Model Composer を使用した PID コントローラー デザイン

XAPP1341 (v1.0) 2019 年 3 月 14 日

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

概要

このアプリケーション ノートでは、『Vivado HLS および System Generator for DSP を使用した浮動小数点 PID コントローラー デザイン』 (XAPP1163) で説明したコンセプトを拡張し、XMC (Xilinx Model Composer) デザイン ツールを使用して MathWorks 社の Simulink® 環境で PID (Proportional-Integral-Derivative: 比例-積分-微分) 制御アルゴリズムをより高い抽象度で実装する方法を説明します。Model Composer は、ザイリンクス デバイスでのプロダクション品質のアルゴリズムの設計、シミュレーション、および実装を可能にする、Simulink のプラグインとして設計されています。アルゴリズム仕様からプロダクション品質の実装への変換は、Vivado® 高位合成 (HLS) ツールを併用し、自動最適化および自動コード生成によって可能です。さらに、自動テストベンチ作成機能を使用すると、Simulink での実行可能な仕様と合成された RTL 間の機能的等価性を検証できます。

このアプリケーション ノートでは、Model Composer を使用して PID アルゴリズムを実装する次の 2 つの方法を解説します。

- Simulink 環境で、Model Composer ブロック ライブラリから提供されるザイリンクス用に最適化されたネイティブ ブロックを使用。
- Simulink 内にカスタム Model Composer ブロックとしてインポートできる、ファームウェアをカスタマイズ可能な C ベース Math Sequencer 関数を使用。

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

MathWorks 社の MATLAB® および Simulink 製品ファミリは、線形および非線形動的システムをモデル化、解析、および調整するための包括的な設計環境を提供します。Model Composer は Simulink 環境に適合したツールで、制御エンジニアはこれを使用してアルゴリズム デザインをザイリンクス デバイスへ展開できます。

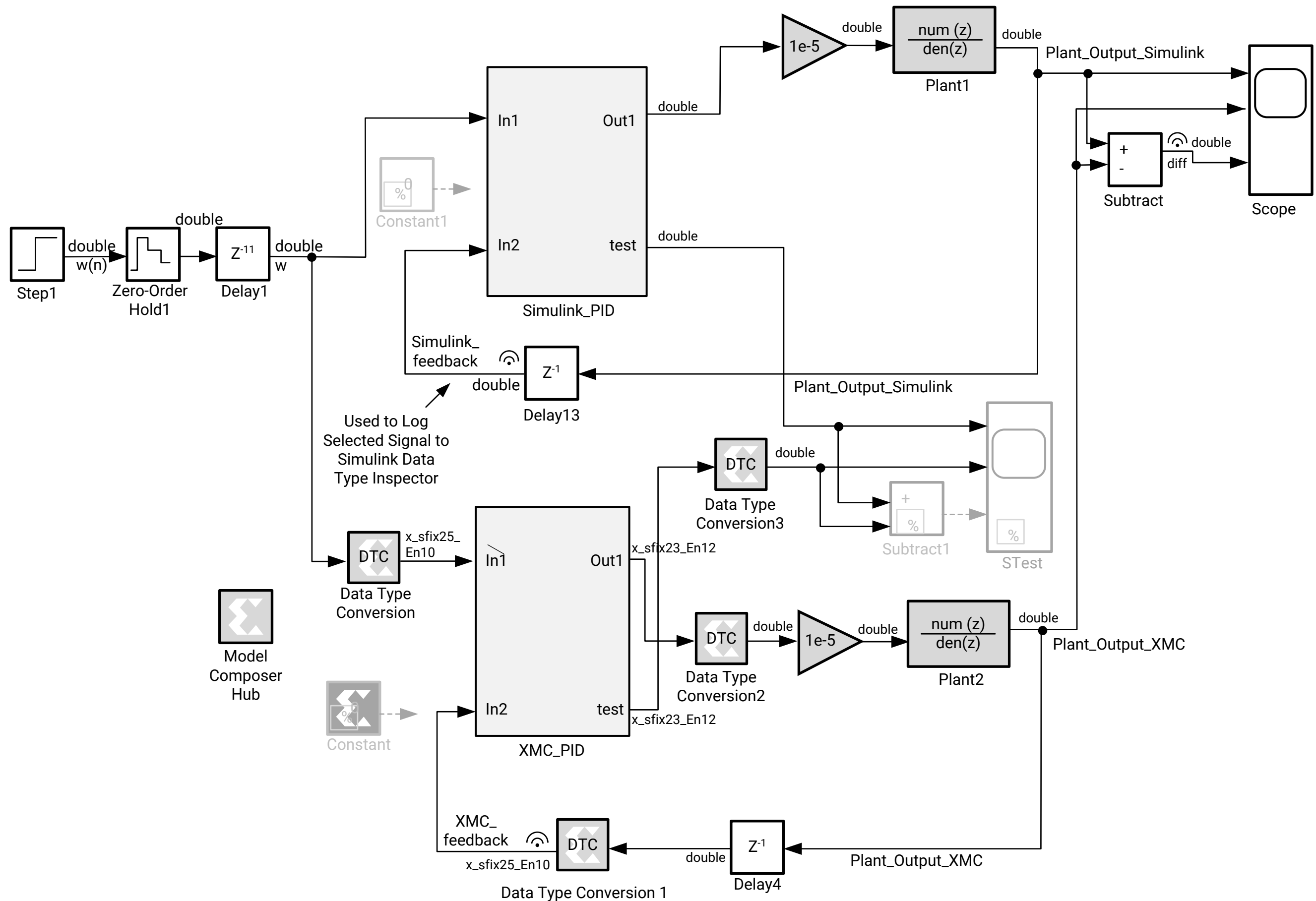
XMC は、次の利点により制御アルゴリズムの展開を高速化できます。

- ザイリンクス用に最適化された math ライブラリおよび線形代数ライブラリを使用して実装モデルを高い抽象度で設計できる。
- Simulink アドオン ツールボックスや MATLAB ソース コードとの統合によってテストベンチ開発が容易になる。
- Simulink の数多くの視覚化およびデバッグ方法を利用することで検証が容易になる。
- ビット精度の高い C++ モデルを使用することでシミュレーションのパフォーマンスが向上する。
- ネイティブの XMC ブロックまたはインポートされた C/C++ を使用した柔軟な実装が可能。
- Simulink シミュレーション時に Microsoft Visual C または GNU デバッガーを使用してインポートされた C/C++ モデルをその時点でデバッグできる。
- C++、Vivado IP カタログ、または System Generator を使用してデザインおよびテストベンチを評価およびエクスポートできる。

アプローチ 1: ネイティブのザイリンクス Model Composer ブロック ライブラリを使用する

次の図は、プラントとエラー フィードバックの両方を含む閉ループ システム モデル内の標準 PID コントローラーを、Simulink ブロック図と XMC ブロック図で対比させて示しています。

図 1: 閉ループ制御システム内の Simulink PID ブロック



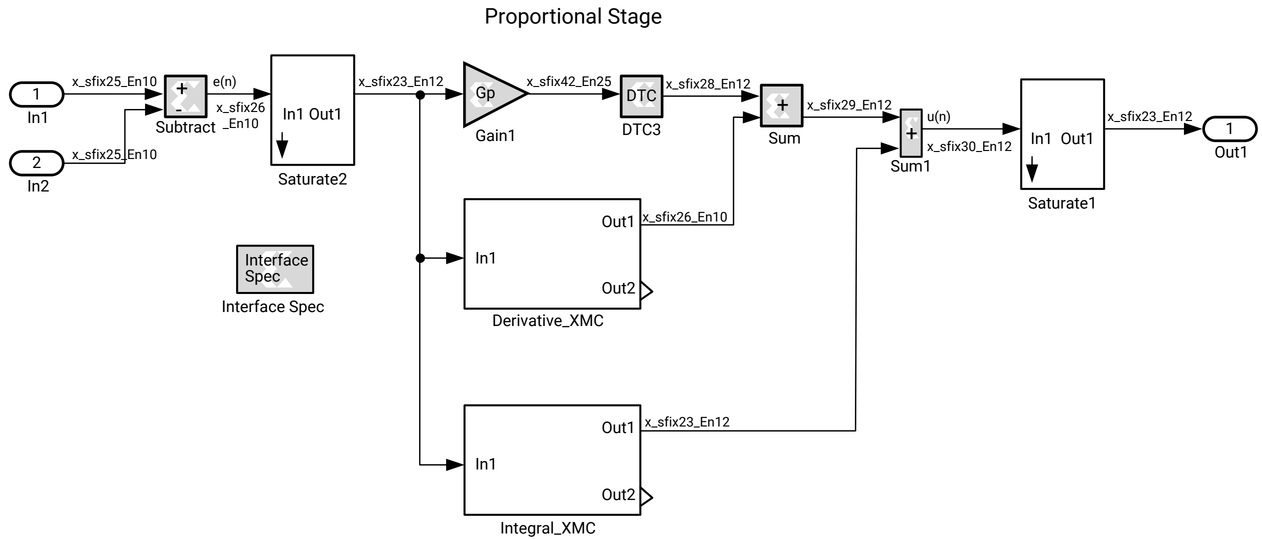
Model Composer で PID コントローラーの Simulink ゴールデン リファレンス モデルから浮動小数点実装モデル (`ClosedLoopPID_XMC_spfp.slx`) に移行するには、Simulink で math を表現するか、ネイティブの Simulink ブロックを Model Composer ブロック ライブラリのザイリンクス用に最適化された同等のビット精度ブロックと置き換える必要があります。これにより、Simulink と同じ抽象度で作業できます。

ゴールデン リファレンス モデルと Model Composer ハードウェア実装モデルの両方を同じ環境で使用できれば、両者の機能的等価性を検証するプロセスが大幅に簡素化されます。このデザインは、同じ入力スティミュラスを使用して両方のサブシステムを駆動し、Simulink 内のデバッグ機能と視覚化機能を活用します。これにより、結果を簡単に比較して実装に必要なデザインのトレードオフを明確にできます。

シミュレーションから Vivado HLS に対応する IP パッケージ化および C++ コードの最適化まで

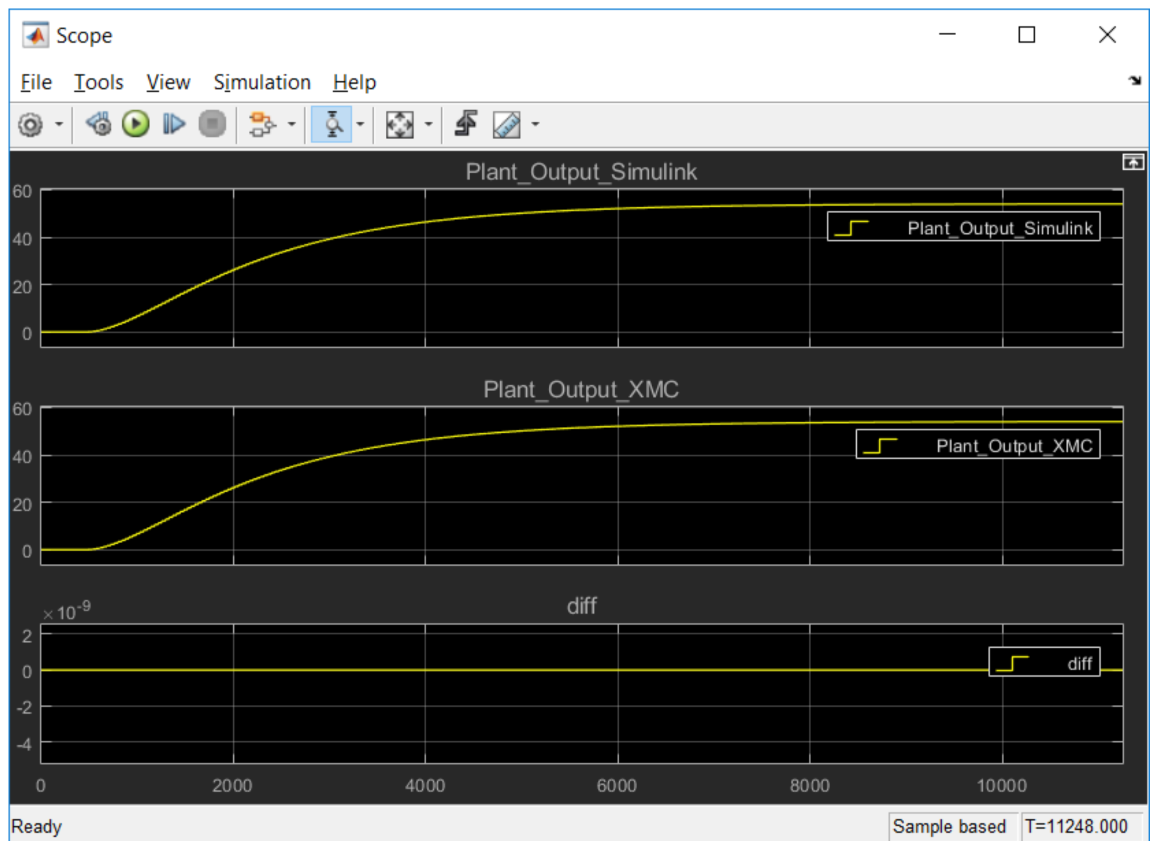
次の図に、固定小数点バージョンの XMC 実装モデル (`ClosedLoopPID_XMC.slx`) を示します。

図 2: XMC_PID 固定小数点デザイン



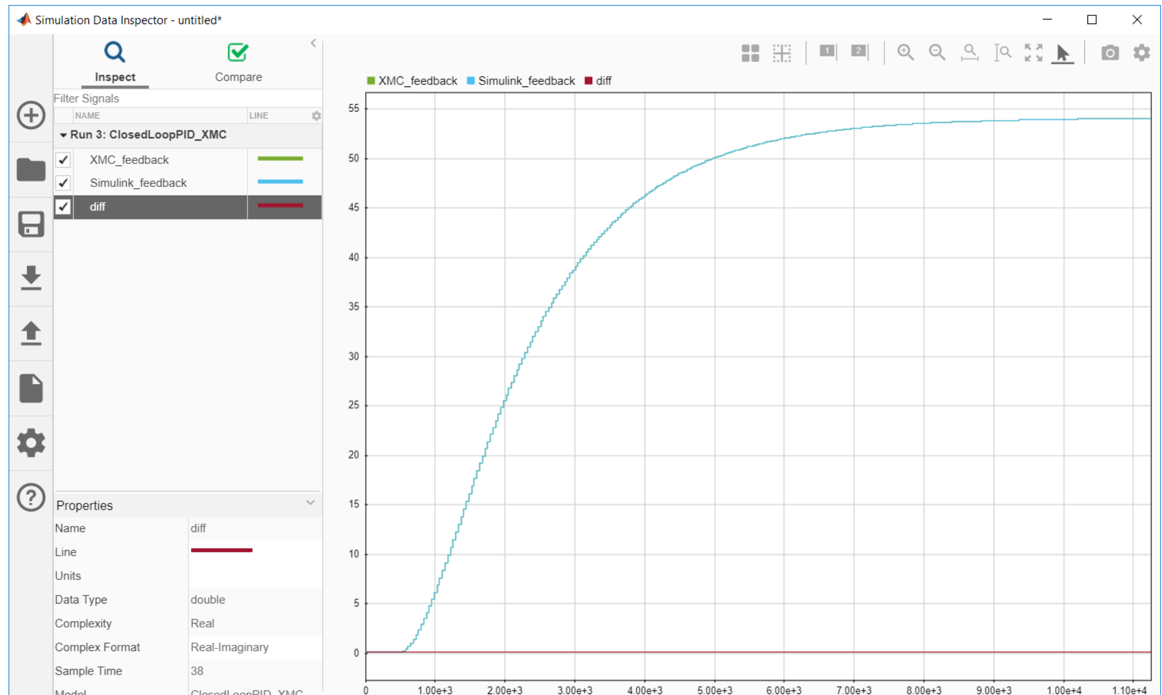
Simulink スコープ、記録された信号のログ、および Simulation Data Inspector を使用すると、Simulink 倍精度浮動小数点のシミュレーション結果をすばやく、かつ簡単に XMC 実装と比較できます。次の図に、従来の Simulink スコープを使用してキャプチャされたフィードバック データパスを示します。倍精度 Simulink と固定小数点 XMC モデル間の誤差を示す、これら 2 つの信号間の差 (diff) も示しています。27 ビットのダイナミック レンジを持つこの PID 実装では、固定小数点と浮動小数点間に大きな差はありません。

図 3: Simulink スコープ シミュレーションの結果



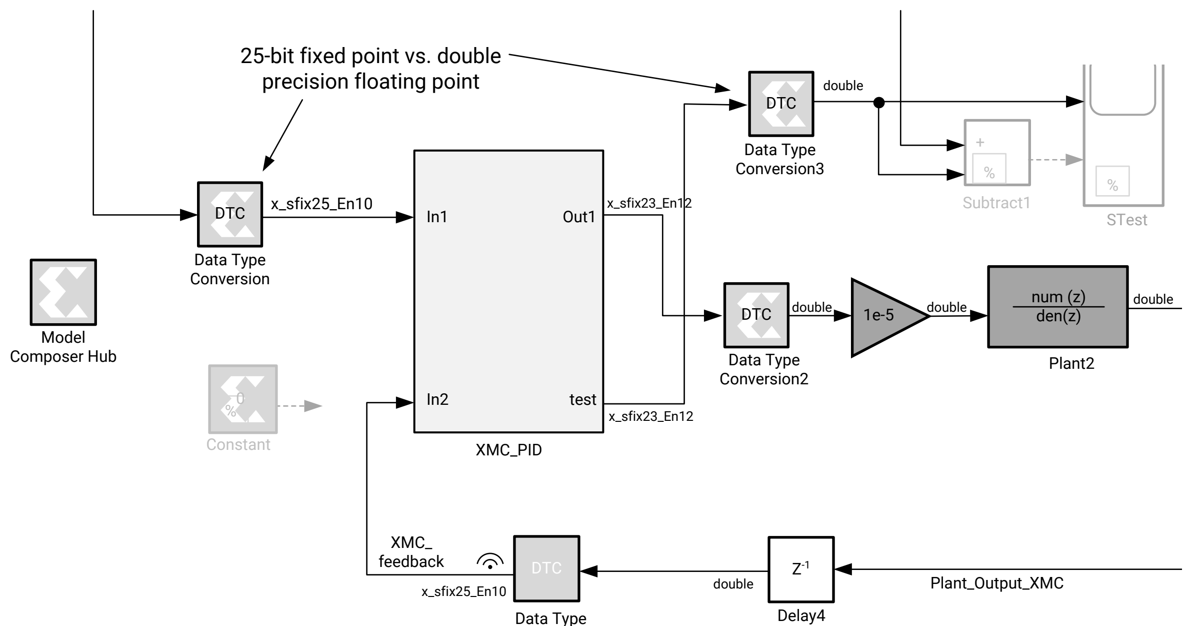
Simulation Data Inspector を使用し、同じデータをキャプチャして解析することもできます。

図 4: Simulation Data Inspector



XMC Data Type Conversion (DTC) 演算子を使用し、浮動小数点データ型と固定小数点データ型の間で量子化レベルを要件に応じて変更し、変更後の精度によって必要なユーザー特性を持つシステム応答がもたらされるかどうかを判断します。たとえば、次の図に示すように、固定小数点と浮動小数点の両方の DTC が同じシミュレーションで使用されているとします。

図 5: XMC DTC の使用

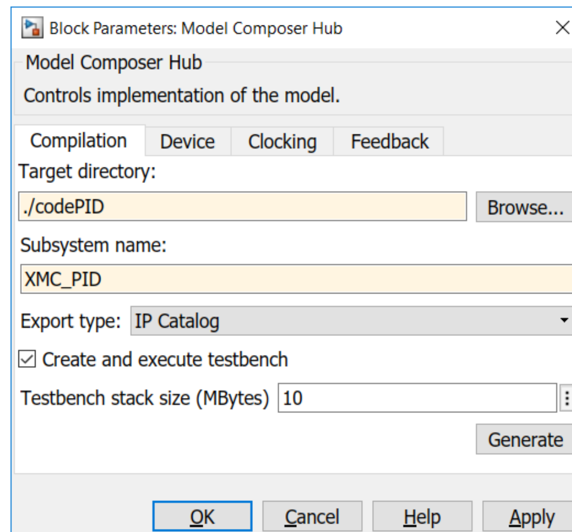


ユーザーは DTC でデータ型を指定するだけで精度を設定できます。

同じデザインでデータ型が異なる場合のリソース、レイテンシ、および達成可能なクロック周波数をデモして比較する目的で、XMC 固定小数点 (`ClosedLoopPID_XMC.slx`) と単精度浮動小数点 (SPFP) PID モデル (`ClosedLoopPID_XMC_spfp.slx`) の両方がリファレンス デザインの一部として含まれています。

自動で生成された Vivado IP カタログに [Create and execute testbench] オプションを指定すると、シミュレーション結果が正常終了かエラーかを含む HLS プロジェクトが作成されます。

図 6: Simulink ソース スティミュラスを使用した HLS プロジェクト (テストベンチの合否を含む) の作成



自動で作成された固定小数点および SPFP の HLS XMC PID プロジェクトを使用すると、結果として得られる C++ ベース デザインをさらに評価または最適化できます。

図 7: PID 固定小数点 HLS 実装の結果

Export Report for 'XMC_PID'

General Information

Report date: Mon Dec 17 10:18:50 -0600 2018
Project: XMC_PID_prj
Solution: solution1
Device target: xc7z020clg484-1
Implementation tool: Xilinx Vivado v.2018.3

Resource Usage

	VHDL
SLICE	221
LUT	459
FF	822
DSP	5
BRAM	0
SRL	0

Final Timing

	VHDL
CP required	3.333
CP achieved post-synthesis	4.066
CP achieved post-implementation	3.216

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	76	80	81	77	81	82

図 8: PID 単精度浮動小数点 HLS 実装の結果

Export Report for 'XMC_PID_SPFP'

General Information

Report date: Mon Dec 17 11:35:21 -0600 2018
Project: XMC_PID_SPFP_prj
Solution: solution1
Device target: xc7z020clg484-1
Implementation tool: Xilinx Vivado v.2018.3

Resource Usage

	Verilog
SLICE	409
LUT	981
FF	1606
DSP	6
BRAM	0
SRL	18

Final Timing

	Verilog
CP required	3.509
CP achieved post-synthesis	4.057
CP achieved post-implementation	3.454

Cosimulation Report for 'XMC_PID_SPFP'

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	138	141	142	139	142	143

表 1: PID 固定小数点と PID 浮動小数点の比較

XMC データ型	DSP	LUT	フリップフロップ	ブロック RAM	レイテンシ (クロック)	クロック (MHz)
固定小数点	5	459	822	0	81	311
SPFP	3	958	1761	0	142	290

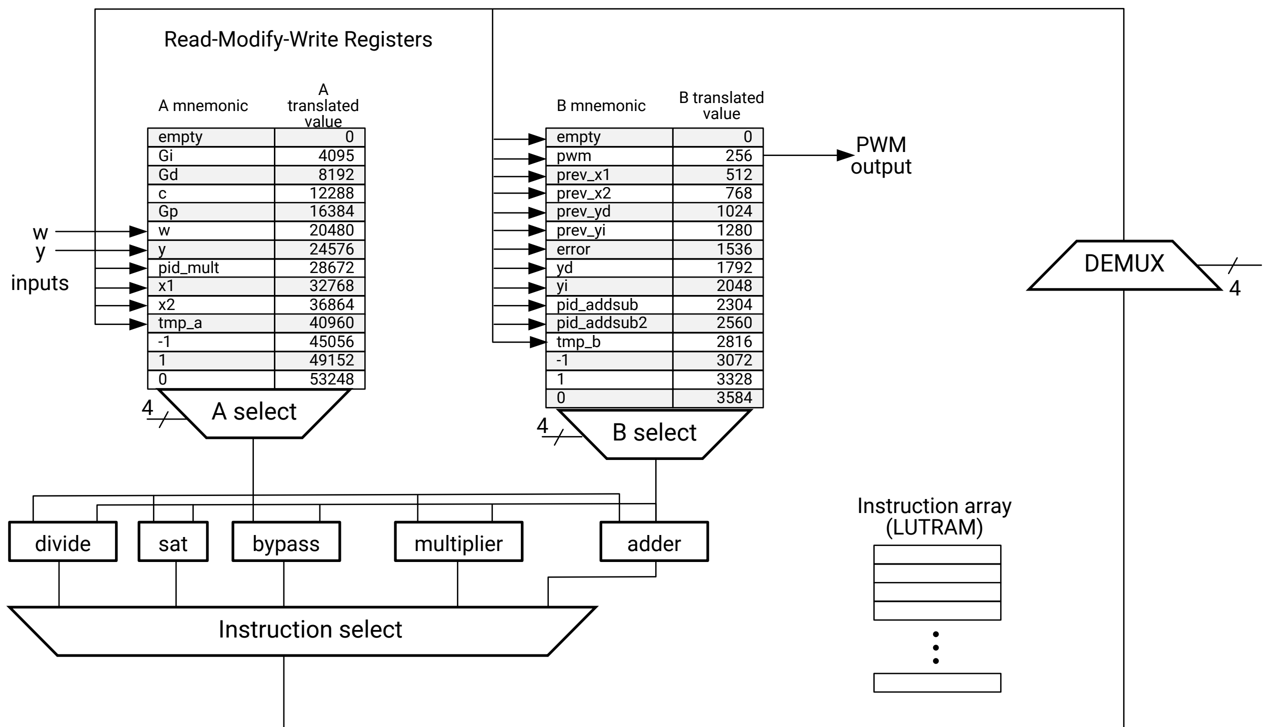
予測どおり、データパスがより広い SPFP の場合、レイテンシが長く、固定小数点での同じ実装よりも多くのリソースが使用されます。

アプローチ 2: カスタム Model Composer ブロックとしてインポートされた C++ Math Sequencer 関数を使用する

パッケージ化された IP または同等の HLS 合成可能コードに自動的に統合できるネイティブの Model Composer ブロック ライブラリを使用してアルゴリズムを表現するほかに、C/C++ ソース コードを用いて関数またはアルゴリズムを XMC 開発環境で記述する方が妥当な場合があります。ここでは、『Vivado HLS および System Generator for DSP を使用した浮動小数点 PID コントローラー デザイン』 (XAPP1163) での議論を拡張し、ハードウェア リソースを修正する必要があり、かつ、制御アルゴリズムを変更する必要がある場合を考察します。PID、PI、またはリード/ラグ コントローラーの使用有無にかかわらず、これらの機能はすべて一連の乗算、加算、減算、および飽和演算になります。

制御アルゴリズムのハードウェア デザインは単純化され、(中間データおよび命令格納用) メモリ、PID コントローラーのサンプルに必要な W、Y 入力などの入力、math 演算子 (乗算、加算、飽和)、および、たとえば DAC を駆動してサーボ モーターを制御するために使用可能な出力を備えるステート マシンが構築されます。数値演算は連続して順次実行でき、汎用プロセッサのように命令メモリを変更することによって修正できます。一連の math 演算である制御アルゴリズムの開始には、基本的に入力データ (W、Y は PID 制御ループに対する入力) の受信が使用されることがあります。これは、次の図に示す Math Sequencer (MS) ブロック図のベースとなっています。

図 9: Math Sequencer ブロック図



次の図に示すように、LUTRAM ベースの命令アレイには、16 ビット命令ワードを格納できます。

図 10: Math Sequencer 命令

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0 Instruction BIT#
A mux	B mux		+,*, sat, bypass
Data Select		Store	Instruction Select

X22296-020819

A オペランド (例: マルチプレクサー A) 用のビット 15 ～ 12 および B オペランド (例: マルチプレクサー B) 用のビット 11 ～ 8 は、ビット 3:0 (+、*、飽和) で指定される math 演算のレジスタ ベースの入力です。ビット 7 ～ 4 は演算結果が格納される場所を示し、これを元にユーザーはデータ ストレージに対する読み出し、変更、および書き込みを実行できます。クロック サイクルは増加しますが、必要なハードウェア量を削減する方法があります。たとえば、error = w - y となる必要がある場合は、tmp_b = y * –1、error = w + tmp_b を使用して 2 段階シーケンスを実行できます。

注記: 0x0 命令は math シーケンスを終了します。

場合によっては、1 組のオペランド データが同じレジスタ アレイに格納されることがあります。この問題を回避するには、データをあるアレイから別のアレイに移動するためのバイパス命令 (たとえば、A アレイ側から B アレイ側にデータを移動する、またはその逆) が必要です。

機能を拡張するために、ほかの演算子や除算、平方根、FFT などの高水準関数を追加することもできます。これらはすべて必要に応じて命令パイプラインに追加できます。このアプリケーション ノートでは、『Vivado HLS および System Generator for DSP を使用した浮動小数点 PID コントローラー デザイン』(XAPP1163) からの PID C++ コードをテンプレートとして利用しているため、飽和、バイパス、乗算、および加算の演算子のみが必要です。

Math Sequencer (MS) のコードは非常に単純で、図 9: Math Sequencer ブロック図 から C++ で簡単に理解できることがわかります。

```
#include "ms.h"

void ms(float w_in, float y_in, float &pwm) {

    // for register & instruction details see math_sequencer_rv2.xls (MS Excel Spreadsheet)

    // A mux  data
    static float a_mux[] = {0, Gi, Gd, c, Gp, 0, 0, 0, 0, 0, 0, minus1, plus1, zeroc};
    // B mux data
    static float b_mux[] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, minus1, plus1, zeroc};
#pragma HLS ARRAY_PARTITION variable=b_mux complete dim=1
    // constant definitions

    // load data from interface
    a_mux[5] = (float) w_in; // cast variable to float
    a_mux[6] = (float) y_in;

    // setup instructions
    unsigned short mnemonics[23] =
{0x6CC2,0x5B31,0x633,0x2652,0x1662,0x3442,0xB272,0x82A1,0x7C42,0x7A11,0x93B1,0xA5A1,0xA03,0x4642,0x
D8B5,0xA791,0x79A1,0xA23,0x8074,0x9084,0x7D5,0x8E5};
    const short num_instr = 22;

    ap_uint<16> instruction;
    ap_uint<4> instr_sel; // instruction mux controls
    ap_uint<4> store;
    ap_uint<4> dsel_b, dsel_a;

    float a_sel_data, b_sel_data; // variables for data management & results storage
    float op_results; // 32 bit results
    float fsat_o, fmul, fadd; // float data types needed to support saturate

#pragma HLS PIPELINE
        instruction_loop : for (short inst_loop = 0; inst_loop < num_instr; inst_loop++) {

            // split up instruction into tasks
            instruction = (ap_uint<16>) mnemonics[inst_loop]; // cast unsigned short to
ap_ufixed<16> data type

            // split out instructions into sub blocks
            instr_sel = instruction & 0x000F; // instruction select
            store = (instruction & 0x00F0) >> 4; // store results where?
            dsel_b = (instruction & 0x0F00) >> 8; // source B side mux from what register?
            dsel_a = (instruction & 0xF000) >> 12; // source A side mux from what register?

            // determine A select
            a_sel_data = a_mux[dsel_a];

            // determine B select
```

```
            b_sel_data = b_mux[dsel_b];

            switch(instr_sel) {
                case 0: break; // invalid instruction
                case 1: fadd = a_sel_data + b_sel_data; op_results = fadd; break; // a+b
                case 2: fmul = a_sel_data * b_sel_data; op_results = fmul; break; // a*b
                case 3: if (b_sel_data < min_limit)
                            fsat_o = min_limit;
                        else if (b_sel_data > max_limit)
                            fsat_o = max_limit;
                        else fsat_o = b_sel_data;
                        op_results = fsat_o; break;
                case 4: op_results = a_sel_data; break; // bypass a
                case 5: op_results = b_sel_data; break; // bypass b
                // case 6: op_results = a_sel_data / b_sel_data; break; // a/b
                break;
            } // end instr_sel

            switch(store) {
                            case 0: b_mux[8] = op_results; break; // yi
                            case 1: b_mux[7] = op_results; break; // yd
                            case 2: b_mux[1] = op_results; break; // pwm
                            case 3: b_mux[6] = op_results; break; // error
                            case 4: a_mux[7] = op_results; break; // pid_mult
                            case 5: a_mux[8] = op_results; break; // x1; a_mux
                            case 6: a_mux[9] = op_results; break; // x2; a_mux
                            case 7: b_mux[2] = op_results; break; // prev_x1
                            case 8: b_mux[3] = op_results; break; //prev_x2
                            case 9: b_mux[9] = op_results; break; // pid_addsub
                            case 10: b_mux[10] = op_results; break; // pid_addsub2
                            case 11: a_mux[10] = op_results; break; // tmp_a; a_mux
                            case 12: b_mux[11] = op_results; break; // tmp_b
                            case 13: b_mux[4] = op_results; break; // prev_yd
                            case 14: b_mux[5] = op_results; break; // prev_yi
                            case 15: break; // invalid
                        } // end store results


        } // end instruction_loop

    pwm = b_mux[1]; // PWM is a pass by reference variable (ie: top level interface); SDSoC
interface requires C99 data types

} // end math sequencer
```

16 進数の命令の作成を単純化するために 『Vivado HLS および System Generator for DSP を使用した浮動小数点 PID コントローラー デザイン』(XAPP1163) で記述された C++ コードを表すニーモニックを用いて命令テーブルを自動的に生成する Microsoft Excel スプレッドシートを使用しました。

図 11: Math Sequencer 命令シーケンスの生成に使用する Excel スプレッドシート

Math Sequencer

A operand	operatc	B operand	store resul	assembly instruction (hex)	XAPP 1163 C++ function	a	operator	b	store	Instr #
y	mult	-1	tmp_b	6CC2	compute error signal (error = w - y)	24576	2	3072	192	0
w	add	tmp_b	error	5B31		20480	1	2816	48	1
empty	sat	error	error	633	check for saturation condition	0	3	1536	48	2
Gd	mult	error	x1	2652	input signal of the derivative stage	8192	2	1536	80	3
Gi	mult	error	x2	1662	input signal of the integration stage (store in x2 reg)	4096	2	1536	96	4
c	mult	prev_yd	pid_mult	3442	calculate derivative	12288	2	1024	64	5
-1	mult	prev_x1	prev_x1	B272		45056	2	512	112	6
x1	add	prev_x1	pid_addsub2	82A1		32768	1	512	160	7
pid_mult	mult	-1	pid_mult	7C42		28672	2	3072	64	8
pid_mult	add	pid_addsub2	yd	7A11	end derivative	28672	1	2560	16	9
x2	add	prev_x2	tmp_a	93B1	calculate integration	36864	1	768	176	10
tmp_a	add	prev_yi	pid_addsub2	A5A1		40960	1	1280	160	11
empty	sat	pid_addsub2	yi	A03	end integration	0	3	2560	0	12
Gp	mult	error	pid_mult	4642	output PWM signal	16384	2	1536	64	13
0	bypb	yi	tmp_a	D8B5		53248	5	2048	176	14
tmp_a	add	yd	pid_addsub	A791		40960	1	1792	144	15
pid_mult	add	pid_addsub	pid_addsub2	79A1		28672	1	2304	160	16
empty	sat	pid_addsub2	pwm	A23		0	3	2560	32	17
x1	bypa	empty	prev_x1	8074	update internal PID states for next iteration	32768	4	0	112	18
x2	bypa	empty	prev_x2	9084		36864	4	0	128	19
empty	bypb	yd	prev_yd	7D5		0	5	1792	208	20
empty	bypb	yi	prev_yi	8E5		0	5	2048	224	21

MS C++ コードをインポートしてシミュレーションを実行した後、Simulink 倍精度モデルと MS にはほとんど違いがありません。

図 12: Math Sequencer 用にインポートされた C++ モデル

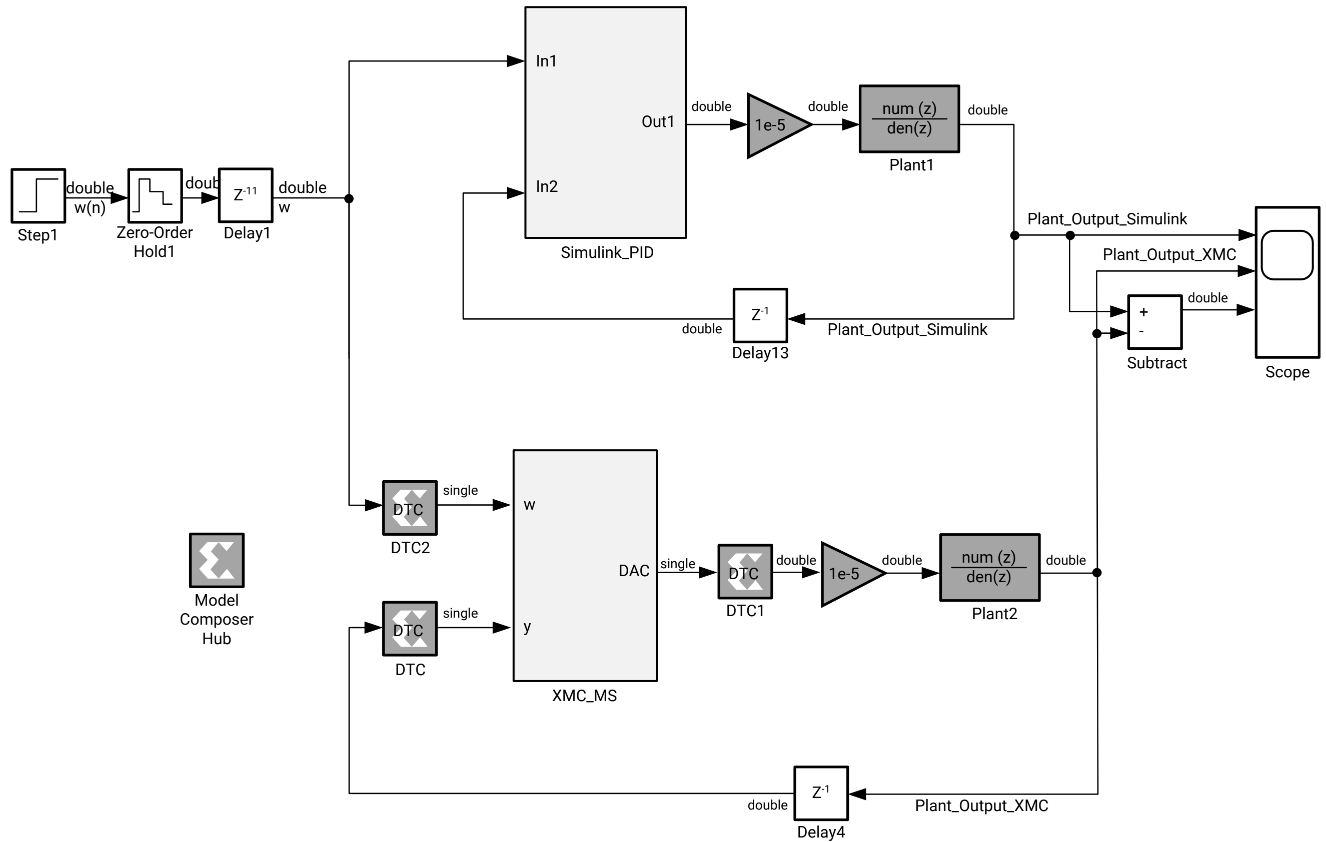
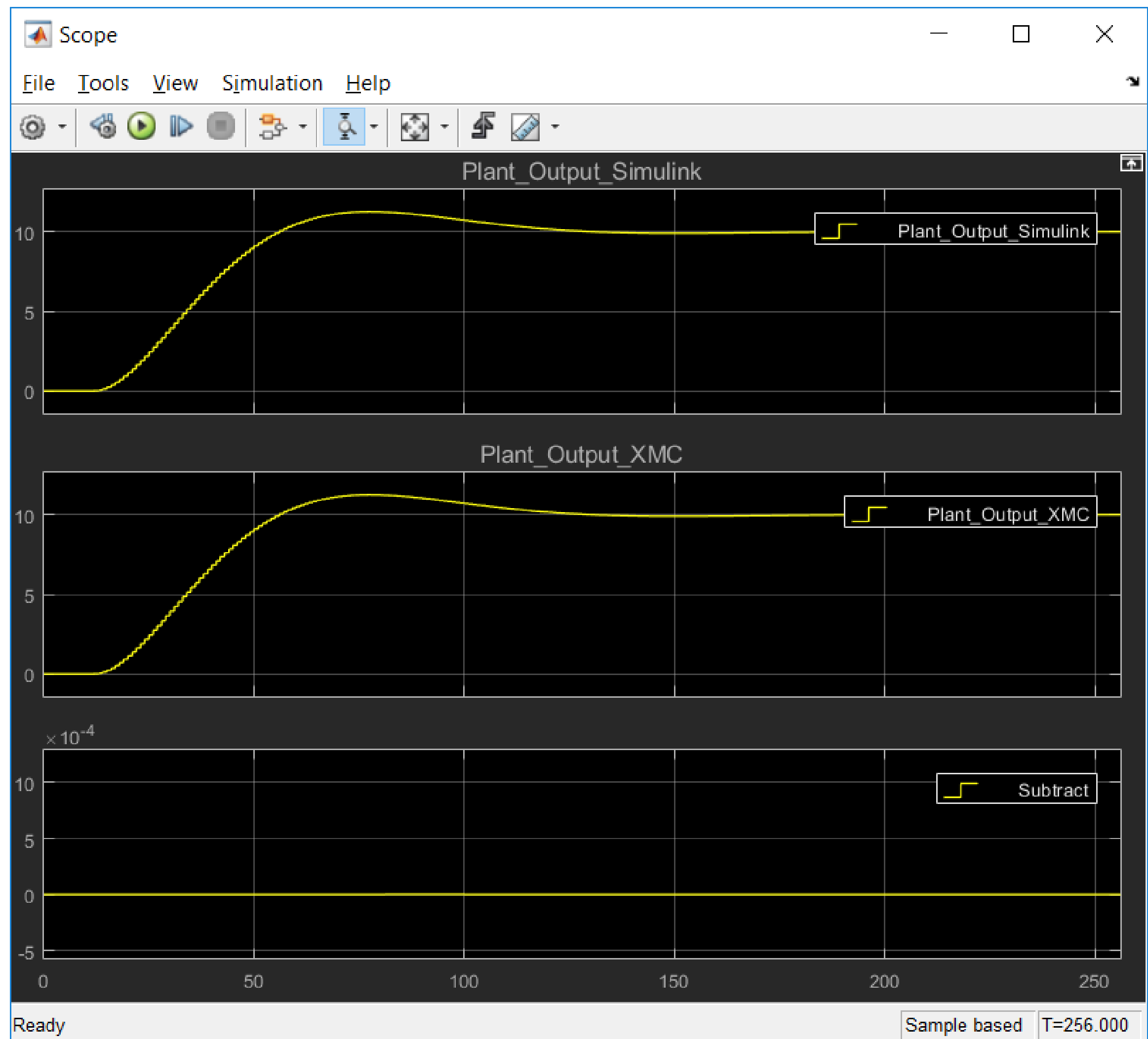


図 13: Simulink および Math Sequencer のシミュレーション結果の比較



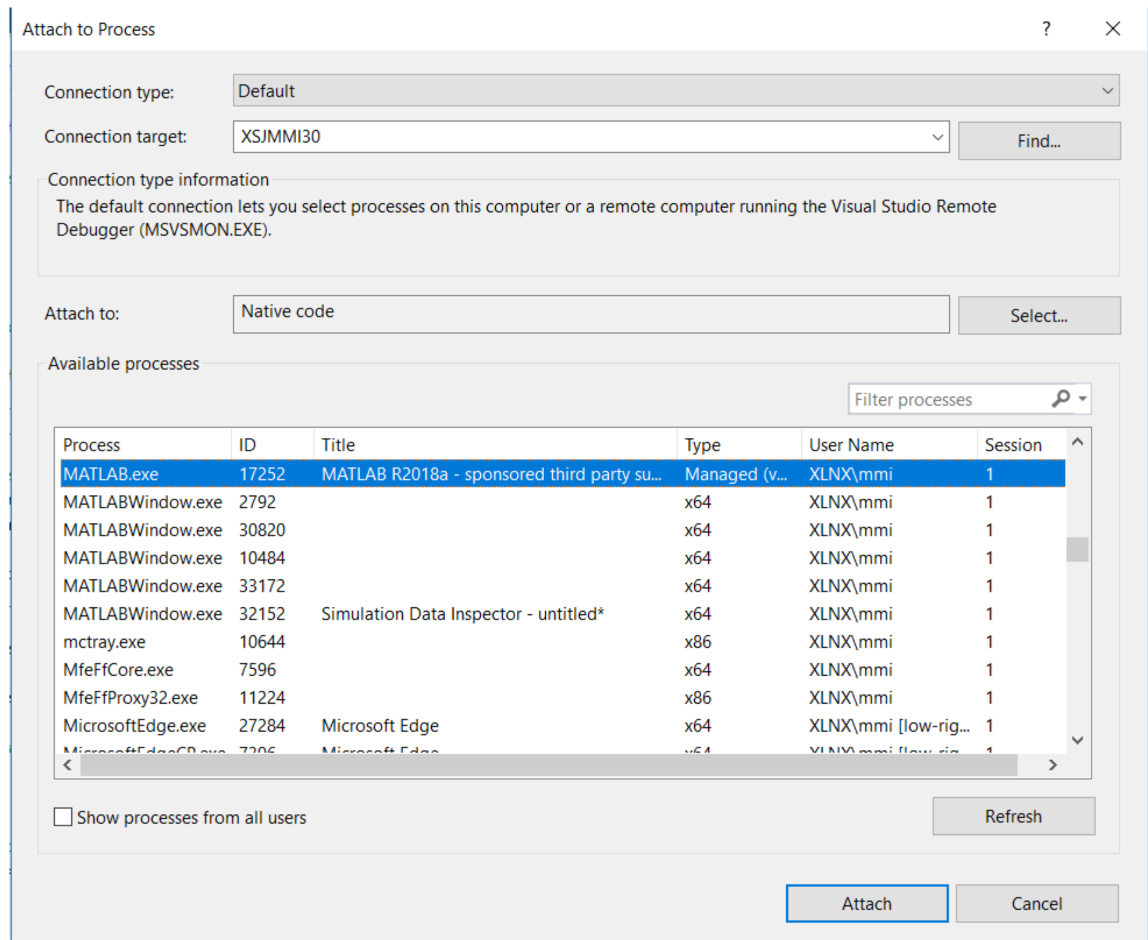
概念的には簡単ですが、エラーのない C/C++ を記述することは難しいため、Microsoft Visual C (MSVC) または GNU デバッガーを利用して C/C++ をデバッグする機能と組み合わせて、ビルド済みテストベンチおよび Simulink の検証環境を備えておくことを推奨します。この機能を用意し、MSVC を使用して `ms.cpp` デザインをシングル ステップ実行またはデバッグする手順は次のとおりです。

1. Simulink モデルを開き、MSVC でシミュレーションの準備ができていることを確認します。
2. Simulink コマンド ライン>> xmcImportFunctionSettings (build、debug、compiler、Visual Studio) から Simulink コンソールで Visual Studio コンパイラを使用してデバッグ DLL をビルドするように設定します (DLL はシミュレーションを実行すると自動的にビルドされる)。
3. Visual Studio を起動します。
4. Visual Studio のメニュー [Debug]→[Attach to Process] から実行中の MATLAB プロセスを接続します。

   [MATLAB.exe] を選択して [Attach] をクリックします。

注記: [Attach to] で [Native code] が設定されていることを確認します。

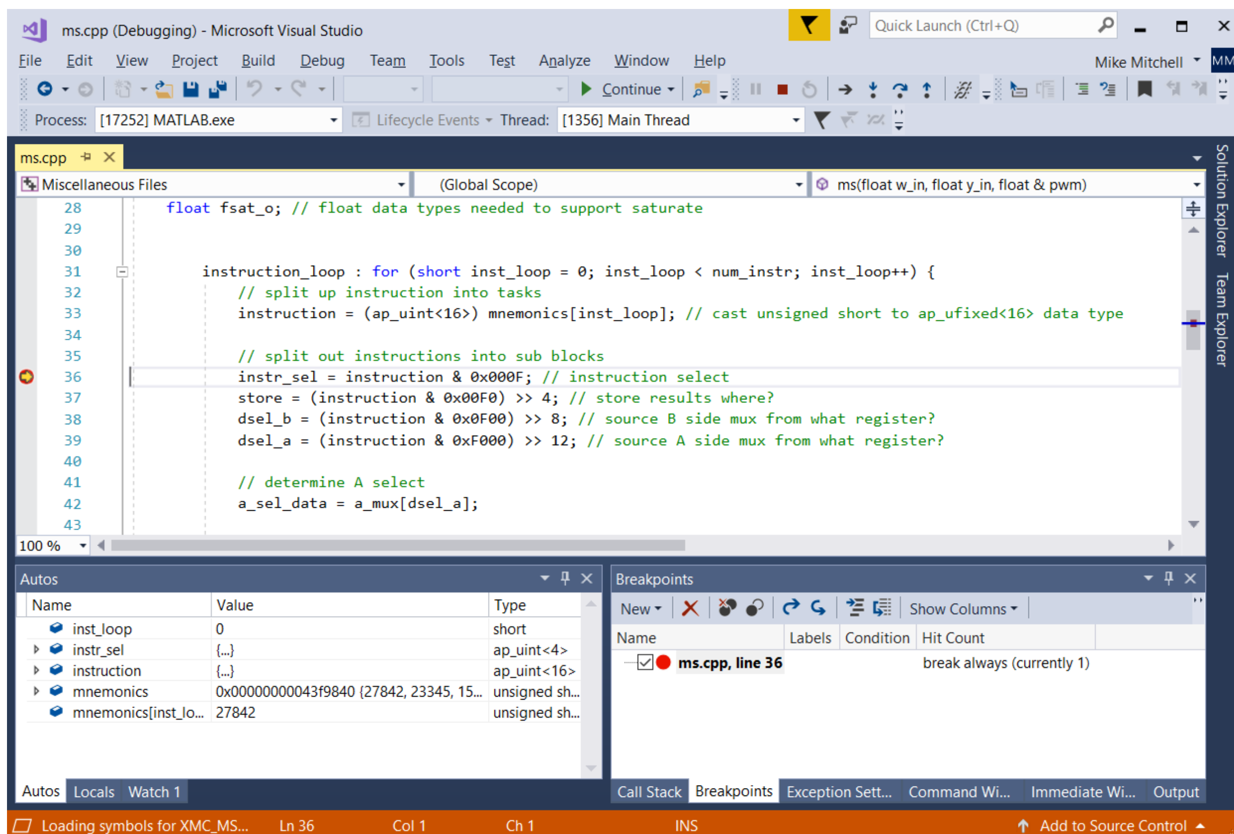
図 14: MATLAB プロセスの接続



5. (Visual Studio で開いたファイルから) C/C++ コードでユーザー ブレークポイントを設定します。
6. Simulink でシミュレーションを実行します。
7. MSVC メニューを使用して C/C++ コードをデバッグします。

次の図から、MSVC では 36 行目にブレークポイントが設定されていることがわかります。

図 15: MSVC のシングル ステップ実行



インポートした C/C++ コードでブレークポイントを設定してデバッグできれば、開発時に大きな利点となります。これで、Simulink を活用し、インポートした C/C++ モジュールに対してテストベンチの作成、検証、および結果の視覚化が可能になります。

最適化を実行せずに `ms.cpp` ソース コードを実行すると、次のような結果となります。

図 16: HLS ベースライン Math Sequencer の結果

Export Report for 'XMC_MS'

General Information

Report date: Mon Dec 17 12:07:00 -0600 2018
Project: XMC_MS_prj
Solution: solution1
Device target: xc7z020clg484-1
Implementation tool: Xilinx Vivado v.2018.3

Resource Usage

	Verilog
SLICE	355
LUT	921
FF	1349
DSP	3
BRAM	2
SRL	19

Final Timing

	Verilog
CP required	3.333
CP achieved post-synthesis	3.723
CP achieved post-implementation	3.228

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	626	638	639	627	639	640

2 つの HLS 命令を適用して命令ループをパイプライン化し、b_mux を分割すると (たとえば、上記の `ms.cpp` ソースコード内の #pragma で始まる行)、次のようにより最適化された HLS 結果が得られます。

図 17: Math Sequencer 実装の結果

Export Report for 'XMC_MS'

General Information

Report date:	Tue Dec 18 10:10:38 -0600 2018
Project:	XMC_MS_prj
Solution:	solution1
Device target:	xc7z020clg484-1
Implementation tool:	Xilinx Vivado v.2018.3

Resource Usage

	Verilog
SLICE	426
LUT	958
FF	1761
DSP	3
BRAM	0
SRL	17

Final Timing

	Verilog
CP required	3.333
CP achieved post-synthesis	4.042
CP achieved post-implementation	3.529

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	145	157	158	146	158	159

次の表は、XMC ブロックセットと Math Sequencer のアプローチを比較したものです。

表 2: Math Sequencer と XMC ネイティブ ブロックのリソース、レイテンシ、およびクロック周波数の比較

SPFP データ型	DSP	LUT	フリップフロップ	ブロック RAM	レイテンシ (クロック)	クロック (MHz)
XMC ネイティブ ブロック	6	981	1606	0	142	289.52
Math Sequencer	3	958	1761	0	158	283.37

MS には制御アルゴリズムや演算子を簡単に変更できる (たとえば、除算演算子または平方根関数を追加してマグニチュードを計算できる) といった潜在的な利点があるため、アルゴリズムを後で (ハードウェア実装を変更することなく) 変更し、レイテンシを犠牲にしてリソースを削減できるように命令シーケンス フィールドをアップグレードできるようにしておきます。

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 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。

リファレンス デザインの詳細

次の表に、リファレンス デザインで使用される手順を示します。

表 3: リファレンス デザインの詳細

パラメーター	説明
全般	
開発者	ザイリンクス
ターゲット デバイス	XC7Z020CLG484-1
ソース コードの提供	あり
ソース コードの形式 (提供される場合)	Simulink (slx) および C++ デザイン ファイル
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、サードパーティ、Vivado ツールからデザインへのコード/IP の使用(使用した場合はその詳細)	あり、『Vivado HLS および System Generator for DSP を使用した浮動小数点 PID コントローラー デザイン』 (XAPP1163)
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	あり
論理シミュレーションおよびタイミング シミュレーション用テストベンチの提供	あり、HLS 自動デザイン フローの一部
テストベンチの形式	Simulink (xls) または C++ (XMC または HLS ツール フローに基づく)
使用したシミュレータ/バージョン	Vivado シミュレータ 2018.3
SPICE/IBIS シミュレーションの実施	なし
インプリメンテーション	
使用した合成ツール/バージョン	Vivado Design Suite 2018.3
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 2018.3
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	なし
使用したプラットフォーム	N/A

セットアップ

リファレンス デザインには、次のファイルが含まれています。

- `create_library.m`: MATLAB ファイル。C++ でシミュレーション可能な XMC モジュールを作成します。
- `ClosedLoopPID_XMC_SPFP.slx`: ネイティブ XMC ブロックを使用する単精度浮動小数点 PID コントローラー。
- `ClosedLoopPID_XMC.slx`: ネイティブ XMC ブロックを使用する固定小数点 PID コントローラー。
- `ClosedLoopPID_MS_XMC.slx`: Math Sequencer デザイン。
- `ms.cpp`, `ms.h`: Math Sequencer C++ ソース ファイル。

- `math_sequencer_rv2.slxs`: Microsoft Excel ファイル。Math Sequencer 命令を作成します。

リファレンス デザインの実行

1. MATLAB コマンド ラインから `create_library.m` ファイルを実行し、`ms.cpp` および `ms.h` C++ ソース ファイルから C++ でシミュレーション可能な XMC モジュールを作成します。
2. Simulink を使用し、ネイティブ XMC ブロックを使用する `ClosedLoopPID_XMC_SPFP.slx` 単精度浮動小数点 PID コントローラーをシミュレーションします。
3. Simulink を使用し、ネイティブ XMC ブロックを使用する `ClosedLoopPID_XMC.slx` 固定小数点 PID コントローラーをシミュレーションします。
4. Simulink を使用し、`ClosedLoopPID_MS_XMC.slx` Math Sequencer デザインをシミュレーションします。
5. Simulink 環境で、Model Composer Hub を使用して手順 2 から手順 4 で使用したモデルに対応する HLS プロジェクト デザインを生成します。
6. HLS フローを使用し、手順 2 から手順 4 までのリソース、タイミング、および RTL 検証を決定します。

まとめ

このアプリケーション ノートでは、ネイティブ XMC ブロックセットを使用して 『Vivado HLS および System Generator for DSP を使用した浮動小数点 PID コントローラー デザイン』 (XAPP1163) で説明した PID コントローラーのデザインを再実装し、固定小数点および浮動小数点の両方のアプローチの違いを比較しています。このアプローチには、C++ ベースの柔軟な Math Sequencer が追加されています。MS は、アルゴリズムの柔軟性が要求され、レイテンシを犠牲にしたリソースの削減 (例: シリアル実装とパラレル実装) が求められるアプリケーションに使用します。

次に、プロセス全体の着目点をまとめます。

- XMC を使用することで、ユーザーは Simulink 固有のさまざまな機能とツールボックスを活用でき、テストベンチの開発、結果の視覚化、およびデバッグの簡素化が可能になります。
- このデザインでは、実装方法としてネイティブの XMC ブロックセットを使用するかまたはカスタムで作成してインポートされた C/C++ を使用するかを柔軟に選択できます。
- C++ ベースのモデルでは、シミュレーションおよび開発時間が短縮されます。
- このデザインでは、インポートされた C/C++ モジュールを Microsoft Visual C または GNU デバッガーを使用してネイティブでデバッグします。
- テストベンチとデザインは、実行可能な C++ デザイン、Vivado IP カタログ、または System Generator IP として作成、評価、およびエクスポートできます。
- エクスポート プロセスで作成された HLS プロジェクトを使用し、デザイン パフォーマンスをさらに最適化することもできます。
- ユーザーは、HLS ベースの C コードで各変数を個別に量子化できます。math 演算後にビットの増加や切り捨てが必要になる可能性がある固定小数点データ型の場合、XMC では、HLS 用に C++ で各変数のデータ サイズを手動で定義するのではなく、math 演算後にデータ型を自動で伝搬できます。

改訂履歴

次の表に、この文書の改訂履歴を示します。

セクション	内容
2018 年 3 月 14 日 バージョン 1.0	
初版	N/A

お読みください: 法的通知

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

自動車用のアプリケーションの免責条項

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

Copyright

© Copyright 2019 Xilinx, Inc. Xilinx、Xilinx のロゴ、Alveo、Artix、ISE、Kintex、Spartan、Versal、Virtex、Vivado、Zynq、およびこの文書に含まれるその他の指定されたブランドは、米国およびその他各国のザイリンクス社の商標です。 MATLAB および Simulink は、MathWorks, Inc. の登録商標です。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。