

Hardware/Software Co-Design

Introductory Lab

February 2018

The objective of this initial laboratory is to introduce you through the process of using Vivado and SDK (Software Development Kit) to create a simple ARM Cortex-A9 based processor design targeting a Zynq device on the Zybo board and execute a small software application on it.

This guide will provide an introduction to:

- Create a Vivado project for a Zynq system
- Use the IP Integrator to create a ARM-based hardware system
- Create a software applications, to execute on the ARM processor, using (SDK)
- Run the example application on the Zybo board

The students can additionally consult the laboratory tutorials of the XUP (Xilinx University Program) available from the site

<http://www.xilinx.com/support/university/vivado/vivado-workshops/Vivado-embedded-design-flow-zynq.html> and/or the tutorials available at the ZynqBook site <http://www.zynqbook.com/download-tuts.html>

1. Create a Vivado Project

1.1. Create New Project targeting the Zybo board

1.1.1. Open Vivado

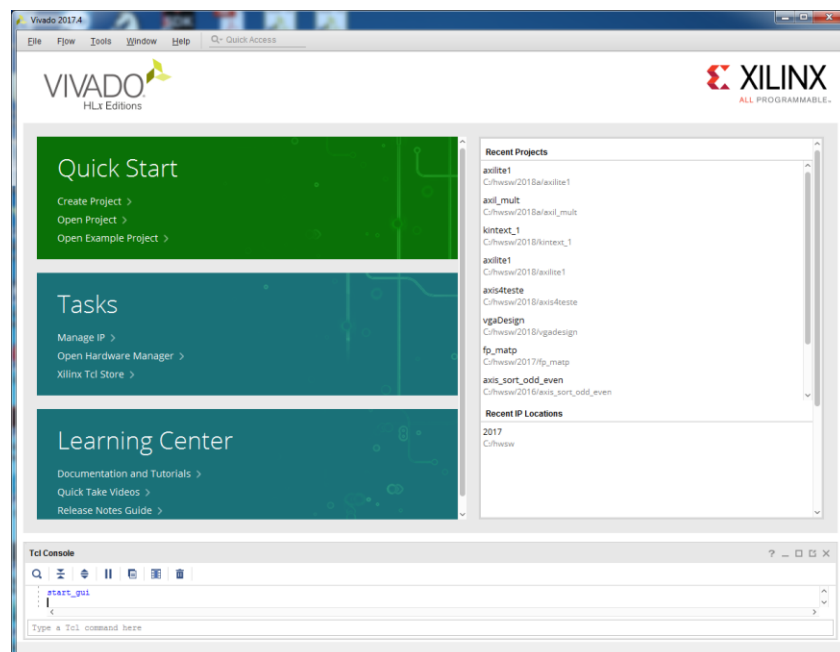


Figure 1. Vivado initial window.

1.1.2. Create New Project

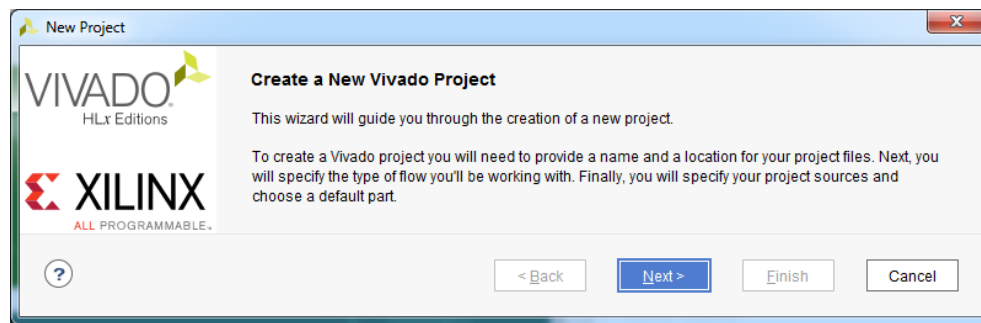


Figure 2. New project window.

1.1.3. Select project name

Next, select project name (in this example, **lab0**) and location (in this example, **C:\hws\2018a**) and create a new project (use the default project file name **system.xmp**) using the AXI interface system.

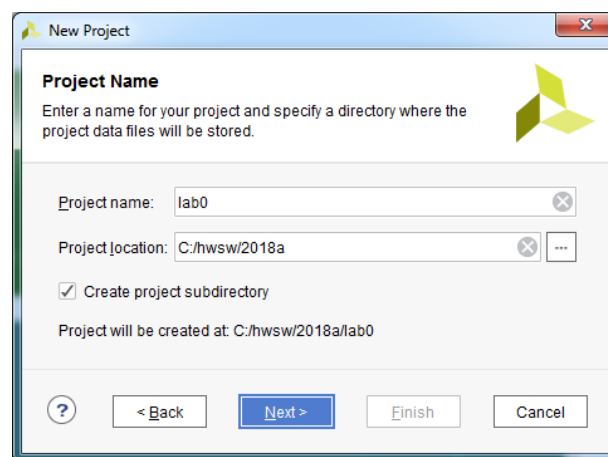


Figure 3. Project Name and Location

Note: avoid using spaces and/or special characters on your directory path and on the filenames.

1.1.4. Select the Project Type

Next,
select project type as
RTL Project.

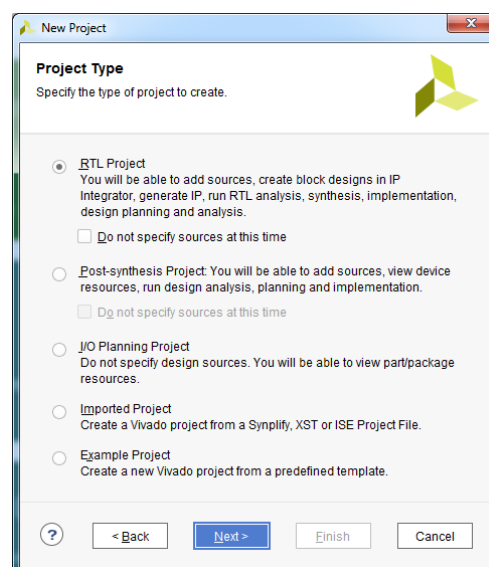


Figure 4. Project Type selection window.

1.1.5. Add Sources

Next in the Add Sources form, select VHDL as the Target language and Mixed as the Simulator language

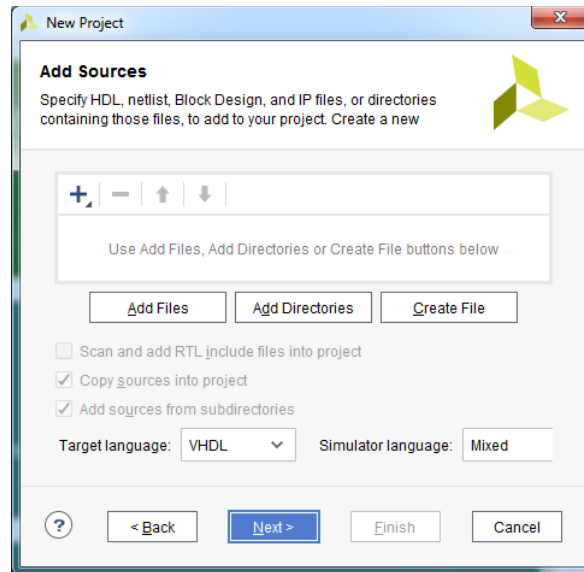


Figure 5. Add Sources window.

1.1.6. Board Selection

Next two more times to skip Adding Existing IP and Add Constraints.

In the Default Part form, select Boards, and then the Zybo board.

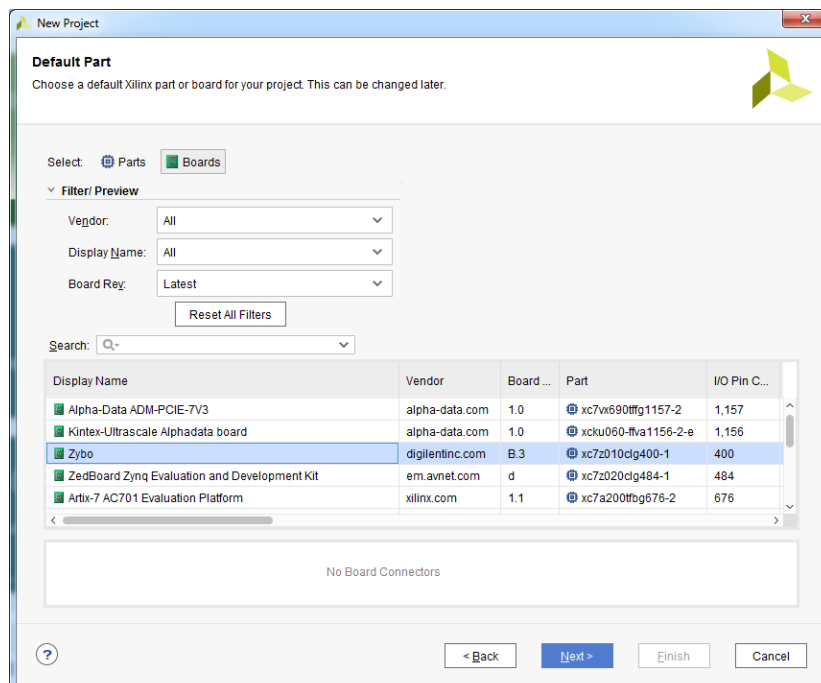


Figure 5. Board Selection window.

Note: if there's no Zybo in the board list, the ZYBO.zip file package of the board must be first downloaded from

<https://www.xilinx.com/support/documentation/university/vivado/workshops/vivado-embedded-design-flow-zynq/materials/2015x/ZYBO/zybo.zip>

and then extracted to the <Vivado_2017_4_install_dir>\Vivado\2017.4\data\boards\board_parts\zynq directory.

Next, check the Project Summary and Finish to create an empty Vivado project.

2. Create the system using the IP integrator

2.1. Create a new Block Design, with the ZYNQ processing system

2.1.1. Create a Block Design

In the Flow Navigator, create a Block Design (under IP Integrator) with the design name **system**.

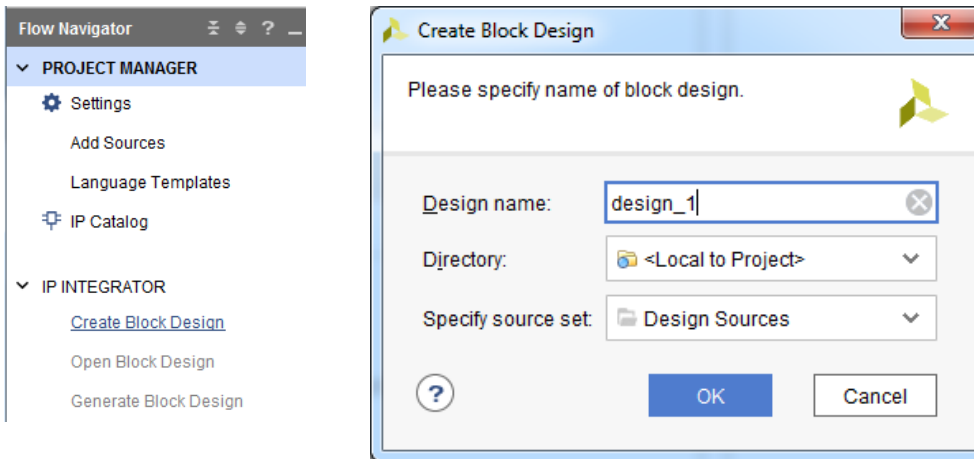


Figure 6. Create new block design.

2.1.2. Add the Processing System

IP components can be added from the catalog by clicking the Add IP icon in the block diagram side bar, or by right-clicking anywhere in the Diagram workspace and selecting Add IP.

Select the Zynq7 Processing System and add it to the design.

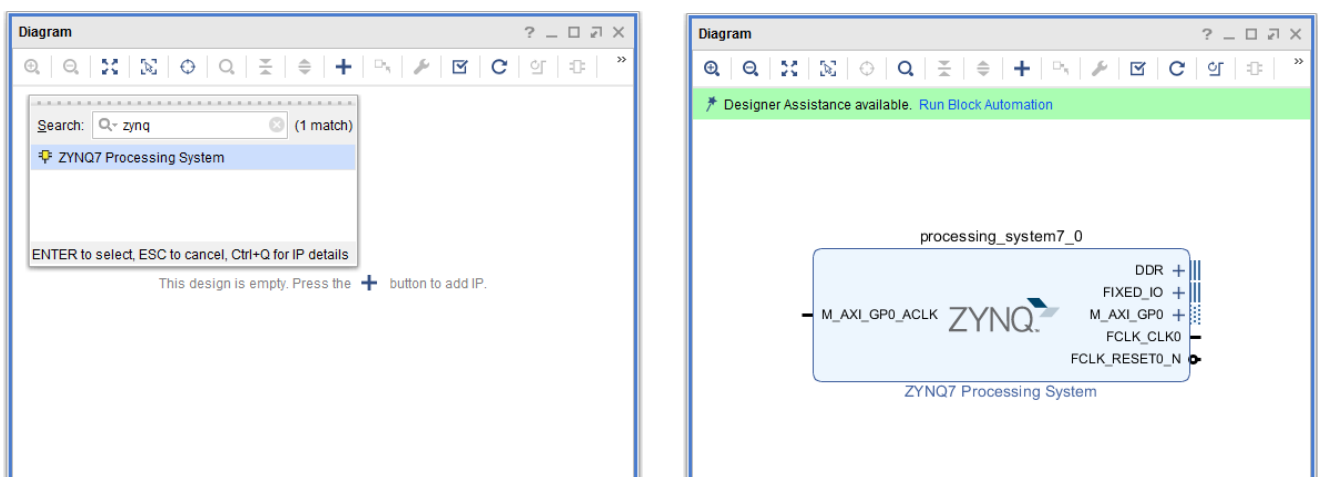


Figure 6. Adding IP.

2.1.3. Automatically connect fixed IOs

Run Block Automation with the default settings

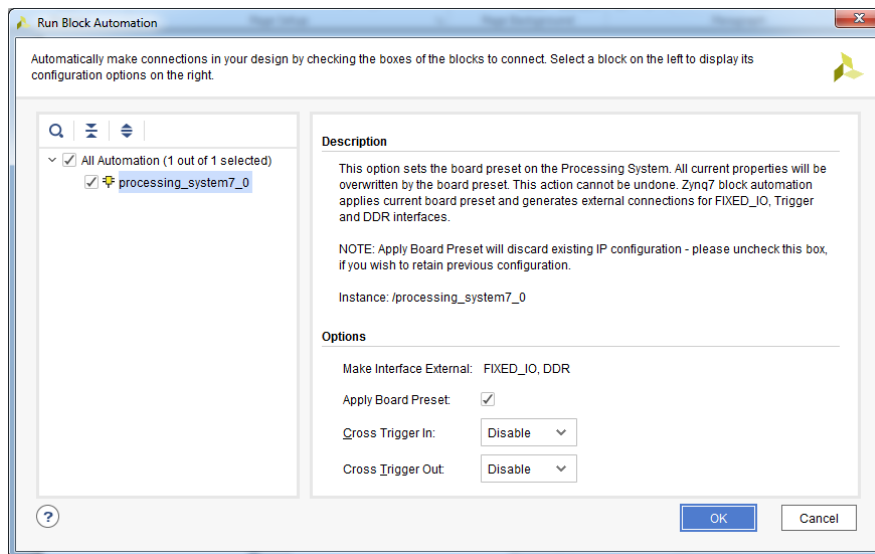


Figure 7. Block Automation settings.

Ports are automatically added for the DDR and Fixed IO. An imported configuration for the Zynq PS system related to the Zybo board has been applied which can be customized.

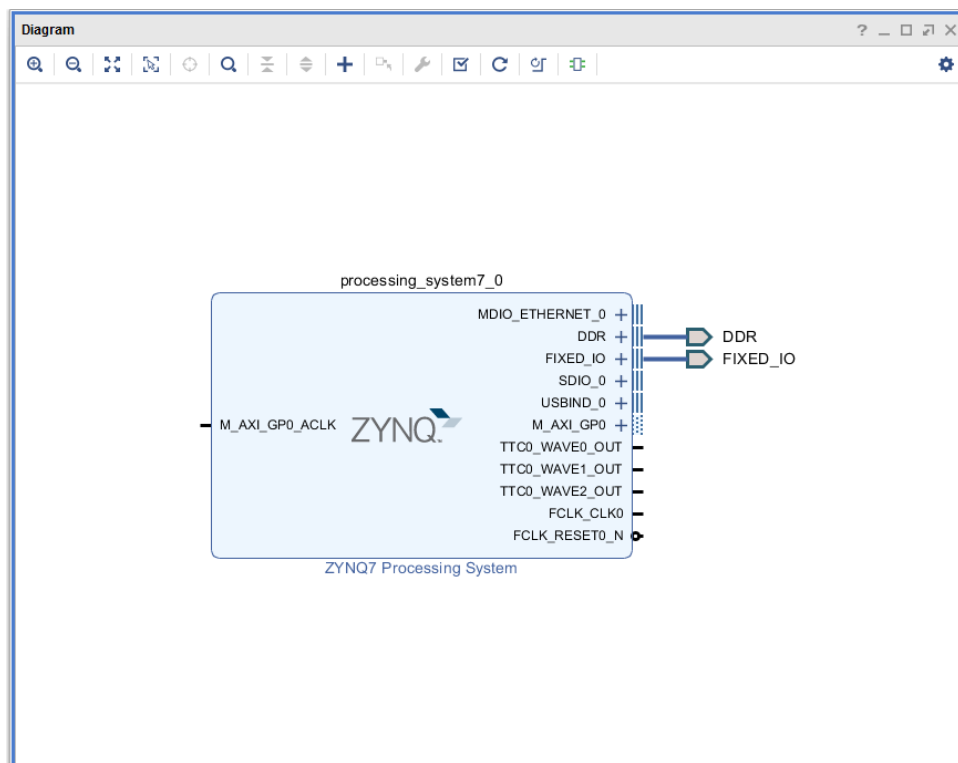


Figure 7. Zynq PS block with DDR and Fixed_IO Ports.

2.2. Customize the Zynq Processing System

2.2.1. Configure the processing block with only one UART peripheral

Double-click on the PS block to open its customization window.

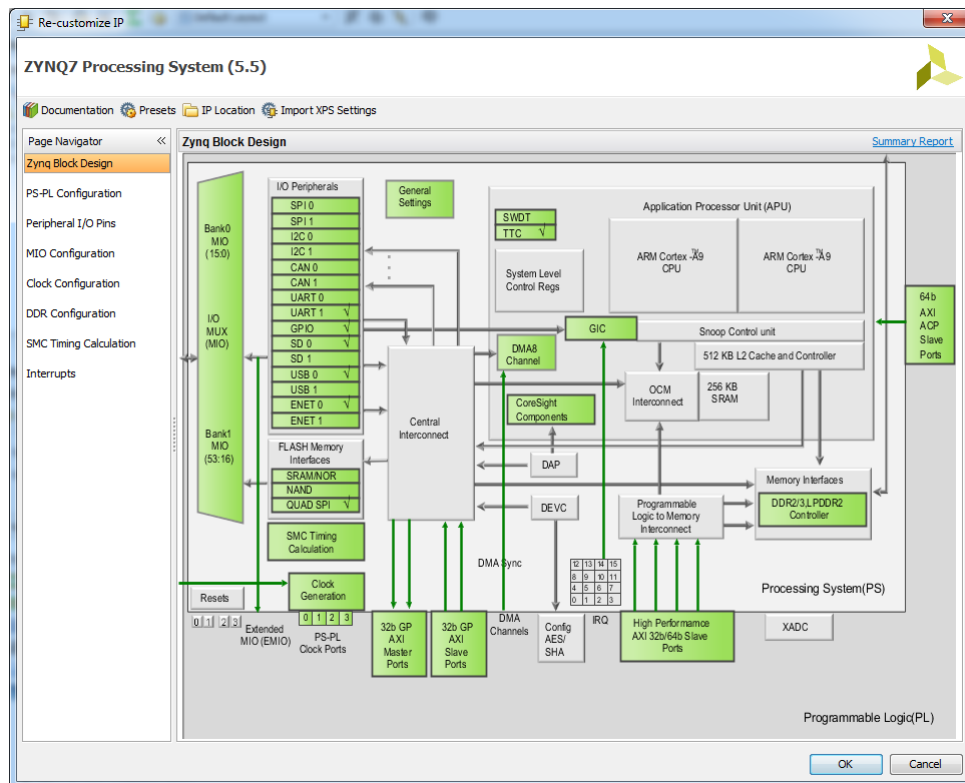


Figure 8. PS Customization Window.

Open the MIO configuration form and ensure all the I/O are deselected except UART 1, i.e. deselect ENET 0, USB 0, SD 0 (I/O Peripherals), GPIO MIO (GPIO), Quad SPI Flash (Memory Interfaces), and Timer 0 (Application Processor Unit).

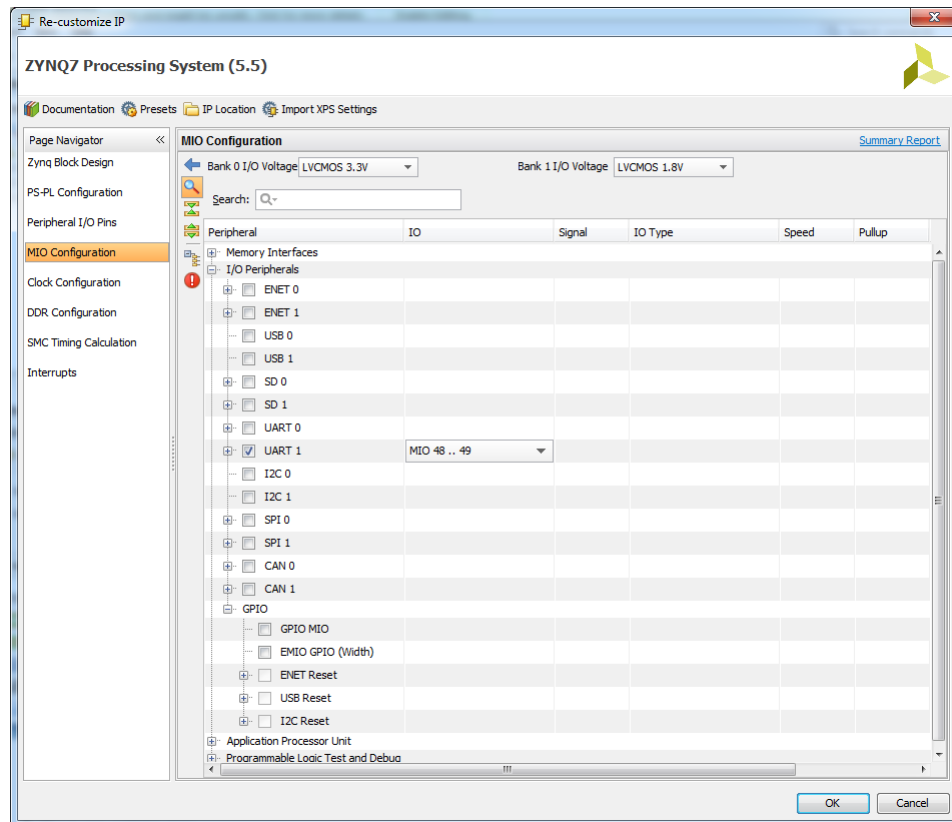


Figure 9. MIO Configuration.



2.2.2. PS-PL Configuration

On the PS-PL Configuration window, deselect the M AXI GP0 interface (AXI Non Secure Enablement → GP Master AXI interface), and deselect the FCLK_RESET0_N option (General → Enable Clock Resets).

2.2.3. Clock Configuration

On the Clock Configuration window, deselect the FCLK_CLK0 option (PL Fabric Clocks).

2.2.4. Finalize the PS Configuration

Finalize the processing system configuration, regenerate the layout (click on the  icon), and validate the design (click on the  icon).

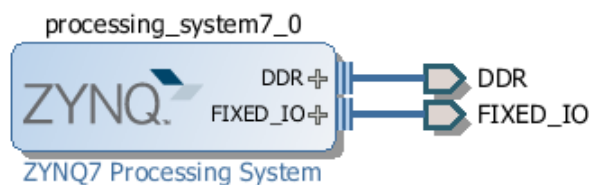


Figure 10. Updated Zynq PS Block.

2.3. System Hardware generation

2.3.1. Generate Block Design

Click on Generate Block Design (on the Flow Navigator) to generate the Implementation, Simulation and Synthesis files for the design.

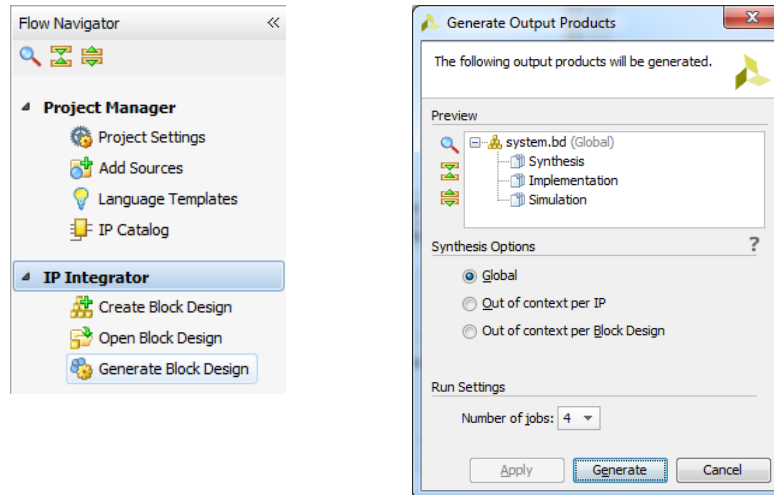


Figure 11. Generate Block Design

2.3.2. Create HDL Wrapper

In the sources panel, right-click on *design_1.bd* and select Create HDL Wrapper... to generate the top-level VHDL model. Leave the “Let Vivado manager wrapper and auto-update” option selected.

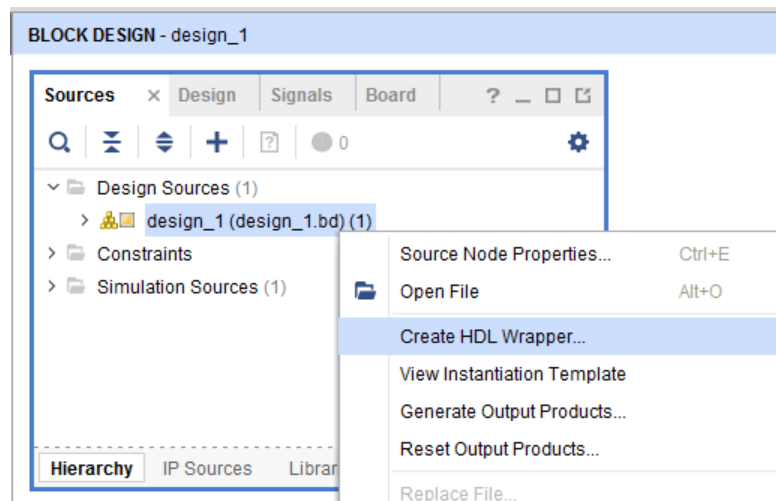


Figure 12. Create HDL Wrapper.

The system_wrapper.vhd file is created, added to the project and set as the top module in the design. (You can double-click on the file name to see the content in the Auxiliary pane).

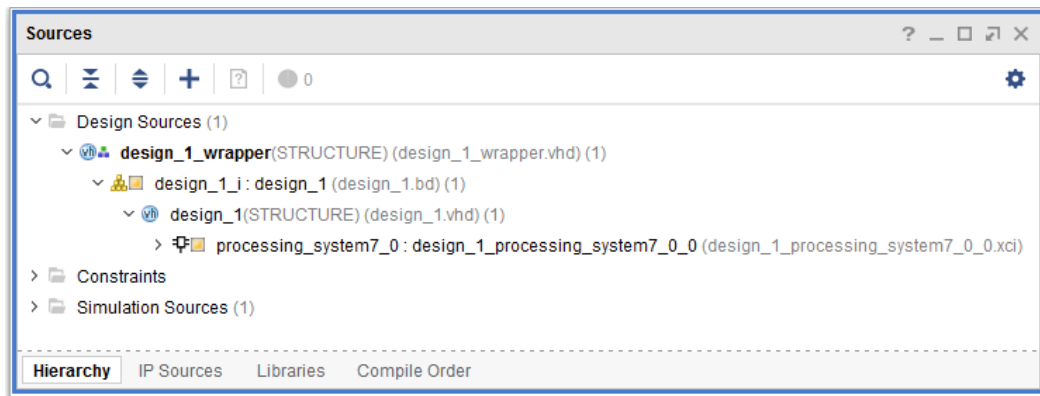


Figure 13. HDL Wrapper file generated and added to the project.

The hardware system has been generated and can now be exported to the SDK.

3. Develop and Test the Software Application

3.1. Start SDK from Vivado

3.1.1. Export hardware

In Vivado, click **File** → **Export** → **Export Hardware** (save the project when prompted).

This design does not have any hardware in Programmable Logic (PL), therefore there is no bitstream to be generated and included.

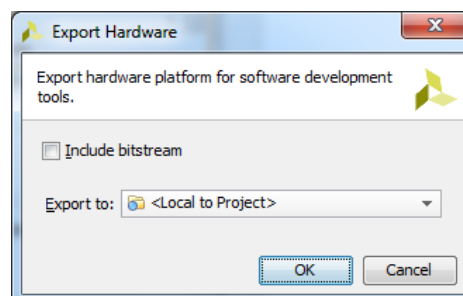


Figure 14. Export hardware.

3.1.2. Launch SDK

Launch SDK by clicking **File** → **Launch SDK**, with the default settings.

SDK will open with a hardware platform project automatically created: a `design_1_wrapper_hw_platform_0` folder will exist in the Project Explorer panel.

In SDK you can see the information about your hardware platform in the **system.hdf** file (hardware description file). This file contains basic information about the hardware configuration of the project, along with the Address maps for the PS systems, and driver information. The .hdf file is used in the software environment to determine the peripherals available in the system, and their location in the address map.

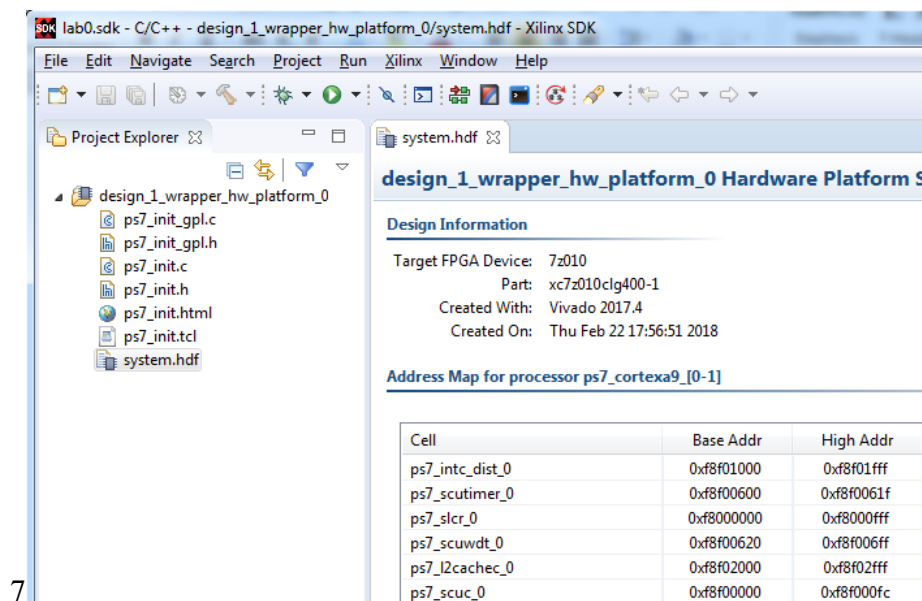


Figure 15. SDK development view.

3.2. Create the Board Support Package

Create a board support package (BSP) for your hardware platform by clicking **File** → **New** → **Board Support Package**. This will create a set of software libraries necessary for the execution of the software in your hardware platform.

Create the BSP with the default settings, namely using the *standalone* software layer, and not including any additional libraries. The standalone BSP contains boot code, cache, exception handling, file and memory management, configuration, time and processor-specific include functions.

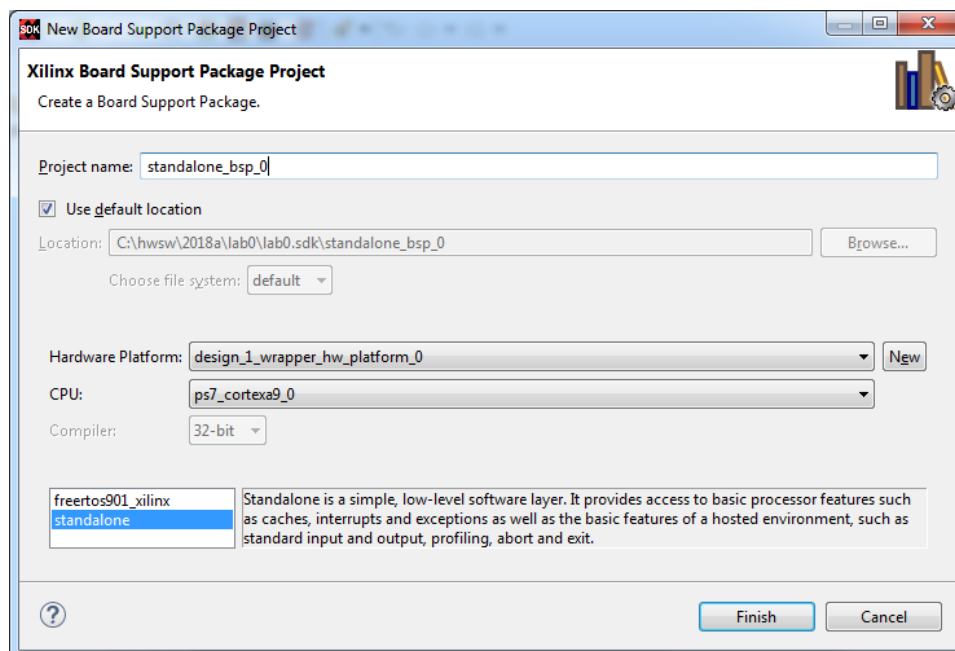
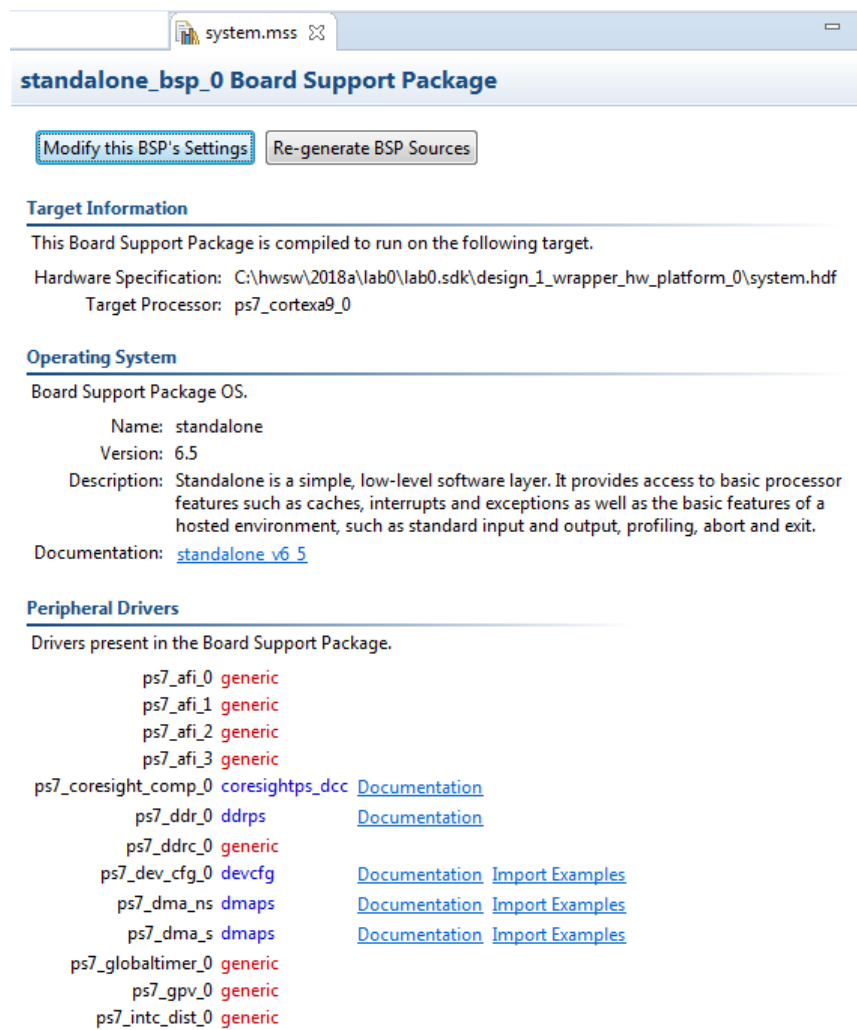


Figure 16. Board Support Package.

The system.mss file gives information about the software drivers and libraries that have been generated. It also includes links for documentation and application examples that you may use in the future.



system.mss

standalone_bsp_0 Board Support Package

[Modify this BSP's Settings](#) [Re-generate BSP Sources](#)

Target Information

This Board Support Package is compiled to run on the following target.

Hardware Specification: C:\hws\2018a\lab0\lab0.sdk\design_1_wrapper_hw_platform_0\system.hdf
 Target Processor: ps7_cortexa9_0

Operating System

Board Support Package OS.

Name: standalone
 Version: 6.5

Description: Standalone is a simple, low-level software layer. It provides access to basic processor features such as caches, interrupts and exceptions as well as the basic features of a hosted environment, such as standard input and output, profiling, abort and exit.

Documentation: [standalone v6.5](#)

Peripheral Drivers

Drivers present in the Board Support Package.

- ps7_afi_0 generic
- ps7_afi_1 generic
- ps7_afi_2 generic
- ps7_afi_3 generic
- ps7_coresight_comp_0 coresightps_dcc [Documentation](#)
- ps7_ddr_0 ddrps [Documentation](#)
- ps7_ddrc_0 generic
- ps7_dev_cfg_0 devcfg [Documentation](#) [Import Examples](#)
- ps7_dma_ns dmaps [Documentation](#) [Import Examples](#)
- ps7_dma_s dmaps [Documentation](#) [Import Examples](#)
- ps7_globaltimer_0 generic
- ps7_gpv_0 generic
- ps7_intc_dist_0 generic

Figure 16. BSP Drivers info.

3.3. Create your C Project

In this example, you will use a previously coded C program “*matprod_v0.c*” that multiplies two 3×3 matrices.

```
#include <stdio.h>

#define MAT_SIZE 3

static int x1[MAT_SIZE][MAT_SIZE] =
    {{-79, -8, 72},
     {-7, -97, 27},
     {-9, 7, 79}};
static int x2[MAT_SIZE][MAT_SIZE] =
    {{-23, 28, -3},
     {-6, 7, -74},
     {-55, -10, 22}};
static int x3[MAT_SIZE][MAT_SIZE];

void print_mat(int *x)
{
    int i, j;

    for (i=0; i<MAT_SIZE; i++) {
        for (j=0; j<MAT_SIZE; j++) {
            printf("%5d ", x[i*MAT_SIZE+j]);
        }
        printf("\n");
    }
    printf("\n");
}

int main()
{
    int i, j, k;

    // print_mat(&(x1[0][0]));
    // print_mat(&(x2[0][0]));

    for (i=0; i<MAT_SIZE; i++) {
        for (j=0; j<MAT_SIZE; j++) {
            x3[i][j] = 0;
            for (k=0; k<MAT_SIZE; k++) {
                x3[i][j] += x1[i][k]*x2[k][j];
            }
        }
    }
    print_mat(&(x3[0][0]));

    return 0;
}
```

Program 1. Matprod_v0.c.

Create a new C project by clicking **File** → **New** → **Application Project**.

Choose a name for the C project (in the example, *matprod_0*) and target the existing *standalone_bsp_0* board support package that you have previously created. Next, select **Empty Application** to create an empty application project.

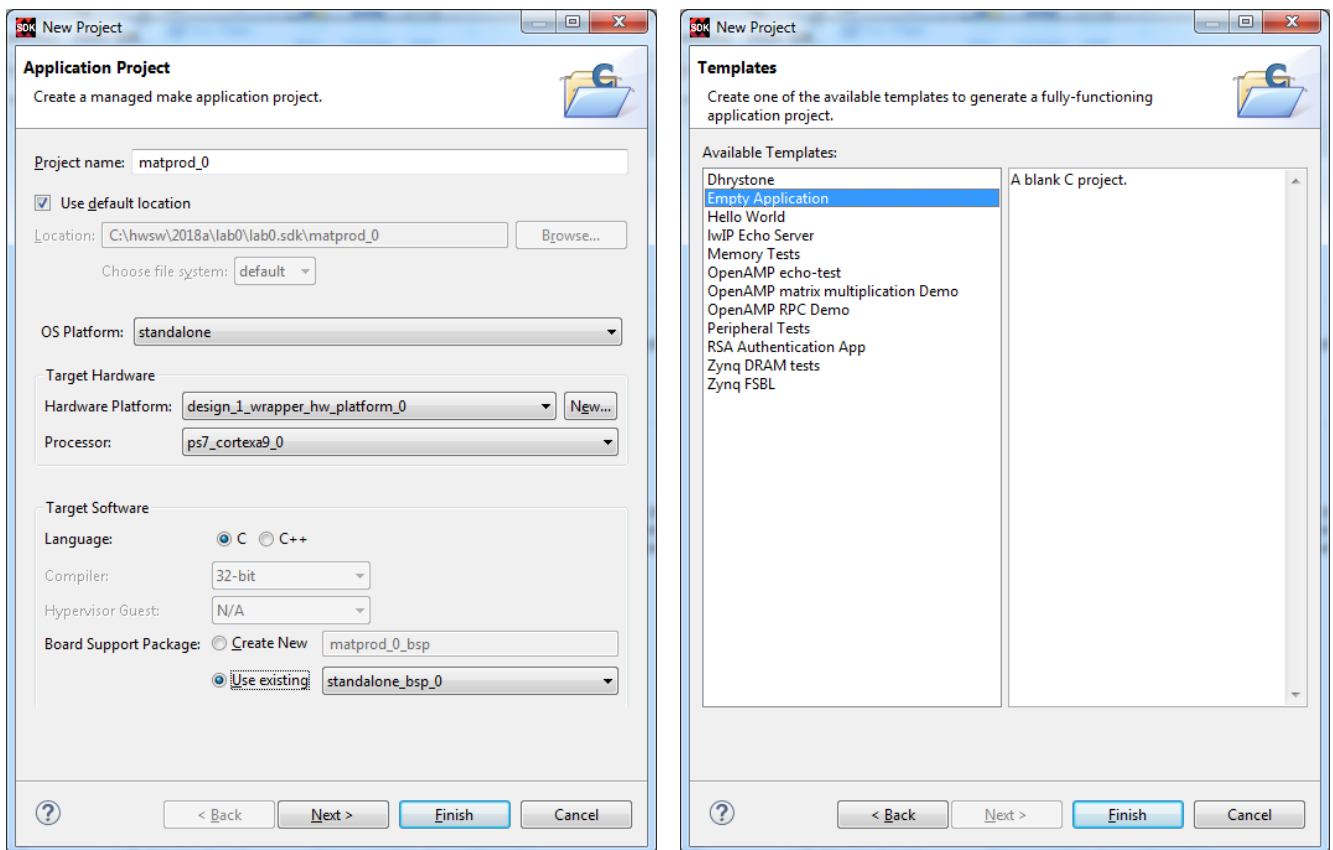


Figure 17. New C Project.

The C project will be created and shown in the SDK Project Explorer window. This window should now look similar to the figure below:

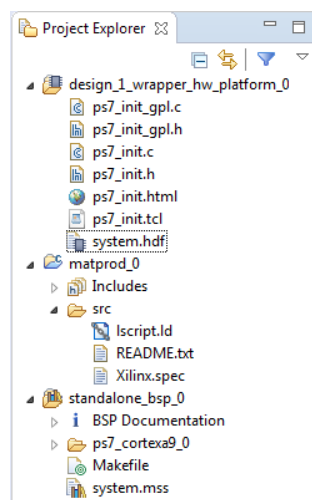


Figure 18. SDK Project Explorer.

Now you can create a new C source file, as in this example, or import an existing file to the folder *matprod_0/src*.

Select the *src* folder under *matprod* in the project view, right-click, and select **New->Source File**.

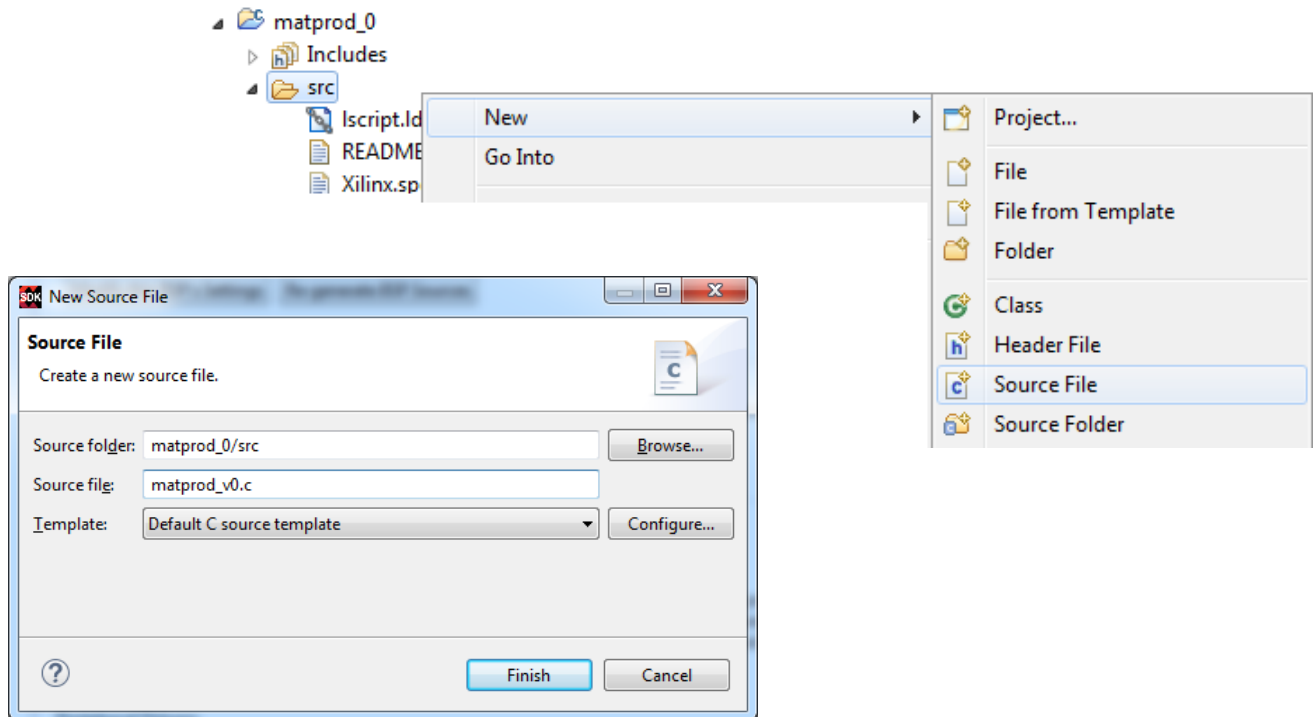


Figure 19. Add new C file.

The Project Explorer view will now also include the new C file, which can be edited in the SDK C Editor.

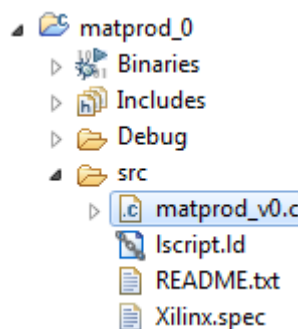


Figure 20. New C file in Project

Fill in the C code for the *matprod_v0.c* function ().

You can set the gcc compiler settings, including the optimization level, by selecting the **C/C++ Build Settings** (right-click on your *matprod_0* project). For now, just leave them as default, namely leave the Optimization Level set to None (-O0).

Select *matprod_0*, right-click and select **Generate Linker Script**. Here you can define where the data and instruction sections of your code will be stored. Place all the sections of your program on the OCM and leave the heap and stack sizes as 1K bytes each.

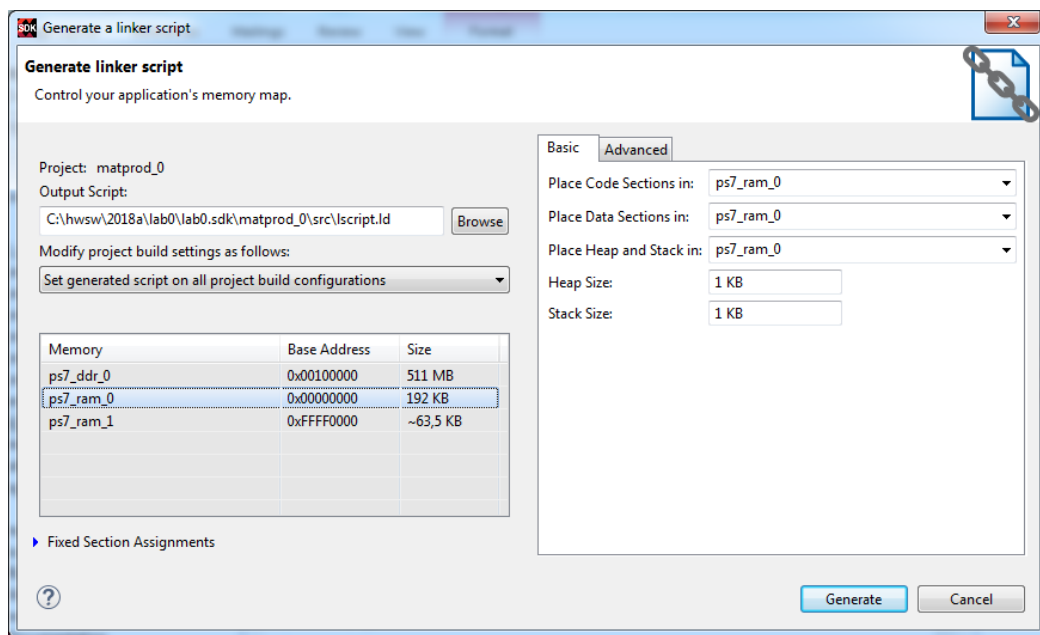


Figure 21. Linker Script.

The linker combines the compiled applications, including libraries and drivers, and produces an executable file, in Executable Linked Format (ELF), that is ready to run on your processor hardware platform.

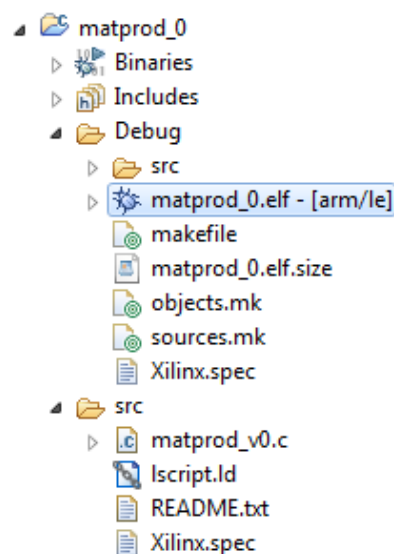


Figure 22. ELF file in Project

Opening the *matprod_0.elf*, you can see the instructions generated for your C code, including their instruction memory addresses. At the top, you can view a list of the sizes and starting addresses of the various sections of the program, where *size* indicates the size of each section and *LMA* is the Loadable Memory Address (start address for each section). Note that, in this case, all sections reside in the OCM memory space (starting with base address 0x00000000).

4. Verify the Design in Hardware

Connect and power up the Zybo Board. You will use 2 cables, one to connect to the JTAG port (to program the FPGA) and the other for fast serial connection through the uart-lite port.

4.1. Program the FPGA

Whenever necessary, the FPGA is programmed by selecting *Xilinx Tools* → *Program FPGA*.

As this design does not have (yet) any hardware in Programmable Logic (PL), there is no bitstream to be programmed and therefore this step is to be skipped.

4.2. Define a configuration to execute your software

The simplest way to debug your system is to define a configuration that associates the *stdout* of your C application to the SDK console.

Define a configuration to configure the execution of your software in the development board by selecting project `matprod_0` (in SDK Project Explorer) and then executing **Run** → **Run Configurations**. In the configurations window, press the New button to create a new C/C++ application (GDB) configuration.

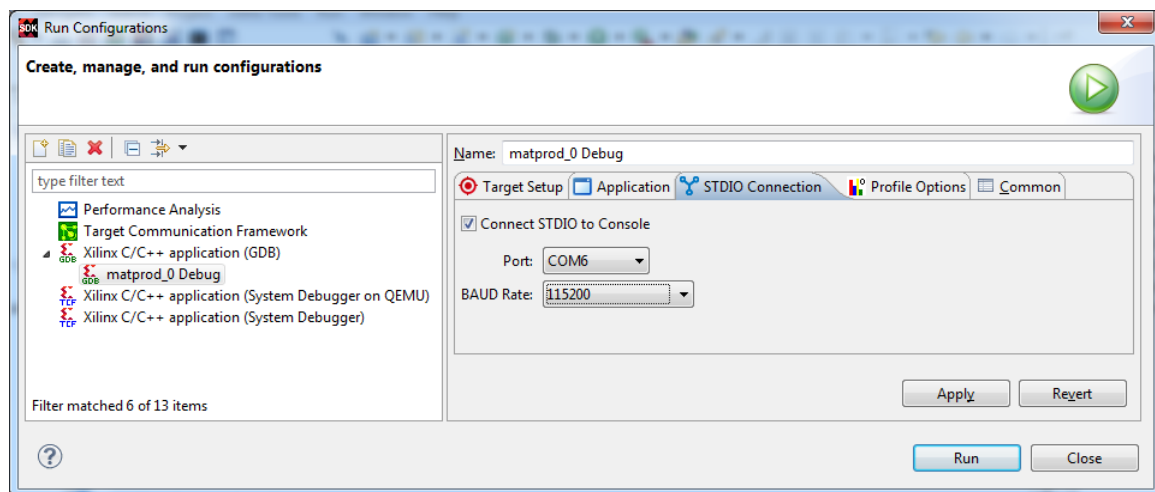


Figure 25. Run Configurations window.

In the STDIO Connection tab, associate the *stdio* to the SDK console through the COM port of your PC to where the uart-lite is connected. Make sure to set the BAUD rate to the same value you defined for the UART in your hardware platform (which by default is 115200).

After clicking **Run**, the application will execute and you will see the *stdout* output in the SDK console.

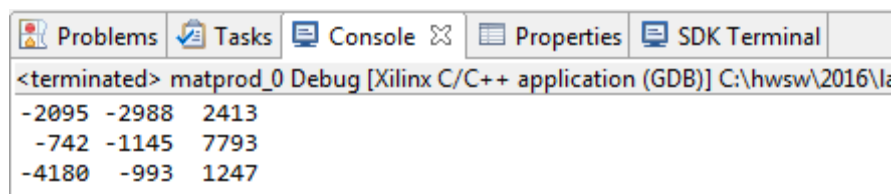


Figure 26. Console output of the 3×3 matrix product.

You can rerun the same configuration just by clicking on the configuration button:

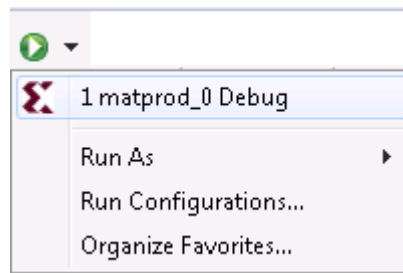


Figure 27. Rerun configuration.

5. Run Application using Memory Initialization from file

You can debug your applications by initializing part of the memory of your system using data from a binary file in your computer. You can also redirect the *stdout* of your program to a text file in your computer.

This basic procedure will be used, in your future projects, to demonstrate the correctness of your designs. In this introduction, you will test a new version of the matrix multiplication application.

5.1. Matprod1 application

The C program in the file *matprod_v1.c* multiplies 2 matrices, previously initialized in a memory zone indicated by the user, and stores the result matrix in an adjacent zone of the same memory. (Note that the matrices size and the matrices base address are not defined in the source provided, therefore syntax error warnings will be issued when you first import the source).

Start a new C project with the *matprod_v1.c* program, using the same BSP. Define the correct size of your matrices, in your C program.

Choose a free memory zone of your system to store the 3 matrices. Be careful that this zone must have **the size adequate** for the required storage and **must not interfere** with the memory zones used for other storage, namely for the program(s) code and data. Remember that **all the memory management is done by you**.

Define the matrices order

```
#define MAT_SIZE ??
```

and set the base address for the matrices storage in your C program

```
#define MATA_START_ADD 0x????????
```

Define the application configuration appropriately (see sections 4.2, 5.2 and 5.3).

5.2. Memory Initialization from a Binary File

In the Run Configuration window go to the **Application** tab. Select a binary file with the contents you want to initialize the memory and define the base address where data will be stored (as shown in figure 28).

You must be careful to select the memory address zone such that it corresponds to **a valid data memory zone** in your system and that it is **free to store your data**. In the figure example, the base address selected defines a memory area to be used in the external memory (DDR).

Note: you can use (in your PC) the *mem_init* program (included in the support material) to generate a binary file with two matrices (of the given size) with random integer elements.

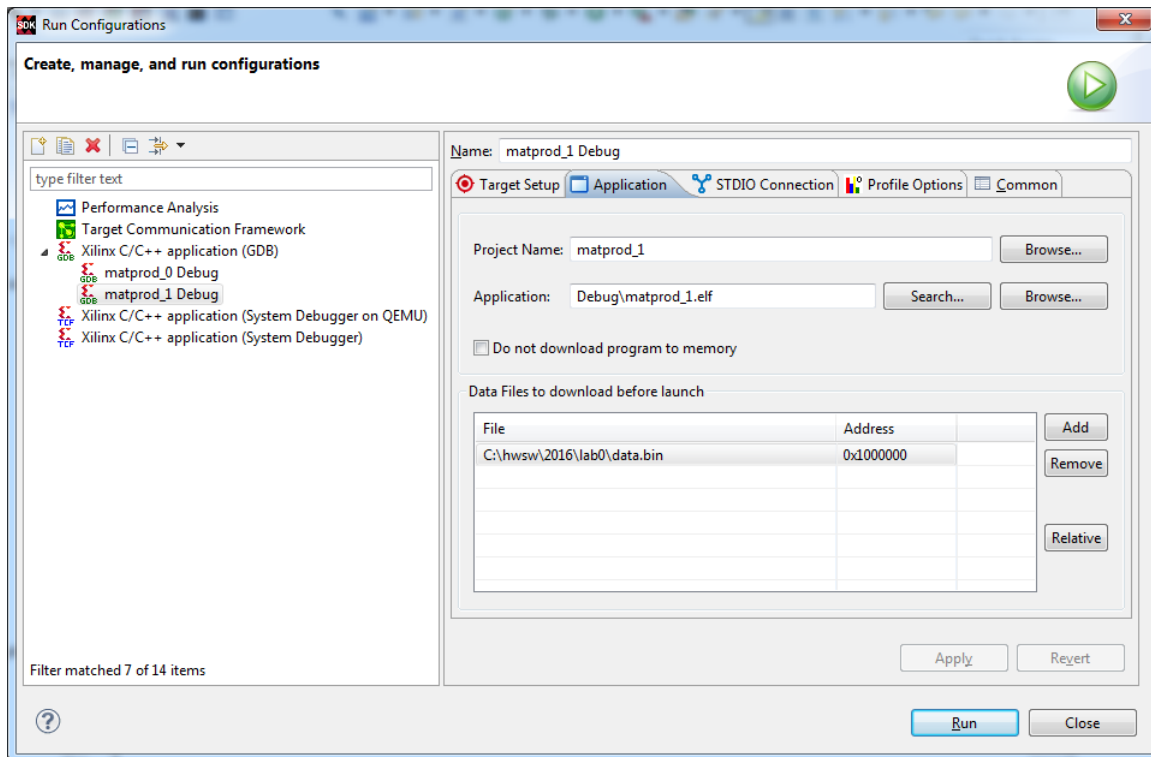


Figure 28. Memory Initialization from file

5.3. Redirect Stdout to File

In the Run Configurations window go to **Common** tab. Below “Standard Input and Output” check the box that says File and specify a text file to redirect the output to (as represented in figure 29).

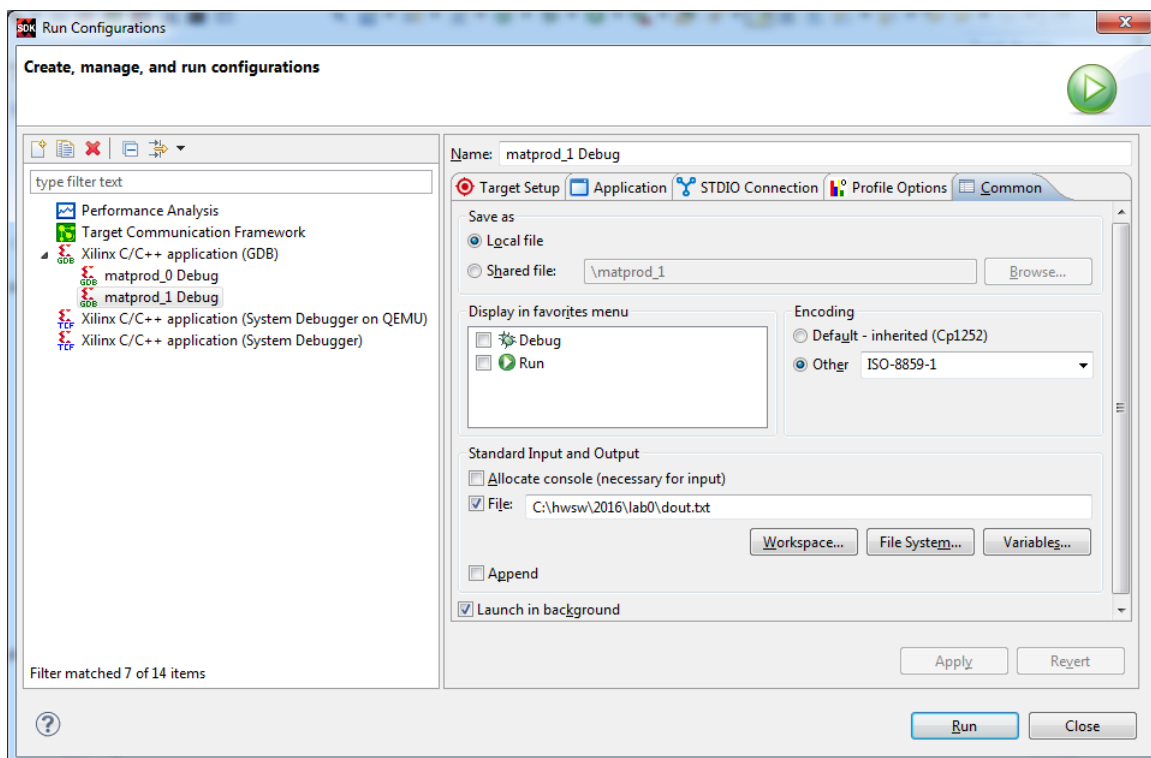


Figure 29. Redirect Output to File

5.4. Measure the execution time of your application

You can use the time-specific function *XTime_GetTime* (*XTime *xtime*) (available in the standalone BSP) to evaluate the execution time of your performance. This function provides direct access to the 64-bit Global Counter in the PMU. This global timer (GT) is a 64-bit incrementing counter with an auto-incrementing feature and is accessible to both Cortex-A9 processors. The global timer is always clocked at 1/2 of the CPU frequency (it increases by one at every 2 processor cycles). The BSP-defined value `COUNTS_PER_SECOND` directly indicates the GT clock frequency (number of counts per second).

```
#include <stdio.h>
#include "xtime_1.h"

int main()
{
    XTime tStart, tEnd;
    // initialize the application
    XTime_GetTime(&tStart);    // start measuring time
    // process the application
    XTime_GetTime(&tEnd);      // end measuring time
    // finalize the application
    printf("Output took %llu clock cycles.\n", 2*(tEnd - tStart));
    printf("Output took %.2f us.\n",
           1.0 * (tEnd - tStart) / (COUNTS_PER_SECOND/1000000));
    return 0;
}
```

Figure 30. XTime_GetTime() Usage Example

5.4.1. Measure the execution time

Measure the execution time of the matrix multiplication application for matrices of size 50×50, by including the example C code in your application.

5.4.2. Check the program memory size requirements

Note that although the SDK libraries fully support the standard C functions for I/O, such as `printf`, these functions are large and might not be suitable for embedded processors with smaller memories.

The *xil_printf*() (`#define xil_printf.h`) function is a light-weight implementation of `printf`(). It is much smaller in size (only 1 kB), but does not have support for floating point numbers and also does not support printing of long (such as 64-bit) numbers.