



XAPP1170 (v2.0) 2016 年 1 月 21 日

# Vivado HLS で設計する浮動小数点 行列乗算の Zynq アクセラレータ

著者 : Daniele Bagni, A. Di Fresco, J. Noguera, F. M. Vallina

## 概要

このアプリケーション ノートでは、Vivado® 高位合成 (HLS) を使用して AXI4-Stream インターフェイス経由で Zynq®-7000 All Programmable SoC (AP SoC) デバイスの ARM CPU のアクセラレータ コヒーレンシ ポート (ACP) へ接続される浮動小数点行列乗算アクセラレータの開発方法について説明します。

Vivado HLS を使用することで、C/C++ コードでモデル化された浮動小数点行列乗算アクセラレータをレジスタ転送レベル (RTL) デザインに素早く実装および最適化できます。ソリューションは IP コアとしてエクスポートされ、自動生成された AXI4-Stream インターフェイスを介して AP SoC Processing Subsystem (PS) の ACP に接続されます。接続には、AP SoC Programmable Logic (PL) サブシステムの DMA (Direct Memory Access) コアを使用します。行列乗算ペリフェラル、DMA エンジン、AXI タイマーなどの AP SoC PL ハードウェアの設計には、Vivado IP インテグレーター (IPI) を使用します。ペリフェラルを管理する AP SoC PS ソフトウェアの設計にはソフトウェア開発キット (SDK) を使用します。

このアプリケーション ノートの [リファレンスデザインファイル](#) は、ザイリンクスのウェブサイトからダウンロードできます。デザイン ファイルの詳細は、「[リファレンス デザイン](#)」を参照してください。

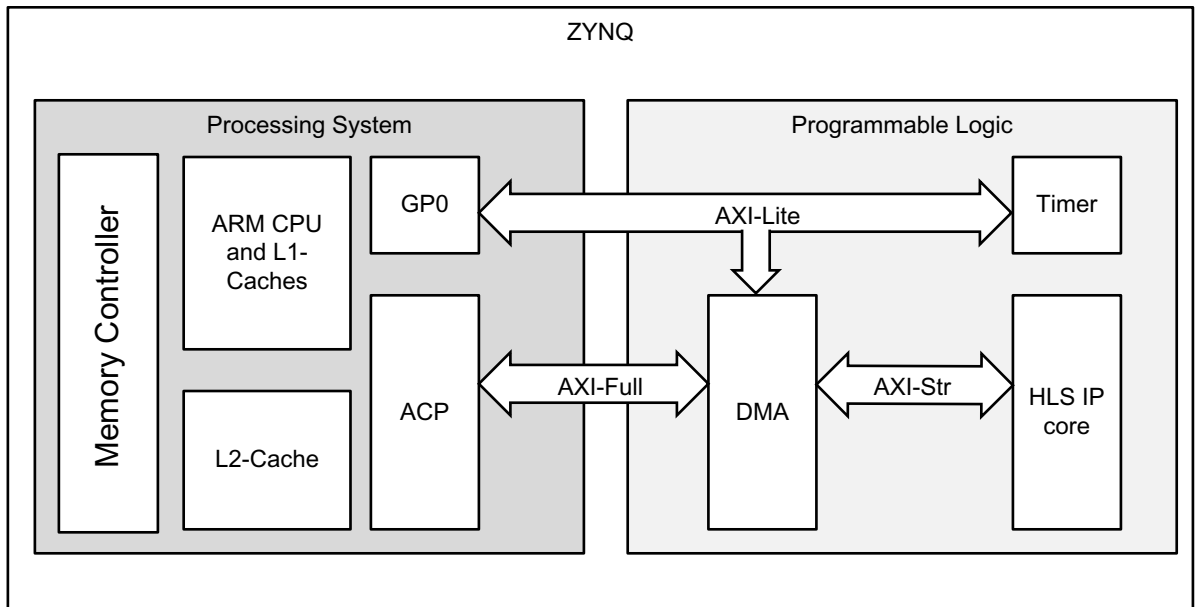
## はじめに

行列乗算は、応用数学のほとんどすべての演算で使用されています。たとえば、最新レーダー システムでは、受信アンテナの位相をコンピューター計算でデジタル制御するビームフォーミング技術で行列乗算が使用されています。ザイリンクスの Vivado HLS ツールを使用することで、浮動小数点アルゴリズムを C/C++ コードで素早く記述でき、Zynq-7000 AP SoC で最適化して実装できます ([\[参照 1\]](#))。この方法は、従来のマイクロプロセッサを用いて浮動小数点アルゴリズムを実装する設計者に、低コスト、高性能、低消費電力といった利点をもたらします ([\[参照 2\]](#)、[\[参照 3\]](#))。

このアプリケーション ノートでは、32x32 行列の浮動小数点乗算の適用をはじめとする、ザイリンクス PL でのデザインフローについて説明します。

1. Vivado HLS を使用して、C/C++ で記述された浮動小数点デザインを高性能ハードウェア アクセラレータにコンパイルして最適化する。
2. Vivado HLS の C++ テンプレートを使用して、ハードウェア アクセラレータの AXI4-Stream インターフェイスを指定して生成する。
3. Vivado IP インテグレーター [\[参照 4\]](#) を使用して、ハードウェア アクセラレータを AP SoC PL の AXI DMA ペリフェラルおよび AP SoC PS の ACP に接続する。
4. ARM CPU で動作するソフトウェアを記述してハードウェア アクセラレータの関数呼び出しを含め、システムレベルの性能を測定する。

[図 1](#) に、Zynq-7000 デバイスに実装されたシステムのブロック図を示します。



X15748-010316

図 1 : Zynq-7000 AP SoC 上の PS と PL のパーティション

このアプリケーション ノートで説明する設計手順は、Zynq-7000 AP SoC 評価キット (ZC702) [参照 5] を使い、Vivado 2015.4 IDE ツールで設計する場合に適用されます。

## Vivado HLS を使用した行列乗算デザイン

行列乗算アルゴリズム  $A*B=C$  は非常にシンプルです。次の 3 つのネストされたループがあります。

- 1 つ目のループ (L1) は、1 行の入力行列 **A** を構成する要素を繰り返します。
- 2 つ目のループ (L2) は、1 列の入力行列 **B** 内の要素を繰り返します。
- 3 つ目のループ (L3) は、行ベクトルの各インデックスを列ベクトル **B** のインデックスと掛け算し、それを累算して 1 行の出力行列 **C** の要素を生成します。

最適化される関数の C++ コードは次のようになります。

```
template <typename T, int DIM>
void mmult_hw(T A[DIM][DIM], T B[DIM][DIM], T C[DIM][DIM])
{
    // matrix multiplication of a A*B matrix
    L1:for (int ia = 0; ia < DIM; ++ia)
    {
        L2:for (int ib = 0; ib < DIM; ++ib)
        {
            T sum = 0;
            L3:for (int id = 0; id < DIM; ++id)
            {
                sum += A[ia][id] * B[id][ib];
            }
            C[ia][ib] = sum;
        }
    }
}
```

アルゴリズムを C++ コードでキャプチャ後に、Vivado HLS を使用してこれを RTL インプリメンテーションに合成できます。C++ ソース コードのほかにも、Vivado HLS はターゲット クロック周波数、ターゲット デバイス仕様、さらには特定の最適化を制御および指示するためのユーザー指示子 (コマンド) を入力として受けることができます。Vivado HLS の機能や特徴を短時間で理解するには、例を用いて実際に使用してみることです。Vivado HLS の詳細は、『Vivado HLS ユーザーガイド』[参照 6] を参照してください。

次の TCL コードは、クロック周期とターゲット デバイスを示しています。

```
set_part {xc7z010clg400-1}
create_clock -period 10
```

mmult\_hw 関数のコードを使用する場合、Vivado HLS は次の動作を実行します。

- C コードで記述されている各動作を同等のハードウェア動作に変換し、これら動作をクロック サイクルに割り当てます。クロック周期とデバイス遅延の情報に基づいて、シングルクロック サイクル内にできるだけ多くの動作を割り当てます。
- インターフェイス合成を使用して、データがどのようにしてハードウェアブロックに送信されて書き込みが実行されるかを自動で合成します。たとえば、データが配列として与えられる場合は、RAM ブロック (別の I/O インターフェイス オプションを指定可能) へアクセスするインターフェイスが自動的に構築されます。
- 各ハードウェア動作を AP SoC 上の同等ハードウェアユニットにマップします。
- 任意のユーザー定義の最適化 (パイプライン動作や並列動作など) を実行します。
- AP SoC に実装するための最終的なデザイン (Verilog および VHDL) をレポートと共に出力します。

サンプル コアのコード合成で生成されたレポートでは、最初の性能評価 (デフォルトの合成結果) を含む動作や性能が評価されています。

この例では、Vivado HLS が C コードで記述された動作を解析し、指定されたターゲット技術とクロック周期で結果を計算するのに必要なクロック サイクルが 329,793 であると判断しています。つまり、このデザインは、最大のクロック周期 8.41ns で実行できると考えられます。

図 2 のエリア評価では、デザインで使用される PL リソース数を示しています (5 DSP48 スライス、473 FF (フリップフロップ)、830 LUT (ルックアップテーブル))。

Synthesis(solution1) ✕

### Performance Estimates

☐ **Timing (ns)**

☐ **Summary**

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.41	1.25

☐ **Latency (clock cycles)**

☐ **Summary**

Latency		Interval		
min	max	min	max	Type
329793	329793	329794	329794	none

### Utilization Estimates

☐ **Summary**

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	63
FIFO	-	-	-	-
Instance	-	5	348	711
Memory	-	-	-	-
Multiplexer	-	-	-	56
Register	-	-	125	-
<b>Total</b>	<b>0</b>	<b>5</b>	<b>473</b>	<b>830</b>
Available	280	220	106400	53200
<b>Utilization (%)</b>	<b>0</b>	<b>2</b>	<b>~0</b>	<b>1</b>

X15749-010416

図 2 : 32 ビット浮動小数点の最初の性能評価

これらの値はあくまでも推定値にすぎません。TRL 合成プロセスでは、RTL コードをさらにゲートレベルのコンポーネントへ変換し、デバイスに配置配線する必要があります。ほかにも、最終的な結果に影響を与えるゲートレベルの最適化が実行される可能性があります。

図 3 に、インターフェイス合成によって I/O ポートに変換された C 関数の引数を示します。このプロセスで、完成したエンベデッド デザインのその他のブロックへポートが接続されます。

Interface						
Summary						
RTL Ports	Dir	Bits	Protocol	Source Object	C Type	
ap_clk	in	1	ap_ctrl_hs	standalone_mmult		return value
ap_rst	in	1	ap_ctrl_hs	standalone_mmult		return value
ap_start	in	1	ap_ctrl_hs	standalone_mmult		return value
ap_done	out	1	ap_ctrl_hs	standalone_mmult		return value
ap_idle	out	1	ap_ctrl_hs	standalone_mmult		return value
ap_ready	out	1	ap_ctrl_hs	standalone_mmult		return value
A_address0	out	10	ap_memory		A	array
A_ce0	out	1	ap_memory		A	array
A_q0	in	32	ap_memory		A	array
B_address0	out	10	ap_memory		B	array
B_ce0	out	1	ap_memory		B	array
B_q0	in	32	ap_memory		B	array
C_address0	out	10	ap_memory		C	array
C_ce0	out	1	ap_memory		C	array
C_we0	out	1	ap_memory		C	array
C_d0	out	32	ap_memory		C	array

X15750-010316

図 3 : 最初の RTL ポート

このプロセスで生じた変更は次のとおりです。

- デザインにクロック信号とリセット信号 (ap\_clk、ap\_rst) が追加されました。
- デザイン レベルのプロトコルが追加されました。これはデフォルト設定ですが、変更可能です。これにより、デザインの動作開始 (ap\_start)、新しい入力に対応する準備が整っている状態、動作が完了した状態 (ap\_done)、またはアイドル状態を示すことができます。
- 配列引数が、ザイリンクスのブロック RAM へアクセスするための適切なアドレス信号、イネーブル信号、および書き込み信号を持つ RAM インターフェイスに変換されました。さらに、DIN ポートがデュアルポートのブロック BRAM (必要に応じてシングルポートブロック RAM に設定可能) を使用することで性能が向上すると Vivado HLS が自動判断しました。
- Vivado HLS によって、C コードの動作と I/O 動作が実装されている RTL インプリメンテーションが生成されました。このとき、Verilog、VHDL などの RTL 言語や RTL デザイン全般に関する知識はユーザーに要求されません。

## 最適化された RTL

HLS で生成された初期デザインは最適化できます。図 4 では、3 つの有効なソリューションを比較しています。行列乗算の計算に必要なクロックサイクルを削減するために、最適化が適用されました。Vivado HLS による最適化の詳細は、『Vivado Design Suite チュートリアル: 高位合成』[参照 7] を参照してください。

Performance Estimates				
Timing (ns)				
Clock		solution1	solution2	solution3
ap_clk	Target	10.00	10.00	10.00
	Estimated	8.41	9.35	8.41
Latency (clock cycles)				
		solution1	solution2	solution3
Latency	min	329793	16535	1190
	max	329793	16535	1190
Interval	min	329794	16536	1191
	max	329794	16536	1191
Utilization Estimates				
		solution1	solution2	solution3
BRAM_18K		0	0	0
DSP48E		5	10	160
FF		473	2312	13420
LUT		830	3450	23293

X15751-010316

図 4: 3 つのソリューションの性能評価の比較

ソリューション 2 は、初期デザイン (ソリューション 1) より約 20 倍高速化されましたが、使用するリソースは増加しています (10 DSP48 スライス、2,312 FF、3,450 LUT)。予測されたクロック周期は 9.35ns で、これは式 1 で算出されるように出力データレートが 6.46 KSPS (千サンプル/秒) となります。

$$16536 \times 9.35\text{ns} = 0.154\text{ms} = 1 / (6.46 \text{ KSPS}) \quad \text{式 1}$$

最も高い性能を示したのは明らかにソリューション 3 で、浮動小数点の行列乗算を計算するのに必要なクロックサイクルは、わずか 1,190 です。この値は、160 DSP48 スライス、13,420 FF、および 23,293 LUT で達成した結果であり、AP SoC で使用可能な各リソースの 72%、12%、および 43% にあたります。ソリューション 3 は、パイプライン初期化インターバルが 1 で、これはスループットが 1 ということです。この例の場合、データレートは 118.9 MSPS (百万サンプル/秒) で、すべての出力行列は 1,190 x 8.41ns 時間 (10µs) 以内に生成されます。

スループット 1 を達成したということは、各クロックサイクルで 1 つの行列出力サンプルが生成されるということです。ソリューションの選択はユーザー要件に基づきます。たとえば、PL リソース重視の観点から、より低コストなデザインを優先する場合はソリューション 2 を選択できます。

次の TCL コードは、ソリューション 3 の最適化指示子を示しています。

```
set_directive_array_partition -type block -factor 16 -dim 2 "mmult_hw" A
set_directive_array_partition -type block -factor 16 -dim 1 "mmult_hw" B
set_directive_pipeline -II 1 "mmult_hw/L2"
```

## Vivado HLS の AXI4-Stream インターフェイス

AXI4-Stream は、アドレス指定や外部バス マスターを必要としないポイント間データ転送用の通信規格です [参照 8]。このプロトコルによって、2つのコアがプロデューサー /コンシューマー モデルを使用するデータに対して動的に同期します。使用される AXI4-Stream のインプリメンテーションに基づいて、通信チャンネルはワイヤーとして構築されたり、両端でのデータ レートの不一致に対応するためにストレージを用いて構築されます。

Vivado HLS で設計された行列乗算コアは、AXI4-Stream インターフェイスを使用して DMA コントローラーへ接続されます。バースト フォーマット、アドレス生成、メモリ トランザクションのスケジューリングは、AXI DMA IP で実行されます。

図 1 に示すシステムのアーキテクチャの場合、ACP ポートを適用して AXI DMA を ARM プロセッサの L2 キャッシュへ接続しています。また、別の手段として、HP (High Performance) ポートを使用して外部 DDR メモリへ接続する方法があります。Vivado HLS コアからすると、DMA を介すメモリ インターフェイスは、メモリが DDR または L2 キャッシュのいずれであっても同じです。DDR または L2 キャッシュのいずれを使用してプロセッサと Vivado HLS コア間でデータを共有するかはシステム アーキテクチャの判断です。DDR は L2 キャッシュより多くのデータを転送できますが、通信時のレイテンシは L2 キャッシュの方が DDR より低くなります。

Vivado HLS の行列乗算ブロックを AXI DMA へ接続するためには、2 ページの「Vivado HLS を使用した行列乗算デザイン」のコードを変更する必要があります。このセクションで示す新たに記述した C++ コードのように、合成される関数をいくつか追加する必要があります。具体的には、AXI4-Stream インターフェイスから要素を抽出する `pop_stream` と AXI4-Stream インターフェイスへ要素を挿入する `push_stream` の 2 つの関数です。これらの関数は、行列の 32 ビット浮動小数点データと AXI4 プロトコルの 32 ビット符号なしデータ間の変換も実行します。

次に示すコードは、AXI\_VAL データ型の使用法を示しています。これは、AXI4-Stream インターフェイスに関連するサイド チャンネル情報を表すユーザー定義データ型です。Vivado HLS では、プロトコル ハンドシェイクに関連しないすべてのサイド チャンネル情報は C/C++ コードで記述され、何らかの方法で使用される必要があります。つまり、Vivado HLS は TREADY 信号と TVALID 信号を抽象化しますが、AXI4-Stream インターフェイスのその他すべての信号はユーザー コードに含まれる必要があります。また、TDATA、TREADY、および TVALID 信号を除く、すべての AXI4-Stream インターフェイス信号はオプションです。サイド チャンネル信号の使用は、Vivado HLS AXI4-Stream インターフェイスへ接続されたブロックに依存します。

```

#include <assert.h>
#include <ap_axi_sdata.h>
typedef ap_axiu<32,4,5,5> AXI_VAL;
template <typename T, int DIM, int SIZE, int U, int TI, int TD>
void wrapped_mmult_hw(AXI_VAL in_stream[2*SIZE], AXI_VAL out_stream[SIZE])
{
    T A[DIM][DIM], B[DIM][DIM], C[DIM][DIM];
    assert(sizeof(T)*8 == 32);
    // stream in the 2 input matrices
    for(int i=0; i<DIM; i++)
        for(int j=0; j<DIM; j++)
            {
                #pragma HLS PIPELINE II=1
                int k = i*DIM + j;
                A[i][j] = pop_stream<T,U,TI,TD>(in_stream[k]);
            }
    for (int i=0; i<DIM; i++)
        for (int j=0; j<DIM; j++)
            {
                #pragma HLS PIPELINE II=1
                int k = i*DIM + j + SIZE;
                B[i][j] = pop_stream<T,U,TI,TD>(in_stream[k]);
            }

    // do multiplication
    mmult_hw<T, DIM>(A, B, C);
    // stream out result matrix
    for (int i=0; i<DIM; i++)
        for (int j=0; j<DIM; j++)
            {
                #pragma HLS PIPELINE II=1
                int k = i*DIM + j;
                out_stream[k] = push_stream<T,U,TI,TD>(C[i][j], k==1023);
            }
    }
// this is the top level design that will be synthesized into RTL
void HLS_accel(AXI_VAL INPUT_STREAM[2048], AXI_VAL OUTPUT_STREAM[1024])
{
    // Map ports to Vivado HLS interfaces
    #pragma HLS INTERFACE s_axilite port=return bundle=CONTROL_BUS
    #pragma HLS INTERFACE axis port=INPUT_STREAM
    #pragma HLS INTERFACE axis port=OUTPUT_STREAM

    wrapped_mmult_hw<float,32,32*32,4,5,5>(INPUT_STREAM, OUTPUT_STREAM);
}

```

図 5 に、AXI4-Stream 行列乗算の合成レポートを示します。この時点のレイテンシは、4,267 クロック サイクルであることに留意してください。合計レイテンシの値は、各行列をアクセラレータから転送したり、アクセラレータへ転送する時間、演算の時間、およびハードウェア関数のセットアップにかかる時間を考慮に入れて計算されます。各行列の転送時間は、1,024 の 32 ビットの浮動小数点値で 1,024 クロック サイクルであるため、合計時間は 3,072 クロック サイクルとなります。行列乗算の計算時間は 1,188 クロック サイクルであり、それに pop\_stream 関数と push\_stream 関数用に 2 サイクルが追加されます。その他、FOR ループのプロローグ/エピローグおよび関数のスタートアップに多少のクロック サイクルが必要になります。結果として、関数の初期化インターバル (II) は 4,268 クロック サイクルで、レイテンシが 4,267 クロック サイクルとなります。

Performance Estimates					Utilization Estimates				
<b>Timing (ns)</b>					<b>Summary</b>				
<b>Summary</b>					Name	BRAM_18K	DSP48E	FF	LUT
Clock	Target	Estimated	Uncertainty		Expression	-	-	0	235
ap_clk	10.00	8.41	1.25		FIFO	-	-	-	-
<b>Latency (clock cycles)</b>					Instance	0	160	11172	22792
<b>Summary</b>					Memory	66	-	0	0
Latency	Interval				Multiplexer	-	-	-	322
min	max	min	max	Type	Register	-	-	2487	440
4267	4267	4268	4268	none	<b>Total</b>	<b>66</b>	<b>160</b>	<b>13659</b>	<b>23789</b>
<b>Detail</b>					Available	280	220	106400	53200
<b>Loop</b>					Utilization (%)	23	72	12	44
Loop Name	Latency		Iteration Latency	Initiation Interval		Trip Count	Pipelined		
- Loop 1	min	max		achieved	target				
- Loop 2	1024	1024	2	1	1	1024	yes		
- Loop 3	1188	1188	166	1	1	1024	yes		
- Loop 4	1025	1025	3	1	1	1024	yes		

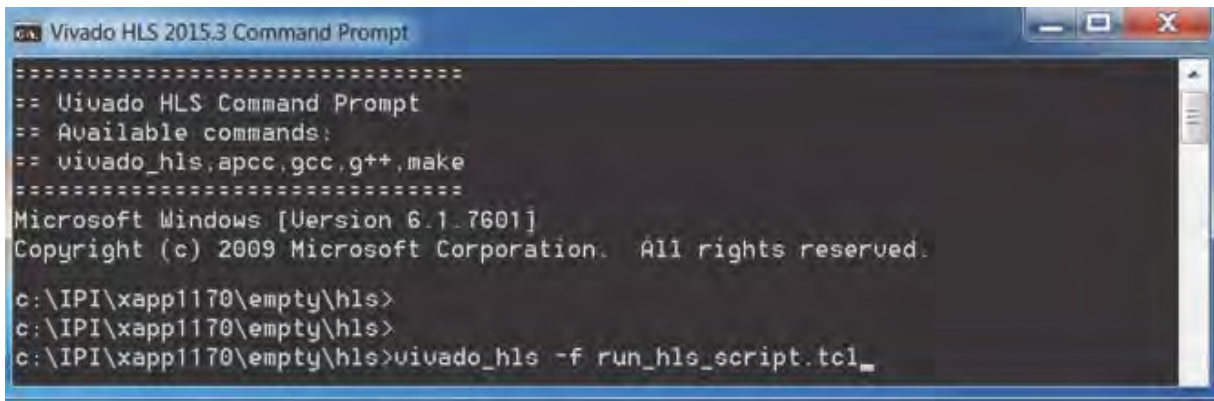
X15752-010316

図 5 : AXI4-Stream 行列乗算コアの合成評価

全体的なリソース使用数の概算は、66 BRAM18K (18K ビット容量のブロック RAM)、160 DSP48 スライス、13,659 FF、および 23,789 LUT となり、Zynq-7000 で使用可能な各リソースの 23%、72%、12%、および 44% にあたります。データレートは 118.9 MSPS のままで、すべての出力行列は 4,268 x 8.41ns 時間 (36μs) 以内に生成されます。

Vivado HLS プロジェクトは、[図 6](#) に示す Vivado HLS Command Prompt シェルを使用して、デザイン アーカイブで提供されている TCL スクリプトを次のコマンドで実行すると作成できます。

```
vivado_hls -f run_hls_script.tcl
```



```

Vivado HLS 2015.3 Command Prompt
=====
== Vivado HLS Command Prompt
== Available commands:
== vivado_hls,apcc,gcc,g++,make
=====
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

c:\IPI\xapp1170\empty\hls>
c:\IPI\xapp1170\empty\hls>
c:\IPI\xapp1170\empty\hls>vivado_hls -f run_hls_script.tcl_

```

図 6: TCL スクリプトを実行して Vivado HLS プロジェクトを作成

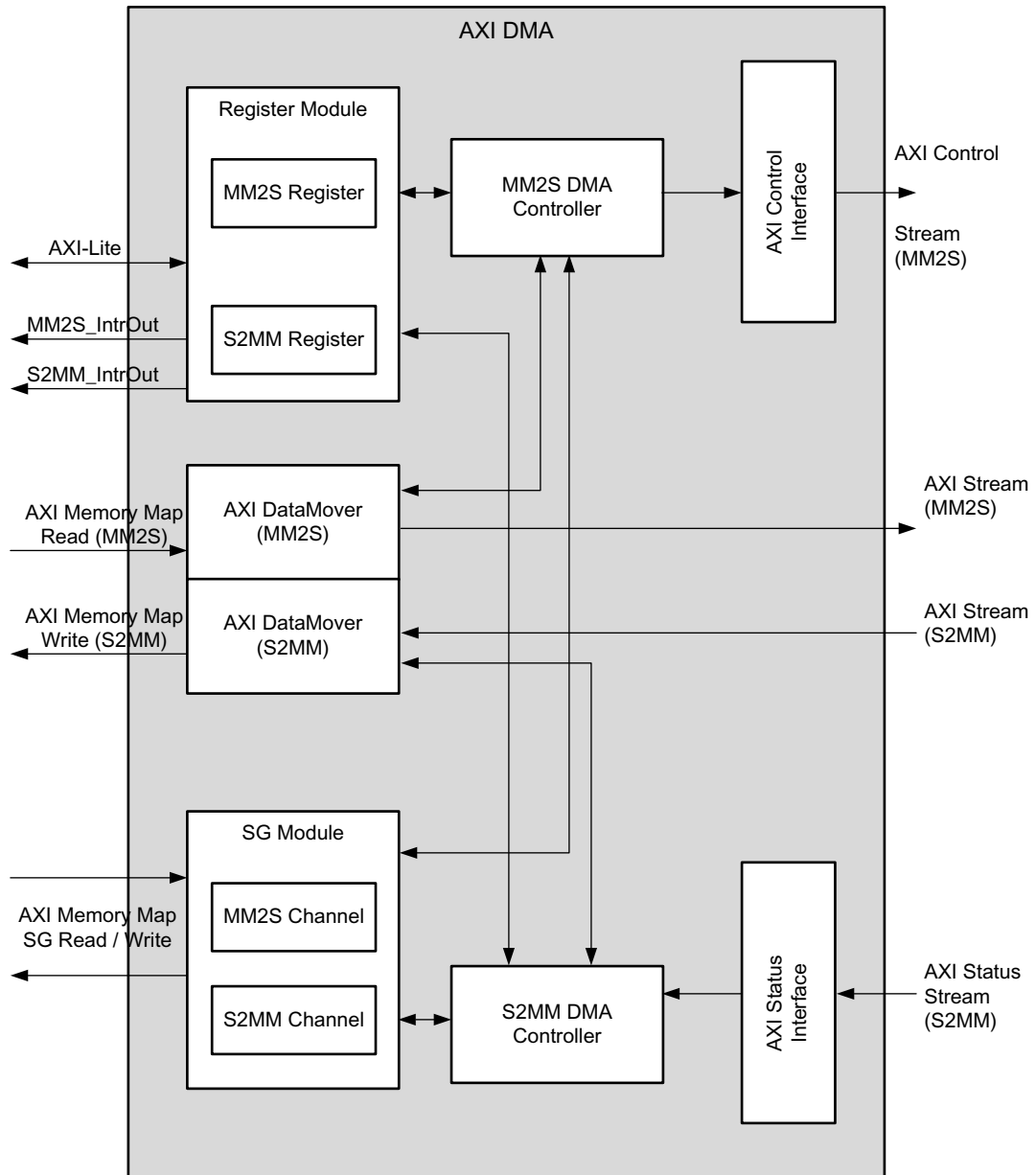
64 ビットの AXI4-Stream インターフェイスを使用することで、システム レベルの性能をさらに向上させることができます。この場合、アクセラレータ レイテンシは 4,267 サイクルから約 2,700 サイクルに削減されます ([図 4](#) に示すループ L1、L2、および L4 のサイクルが半分になる)。

## AXI DMA の概要

AXI DMA コア [\[参照 9\]](#) は、AXI4-Stream インターフェイスを介すメモリとペリフェラル間に高帯域幅のダイレクト メモリ アクセスを提供します。このコア デザインには次の AXI4 インターフェイスがあります。

- AXI4-Lite スレーブ
- AXI4 メモリ マップ読み出しマスター/書き込みマスター
- オプションの AXI4 メモリ マップ スキャッター/ギャザー 読み出し/書き込みマスター
- AXI4 to AXI4-Stream (MM2S) ストリーム マスター
- AXI4-Stream to AXI4 (S2MM) ストリーム スレーブ
- AXI 制御 AXI4-Stream マスター
- AXI ステータス AXI4-Stream スレーブ

[図 7](#) に、AXI DMA インターフェイスとデータ ストリーミングを示します。AXI DMA には 2 つの異なる動作モード (スキャッター/ギャザー モード、シンプル DMA モード) があり、同時に両方は使用できません。各モードには、AXI4-Lite スレーブ インターフェイスを使用して設定された適切なレジスタがあります。ここで示すインプリメンテーションでは、シンプル DMA モードを使用しています。



X15754-010316

図 7: AXI DMA コアのブロック図

シンプル DMA モードは MM2S および S2MM チャンネル上でシンプルな DMA 転送を実行するためのコンフィギュレーションで、使用する FPGA リソース量を抑えることができます。AXI4 Read (MM2S) インターフェイスがマスター外部メモリからデータを読み出して、その後 DMA Data Mover が AXI4-Stream (MM2S) ポートを介してそのデータをスレーブペリフェラルへ転送します。同様に、マスターペリフェラルは AXI4 Write (S2MM) インターフェイスへデータを送信し、その後、AXI4-Stream (S2MM) ポートを介してそのデータをスレーブ外部メモリに書き込みます。

DMA 転送は、制御、ソース、またはデスティネーション アドレスとレンジ レジスタを使用して開始されます。MM2S チャンネルのセットアップ シーケンスは次のとおりです。

1. 制御レジスタの run/stop ビットを設定して、MM2S チャンネルの実行を開始します。
2. 有効なソース アドレスが MM2S ソース アドレス レジスタに書き込まれます。
3. 転送されるバイト数が MM2S レンジ レジスタに書き込まれます。レンジ レジスタは最後に書き込む必要があります。

S2MM チャンネルのセットアップ シーケンスは次のとおりです。

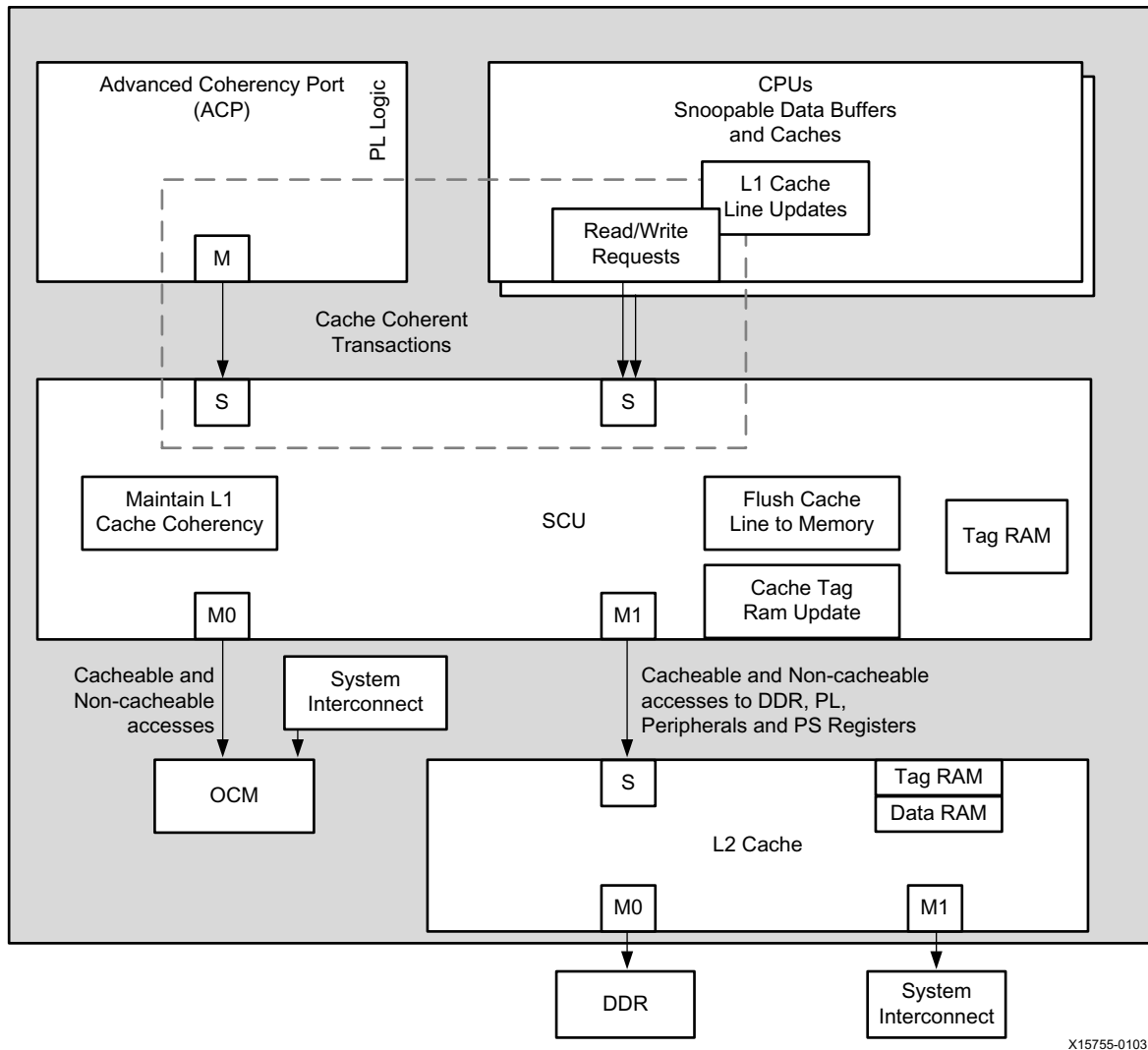
1. 制御レジスタの run/stop ビットを設定して、S2MM チャンネルの実行を開始します。
2. 有効なデスティネーション アドレスが S2MM デスティネーション アドレス レジスタに書き込まれます。
3. 受信バッファのバイト長が S2MM レンジ レジスタに書き込まれます。レンジ レジスタは最後に書き込む必要があります。

---

## ARM ACP の概要

ACP ポートは、スヌープ制御ユニット (SCU) の 64 ビット AXI スレーブ インターフェイスで、Zynq-7000 AP SoC PL と Cortex-A9 CPU プロセッサ サブシステムを直接接続し、キャッシュ コヒーレントな非同期アクセス ポイントとして機能します。この ACP ポートは、PS と PL に実装されたアクセラレータ間に低レイテンシ パスを提供します。システム内のあらゆる PL マスターは、このインターフェイスを介することで、プロセッサと同じようにキャッシュおよびメモリ サブシステムへアクセスできるため、実行するソフトウェア アプリケーションの全体的なシステム性能を向上させることができます。

ACP を経由したコヒーレントなメモリ領域への読み出しトランザクションは、SCU と連携して必要なデータがプロセッサの L1 キャッシュに格納されているかどうかを確認します。L1 キャッシュに必要な情報がある場合、データは要求元のコンポーネントに直接返されます。情報が L1 キャッシュにない場合は、L2 キャッシュにもヒットしなければトランザクションは最後にメイン メモリに転送されます。コヒーレントなメモリ領域に対する書き込みトランザクションの場合、書き込みがメモリ システムに転送される前に SCU がコヒーレンスを強制します。オプションでトランザクションを L2 キャッシュに割り当てることができ、オフチップ メモリへの書き込み動作によって生じる消費電力や性能への影響を回避できます。図 8 に、ACP と CPU へ接続されたメモリ システム間の接続を示します。



X15755-010316

図 8: ARM メモリと ACP の接続

## Vivado IP インテグレーションのデザイン

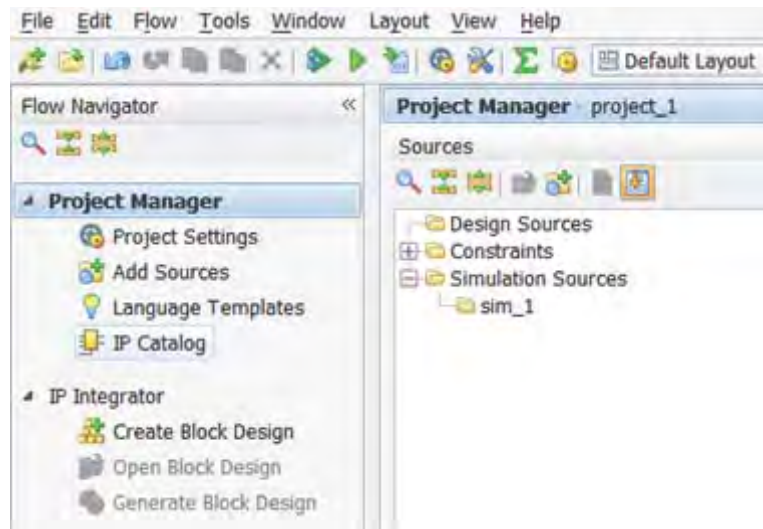
このセクションでは、ZC702 ボードをベースにザイリンクスの Vivado 2015.4 を使用してハードウェア デザインを作成する手順を説明します。以前のバージョン (2013.3、2014.4、2015.2) でも同様の手順を適用できます。

すべてデフォルト設定を選択して、基本の Vivado プロジェクト (project\_1) を作成します。デバイスを指定する画面では、ZC702 ボードを選択します。詳細手順は次のとおりです。

1. Vivado を起動します。作業ディレクトリ (xapp1170/empty/vivado) に新規プロジェクトを作成して、次を選択します。
  - 。 RTL プロジェクト
  - 。 ソース、IP、制約は不要 (現段階では)
2. [Default Part] で、[Boards] → [Zynq-7 ZC702 Evaluation Board] を選択します。
3. [Next] → [Finish] をクリックします。空のプロジェクトで Vivado GUI が表示されます。

IP リポジトリに HLS IP を追加 :

1. Flow Navigator の [Project Manager] で [IP Catalog] を選択します (図 9 参照)。



X15756-010316

図 9: IP カタログの起動

2. [IP Catalog] ウィンドウを表示して、[IP Settings] をクリックします (図 10 参照)。

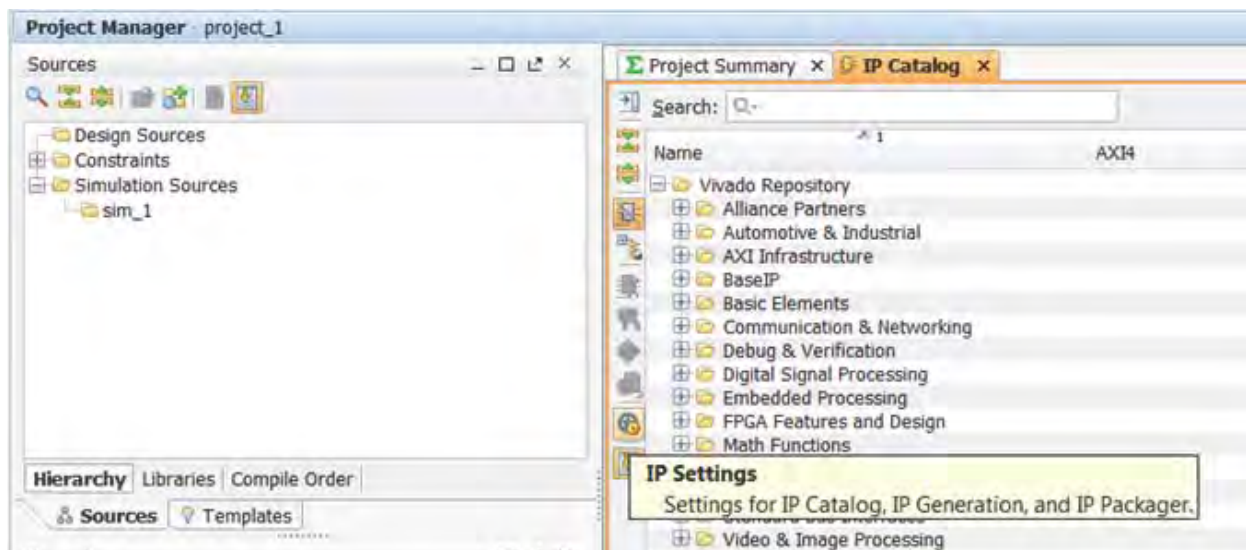
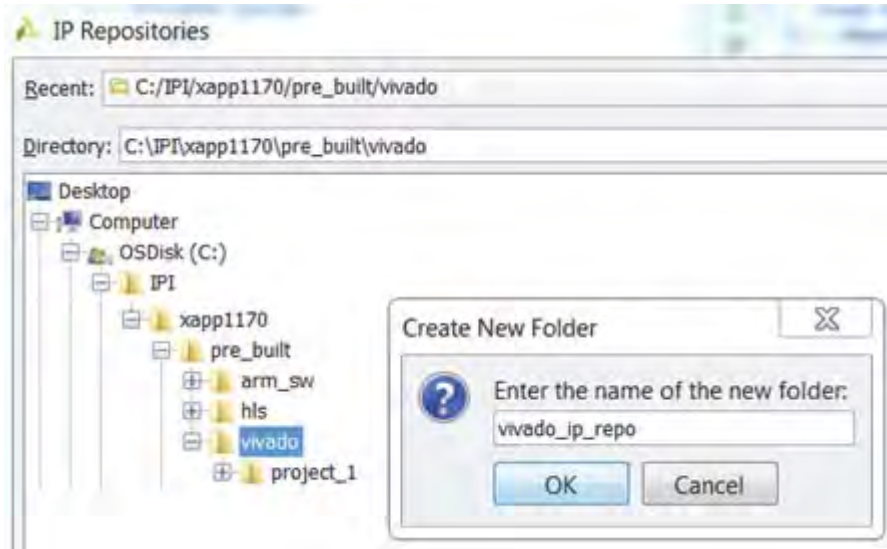


図 10: IP カタログの設定を表示

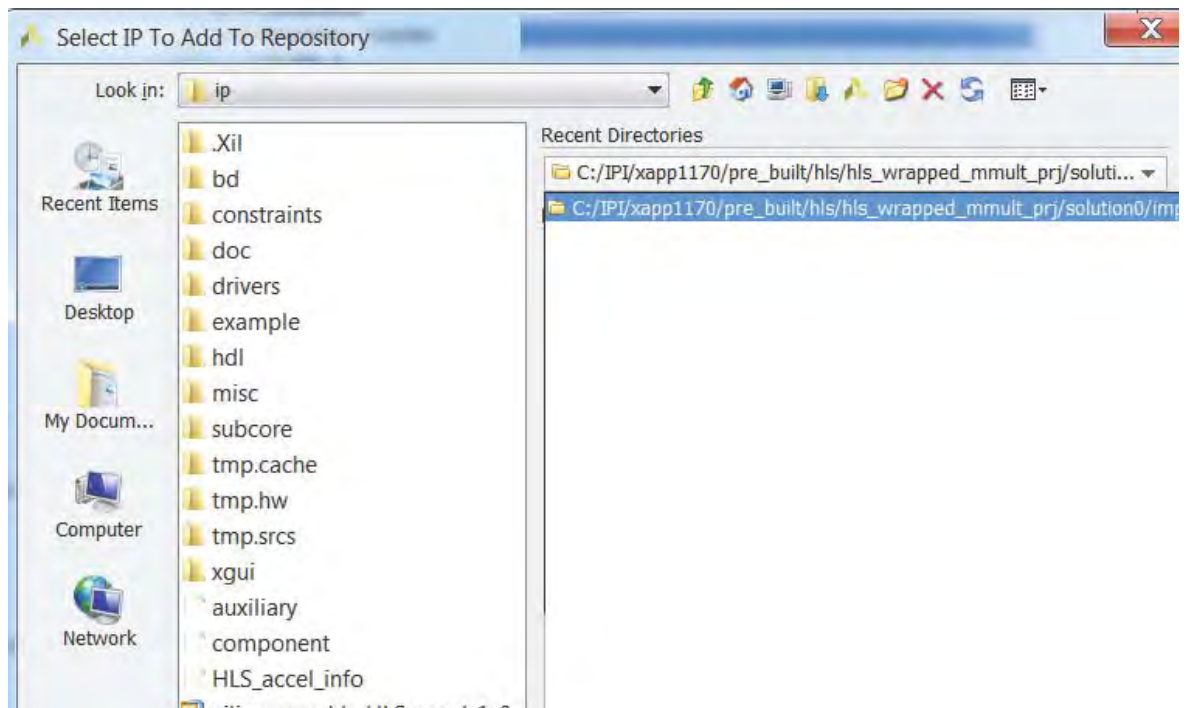
3. [Repository Manager] で [Add Repository] をクリックします。
4. [IP Repositories] ダイアログ ボックスで次を実行します。
  - a. [Create New Folder] アイコンをクリックします。
  - b. 表示されたダイアログ ボックスに「vivado\_ip\_repo」と名前を入力します (図 11 参照)。
  - c. [OK] をクリックして、[Create New Folder] ダイアログ ボックスを閉じます。



X15758-010316

図 11: 新規の IP リポジトリを作成

5. [User Repository] を右クリックします (この時点では空)。
  - a. [Add IP to Repository] を選択します。
  - b. [Select IP to Add to Repository] ダイアログ ボックスで、hls/hls\_wrapped\_mmult\_prj/solution0/impl/ip/ を参照します。
  - c. xilinx\_com\_hls\_HLS\_accel\_1\_0.zip ファイルを選択します (図 12 参照)。



X15759-010316

図 12: リポジトリに HLS IP を追加

- d. [OK] をクリックします。これで [IP Catalog] に Vivado HLS IP が追加されていることを確認できます (図 13 参照)。

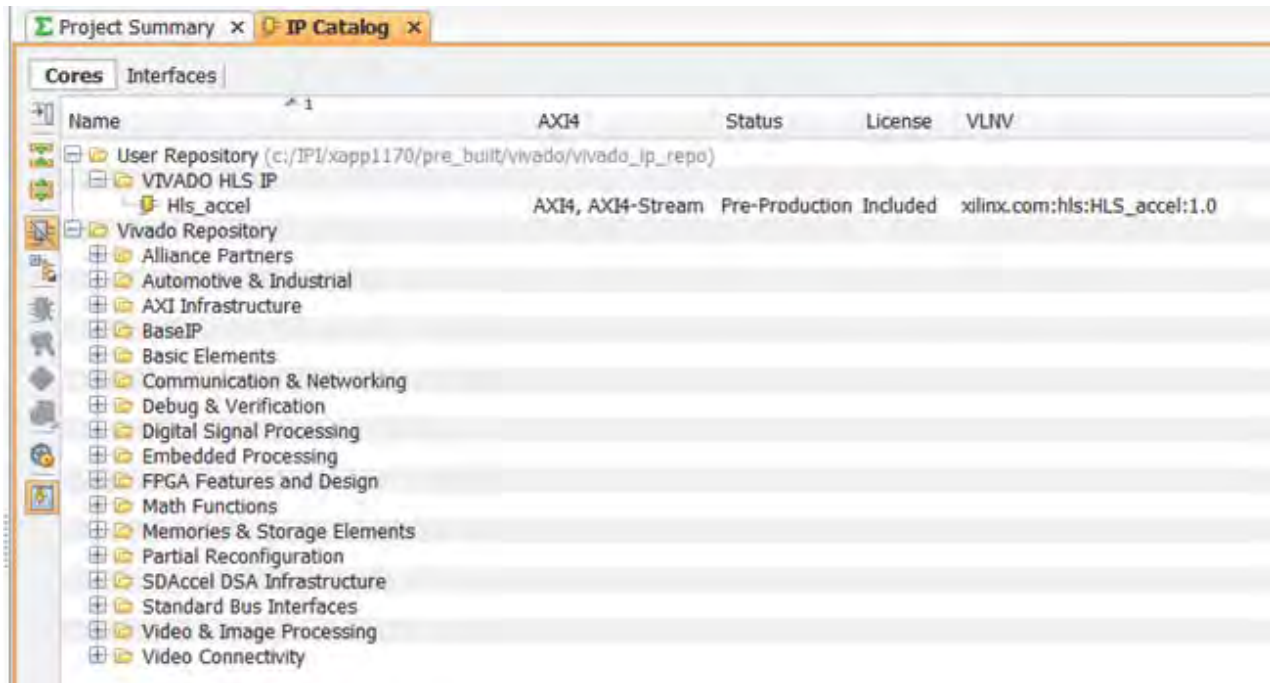


図 13 : HLS IP が追加された IP カタログ

7. [OK] をクリックしてダイアログ ボックスを閉じます。

**IP インテグレーターでブロック図を設計：**  
(詳細は、UG995 [参照 4] を参照してください。)

1. Flow Navigator の [IP Integrator] の下にある [Create Block Design] をクリックします。
  - a. 表示されたダイアログ ボックスに、「system」とデザイン名を入力します (図 14 参照)。

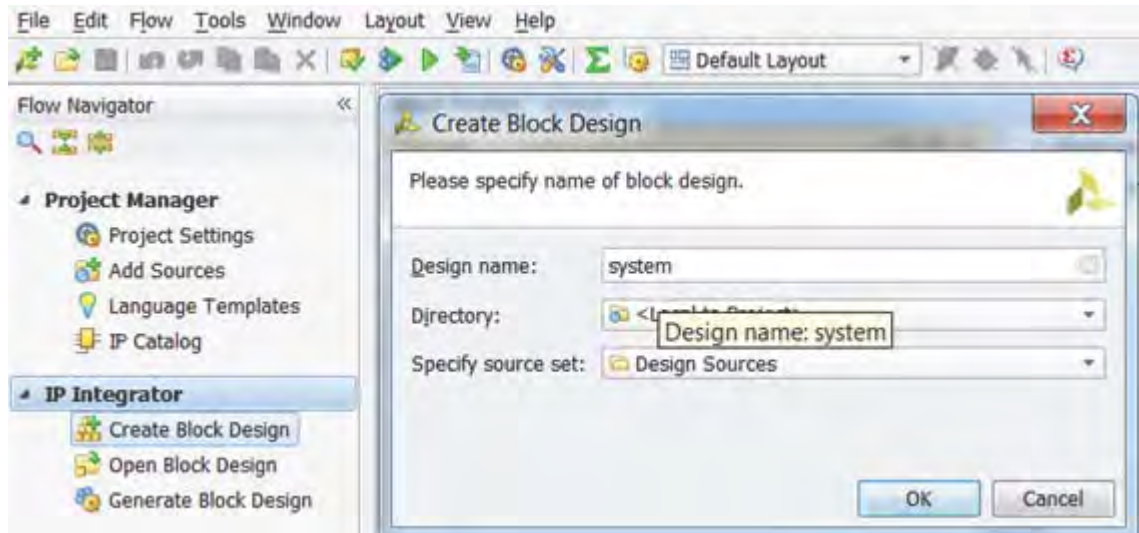
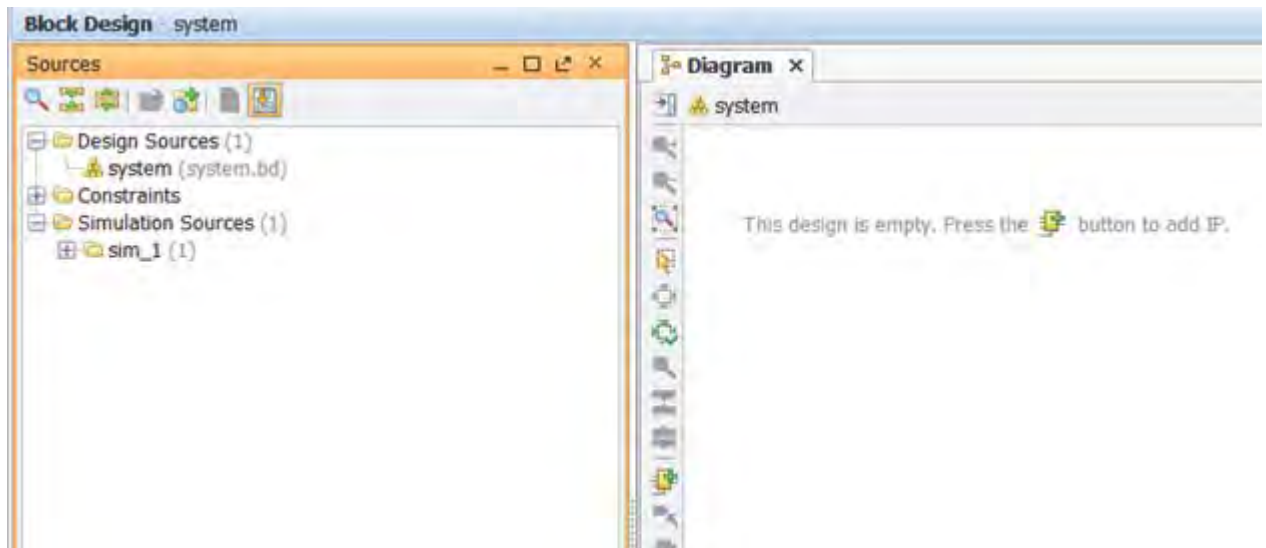


図 14 : [Create Block Design] ダイアログ ボックス

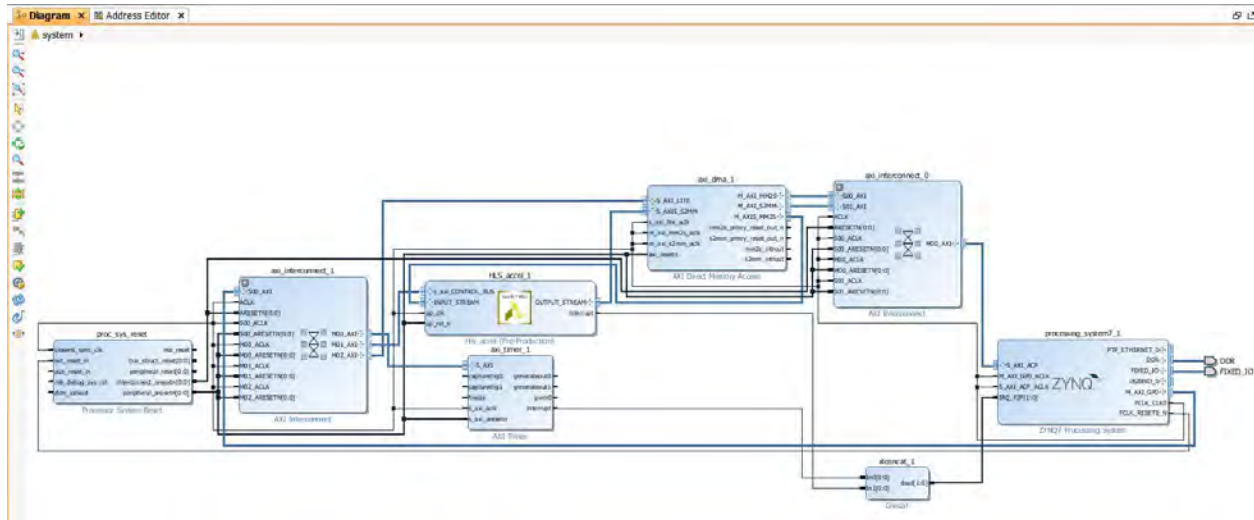
- b. [OK] をクリックします。
  - c. 画面右上に、空の [Diagram] ウィンドウが表示されます (図 15 参照)。



X15762-010316

図 15 : [Diagram] ウィンドウ

2. ブロック デザインを自動で構築するためのスクリプトがあらかじめ用意されています。[TCL Console] に「source 2015v4\_ipi\_xapp1170\_bd.tcl」と入力して、Enter キーを押します。
3. スクリプト終了後、[Diagram] ウィンドウで右クリックして [Regenerate Layout] を選択します。図 16 に示すブロック図が表示されます。



X15763-010316

図 16: ブロック デザイン

4. [Address Editor] ウィンドをクリックして、デザイン内のすべてのスレーブのメモリ マップを表示します。マップされていないスレーブがないことを確認します。マップされていないスレーブがある場合は、[Address Editor] 上で右クリックして [Auto Assign Address] を選択します (詳細は、UG994 [参照 10] の第 3 章を参照)。結果のアドレス マップ テーブルは、図 17 のようになります。

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
axi_dma_1					
Data_MM2S (32 address bits : 4G)					
processing_system7_1	S_AXI_ACP	ACP_IOP	0xE000_0000	4M	0xE03F_FFFF
processing_system7_1	S_AXI_ACP	ACP_DDR_LOWOCM	0x0000_0000	1G	0x3FFF_FFFF
processing_system7_1	S_AXI_ACP	ACP_QSPI_LINEAR	0xFC00_0000	16M	0xFCFF_FFFF
processing_system7_1	S_AXI_ACP	ACP_M_AXI_GP0	0x4000_0000	1G	0x7FFF_FFFF
Data_S2MM (32 address bits : 4G)					
processing_system7_1	S_AXI_ACP	ACP_DDR_LOWOCM	0x0000_0000	1G	0x3FFF_FFFF
processing_system7_1	S_AXI_ACP	ACP_QSPI_LINEAR	0xFC00_0000	16M	0xFCFF_FFFF
processing_system7_1	S_AXI_ACP	ACP_IOP	0xE000_0000	4M	0xE03F_FFFF
processing_system7_1	S_AXI_ACP	ACP_M_AXI_GP0	0x4000_0000	1G	0x7FFF_FFFF
processing_system7_1					
Data (32 address bits : 0x40000000 [ 1G ])					
axi_timer_1	S_AXI	Reg	0x4280_0000	64K	0x4280_FFFF
HLS_accel_1	s_axi_CONTROL_BUS	Reg	0x43C0_0000	64K	0x43C0_FFFF
axi_dma_1	S_AXI_LITE	Reg	0x4040_0000	64K	0x4040_FFFF

図 17: アドレス マップ テーブル

5. processing\_system7\_1 ブロックで [Presets] をクリックして、[図 18](#) に示すように ZC702 がすでに Current Preset であることを確認します。

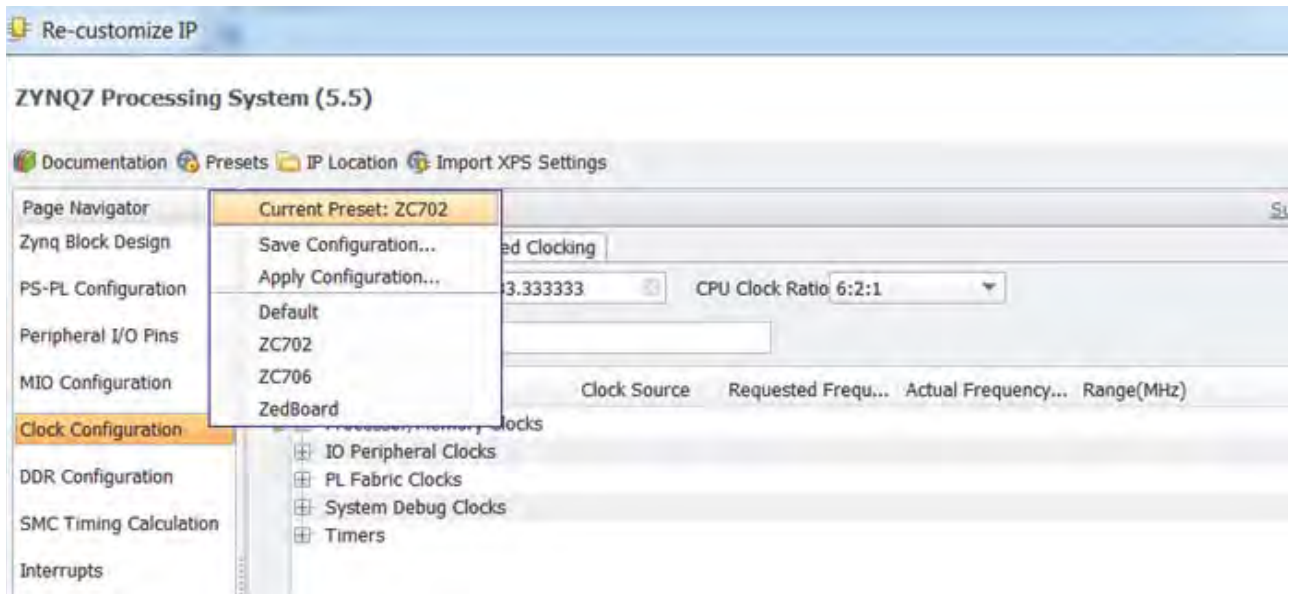
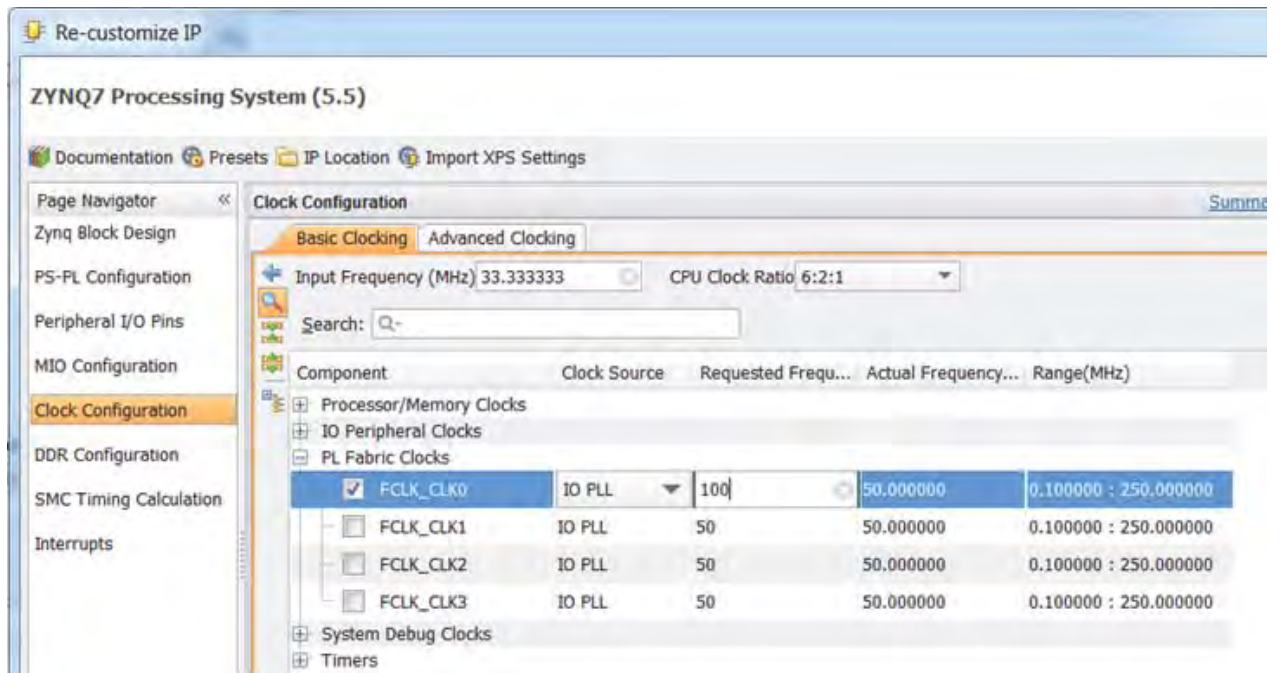


図 18 : Current Preset: ZC702

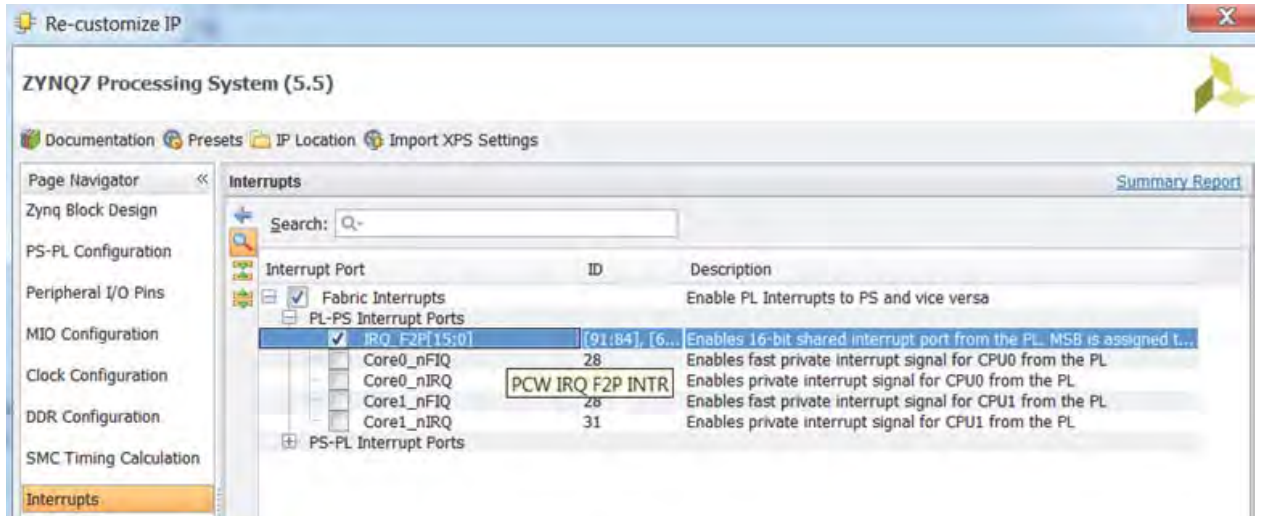
6. [Clock Configuration] → [PL Fabric Clocks] をクリックして、[図 19](#) に示すように FCLK\_CLK0 が 100MHz に設定されていることを確認します。



X15766-010316

図 19 : Zynq-7000 PL FCLK\_CLK0 の設定

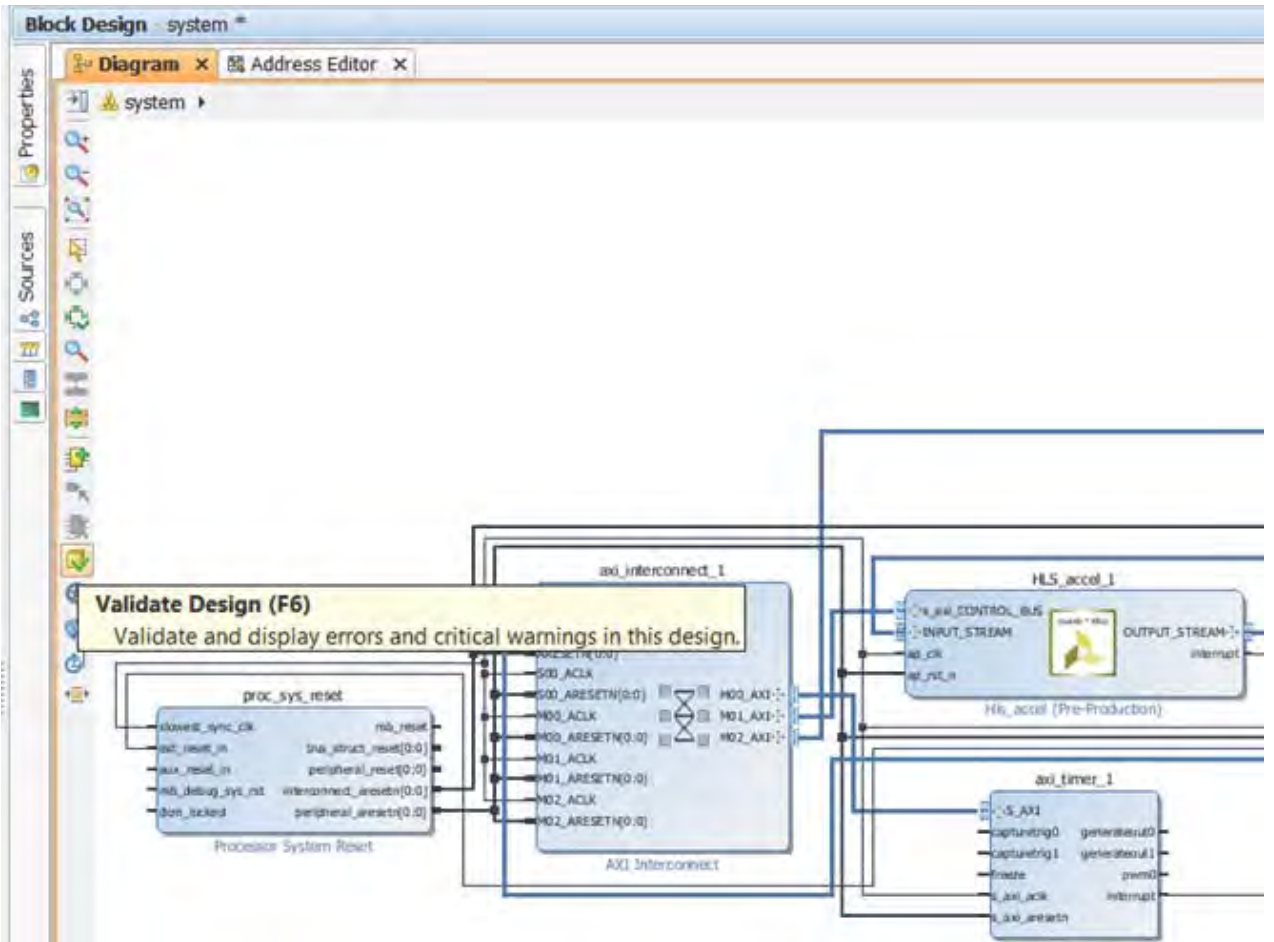
7. [PL-PS Interrupt Ports] が図 20 に示すように設定されていることを確認します。



X15767-010316

図 20: PL-PS 割り込みポートの設定

8. ツールバーの [Validate Design] をクリックして、ブロックデザインを検証します (図 21 参照)。



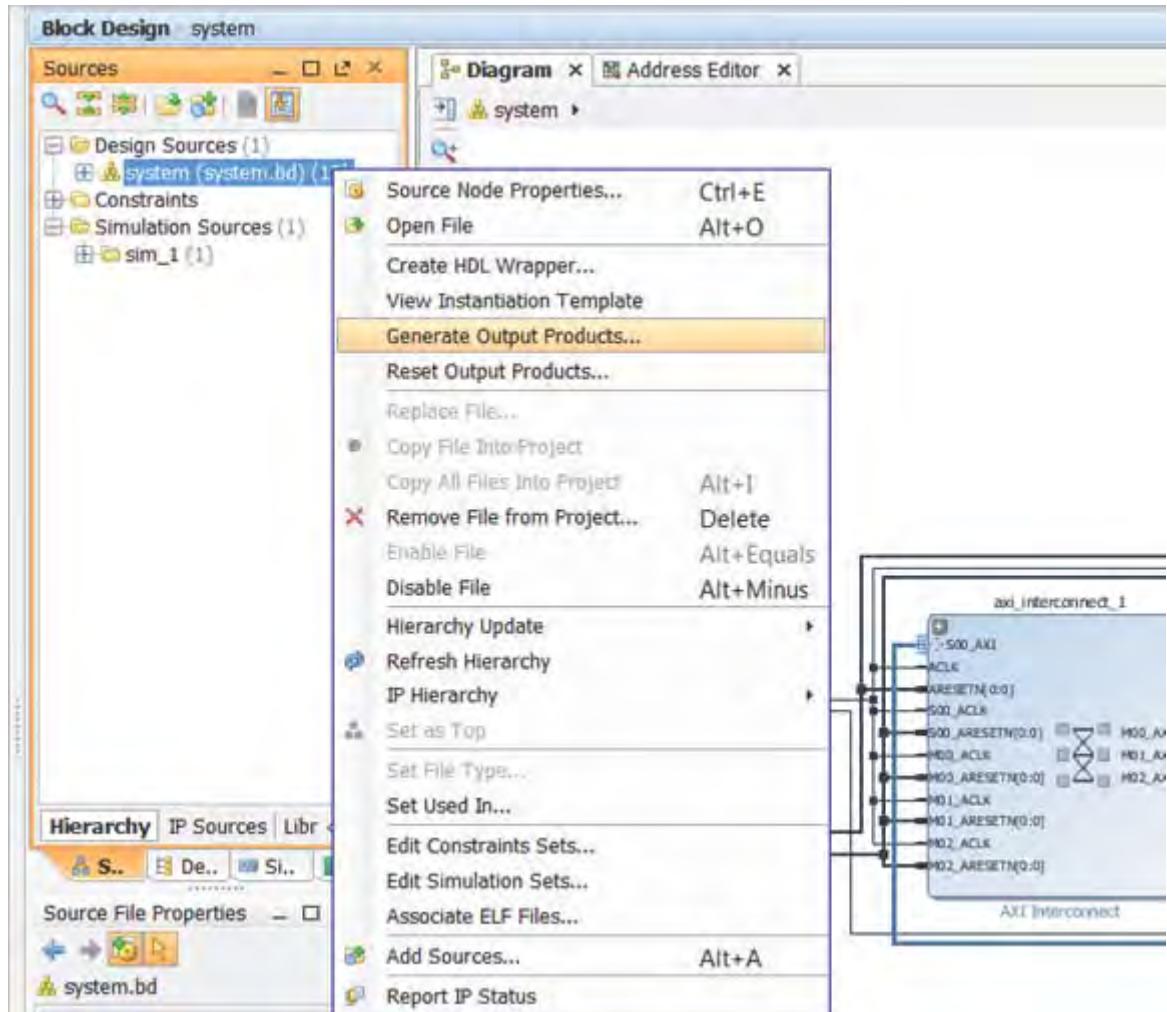
X15768-010316

図 21: デザイン検証

- 検証プロセス完了後、メニューの [File] → [Save Block Design] (または Ctrl + S キー) を使用してブロック デザインを保存します。

#### 出力ファイルの生成 :

- [Project Manager] の [Sources] ビューで `system.bd` を右クリックして [Generate Output Products] をクリックします (図 22 参照)。



X15769-010316

図 22 : 出力ファイルの生成

- 表示されたダイアログ ボックスで [OK] をクリックして、すべての出力ファイルの生成を開始します。
- HDL ラッパーを作成します。
  - [Project Manager] の [Sources] ビュー (前の手順と同じ手順およびメニュー) で `system.bd` を右クリックして [Create HDL Wrapper] をクリックします。
  - [OK] をクリックして通知メッセージを消去します。
- [Generate Bitstream] をクリックします。

5. SDK 用にハードウェアプラットフォームをエクスポートして、SDK を起動します (図 23 参照)。[Export Hardware] が選択されていることを確認します。[Include bitstream] はオンにします。

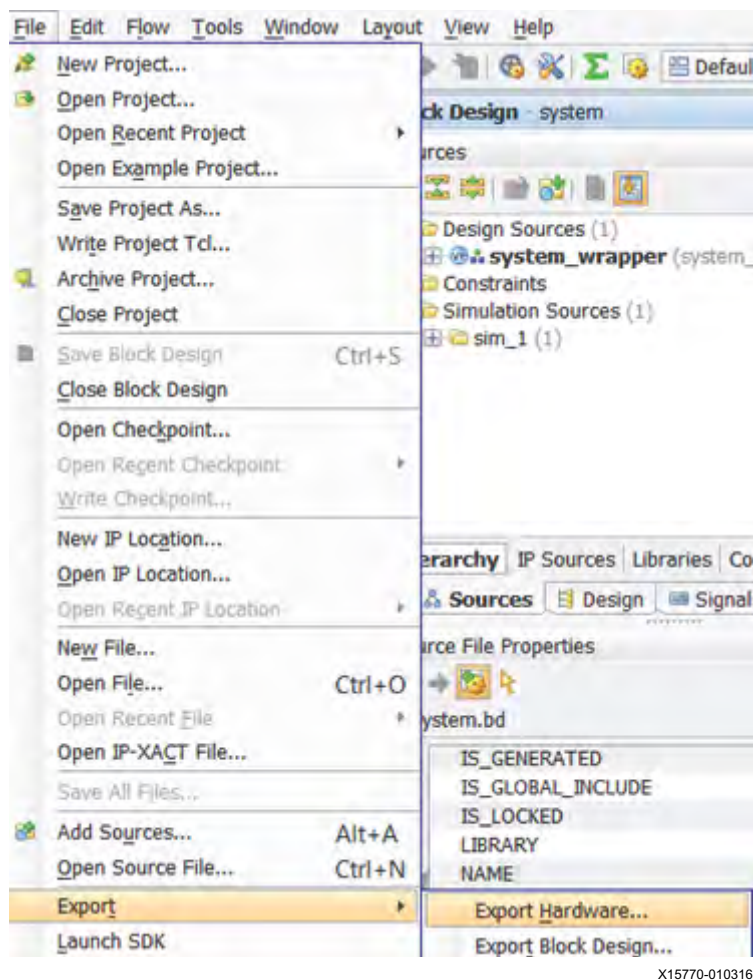
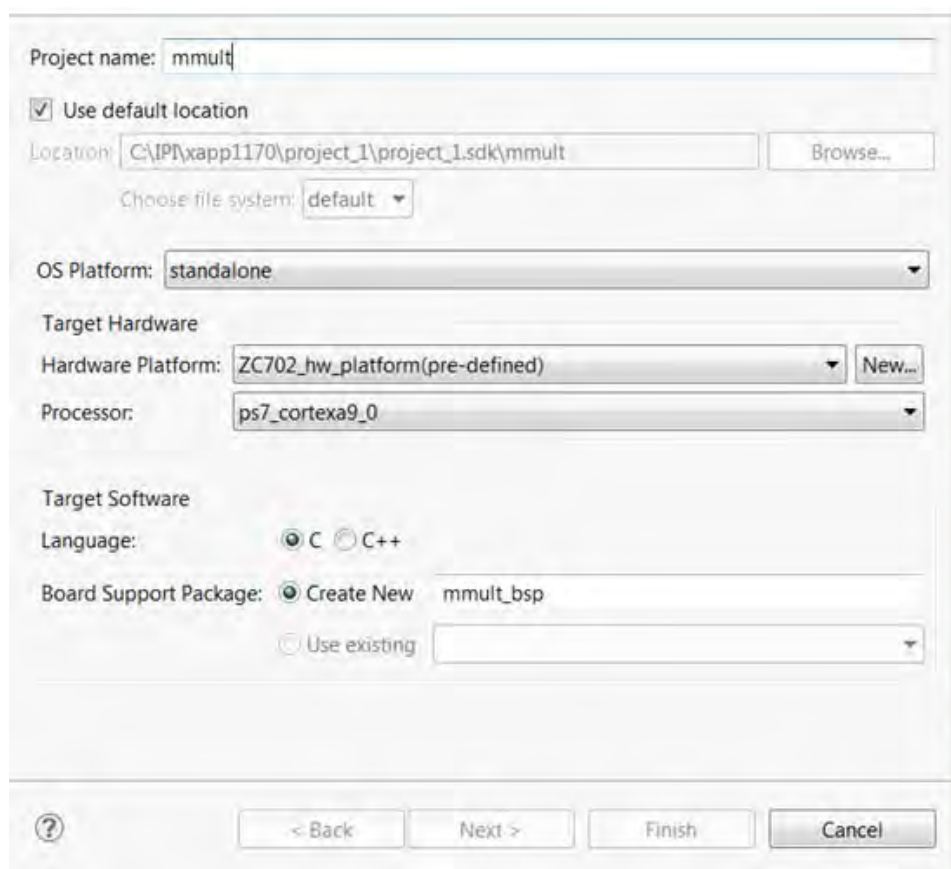


図 23 : ハードウェアのエクスポート

## SDK を使用する Zynq ソフトウェア デザイン

[Launch SDK] をクリックします (図 23 参照)。SDK が開くと、ソフトウェアプロジェクトを開始できます。Hello World アプリケーションを作成するには、次の手順に従います。

1. [File] → [New] → [Application Project] をクリックして新規プロジェクトを作成し、[Project name] に「mmult」と入力します (図 24 参照)。



X15771-010316

図 24 : SDK の New Project ウィザード

2. [Board Support Package] で [Create New] をクリックします。
3. [Next] をクリックします。
4. [Hello World] を選択します。
5. [Finish] をクリックします。この手順で、mmult スタンドアロン アプリケーションが作成および構築されます。

- 出力用にターミナルを使用するため、ターミナルアプリケーション (または SDK のビルトイン ターミナル) を起動して、[図 25](#) のように設定します。

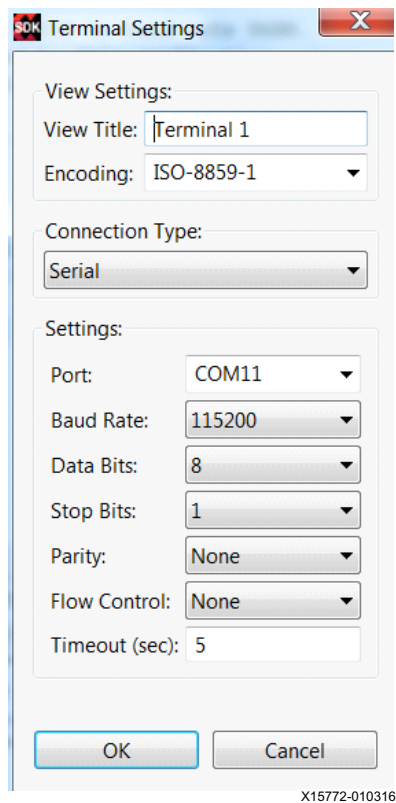


図 25 : UART Terminal の設定

- 右クリックで [Xilinx Tools] → [Program FPGA] を選択して、Zynq-7000 デバイスをプログラムします。

8. 図 26 に示すように、[Build Configurations] を [Release] モードに設定します。

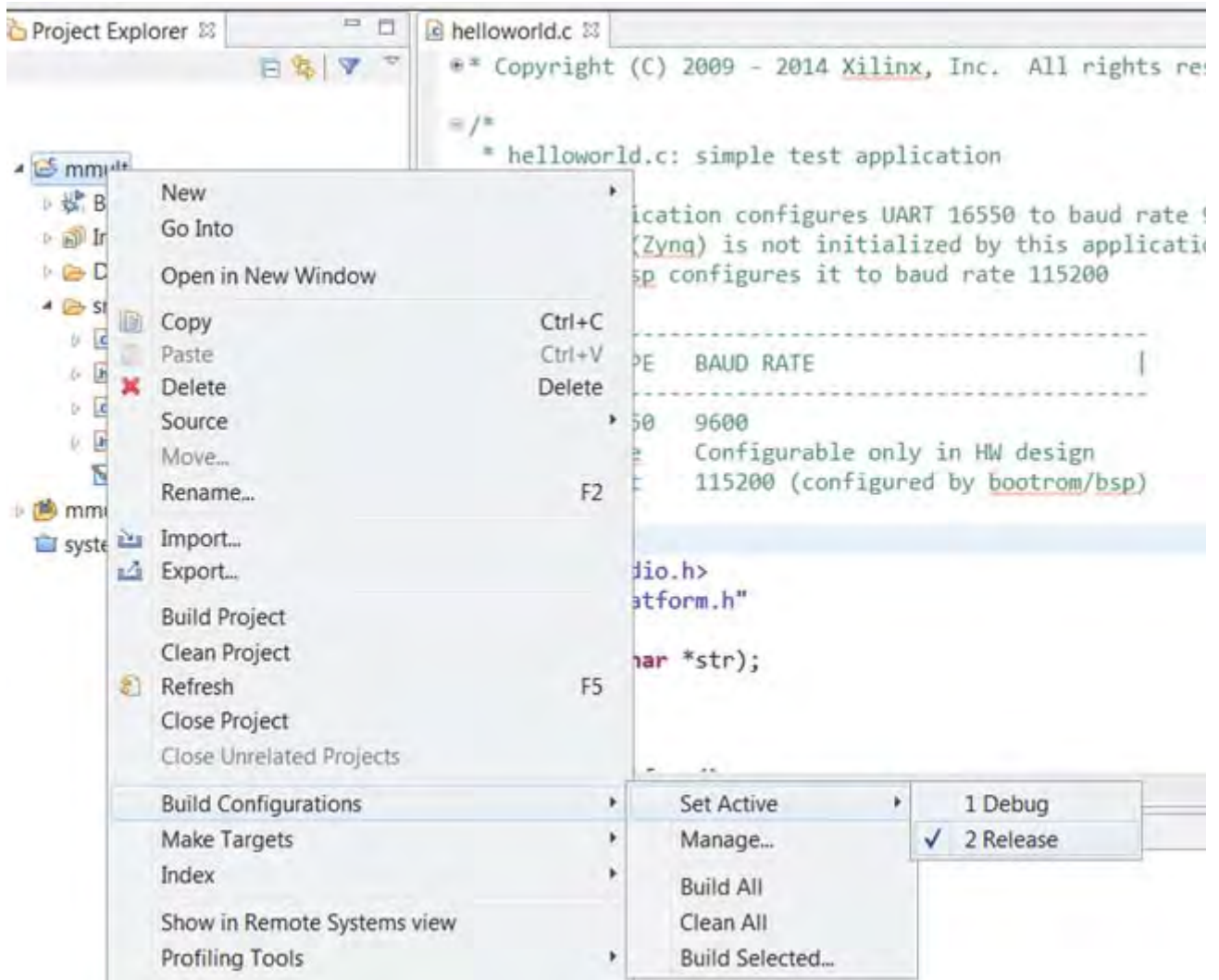
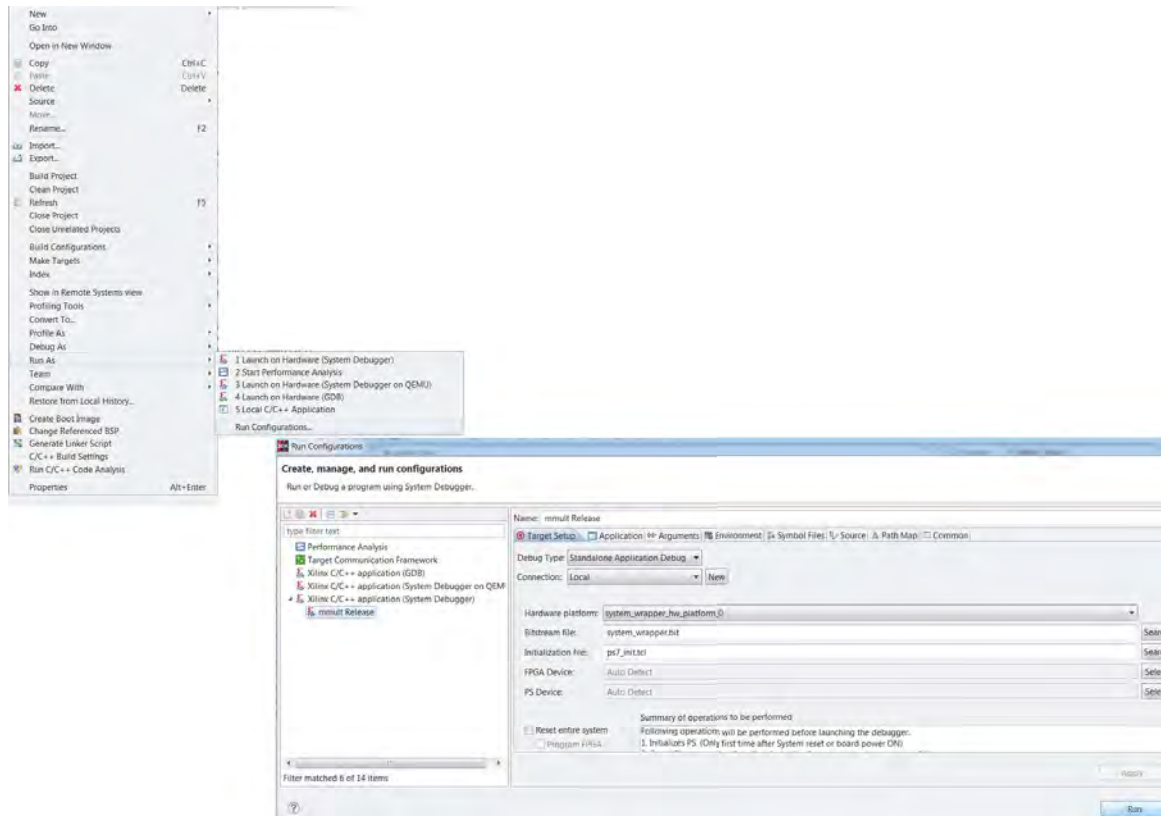


図 26: アクティブ コンフィギュレーションの設定

9. ボードでこのアプリケーションを実行するために、右クリックで [Run] → [Run Configurations] を選択して新しい実行コンフィギュレーションを作成します。GUI では、[Xilinx C/C++ application (System Debugger)] を選択して [New] をクリック (またはエントリをダブルクリック) します。これで、新しいコンフィギュレーションが生成されます。図 27 に示すデフォルト設定を使用してください。



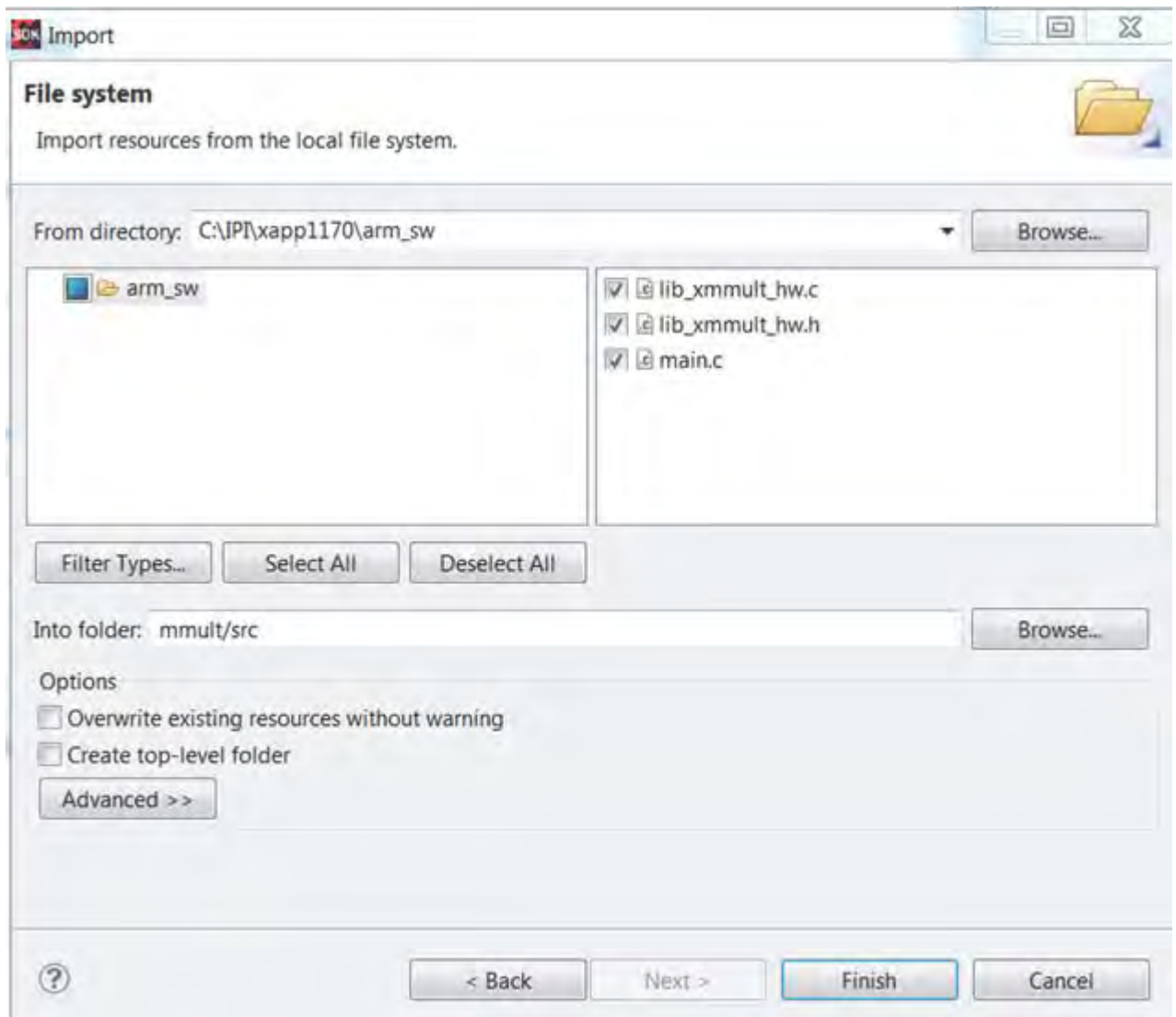
X15774-010316

図 27 : SDK の [Run Configurations]

10. アプリケーションを実行します。ターミナルに「Hello World」と表示されます。

次に、HLS で生成された HW アクセラレータを呼び出すソフトウェアを作成します。arm\_sw フォルダには、DMA の初期化、性能の測定、ハードウェア アクセラレータの呼び出しを実行する 3 つの C コード アプリケーション ファイルが含まれています。残りのすべてのアプリケーション ファイルは、Vivado HLS で自動生成されて SDK でインポートされます。

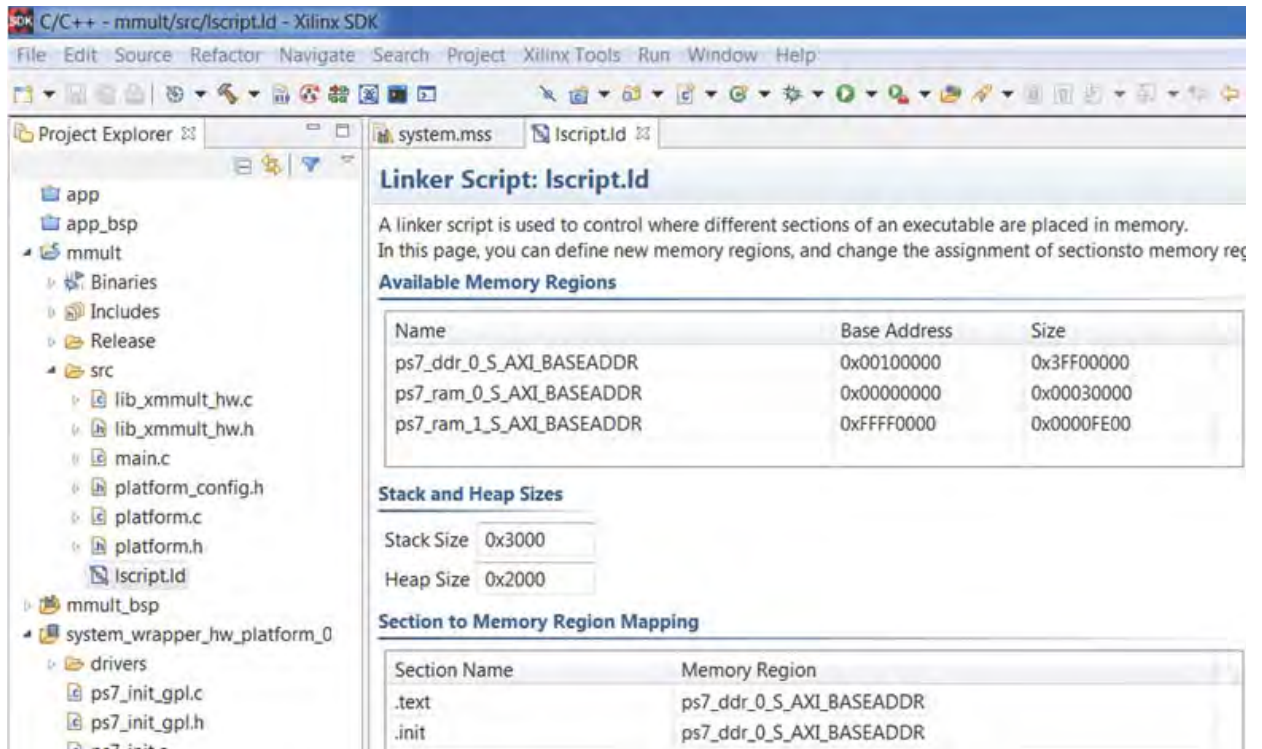
1. helloworld.c ファイルは、必要ないので削除できます。
2. 行列乗算の C ファイルを追加します。
  - a. mmult プロジェクトで src を右クリックします。
  - b. [Import] をクリックします。
  - c. [General] を選択します。
  - d. [File System] を選択して [Next] をクリックします。
  - e. arm\_sw ローカルファイルシステムを選択して、3 つのファイルを選択します (図 28 参照)。



X15775-010316

図 28 : ARM CPU 用のファイルをインポート

3. アプリケーションを実行する前にスタック サイズを変更する必要があります (図 29 参照)。
  - a. [Project Explorer] の mmult/src にある lscript.ld ファイルをダブルクリックします。
  - b. [Stack Size] を [0x2000] から [0x3000] に変更します。
  - c. ファイルを保存します。

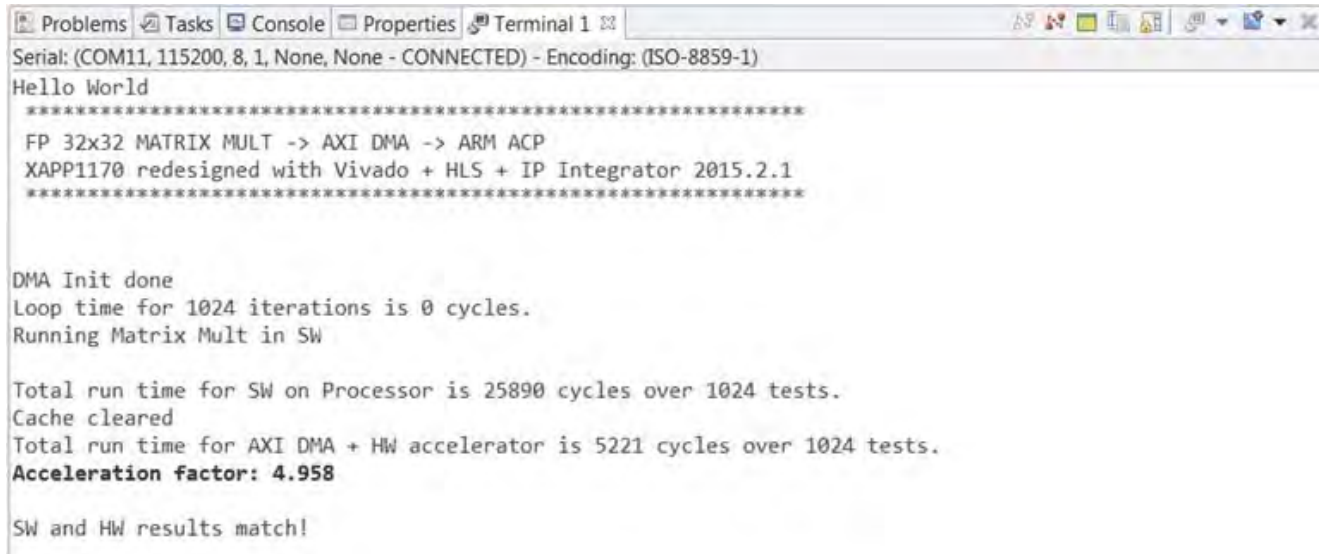


X15776-010316

図 29: スタック サイズの変更

4. ボード上このアプリケーションを実行するために、右クリックで [Run] → [Run Configurations] を選択して新しい実行コンフィギュレーションを作成します。GUI で [Xilinx C/C++ application (System Debugger)] を選択して [New] をクリック (またはエントリをダブルクリック) します。これで、新しいコンフィギュレーションが生成されます。デフォルト設定を使用してください。

5. アプリケーションを実行します。図 30 にターミナルの出力画面を示します。



```
Serial: (COM11, 115200, 8, 1, None, None - CONNECTED) - Encoding: (ISO-8859-1)
Hello World
*****
FP 32x32 MATRIX MULT -> AXI DMA -> ARM ACP
XAPP1170 redesigned with Vivado + HLS + IP Integrator 2015.2.1
*****

DMA Init done
Loop time for 1024 iterations is 0 cycles.
Running Matrix Mult in SW

Total run time for SW on Processor is 25890 cycles over 1024 tests.
Cache cleared
Total run time for AXI DMA + HW accelerator is 5221 cycles over 1024 tests.
Acceleration factor: 4.958

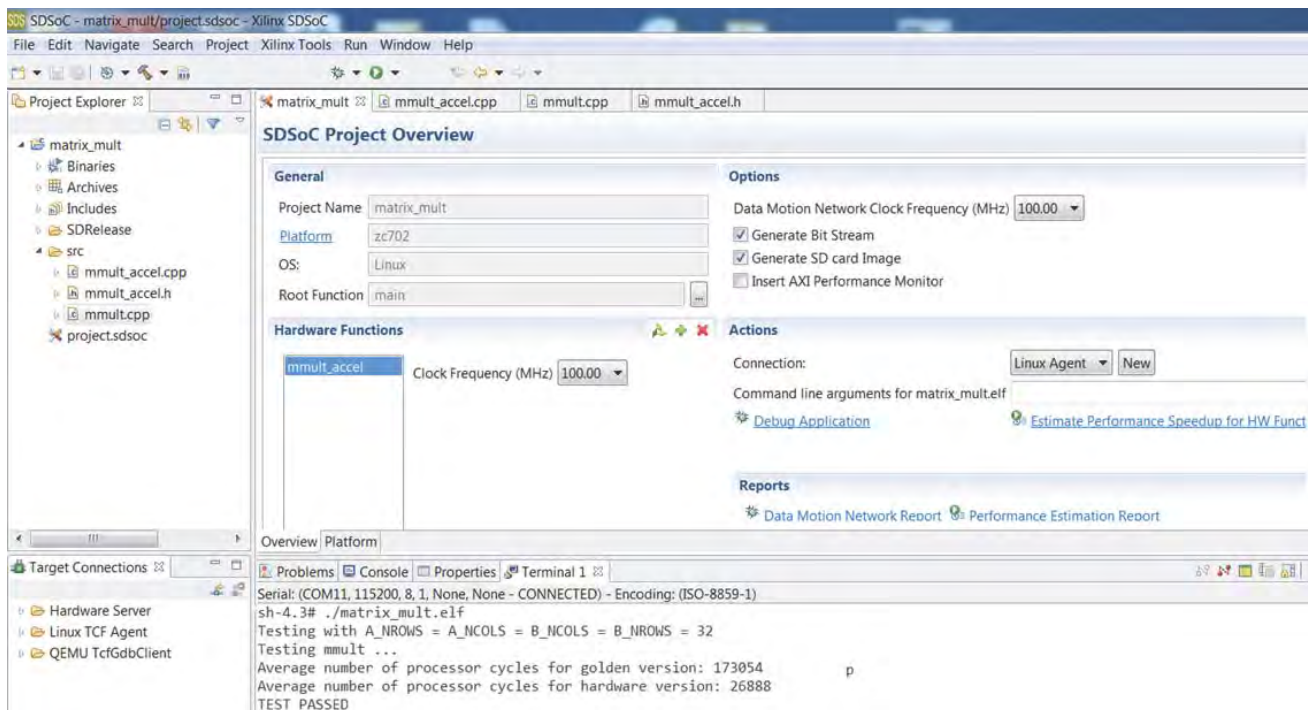
SW and HW results match!
```

図 30: ZC702 ボードでアプリケーションを実行した場合のターミナル出力画面

## SDSoC

ザイリンクスのSDSoCは、Zynq-7000 AP SoCプラットフォーム ([参照 11]) を使用してヘテロジニアスなエンベデッドシステムをインプリメントするための Eclipse ベースの統合開発環境です。SDSoC 環境には、プログラマブル ロジックでのソフトウェア アクセラレーションの自動化やシステム コネクティビティの自動生成機能などを実行し、システム全体の最適化を行う C/C++ コンパイラが含まれます。アプリケーションは C/C++ コードで記述され、プログラマがターゲットプラットフォームとハードウェアにコンパイルするアプリケーション内の関数のサブセットを特定します。その後、SDSoC システム コンパイラによりアプリケーションがハードウェアとソフトウェアにコンパイルされ、ファームウェア、オペレーティングシステム、アプリケーション実行ファイルを含むブート イメージを含めた完全なエンベデッドシステムが Zynq デバイスにインプリメントされます ([参照 12])。

IPI、HLS、SDK ツールを使用して、このアプリケーションをゼロから開発する場合は約 2 日かかりますが、SDSoC を使用すれば 1 時間以内で完成させることができます (図 31 参照)。



X15778-010316

図 31 : 行列乗算アプリケーションのSDSoCプロジェクト

## まとめ

C または C++ で記述された浮動小数点デザインを FPGA デバイスに素早く簡単に実装できるようになりました。このようにデザインを実装することで、FPGA が持つ並列処理性能、低消費電力、組み込まれた CPU、低コストなど、さまざまな利点をいかすことができます。その他の C/C++ フローと同様に、完全なツールチェーンでフロー全体をとおして包括的な解析を行うことで性能のトレードオフを可能にします。サンプルアプリケーションは、Vivado HLS ツールを使用して 32 ビットの浮動小数点精度に最適化された 32x32 行列乗算コアです。

C/C++ コードでモデル化された浮動小数点行列乗算は、Vivado HLS で、RTL デザインに素早く実装および最適化できます。その後、IP コアとしてエクスポートでき、Zynq-7000 AP SoC PL サブシステムの DMA コアを用い、AXI4-Stream インターフェイスを介して Zynq-7000 AP SoC PS の ACP に接続されます。

100MHz クロック周波数で動作する行列乗算ハードウェア パリフェラルは、666MHz クロック周波数で動作する ARM CPU でソフトウェアを実行するよりも約 5 クロック サイクル早く演算を実行します。

最後に、このアプリケーション ノートで説明した設計手順は、SDSoC という新しいシステム デザイン フローを使用することで、すべて自動化できます。

## リファレンス デザイン

このアプリケーション ノートのリファレンス デザインは、次のリンクからダウンロードできます。

<https://secure.xilinx.com/webreg/clickthrough.do?cid=343614>

次の 2 つのフォルダーがあります。

- empty - HLS および IPI プロジェクトを最初から構築するための C++ コードおよび TCL スクリプトが含まれます。
- pre\_built - 開発済みの HLS および IPI プロジェクトが含まれます。

表 1 に、リファレンス デザインの詳細を示します。

表 1: リファレンス デザインの詳細

パラメーター	説明
一般	
開発者	Daniele Bagni、Antonello Di Fresco (ザイリンクス)
ターゲット デバイス	Zynq-7000 AP SoC
ソース コードの提供	あり
ソース コードの形式	C および合成スクリプト
既存のザイリンクス アプリケーション ノート/リファレンス デザイン、またはサードパーティからデザインへのコード/IP の使用	なし
シミュレーション	
論理シミュレーションの実施	あり
タイミングシミュレーションの実施	なし
論理シミュレーションおよびタイミングシミュレーションでのテストベンチの利用	あり
テストベンチの形式	C
使用したシミュレータ/バージョン	Vivado シミュレータ 2015.4
SPICE/IBIS シミュレーションの実施	なし

表 1: リファレンス デザインの詳細 (続き)

パラメーター	説明
使用したインプリメンテーション ツール/バージョン	Vivado Design Suite 2015.4
スタティック タイミング解析の実施	あり
ハードウェア検証	
ハードウェア検証の実施	あり
使用したハードウェア プラットフォーム	ザイリンクス ZC702 ボード

## 参考資料

注記: 日本語版のバージョンは、英語版より古い場合があります。

- 『Zynq-7000 All Programmable SoC: コンセプト、ツール、テクニック ガイド (CTT)』(UG873: [英語版](#)、[日本語版](#))
- 「Floating-Point Design with Xilinx's Vivado HLS」 James Hrica、Xcell Journal ([2012 年 第 4 四半期](#))
- 『Vivado HLS および System Generator for DSP を用いた浮動小数点 PID コントローラー デザイン』([XAPP1163](#))
- 『Vivado Design Suite チュートリアル: IP インテグレーターを使用した IP サブシステムの設計』([UG995](#))
- 『Zynq7000 XC7Z020 All Programmable SoC 向け ZC702 評価ボード ユーザー ガイド』([UG850](#))
- 『Vivado Design Suite ユーザー ガイド: 高位合成』(UG902: [英語版](#)、[日本語版](#))
- 『Vivado Design Suite チュートリアル: 高位合成』(UG871: [英語版](#)、[日本語版](#))
- 『ザイリンクス AXI リファレンス ガイド』([UG761](#))
- 『LogiCORE IP AXI DMA 製品ガイド』(PG021: [英語版](#)、[日本語版](#))
- 『Vivado Design Suite ユーザー ガイド: IP インテグレーターを使用した IP サブシステムの設計』(UG994: [英語版](#)、[日本語版](#))
- 『SDSoC 環境ユーザー ガイド』([UG1027](#))
- 『Using the SDSoC IDE for System-level HW-SW Optimization on the Zynq SoC』Daniele Bagni、Nick Ni、Xcell Software Journal、issue 1 ([2015 年 第 3 四半期](#))

## 改訂履歴

次の表に、この文書の改訂履歴を示します。

日付	バージョン	内容
2016 年 1 月 21 日	2.0	Vivado 2015.4 に基づいてリリース
2013 年 6 月 29 日	1.1	Vivado HLS 2012.2 および PlanAhead (ISE/XPS/SDK) 14.4 に基づいてリリース

## 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」、および全て受領者の責任で(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)、全ての保証および条件を負わない(否認する)ものとします。また、(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、<http://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下さい。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となります。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。  
<http://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。

### 自動車のアプリケーションの免責条項

ザイリンクスの製品は、フェイルセーフとして設計されたり意図されてはならず、また、フェイルセーフの動作を要求するアプリケーション(具体的には、(I)エアバッグの展開、(II)車のコントロール(フェイルセーフまたは余剰性の機能(余剰性を実行するためのザイリンクスの装置にソフトウェアを使用することは含まれません)および操作者がミスをした際の警告信号がある場合を除きます)、(III)死亡や身体傷害を導く使用、に関するアプリケーション)を使用するために設計されたり意図されたりもしていません。顧客は、そのよう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を単独で負います。

© Copyright 2013-2016 Xilinx, Inc. Xilinx, Xilinx のロゴ、Artix、ISE、Kintex、Spartan、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。すべてのその他の商標は、それぞれの保有者に帰属します。

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](mailto: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。