Summary

This application note describes how to use Vivado® High Level Synthesis (HLS) to develop a floating-point matrix multiplication accelerator with an AXI4-Stream interface and connect it to the Accelerator Coherency Port (ACP) of the ARM CPU in the Zynq®-7000 All Programmable SoC. The floating-point matrix multiplication accelerator modeled in the C/C++ code can be quickly implemented and optimized into an RTL design using Vivado HLS. The solution is then exported as a pcore connected with an automatically created AXI4-Stream interface to the ACP of the Zynq-7000 AP SoC Programmable System (PS). The connection is made through a Direct Memory Access (DMA) core in the Programmable Logic (PL) of the Zynq-7000 device. Xilinx Platform Studio (XPS) is used to design the hardware of the Zynq-7000 AP SoC PL, including the matrix multiplier peripheral, the DMA engine and an AXI timer. The Software Development Kit (SDK) is used to design the software on the Zynq-7000 AP SoC PS to manage the peripherals.

Introduction

Matrix multiplication is used in nearly every branch of applied mathematics. For example, matrix multiplication is used by beam-forming, which is the process of phasing a receiving antenna digitally by computer calculation in modern radar systems. The Xilinx Vivado HLS tool allows floating-point algorithms to be quickly specified in C/C++ code, and optimized and implemented on the Xilinx Zynq-7000 AP SoC. This delivers cost, performance and power benefits for designers relying on traditional micro-processors for implementing floating-point algorithms [Ref 1] [Ref 2].

Starting from the application of floating point multiplication on 32x32 matrices, this document explains the following aspects of Xilinx FPGA design flow:

- Compiling and optimizing the C/C++ floating-point design into a high-performance hardware accelerator using Vivado HLS.
- Specifying and generating an AXI4-Stream interface for the hardware accelerator using C++ templates in Vivado HLS.
- Using XPS to connect the hardware accelerator to an AXI DMA peripheral in the Zynq-7000 AP SoC PL and to the ACP of the Zynq-7000 AP SoC PS.
- Writing the software running on the ARM CPU with function calls to the hardware accelerator and measuring the system-level performance.

Figure 1 shows the block diagram of the system to be implemented on the Zynq-7000 device.

The design can be found at:

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

The design is organized in three folders:

- **hls**: C++ code of the matrix multiplier and AXI4-Stream interface wrapper. This can be also seen as a good example of C++ templates.
- **edk**: Zynq-7000 device HW design and SDK project.
- **arm**: C code running on the ARM CPU to be compiled with the SDK.
The following step-by-step design procedure uses Vivado HLS 2012.4 and PlanAhead (ISE/EDK/SDK) 14.4 IDE release tools, targeting the Zynq-7000 All Programmable SC Evaluation Kit (ZC702).

The matrix multiplication algorithm \( A \times B = C \) is very simple. There are three nested loops:

- The first loop (L1) iterates over the elements within a row of the input matrix \( A \).
- The second loop (L2) iterates over the elements within a column of the input matrix \( B \).
- The third loop (L3) multiplies each index of row vector \( A \) with an index of column vector \( B \) and accumulates it to generate the elements of a row of the output matrix \( C \).

The C++ code of the function to be optimized is as follows:

```cpp
template <typename T, int DIM>
void matrix_multiply_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;
    }

    return;
}
```

After the algorithm has been captured in C++ code, Vivado HLS can be used to synthesize this into an RTL implementation.

In addition to the C source code, Vivado HLS accepts as inputs a target clock frequency, a target device specification and user directives (commands) that can be used to control and direct specific optimizations. The easiest way to understand the function and capabilities of Vivado HLS is to step through an example. For more information on Vivado HLS see the Vivado...
The following code specifies the clock period and target device:

```
set_part {xc7z020clg484-1}
create_clock -period 10
```

Given the two input source code examples, Vivado HLS will:

- Transform each of the operations in the C code into an equivalent hardware operation and schedule those operations into clock cycles. Using knowledge of the clock period and device delays, it will put as many operations as possible into a single clock cycle.
- Use interface synthesis to automatically synchronize how the data can be brought into the hardware block and written out. For example if data is supplied as an array, it will automatically construct an interface to access a RAM block (other IO interface options can be specified).
- Map each hardware operation onto an equivalent hardware unit in the Zynq-7000 AP SoC.
- Perform any user specified optimizations, such as pipelined or concurrent operations.
- Output the final design, with reports, in Verilog and VHDL for implementation in the Zynq-7000 AP SoC.

The reports generated by synthesizing the code in the example core can explain the operation and capabilities, including the initial performance characteristics (default synthesis results).

For this example, Vivado HLS analyzes the operations in the C code and determines that it will take 329793 clock cycles to calculate the result using the specified target technology and clock period. This design could execute with a maximum clock frequency of 8.09 ns.

The area estimates in Figure 2 show how many device resources the design is expected to use: five DSP48 slices, about 464 Flip-Flops (FFs) and 842 Look-Up-Tables (LUTs).

These are estimated figures because the RTL synthesis process still needs to transform the RTL code into gate-level components and place and route them in the device. There may be other gate-level optimizations that impact the final results.
Figure 2: Initial Performance Characteristics in 32-bit floating point
Figure 3 shows the C function arguments transformed by interface synthesis into IO ports. This process enables the ports to be connected to other blocks in the design.

Changes during this step include:

- Clock and reset signals were added to the design (ap_clk, ap_rst).
- A design-level protocol were added to the design. This is the default, but also optional. This allows the design to be started (ap_start) and indicates when it is ready for new inputs, has completed (ap_done) or is idle.
- Array arguments were transformed into a RAM interface with the appropriate address, enable and write signals to access a Xilinx block RAM. Additionally, Vivado HLS has automatically determined that the performance can be improved if port din uses a dual-port block RAM (this can be configured to a single-port block RAM if desired).
- Vivado HLS created an RTL implementation where the operations in the C code and the IO operations have been implemented without any requirement to know an RTL design language, such as Verilog or VHDL, or without knowing anything about RTL design in general.

Optimizing the RTL

The initial design created by Vivado HLS can be optimized. Figure 4 shows the comparison of the first four solutions. The optimizations were applied to reduce the amount of clock cycles needed to compute the matrix multiplication. For more details on the optimizations provided by Vivado HLS, see the Vivado HLS Tutorial [Ref 4].

Solution 4 is about 20 times faster than the initial design (Solution 1) at the expense of more resources: 10 DSP48 slices, 2426 FFs and 3261 LUTs. The estimated clock period of 8.09 ns means the output data rate is 7.47 Kilo Samples Per Second (KSPS), as shown in Equation 1.

\[
16535 \times 8.09\text{ns} = 7.47 \text{KSPS} = 0.134\text{ms}
\]  

Equation 1
Figure 4 reports the optimization directives of Solution 4.

The following code shows the optimization directives for Solution 4:

```markdown
set_directive_pipeline -II 1 "matrix_multiply_hw/L2"
```

The performance of Solutions 5 to 8 is detailed in Figure 5. Clearly the highest performance is achieved by solution 8, with only 1190 clock cycles to compute the floating point matrix multiplication. This number is obtained by using 160 DSP48 slices, 13427 FFs and 23113 LUTs, which represent respectively the 72%, 12%, and 43% of the available resources on the Zynq-7000 AP SoC. Solution 8 is the only one that exhibits a pipeline initialization interval of 1, which means a throughput of 1. For this example, the data rate is 123.6 Millions of Samples Per Second (MSPS), and the whole output matrix is generated in the time period of 1190 x 8.09 ns, which is 9.62 s.

Achieving a throughput of 1 means that one matrix output sample is generated on each clock cycle. This is a design choice. If you want a less expensive design in terms of FPGA resources, you can select Solution 4 or Solution 5.
The following code shows the optimization directives of Solution 8.

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

AXI4-Stream is a communication standard for point-to-point data transfer without the need of addressing or external bus masters [Ref 5]. This protocol allows both cores to dynamically synchronize on data using a producer-consumer model. Depending on the implementation of AXI4-stream used, the communication channel can be built as wires or with storage to account for data rate mismatches at each end.

The matrix multiplier core designed with Vivado HLS is connected to the DMA controller using AXI4-Stream interfaces. Burst formatting, address generation and scheduling of the memory transaction is left to the AXI DMA IP.

The architecture of the system, shown in Figure 1, uses the ACP port to connect the AXI DMA to the L2 cache of the ARM processor (an alternative approach is to use the HP ports for connection to the external DDR memory). From the perspective of the Vivado HLS core, the memory interface through the DMA is the same regardless of whether the memory is DDR or L2 cache. It is a system architecture decision to share data between the processor and the Vivado HLS core by either using DDR or the L2 cache. The DDR provides the ability to transfer more data than the L2 cache, but the L2 cache has a lower latency in communication than the DDR.

To connect the Vivado HLS matrix multiplier block to the AXI DMA, we need to change the code shown in Matrix Multiply Design with Vivado HLS and add more functions to be synthesized, as illustrated by the new C++ code shown in this section. In particular pop_stream and push_stream are functions to extract and insert elements from/into an AXI4-Stream interface. These functions also implement the conversion between the 32-bit floating point data of the matrices and the 32-bit unsigned data of AXI.

The code below shows the usage of the AXI_Val data type. This is a user-defined data type that expresses the side channel information associated with the AXI4-Stream interface.

### Vivado HLS Report Comparison

<table>
<thead>
<tr>
<th>Performance</th>
<th>solution5</th>
<th>solution6</th>
<th>solution7</th>
<th>solution8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock Period (ns)</td>
<td>10.00</td>
<td>10.00</td>
<td>10.00</td>
<td>10.00</td>
</tr>
<tr>
<td>C Estimate</td>
<td>8.09</td>
<td>8.09</td>
<td>8.09</td>
<td>8.09</td>
</tr>
<tr>
<td>VHDL Estimate</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Verilog Estimate</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

<p>| Overall performance (clock cycles): |</p>
<table>
<thead>
<tr>
<th>Throughput (B)</th>
<th>solution5</th>
<th>solution6</th>
<th>solution7</th>
<th>solution8</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.35</td>
<td>4.29</td>
<td>2.13</td>
<td>1.19</td>
<td></td>
</tr>
<tr>
<td>Latency</td>
<td>833</td>
<td>4236</td>
<td>1213</td>
<td>1930</td>
</tr>
</tbody>
</table>

### Resource Usage

<table>
<thead>
<tr>
<th>Component</th>
<th>solution5</th>
<th>solution6</th>
<th>solution7</th>
<th>solution8</th>
</tr>
</thead>
<tbody>
<tr>
<td>DYNIE</td>
<td>20</td>
<td>40</td>
<td>80</td>
<td>160</td>
</tr>
<tr>
<td>EXP</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>FIFO</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Memory</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Multiplier</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Register</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Shifter</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Total</td>
<td>20</td>
<td>40</td>
<td>80</td>
<td>160</td>
</tr>
</tbody>
</table>

**Figure 5: Performance Comparison of Solution 5 to 8**

AXI4-Stream Interface with Vivado HLS

AXI4-Stream is a communication standard for point-to-point data transfer without the need of addressing or external bus masters [Ref 5]. This protocol allows both cores to dynamically synchronize on data using a producer-consumer model. Depending on the implementation of AXI4-stream used, the communication channel can be built as wires or with storage to account for data rate mismatches at each end.
Vivado HLS, any side channel information that is not part of the protocol handshake must be expressed in the C/C++ code and used in some way. This means that while Vivado HLS will abstract the \texttt{TREADY} and \texttt{TVALID} signals, all other signals in the AXI4-Stream interface must be part of the user code. Also, besides the \texttt{TDATA}, \texttt{TREADY} and \texttt{TVALID} signals, all other AXI4-Stream interface signals are optional. The use of side channel signals depends on the blocks connected to the Vivado HLS AXI4-Stream interface.

\textbf{Figure 6} shows the synthesis report of the AXI4-Stream matrix multiplier. Note that the latency now is 4269 clock cycles. The total latency values are computed by taking into account the time to transfer each matrix to and from the accelerator, the time of the computation, and the setup for the hardware function. The time to transfer each matrix is 1024 clock cycles for 1024 32-bit floating point values plus an additional two clock cycles for the FOR loop epilogue and prologue. Therefore, the total data transfer time is 3072 clock cycles. The computation time for the matrix multiplication is 1190 clock cycles. This results in a function initialization interval (II) of 4268 clock cycles with a latency of 4269 clock cycles. The additional clock cycle is from the start up of the function.

The overall resource utilization is 34 Block RAM18K (block RAM units of 18 Kbit capacity), 160 DSP48 slices, 13546 FFs and 23620 LUTs. These figures represent respectively the 12\%, 72\% 12\% and 44\% utilization of the available resources on the Zynq-7000 device. The data rate remains unchanged from 123.6 MSPS and the whole output matrix is generated in the time period of 4269 * 8.09 ns (34.53 seconds), which includes the data communication.

In conclusion, the automatically generated AXI4-Stream interface has an overhead of 533 clock cycles (4269 - 1190 - 3 * 1024) of latency and 119 FFs and 507 LUTs (13546 - 13427 and 23620 - 23113) of resources occupation.

The following example code details the new C++ code used to generate an AXI4-Stream matrix multiplier core.

```cpp
#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 dut_mmult_accel_core ( AXI_VAL in_stream[2048], AXI_VAL out_stream[1024])
{
    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
    matrix_multiply_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);
    }

return;
}

// THIS IS THE TOP LEVEL DESIGN THAT WILL BE SYNTHESIZED
void mmult_accel_core (AXI_VAL in_stream[2048], AXI_VAL out_stream[1024])
{
    // Map ports to Vivado HLS interfaces
    #pragma HLS INTERFACE ap_fifo port=in_stream
    #pragma HLS INTERFACE ap_fifo port=out_stream

    // Map HLS ports to AXI interfaces
    #pragma HLS RESOURCE variable=in_stream  core=AXIS metadata="-bus_bundle INPUT_STREAM"
    #pragma HLS RESOURCE variable=out_stream core=AXIS metadata="-bus_bundle OUTPUT_STREAM"
    #pragma HLS RESOURCE variable=return core=AXI4LiteS metadata="-bus_bundle CONTROL_BUS"

dut_mmult_accel_core <float, 32, 32*32, 4, 5, 5>(in_stream, out_stream);

return;
}
The Vivado HLS project is created by running the provided TCL script in the design archive from the Vivado HLS Command Prompt shell (Figure 7).

vivado_hls -f script_run_z.tcl

Figure 6: Synthesis Estimation of the AXI4-Stream Matrix Multiplier Core

The following code example shows the TCL script that creates the Vivado HLS project (mmult_accel_core.cpp). The last command in this example exports the matrix multiplier design and generates the pcore with an AXI4-Stream interface.
Note: C++ code can also be compiled with MS Visual C++ 2010 Express Edition. The MSVC project file is available in the ZIP archive.

open_project hls_fp_matrix_mult_prj -reset
set_top matrix_multiply_hw
add_files mmult_accel_core.cpp
add_files -tb mmult_accel_core.cpp

#solution 1
open_solution "solution1"
set_part (xc7z020clg484-1)
create_clock -period 10
source "./directives/z/directives1.tcl"
csynth_design

#solution 2
open_solution "solution2"
set_part (xc7z020clg484-1)
create_clock -period 10
source "./directives/z/directives2.tcl"
csynth_design

#solution 3
open_solution "solution3"
set_part (xc7z020clg484-1)
create_clock -period 10
source "./directives/z/directives3.tcl"
csynth_design

#solution 4
open_solution "solution4"
set_part (xc7z020clg484-1)
create_clock -period 10
source "./directives/z/directives4.tcl"
csynth_design

#solution 5
open_solution "solution5"
set_part (xc7z020clg484-1)
create_clock -period 10
source "./directives/z/directives5.tcl"
csynth_design

#solution 6
open_solution "solution6"
set_part (xc7z020clg484-1)
create_clock -period 10
source "./directives/z/directives6.tcl"
csynth_design

#solution 7
open_solution "solution7"
set_part (xc7z020clg484-1)
create_clock -period 10
source "./directives/z/directives7.tcl"
csynth_design

#solution 8
open_solution "solution8"
set_part (xc7z020clg484-1)
The following code example shows the other routines of the AXI4-Stream Matrix Multiplier C++ project.

```cpp
template <typename T, int DIM>
void matrix_multiply_ref(T a[DIM][DIM], T b[DIM][DIM], T out[DIM][DIM])
{
    // matrix multiplication of a A*B matrix
    for (int ia = 0; ia < DIM; ++ia)
        for (int ib = 0; ib < DIM; ++ib)
        {
            float sum = 0;
            for (int id = 0; id < DIM; ++id)
                sum += a[ia][id] * b[id][ib];

            out[ia][ib] = sum;
        }

    return;
}

// --------------------------------------------------------
// functions to insert and extract elements from an axi stream
// including conversion to correct data type

template <typename T, int U, int TI, int TD>
inline T pop_stream(ap_axiu <sizeof(T)*8,U,TI,TD> const &e)
{
    assert(sizeof(T) == sizeof(int));
    union
    {
        int ival;
        T oval;
    } converter;
    converter.ival = e.data;
    T ret = converter.oval;

    volatile ap_uint<sizeof(T)> strb = e.strb;
    volatile ap_uint<sizeof(T)> keep = e.keep;
    volatile ap_uint<U> user = e.user;
    volatile ap_uint<1> last = e.last;
    volatile ap_uint<TI> id = e.id;
    volatile ap_uint<TD> dest = e.dest;
}```
return ret;
}
template <typename T, int U, int TI, int TD>
inline ap_axiu sizeof(T)*8,U,TI,TD> push_stream(T const &v, bool last = false)
{
    ap_axiu<sizeof(T)*8,U,TI,TD> e;
    assert(sizeof(T) == sizeof(int));
    union {
        int oval;
        T ival;
    } converter;
    converter.ival = v;
    e.data = converter.oval;
    // set it to sizeof(T) ones
    e.strb = -1;
    e.keep = 15; //e.strb;
    e.user = 0;
    e.last = last ? 1 : 0;
    e.id = 0;
    e.dest = 0;
    return e;
}

// --------------------------------------------------------
// effective test bench
//
template <typename T, int DIM, int SIZE, int U, int TI, int TD>
int test_matrix_mult(void)
{
    int i,j, err;
    T matOp1[DIM][DIM], matOp2[DIM][DIM],
    matMult_sw[DIM][DIM], matMult_hw[DIM][DIM];

    /** Matrix Initiation */
    for(i = 0; i<DIM; i++)
        for(j = 0; j<DIM; j++)
            matOp1[i][j] = (float)(i+j);
    for(i = 0; i<DIM; i++)
        for(j = 0; j<DIM; j++)
            matOp2[i][j] = (float)(i*j);
#ifdef DB_DEBUG
    printf("DEBUGGING AXI4 STREAMING DATA TYPES!\r\n");
#endif
    // prepare data for the DUT
    AXI_VAL inp_stream[2*SIZE];
    AXI_VAL out_stream[SIZE];
    assert(sizeof(T)*8 == 32);
    // stream in the first input matrix
    for(int i=0; i<DIM; i++)
        for(int j=0; j<DIM; j++)
        {
            int k = i*DIM+j;
            inp_stream[k]= push_stream<T,U,TI,TD>(matOp1[i][j],0);
// stream in the second input matrix
for(int i=0; i<DIM; i++)
    for(int j=0; j<DIM; j++)
    {
        int k = i*DIM+j;
        inp_stream[k+SIZE] = push_stream<T,U,TI,TD>(matOp2[i][j],
            k == (SIZE-1));
    }

//call the DUT
dut_mmult_accel_core<T, DIM, SIZE, U, TI, TD>(inp_stream, out_stream);

// extract the output matrix from the out stream
for(int i=0; i<DIM; i++)
    for(int j=0; j<DIM; j++)
    {
        int k = i*DIM+j;
        matMult_hw[i][j] = pop_stream<T,U,TI,TD>(out_stream[k]);
    }

#else
printf("NORMAL MODE\r\n");
matrix_multiply_hw<T, DIM>(matOp1, matOp2, matMult_hw);
#endif

/* reference Matrix Multiplication */
matrix_multiply_ref<T, DIM>(matOp1, matOp2, matMult_sw);

/** Matrix comparison */
err = 0;
for (i = 0; (i<DIM && !err); i++)
    for (j = 0; (j<DIM && !err); j++)
        if (matMult_sw[i][j] != matMult_hw[i][j]) err++;

if (err == 0)
    printf("Matrixes identical ... Test successful!\r\n");
else
    printf("Test failed!\r\n");

return err;

int main(void)
{
    typedef float T;
    int const DIM = 32;
    int const SIZE = DIM*DIM;
    int ret_val = 0;

    ret_val = test_matrix_mult<T, DIM, SIZE, 4,5,5>();
    return ret_val;
}
AXI DMA Overview

The AXI DMA core provides high-bandwidth direct memory access between memory and peripherals through an AXI4-Stream interface. The core design has following AXI4 interfaces:

- AXI4-Lite Slave
- AXI4 Memory Map Read Master/Write Master
- Optional AXI4 Scatter/Gather Read/Write Master
- AXI4 to AXI4-Stream (MM2S) Stream Master
- AXI4-Stream to AXI4 (S2MM) Stream Slave
- AXI Control AXI4-Stream Master
- AXI Status AXI4-Stream Slave

Figure 8 illustrates the AXI DMA interfaces and the data streaming. The AXI DMA works in two different modes, only one at a time: the Scatter/Gather and Simple DMA modes. For each mode, there is a proper register to be configured with AXI4-Lite slave interface. In our implementation, we are using the Simple DMA mode.

The Simple DMA mode provides a configuration for doing Simple DMA transfers on MM2S and S2MM channels that requires less FPGA resource utilization. The AXI4 Read (MM2S) interface reads the data from a master external memory, and then the DMA data mover transmits the data to the slave external memory through the S2MM channel.
data to a slave peripheral through the AXI4-Stream (MM2S) port. Similarly, a master peripheral can send data to the AXI4 Write (S2MM) interface which writes that data to a slave external memory through the AXI4-Stream (S2MM) port.

DMA transfers are initiated by accessing the control, source or destination address and length registers.

MM2S Channel setup sequence:
1. Start the MM2S channel by setting the run/stop bit in the Control register.
2. Write a valid source address to the MM2S Source Address register.
3. Write the number of bytes to transfer in the MM2S Length register. The Length register must be written last.

S2MM Channel setup sequence:
1. Start the S2MM channel by setting the run/stop bit in the Control register.
2. Write a valid destination address to the S2MM Destination Address register.
3. Write the length in bytes of the receive buffer in the S2MM Length register. The Length register must be written last.

The AXI DMA settings in XPS are shown in Figure 17.

---

The ACP port is a 64-bit AXI slave interface on the Snoop Control Unit (SCU) that provides an asynchronous cache-coherent access point directly from the Zynq-7000 SP SoC PL to the Cortex-A9 CPU processor subsystem. The ACP port provides a low-latency path between the PS and the accelerator implemented in the PL. A range of system PL masters can use this interface to access the caches and the memory subsystem exactly in the way the CPU processors accesses the caches to increase the overall system performance of the software application executed.

Any read transactions through the ACP to a coherent region of memory interact with the SCU to check whether the required information is stored within the processor L1 caches. If this is the case, data is returned directly to the requesting component. If the required information is not stored in the L1 cache, it will check the L2 cache before finally being forwarded to the main memory. For write transactions to any coherent memory region, the SCU enforces coherence before the write is forwarded to the memory system. The transaction can also optionally allocate into the L2 cache, removing the power and performance impact of writing through to the off-chip memory. Figure 9 shows the connectivity between the ACP and the memory system connected to the CPU.
This section describes the steps to create the Zynq-7000 device hardware design with Xilinx PlanAhead and Platform Studio 14.4 releases.

Create a basic PlanAhead project (matrix_mult) by selecting all the default settings. When prompted for the part, select the ZC702 board. Here are the detailed instructions:

1. Launch PlanAhead.
2. Create a new project and select:
   - RTL Project
   - Select VHDL as preferred language (optional)
   - No sources, IP, or constraints for now.
3. Select the default part by selecting **Boards > Zynq-7000 ZC702 Evaluation Board**.
4. Click **Finish**.

The PlanAhead GUI opens with an empty project. Next add an embedded subsystem:

1. Select **Add Sources > Add or Create Embedded Sources** (**Figure 10**).
2. Create Sub-Design.
3. Name the new embedded source "system".
4. Click **Finish**. This opens the Xilinx Platform Studio (XPS) where the Zynq-7000 AP SoC PS can be configured and the BSB Wizard appears as shown in **Figure 11**. Click **OK**.

**Figure 10**: PlanAhead Add Embedded Sources
5. Remove the default included peripherals for the Processing System7 (GPIO_SW and LEDs_4Bits). Click Finish. XPS now shows the Zynq-7000 AP SoC PS GUI.

6. Because we are using the ZC702 evaluation board, we import the default configuration of this board. On the Zynq-7000 tab, click Import (Figure 12). Select the ZC702 template and update the MIO and MHS configuration and design files.

7. Copy and paste the folder mmult_accel_core_top_v1_00_a (pcore generated by Vivado HLS) from

   matrix_mult\hls\hls_fp_matrix_mult_prj\solution8_full\impl\pcores to
   matrix_mult\edk\matrix_mult\matrix_mult.srcs\sources_1\edk\system\pcores.

   Select Project > Rescan User Repositories in the XPS GUI. You should see what illustrated in Figure 13.
Adding the Accelerator to the Zynq-7000 Device

The following steps connect the pcore generated by Vivado HLS to the ACP port of the ARM CPU.

1. Click on the green 64b AXI ACP Slave Port block of the GUI and select Enable S_AXI_ACP Interface. Leave all other settings at the default values, as shown in Figure 14.

2. Click on the matrix multiplier pcore available from the User repository in Project Local Pcores to add it to the Zynq-7000 AP SoC PS. Replace GENERIC with XIL_AXI_STREAM_ETH_DATA for both C_INPUT_STREAM_PROTOCOL and C_OUTPUT_STREAM_PROTOCOL, as illustrated in Figure 15. Select the processor instance to make the connections.
3. In the Bus and Bridge section of the IP Catalog, double-click on the AXI interconnect to add the core to the system and accept all the default values (Figure 16).

4. In the DMA and Timer sections of the IP Catalog, double-click on the AXI DMA Engine to add the core to the system. In the configuration Core Configuration screen (Figure 17):
   - Enable Include MM2S Data Realignment Engine
   - Enable Include S2MM Data Realignment Engine
   - Disable Include Scatter Gather Engine
   - Set the Maximum Memory Map Burst Size for MM2S to 256
   - Set the Maximum Memory Map Burst Size for S2MM to 256
5. When requested, select **User will make necessary connections**, and add the core to the system.

6. Add an AXI Timer core. Accept all the default values and select the processor instance to which to connect. This is the last core we need to add.

7. Connect the missing bus interfaces by clicking on the white master squares and slave circles in the XPS assembly view (Figure 18).
   - Connect the S_AXI_LITE from axi_dma_0 to M_AXI_GP0 via interconnect_1.
   - Connect the S_AXI_CONTROL_BUS of mmult_accel_core_top_0 to M_AXI_GP0 via interconnect_1.
   - Connect the AXI timer to M_AXI_GP0 via interconnect_1.
   - Connect M_AXI_MM2S to axi_interconnect_0 (DMA engine).
   - Connect M_AXI_S2MM to axi_interconnect_0 (DMA engine).
   - Connect S_AXI_ACP to axi_interconnect_0 (processor).
   - Connect S_AXIS_S2MM to OUTPUT_STREAM.
   - Connect M_AXIS_MM2S to INPUT_STREAM.

The Bus Interfaces tab should resemble Figure 18.
8. Click the Addresses tab. If anything appears as an unmapped address, click Generate Addresses, as shown in Figure 19.

![Complete Bus Interface on the System Assembly View](image1)

**Figure 18:** Complete Bus Interface on the System Assembly View

9. Connect the unconnected ports to FCLK_CLK0 and FCLK_RESET0_N. Many ports may already be connected by XPS. Check the following items:
   a. Connect ports from axi_interconnect_0 and axi_interconnectc_1:
      - INTERCONNECT_ACLK to processing_system7::FCLK_CLK0
      - INTERCONNECT_ARESETN to processing_system7::FCLK_RESET0_N
   b. Connect ports from processing_system7:
      - S_AXI_ACP_ACLK to processing_system7::FCLK_CLK0
      - M_AXI_GP0_ACLK to processing_system7::FCLK_CLK0
   c. Connect ports from axi_dma_0: s_axi_lite_aclk, m_axi_mm2s_aclk m_axi_s2mm_aclk to processing_system7::FCLK_CLK0
   d. Connect ports from mmult_accel_core_top_0:
      - aclk to processing_system7::FCLK_CLK0
      - reset to processing_system7::FCLK_RESET0_N
   e. Connect the interrupts:
      - from mmult_accel_core_top_0 to processing_system7::IRQ_F2P
      - from axi_timer to processing_system7::IRQ_F2P
   f. Connect ports from axi_timer:
      - s_axi_aclk to processing_system7::FCLK_CLK0

The final result should resemble Figure 20 and Figure 21.
<table>
<thead>
<tr>
<th>Name</th>
<th>Connected Port</th>
<th>Dir</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTERCONNECT ACLK</td>
<td>processing_system7_0:FCLK_CLK0</td>
<td>1</td>
</tr>
<tr>
<td>INTERCONNECT ARESETN</td>
<td>processing_system7_0:FCLK_RESET0_N</td>
<td>1</td>
</tr>
<tr>
<td>axi_interconnect_0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTERCONNECT ACLK</td>
<td>processing_system7_0:FCLK_CLK0</td>
<td>1</td>
</tr>
<tr>
<td>INTERCONNECT ARESETN</td>
<td>processing_system7_0:FCLK_RESET0_N</td>
<td>1</td>
</tr>
<tr>
<td>processing_system7_0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>axi_dma_0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mm2s_introut</td>
<td></td>
<td>O</td>
</tr>
<tr>
<td>s2mm_introut</td>
<td></td>
<td>O</td>
</tr>
<tr>
<td>axi_dma_0.SDVinc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(BUS_E) S_AXI_LITE</td>
<td>Connected to BUS axi_interconnect_1</td>
<td></td>
</tr>
<tr>
<td>s_axi_lite_ack</td>
<td>processing_system7_0:FCLK_CLK0</td>
<td>1</td>
</tr>
<tr>
<td>(BUS_E) M_AXI_MM2S</td>
<td>Connected to BUS axi_interconnect_0</td>
<td></td>
</tr>
<tr>
<td>m_axi_mm2s_ack</td>
<td>processing_system7_0:FCLK_CLK0</td>
<td>1</td>
</tr>
<tr>
<td>(BUS_E) M_AXI_S2MM</td>
<td>Connected to BUS axi_interconnect_0</td>
<td></td>
</tr>
<tr>
<td>m_axi_s2mm_ack</td>
<td>processing_system7_0:FCLK_CLK0</td>
<td>1</td>
</tr>
<tr>
<td>(BUS_E) M_AXI_S2MM_CNTRL</td>
<td>Not connected to BUS or External Ports</td>
<td></td>
</tr>
<tr>
<td>m_axi_s2mm_cntrl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(BUS_E) S_AXI_S2MM_STS</td>
<td>Not connected to BUS or External Ports</td>
<td></td>
</tr>
<tr>
<td>m_axi_s2mm_sts</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(BUS_E) S_AXI/S2MM_axi</td>
<td>Processing System7.0:FCLK_CLK0</td>
<td>1</td>
</tr>
<tr>
<td>axi_timer_0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CaptureTrig0</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>CaptureTrig1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>GenerateOut0</td>
<td></td>
<td>O</td>
</tr>
<tr>
<td>GenerateOut1</td>
<td></td>
<td>O</td>
</tr>
<tr>
<td>PWM0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interrupt</td>
<td>Processing System7.0:BIRQ_F2P</td>
<td>1</td>
</tr>
<tr>
<td>Freeze</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(BUS_E) S_AXI</td>
<td>Connected to BUS axi_interconnect_1</td>
<td></td>
</tr>
<tr>
<td>S_AXI_ACLK</td>
<td>processing_system7_0:FCLK_CLK0</td>
<td>1</td>
</tr>
<tr>
<td>minuit_accel_core_top_0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>interrupt</td>
<td>processing_system7_0:BIRQ_F2P</td>
<td>1</td>
</tr>
<tr>
<td>(BUS_E) S_AXI_CONTROL_BUS</td>
<td>Connected to BUS axi_interconnect_1</td>
<td></td>
</tr>
<tr>
<td>ack</td>
<td>processing_system7_0:FCLK_CLK0</td>
<td>1</td>
</tr>
<tr>
<td>(BUS_E) INPUT_STREAM</td>
<td>Connected to BUS axi_dma_0_M_AXI_MM2S</td>
<td></td>
</tr>
<tr>
<td>ack</td>
<td>processing_system7_0:FCLK_CLK0</td>
<td>1</td>
</tr>
<tr>
<td>(BUS_E) OUTPUT_STREAM</td>
<td>Connected to BUS minuit_accel_core_top_0_OUTPUT_STREAM</td>
<td></td>
</tr>
<tr>
<td>ack</td>
<td>processing_system7_0:FCLK_CLK0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Figure 20**: FCLK_CLK0/FCLK_RESET0_N Ports for Zynq-7000 SoC PL Peripherals
10. Select Project > Design Rule Check. There should be no errors.
11. Close and exit XPS.
12. Create a top-level RTL module in PlanAhead. Because our embedded subsystem is really a "sub" system, we need a top-level RTL file. To generate one automatically, right-click on system in the Sources tab, and select Create Top HDL (Figure 22).

Figure 21: FCLK_CLK0/FCLK_RESET0_N Ports for Zynq-7000 SoC PS Peripherals
13. Generate a bitstream.

14. Export the hardware platform to XPS and launch SDK (Figure 23). Make sure Export Hardware is selected, and to make things easier, select Launch SDK as well. Enable Include Bitstream.

[Figure 22: Top HLD Creation in PlanAhead]

[Figure 23: Export to HW]
When SDK opens, we can start creating our software projects. To create a "Hello World" application, follow these steps (Figure 24):

1. Create a new project by selecting Xilinx C project.
2. Name it "matr_mult2".
3. For Hardware Platform, select ZC702_hw_platform.
5. Click Next.
6. Select hello world.
7. Click Finish.

This creates and builds the matr_mult2 standalone application. Because we use a terminal for output, it is time to start the terminal application (or use the built-in terminal in SDK) and configure it as shown in Figure 25.
To execute this application on the board, create a new Run Configuration by right-clicking **Project > Run As > Run Configurations**. In the GUI, select **Xilinx C/C++ ELF**, and click **New** (or double-click the entry), which generates a new configuration. Accept the defaults shown in **Figure 26**. Run the application. You should see "Hello World" appearing on the terminal.
The next step is to create the SW to call the HW accelerator. You can delete the `helloworld.c` file because we no longer need it.

To add the matrix multipliers files:

a. Right-click on `src` in the matrix_mult2 project.

b. Select `Import`.

c. Select `General`.

d. Select `File System` and click `Next`.

e. Select the include directory of the Vivado HLS generated pcore. This will be in `~work\matrix_mult\edk\matrix_mult\matrix_mult.srcs\sources_1\edk\system\pcores\mmult_accel_core_top_v1_00_a\include`. Pick up the three files generated automatically by Vivado HLS during the pcores generation phase, as shown in Figure 27.

f. Select the include directory of the Vivado HLS generated pcore. This will be in `~work\matrix_mult\arm_code`. Pick up the five files written, as shown in Figure 35.
To execute the application on the board, create a new Run Configuration by right-clicking on Project > Run As > Run Configurations. In the GUI, select Xilinx C/C++ ELF, and click New (or double-click the entry), which will generate a new configuration. Accept the defaults. Run the application. The terminal should resemble Figure 29.

Figure 28: Importing the files written for the ARM CPU

To execute the application on the board, create a new Run Configuration by right-clicking on Project > Run As > Run Configurations. In the GUI, select Xilinx C/C++ ELF, and click New (or double-click the entry), which will generate a new configuration. Accept the defaults. Run the application. The terminal should resemble Figure 29.
This section shows the C code application files that are written to initialize the DMA, instrument performance measurement and hardware accelerator invocation. All remaining files are automatically generated by XPS and Vivado HLS.

- main.c
- lib_xmmult_hw.c
- xtmrctr_low_level_example.c

```
#include <stdio.h>
#include "xil_io.h"
#include "platform.h"
#include "xparameters.h"
#include "xscugic.h"
#include "xaxidma.h"
#include "xmmult_accel_core.h"
#include "xil_printf.h"
#include "lib_xmmult_hw.h"
#include "xtmrctr.h"
#include "temp.h"

#define UPDATE_PACE (100000000)

// AXI DMA Instance
XAxiDma AxiDma;

// TIMER Instance
```
XTmrCtr timer_dev;

int init_dma()
{
    XAxiDma_Config *CfgPtr;
    int status;

    CfgPtr = XAxiDma_LookupConfig(XPAR_AXI_DMA_0_DEVICE_ID);
    if(!CfgPtr){
        print("Error looking for AXI DMA config\n\r");
        return XST_FAILURE;
    }
    status = XAxiDma_CfgInitialize(&AxiDma,CfgPtr);
    if(status != XST_SUCCESS){
        print("Error initializing DMA\n\r");
        return XST_FAILURE;
    }
    //check for scatter gather mode
    if(XAxiDma_HasSg(&AxiDma)){
        print("Error DMA configured in SG mode\n\r");
        return XST_FAILURE;
    }
    /* Disable interrupts, we use polling mode */
    XAxiDma_IntrEnable(&AxiDma,
    XAXIDMA_IRQ_ALL_MASK,XAXIDMA_DEVICE_TO_DMA);
    XAxiDma_IntrEnable(&AxiDma,
    XAXIDMA_IRQ_ALL_MASK,XAXIDMA_DMA_TO_DEVICE);

    return XST_SUCCESS;
}

int main()
{
    int i, j, k;
    int err=0;
    int status;
    float A[DIM][DIM];
    float B[DIM][DIM];
    float res_hw[DIM][DIM];
    float res_sw[DIM][DIM];

    unsigned int dma_size = SIZE * sizeof(float);

    float acc_factor;
    unsigned int init_time, curr_time, calibration;
    unsigned int begin_time;
    unsigned int end_time;
    unsigned int run_time_sw = 0;
    unsigned int run_time_hw = 0;

    init_platform();

    xil_printf("\r********************************************\n\r");
    xil_printf("\rVivado HLS: FP MATRIX MULT + DMA\n\r");

    // Init DMA
    status = init_dma();
    if(status != XST_SUCCESS){
        print("\rError: DMA init failed\n");
        return XST_FAILURE;
    }
    print("\rDMA Init done\n\r");

    //...
/* Initialize Timer */
status = TmrCtrLowLevelExample(XPAR_TMRCTR_0_BASEADDR, TIMER_COUNTER_0);
if (status != XST_SUCCESS) {
    return XST_FAILURE;
}

// Calibrate timer
init_time = XTmrCtr_GetTimerCounterReg(XPAR_TMRCTR_0_BASEADDR, TIMER_COUNTER_0);
curr_time = XTmrCtr_GetTimerCounterReg(XPAR_TMRCTR_0_BASEADDR, TIMER_COUNTER_0);

calibration = curr_time - init_time;

xil_printf("Calibration report:
init_time: %d cycles.
ncurr_time: %d cycles.
calibration: %d cycles.");

// Loop measurement
begin_time = XTmrCtr_GetTimerCounterReg(XPAR_TMRCTR_0_BASEADDR, TIMER_COUNTER_0);
for (i = 0; i < 1000; i++)
end_time = XTmrCtr_GetTimerCounterReg(XPAR_TMRCTR_0_BASEADDR, TIMER_COUNTER_0);
run_time_sw = end_time - begin_time - calibration;
xil_printf("Loop time for 100 iterations is %d cycles.");

// input data
/** Matrix Initiation */
for (i = 0; i < DIM; i++)
    for (j = 0; j < DIM; j++)
    {
        A[i][j] = (float)(i + j);
        B[i][j] = (float)(i * j);
    }

for (k = 0; k < 4; k++)
{
    // call the software version of the function
    begin_time = XTmrCtr_GetTimerCounterReg(XPAR_TMRCTR_0_BASEADDR, TIMER_COUNTER_0);
    matrix_multiply_ref(A, B, res_sw);
    end_time = XTmrCtr_GetTimerCounterReg(XPAR_TMRCTR_0_BASEADDR, TIMER_COUNTER_0);
    run_time_sw = end_time - begin_time - calibration;
    xil_printf("Total run time for SW on Processor is %d cycles.");
    run_time_sw;

    // Run the HW Accelerator
    Setup_HW_Accelerator(A, B, res_hw, dma_size);
    begin_time = XTmrCtr_GetTimerCounterReg(XPAR_TMRCTR_0_BASEADDR, TIMER_COUNTER_0);
    Run_HW_Accelerator(A, B, res_hw, dma_size);
    end_time = XTmrCtr_GetTimerCounterReg(XPAR_TMRCTR_0_BASEADDR, TIMER_COUNTER_0);
    run_time_hw = end_time - begin_time - calibration;
xil_printf( "\rTotal run time for AXI DMA + HW accelerator is %d cycles.\n\n", run_time_hw);

//Compare the results from sw and hw
for (i = 0; i < DIM; i++)
  for (j = 0; j < DIM; j++)
    if (res_sw[i][j] != res_hw[i][j]) { err = 1; }

if (err == 0)
  print("\rSW and HW results match!\n\n");
else
  print("\rERROR: results mismatch\n\n");

// HW vs. SW speedup factor
acc_factor = (float) run_time_sw / (float) run_time_hw;
xil_printf("\r033[1mAcceleration factor: %d.%d \033[0m\r\n\n", (int) acc_factor, (int) (acc_factor * 1000) % 1000);

print("\rEND\n\n");
XTmrCtr_Disable(XPAR_AXI_TIMER_0_BASEADDR, TIMER_COUNTER_0);
cleanup_platform();
return err;
}

lib_xmmult_hw.c

#include "platform.h"
#include "xparameters.h"
#include "xscugic.h"
#include "xaxidma.h"
#include "xmmult_accel_core.h"
#include "lib_xmmult_hw.h"
#include "xil_printf.h"

volatile static int RunExample = 0;
volatile static int ResultExample = 0;
XMmult_accel_core xmmult_dev;
XMmult_accel_core_Config xmmult_config = {
  0,
  XPAR_MMULT_ACCEL_CORE_TOP_0_S_AXI_CONTROL_BUS_BASEADDR
};

//Interrupt Controller Instance
XScuGic ScuGic;

// AXI DMA Instance
extern XAxiDma AxiDma;

int XMmultSetup(){
  return XMmult_accel_core_Initiaize(&xmmult_dev,&xmmult_config);
void XMmultStart(void *InstancePtr){
    XMmult_accel_core *pExample = (XMmult_accel_core *)InstancePtr;
    XMmult_accel_coreInterruptEnable(pExample,1);
    XMmult_accel_coreInterruptGlobalEnable(pExample);
    XMmult_accel_coreStart(pExample);
}

void XMmultIsr(void *InstancePtr){
    XMmult_accel_core *pExample = (XMmult_accel_core *)InstancePtr;

    //Disable the global interrupt
    XMmult_accel_coreInterruptGlobalDisable(pExample);
    //Disable the local interrupt
    XMmult_accel_coreInterruptDisable(pExample,0xffffffff);

    // clear the local interrupt
    XMmult_accel_coreInterruptClear(pExample,1);
    ResultExample = 1;
    // restart the core if it should run again
    if(RunExample){
        XMmultStart(pExample);
    }
}

int XMmultSetupInterrupt()
{
    //This function sets up the interrupt on the ARM
    int result;
    XScuGic_Config *pCfg =
    XScuGic_LookupConfig(XPAR_SCUGIC_SINGLE_DEVICE_ID);
    if (pCfg == NULL){
        print("Interrupt Configuration Lookup Failed\n\r");
        return XST_FAILURE;
    }
    result = XScuGic_CfgInitialize(&ScuGic,pCfg,pCfg->CpuBaseAddress);
    if(result != XST_SUCCESS){
        return result;
    }
    // self test
    result = XScuGic_SelfTest(&ScuGic);
    if(result != XST_SUCCESS){
        return result;
    }
    // Initialize the exception handler
    Xil_ExceptionInit();
    // Register the exception handler
    //print("Register the exception handler\n\r");
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
    (Xil_ExceptionHandler)XScuGic_InterruptHandler,&ScuGic);
    //Enable the exception handler
    Xil_ExceptionEnable();
    // Connect the Adder ISR to the exception table
    //print("Connect the Adder ISR to the Exception handler table\n\r");
    result =
    XScuGic_Connect(&ScuGic, XPAR_FABRIC_MMULT_ACCEL_CORE_TOP_0_INTERRUPT_INTR
    ,
    (Xil_InterruptHandler)XMmultIsr,&xmmult_dev);
    if(result != XST_SUCCESS){
        return result;
    }
}
C Code Running on the ARM CPU

XAPP1170 (v1.1) June 29, 2013  www.xilinx.com

//print("Enable the Adder ISR\n\r");
XScuGic_Enable(&ScuGic,XPAR_FABRIC_MMULT_ACCEL_CORE_TOP_0_INTERRUPT_INTR);
return XST_SUCCESS;

int Setup_HW_Accelerator(float A[DIM][DIM], float B[DIM][DIM], float res_hw[DIM][DIM], int dma_size)
//Setup the Vivado HLS Block
{
  int status = XMmultSetup();
  if(status != XST_SUCCESS){
    print("Error: example setup failed\n");
    return XST_FAILURE;
  }
  status = XMmultSetupInterrupt();
  if(status != XST_SUCCESS){
    print("Error: interrupt setup failed\n");
    return XST_FAILURE;
  }
  //XMmult_accel_core_SetVal1(&xmmult_dev,val1);
  //XMmult_accel_core_SetVal2(&xmmult_dev,val2);
  XMmultStart(&xmmult_dev);
  //flush the cache
  Xil_DCacheFlushRange((unsigned int)A,dma_size);
  Xil_DCacheFlushRange((unsigned int)B,dma_size);
  Xil_DCacheFlushRange((unsigned int)res_hw,dma_size);
  print("\rCache cleared\n\r");
  return 0;
}

void matrix_multiply_ref(float a[DIM][DIM], float b[DIM][DIM], float out[DIM][DIM])
{
  int ia, ib, id;
  // matrix multiplication of a A*B matrix
  for (ia = 0; ia < DIM; ++ia)
    for (ib = 0; ib < DIM; ++ib)
    {
      float sum = 0;
      for (id = 0; id < DIM; ++id)
        sum += a[ia][id] * b[id][ib];
      out[ia][ib] = sum;
    }
}

int Run_HW_Accelerator(float A[DIM][DIM], float B[DIM][DIM], float res_hw[DIM][DIM], int dma_size)
{
  //transfer A to the Vivado HLS block
int status = XAxiDma_SimpleTransfer(&AxiDma, (unsigned int) A, dma_size, XAXIDMA_DMA_TO_DEVICE);
if (status != XST_SUCCESS) {
    //print("Error: DMA transfer to Vivado HLS block failed\n");
    return XST_FAILURE;
}
/* Wait for transfer to be done */
while (XAxiDma_Busy(&AxiDma, XAXIDMA_DMA_TO_DEVICE)) ;

//transfer B to the Vivado HLS block
status = XAxiDma_SimpleTransfer(&AxiDma, (unsigned int) B, dma_size, XAXIDMA_DMA_TO_DEVICE);
if (status != XST_SUCCESS) {
    //print("Error: DMA transfer to Vivado HLS block failed\n");
    return XST_FAILURE;
}
/* Wait for transfer to be done */
while (XAxiDma_Busy(&AxiDma, XAXIDMA_DMA_TO_DEVICE)) ;

//get results from the Vivado HLS block
status = XAxiDma_SimpleTransfer(&AxiDma, (unsigned int) res_hw, dma_size, XAXIDMA_DEVICE_TO_DMA);
if (status != XST_SUCCESS) {
    //print("Error: DMA transfer from Vivado HLS block failed\n");
    return XST_FAILURE;
}
/* Wait for transfer to be done */
while (XAxiDma_Busy(&AxiDma, XAXIDMA_DMA_TO_DEVICE)) ;

/* Waiting for data processing */
/* While this wait operation, the following action would be done */
* First: Second matrix will be sent.
* After: Multiplication will be compute.
* Then: Output matrix will be sent from the accelerator to DDR and
* it will be stored at the base address that you set in the first
SimpleTransfer
*/
while (((XAxiDma_Busy(&AxiDma, XAXIDMA_DEVICE_TO_DMA)) ||
        (XAxiDma_Busy(&AxiDma, XAXIDMA_DMA_TO_DEVICE))) ) ;

return 0;
* @paramTmrCtrBaseAddress is the base address of the device.  
* @paramTmrCtrNumber is the timer counter of the device to operate on.  
* Each device may contain multiple timer counters.  
* The timer number is a zero based number with a range of  
* 0 - (XTC_DEVICE_TIMER_COUNT - 1).  
* @returnXST_SUCCESS to indicate success, else XST_FAILURE to indicate  
* a Failure.  
* @note  
* This function contains a loop which waits for the value of a timer counter  
* to change. If the hardware is not working correctly, this function may  
* not  
* return.  
* *************************************************************************  
***/  
int TmrCtrLowLevelExample(u32 TmrCtrBaseAddress, u8 TmrCtrNumber)  
{  
    u32 ControlStatus;  
    /*/  
* Clear the Control Status Register  
*/  
XTmrCtr_SetControlStatusReg(TmrCtrBaseAddress, TmrCtrNumber, 0x0);  
*/  
* Set the value that is loaded into the timer counter and cause it to  
* be loaded into the timer counter  
*/  
XTmrCtr_SetLoadReg(TmrCtrBaseAddress, TmrCtrNumber, 0x0);  
XTmrCtr_LoadTimerCounterReg(TmrCtrBaseAddress, TmrCtrNumber);  
*/  
* Clear the Load Timer bit in the Control Status Register  
*/  
ControlStatus = XTMTCr_GetControlStatusReg(TmrCtrBaseAddress,  
    TmrCtrNumber);  
ControlStatus = ControlStatus | XTC_CSR_AUTO_RELOAD_MASK;  
ControlStatus = ControlStatus & (~XTC_CSR_LOAD_MASK);  
XTmrCtr_SetControlStatusReg(TmrCtrBaseAddress, TmrCtrNumber,  
    ControlStatus);  
*/  
* Get a snapshot of the timer counter value before it's started  
* to compare against later  
*/  
//Value1 = XTMTCr_GetTimerCounterReg(TmrCtrBaseAddress, TmrCtrNumber);  
*/  
* Start the timer counter such that it's incrementing by default  
*/  
XTmrCtr_Enable(TmrCtrBaseAddress, TmrCtrNumber);  
*/  
* Read the value of the timer counter and wait for it to change,  
* since it's incrementing it should change, if the hardware is not  
* working for some reason, this loop could be infinite such that the  

Floating-point designs written in C or C++ can be quickly and easily implemented on FPGA devices. Implementing designs in this way takes advantage of Xilinx FPGA's parallel performance, low power, embedded CPUs and low cost. As with other C/C++ flows, a full and complete tool chain allows performance trade-offs to be made throughout the flow and comprehensive analysis. The driver application is a 32x32 matrix multiplication core optimized for 32-bit floating point accuracy using Vivado HLS design.

The floating-point matrix multiplication modeled in the C/C++ code can be quickly implemented and optimized into an RTL design using Vivado HLS. It can then be exported as a pcore that is connected with AXI4-Stream interface to the ACP of the Zynq-7000 AP SoC PS through a DMA core in the Programmable Logic subsystem of the Zynq-7000 device. The matrix multiplier HW peripheral is computed in almost 20 times less clock cycles than its SW execution on the ARM CPU.

References

2. Floating-Point PID Controller Design with Vivado HLS and System Generator for DSP (XAPP1163).
5. AXI Reference Guide (UG761)
MERCHANDABILITY, 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 the Limited Warranties which can be viewed at http://www.xilinx.com/warranty.htm; 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 Critical Applications: http://www.xilinx.com/warranty.htm#critapps.