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](image)

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

Figure 3: **Simulink Scope Simulation Results**

Figure 4: **Simulation Data Inspector**
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

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'

<table>
<thead>
<tr>
<th>General Information</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Report date:</td>
<td>Mon Dec 17 10:18:50 -0600 2018</td>
</tr>
<tr>
<td>Project:</td>
<td>XMC_PID_prj</td>
</tr>
<tr>
<td>Solution:</td>
<td>solution1</td>
</tr>
<tr>
<td>Device target:</td>
<td>xc7z020clg484-1</td>
</tr>
<tr>
<td>Implementation tool:</td>
<td>Xilinx Vivado v.2018.3</td>
</tr>
</tbody>
</table>

### Resource Usage

<table>
<thead>
<tr>
<th></th>
<th>VHDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLICE</td>
<td>221</td>
</tr>
<tr>
<td>LUT</td>
<td>459</td>
</tr>
<tr>
<td>FF</td>
<td>822</td>
</tr>
<tr>
<td>DSP</td>
<td>5</td>
</tr>
<tr>
<td>BRAM</td>
<td>0</td>
</tr>
<tr>
<td>SRL</td>
<td>0</td>
</tr>
</tbody>
</table>

### Final Timing

<table>
<thead>
<tr>
<th></th>
<th>VHDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>CP required</td>
<td>3.333</td>
</tr>
<tr>
<td>CP achieved post-synthesis</td>
<td>4.066</td>
</tr>
<tr>
<td>CP achieved post-implementation</td>
<td>3.216</td>
</tr>
</tbody>
</table>

### Result

<table>
<thead>
<tr>
<th></th>
<th>Latency</th>
<th>Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RTL</td>
<td>Status</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VHDL</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Verilog</td>
<td>Pass</td>
<td>76</td>
</tr>
</tbody>
</table>
**Figure 8:** PID Single-Precision Floating-Point HLS Implementation Results

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

<table>
<thead>
<tr>
<th>XMC Data Type</th>
<th>DSP</th>
<th>LUTs</th>
<th>Flip-Flops</th>
<th>Block RAM</th>
<th>Latency (Clocks)</th>
<th>Clock (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed Point</td>
<td>5</td>
<td>459</td>
<td>822</td>
<td>0</td>
<td>81</td>
<td>311</td>
</tr>
<tr>
<td>SPFP</td>
<td>3</td>
<td>958</td>
<td>1761</td>
<td>0</td>
<td>142</td>
<td>290</td>
</tr>
</tbody>
</table>

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

```c
#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[12] = {0, Gi, Gd, c, Op, 0, 0, 0, 0, 0, minus1, plus1, zeroc};
    // B mux data
    static float b_mux[12] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, minus1, plus1, zeroc};
    // 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] =
        [0x6CC2,0x5B31,0x633,0x2652,0x1662,0x3442,0xB272,0x82A1,0x7C42,0x7A11,0x93B1,0xA5A1,0xA03,0x4642,0xD8B5,0xA791,0x79A1,0xA23,0x8074,0x9084,0x7D5,0x8E5];
    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
```

XAPP1341 (v1.0) March 14, 2019
Application Note

www.xilinx.com
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.
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.
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.
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.
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.
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**

<table>
<thead>
<tr>
<th>SPFP Data Type</th>
<th>DSP</th>
<th>LUTs</th>
<th>Flip-Flops</th>
<th>Block RAM</th>
<th>Latency (Clocks)</th>
<th>Clock (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>XMC native blocks</td>
<td>6</td>
<td>981</td>
<td>1606</td>
<td>0</td>
<td>142</td>
<td>289.52</td>
</tr>
<tr>
<td>Math Sequencer</td>
<td>3</td>
<td>958</td>
<td>1761</td>
<td>0</td>
<td>158</td>
<td>283.37</td>
</tr>
</tbody>
</table>

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.
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

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

Setup

The reference design includes the following files:

- `create_library.m`: MATLAB file used to create the C++ simulatable XMC module.
- `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.
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.
Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos.

AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.