

Vivado/Vitis Zynq Quick Start Tutorial for the Digilent Zybo Z7-10 Zynq 7000 ARM/FPGA SoC Platform board – using the Zynq microprocessor

Objectives:

After completing this tutorial, you will be able to:

- Create an embedded system using block design tools
- Customize the Zynq processor
- Use Zynq processor I/O interfaces
- Connect peripherals in the programmable logic part to the Zynq processor
- Use Xilinx Vitis Unified Software Platform to develop software applications for Zynq
- Use the Zynq UART to communicate with the PC
- Configure the FPGA and run software applications on the Zybo Z7-10 board FPGA

This tutorial provides a step-by-step introduction to the operation of the Zynq microprocessor using the Vivado IP Integrator with the Xilinx Vitis Unified Software Platform for the Xilinx Zybo Z7-10 Zynq-7000 ARM/FPGA SoC Platform from Digilent.

The Zybo Z7-10 is equipped with a Zynq Z-7010 (Part Number XC7Z010-1CLG400C), which includes a hard Dual-Core ARM Cortex-A9 MPCore. In this tutorial, we create an application for this processor using the Vivado IP Integrator tool. The aim is to reproduce the functionality of the *Vivado Quick Start Tutorial for the Digilent Zybo Z7-10 Zynq 7000 ARM/FPGA SoC Platform board – a hardware approach* up/down counter but using now a hardware/software co-design approach. But since we are now using a processor, the first application we are going to create is the famous “Hello World”!

The Zynq 7000 ARM/FPGA SoC Platform is divided into two distinct subsystems: the Processing System (PS) and the Programmable Logic (PL). Figure 1 shows an overview of the Zynq architecture, with the PS colored light green and the PL in yellow (adapted from Xilinx).

The PL is nearly identical to a Xilinx 7-series Artix FPGA, except that it contains several dedicated ports and buses that tightly couple it to the PS. It can be configured either directly by the processors or via the JTAG port.

The PS consists of several components, including the Application Processing Unit (APU), which comprises two Cortex-A9 processors, Advanced Microcontroller Bus Architecture (AMBA) Interconnect, DDR3 Memory controller, and various peripheral controllers with their inputs and

outputs multiplexed to 54 dedicated pins (called Multiplexed I/O, or MIO pins). Peripheral controllers that do not have their inputs and outputs connected to MIO pins can instead route their I/O through the PL, via the Extended-MIO (EMIO) interface. The peripheral controllers are connected to the processors as slaves via the AMBA interconnect and contain readable/writable control registers that are addressable in the processors' memory space. The programmable logic is also connected to the interconnect as a slave, and designs can implement multiple cores in the FPGA fabric, with each also containing addressable control registers. Furthermore, cores implemented in the PL can trigger interrupts to the processors and perform Direct Memory Access (DMA) accesses to DDR3 (Double Data Rate 3 Synchronous Dynamic Random-Access Memory (SDRAM)) memory.

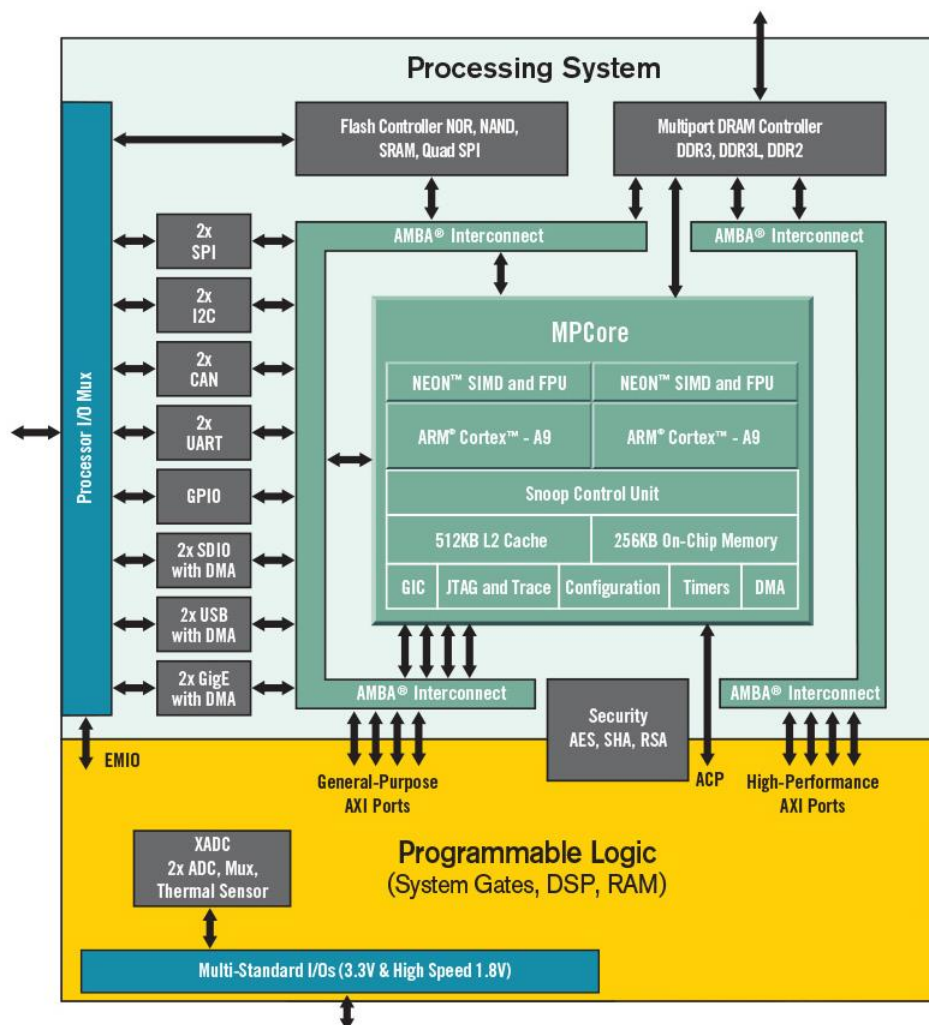


Figure 1

There are many aspects of the Zynq architecture that are beyond the scope of this document. For a complete and thorough description, refer to the Zynq-7000 SoC Technical Reference Manual (UG585) available at the Xilinx website and the Zynq book available at www.zynqbook.com.

1. Starting the Vivado Software

To start Vivado, double-click the desktop icon,



or start Vivado from the Start menu by selecting:

Start → All Programs → Xilinx Design Tools → Vivado20xx.x

Note: Your start-up path is set during the installation process and may differ from the one above.

2. Accessing Help

At any time during the tutorial, you can access online help for additional information about the Vivado tools by selecting **Help → Documentation and Tutorials....**

3. Create a New Project

Create a new Vivado project targeting the FPGA device on the Zybo Z7-10 board from Digilent.

To create a new project:

1. In the **Quick Start** toolbar click **Create New Project** or select **File → Project → New...** in the Vivado menu bar; the **New Project** wizard appears (Figure 2)

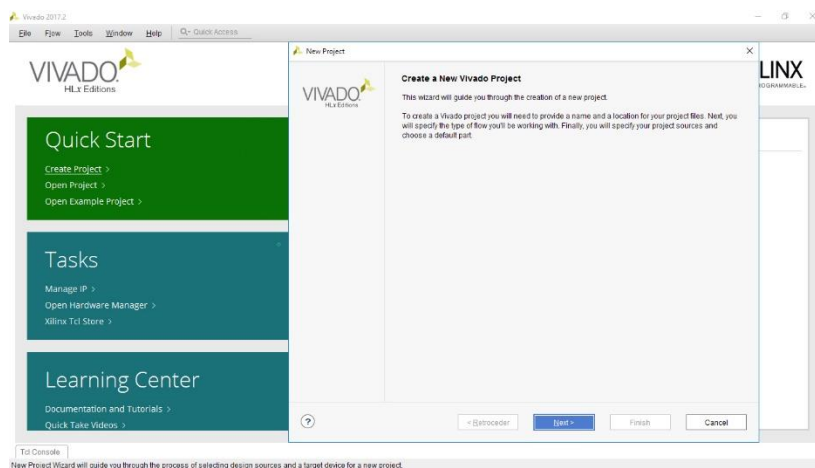


Figure 2

2. Click **Next**
3. Type **tutorial_micro** in the **Project Name** field
4. Enter or browse to a location (directory path) for the new project. A tutorial subdirectory is created if the **Create project subdirectory** checkbox is checked

WARNING: when using any of the Xilinx applications keep your directory path as close as possible to the root, and directory and file names as short as possible! Xilinx applications do not deal well with long directory paths and long directory or file names. Long paths or names

usually lead to hard to detect errors during development. AND IT DOES NOT TOLERATE WHITE SPACES!!!

5. Click **Next**
6. In the **Project Type** window select **RTL Project** and check the **Do not specify sources at this time** checkbox (Figure 3)

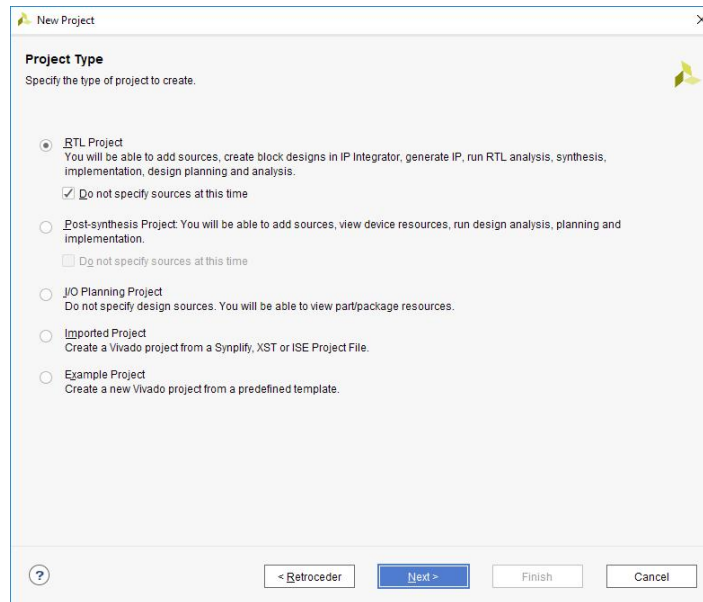


Figure 3

7. In the next window select **Boards**, choose **Vendor: digilentinc.com** and select **Zybo Z7-10** from the list

Notice: if the Digilent vendor does not appear in the list, check section 3 of the [Vivado Quick Start Tutorial for the Digilent Zybo Z7-10 Zynq 7000 ARM/FPGA SoC Platform board – a hardware approach](#).

8. Click **Next** to view the **New Project Summary**
9. Check if everything is correct (compare with Figure 4) and then click **Finish**

The **Project Manager**, where we manage all our project, opens.

4. Create a New Block Design

In this section, we learn how to create an IP block design source.

In electronic design, an IP core or IP block (semiconductor Intellectual Property core) is a reusable block of logic or data that is the intellectual property of one party. IP cores may be licensed for free or under a fee to another party or can be owned and used by a single party alone. The term is derived from the licensing of the patent and/or source code copyright that exist in the design. IP cores can be used as building blocks within ASIC or FPGA logic designs.

Vivado provides an IP-Centric design flow based on the IP Integrator tool that enables us to add IP modules to our projects from various design sources.

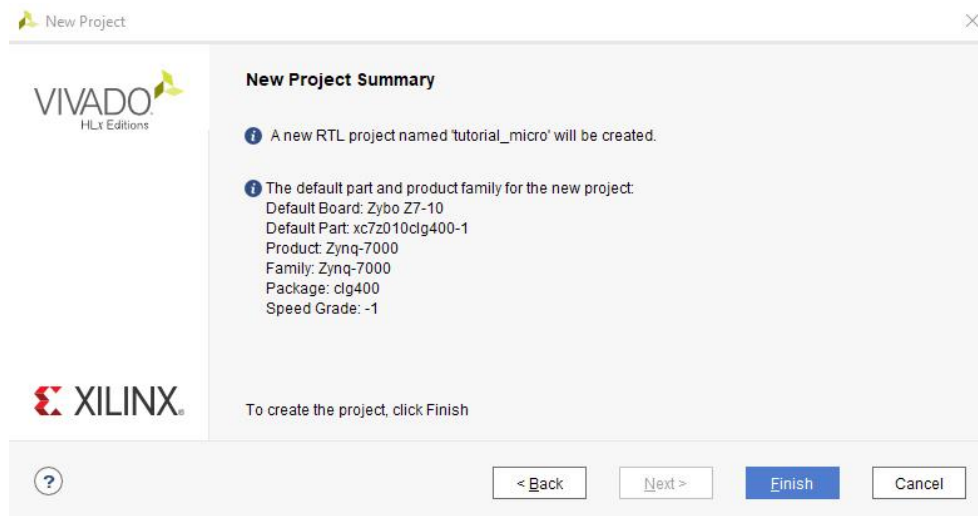


Figure 4

Create a new block design source for the project using the IP Integrator as follows:

1. In the **Flow Navigator** window, on the left side of the **Project Manager** main window, in the **IP Integrator** section, click **Create Block Design**
2. In the new **Create Block Design** window (Figure 5):
 - Design name: **tutorial_design**
 - Directory: **<Local to Project>**
 - Specify source set: **Design Sources**

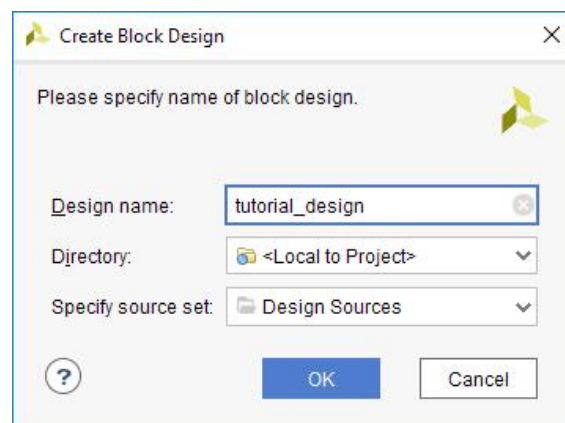


Figure 5

3. Click **OK**
An empty design workspace is created where we can add IP blocks (Figure 6).
4. To view and add an IP core click on the **+** icon in the middle of the right side **Diagram** window to open a catalog of pre-built IP blocks from Xilinx IP repository (Figure 7)

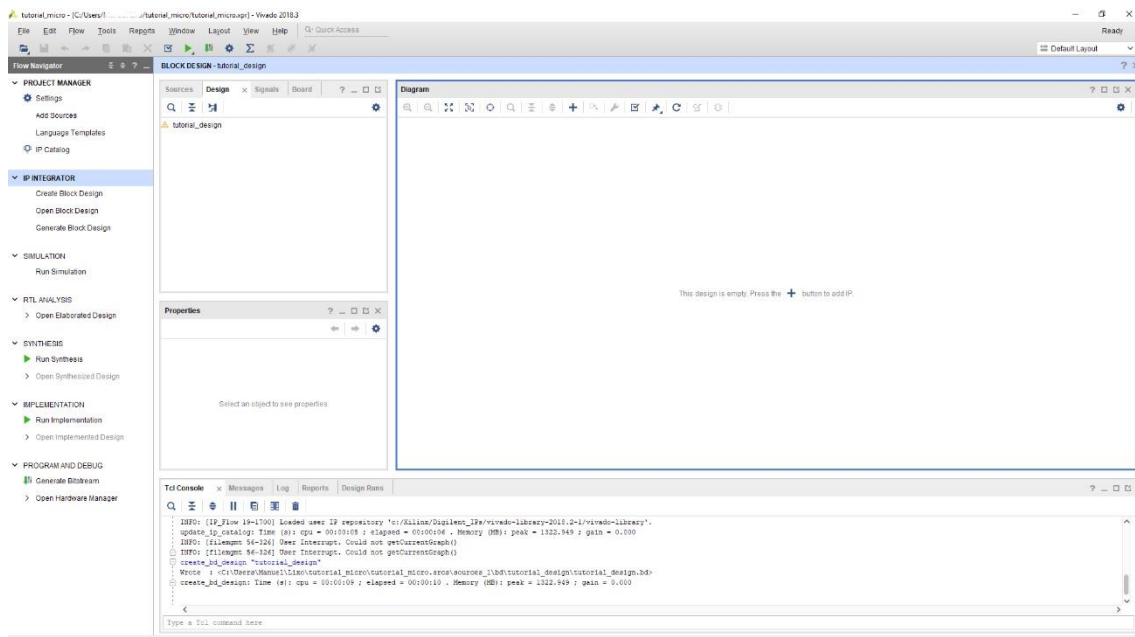


Figure 6

5. Adding and Customizing the ZYNQ7 Processing System

The first step is to add the **ZYNQ7 Processing System** to our design. To do this we have two options: to click on the **+** icon in the middle of the right-side **Diagram** window and look for the **ZYNQ7 Processing System** in the Xilinx IP repository (Figure 7); or, to open the **IP Catalog** in the **Flow Navigator** window. We are going to follow this latter alternative.

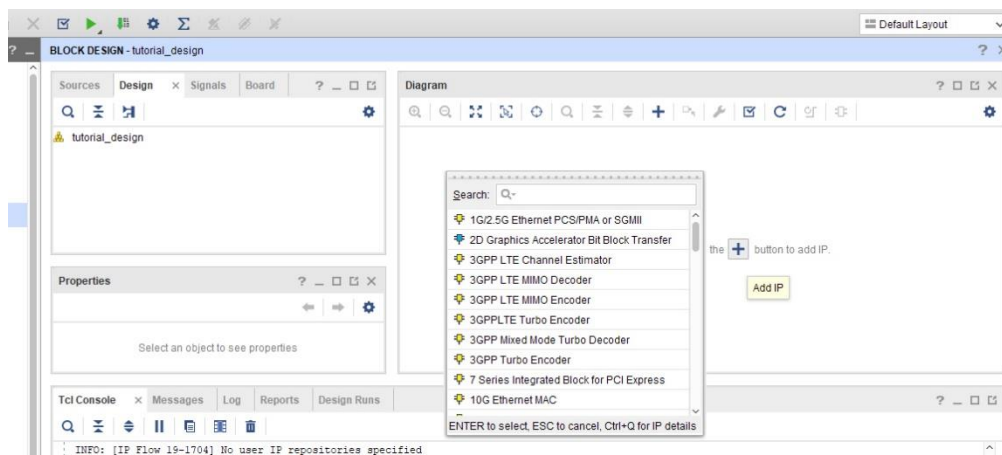


Figure 7

1. In the **Flow Navigator** window, on the left side of the main window, in the **Project Manager** section, click **IP Catalog** to view the list of available IP cores
2. The **IP Catalog** opens on the **Block Design** main window
3. Under **Vivado Repository** → **Embedded Processing** → **Processor**, double click **ZYNQ7 Processing System** to add it to our design (Figure 8)

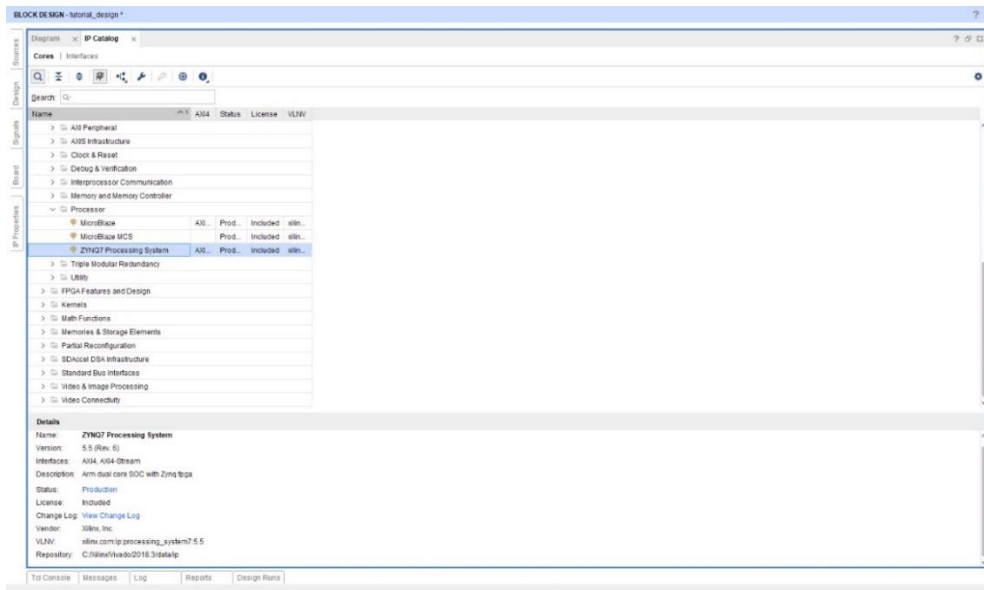


Figure 8

4. In the new **Add IP** floating window select **Add IP to Block Design**
5. Click on **Run Block Automation** at the top of the **Diagram** window and a customization assistant window opens with a set of default settings
6. In the new **Run Block Automation** window make sure that **processing_system7_0** under **All Automation** in the panel to the left and **Apply Board Preset** in the panel to the right are checked. These options set the board preset on the Processing System. ZYNQ7 block automation applies current board presets and generates external connections for FIXED_IO, Trigger and DDR interfaces (Figure 9); the preset values are part of the Zybo Z7-10 board file

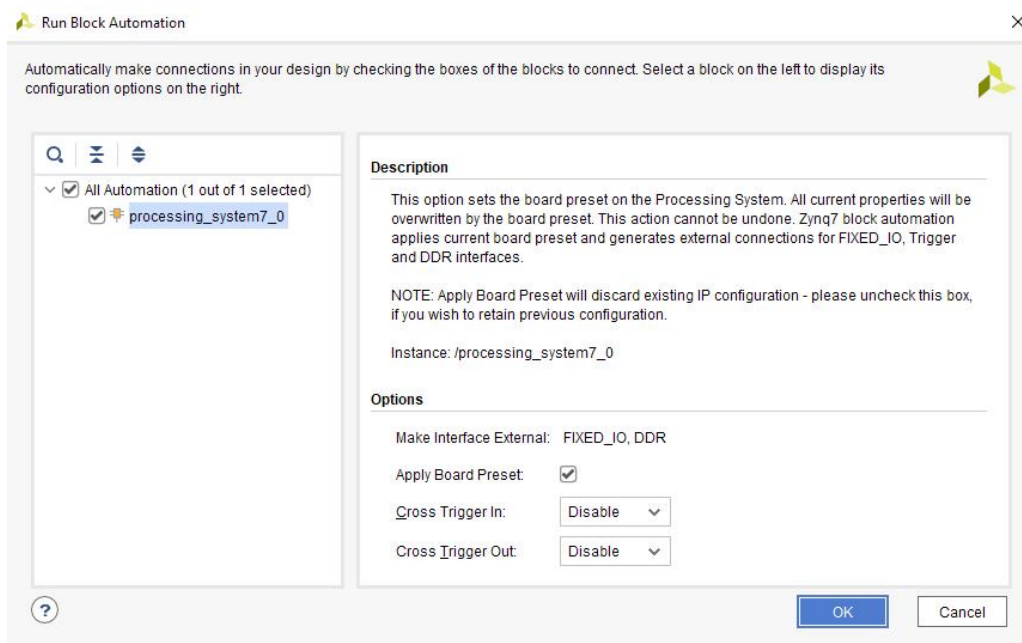


Figure 9

7. Click **OK**

The ZYNQ7 Processing System is a complex, highly customizable, microprocessor. To customize the block properties of any added IP, double click on the block itself. In the case of the ZYNQ7 Processing System, a **ZYNQ7 Processing System Re-customize IP** window (Figure 10) opens giving access to an extended range of features that can be customized depending on our specific application. To facilitate the configuration, a series of predefined configurations are available. Each predefined configuration completely changes the ZYNQ7 Processing System parameters. Access to the full documentation pack is also available on the top left side of the window.

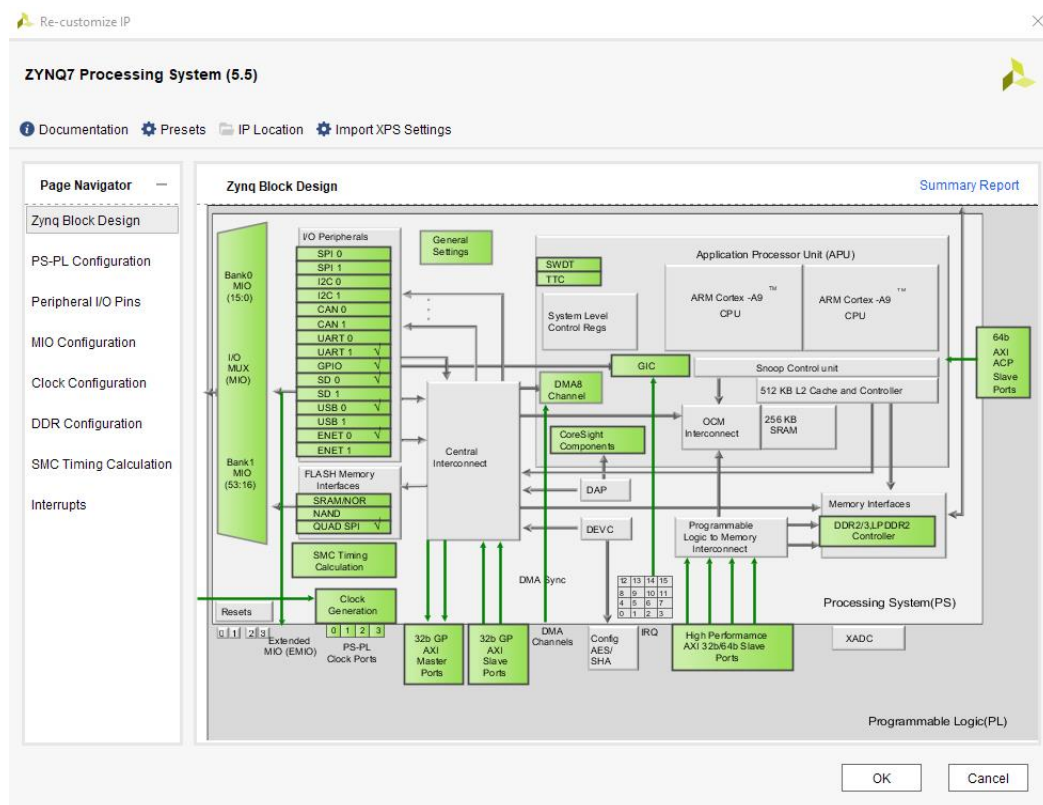


Figure 10

The **Zynq Block Design** shows the design configuration for the Zybo Z7-10 board. Notice that, for example, the I/O Peripherals checked in the diagram correspond to those available on the board. Click on one of the green I/O Peripherals rectangles to open the **MIO Configuration** window. In there, we may see the MIO pins to which each one of the peripherals is connected. If we click on the **PS-PL Configuration** on the left menu and open **General**, we may change the default **Baud Rate** of **UART1** (the processor UART connected in our ZYBO Z7-10 board), which we will use for the communication between the PC and the processor when running the famous “Hello world” (Figure 11). For now, let us keep it with its default value of 115200 bit/s.

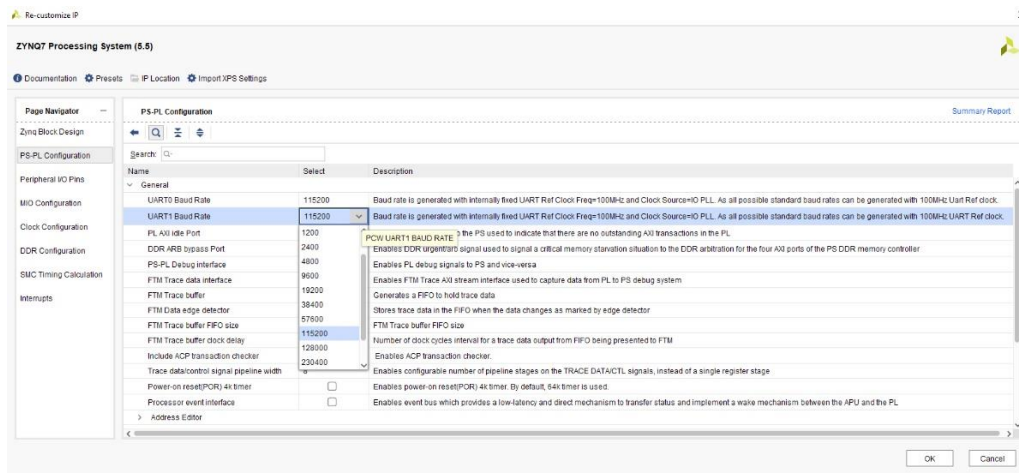


Figure 11

In the top right side of the **ZYNQ7 Processing System Re-customize IP** window, a link called **Summary Report** enables access to the **Zynq PS Register Summary Viewer**, a **Zynq PS Configuration Report** showing the current Zynq configuration and details of all processor registers.

Since we are not going to change any of the presets defined in the board file uploaded from Digilent, click **OK** to close the window.

6. Adding and Customizing a GPIO block

The ZYBO Z7-10 board file we chose in section 3 contains a description of the board's various peripherals, which are already configured to work with several Vivado IPs. The list of these components is available in the **Board** window located on the left-hand side of the **Block Design** window.

To be able to read the board switches and to light up the board LEDs, we need to add a GPIO (General-Purpose Input/Output) IP block to our design. This peripheral communicates with the processor through an Advanced eXtensible Interface (AXI) bus, part of the ARM Advanced Microcontroller Bus Architecture (AMBA), an open standard originally developed by ARM for use in microcontrollers, which had its first version in 1996. AMBA is an on-chip interconnect specification used to connect and manage the communications between the processor(s) and other IP blocks in an embedded system. It facilitates the development of multi-processor designs with large numbers of controllers and peripherals with a bus architecture.

The AMBA standard has been revised and extended, with Xilinx strongly contributing to defining AXI4. To each AXI peripheral implemented into the FPGA, it is assigned an area of the ZYNQ7's memory space that is used to address each of its control registers. Adding a simple AXI interface to design requires the addition of two new blocks, the AXI Interconnect, which defines how the

AXI signals are routed, and the reset controller, which generates the resets of each AXI block and the interface. Much of the basic block design consists of connecting different AXI peripherals to a processor, enabling it to read from and write to peripherals' input and output ports.

To implement the up/down counter we need to read the external switches, to determine if the counter is enabled or not and the counting direction, and to write to the LEDs.

The GPIO block has 32 I/O ports divided into two channels. To the first channel, we connect the 4 LEDs, and to the second channel, we connect the 4 switches (Note: the order here is important only because the software we will develop later will access channel 1 to write to the LEDs and channel 2 to read from the switches; otherwise, the order is irrelevant).

1. Go to the **Board** tab located on the left-hand side of the **Block Design** window and select **4 LEDs** under the **GPIO** section
2. Drag and drop it into the **Diagram** window (Figure 12)

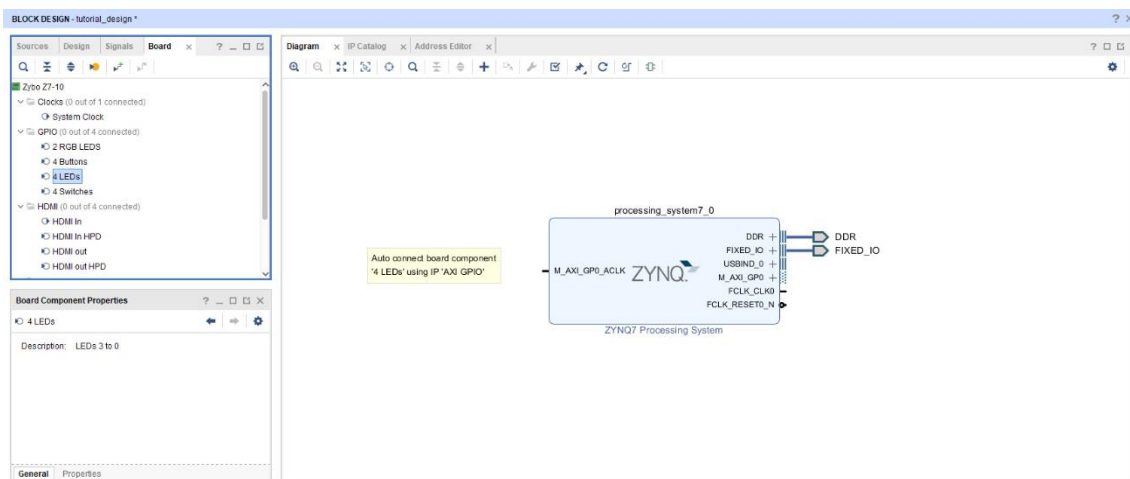


Figure 12

3. In the new **Auto Connect** window click **OK**
Notice that an `axi_gpio_0` block with AXI interface is automatically added to the design, while a 4-bit output bus connects it to the board **LEDs** (Figure 13).

If, when you try to drag and drop the **4 LEDs** nothing seems to happen, or a message appears stating that no GPIO block exists to connect it, you need to add a GPIO block first.

1. In the **Flow Navigator** window, on the left side of the main window, in the **Project Manager** section, click **IP Catalog** to view the list of available IP cores
2. The **IP Catalog** opens on the **Block Design** main window
3. Under **Vivado Repository** → **Embedded Processing** → **AXI Peripheral** → **Low Speed Peripheral**, double click **AXI GPIO** to add it to our design

4. In the new **Add IP** floating window select **Add IP to Block Design**
5. Go to the **Board** tab located on the left-hand side of the **Block Design** window and select **4 LEDs** under the **GPIO** section
6. Drag and drop it into the **Diagram** window

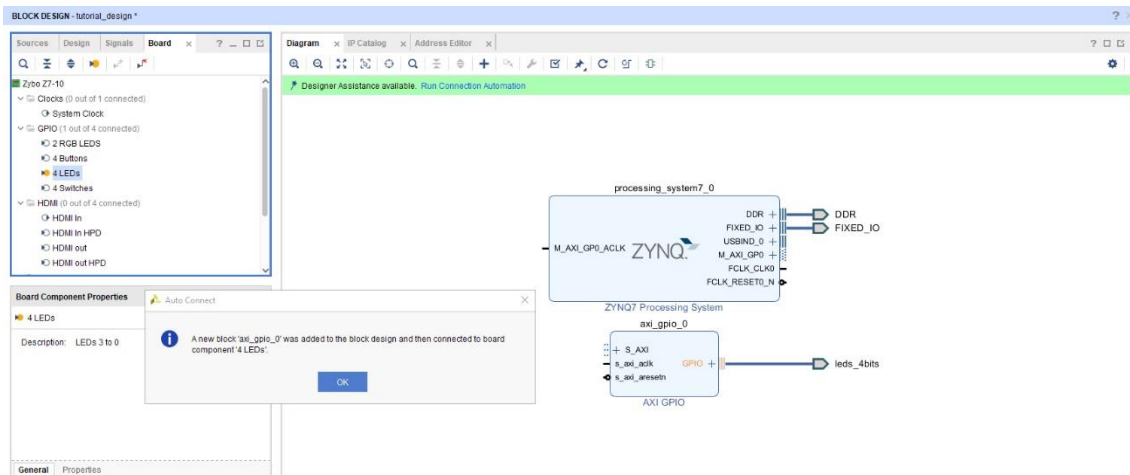


Figure 13

4. Go again to the **Board** tab located on the left-hand side of the **Block Design** window and select **4 Switches** under the **GPIO** section
 5. Drag and drop it into the **Diagram** window
 6. In the new **Auto Connect** window click **OK**
- A new 4-bit input bus is automatically added to the second channel of the same **axi_gpio_0** block to connect the **4 Switches** (Figure 14).

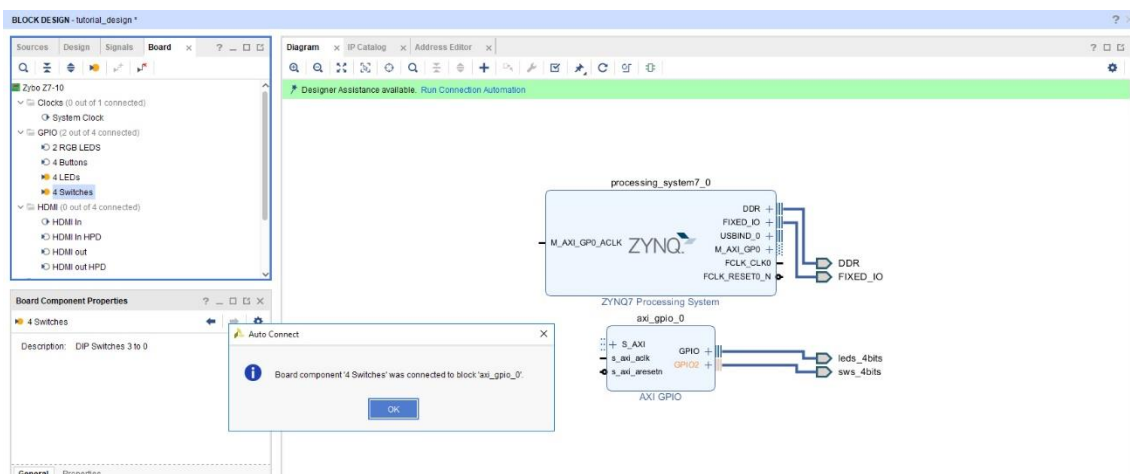


Figure 14

7. Connecting the blocks and validating the Design

Now that all components we need to create our system are added to our design, they need to be properly interconnected.

1. Click on the **Run Connection Automation** message prompt, above our **Diagram** window
2. In the **Run Connection Automation** window, check **All Automation** (Figure 15)
3. Let us keep all options in **Auto** and click **OK**

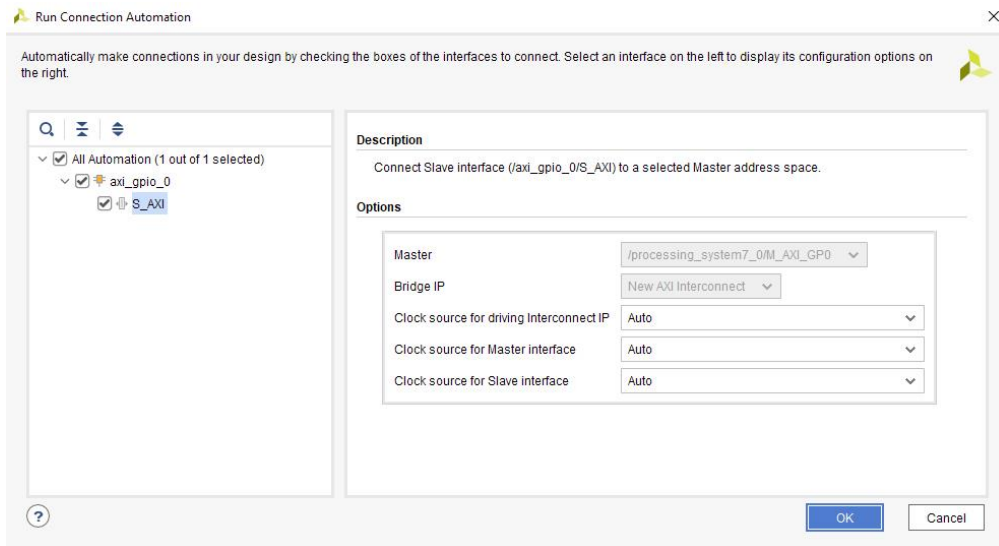


Figure 15

Since we added a block with an AXI interface, Vivado automatically adds an **AXI Interconnect** block and a **Processor System Reset** block, and connects all the blocks in our system. We need now to validate the design, checking for connection errors.

4. Start by regenerating the layout selecting the **Regenerate Layout** icon in the horizontal toolbar of the **Diagram** window (Figure 16) to rearrange our design

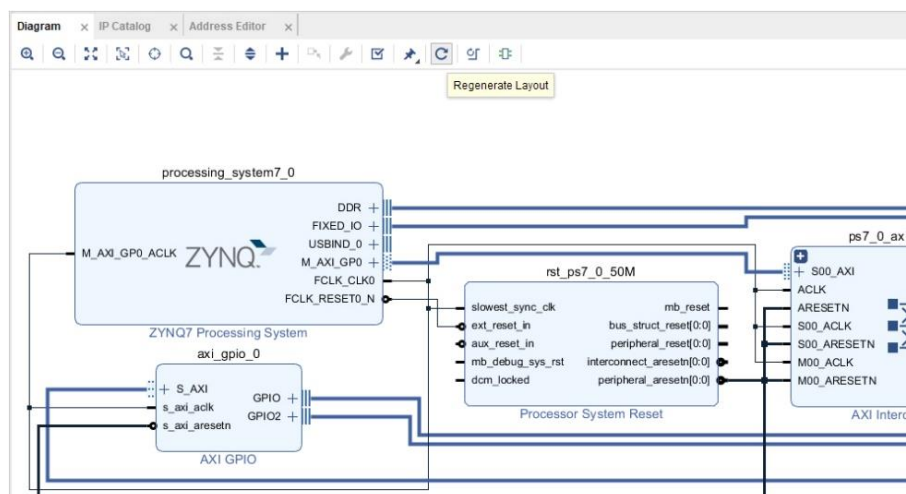


Figure 16

- Then validate the design by selecting in the menu bar **Tools → Validate Design** or selecting the **Validate Design** icon in the horizontal toolbar of the **Diagram** window (Figure 17)

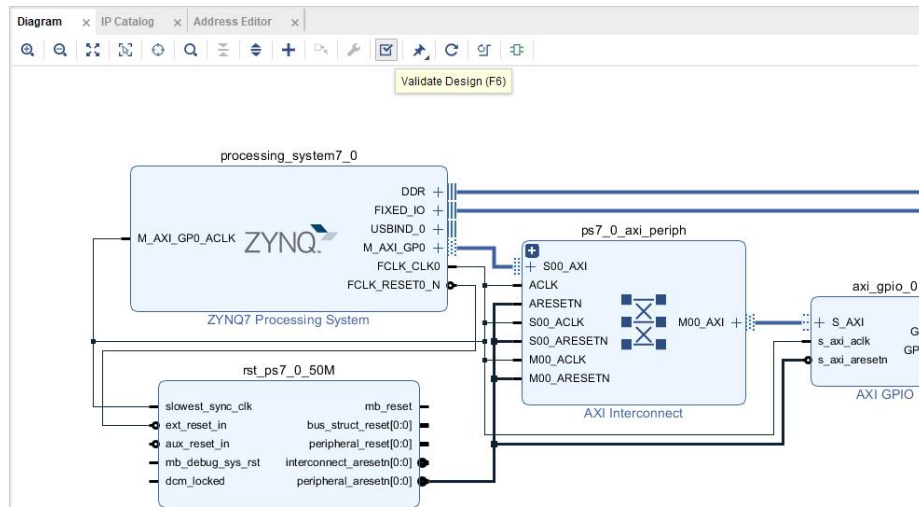


Figure 17

- A new **Validate Design** window appears stating that the validation was successful
- Click **OK**

If, instead, a **Critical Messages** window opens stating that there may be problems with the PS DDR interface, just ignore it by clicking **OK**. You may rerun **Validate Design** again or just ignore and continue.

Like with any other peripheral, the access from the processor to **Pmod connectors JA, JC, JD, and JE** passes through the AXI bus. If we have a **Pmod module** from Digilent connected to one of the **Pmod connectors**, we need an IP core and respective drives to be able to access it from the processor. But even if we do not have any **Pmod module** connected to a **Pmod connector**, we still need to install an IP to gain access to the connector IOs through the AXI bus, in case we want to read/write from/to an external circuit/sensor.

Digilent offers in a GitHub **vivado-library** repository a large number of IP cores intended for use with Digilent boards, including all of Digilent's **Pmod IP cores** and Pmod interface descriptions.

The address of the repository is

<https://github.com/Digilent/vivado-library/releases>

Download the latest release of Digilent's **vivado-library** repository – **vivado-library-
<version>.zip** file. Then choose a location and extract this archive.

In the **Flow Navigator** window, on the left side of the **Block Design** main window, in the **Project Manager** section, go to **Settings**. In the new **Settings** window click **IP Defaults** under **Tool Settings** in the vertical toolbar.

Go to **IP Catalog** → **Default IP Repository Search Path** and click on **+** to add a new path. In the new window **Default IP Repository Search Path** select the **vivado-library** folder created by the extraction of the zip archive.

Close all windows and return to the **Block Design** window.

Go to the **Board** tab located on the left-hand side of the **Block Design** window and select one of the **Connectors** under the **Pmod** section. Drag and drop it into the **Diagram** window.

A new block with an AXI interface is added to the design. Click on the **Run Connection Automation** message prompt, above our **Diagram** window. In the **Run Connection Automation** window, check **All Automation** and click **OK**. The block is automatically connected to the processing system.

During synthesis, you will probably receive a **Critical Warning** stating that **IP xxxx** was packaged with a board value different from our ZYBO Z7-10. You may safely ignore this warning. It is just to let you know that the **IP core** was packaged using a different board.

Digilent provides a **Getting Started with Digilent Pmod IPs** guide in its website, address

<https://reference.digilentinc.com/learn/programmable-logic/tutorials/pmod-ips/2018.2>

8. Making an HDL Wrapper

After the design validation step, we proceed with the creation of an **HDL System Wrapper**. The wrapper defines the block design as the top-level design, so we may simulate, synthesize, implement, and generate a bitstream for the block design.

1. In the **Flow Navigator** window, in the left side of the **Block Design** main window, in the **Project Manager** section, click **Settings**
2. In the new **Settings** window, click **General** in the vertical toolbar
3. Choose **VHDL** as **Target language** (Figure 18), and click **OK**
4. Click on the **Sources** tab located in left hand side of the **Block Design** window
5. Right click on our block design **Design Sources** → **tutorial_design (tutorial_design.bd)** and select **Create HDL Wrapper...** (Figure 19)

If you are making changes in an existing design and need to create a new updated **HDL Wrapper**, right click on the block design **Design Sources** → **tutorial_design_wrapper(STRUCTURE)(tutorial_design_wrapper.vhd)** and select **Remove File from Project**. In the new window, confirm the deletion. To create the new updated **HDL Wrapper**, repeat now step 5.

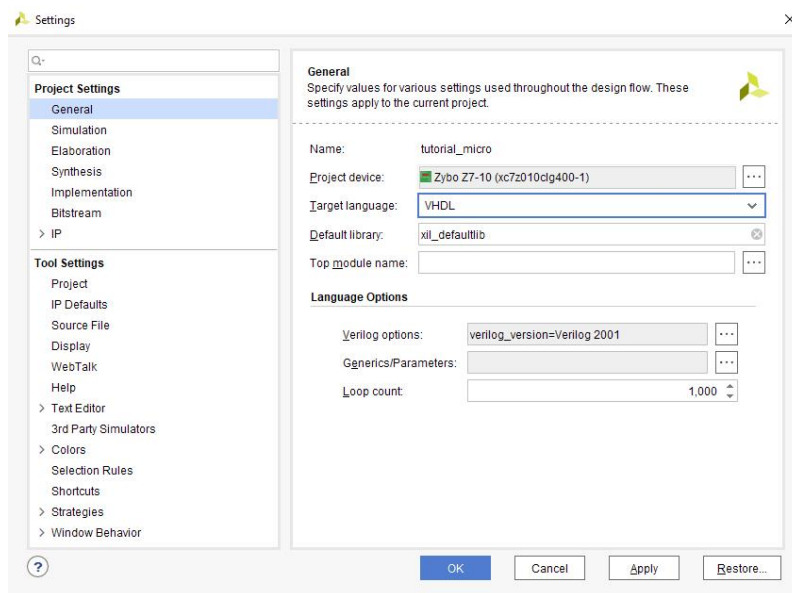


Figure 18

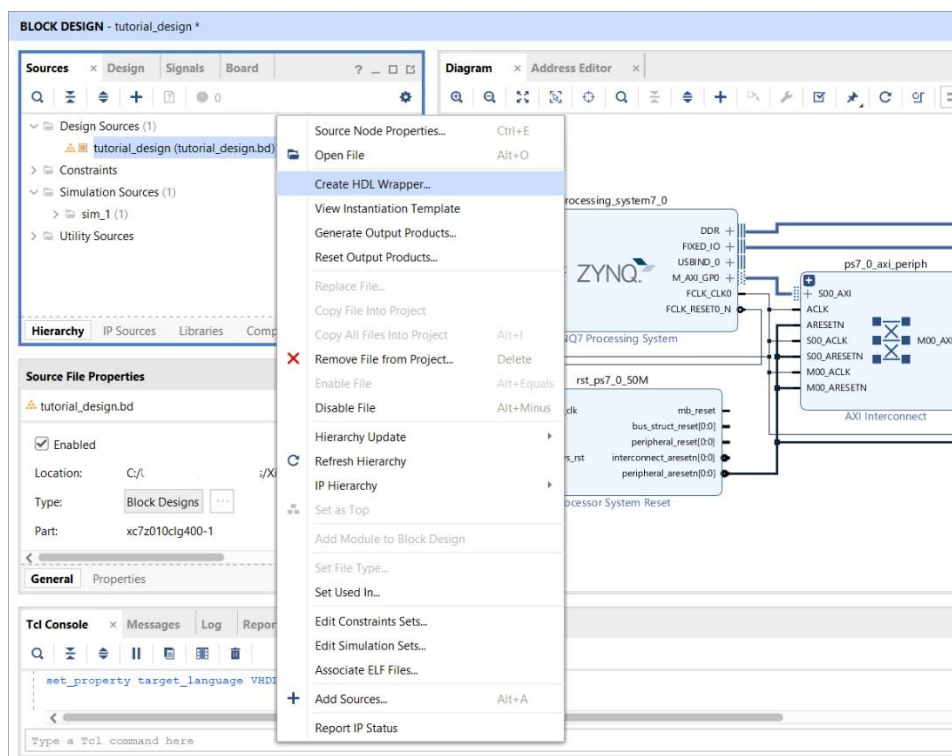


Figure 19

6. In the new **Create HDL Wrapper** window, check that **Let Vivado manage wrapper and auto-update** is selected, and click **OK**
7. In the end, go to the **Sources** window and double-click **Design Sources** → **tutorial_design_wrapper(STRUCTURE) (tutorial_design_wrapper.vhd)**, to open the newly created VHDL description of our design

Notice that the newly created VHDL file contains only a structural description that defines how our blocks are interconnected. Each IP block is treated as a “black box”.

9. Generating the Bitstream

Now that we completed the hardware part of our embedded system design, we are going to synthesize, implement and generate the bitstream necessary for the configuration of the Zynq-7010 FPGA of our ZYBO Z7-10 board.

1. In the **Flow Navigator** window, on the left side of the **Block Design** main window, in the **Program and Debug** section, click **Generate Bitstream**
2. In the new **No Implementation Results Available** window, click **Yes** to synthesize and implement the design, and to generate the bitstream
3. If a **Launch Runs** window appear, just click **OK**
4. In the end of the bitstream generation process, in the new floating window, click **OK** to open the implemented design

In the main **Implemented Design** window, we have access to a bunch of information related to the implementation of our design. In the main window, check the **Device** tab to have an idea of the space occupied by our design inside the FPGA, and of the connections between PS and PL. At the bottom, several tabs give access to the I/O ports map, the estimated power consumed by our design, a summary of the timing analysis, and to a long list of reports generated during the synthesis, implementation, place and route, and bitstream generation. Take time to look at this information. It gives an idea of the complexity of the process, and of all variables that must be considered to implement our design inside a ZYNQ7 Processing System.

10. Vitis Unified Software Platform

Now that the hardware is done, it is time to export it to the Vitis Unified Software Platform to start creating the software that will run in our ZYNQ7 Processing System.

The Vitis Integrated Development Environment (IDE) is part of the Vitis Unified Software Platform. The Vitis IDE is designed to be used for the development of embedded software applications targeted towards Xilinx embedded processors. The Vitis IDE is based on the Eclipse

open-source standard and works with hardware designs created with Vivado Design Suite. The features for software developers include, among others:

- Feature-rich C/C++ code editor and compilation environment
- Project management
- Application build configuration and automatic Makefile generation
- Error navigation
- Integrated environment for seamless debugging and profiling of embedded targets
- Source code version control
- System-level performance analysis
- Focused special tools to configure FPGA
- Bootable image creation
- Flash programming
- Script-based command-line tool

In the Vitis Unified Software Platform, a platform project groups hardware and domains together. A domain can refer to the settings and files of a standalone Board Support Package (BSP), a Linux Operating System (OS), a third-party OS/BSP like FreeRTOS, or a component like a device tree generator. Boot components like First Stage Boot Loader (FSBL) and Platform Management Unit Firmware (PMUFW) are automatically generated in platform projects. A system project groups together applications that run simultaneously on the device.

Some key concepts are essential to understanding the Vitis embedded software development flow (adapted from: Vitis Unified Software Development Platform Documentation, UG1416).

They are:

- Workspace

When you open the Vitis software platform, you create a workspace. A workspace is a directory location used by the Vitis software platform to store project data and metadata. An initial workspace location must be provided when the Vitis software platform is launched;

- Platform

The target platform, or platform, is a combination of hardware components (XSA - Xilinx Support Archive) and software components (domains/BSPs, boot components such as FSBL, and so on);

- Platform Project

A platform project is customizable for adding domains and modifying domain settings. A platform project can be created by importing an XSA file (.xsa), or by importing an existing platform;

- Domain

A domain is a Board Support Package (BSP) or the Operating System (OS), with a collection of software drivers on which to build your application. The created software image contains only the portions of the Xilinx library you use in your embedded design. You can create multiple applications to run on the domain. A domain is tied to a single processor or a cluster of isomorphic processors¹ (for example A53_0 or A53) in the platform;

- System Project

A system project groups together applications that run simultaneously on the device. Two standalone applications for the same processor cannot sit together in a system project. Two Linux applications can sit together in a system project. A workspace can contain multiple system projects;

- Application (Software Project)

A software project contains one or more source files, along with the necessary header files, to allow compilation and generation of a binary output ELF (Executable and Linkable Format) file (.elf). A system project can contain multiple application projects. Each software project must have a corresponding domain;

Vitis IDE provides a variety of software packages, including drivers, libraries, board support packages, and complete operating systems to help us develop a software platform. All this is described in the Xilinx Standalone Library Documentation - OS and Libraries Document Collection – UG643.

11. Exporting Hardware Design to Vitis

1. In the main toolbar, select **File → Export → Export Hardware...**
2. Check the box to **Include Bitstream** and click **OK** (Figure 20)

An XSA file is created in the root folder of the project. XSA is a container that contains:

- One or more .hwh (Hardware Handoff File) files:
 - Vivado tool version, part, and board tag information;
 - IP - instance, name, VLVN (vendor, library, name, and version), and parameters;

¹ “Two processors P1 and P2 are isomorphic iff:

1. all their physical characteristics viz. processing speed, memory capacity, etc. are the same;
2. is there exists r number of k distant neighbor processors for P1, only the same r number of k distant neighbor processors should be there for P2.”

Douglas Antony Louis Piriya Kumar, C. Siva Ram Murthy, and Paul Levi. 1998. A New A* Based Optimal Task Scheduling in Heterogeneous Multiprocessor Systems applied to Computer Vision. In *Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking (HPCN Europe 1998)*. Springer-Verlag, Berlin, Heidelberg, 315–323.

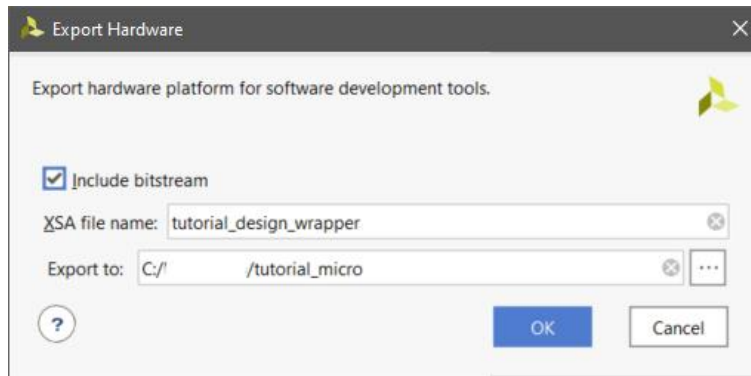


Figure 20

- Memory Map information of the processors;
- Internal Connectivity information (including interrupts, clocks, etc.) and external ports information.
- BMM (Block Memory Map)/MMI (Memory Map Information) and BIT (bitstream) files;
- User and HLS (High-Level Synthesis) driver files;
- Other meta-data files.

12. Create a Platform Project in Vitis

1. In the main toolbar, select **Tools → Launch Vitis**
2. Choose the folder where you want to create the Vitis workspace and click **Launch** (Figure 21)

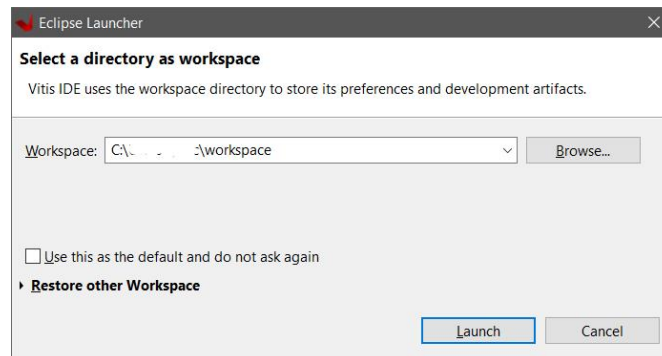


Figure 21

13. The Vitis environment

The software development cycle begins with the handoff files created during the hardware design flow with the Vivado tools. Vitis uses these files in driver and application development. They contain the initialization code for the Zynq SoC Processing System and initialization settings for DDR, clocks, phase-locked loops (PLLs), and MIOs.

The drivers for Xilinx soft IPs and hard processors are shipped with Vitis. These drivers are located in the Xilinx/Vitis/20xx.x/data/embeddedsw/XilinxProcessorIPLib/drivers folder in our Vitis installation folder. Each driver is located in a separate folder. For example, the driver that supports the Xilinx PS GPIO Controller is in the `gpiops_vx_x` (v means version) folder of our Vitis installation. Inside each driver folder, we find four subfolders:

- `data`: contains the MDD file and Tcl script file;
- `doc\html\api`: contains help files that provide details of APIs and data structures implemented in the driver; the `\gpiops_v3_4\doc\html\api` contains an `index.html` that gives access to Documentation, Data Structures, APIs, File List, and implementation Examples;
- `examples`: contains C source files demonstrating examples on how to use the driver; the examples are ready to build and launch on the hardware; they can be used as-is, when following the procedure for building stand-alone applications, or may be modified as needed; the examples may be accessed from the `index.html` file;
- `src`: contains the driver source code (C and H (Header) files); the sources may be accessed from the `index.html` file.

The **Vitis IDE** window (Figure 22) enables the access to different tools and sources of information about Vitis. Explore them at will. We may always return to this welcome window by choosing **Help → Welcome** in the main toolbar.

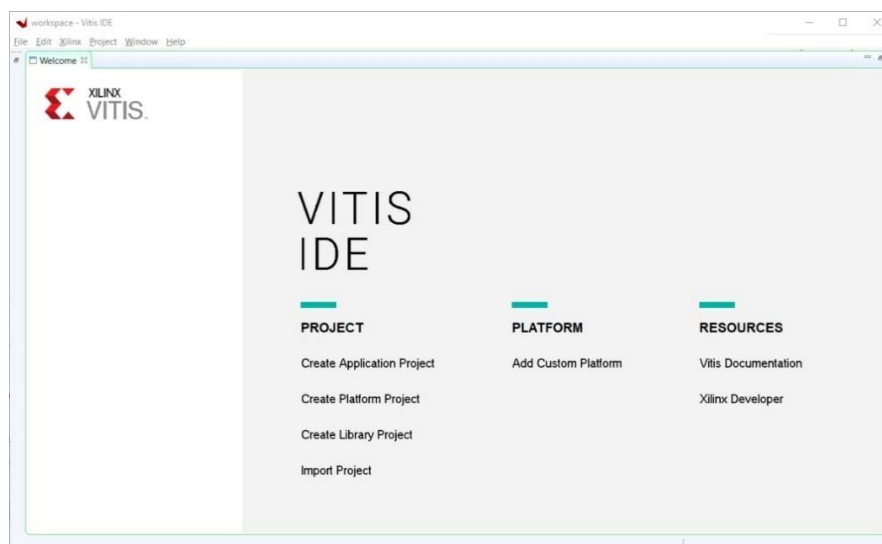


Figure 22

The **Help → Xilinx OS and Libraries Help** link gives access to information about a variety of Xilinx software packages, including drivers, libraries, board support packages, and complete operating systems to help us develop a software platform. Device drivers are documented along with the corresponding peripheral documentation. The embedded drivers and libraries are hosted on the

Xilinx wiki, which can be accessed at the “Embedded Driver and embedded Library Documentation”:

<https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841745/Baremetal+Drivers+and+Libraries>

14. Creating a New Platform Project in Vitis

The first application we are going to create is the famous “Hello World”. Despite being possible to merge the creation of a new platform and a new application project in a single step, we are going to separate it for clarity purposes. So, we start by creating the platform project based on the hardware specification developed in Vivado, and after the applications to run on the Zynq processor, starting with the “Hello World” application, and continuing with the 4-bit up/down counter we created in the *Vivado Quick Start Tutorial for the Digilent Zybo Z7-10 Zynq 7000 ARM/FPGA SoC Platform board – a hardware approach* tutorial, now using a software-based approach.

1. Close the **Vitis IDE Welcome** window
2. In the Vitis workspace main toolbar select **File → New → Platform Project...**
3. The **Create new platform project** dialog box opens in the **New Platform Project** window
4. Type **tutorial_micro_platform** in the **Project Name** field (Figure 23)
5. Leave all the other fields with their default values and click **Next**

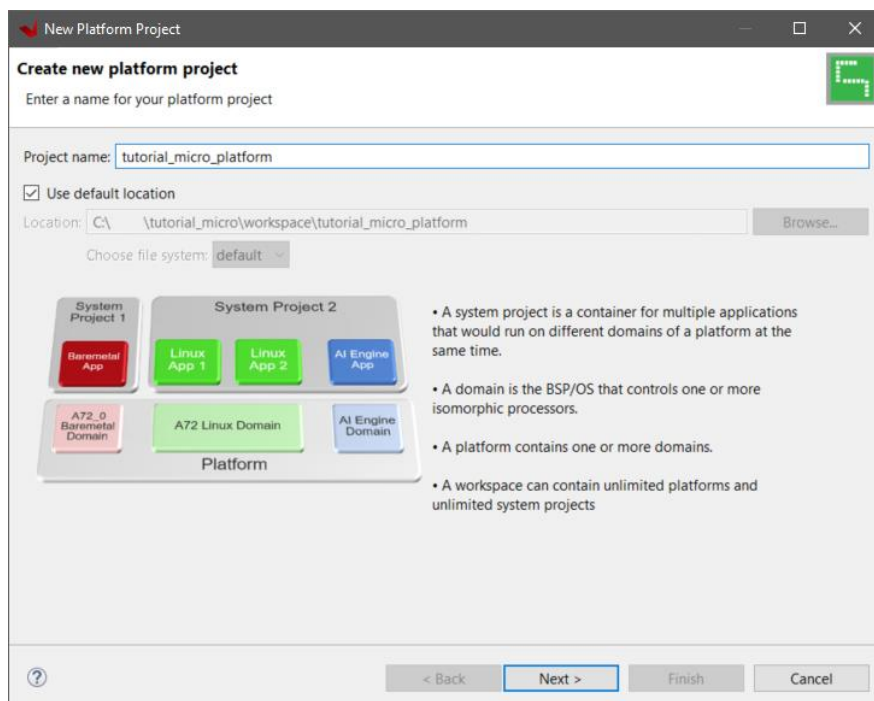


Figure 23

6. In the **Platform Project** page, select the **Create from hardware specification (XSA)** option

7. In the new **Platform Project Specification** page, navigate to the folder where the XSA file **tutorial_design_wrapper.xsa** was created in Vivado to add it (Figure 24)

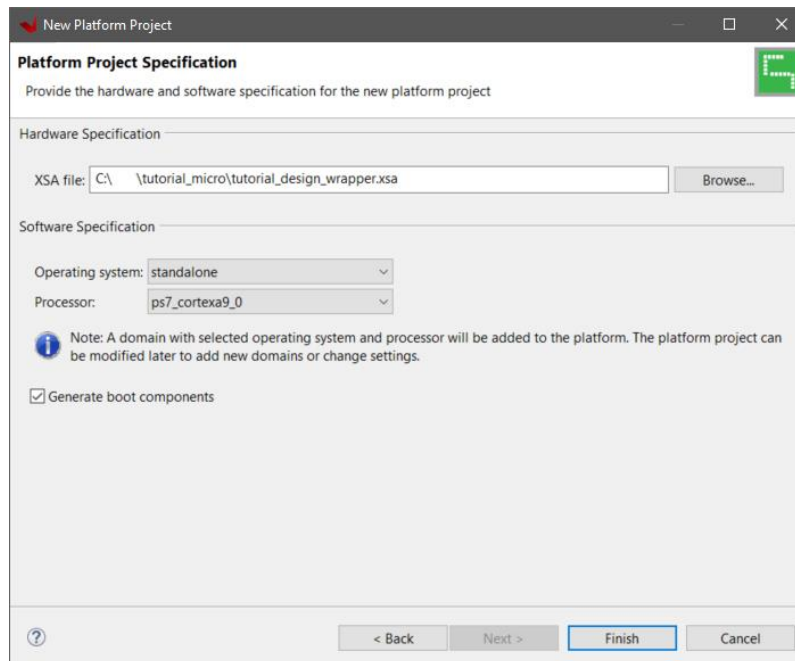


Figure 24

Automatically, some **Software Specifications** are chosen by default: the **Operating system**, **standalone**, a single-threaded, simple operating system (OS) platform that provides the lowest layer of software modules used to access processor-specific functions (check the Standalone v7.1 document, UG647), and the **Processor**, from the two ARM Cortex-A9 available in our Zynq xc7z010, where our applications are going to run - **ps7_cortexa9_0**. Other options for the operating system are **Linux** or **freertos10_xilinx**, a real-time operating system from Xilinx.

The **Generate boot components** checkbox is selected by default, adding the First Stage Bootloader (FSBL) for Zynq to our project, which configures the FPGA with hardware bitstream (if it exists) and loads the Standalone (SA) Image or Operating System (OS) Image and takes the Cortex-A9 out of reset.

8. Leave all the other fields with their default values and click **Finish**

The hardware platform project is created within the Vitis workspace. It is generated with multiple domains, one we specified, and the ones required for the boot components. It can later be modified to add new domains.

In the **Main** window of the **Design** perspective, we have access to the **Board Support Package (BSP)** settings and to the list of **Peripheral Drives** used by our platform. The address map of the system is accessible selecting the **Hardware Specification** tab at the bottom of the **Main** window, just above the Vitis **Console**.

When building the BSP project, files **ps7_init.c** and **ps7_init.h** contain the initialization code for the Zynq SoC Processing System and initialization settings in the form of **#define** constants for DDR, clocks, phase-locked loops (PLLs), and MIOs. Vitis uses these settings when initializing the processing system so that applications can be run on top of it. Software drivers and applications can reference these constants in the code. These files are accessible through the **Explorer** window, on the left side of the Vitis workspace, at **tutorial_micro_platform → zynq_fsbl**

9. To generate the platform, right-click the platform project **tutorial_micro_platform (Out-of-date)** and select **Build Project** (Figure 25)

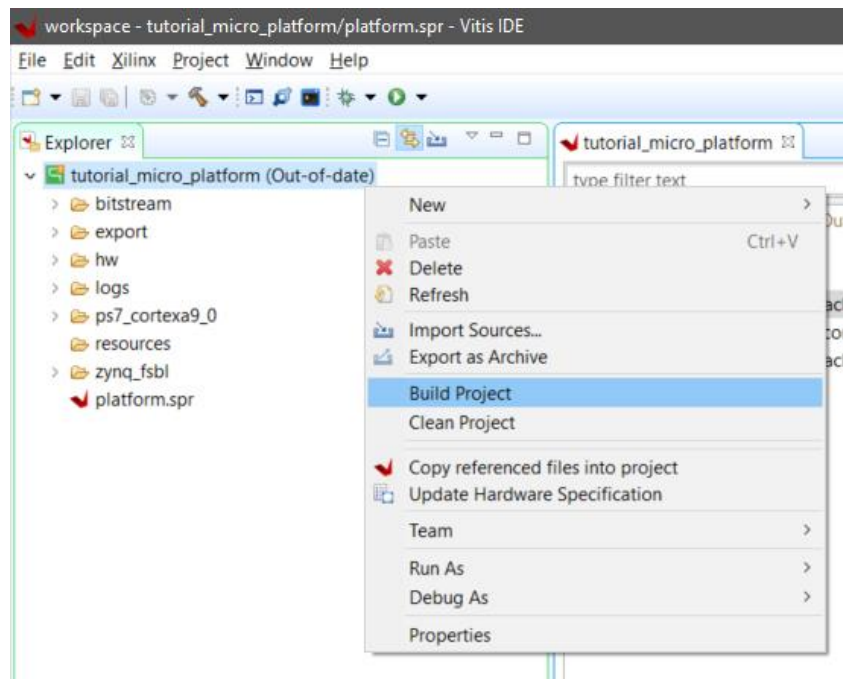


Figure 25

15. Creating a New Application Project in Vitis

Now that we have our platform, the next step is to create our first software application project, the famous “Hello World”. Software application projects are the final application containers. The project directory that is created contains (or links to) your C/C++ source files, executable output file, and associated utility files, such as the Makefiles used to build the project.

1. In the Vitis workspace main toolbar select **File → New → Application Project...**
2. The **Create a New Application Project** dialog box opens in the **New Application Project** window
3. Type **Hello_world** in the **Project Name** field (Figure 26)
4. Leave all the other fields with their default values and click **Next**

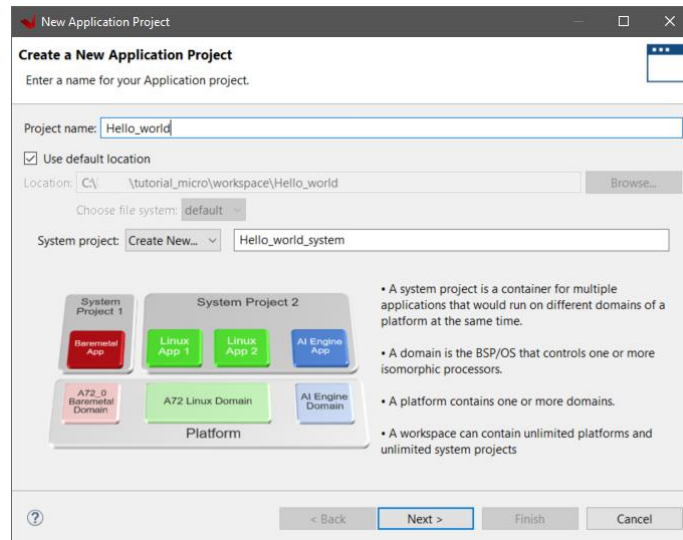


Figure 26

5. In the **Select a platform from repository** tab of the **Platform** page, select the newly created **tutorial_micro_platform** (Figure 27)

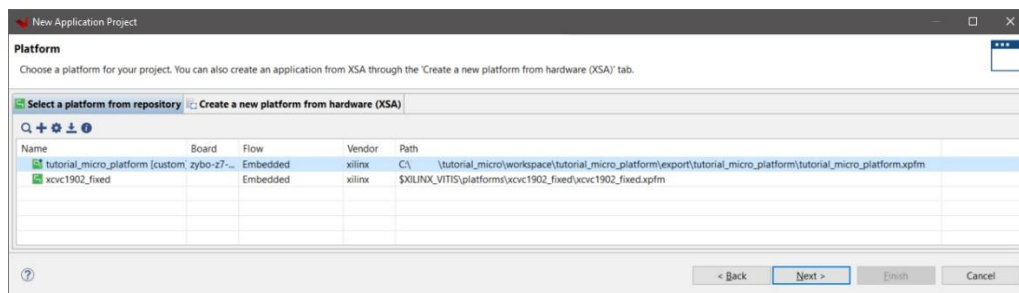


Figure 27

6. In the **Domain** page leave all the fields at their default values and click **Next** (Figure 28)

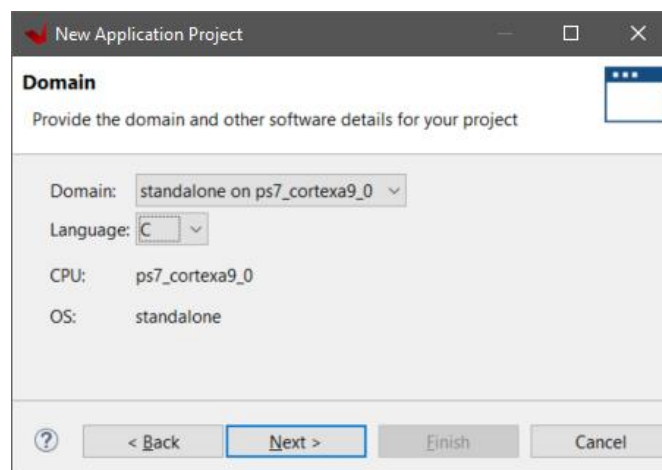


Figure 28

7. In the **Templates** page, select **Hello World** from the list of **Available Templates:** and click **Finish** (Figure 29)

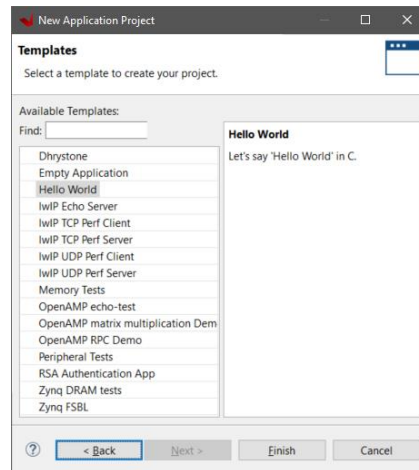


Figure 29

The application project is created in the Vitis workspace.

A new directory tree appears in the **Explorer** window on the left side of the workspace. Under the **Hello_world_system**, the **Hello_world** folder is our main working source folder. It contains all the binaries, .C, and .H files. Inside the **src** folder is our application file, **helloworld.c**. To open the file in the main Vitis workspace window, double-click on it. You may modify its content if you want (Figure 30). This folder also contains an important file, the **lscript.ld**. This is a Xilinx auto-generated linker script file. It is used to control where different sections of an executable are placed in memory. Double-click to open it and to check its content.

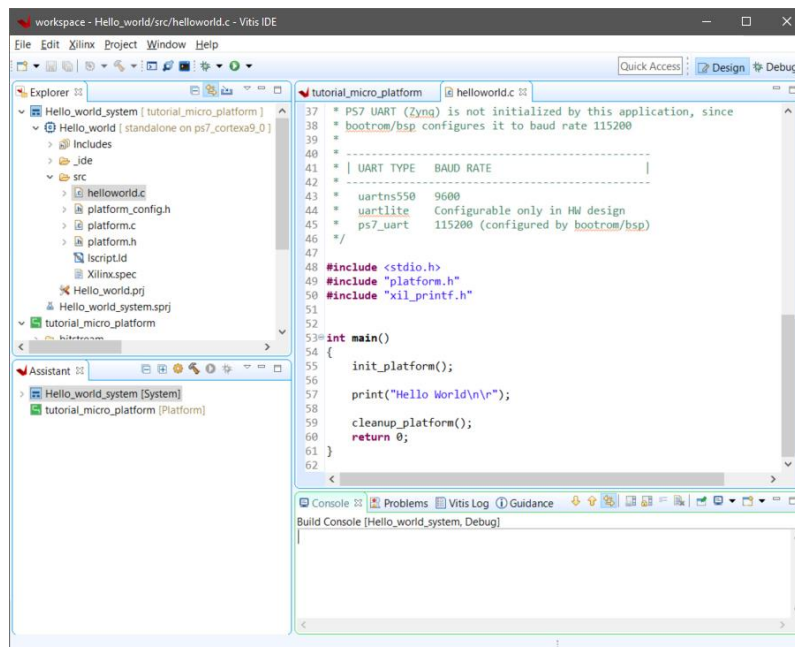


Figure 30

Two other files present in the **Hello_world** folder are the **Hello_world.prj**, which contains the **Application Project Settings**, and the **hello_world_system.sprj**, which contains the **System**

Project Settings and the list of **Application projects**. Double-click each one to open them and check their content.

16. Compiling the Application Project

1. Right-click on the project folder and select **Build Project** (Figure 31)

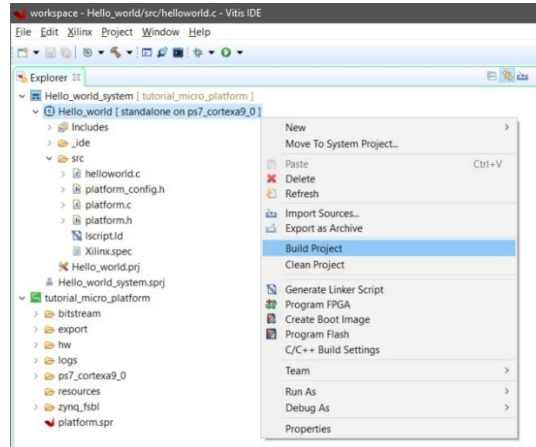


Figure 31

2. Check the **Console** window at the bottom of the Vitis workspace to see when the application project finishes compiling.

The build is the process of converting the source code files into a standalone software file that can be run on the Zynq processor.

17. Setting up the UART Terminal and Running the Application Project

Make sure that the ZYBO Z7-10 is turned on and connected to the host PC via the USB-JTAG port – **PROG UART** – of the board. This port serves a dual purpose as the ZYBO Z7-10 Programming port and as the USB-UART connection to the ZYNQ7 Processing System.

The Zybo Z7 supports three different boot modes: microSD, Quad SPI Flash, and JTAG. The current Zynq configuration boot mode is selected using the **Mode** jumper (JP5) in the Zybo-Z7 board. Since we are configuring the board using the JTAG interface, check that jumper JP5 on the board is selecting **JTAG** (Figure 32).

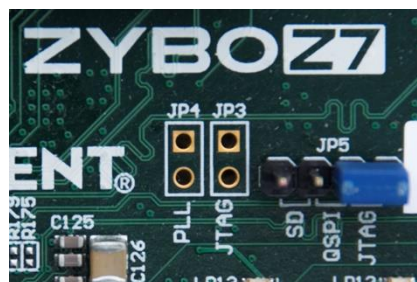


Figure 32

The Zybo Z7 has an onboard 16MB Quad-SPI (Serial Peripheral Interface) Flash memory. Quad-SPI-based interface memories are faster than traditional SPI-based as Quad-SPI uses 4 data lines (I0, I1, I2, and I3) as opposed to just 2 data lines (MOSI and MISO) on the traditional SPI-based interfaces. By default, the board comes with JP5 selecting the Quad-SPI Flash memory (QSPI). This enables the configuration of the FPGA with a manufacture demonstration example stored in the Quad-SPI Flash memory, when the board is powered-up.

The Vitis platform has its own incorporated serial terminal application, the **Vitis Serial Terminal**.

1. To open the **Vitis Serial Terminal** click on **Quick Access** near the top right corner of the Vitis workspace and type **terminal** (Figure 33)
2. Select **Vitis Serial Terminal (Xilinx)** from the list

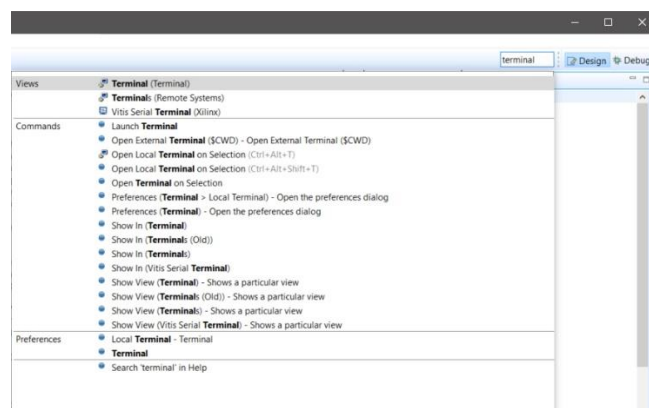



Figure 33

3. In the new **Vitis Serial Terminal** click on the  icon to add a port to the terminal
 Configure the ZYBO Z7-10 UART port to **Port:** COMx (the serial port number depends on your computer) with a **Baud Rate:** 115200 bit/s, the baud rate defined by default for UART1, the one connected to the **PROG UART** connector in the ZYBO Z7-10, in the **PS-PL Configuration** of the **ZYNQ7 Processing System Re-customize IP** window (Figure 11).
4. Configure the other **Advance Settings** to:
 - **Data Bits:** 8
 - **Stop Bits:** 1
 - **Parity:** None
 - **Flow Control:** None
5. Leave **Timeout (sec):** blank (Figure 34)

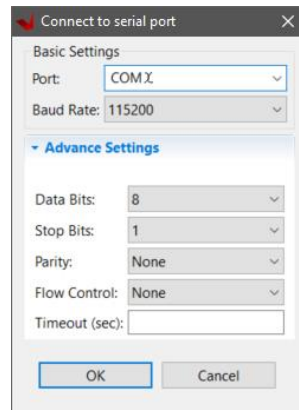


Figure 34

6. Click **OK** to close the configuration window
7. The message **Connected to COMx at 115200** appears in the **Vitis Serial Terminal** window
8. In the **Explorer** window on the left side of the Vitis workspace, select the **Hello_world_system [tutorial_micro_platform]**
9. Right click and select **Run as → 1 Launch on Hardware (System Project Debug)** (Figure 35)

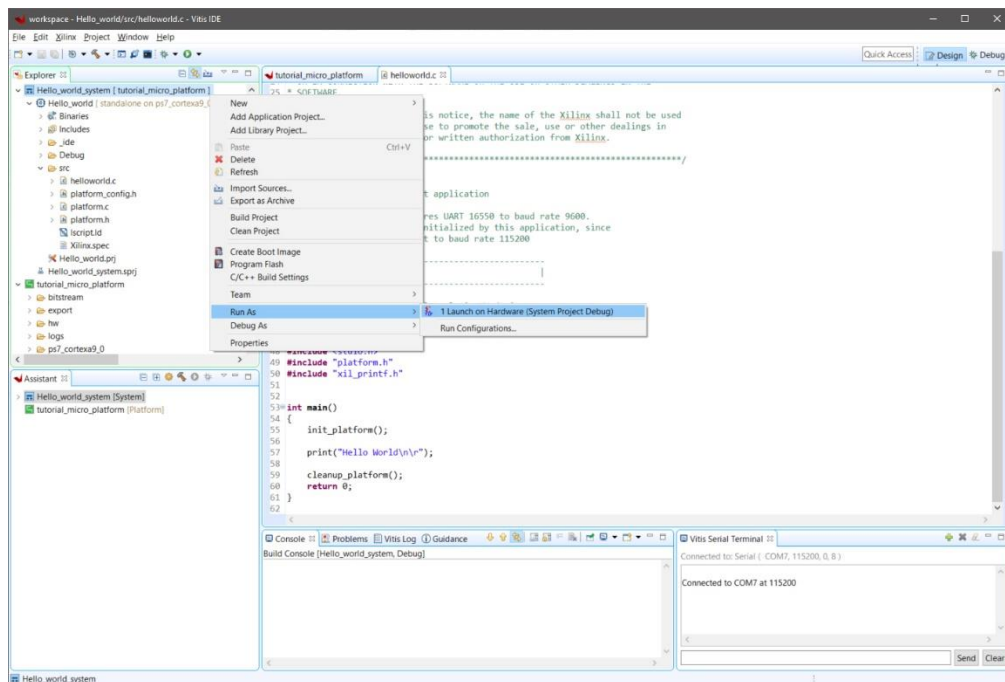


Figure 35

10. Check the **Vitis Serial Terminal** (Figure 36)
11. Everything seems to be working as expected! However, if nothing appears in your **Vitis Serial Terminal**, maybe you picked up the wrong port. Go back and change your serial port and repeat and run the application again.

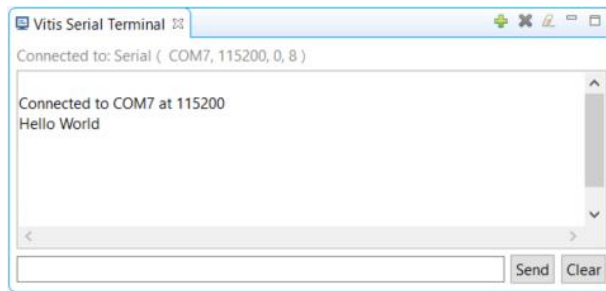


Figure 36

18. Implementing the up/down counter on the ZYNQ7 Processing System Processor

The next step is to implement the up/down counter now using a software approach. The hardware platform we created in Vivado already contains the GPIO IP block that enables the read of the switches and the write of the board LEDs by the ZYNQ7 Processing System processor. In the Vitis environment, the same system project can contain multiple application projects (or applications). Thus, we just need to create the software to run on it.

1. In the main toolbar, select **File → New → Application Project...**
2. In the next window, type **up_down_counter** in the **Project Name** field
3. Leave all the other fields with their default values and click **Next**
4. In the **Select a platform from repository** tab of the **Platform** page, select **tutorial_micro_platform** (see Figure 27), and click **Next**
5. In the **Domain** page leave all the fields at their default values and click **Next** (see Figure 28)
6. In the **Templates** page, select **Empty Application** from the list of **Available Templates:** and click **Finish** (Figure 37)

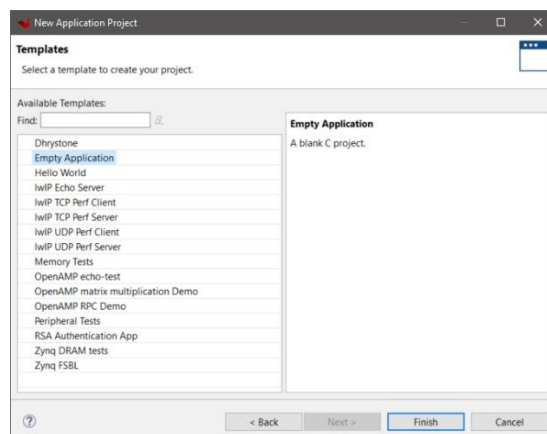


Figure 37

The new application project is created in the Vitis workspace.

7. In the **Explorer** window, open the directory tree **up_down_counter_system → up_down_counter → src**

Obviously, there are no source files yet. We must create our own .c source code file for the application.

8. Right click on **src** → **New** and select **File** (Figure 38)

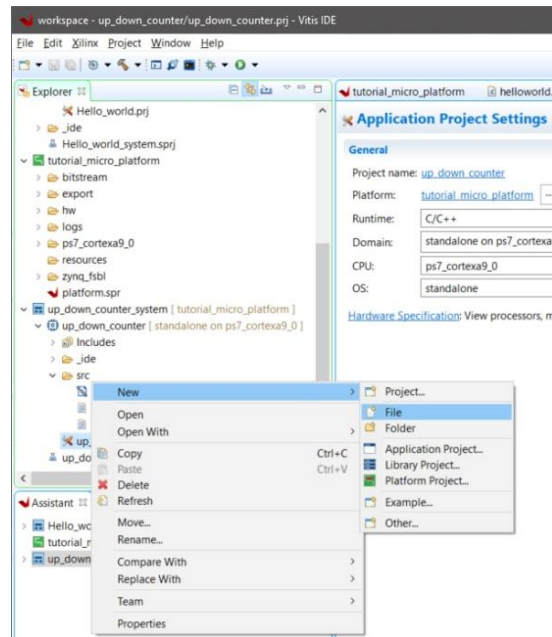


Figure 38

9. In the new window, type **up_down_counter.c** in the **File name:** field, leaving the other fields unchanged, and click **Finish** (Figure 39)

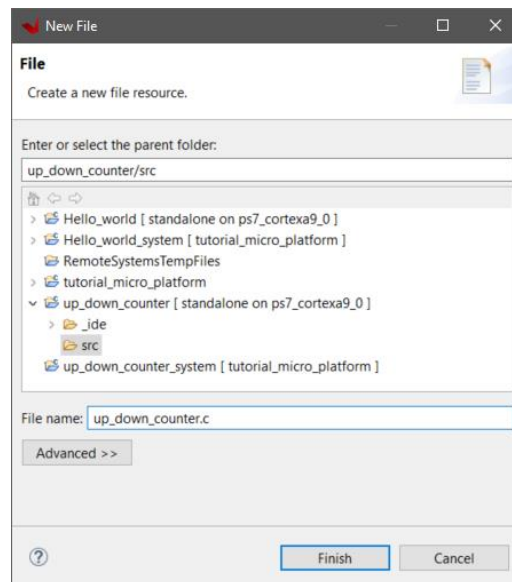


Figure 39

The **up_down_counter.c** file opens in the Vitis workspace main window. Now, it is time to add the C code that will implement our up/down counter.

10. Copy and paste the following code into the file:

```

/* Include Files */
#include "xparameters.h"
#include "xgpio.h"
#include "xstatus.h"
#include "xil_printf.h"

/* Definitions */
#define GPIO_DEVICE_ID      XPAR_AXI_GPIO_0_DEVICE_ID /* GPIO device LEDs & switches are connected to */
#define LED                 0x0000 /* Initial LED value - 0000 */
#define LED_DELAY           10000000 /* Software delay length */
#define LED_CHANNEL         1 /* GPIO port for LEDs */
#define SWITCHES_CHANNEL    2 /* GPIO port for SWITCHES */
#define printf              xil_printf /* smaller, optimized printf */

XGpio LEDInst, SWITCHESInst; /* GPIO Device driver instance */

int CounterLeds(void)
{
    volatile int Delay;
    u32 SWITCHES;
    u32 MASK = 0x0003; /* Masks status of switches SW2 and SW3 */
    int led = LED; /* Hold current LED value. Initialize to LED definition */

    while (1) {
        /* Receives the status of the switches and masks SW2 and SW3 */
        /* Only SW0 and SW1 status determines the counter operation */

        SWITCHES = (MASK & (XGpio_DiscreteRead(&SWITCHESInst, SWITCHES_CHANNEL)));

        /* Verifies if the counter is enabled (SW1=1) */

        if((SWITCHES == 0x0000) || (SWITCHES == 0x0001)){
            led = led;
        }else{
            /* If the counter is enable and counting up (SW0=1) reset it if it overloads */

            if(SWITCHES == 0x0003){
                if (led == 0xFFFF){
                    led = 0x0000;
                }else{
                    /* or increment the counter otherwise */
                    led = led+1;
                }
            }

            /* If the counter is enable and counting down (SW0=0) set it if it reaches zero */

            if(SWITCHES == 0x0002){
                if(led == 0x0000){
                    led = 0xFFFF;
                }else{
                    /* decrement the counter otherwise*/

                    led = led-1;
                }
            }
        }

        /* Write output to the LEDs */

        XGpio_DiscreteWrite(&LEDInst, LED_CHANNEL, led);

        /* Wait a small amount of time so that the LED counting is visible */

        for (Delay = 0; Delay < LED_DELAY; Delay++);
    }

    return XST_SUCCESS; /* Should be unreachable */
}

/* Main function. */

int main(){
    /* Initialize the GPIO */

    int Status;
    Status = XGpio_Initialize(&LEDInst, GPIO_DEVICE_ID);
    if (Status != XST_SUCCESS)
        return XST_FAILURE;

    /* Initialize the SWITCHES GPIO */

```

```

        Status = XGpio_Initialize(&SWITCHESInst, GPIO_DEVICE_ID);
        if (Status != XST_SUCCESS)
            return XST_FAILURE;

/* Set IO directions */
/* Parameters */
/* InstancePtr is a pointer to an XGpio instance to be worked on */
/* Channel contains the channel of the GPIO (1 or 2) to operate on */
/* DirectionMask is a bitmask specifying which IOs are input and which are output. */
/* Bits set to 0 are output and bits set to 1 are input */

/* Set all LEDs IO direction to outputs */

XGpio_SetDataDirection(&LEDInst, LED_CHANNEL, 0x0);

/* Set all switches direction to inputs */

XGpio_SetDataDirection(&SWITCHESInst, SWITCHES_CHANNEL, 0xF);

/* Call the function Counter */
CounterLeds();
return 0;
}

```

11. Save the file by selecting, in the main toolbar, **File → Save**, or clicking in the **Save** toolbar icon (Figure 40)

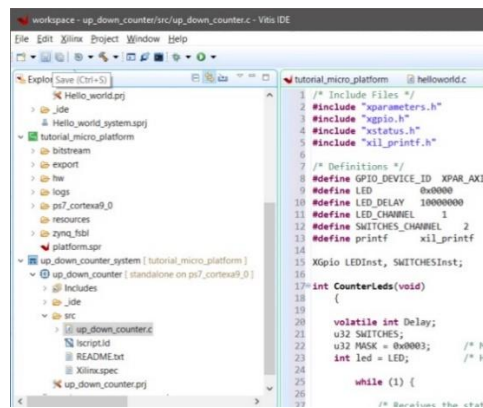


Figure 40

12. To compile the new application project, right-click on the project folder and select **Build Project** (Figure 41)
13. Make sure that the ZYBO Z7-10 is turned on and connected to the host PC via the USB-JTAG port
14. In the **Explorer** window on the left, select the **up_down_counter_system [tutorial_micro_platform]**
15. Right click and select **Run as → 1 Launch on Hardware (System Project Debug)** (Figure 42)
16. If the **Hello_world** application is still running in the ZYNQ7 Processing System, a **Conflict** window appears, asking if we wish to terminate the previous launched configuration (Figure 43)
17. Click **Yes**

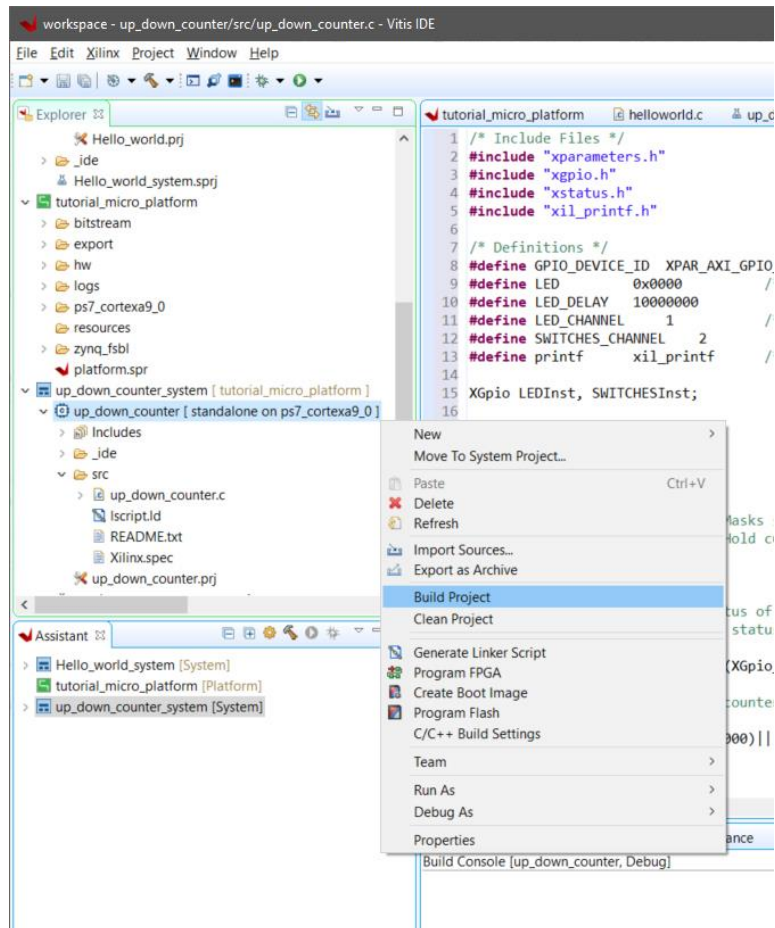


Figure 41

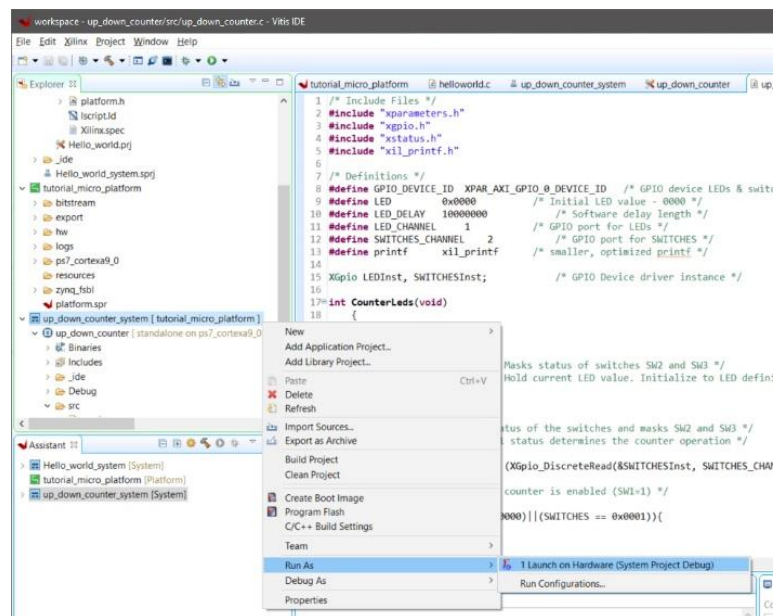


Figure 42

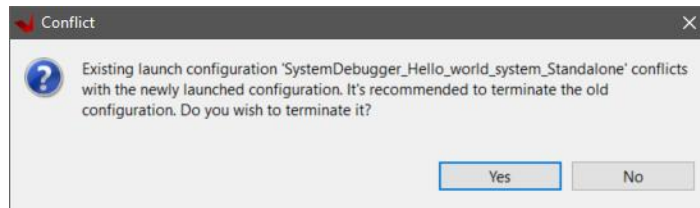


Figure 43

In the end, LED LD(3..0) on the board lit up, indicating that the counter is running. If not, remember that when the enable signal is disabled (SW1) the LEDs do not light, so switch SW1. Switching SW0, the counting direction changes. We may stop the counter changing the state of switch SW1.

19. Some notes about the code

The command **XGpio_Initialize(&Gpio, GPIO_DEVICE_ID);** is a function provided by the GPIO device driver in the file **xgpio.h**. It initializes the XGpio instance, Gpio, with the unique ID of the device specified by **GPIO_DEVICE_ID**.

At the top of the source file, you see that **GPIO_DEVICE_ID** is defined as **XPAR_AXI_GPIO_0_DEVICE_ID**. The value of **XPAR_AXI_GPIO_0_DEVICE_ID** can be found by opening the file, **xparameters.h**, which is automatically generated by Vivado when creating the XSA file used to export the hardware design to Vitis IDE. This file contains definitions of all the hardware parameters of the system.

Both files, **xgpio.h** and **xparameters.h**, may be found at the **include** folder, part of the directory tree that appears in the **Explorer** window under **tutorial_micro_platform** → **export** → **tutorial_micro_platform** → **standalone_domain** → **bspinclude**.

The function **XGpio_SetDataDirection(XGpio *InstancePtr, unsigned Channel, u32 DirectionMask);** is also provided by the GPIO device driver, and sets the direction of the specified GPIO port. **u32 DirectionMask** bits set to '0' mean the corresponding IOs of the 32 IO port are defined as output IOs, and set to '1' as input IOs.

If we are specifying the LEDs, we define the connected IOs as outputs. Since there are 4 LEDs on the ZYBO Z7-10 board, by setting the LED channel - channel 1 of the GPIO_0 - direction value to 0x0 or 0000 in binary, we are setting all 4 LEDs as outputs - **XGpio_SetDataDirection(&LEDInst, LED_CHANNEL, 0x0);**

If we are specifying the switches, we define the connected IOs as inputs. Since there are 4 switches on the ZYBO Z7-10 board, by setting the switches channel - channel 2 of the GPIO_0 - direction value of 0xf or 1111 in binary, we are setting all 4 switches as inputs - **Gpio_SetDataDirection(&SWITCHESInst, SWITCHES_CHANNEL, 0xF);**

Further information on the **Peripheral Drivers** can be found by opening the **platform.spr** file, located under the **tutorial_micro_platform** folder in the **Explorer** window, and selecting, in the new window, the **Board Support Package** under **tutorial_micro_platform** → **ps7_cortexa9_0** → **zynq_fsbl**. A list of all the peripherals in the system is provided, along with links to available documentation and examples, as shown in Figure 44.

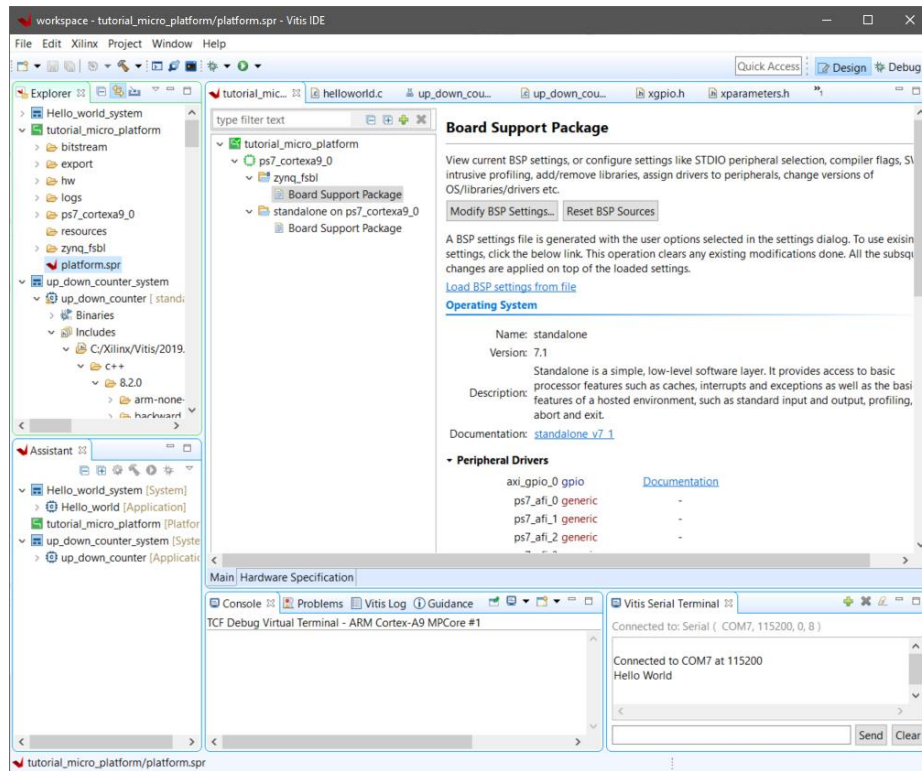


Figure 44

We may add other application projects to the same system project. Figure 45 shows how to access the list of currently available applications from the main horizontal icon toolbar. Just click on the one you want to run it on the board.

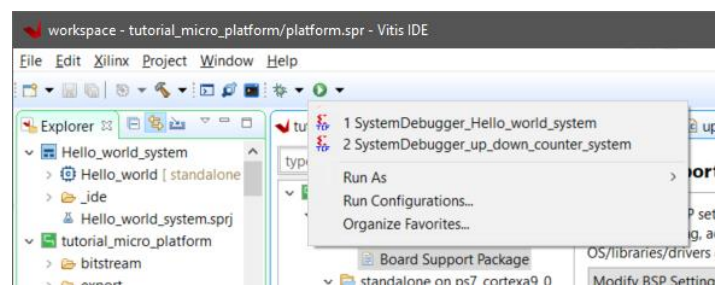


Figure 45

You have completed the *Vivado/Vitis Zynq Quick Start Tutorial for the Digilent Zybo Z7-10 Zynq 7000 ARM/FPGA SoC Platform board – using the Zynq microprocessor.*

References:

Embedded Processor Hardware Design, Vivado Design Suite Tutorial, UG940 (v2019.2) Nov. 26, 2019

Embedded Processor Hardware Design, *Vivado Design Suite User Guide*, UG898 (v2019.2) Oct. 30, 2019

Embedded Software Development, *Vitis Unified Software Platform Documentation*, UG1400 (v2019.2) March 18, 2020

Generating Basic Software Platforms, *Reference Guide*, UG1138 (v2019.2) Oct. 30, 2019

Standalone v7.1, *Xilinx Standalone Library Documentation*, UG647 (2019.2) October 30, 2019

The Zynq Book - Embedded Processing with the ARM® Cortex®-A9 on the Xilinx® Zynq®-7000 All Programmable SoC, Louise H. Crockett, Ross A. Elliot, Martin A. Enderwitz, Robert W. Stewart. 2014. Glasgow: Strathclyde Academic Media.

UltraFast Embedded Design Methodology Guide, UG1046 (v2.3) Apr. 20, 2018

Vitis Unified Software Development Platform Documentation, UG1416 (v2019.2) Mar. 23, 2020

Xilinx Standalone Library Documentation, *OS and Libraries Document Collection*, UG643 (2019.2) Dec. 9, 2019

Xilinx Wiki, available at <https://xilinx-wiki.atlassian.net/wiki/home> Last accessed: April 20, 2020

XMP467 - MicroBlaze Processor Quick Start Guide with Vitis (v1.0) Apr 06, 2020

Zybo Z7 Board Reference Manual, Revised February 21, 2018, *Digilent*, [Online]. Available: <https://reference.digilentinc.com/reference/programmable-logic/zybo-z7/reference-manual>

Zynq-7000 SoC Technical Reference Manual, UG585 (v1.12.2) July 2, 2018

Zynq-7000 SoC: Embedded Design Tutorial, *A Hands-On Guide to Effective Embedded System Design*, UG1165 (v2019.2) October 30, 2019

v. 0	Initial release	May 2019
v. 1	Updated version for Vivado 2019.2 with Vitis Updated reference list	April 2020

Version 1

Manuel Gericota – April 2020