# Embedded Processing with SoC design and High-Level Synthesis

Supervisor: Kim Bjerge
José María Górriz Artigot
Date: 10/11/2014

## Table of Contents

# 1. Preface

The self-study course is a continuation of the Hardware-software Co-design course taught by Kim Bjerge in Aarhus School of Engineering.

The report begins with a general overview of the Zynq device and how to build designs with this device, describing how is possible to implement different design choices. Vivado HLS tool is explained in detail and a general explanation is given about how it is possible to optimize the hardware implementation using directives. [1]

The article *High-Level Synthesis for FPGAs: From Prototyping to Deployment* is a part of this report and it will be presented in the exam. [4]

Finally a couple of exercises have been done in order to get a practical understanding about how to make designs in HLS.

The goal of this study course is to get the basic knowledge about the Vivado® High Level Synthesis as well as getting hands on making some exercises with the tool.

# 2. Introduction

The Zynq Book starts with an introduction about the Zynq device explaining which soft and hard processors are embedded in the device as well as the different peripherals which allows those processors to communicate with other systems. Also a description is given about how to start working with the device and how to setup the hardware and connect the device to the computer.

A comparison is made between Zynq's ARM processor and the FPGA embedded processor MicroBlaze where it is shown that the performance of hardcore processors is higher but less flexible. The Zynq device is built by an ARM processor and offers the possibility to add soft processors which gives a high performance and flexibility.

Chapter 6 from the Zynq book covers the ZedBoard which was provided by Aarhus University to implement different designs and do exercises. The book gives an overview about the features of the ZedBoard and the different physical interfaces.

An explanation about communication between all components within an embedded system is done in chapter 9 of the Zynq book. The functioning of the processors, processor cache, coprocessors, software/hardware interrupts in the embedded system is covered in detail.

Furthermore the partitioning of design components in hardware/software is covered. Different components are divided between software which runs on the PS (Processing System) and hardware which runs in the PL (Programmable Logic) and can accelerate the processes. The Xilinx Vivado High Level Synthesis (HLS) tool performs the transition from C/C++/SystemC code implemented in software to RTL code which can be implemented in hardware.

Designs are becoming more complex and time to market is being shorter. In order to face this problem Vivado allows different ways to create IP blocks which can be stored in an IP catalogue and they can be reused, integrated and connected posteriorly to other systems saving development time.

The Vivado HLS tool is introduced in detail.  HLS allows defining an algorithm in a high level of abstraction using a C/C++ or SystemC and synthesize RTL code. There are different ways to influence and optimize the hardware implementation by using directives and create different solutions. In order to find the best architecture the tool offers the possibility to compare those different solutions generated. The different interfaces used by HLS tool to synthesize the RTL are covered extensively too.

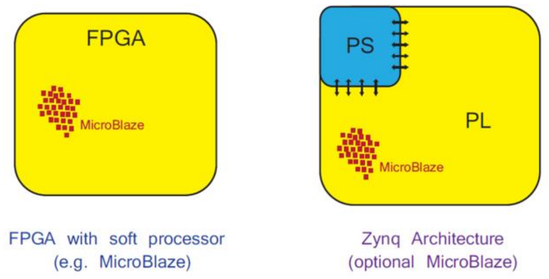
Finally an overview of the AMBA AXI4 interface for IP integrated on a Zynq chip is introduced. The three types of AXI4 have been explained together some examples about how write and read operations are performed.

# 3. The Zynq Device

## 3.1 Locations of hard (ARM Cortex-A9) and soft (MicroBlaze) processors on a Zynq device

MicroBlaze is a soft processor used in FPGA's which configuration is flexible because it has a different number of architectural options which can be added when implementing the processor depending on the requirements of the system.
The Zynq device contains the ARM Cortex-A9 hard processor which has a higher processing performance than MicroBlaze processor but it cannot be configured.

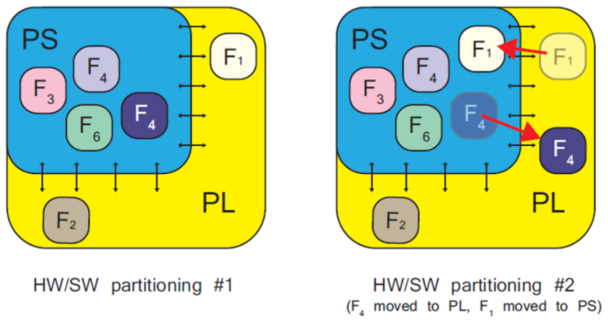
Figure bellow shows the difference between the FPGA's which are built with soft processors (MicroBlaze) and the Zynq device which are built with soft and hard processors. It is possible to configure the Zynq device where only the hard processor is active or where both processors are working at the same time which will have the benefit of the high performance offered by ARM Cortex-A9 processor and the flexibility of MicroBlaze processor.


FPGA
MicroBlaze
FPGA with soft processor (e.g. MicroBlaze)
PS
PL
MicroBlaze
Zynq Architecture (optional MicroBlaze)

[1, p.87]

## 3.2 Hardware/software partitioning based on Zynq

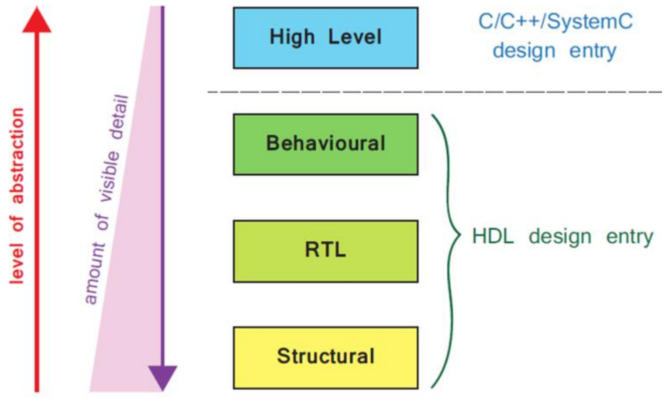
Vivado HLS allows choosing whether the functional elements of the system have to run in software or in hardware. In the figure bellow F4 is moved from software to hardware implementation and F1 is moved from hardware to software implementation. This can be done by writing the code in C/C++ or SystemC and let the Vivado HLS to generate the VHDL of the functional elements that we want to implement in hardware.



[1, p.97]

## 3.3 Levels of abstraction in FPGA designs

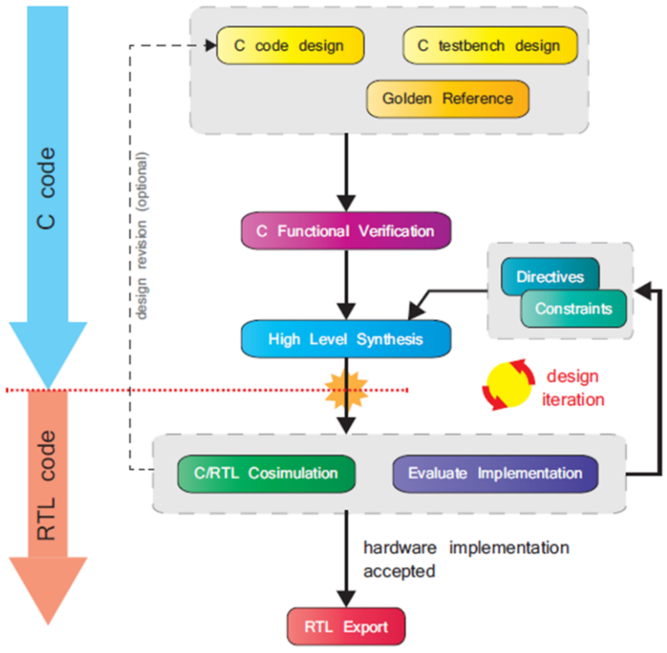
There are different design levels in the implementation of a system. When the level of abstraction is high then the design contains less level of detail in the implementation and for example details like timing, hardware connections or bus transactions are hided.
The levels of abstraction from higher to lower are:
- High-level design: we write our program in C/ C++ or SystemC.
- Behavioural HDL: at this level we concentrate about how the behaviour of the system is going to be.
- Register Transfer Level (RTL) level: the tools use this level of abstraction to translate RTL to hardware.
- Structural: It is the level with the maximum detail where all the hardware elements are connected.



[1, p.256]

## 3.4 Vivado HLS design flow

Figure bellow illustrates the full design flow when generating RTL code and then summarized.



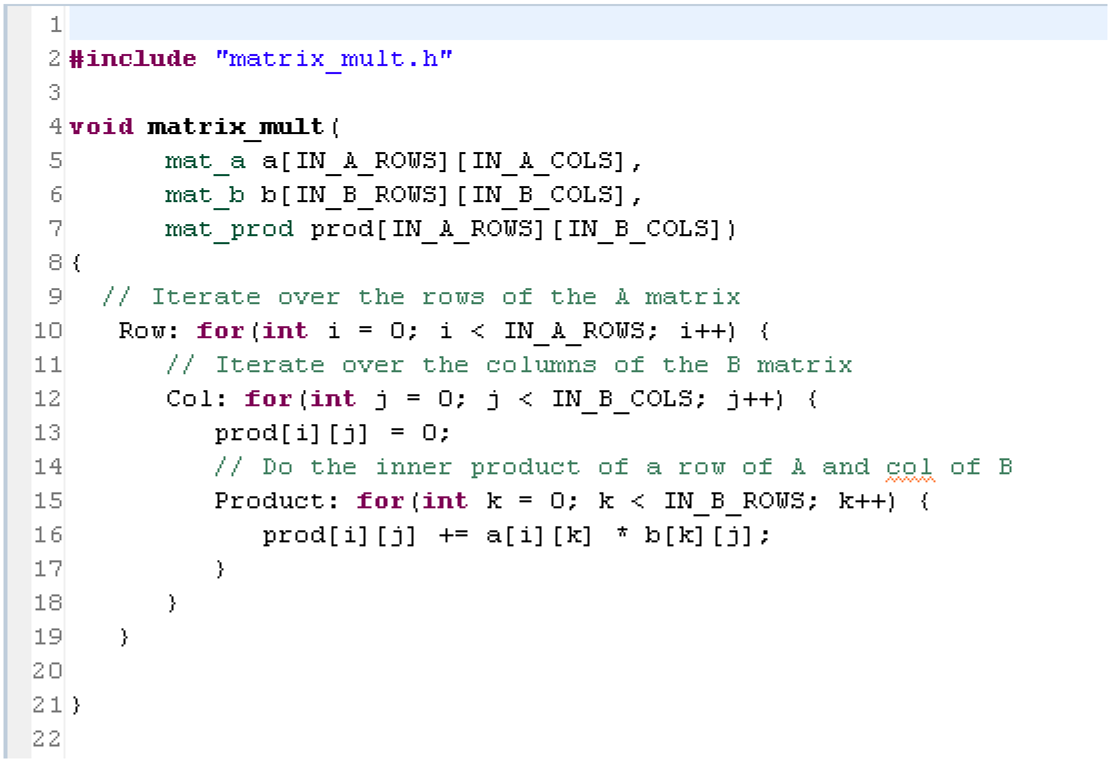
[1, p.267]

## INPUTS TO THE HLS PROCESS

The inputs to the HLS process are:

- ***C code design***: This is our software implementation of the design that we want to synthesize into RTL code.

This is the C file used in the exercise of the matrix multiplication where a multiplication of two matrices is performed in a “for loop”.

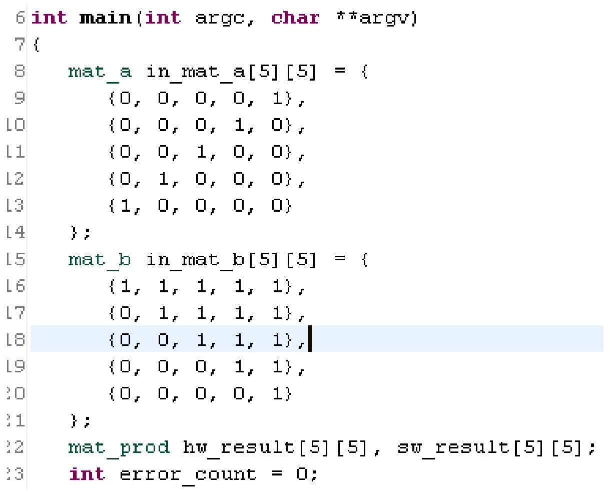


```

#include "matrix_mult.h"

void matrix_mult(
      mat_a a[IN_A_ROWS][IN_A_COLS],
      mat_b b[IN_B_ROWS][IN_B_COLS],
      mat_prod prod[IN_A_ROWS][IN_B_COLS])
{
  // Iterate over the rows of the A matrix
   Row: for(int i = 0; i < IN_A_ROWS; i++) {
      // Iterate over the columns of the B matrix
      Col: for(int j = 0; j < IN_B_COLS; j++) {
         prod[i][j] = 0;
         // Do the inner product of a row of A and col of B
         Product: for(int k = 0; k < IN_B_ROWS; k++) {
            prod[i][j] += a[i][k] * b[k][j];
         }
      }
   }

}

```

[1]

- ***C testbench design***: it is used to verify if the design behaves as we expect.

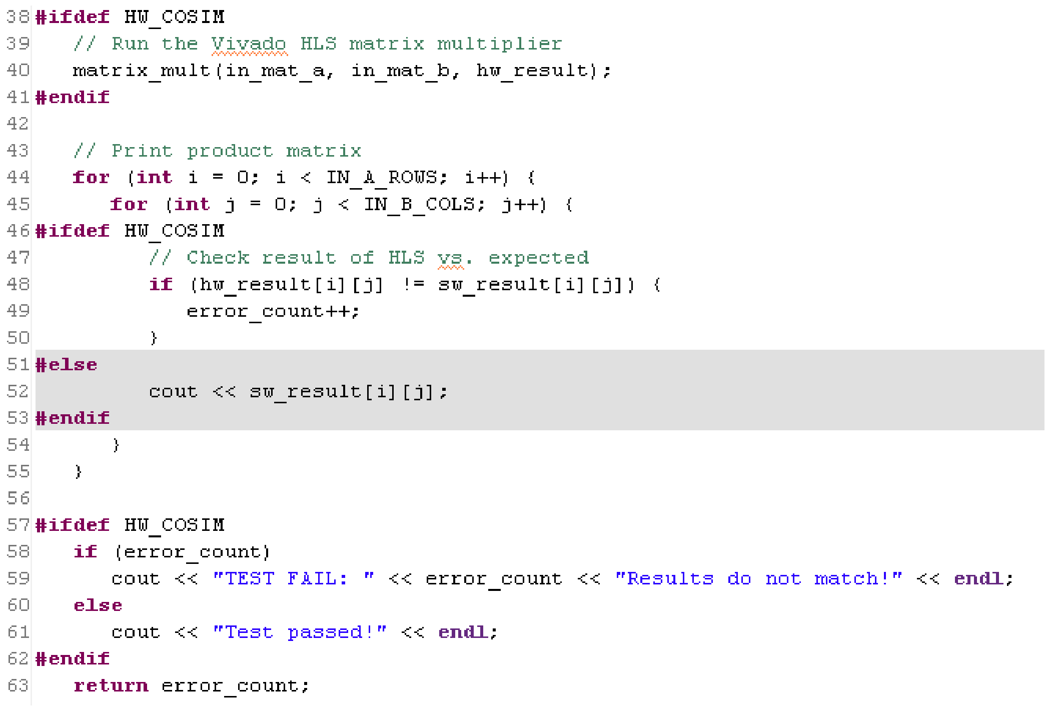
This is the C testbench file used to verify the correct functioning of the matrix multiplication. Four matrices of size 5X5 are created where two are filled with 0’s and 1’s and the other two matrices are used to allocate the result of the multiplication.



```
int main(int argc, char **argv)
{
   mat_a in_mat_a[5][5] = {
      {0, 0, 0, 0, 1},
      {0, 0, 0, 1, 0},
      {0, 0, 1, 0, 0},
      {0, 1, 0, 0, 0},
      {1, 0, 0, 0, 0}
   };
   mat_b in_mat_b[5][5] = {
      {1, 1, 1, 1, 1},
      {0, 1, 1, 1, 1},
      {0, 0, 1, 1, 1},
      {0, 0, 0, 1, 1},
      {0, 0, 0, 0, 1}
   };
   mat_prod hw_result[5][5], sw_result[5][5];
   int error_count = 0;
```

[1]

After the matrix multiplication is called and the matrices are passed in the parameters. The function returns a matrix with the result of the multiplication implemented in hardware and this result is compared with the multiplication implemented in software. If there are errors then the number of errors is count and message of “Test fail” is displayed in the screen and if there are no errors then the text “Test passed” will be displayed in the screen.



```
#ifdef HW_COSIM
   // Run the Vivado HLS matrix multiplier
   matrix_mult(in_mat_a, in_mat_b, hw_result);
#endif

   // Print product matrix
   for (int i = 0; i < IN_A_ROWS; i++) {
      for (int j = 0; j < IN_B_COLS; j++) {
#ifdef HW_COSIM
         // Check result of HLS vs. expected
         if (hw_result[i][j] != sw_result[i][j]) {
            error_count++;
         }
#else
         cout << sw_result[i][j];
#endif
      }
   }

#ifdef HW_COSIM
   if (error_count)
      cout << "TEST FAIL: " << error_count << "Results do not match!" << endl;
   else
      cout << "Test passed!" << endl;
#endif
   return error_count;
```

[1]

- ***Golden reference***: there are some output values that will be used to compare with the outputs generated by our implemented system in order to verify that the design is implemented correctly.

## FUNCTIONAL VERIFICATION

We will compare the outputs of our system with the golden reference and we compare and evaluate the result.
If the functional verification is correct then a “Test passed” message is displayed in the console. Here the matrix verification gave no errors.

```
@I [HLS-10] Opening project 'C:/Zynq_Book/hls/tut3A/matrix_mult_prj'.
@I [HLS-10] Opening solution 'C:/Zynq_Book/hls/tut3A/matrix_mult_prj/solution5'.
@I [SYN-201] Setting up clock 'default' with a period of 5ns.
@I [HLS-10] Setting target device to 'xc7z020clg484-1'
   Compiling ../../../../matrix_mult_test.cpp in debug mode
   Compiling ../../../../matrix_mult.cpp in debug mode
   Generating csim.exe
Test passed!
@I [SIM-1] CSim done with 0 errors.
@I [LIC-101] Checked in feature [HLS]
```

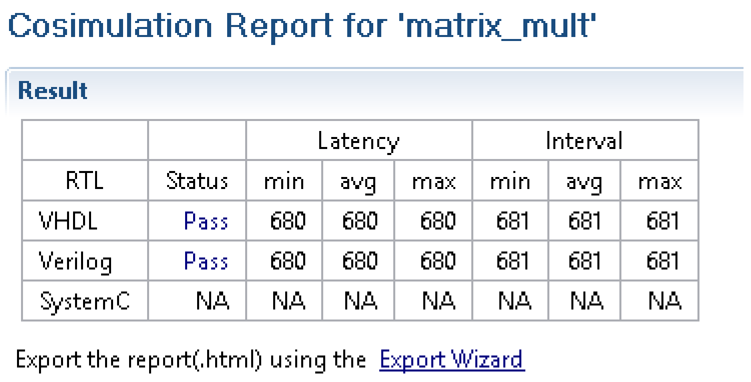
## HIGH-LEVEL SYNTHESIS

It is the HLS process where the C implementation will be transformed into RTL code based in some directives and constraints.
The RTL generation of the matrix multiplication was performed correctly and as it is possible to observe in the figure bellow, SystemC, VHDL and Verilog code were generated.

```
@I [RTGEN-100] Finished creating RTL model for 'matrix_mult'.
@I [HLS-111] Elapsed time: 0.844 seconds; current memory usage: 58.2 MB.
@I [RTMG-282] Generating pipelined core: 'matrix_mult_mul_8s_8s_16_3_MulnS_0'
@I [HLS-10] Finished generating all RTL models.
@I [WSYSC-301] Generating RTL SystemC for 'matrix_mult'.
@I [WVHDL-304] Generating RTL VHDL for 'matrix_mult'.
@I [WVLOG-307] Generating RTL Verilog for 'matrix_mult'.
@I [HLS-112] Total elapsed time: 15.73 seconds; peak memory usage: 58.2 MB.
@I [LIC-101] Checked in feature [HLS]
```

## C/RTL COSIMULATION

We will use the C testbench as an input of our RTL code and we will we compare and evaluate the result with the golden reference. It is not necessary to create a new testbench to verify the RTL code so it improves to possibility of writing some errors if a new testbench was necessary.
A report is generated showing whether the test was successfully passed or not. In the figure bellow is possible to see that VHDL and Verilog tests have been successful passed .The minimum, average and maximum values of the latency and the interval are shown in the cosimulation report too.


Cosimulation Report for 'matrix_mult'

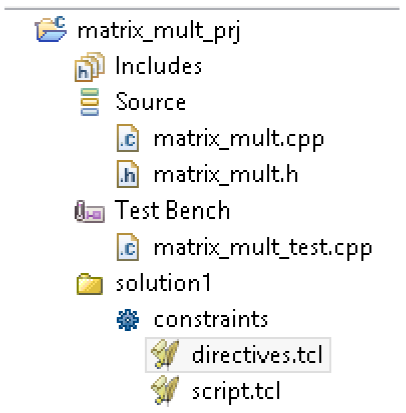
Result

| | | Latency | | | Interval | | |
|---|---|---|---|---|---|---|---|
| RTL | Status | min | avg | max | min | avg | max |
| VHDL | Pass | 680 | 680 | 680 | 681 | 681 | 681 |
| Verilog | Pass | 680 | 680 | 680 | 681 | 681 | 681 |
| SystemC | NA | NA | NA | NA | NA | NA | NA |

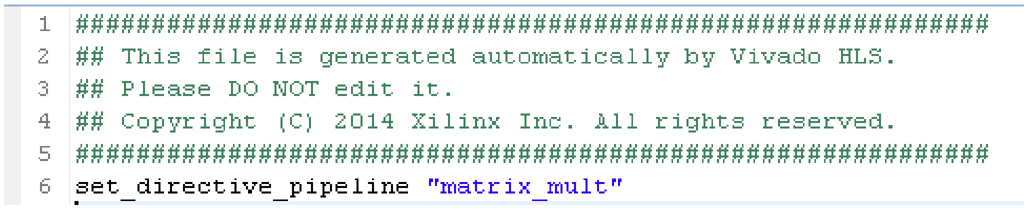
Export the report(.html) using the Export Wizard

## EVALUATION OF IMPLEMENTATION

We will evaluate the design in this phase. There is always the trade-off between the throughput and the hardware resources used in the hardware implementation so we have to find the best architecture which will meet the requirements by changing the directives and/or constraints and evaluating again the design.
The directives file can be found under "constrains" as illustrated in the figure below.



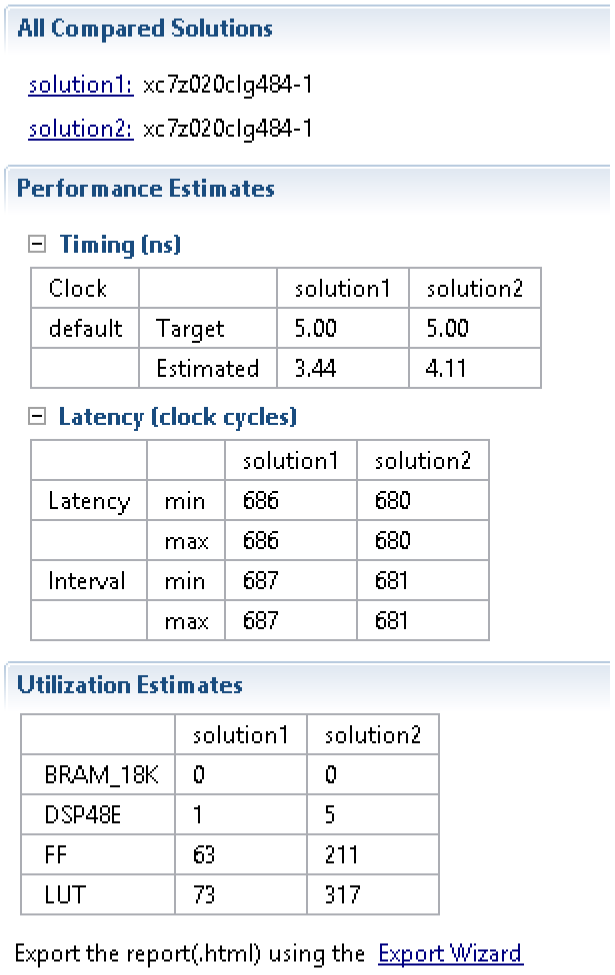
It is possible to write directives in that file and in this case we have a pipeline directive when performing the matrix multiplications.



```
############################################################
## This file is generated automatically by Vivado HLS.
## Please DO NOT edit it.
## Copyright (C) 2014 Xilinx Inc. All rights reserved.
############################################################
set_directive_pipeline "matrix_mult"
```

## DESIGN ITERATIONS

We can have different solutions and make comparisons between them to choose the one that fits best. It is possible to compare the timing, latency and hardware resources used in each implementation. In the example bellow solution 2 is a bit faster than solution 1 with a latency of 680 clock cycles instead of 686 but it uses 4 DSP48E, 148 FF's and 244 LUT's more than solution 1.



All Compared Solutions

solution1: xc7z020clg484-1

solution2: xc7z020clg484-1

Performance Estimates

Timing (ns)

| Clock | | solution1 | solution2 |
|---|---|---|---|
| default | Target | 5.00 | 5.00 |
| | Estimated | 3.44 | 4.11 |

Latency (clock cycles)

| | | solution1 | solution2 |
|---|---|---|---|
| Latency | min | 686 | 680 |
| | max | 686 | 680 |
| Interval | min | 687 | 681 |
| | max | 687 | 681 |

Utilization Estimates

| | solution1 | solution2 |
|---|---|---|
| BRAM_18K | 0 | 0 |
| DSP48E | 1 | 5 |
| FF | 63 | 211 |
| LUT | 73 | 317 |

Export the report(.html) using the Export Wizard

## RTL EXPORT

Finally we can export our design to a larger system or we can use the "Export RTL" button in the main panel which offers the possibility for packaging the IP and store it in the IP catalogue.

### 3.4.1 SystemC [3]

Vivado HLS supports SystemC which is used to model hardware. SystemC is applied to system-level modeling, architectural exploration, performance modeling, software development, functional verification, and high-level synthesis.[2]

Modules are essential in SystemC and can represent hardware components or software classes.

The top-level for synthesis must be inside a SC_MODULE and it can contain methods and threads which describe some functionality.

- **SC_METHOD**
- They can never be suspended
- **SC_CTHREAD**
- Used to model threads in dynamic systems.
- They are sensitive to clock and multiple clock designs can be implemented
- Resets are tolerated by SC_CTHREAD

#### *Communication Channels*

SystemC processes can communicate in a *simulated* real-time environment, using signals of all the datatypes offered by C++, some additional ones offered by the SystemC library, as well as user defined [2]

Communication between threads, methods, and modules is done through channels.

The following signals are used to communicate:

- sc_buffer which can to store some data before to be transmitted.
- sc_signal which is a hardware signal.

#### *Top-Level SystemC Ports*

Modules have ports through which they connect together using channels

The ports available on the top-level interface which are specified in the source code are:

| | |
|---|---|
| - sc_in_clk: | clock input port |
| - sc_in : | input port |
| - sc_out: | output port |
| - sc_inout: | input/output port |
| - sc_fifo_in: | fifo input port |
| - sc_fifo_out: | fifo output port |

#### *SystemC Interface Synthesis*

Vivado HLS does not perform interface synthesis on SystemC but it supports interface for RAM and FIFO ports.

#### *"Unsupported SystemC Constructs*

***Modules and Constructors***

• An SC_MODULE cannot be nested inside another SC_MODULE.
• An SC_MODULE cannot be derived from another SC_MODULE.
• Vivado HLS does not support SC_THREAD.
• Vivado HLS supports the clocked version SC_CTHREAD.

#### *Virtual Functions*

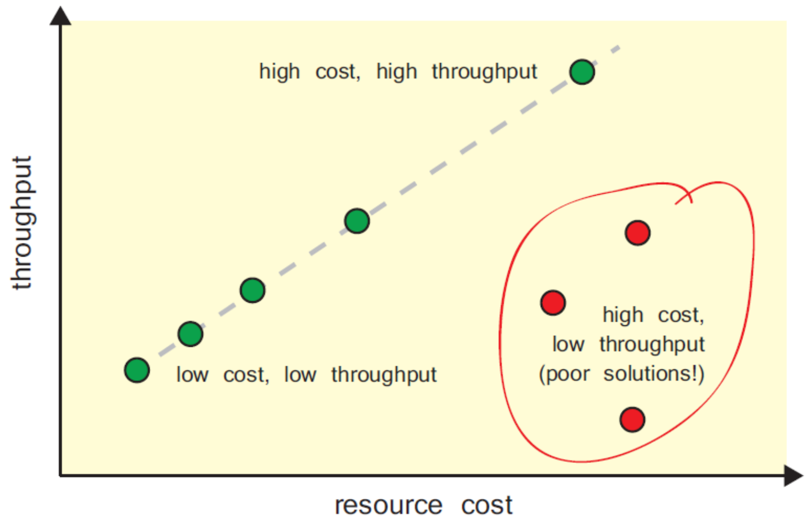
Vivado HLS does not support virtual functions.

#### *Top-Level Interface Ports*

Vivado HLS does not support reading a sc_out port.

## 2. Exploring the Design Space.

The figure below illustrates the trade-off between speed and area when implementing a system in hardware. Increasing the speed of a system will increase the resources necessary to implement the system making it will be more expensive and this is illustrated with the green circles. The red circles are architectures which are using more resources without increasing the speed and they should be discarded. Once we have implemented the system in hardware we have to evaluate and optimize the possible architectures and finally chose the one which meet our design requirements and constrains.

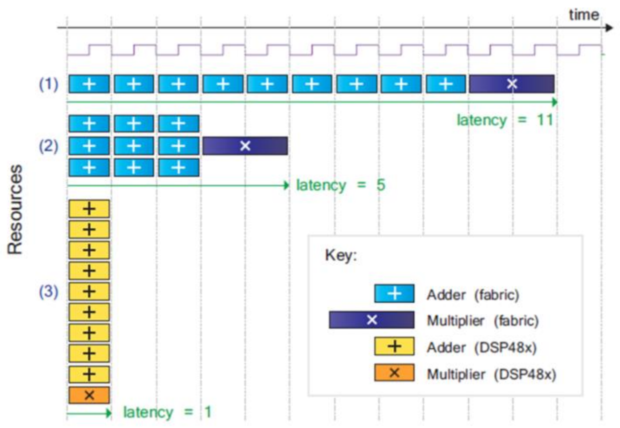


[1, p.260]

### 2.1 Comparison of three possible outcomes from HLS

The figure bellows illustrates an example of how optimization can be done. The average of an array of ten numbers is calculated where nine addition operations and one multiplication are performed.
There are three possible options to do that:

- The first will make the calculations serially using fewest resources and having the lowest throughput (11 clock cycles). Vivado HLS optimize area by default so this is the configuration that the tool will perform when generating the RTL code.
- The second option will make three additions in parallel and finally a multiplication. There is a balance between resources and throughput.
- The third option uses most resources (9 DSP48x) but it will complete the operation in one clock cycle.



[1, p.275]

***Optimisations***
The designer has the possibility of optimizing the design using two methods:

- Constraints

There are some limits in the design like minimum clock period, resource utilisation, etc.

- Directives

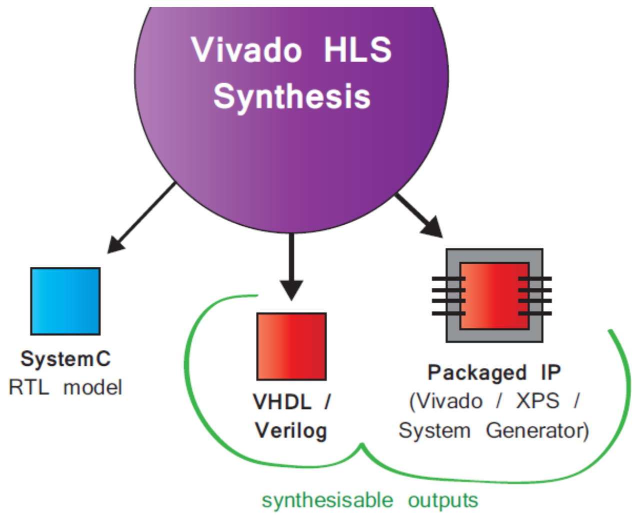
They can be directives about how to treat loops and arrays for example and they have a high influence in the generation of RTL code.

## 3. Vivado HLS: A Closer Look

### 3.1 Overview of Vivado HLS synthesis process

The figure bellow illustrates the different outputs produced after Vivado HLS has generated the RTL code. There are three different outputs that can be chosen, VHDL or Verilog languages and SystemC which is only used for simulation and not to be implemented in hardware.
The output can be packaged in an IP blocks and used later in Vivado, XPS project or System Generator.

.

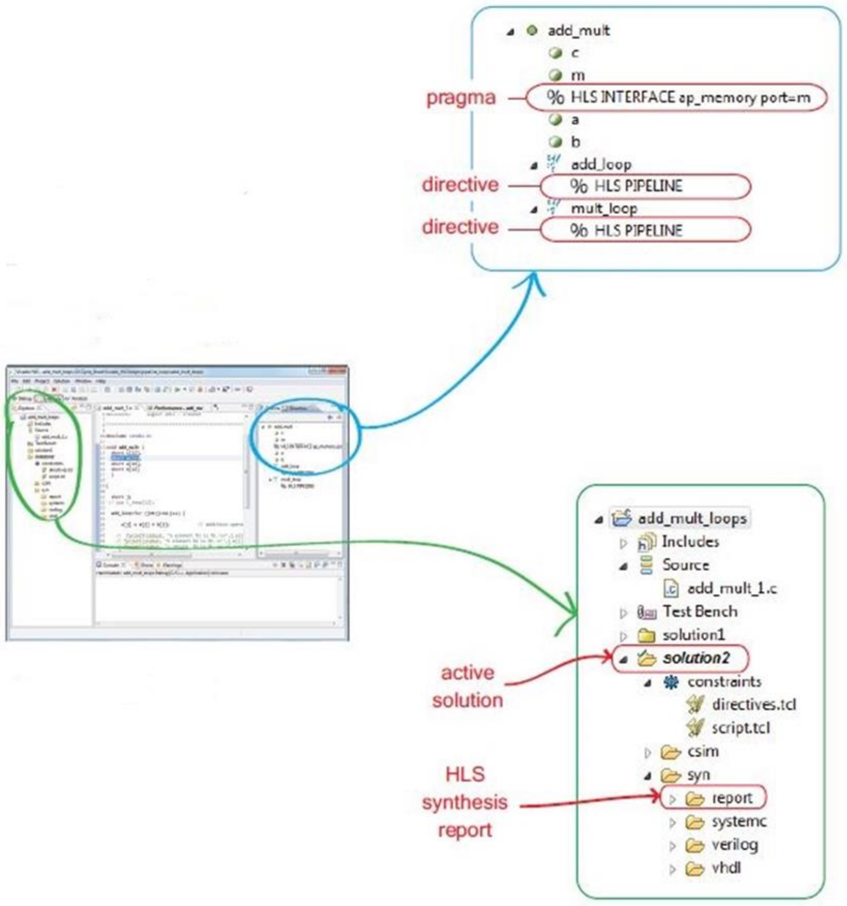


[1, p.283]

After the synthesis process is completed different folders are automatically generated. The folder “report” contains the report generated with the timing, resources and interfaces and the folders “SystemC”, ”Verilog” and “vhdl” contain the RTL code generated in the different formats.

## 3.2 Synthesis Perspective

The Vivado HLS GUI allows managing projects, editing the code, and debugging. A detailed description about the figure and the different perspectives is performed below it.



[1, p.285]

***Vivado HLS User Interfaces***

The Vivado HLS tool has a Graphical User Interface and a Command Line Interface which provides access to the same functionality.

***Graphical User Interface***

The GUI provides three different perspectives: Debug, Synthesis and Analysis.

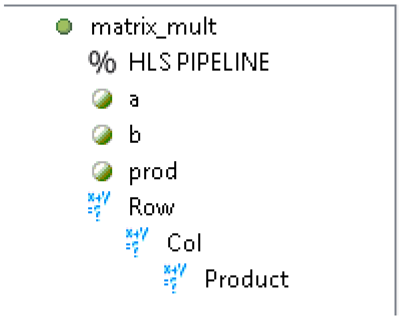
***Synthesis Perspective: Project Organisation***

The project organization view contains the source files with the C code and the testbench files which we use for verification. One of the most important characteristics of Vivado HLS is that allow us to generate different solutions by changing constrains and directives. Each solution contains a folder with a synthesis report and three folders with VHDL, Verilog and SystemC RTL code.

***Synthesis Perspective: Directives Pane***

This pane is used to manage the directives.

If we chose to write the directives in the code then it will appear with the symbol "#", if not they will appear like "%".

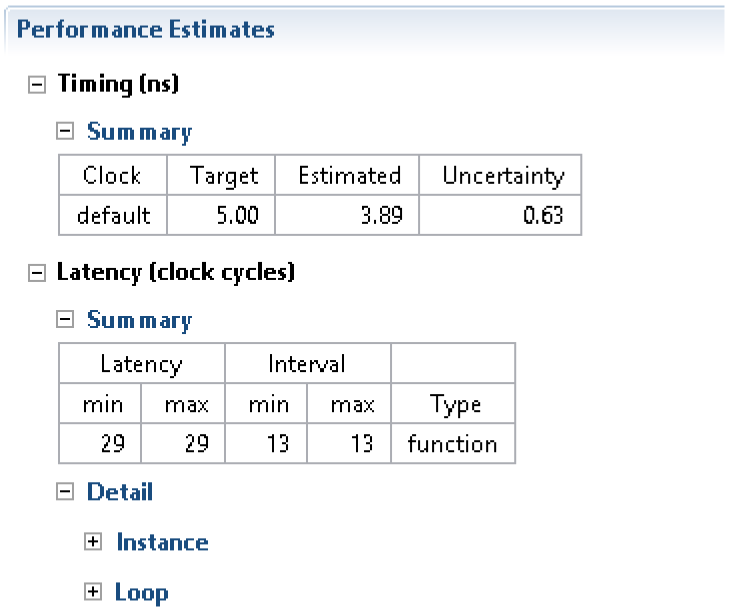


matrix_mult
% HLS PIPELINE
a
b
prod
Row
Col
Product

***Synthesis Perspective: Synthesis Report***

The synthesis report includes the following information:

- " Clock information, and comparison with constraints;
- Details of loops identified within the code (e.g. trip count, latency per loop);
- Latency statistics;



Performance Estimates

Timing (ns)

Summary

| Clock | Target | Estimated | Uncertainty |
|---|---|---|---|
| default | 5.00 | 3.89 | 0.63 |

Latency (clock cycles)

Summary

| Latency | | Interval | | |
|---|---|---|---|---|
| min | max | min | max | Type |
| 29 | 29 | 13 | 13 | function |

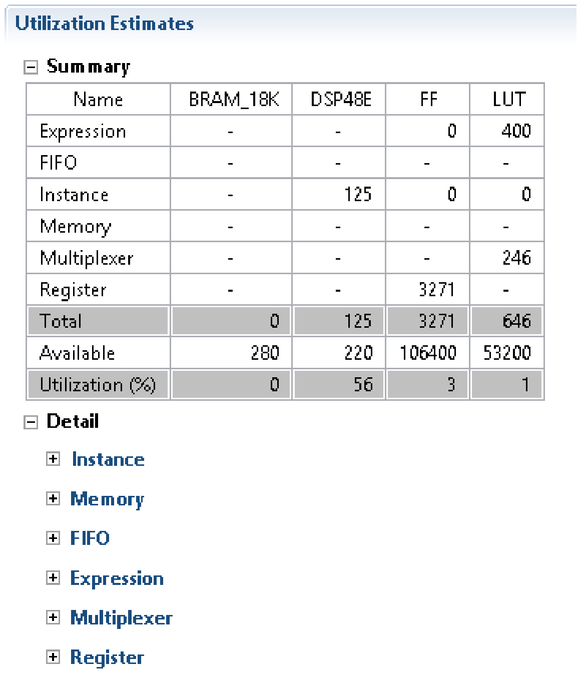
Detail

Instance

Loop

- The estimated cost of the implementation, in terms of different PL resources.

To synthesize the RTL code for the matrix multiplication have been used the following resources: 125 DSP48E's 3271 FF's and 646 LUT's.



Utilization Estimates

⊟ Summary

| Name | BRAM_18K | DSP48E | FF | LUT |
|---|---|---|---|---|
| Expression | - | - | 0 | 400 |
| FIFO | - | - | - | - |
| Instance | - | 125 | 0 | 0 |
| Memory | - | - | - | - |
| Multiplexer | - | - | - | 246 |
| Register | - | - | 3271 | - |
| Total | 0 | 125 | 3271 | 646 |
| Available | 280 | 220 | 106400 | 53200 |
| Utilization (%) | 0 | 56 | 3 | 1 |

⊟ Detail

⊞ Instance

⊞ Memory

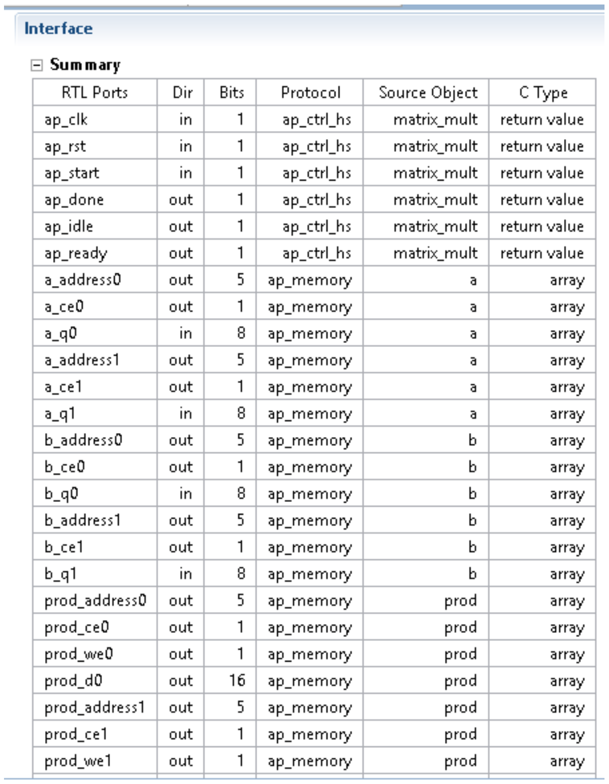
⊞ FIFO

⊞ Expression

⊞ Multiplexer

⊞ Register

- The figure below shows a list of the synthesised RTL interface ports, including their directions, dimensions and associated protocols that have been used to synthesize the RTL code for the matrix multiplication



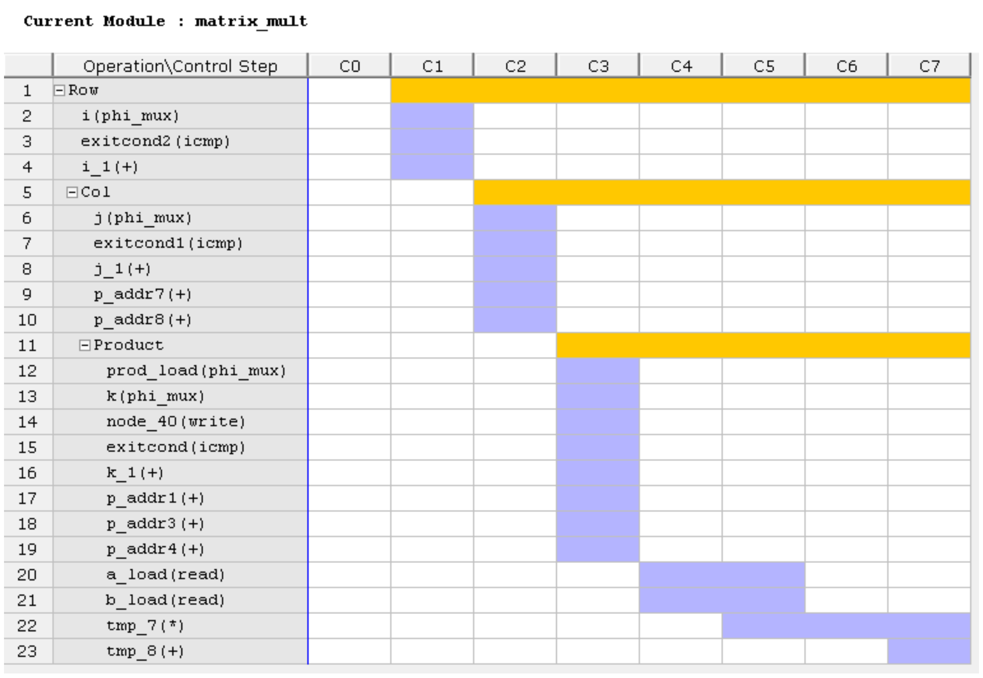
Interface

⊟ Summary

| RTL Ports | Dir | Bits | Protocol | Source Object | C Type |
|---|---|---|---|---|---|
| ap_clk | in | 1 | ap_ctrl_hs | matrix_mult | return value |
| ap_rst | in | 1 | ap_ctrl_hs | matrix_mult | return value |
| ap_start | in | 1 | ap_ctrl_hs | matrix_mult | return value |
| ap_done | out | 1 | ap_ctrl_hs | matrix_mult | return value |
| ap_idle | out | 1 | ap_ctrl_hs | matrix_mult | return value |
| ap_ready | out | 1 | ap_ctrl_hs | matrix_mult | return value |
| a_address0 | out | 5 | ap_memory | a | array |
| a_ce0 | out | 1 | ap_memory | a | array |
| a_q0 | in | 8 | ap_memory | a | array |
| a_address1 | out | 5 | ap_memory | a | array |
| a_ce1 | out | 1 | ap_memory | a | array |
| a_q1 | in | 8 | ap_memory | a | array |
| b_address0 | out | 5 | ap_memory | b | array |
| b_ce0 | out | 1 | ap_memory | b | array |
| b_q0 | in | 8 | ap_memory | b | array |
| b_address1 | out | 5 | ap_memory | b | array |
| b_ce1 | out | 1 | ap_memory | b | array |
| b_q1 | in | 8 | ap_memory | b | array |
| prod_address0 | out | 5 | ap_memory | prod | array |
| prod_ce0 | out | 1 | ap_memory | prod | array |
| prod_we0 | out | 1 | ap_memory | prod | array |
| prod_d0 | out | 16 | ap_memory | prod | array |
| prod_address1 | out | 5 | ap_memory | prod | array |
| prod_ce1 | out | 1 | ap_memory | prod | array |
| prod_we1 | out | 1 | ap_memory | prod | array |

## 3.3 Analysis Perspective

Analysis perspective gives an idea of how the design has been synthesised and can be used to optimize the design (for example the loops).
The figure below shows the Performance view where the different operations of the matrix multiplications within the code are scheduled as clock cycles. If we right click in the cell and select "Go to source" then we will go to the source code that the cell belong to.
We can see that reading the matrices rows happens in cycle 1, reading the matrices columns in cycle 2 and the multiplication occurs in cycle 3.



**Current Module : matrix_mult**

| | Operation\Control Step | C0 | C1 | C2 | C3 | C4 | C5 | C6 | C7 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | ⊟Row | | 🟨 | 🟨 | 🟨 | 🟨 | 🟨 | 🟨 | 🟨 |
| 2 | i(phi_mux) | | 🟦 | | | | | | |
| 3 | exitcond2(icmp) | | 🟦 | | | | | | |
| 4 | i_1(+) | | 🟦 | | | | | | |
| 5 | ⊟Col | | | 🟨 | 🟨 | 🟨 | 🟨 | 🟨 | 🟨 |
| 6 | j(phi_mux) | | | 🟦 | | | | | |
| 7 | exitcond1(icmp) | | | 🟦 | | | | | |
| 8 | j_1(+) | | | 🟦 | | | | | |
| 9 | p_addr7(+) | | | 🟦 | | | | | |
| 10 | p_addr8(+) | | | 🟦 | | | | | |
| 11 | ⊟Product | | | | 🟨 | 🟨 | 🟨 | 🟨 | 🟨 |
| 12 | prod_load(phi_mux) | | | | 🟦 | | | | |
| 13 | k(phi_mux) | | | | 🟦 | | | | |
| 14 | node_40(write) | | | | 🟦 | | | | |
| 15 | exitcond(icmp) | | | | 🟦 | | | | |
| 16 | k_1(+) | | | | 🟦 | | | | |
| 17 | p_addr1(+) | | | | 🟦 | | | | |
| 18 | p_addr3(+) | | | | 🟦 | | | | |
| 19 | p_addr4(+) | | | | 🟦 | | | | |
| 20 | a_load(read) | | | | | 🟦 | 🟦 | | |
| 21 | b_load(read) | | | | | 🟦 | 🟦 | | |
| 22 | tmp_7(*) | | | | | | 🟦 | 🟦 | |
| 23 | tmp_8(+) | | | | | | | | 🟦 |

## 3.4 Vivado HLS Arbitrary Precision Data Types

It is very important to specify the data types used in our implementation in order to optimize and be efficient in the implementation of hardware. If we do not specify the number of bits which will be used then resources used to implement RTL into hardware will increase making the system more expensive.
Vivado HLS provides arbitrary precision data types for C / C++ and special arbitrary precision data types for SystemC.

```
// C code example
#include "ap_cint.h"

void top_level_function (..)
{
      // declarations
        int6 small_signed;
        uint10 big_unsigned;
        int22 vbig_signed;
        ...
}
```

```
// C++ code example
#include "ap_int.h"

void top_level_function (..)
{
      // declarations
         ap_int<6> small_signed;
         ap_uint<10> big_unsigned;
         ap_int<22> vbig_signed;
         ...
}
```

[1, p.290]

## Arbitrary Precision Integer Types

Figure above shows the different data types in used in C, C++ and SystemC, the header required and a small description. In C, C++ the variable N can take values from 1 bit to 1024 bits and for SystemC the variable W can take values from 1 to 64 bits the small and from 1 to 512 bits the large.

| Language | Integer Data Type | Description | Required Header |
|---|---|---|---|
| C | int*N* (e.g. `int7`) | signed integer of *N* bits precision | `#include "ap_cint.h"` |
| | uint*N* (e.g. `uint7`) | unsigned integer of *N* bits precision | |
| C++ | ap_int<*N*> (e.g. `ap_int<7>`) | signed integer of *N* bits precision | `#include "ap_int.h"` |
| | ap_uint<*N*> (e.g. `ap_uint<7>`) | unsigned integer of *N* bits precision | |

SystemC

| SystemC Data Type | Description | Required Preamble |
|---|---|---|
| sc_int<*W*><br>sc_bigint<*W*> | signed integer:<br>(up to 64 bits)<br>(up to 512 bits) | `#include "systemc.h"` |
| sc_uint<*W*><br>sc_ubigint<*W*> | unsigned integer:<br>(up to 64 bits)<br>(up to 512 bits) | |

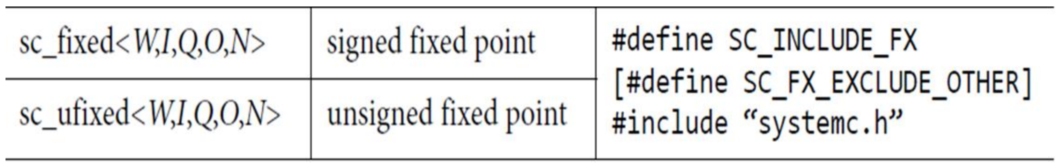
[1, p.290]

## Arbitrary Precision Fixed Point Types

Vivado HLS supports fixed point arithmetic for C++ and SystemC. The wordlength (W) is composed by number of integer bits (I) plus the number of fractional bits (B), which means that W = I+B. Q  specifies the quantisation mode; O specifies the overflow mode and N the number of saturation bits in overflow wrap modes where the N most significant bits are set to 1.

| Language | Fixed Point Data Type | Description | Required Header |
|---|---|---|---|
| C++ | ap_fixed<*W,I,Q,O,N*> | Signed fixed point number of *I* integer bits and *W-I* fractional bits. | `#include "ap_fixed.h"` |
| | ap_ufixed<*W,I,Q,O,N*> | Unsigned fixed point number of *I* integer bits and *W-I* fractional bits. | |

SystemC

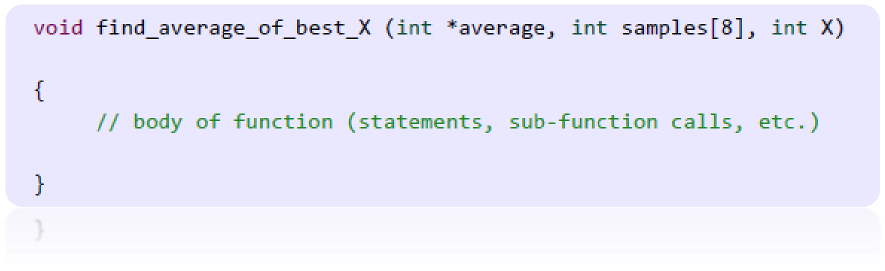


| | | |
|---|---|---|
| sc_fixed<W,I,Q,O,N> | signed fixed point | #define SC_INCLUDE_FX [#define SC_FX_EXCLUDE_OTHER] #include "systemc.h" |
| sc_ufixed<W,I,Q,O,N> | unsigned fixed point | |

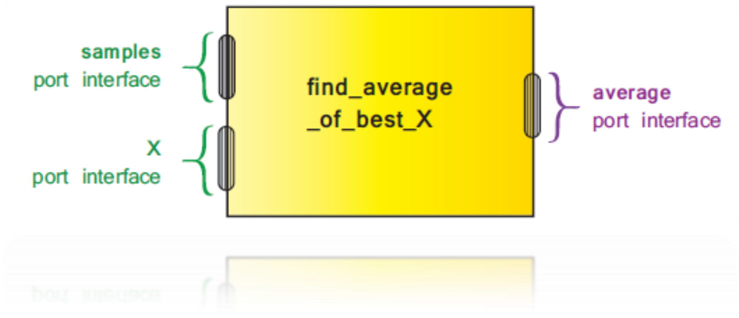
[1, p.291]

## 3.5 Interface Specification and Synthesis

In Vivado HLS, for C and C++, is possible to make a synthesis of the interface and in SystemC the synthesis of the interface should be done manually.
Data ports after the RTL has been synthesized are the parameters passed to the function and the value returned and each port is associated with a protocol.
Port interface is composed by ports and protocols and they are used to communicate with other subsystems.
The figure below shows a function that returns the average value of an array of integers. It takes two parameters as inputs (x and samples) and one parameter as output (average) which they will implemented with a port interface after the RTL has been synthesized.



```
void find_average_of_best_X (int *average, int samples[8], int X)

{
     // body of function (statements, sub-function calls, etc.)

}
```

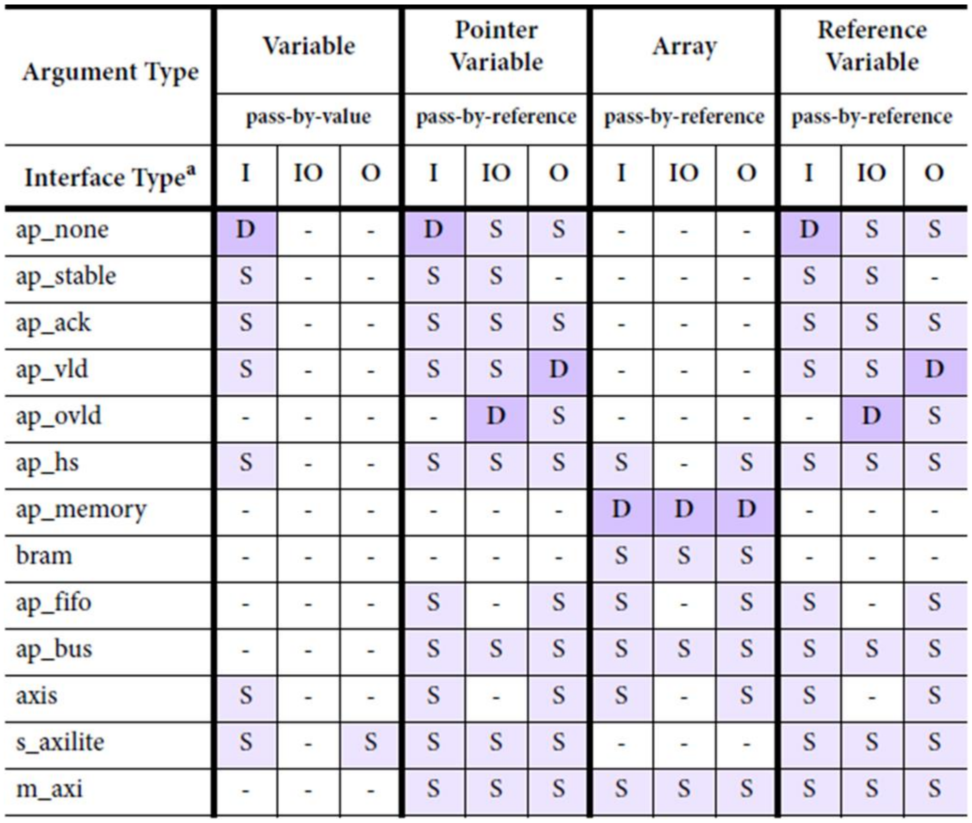


[1, p.296]

***Port Interface Protocol Types***

The figure shows the protocols available for Vivado HLS. The protocol has to be chosen based on the port direction, the type of argument and if the argument is passed by value or is passed by reference. The protocol can be selected by specifying it properly in the directive and the "S"'s in the table are the possible protocols available for each case. If there is no mention about the protocol in the directive then the default protocol noted by "D" will be implemented automatically by Vivado HLS.

The figure bellow shows which protocols are used in the implementation of the find_average_of_best_function_X function. . When we pass an array as an argument the available protocols are ap_hs, ap_memory, bram, ap_fifo, ap_bus, axis and m_axi, and the default one is ap_memory. The default for X is ap_none, and the default for average is ap_vld protocol.



| Argument Type | Variable | | | Pointer Variable | | | Array | | | Reference Variable | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | pass-by-value | | | pass-by-reference | | | pass-by-reference | | | pass-by-reference | | |
| Interface Type[a] | I | IO | O | I | IO | O | I | IO | O | I | IO | O |
| ap_none | D | - | - | D | S | S | - | - | - | D | S | S |
| ap_stable | S | - | - | S | S | - | - | - | - | S | S | - |
| ap_ack | S | - | - | S | S | S | - | - | - | S | S | S |
| ap_vld | S | - | - | S | S | D | - | - | - | S | S | D |
| ap_ovld | - | - | - | - | D | S | - | - | - | - | D | S |
| ap_hs | S | - | - | S | S | S | S | - | S | S | S | S |
| ap_memory | - | - | - | - | - | - | D | D | D | - | - | - |
| bram | - | - | - | - | - | - | S | S | S | - | - | - |
| ap_fifo | - | - | - | S | - | S | S | - | S | S | - | S |
| ap_bus | - | - | - | S | S | S | S | S | S | S | S | S |
| axis | S | - | - | S | - | S | S | - | S | S | - | S |
| s_axilite | S | - | S | S | S | S | - | - | - | S | S | S |
| m_axi | - | - | - | S | S | S | S | S | S | S | S | S |

[1, p.300]

“The protocols below are the default protocols:

**ap_none**

This protocol has no explicit interface protocol, no additional control signals, and no associated hardware overhead.

**ap_vld**

An additional port is provided to validate data. For input ports, a *valid* input control port is added, which qualifies input data as valid. For output ports, a *valid* output port is added, and asserted on clock cycles when output data is valid.

**ap_ovld**

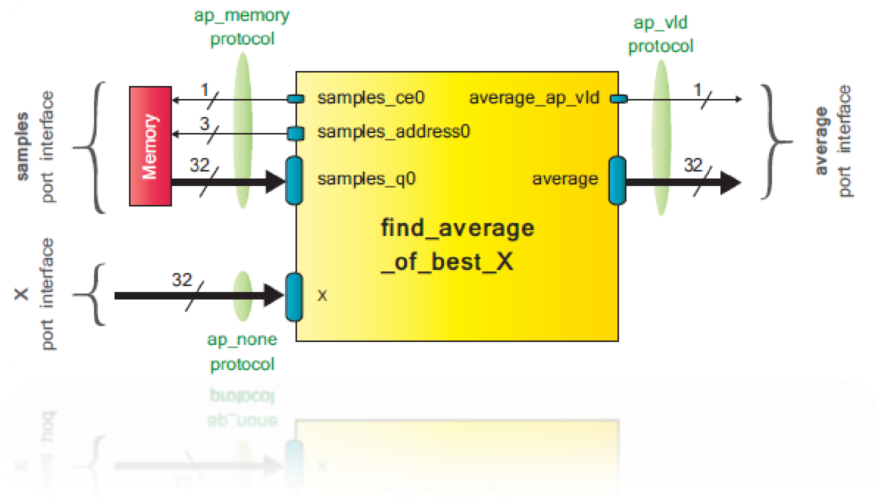
This protocol is the same as *ap_vld*, but can only be implemented on output ports, or the output portion of an inout (bidirectional) port.

**ap_memory**

This memory-based protocol supports random access transactions with a memory, and can be used for both input, output, and bidirectional ports. The only argument type compatible with this protocol is the

array type, which corresponds with the structure of a memory. The *ap_memory* protocol requires control signals for clock and write enables, as well as an address port."
[1, p.299]

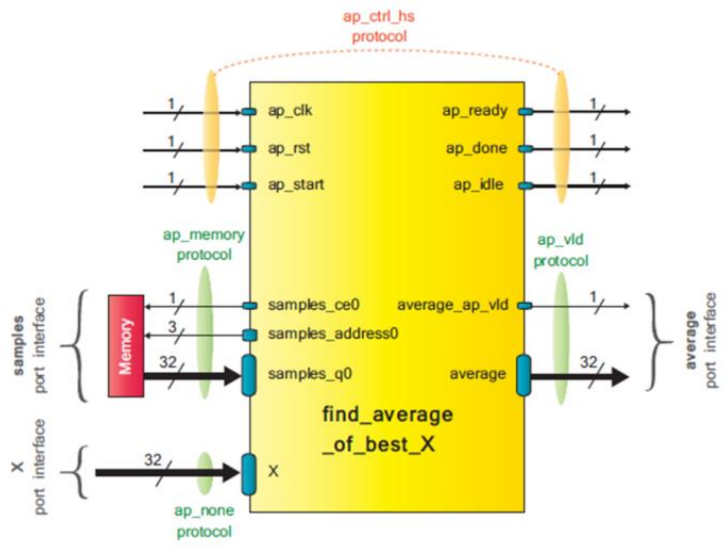
The figure bellow illustrates all the inputs and outputs of the function with the number of bits and the different protocols.



[1, p.301]

## 3.6 Block-Level Interface Ports and Protocols

A block-level protocol can be added to the design when we want to integrate the block into a system and we want to manage the flow of data between the block and the system or between different blocks. There are three types of block-level protocol, ap_ctrl_none., ap_ctrl_hs and ap_ctrl_chain. Two input signals, ap_clk and ap_rst, are added when a block level protocol is implemented and they are used to synchronize the flow of data between blocks and to perform a reset.
AXI4-Lite bus interface can be used in the block-level interface protocol which will add new signals to the block.



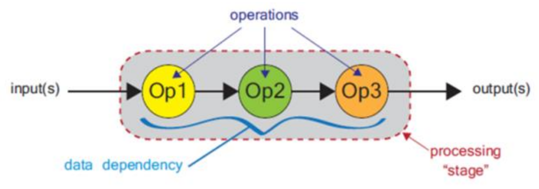
[1, p.304]

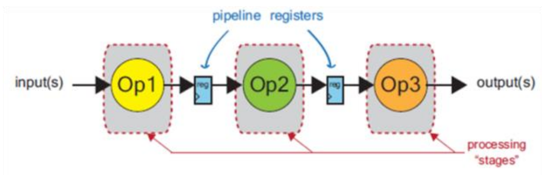
## 3.7 Pipelining

Pipelining is the segmentation of logical processing stages which allows the operations to overlap and be processed in parallel to increase the throughput.
We can apply pipelining as a directive in Vivado HLS in functions and loops.

***Algorithm Execution and Data Dependency***

As an example we see the upper part of the figure below which shows three different operations Op1, Op2, and Op3 executed sequentially. There is a dependency between the operations because in order Op3 to be executed Op2 must be finished because the input of Op2 is the output of Op3. The same case applies to the dependency between Op2 and Op1, between Op1 and the input and between Op3 and the output. This dependency cause that only one output can be executed in each clock cycle.

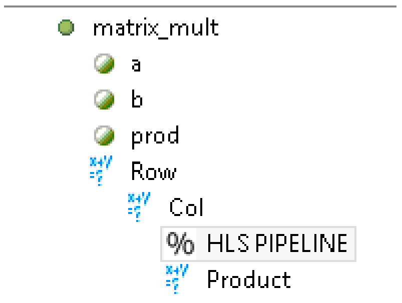




[1, p.313]

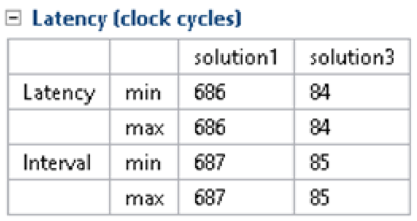
In the lower part of the figure two registers have been added between the operations which allow storing the results of each operation. This breaks the dependency between operations increasing the throughput and reducing the latency because the operations can be executed simultaneously.

In the example of matrix multiplication a pipeline is done to execute the columns simultaneously.



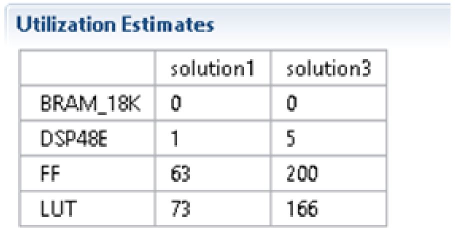
```
matrix_mult
    a
    b
    prod
    Row
        Col
            % HLS PIPELINE
            Product
```

It is possible to compare the timing, latency and hardware resources used in each implementation. In the example bellow the pipeline solution 3 is much faster than solution 1 with a latency of 84 clock cycles instead of 686 but it uses 4 DSP48E, 137 FF's and 93 LUT's more than solution 1. There is always the trade-off between throughput and hardware resources.



Latency (clock cycles)

| | | solution1 | solution3 |
|---|---|---|---|
| Latency | min | 686 | 84 |
| | max | 686 | 84 |
| Interval | min | 687 | 85 |
| | max | 687 | 85 |

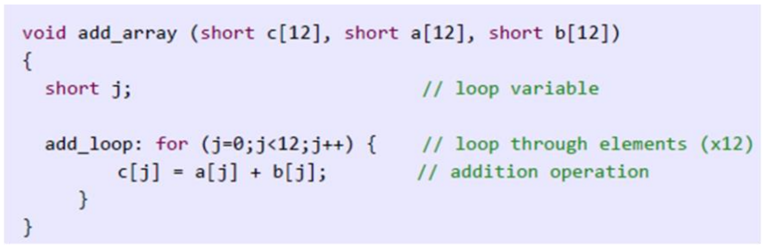


Utilization Estimates

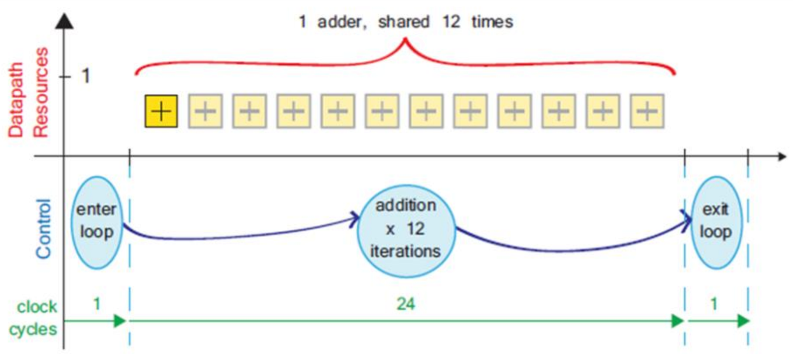
| | solution1 | solution3 |
|---|---|---|
| BRAM_18K | 0 | 0 |
| DSP48E | 1 | 5 |
| FF | 63 | 200 |
| LUT | 73 | 166 |

## 3.8 Loop Synthesis

Loops are well supported by Vivado HLS for hardware synthesis and optimizations can be done in the architecture by using directives.
By default Vivado HLS will implement the loops with the fewest resources which mean that will be executed sequentially.
The figure below shows an example of a function which adds two arrays of 12 elements.
By default Vivado HLS will synthesize the loop with a latency of 26 clock cycles: 2 cycles each for 12 iterations and two clock cycles for entering and exiting the loop.



```
void add_array (short c[12], short a[12], short b[12])
{
  short j;                            // loop variable

  add_loop: for (j=0;j<12;j++) {     // loop through elements (x12)
         c[j] = a[j] + b[j];         // addition operation
     }
}
```



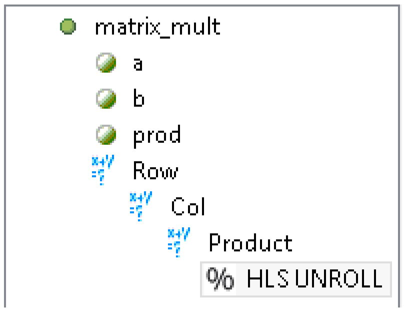
[1, p.320]

Directives can be used too to optimize the loops.

***Unrolled***

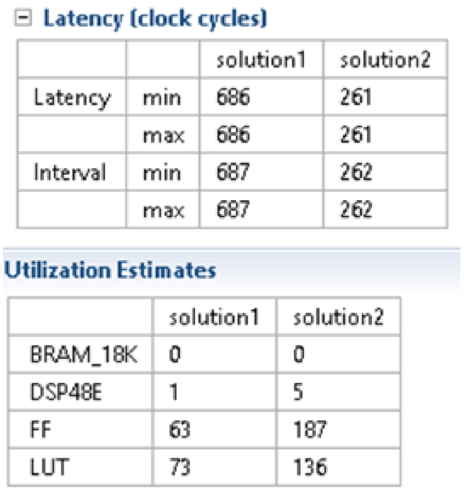
Unrolled use up to N times more resources than the rolled loop synthesis where N is the number of loop iterations but the throughput is increased.

The unrolled directive is set to the product operation of the matrix example.



```
matrix_mult
    a
    b
    prod
    Row
        Col
            Product
                % HLS UNROLL
```

The unrolled solution 2 is much faster than solution 1 with a latency of 261 clock cycles instead of 686 but it uses 4 DSP48E, 124 FF's and 63 LUT's more in hardware resources than solution 1.



Latency (clock cycles)

| | | solution1 | solution2 |
|---|---|---|---|
| Latency | min | 686 | 261 |
| | max | 686 | 261 |
| Interval | min | 687 | 262 |
| | max | 687 | 262 |

Utilization Estimates

| | solution1 | solution2 |
|---|---|---|
| BRAM_18K | 0 | 0 |
| DSP48E | 1 | 5 |
| FF | 63 | 187 |
| LUT | 73 | 136 |

***Partially unrolled***

It is a balance between rolled and unrolled synthesis and it can be used for example when we need to meet some timing requirements where we need to roll some of the loops in order to satisfy those requirements.

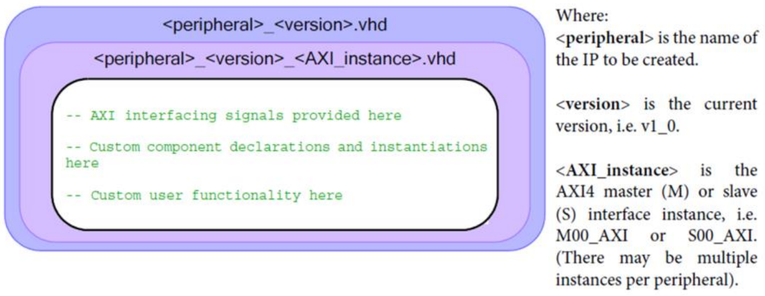
## 4. IP Reuse and Integration

Xilinx provide the necessary tools to create custom IP blocks which can be integrated in embedded system designs via IP Integrator. This allows the possibility to reuse IP blocks saving time and money when designing a system.

### 4.1 IP Core Design Methods

***HDL***

A HDL IP block can be created writing the code in VHDL or Verilog. Using this method allows a full control of all the implementation details but it takes in general much time.
A top-level AXI interface is implemented in order for the IP block to communicate with the other subsystems.


<peripheral>_<version>.vhd
<peripheral>_<version>_<AXI_instance>.vhd
-- AXI interfacing signals provided here
-- Custom component declarations and instantiations here
-- Custom user functionality here
Where:
**<peripheral>** is the name of the IP to be created.
**<version>** is the current version, i.e. v1_0.
**<AXI_instance>** is the AXI4 master (M) or slave (S) interface instance, i.e. M00_AXI or S00_AXI. (There may be multiple instances per peripheral).

[1, p.240]

***System Generator***

System Generator uses the MathWorks Simulink design platform to design systems in FPGA.
IP module implemented with System generator can be added to Vivado IP Catalog with IP Packager and used later in other designs.
IP catalog compilation tool generates automatically testbench files which can be used to verify that the IP block behaves correctly. A HTML file with information about the IP and how can be integrated in a system is automatically generated by the tool too.

***HDL Coder***

HDL Coder generates HDL code from MATLAB functions and Simulink models which can be used to create IP cores for posterior reuse. The tool allows verifying the generated HDL code by comparing it with the original MATLAB/Simulink model.
The tool has some limitations because not all the MATLAB functions and Simulink models can be used to generate HDL code.

## 5. AXI Interfacing

AXI is a protocol developed for high-performance systems which is used to communicate between the different IP cores and it is optimized to be implemented in FPGA.

### 5.1 Variations of AXI4

There are three different types of AXI4 interface:

- AXI4: it is a high-performance interface which allows bursts of up to 256 data transfer cycles per address phase.
- AXI4-Lite: it is slower than AXI4 interface and it does not support burst so only it is only possible to transfer a single data transfer per transaction.

Possible applications are:
Audio and Video/Image Processing, Communications/Networking, Embedded Processing.

- AXI4-Stream: it is the fastest and it was design for the transmission of streaming data because allows bursts of an unlimited amount of data.

Possible applications are:
DSP, Streaming Audio and Streaming Video/Image Processing, Encoders/decoders, interleavers / deinterleavers, Streaming FIFO, Ethernet peripheral,

## 5.2 AXI Architecture

The AXI protocol is used to transfer data between one or more masters to one or more slave through the AXI interconnect.
These are the channels used for the transaction:

*Address Channels*

Contains the control information and indicates where the data has to be written to or where the data has to be read from.

*Write Data Channel*

The write data channel contains:

- a data bus from 8-1024 bits wide
- A byte lane strobe which is passed every eight bits of data to identify valid bytes.

*Read Data Channel*

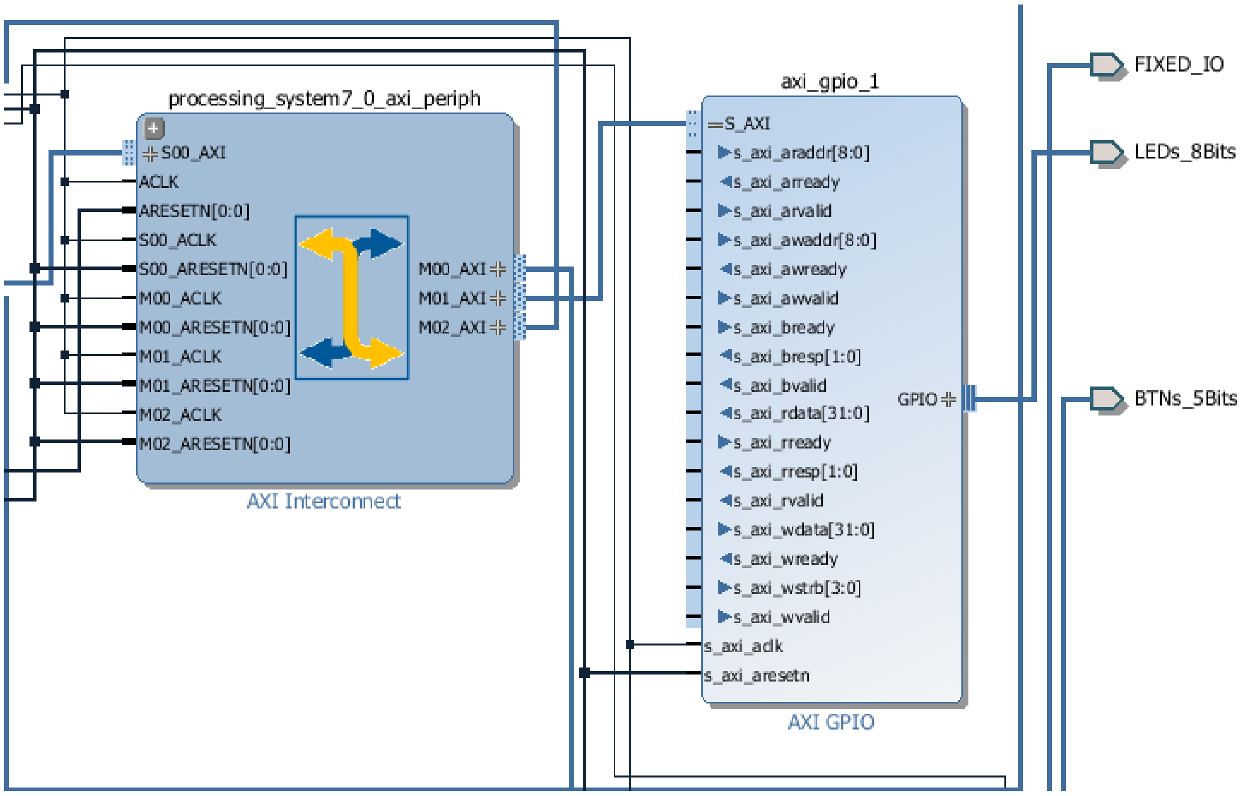
The read data channel contains:

- a data bus from 8-1024 bits wide
- A read response which is sent when the read transaction is completed to indicate that the data has been read correctly.

*Write Response Channel*

A write response is sent from the slave when each burst is completed to indicate that the data has been written correctly.

The figure below illustrates a General-purpose input/output (GPIO) added to the system and connected as a slave via an AXI interface. It is possible to see all the inputs and outputs available which be used to send and receive information.



## 6. High-Level Synthesis for FPGAs: From Prototyping to Deployment

The article gives an overview about the first steps taken to try to transform a program written in C/C ++ and SystemC into RTL code which can be used to hardware implementation and simulation. It describes in detail the AutoESL AutoPilot High-Level Synthesis Tool which was acquired by Xillinx.
The following tests are performed in the article:

- *Quality of results for DQPSK(Dual-polarization quadrature phase shift keying) receiver workload: 18.75 MSamples/second input data at 75MHz clock speed*

A comparison is made about the hardware resources needed to implement a RTL code written in hand and the synthesized by AutoESL. It is possible to see that the tool optimized better the resources than the code written manually.

| Platform | Chip Resource Utilization (Lower is Better) |
|---|---|
| AutoESL AutoPilot plus Xilinx RTL tools targeting the Xilinx XC3D3400A FPGA | 5.6% |
| Hand-written RTL code using Xilinx RTL tools targeting the Xilinx XC3SD3400A FPGA | 5.9% |

[4, p.14]

- **Sphere decoder implementation results.**

The figure bellow gives a detailed view about the hardware resources used by manual and AutoPilot implementation. The tool reduces implementation time by 9 %. Both designs were tested using ISE 12.1, in Xilinx Virtex 5 speed grade 2 at 225 MHz.

| Metric | RTL expert | AutoPilot expert | Diff. (%) |
|---|---|---|---|
| Dev. time (man-weeks) | 16.5 | 15 | -9% |
| LUTs | 32,708 | 29,060 | -11% |
| Registers | 44,885 | 31,000 | -31% |
| DSP48s | 225 | 201 | -11% |
| 18K BRAMs | 128 | 99 | -26% |

[4, p.15]

- **Matrix-Multiply Inverse implementation results**

One of the main advantages of the AutoPilot is the simplicity when implementing changes to the design. Here we can see that besides the optimization in hardware resources the tool reduces the time when changing the design in comparison with manual implementation. The figure shows the resources used to implement a 4X4 multiplication when the implementation is performed manually and with AutoPilot. After the design is changed and the multiplication is done in a 3X3 matrix and finally in a 2X2 matrix.

| Metric | 4x4 RTL | 4x4 AP | 3x3 RTL | 3x3 AP | 2x2 RTL | 2x2 AP |
|---|---|---|---|---|---|---|
| Dev. Time (man-weeks) | 4 | 4 | 1 | 0 | 1 | 0 |
| LUTs | 9,016 | 7,997 | 6,969 | 5,028 | 5,108 | 3,858 |
| Registers | 11,028 | 7,516 | 8,092 | 4,229 | 5,609 | 3,441 |
| DSP48s | 57 | 48 | 44 | 32 | 31 | 24 |
| 18K BRAMs | 16 | 22 | 14 | 18 | 12 | 14 |

[4, p.15]

## 7. Conclusion

The self-study course provided me a good knowledge about the Zynq device, specially the ZedBoard, and the different parts composing embedded devices, as well as the different types of processors.

I have learnt how to use the Vivado® High Level Synthesis tool and some of the features which the tool offers to optimize the RTL code via directives and the possibility of compering different solutions to verify for example if the requirements are met.

Also the practical exercises provided me a practical understanding about how to create a Zynq System design in Vivado and adding further Interrupt Sources. Exercises about how to designing With Vivado High Level Synthesis and creating projects in Vivado HLS have been done too.

The exercises have been used like examples though the entire report to help to understand the theory covered and some exercises have been modified to try illustrating this theory.

The Zynq Book is a very good book with a technical language not so difficult to understand which can be read by people with almost no knowledge of hardware because it covers from the basic principles to more advance hardware implementations.

The lack of knowledge about the self-study course made me use more hours than I expected but in general I feel satisfy with the work done.
The following tasks have been done during the course:

- Installing Vivado HLS with some problems because the version provided in the CD with the ZedBoard does not work with Windows 8.
- Write a course planning
- Appointments with Kim Bjerge
- Read the Zynq book once (except the Linux part) and twice the chapters related to HLS
- Do some practical exercises proposed in the book.
- Read the article “High-Level Synthesis for FPGAs: From Prototyping to Deployment”
- Read the SystemC part from “Vivado Design Suite User Guide”
- Write a report about the theory learnt in this course.
- Prepare a power point presentation for the exam

# 8. Annex

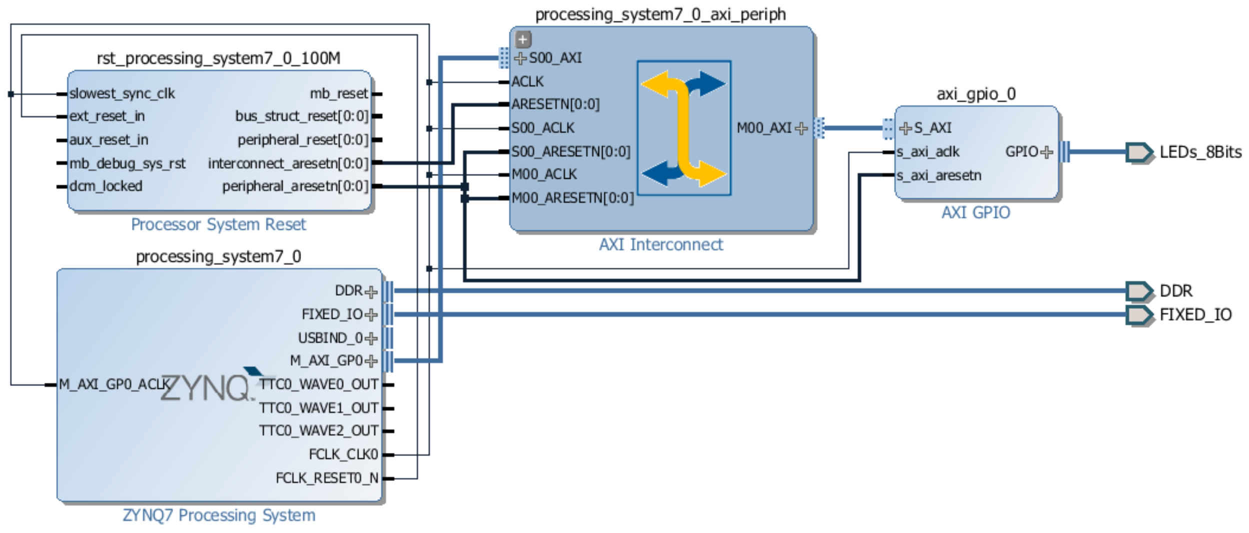
## 8.1 Exercises

These exercises are provided together with the book to have a practical knowledge about to create designs with Vivado and Vivado HLS. The version of Vivado used to develop the exercises is 2014.2 which is the latest available at this moment. Versions earlier than 2014 do not work with Windows 8 operating system.

### First Designs on Zynq

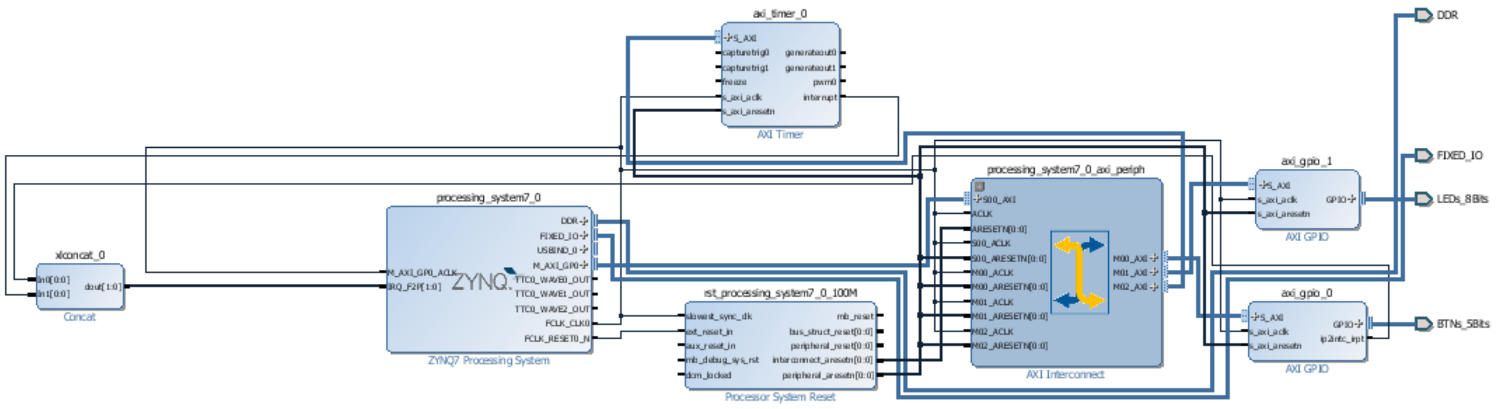
#### *Creating a Zynq System in Vivado*

The first design consists in a GPIO connected to the LEDs and to the Zynq processor via an AXI bus connection. The LEDs are controlled by a software application implemented in Xilinx SDK 2014.2.



#### *Adding a Further Interrupt Source*

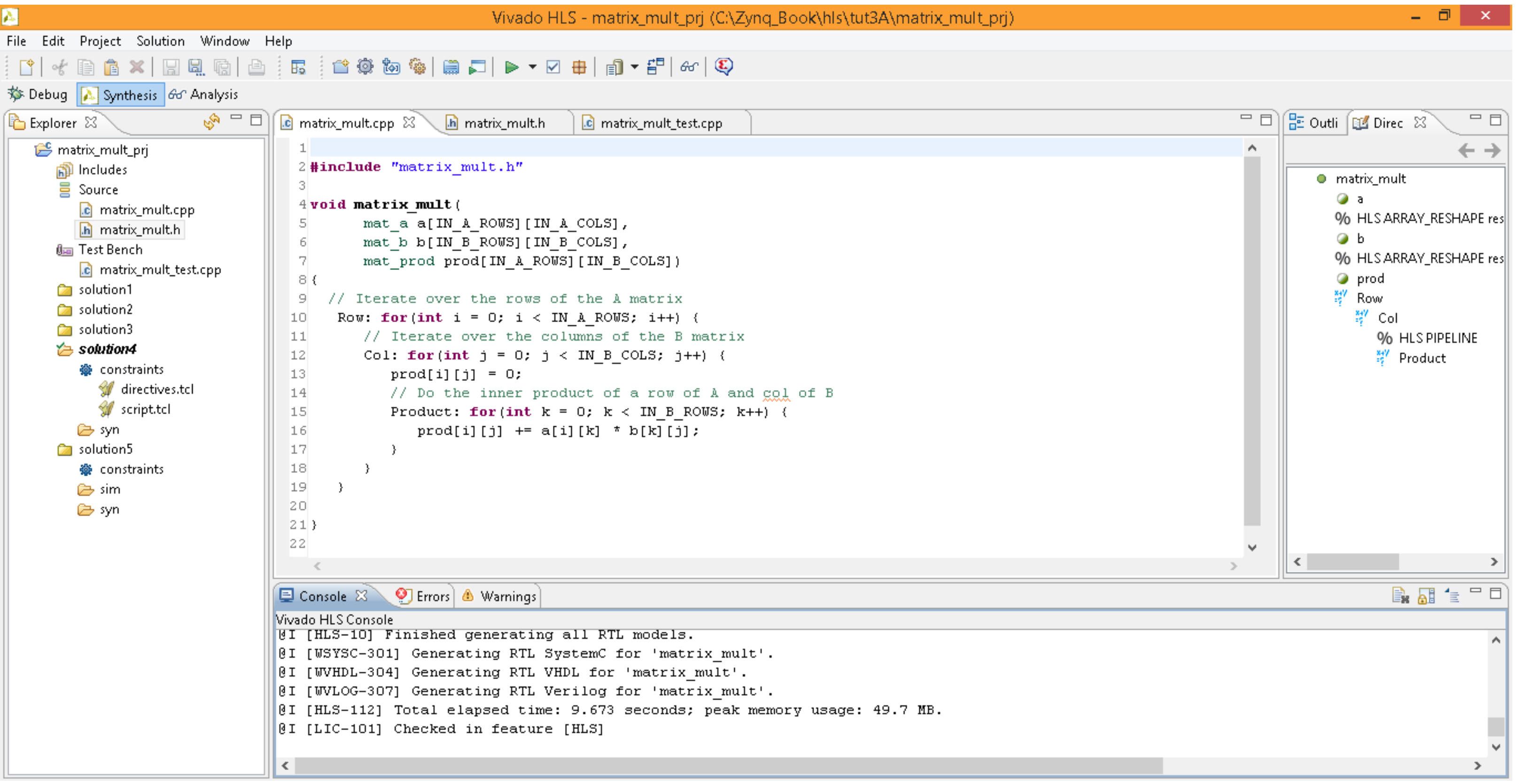
This exercise is implemented reusing the last design. Another GPIO is added to the system to generate hardware interrupts with the push buttons. Also another source of interrupt is added to the system in the form of an AXI Timer which will generate a hardware interrupt periodically (depending on the time established in the timer)



## Designing With Vivado High Level Synthesis

### *Creating Projects in Vivado HLS*

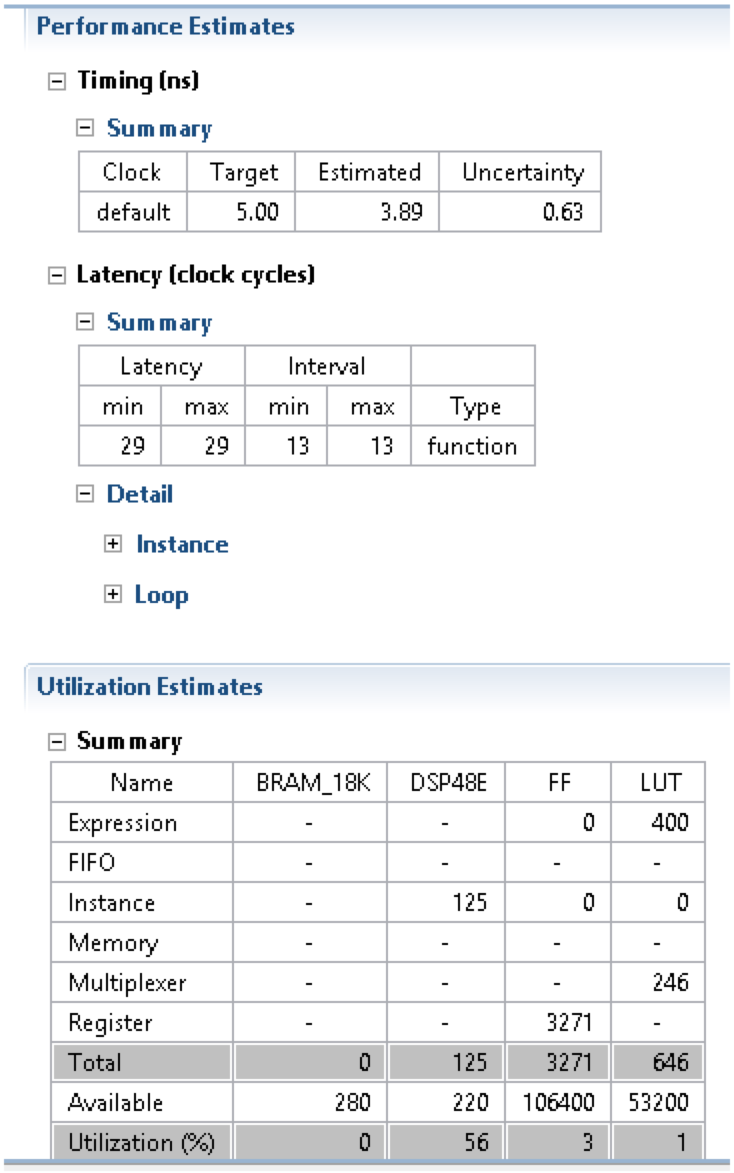
In this exercise a project is created using Vivado HLS. The following files are provided in the book:
***matrix_mult.cpp*** contains the software code of the multiplication of two matrices.
***matrix_mult.h*** header file containing the prototype function for the matrix multiplication.
***matrix_mult_test.cpp*** test bench file which calculates the product of the two matrixes using HLS hardware and software solution. Both will be compared to verify that the system has been implemented correctly.



```
#include "matrix_mult.h"

void matrix_mult(
      mat_a a[IN_A_ROWS][IN_A_COLS],
      mat_b b[IN_B_ROWS][IN_B_COLS],
      mat_prod prod[IN_A_ROWS][IN_B_COLS])
{
  // Iterate over the rows of the A matrix
   Row: for(int i = 0; i < IN_A_ROWS; i++) {
      // Iterate over the columns of the B matrix
      Col: for(int j = 0; j < IN_B_COLS; j++) {
         prod[i][j] = 0;
         // Do the inner product of a row of A and col of B
         Product: for(int k = 0; k < IN_B_ROWS; k++) {
            prod[i][j] += a[i][k] * b[k][j];
         }
      }
   }
}
```

```
@I [HLS-10] Finished generating all RTL models.
@I [WSYSC-301] Generating RTL SystemC for 'matrix_mult'.
@I [WVHDL-304] Generating RTL VHDL for 'matrix_mult'.
@I [WVLOG-307] Generating RTL Verilog for 'matrix_mult'.
@I [HLS-112] Total elapsed time: 9.673 seconds; peak memory usage: 49.7 MB.
@I [LIC-101] Checked in feature [HLS]
```

At the same time that RTL code is generated from the C++ code a Synthesis Report is generated too. This contains information about timing, latency and the resources used in the FPGA to implement the hardware.



Performance Estimates

⊟ Timing (ns)

⊟ Summary

| Clock | Target | Estimated | Uncertainty |
|---|---|---|---|
| default | 5.00 | 3.89 | 0.63 |

⊟ Latency (clock cycles)

⊟ Summary

| Latency | | Interval | | |
|---|---|---|---|---|
| min | max | min | max | Type |
| 29 | 29 | 13 | 13 | function |

⊟ Detail

⊞ Instance

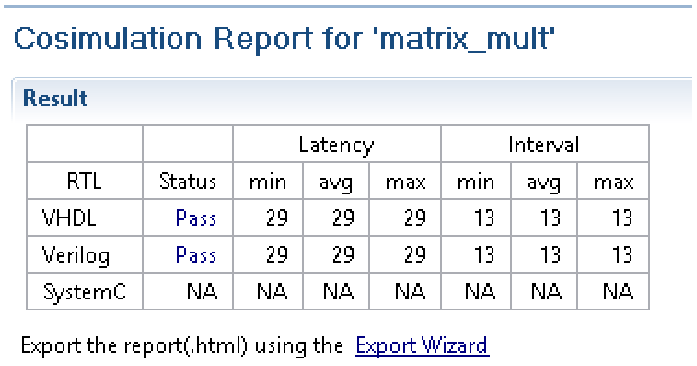
⊞ Loop

Utilization Estimates

⊟ Summary

| Name | BRAM_18K | DSP48E | FF | LUT |
|---|---|---|---|---|
| Expression | - | - | 0 | 400 |
| FIFO | - | - | - | - |
| Instance | - | 125 | 0 | 0 |
| Memory | - | - | - | - |
| Multiplexer | - | - | - | 246 |
| Register | - | - | 3271 | - |
| Total | 0 | 125 | 3271 | 646 |
| Available | 280 | 220 | 106400 | 53200 |
| Utilization (%) | 0 | 56 | 3 | 1 |

A Cosimulation Report is generated after running C/RTL cosimulation which allows verifying that the synthesised RTL produce the same outputs like the C++ code used to generate the RTL code.



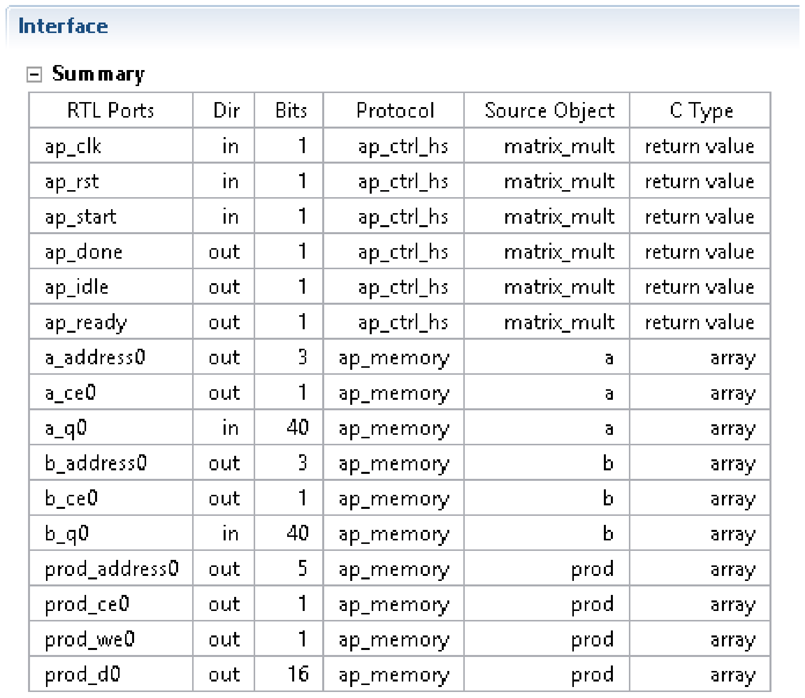
Cosimulation Report for 'matrix_mult'

Result

| | | Latency | | | Interval | | |
|---|---|---|---|---|---|---|---|
| RTL | Status | min | avg | max | min | avg | max |
| VHDL | Pass | 29 | 29 | 29 | 13 | 13 | 13 |
| Verilog | Pass | 29 | 29 | 29 | 13 | 13 | 13 |
| SystemC | NA | NA | NA | NA | NA | NA | NA |

Export the report(.html) using the Export Wizard

The synthesis report contains an interface description with all the interfaces used in the synthesis of the RTL code.

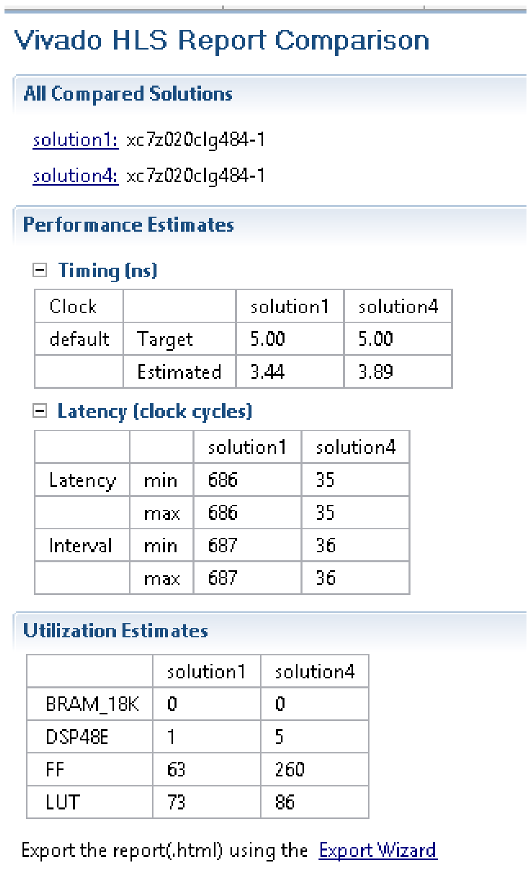


Interface

Summary

| RTL Ports | Dir | Bits | Protocol | Source Object | C Type |
|---|---|---|---|---|---|
| ap_clk | in | 1 | ap_ctrl_hs | matrix_mult | return value |
| ap_rst | in | 1 | ap_ctrl_hs | matrix_mult | return value |
| ap_start | in | 1 | ap_ctrl_hs | matrix_mult | return value |
| ap_done | out | 1 | ap_ctrl_hs | matrix_mult | return value |
| ap_idle | out | 1 | ap_ctrl_hs | matrix_mult | return value |
| ap_ready | out | 1 | ap_ctrl_hs | matrix_mult | return value |
| a_address0 | out | 3 | ap_memory | a | array |
| a_ce0 | out | 1 | ap_memory | a | array |
| a_q0 | in | 40 | ap_memory | a | array |
| b_address0 | out | 3 | ap_memory | b | array |
| b_ce0 | out | 1 | ap_memory | b | array |
| b_q0 | in | 40 | ap_memory | b | array |
| prod_address0 | out | 5 | ap_memory | prod | array |
| prod_ce0 | out | 1 | ap_memory | prod | array |
| prod_we0 | out | 1 | ap_memory | prod | array |
| prod_d0 | out | 16 | ap_memory | prod | array |

Vivado HLS provides a tool for comparing synthesis reports when different solutions of the same C++ code have been implemented.
Figure below shows the comparison of synthesis report for solution1 and solution4 where is possible to see that pipelining the top level function reduce the number of clock cycles from 687 to only 36.



Vivado HLS Report Comparison

**All Compared Solutions**

solution1: xc7z020clg484-1

solution4: xc7z020clg484-1

**Performance Estimates**

⊟ **Timing (ns)**

| Clock | | solution1 | solution4 |
|---|---|---|---|
| default | Target | 5.00 | 5.00 |
| | Estimated | 3.44 | 3.89 |

⊟ **Latency (clock cycles)**

| | | solution1 | solution4 |
|---|---|---|---|
| Latency | min | 686 | 35 |
| | max | 686 | 35 |
| Interval | min | 687 | 36 |
| | max | 687 | 36 |

**Utilization Estimates**

| | solution1 | solution4 |
|---|---|---|
| BRAM_18K | 0 | 0 |
| DSP48E | 1 | 5 |
| FF | 63 | 260 |
| LUT | 73 | 86 |

Export the report(.html) using the Export Wizard

## 9. References

- [1] L. H. Crockett, R. A. Elliot, M. A. Enderwitz and R. W. Stewart, The Zynq Book: **Embedded Processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC**, First Edition, Strathclyde Academic Media, 2014.

- [2] http://en.wikipedia.org/wiki/Main_Page

- [3] Vivado Design Suite User Guide, UG902 (v2014.1) May 30, 2014

- [4] High-Level Synthesis for FPGAs: From Prototyping to Deployment, Jason Cong, Fellow, IEEE, Bin Liu, Stephen Neuendorffer, Member, IEEE, Juanjo Noguera, Kees Vissers, Member, IEEE and Zhiru Zhang, Member, IEEE