XAPP1170 (v2.0) January 21, 2016

A Zynq Accelerator for Floating Point Matrix Multiplication Designed with Vivado HLS (Version Cora Z7)

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

Summary

This application note describes how to use Vivado® High Level Synthesis (HLS) to develop a floating-point matrix multiplication accelerator connected via an AXI4-Stream interface to the Accelerator Coherency Port (ACP) of the ARM CPU in the Zynq®-7000 All Programmable SoC (AP SoC) device.

The floating-point matrix multiplication accelerator modeled in C/C++ code can be quickly implemented and optimized into a Register Transfer Level (RTL) design using Vivado HLS. The solution is then exported as an IP core connected with an automatically-created AXI4-Stream interface to the ACP on AP SoC Processing Subsystem (PS). The connection is made through a Direct Memory Access (DMA) core in the AP SoC Programmable Logic (PL) subsystem. Vivado IP Integrator (IPI) is used to design the AP SoC PL hardware, including the matrix multiplier peripheral, the DMA engine, and an AXI timer. The Software Development Kit (SDK) is used to design the AP SoC PS software to manage the peripherals.

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

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 Zynq-7000 AP SoC [Ref 1]. This delivers cost, performance, and power benefits for designers relying on traditional micro-processors to implement floating-point algorithms [Ref 2] [Ref 3].

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

1. Compiling and optimizing the C/C++ floating-point design into a high-performance hardware accelerator using Vivado HLS.
2. Specifying and generating an AXI4-Stream interface for the hardware accelerator using C++ templates in Vivado HLS.

3. Using Vivado IP Integrator [Ref 4] to connect the hardware accelerator to an AXI DMA peripheral in the AP SoC PL and to the ACP in the AP SoC PS.
4. Writing the software running on the ARM CPU with function calls to the hardware accelerator and measuring system level performance.

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

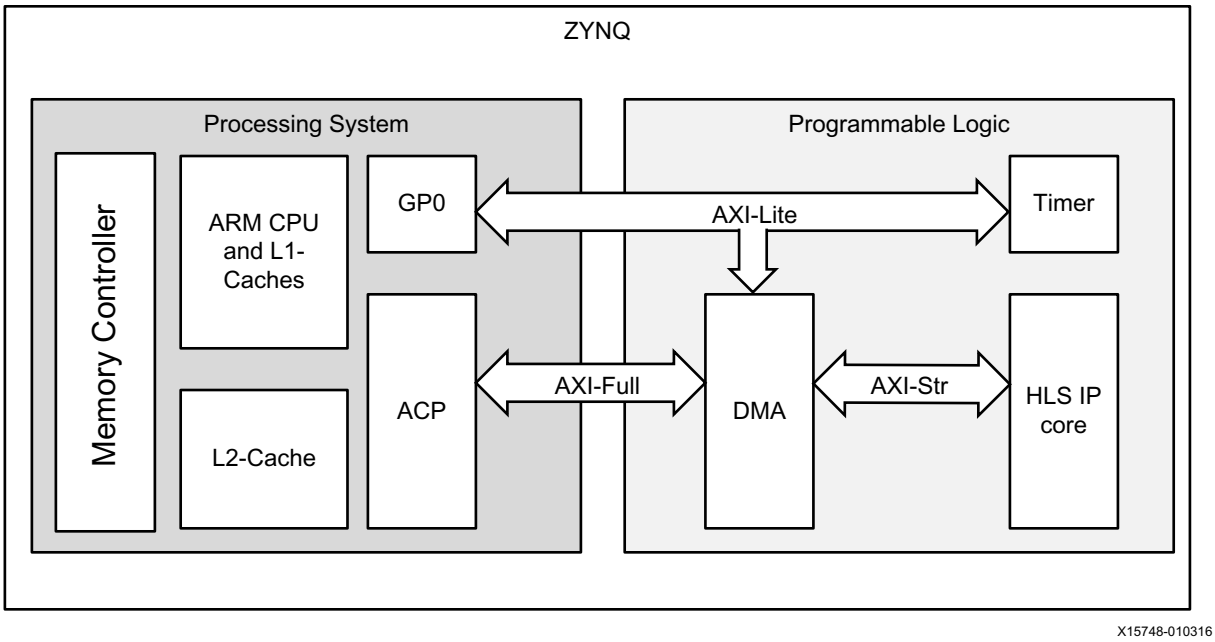


Figure 1: **PS and PL Partitions in the Zynq-7000 AP SoC**

The design procedure described in this document applies to Vivado 2015.4 IDE release tools, targeting the Zynq-7000 AP SoC Evaluation Kit (ZC702) [Ref 5].
Dans le laboratoire vous utiliserez un Cora Z7, et la puce possède moins de ressource (xc7z007sclg400-1)

Matrix Multiply Design with Vivado HLS

The matrix multiplication algorithm A*B=C is very simple. There are three nested loops:

- The first loop (L1) iterates over the elements composing 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:

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

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) which can be applied 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 HLS User Guide [Ref 6].

The following TCL code specifies the clock period and target device:

```
set_part {xc7z007sclg400-1}
create_clock -period 10
```

Given the code of the mmult_hw function, Vivado HLS:

- Transforms each of the operations in the C code into an equivalent hardware operation and schedules those operations into clock cycles. Using knowledge of the clock period and device delays, it places as many operations as possible into a single clock cycle.
- Uses 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 automatically constructs an interface to access a RAM block (other I/O interface options can be specified).
- Maps each of the hardware operations onto an equivalent hardware unit in the AP SoC.
- Performs any user specified optimizations, such as pipelined or concurrent operations.
- Outputs the final design, with reports, in Verilog and VHDL for implementation in the 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 takes 329,793 clocks cycle to calculate the result using the specified target technology and clock period. This design could execute with a maximum clock period of 8.41 ns.

The area estimates in Figure 2 show how many resources on the PL the design is expected to use: 5 DSP48 slices, about 473 FFs (Flip-Flops) and 830 LUTs (Look-Up Tables).

Synthesis(solution1)

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.41	1.25

Latency (clock cycles)

Summary

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

Utilization Estimates

Summary

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

X15749-010416

Attention
la Cora Z7
va peut être donner
des résultats
différents

Figure 2: **Initial Performance Characteristics in 32-bit Floating Point**

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 might be other gate-level optimizations that impact the final results.

Figure 3 shows the C function arguments transformed by interface synthesis into I/O ports. This process enables the ports to be connected to other blocks in the completed embedded design.

Interface

Summary

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

X15750-010316

Figure 3: **Initial RTL Ports**

Changes implemented during this step include:

- A clock and reset signals were added to the design (ap_clk, ap_rst).
- A design-level protocol was added to the design. This is the default, but is also optional. This allows the design to be started (ap_start) and indicates when it is ready for new inputs, has complete (ap_done), or is idle.
- Array arguments were transformed into 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 the port din uses a dual-port block BRAM (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 I/O 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.

Optimized RTL

The initial design created by Vivado HLS can be optimized. Figure 4 shows the comparison of three possible solutions. The optimizations were applied to reduce the amount of clock cycles needed to compute the matrix multiplication. For additional details on the optimizations provided by Vivado HLS, see the Vivado HLS Tutorial [Ref 7].

Performance Estimates

Timing (ns)

Clock		solution1	solution2	solution3
ap_clk	Target	10.00	10.00	10.00
	Estimated	8.41	9.35	8.41

Latency (clock cycles)

		solution1	solution2	solution3
Latency	min	329793	16535	1190
	max	329793	16535	1190
Interval	min	329794	16536	1191
	max	329794	16536	1191

Utilization Estimates

	solution1	solution2	solution3
BRAM_18K	0	0	0
DSP48E	5	10	160
FF	473	2312	13420
LUT	830	3450	23293

X15751-010316

Attention
la Cora Z7
va peut être donner
des résultats
différents

Figure 4: **Performance Estimates Comparison for Three Solutions**

Solution2 is about 20 times faster than the initial design (solution1) at the expense of more resources: 10 DSP48 slices, 2,312 FFs and 3,450 LUTs. The estimated clock period of 9.35ns means the output data rate is 6.46 KSPS (Kilo Samples Per Second), as shown in Equation 1

$$16536 \times 9.35\ \text{ns} = 0.154\ \text{ms} = 1 / (6.46\ \text{KSPS}) \qquad \textit{Equation 1}$$

Clearly the highest performance is achieved by solution3, with only 1,190 clock cycles necessary to compute the floating point matrix multiplication. This number is obtained by using 160 DSP48 slices, 13,420 FFs and 23,293 LUTs, which represent, respectively, the 72%, 12%, and 43%, utilization of the available resources on the AP SoC. Solution3 exhibits a Pipeline Initialization Interval of 1, which means a throughput of 1. For this example the data rate is 118.9 MSPS (millions of samples per second) and the whole output matrix is generated in the time period of 1,190 x 8.41 ns, that is 10 µs.

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 PL resources, you can select, for example, solution2.

The following TCL code shows the optimization directives of solution3.

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

AXI4-Stream Interface with Vivado HLS

AXI4-Stream is a communication standard for point-to-point data transfer without the need for addressing, or external bus masters [Ref 8]. 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 handled by the AXI DMA IP.

The architecture of our system, illustrated in Figure 1, applies the ACP port to connect the AXI DMA to the L2 cache of the ARM processor – an alternative approach is to use the High Performance (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 a lot 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, page 2, and add some additional functions to be synthesized, as illustrated by the new C++ code shown in this section. In particular, pop_stream and push_stream are functions, respectively, 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 AXI4 protocol.

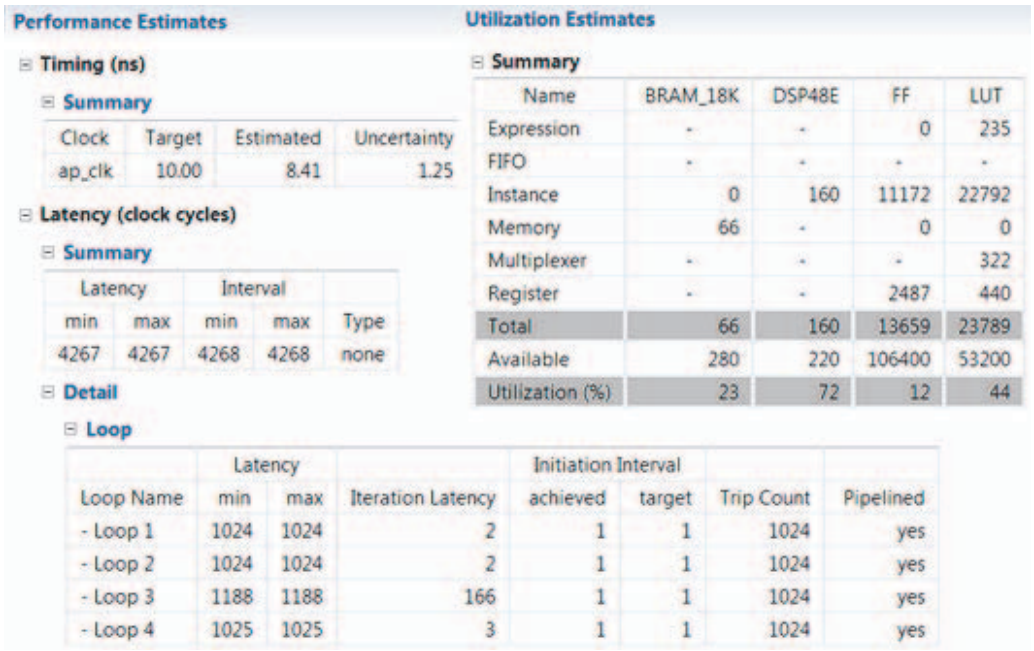
The following code shows the usage of the AXI_VAL data type. This is a user-defined data type that expresses the side channel information associated with the AXI4-Stream interface. In Vivado HLS, any side channel information that is not part of the protocol handshake must be expressed in the C/C++ code and used in some way. This means that while Vivado HLS abstracts the TREADY and TVALID signals, all other signals in the AXI4-Stream interface must be part of the user code. In addition, aside from the TDATA, TREADY, and 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.

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

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

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

Figure 5 shows the synthesis report of the AXI4-Stream matrix multiplier. Note that the latency now is 4,267 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 of the hardware function. The time to transfer each matrix is 1,024 clock cycles for 1,024 32-bit floating point values, therefore the total time is 3,072 clock cycles. The computation time for the matrix multiplication is 1,188 clock cycles, plus an additional two cycles for the pop_stream and push_stream functions. Few additional clock cycle are consumed for the FOR loop prologue and epilogue and for the start up of the function. This results in a function initialization interval (II) of 4,268 clock cycles with a latency of 4,267 clock cycles.



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.41	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
4267	4267	4268	4268	none

Detail

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Loop 1	1024	1024	2	1	1	1024	yes
- Loop 2	1024	1024	2	1	1	1024	yes
- Loop 3	1188	1188	166	1	1	1024	yes
- Loop 4	1025	1025	3	1	1	1024	yes

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	235
FIFO	-	-	-	-
Instance	0	160	11172	22792
Memory	66	-	0	0
Multiplexer	-	-	-	322
Register	-	-	2487	440
Total	66	160	13659	23789
Available	280	220	106400	53200
Utilization (%)	23	72	12	44

Attention
la Cora Z7
va peut être donner
des résultats
différents

X15752-010316

Figure 5: **Synthesis Estimation of the AXI4-Stream Matrix Multiplier Core**

The overall estimated resources utilization is therefore 66 BRAM18K (Block RAM units of 18 Kbit capacity), 160 DSP48 slices, 13,659 FFs, and 23,789 LUTs, which represent, respectively, the 23%, 72%, 12%, and 44% utilization of the available resources on the Zynq-7000 device. The data rate remains unchanged from 118.9 MSPS and the whole output matrix is generated in the time period of 4,268 x 8.41 ns, that is, 36 µs.

The Vivado HLS project is created by running the provided TCL script in the design archive from the Vivado HLS Command Prompt shell, as illustrated in Figure 6, with the following command:

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

Ici on va utiliser un fichier script Tcl/Tk pour faire la synthèse. Prenez le temps d'examiner le fichier de commandes. Vous remarquerezen autre à la ligne 52 la commande: export_design -evaluate verilog -format ip_catalog

Cette dernière va permettre de générer dans un format (IPXACT) le coprocesseur qui se nomme Hls_accel (solution 3) que vous pourrez instancier plus tard dans Vivado. On dit que Hls_accel est mis dans un catalogue IP (Intellectual Property). Cette commande peut prendre du temps car elle synthétise complètement le coprocesseur.

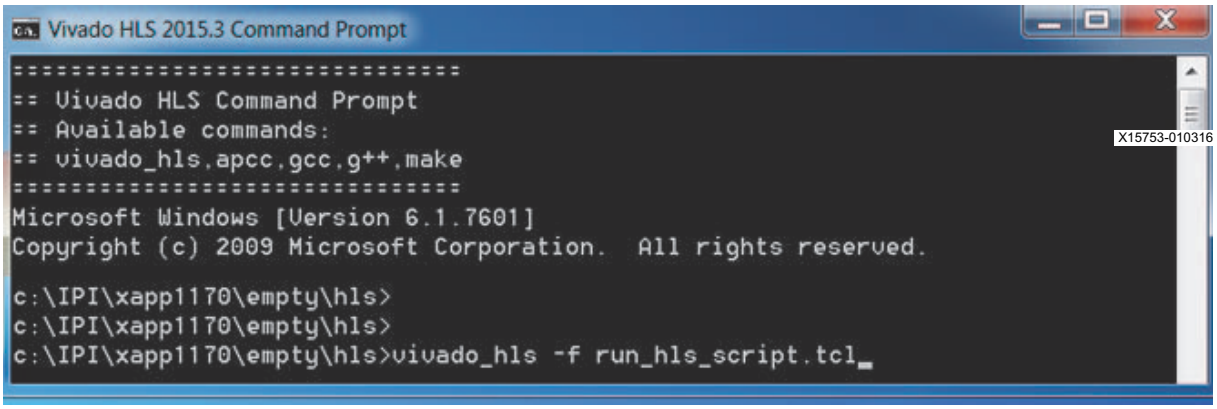


Figure 6: **Launching the TCL Script to Create the Vivado HLS Project**

Further system level improvements could be achieved by using the 64-bit AXI4-Stream interface, which would reduce the accelerator latency from 4,267 cycles to approximately 2,700 cycles (loops L1, L2, and L4, shown in Figure 4, would take half cycles).

AXI DMA Overview

The AXI DMA core [Ref 9] provides high-bandwidth direct memory access between memory and peripherals via an AXI4-Stream interface. The core design has the following AXI4 interfaces:

- AXI4-Lite Slave.
- AXI4 Memory Map Read Master/Write Master
- Optional AXI4 Memory Map 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 7 illustrates the AXI DMA interfaces and the data streaming. The AXI DMA works in two different modes, but only one at a time: Scatter/Gather and Simple DMA modes. For each mode there is a proper register to be configured with the AXI4-Lite slave interface. In our implementation we are using the Simple DMA mode.

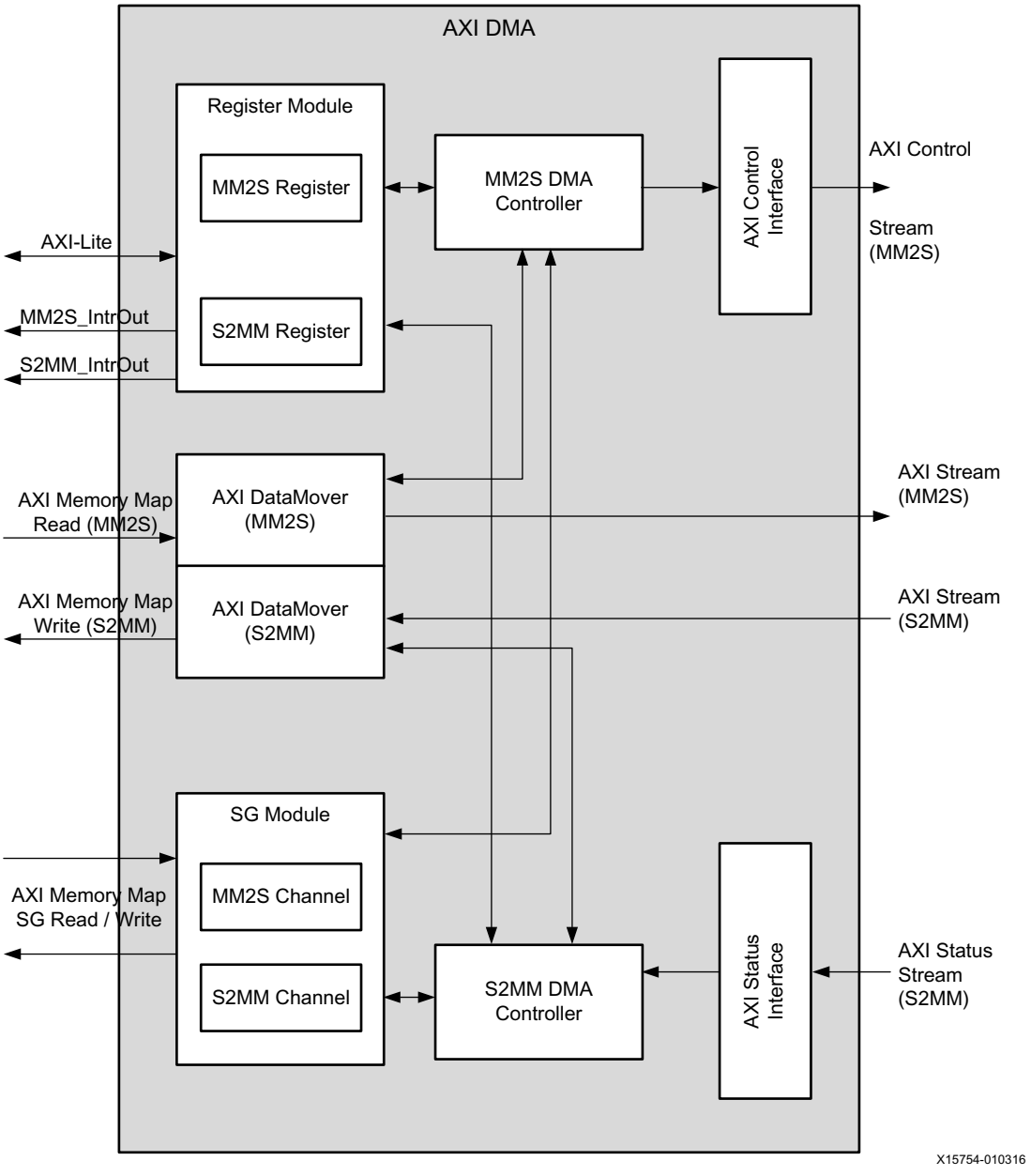


Figure 7: **AXI DMA Core Block Diagram**

The Simple DMA mode provides a configuration for doing simple DMA transfers on MM2S and S2MM channels that require less FPGA resource utilization. The AXI4 Read (MM2S) interface reads the data from a master external memory, then the DMA Data Mover transmits that 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 via the AXI4-Stream (S2MM) port.

DMA transfers are initiated by accessing control, source, or destination address and length registers. The MM2S channel setup sequence is as follows:

1. The MM2S channel run is initiated by setting the run/stop bit in the control register.
2. A valid source address is written to the MM2S Source address register.
3. The number of bytes to transfer is written to the MM2S Length register. The Length register must be written last.

The MM2S channel setup sequence is as follows:

1. The S2MM channel run is initiated by setting the run/stop bit in the control register.
2. A valid destination address is written to the S2MM Destination address register.
3. The length in bytes of the receive buffer is written to the S2MM Length register. The Length register must be written last.

ARM ACP Overview

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 AP 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 as done by the CPU processors 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 it misses in the L1 cache, then there is also the opportunity to hit in 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 8 illustrates the connectivity between ACP and the memory system connected to the CPU.

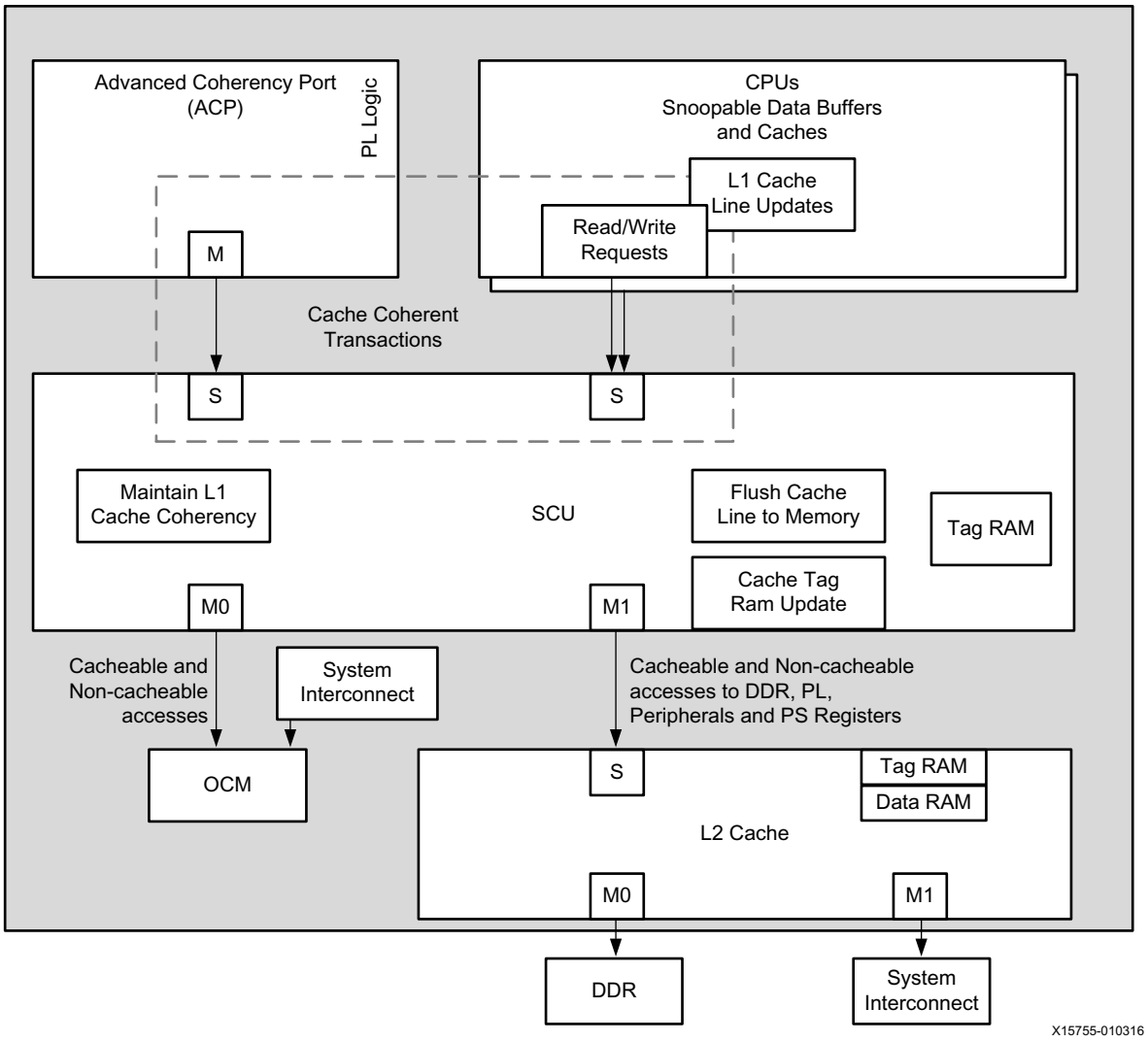


Figure 8: **ARM Memories and ACP Connections**

Vivado IP Integrator design

This section describes the steps to create the hardware design with Xilinx Vivado 2018.3 targeting the Zedboard.

Create a basic Vivado project (project_1) by selecting all of the default settings. When prompted for the part, select the ZC702 board. These are the detailed instructions:

1. Launch Vivado. Create a new project in your working directory
On revient ici au tutoriel de Vivado Lab 1, Partie 2):

   (`xapp1170/empty/vivado`) and select:

   - RTL Project
   - No sources, IP, or constraints (for now)

2. In the **Default Part** select **Boards** > **Zedboard Zynq Evaluation and Development Kit dans Boards**.

3. Click **Next** and then **Finish**. The Vivado GUI opens with an empty project.

Ici on va indiquer à Vivado à quel endroit aller chercher le IP (Hls_accel) que nous avons créée en haut de la page 10. En anglais, Xilinx nomme cette opération:

Add the HLS IP to an IP repository:

1. Tapez Tools - >Settings
2. Dans la fenêtre de droit tapez sur >IP - > Repository
3. Tapez sur le + en dessous de IP Repositories et ajoutez le chemin du path où se trouve votre IP qui a été généré à l'étape synthèse (voir Figure 9).

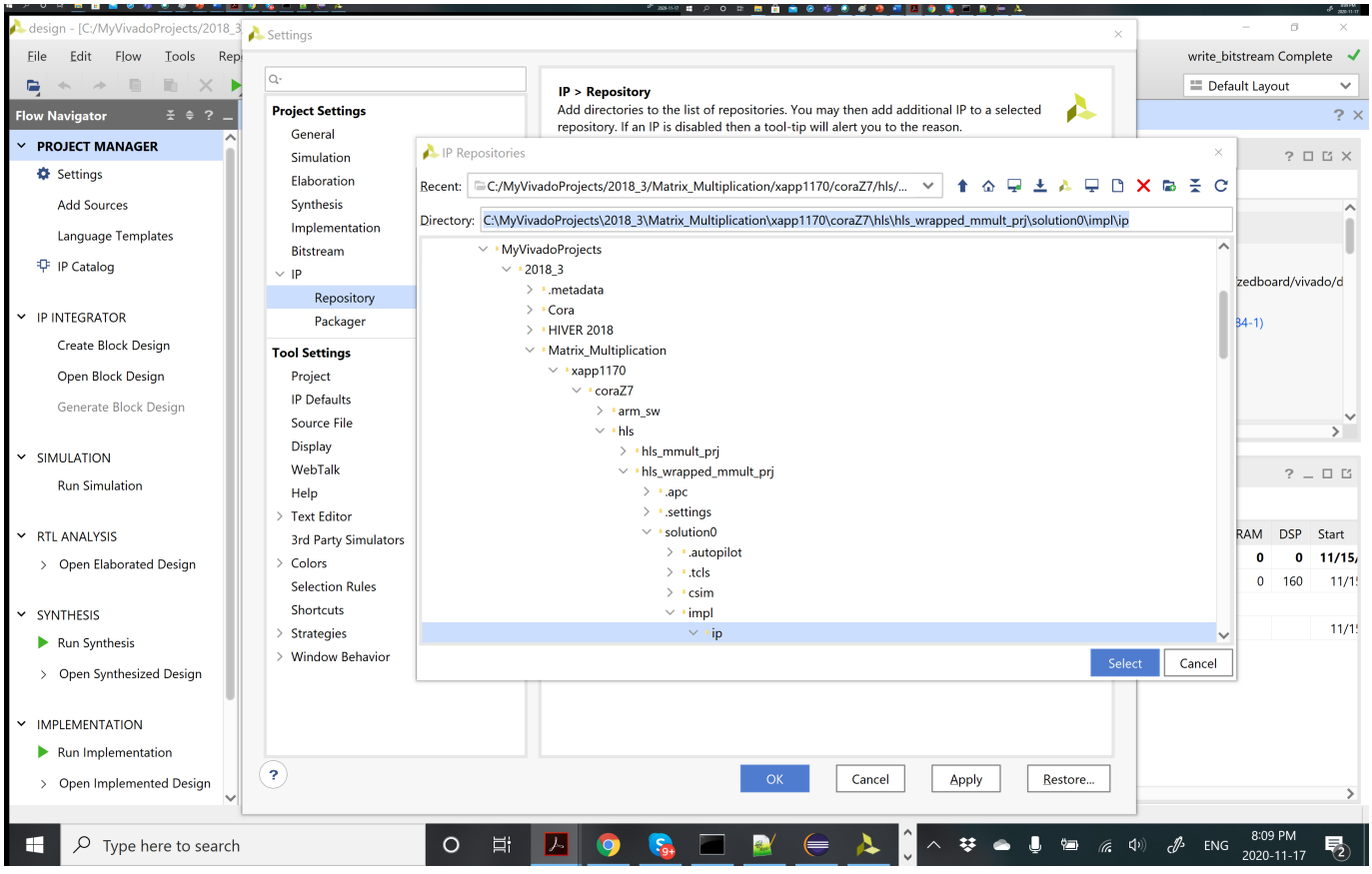


Figure 9

Retenez pour l'instant que l'IP se nomme Hls_accel et sera accéssible lorsque plus loin nous allons créer le Block Design du système à la section suivante.

Start to design the block diagram in IP Integrator:

Ici on va créer le design proprement dit comme vous avez fait dans le tutorial de Vivado

1. Click **Create Block Design** under IP Integrator in the Flow Navigator.
   a. In the resulting dialog, name the design `system` (see Figure 10).
2.

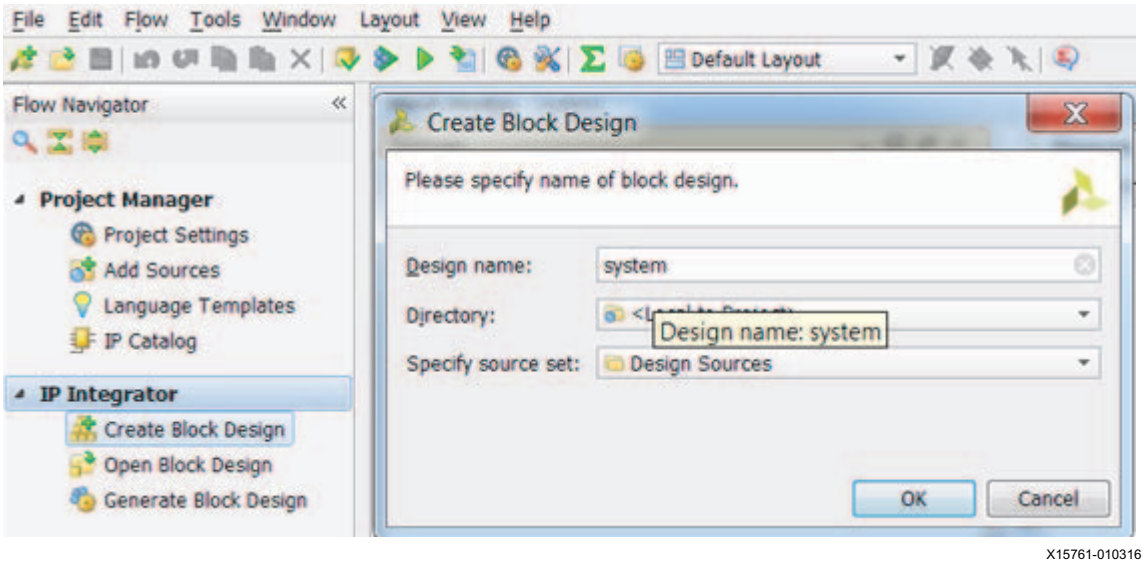


Figure 10: **Create Block Design**

   b. Click **OK**.
   c. The upper-right pane now has an empty **Diagram** tab, as shown in Figure 11.

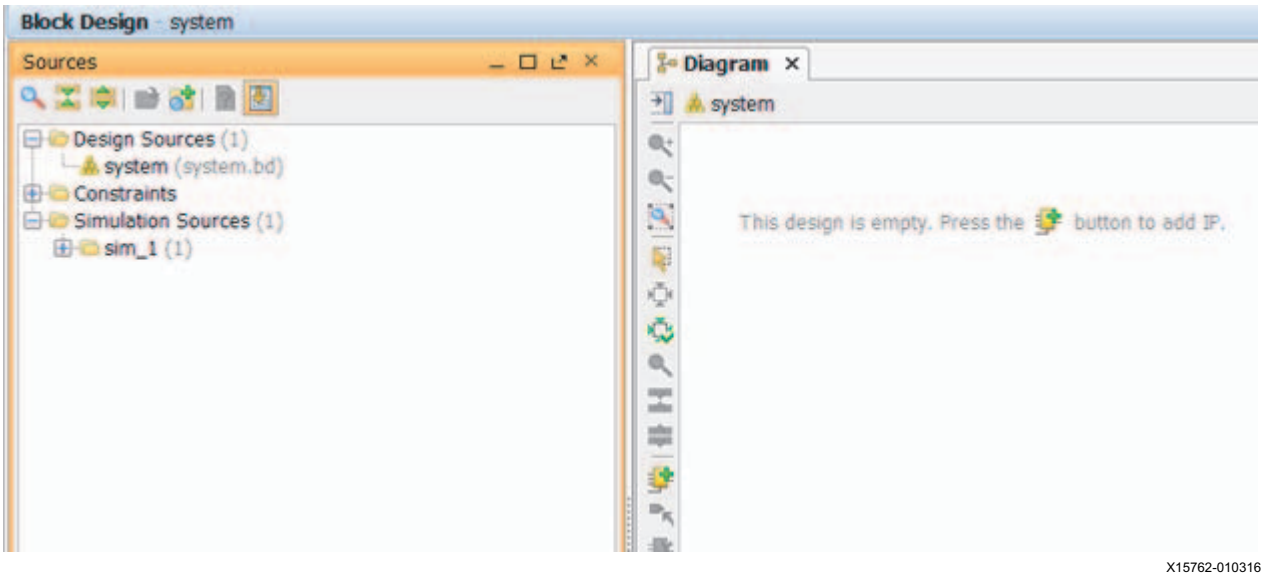


Figure 11: **Diagram Pane**

2. A script is provided to automatically build the block design. In the **TCL Console** enter `source mmult_system_coraZ7.tcl` and press the Enter key.
   Au labo 1, vous avez vous même créer le design à la main (fittimer, GPIO, etc.). Ici on va le faire en utilisant un fichier script Tcl/Tk. Prenez le temps d'examiner ce fichier de commandes.
3. When the script finishes, right-click **Regenerate Layout** in the Diagram pane. You should see the block design, as shown in Figure 12.
   N.B. Notez à la Figure 10, que l'accélérateur Hls_accel mis dans le IP_repository a été instancié car il fesait partie des appels dans le script. On aurait pu l'instantier à la main (comme tous les autres composants d'ailleurs) en tapant le bouton + et en tapant ensuite Hls_accel.

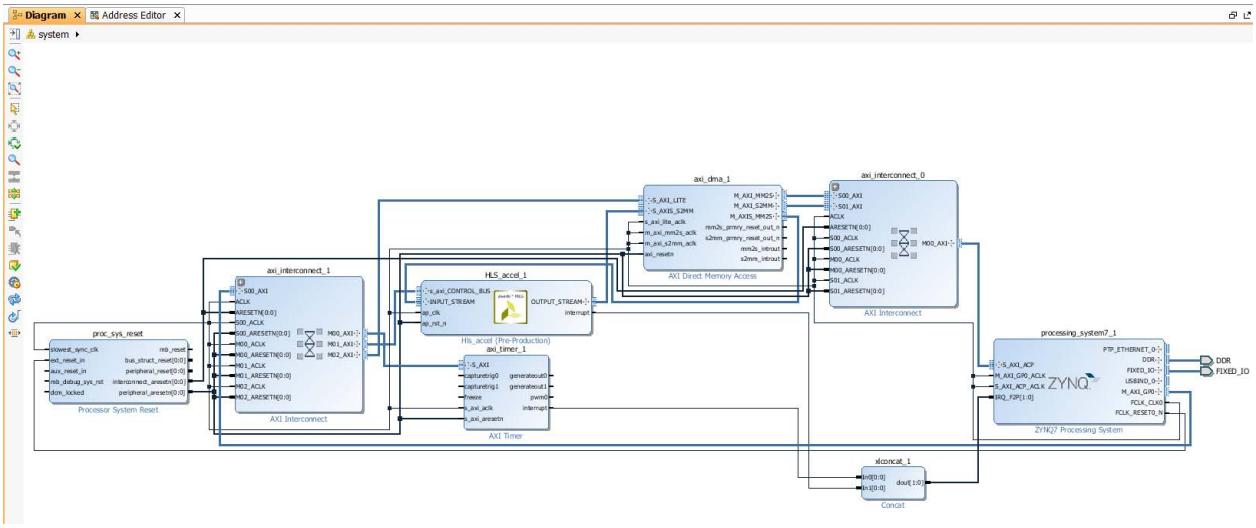


Figure 12: **Block Design**

Avant d'aller à l'étape 4 Ø2ÌÖÌ ljdžŜfljdžes explications:

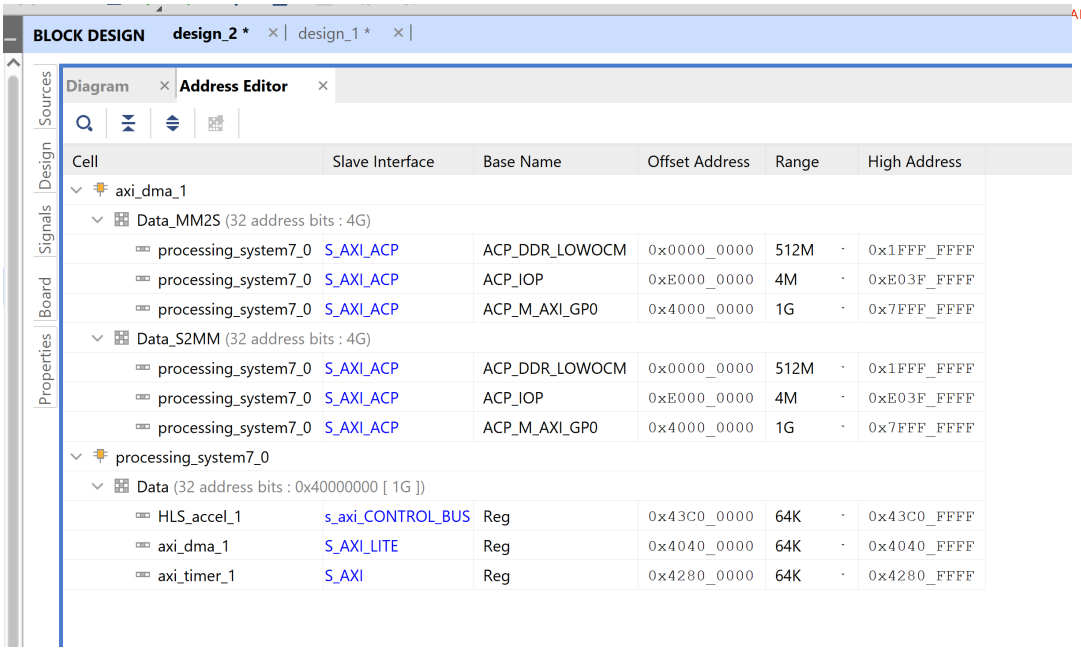
Dans un premier temps prenez le temps d'examiner certains composants et signaux composants du design (nous compléterons l'analyse en classe):

Remarque: pour l'instant ne vous préocuppez pas de la lettre M (pour Master) ou S (pour Slave) ajoutée devant plusieurs nom de port de composants. Par exemple: M_AXI_MM2S (port de axi_dma) devrait se lire simplement comme Memory Mapped 2 Stream alors que S_AXI_S2MM. devrait se lire simplement AXI Stream 2 Mapped Memory.

Notez que Hls_accel (votre coporcesseur) lit les données de INPUT_STREAM qui lui sont apportées par le DMA via son port M_AXI_MM2S. DMA va les lire dans l'interface ACP du ARM (Figure 8). Le DMA passe alors par le axi_interconnect_0 via les ports SS0_AXI et M00_AXI.

À l'opposé Hls_accel écrit les données dans OUT_STREAM via son port S_AXI_S2MM.DMA va les écrire dans l'interface ACP du ARM (Figure 8). Le DMA passe alors par le axi_interconnect_0 via les ports SS1_AXI et M00_AXI.

Tapez maintenant sur Adress Editor (en bas de l'onglet BLOCK DESIGN). Vous deviez voir la figure 13.



Cell	Slave Interface	Base Name	Offset Address	Range	High Address
axi_dma_1					
Data_MM2S (32 address bits : 4G)					
processing_system7_0	S_AXI_ACP	ACP_DDR_LOWOCM	0x0000_0000	512M	0x1FFF_FFFF
processing_system7_0	S_AXI_ACP	ACP_IOP	0xE000_0000	4M	0xE03F_FFFF
processing_system7_0	S_AXI_ACP	ACP_M_AXI_GP0	0x4000_0000	1G	0x7FFF_FFFF
Data_S2MM (32 address bits : 4G)					
processing_system7_0	S_AXI_ACP	ACP_DDR_LOWOCM	0x0000_0000	512M	0x1FFF_FFFF
processing_system7_0	S_AXI_ACP	ACP_IOP	0xE000_0000	4M	0xE03F_FFFF
processing_system7_0	S_AXI_ACP	ACP_M_AXI_GP0	0x4000_0000	1G	0x7FFF_FFFF
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
HLS_accel_1	s_axi_CONTROL_BUS	Reg	0x43C0_0000	64K	0x43C0_FFFF
axi_dma_1	S_AXI_LITE	Reg	0x4040_0000	64K	0x4040_FFFF
axi_timer_1	S_AXI	Reg	0x4280_0000	64K	0x4280_FFFF

Figure 13

On a ici une description du memory map de notre système.

En haut on voit le Data MM2S (4G addressable). Ce qui nous intéresse ici c'est le 1/2 G de DDR nommé ACP_DDR_LOWOCM (0x0000_0000 à 0x1FFFF FFFF) qui servira aux lectures de Hls_accel.

Au milieu le Data S2MM (4G addressable). Ce qui nous intéresse ici c'est le 1G de DDR nommé ACP_DDR_LOWOCM (0x0000_0000 à 0x1FFFF FFFF) qui servira aux écritures de Hls_accel.

Notez que sur la Zedboard on a 1 Gig de DDR donc des adresses allant de 0x0000_0000 à 3FFF_FFFF.

En bas, le Data représente 3 ranges de 64K qui sont utilisés pour initialiser le DMA. Encore une fois nous verrons en classe le détail de ce memory map en classe.

En terminant remarquez l'utilisation d'un timer qui va nous servir dans SDK pour calculer le temps à Hls_accel de faire une multiplication de matrice. Notez aussi que le DMA communique avec le coprocesseur via une interruption. Vous avez donc deux interruptions possible qui rentrent sur le ARM via xlconc.

4. Tapez la commande Validate Design avec le bouton de droite n'importe où dans le design. Vous serez ainsi aviser si il manque ou s'il y a de mauvaises connexions.

5. À partir d'ici il ne vous reste qu'à compléter avec Create HDL Wrapper, Genreate Bitstream, Export Hardware et Launch SDK (étapes 17 à 21 du tutoriel de Vivado Lab 1).

X15766-010316

The Zynq Software Design with SDK

Click **Launch SDK**. When the SDK opens, software projects can be started. Follow these steps to create a *Hello World* application:

1. Create a new project by selecting **File** > **New** > **Application Project** and name it `mmult` (see Figure 14).

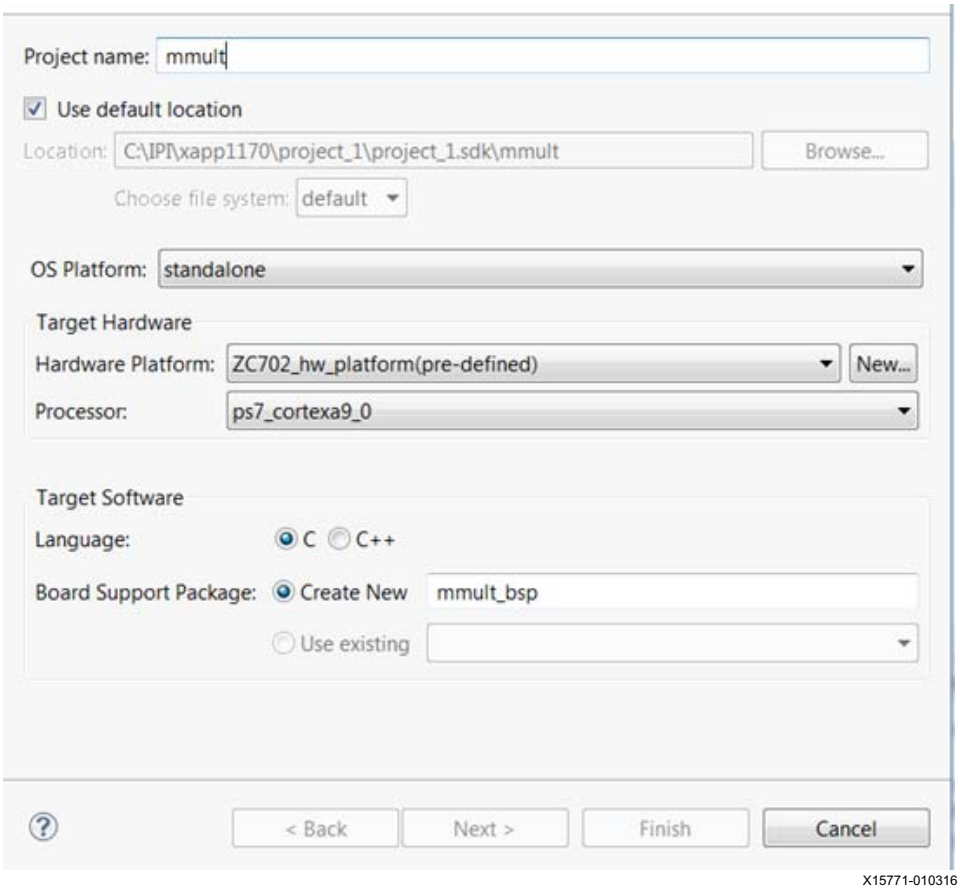


Figure 14: **SDK New Project Wizard**

2. Click **Board Support Package: Create New**.
3. Click **Next**.
4. Select **Hello World**.
5. Click **Finish**. This creates and builds the `mmult` standalone application.

6. Because a terminal is used for the output, start the terminal application (or use the built-in terminal in SDK) and configure it as shown in Figure 15.

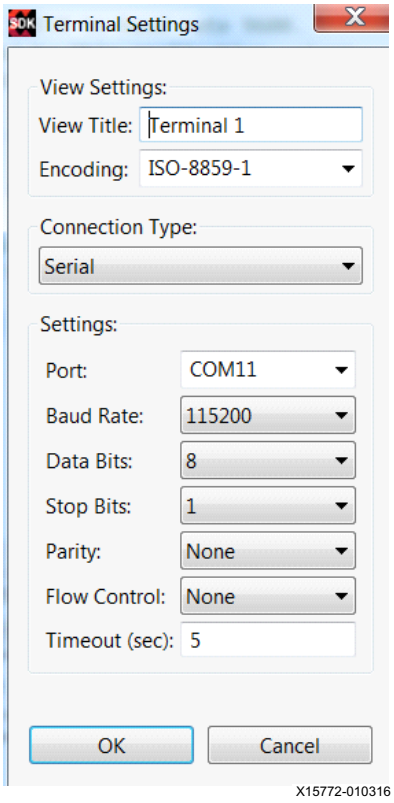


Figure 15: **UART Terminal Configuration Settings**

7. Program the Zynq-7000 device by right-clicking **Xilinx Tools** > **Program FPGA**.

8. Set the **Build Configurations** to **Release** mode, as shown in Figure 16.

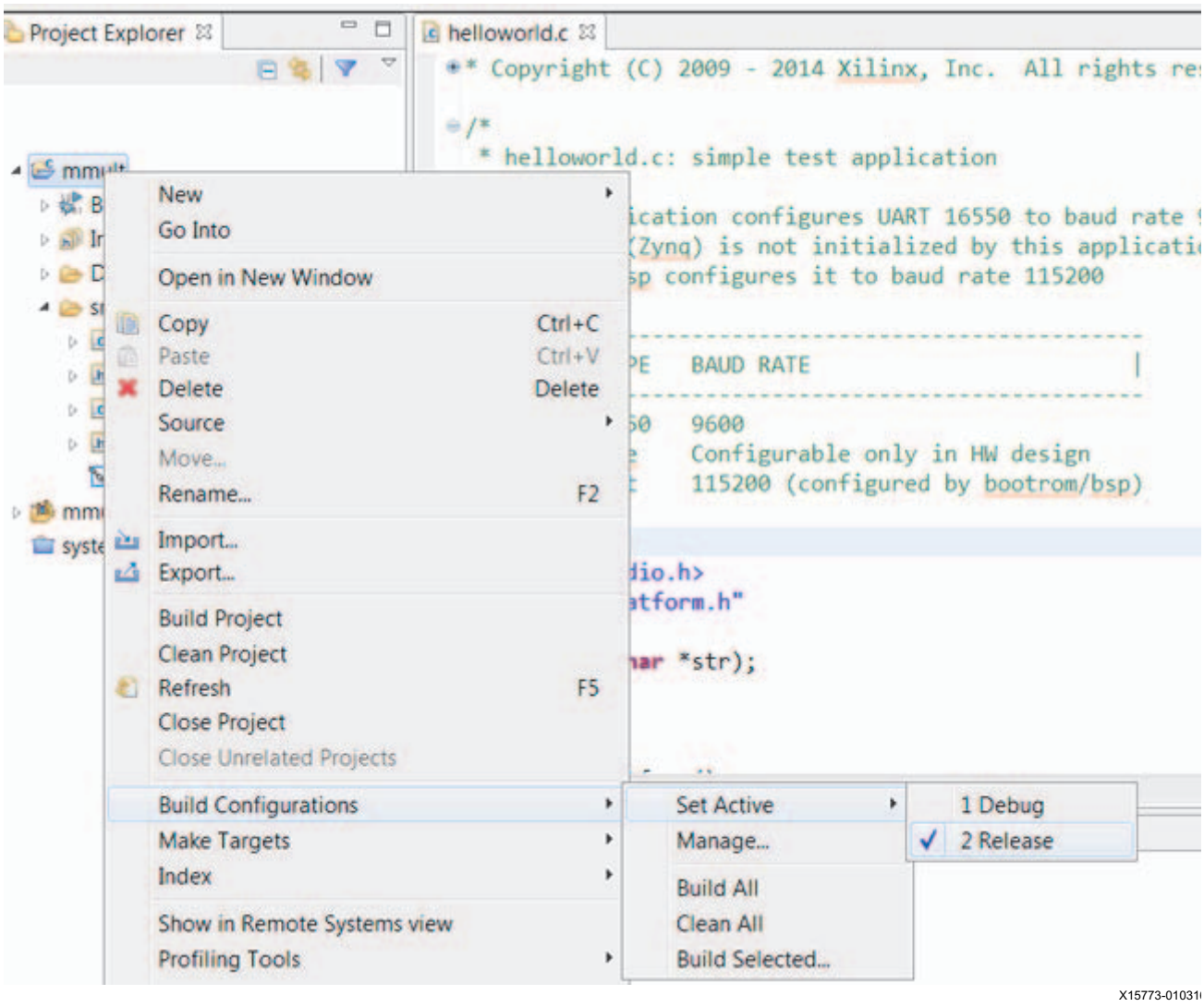
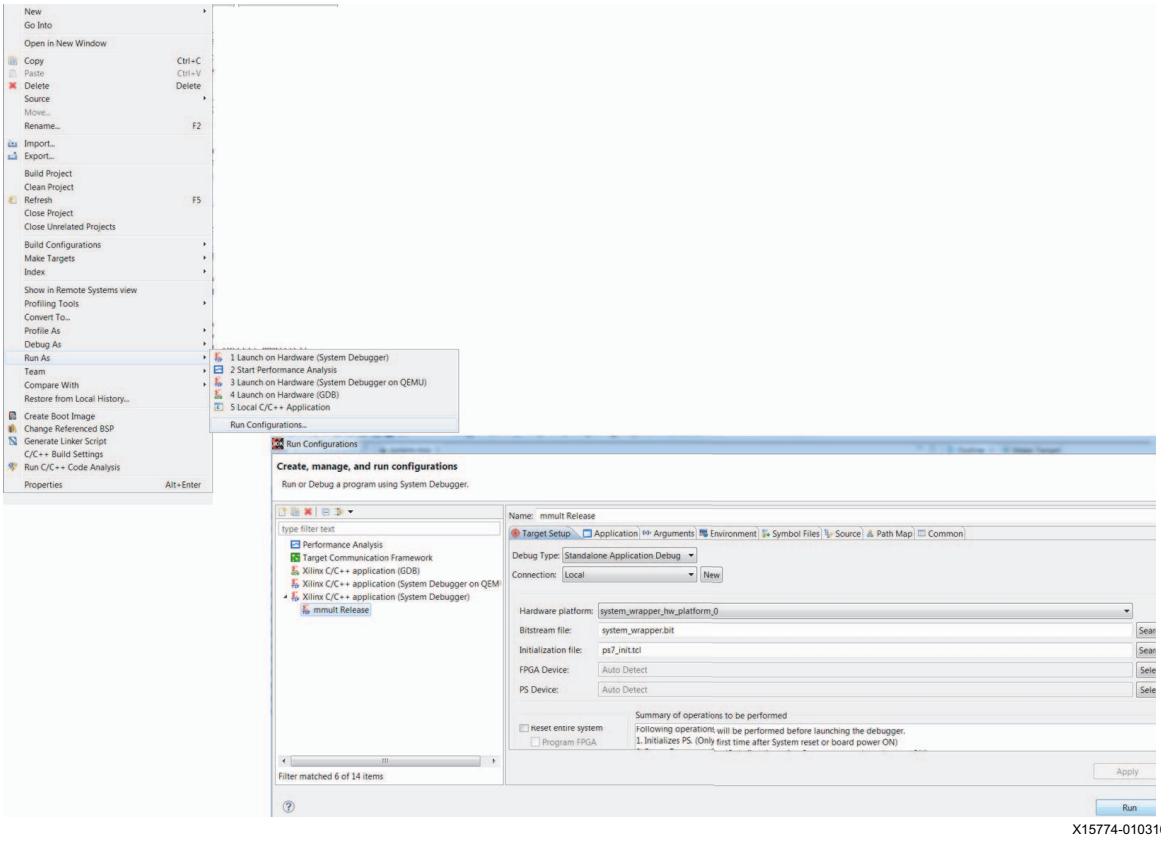


Figure 16: **Set Active Configuration**

9. To execute this application on the board, create a new Run Configuration by right-clicking **Run** > **Run Configurations**. In the GUI, select the **Xilinx C/C++ application (System Debugger)**, and click **New** (or double click the entry). This generates a new configuration. Accept the defaults shown in Figure 17.



X15774-010316

Figure 17: **SDK Run Configurations**

10. Run the application. *Hello World* appears on the terminal.

The next step is to create the software to call the HLS-generated HW accelerator. The `arm_sw` folder contains three C code application files that initialize the DMA, instrument performance measurement, and invoke the hardware accelerator. All of the remaining application files are automatically generated by Vivado HLS and imported by SDK.

1. You can delete the `helloworld.c` file because it is no longer needed.
2. To add the matrix multipliers C files:
   a. Right-click on `src` in the `mmult` project.
   b. Select **Import**.
   c. Select **General**.
   d. Select **File System** and click **next**.
   e. Select the `arm_sw` local file system and select the three files (Figure 18).
3.

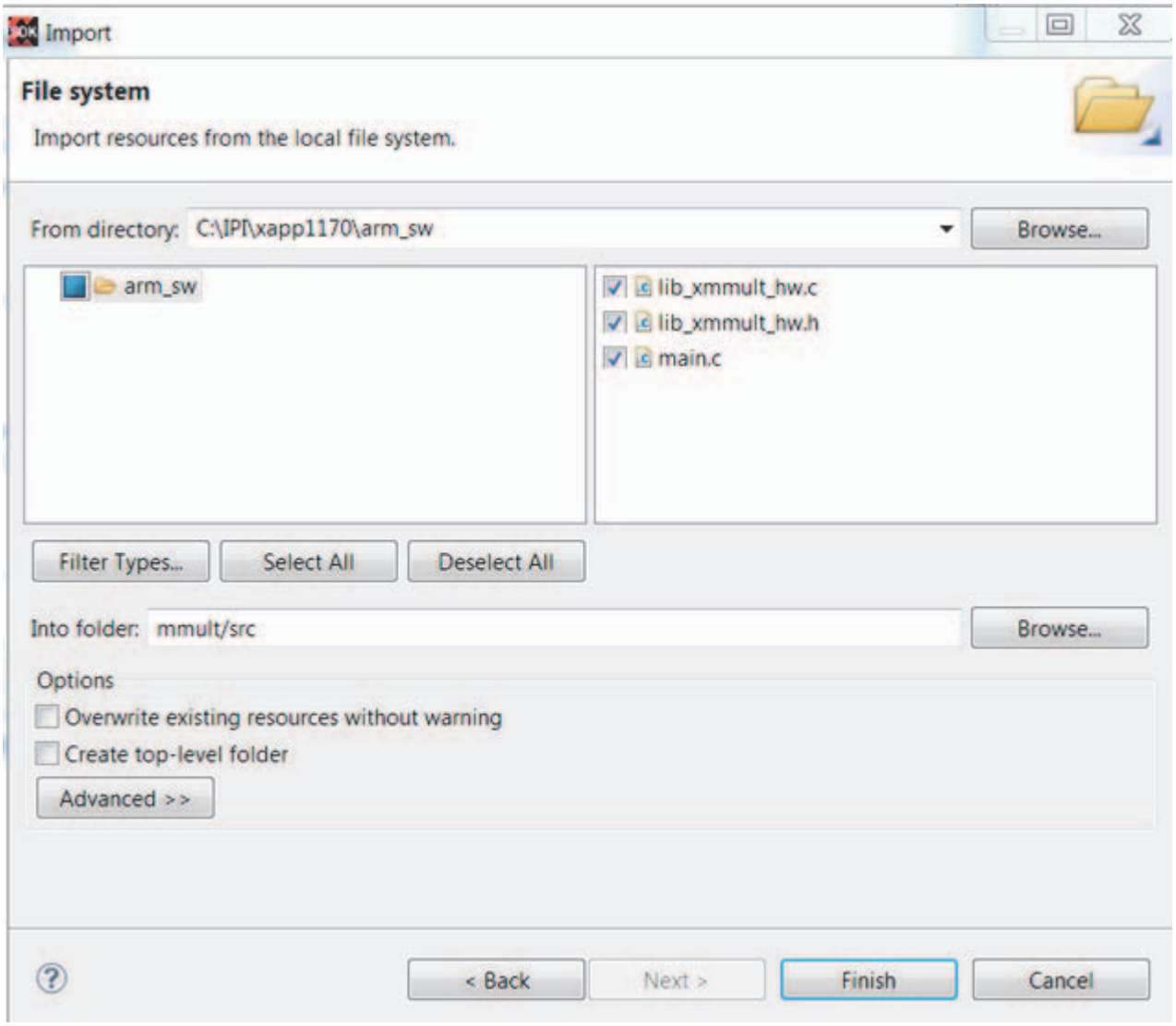


Figure 18: **Importing the Files for the ARM CPU**

3. Prior to running the application, the Stack Size must be changed (Figure 19).
   a. In the Project Explorer `mmult/src`, double-click on the file `lscript.ld`.
   b. Change the Stack Size from `0x2000` to `0x3000`.
   c. Save the file.
4.

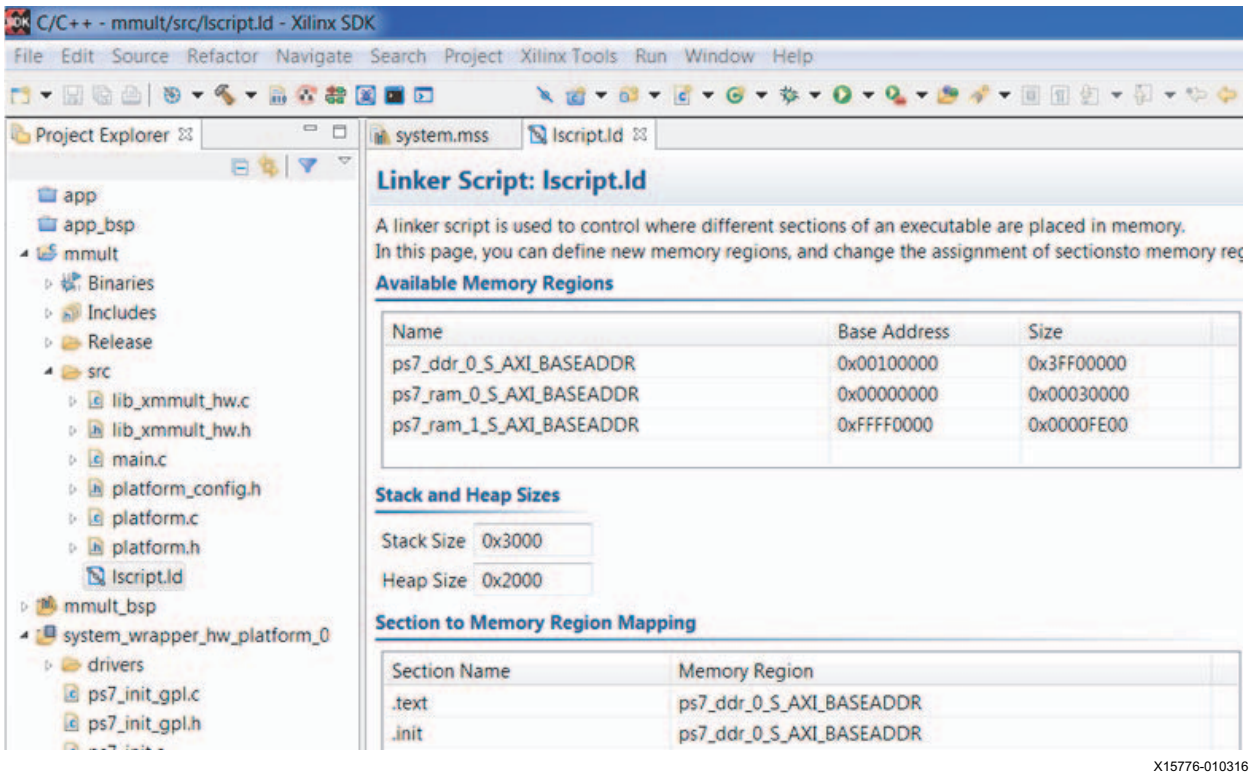


Figure 19: **Changing the Stack Size**

4. To execute the application on the board, create a new Run Configuration by right-clicking on **Run** > **Run Configurations**. In the GUI, select **Xilinx C/C++ application (system Debugger)**, and click **New** (or double-click the entry), which generates a new configuration. Accept the defaults.

5. Run the application. The terminal should display the output shown in Figure 20.

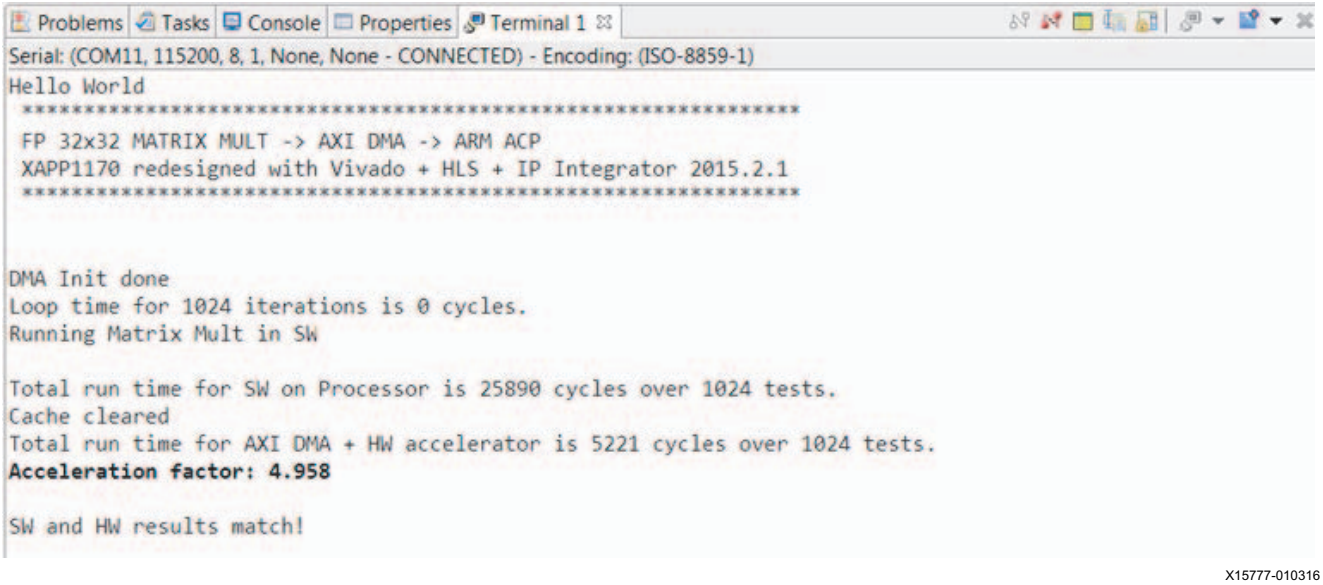


Figure 20: **Terminal Output of the Application Running on the ZC702 Board**

Conclusion

Floating-point designs written in C or C++ can now 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 example application is a 32x32 matrix multiplication core optimized for 32-bit floating point accuracy using the Vivado HLS tool.

The floating-point matrix multiplication modeled in C/C++ code can be quickly implemented and optimized into an RTL design using Vivado HLS. It can then be exported as an IP core that is connected with AXI4-Stream interface to the ACP of the Zynq-7000 AP SoC PS through a DMA core in the PL subsystem of the Zynq-7000 device.

The matrix multiplier HW peripheral running at a 100 MHz clock frequency is computed in almost five fewer clock cycles than its software execution on the ARM CPU running at a 666 MHz clock frequency.

In conclusion, the entire design procedure illustrated in this document can be fully automatized by using the new system design flow called SDSoC.

Reference Design

The reference design files for this application note can be downloaded from:

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

There are two folders:

- `empty` - contains just C++ code and TCL scripts to build the HLS and IPI projects from scratch.
- `pre_built` - contains the whole HLS and IPI projects already developed

Table 1 shows the reference design matrix.

Table 1: **Reference Design Matrix**

Parameter	Description
General	
Developer name	Daniele Bagni, Antonello Di Fresco (Xilinx)
Target devices	Zynq-7000 AP SoC
Source code provided	Yes
Source code format	C and synthesize script
Design uses code and IP from existing Xilinx application note and reference designs or third party	No

Table 1: **Reference Design Matrix *(Cont'd)***

Parameter	Description
Simulation	
Functional simulation performed	Yes
Timing simulation performed	No
Testbench provided for functional and timing simulation	Yes
Testbench format	C
Simulator software and version	Vivado Simulator 2015.4
SPICE/IBIS simulations	No
Implementation software tools/versions used	Vivado Design Suite 2015.4
Static timing analysis performed	Yes
Hardware Verification	
Hardware verified	Yes
Hardware platform used for verification	Xilinx ZC702 board

References

1. *Zynq-7000 All Programmable SoC: Concepts, Tools and Techniques (CTT)* (UG873)
2. *Floating-Point Design with Xilinx's Vivado HLS*, James Hrica, Xcell Journal, Fourth Quarter 2012
3. *Floating-Point PID Controller Design with Vivado HLS and System Generator for DSP* (XAPP1163)
4. *Vivado Design Suite Tutorial: Designing IP subsystems using IP Integrator* (UG995)
5. *ZC702 Evaluation Board for the Zynq-7000 XC7020 All Programmable SoC* (UG850)
6. *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)
7. *Vivado Design Suite Tutorial: High-Level Synthesis* (UG871)
8. *UG761 AXI Reference Guide* (UG761)
9. *AXI DMA v7.1 LogiCORE IP Product Guide* (PG021)
10. *Vivado Design Suite User Guide: Designing IP subsystems Using IP Integrator* (UG994)
11. *SDSoC Environment User Guide* (UG1027)
12. *Using the SDSoC IDE for System-level HW-SW Optimization on the Zynq SoC*, Daniele Bagni, Nick Ni, Xcell Software Journal, issue 1, Third Quarter 2015

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
01/21/2016	2.0	Release based on Vivado 2015.4
06/29/2013	1.1	Release based on Vivado HLS 2012.2 and PlanAhead (ISE/XPS/SDK) 14.4.

Please Read: Important Legal Notices