \text{ Msps} = 1 \text{ GSP}$ の計算スループットを達成できます。これらの 4 つのカーネルは、カスケード ストリーミングにより接続されます。そのため、AI エンジンの計算バウンドが AI エンジン インターフェイスのスループットに一致します。

図 39: カスケード接続した 4 つのカーネルを使用した 1 Gsps インプリメンテーション

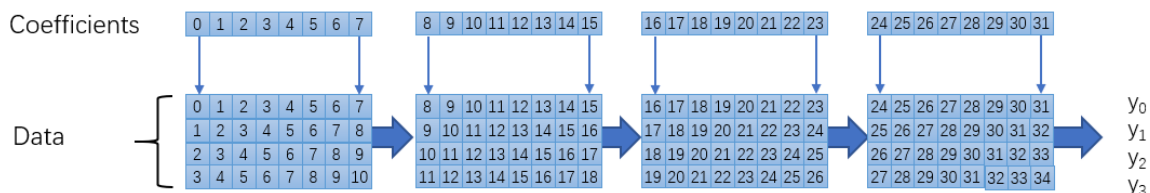


X24973-122120

組み込み関数を使用したコード記述

1 GSP インプリメンテーションの 4 つのカーネルにそれぞれ異なる係数のセットを使用し、その間でストリームをカスケード接続できます。このインプリメンテーションを次の図に示します。

図 40: 分割係数とカスケード ストリームを使用した 4 つのカーネル



入力データは、ストリームからこれらの 4 つのカーネルに流れます。ただし、2 番目のカーネルでは、最初の 8 つの入力データが破棄されます。3 番目のカーネルでは、最初の 16 個の入力データが破棄されます。同様に、4 番目のカーネルでは、最初の 24 個の入力データが破棄されます。

最初のカーネルのコードは、次のとおりです。

```
#include <adf.h>
#include "fir_32tap.h"
// buffer to keep state
static v16cint16 delay_line;

void fir_32tap_core0(
    input_stream_cint16 * sig_in,
    output_stream_cacc48 * cascadeout)
{
    const cint16_t * __restrict coeff = eq_coef0;
    const v8cint16 *coef_ = (v8cint16 const*)coeff;
    const v8cint16 coe = *coef_;
```

```

v16cint16 buff = delay_line;
v4cacc48 acc;
const unsigned LSIZE = (samples/4/4); // assuming samples is integer
power of 2 and greater than 16

for (unsigned int i = 0; i < LSIZE; ++i)
chess_prepare_for_pipelining
chess_loop_range(4,)
{
    acc = mul4(buff, 0 , 0x3210, 1, coe, 0, 0x0000, 1);
    acc = mac4(acc, buff, 2 , 0x3210, 1, coe, 2, 0x0000, 1);
    buff = upd_v(buff, 2, readincr_v4(sig_in));
    acc = mac4(acc, buff, 4 , 0x3210, 1, coe, 4, 0x0000, 1);
    acc = mac4(acc, buff, 6 , 0x3210, 1, coe, 6, 0x0000, 1);
    writeincr_v4(cascadeout, acc);

    acc = mul4(buff, 4 , 0x3210, 1, coe, 0, 0x0000, 1);
    acc = mac4(acc, buff, 6 , 0x3210, 1, coe, 2, 0x0000, 1);
    buff = upd_v(buff, 3, readincr_v4(sig_in));
    acc = mac4(acc, buff, 8 , 0x3210, 1, coe, 4, 0x0000, 1);
    acc = mac4(acc, buff, 10, 0x3210, 1, coe, 6, 0x0000, 1);
    writeincr_v4(cascadeout, acc);

    acc = mul4(buff, 8 , 0x3210, 1, coe, 0, 0x0000, 1);
    acc = mac4(acc, buff, 10, 0x3210, 1, coe, 2, 0x0000, 1);
    buff = upd_v(buff, 0, readincr_v4(sig_in));
    acc = mac4(acc, buff, 12, 0x3210, 1, coe, 4, 0x0000, 1);
    acc = mac4(acc, buff, 14, 0x3210, 1, coe, 6, 0x0000, 1);
    writeincr_v4(cascadeout, acc);

    acc = mul4(buff, 12, 0x3210, 1, coe, 0, 0x0000, 1);
    acc = mac4(acc, buff, 14, 0x3210, 1, coe, 2, 0x0000, 1);
    buff = upd_v(buff, 1, readincr_v4(sig_in));
    acc = mac4(acc, buff, 0 , 0x3210, 1, coe, 4, 0x0000, 1);
    acc = mac4(acc, buff, 2 , 0x3210, 1, coe, 6, 0x0000, 1);
    writeincr_v4(cascadeout, acc);
}
delay_line = buff;
}

void fir_32tap_core0_init()
{
    // Drop samples if not first block
    int const Delay = 0;
    for (int i = 0; i < Delay; ++i)
    {
        get_ss(0);
    }
};

```

関数 `fir_32tap_core0_init` は、AI エンジン カーネル `fir_32tap_core0` の初期化関数となり、カーネルの起動時に一度だけ実行されることに注意してください。この初期化関数の目的は、入力ストリームを揃えるために不要なサンプルを破棄することです。

同様に、関数 `fir_32tap_core1_init` は AI エンジン カーネル `fir_32tap_core1` の初期化関数となり、コードは次のとおりです。初期化関数 `fir_32tap_core2_init` および `fir_32tap_core3_init` も同様です。

2 番目のカーネルのコードは次のとおりです。

```
#include <adf.h>
#include "fir_32tap.h"
// buffer to keep state
static v16cint16 delay_line;

void fir_32tap_core1(
    input_stream_cint16 * sig_in,
    input_stream_cacc48 * cascadein,
    output_stream_cacc48 * cascadeout)
{
    const cint16_t * __restrict coeff = eq_coef1;
    const v8cint16 *coef_ = (v8cint16 const*)coeff;
    const v8cint16 coe = *coef_;

    v16cint16 buff = delay_line;
    v4cacc48 acc;
    const unsigned LSIZE = (samples/4/4); // assuming samples is integer
    power of 2 and greater than 16

    for (unsigned int i = 0; i < LSIZE; ++i)
        chess_prepare_for_pipelining
        chess_loop_range(4,)
        {
            acc = readincr_v4(cascadein);
            acc = mac4(acc, buff, 0 , 0x3210, 1, coe, 0, 0x0000, 1);
            acc = mac4(acc, buff, 2 , 0x3210, 1, coe, 2, 0x0000, 1);
            buff = upd_v(buff, 2, readincr_v4(sig_in));
            acc = mac4(acc, buff, 4 , 0x3210, 1, coe, 4, 0x0000, 1);
            acc = mac4(acc, buff, 6 , 0x3210, 1, coe, 6, 0x0000, 1);
            writeincr_v4(cascadeout, acc);

            acc = readincr_v4(cascadein);
            acc = mac4(acc, buff, 4 , 0x3210, 1, coe, 0, 0x0000, 1);
            acc = mac4(acc, buff, 6 , 0x3210, 1, coe, 2, 0x0000, 1);
            buff = upd_v(buff, 3, readincr_v4(sig_in));
            acc = mac4(acc, buff, 8 , 0x3210, 1, coe, 4, 0x0000, 1);
            acc = mac4(acc, buff, 10, 0x3210, 1, coe, 6, 0x0000, 1);
            writeincr_v4(cascadeout, acc);

            acc = readincr_v4(cascadein);
            acc = mac4(acc, buff, 8 , 0x3210, 1, coe, 0, 0x0000, 1);
            acc = mac4(acc, buff, 10 , 0x3210, 1, coe, 2, 0x0000, 1);
            buff = upd_v(buff, 0, readincr_v4(sig_in));
            acc = mac4(acc, buff, 12 , 0x3210, 1, coe, 4, 0x0000, 1);
            acc = mac4(acc, buff, 14 , 0x3210, 1, coe, 6, 0x0000, 1);
            writeincr_v4(cascadeout, acc);

            acc = readincr_v4(cascadein);
            acc = mac4(acc, buff, 12 , 0x3210, 1, coe, 0, 0x0000, 1);
            acc = mac4(acc, buff, 14 , 0x3210, 1, coe, 2, 0x0000, 1);
            buff = upd_v(buff, 1, readincr_v4(sig_in));
            acc = mac4(acc, buff, 0 , 0x3210, 1, coe, 4, 0x0000, 1);
            acc = mac4(acc, buff, 2 , 0x3210, 1, coe, 6, 0x0000, 1);
            writeincr_v4(cascadeout, acc);
        }
    delay_line = buff;
}

void fir_32tap_core1_init()
{
    // Drop samples if not first block
}
```



```
int const Delay = 8;
for (int i = 0; i < Delay; ++i)
{
    get_ss(0);
}
};
```

3 番目のカーネルのコードは、2 番目のカーネルのコードと同様です。最後のカーネルのコードは次のとおりです。

```
#include <adf.h>
#include "fir_32tap.h"
// buffer to keep state
static v16cint16 delay_line;

void fir_32tap_core3(
    input_stream_cint16 * sig_in,
    input_stream_cacc48 * cascadein,
    output_stream_cint16 * data_out)
{
    const cint16_t * __restrict coeff = eq_coef3;
    const v8cint16 *coef_ = (v8cint16 const*)coeff;
    const v8cint16 coe = *coef_;

    v16cint16 buff = delay_line;

    v4cacc48 acc;

    set_rnd(2);
    set_sat();
    const unsigned LSIZE = (samples/4/4); // assuming samples is integer
    power of 2 and greater than 16

    for (unsigned int i = 0; i < LSIZE; ++i)
    chess_prepare_for_pipelining
    chess_loop_range(4,)
    {
        acc = readincr_v4(cascadein);
        acc = mac4(acc, buff, 0 , 0x3210, 1, coe, 0, 0x0000, 1);
        acc = mac4(acc, buff, 2 , 0x3210, 1, coe, 2, 0x0000, 1);
        buff = upd_v(buff, 2, readincr_v4(sig_in));
        acc = mac4(acc, buff, 4 , 0x3210, 1, coe, 4, 0x0000, 1);
        acc = mac4(acc, buff, 6 , 0x3210, 1, coe, 6, 0x0000, 1);
        writeincr_v4(data_out, srs(acc, shift));

        acc = readincr_v4(cascadein);
        acc = mac4(acc, buff, 4 , 0x3210, 1, coe, 0, 0x0000, 1);
        acc = mac4(acc, buff, 6 , 0x3210, 1, coe, 2, 0x0000, 1);
        buff = upd_v(buff, 3, readincr_v4(sig_in));
        acc = mac4(acc, buff, 8 , 0x3210, 1, coe, 4, 0x0000, 1);
        acc = mac4(acc, buff, 10, 0x3210, 1, coe, 6, 0x0000, 1);
        writeincr_v4(data_out, srs(acc, shift));

        acc = readincr_v4(cascadein);
        acc = mac4(acc, buff, 8 , 0x3210, 1, coe, 0, 0x0000, 1);
        acc = mac4(acc, buff, 10, 0x3210, 1, coe, 2, 0x0000, 1);
        buff = upd_v(buff, 0, readincr_v4(sig_in));
        acc = mac4(acc, buff, 12 , 0x3210, 1, coe, 4, 0x0000, 1);
        acc = mac4(acc, buff, 14 , 0x3210, 1, coe, 6, 0x0000, 1);
        writeincr_v4(data_out, srs(acc, shift));

        acc = readincr_v4(cascadein);
        acc = mac4(acc, buff, 12 , 0x3210, 1, coe, 0, 0x0000, 1);
        acc = mac4(acc, buff, 14 , 0x3210, 1, coe, 2, 0x0000, 1);
```

```

        buff = upd_v(buff, 1, readincr_v4(sig_in));
        acc = mac4(acc, buff, 0, 0x3210, 1, coe, 4, 0x0000, 1);
        acc = mac4(acc, buff, 2, 0x3210, 1, coe, 6, 0x0000, 1);
        writeincr_v4(data_out, srs(acc, shift));
    }
    delay_line = buff;
}

void fir_32tap_core3_init()
{
    // Drop samples if not first block
    int const Delay = 24;
    for (int i = 0; i < Delay; ++i)
    {
        get_ss(0);
    }
};

```

グラフ コードは次のとおりです。

```

#include <adf.h>
#include "kernels.h"
using namespace adf;
class firGraph : public graph {
public:
    kernel k0,k1,k2,k3;
    input_port in0123;
    output_port out;
    firGraph()
    {
        k0 = kernel::create(fir_32tap_core0);
        runtime<ratio>(k0) = 0.9;
        source(k0) = "fir_32tap_core0.cpp";
        connect<stream> n0(in0123,k0.in[0]);

        k1 = kernel::create(fir_32tap_core1);
        runtime<ratio>(k1) = 0.9;
        source(k1) = "fir_32tap_core1.cpp";
        connect<stream> n1(in0123,k1.in[0]);
        connect<cascade> (k0.out[0],k1.in[1]);

        k2 = kernel::create(fir_32tap_core2);
        runtime<ratio>(k2) = 0.9;
        source(k2) = "fir_32tap_core2.cpp";
        connect<stream> n2(in0123,k2.in[0]);
        connect<cascade> (k1.out[0],k2.in[1]);

        k3 = kernel::create(fir_32tap_core3);
        runtime<ratio>(k3) = 0.9;
        source(k3) = "fir_32tap_core3.cpp";
        connect<stream> n3(in0123,k3.in[0]);
        connect<cascade> (k2.out[0],k3.in[1]);
        connect<stream> (k3.out[0],out);

        initialization_function(k0) = "fir_32tap_core0_init";
        initialization_function(k1) = "fir_32tap_core1_init";
        initialization_function(k2) = "fir_32tap_core2_init";
        initialization_function(k3) = "fir_32tap_core3_init";
    }
};

```

カスケード ストリームで接続されたカーネルは、同期動作します。カスケード ストリームで競合が発生すると、カーネルがストールする可能性があります。カーネル内のループをスムーズに動作させるためには、入力データが使用可能であることが必要です。入力ストリームが各カーネルに適切なときに到着することが重要です。入力ストリームのストールは、AI エンジン カーネルに接続されているネットに十分に大きな FIFO を追加することで解決できます。次に例を示します。

```
fifo_depth(n0)=175;
fifo_depth(n1)=150;
fifo_depth(n2)=125;
fifo_depth(n3)=100;
```

すべてのネットに同じ FIFO 深さを使用すると、自動的に FIFO が結合される可能性があるため、上記の例では異なる FIFO 深さが指定されています。

FIFO リソースを節約するため、各カーネルでイベント `CORE_INSTREAM_WIDE` がいつ発生するかを調べることで、個々の FIFO 深さを設定できます。イベントが早く発生するほど、FIFO を深くする必要があります。次に例を示します。

```
fifo_depth(n0)=45;
fifo_depth(n1)=33;
fifo_depth(n2)=23;
fifo_depth(n3)=10;
```

グラフ コード記述の詳細は、『Versal ACAP AI エンジン プログラミング環境ユーザー ガイド』 ([UG1076](#)) を参照してください。

その他のリソースおよび法的通知

ザイリンクス リソース

アンサー、資料、ダウンロード、フォーラムなどのサポート リソースは、[ザイリンクス サポート](#) サイトを参照してください。

Documentation Navigator およびデザイン ハブ

ザイリンクス Documentation Navigator (DocNav) では、ザイリンクスの資料、ビデオ、サポート リソースにアクセスでき、特定の情報を取得するためにフィルター機能や検索機能を利用できます。DocNav を開くには、次のいずれかを実行します。

- Vivado[®] IDE で [Help] → [Documentation and Tutorials] をクリックします。
- Windows で [スタート] → [すべてのプログラム] → [Xilinx Design Tools] → [DocNav] をクリックします。
- Linux コマンド プロンプトに「docnav」と入力します。

ザイリンクス デザイン ハブには、資料やビデオへのリンクがデザイン タスクおよびトピックごとにまとめられており、これらを参照することでキー コンセプトを学び、よくある質問 (FAQ) を参考に問題を解決できます。デザイン ハブにアクセスするには、次のいずれかを実行します。

- DocNav で [Design Hub View] タブをクリックします。
- ザイリンクス ウェブサイトで[デザイン ハブ](#) ページを参照します。

注記: DocNav の詳細は、ザイリンクス ウェブサイトの [Documentation Navigator](#) ページを参照してください。DocNav からは、日本語版は参照できません。ウェブサイトのデザイン ハブ ページをご利用ください。

参考資料

このガイドの補足情報は、次の資料を参照してください。

1. 『Versal ACAP AI エンジン アーキテクチャ マニュアル』 (AM009: [英語版](#)、[日本語版](#))
2. 『Versal ACAP AI エンジン プログラミング環境ユーザー ガイド』 ([UG1076](#))
3. 『Versal ACAP AI エンジン イントリクション資料』 ([UG1078](#))

お読みください: 重要な法的通知

本通知に基づいて貴殿または貴社 (本通知の被通知者が個人の場合には「貴殿」、法人その他の団体の場合には「貴社」。以下同じ) に開示される情報 (以下「本情報」といいます) は、ザイリンクスの製品を選択および使用することのためにのみ提供されます。適用される法律が許容する最大限の範囲で、(1) 本情報は「現状有姿」、およびすべて受領者の責任で (with all faults) という状態で提供され、ザイリンクスは、本通知をもって、明示、黙示、法定を問わず (商品性、非侵害、特定目的適合性の保証を含みますがこれらに限られません)、すべての保証および条件を負わない (否認する) ものとし、(2) ザイリンクスは、本情報 (貴殿または貴社による本情報の使用を含む) に関係し、起因し、関連する、いかなる種類・性質の損失または損害についても、責任を負わない (契約上、不法行為上 (過失の場合を含む)、その他のいかなる責任の法理によるかを問わない) ものとし、当該損失または損害には、直接、間接、特別、付随的、結果的な損失または損害 (第三者が起こした行為の結果被った、データ、利益、業務上の信用の損失、その他あらゆる種類の損失や損害を含みます) が含まれるものとし、それは、たとえば当該損害や損失が合理的に予見可能であったり、ザイリンクスがそれらの可能性について助言を受けていた場合であったとしても同様です。ザイリンクスは、本情報に含まれるいかなる誤りも訂正する義務を負わず、本情報または製品仕様のアップデートを貴殿または貴社に知らせる義務も負いません。事前の書面による同意のない限り、貴殿または貴社は本情報を再生産、変更、頒布、または公に展示してはなりません。一定の製品は、ザイリンクスの限定的保証の諸条件に従うこととなるので、<https://japan.xilinx.com/legal.htm#tos> で見られるザイリンクスの販売条件を参照してください。IP コアは、ザイリンクスが貴殿または貴社に付与したライセンスに含まれる保証と補助的条件に従うことになります。ザイリンクスの製品は、フェイルセーフとして、または、フェイルセーフの動作を要求するアプリケーションに使用するために、設計されたり意図されたりしていません。そのような重大なアプリケーションにザイリンクスの製品を使用する場合のリスクと責任は、貴殿または貴社が単独で負うものです。<https://japan.xilinx.com/legal.htm#tos> で見られるザイリンクスの販売条件を参照してください。

自動車用のアプリケーションの免責条項

オートモーティブ製品 (製品番号に「XA」が含まれる) は、ISO 26262 自動車用機能安全規格に従った安全コンセプトまたは余剰性の機能 (「セーフティ 設計」) がない限り、エアバッグの展開における使用または車両の制御に影響するアプリケーション (「セーフティ アプリケーション」) における使用は保証されていません。顧客は、製品を組み込むすべてのシステムについて、その使用前または提供前に安全を目的として十分なテストを行うものとし、セーフティ設計なしにセーフティ アプリケーションで製品を使用するリスクはすべて顧客が負い、製品の責任の制限を規定する適用法令および規則にのみ従うものとし、

商標

© Copyright 2021 Xilinx, Inc. Xilinx、Xilinx のロゴ、Alveo、Artix、Kintex、Spartan、Versal、Virtex、Vivado、Zynq、およびこの文書に含まれるその他の指定されたブランドは、米国およびその他の各国のザイリンクス社の商標です。AMBA、AMBA Designer、Arm、ARM1176JZ-S、CoreSight、Cortex、PrimeCell、Mali、および MPCore は、EU およびその他の各国の Arm Limited の商標です。

この資料に関するフィードバックおよびリンクなどの問題につきましては、jpn_trans_feedback@xilinx.com まで、または各ページの右下にある [フィードバック送信] ボタンをクリックすると表示されるフォームからお知らせください。フィードバックは日本語で入力可能です。いただきましたご意見を参考に早急に対応させていただきます。なお、このメール アドレスへのお問い合わせは受け付けておりません。あらかじめご了承ください。