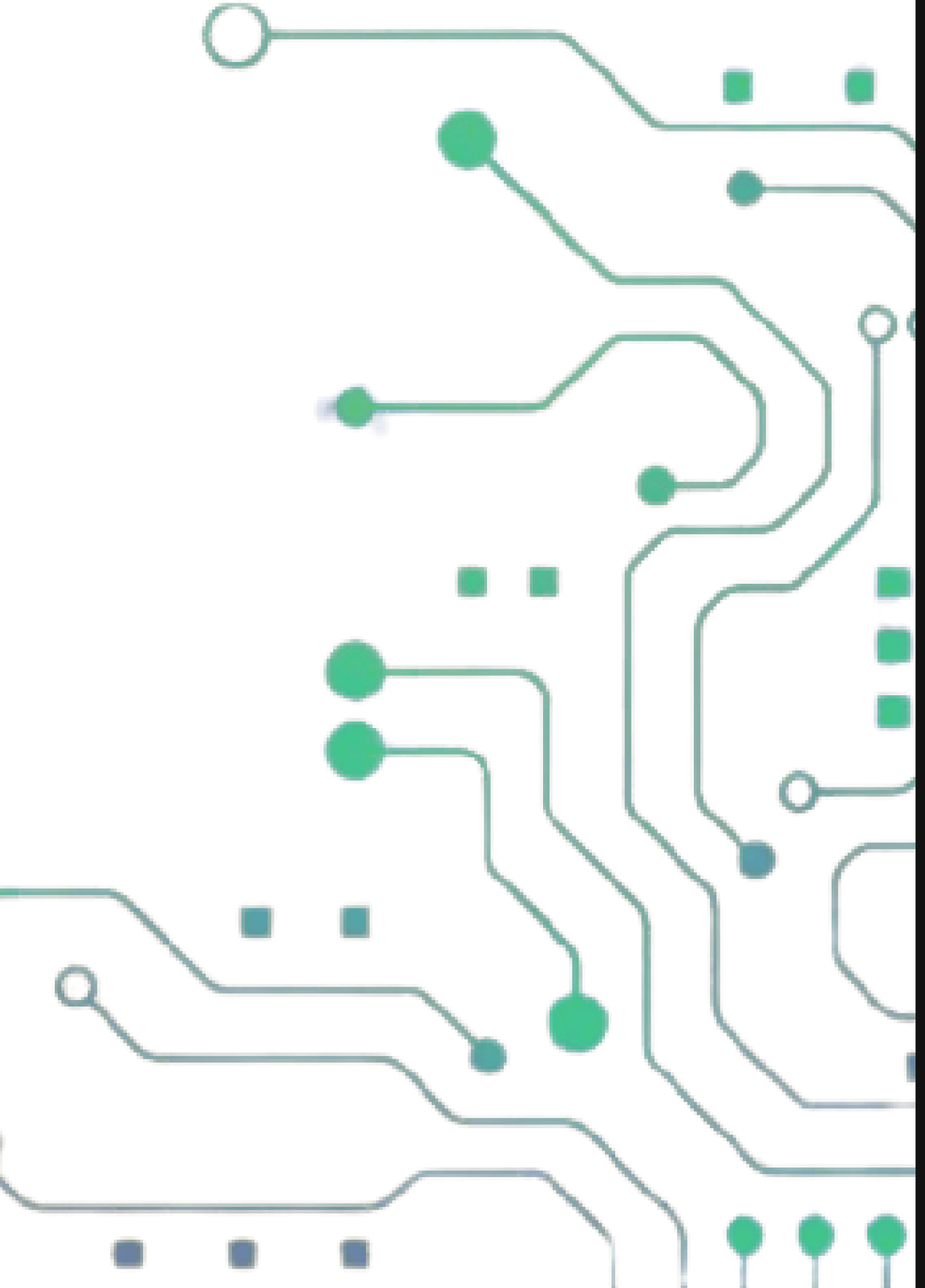


GROUP 02 - NOVEMBER 2023

AES Cryptocore

Operating Systems final project - type 3

Goals of the project

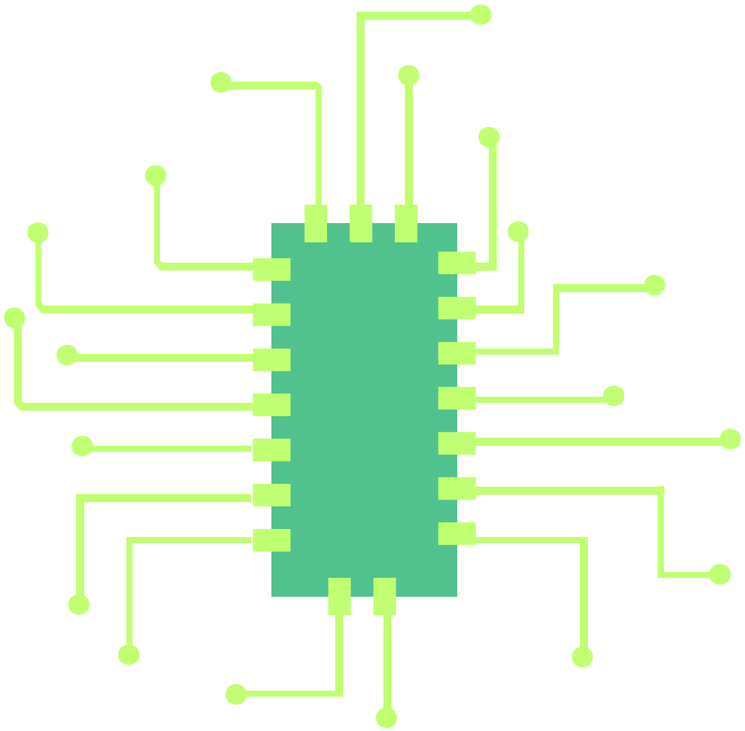
BUILD AN AES CRYPTOCORE INTO AN FPGA PLATFORM WITH A REALTIMEOS IN 3 MAIN STEPS:

- HARDWARE DEVELOPMENT & DEPLOYMENT
- OS IMAGE CREATION
- DRIVER DEVELOPMENT

First Part: HW Development and Deployment

We will use Vivado to instantiate our component (which is an AES cryptocore) inside the AXI4 Interface in order to communicate with the pynq-z2 board

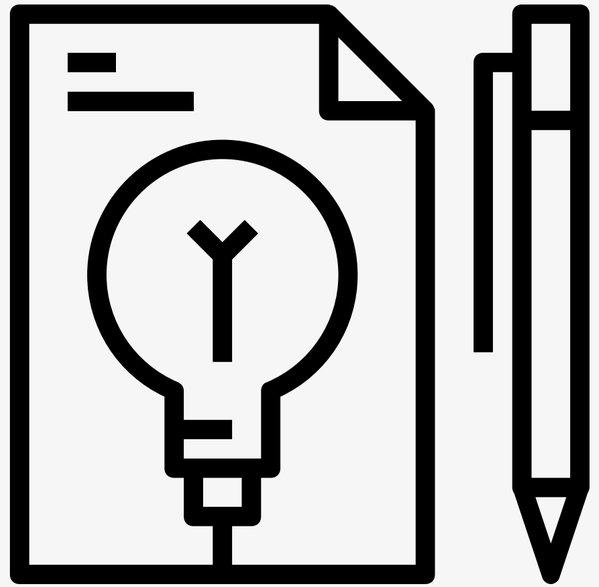
In the following slides it is reported the workflow to obtain a working Vivado project



How to set the core on the Pynq-z2 board

01 Download the core

02 Vivado IP and Project creation



03 VITIS

01 - Download the Core

Download the requested core from http://opencores.org/
It's mandatory to be registered with a student account in order to be able to download it. The confirmation of the registration could take some days. The core will be in VHDL or Verilog format. Both the HDL formats will work correctly with the following parts of the task.

02 - Vivado IP and Project Creation

We will use vivado to create the IP, including the core we need.

- Create a Vivado Project.
- Create new Project -> RTL project
- Then we choose the board, which is the Pynq-Z2.

NOTE: the board package needs to be included during the installation process of Vivado

- Then, clicking on Finish, the project is created.
- From the Tools menu, click on Create and package new IP, then Create AXI4 Peripheral

Pay attention to set light as interface type.

In this phase is important to specify the total number of registers needed in the IP (considering they have 32 bit, so signals with higher parallelism will take more than one register).

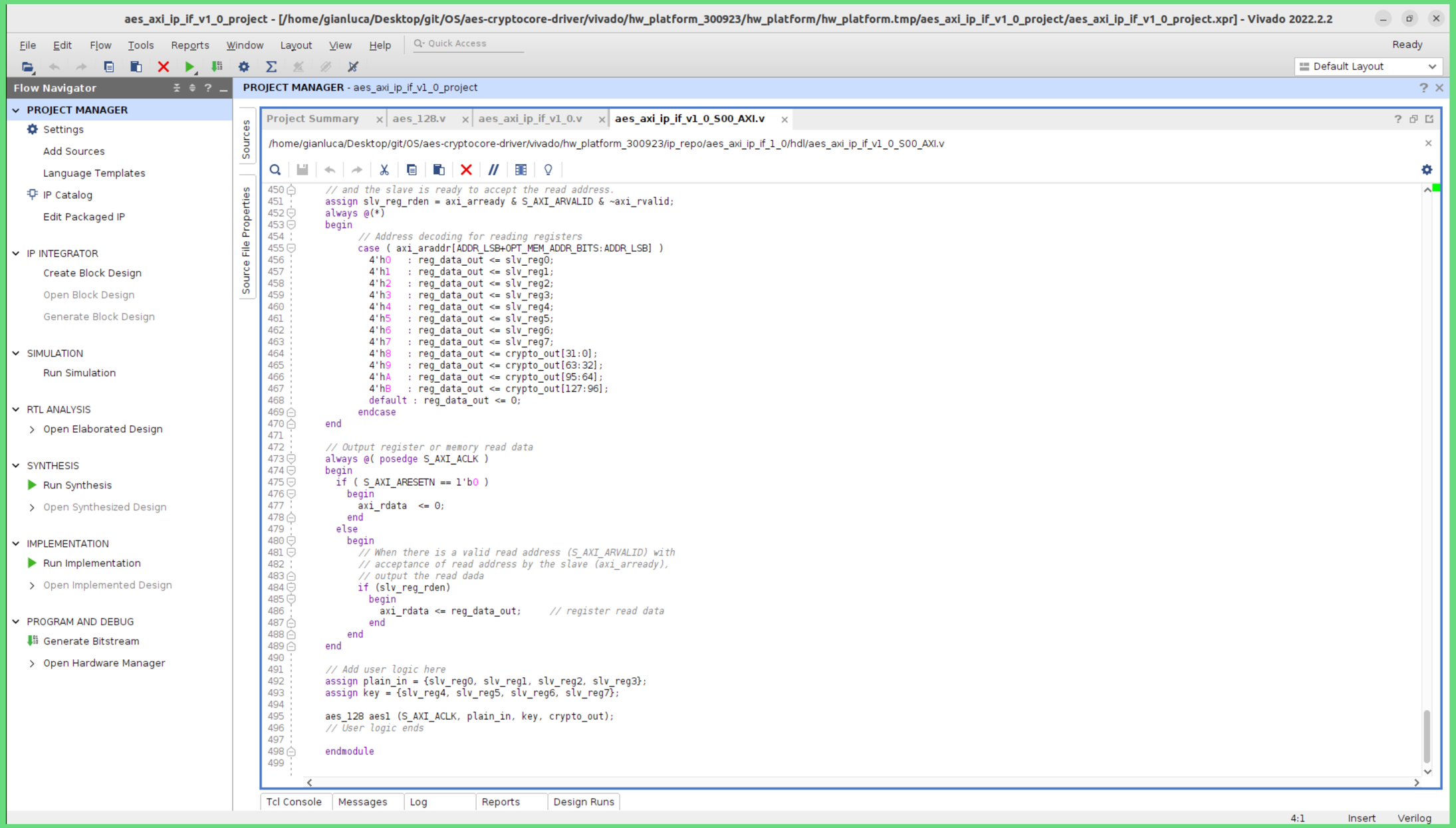
Each register will create an address (made by an offset with respect to a common base address) and each of these will be used only for an input or an output. The same address can't be used for more than one signal.

02 - Vivado IP and Project Creation

Core instantiation in the interface

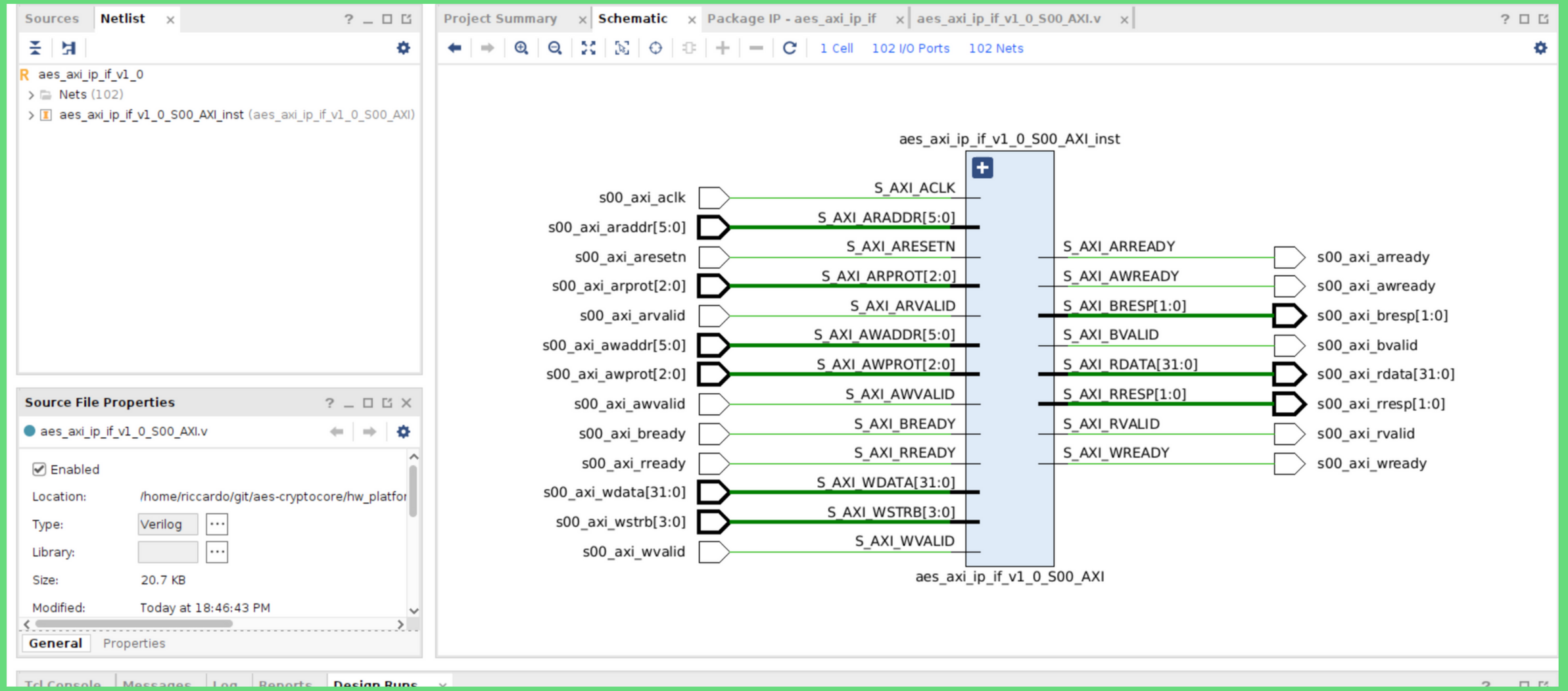
After having created the IP, click on the Edit IP choice.
Then from the Sources panel click on the + and add the HDL files for the core.
The core will be instantiated using these files as a component.
For interfacing properly the core through the AXI4 interface, it has to be instantiated and the ports have to be connected to the signals of the AXI4 that can be used as input/output. It can be done by writing HDL code into the verilog file of the IP (the file which name terminates with …SOO_AXI.vhd/.v)

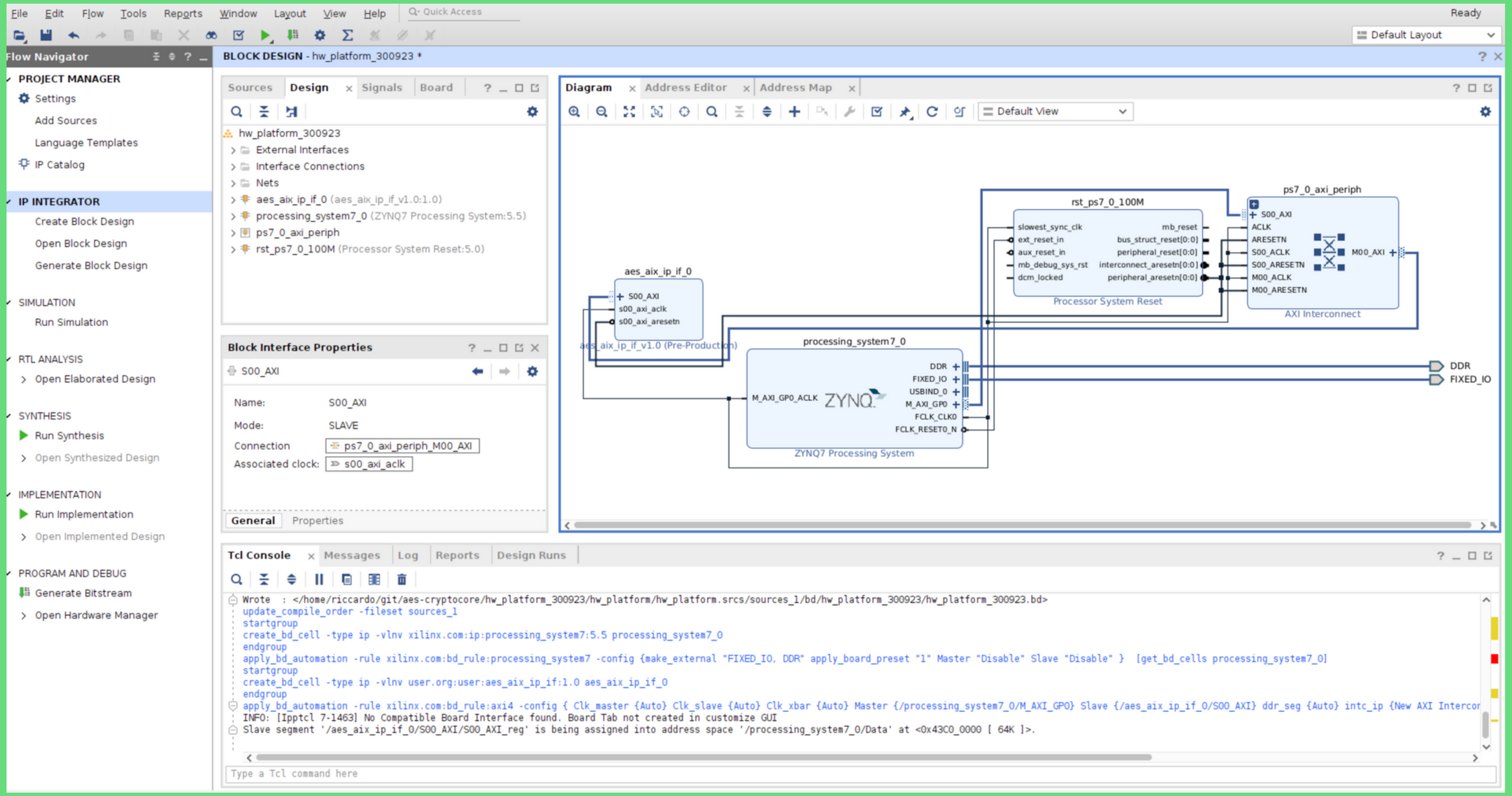
Core instantiation in the interface

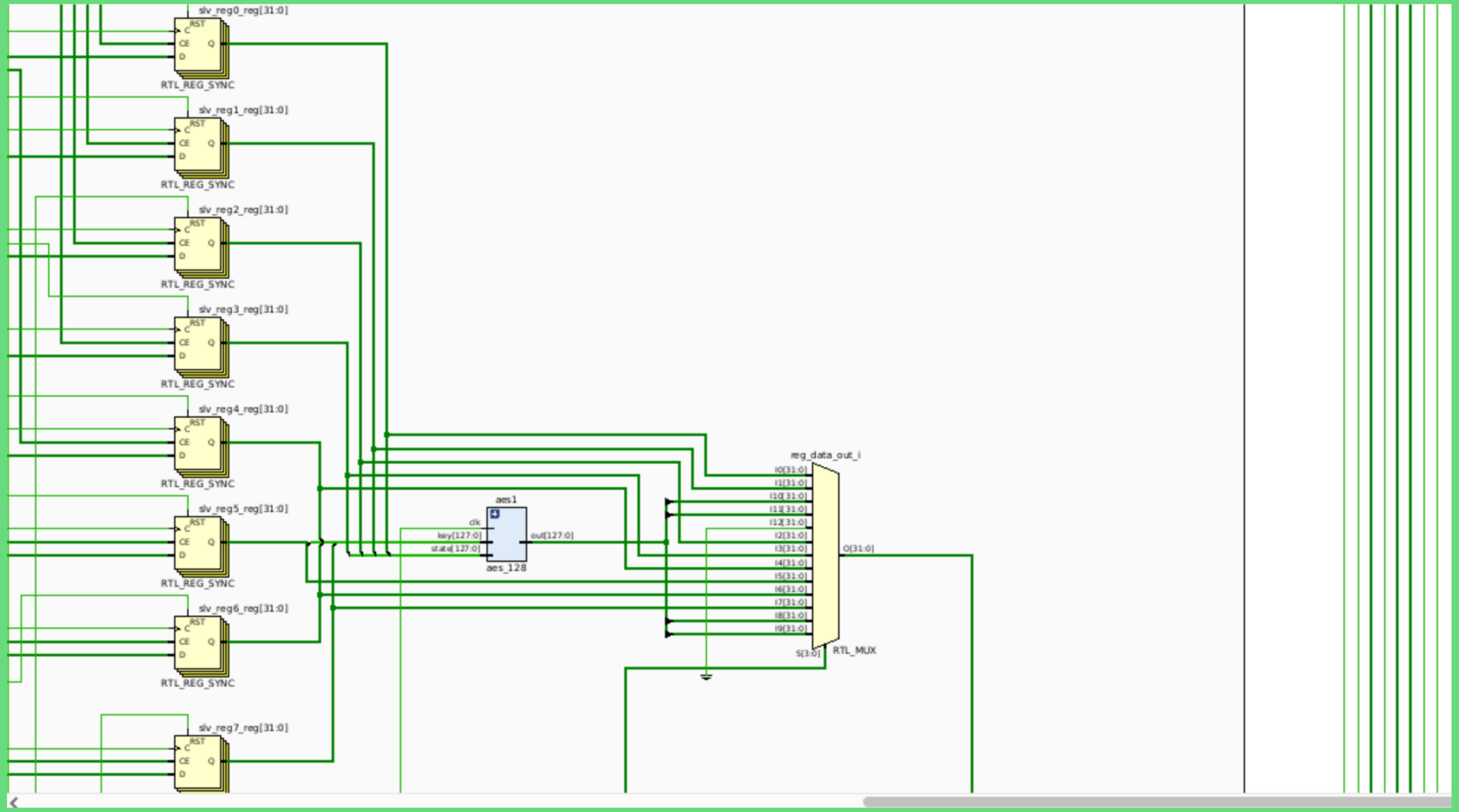


02 - Vivado IP and Project Creation

After having made the proper connections, click on Open elaborated design. This option will automatically allocate our core inside the interface in the way defined in the IP.
Now the schematics of the elaborated design are available.
It's important to verify from the schematics that all the connections have correctly been implemented, especially where there is a parallelism higher than 32 bits and so the data has to be managed using registers in parallel


Sources
Netlist
aes_axi_ip_if_v1_0
Nets (102)
aes_axi_ip_if_v1_0_S00_AXI_inst (aes_axi_ip_if_v1_0_S00_AXI)
Source File Properties
aes_axi_ip_if_v1_0_S00_AXI.v
Enabled
Location: /home/riccardo/git/aes-cryptocore/hw_platfor
Type: Verilog
Library:
Size: 20.7 KB
Modified: Today at 18:46:43 PM
General
Properties
Project Summary
Schematic
Package IP - aes_axi_ip_if
aes_axi_ip_if_v1_0_S00_AXI.v
1 Cell
102 I/O Ports
102 Nets
aes_axi_ip_if_v1_0_S00_AXI_inst
s00_axi_aclk
s00_axi_araddr[5:0]
s00_axi_aresetn
s00_axi_arprot[2:0]
s00_axi_arvalid
s00_axi_awaddr[5:0]
s00_axi_awprot[2:0]
s00_axi_awvalid
s00_axi_bready
s00_axi_rready
s00_axi_wdata[31:0]
s00_axi_wstrb[3:0]
s00_axi_wvalid
S_AXI_ACLK
S_AXI_ARADDR[5:0]
S_AXI_ARESETN
S_AXI_ARPROT[2:0]
S_AXI_ARVALID
S_AXI_AWADDR[5:0]
S_AXI_AWPROT[2:0]
S_AXI_AWVALID
S_AXI_BREADY
S_AXI_RREADY
S_AXI_WDATA[31:0]
S_AXI_WSTRB[3:0]
S_AXI_WVALID
S_AXI_ARREADY
S_AXI_AWREADY
S_AXI_BRESP[1:0]
S_AXI_BVALID
S_AXI_RDATA[31:0]
S_AXI_RRESP[1:0]
S_AXI_RVALID
S_AXI_WREADY
s00_axi_aresetn
s00_axi_awready
s00_axi_bresp[1:0]
s00_axi_bvalid
s00_axi_rdata[31:0]
s00_axi_rresp[1:0]
s00_axi_rvalid
s00_axi_wready
aes_axi_ip_if_v1_0_S00_AXI
Tcl Console
Messages
Log
Reports
Design Runs


File Edit Flow Tools Reports Window Layout View Help
Quick Access
Ready
Default Layout
Flow Navigator
PROJECT MANAGER
Settings
Add Sources
Language Templates
IP Catalog
IP INTEGRATOR
Create Block Design
Open Block Design
Generate Block Design
SIMULATION
Run Simulation
RTL ANALYSIS
Open Elaborated Design
SYNTHESIS
Run Synthesis
Open Synthesized Design
IMPLEMENTATION
Run Implementation
Open Implemented Design
PROGRAM AND DEBUG
Generate Bitstream
Open Hardware Manager
BLOCK DESIGN - hw_platform_300923 *
Sources Design Signals Board
hw_platform_300923
External Interfaces
Interface Connections
Nets
aes_aix_ip_if_0 (aes_aix_ip_if_v1.0:1.0)
processing_system7_0 (ZYNQ7 Processing System:5.5)
ps7_0_axi_periph
rst_ps7_0_100M (Processor System Reset:5.0)
Block Interface Properties
S00_AXI
Name: S00_AXI
Mode: SLAVE
Connection ps7_0_axi_periph_M00_AXI
Associated clock: s00_axi_aclk
General Properties
Diagram Address Editor Address Map
Default View
rst_ps7_0_100M
slowest_sync_clk
ext_reset_in
aux_reset_in
mb_debug_sys_rst
dcm_locked
mb_reset
bus_struct_reset[0:0]
peripheral_reset[0:0]
interconnect_aresetn[0:0]
peripheral_aresetn[0:0]
Processor System Reset
ps7_0_axi_periph
S00_AXI
ACLK
ARESETN
S00_ACLK
S00_ARESETN
M00_ACLK
M00_ARESETN
M00_AXI
AXI Interconnect
aes_aix_ip_if_0
S00_AXI
s00_axi_aclk
s00_axi_aresetn
aes_aix_ip_if_v1.0 (Pre-Production)
processing_system7_0
DDR
FIXED_IO
USBIND_0
M_AXI_GP0
FCLK_CLK0
FCLK_RESET0_N
M_AXI_GP0_ACLK
ZYNQ
ZYNQ7 Processing System
DDR
FIXED_IO
Tcl Console Messages Log Reports Design Runs
Wrote : </home/riccardo/git/aes-cryptocore/hw_platform_300923/hw_platform/hw_platform.srcs/sources_1/bd/hw_platform_300923/hw_platform_300923.bd>
update_compile_order -fileset sources_1
startgroup
create_bd_cell -type ip -vlnv xilinx.com:ip:processing_system7:5.5 processing_system7_0
endgroup
apply_bd_automation -rule xilinx.com:bd_rule:processing_system7 -config {make_external "FIXED_IO, DDR" apply_board_preset "1" Master "Disable" Slave "Disable" } [get_bd_cells processing_system7_0]
startgroup
create_bd_cell -type ip -vlnv user.org:user:aes_aix_ip_if:1.0 aes_aix_ip_if_0
endgroup
apply_bd_automation -rule xilinx.com:bd_rule:axi4 -config { Clk_master {Auto} Clk_slave {Auto} Clk_xbar {Auto} Master {/processing_system7_0/M_AXI_GP0} Slave {/aes_aix_ip_if_0/S00_AXI} ddr_seg {Auto} intc_ip {New AXI Intercor
INFO: [Ipptcl 7-1463] No Compatible Board Interface found. Board Tab not created in customize GUI
Slave segment '/aes_aix_ip_if_0/S00_AXI/S00_AXI_reg' is being assigned into address space '/processing_system7_0/Data' at <0x43C0_0000 [64K]>.
Type a Tcl command here


slv_reg0_reg[31:0]
RST
C
CE
D
Q
RTL_REG_SYNC
slv_reg1_reg[31:0]
slv_reg2_reg[31:0]
slv_reg3_reg[31:0]
slv_reg4_reg[31:0]
slv_reg5_reg[31:0]
slv_reg6_reg[31:0]
slv_reg7_reg[31:0]
aes1
clk
key[127:0]
state[127:0]
out[127:0]
aes_128
reg_data_out_i
I0[31:0]
I1[31:0]
I10[31:0]
I11[31:0]
I12[31:0]
I2[31:0]
I3[31:0]
I4[31:0]
I5[31:0]
I6[31:0]
I7[31:0]
I8[31:0]
I9[31:0]
O[31:0]
S[3:0]
RTL_MUX

02 - Vivado IP and Project Creation

Review, package, block design, wrapper

At this point, if everything corresponds to what is expected, click on the Package IP - <name> panel, and proceed clicking on File Groups -> Merge changes from File Groups Wizard.
Now click on Review and Package -> Re-Package IP
After having confirmed to close the project by clicking on Yes, next step is to create the block design
On the left menu: Create block design, the specify a name and click OK.
Now add the board processing system by clicking on the + symbol in the Diagram panel, and then selecting the board.
Click on Run Block Automation, then confirm with OK.

02 - Vivado IP and Project Creation

Review, package, block design, wrapper

Then add the custom IP made in the previous steps by clicking again on the + symbol, then Run Connection Automation and OK in the window that will appear.
Now validateby clicking on the blue check mark √ in Diagram panel, click OK.
Now it's time to create the wrapper that will be exported:
right click on the core name under the Design Sources section -> Create HDL Wrapper -> Let Vivado manage wrapper and auto-update.
Now the wrapper should appear in the Design Sources in the Sources panel.
Then on the left menu: Generate Bitstream, and OK in "Launch Runs" window.
Now File > Export > Export Hardware
In the window "Export Hardware Platform" :
Include bitstream, give the file name for the XSA and a path where to be exported.
Finish.

• EXERCISE 1: IP generation

Now it’s time to try to build your own HW description file!

Try to modify by yourself the hw description files of your system in order to communicate through the AXI4 interface. Instantiate your hw, the processor and the communication needed.

A solution for this exercise is inside the repository.

03 - VITIS - bare-metal test on the HW platform

- Open Vitis
- Create Application Project -> Next -> Create a new platform from hardware (XSA)

Here click on Browse... and select the wrapper file (.xsa).
Verify the platform name and proceed with Next.

- Select the ps7_cortexa9_0, then Next.
- In "select a domain" click on Create new, and in "Operating System" option select standalone Next -> select HelloWorld -> Finish

Now the project is ready for adding some user code

03 - VITIS - bare-metal test on the HW platform

- On the left menu, in src, then helloworld.c, read the code generated by Vitis.
- On the left menu, in hw, drivers, <your ip name>, src open the <your ip name>.h file and verify the generation of the correct offset constants for the address management. By scrolling down in the code, there are the methods for writing registers and reading registers, that can be used for our purposes.
- Be sure to have included libraries:

```
#include "platform.h"
#include "xil_io.h"
#include "<your ipname>.h""
#include "xparameters.h"
```

03 - VITIS - bare-metal test on the HW platform

Now we can use the _mWriteReg and _mReadReg methods to properly write and read data to/from the core.
Once finished, we can click on the build icon: 

IMPORTANT

From now on, the board should be properly connected to the PC.
Check if the jumper that allows to choose between JTAG and SD is in the correct position. Verify the correct behavior from the status LED on the board.

03 - VITIS - bare-metal test on the HW platform

The next step is to click on this icon:

in the upper toolbar.

This opens the menu Target connections, then in Hardware Server section you can click on Local [default] In the window that wil appear, click on OK with default values.

Now click on the Debug key on the upper toolbar:

This will program the FPGA and allow you to use the Vitis Serial Terminal (in the lower part of the screen), and add the correct port by clicking on the + symbol.
With the serial terminal, you can verify the output from the board, and see if it's the correct response to the data applied through the C program.

- # EXERCISE 2: Test your HW

Now it’s time to try to test your own HW description file!

Modify the helloworld.c template in Vitis in order to write and read on the registers you settled up in the previous exercise.

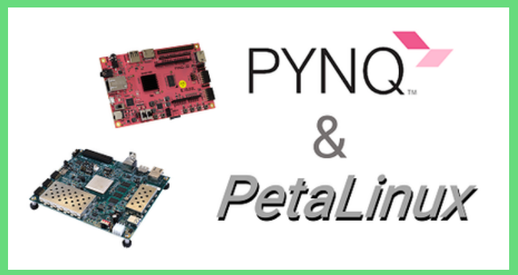
A solution for this exercise is inside the repository.

Second Part: OS IMAGE CREATION

In the second phase of our project, we utilize PetaLinux to create a customized Linux operating system image for our embedded system. This involves configuring the Linux kernel, building a root filesystem, and generating a bootable image specifically designed for our FPGA platform.

PETALINUX

- Optimized for FPGA and Zynq SoC
- Widely Adopted in Automotive,Aerospace,Telecommunications and Industrial Automation for Robust Embedded Systems.
- Seamless integration into hardware designs



Embedded Linux Installation Guide

1. **Download PetaLinux:**
   - Visit the following link to download the PetaLinux installer: PetaLinux Download
2. **Installation Steps:**

   First of all, make sure you have the latest version of the OS.

```
sudo apt update
sudo apt upgrade
```

Embedded Linux Installation Guide

- Then, after downloading, move the .run file to the desired installation folder.
- Open your terminal and navigate to the installation folder.
- Run the installer using the following command:

```
bash petalinux-v2023.1-05012318-installer.run
```

If an Error comes out saying you are missing some dependencies, run the following command:

```
sudo apt install gawk zlib1g-dev net-tools xterm autoconf libtool
texinfo gcc-multilib
```

Embedded Linux Installation Guide

Maybe they aren't all the packages and dependencies missing.
You can go at that link. By scrolling down, you can see a script called "plnx-evn-setup.sh". Download that script and run it. It should install all the packages needed. If is that not the case, check the error messages and search on google how to download that specific library by terminal. You will easily find a command to do it.

- Follow the on-screen prompts to accept the license agreements and complete the installation.
- Run the settings.sh script source ./settings.sh (ignore the warning)

Embedded Linux Installation Guide

Create the project

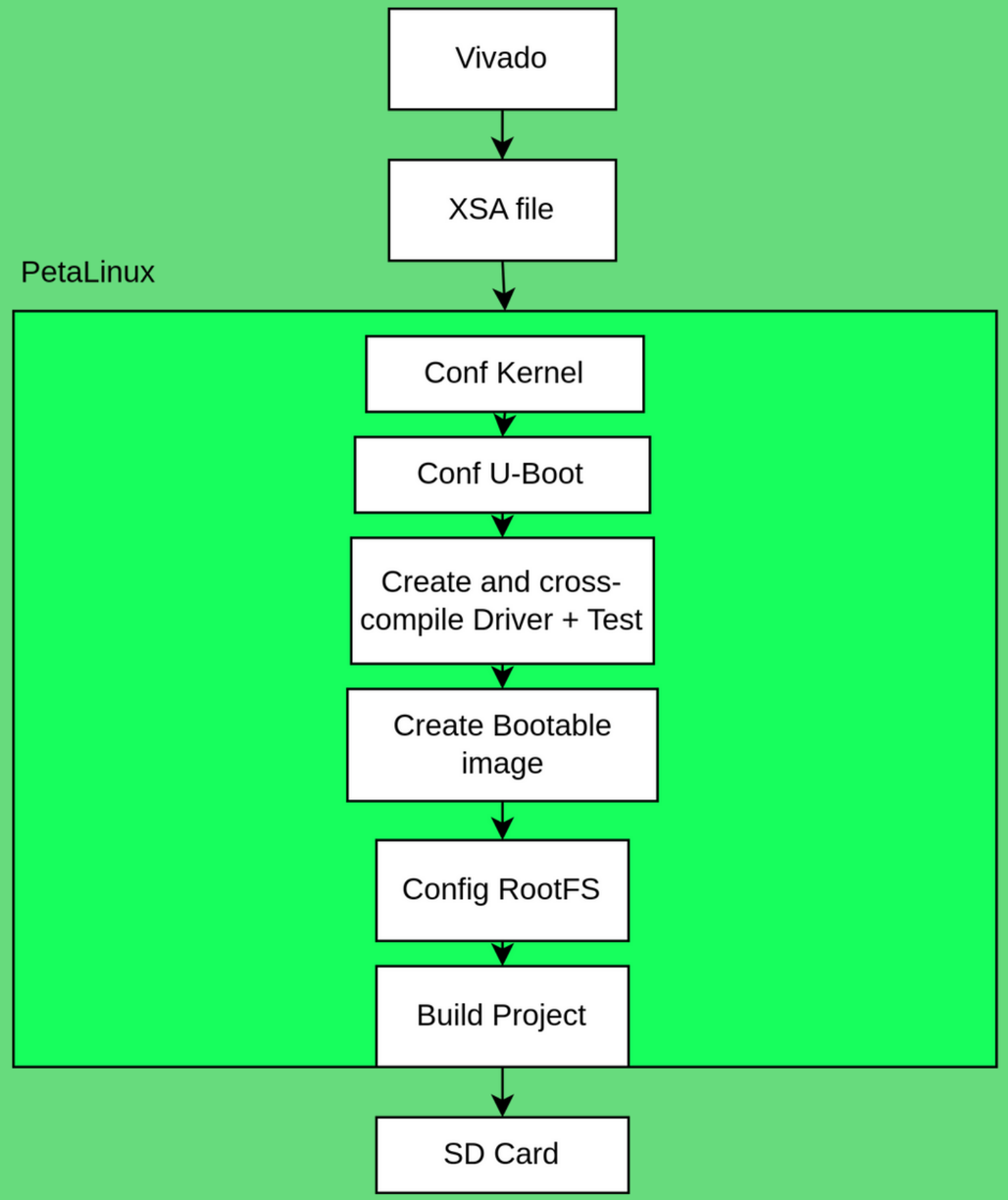
Go to your terminal and change directory to where you would like to create your new PetaLinux project directory

Run:

```
petalinux-create -t project --template zynq --name aes-petalinux
```

Here '-t' is equivalent to '--type'. You can see that there is a new folder created and you can enter inside it.

```
cd aes-petalinux
```


Vivado
XSA file
PetaLinux
Conf Kernel
Conf U-Boot
Create and cross-compile Driver + Test
Create Bootable image
Config RootFS
Build Project
SD Card

Embedded Linux Installation Guide

Configure using XSA file

You have previously created a XSA file as output of VIVADO that describes you hardware platform. You will need this file now for configure petalinux build and tell to petalinux what devices are available for it and it will create a Device Tree based on this hw definition file.

Write this command into the project directory

```
petalinux-config --get-hw-description <PATH-TO-XSA DIRECTORY>
```

This will open the first configuration screen of the petalinux tool. In this you can navigate and check what is inside, but there is no need to change anything here.You can exit by pressing double time the . Save it. This can needs some time.

Embedded Linux Installation Guide

Configure the Kernel

```
petalinux-config -c kernel
```

This will need some time. After that, a similar window to the previous one should open. As before, there are many options, you can browser it for checking if you have to change something. Also here, there is no need to change anything for the purpose of this Lab.

Embedded Linux Installation Guide

Configure U-Boot

Another configuration command we have to run

```
petalinux-config -c u-boot
```

In the configuration window, you have to enable the boot options --> boot media --> "Support for booting from QSPI flash" and "Support for booting from SD/EMMC".

Save and exit.

Now there will be another configuration command needed, the one for the RootFS. But first, in order to compile it correctly for this lab, you want to create an application and a module.

Create and compile the driver

The skeleton of the driver that will contain your custom code can be created by terminal with this command.

```
petalinux-create -t modules --name aes-core-driver --enable
```

You can check that a new file it is created. The file is in

```
$(PETAPROJECT)/project-spec/meta-user/recipes-modules/aes-coredriver/files/aes-core-driver.c
```

There you can customize the init, write, read, open, close functions to your needs.
Once you are done you can build it with petalinux (cross-compile it)

```
petalinux-build -c aes-core-driver
```

Thanks to that command and to nexts that we will see, the driver will be already mounted on the device (i.e. the *.ko file).

Creating and Compiling the test- driver

You can also create an application that tests your driver using the petalinux tools and avoiding also here to crosscompile it by yourself.

```
petalinux-create -t apps --template c --name aes-core-test --enable
```

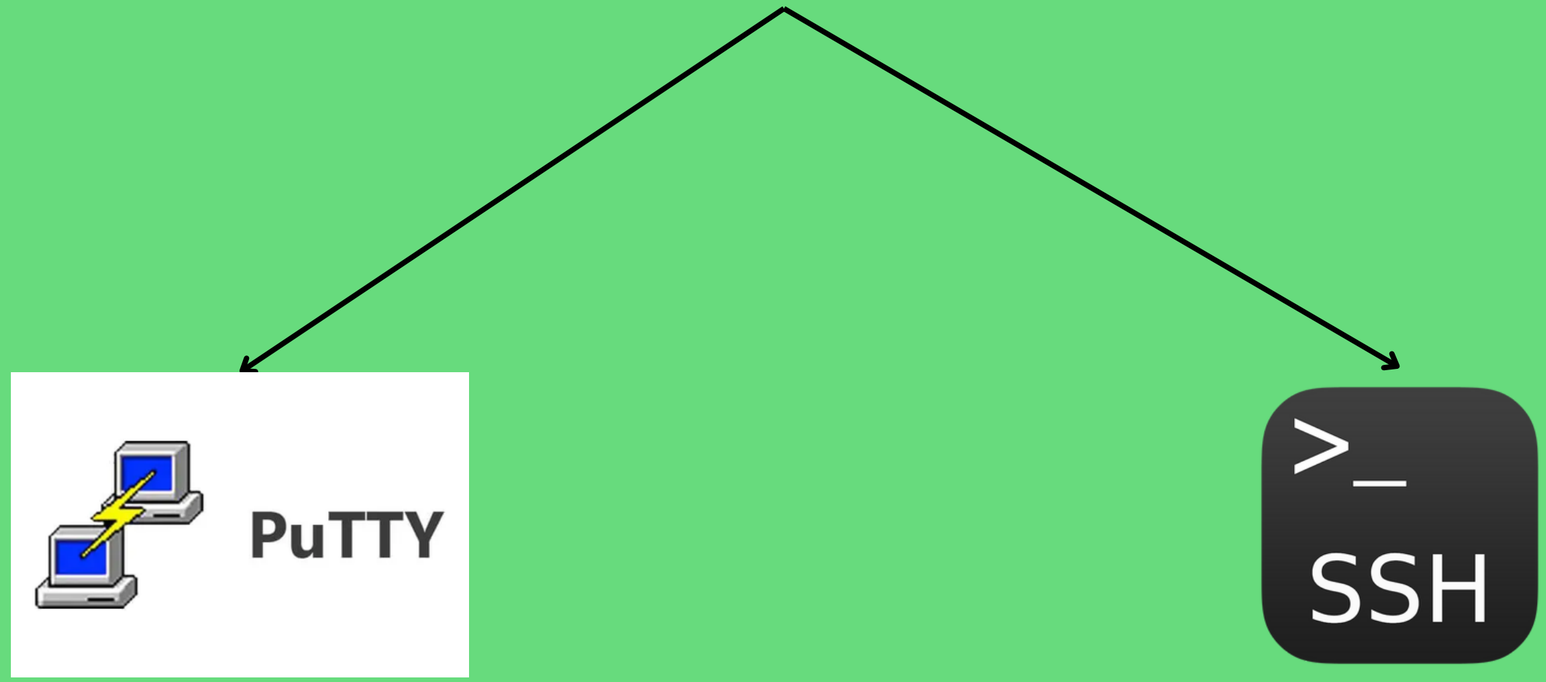
The new file created is in

```
$(PETAPROJECT)/project-spec/meta-user/recipes-apps/aes-core-test
/files/aes-core-test.c
```

There you can customize the C file to your testing needs. Once you are done you can build it with petalinux

```
petalinux-build -c aes-core-test
```

How do we interface with the OS ?



- # EXERCISE 3: Build your linux system!

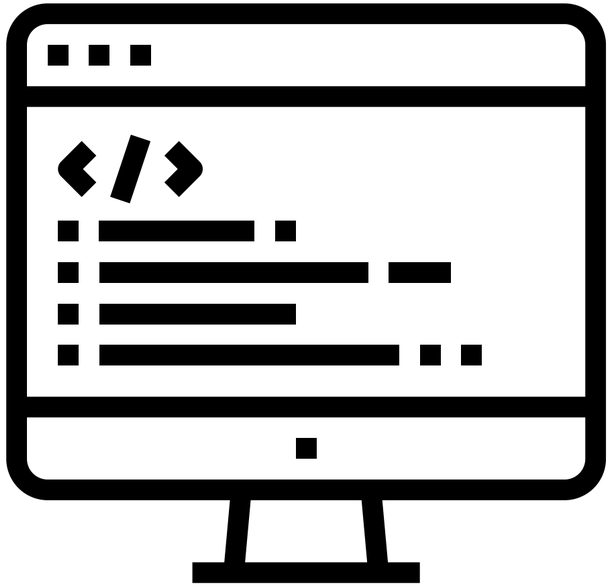
Now it’s time to try to test what you learned in the previous steps.

Try to follow the guide in the repository and build your linux system.

Flash the system through JTAG and then SD card

Third Part: Driver development

The main goal of this LAB is to write a crypto-core driver in C language. Now, you can do it from scratch and try to cross-compile this driver for the system that you are building. This is not easy. Fortunately, Petalinux comes to help us. Indeed, in order to cross-compile the driver for the cripto-core easily, we will use the petalinux's recipes and buildtools.



Structure of the driver

#define

Here there are the constant values used in our IP to access registers, according to the right addresses

```
#define BASE_ADDR                       (0x43C00000U)

#define OFFSET_STATUS_0         (0x00000000U)
#define OFFSET_STATUS_1         (0x00000004U)
#define OFFSET_STATUS_2         (0x00000008U)
#define OFFSET_STATUS_3         (0x0000000CU)

#define OFFSET_KEY_0            (0x00000010U)
#define OFFSET_KEY_1            (0x00000014U)
#define OFFSET_KEY_2            (0x00000018U)
#define OFFSET_KEY_3            (0x0000001CU)

#define OFFSET_OUTPUT_0         (0x00000020U)
#define OFFSET_OUTPUT_1         (0x00000024U)
#define OFFSET_OUTPUT_2         (0x00000028U)
#define OFFSET_OUTPUT_3         (0x0000002CU)

#define DEST_ADDR(x,y)      (uint8_t*)((uint8_t)(x) + (uint8_t)(y))
```

Structure of the driver

aes_core_driver_probe():

This function is responsible of mapping the memory according to our hardware, thus checking possible incorrect addresses

```
static int aes_core_driver_probe(struct platform_device *pdev)
{
        struct resource *r_irq; /* Interrupt resources */
        struct resource *r_mem; /* IO mem resources */
        struct device *dev = &pdev->dev;
        struct aes_core_driver_local *lp = NULL;

        int rc = 0;
        dev_info(dev, "Device Tree Probing\n");
        /* Get iospace for the device */
        r_mem = platform_get_resource(pdev, IORESOURCE_MEM, 0);
        if (!r_mem) {
                dev_err(dev, "invalid address\n");
                return -ENODEV;
        }
        lp = (struct aes_core_driver_local *) kmalloc(sizeof(struct aes_core_driver_local), GFP_KERNEL);
        if (!lp) {
                dev_err(dev, "Cound not allocate aes-core-driver device\n");
                return -ENOMEM;
        }
        dev_set_drvdata(dev, lp);
        lp->mem_start = r_mem->start;
        lp->mem_end = r_mem->end;

        if (!request_mem_region(lp->mem_start,
                                lp->mem_end - lp->mem_start + 1,
                                DRIVER_NAME)) {
                dev_err(dev, "Couldn't lock memory region at %p\n",
                        (void *)lp->mem_start);
```

Structure of the driver

dev_read():

This function is responsible of reading the buffer using the appropriate offset, thus reporting cases of error or succes

```
static ssize_t dev_read
(
        struct file *fil,
        char *buf,
        size_t len,
        loff_t *off
)
{
        uint8_t err = 0U;

        printk(KERN_ALERT "Device starting to read");

        local_buf = readl(DEST_ADDR(vi_baddr, *off));

        err = copy_to_user(buf, &local_buf, sizeof(buf));
        if (err != 0U)
        {
                printk(KERN_ALERT "ERROR_R: impossible to copy to user space");
        }

        printk(KERN_ALERT "Succesfully read: data %0X", local_buf);

        return 0;
}
```

Structure of the driver

dev_write():

This function is responsible of writing the buffer, provided the correct offset, thus it is checking for possible offsets out of the correct range

```
static ssize_t dev_write
(
        struct file *fp,
        const char *user_buf,
        size_t len,
        loff_t *off
)
{
        uint8_t  err        = 0U;


        printk(KERN_ALERT "device starting to write");
        printk(KERN_ALERT "Write function.\nVirtual address = %08X\n local_buf = %08X\n", *vi_baddr, local_buf);
        printk(KERN_ALERT "user_buf = %08X\n", *user_buf);


        if((*off % 4) != 0 || *off > 16)
        {
                printk(KERN_INFO"Offset out of range\n");
                return -1;
        }

        err = copy_from_user(&local_buf, user_buf, sizeof(user_buf));
        if (err != 0U)
        {
                printk(KERN_ALERT "ERROR_W: impossible to copy from user space");
        }


        writel(local_buf,DEST_ADDR(vi_baddr,*off));

        printk(KERN_ALERT "Succesfully write: data %s", local_buf);
```

Structure of the driver

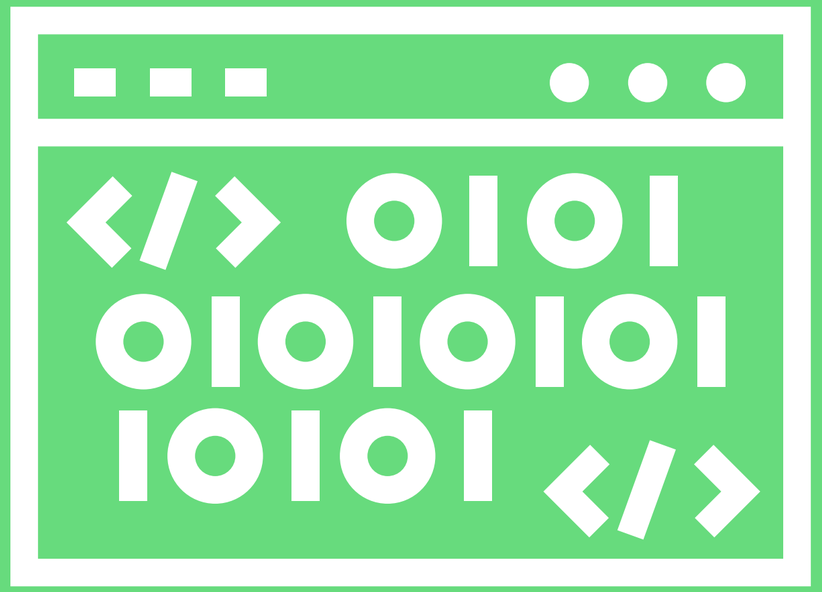
Open, close and exit functions

```
static int dev_close(struct inode *inod, struct file *fil){
        printk(KERN_ALERT "device closed\n");
        return 0;
}


static void __exit aes_core_driver_exit(void)
{
        platform_driver_unregister(&aes_core_driver_driver);
        printk(KERN_ALERT "Goodbye module world.\n");
}

static int dev_open(struct inode *inod, struct file *fil){
        printk(KERN_ALERT "device opened");

        printk(KERN_ALERT "Open function.\nVirtual address = %08X\n local_buf = %08X\n", *vi_baddr, local_buf);
        return 0;
}
```

Final exercise: The driver

https://github.com/EMNESS-project-group2

Inside the above GIT repo, you can find all the reference code, with more technical detail



Now It’s up to you to develop a working driver with a c test file to check its functionality