EPFL

EMBEDDED SYSTEMS LABORATORY

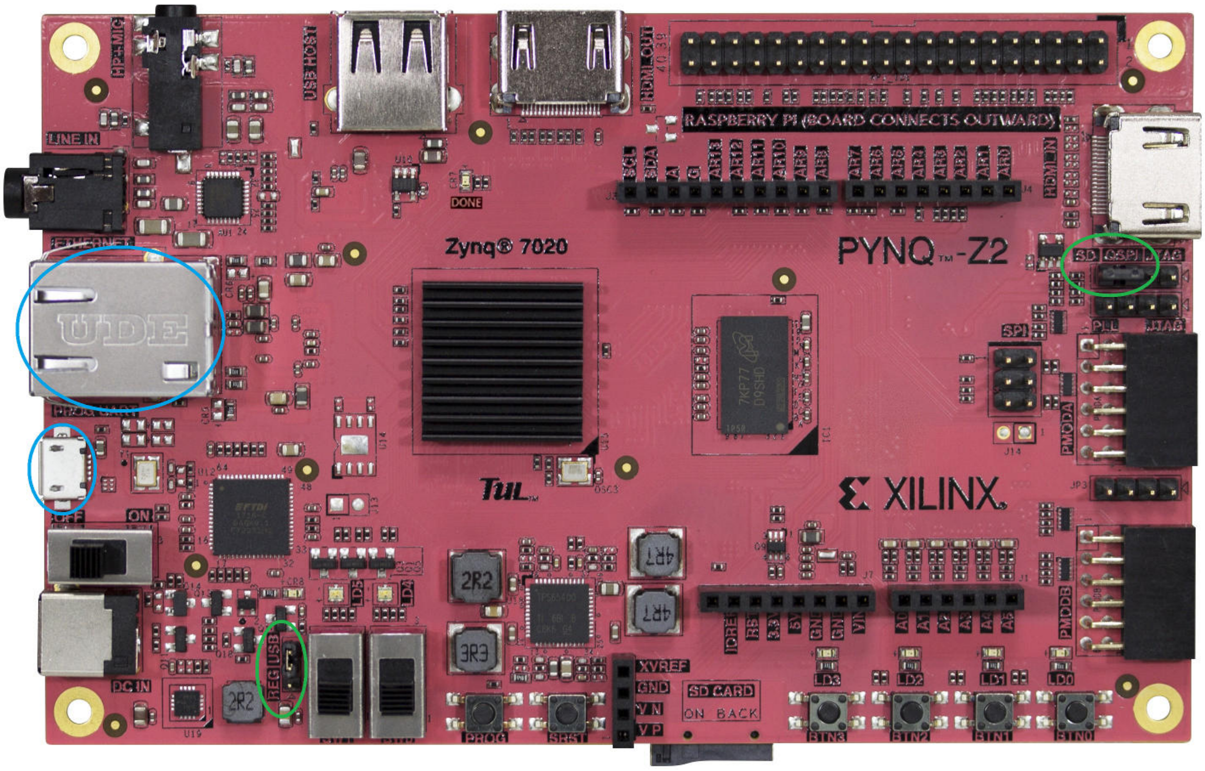
Lab on HW-SW digital systems codesign
SEL B.Sc. Curriculum (Spring Semester)
Embedded Systems Laboratory
EPFL-STI-IEL

Exercises Session 1
Out: 2025-02-20

Setting up Linux on the Pynq board. Basic peripherals in a SoC with Linux

1 Basic board and PC setup

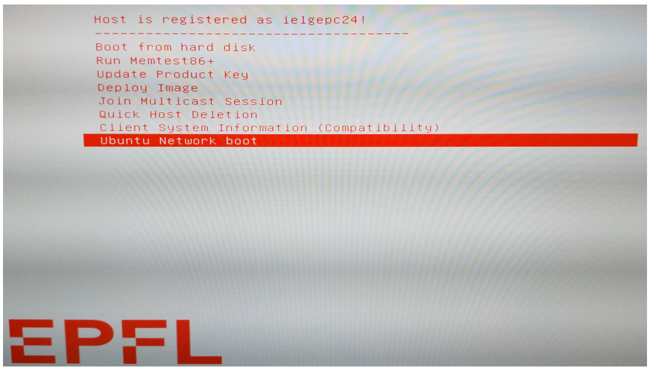
During this course, we will use the TUL Pynq-Z2 board, shown in the figure below. Before using the board for the first time, check that the jumpers are in the correct position:



The position of the jumpers for boot selection and power source is indicated with green circles. The ports for the micro USB and Ethernet cables are indicated by blue circles.

- The boot selection jumper must be in the position “SD”.
- The power source jumper must be in the position “USB”.

During this course, we will use Xilinx Vivado 2022.2 on Linux computers. To start Vivado 2022.2, turn on the computer and select the option “Ubuntu Network boot” in the boot menu:



```
Host is registered as ielgepc24!
--------------------------------------------
Boot from hard disk
Run Memtest86+
Update Product Key
Deploy Image
Join Multicast Session
Quick Host Deletion
Client System Information (Compatibility)
Ubuntu Network boot
```

Log into the machine with your Gaspar credentials, open a terminal and start Vivado with the following command:

```
$ /scrap/users/Xilinx2022.2/Vivado/2022.2/bin/vivado &
```

The & at the end of the command opens the program in the background so that you can continue using the terminal.

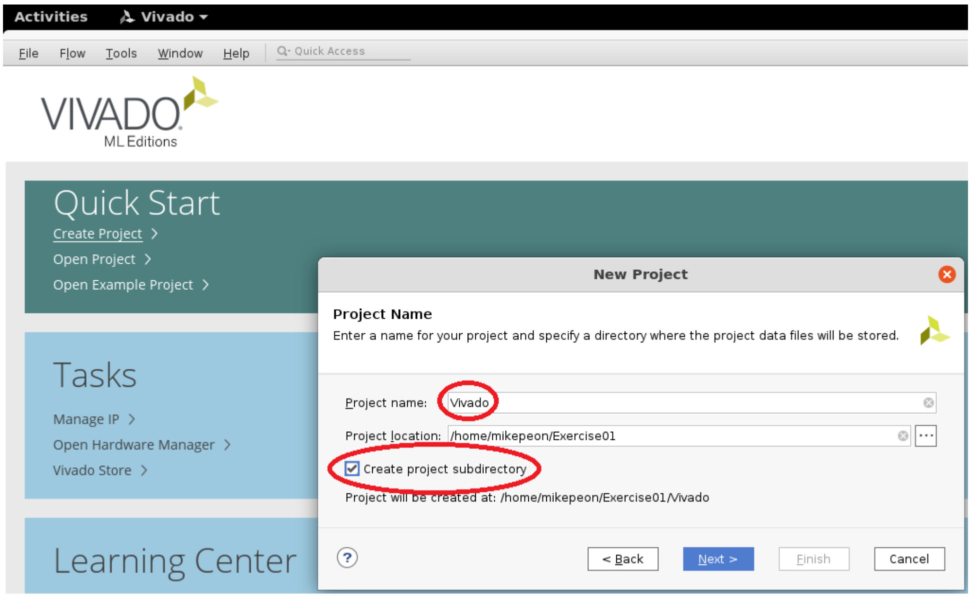
2 Exercise 1: Simple VHDL peripheral

This example serves the purpose of verifying that the board is in correct state and that it can be correctly programmed. We will create a simple design that connects the board switches directly to the LEDs.

→ Please, ensure that the SD card is not in the slot for this experiment. ←

2.1 Building the HW project

Create a new project and select the destination folder. Name the project “Vivado” and select the option “Create project subdirectory,” as seen in the figure:[1]



New Project

Project Name

Enter a name for your project and specify a directory where the project data files will be stored.

Project name: Vivado

Project location: /home/mikepeon/Exercise01

☑ Create project subdirectory

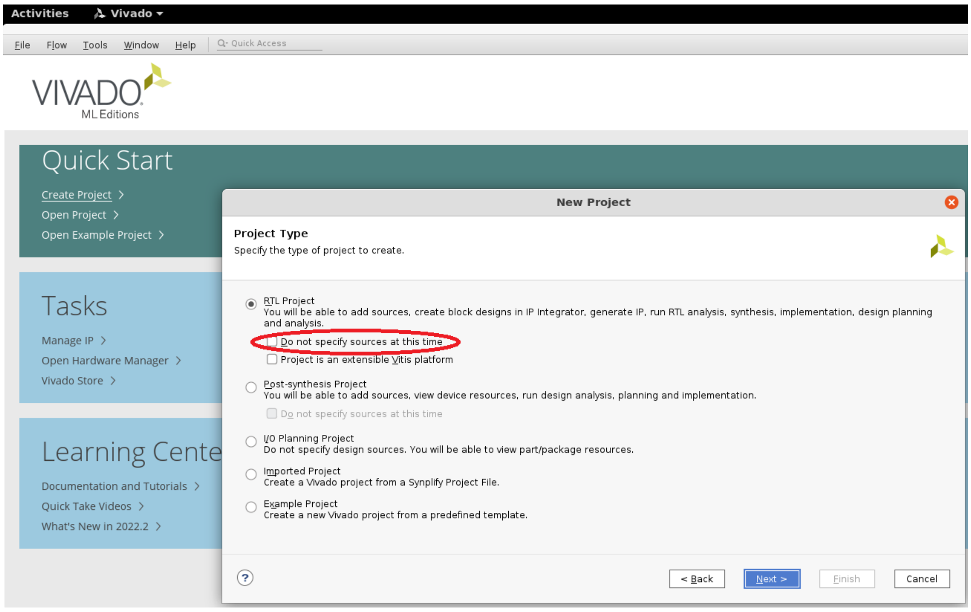
Project will be created at: /home/mikepeon/Exercise01/Vivado

We will create a structure for the projects with separated folders for HDL and HLS sources, test benches, and for the Vivado project itself. In future sessions we will see how to export the Vivado project to a TCL script and recreate it from that file. This will allow us to reduce significantly the size of the projects when we want to save or upload them to a repository. We will have a folder structure similar to this:

[1] Do not use spaces in the path or name of your projects.

```
/Exercise01
*build_project.sh      --- Bash script to call the TCL script
*Exercise01.tcl        --- TCL script to recreate the Vivado project
|-->HDL                --- Contains your VHDL/Verilog sources
|-->HLS                --- Contains your HLS sources
|-->SW                 --- Contains the sources for the SW project in Linux
|-->Vivado             --- Contains the Vivado project and can be deleted
```

In the next dialog box, select "RTL project" and clear the mark "Do not specify sources at this time." This will allow us to select any HDL sources that we might have already written.



New Project

Project Type
Specify the type of project to create.

● RTL Project
You will be able to add sources, create block designs in IP Integrator, generate IP, run RTL analysis, synthesis, implementation, design planning and analysis.
☐ Do not specify sources at this time
☐ Project is an extensible Vitis platform

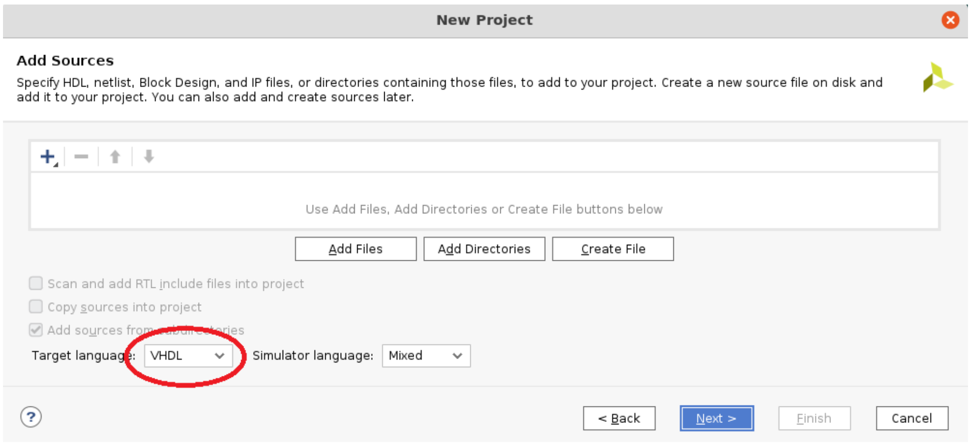
○ Post-synthesis Project
You will be able to add sources, view device resources, run design analysis, planning and implementation.
☐ Do not specify sources at this time

○ I/O Planning Project
Do not specify design sources. You will be able to view part/package resources.

○ Imported Project
Create a Vivado project from a Synplify Project File.

○ Example Project
Create a new Vivado project from a predefined template.

In the following dialog, select VHDL as target language. If this step is omitted, it is still possible to add both VHDL or Verilog files to the project. However, the tool uses this selection to determine the language in which it will produce automatically generated files (e.g., wrapper and simulation files). It is also possible to change the default project language in the project settings dialog.



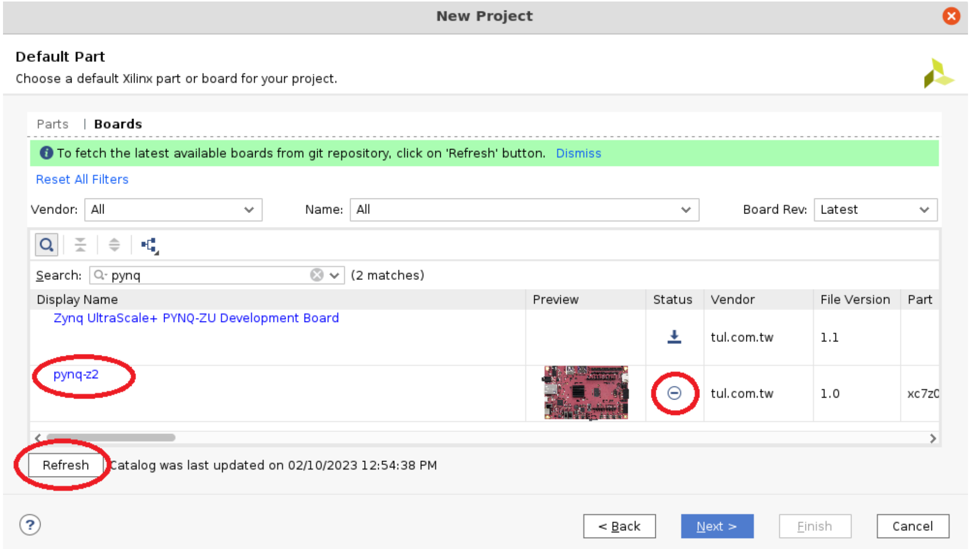
New Project

Add Sources
Specify HDL, netlist, Block Design, and IP files, or directories containing those files, to add to your project. Create a new source file on disk and add it to your project. You can also add and create sources later.

Use Add Files, Add Directories or Create File buttons below

☐ Scan and add RTL include files into project
☐ Copy sources into project
☑ Add sources from subdirectories
Target language: VHDL   Simulator language: Mixed

Now, we can select the board for our project. First, select the option "Board" rather than "Parts" in the dialog. The first time that Vivado is started by a user, it is necessary to refresh the board catalog to download the files for our board. Simply press the button "Refresh" and wait for the refresh process to complete. Then, write the name "pynq" or scroll in the updated list to select the "pynq-z2". Also only the first time that Vivado is started, it is necessary to download the specific files for the board. Click on the download arrow in the "Status" column and wait for the download to complete. Finally, select the row and click "Next."



New Project

Default Part

Choose a default Xilinx part or board for your project.

To fetch the latest available boards from git repository, click on 'Refresh' button. Dismiss

Vendor: All

Name: All

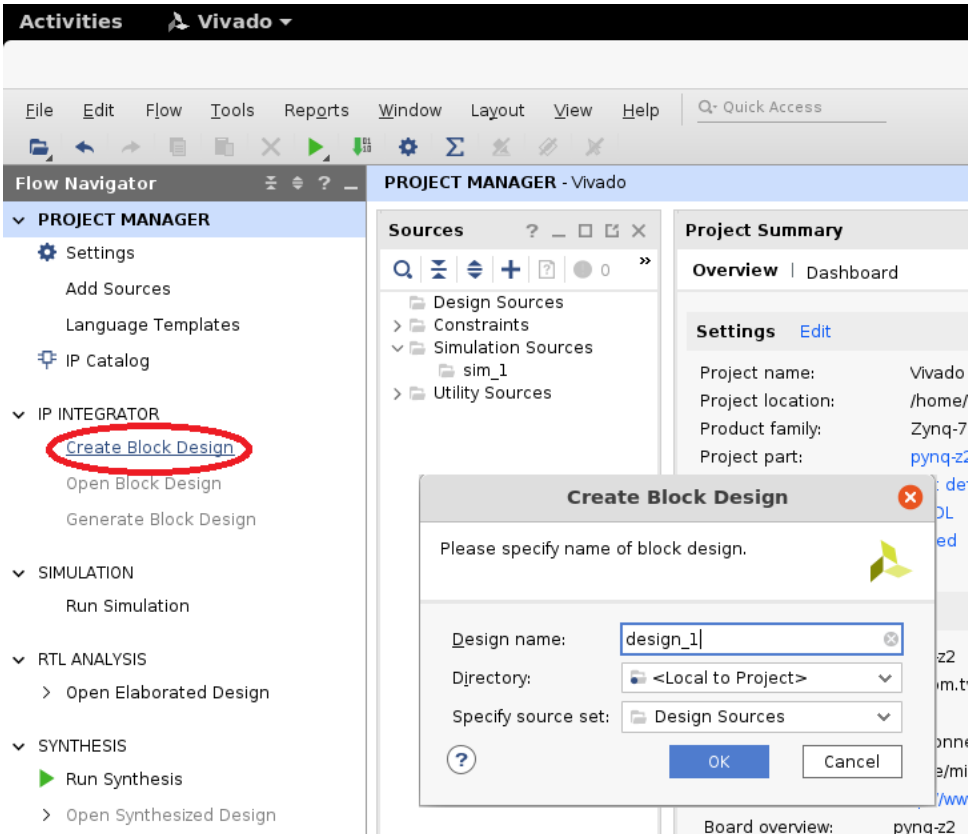
Board Rev: Latest

Search: pynq (2 matches)

Display Name	Preview	Status	Vendor	File Version	Part
Zynq UltraScale+ PYNQ-ZU Development Board			tul.com.tw	1.1	
pynq-z2			tul.com.tw	1.0	xc7z0

Catalog was last updated on 02/10/2023 12:54:38 PM

The main project window will appear. On the left pane, we have a list of actions that we can perform during the project lifetime. Click on the line "Create Block Design" to create the main schematic of the project:



Create Block Design

Please specify name of block design.

Design name: design_1

Directory: <Local to Project>

Specify source set: Design Sources

Once the schematic is created, we have an empty block diagram in which we can add components. First, we are going to add our HDL sources to the project. Create a folder named "HDL" according to the mentioned folder structure and create inside a file named "buttons2leds.vhdl" with the VHDL code for this example:[2]

```
library ieee;
use ieee.std_logic_1164.ALL;
use ieee.numeric_std.ALL;
```

[2] The source code can be downloaded from Moodle.

```
entity Buttons2Leds is
  port (
    btns : in std_logic_vector(3 downto 0);
    leds : out std_logic_vector(3 downto 0)
  );
end Buttons2Leds;

library ieee;
use ieee.std_logic_1164.ALL;
use ieee.numeric_std.ALL;

architecture rtl of Buttons2Leds is
begin
  leds <= btns;
end rtl;
```

Create also a *constraints file* named "`pynqz2.xdc`". We will use it to specify the physical location of the design ports:

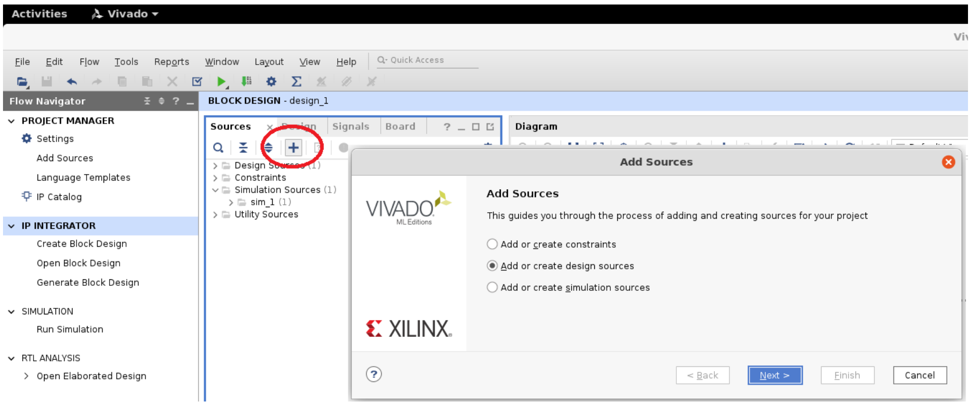
```
## This file is a general .xdc for the PYNQ-Z2 board
## To use it in a project:
## - uncomment the lines corresponding to used pins
## - rename the used ports (in each line, after get_ports) according to the top level signal names
##   in the project

## Clock signal 125 MHz
#set_property -dict { PACKAGE_PIN H16 IOSTANDARD LVCMOS33 } [get_ports { sysclk }];
#create_clock -add -name sys_clk_pin -period 8.00 -waveform {0 4} [get_ports { sysclk }];

#LEDs
set_property -dict { PACKAGE_PIN R14 IOSTANDARD LVCMOS33 } [get_ports { leds[0] }];
set_property -dict { PACKAGE_PIN P14 IOSTANDARD LVCMOS33 } [get_ports { leds[1] }];
set_property -dict { PACKAGE_PIN N16 IOSTANDARD LVCMOS33 } [get_ports { leds[2] }];
set_property -dict { PACKAGE_PIN M14 IOSTANDARD LVCMOS33 } [get_ports { leds[3] }];

#Buttons
set_property -dict { PACKAGE_PIN D19 IOSTANDARD LVCMOS33 } [get_ports { btns[0] }];
set_property -dict { PACKAGE_PIN D20 IOSTANDARD LVCMOS33 } [get_ports { btns[1] }];
set_property -dict { PACKAGE_PIN L20 IOSTANDARD LVCMOS33 } [get_ports { btns[2] }];
set_property -dict { PACKAGE_PIN L19 IOSTANDARD LVCMOS33 } [get_ports { btns[3] }];
```

Then, click on the *plus* sign in the sources pane to add our sources. Select "Add or create design sources" first to add the VHDL file and repeat the operation selecting "Add or create constraints" to add the constraints file.



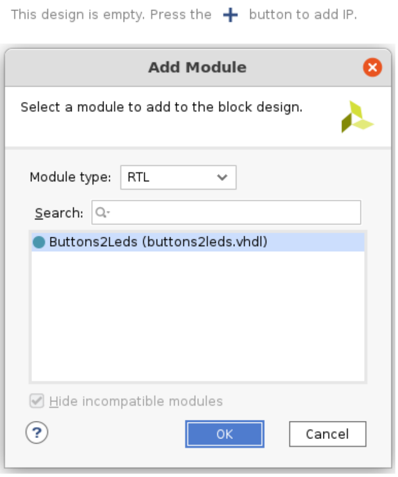
Add Sources

Add Sources

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

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

Once the sources have been added to the project, right-click on the schematic and select the option "Add Module." A new dialog box will open with a list of RTL modules in the project. Select the component `Buttons2Leds` from our VHDL file:



This design is empty. Press the + button to add IP.

Add Module

Select a module to add to the block design.

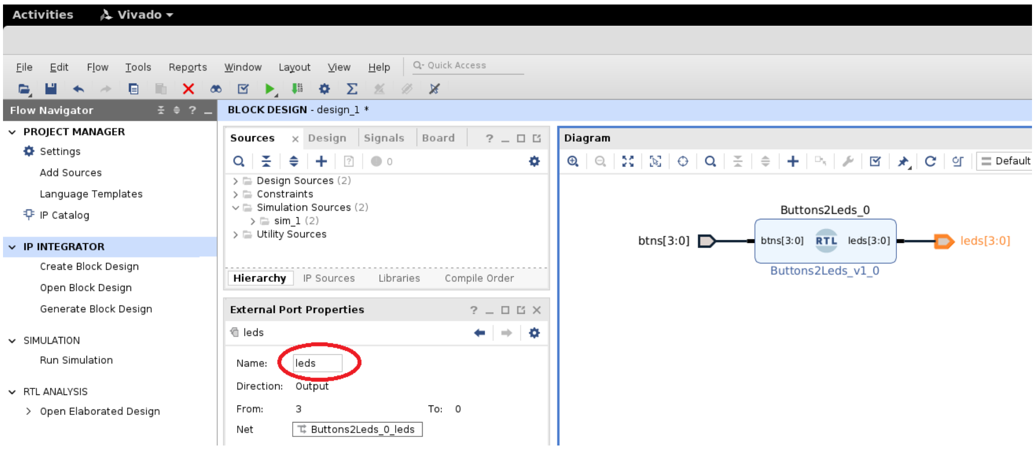
Module type: RTL

Search:

● Buttons2Leds (buttons2leds.vhdl)

☑ Hide incompatible modules

The schematic will be updated to contain our new component. Click on each of the ports and select "Make external." Vivado will automatically insert the I/O symbols to connect the ports to the FPGA pins at the location specified in the constraints file. However, Vivado renames the pins and they will not match the definition in the constraints file. To solve this problem, click on the external pins and modify their names:



BLOCK DESIGN - design_1 *

Sources: Design Sources (2), Constraints, Simulation Sources (2), sim_1 (2), Utility Sources

External Port Properties

leds

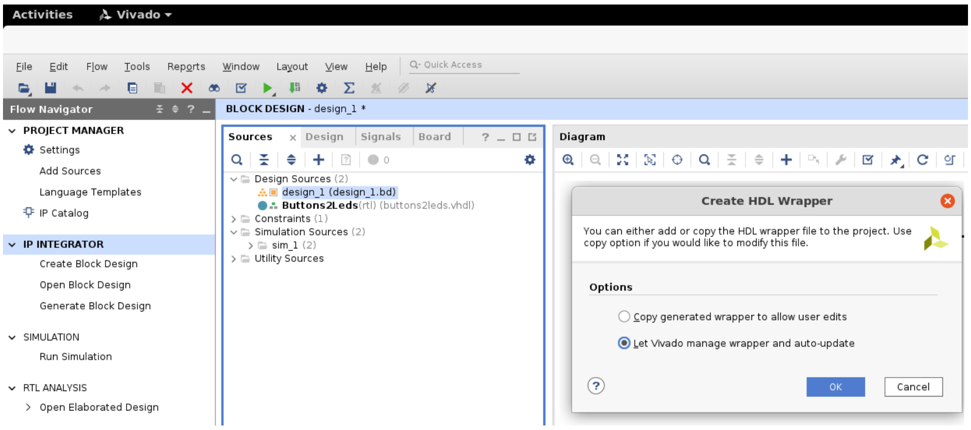
Name: leds

Direction: Output

From: 3 To: 0

Net Buttons2Leds_0_leds

In order for Vivado to be able to synthesize a block diagram schematic, we need to create an HDL wrapper for it. Right-click on the schematic name and select the option "Create HDL wrapper." Then, allow Vivado to manage the updates to the HDL wrapper itself, so that we do not need to update the wrapper manually each time we change the schematic:



BLOCK DESIGN - design_1 *

Sources: Design Sources (2), design_1 (design_1.bd), Buttons2Leds(rtl) (buttons2leds.vhdl), Constraints (1), Simulation Sources (2), sim_1 (2), Utility Sources

Create HDL Wrapper

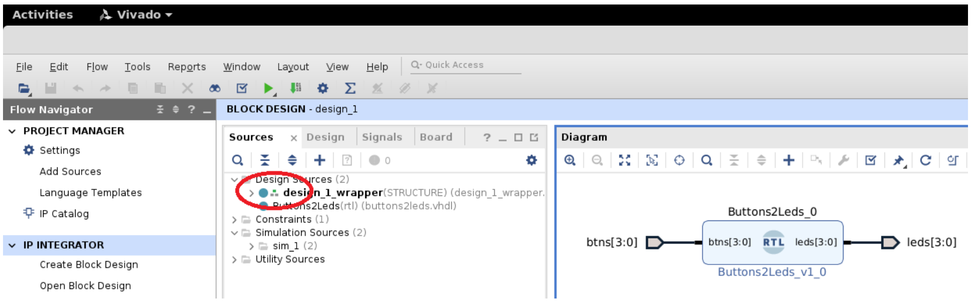
You can either add or copy the HDL wrapper file to the project. Use copy option if you would like to modify this file.

Options

○ Copy generated wrapper to allow user edits

● Let Vivado manage wrapper and auto-update

Once the HDL wrapper for the schematic is completed, Vivado will generally recognize correctly the project hierarchy by itself and select the correct top module. If this is not the case for your project, select manually the top module:

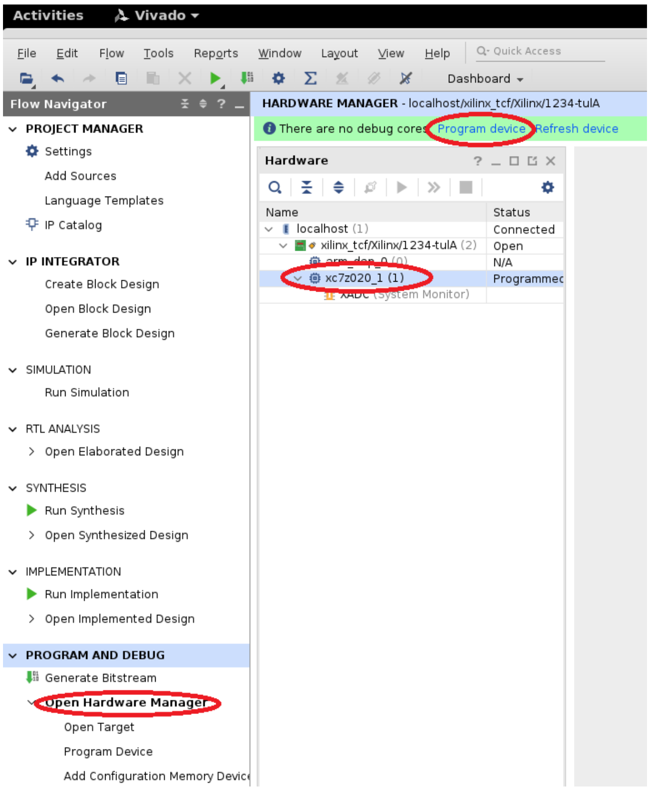


Once the design is complete, click on the line "Generate bitstream" to start and complete the bitstream generation process.

2.2 Programming the FPGA and testing the design

Please, ensure that the SD card is not in the slot before programming the FPGA.

Once the bitstream is correctly generated, connect the board to the PC using the USB cable and turn on its power switch. Click on the line "Open Hardware Manager." Select the option "Open target" and allow the tool to select automatically a local device. After a few seconds, the hardware manager will show the list of detected devices. Select the FPGA of the Pynq board (`xc7z020`) and program the device.



Name	Status
localhost (1)	Connected
xilinx_tcf/Xilinx/1234-tulA (2)	Open
arm_dap_0 (0)	N/A
xc7z020_1 (1)	Programmec
XADC (System Monitor)	

Once the design is loaded into the FPGA, the status of the four push buttons will be automatically reflected into the green LEDs of the board.

3 Connecting to the Pynq board for the first time

When using an SD card with an appropriate operating system (OS) image, the ARM cores boot from it and start running as soon as power is applied, without needing any design in the programmable logic. In that aspect, they behave as a stand-alone dual-core computer with 512 MiB of RAM memory. In our case, we are booting the system with an image based on Ubuntu 18.04 prepared by Xilinx

(Pynq v2.6). The system is a complete Linux computer with most of the functionality expected in this type of systems. In particular, we can use `ssh` to log into the board. The Pynq OS can be used to test overlays (HW designs) directly from Python using Jupyter notebooks. For more information on how to use the Pynq Python environment, refer to the official documentation at `https://pynq.readthedocs.io/` . In this course, we are going to interact with our peripherals in a Linux environment using C/C++ applications.

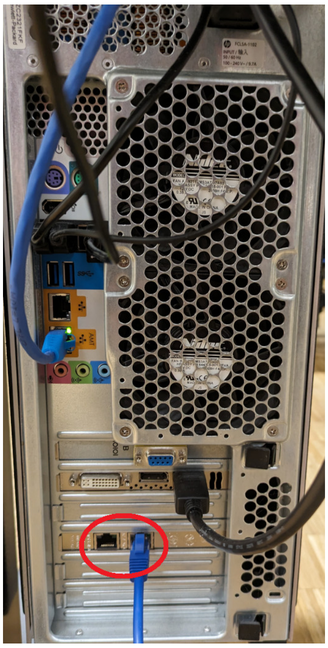
In the event of file system corruption in your SD card, you can write a fresh image into it. For this, you can use any usual tool for programming SD cards, including the command `dd` in Linux or the free software Balena Etcher (`https://www.balena.io/etcher/`). Our experiments will not work with version 2.7 or higher of the Pynq Linux image. The image can be downloaded from:
`https://dpoauwgwqsy2x.cloudfront.net/Download/pynq_z2_v2.6.0.zip`
MD5 checksum: *1e836bed78c3ddeb2c69a074925df8fe*
SHA256 checksum: *88c86412188cfb878217eb3d0a7ab5898ec52dfbc881a7f4f7d3526a55ba3978*
Manufacturer web: `https://www.tulembedded.com/FPGA/ProductsPYNQ-Z2.html`

3.0.1 Setting up the Ethernet connection

The Pynq board has a wired Ethernet interface that we can use to connect it directly to a computer or to a router. An Ethernet cable is included in the board box. By default, dual virtual interfaces are configured for the Ethernet interface. Therefore, it will use any address received through DHCP during boot (e.g., from a router) and also the address 192.168.2.99 . If the board is connected directly to the computer, we need to set up the Ethernet configuration of the computer to use the subnet of the board:

Board IP address: 192.168.2.99
Subnet mask: 255.255.255.0
Computer IP address: 192.168.2.x , where x is any number between 3 and 254.

The computers in the lab are already configured. The Ethernet cable of the board can be attached directly to the lower right port (the ones in horizontal) of the PC:



If the board is connected to a router, it uses DHCP to obtain its IP address automatically from the router. In this case, we can use the serial terminal to obtain its IP address. Use the `ifconfig` command to retrieve the *inet* address of the *eth0* interface:

```
 [  OK  ] Started Serial Getty on ttyPS0.
 [  OK  ] Reached target Login Prompts.

PYNQ Linux, based on Ubuntu 18.04 pynq ttyPS0
pynq login: xilinx (automatic login)

Welcome to PYNQ Linux, based on Ubuntu 18.04 (GNU/Linux 4.19.0-xilinx-v2019.1 armv7l)

The programs included with the PYNQ Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

PYNQ Linux comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

xilinx@pynq:~$ ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
inet 192.168.1.14  netmask 255.255.255.0  broadcast 192.168.1.255
...

eth0:1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
inet 192.168.2.99  netmask 255.255.255.0  broadcast 192.168.2.255
...
```

In the previous example, the board has the IP address 192.168.1.14, which should be directly accessible from our computer if it is connected to the same router.

The serial interface is always available and can be reached using picocom, minicom, screen or any other serial console software in Linux, or Putty in Windows. The ttyUSB number of the peripheral can be determined seeing the kernel messages with dmesg.

```
sudo picocom -b 115200 -f n -p n -d 8 /dev/ttyUSB2
```

If no messages are being displayed on the serial terminal, press `enter`. The default user is already logged in. This terminal can be used (with the same configuration that we normally use for all the exercises) in case we have problems to identify the IP address of the network interfaces, or if we need to modify the network configuration in the board. Connecting the terminal immediately after powering up the board allows inspecting the boot messages of the Linux kernel.

To connect to the board, simply use any SSH client (e.g., putty or command-line ssh on Windows, or ssh on Linux/Mac):

```
ssh xilinx@192.168.2.99
```

The default user name and password are “xilinx”/“xilinx”. This user has sudo permissions, in case you need to install any additional packages. The Ubuntu repositories are correctly configured in the SD card image; therefore, you can install any additional packages needed for your final project. For this, the board needs to be connected directly to a router or the computer has to be configured to share its Internet connection.

Change the default password of the xilinx board, particularly if you plan on connecting it to a network. Even better, configure it to accept only authentication via SSH keys.

3.1 Programming bitstreams

After booting the board, wait about one minute before reprogramming it with your own bitstream because the boot script programs it with an example overlay that flashes the LEDs to show that the board is active. You can know that the board is ready when the RGB LEDs flash in blue three times.

We can use two methods to program the bitstreams into the FPGA once the board is running Linux:

1. After copying the bitstreams to the board (using `scp`), use the script "`programOverlay.py`" provided in Moodle:

   ```
   sudo ./programOverlay.py design_1.bit
   ```

   This script requires the bitstream file (`.bit`) and the hardware description file (`.hwh`). Both files need to have the same name; rename them manually if necessary. Read the contents of the script "`extractBitstream.sh`" to learn where to find both files inside the Vivado project.

2. Program the FPGA from Vivado using the Hardware Manager. In this case, the board has to be connected to the computer using the USB programming cable (the same used to power it). Open the Vivado project and then use the Hardware Manager to connect to the board locally. You can program the board directly as many times as required. Make sure that the processors (PS) are not using the FPGA side before programming, or the board will crash.

Once the bitstream is programmed, we can interact freely with it. The bitstream does not need to be reloaded unless the board is rebooted, the PL side (FPGA design) is in an unrecoverable state due to design errors, or a change of bitstream is desired.

The board programming scripts check the configuration of the PS-generated clocks in the bitstream and correctly configure the PS. However, exert care when using clocks different than the default one (100 MHz).

3.1.1 Additional notes on the Pynq board

The board does not have a battery to maintain a real-time clock. If the board does not have an Internet connection, you will need to set up its clock manually for `make` to work correctly with files copied from a computer, **every time you power on the board**:

```
xilinx@pynq:~$ sudo date -s "2024-02-19 13:39:00 UTC+1"
[sudo] password for xilinx:
Mon Feb 19 12:39:00 UTC 2024
```

Once connected to the board, verify the characteristics of the system using standard commands:

```
xilinx@pynq:~$ lscpu
Architecture:        armv7l
Byte Order:          Little Endian
CPU(s):              2
On-line CPU(s) list: 0,1
Thread(s) per core:  1
Core(s) per socket:  2
Socket(s):           1
Vendor ID:           ARM
Model:               0
Model name:          Cortex-A9
Stepping:            r3p0
BogoMIPS:            650.00
Flags:               half thumb fastmult vfp edsp neon vfpv3 tls vfpd32

xilinx@pynq:~$ free -h
total        used        free      shared  buff/cache   available
Mem:           494M        124M        215M        1.2M        154M        359M
Swap:          1.0G          0B        1.0G
```

Table 1: Registers presented by a hypothetical AXI4 peripheral.

Offset	Direction	Description
0x00	R/W	Control & status register (write 1 to start, read 0 when done)
0x04	R/W	Parameter 1
0x08	R/W	Parameter 2
0x0C	R	Result

```
xilinx@pynq:~$ df -hT
Filesystem     Type      Size  Used Avail Use% Mounted on
/dev/root      ext4       15G  7.1G  6.8G  52% /
devtmpfs       devtmpfs  183M     0  183M   0% /dev
tmpfs          tmpfs     248M     0  248M   0% /dev/shm
tmpfs          tmpfs     248M  1.3M  247M   1% /run
tmpfs          tmpfs     5.0M     0  5.0M   0% /run/lock
tmpfs          tmpfs     248M     0  248M   0% /sys/fs/cgroup
/dev/mmcblk0p1 vfat      100M  6.5M   94M   7% /boot
tmpfs          tmpfs      50M     0   50M   0% /run/user/1000
tmpfs          tmpfs      50M     0   50M   0% /run/user/1001
```

You can also use `htop` or `top` to check the processor and memory usage of the system in real time.

4 Exercise 2: Accessing HW peripherals from a Linux application

In this exercise we will learn how to interact with a HW peripheral from a Linux application running on the ARM cores of the Zynq system-on-chip (SoC).

During this course we will always use memory-mapped peripherals. These peripherals present a set of registers on the address map of the system that the applications running on the processors can use to interact with them. In general, the applications only need to create a pointer to the base address of the peripheral register file and access the individual registers using offsets. An example of a hypothetical peripheral that takes two parameters and produces a result operating on them is shown in Table 1. With that register file, the application could use the following code to interact with the peripheral:

```
volatile uint32_t * regs = (uint32_t *)0x41200000; // Base address in Vivado
*(leds+1) = 23;          // First parameter
*(leds+2) = 25;          // Second parameter
*(leds) = 1;             // Signal start
while ( *(leds) != 0);   // Wait for completion
return *(leds+2);        // Read the result
```

The main obstacle when accessing memory-mapped peripherals in an operating system with *virtual memory*, such as Linux, Android, Windows or macOS, is that applications cannot access the physical address map of the system. Instead, each application is confined into its own *virtual address space*, which prevents it from accessing system resources directly and memory from other applications. We will delve deeper into the concept of virtual memory in the following lessons.

To overcome this restriction, our applications have to map the physical addresses of our peripherals into their own virtual address space. We use a library developed by Xilinx that offers a function called `cma_mmap()` specifically for this purpose. We will encapsulate the interaction with Xilinx's library inside a class, named `CAccelDriver`.

Table 2: Registers presented by the Xilinx AXI4 GPIO IP core. In the 3-state registers: 0 → pin configured as output; 1 → pin configured as input. In this system, we use only the four LSBs of each data register, which are connected to the physical buttons and LEDs.

Offset	Direction	Description
0x00	R/W	Channel 1, data register
0x04	R/W	Channel 1, 3-state register
0x08	R/W	Channel 2, data register
0x0C	R/W	Channel 2, 3-state register

> ▶ **Observation:** *Download the sources for this session from Moodle and read the code of the `CAccelDriver` class. For now, just look into the class declaration and the implementation of the `Open()` function.*

We have prepared a HW design that contains one general-purpose input/output (GPIO) peripheral with two channels. The first channel is connected to the Pynq's four green LEDs. The second channel is connected to the four push buttons. The goal of this exercise is to write an application that maps the register file of the GPIO into its own virtual address space and interacts with the GPIO to copy the value of the buttons to the LEDs.

4.1 The GPIO peripheral

The documentation for the AXI4 GPIO IP core is available in the following URL:
`https://docs.xilinx.com/v/u/1.01b-English/ds744_axi_gpio`
The GPIO core offers two independent logic channels with a width of up to 32 bits. Each bit in a channel can be configured independently as input or output, using a tri-state gate. The core presents two registers per channel to the processor: one to control the tri-state gates (i.e., to configure each pin as input or output), one to access the physical pins. The processor has to write the configuration in the tri-state control register; then, it can read the data register to know the state of the external pins configured as inputs, or write into the data register to change the value driven towards the outside world by the output pins. Table 2 describes the available registers.

The GPIO IP core can also be configured to generate interrupts if necessary. In this example, we will keep the interrupts disabled.

To drive an output pin (e.g., to an LED), the processor has to configure the corresponding bit of the 3-state register as output and then write into the data register the new output value. Since the processor writes to the complete register, all the output pins are affected at the same time. Conversely, to read the value of the input pins, the processor has to configure the bits of the 3-state register as inputs and then read from the data register. In our example, all the outputs are connected to one GPIO channel and all the inputs to another, which eases programming, but this is not a general requirement.

To know the address of each register, we need the base address of the peripheral on the system memory map, which we can find in the address editor of Vivado, then we add the offset of the register we want to access. However, since Linux uses virtual memory, we cannot access the physical address directly from our application. Instead, we need to obtain a suitable virtual address in the virtual address space of our application that maps to the same physical address. We use the functions `CAccelDriver::Open()` and `cma_mmap()` to obtain that virtual address.

4.2 Writing SW to interact with the GPIO peripheral

Download the sources of this exercise. Copy the file "`exer02.txz`" to the Pynq board using `scp` and uncompress it:

```
FROM THE PC:
scp Exer02_GPIO_SW_Exercise.txz xilinx@192.168.2.99:./

IN THE PYNQ BOARD:
tar xJvf Exer02_GPIO_SW_Exercise.txz
cd Exer02_GPIO_SW_Exercise
```

Read carefully the sources of the class `CAccelDriver` (for now, ignore the direct memory access (DMA) allocation functions) and the sources of file "`exer02.cpp`". The address of the GPIO peripheral in the physical address map of the SoC is `0x41200000`. This address is determined in Vivado while creating the system. Complete the source code of the application (the missing parts are marked with . . .).

Before starting the application, the bitstream has to be programmed in the FPGA. For that, use the script "`programOverlay.py`" provided in Moodle with the bitstream files of the example:

```
xilinx@pynq:~$ sudo ./programOverlay.py exercise02.bit
```

Tip: Ensure that the script has execution permissions in the Pynq board.

Execute the application — using `sudo` — and observe the generated messages.

> ▶ **Question:** *What is the physical address of the peripheral? Using the slides shown in class during this lesson, where is the peripheral mapped in the address space of the Zynq-7000 SoC?*

> ▶ **Question:** *What is the virtual address used by the application? How much does this address represent in decimal (base-10)? Does this address correspond to any DRAM or peripheral range in the address space of the Zynq-7000 SoC? Does this address make any sense for you?*

> ▶ **Observation:** *Analyze how the program defines the base address for the peripheral and then uses offsets to access each of the HW registers. This is pointer arithmetic, in which the offset values are multiplied by the size of the data type.*

> ▶ **Question:** *Use the command `htop` in a second terminal connected to the board to determine the CPU utilization of our program. How much is it? What happens if you remove the sleeping function at the end of the main loop?*

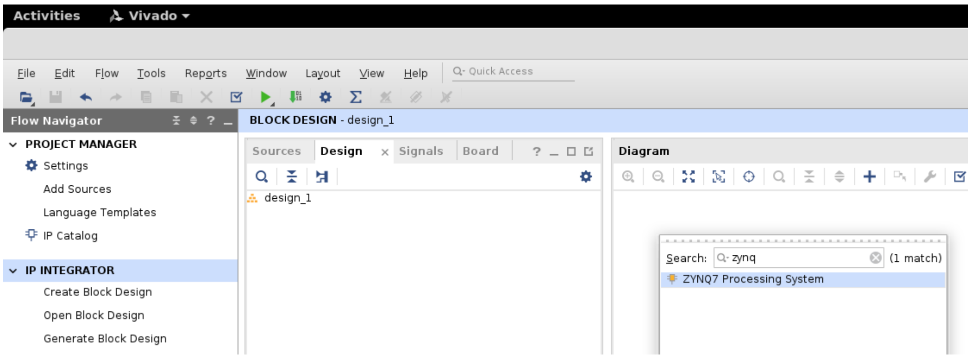
> ▶ **Question:** *Why do we need to use `sudo` to execute our application? Try to execute it normally and observe what happens. Why do you think the system does not allow normal users to execute this type of applications?*

5 Exercise 3: Designing and integrating a HW peripheral in a SoC

In this exercise, we will re-create the same SoC architecture that was used in the previous exercise. Therefore, we will be able to use the same SW in Linux to interact with the HW peripherals.

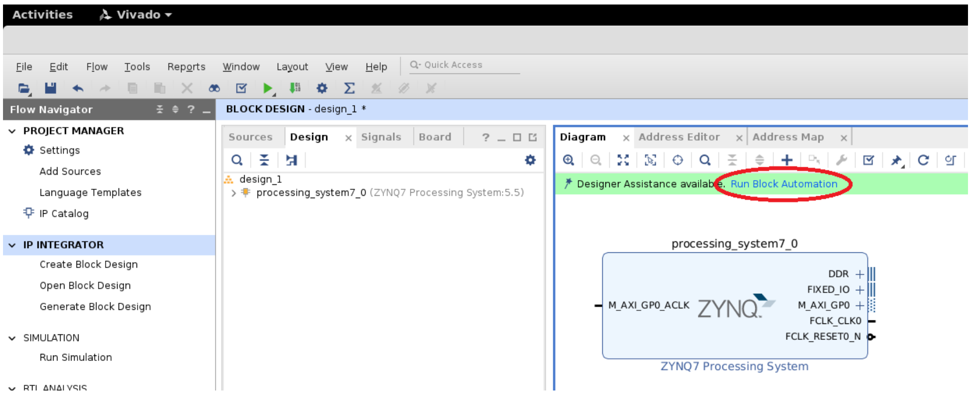
5.1 Designing the HW in Vivado

Open Vivado and generate a new project with a new block design. Click on the "plus" sign in the schematic view and select the ZYNQ7 Processing System:



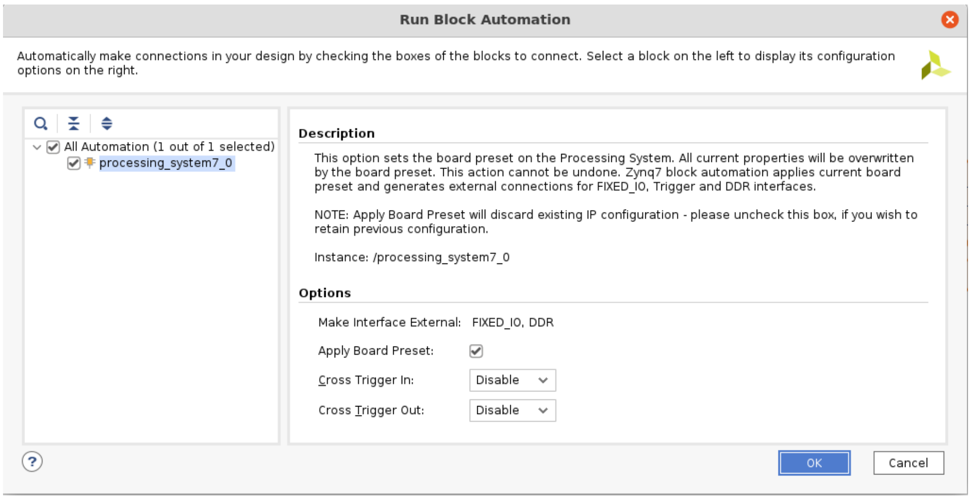
The ZYNQ7 Processing System represents the fixed part of the Zynq chip, that is, the two ARM Cortex-A9 cores, their L_1/L_2 cache memories, the DDR memory controller, clock generators and all the fixed peripherals. It interfaces with the programmable logic via a set of master/slave AXI4 ports. The ZYNQ7 block can also generate the reference clock and reset signal for the designs in the FPGA.

The ZYNQ7 Processing System presents in the diagram a set of ports to communicate with the programmable logic:



- M_AXI_GP0: This port allows the ARM cores to act as AXI4 masters with respect to the peripherals created in the programmable logic.
- FCLK_CLK0: This is the reference clock for the logic. Its frequency can be adjusted according to the design needs. Up to four clocks of different frequencies can be generated for the programmable logic.
- FCLK_RESET0_N: This is the reference reset for programmable logic that uses FCLK_CLK0 as clock.
- M_AXI_GP0_ACLK: This is the clock for the AXI4 bus corresponding to port M_AXI_GP0. Normally, it will be connected to FCLK_CLK0.
- DDR: This is the set of signals that control the DDR memory controller of the Zynq chip to the external DDR-SDRAM. The Pynq-Z2 board has 512 MiB of DDR2-SDRAM with a bus of 16 bits. The memory controller can be configured up to 533 MHz.
- FIXED_IO: These signals connect the Zynq chip (cores and programmable logic) to external I/O devices such as the UART pins. These signals are hardwired in the board and are independent of whatever peripherals are instantiated in the programmable logic.

Vivado offers a wide range of automation. For every IP core, the designer can specify actions that need to be taken in order to fully configure the peripheral. After adding an IP core, we will see a green bar at the top of the window suggesting the type of automation available. Click on the bar to run the block automation:



Run Block Automation

Automatically make connections in your design by checking the boxes of the blocks to connect. Select a block on the left to display its configuration options on the right.

☑ All Automation (1 out of 1 selected)

☑ processing_system7_0

Description

This option sets the board preset on the Processing System. All current properties will be overwritten by the board preset. This action cannot be undone. Zynq7 block automation applies current board preset and generates external connections for FIXED_IO, Trigger and DDR interfaces.

NOTE: Apply Board Preset will discard existing IP configuration - please uncheck this box, if you wish to retain previous configuration.

Instance: /processing_system7_0

Options

Make Interface External: FIXED_IO, DDR

Apply Board Preset: ☑

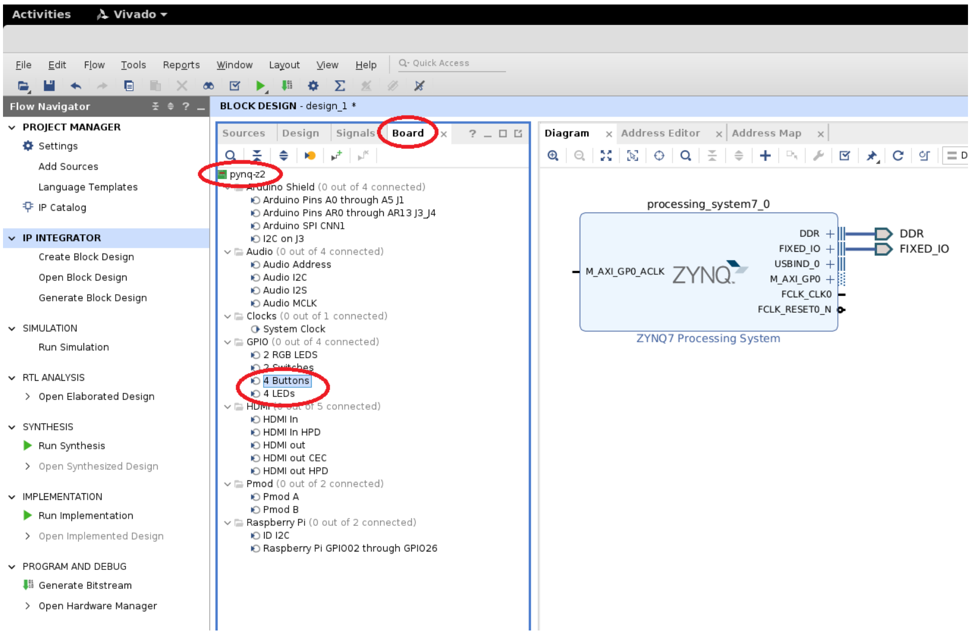
Cross Trigger In: Disable

Cross Trigger Out: Disable

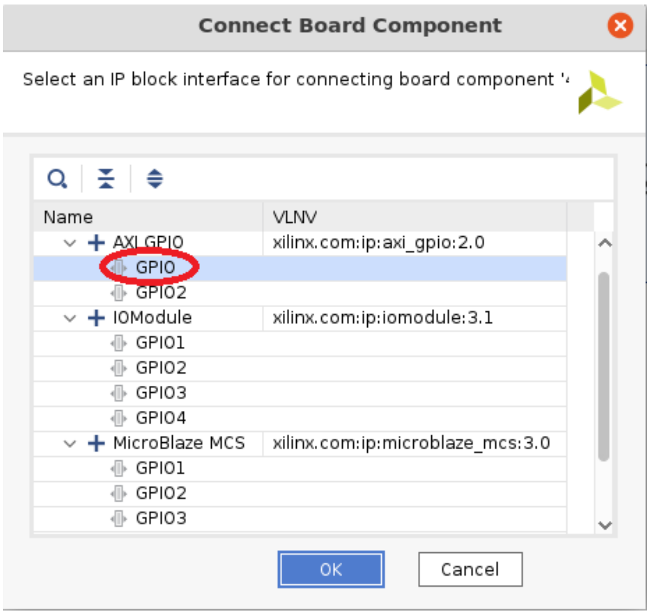
The block automation dialog for the ZYNQ7 Processing System proposes us to apply the presets defined in the Pynq-Z2 board support package (BSP). Among others, it will connect the DDR and FIXED_IO ports to the external pins of the chip — these connections are hardwired and specific for this board. Accept all the configurations with the default presets.

Now, we can add any of the components defined in the BSP of our board to the design. The board manufacturer defines in the BSP file the peripherals of the board, their physical pin assignments, electrical characteristics and common names for the user. Additionally, the type of connections in the system can also be defined via automation.

Click on the “Board” tab and select the “4 LEDs”:



Once we select an I/O element to add to the design, a dialog to define the type of system connection that we want to use will appear:



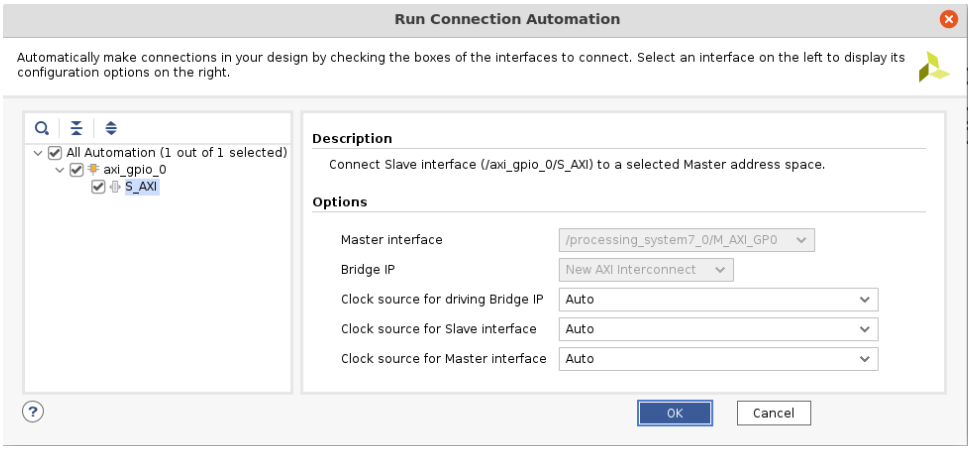
Connect Board Component

Select an IP block interface for connecting board component '4

Name	VLNV
AXI GPIO	xilinx.com:ip:axi_gpio:2.0
GPIO	
GPIO2	
IOModule	xilinx.com:ip:iomodule:3.1
GPIO1	
GPIO2	
GPIO3	
GPIO4	
MicroBlaze MCS	xilinx.com:ip:microblaze_mcs:3.0
GPIO1	
GPIO2	
GPIO3	

In this case, we are offered to connect the four LEDs using a GPIO, which is an IP core created by Xilinx to connect bus masters (e.g., a processor) directly to physical pins of the FPGA chip. A GPIO peripheral offers two logical ports (named `GPIO` and `GPIO2`), each of which can have up to 32 bits that can be connected to the corresponding 32 external pins of the chip. Select the default option — this will create a new GPIO peripheral connected through an AXI4 bus and will connect the LEDs to its first logical port. The script will also create an external port, which is connected to physical pins of the FPGA as defined in the BSP.

Run the automation script. We have to select, for each of the slave ports in our design, to which master port they are connected. In our case, we want to connect the slave port of the GPIO to the master port of the Zynq7 block. Since we only have one AXI4 master in the system (the M_AXI_GP0 port of the fixed logic), the option to select the master interface is disabled:



Run Connection Automation

Automatically make connections in your design by checking the boxes of the interfaces to connect. Select an interface on the left to display its configuration options on the right.

☑ All Automation (1 out of 1 selected)
  ☑ axi_gpio_0
    ☑ S_AXI

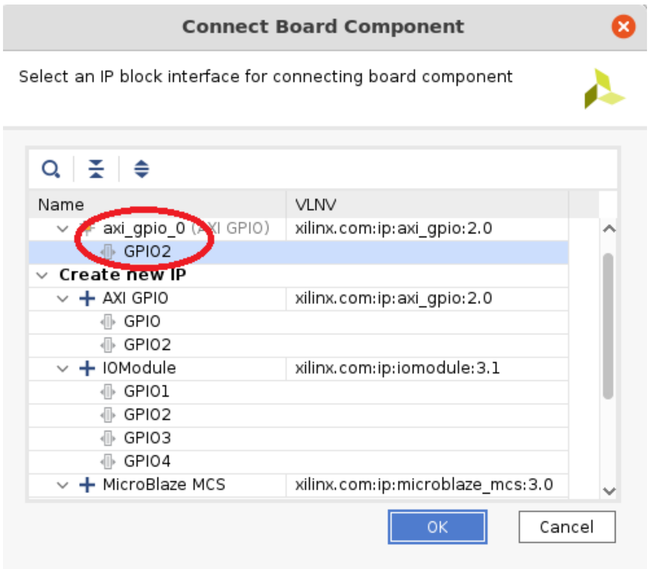
Description

Connect Slave interface (/axi_gpio_0/S_AXI) to a selected Master address space.

Options

Master interface	/processing_system7_0/M_AXI_GP0
Bridge IP	New AXI Interconnect
Clock source for driving Bridge IP	Auto
Clock source for Slave interface	Auto
Clock source for Master interface	Auto

Now, add the component “4 Buttons” from the board elements. In this case, we will be offered to connect the buttons to a second port of the same GPIO IP core:



Connect Board Component

Select an IP block interface for connecting board component

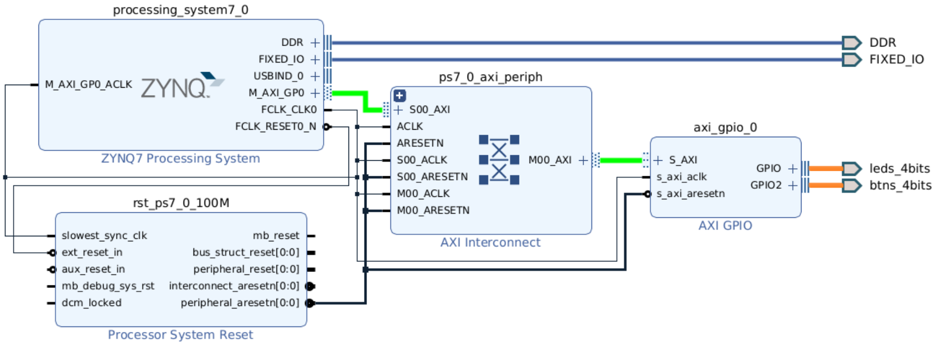
Name	VLNV
axi_gpio_0 (AXI GPIO)	xilinx.com:ip:axi_gpio:2.0
GPIO2	
Create new IP	
AXI GPIO	xilinx.com:ip:axi_gpio:2.0
GPIO	
GPIO2	
IOModule	xilinx.com:ip:iomodule:3.1
GPIO1	
GPIO2	
GPIO3	
GPIO4	
MicroBlaze MCS	xilinx.com:ip:microblaze_mcs:3.0

Accept the selection and observe how the final system is connected. Double-click on the element named `axi_gpio_0` to observe its configuration.

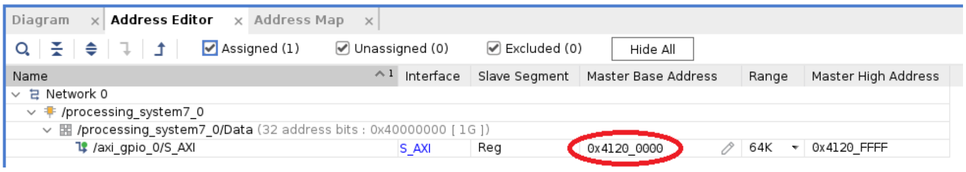
► **Question:** *Observe the number of connected pins for each logical port of the GPIO module. How many are used for the LEDs? How many for the GPIOs? Why? How does Vivado know the number of pins to connect for each of them?*

If we are using only elements defined in the BSP (i.e., from the board tab), we do not need to add our own `xdc` file since the pin locations are added automatically as well.[3] Otherwise, when connecting your own peripherals, do not forget to specify their corresponding physical pins and electrical configurations in a constraints file.

The following figure shows the final aspect of the design. The AXI buses from the processors to the AXI4 GPIO peripherals are highlighted in green. The wires (i.e., as opposed to a bus) from the GPIO peripherals to the physical pins are highlighted in orange.



Finally, click on the tab "Address Editor." Here, you will find the description of the address map of the system. Find out the address at which the GPIO peripheral is mapped, since this is the address that we will need to use in SW to access the GPIO peripherals:



Name	Interface	Slave Segment	Master Base Address	Range	Master High Address
Network 0					
/processing_system7_0					
/processing_system7_0/Data (32 address bits : 0x40000000 [1G])					
/axi_gpio_0/S_AXI	S_AXI	Reg	0x4120_0000	64K	0x4120_FFFF

[3] However, it is possible to add an `xdc` file specifying also the location of these ports. Just verify that the names of the external ports and the ports in the constraints file match.

If the base address assigned to the GPIO module in Vivado does not correspond to the address we used in the SW of the previous exercise, you have to adjust one of them for this example to work correctly!

> ▶ **Question:** *Double-click on the ZYNQ7 Processing System and observe its components. Click on the block named "32b GP AXI Master Ports" and try to find out how many independent master ports can be created. How many clocks can be generated for the programmable logic and how can we define their frequencies?*

> ▶ **Question:** *Synthesize the system and generate the bitstream. Transfer the design* `.hwh` *and* `.bit` *files to the Pynq board. Use the script "*`programOverlay.py`*" to program the bitstream in the FPGA and test it with the SW application we developed in the previous exercise. Do you obtain exactly the same behavior?*

6 Exercise 4: In-chip debugging with the System ILA IP core

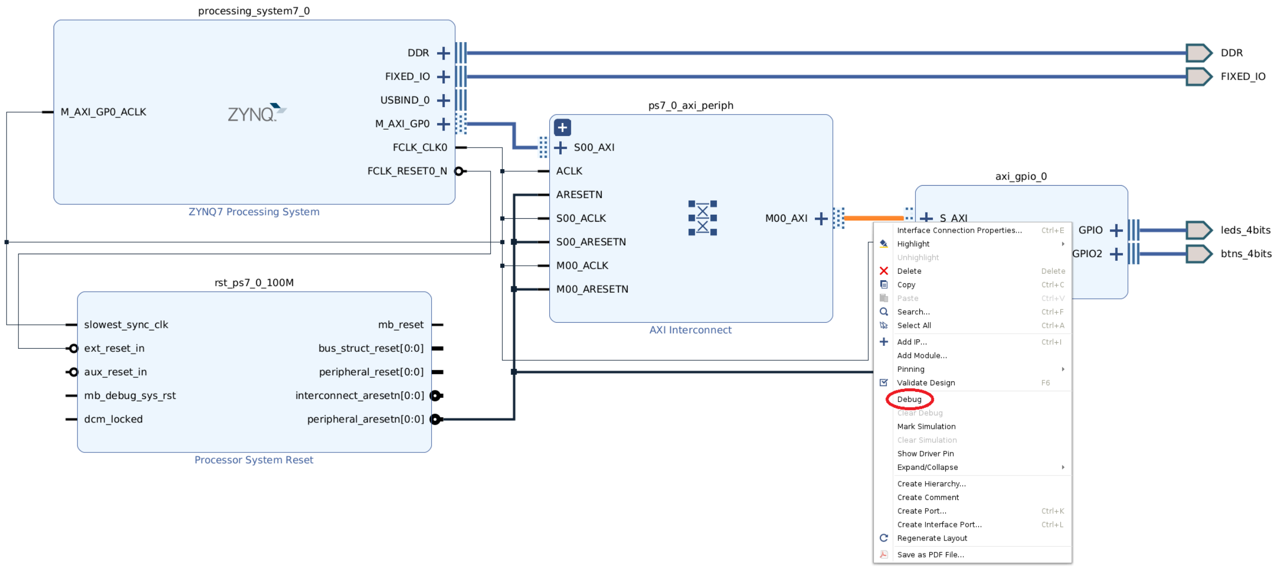
At this point, you know how to design a SoC integrating processors and IP cores, and how to write a Linux application that accesses the peripherals. However, what happens if something goes wrong? How can we observe the interactions between the SoC processor and the peripherals?

6.1 Preparing the HW design for debugging

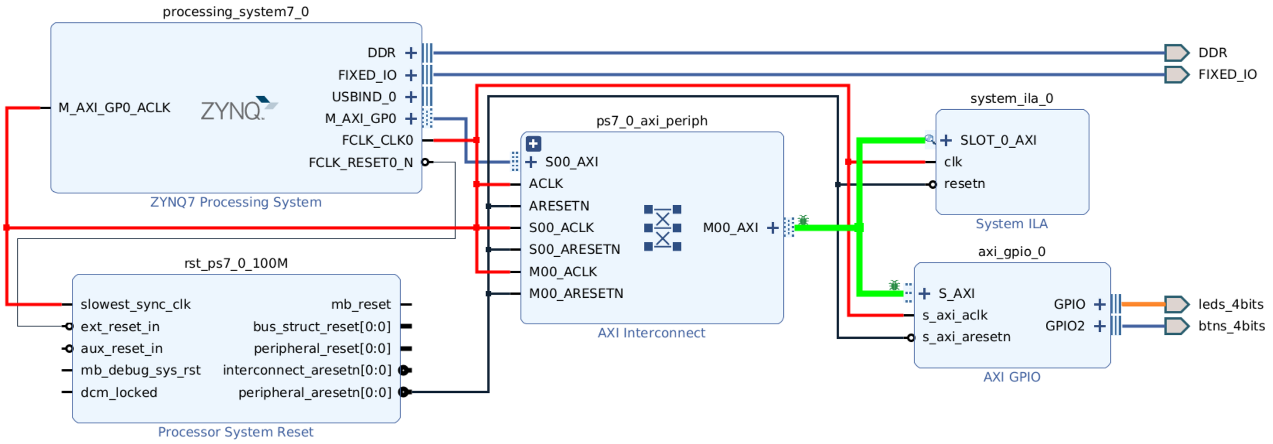
In this exercise we are going to use the System ILA IP core to debug the system while it is working. System ILA works like an oscilloscope, internal to the FPGA. It enables monitoring of individual signals, but it can also interpret transactions in an AXI4 bus.

The System ILA uses *trigger* conditions to start capturing signals. It is possible to trigger the capture based on logical conditions on the address and data lines of a bus, or on individual design signals. It is also possible to use external signals as trigger (i.e., an external port), or even drive the trigger condition outside. A design may contain multiple System ILA instantiations, and their trigger conditions can be cascaded. This is useful, for example, if we want to trigger the capture based on a small set of signals, but then we want to capture a long period in a second bus. With two ILAs, the first one can store the minimum number of samples — and thus reduce the number of internal BRAMs used — whereas the second ILA can have a deeper capture window. This saves internal FPGA memory to capture the more relevant signals.

To start debugging a bus, right click on it and select the "Debug" option:

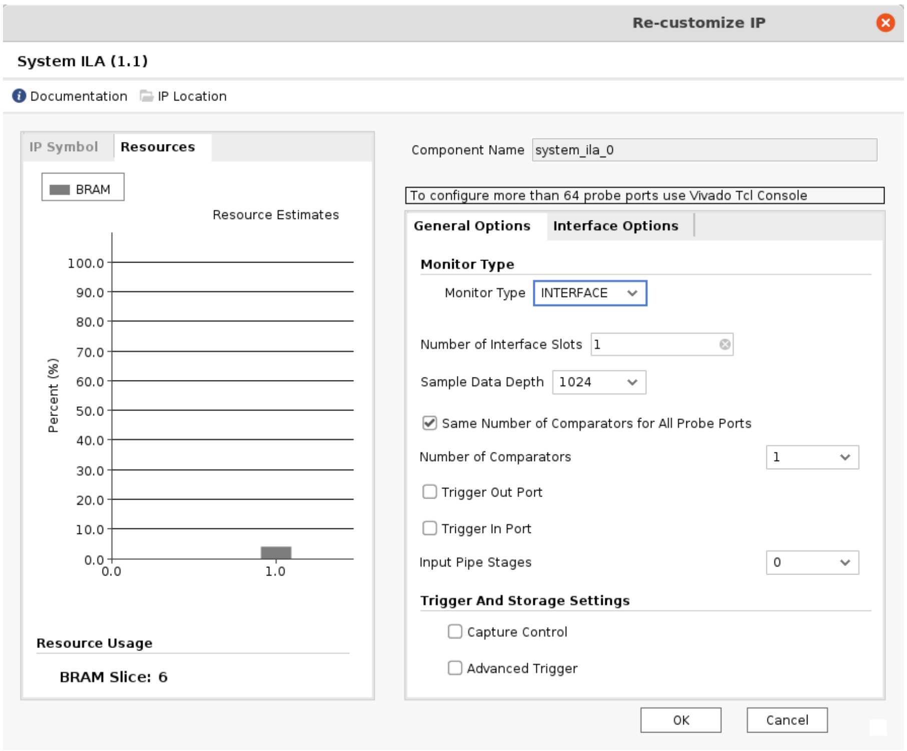


Once we have marked one or more signals (or buses) for debugging, Vivado will offer us an automation to add the corresponding System ILA instantiations to our design. Click on the automation. Keep active (i.e., do not change) the options to treat the address and data lines as both data and triggers. The resulting design should be similar to the following:



In the previous image, the lines of the clock tree deriving from the main 100 MHz clock are drawn in red; the bus that is being monitored is drawn in green. The System ILA instance is connected to the same clock than the bus it is monitoring. If one instance monitors multiple buses, it will accept one clock for each.

The System ILA instantiation can be configured clicking on it:



System ILA (1.1)

Documentation IP Location

Resource Usage

BRAM Slice: 6

Component Name system_ila_0

To configure more than 64 probe ports use Vivado Tcl Console

General Options Interface Options

Monitor Type

Monitor Type INTERFACE

Number of Interface Slots 1

Sample Data Depth 1024

☑ Same Number of Comparators for All Probe Ports

Number of Comparators 1

☐ Trigger Out Port

☐ Trigger In Port

Input Pipe Stages 0

Trigger And Storage Settings

☐ Capture Control

☐ Advanced Trigger

In particular, we can change the number of buses (“interface slots”) to be debugged. We can also select whether to monitor only buses (“interface”), independent logic signals (“native”) or both at the same time (“mixed”). In the case of adding native signals, we can specify how many we need to debug (number of probes). By default, Vivado will determine automatically the width of the native probes based on the width of the signals to which they connect. The options “Trigger out port” and “Trigger in port” activate the in/out ports to cascade triggers along a chain of System ILA instantiations. Finally, we can select the number of samples (i.e., depth) of the capture. A higher depth enables the capture of more consecutive events, but it requires more BRAMs — and the time to synthesize the resulting design will quickly increase.

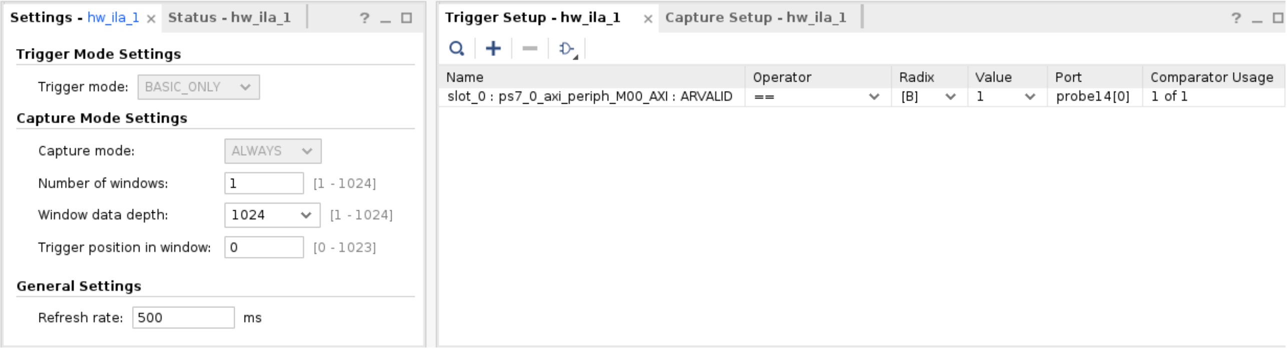
Once the design is configured, synthesize and implement it, and transfer the bitstream to the board.

6.2 Debugging the live system with System ILA

For this example, we are going to use the same software than in the previous one — after all, the functionality of the design is exactly the same. Therefore, program the FPGA with the new bitstream and start the software application in the board.

To connect to the System ILA, open the hardware manager in Vivado and connect to the board — it is possible to program the bitstream from Vivado rather than from Linux; just be sure that the application is not running before programming the bitstream or the system will hang. The hardware manager will identify the debugging probes in our design and group the bus signals by their corresponding channels for us.

The first step for debugging is deciding what are our trigger conditions. In the lower right pane, configure a trigger to start capturing when the processor reads the status of the switches. We can identify this condition when the signal `ARVALID` is '1' and the signal `ARADDR` has the address of our peripheral. In this case, we want to check when the switches are read, which corresponds to the memory address `0x41200008`.[4] The capture is done over a circular buffer; therefore, it is possible to determine in which position of the buffer the trigger should be at. For example, with 1024 samples, we can display 512 samples before the trigger condition, and 512 samples after it. For this exercise, configure the trigger position at 0. The following figure shows how to set the trigger based only on the first condition (i.e., the System ILA will trigger when the processor tries to read from any GPIO register):



Settings - hw_ila_1 | Status - hw_ila_1

Trigger Mode Settings

Trigger mode: BASIC_ONLY

Capture Mode Settings

Capture mode: ALWAYS

Number of windows: 1 [1 - 1024]

Window data depth: 1024 [1 - 1024]

Trigger position in window: 0 [0 - 1023]

General Settings

Refresh rate: 500 ms

Trigger Setup - hw_ila_1 | Capture Setup - hw_ila_1

Name	Operator	Radix	Value	Port	Comparator Usage
slot_0 : ps7_0_axi_periph_M00_AXI : ARVALID	==	[B]	1	probe14[0]	1 of 1

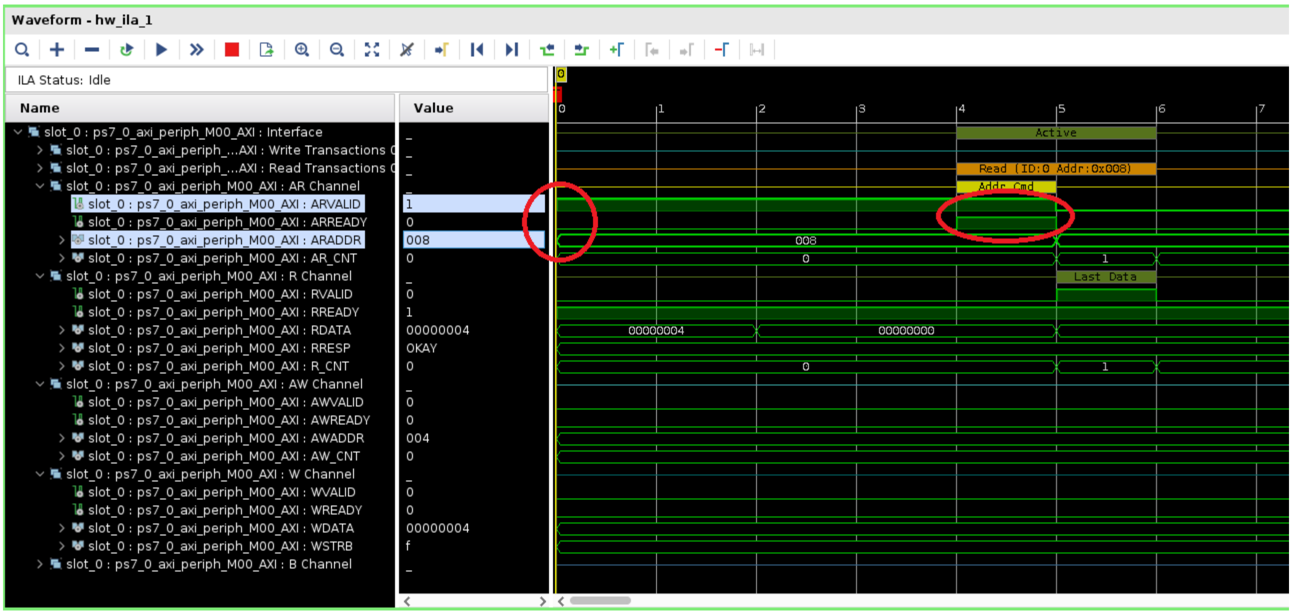
The AXI4 bus has five independent channels: read addresses (prefix `AR-`), read data (`R-`), write addresses (`AW-`), write data (`W-`) and write responses (`B-`). Once everything is configured, expand the bus channels in the wave panel to see all their signals. Press and hold one button on the board to have some distinct value to read — in the figures below, the second button was pressed. Then, arm the trigger (by pressing the "play" button). In this example, the ILA will trigger immediately, since our application is continuously reading the switches.

6.3 Analyzing bus transactions with System ILA

Select the signal `ARVALID` and go to the instant in which it gets the value '1'. If the trigger has been configured at position 0 of the capture window, then the condition will be true at cycle 0 in the

[4] However, the AXI interconnect already decodes the upper bits of the address to determine which peripheral to activate; therefore, we will only see the LSBs of the address. If instead we place the ILA between the processor and the interconnect, we would see the complete addresses for all the peripherals present in the system. You can easily modify your design to see the differences.

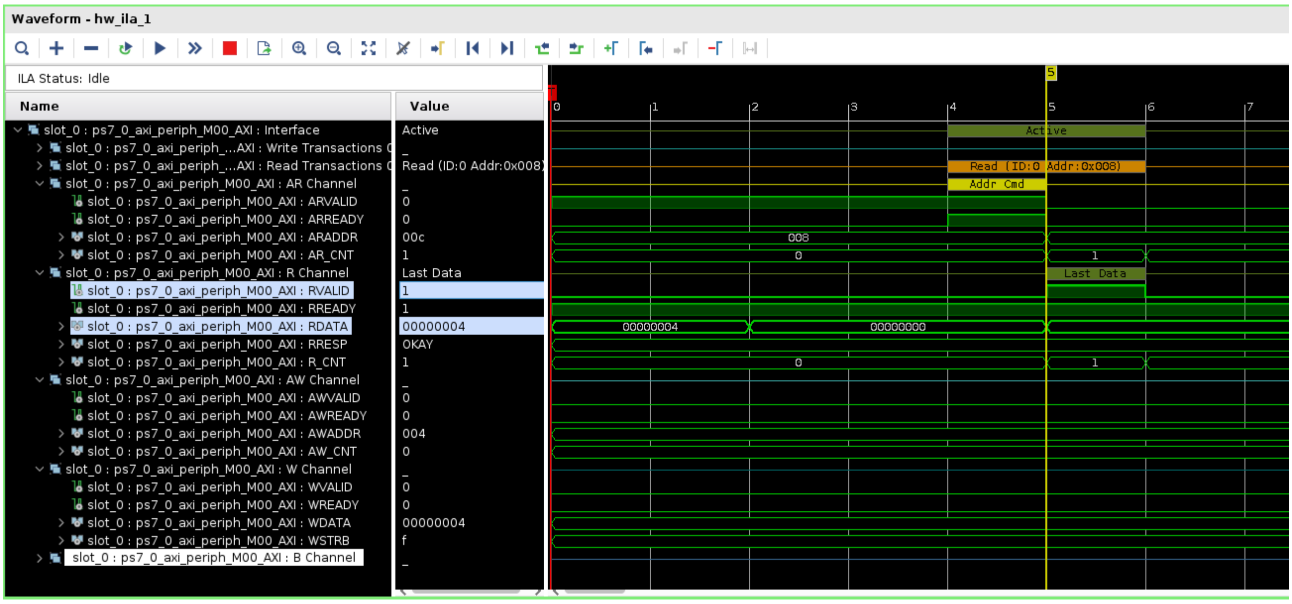
capture data. Check the value of ARADDR: it should have the value 0x8, which corresponds to the third register of our GPIO, the value read from the input pins connected to the buttons.[5]


Waveform - hw_ila_1

ILA Status: Idle

Name	Value
slot_0 : ps7_0_axi_periph_M00_AXI : Interface	_
slot_0 : ps7_0_axi_periph_...AXI : Write Transactions (	_
slot_0 : ps7_0_axi_periph_...AXI : Read Transactions (	_
slot_0 : ps7_0_axi_periph_M00_AXI : AR Channel	_
slot_0 : ps7_0_axi_periph_M00_AXI : ARVALID	1
slot_0 : ps7_0_axi_periph_M00_AXI : ARREADY	0
slot_0 : ps7_0_axi_periph_M00_AXI : ARADDR	008
slot_0 : ps7_0_axi_periph_M00_AXI : AR_CNT	0
slot_0 : ps7_0_axi_periph_M00_AXI : R Channel	_
slot_0 : ps7_0_axi_periph_M00_AXI : RVALID	0
slot_0 : ps7_0_axi_periph_M00_AXI : RREADY	1
slot_0 : ps7_0_axi_periph_M00_AXI : RDATA	00000004
slot_0 : ps7_0_axi_periph_M00_AXI : RRESP	OKAY
slot_0 : ps7_0_axi_periph_M00_AXI : R_CNT	0
slot_0 : ps7_0_axi_periph_M00_AXI : AW Channel	_
slot_0 : ps7_0_axi_periph_M00_AXI : AWVALID	0
slot_0 : ps7_0_axi_periph_M00_AXI : AWREADY	0
slot_0 : ps7_0_axi_periph_M00_AXI : AWADDR	004
slot_0 : ps7_0_axi_periph_M00_AXI : AW_CNT	0
slot_0 : ps7_0_axi_periph_M00_AXI : W Channel	_
slot_0 : ps7_0_axi_periph_M00_AXI : WVALID	0
slot_0 : ps7_0_axi_periph_M00_AXI : WREADY	0
slot_0 : ps7_0_axi_periph_M00_AXI : WDATA	00000004
slot_0 : ps7_0_axi_periph_M00_AXI : WSTRB	f
slot_0 : ps7_0_axi_periph_M00_AXI : B Channel	_

The peripheral will reply to the read request of the processor some time after. In this example, the answer arrives at cycle 5, when RVALID becomes '1':

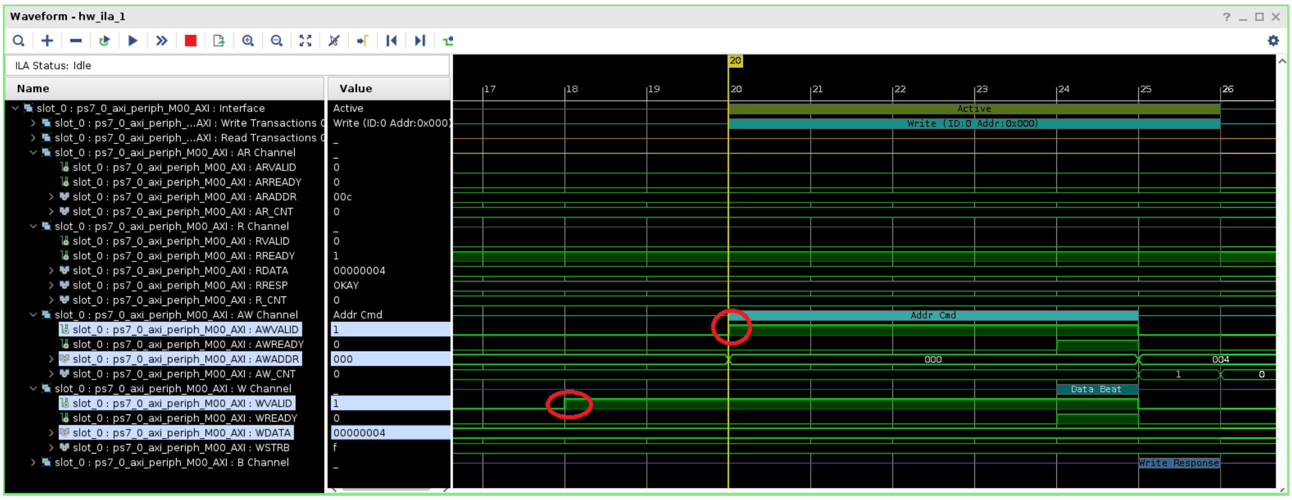

Waveform - hw_ila_1

ILA Status: Idle

Name	Value
slot_0 : ps7_0_axi_periph_M00_AXI : Interface	Active
slot_0 : ps7_0_axi_periph_...AXI : Write Transactions (	_
slot_0 : ps7_0_axi_periph_...AXI : Read Transactions (	Read (ID:0 Addr:0x008)
slot_0 : ps7_0_axi_periph_M00_AXI : AR Channel	_
slot_0 : ps7_0_axi_periph_M00_AXI : ARVALID	0
slot_0 : ps7_0_axi_periph_M00_AXI : ARREADY	0
slot_0 : ps7_0_axi_periph_M00_AXI : ARADDR	00c
slot_0 : ps7_0_axi_periph_M00_AXI : AR_CNT	1
slot_0 : ps7_0_axi_periph_M00_AXI : R Channel	Last Data
slot_0 : ps7_0_axi_periph_M00_AXI : RVALID	1
slot_0 : ps7_0_axi_periph_M00_AXI : RREADY	1
slot_0 : ps7_0_axi_periph_M00_AXI : RDATA	00000004
slot_0 : ps7_0_axi_periph_M00_AXI : RRESP	OKAY
slot_0 : ps7_0_axi_periph_M00_AXI : R_CNT	1
slot_0 : ps7_0_axi_periph_M00_AXI : AW Channel	_
slot_0 : ps7_0_axi_periph_M00_AXI : AWVALID	0
slot_0 : ps7_0_axi_periph_M00_AXI : AWREADY	0
slot_0 : ps7_0_axi_periph_M00_AXI : AWADDR	004
slot_0 : ps7_0_axi_periph_M00_AXI : AW_CNT	0
slot_0 : ps7_0_axi_periph_M00_AXI : W Channel	_
slot_0 : ps7_0_axi_periph_M00_AXI : WVALID	0
slot_0 : ps7_0_axi_periph_M00_AXI : WREADY	0
slot_0 : ps7_0_axi_periph_M00_AXI : WDATA	00000004
slot_0 : ps7_0_axi_periph_M00_AXI : WSTRB	f
slot_0 : ps7_0_axi_periph_M00_AXI : B Channel	_

We know that the application writes into the LEDs register immediately after reading the switches. Therefore, move forward in the capture until the signal AWVALID becomes '1'. Since the five AXI4 bus channels are truly independent, it is possible that a master sends the address and the data for a write at different times. In the figure below we can clearly see how the master sends the data (WVALID = '1') before sending the address. It is responsibility of the slave peripheral to tackle this situation, either by blocking the first channel until the second arrives, or buffering internally the values if one channel is faster than the other (more parallelism available in the bus at expense of slave complexity). The figure below shows a write transaction at address 0x0 (the first register of the GPIO) with a value of 0x00000004, which mimics the value read from the switches in this case:

[5] In the bus, we see 32-bit addresses, not offsets. The GPIO registers are at addresses 0 (0x0), 4 (0x4), 8 (0x8) and 12 (0xC).

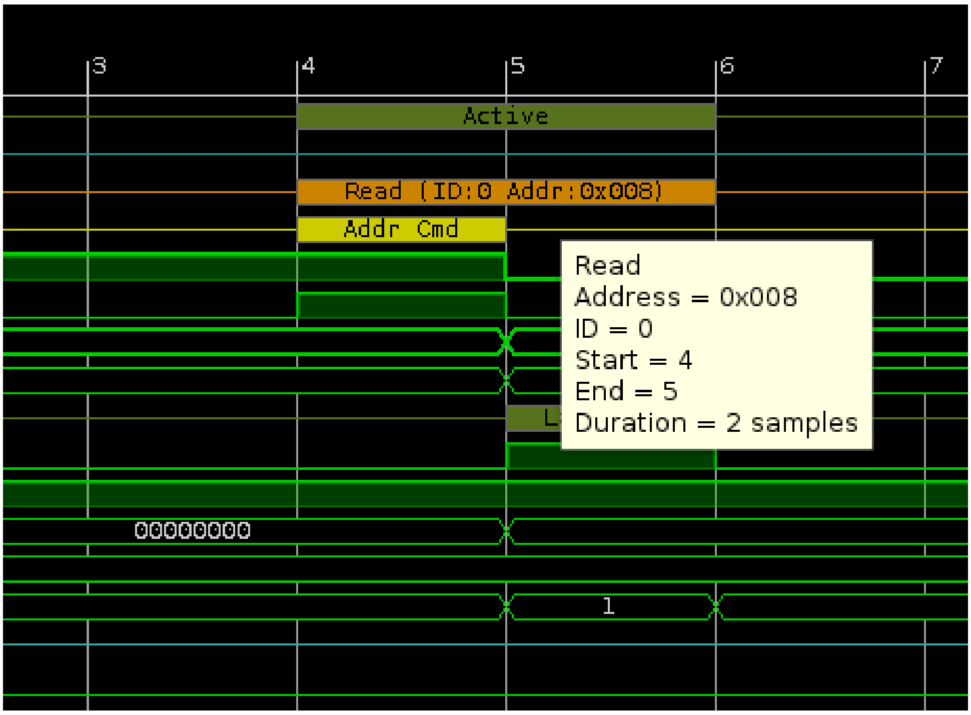


ILA Status: Idle

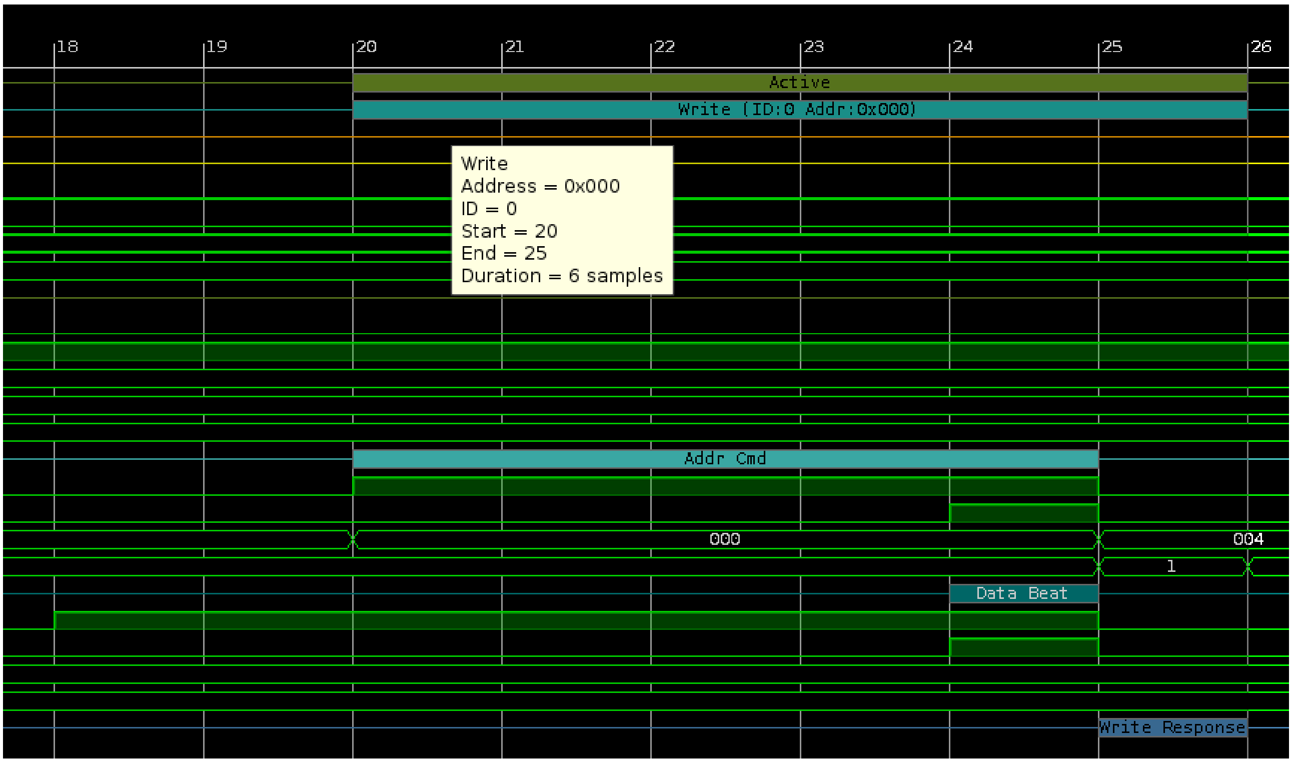
Name	Value
slot_0 : ps7_0_axi_periph_M00_AXI : Interface	Active
slot_0 : ps7_0_axi_periph_...AXI : Write Transactions (	Write (ID:0 Addr:0x000)
slot_0 : ps7_0_axi_periph_...AXI : Read Transactions (	_
slot_0 : ps7_0_axi_periph_M00_AXI : AR Channel	_
slot_0 : ps7_0_axi_periph_M00_AXI : ARVALID	0
slot_0 : ps7_0_axi_periph_M00_AXI : ARREADY	0
slot_0 : ps7_0_axi_periph_M00_AXI : ARADDR	00c
slot_0 : ps7_0_axi_periph_M00_AXI : AR_CNT	0
slot_0 : ps7_0_axi_periph_M00_AXI : R Channel	_
slot_0 : ps7_0_axi_periph_M00_AXI : RVALID	0
slot_0 : ps7_0_axi_periph_M00_AXI : RREADY	1
slot_0 : ps7_0_axi_periph_M00_AXI : RDATA	00000004
slot_0 : ps7_0_axi_periph_M00_AXI : RRESP	OKAY
slot_0 : ps7_0_axi_periph_M00_AXI : R_CNT	0
slot_0 : ps7_0_axi_periph_M00_AXI : AW Channel	Addr Cmd
slot_0 : ps7_0_axi_periph_M00_AXI : AWVALID	1
slot_0 : ps7_0_axi_periph_M00_AXI : AWREADY	0
slot_0 : ps7_0_axi_periph_M00_AXI : AWADDR	000
slot_0 : ps7_0_axi_periph_M00_AXI : AW_CNT	0
slot_0 : ps7_0_axi_periph_M00_AXI : W Channel	_
slot_0 : ps7_0_axi_periph_M00_AXI : WVALID	1
slot_0 : ps7_0_axi_periph_M00_AXI : WREADY	0
slot_0 : ps7_0_axi_periph_M00_AXI : WDATA	00000004
slot_0 : ps7_0_axi_periph_M00_AXI : WSTRB	f
slot_0 : ps7_0_axi_periph_M00_AXI : B Channel	_

Observe how the transaction does not finish immediately. AXI4 write transactions are considered completed when the valid-ready handshake is complete. Even if the master asserts the write data and write address valid signals at cycles 18 and 20, respectively, the peripheral does not signal `AWREADY` and `WREADY` until cycle 24. The master must keep all the signals stable until that moment, which is when the slave acknowledges the transaction. Finally, at cycle 25 there is a single-cycle transaction in the write response channel to signal that the write was correctly performed by the peripheral.

System ILA identifies automatically the activity periods of each channel. Hovering the mouse over the marker brings up a tooltip with a summary of the transaction, which contains among other information its length in cycles. The following two figures show the information for the read and write transactions captured in our example.



Read
Address = 0x008
ID = 0
Start = 4
End = 5
Duration = 2 samples



Write
Address = 0x000
ID = 0
Start = 20
End = 25
Duration = 6 samples

6.4 Additional work

Add a second bus connection to the System ILA. In this case, debug as well the segment of the AXI4 bus between the Zynq7 and the AXI interconnect. Synthesize the design and place a trigger exactly as in the previous examples, but modify the trigger position to the middle of the window (512 samples). Then, compare the addresses used in the first segment with those used in the second segment. Try to understand the differences.

Modify the software application so that it writes to the LEDs only when the value of the switches changes. That is, instead of writing in every while iteration, write only if the status of the switches has changed since the previous write operation to the LEDs. Then, prepare the System ILA to trigger at a write event at the specific address of the LEDs register (offset 0). Check that the System ILA does not trigger until a button is pressed and the application performs a write.