XILINX®

PID Controller Design with Model Composer

XAPP1341 (v1.0) March 14, 2019

Summary

This application note extends the concepts discussed in *Floating-Point PID Controller Design with Vivado HLS and System Generator for DSP* (XAPP1163) and demonstrates how you can implement the Proportional-Integral-Derivative (PID) control algorithm at a higher level of abstraction within MathWorks Simulink® using the Xilinx Model Composer (XMC) design tool. Model Composer is designed as a plug-in to Simulink for design, simulation, and implementation of production-quality algorithms on Xilinx devices. Transformation from algorithmic specification to production-quality implementation is enabled through automatic optimizations and automatic code generation that extends the Vivado® High-Level Synthesis (HLS) tool. Additionally, automatic test bench generation helps validate functional equivalence between the executable specification in Simulink and the synthesized RTL.

This application note describes two different ways to implement the PID algorithm with Model Composer:

- Native Xilinx-optimized blocks from the Model Composer block libraries within Simulink.
- A C-based, firmware customizable, Math Sequencer function that can be imported as a custom Model Composer block within Simulink.

Download the reference design files for this application note from the Xilinx website. For detailed information about the design files, see Reference Design.

Introduction

MATLAB® and Simulink product families from MathWorks provide a comprehensive design environment to model, analyze, and tune linear and non-linear dynamic systems. Model Composer fits into the Simulink environment and offers controls engineers a path for moving from algorithm design to deployment on Xilinx devices.

XMC offers the following advantages for accelerating the deployment of controls algorithm:

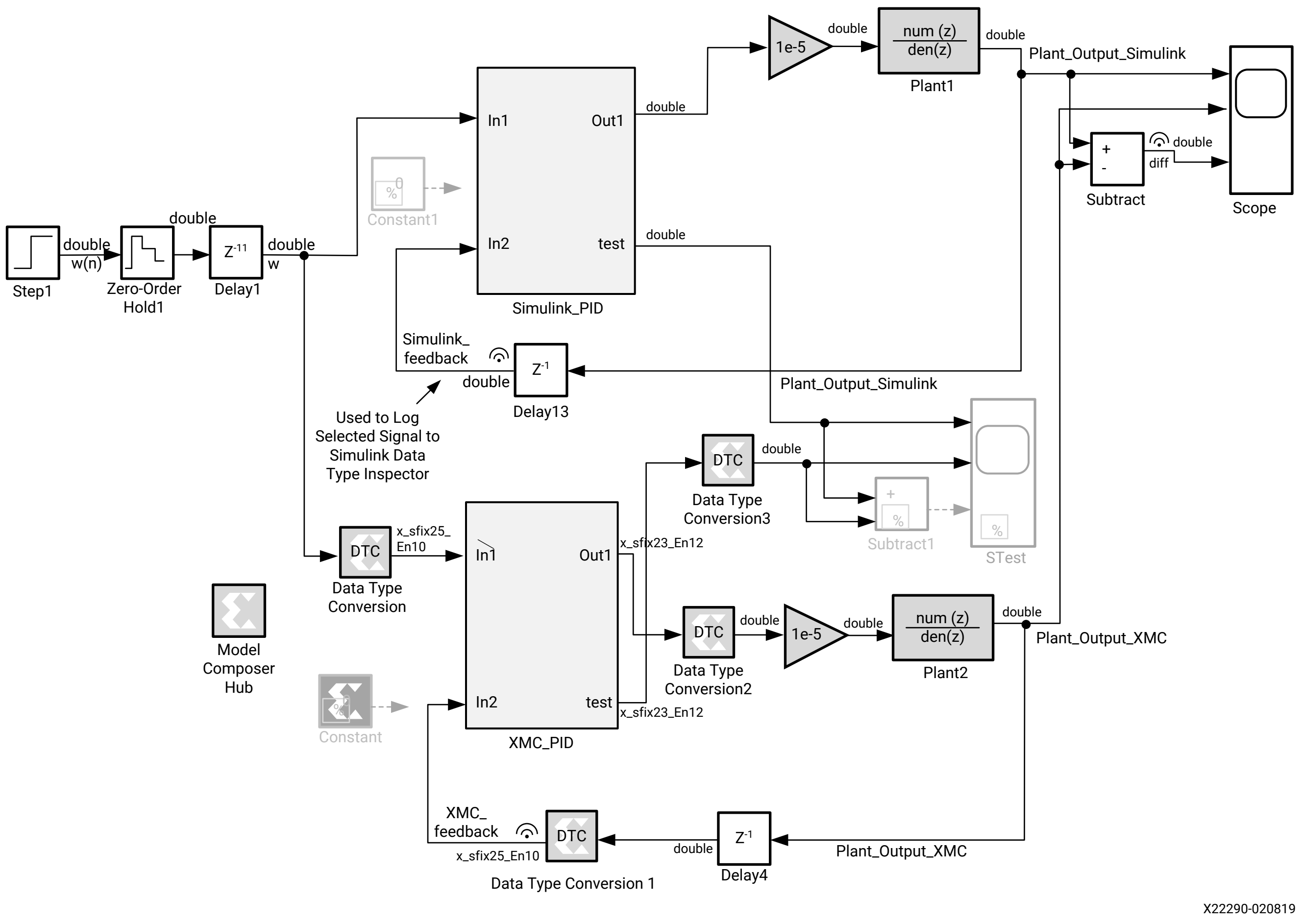
- Access to Xilinx-optimized math and linear algebra libraries for designing the implementation model at a high level of abstraction.
- Simplification of test bench development via integration with Simulink add-on toolboxes or MATLAB source code.
- Ease of verification by taking advantage of the many Simulink visualization and debug methodologies.
- Simulation performance advantages using bit-accurate C++ models.
- Flexible implementation approaches using either native XMC blocks or imported C/C++.

- In situ debug of imported C/C++ models using Microsoft Visual C or GNU Debugger during Simulink simulations.
- Ability to evaluate and export the design and the test bench using C++, Vivado IP catalog, or System Generator.

Approach 1: Using Native Xilinx Model Composer Block Libraries

The following figure contrasts the Simulink and XMC block diagrams for a standard PID controller as part of a closed loop system model that includes both the plant and the error feedback.

Figure 1: **Simulink PID Block within a Closed-Loop Control System**



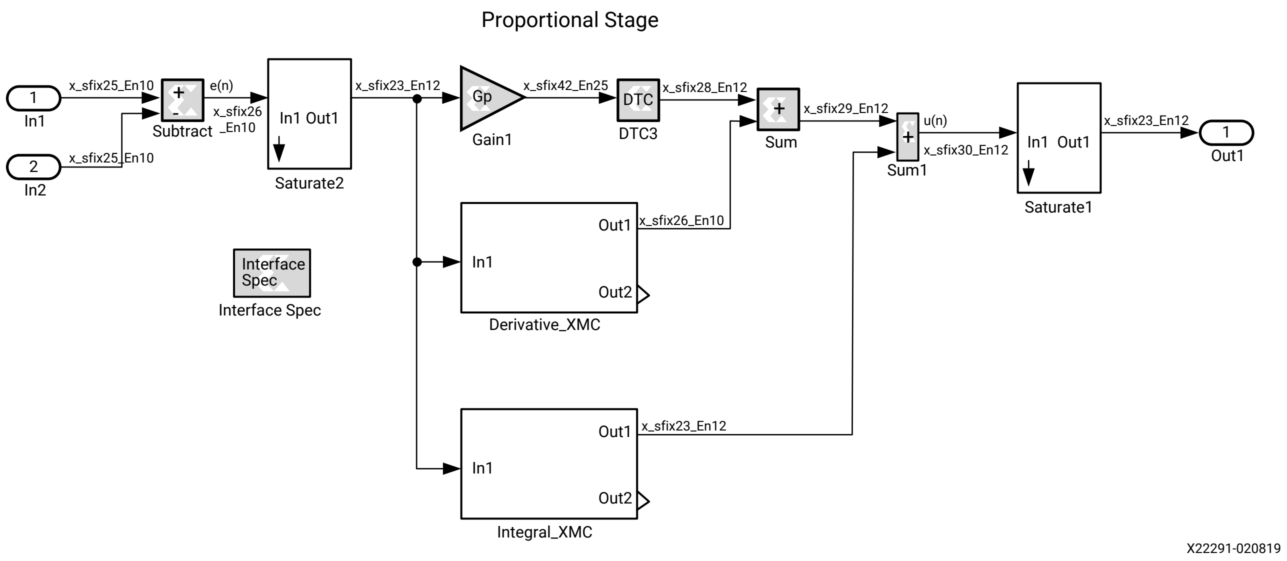
Moving from the Simulink golden reference model for the PID controller to a floating-point implementation model (`ClosedLoopPID_XMC_spfp.slx`) in Model Composer entails expressing the math or replacing the native Simulink blocks with equivalent Xilinx-optimized bit-accurate blocks from the Model Composer block libraries within Simulink—this enables you to work at the same level of abstraction as Simulink.

The ability to have both the golden reference model and the Model Composer hardware implementation model within the same environment significantly simplifies the process of verifying the functional equivalence between them. This design drives both subsystems using the same input stimulus and leverages the debug and visualization capabilities within Simulink. This allows you to easily compare the results and make informed design tradeoffs for the implementation.

From Simulation to Packaged IP or Optimized C++ Code for Vivado HLS

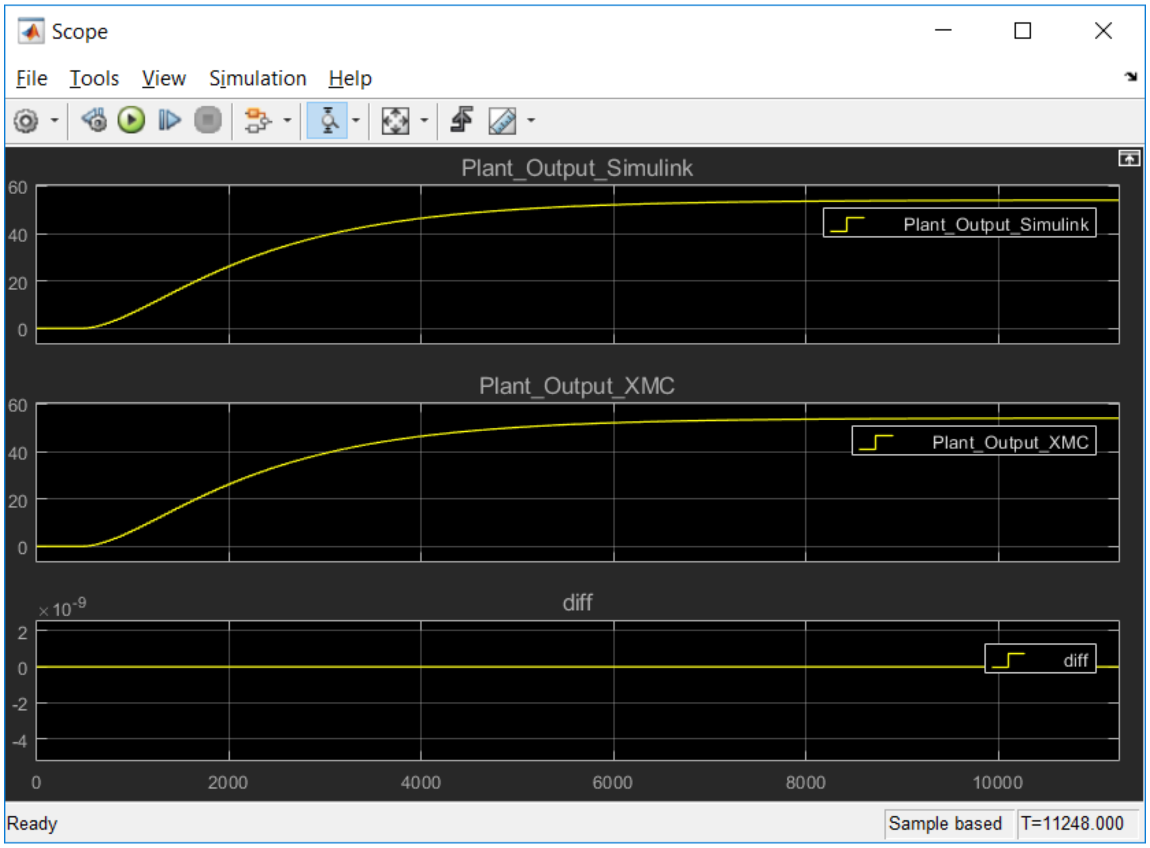
The following figure shows the fixed-point version of the XMC implementation model (`ClosedLoopPID_XMC.slx`).

Figure 2: **XMC_PID Fixed-Point Design**



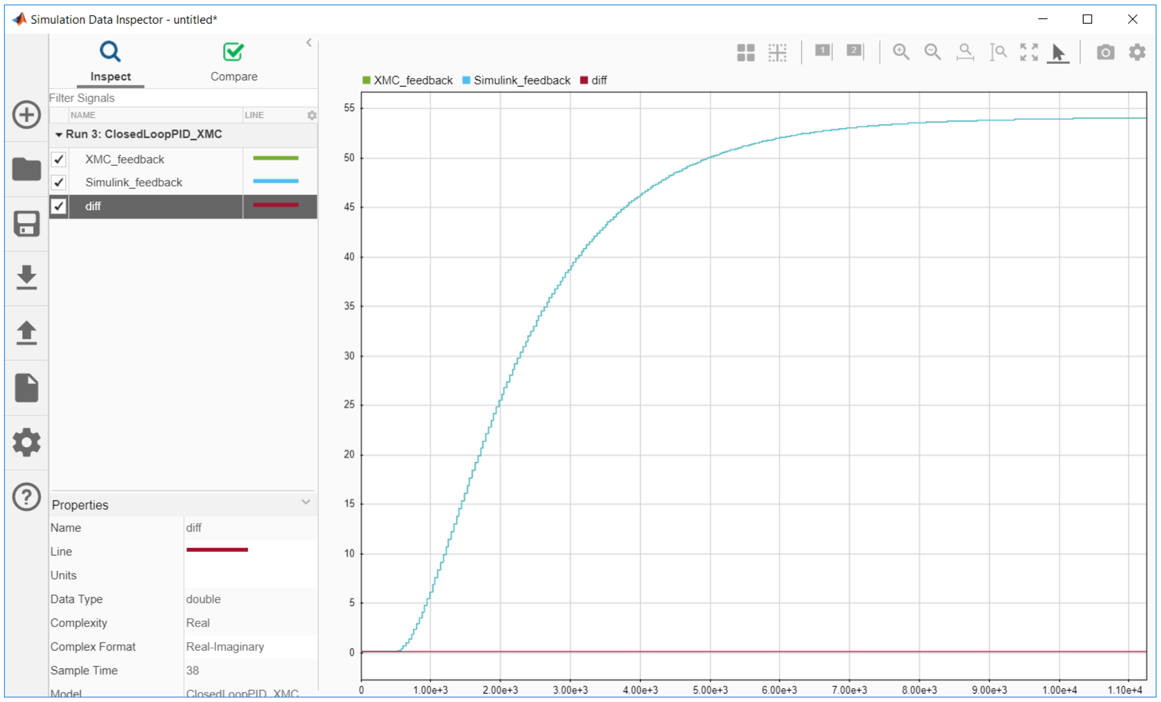
The Simulink scopes, signal logging, and simulation data inspector enable you to quickly and easily compare and contrast the simulation results for the Simulink double precision floating point to the XMC implementation. The following figure shows the feedback datapaths captured using a traditional Simulink scope. The difference ("diff") between the two signals is also plotted to demonstrate the error between the double precision Simulink and fixed-point XMC model. For this PID implementation with 27 bits of dynamic range, there is no noticeable difference between fixed- and floating-point.

Figure 3: **Simulink Scope Simulation Results**



The same data can also be captured and analyzed using the simulation data inspector.

Figure 4: **Simulation Data Inspector**


Simulation Data Inspector - untitled*
Inspect
Compare
Filter Signals
NAME
LINE
XMC_feedback Simulink_feedback diff
0 1.00e+3 2.00e+3 3.00e+3 4.00e+3 5.00e+3 6.00e+3 7.00e+3 8.00e+3 9.00e+3 1.00e+4 1.10e+4

Run 3: ClosedLoopPID_XMC

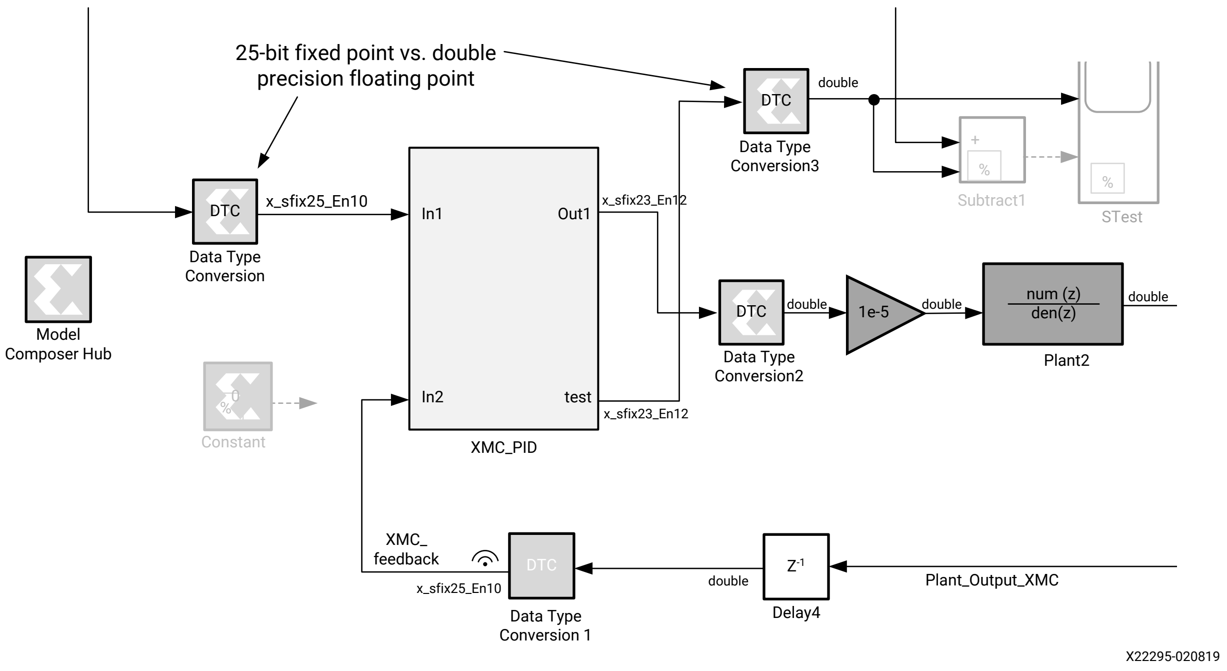
NAME	LINE
✓ XMC_feedback	🟩
✓ Simulink_feedback	🟦
✓ diff	🟥

Properties

Name	diff
Line	🟥
Units	
Data Type	double
Complexity	Real
Complex Format	Real-Imaginary
Sample Time	38
Model	ClosedLoopPID_XMC

An XMC Data Type Conversion (DTC) operator is used to change the quantization levels between floating-point or fixed-point data types as needed to determine if the modified precision leads to a system response with the desired user characteristics. For example, as shown in the following figure, both a fixed-point and floating-point DTC are used in the same simulation.

Figure 5: **Using the XMC DTC**

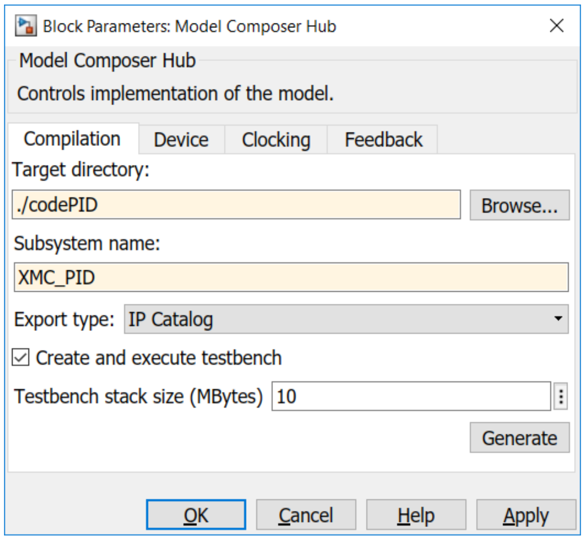


A user only needs to specify the data type in the DTC in order to set the precision.

Both an XMC fixed-point (`ClosedLoopPID_XMC.slx`) and single-precision floating-point (SPFP) PID model (`ClosedLoopPID_XMC_spfp.slx`) are included as part of the reference design to demonstrate and contrast resources, latency, and achievable clock frequency for the same design with different data types.

The automatically generated Vivado IP catalog option using the create and execute test bench option creates the HLS project that includes pass/fail results.

Figure 6: **Creating an HLS Project that Includes a Pass/Fail Test Bench Using the Simulink Source Stimulus**


Block Parameters: Model Composer Hub

Model Composer Hub

Controls implementation of the model.

Target directory: ./codePID

Subsystem name: XMC_PID

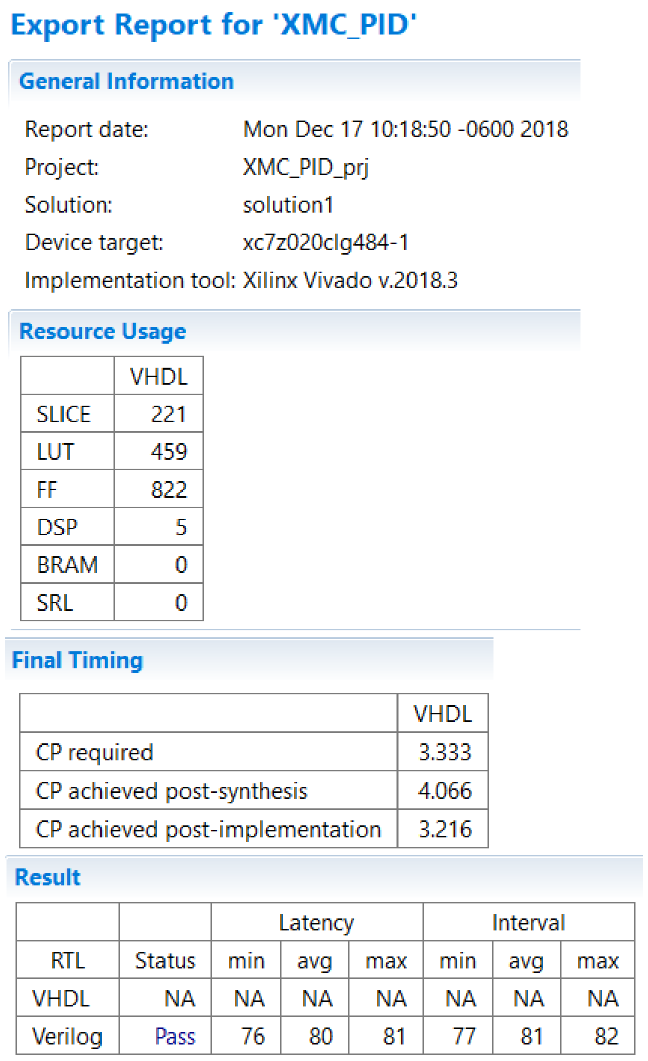
Export type: IP Catalog

☑ Create and execute testbench

Testbench stack size (MBytes) 10

The resulting auto-created fixed-point and SPFP HLS XMC PID projects can be used to further evaluate or optimize the resulting C++ based designs.

Figure 7: **PID Fixed-Point HLS Implementation Results**



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

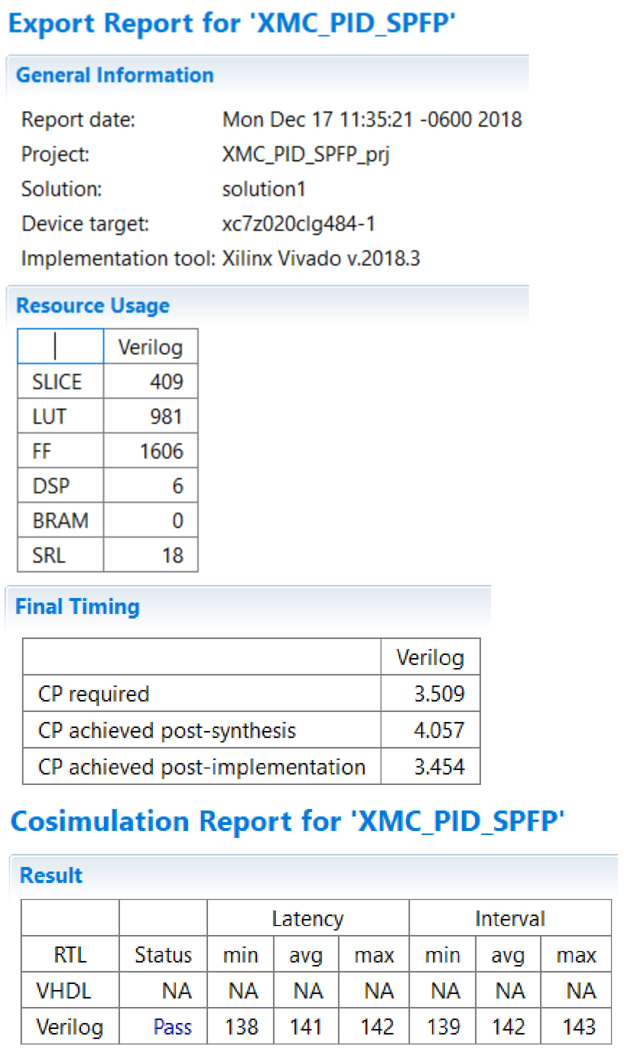
Final Timing

	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

Figure 8: **PID Single-Precision Floating-Point HLS Implementation Results**



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

Table 1: **PID Fixed-Point vs. Float-Point Comparison**

XMC Data Type	DSP	LUTs	Flip-Flops	Block RAM	Latency (Clocks)	Clock (MHz)
Fixed Point	5	459	822	0	81	311
SPFP	3	958	1761	0	142	290

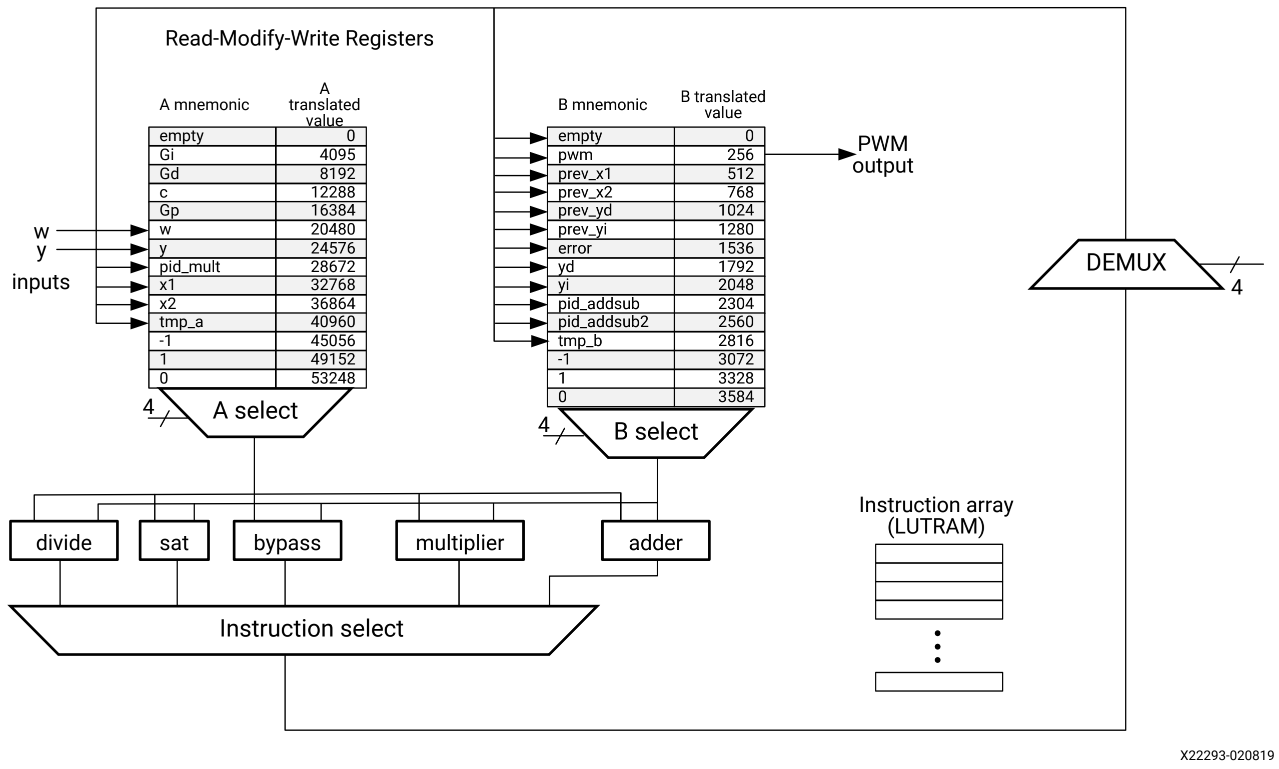
As expected, the wider SPFP datapaths are slower, have higher latency, and use more resources than the same implementation in fixed point.

Approach 2: Using a C++ Math Sequencer Function Imported as a Custom Model Composer Block

In addition to expressing the algorithm using native Model Composer block libraries, which can be automatically synthesized into a packaged IP or equivalent HLS synthesizable code, sometimes the user might find it more natural to start with C/C++ source code to describe a function or algorithm in the XMC development environment. To extend the discussion started in *Floating-Point PID Controller Design with Vivado HLS and System Generator for DSP* (XAPP1163), what if the user wanted to fix the hardware resources but change the control algorithm? Whether you have a PID, PI, or lead-lag controller, these functions all boil down to a series of multiply, add, subtract, and saturate operations.

The control algorithm hardware design could be simplified, and a state machine built that has memory (for intermediate data and instruction storage), inputs (e.g., the W, Y input needed for the PID controller example), math operators (mult, add, saturate) and an output that could, for example, be used to drive a DAC to control a servo motor. The arithmetic operations could be operated serially over time and modified by changing the instruction memory much like a generic processor. The arrival of input data (W, Y are inputs for the PID control loop) could be used to kick off a control algorithm that is essentially a sequence of math operations. This is the basis for the Math Sequencer (MS) block diagram in the following figure.

Figure 9: **Math Sequencer Block Diagram**

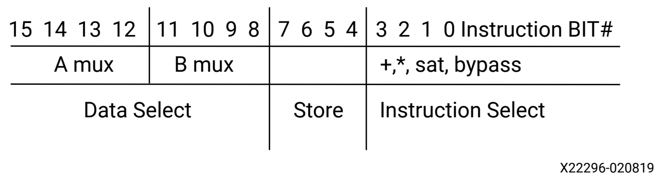


A mnemonic	A translated value
empty	0
Gi	4095
Gd	8192
c	12288
Gp	16384
w	20480
y	24576
pid_mult	28672
x1	32768
x2	36864
tmp_a	40960
-1	45056
1	49152
0	53248

B mnemonic	B translated value
empty	0
pwm	256
prev_x1	512
prev_x2	768
prev_yd	1024
prev_yi	1280
error	1536
yd	1792
yi	2048
pid_addsub	2304
pid_addsub2	2560
tmp_b	2816
-1	3072
1	3328
0	3584

A LUTRAM-based instruction array provides storage for the 16-bit instruction word as shown in the following figure.

Figure 10: **Math Sequencer Instruction**



Bits 15–12 for the A operand (e.g., multiplexer A) and bits 11–8 for the B operand (e.g., multiplexer B) are register-based inputs for the math operation detailed in bits 3:0 (i.e., +, *, saturate, bypass). Bits 7–4 indicate where the operation results are stored, giving you read, modify, and write capabilities for your data storage. There are ways to reduce the amount of hardware needed at the expense of clock cycles. For example, if you need error = w – y, you could perform a two-step sequence using tmp_b = y * –1; error = w + tmp_b.

Note: A 0x0 instruction ends the math sequence.

Sometimes a pair of operand data ends up in the same register array. To work around this problem, you need a bypass instruction to move data from one array to another (for example, to move data from the A array side to the B array side, or vice versa).

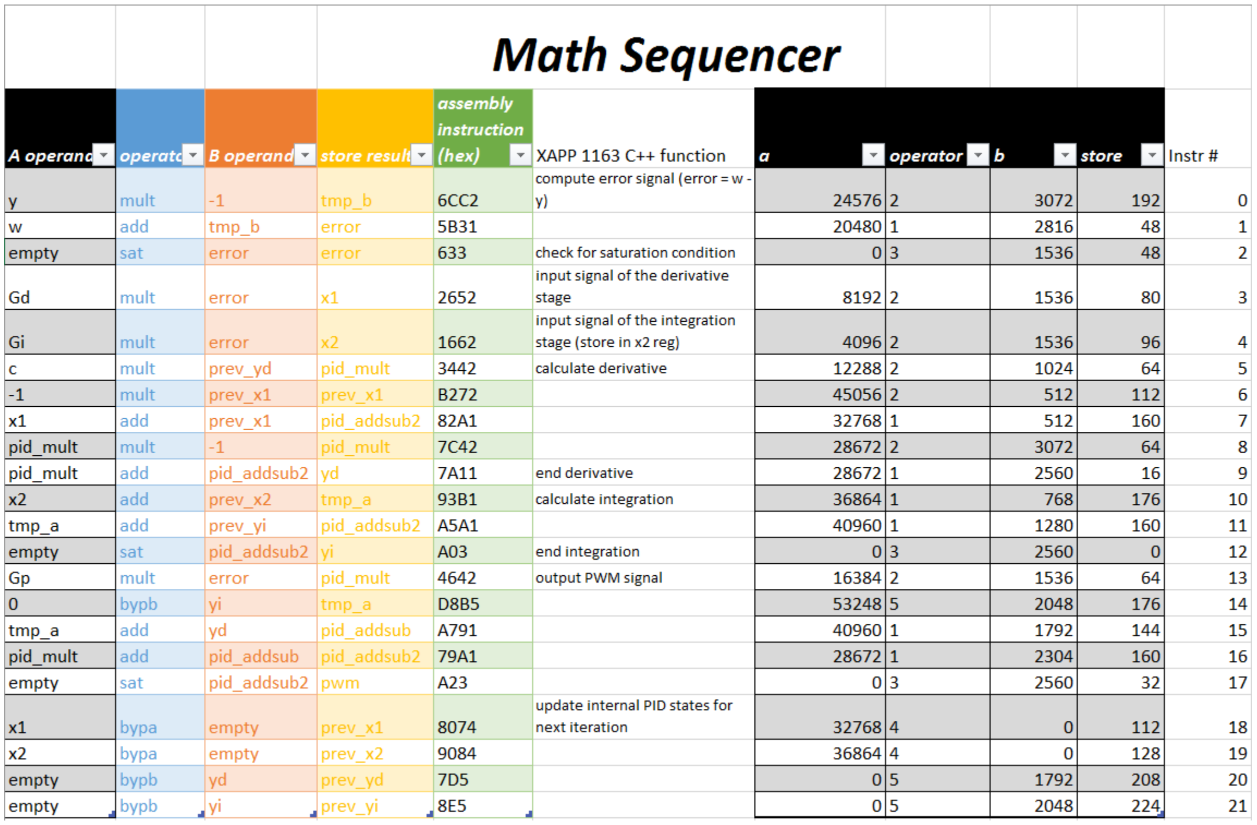
To expand the capabilities, you could also add additional operators or higher level functions like a divide, square root, and FFT, which can all be added to the instruction pipeline as needed. For this exercise, we are using the PID C++ code from *Floating-Point PID Controller Design with Vivado HLS and System Generator for DSP* (XAPP1163) as our template, so we only need saturate, bypass, multiplier, and adder operators.

The code for the MS is very simple and easy to understand in C++ in reference to Figure 9: Math Sequencer Block Diagram.

```
#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
```

A Microsoft Excel spreadsheet was used to automatically generate the instruction table using mnemonics to represent the C++ code written in *Floating-Point PID Controller Design with Vivado HLS and System Generator for DSP* (XAPP1163) to simplify hexadecimal instruction creation.

Figure 11: **Excel Spreadsheet Used to Generate Math Sequencer Instruction Sequence**



Math Sequencer

A operan	operat	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

After importing the MS C++ code and running the simulation, almost no difference is found between the Simulink double precision model and the MS.

Figure 12: **Imported C++ Model for the Math Sequencer**

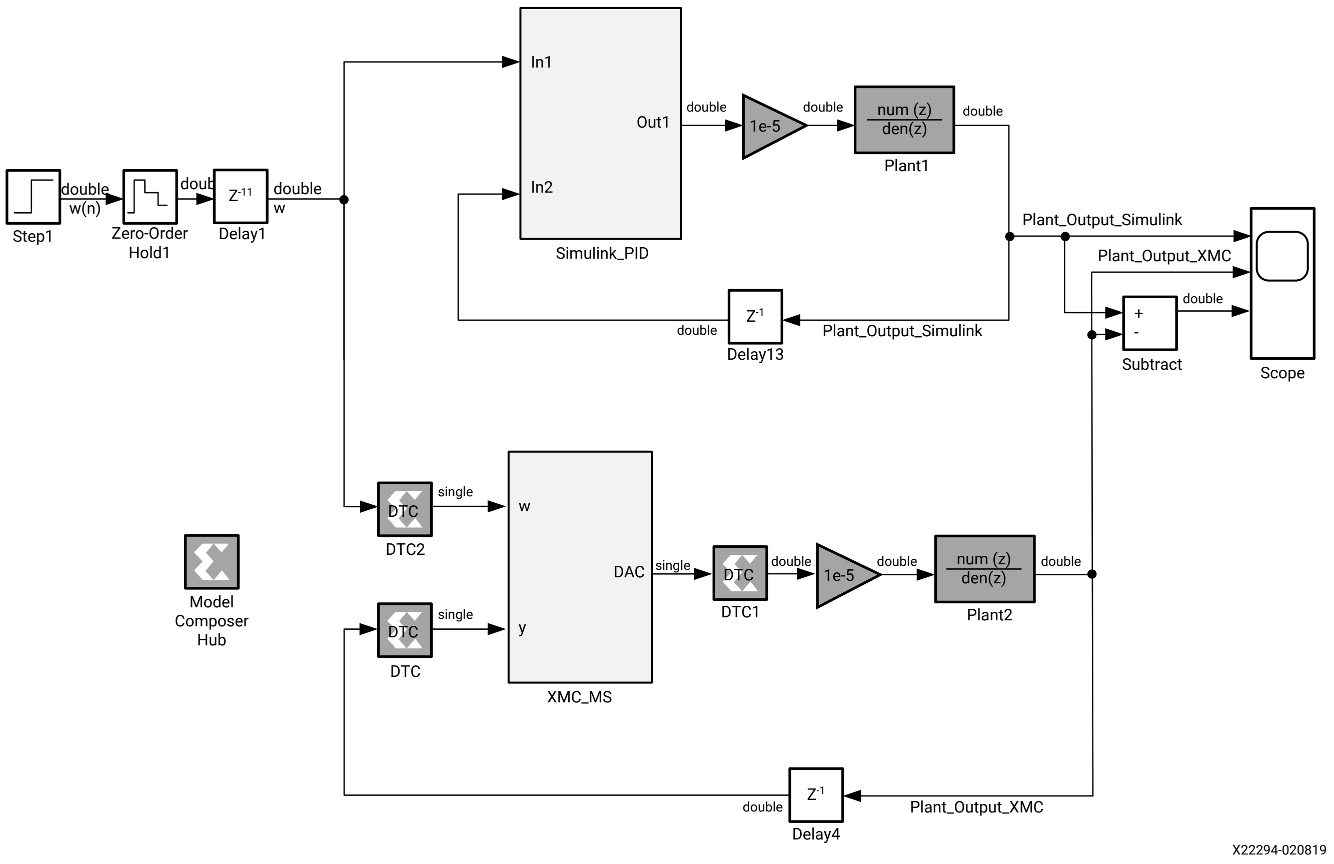
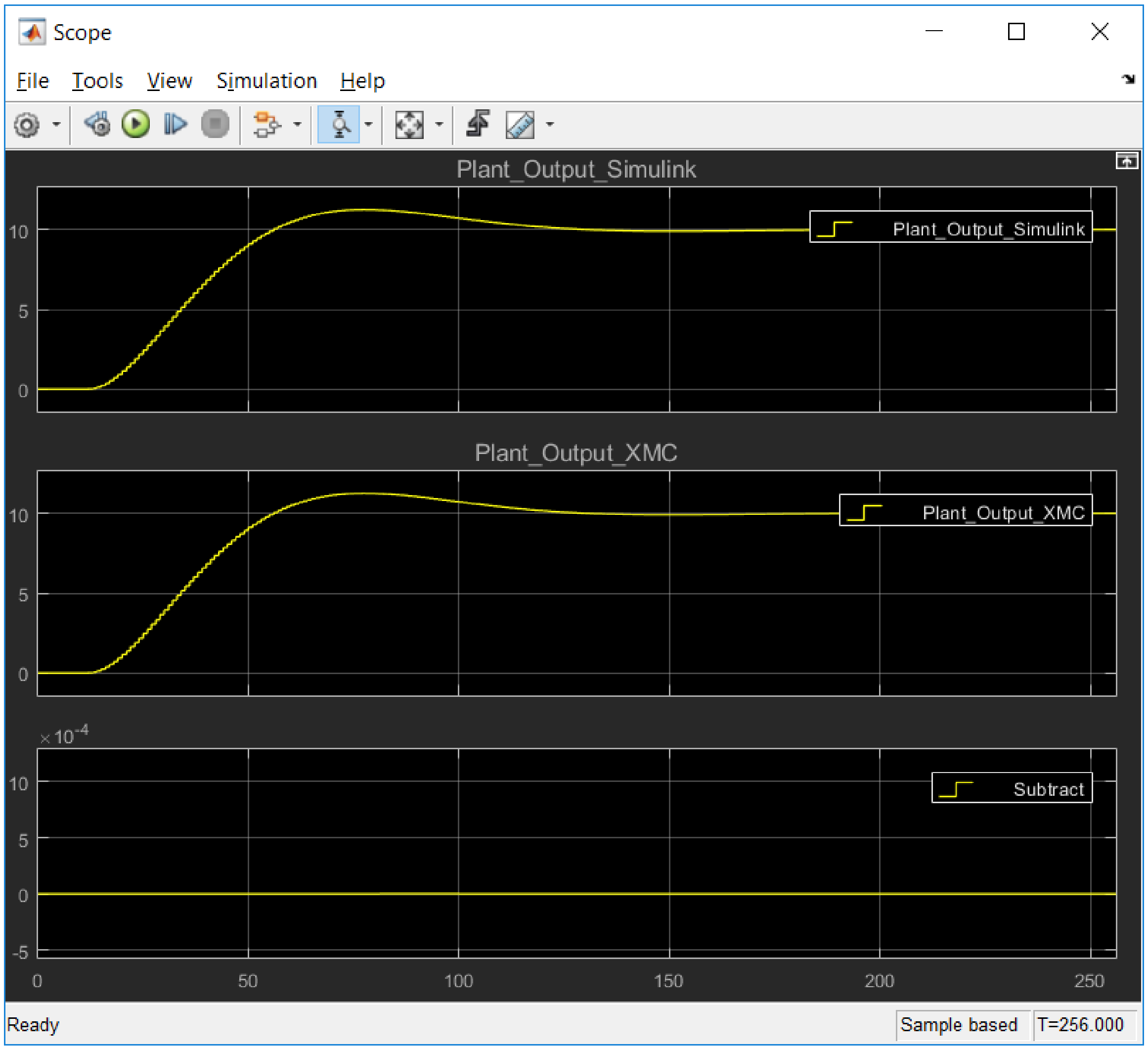


Figure 13: **Simulink vs. Math Sequencer Simulation Results**



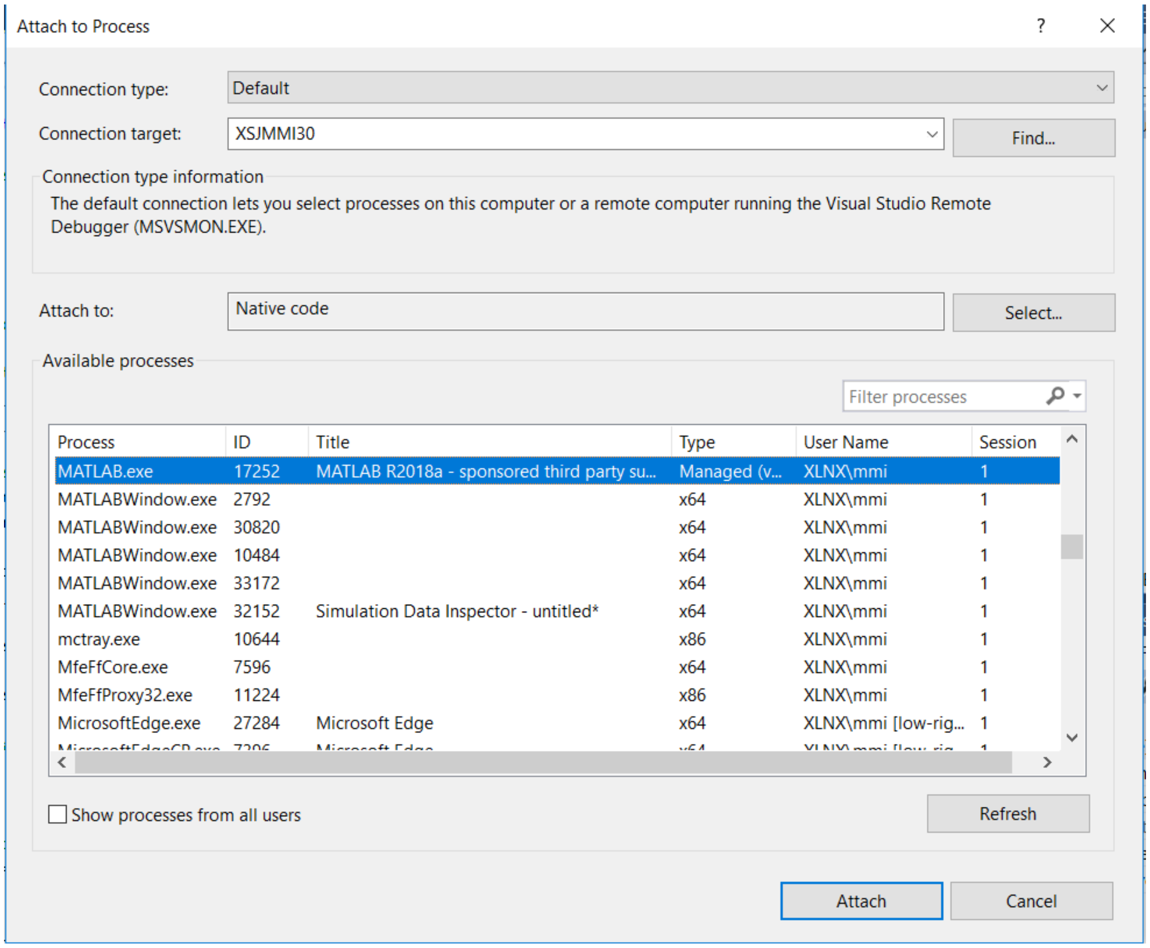
Conceptually this is all simple, but because it is difficult to write error free C/C++, it is desirable to have the pre-built test benches and verification environment of Simulink combined with the ability to debug C/C++ using Microsoft Visual C (MSVC) or GNU debugger. This capability exists, and the steps to single step or debug the `ms.cpp` design using MSVC are:

1. Make sure the Simulink model is opened and ready to simulate in MSVC.
2. Set to build the debug DLL with Visual Studio compiler at the Simulink console (the DLL is automatically built when you run the simulation) from the Simulink command line >> **xmcImportFunctionSettings** ('build', 'debug', 'compiler', 'Visual Studio').
3. Launch Visual Studio.
4. Attach the running MATLAB process from the Visual Studio menu **Debug → Attach to Process...**.

   Select the **MATLAB.exe** process and click on the **Attach** button.

Note: Make sure the **Attach to: Native code** option is set.

Figure 14: **Attach the MATLAB Process**


Attach to Process

Connection type: Default

Connection target: XSJMMI30

Connection type information

The default connection lets you select processes on this computer or a remote computer running the Visual Studio Remote Debugger (MSVSMON.EXE).

Attach to: Native code

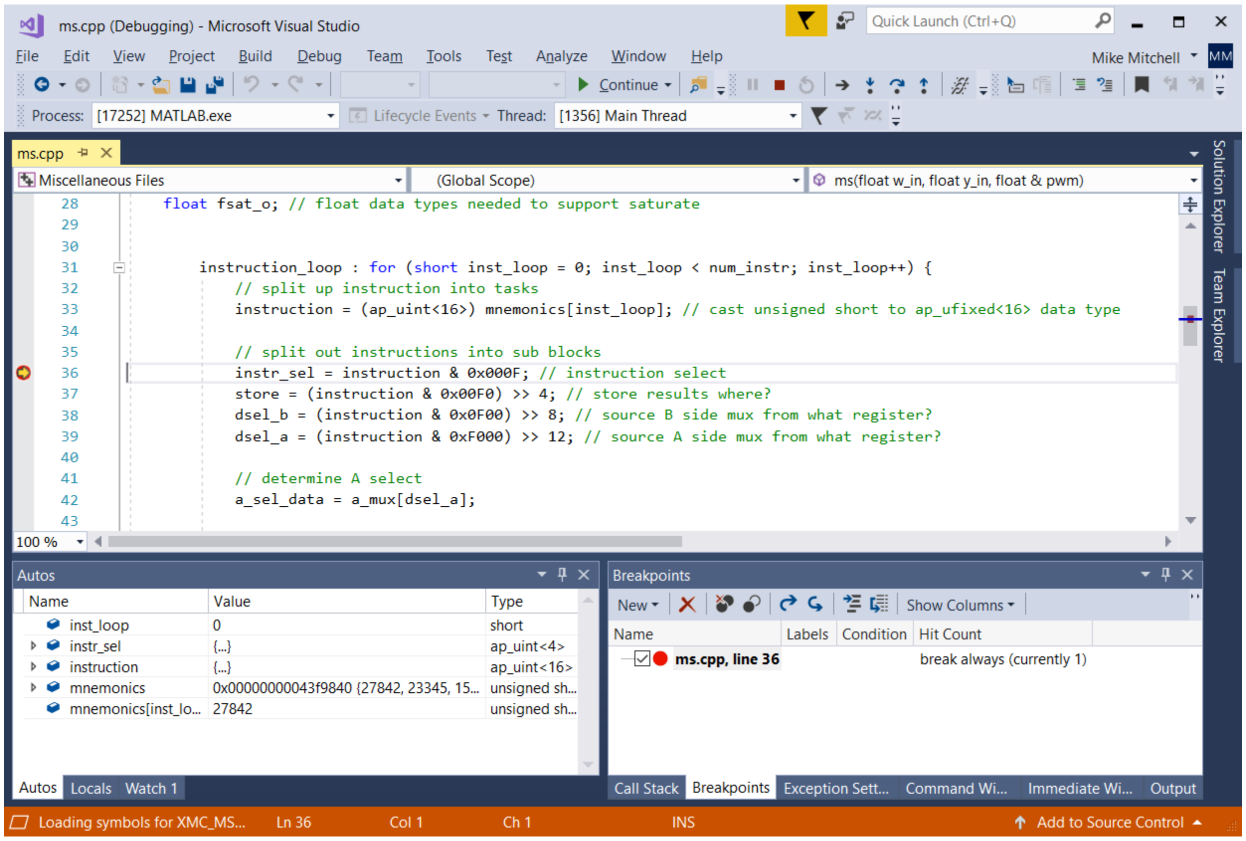
Available processes

Process	ID	Title	Type	User Name	Session
MATLAB.exe	17252	MATLAB R2018a - sponsored third party su...	Managed (v...	XLNX\mmi	1
MATLABWindow.exe	2792		x64	XLNX\mmi	1
MATLABWindow.exe	30820		x64	XLNX\mmi	1
MATLABWindow.exe	10484		x64	XLNX\mmi	1
MATLABWindow.exe	33172		x64	XLNX\mmi	1
MATLABWindow.exe	32152	Simulation Data Inspector - untitled*	x64	XLNX\mmi	1
mctray.exe	10644		x86	XLNX\mmi	1
MfeFfCore.exe	7596		x64	XLNX\mmi	1
MfeFfProxy32.exe	11224		x86	XLNX\mmi	1
MicrosoftEdge.exe	27284	Microsoft Edge	x64	XLNX\mmi [low-rig...	1

5. Set user breakpoint(s) in the C/C++ code (from opened files in Visual Studio).
6. Run the simulation in Simulink.
7. Use the MSVC menu to debug the C/C++ code.

In MSVC, you can see a breakpoint set at line 36 in the following figure.

Figure 15: **MSVC Single Step Implementation**


ms.cpp (Debugging) - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
File Edit View Project Build Debug Team Tools Test Analyze Window Help
Mike Mitchell
Continue
Process: [17252] MATLAB.exe
Lifecycle Events
Thread: [1356] Main Thread
ms.cpp
Miscellaneous Files
(Global Scope)
ms(float w_in, float y_in, float & pwm)
Solution Explorer
Team Explorer
100 %

```
    float fsat_o; // float data types needed to support saturate


        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];

```

Autos

Name	Value	Type
inst_loop	0	short
instr_sel	{...}	ap_uint<4>
instruction	{...}	ap_uint<16>
mnemonics	0x00000000043f9840 {27842, 23345, 15...	unsigned sh...
mnemonics[inst_lo...	27842	unsigned sh...

Breakpoints

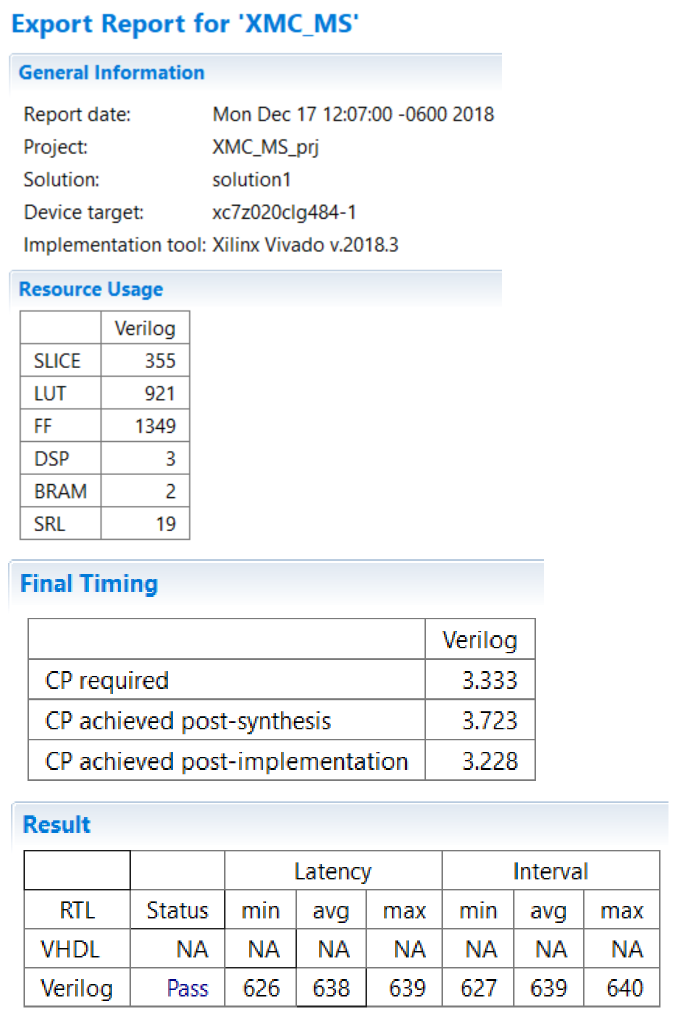
Name	Labels	Condition	Hit Count
ms.cpp, line 36			break always (currently 1)

Call Stack Breakpoints Exception Sett... Command Wi... Immediate Wi... Output
Loading symbols for XMC_MS... Ln 36 Col 1 Ch 1 INS
Add to Source Control

Having the ability to set breakpoints and debug in your imported C/C++ code is a huge advantage for development. Now you have the power of Simulink for test bench creation, verification, and visualization of results for any imported C/C++ modules.

Running the `ms.cpp` source code without any optimizations gives the following results.

Figure 16: **HLS Baseline Math Sequencer Results**



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

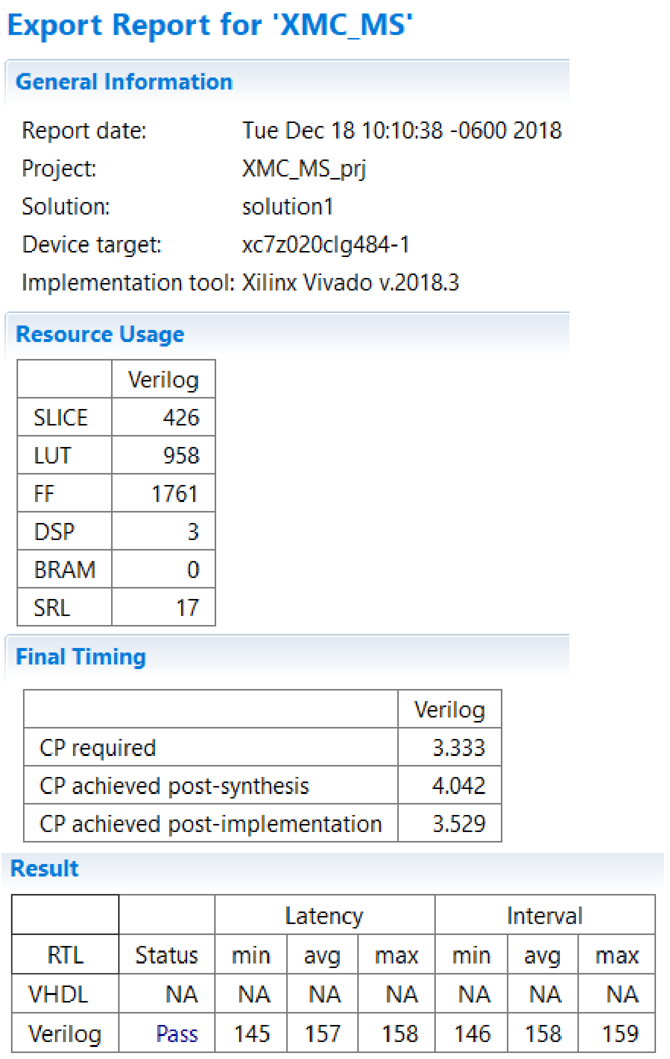
	Verilog
CP required	3.333
CP achieved post-synthesis	3.723
CP achieved post-implementation	3.228

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

Applying two HLS instructions to pipeline the instruction loop and partition the b_mux (e.g., the lines that start with #pragma in the `ms.cpp` source code above) gives the following more optimized HLS results.

Figure 17: **Math Sequencer Implementation Results**



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

The following table compares the XMC blockset with the Math Sequencer approach.

Table 2: **Math Sequencer vs. XMC Native Block Resource, Latency, and Clock Frequency Comparison**

SPFP Data Type	DSP	LUTs	Flip-Flops	Block RAM	Latency (Clocks)	Clock (MHz)
XMC native blocks	6	981	1606	0	142	289.52
Math Sequencer	3	958	1761	0	158	283.37

The MS has some potential advantage because you can easily change the control algorithm or operators (for example, you could add a divide operator or a square root function to calculate magnitude), make the instruction sequence field upgradeable so the algorithm can change later (without changing the hardware implementation), and reduce resources at the expense of latency.

Reference Design

Download the reference design files for this application note from the Xilinx website.

Reference Design Matrix

The following checklist indicates the procedures used for the provided reference design.

Table 3: **Reference Design Matrix**

Parameter	Description
General	
Developer name	Xilinx
Target devices	XC7Z020CLG484-1
Source code provided?	Yes
Source code format (if provided)	Simulink (slx) and C++ design files
Design uses code or IP from existing reference design, application note, 3rd party or Vivado software? If yes, list.	Yes, *Floating-Point PID Controller Design with Vivado HLS and System Generator for DSP* (XAPP1163)
Simulation	
Functional simulation performed	Yes
Timing simulation performed?	Yes
Test bench provided for functional and timing simulation?	Yes, as part of HLS automated design flow
Test bench format	Simulink (xls) or C++ based on XMC or HLS tools flow
Simulator software and version	Vivado simulator 2018.3
SPICE/IBIS simulations	No
Implementation	
Synthesis software tools/versions used	Vivado Design Suite 2018.3
Implementation software tool(s) and version	Vivado Design Suite 2018.3
Static timing analysis performed?	Yes
Hardware Verification	
Hardware verified?	No
Platform used for verification	N/A

Setup

The reference design includes the following files:

- `create_library.m`: MATLAB file used to create the C++ simulatable XMC module.
- `ClosedLoopPID_XMC_SPFP.slx`: Single-precision floating-point PID controller using native XMC blocks.
- `ClosedLoopPID_XMC.slx`: Fixed-point PID controller using native XMC blocks.
- `ClosedLoopPID_MS_XMC.slx`: Math sequencer design.
- `ms.cpp, ms.h`: Math Sequencer C++ source files.

- `math_sequencer_rv2.slxs`: Microsoft Excel file used to create Math Sequencer instructions.

Running the Reference Design

1. From the MATLAB command line execute the `create_library.m` file to create the C++ simulatable XMC module from the `ms.cpp` and `ms.h` C++ source files.
2. Use Simulink to simulate the `ClosedLoopPID_XMC_SPFP.slx` single-precision floating-point PID controller using native XMC blocks.
3. Use Simulink to simulate the `ClosedLoopPID_XMC.slx` fixed-point PID controller using native XMC blocks.
4. Use Simulink to simulate the `ClosedLoopPID_MS_XMC.slx` math sequencer design.
5. From within the Simulink environment, use the Model Composer Hub to generate the corresponding HLS project design for the models used in step 2 through step 4.
6. Use the HLS flow to determine the resources, timing, and RTL verification for step 2 through step 4.

Conclusion

This application note reimplements the PID controller design of *Floating-Point PID Controller Design with Vivado HLS and System Generator for DSP* (XAPP1163) using the native XMC blockset and contrasts the differences for both fixed- and floating-point approaches. In addition, the approach is augmented with a C++ based, flexible Math Sequencer used for applications requiring additional algorithm flexibility and reduced resources at the expense of latency (e.g., serial implementation vs. parallelism).

Throughout the process it is notable that:

- XMC simplifies test bench development, visualization of results, and debug by allowing the user to take advantage of the many inherent Simulink capabilities and toolboxes.
- The design provides a flexible implementation approach using either native XMC blocksets or custom created and imported C/C++.
- Having C++ based models reduces simulation and development time.
- The design provides native debug of imported C/C++ modules during development using Microsoft Visual C or a GNU debugger.
- A test bench and design can be created, evaluated, and exported as an executable C++ design, Vivado IP catalog, or System Generator IP.
- An HLS project created as part of the export process can also be used to further optimize the design performance.
- The user has the ability to independently quantize each variable in the HLS based C code. For fixed-point data types that might require bit growth and truncation after a math operation, XMC allows you to take advantage of automated data type propagation after a math operation as opposed to having to manually define the data size for each variable in C++ for HLS.

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
03/14/2018 Version 1.0	
Initial release.	N/A

Please Read: Important Legal Notices

AUTOMOTIVE APPLICATIONS DISCLAIMER

Copyright