Imperial College
London

ADVANCED DSD COURSEWORK

V2.1 – Christos Bouganis

TABLE OF CONTENTS

Requirements to complete this coursework:

Xilinx Vivado 2018.2 or 2019.1 (including SDK and HLS);

ZedBoard (Zynq system);

Vivado Design Suite Tutorial: High-Level Synthesis (UG871);

Vivado Design Suite User Guide: High-Level Synthesis (UG902).

Legend:

ⓘ Important information.

📖 Reference material that you should read before proceeding.

🛠 Work you should deliver. A set of tasks for you to do, and deliverables to include in your report.

🕷 Debug hints

☺ Helpful information

Please read this document carefully before starting the coursework as it will guide you to complete the coursework efficiently. There are links to references that explain, and exemplify, procedures you will need to know how to do. You're advised to read and understand them.

INTRODUCTION

In this coursework you will acquire experience in the design of digital systems. As a case study, we will focus on the acceleration of computationally intensive applications, and we will see how having customised hardware blocks tuned for a certain application can lead to a better performing system compared to one that does not have these computational units.

In this coursework, we will start by running applications on a generic central processing unit (CPU), assess the system's performance and then focus on the design of dedicated hardware processing elements for the acceleration of the same applications. The hardware design will be based on an HLS language, the Vivado HLS, which is a C-based high-level description language that allows you to design hardware components faster than RTL languages.

The system design will target a Field-Programmable Gate Array (FPGA) from Xilinx (now AMD). It is a generic programmable digital device that supports the implementation of any digital hardware design after its fabrication. The FPGA board that you're going to use is the ZedBoard that contains the Zynq-7000 All Programmable SoC, a SoC that contains a hard ARM processor and FPGA fabric. More information about ZedBoard can be found here:

https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

The main application for the configuration of the FPGA is Vivado from Xilinx. It allows to design and program digital systems based on FPGAs. The ARM processor can be coded using Xilinx SDK, where you will learn how to design new hardware components using Vivado HLS.

Please note that the coursework assumes that you have successfully completed the MSc ADIC Lab – Digital System Design. If you have not done so, you will need to go through that lab in order to have the necessary background.

The coursework is structured in two main parts.

- Part 1. In the first part of the coursework, you will become familiar with how to design a hardware module using Vivado HLS and how to interface it with the Zynq-based system.
- Part 2. In this part, you will focus on the mapping of the targeted application to the Zynq system. You will first implement a SW-based version of the application. You will then profile the system to identify the components that need acceleration, design the correspondent HW accelerators using Vivado HLS, and integrate them with the rest of the system in order to improve the performance of the targeted application.

Before embarking on your project, you need to get some background on the Zynq processor. Read Chapters 1 to 3 from the "The Zynq Book". If you want, you can read chapters 4 and 5 of the same book that focus on the performance of the Zynq processor and its applications. Finally, Chapter 6 introduces the Zedboard.

Objective: Acceleration of a Support Vector Machine Classifier

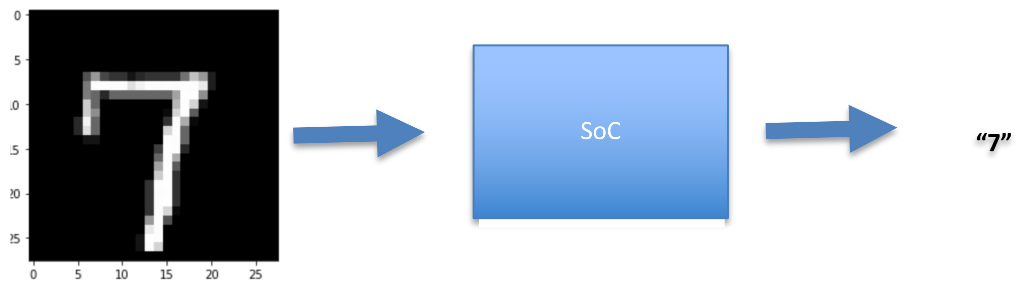
The main purpose of this coursework is to create a digital system that implements and optimises the performance of a Support Vector Machine classifier in an embedded system. Support Vector Machines are a powerful machine learning method that provides good generalization performance for a wide range of regression and classification tasks. SVMs is a supervised learning method. SVMs use a decision function, whose parameters are tuned during the training stage, in order to infer the classification of an unknown input during the classification phase. In SVMs, a training dataset, consisting of pairs of input vectors and desired outputs, is used to model and construct the decision function of the system. During the training phase, the system identifies the *Support Vectors* (SVs), which are those data points that can best build a separation plane for the classes. Those vectors are then used to predict the class of any future data point during the classification phase.

On the classification (inference) phase, any new datum x is classified according to the output of the decision function:

$$f(x) = sgn\left(\sum_{i=1}^{N_{SV}} y_i a_i K(x_i, x) + b\right)$$

where N_{SV} is the total number of Support Vectors x_i identified in the training phase, along with their weights a_i, b is the offset to the origin, y_i is the ground truth label of SV x_i, and K( ) denotes a kernel function that allows the SVM to perform well with problems that require non-linear separation boundaries.

The system that you need to design will take as input a 28x28 pixels input image of a digit and should be able to classify the digit in the image. An example of classifying the digit 7 is shown below.



The aim is to design a system that would classify as many input images as possible per unit of time (i.e. you need to optimise your system for throughput). In your application you will use the following kernel known as the Radial Basis Function (RBF) kernel:

$$K(x_i, x_j) = \exp\left(-\gamma \, || x_i - x_j ||^2\right)$$

where || … || refers to the L2-norm of a vector.

The parameters of the model have been tuned for you (i.e. the training of the model has been performed) and are provided. The training was performed to classify between the digits 0 and 1 of the MNIST dataset and the accuracy achieved is 99.65%.

Thus, given a set of SVs and parameters y_i, a_i, b and γ, your aim is to design an embedded system that performs classification of new data points x. You can assume that the above parameters are known during the design time (i.e. no need to treat x_i , y_i, a_i, b and γ as runtime parameters).

Your design should be assessed under the following metrics (when they are applicable).

- FPGA resources used by your system,
- Throughput – Number of data classifications per second. (Primary objective)

- Latency or Execution time – This is the time that is needed for your system to compute a single point classification

ⓘ **Please note that at the end of the coursework you need to provide the code for each Task. Please check the coursework submission details for more information on this to avoid any duplication of work.**

Please have in mind best solution for latency driven applications (i.e. lowest latency) may not give the best throughput, and vice versa.

Part 1: Lab – Dot-product Module Design

In this part, we will go through the process of creating and optimising a dot-product module using Vivado HLS. After finishing this lab, you will learn:

- the design process of HLS, which is Validation, High-Level Synthesis, RTL Verification, and finally IP creation.
- the usage of fixed-point data types to make your design smaller and faster.
- the optimisation of the design using pragmas, such as Pipeline, Loop Unrolling, Array and Interface.
- appreciate how HLS improves productivity when designing hardware.

The objective of this part is to design a module that takes as input two vectors, calculates their dot-product, and returns the computed output.

STEP 1A: CREATE THE HLS PROJET

📖 You have already performed Exercise 4C of "The Zynq Book Tutorials" in the MSc ADIC Lab, which introduces the design process of HLS. Please read that section again before you start.

Open the Vivado HLS, and create a new project named **dot_prod_prj**

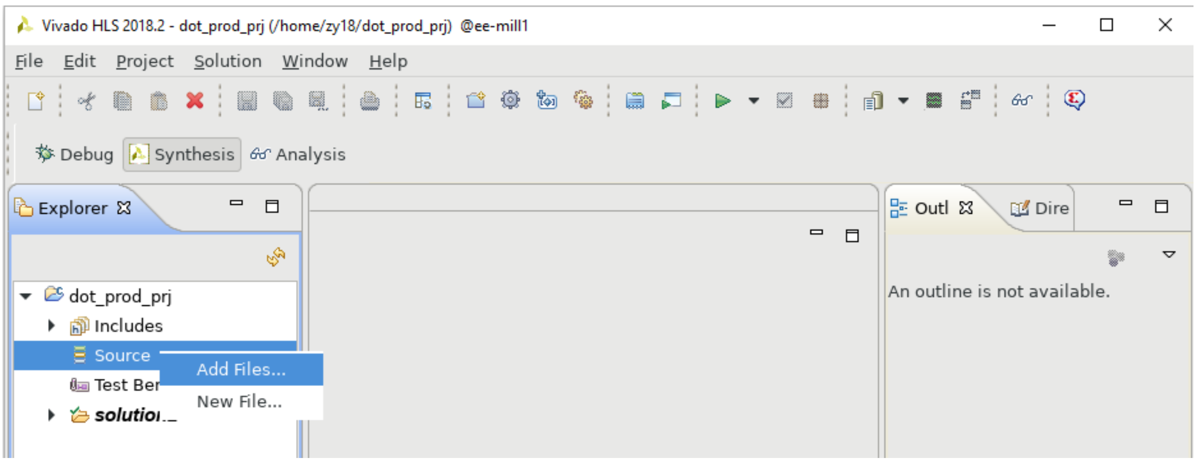
Click **Next** twice, as we will create the source files and the testbench later

Choose **Zedboard** in **Part Selection**, and then **Finish**

STEP 1B: CREATE THE SOURCE FILES

In the **Explorer** panel, right-click on **Source->New File …**

Create a new source file named **dot_prod.cpp** and a header file named **dot_prod.h**


Vivado HLS 2018.2 - dot_prod_prj (/home/zy18/dot_prod_prj) @ee-mill1
File Edit Project Solution Window Help
Debug
Synthesis
Analysis
Explorer
Outl
Dire
An outline is not available.
dot_prod_prj
Includes
Source
Test Ber
solutio
Add Files...
New File...

The dot-product module takes two input vectors of predefined length N and generates their dot-product at its output. For now, we assume the length N=10 and the data type to be floating point. We should have the following definitions and the function declaration in **dot_prod.h**:

```
#ifndef DOT_PROT_H
#define DOT_PROT_H

#define N 10
typedef float data_t;
void dotProduct(data_t x[N], data_t y[N], data_t *output);

#endif
```

Next, implement the **dotProduct** function in **dot_prod.cpp**, and the function is a for-loop with the multiplication and accumulation inside.

```
#include "dot_prod.h"

void dotProduct(data_t x[N], data_t y[N], data_t *output)
{
  data_t z = 0;

  int i;
  for(i=0;i<N;i++)
    z += x[i] * y[i];

  *output = z;
}
```

STEP 1C: CREATE THE TESTBENCH

In the **Explorer** panel, right-click on **Test Bench->New File …**

Create a new file named **dot_prod_tb.cpp**, where we need to define the main function as the testbench.

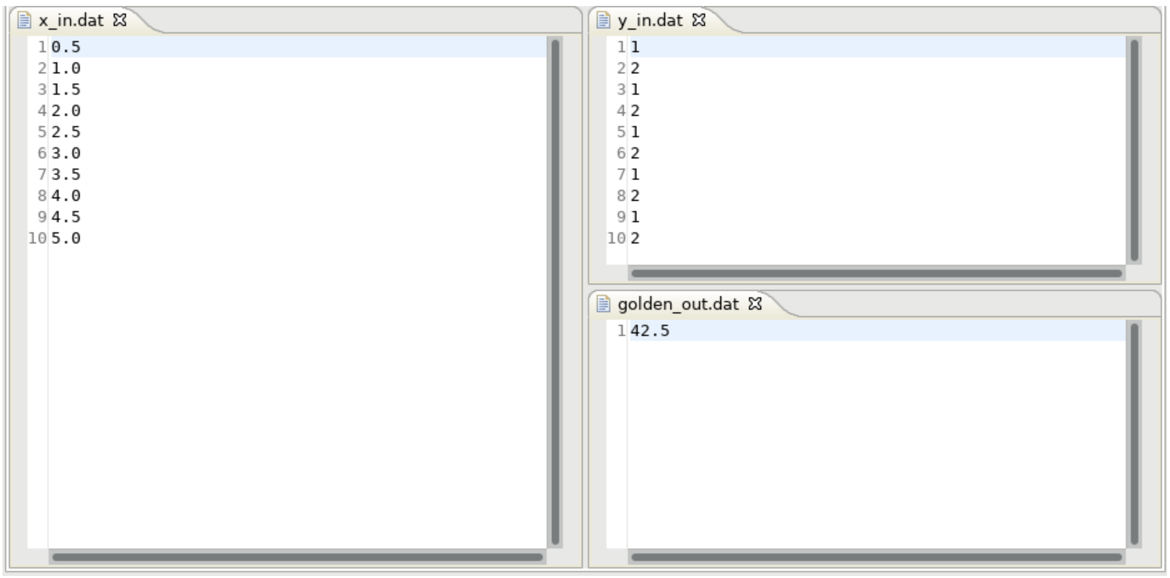
The testbench structure for HLS is similar to that of the RTL design, which initialises the input, calls (instantiates) the design under test, and checks the correctness of the output.

```
#include <stdio.h>
#include "dot_prod.h"

int main(){
  data_t x[N], y[N];
  data_t out, golded_output;
  float tmp;
  FILE *fp1, *fp2, *fp3;
  int i = 0;

  // read inputs and golden output from files
  fp1 = fopen("x_in.dat", "r");
  fp2 = fopen("y_in.dat", "r");
  fp3 = fopen("golden_out.dat","r");
  for(i = 0; i < N; i++){
    fscanf(fp1, "%f", &tmp);
    x[i] = data_t(tmp);
    fscanf(fp2, "%f", &tmp);
```

```
        y[i] = data_t(tmp);
    }
    fscanf(fp3, "%f", &tmp);
    golded_output = data_t(tmp);
    fclose(fp1);
    fclose(fp2);
    fclose(fp3);

    // execute the function
    dotProduct(x,y,&out);

    // compare the outputs
    if (golded_output == out) {
            fprintf(stdout, "PASS: The output matches the golden output!\n");
            return 0;
    } else {
            fprintf(stdout, "FAIL: Output DOES NOT match the golden output\n");
            return 1;
    }
}
```

In the **Explorer** panel, right-click on **Test Bench->New File…** to create "**x_in.dat**", "**y_in.dat**" and "**golden_out.dat**", where the first two files are used to initialise the vector *x* and *y*, while the last file is to store the ground truth of the dot-product computation.

Here is an example of what these files should look like and you are always welcome to try out different values of inputs for better test coverage.


x_in.dat

```
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
```

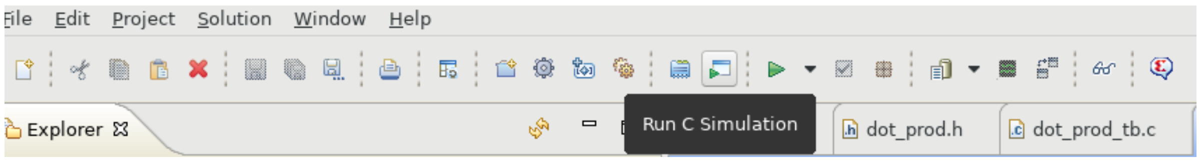
y_in.dat

```
1
2
1
2
1
2
1
2
1
2
```

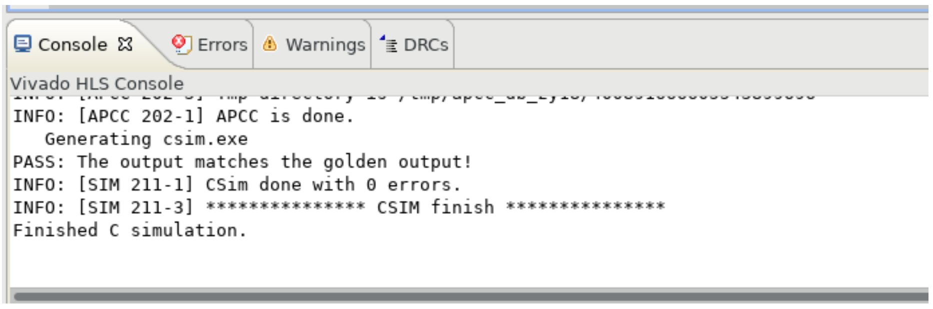
golden_out.dat

```
42.5
```

Next, click the **Run C Simulation** button, or use **Project -> Run C Simulation**, and then click **OK**



Wait a few seconds, and if everything is alright, you should see the following message shown in the **Console**.



```
INFO: [APCC 202-1] APCC is done.
   Generating csim.exe
PASS: The output matches the golden output!
INFO: [SIM 211-1] CSim done with 0 errors.
INFO: [SIM 211-3] *************** CSIM finish ***************
Finished C simulation.
```

Please note that we have only described the functionality of the dotProduct function and simulated its behaviour.

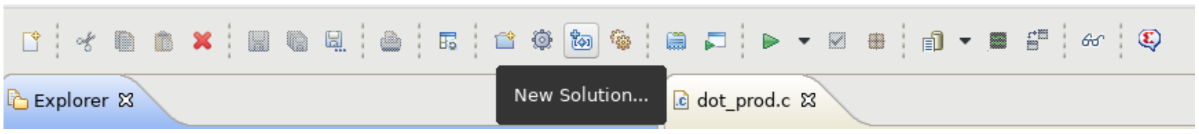
Please read and the section "*C Test Bench*" in *Chapter 3* of *Vivado Design Suite User Guide: High-Level Synthesis (UG902)*, to learn more about writing a good testbench.

STEP 1D: SOLUTION AND DIRECTIVES

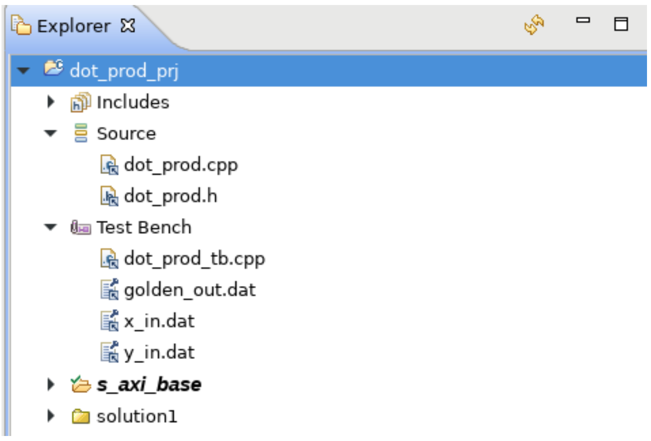
So far, we have already described the function of dot-product in the **C Source file**. In the next steps we will introduce hardware-specific information such as how that block should be mapped to hardware, as well as how it should interface with the rest of the system.

For the same source description, there are many possible versions of hardware implementation. We can have the smallest design, the fastest design, or a balanced design. Therefore, an HLS project can have multiple **solution**s, where each solution corresponds to a different version of hardware implementation.

In the **Explorer** panel, you can see that **solution1** is activated by default. Now click the button **New Solution …**, to create a new solution named **s_axi_base**, keep other options as the defaults and then click **Finish**.

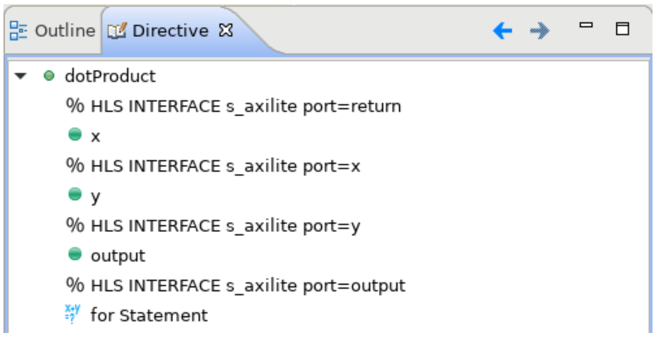


If you look at the **Explorer** panel again, you will find the project is now switched to the new created solution, as ***s_axi_base*** is shown in bold.



Inside each solution, HLS uses directives to specify the behaviour or optimisation of the hardware. To add directives, open **dot_prod.cpp** and click the **directives** tab at the right-hand side. The **directives** tab lists all the objects that can be optimised, which usually include the function name, variables and for loops.

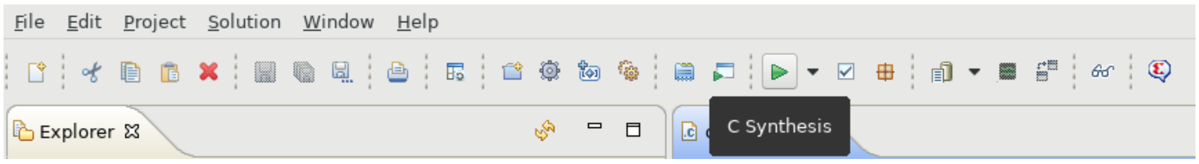
In the **directives** tab, right click the function name **dotProduct** and then click Insert **Directive…** . Select **INTERFACE** from the Directive drop-down menu, make sure the destination is **Directive File** and choose **s_axilite** from the mode menu. Please now repeat the above step for variables **x**, **y** and **outputs** to constrain all the input/output interfaces to be **AXI4-Lite**. Eventually, your **directives** tab should look like this:



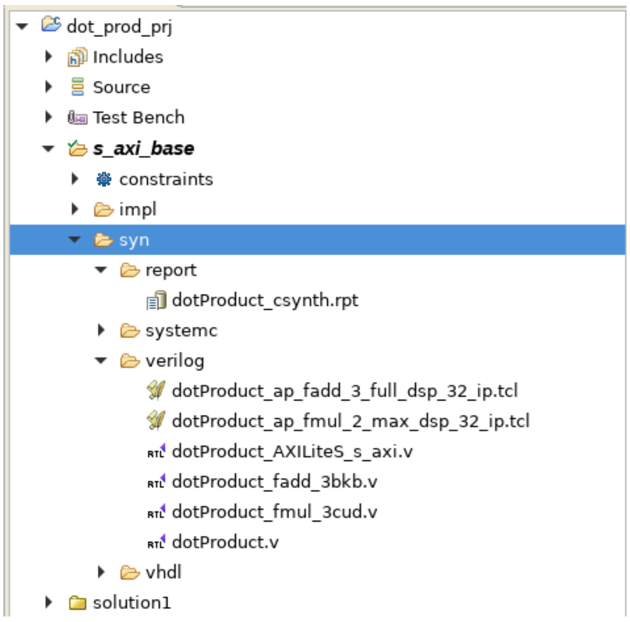
```
dotProduct
    % HLS INTERFACE s_axilite port=return
    x
    % HLS INTERFACE s_axilite port=x
    y
    % HLS INTERFACE s_axilite port=y
    output
    % HLS INTERFACE s_axilite port=output
    for Statement
```

STEP 1E: SYNTHESIS AND EXPORT IP

Click **Project -> Project Settings -> Synthesis -> Browse…** to select **dotProduct** as the top-level function, and then click the **C Synthesis** button, which compiles the C source files + the directives into the RTL designs.



After the synthesis completes, you can find the generated RTL designs under the folder “syn/verilog”. You can also find a report under “syn/report” which contains the **estimated** performance and resource utilisation.

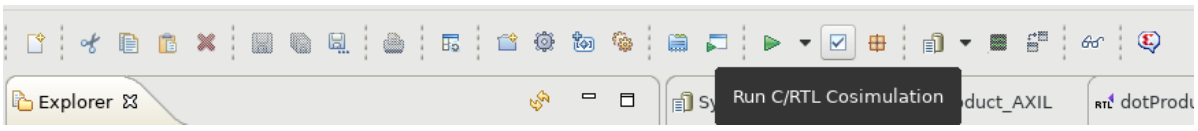


```
dot_prod_prj
    Includes
    Source
    Test Bench
    s_axi_base
        constraints
        impl
        syn
            report
                dotProduct_csynth.rpt
            systemc
            verilog
                dotProduct_ap_fadd_3_full_dsp_32_ip.tcl
                dotProduct_ap_fmul_2_max_dsp_32_ip.tcl
                dotProduct_AXILiteS_s_axi.v
                dotProduct_fadd_3bkb.v
                dotProduct_fmul_3cud.v
                dotProduct.v
            vhdl
    solution1
```

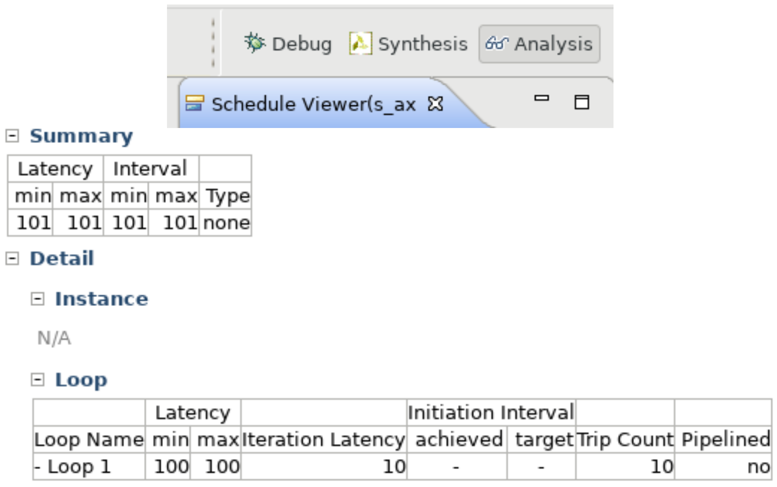
As the report shows, the current solution takes 101 clock cycles, in which the for-loop takes 100 cycles. **Iteration Latency** represents the latency of each loop iteration, and **Trip Count** represents the number of iteration which is same as N.

You can also switch to the **Analysis** perspective to understand how the operations are scheduled. Answer the question: which operation inside the for-loop is most time-consuming in the current hardware implementation?

Click **Run C/RTL Cosimulation** to verify the generated RTL code. A report will also be generated once the cosimulation completes. Again, the numbers shown in the cosimulation report are estimates and they can differ from the C synthesis report and the actual latency measured on the board.



Next, click **Export RTL**, which will package the design as an IP block, and you can find the exported IP at s_axi_base/impl/ip/xilinx_com_hls_dotProduct_1_0.zip



Summary

Latency		Interval		
min	max	min	max	Type
101	101	101	101	none

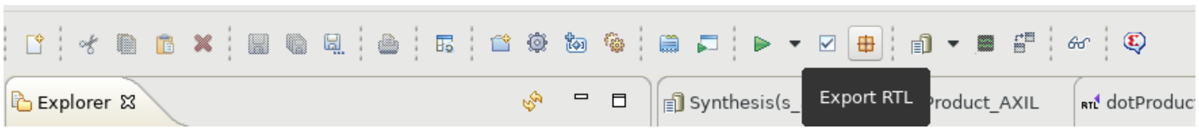
Detail

Instance

N/A

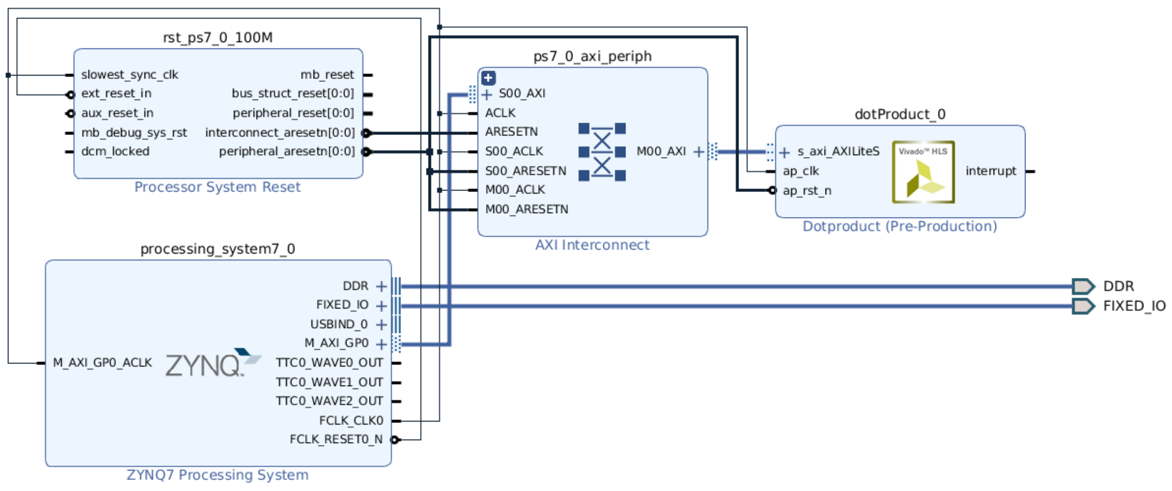
Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Loop 1	100	100	10	-	-	10	no



STEP 1F CREATE THE ZYNQ SYSTEM

You have already learned the integration of an HLS IP into the Zynq platform in Exercise 4C of “The Zynq Book Tutorials”. Please follow the same process to create a Vivado project and import your dotProduct IP. Your block diagram should resemble the figure below.



Run RTL Synthesis, Implementation, and generate the bitstream. You can also find the actual resource utilisation and the timing information by clicking **Implementation->Open Implemented Design** on the left side. Afterwards, launch the SDK and create an application project named **dot_product_app**, which contains the following C source file named **main.c**.

```
#include <stdio.h>
#include "xdotproduct.h"
#include "xtime_l.h"


#define N 10
typedef float data_t;
// test data
data_t x[N] = {0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0};
data_t y[N] = {1.0, 2.0, 1.0, 2.0, 1.0, 2.0, 1.0, 2.0, 1.0, 2.0};

void dotProductSw(data_t x[N], data_t y[N], data_t *output);

int main() {
  XDotproduct HlsDotProduct; // set up HW instance
  XDotproduct_Initialize(&HlsDotProduct, XPAR_DOTPRODUCT_0_DEVICE_ID);

  XTime TimeEnd, TimeStart;
  XTime_GetTime(&TimeStart);
  XDotproduct_Write_x_Words(&HlsDotProduct, 0, (int *) x, sizeof(x)/sizeof(data_t));
  XDotproduct_Write_y_Words(&HlsDotProduct, 0, (int *) y, sizeof(y)/sizeof(data_t));
  XDotproduct_Start(&HlsDotProduct); // start HW execution

  data_t ResHw;
  do {
        u32 ResTmp = XDotproduct_Get_output_r(&HlsDotProduct);
        ResHw = *((data_t *) (&ResTmp));
  } while (!XDotproduct_IsReady(&HlsDotProduct)); // wait until HW finishes

  XTime_GetTime(&TimeEnd);
  u32 TimeHw = ((TimeEnd-TimeStart)*1000000000)/(COUNTS_PER_SECOND);
  // execution time in nanosecond
   xil_printf("HW Time (ns): %d\n\r", TimeHw);

  XTime_GetTime(&TimeStart);

  data_t ResSw;
  dotProductSw(x, y, &ResSw); // SW execution
```

```
   XTime_GetTime(&TimeEnd);
   u32 TimeSw = ((TimeEnd-TimeStart)*1000000000)/(COUNTS_PER_SECOND);
   xil_printf("SW Time (ns): %d\n\r", TimeSw);

   if (ResHw == ResSw)
           xil_printf("Correct!\n\r");

   return 0;
}

void dotProductSw(data_t x[N], data_t y[N], data_t *output) {
   data_t z = 0;
   int i;
   for(i=0;i<N;i++)
     z += x[i] * y[i];
   *output = z;
}
```

From the Project Explorer at the left side, you can find the header file **xdotproduct.h** under the directory of **dot_product_app_bsp/ps7_cortexa9_0/include**. This header file contains a set of auto-generated APIs which allow the ARM processor to communicate with the HLS IP using the AXI4-Lite slave interface. The implementations of these APIs are in **xdotproduct.c,** which can be found under the directory of **dot_product_app_bsp/ps7_cortexa9_0/libsrc/dotProduct_v1_0/src**.

For a detailed explanation of these APIs, please read the "*AXI4-Lite Slave C Driver Reference*" section in *Chapter 4* of *Vivado Design Suite User Guide: High-Level Synthesis (UG902)*.

Through these APIs, the ARM processor (Processing System, PS) can command the HLS IP and calculate the dot-product using FPGA (Programmable Logic, PL). In addition, the above program also calls the **dotProductSw** function which completes the same calculation but on the ARM processor.

Now build the project and run the program. You shall see the messages printed in the SDK terminal showing the latency numbers and comparing the results. Are the results produced by PL (HW) and PS (SW) the same? How about the execution time, and which implementation is faster? Try to explain what you observe?

STEP 1G: FIXED-POINT REPRESENTATION

Having seen how we can design a hardware module using Vivado HLS, the focus is now shifted on the optimisation of the given design. The first type of optimisation we are going to apply is reducing the precision of the arithmetic operations and variables.

Please read the section "*Arbitrary Precision Data Types Library*" in *Chapter 2* of *Vivado Design Suite User Guide: High-Level Synthesis (UG902)* before we move on.

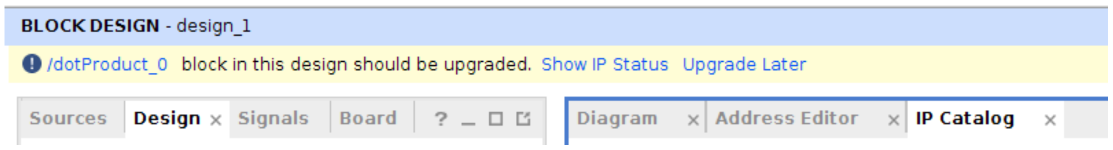
Go back to the HLS project **dot_product_prj** and create a new solution named **s_axi_fp**. Then include the header file **"ap_fixed.h"** in **dot_prod.h** and change the definition of **data_t** to **ap_fixed<32,16>**, which will apply a fixed-point representation to our design where our variables will be using 32 bits, and 16 bits are used for the integer part.. Please also update the testbench file as follows:

```
#include <stdio.h>
#include "dot_prod.h"

int main(){
   data_t x[N], y[N];
```

```
  data_t out, golded_output;
  float tmp;
  FILE *fp1, *fp2, *fp3, *fp4, *fp5, *fp6;
  int i = 0;

  // read inputs and golden output from files
  fp1 = fopen("x_in.dat", "r");
  fp2 = fopen("y_in.dat", "r");
  fp3 = fopen("golden_out.dat","r");

  fp4 = fopen("x_in_fp.dat", "w");
  fp5 = fopen("y_in_fp.dat", "w");
  fp6 = fopen("golden_out_fp.dat","w");

  for(i = 0; i < N; i++){
    fscanf(fp1, "%f", &tmp);
    x[i] = data_t(tmp);
    fprintf(fp4, "%d\n",*((int *)(&x[i])));

    fscanf(fp2, "%f", &tmp);
    y[i] = data_t(tmp);
    fprintf(fp5, "%d\n",*((int *)(&y[i])));
  }

  fscanf(fp3, "%f", &tmp);
  golded_output = data_t(tmp);
  fprintf(fp6, "%d\n",*((int *)(&golded_output)));

  fclose(fp1);
  fclose(fp2);
  fclose(fp3);
  fclose(fp4);
  fclose(fp5);
  fclose(fp6);

  // execute the function
  dotProduct(x,y,&out);

  // compare the outputs
  if (golded_output == out) {
        fprintf(stdout, "PASS: The output matches the golden output!\n");
        return 0;
  } else {
        fprintf(stdout, "FAIL: Output DOES NOT match the golden output\n");
        return 1;
  }
}
```

After running the simulation, you will find three generated data files **x_in_fp.dat**, **y_in_fp.dat** and **golden_out_fp.dat** under **s_axi_fp/csim/build**. The usage of these files will be explained later. At present, re-synthesise the design and export the IP. The new generated IP can be found under **s_axi_fp/impl/ip/xilinx_com_hls_dotProduct_1_0.zip**. Please compare the new reports with those from the previous floating-point implementation.

Go to the Vivado project and make sure the newly generated IP is added to your IP repository. You will then see a message pop up at the top of the window, which reminds you to update the IP in the block design. Click **Show IP Status**, then click **Upgrade Selected** at the bottom.


BLOCK DESIGN - design_1

/dotProduct_0 block in this design should be upgraded. Show IP Status Upgrade Later

Now click generate the bitstream again and re-export the new hardware. However, after you launch SDK, you will find some errors showing up in the console, if you try to re-build the application project **dot_product_app**. To fix these errors, you need to update all the drivers by right-clicking the **dot_product_app_bsp** folder and selecting **Re-generate BSP sources**. You may also notice that some functions in **xdotproduct.h** have been automatically renamed after the design is switched to fixed-point representation. Therefore, please make the following changes in **main.c.**

Before: XDotproduct_Write_x_Words **After**: XDotproduct_Write_x_V_Words
Before: XDotproduct_Write_y_Words **After**: XDotproduct_Write_y_V_Words
Before: XDotproduct_Get_output_r **After**: XDotproduct_Get_output_V

We also need to change the data type in **main.c.** However, it is not recommended to use "ap_fixed" library in SDK as the ARM processor will have to simulate this data structure as it has no dedicated hardware and will lead to poor performance because of it. However, we can cast the data to integer type instead by replacing the data with the contents of **x_in_fp.dat**, **y_in_fp.dat** which were generated during the simulation of the HLS project.

Before: typedef float data_t; **After**: typedef int data_t;

Before: data_t x[N] = {0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0};
After: data_t x[N] = {32768, 65536, 98304, 131072, 163840, 196608, 229376, 262144, 294912, 327680};

Before: data_t y[N] = {1.0, 2.0, 1.0, 2.0, 1.0, 2.0, 1.0, 2.0, 1.0, 2.0};
After: data_t y[N] = {65536, 131072, 65536, 131072, 65536, 131072, 65536, 131072, 65536, 131072};

During this conversion, the floating point data has been multiplied by 65536=2^16 and rounded, since ap_fixed<32,16> represents the wordlength of 32 bits among which 16 bits are used for integers. In addition, for the software implementation, the product needs to be divided by 65536 (right shift by 16 bits), as both x and y have been multiplied by 65536.

Before: z += x[i] * y[i]; **After**: z += ((long long) x[i] * y[i]) >> 16;

Program FPGA, launch the application and compare the results with the previous implementation.

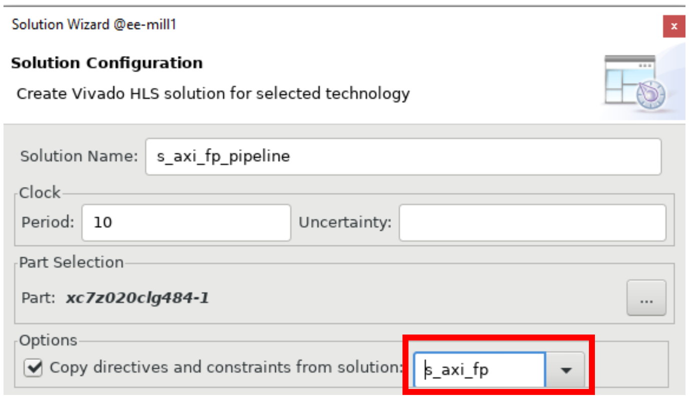
STEP 1H: OPTIMISE THE LOOPS

Please go back to HLS. If we look at the C Synthesis report of the **s_axi_fp** solution, you can see that the execution of the for-loop is the most timing-consuming part of our design. So far, the loop iterations are executed in sequence. HLS provides **UNROLL** and **PIPELINE** directives which allow multiple iterations to occur in parallel. Given that there is no dependencies between each loop iteration computation, the code in the loop can be executed in parallel.

Please create a new solution named **s_axi_fp_unroll** and insert the **UNROLL** directive to the for statement. Leave the factor option empty so that the tool will completely unroll the loop. The tool will also ask you to add a label to the loop. Put **dotProduct_loop** in the textbox and click OK to proceed. Run C Synthesis for the **s_axi_fp_unroll** solution and you will find that the latency decreases but at a cost of 10 times DSP than the **s_axi_fp** solution. In total, there are 10 iterations of the loop, and each iteration now has its own multiplier (hardware block) to allow parallel execution.

Another possibility to optimise the loop, is to create a pipeline for the inner part of the loop. The main body of the loop takes more than one clock cycles, but the computations do not have any dependencies and can be converted in a feed-forward pipeline allowing each loop iteration to start before the computation of the loop body completes.

Please create another solution named **s_axi_fp_pipeline** and make sure you select **Copy directives and constraints from solution: s_axi_fp** in the drop-down menu.


Solution Wizard @ee-mill1

Solution Configuration

Create Vivado HLS solution for selected technology

Solution Name: s_axi_fp_pipeline

Clock

Period: 10 Uncertainty:

Part Selection

Part: *xc7z020clg484-1*

Options

✓ Copy directives and constraints from solution: s_axi_fp

Insert the **PIPELINE** directive to the **dotProduct_loop** label. Leave the **II** (Initial Interval, which is the number of clock cycles before the next iteration of the loop starts to process data) option empty and the tool will target **II** to be 1 by default. Please compare the synthesis report between UNROLL and PIPELINE, and switch to the **Analysis** perspective to understand how the operations are scheduled for each solution.

PIPELINE and **UNROLL** directives can also be combined. Please read Pages 140-157 in *Chapter 1* of *Vivado Design Suite User Guide: High-Level Synthesis (UG902)* for more information.

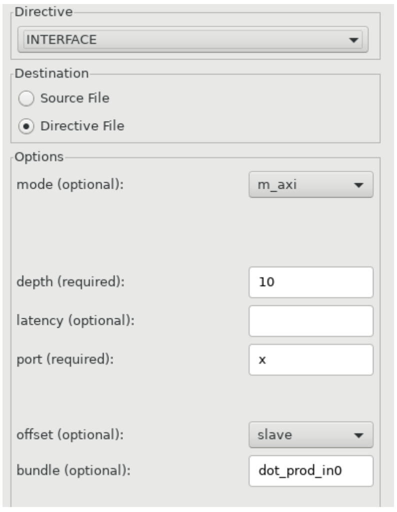
STEP 1I: OPTIMISE THE INTERFACES

A system can either be "compute bound", the computations performed are slower than the memory interactions and reading/writing memory any faster does not result in performance improvements, or "memory bound", the compute is completing faster than the memory interactions and improving the rate of compute will not yield any more overall execution time benefits. In the latter case, the system is waiting for new data to be read in for memory to be able to execute.

Consider you current designs and conclude whether they are compute or memory bounded.

In the case of memory bounded designs, switching to a faster interface can help to improve the performance. At the moment, your design is using an AXI4 lite interface. AXI4 master interface is suitable for transferring arrays between PL and PS in high throughput. The interface can burst read/write multiple values from/to DDR using a single read/write request.

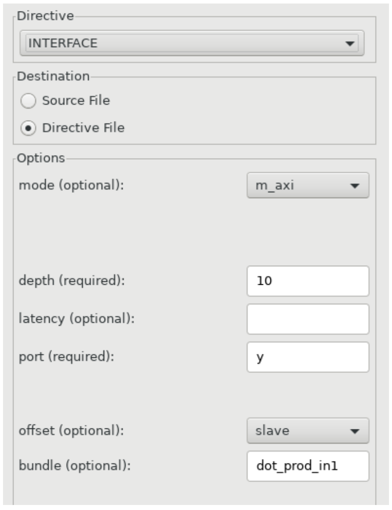
Create a new solution named **m_axi_fp_pipeline** and select **Copy directives and constraints from solution: s_axi_fp_pipeline**. Modify the **INTERFACE** directives of port x and y as the following configurations.


Directive: INTERFACE

Destination: Source File [✓ Directive File]

Options

mode (optional):	m_axi
depth (required):	10
latency (optional):	
port (required):	x
offset (optional):	slave
bundle (optional):	dot_prod_in0


Directive: INTERFACE

Destination: Source File [✓ Directive File]

Options

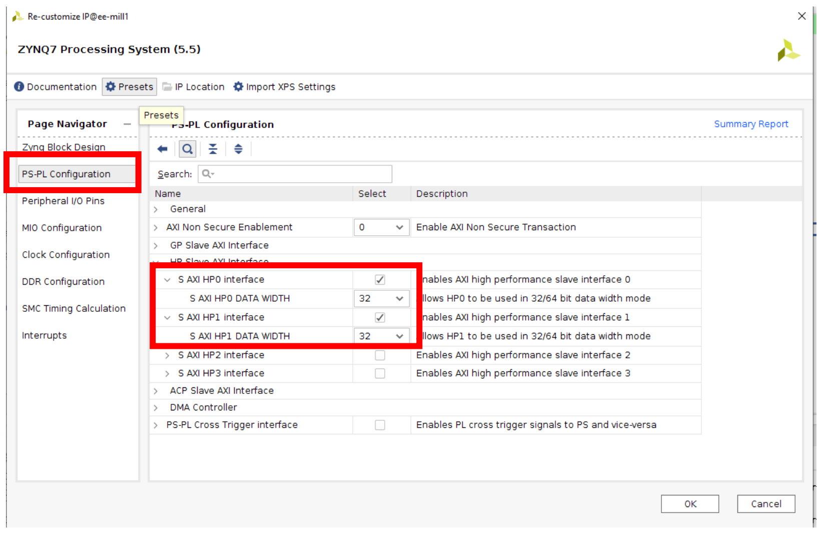
mode (optional):	m_axi
depth (required):	10
latency (optional):	
port (required):	y
offset (optional):	slave
bundle (optional):	dot_prod_in1

The **depth** option is specified as the size of the array, which is 10 in the current experiment. It is required for C/RTL co-simulation to work. The **offset** option is set as slave. Therefore, an extra s_axilite interface will be generated alongside the AXI master interface so that PS can specify the base address of the array. Without this, the IP will by default read 0x0 in the address space, which is not necessarily where the array has been declared. The **bundle** option superficies the name of the bundle where the port is placed. By giving port x and y different bundle names, we place each port in a separate bundle to maximise performance. Otherwise, multiple ports will be grouped together to share the same interface by default.

After running C Synthesis, you will find the reported latency estimation of **m_axi_fp_pipeline** is surprisingly worse than **s_axi_fp_pipeline**. Don't be worried about this result, as the C Synthesis underestimates the latency of s_axilite too much. After comparing the C/RTL co-simulation results, you will find the performance of m_axi is better than s_axilite.

Re-export the IP from HLS and import this new IP into the Vivado IP Catalog. Afterwards, re-draw the block design through the following steps:

- add the ZYNQ7 Processing System and the Dotproduct IP.
- Double-click the ZYNQ7 Processing System IP. Enable S AXI HP0 and HP1 interfaces and set the data width to be 32 bits.



Re-customize IP@ee-mill1

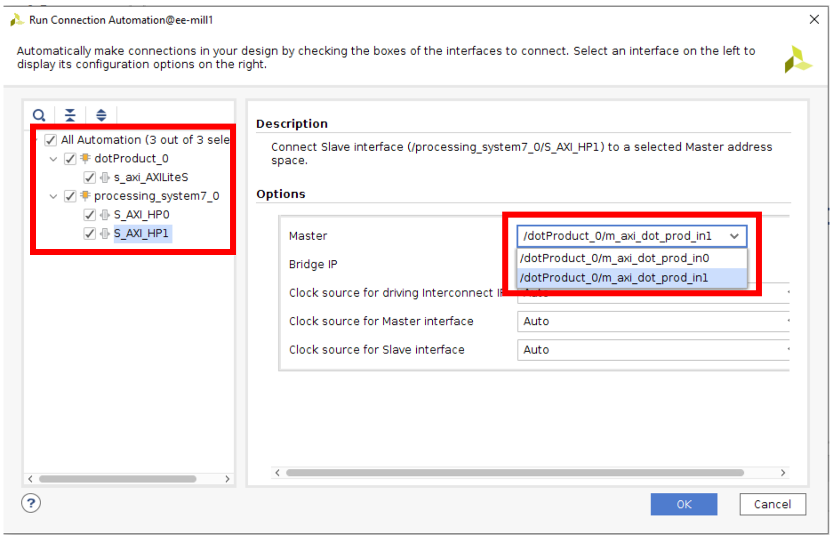
ZYNQ7 Processing System (5.5)

PS-PL Configuration

Search:

Name	Select	Description
General		
AXI Non Secure Enablement	0	Enable AXI Non Secure Transaction
GP Slave AXI Interface		
HP Slave AXI Interface		
S AXI HP0 interface	✓	nables AXI high performance slave interface 0
S AXI HP0 DATA WIDTH	32	llows HP0 to be used in 32/64 bit data width mode
S AXI HP1 interface	✓	nables AXI high performance slave interface 1
S AXI HP1 DATA WIDTH	32	llows HP1 to be used in 32/64 bit data width mode
S AXI HP2 interface	☐	Enables AXI high performance slave interface 2
S AXI HP3 interface	☐	Enables AXI high performance slave interface 3
ACP Slave AXI Interface		
DMA Controller		
PS-PL Cross Trigger interface	☐	Enables PL cross trigger signals to PS and vice-versa

- Click Run Connection Automation. Tick all boxes, and for S_AXI_HP1, choose its Master to be /dotProduct_0/m_axi_dot_prod_in1 from the drop-down menu.



Run Connection Automation@ee-mill1

Automatically make connections in your design by checking the boxes of the interfaces to connect. Select an interface on the left to display its configuration options on the right.

- ✓ All Automation (3 out of 3 sele
  - ✓ dotProduct_0
    - ✓ s_axi_AXILiteS
  - ✓ processing_system7_0
    - ✓ S_AXI_HP0
    - ✓ S_AXI_HP1

Description

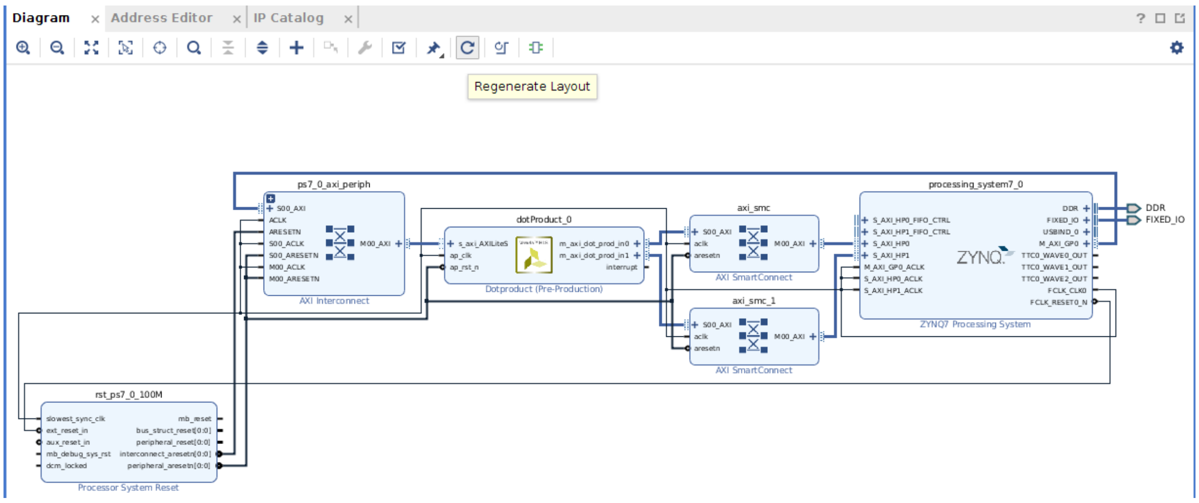
Connect Slave interface (/processing_system7_0/S_AXI_HP1) to a selected Master address space.

Options

Master	/dotProduct_0/m_axi_dot_prod_in1
Bridge IP	
Clock source for driving Interconnect IP	Auto
Clock source for Master interface	Auto
Clock source for Slave interface	Auto

/dotProduct_0/m_axi_dot_prod_in0
/dotProduct_0/m_axi_dot_prod_in1
OK
Cancel

- Click Regenerate Layout and the final design should look like the following graph.



Now generate the bitstream, re-export hardware and launch SDK. In SDK, re-generate BSP sources and apply the following changes to **main.c:**

Before: XDotproduct_Write_x_V_Bytes **After**: XDotproduct_Set_x_V(&HlsDotProduct, (u32)x);
Before: XDotproduct_Write_y_V_Bytes **After**: XDotproduct_Set_y_V(&HlsDotProduct, (u32)y);

Add at the beginning of the file: #include "xil_cache.h"
Add at the beginning of the main function: Xil_DCacheFlush();

The **Xil_DCacheFlush** function is important as the PS has an internal data cache. The function forces the system to flush the cache, which means that the PS pushes any data from the cache to the DDR. As such, if data are read from DDR (which is what happens with our current FPGA design), the most updated values will be retrieved.

XDotproduct_Set_x_V and **XDotproduct_Set_y_V** functions pass the base address of array x and y through the AXI Lite interface, as we have already enabled the offset option in the m_axi directive. Afterwards, the data will be transferred through the AXI Master interface whenever PL sends the request.

Upon implementing the above modifications, you should find the performance gap between the SW and HW implementation to become smaller. You may wonder why the HW implementation is still slower than SW. The reason is that computing the dot-product of two vectors length of 10 is a simple task, and the overhead of transferring data between PS and PL outstrips any potential performance gains due to parallel computations performed in the FPGA.

By increasing the value of **N**, we can reduce the impact of the communication overheads**.**

Please note that In SDK, as the size of array x and y is increased, you need to manually change the size of the stack by editing the **lscript.ld** under the **dot_prod_app/src** folder. If you use **malloc** to declare the array, increase the heap size instead. Assume N=1000, an array of size 1000, where each value is 32-bits (4 bytes) needs 4000 bytes (~4KB) of memory. In total, the program now needs at least 8KB to store array x and y, in that case.

Investigate the performance gap as **N increases** to 2000. Sample different N values and discuss your findings.

Part 2: Project – CORDIC and System Design

The focus of this part of the coursework is to design and optimise an embedded system that performs SVM classification. Your aim is to design and optimise your system for performing SVM classification given a description of the targeted classifier. **The targeted classifier should be implemented as an IP core and connected to the ARM processor through an appropriate AXI bus interface.**

Your system should be able to support the Radial Basis Function (RBF) kernel classifier:

$$y = sgn\left(\sum_{i=1}^{N_{SV}} y_i a_i K(x_i, x) + b\right)$$

- $\gamma = 0.001$, scaling parameter of the classifier, scalar
- $N_{SV} = 165$, number of Support Vectors (SVs), scalar
- $y_i a_i$, the product of SVs labels and weights, vector with the size of (1×165)
- b, bias of the classifier, scalar
- x_i, features of SVs, vector with the size of $(1 \times 165 \times 784)$
- x, 2601 scaled and normalised images each of size 784
- $K(x_i, x)$, kernel function, which is exp $(-\gamma \mid\mid x_i - x \mid\mid^2)$
- y, prediction result, vector with the size of (1×2601)

All the necessary files are provided through BB, which include:

- c_headers/svs.h : File containing x_i,
- c_headers/alphas.h : File containing $y_i a_i$,
- c_headers/bias.h : File containing b
- c_headers/test_data.h : File containing x
- c_headers/ground_truth.h : File containing the ground truth of y
- test_classification.cpp : Reference implementation that reads the files above, computes a prediction for each of the images and calculates the classification accuracy and confusion matrix. The “score”, value of the output before the sign is taken is written to a file scores.txt which can be used as a reference to check correctness of designs.
- training/ : Directory with Python files used to train the SVM and MATLAB files used to quantise the values with a README.md describing how to use the files. This also contains a reference implementation of an SVM in Python to which your C implementation can be compared to.

Hints:

1. As you may be aware from the Part 1 lab, transferring data between PS and PL can be time-consuming. Therefore, it is recommended to store all the SVM model parameters $(\gamma, N_{SV}, y_i a_i, b, x_i)$ inside the BRAM of PL, and only leave x and y to be the input and output of the IP.
2. Due to the BRAM constraints of the board, the values stored in the above header files need to be quantised to the 8-bit fixed point precision to fit all the SVM model parameters as well as at least 1 image worth of data onto the board. The recommended configurations of quantisation are:

x_i, 7 integer and 1 fractional bits

$y_i a_i$, 5 integer and 3 fractional bits

b, 1 bit integer and 7 fractional bits

x, 7 bits integer, 1 fractional bit

3. The exp() function can be implemented using CORDIC, which is an iterative algorithm. You need to tune the number of iterations so that your IP is a faithful implementation with the classification accuracy to be 99.65%. Afterwards, the extension task is to think about the trade-off between performance of the system and utilised resources. The best designs are the ones that belong to the Pareto curve Performance-Resources. The performance is defined as the execution time for the classification of the input set of data (i.e. your system should be optimised for throughput). **Do not forget that the aim of this work is to accelerate the SVM classification process.** For example, you can consider approximating exp() with a piecewise function. Apart from tuning the exp() function , you can also use less precision than the 8 bits defined above.

4. There are also some other techniques that are not covered in the lab material, but you are welcome to try them **AFTER** you have a system that is already working.

   a. Data packing with struct:
   Pages 92-94 of *Vivado Design Suite User Guide: High-Level Synthesis (UG902)*

   b. AXI-Stream interface:
   Pages 108-113, 229-237 of *Vivado Design Suite User Guide: High-Level Synthesis (UG902)*
   https://youtu.be/3So1DPe2_4s

   c. Using both ARM cores:
   https://www.xilinx.com/support/documentation/application_notes/xapp1079-amp-bare-metal-cortex-a9.pdf;
   https://www.hackster.io/whitney-knitter/dual-arm-hello-world-on-zynq-using-vitis-9fc8b7

MARKING SCHEME

The distribution of marks to the parts of the coursework is shown below. Please note that the marking will take into consideration your design approach, optimisations that you have performed, your achieved results, as well as the quality of the submitted report including the analysis of the design and its bottlenecks.

Part	Mark
Part 1: Lab – Dot-product Module Design	15 %
Part 2: Project – CORDIC and System Design (ARM core utilisation, transfer overhead minimisation/hiding, IP core optimisation, CORDIC optimisation/analysis)	75 %
Final Presentation + Interview	10%

COURSEWORK SUBMISSION

Proposed work schedule and Intermediate deliverables deadlines

You should register in a logbook all that you have accomplished including results. Please note that the logbook doesn't replace the reports.

Schedule

Part	**Week starting on:**	**Deliverable deadline** **(end of week, Friday@16:00)**
Part 1: Introduction to Vivado HLS	20/1	
	27/1	Report 1 (Part 1)
Part 2: Acceleration of an SVM classifier	3/2	
	10/2	
	17/2	
	24/2	
	3/3	
	10/3	Report 2 (Part 2)
The oral examination will take place between Monday 17th March and Friday 21st March.		

If you are late in submitting your report, the usual College penalty on late submission is applied

Reports

Each group needs to submit two reports on their project according to the schedule above.

The reports should contain the following:

a) Answers to any questions raised in the project hand-out, e.g. performance, FPGA resources, etc.;

b) Any related design in a block diagram;

c) A detailed justification of the design choices that you have made;

d) Any limitations of your design;

e) An elaboration of what you would have done differently if you were doing the project from the beginning (for example you may have found that you took a wrong design decision but it is too late to go back and fix it. It is important to elaborate on this (why you believe it is a wrong decision and what would be a better one).

Please note that special attention will be given on the justification of your design choices and how you have approached the problem. So please provide enough details around these issues.

Also note that the report should **not** contain any background info like literature review on Zynq or SVMs.

Below I have listed some general points and good practices on the report writing. Please note that the list below is not exhaustive.

General points

1. Describe clearly the aim of the report and its task
2. Please also make sure that you include a high level diagram of the architecture of your system as well as an expectation on its performance, and resources utilization.
3. Provide discussion on every result that you include. Does it make sense? What conclusions you can draw from the reported results?
4. Be precise when you try to justify the results. Do not just use phrases like "… good enough..", "..very slow…"
5. Avoid using screenshots that do not have any useful info.
6. Do not provide a description of the tutorial that you followed. If there is a specific step that gave you trouble or you think is important to mention, then do just that.
7. Have a conclusion section (task level/report). What did you see? (do not summarize what you did)
8. Pay attention to the structure of the report.
9. Think about adding graphs to make your points clearer.

Format of the report

1. The report should be in pdf and you should submit your code only for the final report.
2. Provide names and group id on the first page of your report.
3. The first report should be up to 4 pages long.
4. The second report should be up to 8 pages long. Please do not use less than 11pts font size.

Code

Each group needs to upload the source files of their project for the most advanced design of each part, at the end of the project. Please submit a zip file of your code.

*** end of document ***