The Abdus Salam International Centre for Theoretical Physics

HLS optimization and integration

Advanced Workshop on FPGA-based Systems-On-Chip for Scientific Instrumentation and Reconfigurable Computing

Trieste, 26st November-7 December 2018

Fernando Rincón
Fernando.rincon@uclm.es

Contents

Objectives

After the completion of the lab, you will be able to:

- build a hardware core from a C high-level description
- optimize the C code adding directives in your design
- specify the type of interface for the inputs/outputs of the core
- export the the core to be used in a Vivado project
- Write a simple application using the driver generated after the synthesis

Create HLS project

Project Files

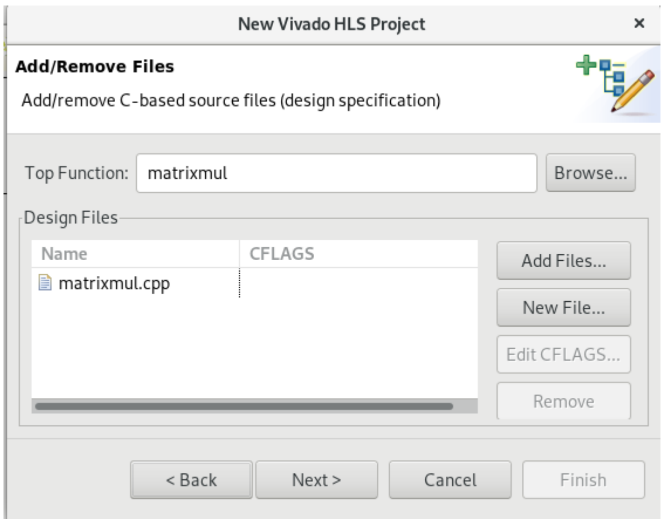
The code for the new core is composed of the five following files:

- matrix_mult.h/.c: The naive matrix multiplication algorithm.
- matrix_mult_test.c: Test bench for the matrix multiplication

Create the project

The first steps will be the creation of the Vivado_HLS project and setting up of the source and test-bench files, the clock cycle and the board type

1. Launch Vivado HLS and select the **Create New Project** flow:
    1. Select **matrixmul** as the **Project Name**.
    2. Add **matrixmul.cpp** as the source file-system, and **matrixmul** as the **top-level** function. Note that the top-level function should be the function of the source code that is the entry point of the design. In this case there’s only one function. Click Next.


New Vivado HLS Project

Add/Remove Files

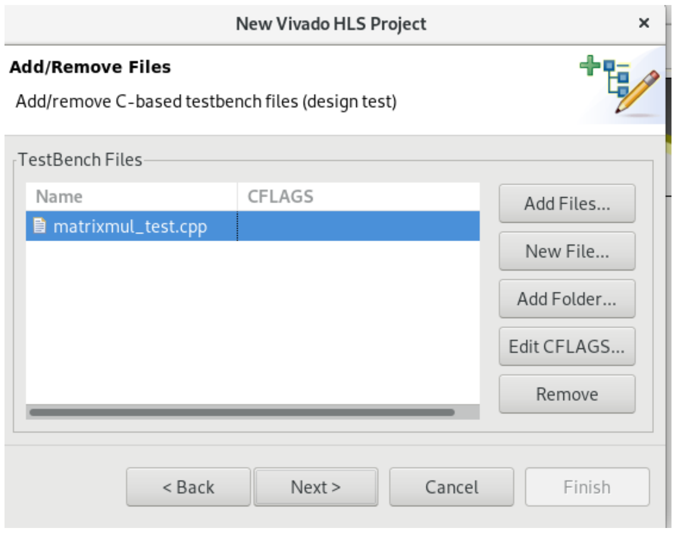
Add/remove C-based source files (design specification)

Top Function: matrixmul

Design Files

Name	CFLAGS
matrixmul.cpp	

3. Add **matrixmul_test.cpp** as the test-bench source file and ckick **Next**.


New Vivado HLS Project

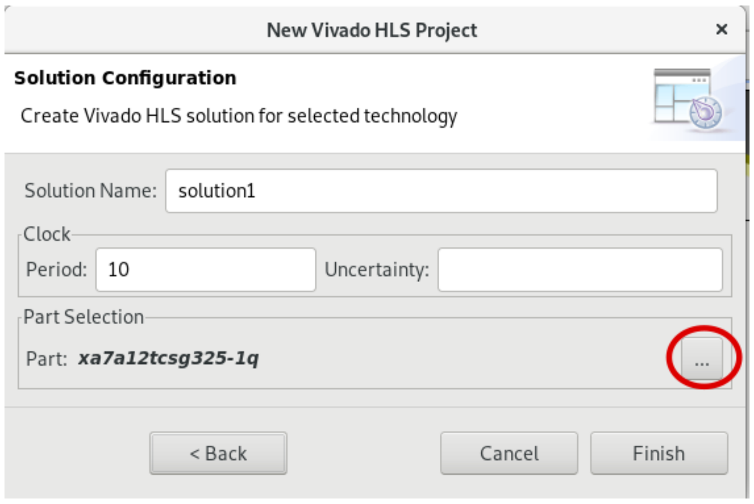
Add/Remove Files

Add/remove C-based testbench files (design test)

TestBench Files

Name	CFLAGS
matrixmul_test.cpp	

4. In the Solution Configuration page, leave the default values set and click on the **Part's Selection** Browse button (shown in red in the picture) to select the *ZedBoard* as the target device.



Create Vivado HLS solution for selected technology

Solution Name: solution1

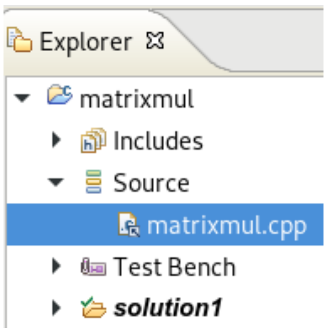
Clock

Period: 10 Uncertainty:

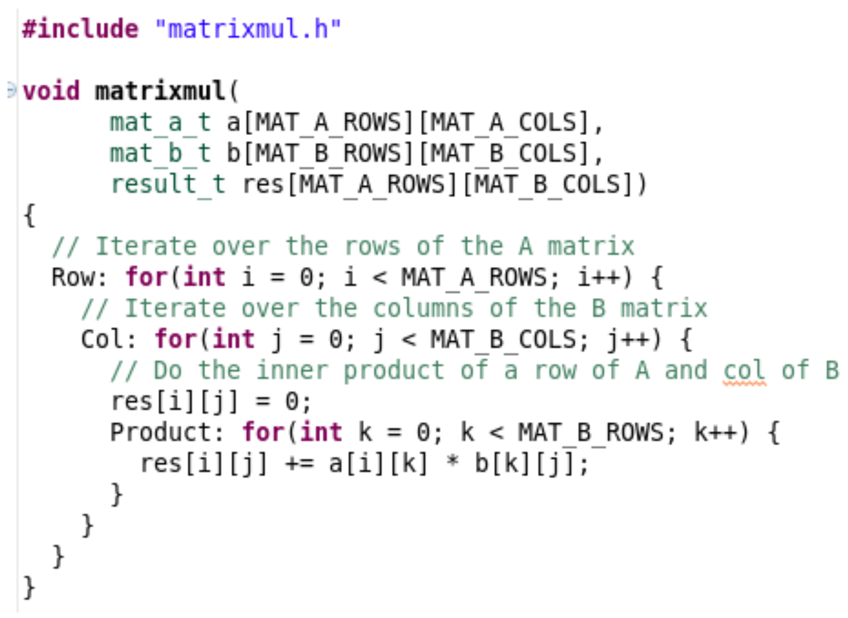
Part Selection

Part: *xa7a12tcsg325-1q*

5. Note the 10 value in the clock period box. The default target clock will be of a 100MHz for the cores synthesized. Leave the value by default. I can later be changed in the project.
6. Then click on **Finish**.
7. Once the project generated, you will find on the left side of the window the project *Explorer*, that provides a hierarchical view of the different files contained.



8. Look for the Source subfolder, and double-click on the *matrixmul.cpp* to open its contents in the editor.



```
#include "matrixmul.h"

void matrixmul(
      mat_a_t a[MAT_A_ROWS][MAT_A_COLS],
      mat_b_t b[MAT_B_ROWS][MAT_B_COLS],
      result_t res[MAT_A_ROWS][MAT_B_COLS])
{
  // Iterate over the rows of the A matrix
  Row: for(int i = 0; i < MAT_A_ROWS; i++) {
    // Iterate over the columns of the B matrix
    Col: for(int j = 0; j < MAT_B_COLS; j++) {
      // Do the inner product of a row of A and col of B
      res[i][j] = 0;
      Product: for(int k = 0; k < MAT_B_ROWS; k++) {
        res[i][j] += a[i][k] * b[k][j];
      }
    }
  }
}
```

You can observe how the product is performed through three nested loops. The first two iterate over all rows in A and columns in B respectively, while the third one performs the partial products addition for the selected cell of the output matrix.

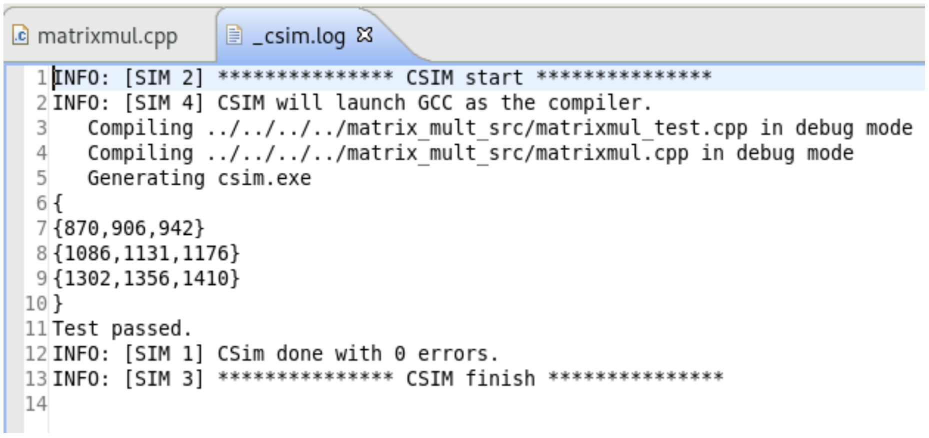
Code simulation

Before proceeding to the synthesis of the design, it is advisable to check that the C code is correct from the functional point of view. This step in call “C simulation” in Vivado HLS, but in fact simply consist in the C code compilation of the matrixmult algorithm plus its test-bench in the host machine, and the execution of the resulting program.

1. Run **Project→ C Simulation** to for the verification of the project.

   This will pop-up a dialog box that you can leave with the default values.

2. As a result of the execution the console window should open, the cross-compilation should take place, and a message telling that there where no error should be shown at the end of the process.



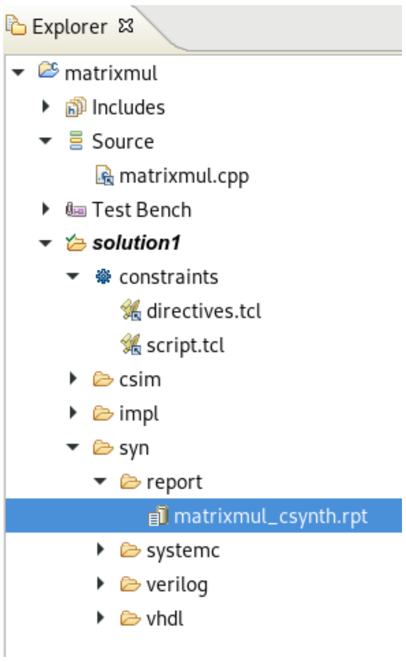
```
INFO: [SIM 2] *************** CSIM start ***************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
   Compiling ../../../../matrix_mult_src/matrixmul_test.cpp in debug mode
   Compiling ../../../../matrix_mult_src/matrixmul.cpp in debug mode
   Generating csim.exe
{
{870,906,942}
{1086,1131,1176}
{1302,1356,1410}
}
Test passed.
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```

Synthesize the code and analyze the results

The next step in the design flow is the synthesis of the code into an RTL description:

1. Perform an initial synthesis run, **Solution→Run C Synthesis→Active solution**.

2. Once the process has been correctly completed a report tab will appear with a summary of the results. You can also have access to the report from the Explorer.



3. Observe the results shown in the report. Notice the estimated time and the latency interval values, as well as the % of usage of resources.

4. Now click on the **Detail→Loop** to view the latency estimations for the two loops in the table shown.

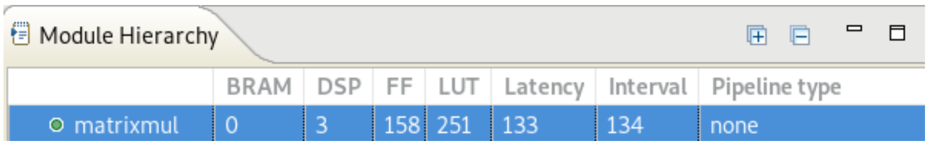
Questions

1. Which is the estimated **clock period**?
2. Which is the overall **latency**?
3. Now have a look at the Loop data. Which do you think it is the relationship between latency, iteration latency and trip count?
4. Which is the total number of FFs and LUTs used?

The analysis perspective

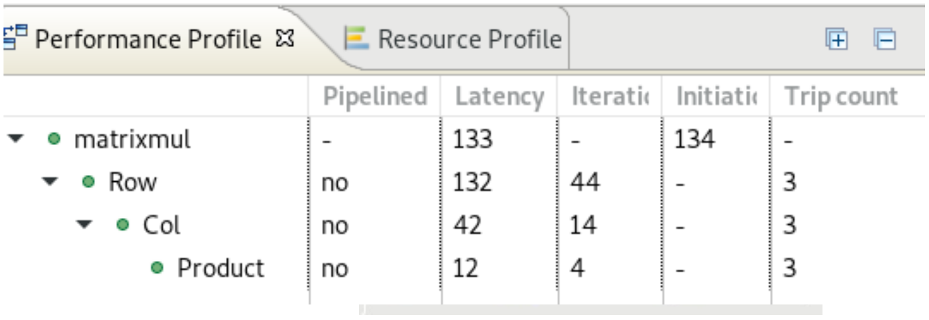
The results shown in the report are just a summary of the performance and resource usage of the generated RTL solution. You can have access to a deeper analysis through the Analysis perspective:

1. Switch to the Analysis perspective by selecting **Solution→Open Analysis perspective**. Alternatively you can click on the corresponding button on the right upper part of the window
2. You sould see three parts in the screen. On the left side, at the upper part you can find the *Module Hierarchy* tab. Since the multiplier is composed of a single function, only one is displayed with a brief summary of the results:



	BRAM	DSP	FF	LUT	Latency	Interval	Pipeline type
matrixmul	0	3	158	251	133	134	none

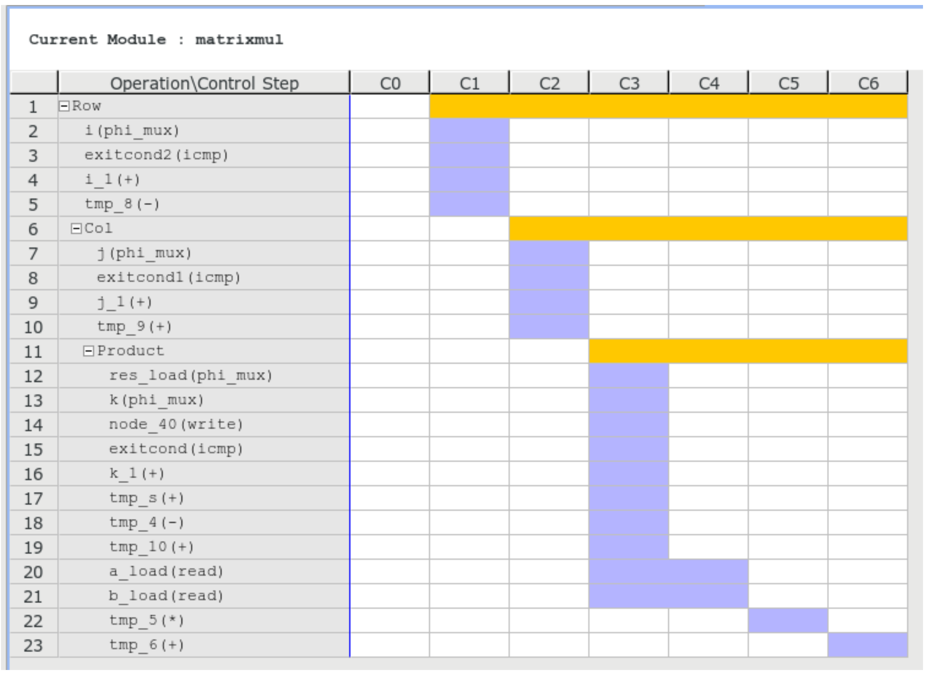
3. On the lower part you can find the *Performace* and *Resource Profiles*. Expand the hierarchy in the *Performace Profile* and you'll discover that there is an entry per each foor loop with the label name set on the source code: *Row*, *Col* and *Product.* Look how these are the same results that you obtained in the Loop detail of the report



	Pipelined	Latency	Iteratic	Initiatic	Trip count
matrixmul	-	133	-	134	-
Row	no	132	44	-	3
Col	no	42	14	-	3
Product	no	12	4	-	3

4. Right-click on the matrixmult text in the *Performance Profile* tab on the lower left pane, and select the **Performance/Resource** view. That should open a different table graph in the main pane, as the one described in the next step.

5. On the right part of the screen you can see de detailed scheduling of the operations and control steps they are assigned to. Click on all **+** parts in the hierarchy to get the full vision in the following picture:



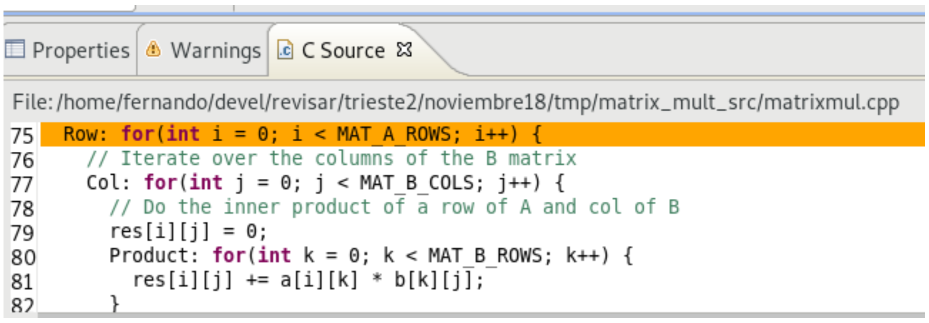
Current Module : matrixmul

	Operation\Control Step	C0	C1	C2	C3	C4	C5	C6
1	⊟Row		🟨	🟨	🟨	🟨	🟨	🟨
2	i(phi_mux)		🟦					
3	exitcond2(icmp)		🟦					
4	i_1(+)		🟦					
5	tmp_8(-)		🟦					
6	⊟Col			🟨	🟨	🟨	🟨	🟨
7	j(phi_mux)			🟦				
8	exitcond1(icmp)			🟦				
9	j_1(+)			🟦				
10	tmp_9(+)			🟦				
11	⊟Product				🟨	🟨	🟨	🟨
12	res_load(phi_mux)				🟦			
13	k(phi_mux)				🟦			
14	node_40(write)				🟦			
15	exitcond(icmp)				🟦			
16	k_1(+)				🟦			
17	tmp_s(+)				🟦			
18	tmp_4(-)				🟦			
19	tmp_10(+)				🟦			
20	a_load(read)				🟦	🟦		
21	b_load(read)				🟦	🟦		
22	tmp_5(*)						🟦	
23	tmp_6(+)							🟦

6. Notice that there are 7 control step in the control machine, and that the yellow lines identify those related to every label in the source code. Notice how the *Product* requires 4 control steps, which is also the iteration latency previously identify (see on the left).

7. The operations that you see on the left correspond to the primitives used for the implementation of the different source code sentences. Note that in many cases there is no one to one matching since high-level sentences may imply the use of more than one primitive.

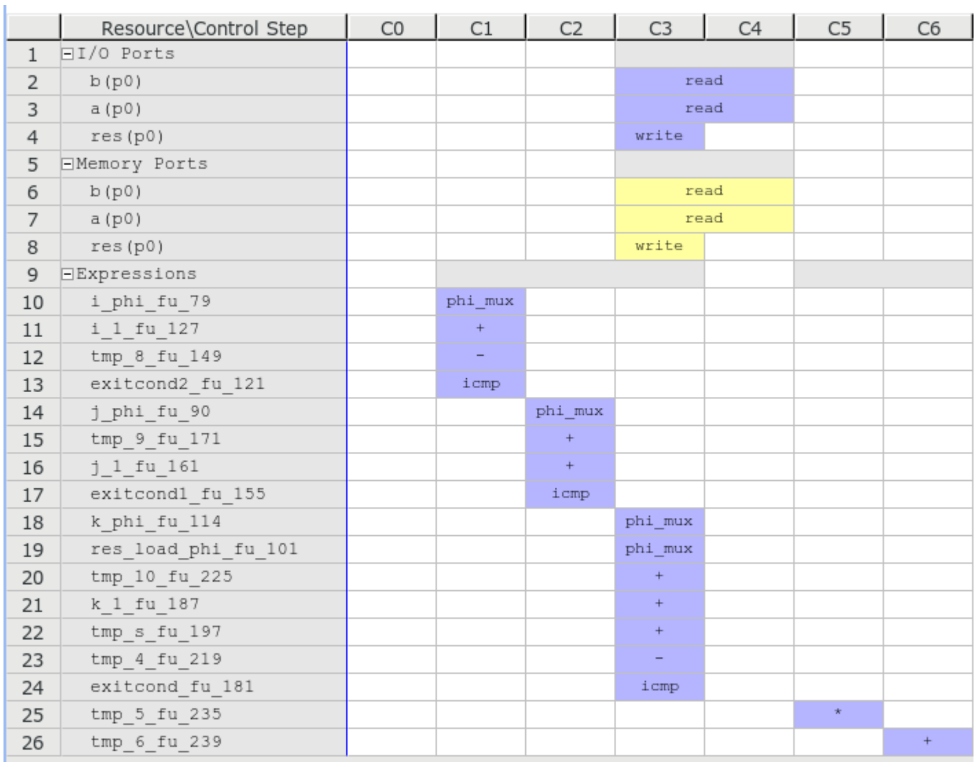
8. You can locate the exact source code sentence each primitive has been derived from. To do so, select a cell in the table, for example the one at line 4 and column C1 (*i_1 (+)*). This seems to be an increment of a variable. Then **right-click→Goto Source**. You can see how the increment is related to the i++ operation in the Row loop, at line 75.



File: /home/fernando/devel/revisar/trieste2/noviembre18/tmp/matrix_mult_src/matrixmul.cpp

```
Row: for(int i = 0; i < MAT_A_ROWS; i++) {
  // Iterate over the columns of the B matrix
  Col: for(int j = 0; j < MAT_B_COLS; j++) {
    // Do the inner product of a row of A and col of B
    res[i][j] = 0;
    Product: for(int k = 0; k < MAT_B_ROWS; k++) {
      res[i][j] += a[i][k] * b[k][j];
    }
```

9. Now click on the *Resource* tab of the central window to get a table similar to this:



	Resource\Control Step	C0	C1	C2	C3	C4	C5	C6
1	⊟I/O Ports							
2	b(p0)				read			
3	a(p0)				read			
4	res(p0)				write			
5	⊟Memory Ports							
6	b(p0)				read			
7	a(p0)				read			
8	res(p0)				write			
9	⊟Expressions							
10	i_phi_fu_79		phi_mux					
11	i_1_fu_127		+					
12	tmp_8_fu_149		-					
13	exitcond2_fu_121		icmp					
14	j_phi_fu_90			phi_mux				
15	tmp_9_fu_171			+				
16	j_1_fu_161			+				
17	exitcond1_fu_155			icmp				
18	k_phi_fu_114				phi_mux			
19	res_load_phi_fu_101				phi_mux			
20	tmp_10_fu_225				+			
21	k_1_fu_187				+			
22	tmp_s_fu_197				+			
23	tmp_4_fu_219				-			
24	exitcond_fu_181				icmp			
25	tmp_5_fu_235						*	
26	tmp_6_fu_239							+

10. Here you can see the different operations (and therefore the resources used) at each control step and for every primitive in the code.

Questions

1. At which control step is the exit condition of the row loop evaluated?
2. Having a look at the resource/control step detail, which do you think it is the bottleneck of this design?

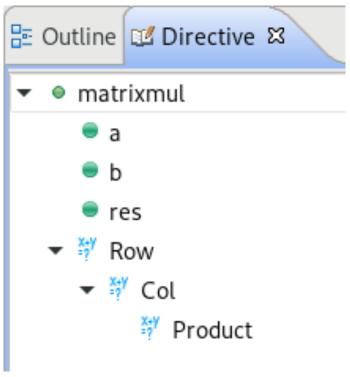
Optimizing the code

There are a number of techniques that can be applied to optimize the code, both in resources and performance. In the following steps we'll make use of two simple but effective ones.

The PIPELINE directive

Vivado HLS supports the implementation of different solutions for the same source code, so multiple different optimizations can be tested without overwriting the previous one.

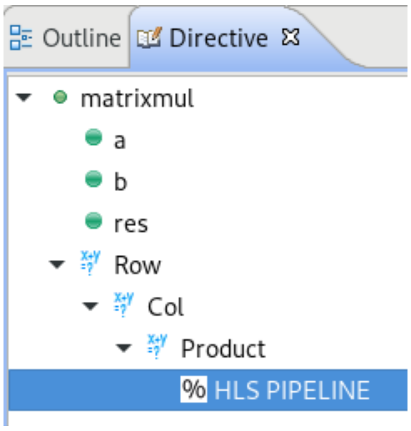
1. Go back to the **Synthesis** view using the button in the upper right part of the window.
2. Select **Project -> New Solution** and check that the **Copy directives and constraints box** is *checked* to copy the current solution into a new one (even if we haven't included any yet). Note how after the creation, the new solution is highlighted in bold This means that now it is the solution chosen by default.
3. Open the source code for the *matrixmul.cpp* file.
4. Look at the right pane of the window, where there are two tabs (*Outline* and *Directive*). Select the **Directive** one. You should see a list of the parameters, variables and loops identified in the source code.



5. Select the **Product** label *Directive* pane, **right-click** on it and select **Insert Directive…**
6. Edit the type of directive to associate to the function in the resulting dialog box, filling up the following fields, and letting the rest of the values as set by default:
   - *Directive*: **PIPELINE**
   - *Destination*: **Directive File**
7. The directive will be shown below the *Product* label in the hierarchy. This will cause the pipelining of the product computation in the next synthesis

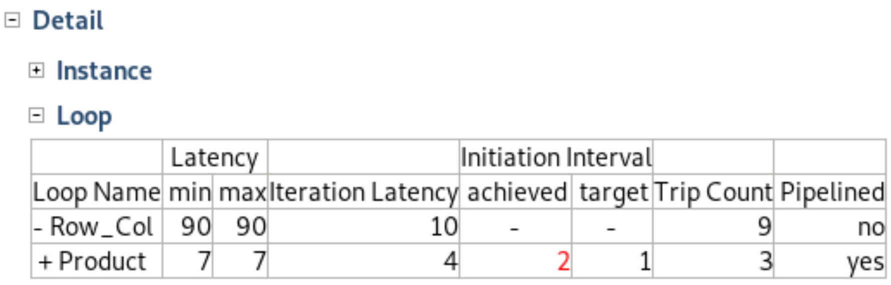
run.

8. At this point, the Directive tab should look like as follows.



9. **Synthesize** the design again

10. Once the synthesis has finished you should see a critical error about the tool not being able to achieve an initiation interval of 1. You can also see the effect in the detailed loop view of the report:



⊟ Detail

⊞ Instance

⊟ Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Row_Col	90	90	10	-	-	9	no
+ Product	7	7	4	2	1	3	yes

The main reason is that as the table shows, it is not possible to start a new product operation at each clock cycle, but at two (that is the best solution achieved).

11. To fix the problem go back to the *Directives* tab, double-click on the **HLS PIPELINE** directive and in the resulting window select 2 for the fileld *II*, in the *Options* part. This will tell the tool that the pipeline will start a new iteration every two clock cycles.

12. Synthesize the design again, and check that the problem has been fixed.

13. When the synthesis is completed, select **Project > Compare Reports…**, and in the resulting dialog box select *Solution1* and *Solution2* from the *Available Reports*, and click on the **Add>>** button.

Questions

14. Which have been the improvements in the second solution with respect to the first one?
15. What happened to the loops shown in the details view of the solution 2 synthesis report?
16. Open the Analysis view for Solution 2. What are the main differences in the detail scheduling with respect to the previous version?

Appropriate data types

One of the problems with the use of standard data types is that they may imply the use of more bits than those strictly necessary. We'll assume that the values in our source matrices are 8-bits wide. In that case we don't need full *int* types, but just 8 bit unsigned ones. This done through the use of a special include library: **ap_int.h**. In the following step we'll modify the source code accordingly.

1. Create a **Project→New Solution**, making a copy of the constraints and directives from Solution 2.
2. Go back to the *matrixmul.cpp* code window, and locate the **#include <matrixmul.h>** line. Put the cursor over any part of the matrixmul text and hit **F3**. This should open the header file.
3. Now replace the 3 definition types for mat_a_t, mat_b_t and mat_result_t to **ap_uint<8>**, **ap_uint<8>** and **ap_uint<16>** respectively.

```
typedef ap_uint<8> mat_a_t;
typedef ap_uint<8> mat_b_t;
typedef ap_uint<16> result_t;
```

4. Perform a new **synthesis**.
5. Compare the results of the last two solutions.

Questions

1. Which is the main difference between this and the previous solution?
2. How many cycles are required for the computation of each product? And which is the initiation interval?

Final Verification

Once the design has been completed, you could perform one final verification after the synthesis process, where the resulting RTL code is simulated against the original test bench. The output result is expected to be exactly the same to the C version:

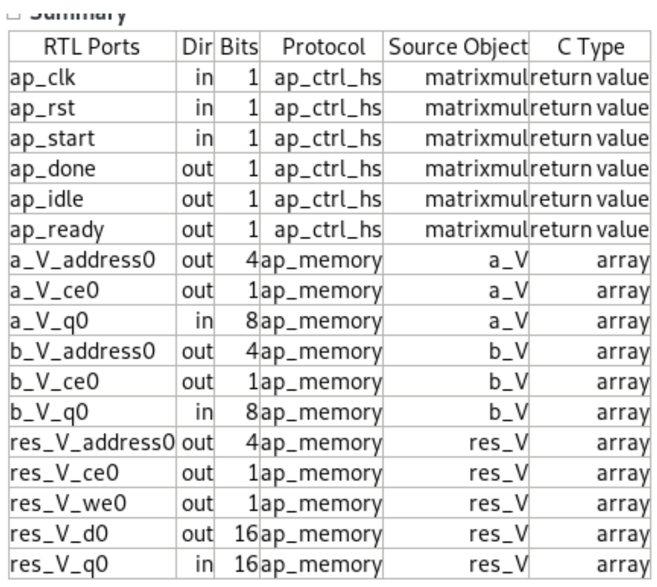
1. Select **Solution > Run C/RTL Cosimulation**

2. In the resulting dialog box make sure that VHDL is selected and the rest of the values left by default. Then click **OK**.
3. The C/RTL Co-simulation will run, generating and compiling several files, and then simulating the design. In the console window you can see the progress and also a message that the test is passed.
4. Once the simulation has finished, a report tab will open and show the results.

Core packaging

Once the design is ready and has already been tested, there are a few steps still required to generate an IP core suitable to be inserted in our Vivado design. The first issue to be considered is the desired interface. We'll analyze the default one that has been generated and modify it to turn the core into an AXI_Lite core. Next we'll export the design, so it can then be used in a block design.

1. Open the last synthesis report and look at the *Summary* by the end of the report. There you will see the signal interface generated from the C description. You will identify four parts in it:
    1. a set of signals with the *ap_* prefixes. This interface does not correspond to any input or output parameter in the matrixmul function, but is automatically generated to control the execution of the core. The *start* signal, for example, implies the initiation of the computation, while he *done* is set to high when the whole computation has finished.
    2. three more sets with prefixes *a*, *b*, and *res*, respectively. Since their data types are vectors, they have been mapped by default to block memories, so you can identify the address, chip select and data signals in each case.



RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	matrixmul	return value
ap_rst	in	1	ap_ctrl_hs	matrixmul	return value
ap_start	in	1	ap_ctrl_hs	matrixmul	return value
ap_done	out	1	ap_ctrl_hs	matrixmul	return value
ap_idle	out	1	ap_ctrl_hs	matrixmul	return value
ap_ready	out	1	ap_ctrl_hs	matrixmul	return value
a_V_address0	out	4	ap_memory	a_V	array
a_V_ce0	out	1	ap_memory	a_V	array
a_V_q0	in	8	ap_memory	a_V	array
b_V_address0	out	4	ap_memory	b_V	array
b_V_ce0	out	1	ap_memory	b_V	array
b_V_q0	in	8	ap_memory	b_V	array
res_V_address0	out	4	ap_memory	res_V	array
res_V_ce0	out	1	ap_memory	res_V	array
res_V_we0	out	1	ap_memory	res_V	array
res_V_d0	out	16	ap_memory	res_V	array
res_V_q0	in	16	ap_memory	res_V	array

2. We want to be able to both control the core and move data around to/from the PS using an AXI_Lite interface, so the default interface is not the appropriate one. Luckily the interface can easily be changed by just setting a few more directives.

3. Open the **matrixmul.cpp** source code window, and select the **Directives** tab on the right pane.

4. Double-click over the **a** interface. In the resulting dialog fill up the following fields, letting the rest of the values as set by default:

    1. *Directive*: **INTERFACE**

    2. *Destination*: **Directive File**

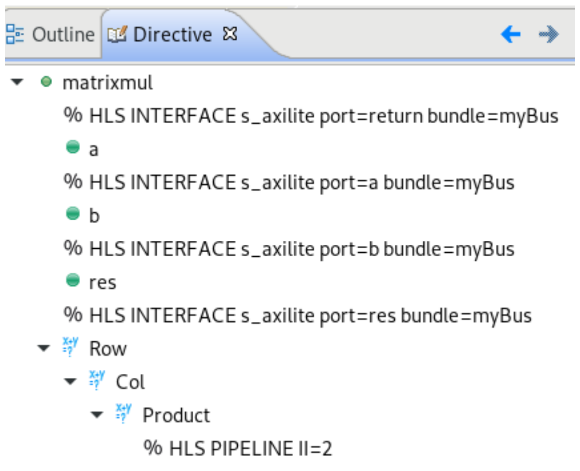
    3. *Mode*: **s_axilite**

    4. *Bundle*: **myBus**

    This will select an AXI_Lite interface for parameter **a**. The *bundle* will later be used to map all parameters to the same AXI bus, instead of having separated buses for each one.

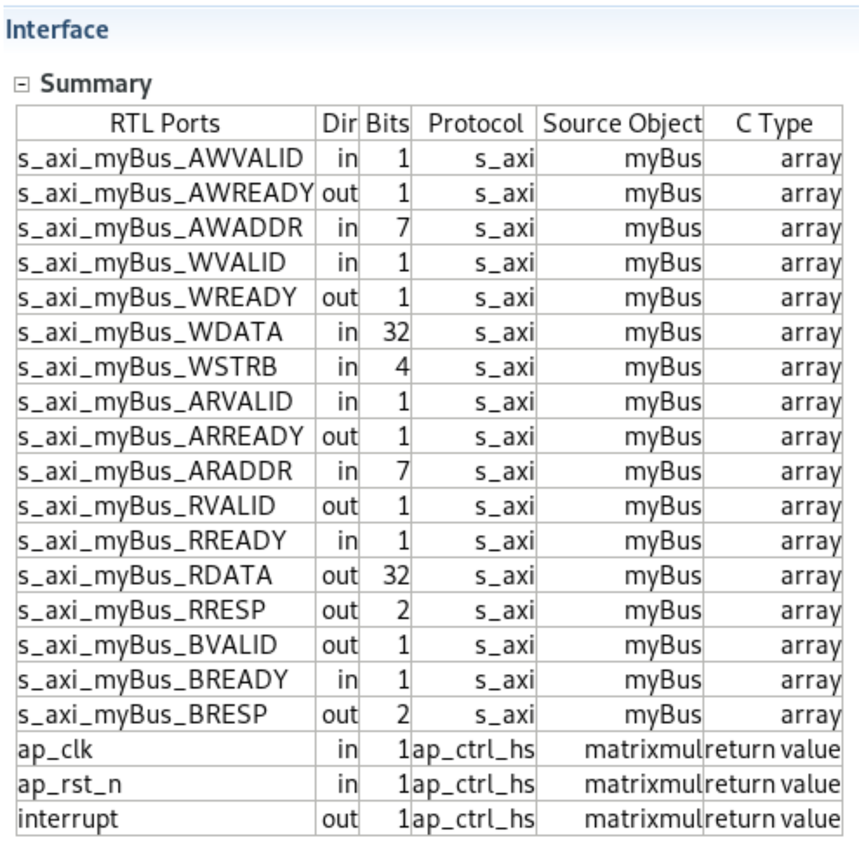
5. Repeat the procedure for parameters **b** and **res**.

6. And finally do exactly the same with the top level function *matrixmul*. While this is not a parameter, the effect of the directive will be that the control signals of the core (start, done, idle, …) will also be mapped to the same AXI_Lite bus. Thus all I/O and control will only be using one bus.



```
matrixmul
    % HLS INTERFACE s_axilite port=return bundle=myBus
    a
    % HLS INTERFACE s_axilite port=a bundle=myBus
    b
    % HLS INTERFACE s_axilite port=b bundle=myBus
    res
    % HLS INTERFACE s_axilite port=res bundle=myBus
  Row
    Col
      Product
        % HLS PIPELINE II=2
```

7. Resynthesize the design and check that the new set of signals correspond to the AXI_Lite bus.



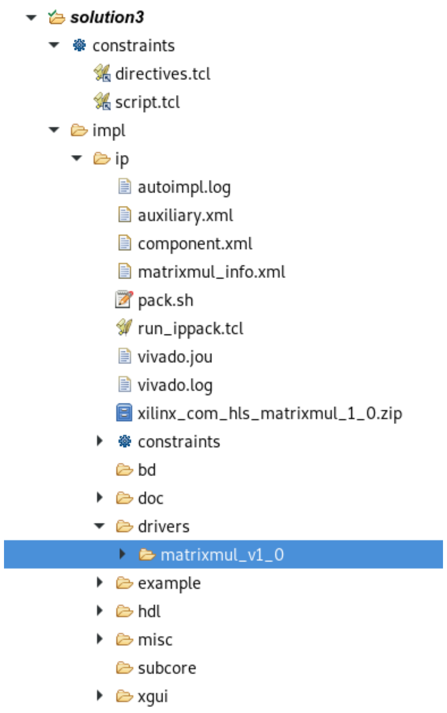
RTL Ports	Dir	Bits	Protocol	Source Object	C Type
s_axi_myBus_AWVALID	in	1	s_axi	myBus	array
s_axi_myBus_AWREADY	out	1	s_axi	myBus	array
s_axi_myBus_AWADDR	in	7	s_axi	myBus	array
s_axi_myBus_WVALID	in	1	s_axi	myBus	array
s_axi_myBus_WREADY	out	1	s_axi	myBus	array
s_axi_myBus_WDATA	in	32	s_axi	myBus	array
s_axi_myBus_WSTRB	in	4	s_axi	myBus	array
s_axi_myBus_ARVALID	in	1	s_axi	myBus	array
s_axi_myBus_ARREADY	out	1	s_axi	myBus	array
s_axi_myBus_ARADDR	in	7	s_axi	myBus	array
s_axi_myBus_RVALID	out	1	s_axi	myBus	array
s_axi_myBus_RREADY	in	1	s_axi	myBus	array
s_axi_myBus_RDATA	out	32	s_axi	myBus	array
s_axi_myBus_RRESP	out	2	s_axi	myBus	array
s_axi_myBus_BVALID	out	1	s_axi	myBus	array
s_axi_myBus_BREADY	in	1	s_axi	myBus	array
s_axi_myBus_BRESP	out	2	s_axi	myBus	array
ap_clk	in	1	ap_ctrl_hs	matrixmul	return value
ap_rst_n	in	1	ap_ctrl_hs	matrixmul	return value
interrupt	out	1	ap_ctrl_hs	matrixmul	return value

8. Now the final step will be the generation of the IP core. To do so click on **Solution→Export RTL**. You can leave the default configuration in the resulting pop-up dialog and simply select **OK**.

9. After the generation a new folder *impl* will appear in the project hierarchy.

Inside it the **ip** subfolder will contain all the information regarding the core. This is the route to be used later in the Vivado tool to add the new created core into the Vivado IP cores catalog.

10. Note how the **ip** subfolder contains another subfolder called **drivers**. Inside an appropriate driver for the core has been generated. This driver is customized to the exact parameters, data types and bus protocols finally chosen.

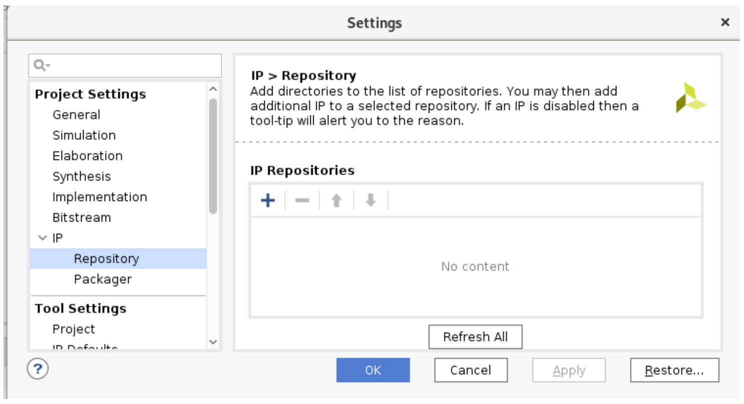


```
solution3
  constraints
    directives.tcl
    script.tcl
  impl
    ip
      autoimpl.log
      auxiliary.xml
      component.xml
      matrixmul_info.xml
      pack.sh
      run_ippack.tcl
      vivado.jou
      vivado.log
      xilinx_com_hls_matrixmul_1_0.zip
      constraints
      bd
      doc
      drivers
        matrixmul_v1_0
      example
      hdl
      misc
      subcore
      xgui
```

Vivado project integration

Once the IP has been generated, we will create a simple Vivado project where the core will be inserted. This core will be connected to the PS through an AXI_Lite interface. Later the project will be exported to the SDK and a simple application that writes data and retrieves the result will also be written.

1. Launch Vivado, and create a new empty project for the Zedboard
2. Create a new block design
3. Click on **Tools->Settings** in the main menu and in the dialog that will pop expand the IP category and click on the **Repository** option.



Settings

Project Settings
General
Simulation
Elaboration
Synthesis
Implementation
Bitstream
IP
Repository
Packager
Tool Settings
Project
IP Defaults

IP > Repository
Add directories to the list of repositories. You may then add additional IP to a selected repository. If an IP is disabled then a tool-tip will alert you to the reason.

IP Repositories

No content

Refresh All

OK Cancel Apply Restore...

4. Click on the **+** icon to add the path to a new repository. This path should be the location of the **impl/ip** folder of the Vivado HLS project of the maltmul. The relative path for this lab should be the following one:

   <matmul project>/solution3/impl/ip

5. A confirmation message should show that 1 IP core has been detected and added to the project. Click **OK** to exit the message and **OK** again to close the settings.

6. Back in the *Diagram* editor, add the following two cores using the **+** icon :

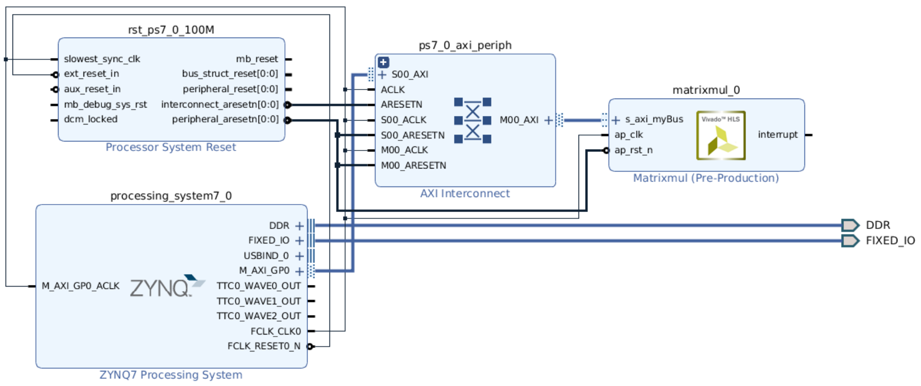
   1. ZYNQ7 Processing System

   2. Matrixmul

7. Now run the two assistants available in the top of the panel:

   1. The *Block Automation assistant* will configure the PS part and generate the external connections for the *DDR* and *Fixed I/O*

   2. The *Connection Automation* will connect the *Matrixmul* core to the AXI bus of the PS.



8. Save the design and run **Tools→Validate Design** to check that everything in the block design is correct.

9. Now switch to the *Sources* tab in the *Block Design* pane, right-click over the design block file (**design_1.bd**) by default, and select **Create HDL Wrapper ...**

10. Then click on the **Generate Bitstream** commnand on the *Flow Navigator* pane and wait for the completion of the process.

11. Once finished Run **File→Export→Export Hardware** and check the **Include bitstream** box.

12. Now that the hardware part has been generated select **File→Launch SDK** to build the minimal software to test the system.

Sofware Application

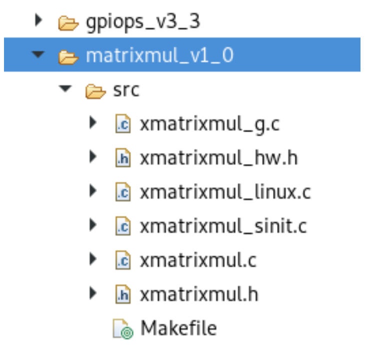
1. The first operation that the SDK will perform after the opening will be the generation of the hardware platform initialization files. Once the process has completed create the hello world application on a standalone configuration: **File→New→Application Project ...**

2. Compile and **Run** the project, and look for the *hello world* message to test that the platform is working correctly.

3. If everything was correct, the next step should be the modification of the program to perform the matrix multiplication computation on the PL, which can easily be done with the driver generated by *Vivado HLS*.

4. The driver should be located in the BSP , in the following path:

```
<bsp name>/ps7_cortexa9_0/libsrc/matrixmul_v1_0/src
```



```
▸ gpiops_v3_3
▾ matrixmul_v1_0
    ▾ src
        ▸ xmatrixmul_g.c
        ▸ xmatrixmul_hw.h
        ▸ xmatrixmul_linux.c
        ▸ xmatrixmul_sinit.c
        ▸ xmatrixmul.c
        ▸ xmatrixmul.h
          Makefile
```

5. Now open the **helloworld.c** source file in the application *src* folder.

6. Add the driver include at the top of the file:

```
#include "xmatrixmul.h"
```

7. After the includes, and before the main function we'll add some

declarations with the test values for **a** and **b** vectors, and for the **result**. Remember that **a** an **b** values are 8 bits wide, while for the **result** we'll need double number of bits, therefore a *short int*, instead a *char*:

```
char a[9] = {11, 12, 13, 14, 15, 16, 17, 18 ,19};

char b[9] = {21, 22, 23, 24, 25, 26, 27, 28, 29};

short int res[9];
```

8. Now locate the "hello world" message, after the initialization of the platform. Here we will append the following code for the configuration of the *matrixmul* peripheral.

```
XMatrixmul pmatrix;

XMatrixmul_Config *pmatrix_config;

pmatrix_config=XMatrixmul_LookupConfig(XPAR_XMATRIXMUL_0_DEVICE_
ID);

XMatrixmul_CfgInitialize(&pmatrix, pmatrix_config);
```

9. The next step will be passing the input values to the core:

```
XMatrixmul_Write_a_V_Bytes(&pmatrix, 0, a, 9);

XMatrixmul_Write_b_V_Bytes(&pmatrix, 0, b, 9);
```

10. Then we need to tell the core to perform the computation, and we should wait until it has been completed:

```
XMatrixmul_Start(&pmatrix);

while (!XMatrixmul_IsDone(&pmatrix));
```

11. Finally, we just need to retrieve the result and print it. Note how for the retrieval we request 18 bytes, that we upload to the **res** vector, that has been casted as a *"char *"*. This is because the *short ints* are stored one after another, as two consecutive bytes. Later, in the printing we use the res result as a short int type.

```
XMatrixmul_Read_res_V_Bytes(&pmatrix, 0, (char*)res, 18);

for (int i = 0; i < 9; i++)

    xil_printf("%d\r\n", res[i]);
```

12. Compile and run the code, and you should get the same result that we tested during the C simulation of the test-bench in Vivado HLS.