ToolsXilinxLabsRTLHLSIP

Creating and using custom IP blocks both in Verilog and using High-Level Synthesis

Back to Xilinx Labs

Objectives

- Learn to create custom IP blocks at RTL level (Verilog, VHDL)
- Use AXI bus to connect an IP block with the Zynq PS
- Learn to use High-level Synthesis (HLS) to create a similar IP block in C/C++
- Test both IP blocks using the SDK

Custom IP block at RTL level

A system on a chip consisting of both a Hard Processor System and FPGA fabric, such as the Zynq-7000, offers the opportunity of offloading computation to the FPGA. Parallelizable algorithms can thus be accelerated, or more computations can be executed in parallel.

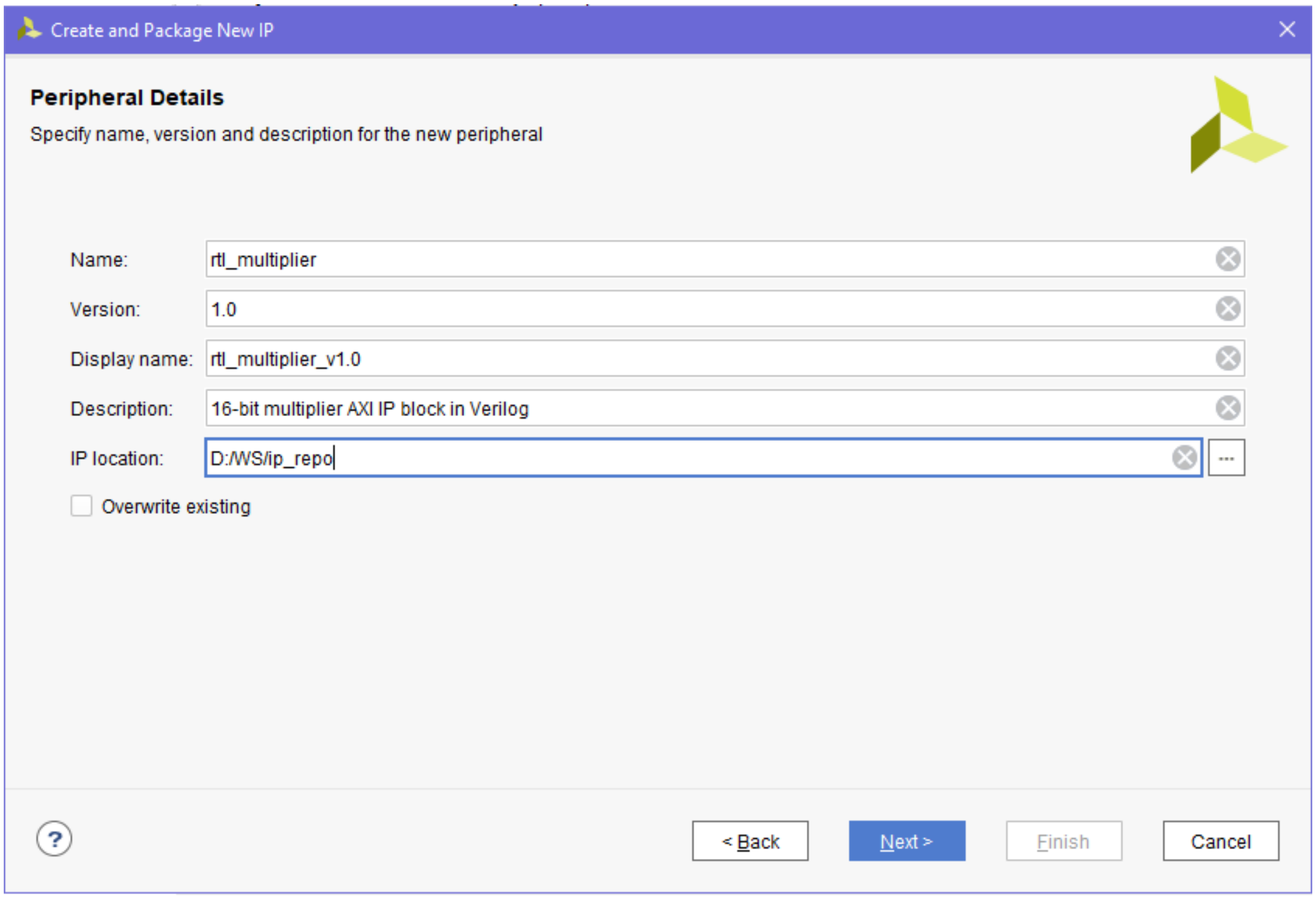
In this section, we will create a simple custom AXI IP block that multiplies two numbers and will connect it to the Zynq PS. The multiplier will take as input two 16-bit unsigned numbers and will output the product as one 32-bit unsigned number. A single 32-bit write to the IP block will contain the two 16-bit inputs, separated into the lower and higher 16 bits. A single 32-bit read from the peripheral will contain the result from the multiplication of the two 16-bit inputs.

This design doesn’t really make much sense as an accelerator, but it is a good learning example.

Create and Package the IP block

Feel free to start with any previously made Vivado project that contains a Zynq system. For example, you can start with the system you created in the previous lab, *Building a basic ZYNQ system on the PYNQ-Z1 board*. To make your work easier, you can copy-paste the project *ZynqComputer* to *ZynqComputerExtended* and open it in Vivado. Now that you have a project with a Zynq PS System open in Vivado, follow the instructions below.

- Start by going to menu *Tools -> Create and Package New IP...*.
  - Read the overview of the *Create and Package New IP* wizard and then click *Next*.
- We are interested in a new AXI4 peripheral, therefore select the *Create a new AXI4 peripheral* and click *Next*.
  - *Note*: If you've been attentive in the previous tutorial and homework assignment, you might still remember what the AXI stands for. It is also reasonable that you forgot, considering spring break, etc. Either way, you are encouraged to consult this reference guide and other sources to learn more about AXI, AMBA, and how it compares to the Avalon Interface.
- Fill in the *Peripheral Details* fields with proper values. For the *IP location*, select a directory in your group folder, in which you will store all your custom IP blocks.



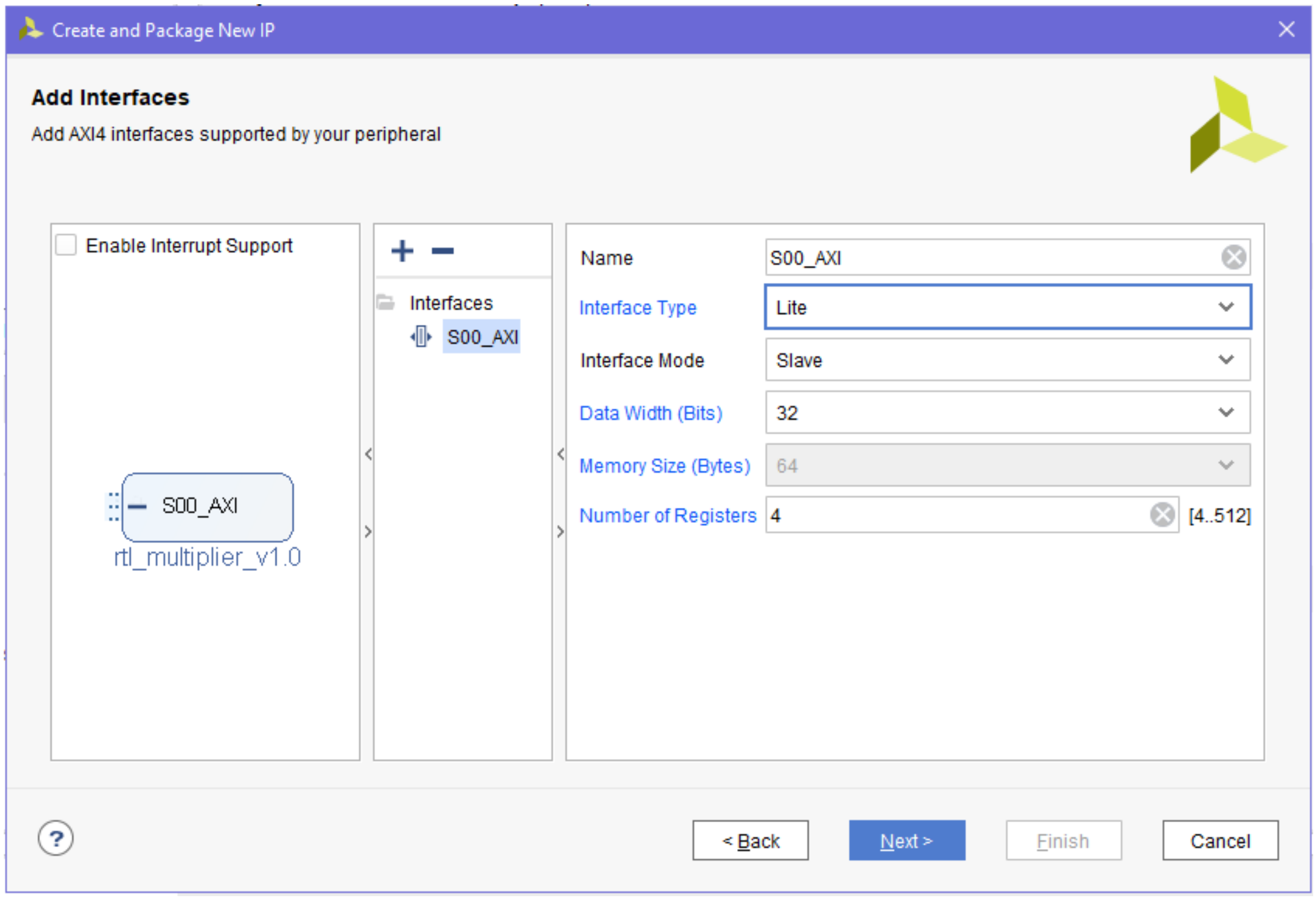
Peripheral Details

Specify name, version and description for the new peripheral

Name:	rtl_multiplier
Version:	1.0
Display name:	rtl_multiplier_v1.0
Description:	16-bit multiplier AXI IP block in Verilog
IP location:	D:/WS/ip_repo

☐ Overwrite existing

- On the *Add Interfaces* page, use the default 32-bit AXI4 Lite Slave interface.



Add Interfaces

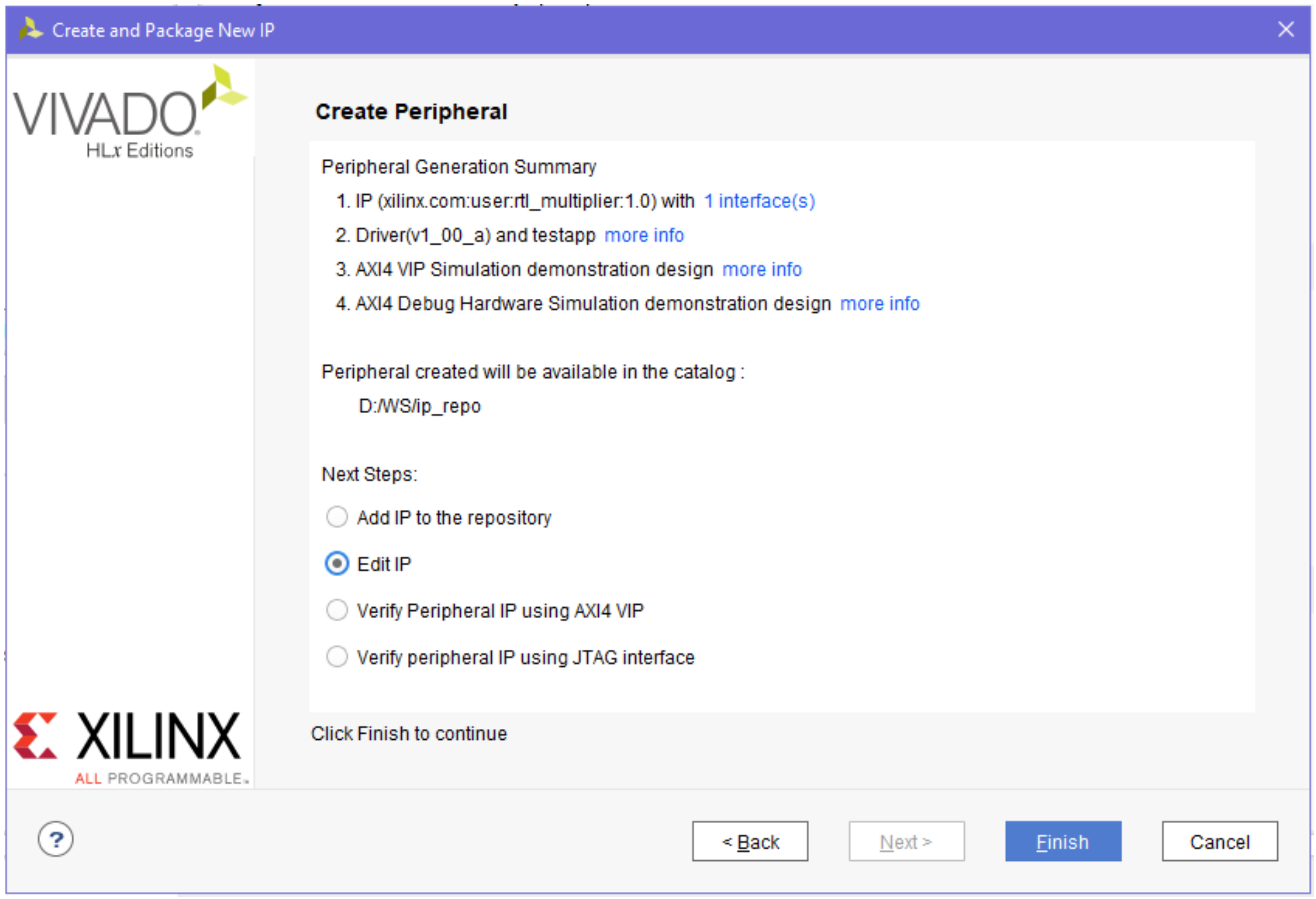
Add AXI4 interfaces supported by your peripheral

Enable Interrupt Support

Interfaces
S00_AXI

Name	S00_AXI
Interface Type	Lite
Interface Mode	Slave
Data Width (Bits)	32
Memory Size (Bytes)	64
Number of Registers	4 [4..512]

- On the last page, select *Edit IP* and click *Finish*. This will open another Vivado window in which we will implement the peripheral.



Create Peripheral

Peripheral Generation Summary
1. IP (xilinx.com:user:rtl_multiplier:1.0) with 1 interface(s)
2. Driver(v1_00_a) and testapp more info
3. AXI4 VIP Simulation demonstration design more info
4. AXI4 Debug Hardware Simulation demonstration design more info

Peripheral created will be available in the catalog :
D:/WS/ip_repo

Next Steps:
- Add IP to the repository
- [✓ Edit IP]
- Verify Peripheral IP using AXI4 VIP
- Verify peripheral IP using JTAG interface

Click Finish to continue

Edit the IP block

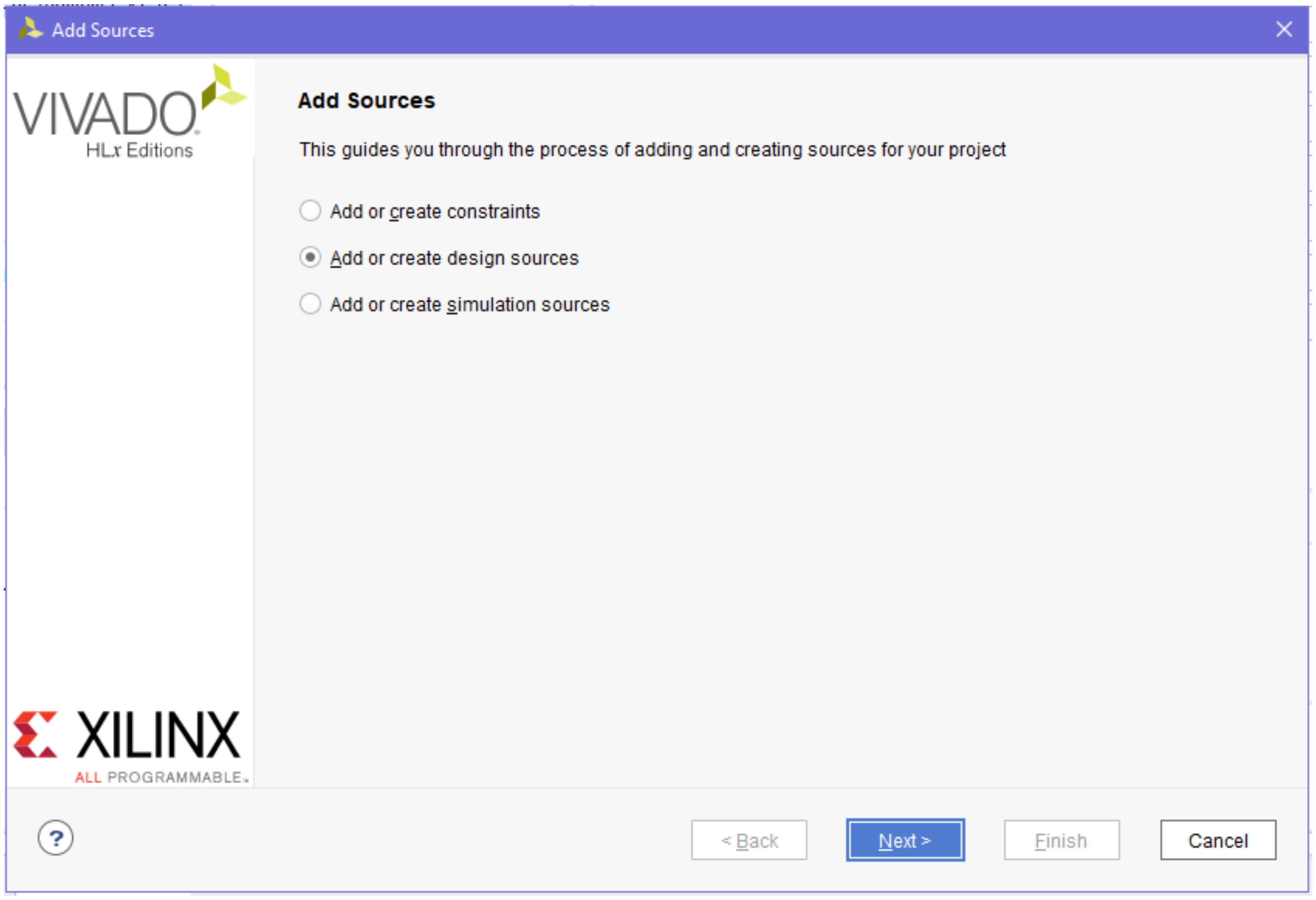
The multiplier Verilog code is simple since it only multiplies two numbers. For example, this code will do:

```
module rtl_multiplier(
    input clk,
    input [15:0] a,
    input [15:0] b,
    output [31:0] product
    );

    reg [31:0] productReg;
    assign product = productReg;
    always @(posedge clk) begin
        productReg <= a * b;
    end
```

```
endmodule
```

Media:rtl_multiplier.v

- Save this file as *rtl_multiplier.v* in a directory such as *ip_repo|src|* for later access.
- From the pannel *Flow Navigator* on the left, click *Add Sources* and, in the new *Add Sources* window, select *Add or create design sources*.

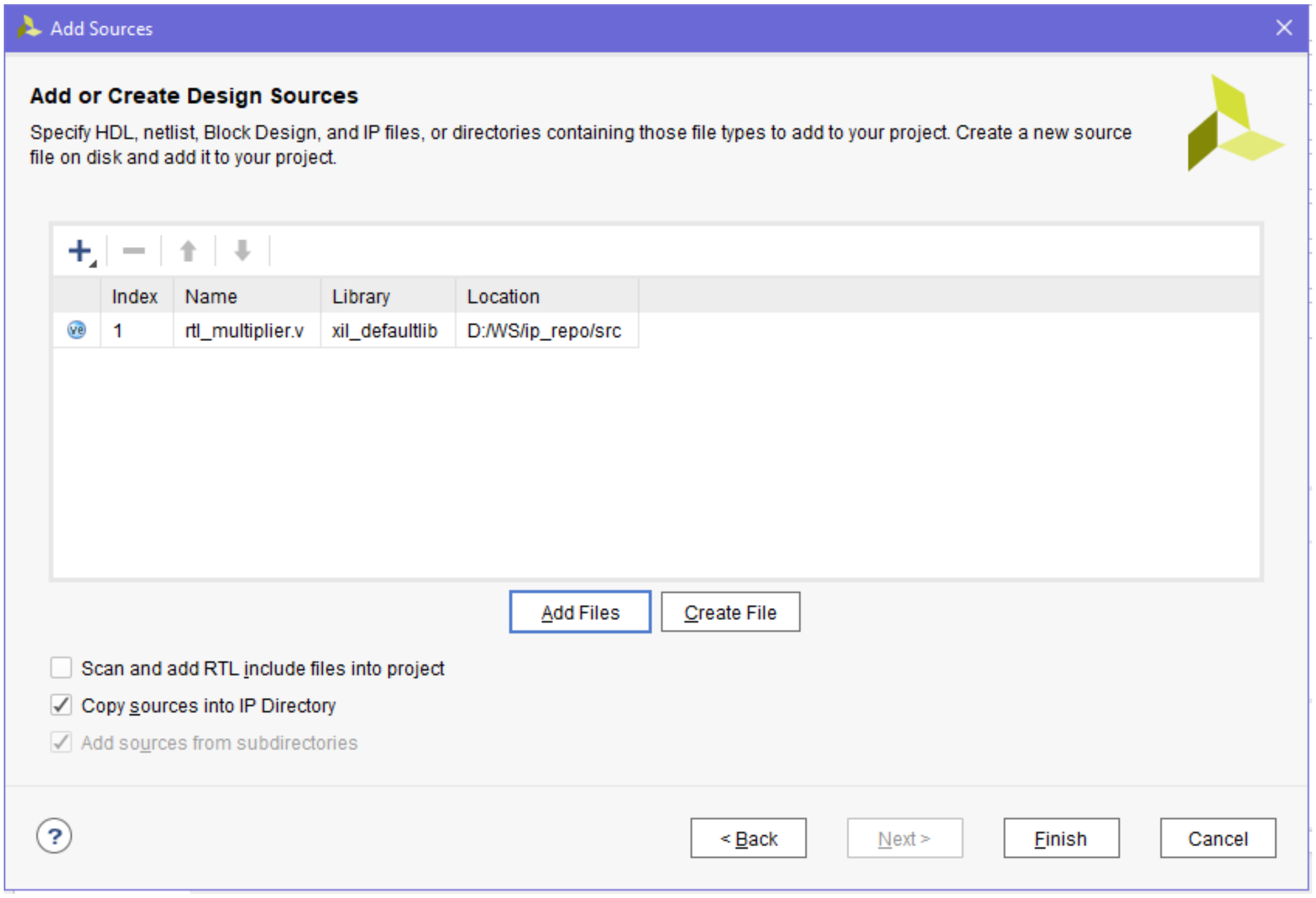


Add Sources

This guides you through the process of adding and creating sources for your project

- ○ Add or create constraints
- ● Add or create design sources
- ○ Add or create simulation sources

- Next, select the previously saved *ip_repo|src|rtl_multiplier.v* file and have the option *Copy sources into IP Directory* checked. Click *Finish* and the file will be added to the *Design Sources*.


Add Sources

Add or Create Design Sources

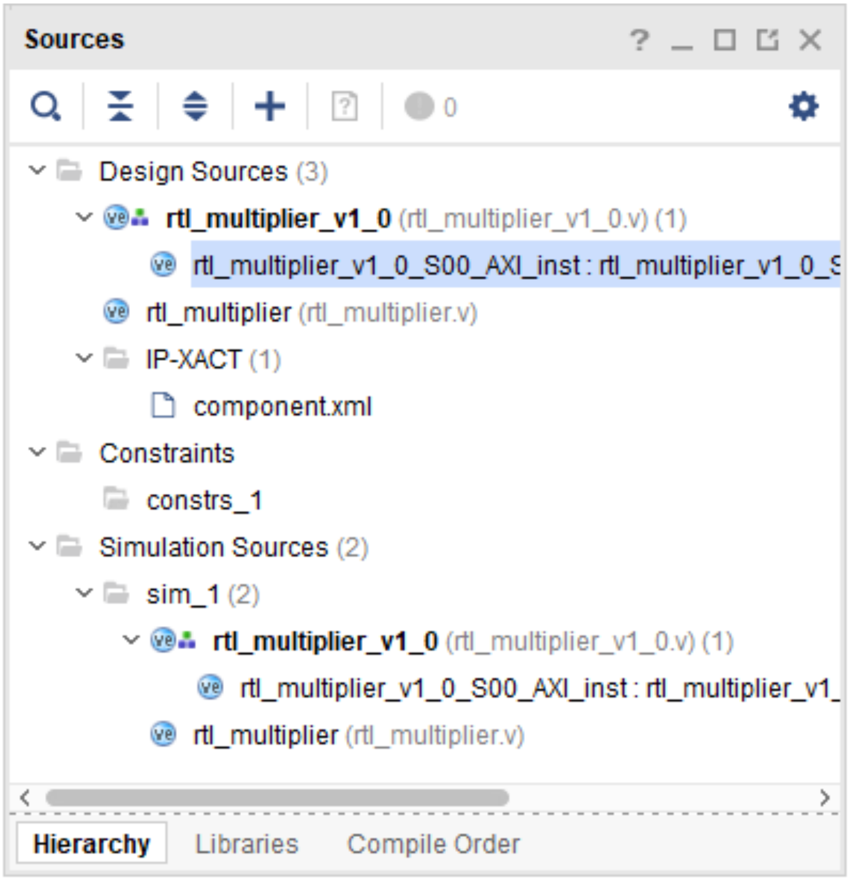
Specify HDL, netlist, Block Design, and IP files, or directories containing those file types to add to your project. Create a new source file on disk and add it to your project.

	Index	Name	Library	Location
	1	rtl_multiplier.v	xil_defaultlib	D:/WS/ip_repo/src

☐ Scan and add RTL include files into project
☑ Copy sources into IP Directory
☑ Add sources from subdirectories

At this point the *rtl_multiplier.v* file is separately part of the *Design Sources*. Let's connect it to the AXI IP block.

- Expand the top branch (*rtl_multiplier_v1_0.v*) and open the file *rtl_multiplier_v1_0_S00_AXI_inst* by double-clicking on it.

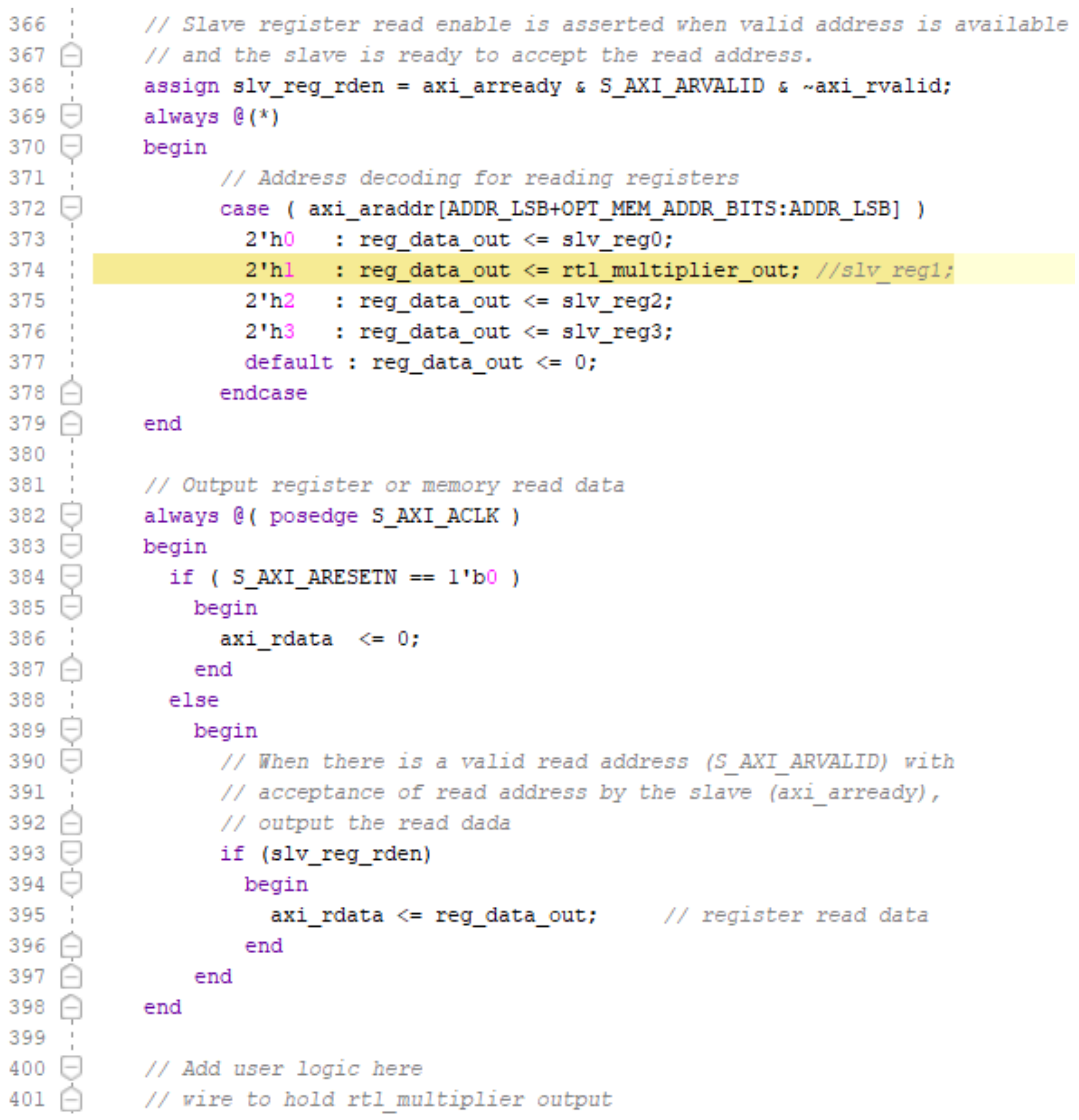


```
Design Sources (3)
  rtl_multiplier_v1_0 (rtl_multiplier_v1_0.v) (1)
    rtl_multiplier_v1_0_S00_AXI_inst : rtl_multiplier_v1_0_S
  rtl_multiplier (rtl_multiplier.v)
  IP-XACT (1)
    component.xml
Constraints
  constrs_1
Simulation Sources (2)
  sim_1 (2)
    rtl_multiplier_v1_0 (rtl_multiplier_v1_0.v) (1)
      rtl_multiplier_v1_0_S00_AXI_inst : rtl_multiplier_v1_
    rtl_multiplier (rtl_multiplier.v)
```

- Scroll down to the end of the file where the comment *Add user logic here* is and insert the code below. The code below instantiates the *rtl_multiplier* module inside the AXI IP block and connects the clock to the AXI clock, the inputs *a* and *b* to the 16 MSB and LSB of the first register (*slv_reg0*) and the output product to a created wire *rtl_multiplier_out*.

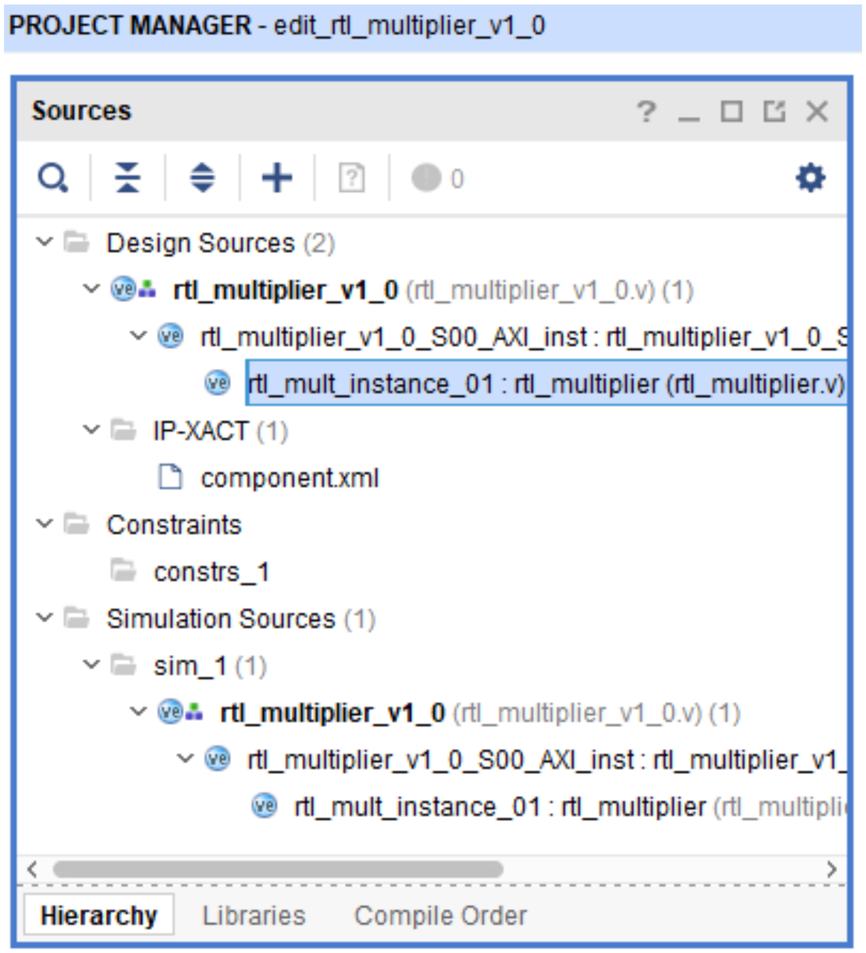
```
// Add user logic here
// wire to hold rtl_multiplier output
wire [31:0] rtl_multiplier_out;
// instantiate the rtl_multiplier
rtl_multiplier rtl_mult_instance_01(
    .clk(S_AXI_ACLK),
    .a(slv_reg0[31:16]),
    .b(slv_reg0[15:0]),
    .product(rtl_multiplier_out)
);
// User logic ends
```

- Finally, set the *rtl_multiplier_out* as the output of one of the AXI registers, as shown below.



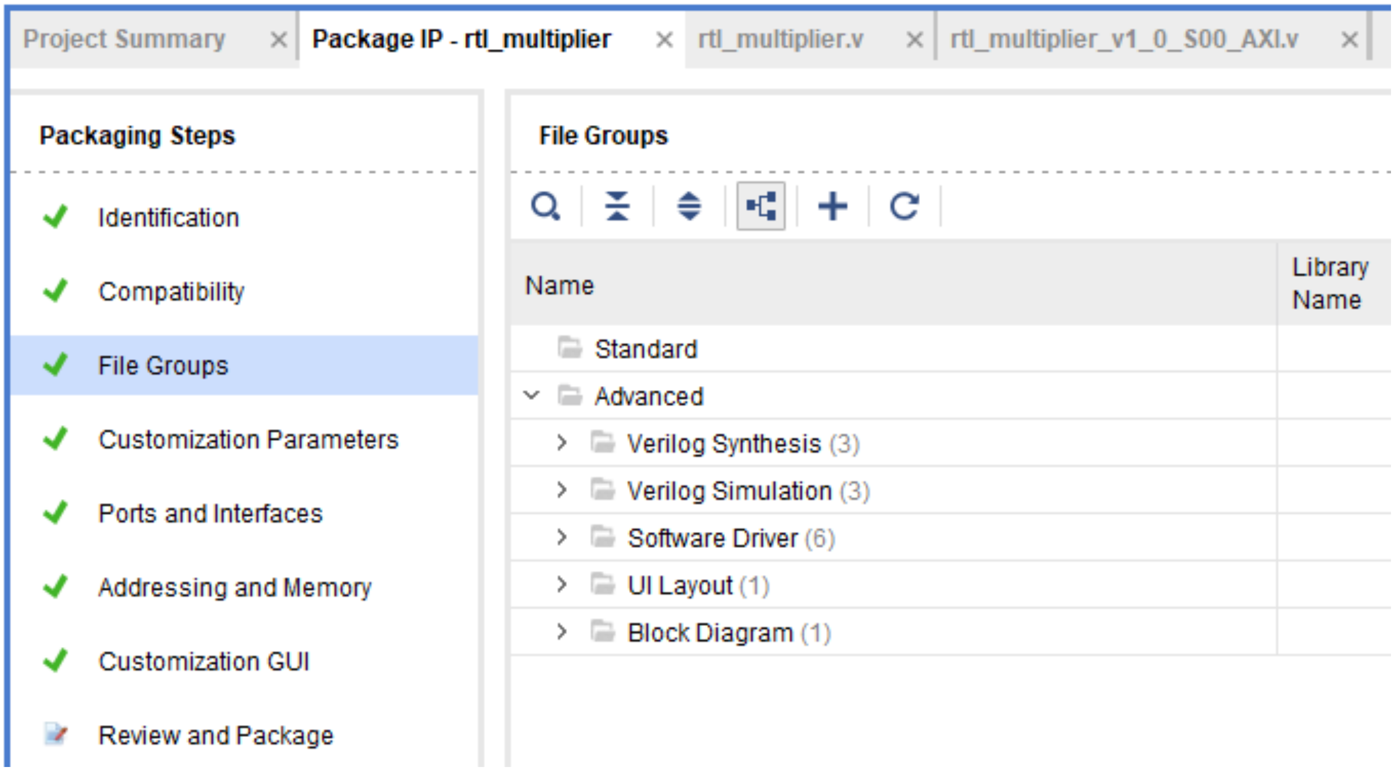
```
    // Slave register read enable is asserted when valid address is available
    // and the slave is ready to accept the read address.
    assign slv_reg_rden = axi_arready & S_AXI_ARVALID & ~axi_rvalid;
    always @(*)
    begin
          // Address decoding for reading registers
          case ( axi_araddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
            2'h0   : reg_data_out <= slv_reg0;
            2'h1   : reg_data_out <= rtl_multiplier_out; //slv_reg1;
            2'h2   : reg_data_out <= slv_reg2;
            2'h3   : reg_data_out <= slv_reg3;
            default : reg_data_out <= 0;
          endcase
    end

    // Output register or memory read data
    always @( posedge S_AXI_ACLK )
    begin
      if ( S_AXI_ARESETN == 1'b0 )
        begin
          axi_rdata  <= 0;
        end
      else
        begin
          // When there is a valid read address (S_AXI_ARVALID) with
          // acceptance of read address by the slave (axi_arready),
          // output the read dada
          if (slv_reg_rden)
            begin
              axi_rdata <= reg_data_out;     // register read data
            end
        end
    end

    // Add user logic here
    // wire to hold rtl_multiplier output
```

After saving the file(s) you will notice that the *rtl_multiplier.v* file has been integrated under the AXI file in the hierarchy. Good work so far! Almost there - just a few more rudimentary steps.


PROJECT MANAGER - edit_rtl_multiplier_v1_0

```
Design Sources (2)
  rtl_multiplier_v1_0 (rtl_multiplier_v1_0.v) (1)
    rtl_multiplier_v1_0_S00_AXI_inst : rtl_multiplier_v1_0_S
      rtl_mult_instance_01 : rtl_multiplier (rtl_multiplier.v)
  IP-XACT (1)
    component.xml
Constraints
  constrs_1
Simulation Sources (1)
  sim_1 (1)
    rtl_multiplier_v1_0 (rtl_multiplier_v1_0.v) (1)
      rtl_multiplier_v1_0_S00_AXI_inst : rtl_multiplier_v1_
        rtl_mult_instance_01 : rtl_multiplier (rtl_multipli
```

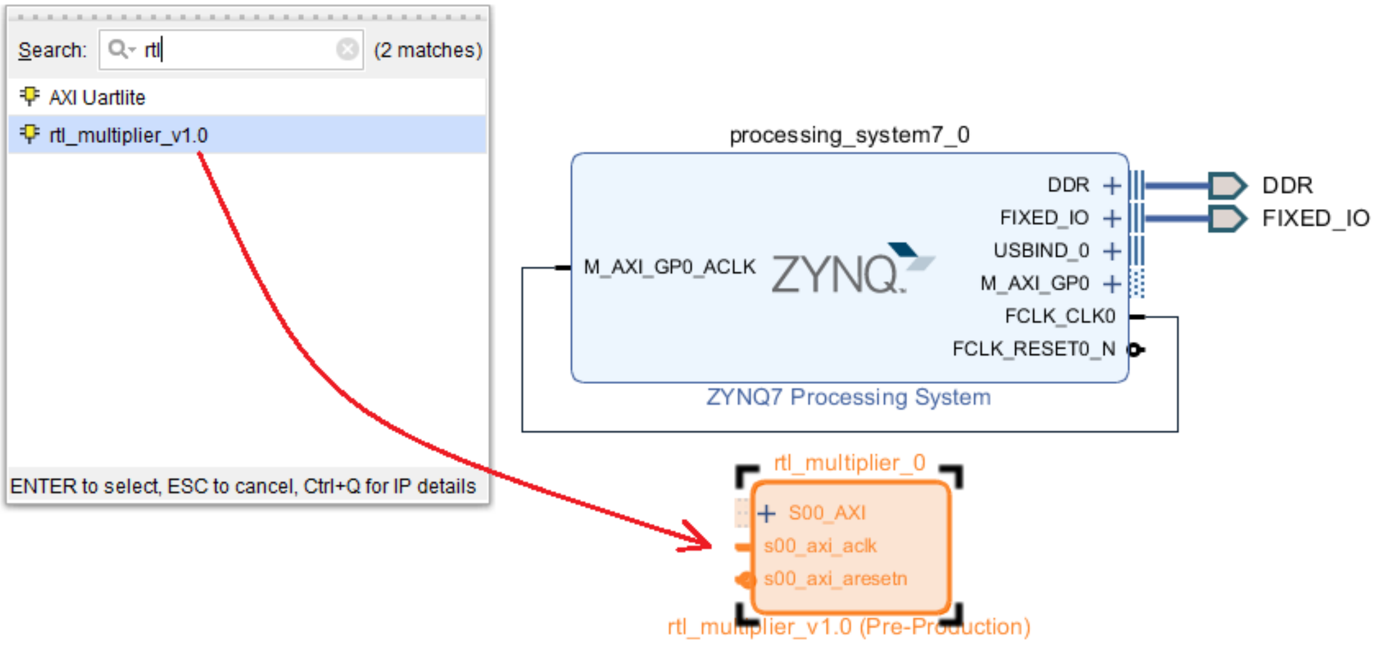
- In the *Package IP* tab, click on *File Groups* and then on *Merge changes from the File Groups Wizard*. This will *OK* the *File Groups* step with a flattering green tick badge of great success. :)



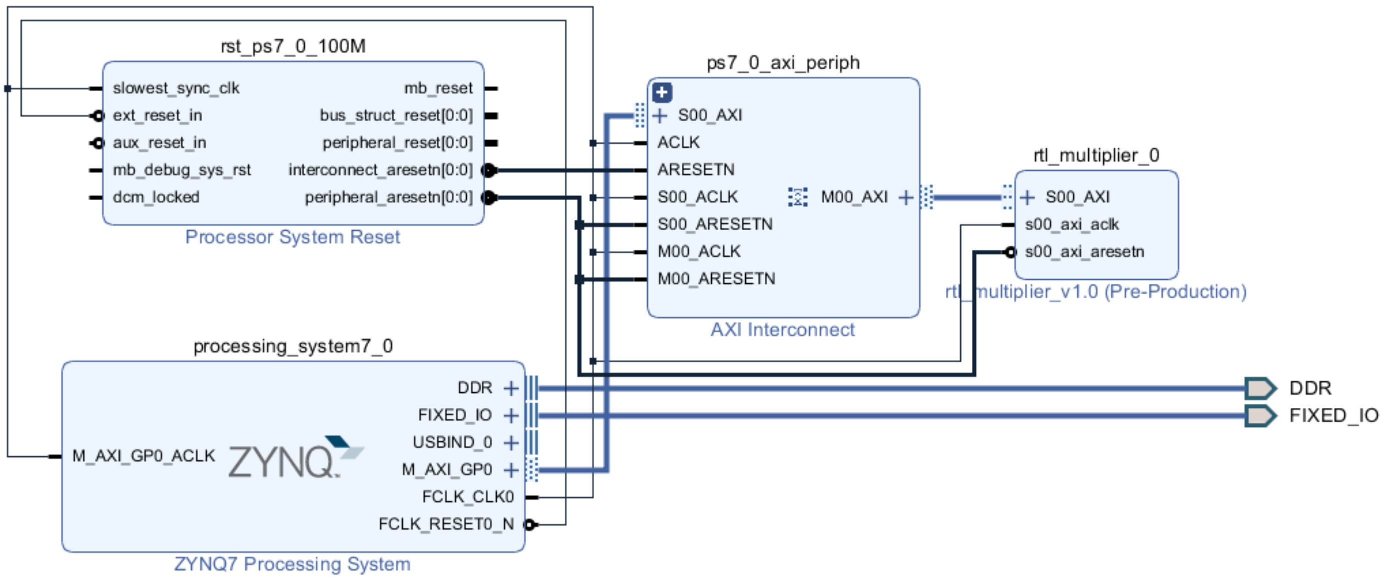
- Next, click on *Review and Package* and proceed with *Re-Package IP*. The IP block will be packaged and you can safely close the project.

Add the IP block to the Zynq PS System

- In the original Vivado project containing the Zynq PS system, click on *Open Block Design* in the *IP Integrator* section to open the design.
- As in the previous lab, to browse for an IP block, click on the *Add IP* (**+**) button and search for our newly custom created *rtl_multiplier*. Double click it to add it to the design.



- Now let Vivado do the "magic" of connecting it to the *ZYNQ7 PS* by clicking on *Run Connection Automation* and use the default settings in the new dialog window. The *Connection Automation* will add a few necessary intermediate IP blocks. It might look scary at first, but fear not! It's just a *Processor System Reset* and an *AXI Interconnect* here and there, no biggie. To make it more clear (and hopefully less scary), click on the *Regenerate Layout* (looks like a Refresh) button. You should see a "neat" design as shown below.



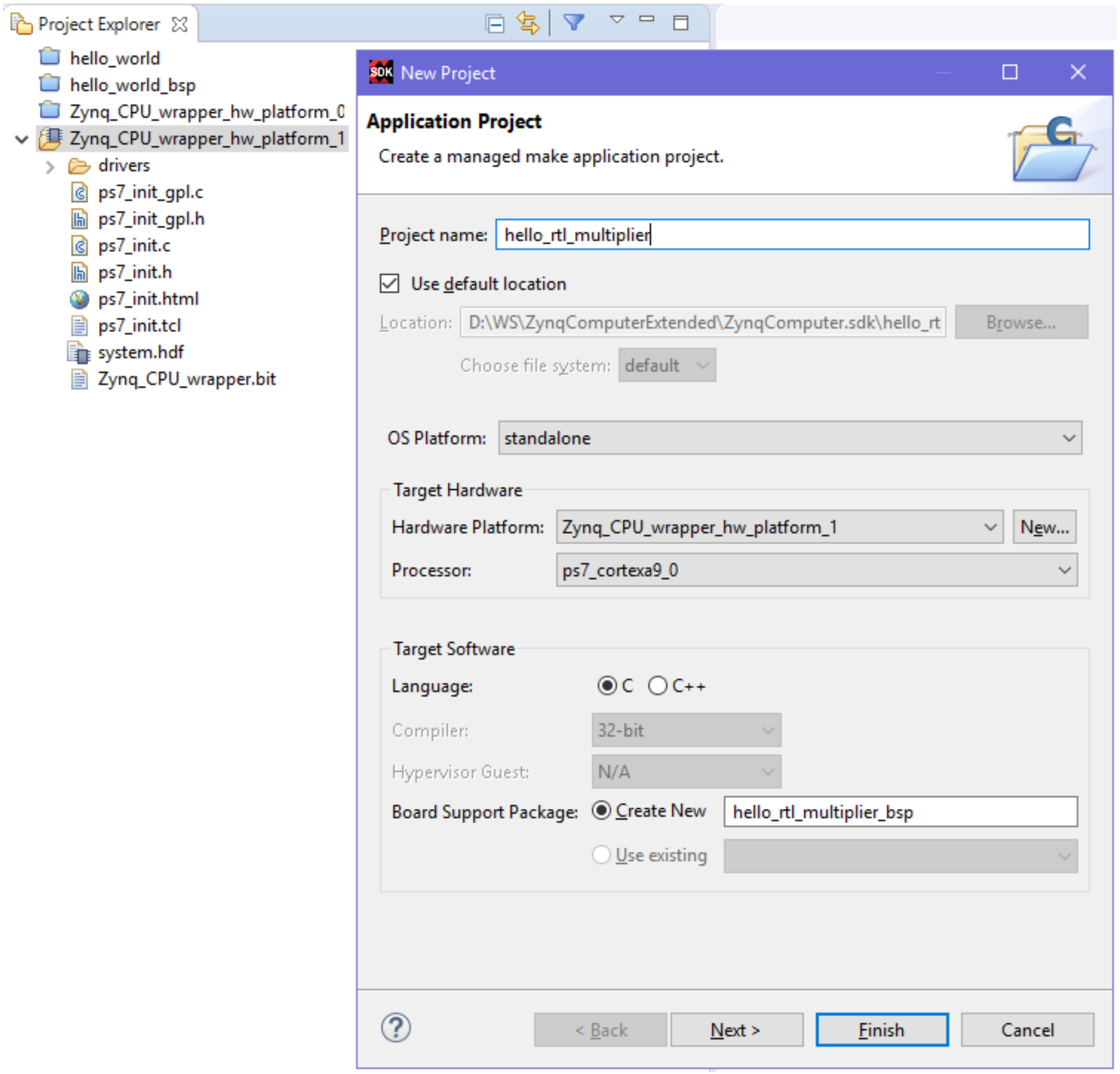
This is all! Now, as in the previous tutorial, save all files and click on *Generate Bitstream*. This will run through Synthesis, Implementation and will generate the bitstream file. Compilation takes a while - see the status in the upper-right corner. When completed, select *Open Implemented Design*. Next, export the hardware design to SDK including the bitstream and then lunch the SDK.

In the Implementation reports, check the resource utilization. Is it reasonable?

Interfacing with the IP Block in Software

So far we have the Zynq PS and the RTL Multiplier as an AXI Lite slave IP block all nicely interconnected, but it is all useless unless we use it in an application - so let's do that.

- *Note*: In case you have previous projects in the SDK, close them and start with the most recent exported hardware platform.
- Create a *File -> New -> Application Project* and give it a name as shown below.



Application Project

Create a managed make application project.

Project name: hello_rtl_multiplier

[✓] Use default location

Location: D:\WS\ZynqComputerExtended\ZynqComputer.sdk\hello_rt Browse...

Choose file system: default

OS Platform: standalone

Target Hardware

Hardware Platform: Zynq_CPU_wrapper_hw_platform_1 New...

Processor: ps7_cortexa9_0

Target Software

Language: [✓ C] C++

Compiler: 32-bit

Hypervisor Guest: N/A

Board Support Package: [✓ Create New] hello_rtl_multiplier_bsp Use existing

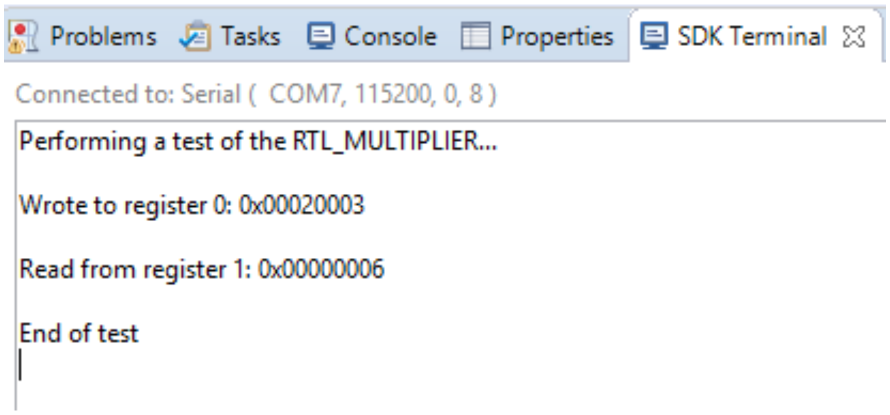
- In the next step, select the *Hello World* template and click *Finish*. The SDK will generate a new application which will appear in the *Project Explorer* as *hello_rtl_multiplier* and *hello_rtl_multiplier_bsp*.
  - *Optional*: Feel free to test that this part works by programming the FPGA, connecting the SDK Terminal and then running the default "Hello World" on the PYNQ-Z1 board.
- Open the *hello_rtl_multiplier/src/helloworld.c* C-source file and update

the file with the code provided below. Understand the code well - it's quite simple.

```
#include "platform.h"
#include "xbasic_types.h"
#include "xparameters.h" // Contains definitions for peripheral RTL_MULTIPLIEF

// we will use the Base Address of the RTL_MULTIPLIER
Xuint32 *baseaddr_p = (Xuint32 *) XPAR_RTL_MULTIPLIER_0_S00_AXI_BASEADDR;

int main() {
    init_platform();
    xil_printf("Performing a test of the RTL_MULTIPLIER... \n\r");

    // Write multiplier inputs to register 0
    *(baseaddr_p + 0) = 0x00020003;
    xil_printf("Wrote to register 0: 0x%08x \n\r", *(baseaddr_p + 0));

    // Read multiplier output from register 1
    xil_printf("Read from register 1: 0x%08x \n\r", *(baseaddr_p + 1));

    xil_printf("End of test\n\n\r");

    cleanup_platform();
    return 0;
}
```

- Save the updated files and run the application on the PYNQ-Z1 board as described in the previous tutorial.
  - Program the FPGA
  - Connect the SDK Terminal to the serial port
  - *Run As -> 4 Launch on Hardware (GDB)*

You should see a number of warnings, but also the successful completion of the test in the SDK Terminal, as shown below.



```
Performing a test of the RTL_MULTIPLIER...

Wrote to register 0: 0x00020003

Read from register 1: 0x00000006

End of test
```

Custom IP block using High-Level Synthesis

There are a number of reasons why High-Level Synthesis (HLS) tools have been developed. Most of all, it has to do with developer productivity and code reuse, as well as other business-related reasons. RTL development in hardware description languages such as Verilog and VHDL is slow, difficult to debug and verify, difficult to update, etc. Moreover, it can not be done by software engineers without an intense training on hardware development. These translate into a high expense for businesses. High-level synthesis attempts to partially solve this problem, by generating HDL code from a higher level language such as C/C++. Further hardware control is enabled by the use of *HLS PRAGMAS*. Designs made using HLS are typically not the most optimal possible solutions, even after several optimizations, but are often considered acceptable considering the low engineering cost and the throughput gains.

In this lab, we will only get started with HLS, by reimplementing the 16-bit multiplier using C++ and HLS to generate the AXI-Lite slave IP block.

Creating the Vivado HLS project

- Open *Vivado HLS* and proceed with *Create New Project*.
- Call the project *hls_multiplier* and locate it in your group's working

folder.

- Similarly, in the *Top Function* field write *hls_multiplier*. The *Top Function* is the C/C++ function that will be translated to HDL by the HLS algorithm. If it calls other functions, they will also be translated to HDL.
- Leave the *Solution Name* as default, the *Period* is 10 ns since the board frequency is 100 MHz, and for the *Part* please select *xc7z020clg400-1*, which is the chip on the PYNQ-Z1 board.

You will notice that Vivado HLS looks less "busy" compared to Vivado. This is because Vivado HLS only simulates, synthesizes and packages the IP block, but does not interface with the hardware directly. The output from Vivado HLS is to be later imported into a Vivado project. Let's add the necessary code to implement and test the HLS multiplier AXI Lite slave IP block.

- In the *Explorer*, right click on *Source* and select *New File...*. Locate the file in the suggested directory and name it *hls_multiplier.cpp*. The code for this file is given below.

```
unsigned int hls_multiplier(unsigned short int a, unsigned short int b) {
#pragma HLS INTERFACe s_axilite port=return bundle=CRTLS
#pragma HLS INTERFACe s_axilite port=a bundle=CRTLS
#pragma HLS INTERFACe s_axilite port=b bundle=CRTLS
    unsigned int p;
    p = a * b;
    return p;
}
```

The code is really quite simple. The *hls_multiplier* C++ function takes in two *unsigned short int* arguments *a* and *b*, 16-bits each, and returns an *unsigned int* which is 32-bit. Inside, the product *p* is declared as a 32-bit *unsigned int*, is given the value *a * b* and is returned. The *pragma HLS INTERFACE* lines specify that the interface to be used is a slave AXI Lite and are all into the same bundle.

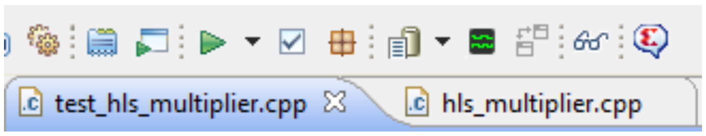
- Next, right-click on *Test Bench* and select *New File...*. Similarly, store the file in the suggested location and name it *test_hls_multiplier.cpp*. The code for this file is presented below.

```
#include <stdio.h>
unsigned int hls_multiplier(unsigned short int a, unsigned short int b);

int main() {
    unsigned short int a, b;
    unsigned int p;
    a = 2;
    b = 3;
    p = 0;
    printf("initialized variables: a=%d, b=%d, p=%d \n", a, b, p);
    p = hls_multiplier(a, b);
    printf("testing hls_multiplier: %d * %d = %d \n", a, b, p);
    return 0;
}
```

Again, the code is very simple - please read it and understand what is going on.

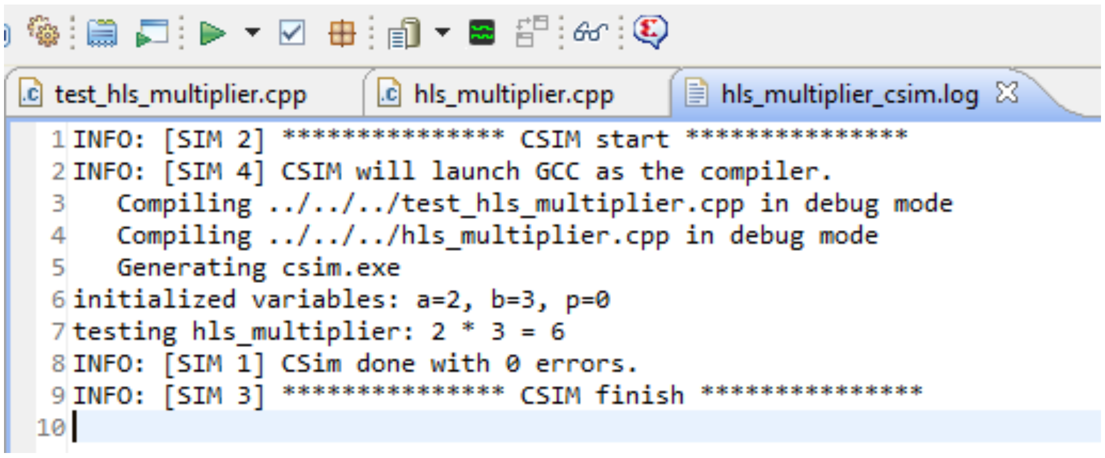
Testing the *hls_multiplier* and exporting the IP block

Vivado HLS offers a few ways to test your design, both as C Simulation and as C/RTL Cosimulation. To start a simulation - use the comfortably ordered buttons on the top area.



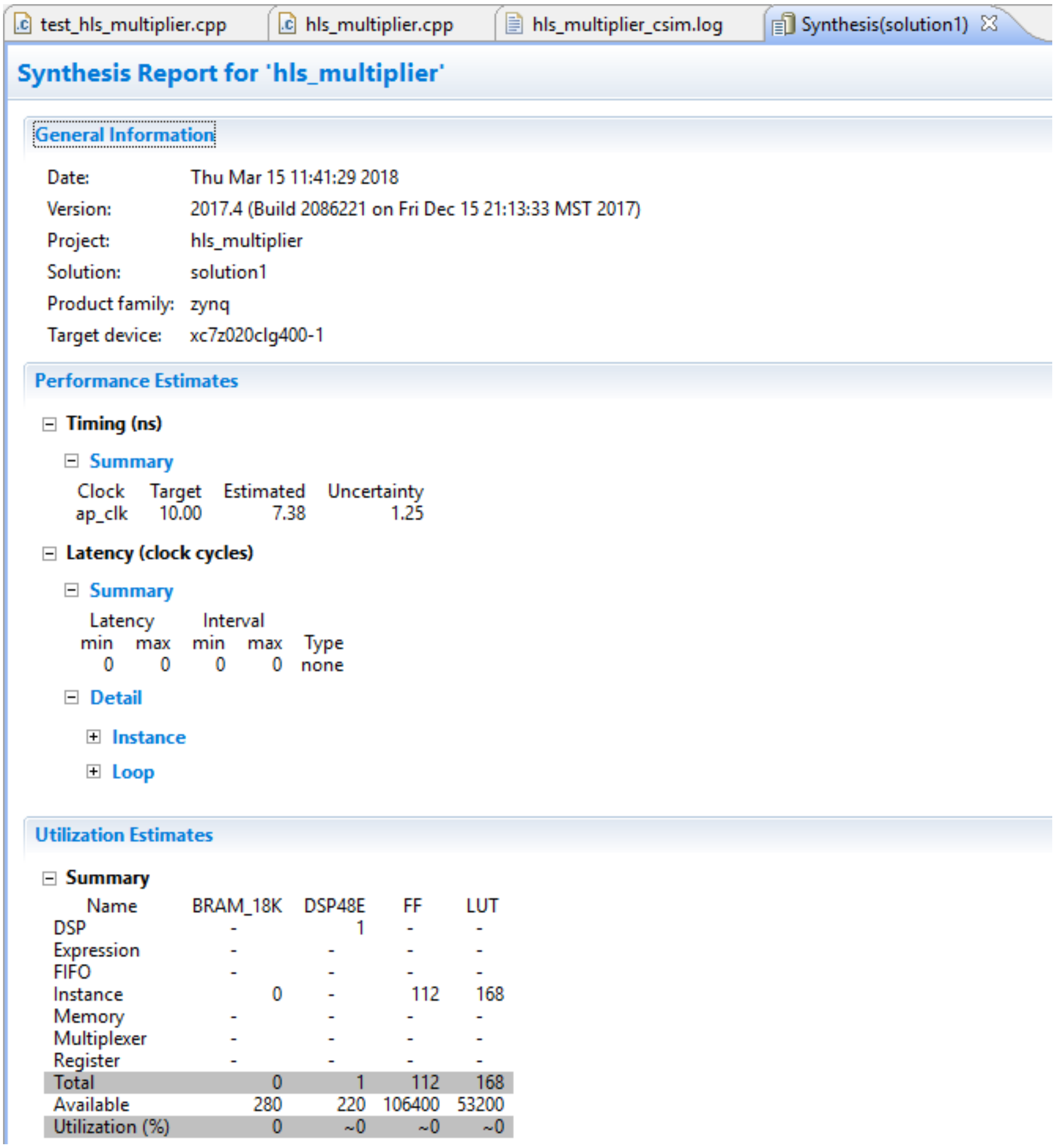
- First, click the *Index C Source* button (and watch as it doesn't really do anything visible).

- Next, click on the *Run C Simulation* button. In the *C Simulation Dialog* feel free to enable the *Clean Build* option. You should see the success message *testing hls_multiplier: 2 * 3 = 6*.



```
INFO: [SIM 2] *************** CSIM start ***************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
   Compiling ../../../test_hls_multiplier.cpp in debug mode
   Compiling ../../../hls_multiplier.cpp in debug mode
   Generating csim.exe
initialized variables: a=2, b=3, p=0
testing hls_multiplier: 2 * 3 = 6
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```

- Next, click on the *C Synthesis* button. This button starts the translation of the C/C++ code to HDL (both Verilog and VHDL), synthesizes it for the targeted chip and the targeted frequency, and generates a report containing timing information and even an *Utilization Estimate*. **Do you observe anything interesting in the Utilization Estimate?**


test_hls_multiplier.cpp | hls_multiplier.cpp | hls_multiplier_csim.log | Synthesis(solution1)

Synthesis Report for 'hls_multiplier'

General Information

Date:	Thu Mar 15 11:41:29 2018
Version:	2017.4 (Build 2086221 on Fri Dec 15 21:13:33 MST 2017)
Project:	hls_multiplier
Solution:	solution1
Product family:	zynq
Target device:	xc7z020clg400-1

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	7.38	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
0	0	0	0	none

Detail

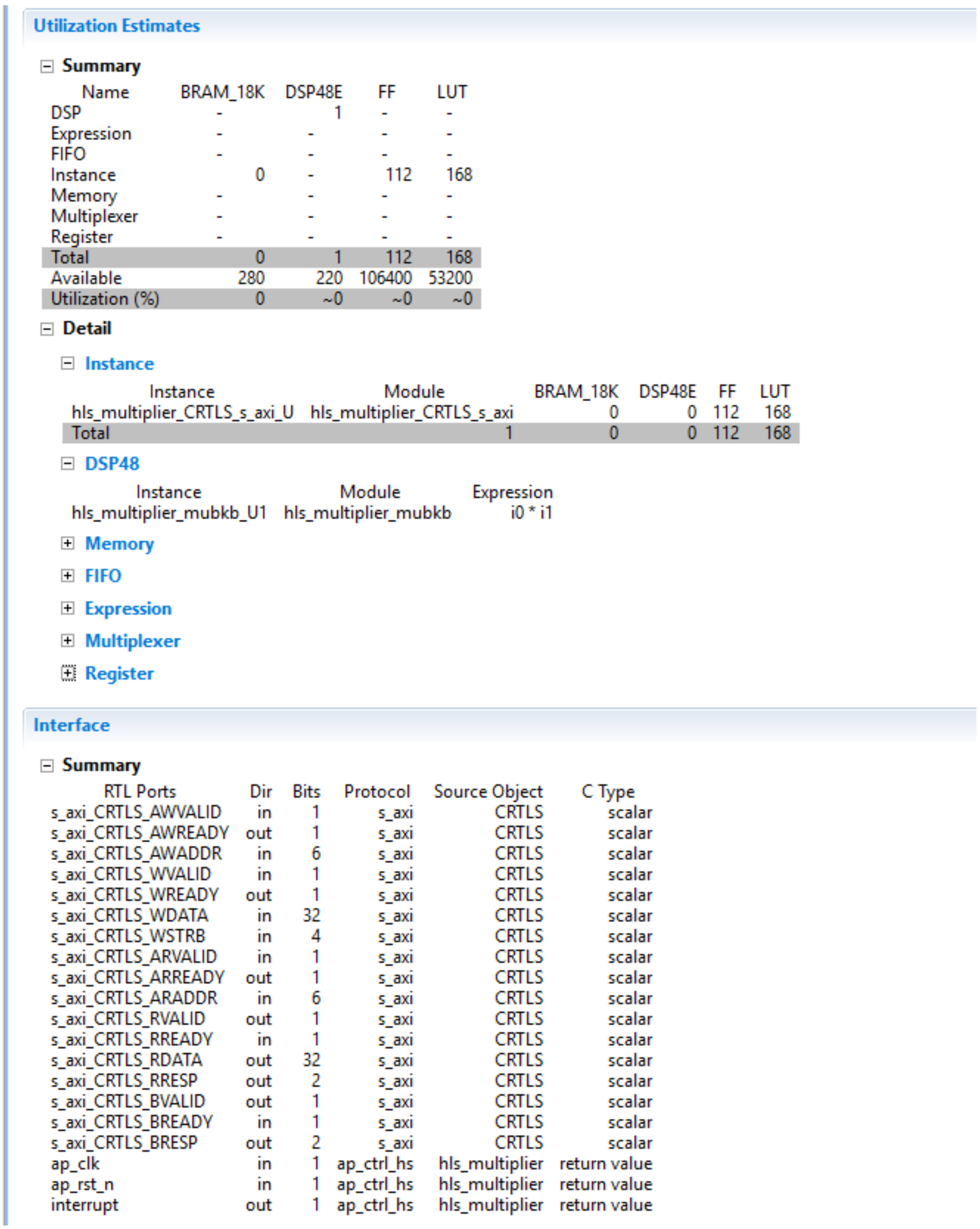
Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	1	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	0	-	112	168
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	0	1	112	168
Available	280	220	106400	53200
Utilization (%)	0	~0	~0	~0



Utilization Estimates

⊟ Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	1	-	-
Expression	-	-	-	-
FIFO	-	-	-	-
Instance	0	-	112	168
Memory	-	-	-	-
Multiplexer	-	-	-	-
Register	-	-	-	-
Total	0	1	112	168
Available	280	220	106400	53200
Utilization (%)	0	~0	~0	~0

⊟ Detail

⊟ Instance

Instance	Module	BRAM_18K	DSP48E	FF	LUT
hls_multiplier_CRTLS_s_axi_U	hls_multiplier_CRTLS_s_axi	0	0	112	168
Total	1	0	0	112	168

⊟ DSP48

Instance	Module	Expression
hls_multiplier_mubkb_U1	hls_multiplier_mubkb	i0 * i1

⊞ Memory

⊞ FIFO

⊞ Expression

⊞ Multiplexer

⊞ Register

Interface

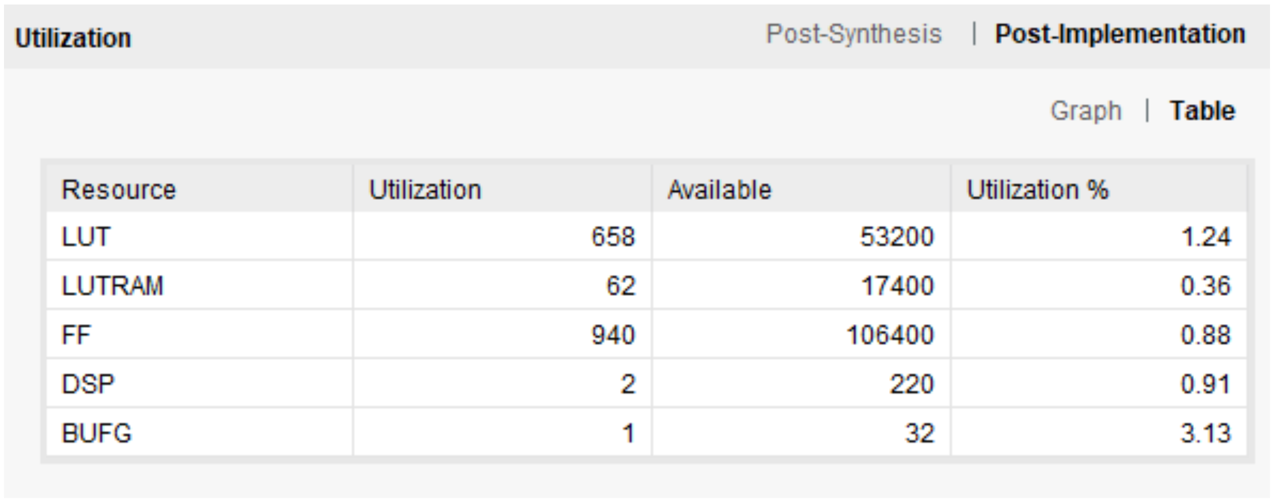
⊟ Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
s_axi_CRTLS_AWVALID	in	1	s_axi	CRTLS	scalar
s_axi_CRTLS_AWREADY	out	1	s_axi	CRTLS	scalar
s_axi_CRTLS_AWADDR	in	6	s_axi	CRTLS	scalar
s_axi_CRTLS_WVALID	in	1	s_axi	CRTLS	scalar
s_axi_CRTLS_WREADY	out	1	s_axi	CRTLS	scalar
s_axi_CRTLS_WDATA	in	32	s_axi	CRTLS	scalar
s_axi_CRTLS_WSTRB	in	4	s_axi	CRTLS	scalar
s_axi_CRTLS_ARVALID	in	1	s_axi	CRTLS	scalar
s_axi_CRTLS_ARREADY	out	1	s_axi	CRTLS	scalar
s_axi_CRTLS_ARADDR	in	6	s_axi	CRTLS	scalar
s_axi_CRTLS_RVALID	out	1	s_axi	CRTLS	scalar
s_axi_CRTLS_RREADY	in	1	s_axi	CRTLS	scalar
s_axi_CRTLS_RDATA	out	32	s_axi	CRTLS	scalar
s_axi_CRTLS_RRESP	out	2	s_axi	CRTLS	scalar
s_axi_CRTLS_BVALID	out	1	s_axi	CRTLS	scalar
s_axi_CRTLS_BREADY	in	1	s_axi	CRTLS	scalar
s_axi_CRTLS_BRESP	out	2	s_axi	CRTLS	scalar
ap_clk	in	1	ap_ctrl_hs	hls_multiplier	return value
ap_rst_n	in	1	ap_ctrl_hs	hls_multiplier	return value
interrupt	out	1	ap_ctrl_hs	hls_multiplier	return value

- Now that everything seems reasonable at the C Simulation level and at the C Synthesis level, click on the *Run C/RTL Cosimulation* button. Use the default settings in the *Co-simulation Dialog*. This can take a long time since this is quite a powerful operation. It simulates both the C and the generated RTL and compares the final results and ensures that they match. In the generated report you are expecting the word *Pass* - which means the test was successful. Now that all these tests passed, it's time to export the design to an IP block.
- Click on the *Export RTL* button and go with the default options. You can now close Vivado HLS.

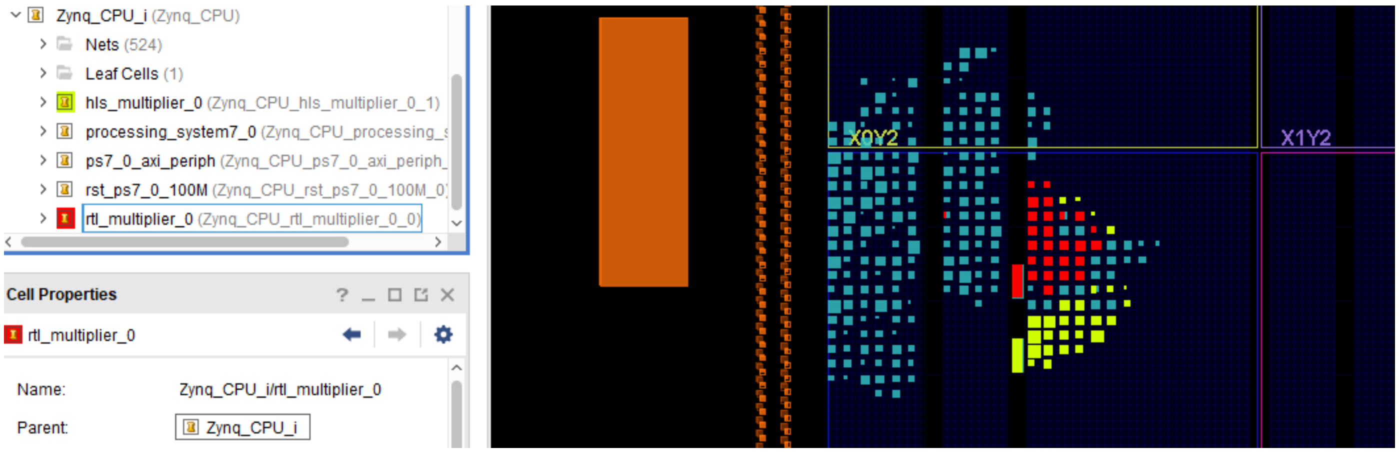
Adding the *hls_multiplier* AXI Lite Slave IP Block to your Vivado design

- In Vivado, go back to *Tools -> Settings...* and under *IP -> Repository* add the *hls_multiplier/solution1/impl/ip* directory. This will allow Vivado to find the IP we just created using Vivado HLS.
- Next, in the *Block Design*, click on *Add IP* and search for the *hls_multiplier*. Add it to your design and *Run Connection Automation* to automatically have it connected to the *AXI Interconnect*.
- Save all files and click on *Generate Bitstream* to re-run Synthesis and Implementation over the updated design. Check the resource utilization and look over the implementation. Example results are shown below. In the second image, the *hls_multiplier* area is highlighted yellow, while the *rtl_multiplier* area is highlighted red.



Utilization
Post-Synthesis | Post-Implementation
Graph | Table

Resource	Utilization	Available	Utilization %
LUT	658	53200	1.24
LUTRAM	62	17400	0.36
FF	940	106400	0.88
DSP	2	220	0.91
BUFG	1	32	3.13



Zynq_CPU_i (Zynq_CPU)
Nets (524)
Leaf Cells (1)
hls_multiplier_0 (Zynq_CPU_hls_multiplier_0_1)
processing_system7_0 (Zynq_CPU_processing_s
ps7_0_axi_periph (Zynq_CPU_ps7_0_axi_periph_
rst_ps7_0_100M (Zynq_CPU_rst_ps7_0_100M_0)
rtl_multiplier_0 (Zynq_CPU_rtl_multiplier_0_0)
Cell Properties
rtl_multiplier_0
Name: Zynq_CPU_i/rtl_multiplier_0
Parent: Zynq_CPU_i
X0Y2
X1Y2

Now let's interface both multipliers into Xilinx SDK.

Interfacing with the RTL and HLS blocks in Software

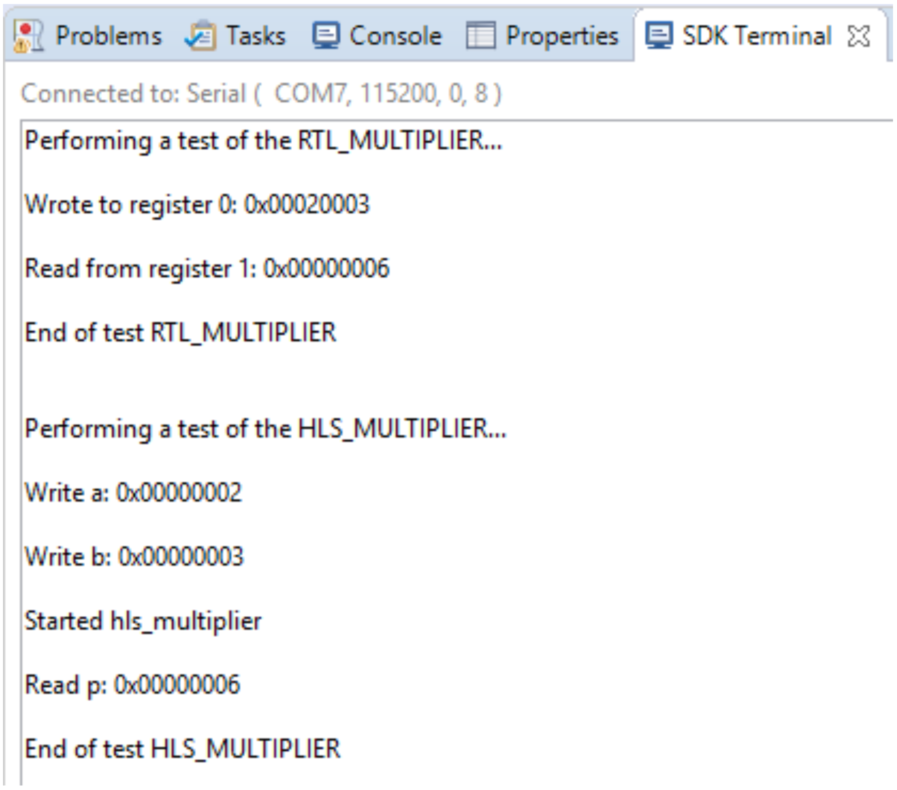
- As before, *Export Hardware*... and *Launch SDK*.
- Close the previous *hello_rtl_multiplier* and *hello_rtl_multiplier_bsp* projects and create a new application called *hello_hls_rtl_m* starting with the *Hello World* template.
- In the *helloworld.c* file paste the following code.

Media:Hello_hls_rtl_m.c

The first part of this code does the same as before, testing the *rtl_multiplier*. The second part of the code tests the *hls_multiplier*. Vivado

HLS generates not only an HDL implementation of the IP block, but also functions that enable the user to use the IP block correctly. The Code might appear more involved, but try to understand it.

- Build and run the code on the PYNQ-Z1 board. The result from the SDK Terminal is shown below.



```
Connected to: Serial ( COM7, 115200, 0, 8 )
Performing a test of the RTL_MULTIPLIER...

Wrote to register 0: 0x00020003

Read from register 1: 0x00000006

End of test RTL_MULTIPLIER


Performing a test of the HLS_MULTIPLIER...

Write a: 0x00000002

Write b: 0x00000003

Started hls_multiplier

Read p: 0x00000006

End of test HLS_MULTIPLIER
```

Questions

- How does the Zynq PS communicate with the IP blocks we created in this lab?
- What was the resource utilization of the *rtl_multiplier* and the *hls_multiplier*?
- Imagine that you have to create an IP block implementing 10 different sorting algorithms. Would you rather use a hardware description language or High-level Synthesis? Why?

Assignment

Using Vivado HLS, implement an IP block as an AXI Lite Slave that takes in two numbers as arguments and returns the *div* and *mod* of them. Add the *hls_divider* to your Block Design and test it in the SDK. Next, implement an ALU (Arithmetic Logic Unit), i.e. a block that can perform a number of arithmetic operations (product, addition, subtraction, division, etc.). The ALU takes as arguments the number values and also the operation, and returns the result.

Bonus: If you would like a bit of a challenge - measure the execution time of AES when using only the ARM CPU, and then when using the FPGA AES IP block, and observe the difference.

References

This tutorial was used as a reference for the RTL multiplier

This YouTube tutorial is similar for the RTL multiplier part

Xilinx Vivado User Guide on Creating and Packaging Custom IP

"High-Level Synthesis for FPGAs: From Prototyping to Deployment" - great paper by Jason Cong

Great video tutorial on making an AXI Lite slave IP block

Get the PYNQ-Z1 board files for Vivado