**Embedded linux: vdma and hdmi integration**

Oliwier Jaworski

PXL hogeschool

Elektronica ICT

oliwier.jaworski@student.pxl.be

**ABSTRACT**

This tutorial provides a complete walkthrough from concept to implementation of an embedded Linux application using Xilinx Vivado and PetaLinux. It covers the full process of designing the hardware, integrating AXI-based peripherals in the programmable logic (PL), and building a custom Linux distribution for the ARM processing system (PS). The implementation demonstrates how to enable HDMI output, USB functionality, and PL-connected peripherals throuygh various configurations.It also details the necessary device tree modifications and driver integrations required to achieve a fully functional embedded system.

**Keywords**: PYNQ-Z2, Petalinux, Userspace I/O driver, Simple framebuffer, USB host, AXI-gpio, embedded linux, User-space, Kernel-space, VDMA to hdmi buffering

# 1. INTRODUCTION

Developing an embedded Linux application on Zynq-based platforms involves a complete workflow from hardware design in Vivado to software execution on the ARM processing system (PS). In this project, the focus is on integrating HDMI, USB, and AXI-based peripherals within the programmable logic (PL) to create a functional embedded system. Communication between the PS and PL is achieved through the AXI (Advanced eXtensible Interface) interconnect, which enables data transfer between custom hardware blocks and the Linux operating system.

Vivado is used to implement the hardware design and generate the bitstream that defines the system architecture. Once the hardware is validated, the PetaLinux build framework is employed to create a customized Linux distribution for the target board. PetaLinux simplifies kernel configuration, root filesystem customization, and device tree modification—steps necessary to integrate custom peripherals, drivers, and display interfaces such as VDMA and simple-framebuffers.

After building and deploying the image, the system boots into the tailored Linux environment, where the hardware components can be accessed from user space or through kernel drivers. This workflow provides a foundation for developing complex embedded applications that leverage both the processing capabilities of the ARM cores and the parallel performance of the programmable logic.

# 2. HARDWARE REQUIREMENTS

This is the minimal hardware list to be able to reproduce this tutorial, take note that this tutorial with a little bit of research, can be ported to other hardware too.

* Host machine ([compatible linux distribution](https://docs.amd.com/r/2024.1-English/ug1144-petalinux-tools-reference-guide/Installation-Requirements))
* Vivado, Petalinux, Vitis (version 2024.1)
* PYNQ-Z2
* Ethernet and micro-USB data cable
* SD-card(atleast 8GB)

# 3. HARDWARE DESIGN

This project includes a prebuilt Vivado hardware design to save development time. The design implements **triple buffering of RGBA frame data**, which is required for the X11 display server to operate correctly without startup errors or crashes. Additionally, two AXI GPIO IP blocks are integrated to demonstrate how peripherals can be easily interfaced through the AXI bus and corresponding linux drivers.

Once you have reviewed the hardware design and identified the register address ranges assigned to each IP core, the next step is to verify and finalize the hardware platform by generating the **Xilinx Support Archive (XSA)** file. This file contains the hardware definition, bitstream, and associated metadata required for integration into the PetaLinux build environment.

1. **Validate the design using the option in the top tray of the block design window.**

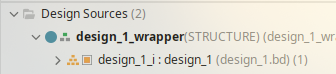
This step ensures that any connection mismatches, data width inconsistencies, or protocol violations are detected early, before proceeding to design packaging or export. Refer to Reference Image 1 or the provided **.pdf** file for a visual reference of the expected design layout.

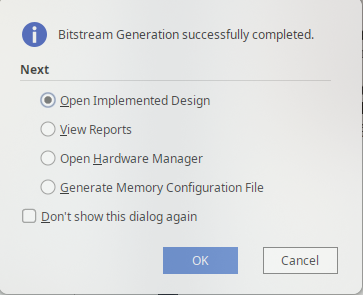
1. **Create the HDL wrapper** by selecting the block design (usually called design\_1) under the Sources section and clicking “Create HDL Wrapper.” Let Vivado manage and auto-update the wrapper.

* Once successful, a wrapper will appear over your block design file.

1. **Generate the bitstream** under the PROGRAM AND DEBUG section, leaving default settings.

* ***Note***: The bitstream is generated for the design set as the top design, which is distinguished by bold text in the Sources panel.

1. Verify bitstream generation



* Once the bitstream generation completes, Vivado will display a message indicating that the design has been successfully implemented and compiled to FPGA logic.
* This confirms that your hardware design is ready to be exported for use in the PetaLinux build process.

1. **Export the hardware design.** Go to File → export → export hardware

* Make sure to include bitstream
* Save the exported hardware platform (XSA file) in your workspace directory for later use in the PetaLinux build.

# 4. BUILD DEVICE TREE BLOB

Once the hardware design has succesfully been exported. The next step is to generate the **device tree** for the given design.

* A **device tree** is a data structure used by the Linux kernel to describe the hardware components of a system. It informs the kernel about available peripherals, memory regions, interrupts, and other hardware-specific configurations, allowing the operating system to interface correctly with the hardware without hard-coded assumptions.
* In this workflow, the device tree will include nodes for the AXI GPIO blocks and any other custom IP, enabling user-space access via the UIO driver.

The automatically generated device tree does not assign **UIO (Userspace I/O)** as the compatible driver for the AXI GPIO peripherals. By default, Vivado and PetaLinux use the xlnx,axi-gpio binding, which lacks user-space mapping support. As a result, no /dev/uio\* device nodes are created by the kernel, preventing direct access from user applications.

To address this, we will first extract the device tree source (DTS) from the project’s .xsa hardware definition file. The following section will describe how to modify the relevant nodes to use the **generic-uio** binding, allowing user-space access to the AXI GPIO peripherals.

In addition, we will adjust the device tree to properly configure the **USB controller** and **simple-framebuffer** node required for the **X11 display server**. Unlike typical GPU-based systems, the framebuffer in this design is not automatically bound to the HDMI output through the VDMA. Instead, the VDMA engine must be explicitly configured at runtime to transfer frame data from memory to the HDMI output pipeline. These adjustments ensure proper USB initialization and a valid framebuffer interface for X11 to operate without errors.

1. **Source the **Xilinx design tools****

* **Before using PetaLinux or related build tools, the Xilinx environment must be sourced by running the setup script, which is typically located under**

~/tools/Xilinx/S\*/V\*/settings64.sh

* **where S\* and V\* correspond to the specific software (e.g., Vitis) and version you want to use.**

1. ****Obtain the Xilinx device tree sources** by cloning the official repository into your workspace:**

git clone -b xilinx\_v2024.1 https://github.com/Xilinx/device-tree-xlnx

1. ****Change** into the newly cloned directory**

cd device-tree-xlnx

1. **Run **XSCT** command**

* ***Note*: This command must be executed in the same terminal where the Xilinx environment was sourced.**

1. ****Open XSA/HDF** file using the following command:**

hsi open\_hw\_design <design\_name>.<xsa|hdf>

1. **Set repository path to the cloned device-tree-xlnx repository**

hsi set\_repo\_path <path to device-tree-xlnx repository>

1. **Extract the processor cell name which will return a list of valid processors**

set procs [hsi get\_cells -hier -filter {IP\_TYPE==PROCESSOR}]

1. ****generate a device tree source** tailored to the exported hardware.**

hsi create\_sw\_design device-tree -os device\_tree -proc ps7\_cortexa9\_0

* **folder named device-tree**
* ****-os device\_tree** specifies that the operating system for this project is a device tree generator, not a full OS.**
* ****-proc psv\_cortexa72\_0** specifies the processor instance that the software design will target.**

1. **Generate DTS/DTSI files to folder**

hsi generate\_target -dir DT\_EXTRACT

1. **Clean up.**

→ hsi close\_hw\_design [hsi current\_hw\_design]

→ exit

Take note of the generated pl.dtsi and zynq-7000.dtsi file in the output folder. These files contains the device tree entries for the components instantiated in the hardware design, and these entries will later be modified to bind the correct driver. For user-space access.

# 5. BUILDING CUSTOM PETALINUX IMAGE

Building the custom Linux image with **PetaLinux** follows a straightforward **workflow**. First, a new PetaLinux project is created and configured according to the hardware design and system requirements. During this configuration step, additional packages can be selected to be included in the root file system. Once the project is set up, the image is compiled and written to an **SD card**, either with dedicated tools or by manually copying the output files.

Although the process is documented in detail in Xilinx reference material, several difficulties may arise. In particular, some configurations provided by the board support package **(BSP)** are not automatically applied when using an exported hardware design **(XSA)**. This requires manual adjustment to enable certain features.

This section outlines the complete workflow for building the software platform. It covers the creation of the **PetaLinux project**, configuration of the **kernel**, **root filesystem**, and **project settings**, along with the required modifications to integrate and enable the drivers responsible for initializing and managing the system peripherals.

1. Creating the petalinux project

* the following command can be used to create a petalinux project from an xsa file.

petalinux-create project -n PYNQZ2\_SDL2 --template zynq --force

* **-n** can be replaced by the user desired name
* **--template zynq**, we select the option to use a template project based on our processing system architecture.
* **--force** makes petalinux create the project folder even if it already exists overwriting old contents.

1. Adding **hw-description** using .xsa

petalinux-config –get-hw-description=${full-path-to-xsa}/\*.xsa --silentconfig

* replace the placeholder with the full-path to your \*.xsa file
* this step will incorporate your custom hardware design into the project.

1. Adding kernel **boot arguments**

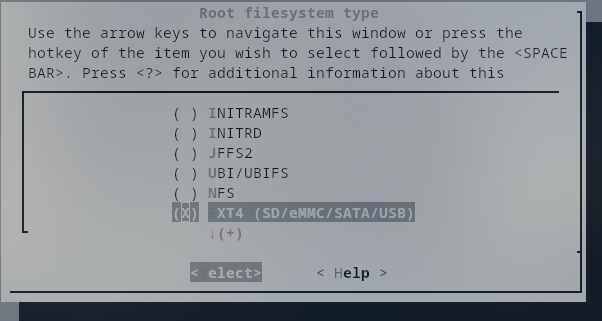
petalinux-config

* This will open a window inside the command line, in which the petalinux system configuration can be done. We are interested in adding boot arguments so we select DTG Settings → kernel bootargs → Add extra boot argument
* A new window will pop up in which you will need to write the following line to enable binding the UIO driver to device.

uio\_pdrv\_genirq.of\_id=generic-uio

* ***Note***: Don’t forget to save your configuration before exiting the configuration panel.

1. Setting the **root file system type** to **EXT4.**

* This is necessary so that the bootloader can correctly locate and mount the root file system during boot.
* To achieve this go back to the root of the petalinux system configuration and select Image Packaging Configuration→ Root filesystem type( press space to select ) EXT4(SD/eMMC/SATA/USB)
* ***Note***: Don’t forget to save your configuration before exiting the configuration panel.
* Once you exit, the configuration changes will be applied automatically; at this stage, the only concern is whether the process completed successfully.

1. Enabling root login

* By default, root login is disabled, leaving no way to log in to the system. This can be corrected by adjusting the root file system settings. In this guide, only the simplest method will be shown. Note that alternative approaches exist for creating users or changing passwords within the PetaLinux project itself.
* To enable root login we will configure the rootfs with the following command:

petaliinux-config -c rootfs

* In ****Image Features****, enable **debug-tweaks** to allow root login, and optionally enable **empty-root-password** to skip the login process entirely.

1. changing the **GPIO driver** inside the petalinux **device tree**

* navigate to the following directory inside your petalinux project

project-spec/meta-user/recipes-bsp/device-tree/files/system-user.dtsi

* Open the file with your prefered text editor. It will look as following:

/include/ "system-conf.dtsi"

/ {

};

* The previously generated device tree will now be used to modify specific entries for the AXI GPIO hardware. By adding the following lines, the default driver for the GPIO blocks is replaced with the ****UIO** driver**, enabling user-space access.

/include/ "system-conf.dtsi"

/ {

};

&axi\_gpio\_0{

compatible = "generic-uio";

};

&axi\_gpio\_1{

compatible = "generic-uio";

};

* By doing this, both GPIO peripherals will appear as /dev/uioX devices, allowing direct access from user-space applications using **mmap**

1. Enabling **UIO kernel driver** on boot**.**

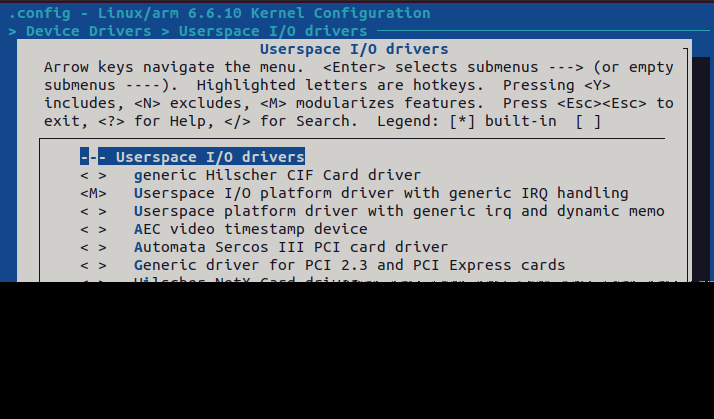
* To ensure the UIO driver is loaded automatically at boot the kernel configuration must have **UIO support enabled**. This can be achieved by changing a value inside the kernel configuration
* accessing the kernel configuration happens with the following command

petalinux-config -c kernel

* Inside the kernel configuration open the following sections

Device Drivers → Userspace I/O drivers

* Select **Userspace I/O (UIO) platform drivers** with **generic IRQ handling** and set it to \* (built-in) instead of M (modular). This ensures that the UIO driver is compiled directly into the kernel and available immediately at boot, without requiring manual module loading.



***!Note*:** No additional packages were added to this PetaLinux build, so it will include only the default packages provided by PetaLinux.Links to instructions for adding custom packages are provided in the Noteworthy section of this tutorial.

Once all configurations have been applied to meet the user’s requirements and ensure the minimum settings for driver support, the final step is to build the custom Linux system using the following command:

petalinux-build

***!Note*:** The build process can take a significant amount of time, may fail on fetching data. If that happens just rerun the command. and may consume large amounts of **RAM**. Based on experience, it is recommended to **enable swap memory** or increase it to at least **32 GB**. Without sufficient memory, the build can quickly max out system RAM, causing the system to become unresponsive and potentially stalling the build indefinitely.

# 6. PACKAGING CUSTOM LINUX

After the build completes, a new directory is created inside the PetaLinux project at **/images/linux/**. This directory contains all files required to create the boot image and package the Linux distribution into a .**wic** file. The resulting .**wic** file can then be written to the SD card using a tool such as **Balena** **Etcher**, or alternatively by manually copying the relevant files to the appropriate SD card partitions.

1. Generation of boot image(**BOOT.BIN**)

* The boot image (**BOOT.BIN**) typically contains the **First Stage Boot Loader (FSBL**), which initializes the processing system (DDR, clocks, and I/O) and then loads the **Second Stage Boot Loader (U-Boot)**. U-Boot provides the environment to load and start the Linux kernel.
* Optionally, the boot image can also include the FPGA bitstream to configure the programmable logic. In this tutorial, the bitstream (**system.bit**) is included. However, Xilinx also provides the **fpgautil** package with Zynq processing systems, which allows users to load FPGA logic dynamically at runtime. This tutorial does not cover the use of fpgautil or the steps required to enable UIO on such a setup.
* The **BOOT.BIN** file is generated inside the **images/linux/** directory, as the command structure and available arguments make this the most convenient working location.

petalinux-package boot --fsbl zynq\_fsbl.elf --fpga system.bit --u-boot u-boot.elf

* The following command must be executed from the base directory of the **PetaLinux project**; otherwise, the **rootfs.tar.gz** file will not be found.

petalinux-package wic -b "BOOT.BIN, uImage, boot.scr"

This command creates a .wic image by bundling the following list of components:

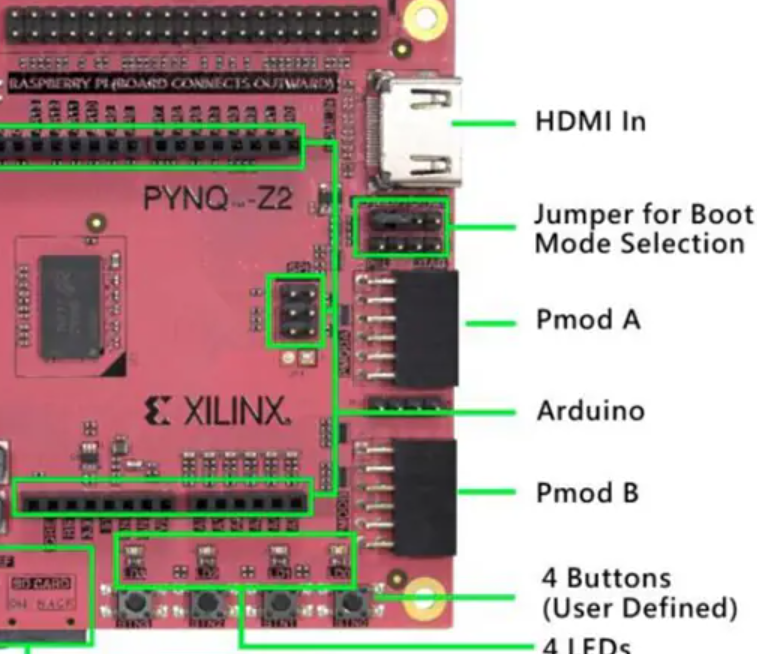
* **BOOT.BIN -** Contains the First Stage Boot Loader (FSBL), U-Boot, and optionally the FPGA bitstream. It is the first image loaded by the Zynq boot ROM.
* **Uimage** - The Linux kernel image in U-Boot–compatible format. It is the binary executed after U-Boot passes control to the kernel.
* **Boot.scr** - A compiled U-Boot script (boot.cmd → boot.scr) that contains the boot commands. It instructs U-Boot how to load the kernel, device tree, and root filesystem.

Together, these files form the minimal set required for booting the custom Linux system from the SD card.

# 7. IMAGE FLASHING

To flash the image onto the **SD** card, **Balena** **Etcher** was used. Simply select the generated .**wic** file, choose the target SD card, and click Flash. Wait for the process to complete before removing the card.

*Note*: Ensure the boot mode jumper is set to SD rather than **JTAG**; otherwise, the board will not boot from the SD card.



# 7. FUNCTIONALITY CHECK

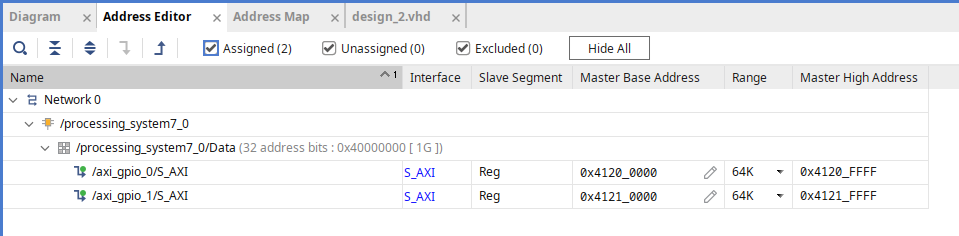
Once the board has booted and the login screen appears, log in as **root**. To verify that the **UIO driver** has been bound correctly, check for the presence of device files under **/dev/uio\***. These entries should exist if the configuration was done correctly.



After verifying the presence of the **/dev/uio\*** entries, you can inspect the details of a bound device (such as its name, base address, offset, and size) by reading from the sysfs interface:

cat /sys/class/uio/uio0/maps/

It is up to the user to determine which device corresponds to each UIO entry by comparing the reported base addresses with the address mapping defined in the Vivado project. These mappings can be found in the **Address Editor** section of **Vivado**.



There are two common methods to obtain the register mapping for each device:

1. Using **Vitis**:

* Create a new Vitis project and locate the header file corresponding to the device, e.g., **xgpio\_l.h.**
* This file defines the offsets for each register:

#define XGPIO\_DATA\_OFFSET 0x0 /\*\*< Data register for 1st channel \*/

#define XGPIO\_TRI\_OFFSET 0x4 /\*\*< I/O direction reg for 1st channel \*/

#define XGPIO\_DATA2\_OFFSET 0x8 /\*\*< Data register for 2nd channel \*/

#define XGPIO\_TRI2\_OFFSET 0xC /\*\*< I/O direction reg for 2nd channel \*/

#define XGPIO\_GIE\_OFFSET 0x11C /\*\*< Glogal interrupt enable register \*/

#define XGPIO\_ISR\_OFFSET 0x120 /\*\*< Interrupt status register \*/

#define XGPIO\_IER\_OFFSET 0x128 /\*\*< Interrupt enable register \*/

1. Using the **generated AXI implementation** **files** (for custom AXI IP only

* Open the generated VHDL implementation file for the custom IP
* There is usually a commented section that outlines each register offset and its function, which can be used to determine the mapping.

For standard IP blocks like AXI GPIO, the Vitis header file method is preferred as it is easier and more reliable. For custom AXI IP, the VHDL implementation file is often the only source of detailed register information.

# 7. APPLICATION CROSS COMPILATION

There are many ways to cross-compile an executable for the ARMv7 architecture, but this guide will focus on the method that requires the least effort. An alternative approach will be briefly mentioned for reference.

Cross compiling using the petalinux SDK

1. Navigate to your **PetaLinux project folder**.
2. Build the **SDK** by running:

petalinux-build –sdk

!Note: The build process can take a significant amount of time and may consume substantial system resources.

1. After the build completes, locate and run the generated **sdk.sh** script in **images/linux/**. You will be prompted to provide a path where the root filesystem and cross-compiler should be installed.
2. Once the installation finishes, the script will provide instructions on how to set up the cross-compilation environment. Follow these instructions to source the environment.
3. To verify the cross-compiler, run:

echo $CC

* The output should show the name of the cross-compilation toolchain, typically:

arm-linux-gnueabihf-gcc

when targeting an ARM processor.

An example application that uses a Makefile to build the executable can be found in the Github repository, inside the **gpio\_app folder**.

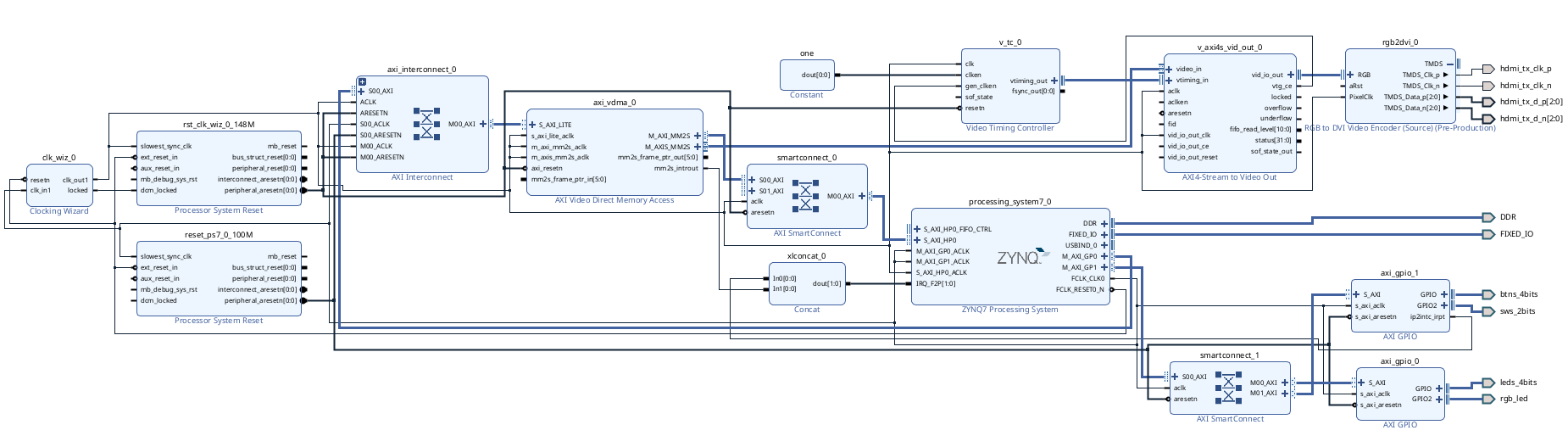
# 8. RESULTS

Finally, the application can be tested. Using mmap, the device memory is mapped to virtual memory, allowing access from user-space. By configuring the registers as shown in **Example 1** and writing to the data registers, the **LED** states can be controlled.

***Note:*** *As illustrated in the example and hardware design, no interrupt handling is implemented in this tutorial. Handling button interrupts and registering them will be covered in a future tutorial.*

Code example will come here

* Vivado design



**REFERENCES**

* to be added