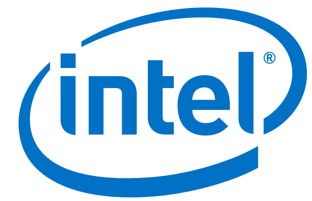


インテル® HLS コンパイラー

リファレンス･マニュアル

インテル® Quartus® Prime 開発デザインスイートの更新情報: **17.0**

MNL-1083 | 2017.06.23

最新版をウェブからダウンロード: PDF | HTML

目次

1. インテル® HLS コンパイラー・リファレンス・マニュアル.. 3
1.1. HLS コンパイラー・コマンドオプション... 3
1.2. コンポーネント合成でサポートされるサブセット 5
1.3. 並行処理モデル...6
1.3.1. メモリースペースまたは I/O 内のシリアルの等価性.. 6
1.3.2. `hls_max_concurrency` を使用した並行処理の制御.. 6
1.4. コンポーネント・インターフェイス..7
1.4.1. コンポーネント・インターフェイスの構造体.. 7
1.4.2. パラメーター・プロトコル.. 7
1.4.3. コンポーネント呼び出しインターフェイスを制御するマクロと属性.............................. 15
1.4.4. スレーブ・インターフェイス..18
1.4.5. ポインター・エイリアシングの回避.. 21
1.5. 標準 C および C++ ライブラリー.. 21
1.6. 静的変数..22
1.7. ループ...23
1.7.1. ループ展開 24
1.7.2. ループ結合 24
1.7.3. ループ・イニシエーション・インターバル (II) 25
1.7.4. ivdep プラグマを使用したループ伝搬依存性の削除... 26
1.7.5. ループ並行処理の制御...28
1.8. 任意高精度整数のサポート...28
1.8.1. コンポーネントの `ac_int` データ型の定義..29
1.8.2. `ac_int` データ型のデバッグ用ツール... 30
1.8.3. コンポーネント内の `ac_fixed` データ型の定義.. 31
1.9. エリアの最小化およびオンチップ・メモリー・アーキテクチャーの制御...31
1.9.1. メモリー属性の使用例... 32
1.9.2. hls_merge 属性の使用例.. 34
1.9.3. hls_bankbits 属性での使用例..38
1.10. 改訂履歴 41

A. HLS コンパイラー・クイック・リファレンス.. 43

B. HLS コンパイラーでサポートされる標準の数学関数...52
B.1. 改訂履歴 56

1. インテル® HLS コンパイラー・リファレンス・マニュアル

インテル® HLS コンパイラー・リファレンス・マニュアル は、高位合成 (HLS) コンポーネント・デザイン・フローに関するリファレンス情報を提供します。インテル HLS コンパイラーは、コンパイラー・コマンドの名前を反映し、しばしば i++コンパイラーと呼ばれることもあります。

HLS コンパイラー・コマンドオプション (3 ページ)
コンポーネント合成でサポートされるサブセット (5 ページ)
並行処理モデル (6 ページ)
コンポーネント・インターフェイス (7 ページ)
標準 C および C++ ライブラリー (21 ページ)
静的変数 (22 ページ)
ループ (23 ページ)
任意高精度整数のサポート (28 ページ)
エリアの最小化およびオンチップ・メモリー・アーキテクチャーの制御 (31 ページ)
改訂履歴 (41 ページ)

1.1. HLS コンパイラー・コマンドオプション

HLS コンパイラーは、一般的な関数を実行するための呼び出しやファイル・リンキング、またはコンパイルのカスタマイズが可能なコマンドオプションを含んでいます。

表 1. 一般的なコマンドオプション

これらの i++コマンドオプションは、一般的なコンパイラー関数を実行します。

コマンドオプション	説明
`--debug-log`	コンパイラーに診断情報を含めたログファイルを生成するように指示します。 デフォルトでは、`debug.log` ファイルは現在の作業ディレクトリー内の `a.prj` サブディレクトリーにあります。`-o` *<result>*コマンドオプションも含める場合は、`debug.log` ファイルは *<result>*`.prj` サブディレクトリーに格納されます。
`-h` または `--help`	コンパイラーにすべてのコマンドオプションと画面上のそれらの説明を表示するように指示します。 コマンド : `i++ -h` または `i++ --help`
`-o` *<result>*	コンパイラーに *<result>* 実行可能ファイルと *<result>*`.prj` ディレクトリーの中に出力を配置するように指示します。 コンパイラーはデフォルトで Linux 用に `a.out` ファイルを出力し、Windows 用に `a.exe` ファイルを出力します。`-o` *<result>* コマンドオプションは、コンパイラー出力の名前を指定できます。 コマンド例 : `i++ -o hlsoutput multiplier.c` このコマンド例を呼び出すと、現在の作業ディレクトリー内に Linux 用の `hlsoutput` 実行可能ファイルと Windows 用の `hlsoutput.exe` を作成します。
`-v`	コンパイルの進行状況を記述するメッセージの表示をコンパイラーに指する Verbose モードです。

continued...

Intel Corporation.無断での引用、転載を禁じます。Intel、インテル、Intel ロゴ、Altera、ARRIA、CYCLONE、ENPIRION、MAX、NIOS、QUARTUS および STRATIX の名称およびロゴは、アメリカ合衆国および/ またはその他の国における Intel Corporation の商標です。インテルは FPGA 製品および半導体製品の性能がインテルの標準保証に準拠することを保証しますが、インテル製品およびサービスは、予告なく変更される場合があります。インテルが書面にて明示的に同意する場合を除き、インテルはここに記載されたアプリケーション、または、いかなる情報、製品、またはサービスの使用によって生じるいっさいの責任を負いません。インテル製品の顧客は、製品またはサービスを購入する前、および、公開済みの情報を信頼する前には、デバイスの仕様を最新のバージョンにしておくことをお勧めします。
*その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。
ISO 9001:2015 登録済

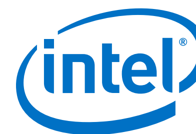
コマンドオプション	説明
	コマンド例は、 `i++ -v hls/multiplier/multiplier.c` で、`multiplier.c` は入力ファイルを示します。
`--version`	コンパイラーに画面上にバージョン情報を表示するように指示します。 コマンド : `i++ --version`

表 2. コンパイルをカスタマイズするコマンドオプション

これらの i++コマンドオプションは、ソースファイルからオブジェクト・ファイルへの翻訳に影響するコンパイラー関数を実行します。

オプション	説明
`-c`	コンパイラーに現在の作業ディレクトリーにオブジェクト・ファイル (`.o`) を前処理、解析、および生成するように指示します。 コマンド例 : `i++ -c multiplier.c` このコマンド例を呼び出すと、`multiplier.o` ファイルを作成します。
`--component` *<components>*	コンパイラーが RTL に合成する関数名のコンマ区切り値リストを指定できます。 コマンド例 : `i++ --component` *<components> <input_files>*
`-D` *<macro>* `[=` *<val>* `]`	コンパイラーにマクロ定義 (*<macro>*) とその値 (*<val>*) を渡すことができます。 *<val>* で値を指定しない場合、デフォルト値は 1 になります。
`-I` *<dir>*	ディレクトリー (*<dir>*) をインクルード・パス・リストの最後に追加します。
`-march= [X86-64 | "`*<FPGA_family>*`" | "`*<FPGA_part_number>*`"`	コンポーネントを指定されたアーキテクチャーまたは FPGA ファミリーにコンパイルするように、コンパイラーに指示します。 `-march` コンパイラー・オプションは次のいずれかの値を取ります。 *X86-64* : コンパイラーにエミュレーター・フローでのコードをコンパイルするように指示します。 *"<FPGA_family>"* : コンパイラーにターゲット FPGA デバイスファミリーのコードをコンパイルするように指示します。*<FPGA_family>* 値は次のデバイスファミリーのいずれかになります。 • `Arria V` • `Arria 10` • `Cyclone V` • `MAX 10` • `Stratix V` • `Stratix 10` *"<FPGA_part_number>"* : コンパイラーにターゲットデバイスのコードをコンパイルするように指示します。コンパイラーはここで指定する FPGA パーツナンバーから FPGA デバイスファミリーを決定します。 コマンド例 : `i++ -march=x86-64 multiplier.c` このコマンド例を呼び出すと、現在の作業ディレクトリーに `a.out` ファイルを作成します。 コンパイルが失敗した場合は、プロジェクト・ディレクトリー (`a.prj` など) にある `debug.log` ファイルで詳細を参照してください。
`--promote-integers`	コンパイラーに g++ 整数昇格を模倣するための追加の FPGA リソースを使用するように指示します。 `--promote-integers` コマンドオプションについては *<path to i++ installation>*`/examples/tutorials/best_practices/integer_promotion` デザイン例を参照にしてください。

continued...



オプション	説明
	コンパイルに g++ 整数昇格ルールを適用すると、大量のエリア・オーバーヘッドを引き起こします。インテルは、プロダクション・コンパイルでの整数昇格を推奨していません。プロダクション・コンパイルでは、未定義の算術オーバーフロー動作に依存しなくてもいいように、コンポーネント内のデータ型のサイズを適切にします。
`--quartus-compile`	Quartus® Prime ソフトウェアを使用して HDL ファイルをコンパイルします。 コマンド例 : `i++ --quartus-compile` *<input_files>* 出力ファイルは、 *<result>*`.prj/quartus` ディレクトリー内のコンパイルした Quartus Prime プロジェクトと *<result>*`.prj/reports` ディレクトリー内のレポートです。このコンパイルは、コンポーネントを FPGA アーキテクチャーにコンパイルする際に作成されるすべてのファイルも生成します。
`--simulator` *<name>* `--simulator none`	検証の実行に使用しているシミュレーターを指定します。 *重要:* `--simulator` *<name>* コマンドオプションは`-march="` *<FPGA_family_or_part_number>* `"` コマンドオプションと併せてのみ動作します。 デフォルトでは、i++コマンドに`-march="` *<FPGA_family_or_part_number>* `"` を含める際、`--simulator` *<name>* は自動的に`--simulator modelsim` に設定されるため、`--simulator` *<name>* オプションを指定する必要はありません。 `--simulator none` オプションは、検証フローをスキップし、対応するテストベンチを生成せずにコンポーネントの RTL を生成するように、HLS コンパイラーに指示します。このオプションは、生成された HLD レポート (`report.html`) でコンポーネント・デザインを反復する必要がある場合に使用します。 コマンド例 : `i++ -march="`*<FPGA_family_or_part_number>*`" --simulator none multiplier.c`

表 3. ファイルリンキングをカスタマイズするコマンドオプション

これらの HLS コマンドオプションは、バイナリー または RTL コンポーネントへのオブジェクト・ファイルの翻訳に影響するコンパイラーの動作を指定します。

オプション	説明
`-clock` *<clock_spec>*	指定されたクロック周波数または期間で RTL を最適化します。
`--fpc`	可能な場合、中間丸めと変換を削除します。
`--fp-relaxed`	算術演算の順序を緩和します。
`-ghdl`	フルデバッグの可視化を有効にし、検証実行可能の実行時にすべての信号をログ付けします。実行可能が実行された後、シミュレーターは波形を `a.prj/verification/vsim.wlf` ファイルにログ付けします。
`-L` *<dir>*	ディレクトリー (*<dir>*) をライブラリー・ファイルの検索パスの最後に追加します。
`-l` *<library>*	オブジェクト・ファイルがバイナリーにリンクする際に、ライブラリー・ファイル名を指定します。

1.2. コンポーネント合成でサポートされるサブセット

HLS コンパイラーには、サポートされる C99 サブセットと C++ に関連するいくつかの合成制限があります。

- 動的ループ境界または分岐は、エリア・オーバーヘッドを引き起こす可能性があります。
- コンパイラーは、動的メモリー割り当て、仮想関数、関数ポインター、およびこの資料の付録で明示的に言及しているサポートされる数学関数を除く C++ または C ライブラリーの関数を合成できません。一般的に、コンパイラーは、クラス、構造、関数、テンプレート、およびポインターを含む関数を合成できます。

コンパイル時にコードがコンパイラーに提供できる情報が多いほど、結果となるハードウェアはより小さく、より速くなります。いくつかの C++ コンストラクターは合成可能ですが、可能な場合は C99 でコンポーネント関数を作成してください。

注意: これらの合成制限はテストベンチ・コードには適用されません。

1.3. 並行処理モデル

HLS コンパイラーは、コンポーネントの完全にパイプラインされたデータパスが必要であることを前提としています。C++ の実装では、最初のコールが返される前に関数を複数回コールして (例えば、複数のスレッドによって) 、完全にパイプラインされたデータパスを考慮するかもしれません。合成されたデータパス内のスレッドの動作は、並行処理モデルの対象となります。

1.3.1. メモリースペースまたは I/O 内のシリアルの等価性

単一メモリースペースまたは I/O 内では、コンポーネントへの各コール、つまり、`start` 信号がアサートされ、`busy` 信号がディアサートされるサイクルごとに、関数プロトコル・インターフェイスで前の関数コールが完全に実行されたかのように動作します。

単一共有メモリースペースを表示する場合、複数の関数コールが順次実行されると考える必要があります。この場合、`done` 信号がアサートされると、ハードウェアのコンポーネント呼び出しの結果は次のコンポーネント呼び出しと外部システムの両方からの可視化が保証されます。

関連する依存性の並列実行を許容する場合、HLS コンパイラーはパイプライン並列化を利用してコンポーネント呼び出しを実行し、並列にループ反復処理を行います。HLS コンパイラーはコンポーネント呼び出しに広がる依存性のトラックし続けるハードウェアを生成するため、メモリースペースに広がるシリアルの等価性を保証しながらパイプライン並列化をサポートすることができます。I/O 命令間の順序は保証されません。

1.3.2. `hls_max_concurrency` を使用した並行処理の制御

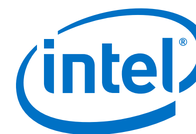
`hls_max_concurrency` コンポーネント属性を使用して、コンポーネントの最大並行処理を増減することができます。コンポーネントの並行処理は、一度に実行できるコンポーネント呼び出し数です。デフォルトでは、インテル HLS コンパイラーはコンポーネントがピーク・スループットで動作するため、並行処理を最大化しようと試みます。

コンポーネントを宣言する前に、次に示すように `hls_max_concurrency` 属性を追加することでコンポーネントの最大並行処理を制御できます。

```
#include "HLS/hls.h"

hls_max_concurrency(3)
component void foo ( /* arguments */ ){
  // Component code
}
```

次のような場合、インテル HLS コンパイラーはコンポーネント並行処理を最大化しようとしません。

- インテル HLS コンパイラーは、コンポーネント・レベルでのスループットの増加のためにローカルメモリーを自動的に複製しません。コンポーネント呼び出しが、コンポーネント呼び出しによって使用される (静的でない) ローカル・メモリー・システムを使用する場合、前の呼び出しがローカルメモリーへ、およびローカルメモリーからのすべてのアクセスを終了するまで、次の呼び出しを開始できません。この制限は、ループ分析レポートにアレイでのロードストアーの依存関係として表示されます。コンポーネントで `hls_max_concurrency(N)` 属性を追加することで、ローカルメモリーを複製し、同時にコンポーネントの複数の呼び出しが可能です。
- 場合によっては、コンパイラーは大量のエリアを節約するために並行処理を減らします。これらの場合は、`hls_max_concurrency(N)` 属性は並行処理を 1 つから増やすことができます。



また、max_concurrency(*N*)プラグマを使用して、ループの並行処理を制御することも可能です。max_concurrency(*N*)プラグマについて詳しくは、ループ並行処理の制御 (28 ページ)を参照してください。

1.4. コンポーネント・インターフェイス

HLS コンパイラーが生成するすべてのコンポーネントには、パラメーター・インターフェイスとコンポーネント呼び出しインターフェイスの 2 つのインターフェイスがあります。

*パラメーター・インターフェイス*は、関数の引数の中にデータをプッシュしたり関数からのデータを返したりするために使用するプロトコルです。次の関数の定義を考察します。

```
component int interfaces(int a, int b);
```

関数の定義は 2 つの引数 (すなわち a と b) があります。ハードウェアで使用するプロトコルを各引数に渡す際にどのように指定するかについて詳しくは、パラメーター・プロトコル (7 ページ)を参照してください。

コンポーネントへのコール動作やコンポーネントから返す動作を表すには、*コンポーネント呼び出しインターフェイス*が必要です。このインターフェイスは、戻り値 (nonvoid 関数で) と、コンポーネントに制御を渡し、関数が実行を完了すると制御を戻すハンドシェイク信号が含まれています。

1.4.1. コンポーネント・インターフェイスの構造体

コンポーネント内の構造体での実装インターフェイスのパディングおよび Packed-ness に関する情報は、 *<a.prj>*/components/*<component_name>* フォルダー内の interface_structs.sv ファイルを確認し、参照してください。

interface_structs.sv ファイルには、コンポーネント・インターフェイスで見つかった Verilog 形式の定義が含まれています。

1.4.2. パラメーター・プロトコル

スカラー・パラメーターおよび Avalon Streaming インターフェイス (7 ページ)

ポインター・パラメーター、参照パラメーター、および Avalon Memory-Mapped インターフェイス (10 ページ)

メモリーマップド・マスター・テストベンチ・コントラクター (11 ページ)

参照パラメーター (12 ページ)

グローバル変数 (12 ページ)

引数の実装を制御するマクロと属性 (13 ページ)

1.4.2.1. スカラー・パラメーターおよび Avalon Streaming インターフェイス

コンポーネントのすべてのスカラー・パラメーター関数引数には、生成した Verilog モジュール内に入力ポートがあります。これらのポートは、Avalon® Streaming (Avalon-ST) インターフェイス仕様で示すように、start と busy 入力に関連するパイプラインされた入力で、start は valid 信号、busy は反転した ready 信号です。

スカラー引数入力に加えて、コンポーネントは、Avalon-ST インターフェイス仕様に準拠する明示的な入力ストリームおよび出力ストリームを有することができます。各ストリームは、コンポーネント内部に 1 つのリードまたはライトのみを有する場合があります。これらの入力ストリームおよび出力ストリーム

は、提供された ihc::stream_in<> と ihc::stream_out<> テンプレート・クラスをコンポーネントへの関数引数として使用することで、C ソースで表されます。これらのクラスをコンポーネント・シグネチャーの参照パラメーターとして定義する必要があります。これらのクラスは他のクラスの基本クラスとして使用したり、構造体や配列などのその他の形式でカプセル化したりすることはできません。ただし、他のクラス内部の参照として使用することができ、つまり、データメンバーとしてストリームへの参照を持つクラスが構築できるということを意味しています。

ストリーミング・インターフェイスについて詳しくは、*Avalon Interface Specifications* を参照してください。

表 4. Avalon-ST インターフェイスで使用可能なスカラー・パラメーター

テンプレート引数	値	デフォルト	説明
ihc::type	任意の有効な C++ 型	N/A	ストリームにより実行されるデータ型。
ihc::buffer	正の整数値	0	ストリームに関連するデータ上のワード単位での FIFO バッファーの容量。 このパラメーターは入力ストリームでのみ有効です。
ihc::readyLatency	正の整数値 値の範囲は 0 ~ 8	0	ready 信号がディアサートされてから、ストリームシンクが新しい入力を受け入れられなるまでの間のサイクル数。概念的には、このパラメーターはストリームに関連するデータの入力 FIFO バッファーでの almost ready レイテンシーとして確認できます。
ihc::bitsPerSymbol	正の整数値 値はデータ型サイズにより均等に分割される必要があります。	*データ型のサイズ*	データがどのようにデータバスのシンボル内で分割されるかを記述します。データは常にリトル・エンディアンの順序で分割されます。
ihc::usesPackets	true または false	false	ストリーム・インターフェイスの startofpacket と endofpacket のサイドバンド信号を抽出します。信号はパケットベースのリード / ライトによりアクセス可能です。
ihc::usesReady	true または false	true	stream_out<> インターフェイスでのみサポートされます。 ready 信号が存在するかどうかを制御します。true の場合、ダウンストリーム・シンクは valid がアサートされるサイクルごとにデータを受け入れます。 これは、ストリーム書き込みコールを tryWrite に変更し、success が常に true であることと同じことです。 false に設定すると、readyLatency は 0 になります。
ihc::usesValid	true または false	true	stream_in<>インターフェイスでのみサポートされます。 valid 信号がストリーム・インターフェイスに存在しているかどうかを制御します。false の場合、アップストリーム・ソースは ready 信号がアサートされるサイクルごとに有効データを提供します。 これは、ストリーム書き込みコールを tryRead に変更し、success が常に true であることと同じことです。 false に設定すると、buffer と readyLatency は 0 になります。

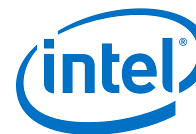


表 5. テストベンチとコンポーネント･コードの両方でストリームとインタラクトする使用可能な関数

注意: 入力および出力ストリームとしてコンポーネント内で使用できるストリーム･アプリケーション･プログラミング･インターフェイス (API) には制限があります。

API	ブロックキング	サイドバンド信号	コンポーネントの stream_in による使用	コンポーネントの stream_out による使用	説明
`T read()`	可能	不可能	可能	不可能	この関数は、ストリームから次の使用可能なデータ項目を読み込みます。コンポーネント内に使用可能なデータがない場合、関数は有効なデータが存在するまでデータパスをストールします。
`T read(bool& sop, bool& eop)`	可能	可能	可能	不可能	この関数は usesPackets が true に設定されている場合のみ使用可能です。 この関数は、ストリームから次の使用可能なデータ項目を読み込みます。コンポーネント内に使用可能なデータがない場合、関数は有効なデータが存在するまでデータパスをストールします。
`T tryRead(bool& success)`	不可能	不可能	可能	不可能	この関数は、使用可能な場合にストリームから次の使用可能なデータ項目を読み込みます。この関数コールはストールを引き起こしません。bool 引数はデータが有効かどうかを表示します。
`T tryRead(bool& success, bool& sop, bool& eop)`	不可能	可能	可能	不可能	この関数は、usesPackets が true に設定されている場合のみ使用可能です。 この関数は、次のデータが使用可能な場合にストリームからそのデータを読み込みます。この関数コールはストールを引き起こしません。bool 引数はデータが有効かどうかを表示します。
`void write(T arg)`	可能	不可能	不可能	可能	この関数はデータをストリームに書き込みます。ストリームが新しいデータを受け入れられない場合 (例えば、ストリームの他のエンドからのストールのため)、この関数はストリームにデータを書き込めるまでデータパスをストールします。
`void write(T data, bool sop, bool eop)`	可能	可能	不可能	可能	この関数は、usesPackets が true に設定されている場合のみ使用可能です。 この関数はデータをストリームに書き込みます。ストリームが新しいデータを受け入れられない場合、この関数はストリームにデータを書き込めるまでデータパスをストールします。
`bool tryWrite(T data)`	不可能	不可能	不可能	可能	この関数は、新しいデータを受け入れられる場合、ストリームにデータを書き込みます。この関数コールはストールを引き起こしません。bool 戻り値はライト命令が実行されたかどうかを表示します。
`bool tryWrite(T data, bool sop, bool eop)`	不可能	可能	不可能	可能	この関数は、usesPackets が true に設定されている場合のみ使用可能です。 この関数は、新しいデータを受け入れられる場合にストリームにデータを書き込みます。この関数コールはストールを引き起こしません。bool 戻り値はライト命令が実行されたかどうかを表示します。

注意: コンポーネント内では、ストリーム間に存在するデータの依存関係を除き、異なるストリームの実行順序の保証はありません。

関連情報

Avalon Interface Specifications

1.4.2.2. ポインター・パラメーター、参照パラメーター、および Avalon Memory-Mapped インターフェイス

コンポーネントは、Avalon Memory-Mapped (Avalon-MM) インターフェイスを介して外部メモリーとインターフェイスすることができます。Avalon-MM インターフェイスは、ポインター関数引数を黙示的に使用するか、または `hls.h` ファイルで定義される `mm_master<>` クラスを明示的に使用して指定できます。`mm_master<>` クラスはコンポーネント・シグネチャーの参照パラメーターとして機能しなければなりません。

表 6. Avalon-MM インターフェイスのコンフィグレーションで使用可能なテンプレート引数

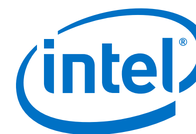
すべてのポインターは、デフォルト設定で使用する信号インターフェイスで調停されます。ポインターのデフォルトのアドレススペースは 0 です。

Avalon-MM マスターでの入力テンプレート構文は、変化の対象となります。

テンプレート引数	値	デフォルト	説明
`type`	任意の有効な C++ 型	N/A	基となるポインター型です。マスター・オブジェクトで実行されるポインター演算はこの型に準じます。マスターの依存関係は、`sizeof(`*`type`*`)`の幅を有するロード・ストアー・サイトの結果となります。ポインター引数は、少なくともバイト単位で `sizeof(`*`type`*`)`に揃えられている必要があります。
`ihc::dwidth<`*`value`*`>`	8、16、32、64、128、256、512、または 1024	64	ビット単位のメモリーマップド・データバスの幅。
`ihc::awidth<`*`value`*`>`	1 – 64 の範囲の整数値	64	ビット単位のメモリーマップド・アドレスバスの幅。
`ihc::aspace<`*`value`*`>`	0 よりも大きい整数値	1	マスターに関連するインターフェイスのアドレススペースです。同じアドレススペースのすべてのマスターは、コンポーネント内で単一インターフェイスに調停されます。これらのマスターは、インターフェイスを記述する同じテンプレート・パラメーターを共有している必要があります。
`ihc::latency<`*`value`*`>`	正の整数値	1	外部メモリーが有効な読み出しデータを返す際に、read コマンドがコンポーネントを終了してから保証されるレイテンシーです。 このレイテンシーが変数の場合、0 に設定されます。
`ihc::maxburst<`*`value`*`>`	1 – 1024 の範囲の整数値	1	リードおよびライト・トランザクションに関連付けることができるデータ転送の最大数です。固定レイテンシー・インターフェイスでは、値は 1 に設定される必要があります。 詳しくは、*Avalon Interface Specifications* を参照ししてください。
`ihc::align<`*`value`*`>`	`type` 引数の整列の整数倍である整数値	型のアライメント	基本ポインターアドレスのバイト境界合わせです。HLS コンパイラーはこの情報を使用してこのポインターへのロードとストアーで可能な結合の量を決定します。 データ型の幅がマスターデータの幅より小さい場合、インテル はこの引数を少なくともマスター幅と同じに設定することを推奨しています。 例えば、型が `char` でマスター幅を 64 ビットになるように設定している場合、各クロックサイクルごとに 8 文字をリードまたはライトできるように、`align` テンプレート引数を 8 バイトに設定します。 ***注意:*** 呼び出し元は、`align` 引数の設定値にデータを整列する必要があります。そうでない場合、機能上で障害が起こる場合があります。

関連情報

Avalon Interface Specifications



1.4.2.3. メモリーマップド·マスター·テストベンチ·コントラクター

これらのメモリー·インターフェイスを記述するためのメモリーマップド·マスター·クラス (mm_master<>) が使用するコンポーネントでは、各 mm_master 引数でのテストベンチで mm_master<>オブジェクトを作成する必要があります。

mm_master<>オブジェクトを作成するには、コードに次のコンストラクターを追加します。

```
ihc::mm_master<int, … > mm(void* ptr, int size, bool use_socket=false);
```

位置

- void* ptr はテストベンチでのメモリーへの基となるポインターです。
- int size はバイト単位のバッファーの合計サイズです。
- bool use_socket はメモリーバッファーのコピーをオーバーライドして、すべてのメモリーアクセスをテストベンチ·メモリーに戻すオプションです。

  デフォルトでは、HLS コンパイラーはメモリーバッファーをシミュレーターにコピーし、コンポーネントが実行された後にそれをコピーして戻します。リンクされたリストのポインターチェーシングなどの場合、メモリーバッファーを前後にコピーすることは好ましくありません。この動作は bool use_socket を true に設定することでオーバーライドできます。

  注意: bool use_socket を true に設定すると、64 ビット幅アドレスの mm_masters のみがサポートされます。更に、このオプション設定はシミュレーションのランタイムを増加させます。

1.4.2.3.1. 暗黙的および明示的なメモリー·インターフェイスの記述例

明示的な mm_master クラスを指定することで、メモリー·インターフェイスを記述するコンポーネント·コードを最適化します。

暗黙的な例

次のコード例は、両方のポインター逆参照からのロード / ストアー命令をコンポーネントのトップレベル·モジュールの単一インターフェイスに調停します。このインターフェイスは、64 ビットのデータバス幅と 64 ビットアドレス幅、および固定レイテンシー 1 を有します。

```
#include "HLS/hls.h"
component void dut(int *ptr1, int *ptr2) {
 *ptr1 += *ptr2;
 *ptr2 += ptr1[1];
}

int main(void) {
  int x[2] = {0, 1};
  int y = 2;

  dut(x, &y);

  return 0;
}
```

明示的な例

この例は、明示的な mm_master クラスを使用して、特定のメモリー・インターフェイスに対して前のコードスニペットを最適化する方法を示します。mm_master クラスは定義されたテンプレートであり、次の特性があります。

- 各インターフェイスには、2 つの独立したインターフェイスを推測してコンポーネント内の調停量を減らす一意の ID が与えられる。
- デフォルトの 64 ビット幅より大きいデータバス幅
- デフォルトの 64 ビット幅より小さいアドレスビット幅
- 2 の固定レイテンシーを有するインターフェイス

これらの特性を定義することで、システムが正確に 2 クロックサイクル後に有効な読み込みデータを返し、読み込みと書き込みの両方でインターフェイスがストールしないことが保証されますが、システムは 2 つの異なるメモリーが提供できなければなりません。

```
#include "HLS/hls.h"
component void dut(ihc::mm_master<int, ihc::dwidth<256>,
                                       ihc::awidth<32>,
                                       ihc::aspace<1>,
                                       ihc::latency<2> > &mm1,
                   ihc::mm_master<int, ihc::dwidth<256>,
                                       ihc::awidth<32>,
                                       ihc::aspace<4>,
                                       ihc::latency<2> > &mm2) {
  *mm1 += *mm2;
  *mm2 += mm1[1];
}
int main(void) {
  int x[2] = {0, 1};
  int y = 2;

  ihc::mm_master<int, ihc::dwidth<256>, ihc::awidth<32>, ihc::aspace<1>,
                   ihc::latency<2> > mm_x(x,2*sizeof(int),false);
  ihc::mm_master<int, ihc::dwidth<256>, ihc::awidth<32>, ihc::aspace<1>,
                   ihc::latency<2> > mm_y(&y,sizeof(int),false);

  dut(mm_x, mm_y);

  return 0;
}
```

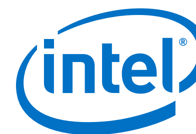
1.4.2.4. 参照パラメーター

HLS コンパイラーはポインター・パラメーターと同様の方法で参照パラメーターを扱います。現在、明示的なメモリーマスター・インターフェイスとストリーミング・インターフェイス (つまり、stream_in と stream_out) もコンパイラーを参照して渡されます。

1.4.2.5. グローバル変数

コンポーネントはグローバル変数の使用と更新ができます。HLS コンパイラーは、コンポーネントへのポインター引数と同様の方法でグローバル変数をモデル化します。

コンポーネントが使用する各グローバル変数は、@<*global variable name*> という名前のコンデュイット引数があり、この変数はシステムメモリー内の特定のグローバル変数のアドレスが指定されている必要があります。その結果、コンポーネント内のグローバル変数を使用すると、グローバル変数を定数として宣言する場合と比べて、デザインに必要なエリアが増えます。追加エリアは、グローバル変数とロード・ストアー・ユニットのやり取りに必要です。



定数グローバル変数の最適化

グローバル変数が定数の場合、const として宣言するとグロバール変数の余白エリアの使用はされなくなります。

1.4.2.6. 引数の実装を制御するマクロと属性

デフォルトでは、HLS コンパイラーは関数引数のインターフェイスをコンポーネントの呼び出し (start または busy) インターフェイスに同期する入力コンデュイットとして実装します。ソースコード内のマクロまたは属性を個別のコンポーネント引数に追加することにより、これらのインターフェイスの実装をポインター・パラメーターと同様の方法で制御するオプションがあります。

表 7. 可能なインターフェイス合成マクロと属性

<path to i++ installation>/include/HLS/hls.h ファイル内のマクロ定義	属性	説明
hls_conduit_argument	__attribute__((argument_interface("wire")))	デフォルト実装。 コンパイラーは、Avalon-ST プロトコルに従って、コンポーネントの呼び出し (start または busy) インターフェイスに同期する入力コンデュイットとして引数を実装します。
hls_avalon_slave_register_argument	__attribute__((argument_interface("avalon_mm_slave")))	コンパイラーは、Avalon-MM スレーブ・インターフェイスを介して読み出しと書き込みができるレジスターとして引数を実装します。引数は、コンデュイットの実装と同様にコンポーネントのパイプラインに読み込まれます。実装は、start または busy インターフェイスに同期します。
hls_avalon_slave_memory_argument(*size_of_slave_memory_in_bytes*)	N/A	コンパイラーは、専用スレーブ・インターフェイスを介して読み出しと書き込みができるオンチップ・メモリーブロックに引数を実装します。 生成されたメモリーは、他のすべての内部コンポーネント・メモリー (つまり、バンキング、結合など) と同じアーキテクチャーの最適化を行います。コンパイラーが静的結合の最適化を実行する場合、スレーブ・インターフェイスのデータ幅は結合した幅になります。 *注意:* この属性はポインター引数にのみ適用されます。

表 8. 引数の意図する動作の指定におけるマクロと属性

引数の実装の制御に加え、argument_interface 属性と組み合わせて属性を使用して引数の意図した動作を指定することができます。または、対応するマクロを挿入することもできます。

意図する動作を指定しない場合、引数のデフォルト動作は不安定です。不安定引数は、コンポーネントにライブデータがある間 (つまり、パイプラインされた関数呼び出しの間) に変更されることがあります。

<path to i++ installation>/include/HLS/hls.h ファイル内のマクロ定義	属性	説明
hls_stable_argument	__attribute__((stable_argument))	安定引数とは、コンポーネントにライブデータがある間 (つまり、パイプラインされた関数呼び出しの間) に変更されない引数のことです。 コンポーネント実行中の安定引数の変更には予期しない動作が発生する可能性があります。特に、安定引数の使用ごとに古い値または新しい値になる可能性があり、呼び出し内での一貫性が保証されません。すなわち、同じ呼び出しでの同じ変数が複数の値で表示されます。

<path to i++ installation>/include/HLS/hls.h ファイル内のマクロ定義	属性	説明
		適切な場合は、安定引数の使用によりデザインで大量のレジスター数の節約が可能です。

1.4.2.6.1. インターフェイス合成マクロの使用例

次のコード例は、関数引数にインターフェイス合成マクロのインクルードがどのようにコンポーネント dut での実装に影響するかを示しています。

図 -1: hls_conduit_argument および hls_avalon_slave_register_argument マクロを使用した関数の引数の実装

```
#include "HLS/hls.h"
component int dut(hls_conduit_argument int a,
                  hls_avalon_slave_register_argument int b)
{
    return a * b;
}
```

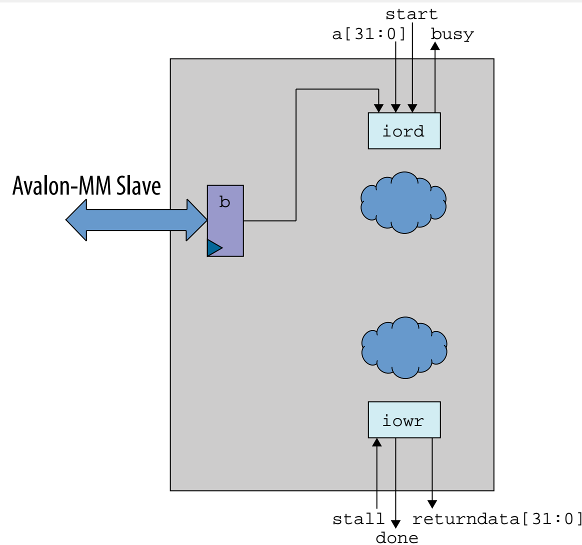
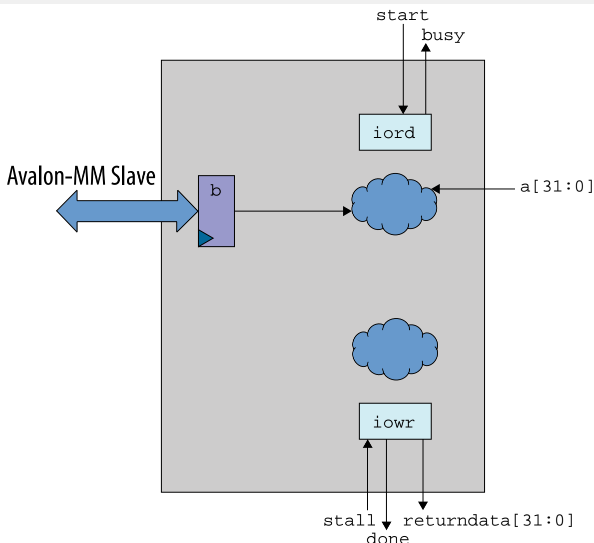


図 -2: hls_stable_argument マクロを使用した関数の引数の実装

```
#include "HLS/hls.h"
component int dut(hls_stable_argument
                  hls_conduit_argument int a,
                  hls_stable_argument
                  hls_avalon_slave_register_argument int b)
{
    return a * b;
}
```



1.4.3. コンポーネント呼び出しインターフェイスを制御するマクロと属性

コンポーネント呼び出しインターフェイスは呼び出している関数の動作に対応する制御信号に言及します。すべての不安定なコンポーネント引数の入力は、このコンポーネント呼び出しプロトコルに従って同期されます。

表 9. 可能なコンポーネント呼び出しプロトコルマクロおよび属性

マクロ	属性	説明
hls_avalon_streaming_ component	__attribute__((component_interface("avalon_streaming")))	この属性は関数コールとリターンストリームの両方の Avalon-ST プロトコルに従います。コンポーネントは、start 信号がアサートされ、busy 信号がディアサートされると、不安定引数を消費します。コンポーネントは done 信号がアサートされると戻りデータを生成します。 このインターフェイスはデフォルトのコンポーネント呼び出しインターフェイスです。 トップレベル・モジュール・ポート 関数コール—start、busy 関数戻り値—done、stall
hls_avalon_slave_component	__attribute__((component_interface("avalon_mm_slave")))	start、done、および returndata (可能な場合) 信号は、コンポーネントのスレーブ・メモリーマップにレジスターされます。詳細については、*CSR スレーブ*の項を参照してください。

continued...

マクロ	属性	説明
		トップレベル·モジュール·ポート : Avalon-MM スレーブ·インターフェイス
hls_always_run_component	__attribute__((component_interface("always_run")))	start 信号はコンポーネントの内部で 1 に結び付けられます。done 信号の出力はありません。コンポーネントのデータパスがデータの入力と出力の明示的なストリームにのみ依存する場合は、このプロトコルを使用します。 **注意:** IP 検証は、このコンポーネント呼び出しプロトコルを使用するコンポーネントをサポートしていません。 トップレベル·モジュール·ポート : None

関連情報

CSR スレーブ (18 ページ)

1.4.3.1. コンポーネント呼び出しプロトコルマクロの使用例

次のコード例は、関数引数のコンポーネント呼び出しプロトコルマクロのインクルードがどのようにコンポーネント dut の実装に影響するかを示しています。

図 -3: hls_avalon_streaming_component マクロの実装

```
#include "HLS/hls.h"
hls_avalon_streaming_component
component int dut(hls_conduit_argument int a,
                  hls_avalon_slave_register_argument int b)
{
    return a * b;
}
```

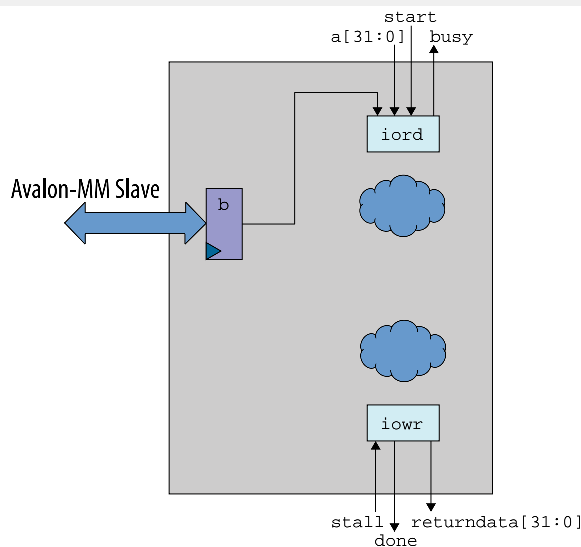


図 -4: Hls_avalon_slave_component マクロの実装

```
#include "HLS/hls.h"
hls_avalon_slave_component
component int dut(hls_conduit_argument int a,
                  hls_avalon_slave_register_argument int b)
{
    return a * b;
}
```

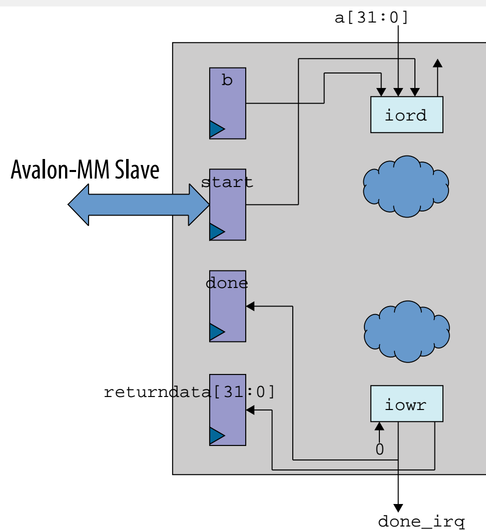
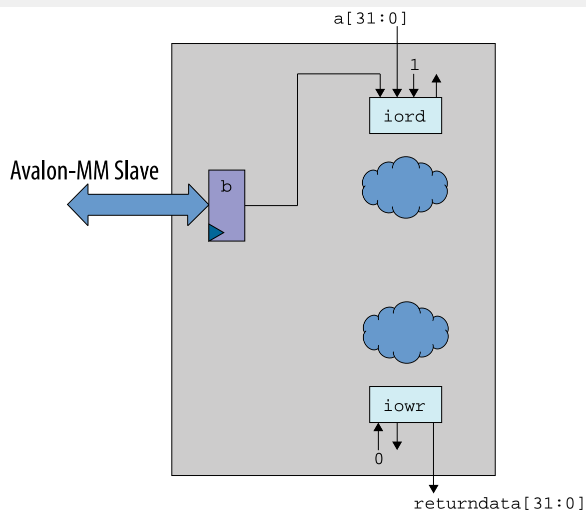


図 -5: hls_always_run_component マクロの実装

```
#include "HLS/hls.h"
hls_always_run_component
component int dut(hls_conduit_argument int a,
                  hls_avalon_slave_register_argument int b)
{
    return a * b;
}
```



1.4.4. スレーブ・インターフェイス

HLS コンパイラーはコンポーネントで使用できる 2 種類の異なるスレーブ・インターフェイスを提供しています。一般的に、より小さいスカラー入力はスレーブレジスターを使用します。これらのアレイをコンポーネント内外へコピーする必要がある場合、大きいアレイはスレーブメモリーを使用する必要があります。

表 10. スレーブ・インターフェイスの型

スレーブ型	実装	関連するスレーブ・インターフェイス	リード / ライト動作	同期化	リード・レイテンシー	インターフェイス・データ幅の制御
Register	レジスター	コンポーネントの制御およびステータスレジスター (CSR) スレーブ	書き込まれた値を読み出し直す場合にのみ使用します。コンポーネントは、これらのレジスターをデータパスから更新できません。	コンポーネントの start 信号を使用する同期化を制御するために `hls_stable_argument` マクロを使用します。	固定値 1	常に 64 ビット
Memory	M20K	コンポーネントの専用スレーブ・インターフェイス	コンポーネントのデータパスからの更新はメモリーで可視化されます。	このメモリーでのすべての変更は、コンポーネントのデータパスに即時に表示されます。したがって、コンポーネントの外部からのリードとライトは、コンポーネント内で実行中のスレッドがない場合にのみ発生します。	固定値はコンポーネントに依存します。 詳細はコンポーネントの `.qsys` ファイルを参照してください。	データ幅はスレーブのデータ型の倍数であり、倍数は内部アクセスを合わせて決定します。

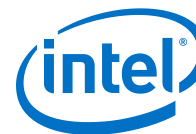
1.4.4.1. CSR スレーブ

コンポーネントは最大で 1 つの CSR スレーブ・インターフェイスを有することができます。`hls_avalon_slave_component` マクロのステータスおよびコントロール・レジスター (つまり、関数コールと戻り値) はこのインターフェイスに実装されています。`hls_avalon_slave_register_argument` としてラベルされた任意の引数はこのメモリースペースに配置されます。結果のメモリーマップは自動的に生成されるヘッダーファイル `_component_name_csr.h` に記述されます。このファイルは、マスターがスレーブとやり取りするための C マクロも提供します。

HLS コパイラーはレジスターに CSR スレーブを実装します。

CSR スレーブを使用するコンポーネントのコード例 :

```
#include "HLS/hls.h"

struct MyStruct {
    int f;
    double j;
    short k;
};


hls_avalon_slave_component
component MyStruct mycomp_xyz (hls_avalon_slave_register_argument int y,
                hls_avalon_slave_register_argument MyStruct struct_argument,
                hls_avalon_slave_register_argument unsigned long long mylong,
                hls_avalon_slave_register_argument char char_arg
                            ) {
    return struct_argument;
}
```



コンポーネント mycomp_xyz で生成される C ヘッダーファイル：

```
/* This header file describes the CSR Slave for the mycomp_xyz component */

#ifndef __MYCOMP_XYZ_CSR_REGS_H__
#define __MYCOMP_XYZ_CSR_REGS_H__



/******************************************************************************/
/* Memory Map Summary                                                         */
/******************************************************************************/

/*
  Register  | Access  |   Register Contents      | Description
  Address   |         |      (64-bits)           |
------------|---------|--------------------------|-----------------------------
        0x0 |       W |         {reserved[62:0], |  Write 1 to signal start to
            |         |              start[0:0]} |               the component
------------|---------|--------------------------|-----------------------------
        0x8 |     R/W |         {reserved[62:0], |      0 - Disable interrupt,
            |         |   interrupt_enable[0:0]} |        1 - Enable interrupt
------------|---------|--------------------------|-----------------------------
       0x10 |  R/Wclr |         {reserved[61:0], | Signals component completion
            |         |               done[0:0], |       done is read-only and
            |         |   interrupt_status[0:0]} | interrupt_status is write 1
            |         |                          |                    to clear
------------|---------|--------------------------|-----------------------------
       0x18 |       R |       {returndata[63:0]} |         Return data (0 of 3)
------------|---------|--------------------------|-----------------------------
       0x20 |       R |     {returndata[127:64]} |         Return data (1 of 3)
------------|---------|--------------------------|-----------------------------
       0x28 |       R |    {returndata[191:128]} |         Return data (2 of 3)
------------|---------|--------------------------|-----------------------------
       0x30 |     R/W |         {reserved[31:0], |                  Argument y
            |         |                 y[31:0]} |
------------|---------|--------------------------|-----------------------------
       0x38 |     R/W |  {struct_argument[63:0]} | Argument struct_argument (0
of 3)
------------|---------|--------------------------|-----------------------------
       0x40 |     R/W | {struct_argument[127:64]} | Argument struct_argument (1
of 3)
------------|---------|--------------------------|-----------------------------
       0x48 |     R/W | {struct_argument[191:128]} | Argument struct_argument
(2 of 3)
------------|---------|--------------------------|-----------------------------
       0x50 |     R/W |           {mylong[63:0]} |             Argument mylong
------------|---------|--------------------------|-----------------------------
       0x58 |     R/W |         {reserved[55:0], |           Argument char_arg
            |         |           char_arg[7:0]} |

NOTE: Writes to reserved bits will be ignored and reads from reserved
      bits will return undefined values.
*/


/******************************************************************************/
/* Register Address Macros                                                    */
/******************************************************************************/

/* Byte Addresses */
#define MYCOMP_XYZ_CSR_START_REG (0x0)
#define MYCOMP_XYZ_CSR_INTERRUPT_ENABLE_REG (0x8)
#define MYCOMP_XYZ_CSR_INTERRUPT_STATUS_REG (0x10)
#define MYCOMP_XYZ_CSR_RETURNDATA_0_REG (0x18)
#define MYCOMP_XYZ_CSR_RETURNDATA_1_REG (0x20)
#define MYCOMP_XYZ_CSR_RETURNDATA_2_REG (0x28)
#define MYCOMP_XYZ_CSR_ARG_Y_REG (0x30)
#define MYCOMP_XYZ_CSR_ARG_STRUCT_ARGUMENT_0_REG (0x38)
```

```
#define MYCOMP_XYZ_CSR_ARG_STRUCT_ARGUMENT_1_REG (0x40)
#define MYCOMP_XYZ_CSR_ARG_STRUCT_ARGUMENT_2_REG (0x48)
#define MYCOMP_XYZ_CSR_ARG_MYLONG_REG (0x50)
#define MYCOMP_XYZ_CSR_ARG_CHAR_ARG_REG (0x58)

/* Argument Sizes (bytes) */
#define MYCOMP_XYZ_CSR_RETURNDATA_0_SIZE (8)
#define MYCOMP_XYZ_CSR_RETURNDATA_1_SIZE (8)
#define MYCOMP_XYZ_CSR_RETURNDATA_2_SIZE (8)
#define MYCOMP_XYZ_CSR_ARG_Y_SIZE (4)
#define MYCOMP_XYZ_CSR_ARG_STRUCT_ARGUMENT_0_SIZE (8)
#define MYCOMP_XYZ_CSR_ARG_STRUCT_ARGUMENT_1_SIZE (8)
#define MYCOMP_XYZ_CSR_ARG_STRUCT_ARGUMENT_2_SIZE (8)
#define MYCOMP_XYZ_CSR_ARG_MYLONG_SIZE (8)
#define MYCOMP_XYZ_CSR_ARG_CHAR_ARG_SIZE (1)

/* Argument Masks */
#define MYCOMP_XYZ_CSR_RETURNDATA_0_MASK (0xffffffffffffffffULL)
#define MYCOMP_XYZ_CSR_RETURNDATA_1_MASK (0xffffffffffffffffULL)
#define MYCOMP_XYZ_CSR_RETURNDATA_2_MASK (0xffffffffffffffffULL)
#define MYCOMP_XYZ_CSR_ARG_Y_MASK (0xffffffff)
#define MYCOMP_XYZ_CSR_ARG_STRUCT_ARGUMENT_0_MASK (0xffffffffffffffffULL)
#define MYCOMP_XYZ_CSR_ARG_STRUCT_ARGUMENT_1_MASK (0xffffffffffffffffULL)
#define MYCOMP_XYZ_CSR_ARG_STRUCT_ARGUMENT_2_MASK (0xffffffffffffffffULL)
#define MYCOMP_XYZ_CSR_ARG_MYLONG_MASK (0xffffffffffffffffULL)
#define MYCOMP_XYZ_CSR_ARG_CHAR_ARG_MASK (0xff)

/* Status/Control Masks */
#define MYCOMP_XYZ_CSR_START_MASK   (1<<0)
#define MYCOMP_XYZ_CSR_START_OFFSET (0)

#define MYCOMP_XYZ_CSR_INTERRUPT_ENABLE_MASK   (1<<0)
#define MYCOMP_XYZ_CSR_INTERRUPT_ENABLE_OFFSET (0)

#define MYCOMP_XYZ_CSR_INTERRUPT_STATUS_MASK   (1<<0)
#define MYCOMP_XYZ_CSR_INTERRUPT_STATUS_OFFSET (0)
#define MYCOMP_XYZ_CSR_DONE_MASK   (1<<1)
#define MYCOMP_XYZ_CSR_DONE_OFFSET (1)


#endif /* __MYCOMP_XYZ_CSR_REGS_H__ */`
```

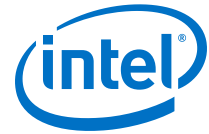
1.4.4.2. スレーブメモリー

デフォルトでは、コンポーネントへのポインター引数はコンポーネントにメモリーマップド・マスター・インターフェイスを生成します。ポインターはコンポーネントの外部でシステムメモリーに格納されているバッファーを参照します。

ポインター引数によるコンポーネントにメモリーマップド・マスター・インターフェイスを生成には、潜在的な欠点が 2 つあります。

- マスター・インターフェイスはシングルポートを有します。コンポーネントに複数のロード・ストアー・サイトがある場合、そのポートの調停により、ストール可能なロジックが作成される可能性があります。
- コンポーネントがインスタンス化されているシステムによっては、コンポーネントが動作中に他のマスターがメモリーバスを使用して、バス上に好ましくないストールを作成する場合があります。

小から中サイズのオンチップ・メモリーでは、スレーブメモリーのポインター引数は上記の欠点にアドレスする可能性があります。ポインター引数をスレーブメモリーとして宣言することは、コンポーネントの中で静的アレイを宣言することと類似しています。静的アレイの宣言とは異なり、内部初期化はメモリーに外部から公開されたポートに置き換えられます。メモリーは現在コンポーネントの内部にあるため、HLS コンパイラーは通常のメモリー最適化をすべて利用して、コンポーネントのアクセスパターン (つまり、バンキング、結合など) に最適化されたメモリー・アーキテクチャーを作成することができます。



HLS コンパーラーはオンチップ・メモリーブロックにスレーブメモリーを実装します。

コンポーネントは多数のメモリー・インターフェイスを有することができます。CSR スレーブ・インターフェイスとともにグループ化されたスレーブレジスター引数とは異なり、スレーブメモリーはコンポーネントで独自のスレーブ・インターフェイスを有します。スレーブメモリー・インターフェイスのデータバス幅はスレーブタイプの幅により決定します。メモリーへの内部アクセスが結合した場合、スレーブメモリー・インターフェイスのデータバス幅はスレーブタイプの幅の倍数になる可能性があります。

1.4.5. ポインター・エイリアシングの回避

`restrict` キーワードを可能な場合にポインター引数に挿入します。`restrict` キーワードを含めることで、インテル HLS コンパイラーは競合しないリードおよびライト動作の間に不要なメモリーの依存関係の作成を回避します。

各反復があるアレイからデータを読み出し、データを同じ物理メモリー内の別のアレイに書き込むループを考慮します。これらのポインター引数に `restrict` キーワードがない場合、コンパイラーは 2 つのアレイ間で依存関係を仮定し、結果としてパイプライン並列化をより少なく抽出する必要があります。

1.5. 標準 C および C++ ライブラリー

HLS コンパイラーは次のヘッダーファイルにより定義された特定の標準 C および C++ 関数の効果的な実装をサポートしています。

math.h

合成されるコンポーネントから `math.h` 関数にアクセスするには、ソースコードに `HLS/math.h` ファイルをインルードします。コンパイラーは、コンポーネントがハードウェア・バージョンの数学関数を呼び出すようにします。

サポートされる `math.h` 関数について詳しくは、この資料の *HLS コンパイラーでサポートされる標準数学関数*の章を参照してください。

stdio.h

合成したコンポーネント関数は一般的に FILE ポインターといった C および C++ 標準ライブラリー関数をサポートしません。

コンポーネントはヘッダーファイル `HLS/stdio.h` をインクルードすることで `printf` をコールできます。このヘッダーはコンパイルの対象に応じて `printf` の動作を変更します。

- x86-64 アーキテクチャー (つまり、`-march=x86-64`) を対象とするコンパイルの場合、`printf` コールが通常通りに動作します。
- FPGA アーキテクチャー (つまり、`-march="`*`<FPGA_family_or_part_number>`*`"`) を対象とするコンパイルの場合、コンパイラーは `printf` コールを削除します。

初めにコードに`#include "HLS/stdio.h"`行を含めずに ハードウェアを FPGA アーキテクチャーにコンパイルすると、次に類似したエラーメッセージが表示されます。

```
$ i++ -march="<FPGA_family_or_part_number>" -I${<path to i++ installation>}/
include --component dut test.cpp
Error: HLS gen_qsys FAILED.
See ./dut.prj/dut.log for details.
```

`fopen` や `printf` などの C および C++ 標準ライブラリー関数は、すべての非コンポーネント関数で通常通りに使用できます。

iostream

合成したコンポーネント関数は一般的に C++ ストリーム・オブジェクト (cout など) のような C++ 標準ライブラリー関数をサポートしません。

コンポーネントはヘッダーファイル HLS/iostream をインクルードすることで cout または cerr をコールできます。このヘッダーはコンパイルの対象に応じて cout と cerr の動作を変更します。

- x86-64 アーキテクチャー (つまり、-march=x86-64) を対象とするコンパイルの場合、cout または cerr コールが通常通りに動作します。
- FPGA アーキテクチャー (つまり、-march="*<FPGA_family_or_part_number>*") を対象とするコンパイルの場合、コンパイラーは cout または cerr コールを削除します。

初めにコードに#include "HLS/iostream"行を含めずに cout または cerr をコンポーネント関数で使用すると、ハードウェアを FPGA アーキテクチャーにコンパイルすると、次に類似したエラーメッセージが表示されます。

```
$ i++ -march="<FPGA_family_or_part_number>" run.cpp run.cpp:5:
Compiler Error: Cannot synthesize std::cout used inside of a component.
HLS Main Optimizer FAILED.
```

重要: ヘッダーファイル HLS/iostream をインクルードすると、cout および cerr への書き込みのみに影響を与えます。他の標準入出力ストリーム・オブジェクトを使用すると、コンパイル時エラーが発生します。標準の入出力ストリーム・オブジェクトを使用するテストベンチ・コードのセクションが多い場合は、HLS/iostream ヘッダーファイルを使用しないでください。

関連情報

HLS コンパイラーでサポートされる標準の数学関数 (52 ページ)

1.6. 静的変数

HLS コンパイラーは C および C++ と同一のセマンティックを有する関数スコープの静的変数をサポートしています。

関数スコープの静的変数はリセット時に指定された値に初期化されます。加えて、これらの変数への変更はコンポーネントの起動時に表示されるため、コンポーネント内の状態の保存に最適です。

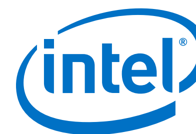
静的変数の初期化のために、コンポーネントは余分なロジックを必要とし、そのロジックがアクティブな間はコンポーネントがリセット状態を終了するまで時間がかかる場合があります。

HLS コンパイラーは単一コンポーネントでのみアクセスされるファイルスコープの静的変数をサポートしています。HLS コンパイラーはこれらのファイルスコープの静的変数をそのコンポーネントにおける関数スコープの静的変数であるかのようにコンパイルします。

静的変数の初期化

典型的なプログラムとは異なり、コンポーネント内の静的変数がいつ初期化されるかを制御することができます。静的変数は、コンポーネントの電源投入時またはリセット時に初期化できます。

コンポーネントの電源投入時に静的変数を初期化することは、プログラムの実行開始後に静的変数の値を変更できない従来のプログラミング・モデルに類似しています。



コンポーネントがリセットされた際に静的変数を初期化すると、電源投入時を含めてコンポーネントがテストベンチから呼び出されるたびに、静的変数が初期化されます。しかしながら、このタイプの静的変数の初期化には余分なロジックが必要です。この余分なロジックは、スタートアップ・レイテンシーおよびコンポーネントに必要な FPGA エリアに影響を与えます。

デフォルト動作は、コンポーネントのリセット時に静的変数を初期化します。

静的変数の初期化は、静的変数の宣言に次のいずれかのキーワードを追加することで明示的に設定できます。

hls_init_on_reset

静的変数値はコンポーネントのリセット後に初期化されます。

次の例で示すように、このキーワードを静的変数の宣言に追加します。

```
static char arr[128] hls_init_on_reset;
```

これは静的変数を初期化するためのデフォルトの動作です。この動作を取るために静的変数の宣言で hls_init_on_reset のキーワードを指定する必要はありません。

例えば、次の例の静的変数もコンポーネントのリセット時に初期化されます。

```
static in arr[64];
```

hls_init_on_powerup

静的変数は、電源投入時にのみ初期化されます。この初期化はメモリー初期化ファイル (.mif) を使用してメモリーを初期化し、これによりリソース使用率とコンポーネントのスタートアップ・レイテンシーを減らします。

次の例で示すように、このキーワードを静的変数の宣言に追加します。

```
static char arr[128] hls_init_on_powerup;
```

一部の静的変数は、静的変数の複雑さのためにこの初期化を利用できない場合があります。このような場合、コンパイラーはエラーを返します。

静的変数の初期化の説明について詳しくは、 *<path to HLS compiler installation>*/examples/tutorials/static_var_init にある static_var_init デザインのチュートリアルをご覧ください。

コンポーネントのリセットについて詳しくは、*インテル HLS コンパイラー・ユーザーガイド*の「リセット動作」を参照してください。

1.7. ループ

HLS コンパイラーはループをパイプライン化して定義するさまざまなコンポーネントのスループットを最大化しようと試みます。

ループのパイプライン化は、HLS コンパイラーがループの後続の反復をパイプライン並列処理方式で実行できるようにします。パイプライン並列処理実行とは、複数のループの反復が実行中の異なる時点で同時に実行することを意味します。また、パイプライン化ループは、生成されたハードウェアの使用を最大限にするのに役立ちます。

一部のケースではパイプライン化ができない場合があります。ループの新しい反復処理は、前の反復処理後 N サイクルまで開始されません。ループ反復処理が開始されるまで待機しなければならないサイクル数は、ループのイニシエーション・インターバル (II) と呼ばれます。一般的にイニシエーション・インターバルの値は 1 が理想的です。II > 1 の場合の通常のケースは、最適化レポートに示されているように、後続のループ反復間にループ伝搬依存性がある場合です。回路はループの新しい反復処理の開始前にこれらのループ伝搬依存性が解決されるのを待機する必要があります。可能であれば、コンパイラーが低い II 値を持つループのスケジュールを生成できるように、ループ伝搬依存性を削減します。

ネストされたループの場合、クリティカルな内側のループが作業の大部分を実行すると、外側のループの II > 1 は重要なパフォーマンス・リミッターとはみなされません。一般的なパフォーマンス・リミッターの 1 つは、HLS コンパイラーが内側のループのトリップ数 (可変内部ループトリップ数など) を静的に計算できない場合です。既知のトリップ数がない場合、コンパイラーは外側のループをパイプライン処理できません。

1.7.1. ループ展開

HLS コンパイラーは複数のループのコピーを展開するための `unroll` プラグマをサポートしています。

コード例 :

```
1  #pragma unroll <N>
2  for (int i = 0; i < M; ++i) {
3      // Some useful work
4  }
```

この例では、*N* はアンロール係数、すなわち、HLS コンパイラーが生成するループのコピーの数をサポートしています。アンロール係数を指定しない場合、HLS コンパイラーはループを完全に展開します。最適化レポートで各ループのアンロールステータスを確認することができます。

注意: ストリームが単一の読み出しと書き込みのみを有するため、HLS コンパイラーはストリーム動作を含んだループを展開できません。

1.7.2. ループ結合

`loop_coalesce` プラグマを使用して、ループ機能に影響を与えずにネスト化ループを 1 つのループに統合するように インテル 高位合成 (HLS) コンパイラーに指示します。ループを結合すると、ループに必要なオーバーヘッドを減らすようにコンパイラーに指示することで コンポーネントのエリア使用率の削減ができます。

ネスト化ループを結合するには、次のようにプラグマを指定します。

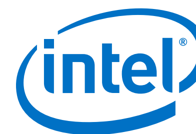
```
#pragma loop_coalesce <loop_nesting_level>
```

<loop_nesting_level> パラメーターはオプションであり、コンパイラーに結合させたいネスト化ループレベルの数を指定する整数です。*<loop_nesting_level>* パラメーターを指定しない場合、コンパイラーはすべてのネスト化ループを結合しようとします。

例えば、次のような一連のネスト化ループを考察します。

```
for (A)
  for (B)
    for (C)
      for (D)
    for (E)
```

ループ (A) の前にプラグマを置くと、これらのネスト化ループレベルは次のようになります。



- ループ (A) はネスト化ループレベル 1
- ループ (B) はネスト化ループレベル 2
- ループ (C) はネスト化ループレベル 3
- ループ (D) はネスト化ループレベル 4
- ループ (E) はネスト化ループレベル 3

指定するループのネストレベルに応じて、コンパイラーはループを別々に結合しようとします。

- `#pragma loop_coalesce 1` を指定する場合、コンパイラーはどのネスト化ループも結合しようとしない。
- `#pragma loop_coalesce 2` を指定する場合、コンパイラーはループ (A) と (B) を結合しようとする。
- `#pragma loop_coalesce 3` を指定する場合、コンパイラーはループ (A)、(B)、(C)、および (E) を結合しようとする。
- `#pragma loop_coalesce 4` を指定する場合、コンパイラーはループ (A) ~ (E) の全ループを結合しようとする。

例

次の簡単な例はコンパイラーが 2 つのループを 1 つのループに結合する方法を示しています。

次のように記述された簡単なネスト化ループを考察します。

```
#pragma loop_coalesce
for(int i = 0; i < N; i++)
  for(int j = 0; j < M; j++)
    sum[i][j] += i+j;
```

コンパイラーは 2 つのループを結合し、次のように記述された 1 つのループのように実行します。

```
int i = 0;
int j = 0;
while(i < N){

  sum[i][j] += i+j;
  j++;

  if(j == M){
    j=0;
    i++;
  }
}
```

1.7.3. ループ・イニシエーション・インターバル (II)

`ii` プラグマを使用して、プラグマ宣言に続くループのループ・イニシエーション・インターバル (II) を設定するように、インテル 高位合成 (HLS) コンパイラーに指示します。コンパイラーがループに対して指定された II を達成できない場合、コンパイルエラーが出力されます。

反復インターバル、または II は、連続するループ反復を開始する間のクロックサイクル数です。II 値が高いほど、次のループ反復までの待機時間が長くなります。

コンポーネントのいくつかのループでは、コンパイラーが選択するよりもより高い II を `ii` プラグマで指定すると、スループットを損なうことなく component の最大動作周波数 (f_{max}) を上げることができます。

ループが次の条件を満たしている場合、ii プラグマを適用するのに適しています。

- ループが コンポーネントのスループットにとって重大ではない場合
- ループの実行時間が含まれる可能性のある他のループと比較して短時間の場合

ループでのループ·イニシエーション·インターバルを指定するには、次のようにループの前にプラグマを指定します。

```
#pragma ii <desired_initiation_interval>
```

<desired_initiation_interval> パラメーターは必須であり、連続するループ·インターバルの間に待機するクロックサイクル数を指定する整数です。

例

コンポーネントに 2 つの異なるパイプライン可能なループがある場合、すなわち、ループ伝搬依存性を持つショートトラニング初期化ループと処理の大半を実行するロングラニング·ループの場合を考慮します。この場合、コンパイラーは初期化ループがデザインの全体的なスループットに及ぼす影響がはるかに小さいことを認識しません。コンパイラーは可能であれば II の値が 1 の両方のループをパイプライン化しようとします。

初期化ループはループ伝搬依存性があるため、生成されたハードウェアにはフィードバック·パスがあります。このようなフィードバック·パスのある II を達成するには、いくつかのクロック周波数を犠牲にする能性があります。メインループのフィードバック·パスに応じて、残りのデザインがより高い周波数で動作する可能性があります。

初期化ループで#pragma ii 2 プラグマを指定すると、このループで II を最適化するアグレッシブが少ないことがコンパイラーに通知されます。少ないアグレッシブの最適化は、コンパイラーがパスをパイプライン化して最大動作周波数 (f_{max}) を制限できるようにし、コンポーネントデザイン全体でより高い f_{max} が達成できます。

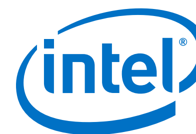
初期化ループは、新しい II で実行するのに長時間を要します。しかしながら、f_{max} が高いため、ロングランニング·ループの実行時間が短くなると、初期化ループの実行時間の長さは増加します。

1.7.4. ivdep プラグマを使用したループ伝搬依存性の削除

コンポーネントをコンパイルする際、HLS コンパイラーはロード / ストアー命令間のデータハザードを回避するためにハードウェアを生成します。特に、現在の反復がロード / ストアー命令の実行を終了する前にコンパイラーが新しいループ反復を開始しないようにするために、読み出しと書き込みの依存性はパフォーマンスを制限します。HLS コンパイラーにはコード内に ivdep プラグマを追加してコンポーネントのループ反復に依存しないことを保証するオプションがあります。ivdep プラグマが存在すると、データハザードを回避するためにコンパイラーがハードウェアを生成することがなくなり、エリアを節約して影響を受けるループの II の値を低下させます。

ループ依存性に関する詳細は、safelen(N)クローズを ivdep プラグマに追加することで取得できます。safelen(N)クローズはループ伝搬依存性のない連続ループ反復の最大数を指定します。例えば、#pragma ivdep safelen(32)はコンパイル時にループ伝搬依存性が導入される前に最大 32 回のループ反復があることをコンパイラーに示します。つまり、#pragma ivdep はこのループの反復間に暗黙的なメモリー依存性がないことを保証し、#pragma safelen(32)はこの反復に依存する可能性のある最も近い反復が 32 回の反復であることを保証します。

ループ内部の特定のメモリーアレイへのアクセスがループ伝搬依存性を引き起こさないように指定するには、コンポーネント·コードのループの前に#pragma ivdep array (*array_name*)行を追加します。ivdep プラグマで指定されたアレイは、ローカルまたはプライベートのメモリーアレイ、もしくはグローバル、ローカル、またはプライベートのメモリーストレージを指すポインター変数でなければ



なりません。指定されたアレイがポインターの場合、ivdep プラグマは指定されたポインターでエイリアスする可能性があるすべてのアレイにも適用されます。ivdep プラグマで指定されたアレイは、構造体のアレイまたはポインターメンバーでもあります。

注意: ivdep プラグマを誤って使用すると、ハードウェアで機能エラーを起こす可能性があります。

使用ケース 1:

ループ内部のメモリーアレイへのすべてのアクセスがループ伝搬依存性を引き起こさない場合、ループの前に#pragma ivdep を追加します。

```
1  // no loop-carried dependencies for A and B array accesses
2  #pragma ivdep
3  for(int i = 0; i < N; i++) {
4      A[i] = A[i + N];
5      B[i] = B[i + N];
6  }
```

使用ケース 2:

#pragma ivdep array (*array_name*)はすべてのアレイアクセスに代わり、特定のメモリーアレイに指定できます。このプラグマはアレイ、ポインター、または構造体のポインターメンバーに適用されます。指定したアレイがポインターの場合、ivdep プラグマは指定したポインターでエイリアスする可能性のあるすべてのアレイに適用します。

```
 1  // No loop-carried dependencies for A array accesses
 2  // Compiler inserts hardware that reinforces dependency constraints for B
 3  #pragma ivdep array(A)
 4  for(int i = 0; i < N; i++) {
 5      A[i] = A[i - X[i]];
 6      B[i] = B[i - Y[i]];
 7  }
 8
 9  // No loop-carried dependencies for array A inside struct
10  #pragma ivdep array(S.A)
11  for(int i = 0; i < N; i++) {
12      S.A[i] = S.A[i - X[i]];
13  }
14
15  // No loop-carried dependencies for array A inside the struct pointed by S
16  #pragma ivdep array(S->X[2][3].A)
17  for(int i = 0; i < N; i++) {
18      S->X[2][3].A[i] = S.A[i - X[i]];
19  }
20
21  // No loop-carried dependencies for A and B because ptr aliases
22  // with both arrays
23  int *ptr = select ? A : B;
24  #pragma ivdep array(ptr)
25  for(int i = 0; i < N; i++) {
26      A[i] = A[i - X[i]];
27      B[i] = B[i - Y[i]];
28  }
29
30  // No loop-carried dependencies for A because ptr only aliases with A
31  int *ptr = &A[10];
32  #pragma ivdep array(ptr)
33  for(int i = 0; i < N; i++) {
34      A[i] = A[i - X[i]];
35      B[i] = B[i - Y[i]];
36  }
```

1.7.5. ループ並行処理の制御

max_concurrency プラグマを使用して、コンポーネントでループの並行処理の増減を制限することができます。ループの並行処理とは、一度にそのループでどの位の反復処理が実行されるかです。デフォルトでは、インテル HLS コンパイラーはコンポーネントがピーク・スループットで動作するため、並行処理を最大化しようと試みます。

ループで最大の並行処理を達成するには、ループが完全にパイプラインされないようにする基となるハードウェアの依存関係を解除するために、ローカルメモリーを複写する必要があります。これは、コンポーネント HLD レポート (report.html)の Details ペインのループ分析レポートで同時実行の最大数が N に制限されていることを示すメッセージとして、確認できます。この場合のローカルメモリーの重複は、ポート数の増加のために複製するメモリーと同一ではありません。

ローカルメモリーの節約のためにいくつかの性能を交換する場合は、ループに#pragma max_concurrency *<N>* を適用します。このプラグマを適用すると、重複ファクターは次の例のようにループに入るスレッド数を変更し、制御します。

```
#pragma max_concurrency 1
for (int i = 0; i < N; i++) {
  int arr[M];
  // Doing work on arr
}
```

また、hls_max_concurrency(N)コンポーネント属性を使用して、コンポーネントの並行処理を制御することも可能です。hls_max_concurrency(N)コンポーネント属性について詳しくは、hls_max_concurrency を使用した並行処理の制御 (6 ページ)を参照してください。

1.8. 任意高精度整数のサポート

Algorithmic C (AC) データ型は、Mentor Graphics®が Apache ライセンスで提供する一連のヘッダーファイルです。Algorithmic C (AC) データ型について詳しくは、*<path to i++ installation>*/include/ref/ac_datatypes_ref.pdf として入手可能な *Mentor Graphics Algorithmic C (AC) Datatypes* を参照してください。

表 11. HLS コンパイラーでの AC データ型の使用

HLS コンパイラーは ac_int および ac_fixed データ型をサポートしています。

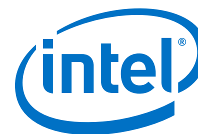
AC データ型	インテル・ヘッダーファイル	説明
ac_int	HLS/ac_int.h	任意幅の整数サポート
ac_fixed	HLS/ac_fixed.h	任意精度の固定小数点サポート

インテル は HLS コンパイラーが以下の理由のためにハードウェアの生成に使用するヘッダーファイルの最適化バージョンを開発しました。

- Mentor Graphics の一部のヘッダーファイルは インテル 高位合成 (HLS) コンパイラーと互換性がない。
- Mentor Graphics のヘッダーファイルを使用する場合、HLS コンパイラーは最適な品質の結果を得られない。

ac_int および ac_fixed データ型の使用による利点：

- より小さいデータ型および回路のさまざまな動作でのエレメント処理が実現できる。
- データ型は、コンテナーサイズを 32 または 64 ビットに昇格せずに、十分なサイズのコンテナーでのさまざまな動作結果を返すための整数昇格の優れたサポートを提供する。



HLS コンパイラーの ac_int および ac_fixed データ型の現在の実装での制限事項：

- 512 ビットの結果の生成に限定された乗算器
- 最大 64 ビットに制限された除算器
- cout 出力ストリームはコンポーネントでサポートされるが、コンポーネントを FPGA アーキテクチャーにコンパイルする際には含まれないようにする必要がある。

関連情報

Mentor Graphics ウェブサイトの AC Datatypes Download のページ

1.8.1. コンポーネントの ac_int データ型の定義

HLS コンパイラー・パッケージはコンポーネント内に任意の精度整数を使用するためのインクルード用の ac_int.h ヘッダファイルが含まれています。

ac_int データ型について詳しくは、任意高精度整数のサポート (28 ページ)を参照してください。

1. コンポーネントに次のように ac_int.h ヘッダーファイルをインクルードします。

```
#include "HLS/hls.h"
#include "HLS/ac_int.h"
```

2. ヘッダーファイルをインクルードした後、次のいずれかの方法で ac_int 変数を宣言します。
   - テンプレートベースの宣言
     - ac_int<N, true> var_name; //Signed N bit integer
     - ac_int<N, false> var_name; //Unsigned N bit integer
   - 63 ビットまでの組み込み typedefine 宣言
     - intN var_name; //Signed N bit integer
     - uintN var_name; //Unsigned N bit integer

1.8.1.1. ac_int データ型での重要な使用情報

ac_int データ型には、インテル HLS コンパイラーインストール・パッケージに含まれている ac_int ドキュメントに記載されている多数の API コールがあります。AC データ型のより詳しい情報は、*<path to i++ installation>*/include/ref ディレクトリー内の ac_datatypes_ref.pdf ファイルを参照してください。

ac_int データ型は整数昇格サポートしています。次の例を考察します。

```
// For a 14-bit addition, the result should be stored in a 15-bit container
int15 s_adder(int14 a, int14 b) {
    return (a + b);
}
```

この場合、加算演算は自動的に 15 ビットの結果を出力します。しかしながら、結果を適切なデータ型に格納するのはユーザーの責任となります。HLS コンパイラーは指定されたコンテナーのサイズに自動的に切り捨てまたは拡張します。

ac_int データ型は C および Verilog を含め、他の言語とはビットシフトが異なります。デフォルトでは、シフト量が符号付きデータ型の場合、下のコード例にように ac_int は負のシフトを許容します。ハードウェアでは、この負のシフトは左シフトと右シフトの両方の実装をもたらします。

```
int14 shift_left(int14 a, int14 b) {
    return (a << b);
}
```

シフトが常に一方向にあるとわかっている場合、効率的なシフト演算子を実装するには、次のようにシフト量を符号なしのデータ型として宣言します。

```
int14 efficient_left_only_shift(int14 a, uint14 b) {
    return (a << b);
}
```

HLS コンパイラー・インストレーション・パッケージはチュートリアル・デザインのいくつかの例が含まれています。推奨される例については、これらのデザインを参照してください。

1.8.2. ac_int データ型のデバッグ用ツール

インテル HLS コンパイラーはコンポーネントのエミュレーション実行時に、コンポーネントのオーバーフローのための ac_int 操作と割り当ての確認に役立つ DEBUG_AC_INT_WARNING と DEBUG_AC_INT_ERROR のツールを提供しています。これらのツールは、コード内の#define マクロとして、あるいは i++コマンドのオプションとして使用します。

これらのツールでコンポーネントにオーバーフローがあることを確認した後、コンポーネントで gdb デバッガーを使用してプログラムを再度実行し、オーバーフローが発生した場所をバックトレースします。

DEBUG_AC_INT_WARNING

DEBUG_AC_INT_WARNING を使用して、コンポーネントの x86 エミュレーション中に ac_int データ型のランタイム・トラッキングを有効にします。このツールはオーバーヘッドを追跡するために追加のオーバーヘッドを使用し、検出されたオーバーフローごとに警告を出します。

DEBUG_AC_INT_WARNING はコンポーネント・コード内のマクロとして使用するか、i++コマンドのオプションとして次のように指定することもできます。

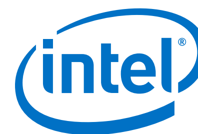
マクロ	`#define DEBUG_AC_INT_WARNING`
コンパイラー・コマンドライン・オプション	`-D DEBUG_AC_INT_WARNING`

DEBUG_AC_INT_ERROR

DEBUG_AC_INT_ERROR を使用して、コンポーネントの x86 エミュレーション中に ac_int データ型のランタイム・トラッキングを有効にします。このツールはオーバーヘッドを追跡するために追加のオーバーヘッドを使用し、最初に検出されたオーバーフローのメッセージを出力して、エラーが発生したコンポーネントを終了します。

DEBUG_AC_INT_ERROR はコンポーネント・コード内のマクロとして使用するか、i++コマンドのオプションとして次のように指定することもできます。

マクロ	`#define DEBUG_AC_INT_ERROR`



コンパイラー・コマンドライン・オプション	`-D DEBUG_AC_INT_ERROR`

1.8.3. コンポーネント内の ac_fixed データ型の定義

HLS コンパイラーのパッケージには任意高精度整数のサポートに関する `ac_fixed.h` ヘッダーファイルが含れています。コンポーネントの `ac_fixed.h` ヘッダーファイルをプレディケート命令に含めます。

`ac_fixed` データ型について詳しくは、任意高精度整数のサポート (28 ページ)を参照してください。

1. 次の方法でコンポーネントに `ac_fixed.h` ヘッダーファイルをインクルードします。

```
#include "HLS/hls.h"
#include "HLS/ac_fixed.h"
```

2. ヘッダーファイルをインクルードした後、次のように `ac_fixed` 変数を宣言します。
   - `ac_fixed<N, I, true, Q, O> var_name; //Signed fixed-point number`
   - `ac_fixed<N, I, false, Q, O> var_name; //Unsigned fixed-point number`

   テンプレートの属性の位置は次のように定義されます。

   N ビット単位の固定小数点の合計の長さ

   I 固定小数点の整数値を表すために使用されるビット数

   Q 小数部を表すために、生成された精度 (小数点以下の桁数) が変数で使用可能なビット数を超える値を扱う方法を決定する量子化モード

   量子化モードおよびその詳細については、`<path to i++ installation>/include/ref/ac_datatypes_ref.pdf` として入手可能な *Mentor Graphics Algorithmic C (AC) Datatypes* の「2.1. Quantization and Overflow」を参照してください。

   O 生成された値が変数で使用可能なビット数よりも多いビットのハンドル方法を決定するオーバーフロー・モード

   オーバーフロー・モードおよびその詳細については、`<path to i++ installation>/include/ref/ac_datatypes_ref.pdf` で入手可能な *Mentor Graphics Algorithmic C (AC) Datatypes* の「2.1. Quantization and Overflow」を参照してください。

1.9. エリアの最小化およびオンチップ・メモリー・アーキテクチャーの制御

HLS コンパイラーは可能な場合は最大スループットを提供しようとします。場合によっては、特に HLS コンパイラーがスループットのためにローカルメモリー・コンフィグレーションを最適化する場合は、より小さいエリアである程度のスループットを交換すると有益な場合があります。 属性をローカル変数に適用してローカルメモリー・システムのジオメトリーをカスタマイズし、回路のエリア使用率を節約できます。これらの属性は `hls.h` ヘッダーファイルで定義され、コード内に含めることができます。

注意: これらの属性はプリミティブおよびオブジェクトに適用できますが、クラスメンバーには適用できません。

表 12. オンチップ・メモリー・アーキテクチャーの制御における属性

HLS 属性	説明
`hls_register`	コンパイラーがレジスターに実装する必要があるローカル変数を指定します。
`hls_memory`	コンパイラーがエンベデッド・メモリーに実装する必要があるローカル変数を指定します。
`hls_numbanks(n)`	ローカル変数を実装しているメモリーが、n が 0 より大きい 2 のべき乗定数である *n* バンクを有することを指定します。
`hls_bankwidth(n)`	ローカル変数を実装しているメモリーが、*n* が 0 より大きい 2 のべき乗定数である *n* バイト幅のバンクを有することを指定します。
`hls_singlepump`	ローカル変数を実装しているメモリーがシングルポンピングする必要があることを指定します。
`hls_doublepump`	ローカル変数を実装しているメモリーがダブルポンピングする必要があることを指定します。
`hls_numports_readonly_writeonly(m,n)`	ローカル変数を実装しているメモリーに、*m* および *n* が 0 より大きい定数である *m* リードポートおよび *n* ライトポートを有することを指定します。
`hls_simple_dual_port_memory`	次の属性を指定することと同等のコンフィグレーションを指定します。 • `hls_singlepump` • `hls_numports_readonly_writeonly(1,1)`
`hls_merge("label", "direction")`	2 つ以上の変数を同じメモリーシステムに実装するように強制します。 *label* は任意の文字列です。マージするすべての変数に同じラベルを割り当てます。 メモリーを幅方向または深さ方向にそれぞれマージするかどうかを判断するために、*direction* を `width` または `depth` のいずれかとして指定します。
`hls_bankbits(b 0 , b 1 , ..., b n )`	バンク選択ビットを形成する{$b_0, b_1, ..., b_n$} を使用して、2^n バンク内を分割するようにメモリーシステムに強制します。 *重要:* $b_0, b_1, ..., b_n$ は連続的であり、正の整数です。 `bank_bits` 属性なしで `numbanks(n)` 属性を指定する場合、バンク選択ビットはデフォルトで最下位ビット (つまり、0, 1, ..., log2(numbanks)-1) になります。

1.9.1. メモリー属性の使用例

コードにさまざまな組み合わせのメモリー属性を組み込むことで、HLS コンパイラーがコンポーネントに実装するメモリー・アーキテクチャーをオーバーライドすることができます。

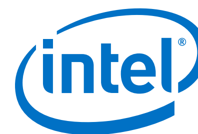
オンチップ・メモリーブロックを節約するための統合メモリーのオーバーライド例

次のコード例は、次のメモリー属性を使用して結合メモリーをオーバーライドする方法を示しています。

- `hls_bankwidth(n)`
- `hls_numbanks(n)`
- `hls_singlepump`
- `hls_numports_readonly_writeonly(m,n)`

オリジナルコードは、64 ビット幅の 256 の深さ (256x64 ビット) のメモリー、つまり、2 つのオンチップ・メモリーブロックを統合します。

```
component unsigned int mem_coalesce(unsigned int raddr,
                                    unsigned int waddr,
                                    unsigned int wdata){
```



```
    unsigned int data[512];
    data[2*waddr] = wdata;
    data[2*waddr + 1] = wdata + 1;
    unsigned int rdata = data[2*raddr] + data[2*raddr + 1];
    return rdata;
}
```

変更されたコードは、32 ビット幅の 512 の深さ (512×32 ビット) で、ストール可能な調停を備えたシンプル・デュアルポート・オンチップ・メモリーブロックを実装しています。

```
component unsigned int mem_coalesce(unsigned int raddr,
                                    unsigned int waddr,
                                    unsigned int wdata){
    //Attributes that stop memory coalescing
    hls_bankwidth(4) hls_numbanks(1)
    //Attributes that specify a simple dual port
    hls_singlepump hls_numports_readonly_writeonly(1,1)
    unsigned int data[512];
    data[2*waddr] = wdata;
    data[2*waddr + 1] = wdata + 1;
    unsigned int rdata = data[2*raddr] + data[2*raddr + 1];
    return rdata;
}
```

注意: hls_singlepump と hls_numports_readonly_writeonly(1,1) 属性の指定の代わりに、hls_simple_dual_port_memory 属性でこのコンフィグレーションを指定することができます。

オンチップ・メモリーブロックを節約するバンクメモリー・オーバーライドの例

次のコード例は、下のメモリー属性を使用してバンクメモリーをオーバーライドする方法を示しています。

- hls_bankwidth(*n*)
- hls_numbanks(*n*)
- hls_singlepump
- hls_doublepump

オリジナルコードは、シングルポンプ・オンチップ・メモリーブロックのバンクを 2 つ作成します。

```
component unsigned short mem_banked(unsigned short raddr,
                                    unsigned short waddr,
                                    unsigned short wdata){
    unsigned short data[1024];

    data[2*waddr] = wdata;
    data[2*waddr + 9] = wdata +1;

    unsigned short rdata = data[2*raddr] + data[2*raddr + 9];

    return rdata;
}
```

バンクメモリーの節約のために、data[1024] の宣言前に次の属性を追加することで、ダブルポンプ・オンチップ・バンクメモリーのバンクを 1 つ実装するオプションがあります。

```
hls_bankwidth(2) hls_numbanks(1)hls_doublepump
unsigned short data[1024];
```

あるいは、次の属性を data[1024] の宣言前に追加することでストール可能な調停のシングルポンプ・オンチップ・メモリーブロックのバンクを 1 つ実装することもできます。

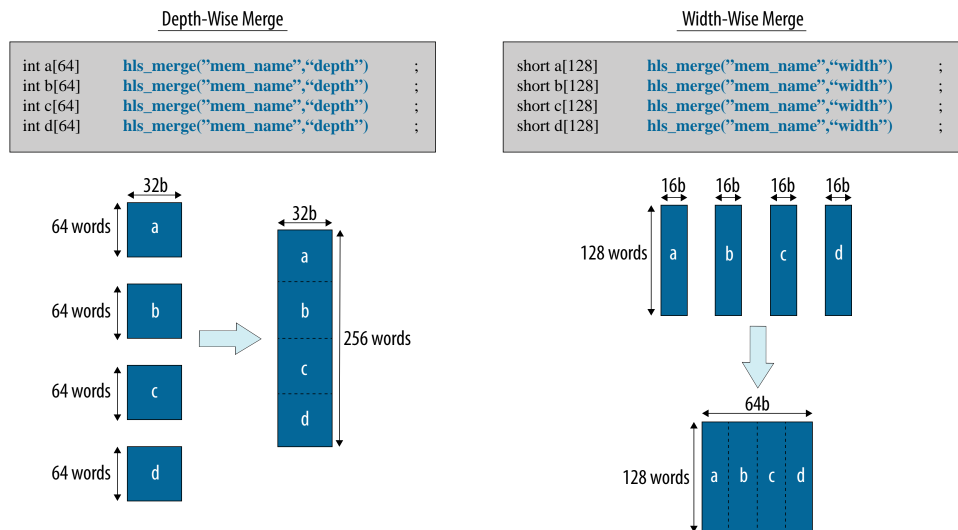
```
hls_bankwidth(2) hls_numbanks(1)hls_singlepump
unsigned short data[1024];
```

1.9.2. hls_merge 属性の使用例

HLS コンパイラーに同じメモリーシステム内の変数を幅方向または深さ方向のいずれかにマージして実装することを強制するオプションがあります。

注意: HLS コンパイラーは不一致のデータ型をサポートします。

図 -6: hls_merge("label", "direction") 属性の実装の概要



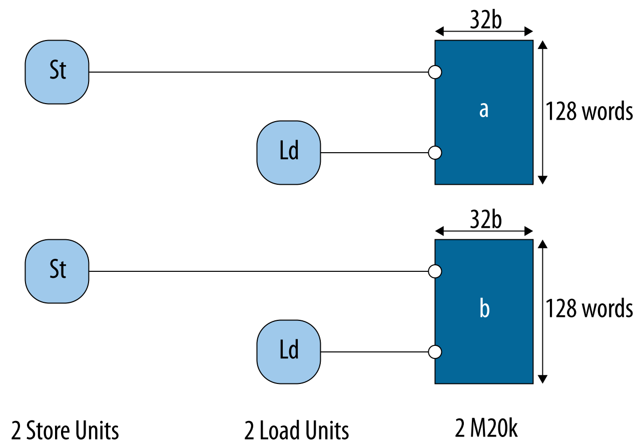
深さに関する hls_merge 属性の実装

次のコンポーネントコードを考察します。

```
component int depth_manual(bool use_a, int raddr, int waddr, int wdata) {
  int a[128];
  int b[128];

  int rdata;

  // mutually exclusive write
  if (use_a) {
    a[waddr] = wdata;
  } else {
    b[waddr] = wdata;
  }

  // mutually exclusive read
  if (use_a) {
    rdata = a[raddr];
  } else {
    rdata = b[raddr];
  }

  return rdata;
}
```

コードはローカルメモリー a と b をそれぞれ独自のロード / ストアー命令を有する 2 つのオンチップ・メモリーブロックとして HLS コンパイラーに命令します。

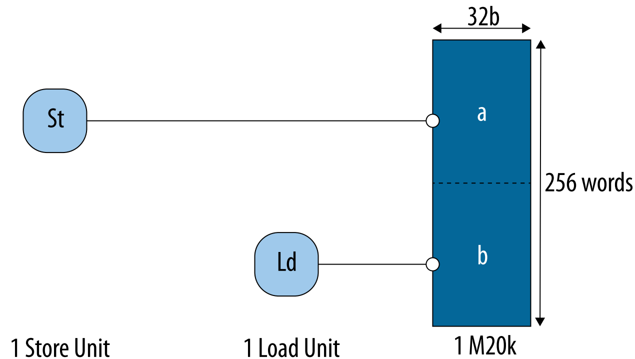
図 -7: コンポーネント depth_manual でのローカルメモリーの実装



ローカルメモリー a と b のロード / ストアー命令が相互に排他的なため、下のサンプルコードに示すようにアクセスをマージすることができ、これによりロード / ストアー命令の数が減り、オンチップ・メモリーブロックの数も半減します。

```
component int depth_manual(bool use_a, int raddr, int waddr, int wdata) {
  int a[128] hls_merge("mem","depth");
  int b[128] hls_merge("mem","depth");;

  int rdata;

  // mutually exclusive write
  if (use_a) {
    a[waddr] = wdata;
  } else {
    b[waddr] = wdata;
  }

  // mutually exclusive read
  if (use_a) {
    rdata = a[raddr];
  } else {
    rdata = b[raddr];
  }

  return rdata;
}
```

図 -8: コンポーネント depth_manual のローカルメモリーの深さ方向のマージ

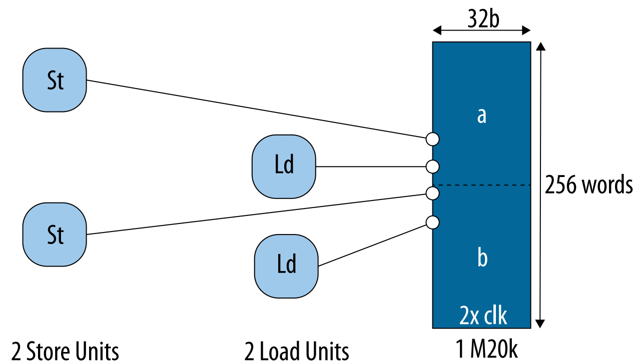


深さに関してローカルメモリーをマージするとメモリーアクセスの効率が低下する場合があります。深さに関するローカルメモリーをマージするかどうかを決定する前に、HLD レポート
(*<result>*.prj/reports/report.html) を参照し、予測されたロード / ストアー命令の数で予測されたメモリー構成を生成しているかを確認する必要があります。下の例では、各メモリーへのロード / ストアー命令は互いに排他的ではないため、HLS コンパイラーはローカルメモリー a と b へのアクセスをマージすべきではありません。

```
component int depth_manual(bool use_a, int raddr, int waddr, int wdata) {
  int a[128] hls_merge("mem","depth");
  int b[128] hls_merge("mem","depth");

  int rdata;

  // NOT mutually exclusive write

  a[waddr] = wdata;
  b[waddr] = wdata;

  // NOT mutually exclusive read

  rdata = a[raddr];
  rdata += b[raddr];

  return rdata;
}
```

この場合、HLS コンパイラーはメモリーシステムを ダブルポンプし、すべてのアクセスに十分なポートを提供する可能性があります。それ以外の場合、アクセスはストールフリーのアクセスを防止するポートを共有する必要があります。

図 -9: 相互に排他的ではないアクセスを有するコンポーネント depth_manual のローカルメモリー

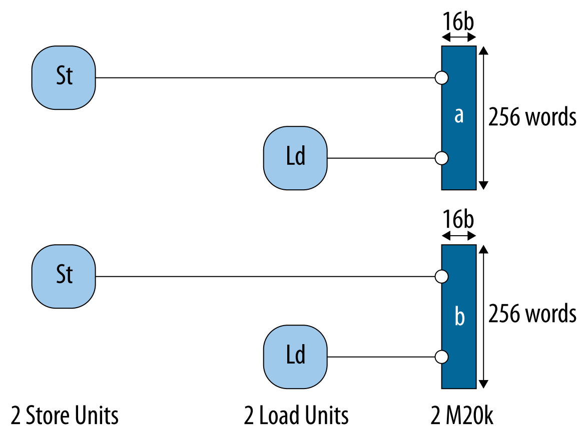


幅に関する hls_merge 属性の実装例

次のコンポーネント・コードを考察します。

```
component short width_manual (int raddr, int waddr, short wdata) {

  short a[256];
  short b[256];

  short rdata = 0;

  // Lock step write
  a[waddr] = wdata;
  b[waddr] = wdata;

  // Lock step read
  rdata += a[raddr];
  rdata += b[raddr];

  return rdata;
}
```

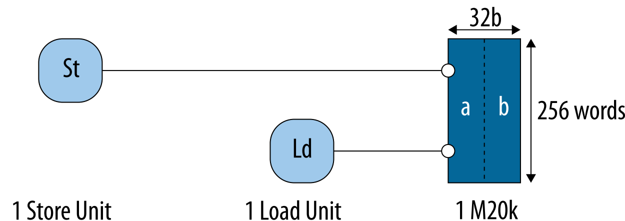
図 -10: コンポーネント width_manual のローカルメモリーの実装



この場合、HLS コンパイラーは、ロード / ストアー命令は同じアドレスへのアクセスであるため、ロード / ストアー命令をローカルメモリー a と b にマージすることができ、次のように統合されます。

```
component short width_manual (int raddr, int waddr, short wdata) {

  short a[256] hls_merge("mem","width");
  short b[256] hls_merge("mem","width");

  short rdata = 0;

  // Lock step write
  a[waddr] = wdata;
  b[waddr] = wdata;

  // Lock step read
  rdata += a[raddr];
  rdata += b[raddr];

  return rdata;
}
```

図 -11: コンポーネント width_manual でのローカルメモリーの幅方向のマージ



1.9.3. hls_bankbits 属性での使用例

コンポーネント·コードに hls_bankbits(*b0*, *b1*, ..., *bn*) 属性を含むことにより、特定のアドレスビットにローカルメモリーをバンクするように HLS コンパイラーに指示するオプションがあります。

{b0, b1, ... ,bn} 引数は HLS コンパイラーがバンク選択ビットで使用すべきワードアドレス·ビットを参照します。hls_bankbits(*b0*, *b1*, ..., *bn*) 属性を指定することは、バンク数が 2^(*バンクビット数*) と同等であることを意味します。

注意: 現在、hls_bankbits(*b0*, *b1*, ..., *bn*) 属性は連続するバンクビットのみをサポートしています。

hls_bankbits 属性の実装例

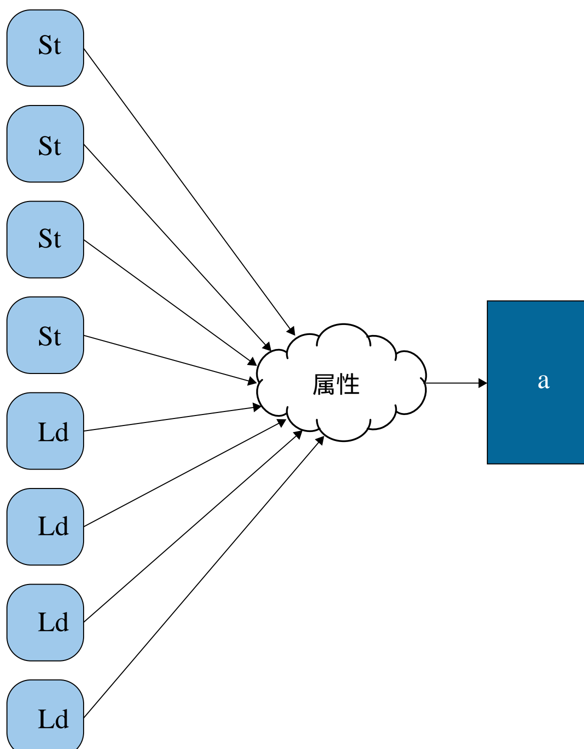
次のコンポーネント·コードを考察します。

```
component int bank_arb_consecutive_multidim (int raddr, int waddr, int wdata,
int upperdim) {

  int a[2][4][128];

  #pragma unroll
  for (int i = 0; i < 4; i++) {
    a[upperdim][i][(waddr & 0x7f)] = wdata + i;
  }

  int rdata = 0;

  #pragma unroll
```

```
  for (int i = 0; i < 4; i++) {
    rdata += a[upperdim][i][(raddr & 0x7f)];
  }

  return rdata;
}
```

下の図に示すように、このコード例は複数のロード / ストアー命令を生成します。ただし、命令数に対するポート数が不足しているため、ストール可能なアクセスと II の値は 66 になります。

図 -12: コンポーネント bank_arb_consecutive_multidim のローカルメモリーへのアクセス

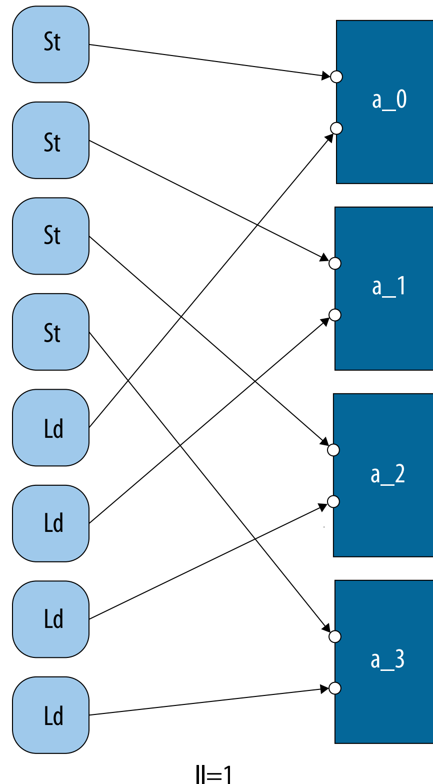


hls_bankbits 属性を指定することで、ロード / ストアー命令がローカルメモリーにアクセスする方法を制御できます。変更されたコード例と下の図に示すように、ローカルメモリー a にアクセスするたびに一定のバンク選択ビットを選択すると、ロード / ストアー命令の各ペアはメモリーバンクの 1 つに接続するためのみに必要になります。このアクセスパターンは II の値を 66 から 1 に大幅に減少させます。

```
component int bank_arb_consecutive_multidim (int raddr, int waddr, int wdata,
int upperdim) {

  int a[2][4][128] hls_bankbits(8,7);

  #pragma unroll
  for (int i = 0; i < 4; i++) {
    a[upperdim][i][(waddr & 0x7f)] = wdata + i;
  }

  int rdata = 0;

  #pragma unroll
```

```
  for (int i = 0; i < 4; i++) {
    rdata += a[upperdim][i][(raddr & 0x7f)];
  }

  return rdata;
}
```

図 -13: hls_bankbits 属性のコンポーネント bank_arb_consecutive_multidim のローカルメモリーへのアクセス

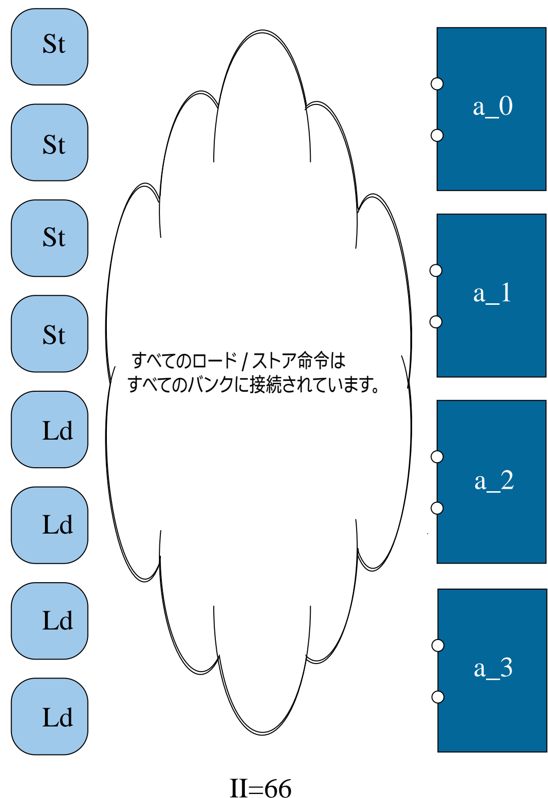


hls_bankbits 属性でワードアドレス・ビットを指定する場合、バンク選択ビットの結果がローカルメモリーへのアクセスごとに一定であることを確認します。下の図に示すように、ローカルメモリーのアクセスパターンは、選択されたバンク選択ビットがアクセスごとに一定であるという保証がないようなものです。結果として、ロード / ストアー命令の各ペアは、すべてのローカルメモリー・バンクに接続する必要があり、ストール可能なアクセスとなります。

```
component int bank_arb_consecutive_multidim (int raddr,
                                             int waddr,
                                             int wdata,
                                             int upperdim){

  int a[2][4][128] hls_bankbits(5,4);

  #pragma unroll
  for (int i = 0; i < 4; i++) {
    a[upperdim][i][(waddr & 0x7f)] = wdata + i;
  }

  int rdata = 0;

  #pragma unroll
  for (int i = 0; i < 4; i++) {
```

```
    rdata += a[upperdim][i][(raddr & 0x7f)];
  }

  return rdata;
}
```

図 -14: hls_bankbits 属性のコンポーネント bank_arb_consecutive_multidim のローカルメモリーへのストール可能なアクセス

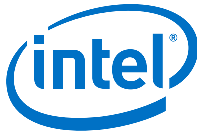


1.10. 改訂履歴

表 13. インテル HLS コンパイラー・リファレンス・マニュアルの改訂履歴

日付	バージョン	変更内容
2017 年 6 月	2017.06.23	• 項静的変数 (22 ページ)静的変数初期化およびその制御方法に関する情報を追加し、更新。 • メンテナンスによる修正および変更。
2017 年 6 月	2017.06.09	• コンポーネントの ac_int データ型の定義 (29 ページ) ac_int.h を含める方法を変更し、更新。 • 項任意高精度整数のサポート (28 ページ) Algorithmic C データ型のサポートを明確にし、更新。 • --device コンパイラー・オプションの全内容を削除。このオプションは -march コンパイラー・オプションの関数に置き換え済。-march コンパイラー・オプションの変更した関数については#unique_2/unique_2_Connect_42_section_N10130_N1001B_N10001 (4 ページ)詳細を参照。

continued...



日付	バージョン	変更内容
		• CSR スレーブ (18 ページ)コンポーネント mycomp_xyz で生成される C ヘッダーファイルを更新。 • 項コンポーネント・インターフェイス (7 ページ)コンポーネント・インターフェイスの構造体に関する情報を追加。 • 項標準 C および C++ ライブラリー (21 ページ) iostream の動作を訂正。
2017 年 2 月	2017.02.03	• *スカラー・パラメーターおよび Avalon Streaming インターフェイスの Avalon-ST インターフェイスで使用可能なスカラー・パラメーター*の表で情報を更新。 • *ポインター・パラメーター、参照パラメーター、および Avalon Memory-Mapped マスター・インターフェイスの Avalon-MM インターフェイスのコンフィグレーションで使用可能なテンプレート引数*の表での情報を更新 • グローバル定数、ポインター、および変数のエリア使用率の最適化についての情報を*グローバル変数*に追加。
2016 年 11 月	2016.11.30	• *「HLS コンパイラー・コマンド・オプション」のコンパイルをカスタマズするコマンドオプション*の表で次の方法を変更。 — 未使用の--rtl-only コマンドオプションとその説明を削除。 — --simulator *<name>* コマンドオプションおよびその説明を追加。 — HLS コンパイラーは Windows および Linux の両方でデフォルトでレポートにデバッグ情報を生成するため、-g コマンドオプションを削除。また、Linux の最終バイナリーではデバッグデータはデフォルトにより使用可能。 • *ポインター・パラメーター、参照パラメーター、および Avalon Memory-Mapped マスター・インターフェイス*で、表に altera::align<*value*> テンプレート引数に関する情報を追加。 • *メモリーマップド・マスター・テストベンチ・コントラクター*および*メモリーマップド・マスター・テストベンチ作成の暗黙的および明示的な例*のトピックを追加。 • *コンポーネント呼び出しプロトコルマクロの使用例*で、対応するマクロを使用したコード例のコンポーネント呼び出しプロトコルの属性を置き換え済。 • 次のセクションで、コードスニペットに #include "HLS/hls.h" 行を追加。 — *インターフェイス合成マクロの使用例* — *コンポーネント呼び出しプロトコルマクロの使用例* • 項*任意高精度整数のサポート*を追加し、ac_int データ型およびインテル提供の ac_int.h ヘッダーファイルを導入し、次の項目を記述。 — *任意高精度整数のサポートにおけるコンポーネントの ac_int データ型の定義* — *ac_int データ型での重要な使用情報* • 項*エリアの最小化およびオンチップ・メモリー・アーキテクチャーの制御*の内容を更新。 — numreadports(*n*)および numwriteports(*n*)エントリーを*オンチップ・メモリー・アーキテクチャーの制御*の表の属性で 1 つの numports_readonly_writeonly(*m*,*n*) エントリーに置き換え。 — hls_simple_dual_port_memory マクロに関する情報を追加。 — hls_merge ("*label*","*direction*") および hls_bankbits(*b0*, *b1*, ..., *bn*) の属性に関する情報を追加。 • hls_merge("*label*","*direction*")および hls_bankbits(*b0*, *b1*, ..., *bn*) の属性での使用例を追加。 • hls_bankbits と hls_bankwidth 属性を既存のメモリーアドレスの導出を説明するために、*hls_bankbits 仕様とメモリー・アドレス・ビット間の関係性*を追加。
2016 年 9 月	2016.09.12	初版

A. HLS コンパイラー・クイック・リファレンス

表 14. i++ コマンドライン引数

コマンドオプション	デフォルト値	説明	例
-c		オブジェクト・ファイルの前処理、解析、生成。	
--clock <clock target>	240 MHz	指定したクロック周波数または周期で RTL を最適化する。	`i++ -march="Arria 10" test.cpp --clock 100MHz` `i++ -march="Arria 10" test.cpp --clock 10ns`
--component *<component name>*		RTL に合成するための関数名のコンマ区切りのリスト。	
-D*<macro>*[=*<val>*]		値としての *<val>* での *<macro>* を定義する。	
--fpc		可能時に中間丸めと変換を削除する。	
--fp-relaxed		浮動小数点演算動作の順序を緩和する。	
-ghdl		シミュレーションでのすべての HDL 信号の完全なデバッグの可視化およびログ付けを有効にする。	
-L*<dir>*		-i を検索できるディレクトリーのリストにディレクトリー *<dir>* を追加する。	
-l*<library>*		リンクキングの際にライブラリー名 *<library>* を検索する。	
-march=[x86-64 | "<FPGA_family>" | "<FPGA_part_number>"	x86-64	エミュレーター・フロー (x86-64)、または指定された FPGA ファミリーもしくは FPGA パートナンバーのコードを生成する。	
--promote-integers		余分な FPGA リソースを使用し g++ 整数昇格を模倣する。	
--quartus-compile		Quartus Prime から生成された HDL を実行する。	
--simulator none		テストベンチなしでコンポーネントの RTL を生成する。	

Intel Corporation. 無断での引用、転載を禁じます。Intel、インテル、Intel ロゴ、Altera、ARRIA、CYCLONE、ENPIRION、MAX、NIOS、QUARTUS および STRATIX の名称およびロゴは、アメリカ合衆国および/ またはその他の国における Intel Corporation の商標です。インテルは FPGA 製品および半導体製品の性能がインテルの標準保証に準拠することを保証しますが、インテル製品およびサービスは、予告なく変更される場合があります。インテルが書面にて明示的に同意する場合を除き、インテルはここに記載されたアプリケーション、または、いかなる情報、製品、またはサービスの使用によって生じるいっさいの責任を負いません。インテル製品の顧客は、製品またはサービスを購入する前、および、公開済みの情報を信頼する前には、デバイスの仕様を最新のバージョンにしておくことをお勧めします。
*その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

ISO 9001:2015 登録済

表 15. HLS ヘッダーファイル

ヘッダーファイル	説明
`#include "HLS/hls.h"`	コンポーネントの識別と明示的なインターフェイスで必要である。
`#include "HLS/math.h"`	動作システム用の `math.h` からのすべての数学関数を含む。
`#include "HLS/extendedmath.h"`	`math.h` ではない追加の数学関数を含む。
`#include "HLS/ac_int.h"`	`ac_int` ヘッダーファイルのインテル HLS バージョンであり、最適化された任意のサイズの整数サポートを提供する。
`#include "HLS/ac_fixed.h"`	`ac_fixed` ヘッダーファイルのインテル HLS バージョンであり、任意の長さの固定小数点サポートを提供する。

表 16. HLS キーワード

キーワード	デフォルト値	説明	例
`component`		関数がコンポーネントであることを示すために使用される。	`component void foo()`

表 17. HLS シミュレーション API (テストベンチのみ)

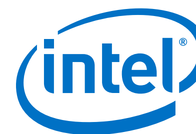
<table>
<tr><th>関数</th><th>説明</th><th>例</th></tr>
<tr><td>ihc_hls_enqueue(/*ptr to return type*/, /*function arguments*/)</td><td>この関数は HLS コンポーネントの 1 つの呼び出しをエンキューする。戻り値は、戻り型へのポインターである最初の引数に格納される。コンポーネントは ihc_hls_component_run_all()関数が呼び出されるまで実行されない。</td><td rowspan="4">component int foo(int val) {
  // function definition
}

component void bar (int val) {
  // function definition
}
int main() {
  // ……
  int input = 0;
  int res;
  ihc_hls_enqueue(&res, &foo, input);
  ihc_hls_enqueue_noret(&bar, input);
  input = 1;
  ihc_hls_enqueue(&res, &foo, input);
  ihc_hls_enqueue_noret(&bar, input);
  ihc_hls_component_run_all(foo);
  ihc_hls_component_run_all(bar);
  }</td></tr>
<tr><td>ihc_hls_enqueue_noret(/*function arguments*/)</td><td>この関数は HLS コンポーネントの 1 つの呼び出しをエンキューする。この機能は HLS コンポーネントの戻り値が void の場合に使用される。コンポーネントは ihc_hls_component_run_all() 関数が呼び出されるまで実行されない。</td></tr>
<tr><td>ihc_hls_component_run_all(/*function name*/)</td><td>この関数は HLS コンポーネントの名前を受け入れる。実行時、コンポーネントのエンキューされたすべての呼び出しは、コンポーネントが新しい呼び出しをできるだけ早く受け入れられるように、HDL シミュレーターのコンポーネント内にプッシュされる。</td></tr>
<tr><td>int ihc_hls_sim_reset(void)</td><td>この関数は自動シミュレーション中にリセット信号をコンポーネントに送る。リセットが実行されると 1 を返し、それ以外の場合は 0 を返す。</td></tr>
</table>

表 18. HLS コンポーネント·メモリー属性

属性	デフォルト値	説明
`hls_register`	アクセスパターンに基づいた値	コンポーネント内の変数 / アレイを強制的にレジスターとして実装する。
`hls_memory`	アクセスパターンに基づいた値	コンポーネント内の変数 / アレイを強制的にエンベデッド·メモリーとして実装する。
`hls_singlepump`	アクセスパターンに基づいた値	ローカル変数を実装しているメモリーがシングルポンプする必要があることを指定する。

continued...



属性	デフォルト値	説明
`hls_doublepump`	アクセスパターンに基づいた値	ローカル変数を実装しているメモリーがダブルポンプする必要があることを指定する。
`hls_numbanks(N)`	アクセスパターンに基づいた値	ローカル変数を実装しているメモリーが、*N* が 0 より大きい 2 のべき乗の定数である *N* バンクを有する必要があることを指定する。
`hls_bankwidth(N)`	アクセスパターンに基づいた値	ローカル変数を実装しているメモリーが、*N* が 0 より大きい 2 のべき乗の定数である *N* バイト幅のバンクを有する必要があることを指定する。
`hls_bankbits(` b_0 , b_1 , ..., b_n `)`	バンク数に基づくアドレスの最下位ビット	{b_0 , b_1 , ..., b_n } がバンク選択ビットを形成するように、メモリーシステムを強制的に 2^n のバンクに分割するようにする。 *重要:* b_0 , b_1 , ..., b_n は連続した正の整数である必要がある。
`hls_numports_readonly_writeonly(M, N)`	アクセスパターンに基づいた値	ローカル変数を実装しているメモリーが、*M* および *N* が 0 より大きい定数である *M* リードポートと *N* ライトポートを有することを指定する。
`hls_simple_dual_port_memory`		`hls_singlepump` および `hls_numports_readonly_writeonly(1,1)` マクロの両方の存在により定義される構成を指定する。
`hls_merge("mem_name", "depth")`		2 つ以上のローカル変数を単一マージド・メモリーシステムとして深さ方向にコンポーネント・メモリー内に実装できることを可能にする。
`hls_merge("mem_name", "width")`		2 つ以上のローカル変数を単一マージド・メモリーシステムとして横方向にコンポーネント・メモリー内に実装できることを可能にする。
`hls_init_on_reset`	静的変数でのデフォルト動作の値	この属性は、コンポーネント・リセット信号がアサートされると、コンポーネント内部の静的変数を強制的にリセットする。これはコンポーネント・メモリーへの追加のライトポートの実装を必要とし、コンポーネントのリセット時にパワーアップ・レイテンシーを増加できる。
`hls_init_on_powerup`		この属性は、FPGA がプログラムされている際に静的変数を実装するコンポーネント・メモリーを電源投入に設定するように設定する。コンポーネントがリセットの場合、コンポーネント・メモリーは静的初期化値にリセットが戻されない。

表 19. HLS ループプラグマ

プラグマ	説明	例

`#pragma ii <N>`	これが適用されるループに <*N*> の値が 0 より大きい II の <*N*> を強制する。	`#pragma ii 2`
		`for (int i = 0; i < 8; i++) {`
		`  // Loop body`
		`}`

`#pragma ivdep safelen(<N>) array(<array_name>)`	このループの反復間のメモリー依存をコンパイラーに無視するように通知する。アレイの名前を指定するオプションの引数を受け入れることができる。これを指定しない場合、すべてのコンポーネント・メモリー依存は無視される。`safelen` パラメーターはオプションであるが、推奨されている。ループ内のロード / ストアー間の依存距離を指定	`#pragma ivdep`
		`safelen(2)`
		`for (int i = 0; i`

continued...

プラグマ	説明	例

	する。コンパイラーはこの情報を使用して依存関係に適切な注釈を付ける。safelen を含まないことが安全である唯一のケースは、依存距離が無限大 (つまり、実際の依存関係がない) 場合である。	< 8; i++) {
		 // Loop body
		}

#pragma loop_coalesce <N>	コンパイラーはこのループ内にネストされたすべてのループを 1 つのループに結合しようと試みる。このプラグマは一緒に結合するループの数を示すオプションの <N> を受け入れる。	#pragma
		loop_coalesce
		for (int i = 0; i
		< 8; i++) {
		 for (int j = 0; j
		< 8; j++) {
		 // Loop body
		 }
		}

#pragma unroll <N>	このプラグマはループを完全に、または <N> 回 (<N> はオプションで正の整数値) で、アンロールする。	#pragma unroll
		for (int i = 0; i
		< 8; i++) {
		 // Loop body
		}

#pragma max_concurrency <N>	このプラグマはいつでも同時に実行できるループの反復の数を制限する。主にループのスループットを向上させるためにコンポーネント・メモリーが複製されている場合に役立つ (ループ分析ペイン内のループの詳細ペインに記載されている)。これはコンポーネント・メモリーのスコープ (宣言またはアクセスパターンによる) がこのループに制限されている場合にのみ発生する。このプラグマを追加すると、いくつかのスループットを犠牲にして消費するループのエリアを節約できる。	// Without this
		pragma, multiple
		copies
		// of the array
		"arr"
		#pragma
		max_concurrency 1
		for (int i = 0; i
		< 8; i++) {
		 int arr[1024];
		 // Loop body
		}

表 20. HLS コンポーネント属性

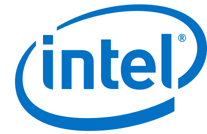
コンポーネント属性	説明	例

hls_max_concurrency(<N>)	コンポーネント・メモリーがインスタンス化されている場合、(つまり、静的変数ではない)、コンポーネント・メモリが一インスタンス化されている場合、(およびコンポーネント・メモリーのスコープがループだけでなくコンポーネント全体である場合)、単一のコピーのみがインスタンス化される。これによりコンポーネントがパイプラインされ、複数の呼び出しが一度に実行される際にパフォーマンスを制限することができる。この属性はコンポーネントに追加でき、メモリーの複数のコピーが作成され、メモリー内の異なるスペースで異なる呼び出しが並列実行できるようになる。異なるコピーがすべて同じメモリーシステムにマップされることは、レプリケーションが深刻であることに注意する必要がある。	hls_max_concurrenc
		y(2)
		void
		foo(ihc::stream_in<
		int> &data_in,
		
		ihc::stream_out<int
		> &data_out) {
		 int arr[N];
		 for (int i = 0; i
		< N; i++) {
		 arr[i] =
		data_in.read();
		 }
		 // Operate on the
		data and modify in
		place
		 for (int i = 0; i
		< N; i++) {
		
		data_out.write(arr[
		i]);
		 }
		}

表 21. HLS デフォルト・インターフェイス

デフォルト・インターフェイス	説明
コンポーネント呼び出しと戻り値	コンポーネント呼び出しインターフェイスは、start と busy コンデュイットから成るストリーミング・インターフェイスとして実装される。

continued...



デフォルト・インターフェイス	説明
	コンポーネント戻り値インターフェイスも done と stall 信号を含むストリーミング・インターフェイスとして実装される。
スカラー引数	スカラー引数はコンポーネント呼び出しインターフェイスに同期されるコンデュイットとして実装される。
ポインター引数	ポインター引数はデフォルトのパラメーター化での `mm_master` インターフェイスとして実装される。 デフォルトでは、ベースアドレスはスカラー引数として扱われるため、コンポーネント呼び出しインターフェイスに同期されるコンデュイットとして実装される。 メモリーマップド・インターフェイスもコンポーネント上に公開される。

表 22. HLS コンポーネント呼び出しインターフェイス引数

コンポーネント呼び出しインターフェイス引数	説明	例
`restrict`	`restrict` キーワードを含めることで、可能な場合に インテル HLS コンパイラーが非競合のリードおよびライト動作間の不要なメモリー依存関係の作成を回避できる。	`component void foo (restrict int *ptr1, restrict int *ptr2);`
`hls_avalon_streaming_component` これはデフォルトのコンポーネント呼び出しインターフェイスである。	この属性は関数コールとリターンストリームの両方のストリーミング・プロトコルに従う。コンポーネントは start 信号がアサートされて busy 信号がディアサートされると、不安定引数を消費する。コンポーネントは done 信号がアサートされると戻り値データを生成する。 トップレベルのモジュールポート : 関数コール—start、busy 関数リターン—done、stall	`hls_avalon_streaming_component void foo(/*component arguments*/)`
`hls_avalon_slave_component`	start、done、および returndata (可能な場合) 信号はコンポーネントのスレーブ・メモリー・マップにレジスターされる。詳しくは、CSR スレーブの項を参照。 トップレベルのモジュールポート : Avalon-MM スレーブ・インターフェイスおよび irq_done 信号	`hls_avalon_slave_component void foo(/*component arguments*/)`
`hls_always_run_component`	Start 信号はコンポーネント内で 1 に結び付けられる。done 信号は出力されない。コンポーネントのデータパスが入力 / 出力用の明示的なストリームでのみ依存する場合にこのプロトコルを使用する。 注意 : IP 検証ではこのコンポーネント呼び出しプロトコルを使用するコンポーネントはサポートされていない。 トップレベルのモジュールポート : なし	`hls_always_run_component void foo(/*component arguments*/)`

表 23. HLS コンポーネント引数マクロ

コンポーネント引数マクロ	説明	例
`hls_conduit_argument` これはスカラー引数のデフォルト・インターフェイスである。	コンパイラーはコンポーネントの呼び出し (start と busy) に同期する入力コンデュイットとして引数を実装する。	`void foo( hls_conduit_argument int b)`

continued...

コンポーネント引数マクロ	説明	例
hls_avalon_slave_register_argument	コンパイラーは Avalon-MM スレーブ・インターフェイス上で読み書きできるレジスターとして引数を実装する。引数は、コンデュイット実装と同様にコンポーネントのパイプラインにリードされる。実装は、start および busy インターフェイスに同期する。	void foo(hls_avalon_slave_register_argument int b)
hls_avalon_slave_memory_argument(N)	コンパイラーはオンチップ・メモリーブロックに引数を実装し、これは専用スレーブ・インターフェイス上で読み書きできる。生成されたメモリーは他のすべての内部コンポーネント・メモリー（つまり、バンキング、結合など）と同じアーキテクチャーの最適化を行う。 コンパイラーが静的結合最適化を実行すると、スレーブ・インターフェイスのデータ幅は結合された幅になる。この属性は、ポインター引数でのみ適用する。 コンポーネントのデータパスで行われたこの引数の値の変更はこのレジスターに反映されない。	component void foo(hls_avalon_slave_memory_argument(128) *a)
hls_stable_argument	stable 引数は、ライブデータがコンポーネントにある間（つまり、パイプライン化関数呼び出しの間）は変更されない引数である。 コンポーネント実行中の stable 引数の変更は、未定義の動作を引き起こす。つまり、stable 引数の使用のたびに古い値または新しい値が使用される可能性があるが、一貫性の保証はない。同じ呼び出しでの同じ変数では複数の値が現れる場合がある。stable 引数を使用すると、適切な場合にデザインの多くのレジスター数を節約できる場合がある。 stable 引数はコンデュイット、mm_master インターフェイス、slave_registers で使用できる。	component int dut(hls_stable_argument int a, hls_stable_argument int b) { return a * b;}

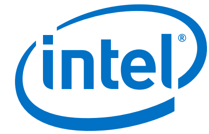
表 24. HLS ストリーミング用入力インターフェイス

入力インターフェイス	デフォルト値	有効値	説明	例

stream_in 宣言				void foo(ihc::stream_in<int> &a) {
				 int x = a.read(); // Blocking read
				}
				void foo_nb(ihc::stream_in<int> &a) {
				 bool success = false;
				 int x = a.tryRead(&success); // Blocking read
				 if (success) {
				 // x is valid
				 }
				}
				
				int main() {
				
				ihc::stream_in<int> a;
				
				ihc::stream_in<int> b;
				 for (int i = 0; i < 10; i++) {
				 a.write(i);
				 b.write(i);
				 }

ihc::stream_in<datatype>			コンポーネントへのストリーミング用の入力インターフェイスであり、テストベンチはこのバッファーをコンポーネントに送信する前に配置する必要がある。	
ihc::buffer	0	正の整数値	ストリームに関連する入力データ上の FIFO バッファーのワード単位での容量。 このパラメーターは入力ストリーム上でのみ使用可能。 このパラメーターは void foo (ihc::stream_in<int, ihc::buffer<16 > &a); など、関数引数に追加される。	
ihc::readylatency	0	正の整数値（0-8 の間）	リード信号がディアサートされてから入力ストリームが新しい入力を受け入れられなくなるまでのサイクル数。概念的には、このパラメーターはストリームと関連するデータの入力 FIFO バッファー上の almost ready レイテンシーとして確認することができる。	
ihc::bitsPerSymbol	データ型サイズ	データ型サイズを均等に分割する正の整数値	データがデータパス上のシンボルにどのようにブレークされるかを記述する。データは常にリトル・エンディアンの順に中断される。	
ihc::usesPackets	false	true または false	ストリーム・インターフェイス上の startofpacket と endofpacket 信号を公開する。これは、パケットベースの読み出し / 書き込みによりアクセス可能。	
ihc::usesValid	true	true または false	有効な信号がストリーム・インターフェイス上に存在するかどうかを制御する。false の場合は、アップストリーム・ソースは ready がアサートされるサイクルごとに有効なデータを提供する必要がある。 これはストリームリード呼び出しを tryRead に変更し、success を常に true を保つことと同じである。	

continued...



入力インターフェイス	デフォルト値	有効値	説明	例
			true に設定する場合は、buffer と readyLatency は 0 である必要がある。	
stream_in 関数 API				
T read()			コンポーネント内から使用されるブロッキング・リードコール。	
T read(bool& sop, bool& eop)			usesPackets<true>が設定されている場合にのみ有効である。 sideband 信号によるブロッキング・リード。	

T tryRead(bool &success)			コンポーネント内から使用されるノンブロッキング・リードコール。読み込みが有効であれば、success bool は true に設定される。	foo(&a);
				foo_nb(&b);
				}

T tryRead(bool& success, bool& sop, bool& eop)			usesPackets<true>が設定されている場合にのみ有効である。 sideband 信号によるノンブロッキング・リード。	
void write(T data)			コンポーネントに送信される FIFO を配置するためにテストベンチから使用されるブロッキング・ライトコール。	
void write(T data, bool sop, bool eop)			usesPackets<true>が設定されている場合にのみ有効である。 sideband 信号によるブロッキング・ライトコール。	

表 25. HLS ストリーミング用出力インターフェイス

出力インターフェイス	デフォルト値	有効値	説明	例

Stream_out 宣言				void foo (ihc::stream_out<int> &a) {
				static int count = 0;
				
				a.write(count++); // Blocking write
				}
				void foo (stream_in<int> &a) {
				static int count = 0;
				bool success = a.tryWrite(count++); // Non-blocking write
				if (success) {
				// write was successful
				}
				}

ihc::stream_out<datatype>			コンポーネントからのストリーミング用の出力インターフェイス。テストベンチはコンポーネントが返されるとこのバッファーから読み出しが可能。	
ihc::readylatency	0	正の整数値 (0-8 の間)	リード信号がディアサートされてから入力ストリームが新しい入力を受け入れられなるまでのサイクル数。 概念的には、このパラメーターはストリームと関連するデータの入力 FIFO バッファー上の almost ready レイテンシーとして確認することができる。	
ihc::bitsPerSymbol	データ型サイズ	データ型サイズを均等に分割する正の整数値	データがデータバス上のシンボルにどのようにブレークされるかを記述する。データは常にリトル・エンディアンの順に中断される。	
ihc::usesPackets	false	true または false	ストリーム・インターフェイス上の startofpacket と endofpacket 信号を公開する。これは、パケットベースの読み出し / 書き込みによりアクセス可能。	
ihc::usesReady	true	true または false	ready 信号が存在するかどうかを制御する。true の場合は、ダウンストリーム・シンクは valid がアサートされるサイクルごとにデータを受け入れる必要がある。 これはストリーム・リードコールを tryWrite に変更し、success を常に true を保つこととと同じである。	

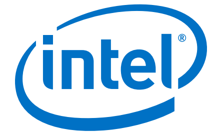
continued...

出力インターフェイス	デフォルト値	有効値	説明	例
			false に設定する場合は、buffer と readyLatency は 0 である必要がある。	
stream_out 関数呼び出し API				
void write(T data)			コンポーネント内からのブロッキング・ライトコール。	
void write(T data, bool sop, bool eop)			usesPackets<true>が設定されている場合にのみ有効。 sideband 信号によるコンポーネントからのブロッキング・ライトコール。	
bool tryWrite(T data)			コンポーネント内からのノンブロッキング・ライトコール。書き込みが成功したかどうかの戻り値を表す。	
bool tryWrite(T data, bool sop, bool eop)			usesPackets<true>が設定されている場合にのみ有効。 コンポーネント内からのノンブロッキング・ライトコールし。書き込みが成功したかどうかの戻り値を表す。	
T read()			コンポーネントからデータを読み戻すためにテストベンチで使用されるブロッキング・リードコール。	
T read(bool &sop, bool &eop)			usesPackets<true>が設定されている場合にのみ有効。 sideband 信号によるコンポーネントからデータを読み戻すためにテストベンチで使用されるブロッキング・リードコール。	

表 26. HLS mm_master インターフェイス

mm_master インターフェイス	有効値	デフォルト値	Description	例
ihc::mm_master<datatype, / *template arguments*/ >		ポインター引数におけるデフォルト・インターフェイス	メモリーマスター・インターフェイスの引数 : 複数のテンプレート引数がサポートされている。テンプレートの引数は下記のとおりである。有効なハードウェア・コンフィグレーションを要求する限り、任意の組み合わせを使用できる。各コンフィグレーションのデフォルト値は、HLS リファレンス・ガイドに記載されている。	component int dut(ihc::mm_master<int, ihc::aspace<2>, ihc::latency<3>, ihc::awidth<10>, ihc::dwidth<32> > &a)
ihc::dwidth<value>	8、16、32、64、128、256、512、または 1024	64	ビット単位でのメモリーマップド・データバスの幅	
ihc::awidth<value>	範囲 1 – 64 の整数値	64	ビット単位でのメモリーマップド・アドレスバスの幅	
ihc::aspace<value>	0 より大きい整数値	1	マスターに関連するインターフェイスのアドレススペース。同じアドレススペースのすべてのマスターは、コンポーネント内で 1 つのインターフェイスに調停される。したがって、これらのマスターはインターフェイスを記述する同じテンプレート・パラメーターを共有する必要がある。	
ihc::latency<value>	正の整数値	1	外部メモリーが有効な読み出しデータを返す際に、リードコマンドがコンポーネントを終了してからの保証されたレイテンシー。このレイテンシーが変数であれば、0 に設定します。	

continued...



mm_master インターフェイス	有効値	デフォルト値	Description	例
`ihc::maxburst<value>`	範囲 1 - 1024 の整数値	1	リード・トランザクションまたはライト・トランザクションに関連されるデータ転送の最大数。 固定レイテンシー・インターフェイスでは、この値は 1 に設定される必要がある。 詳しくは *Avalon Interface Specifications* を参照。	
`ihc::align<value>`	データ型のアライメントより大きい整数値	データ型のアライメント	ベースポインター・アドレスのバイト・アライメント。HLS コンパイラーはこの情報を使用して、このポインターへのロードとストアーで可能な結合の量を決定する。 データ型幅がマスターデータ幅以下の場合、この引数を少なくともマスター幅以上に設定する。 例えば、型 `char` が 64 ビットのマスター幅に設定されている場合、各クロックサイクルで 8 文字の読み出しと書き込みが可能となるように、`align` テンプレート引数は 8 バイトに設定する。 *重要:* 呼び出し元は align 引数の設定値にデータを揃える必要があり、そうでない場合は機能障害が発生する可能性がある。	

`getInterfaceAtIndex(int index)`			この関数は、mm_master オブジェクトへのインデックスに使用される。これは、アレイを反復処理してアレイの異なるインディケーター上のコンポーネントを呼び出す際に便利である。この機能はテストベンチでのみサポートされている。	`int main() {`
				`// …….`
				`for(int index = 0; index < N; index++) {`
				`dut(src_mm.getInterfaceAtIndex(index));`
				`}`
				`// …….`
				`}`

表 27. HLS コンパイラーでサポートされている AC データバイト

AC データ型	インテル・ヘッダーファイル	内容
ac_int	HLS/ac_int.h	任意の幅の整数サポート
ac_fixed	HLS/ac_fixed.h	任意精度の固定小数点サポート

表 28. HLS デバッグツール

デバッグツール		説明
マクロ *コンパイラー・コマンドライン・オプション*	#define DEBUG_AC_INT_WARNING -D DEBUG_AC_INT_WARNING	ac_int データ型の x86 エミュレーション中にランタイムでのトラッキングを有効にする。 これは、オーバーフローを追跡する際に追加のオーバーヘッドを使用し、オーバーフローが検出されると警告を出す。
マクロ *コンパイラー・コマンドライン・オプション*	#define DEBUG_AC_INT_ERROR -D DEBUG_AC_INT_ERROR	ac_int データ型のオーバーフローのトラッキングを有効にする。 これは、オーバーフローの追跡で追加のオーバースレッドを使用し、オーバーフローが検出されるとランタイムでエラーを出す。

B. HLS コンパイラーでサポートされる標準の数学関数

インテル HLS コンパイラーには、math.h の C ヘッダーファイルに存在する標準の数学関数から効率的な IP を生成するための内蔵サポートが含まれています。コンパイラーは非標準の数学関数もサポートしており、これらの関数は extendedmath.h の C ヘッダーファイルで提供されています。

FPGA 用に最適化された math.h の インテル 実装を使用するには、関数内に次の行を追加して HLS/math.h をインクルードします。

```
#include "HLS/math.h"
```

FPGA 用に最適化された math.h の インテル 実装と同様に非標準の数学関数を使用するには、関数内に次の行を追加して HLS/extendedmath.h をインクルードします。

```
#include "HLS/extendedmath.h"
```

extendedmath.h ヘッダーファイルは HLS/math.h ヘッダーファイルを含んでいます。

HLS コンパイラーでサポートされる Math.h 関数

HLS コンパイラーは math.h に存在する関数のサブセットをサポートしています。

各 math.h 関数を下にリストしています。「●」は HLS コンパイラーがサポートする関数を表し、「X」はサポートされない関数を表します。

表 29. 三角関数

三角関数		HLS コンパイラーによるサポート
倍精度浮動小数点関数	cos	●
	sin	●
	tan	●
	acos	●
	asin	●
	atan	●
	atan2	●
単精度浮動小数点関数	cosf	●
	sinf	●
	tanf	●
	acosf	●
	asinf	●
	atanf	●
	atan2f	●

Intel Corporation.無断での引用、転載を禁じます。Intel、インテル、Intel ロゴ、Altera、ARRIA、CYCLONE、ENPIRION、MAX、NIOS、QUARTUS および STRATIX の名称およびロゴは、アメリカ合衆国および/ またはその他の国における Intel Corporation の商標です。インテルは FPGA 製品および半導体製品の性能がインテルの標準保証に準拠することを保証しますが、インテル製品およびサービスは、予告なく変更される場合があります。インテルが書面にて明示的に同意する場合を除き、インテルはここに記載されたアプリケーション、または、いかなる情報、製品、またはサービスの使用によって生じるいっさいの責任を負いません。インテル製品の顧客は、製品またはサービスを購入する前、および、公開済みの情報を信頼する前には、デバイスの仕様を最新のバージョンにしておくことをお勧めします。
*その他の社名、製品名などは、一般に各社の表示、商標または登録商標です。

ISO 9001:2015 登録済

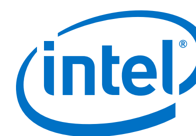


表 30. 双曲線関数

双曲線関数	HLS コンパイラーによるサポート
cosh	•
sinh	•
tanh	•
acosh (C++11)	X
asinh (C++11)	X
atanh (C++11)	X

表 31. 指数関数および対数関数

指数関数または対数関数	HLS コンパイラーによるサポート
exp	•
frexp	•
ldexp	•
log	•
log10	•
modf	•
exp2 (C++11)	•
expm1 (C++11)	X
ilogb (C++11)	•
log1p (C++11)	X
log2 (C++11)	•
logb (C++11)	•
scalbn (C++11)	X
scalbln (C++11)	X

表 32. べき関数

べき関数	HLS コンパイラーによるサポート
pow	•
sqrt	•
cbrt (C++11)	X
hypot (C++11)	X

表 33. エラー関数およびガンマ関数

エラー関数またはガンマ関数	HLS コンパイラーによるサポート
erf (C++11)	X
erfc (C++11)	X
tgamma (C++11)	X
lgamma (C++11)	X

表 34. 丸め関数と余り関数

丸め関数と余り関数	HLS コンパイラーによるサポート
ceil	•
floor	•
fmod	•
trunc (C++11)	•
round (C++11)	•
lround (C++11)	X
llround (C++11)	X
rint (C++11)	•
lrint (C++11)	X
llrint (C++11)	X
nearbyint (C++11)	X
remainder (C++11)	X
remquo (C++11)	X

表 35. 浮動小数点操作関数

浮動小数点操作関数	HLS コンパイラーによるサポート
copysign (C++11)	X
nan (C++11)	X
nextafter (C++11)	X
nexttoward (C++11)	X

表 36. 最小、最大、および差分関数

最小、最大、または差分関数	HLS コンパイラーによるサポート
fdim (C++11)	•
fmax (C++11)	•
fmin (C++11)	•

表 37. その他の関数

関数	HLS コンパイラーによるサポート
fabs	•
abs	X
fma (C++11)	X

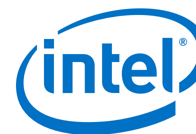


表 38. マクロ分類

マクロ分類	HLS コンパイラーによるサポート
fpclassify (C++11)	X
isfinite (C++11)	•
isinf (C++11)	•
isnan (C++11)	•
isnormal (C++11)	X
signbit (C++11)	X

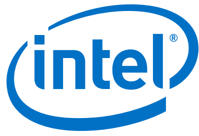
表 39. 比較マクロ

比較マクロ	HLS コンパイラーによるサポート
isgreater (C++11)	X
isgreaterequal (C++11)	X
isless (C++11)	X
islessequal (C++11)	X
islessgreater (C++11)	X
isunordered (C++11)	X

extendedmath.h ヘッダーファイルで提供される数学関数

HLS/extendedmath.h ヘッダーファイルを追加すると、次の関数もサポートされます。

- sincos
- acospi
- asinpi
- atanpi
- cospi
- sinpi
- tanpi
- pown
- powr
- rsqrt



B.1. 改訂履歴

表 40. インテル HLS コンパイラーリファレンス･マニュアル付録の改定履歴：HLS コンパイラー･サポートの標準数学関数

日付	バージョン	変更内容
2017 年 6 月	2017.06.09	`extendedmath.h` ヘッダーファイルでサポートされる数学関数に関する情報を追加。
2017 年 2 月	2017.02.03	メンテナンス･リリース
2016 年 11 月	2016.11.30	メンテナンス･リリース
2016 年 9 月	2016.09.12	初版