2025

FPGA Development Guide

USING XILINX ULTRASCALE+ MPSOC

FPGA and MPSoC Guide

1.Introduction to FPGA

What is an FPGA?

A field-programmable gate array (FPGA) is a reconfigurable silicon chip that can be programmed to implement almost any digital circuit. An FPGA might seem similar to a conventional CPU. But they differ in the architecture level. We'll discuss the architecture of an FPGA and see how it's different from a microcontroller or a microprocessor.

The need for FPGA

Flexibility and reconfigurability: FPGAs can be reprogrammed anytime, allowing developers to update or modify functionality without changing the physical hardware. This is ideal for rapid prototyping, updates, or adapting to new requirements.

Hardware prototyping: FPGAs provide a practical way to test hardware designs before manufacturing custom chips (ASICs). They help verify performance, functionality, and reduce design risks early in development.

High performance and low latency: With parallel processing capabilities, FPGAs can handle multiple tasks simultaneously. This leads to faster execution and minimal delay, making them perfect for real-time and high-speed applications.

Customization: FPGA logic can be tailored exactly to the needs of a specific application or system. This allows for optimized performance, reduced power usage, and better control over system behavior.

FPGA Building blocks:

Typical FPGA building Blocks consisting of three major elements:

Configurable logic Blocks (CLB): The blue color blocks present in the figure are the CLB's which are responsible for the implementation of logic functions,

Interconnects: Interconnects are used in FPGA to connect the long and short interconnection wires together in a flexible combination. It also contains the transistors to turn on/off connections between different lines.

Input/Output Blocks: Available in FPGAs periphery facilitate signals to come into or from the FPGA chip.

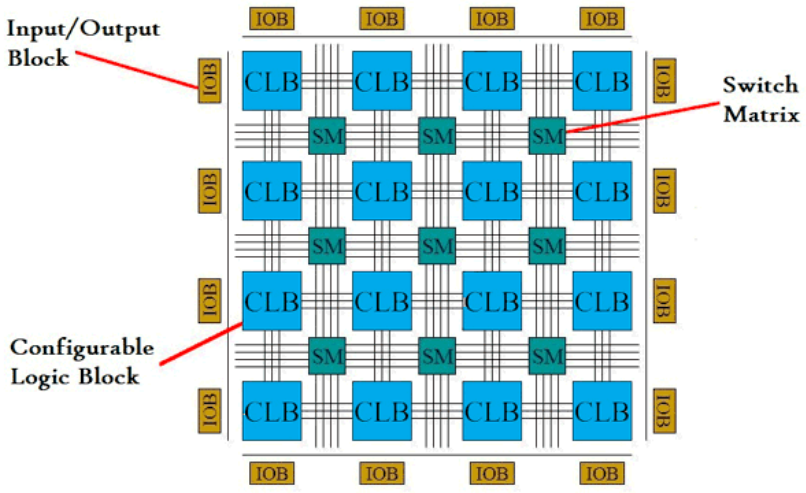


Figure 1: Internal structure of FPGA fabric

Configurable Logic Block: CLB

CLB (Configurable Logic Block). These are the main "building blocks." They do the actual logic work, like adding, comparing, or checking conditions. They can be used as both- combinational or sequential logic blocks. The components of a CLB are:

LUT (Look-Up Table): LUT is a small memory. Like a small truth table that tells the circuit what output to give based on inputs. This LUT is configured when an FPGA is configured. To implement an n-bit combinational logic, the size of the LUT should be 2n bits.

MUX (Multiplexer): Chooses which output to use. Its main purpose is to bypass the flip flop, configuring the CLB to act as either combinational or sequential block.

D Flip-Flop: Remembers or stores a value for timing or sequences (used for sequential circuits).

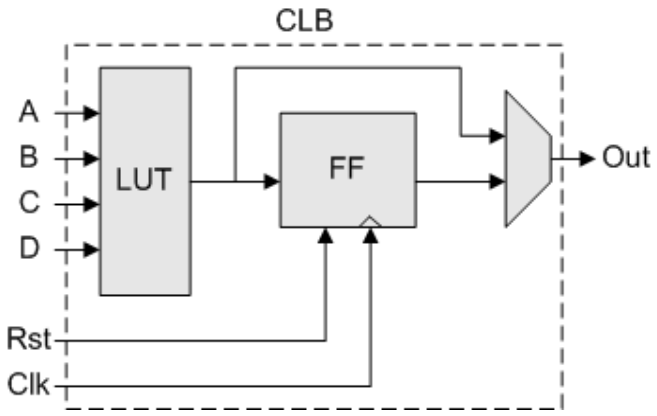
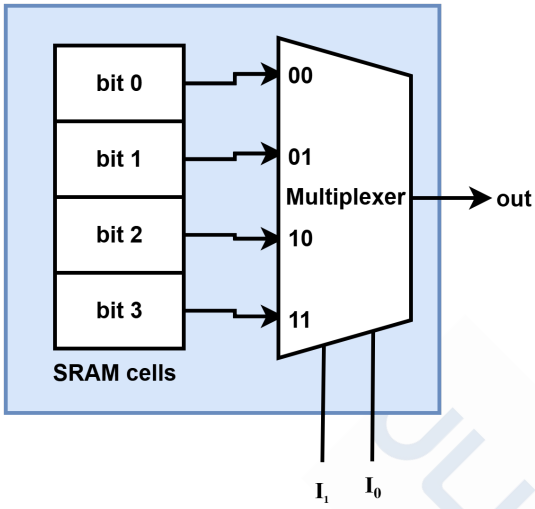


Figure 2: Structure of a CLB

Look-up table

Consider a digital memory which has 1 column and 4 rows. Each cell can store 1 bit data. This data can be accessed at the out terminal using address lines- I_0 and I_1.



I_1	I_0	out
0	0	0
0	1	0
1	0	0
1	1	1

If we somehow store this value in the memory as shown above, the relation between input (address lines) and the output terminals are given on the right. They act as AND gates. So, this memory emulates AND gate. The memory can be used to emulate any 2-bit combinational logic.

A 4-bit memory can emulate any 2-bit combinational logic.
A 2n-bit memory can emulate any n-bit combinational logic.

A LUT is typically built out of SRAM bits to hold the configuration memory and a set of multiplexers to select the bit of CRAM that is to drive the output.

To implement a K-input LUT i.e., a LUT that can implement any function of K-inputs. For that 2k SRAM bits and a 2k:1 multiplexer is needed. The Figure beside shows a 4-input LUT which consists of a 16 bit-SRAM and a 16:1 mux implemented as a tree of 2:1 mux.
Connecting LUTs: To emulate combinational circuits with more than 2 inputs, we interconnect LUTs using MUXs.

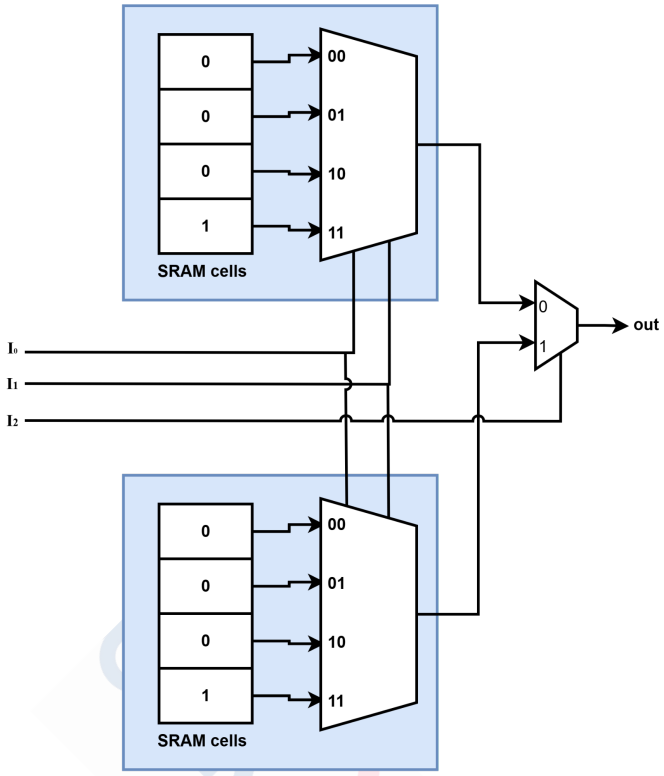


Figure 3: CLB implementing a combinational logic

What's special about the LUT (Look-Up Table)?

The LUT is the brain of the CLB. It stores bits to emulate simple logic functions (like AND, OR, NOT). Depending on the chip, a LUT can take 3, 4, 6, or even 8 inputs and produce one or two outputs. Some advanced FPGAs have adaptive LUTs that can do more than one function at the same time.

A LUT is combined with a flip-flop and a 2:1 mux to implement CLB (Configurable Logic Block)

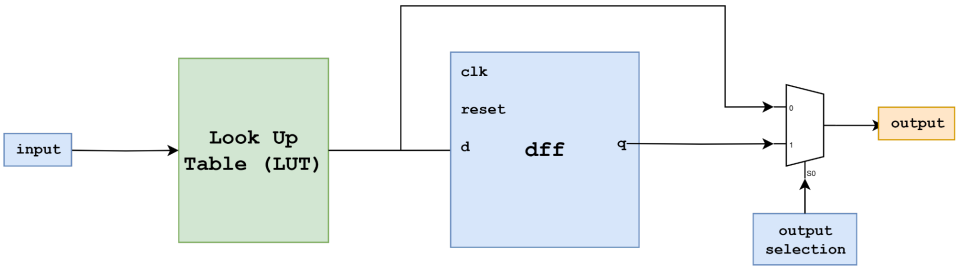


Figure 4: LUT with a DFF, for sequential logic

CLB is the basic structural unit for Xilinx FPGAs (The term CLB is not generic and is used exclusively for Xilinx FPGA) CLB helps in implementing both combinational and sequential circuits.

Modern FPGAs are much more complex, Xilinx ZYNQ Ultrascale FPGAs have 8 LUTs and 16 flip-flops in a CLB. Each LUT has 6 inputs and 2 outputs.

Interconnects

Interconnects are used to create connections between CLBs, I/O blocks and other FPGA elements. The interconnect consists of switch boxes and several wires. The wires connect CLB inputs and output with the switch boxes. The switch boxes contain a crossbar array, which enables connecting a wire coming from any direction to any other direction. The transistors that control the connection are controlled by BRAM which are configured when the FPGA is programmed, transforming the device to act as specified hardware.

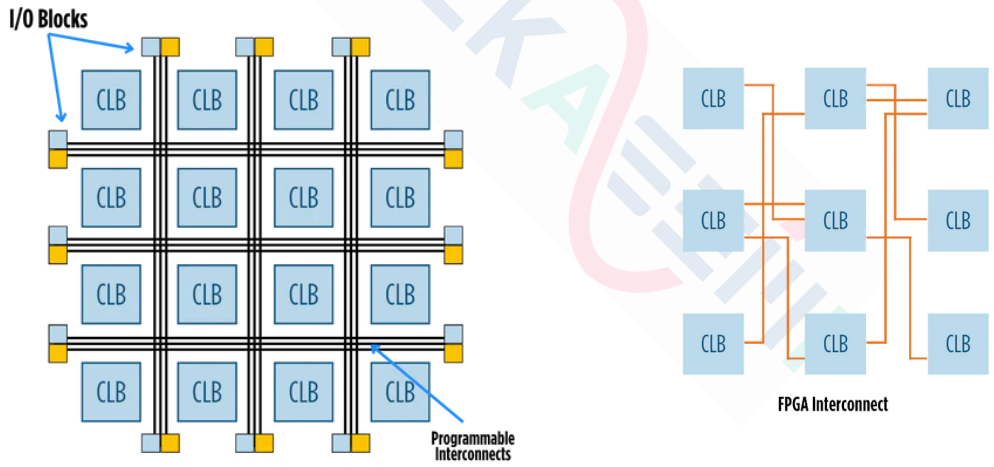
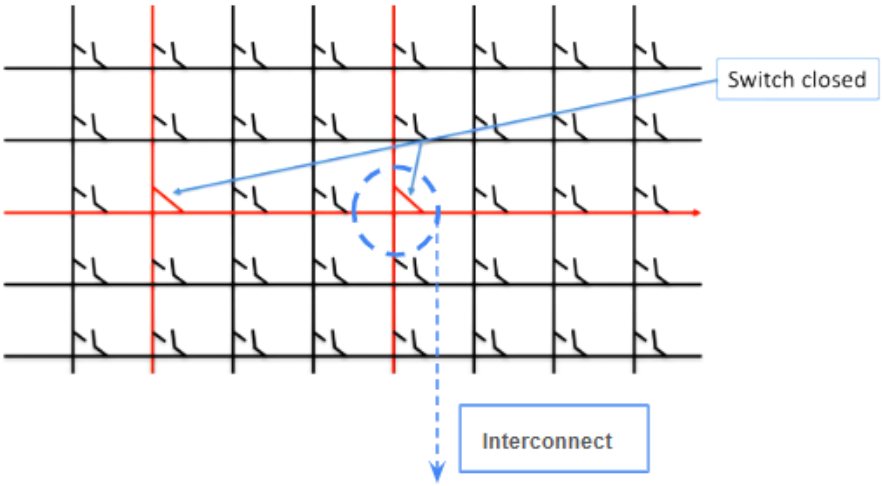
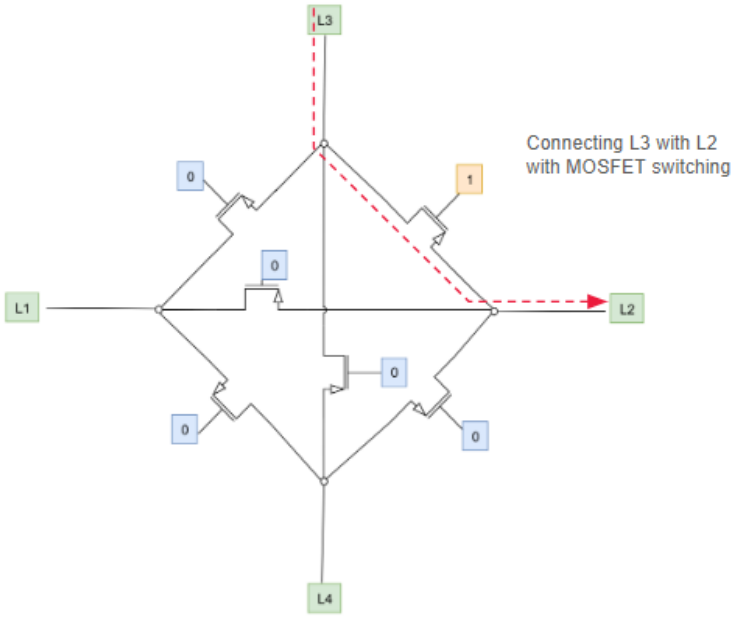
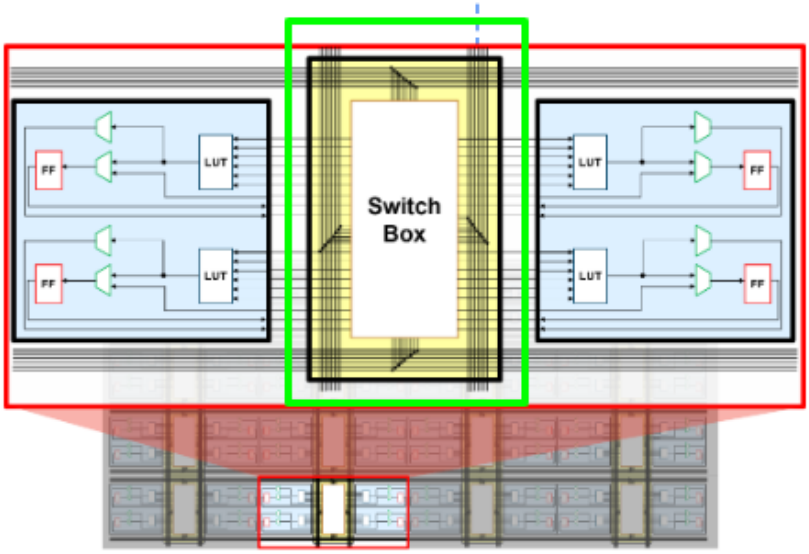


Figure 5: Interconnects altering the connection between the CLBs

After configuring the interconnects, the CLBs can be connected in any manner. This is a key aspect of FPGA. In this way, it can alter how the device behaves at the hardware level.



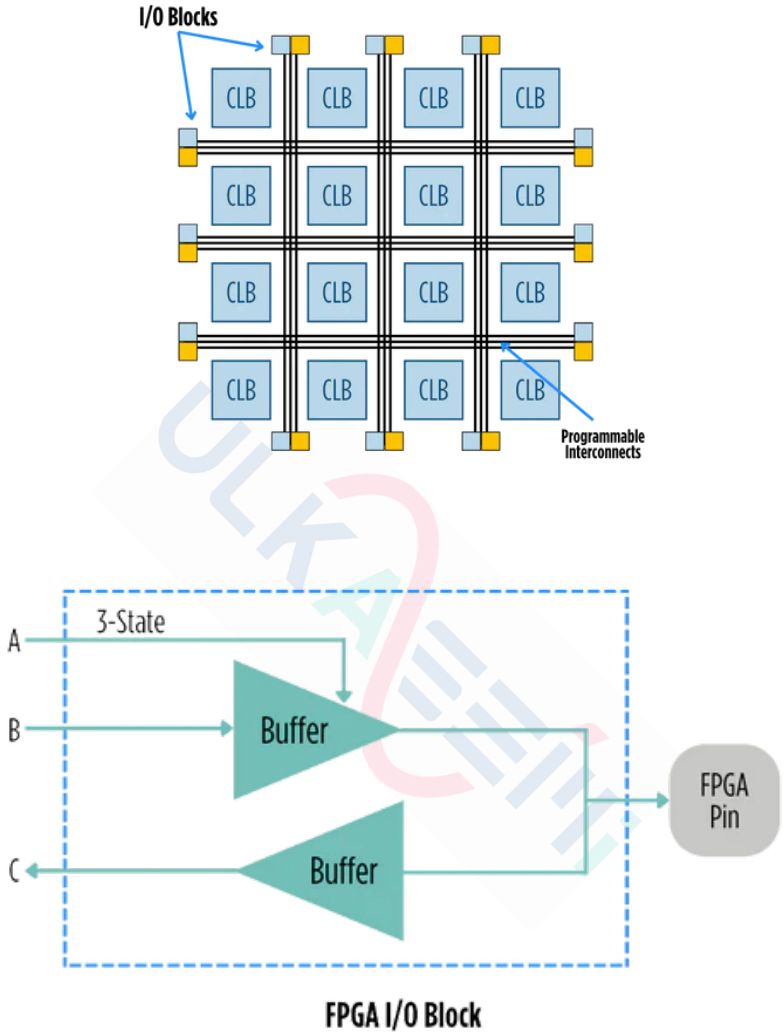




A bunch of them combined together forming switch box

I/O block:

Located next to every physical input or output pin, these blocks connect the internal logic of the FPGA to the external environment. They can be programmed to act as inputs, outputs or tri-states, enabling the FPGA to communicate with external devices and systems.



FPGA I/O Block

FPGA Vs CPU

What is a CPU?

CPU – Central Processing Unit

- General-purpose processor
- Executes instructions sequentially
- Optimized for flexibility and software compatibility
- Runs operating systems and high-level code
- Limited parallelism (few cores)

Example:

Performs tasks like:

Fetch ⟶ Decode ⟶ Execute for each instruction

What is an FPGA?

FPGA – Field-Programmable Gate Array

- Reconfigurable hardware
- Executes logic in parallel
- Customized circuits designed with VHDL or Verilog
- Great for real-time and high-throughput applications
- Logic blocks and routing are user-defined

Example:

You can create 10 adders to run at the same time

Feature	**CPU**	**FPGA**
Execution	Sequential	Parallel
Programming	C/C++, Python, Java	VHDL, Verilog
Flexibility	General purpose	Task-specific, reconfigurable
Latency	Varies, non-deterministic	Very low, deterministic timing
Power	Higher, less efficient	Efficient for specific workload

2.MPSoCs

A multiprocessor system-on-chip is a heterogeneous computational device. It consists of Processor, FPGA fabric, GPUs. Many variants even contain specialized engines for ML/DL application, video/image processing.

MPSoC structure

Multiprocessor System on Chip. The entire ZYNQ Ultrascale+ chip is an MPSoC. An MPSoC mainly contains:

1. Processing system (**PS**)
2. Programmable logic (**PL**)
3. Power management unit (**PMU**)
4. Security logic (encryption)
5. Bridges. (PS - PL)

Processing system: It's a part of the MPSoC. It's the fixed function hard logic inside the MPSoC. Meaning, the hardware can not be configured like FPGA fabric. It's basically a CPU with various components such as:

1. Application processing unit (**APU**)
2. Real-time processing unit (**RPU**)
3. Memory controller (DDR, LPDDR)
4. Peripherals (I2C, UART, USB, SPI, Ethernet)
5. Boot logic (What to do when the chip starts up)
6. AXI interfaces (To connect with PL)

The ZYNQ Ultrascale+ has multiple cores for application processor (ARM cortex A53) and real-time processor (ARM cortex R5). There is no direct connection between the PS peripherals and Programmable logic. The whole thing can be visualized as follow:

MPSoCs Architecture Block Diagram:

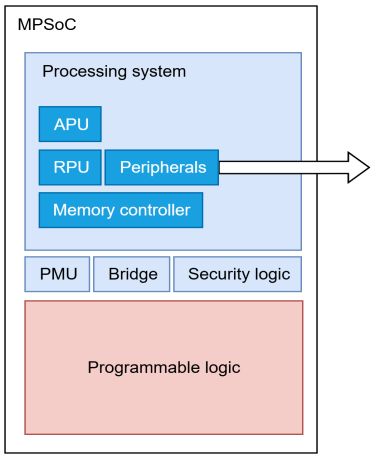


Figure 6: Overall structure of a MPSoC

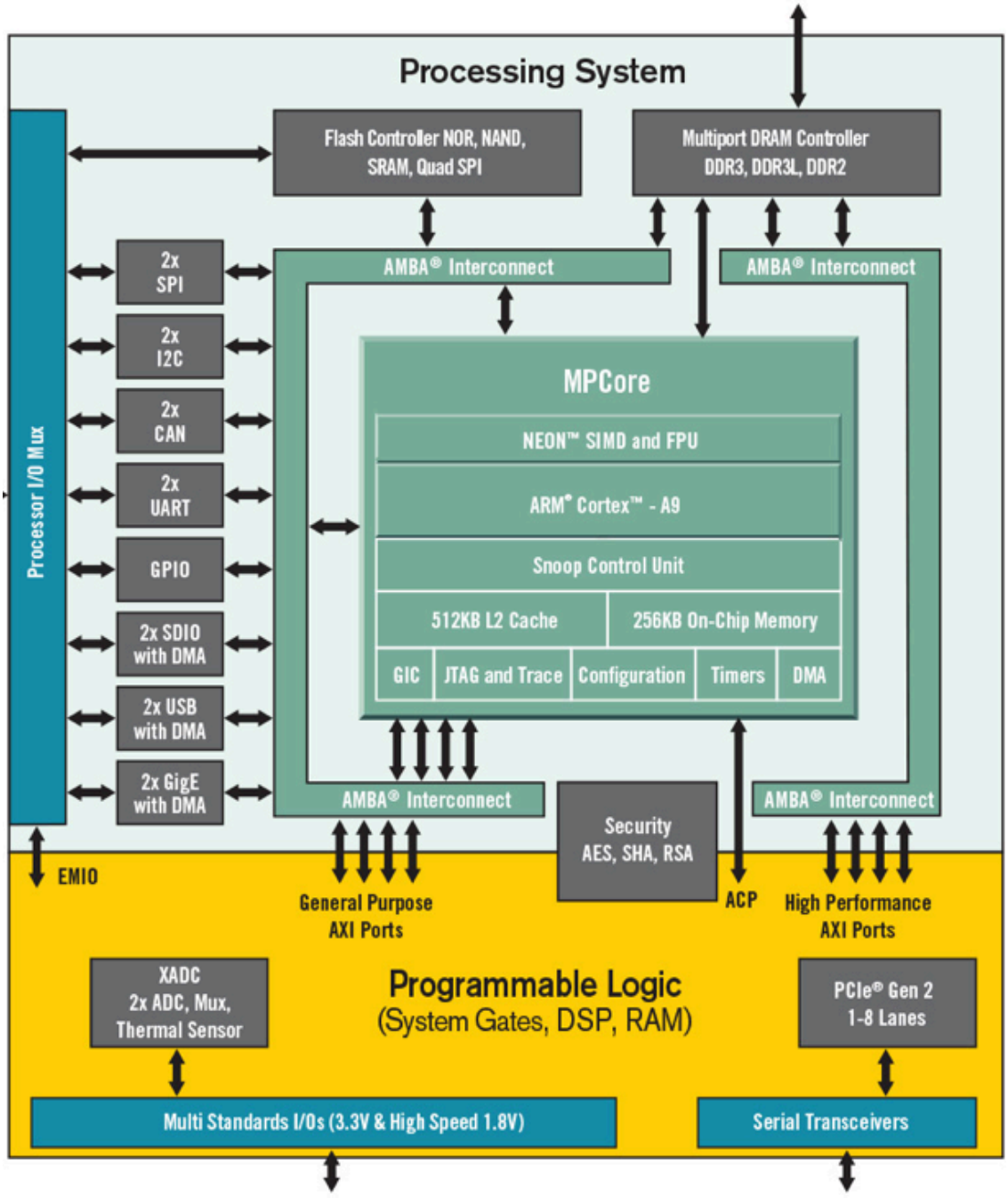


Figure 7: Detailed structure of a Xilinx MPSoC

MPSoC operation overview

MPSoC has to separate regions depending on the hardware architecture and they run differently

PS: The processing system is a hard-logic device. Its hardware can not be reconfigured and runs the 'software' we provide it.

PL: Programmable logic is the FPGA fabric. The hardware can be reconfigured to emulate any digital circuit.

The high-level view of the MPSoC operation can be described as follows:

1. First, we design the hardware, which is to be implemented in the FPGA fabric or PL.
2. This design (after verification, synthesis and exported as bitstream, resulting in a .bit or .xsa file) is then uploaded to the device board using bootloader.
3. To control the hardware uploaded on the FPGA fabric, we write software code in Vitis and upload it to the PS.
4. The PS controls (configure, read and write data using AXI interface, memory mapped register* or interrupts) the hardware in PL

The PS doesn't come with an operating system. It only has basic boot configuration burnt into it. No file system or scheduler.

When an application is uploaded to the board using Vitis platform, it provides linker scripts, low-level drivers to run the application. This is called a **bare-metal application** - No operating system, just the code running at full control.

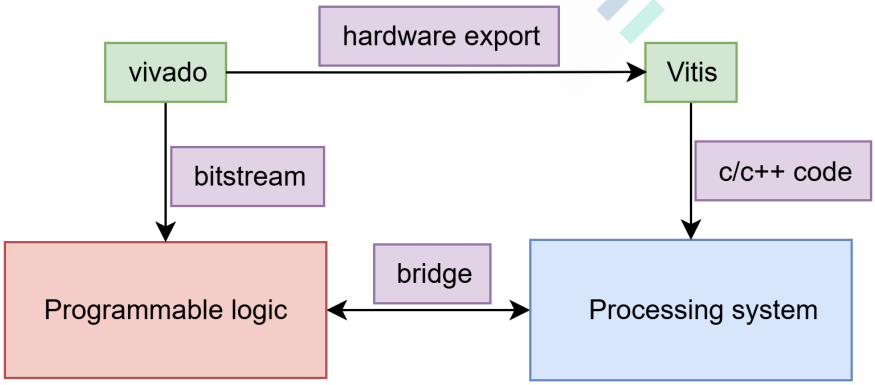
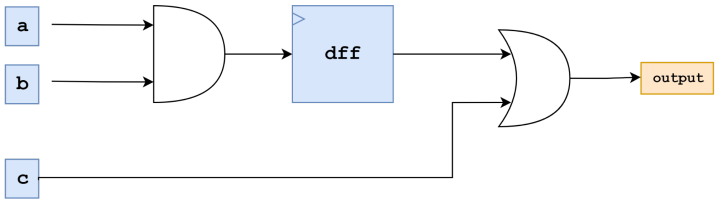


Figure 8: Overview of MPSoC workflow

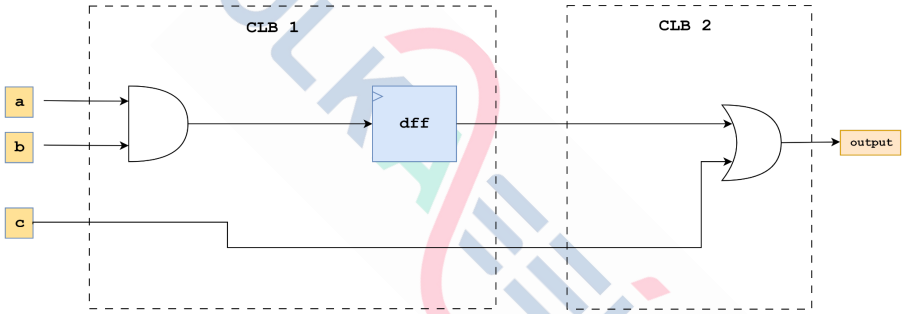
3.RTL to Bitstream Flow

Now, let's see how a simple RTL code is translated to something that can be uploaded to the FPGA (a bitstream file, which contains information about the RTL, fabric configuration so it can emulate that RTL)

Consider a logic circuit as below:



The Vivado design suite tool upon synthesis, implements the whole logic circuits with two CLBs.



In reality, the CLB and interconnects are configured in the following manner-

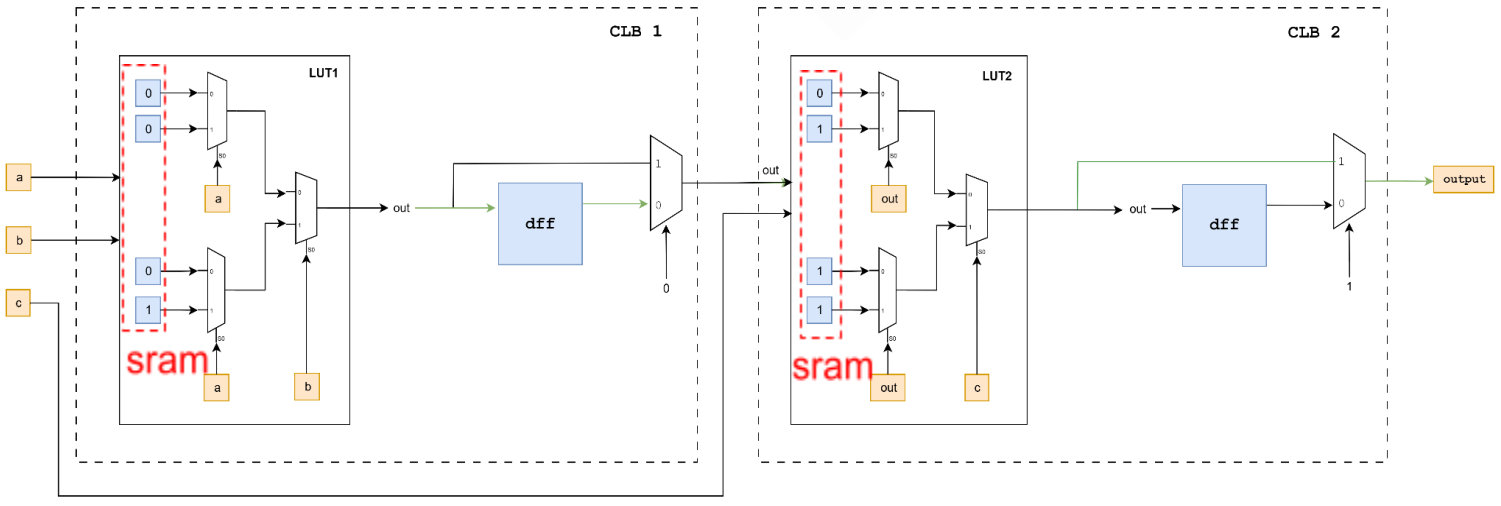
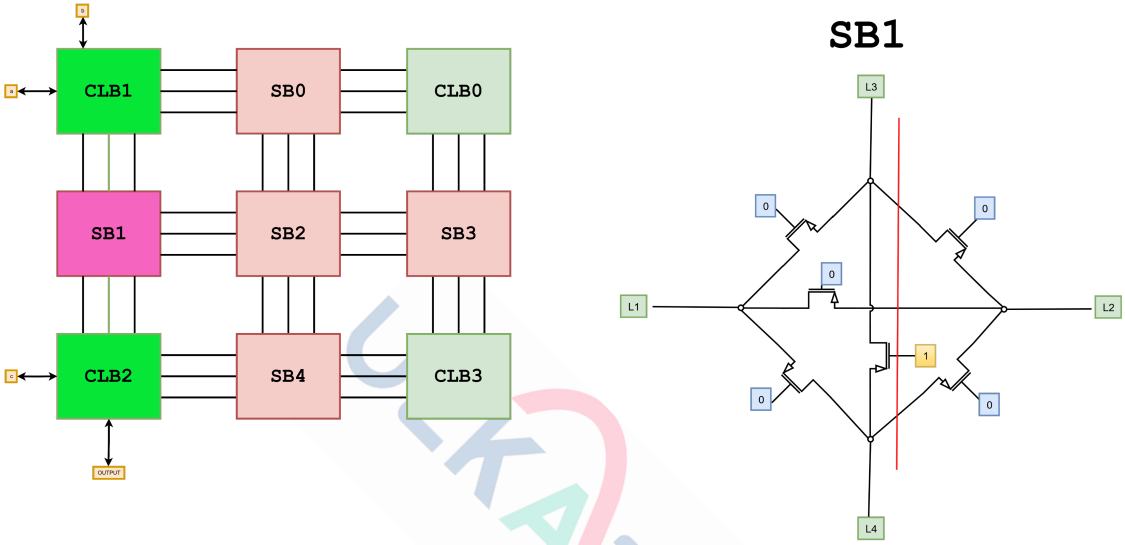


Figure 9:Synthesis tool using CLBs to implement circuit given RTL code

The bitstream generated from Vivado design suite loaded in FPGA is used for:

- To load the SRAM memory to configure the LUT function generator of the respective CLBs.
- Bits generated configure whether outputs are sequential or combinational in nature through the selection pin of the output mux.
- Bits generated configure the switch matrix of the Switch Block to interconnect multiple CLBs and establish the required logic.



Inside the SB1, the MOSFETs are switched in such a way that it connects **L3** and **L4**, which establishes the desired connection between CLB1 and CLB2 through the Switch Block **SB1** to implement the previously mentioned logic circuits.
See here for details

For Input (a, b, c) = (1, 0, 1)

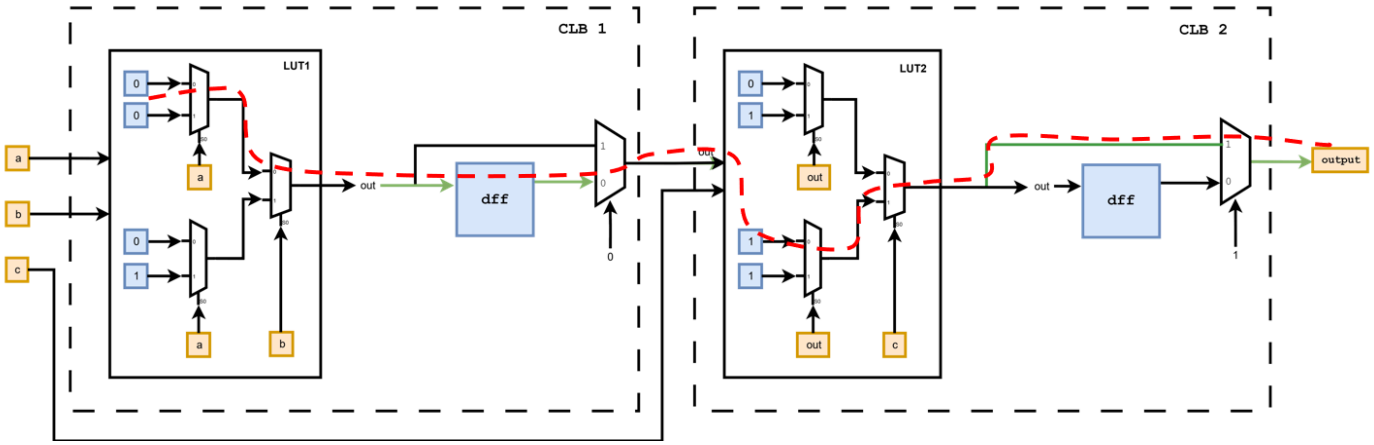
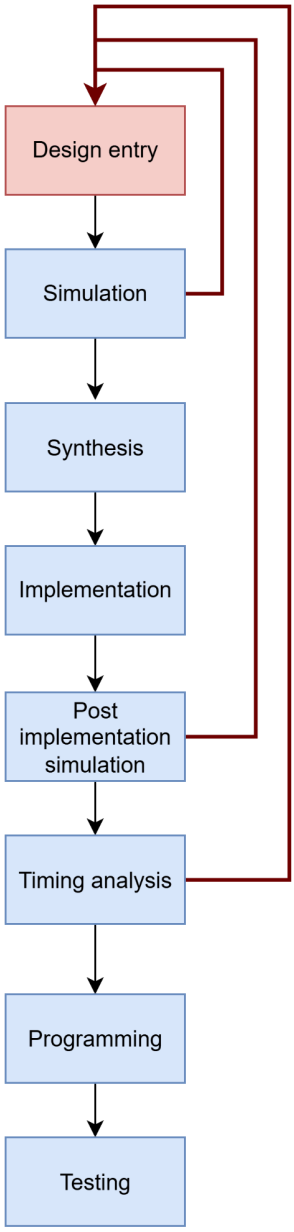


Figure 10: Data flow to output for a given input in CLBs

4.FPGA Design Flow

Developing FPGA usually follows this sequence:



Design entry consists of-

- Different entry points
- Schematic
- Hardware Description Languages (HDL) • High
- Level Synthesis (HLS)

Constraints

- Clock frequency
- Signal delay
- Power consumption
- Connection to pins of FPGA

Simulation: RTL simulation for-

- Verifying functionality of design
- Fast simulation
- No delay or resources information

Synthesis:

- Translates/maps design to device recourses
- Lock-Up tables (LUTS) Flip-flops
- Latches
- Adders Multipliers • Memory
- Generates a netlist of the design Resource utilization
- Coarse delay
- Coarse power consumption

Implementation:

- Optimization
- Placement
- Route wires
- Route wires to external pins Generate configuration file
- Functional simulation of the design
    - Simulation closes to the actual
    - Hardware

Post implementation simulation:

Simulation closes to the actual hardware

Timing analysis:

Can the system run at intended clock frequency?
Delay of signals

Programming:

Configuration is transferred to the FPGA
Most FPGAs use SRAM to store the configuration. SRAM is volatile

Testing

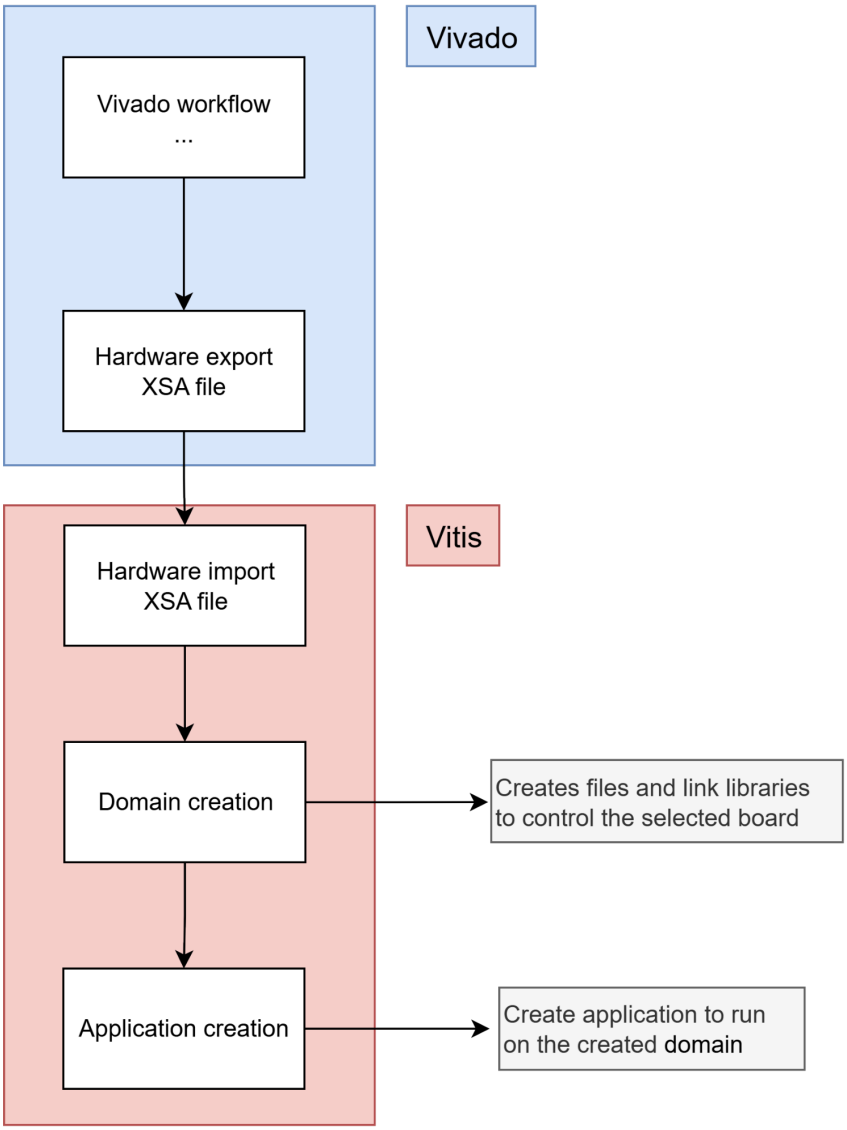
Verify that the design works as intended

- Pattern generators
- Measure power consumption

5.Vivado & Vitis

The AMD Vivado is an integrated design environment (IDE) and software from AMD for designing and developing hardware using HDLs for SoCs and FPGAs.The software supports the entire hardware design flow, including **design entry, simulation, synthesis, implementation** (PnR), high-level synthesis (HLS), IP integration etc.

The AMD Vitis unified software platform combines all aspects of AMD software development into one unified environment. The Vitis software platform supports both the Vitis **embedded software development** flow and the Vitis **application acceleration** development flow.



Vivado terminology

Block diagram

Block diagram is where we create designs, bring IPs from the library and our custom IPs (RTL blocks) together. It's a visual editor.

HDL wrapper

The block diagram is converted to a HDL file. This file is called HDL wrapper.

Constraint file (XDC)

If an IP is interfacing with external peripherals on the board, its I/O terminals are connected with physical pins of the board. These pins require voltage, clock and other configurations to operate correctly. These configurations are stored in an XDC file.

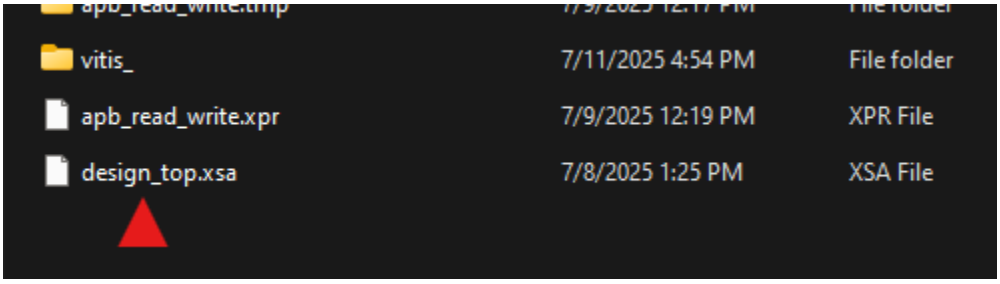
Vitis terminology

Workspace

A workspace is a directory location used by the Vitis unified software platform to store project data. It's basically a folder.

XSA

XSAs are exported from Vivado. It has the hardware specifications like processor configuration properties, peripheral connection information, address map, and device initialization code and custom hardware.



Platform

The Platform is a combination of hardware components (XSA) and software components like domains/BSPs, boot components such as FSBL (First stage boot loader), and so on. It basically tells the application-

1. The PS configuration.
2. Custom IP on the PL side.
3. Their configuration, bridge,
4. Clocks, resets, memory map.
5. Drivers for peripherals.

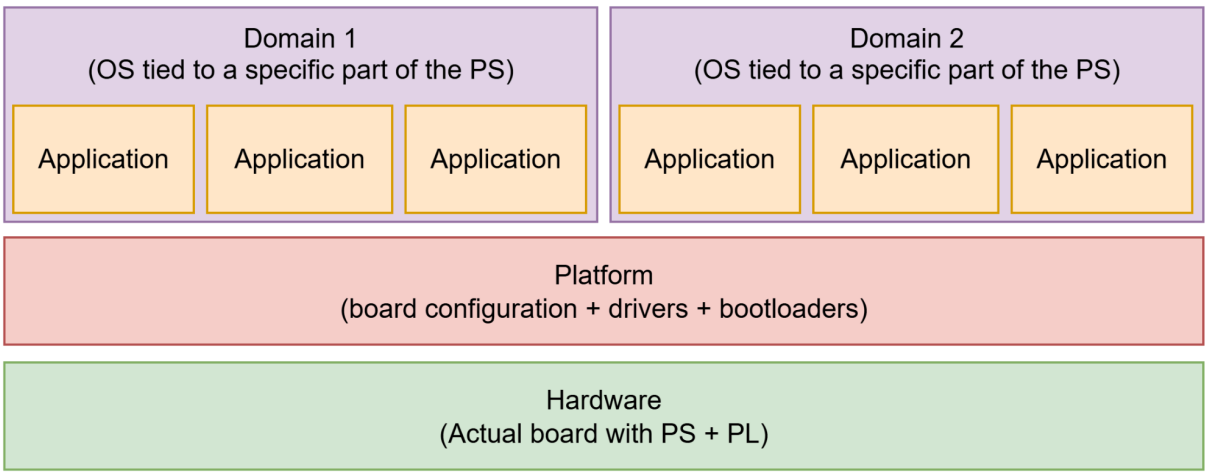
Domain

A domain is a board support package (BSP) or the operating system (OS) with a collection of software drivers on which to build our application. In essence, a domain is hardware + its OS.

Application (Software Project)

An application is a software code that runs on a selected domain.

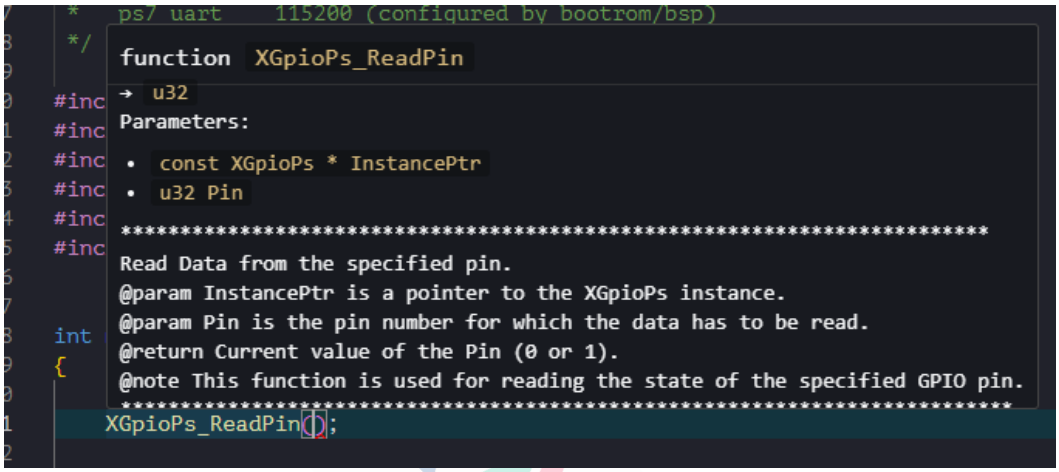
The overall structure of an application running on a hardware device is as follows-



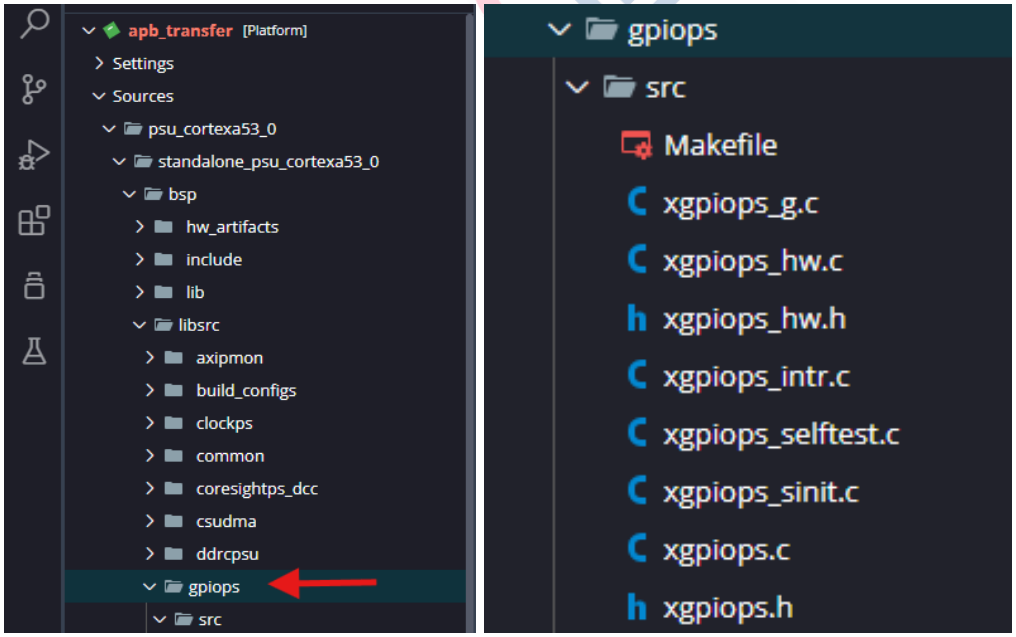
Navigating Vitis documentation

To go through the documentation, we follow the header file of the particular library we want to know about. Here's how-

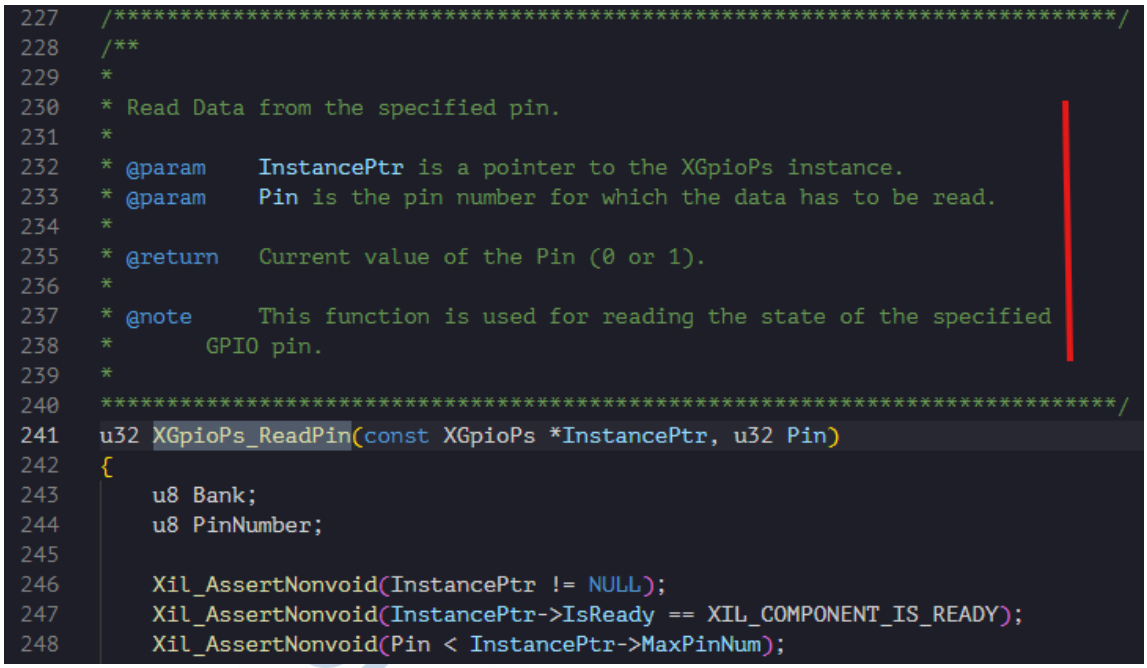
1. At the beginning, we hover over the function and see if it reveals the documentation for that particular function. (It does that if the header file contains the function prototype and the documentation)



2. Otherwise, we go to the definition of the header, the actual C file. For example, if we want to know about the function "xgpio_readpin" from the xgpio.h header, we go to the xgpio.c file which is located under platform > Sources > PSU* > standalone* > bsp > librc.



3. Next, we try to find the function definition.



```
227  /*****************************************************************************/
228  /**
229  *
230  * Read Data from the specified pin.
231  *
232  * @param    InstancePtr is a pointer to the XGpioPs instance.
233  * @param    Pin is the pin number for which the data has to be read.
234  *
235  * @return   Current value of the Pin (0 or 1).
236  *
237  * @note     This function is used for reading the state of the specified
238  *       GPIO pin.
239  *
240  ******************************************************************************/
241  u32 XGpioPs_ReadPin(const XGpioPs *InstancePtr, u32 Pin)
242  {
243      u8 Bank;
244      u8 PinNumber;
245
246      Xil_AssertNonvoid(InstancePtr != NULL);
247      Xil_AssertNonvoid(InstancePtr->IsReady == XIL_COMPONENT_IS_READY);
248      Xil_AssertNonvoid(Pin < InstancePtr->MaxPinNum);
```

This is the functionality, parameter definition, return type and remarks.

NOTE: Not all functions in the standard libraries contain documentation. Good luck👍

6.Integrated logic analyzer (ILA)

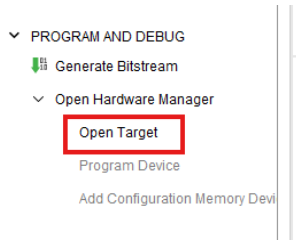
ILA is a tool we use to retrieve the waveform of the signal inside the PL fabric in real-time.

To use it, we instantiate an ILA in the PL fabric. Usually we use the native monitor type

We can change the port number, width of the port and how much data it can sample. Usually we keep it around 2048. ILA is like an oscilloscope. It continuously samples data. So if you want to see data at a particular point in, you need to set up a trigger. That comes after you generate bitstream.
Once the bitstream and XSA is generated, an application in the vitis is written, run it in the vitis console.

NOTE: Put a ***scanf*** at the beginning of your app when testing. This way, the program is halted until you've initialized the ILA from Vivado.

After starting the app run, go to Vivado, open target - auto detect



Then program device-> select your bitstream and program.

Design Lab

Prerequisites

Hardware-

- Xilinx MPSoC. (preferably Zynq Ultrascale+ or any other board with a hardcore processor)

Software-

- Vivado design suit
- Vitis unified IDE

Vivado installation caution

Install Visual C redistributable (VCRedist) beforehand. In the Vivado installation, select **Vivado ML standard edition**. Select the board you have.

Lab-1: Implementing blinking LED on FPGA board (PL fabric only)

Title: Implementing blinking LED on FPGA board (PL fabric only)

Objective

- Control 4 LEDs present in the Zynq Ultrascale+ MPSoC using an FPGA board in a cyclic manner.
- Design and implement a project in the Vivado IDE.
- Perform Block level integration between MPSoC in PS side and RTL block in PL side in Vivado design suite
- Develop XDC file to configure the IP to use external pins (LEDs).

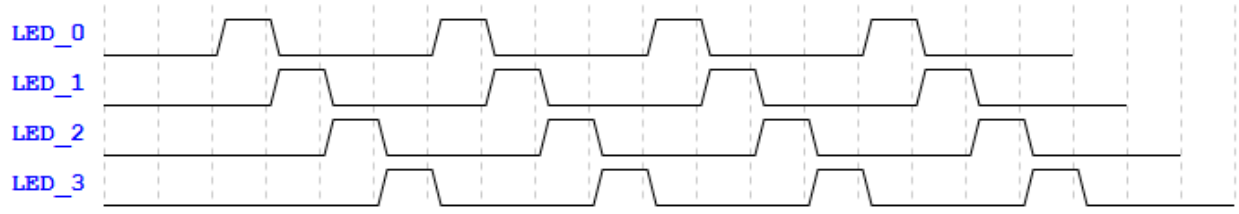
Functional Description

The objective of this project is to control the LEDs in a cyclic manner, through PL fabric only. The timing of the LED is- 1 sec on, 3 sec off.

Block Diagram

Added task

Configure the four LEDs to run in cyclic mode as follows:



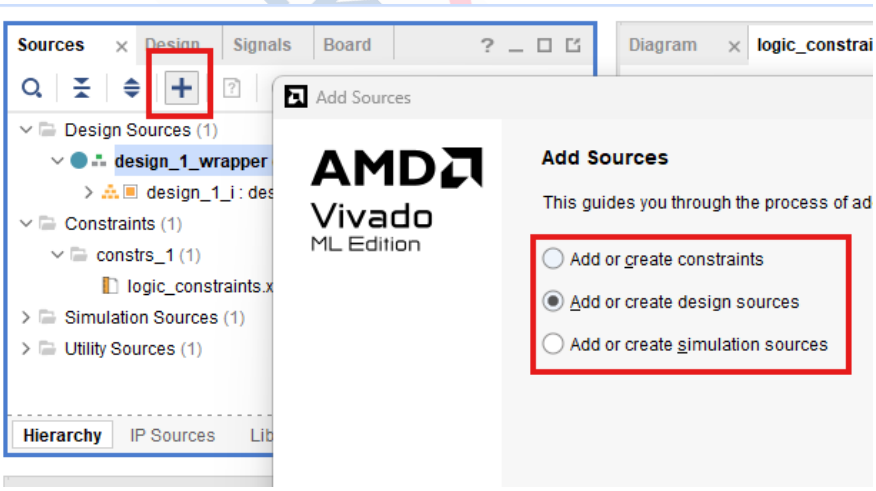
Solution

Vivado guideline

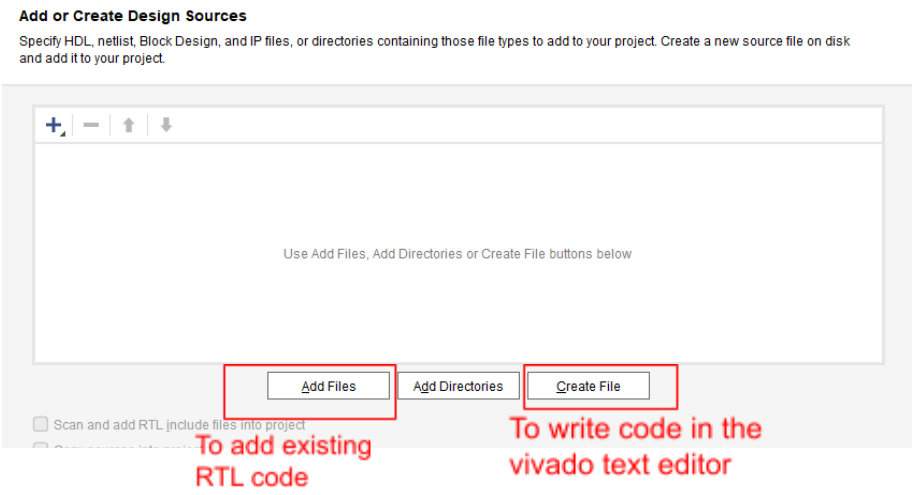
This workflow goes through the steps to complete the design specified for Lab-1. We go through creating a new project, setting up a block diagram, adding constraints file, running synthesis and implementation and finally generating bitstream.

Step 1: Creating a new project

1. Go to "**Create Project".**
2. Provide "project name" and "project location".
3. Select the "RTL Project" option and then next. Then we were to select the board over which the RTL is going to be implemented.
4. Go to **Board →Name →Zynq Ultrascale+ ZCU104 Evaluation Board** and click on next.
5. If all information is provided correctly then a UI of below mentioned snippets will get shown with all the information. Click on **Finish** to create the project.
6. Go to the "Add Sources" option. Select "Add or create design sources" to write/add the RTL code and **"Add or create simulation sources "** to write/add testbenches.



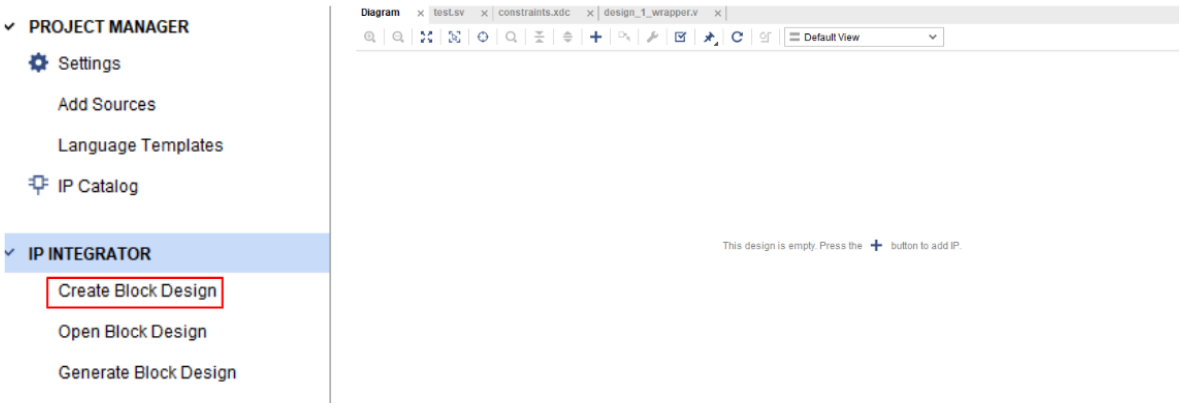
You can add existing RTL files or create new ones and edit them in the built in editor.



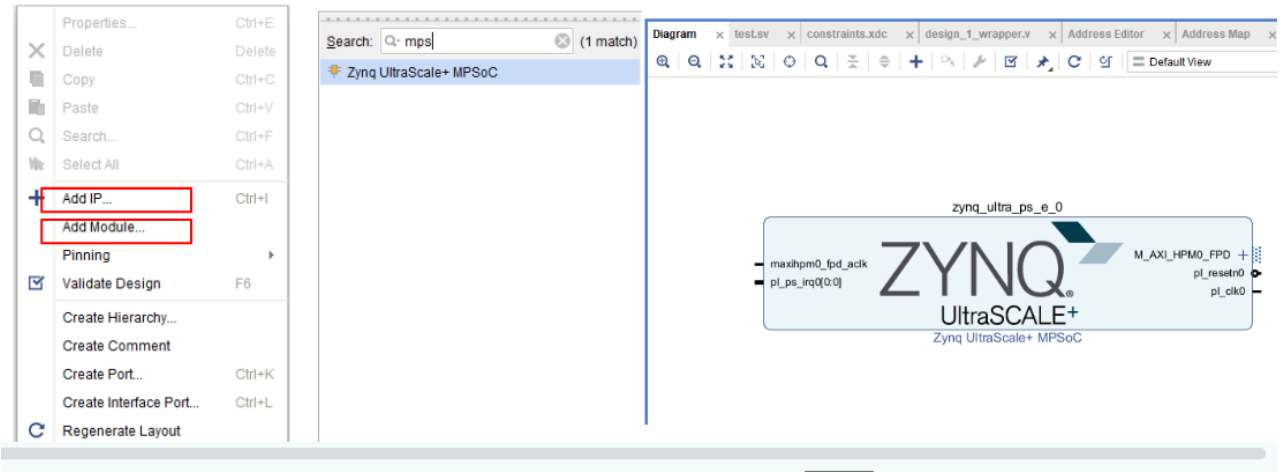
Step 2: Creating block diagram

Block level Design and integration of RTL module with Processing System

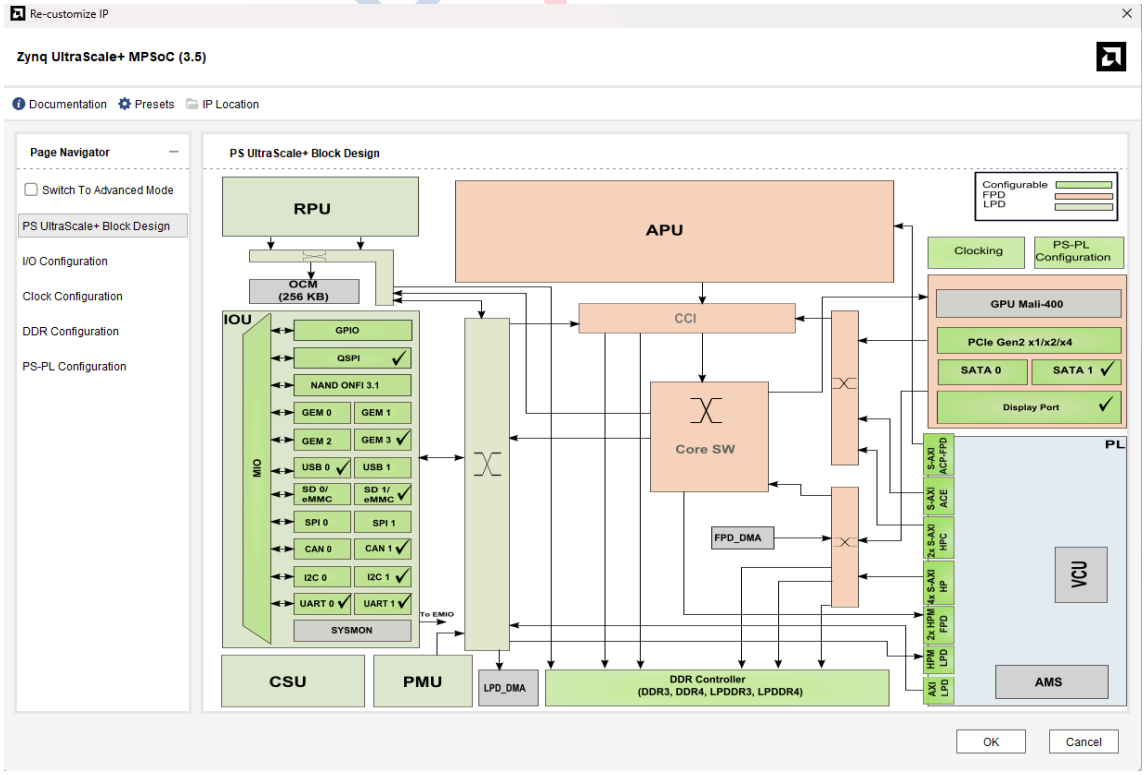
1. Select Create Block Design option from the flow navigator tab.
2. It will open the Block Diagram window with the configured name.



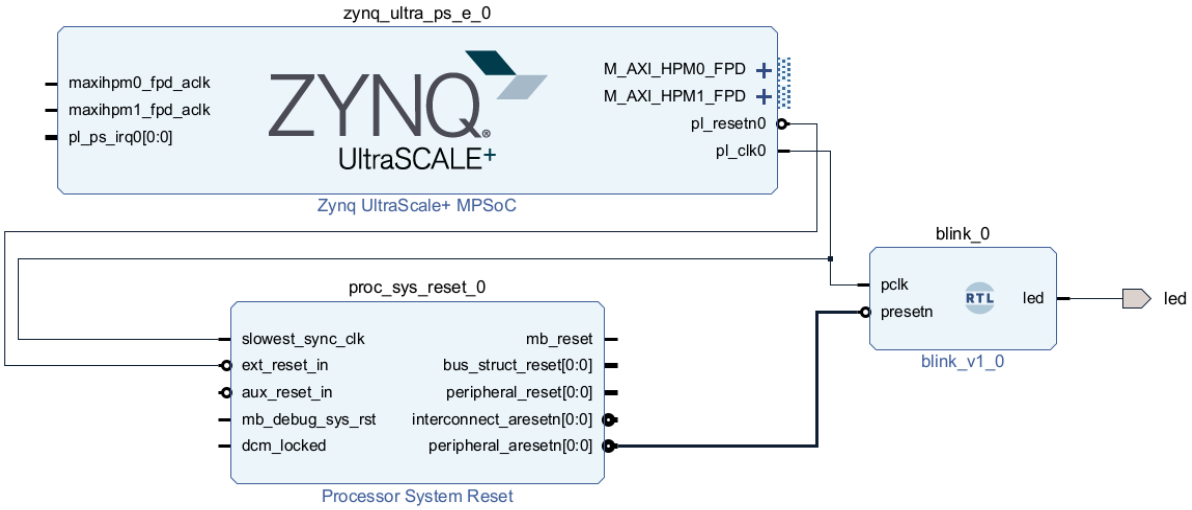
3. Right click on the Diagram window. Select Add IP.
4. Search for the **Zynq Ultrascale+ MPSoC** PS unit with which the RTL module configured in the FPGA fabric of the PL unit is connected.
5. Next click on Add Module and select the designed verilog model. Note that only verilog type is compatible for block design.



7. Right clicking on Zynq -> run automation. Double clicking on the Block will open the IP customization window, where we can configure the PS. Leave it as it is for now. Necessary things for PS to work properly are DDR, UART0, QSPI etc. These are automatically configured when we run “run automation”.

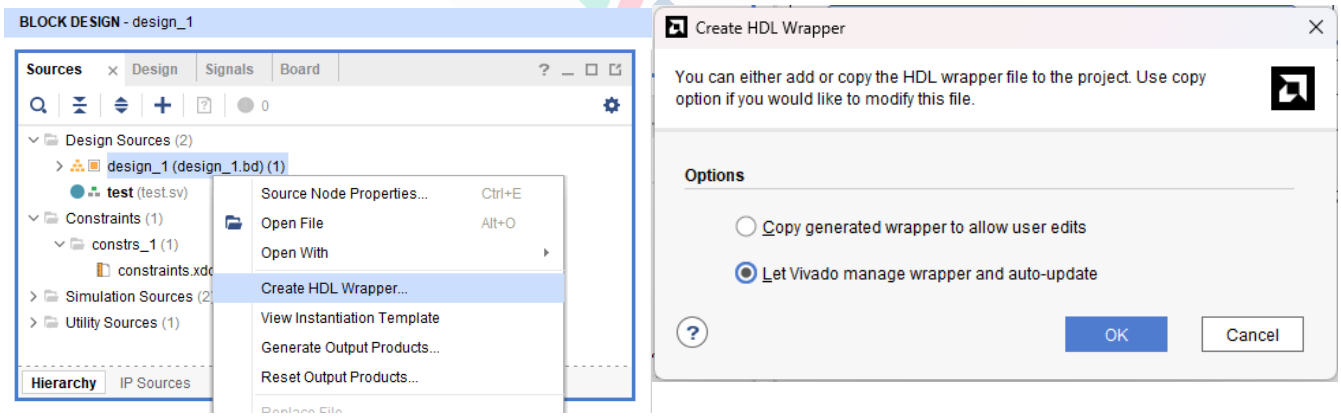


8. Next, we drag and drop the RTL from the hierarchy viewer. It should be a verilog file or a system verilog file wrapped in a verilog wrapper.
9. Then, we connect the ports. Remember to instantiate processor system reset to use the Zynq reset through it. The block diagram looks like this-



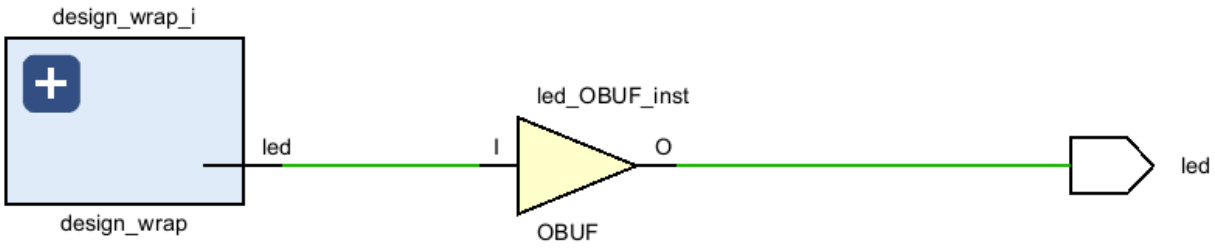
Step 3: Wrapping things up

1. After creating the block diagram, we create an HDL file that represents the block diagram instantiations and connections. This is done by creatine HDL wrapper. Use the following steps to create HDL wrapper:



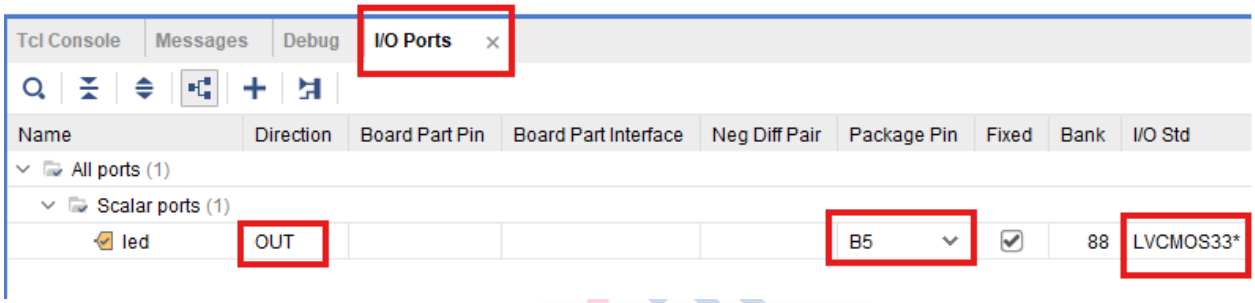
2. After creating the wrapper, the design is almost complete. At this point, we can run simulations using our testbench files. Simulation files can be added by using add source -> add simulation files. Timescale has to be determined in each verilog file used in the simulation. Then in the flow navigator, we can run simulations. Although it is recommended that, simulations should be done in robust tools like simvision.

3. Next, we run the synthesis that generates the actual circuit that is to be uploaded on the board.

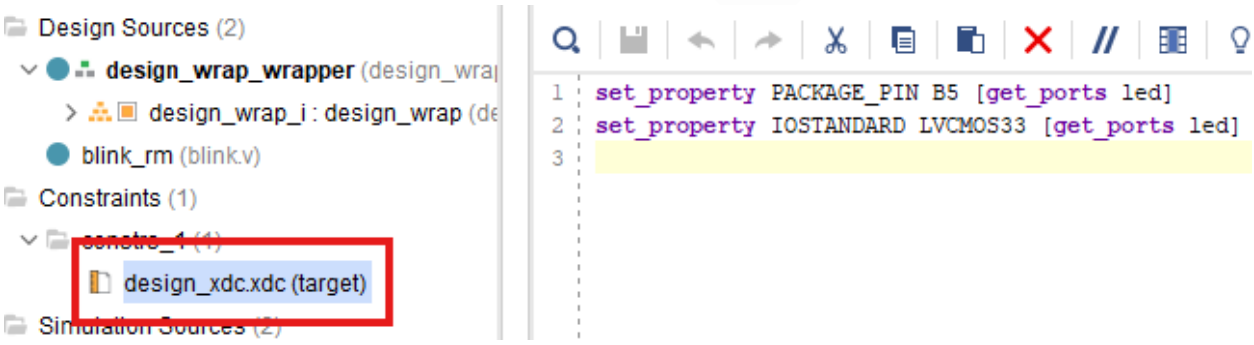


Step 4: Adding constraints

We add design constraint files (.XDC): It can be done in two ways. In the first way, we open the schematic, then go to windows -> **I/O ports** and select the I/O configurations manually. For LEDs, it's **OUT**, **LVCMOS33**. The exact pin number is specified in the User guide.



Or, we can add a constraint file from the add file menu. Using the master XDC file (check drive), we can easily configure it by commenting out things that we don't need.

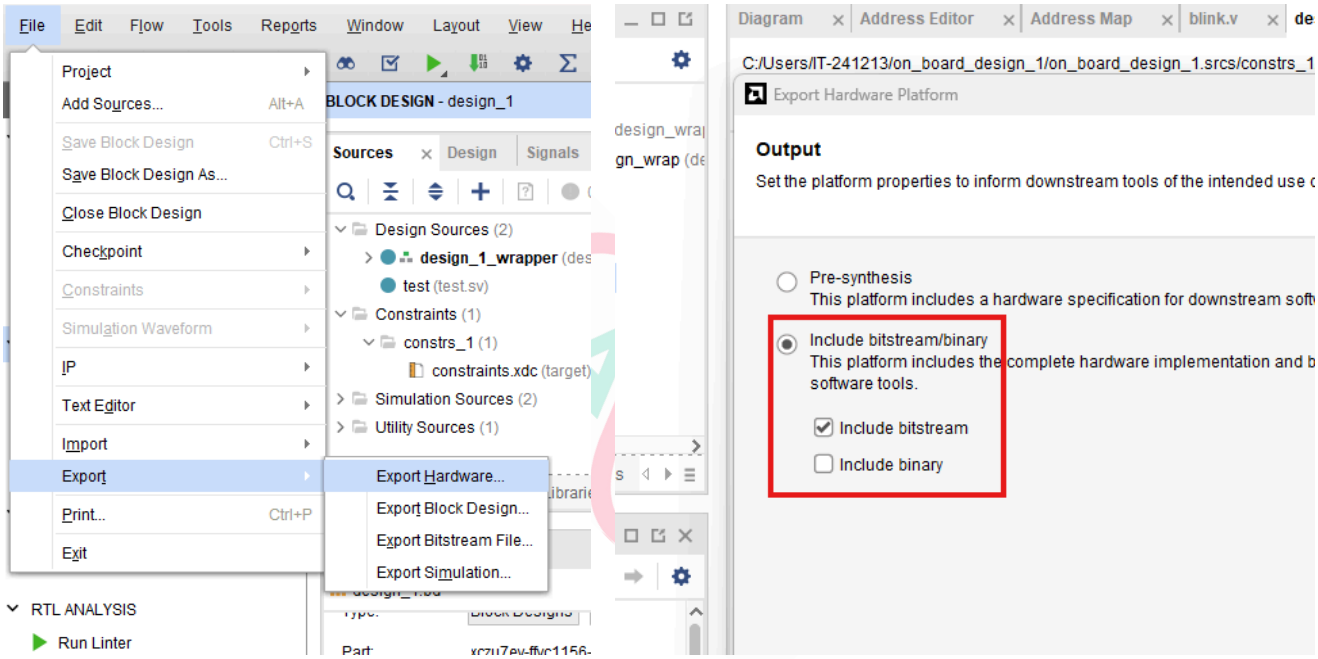


After this, we can generate bitstream directly (which runs the elaboration step automatically) or run implementation and then generate bitstream.

Note: You do NOT need to set package pin constraints for internal signals that come from the Processing System (PS) into your PL logic.The PS pins are fixed and managed internally by the Zynq UltraScale+ device, so you don't constrain them in the XDC.

We need to set design constraints related to package pins if we need to connect any ports of the top level to the external pin physically present in the board. For example: switches connected to input or led connected to output.

After generating bitstream. The export hardware generates a (.xsa) file. This file is loaded in Vitis IDE where the board platform is established with all the drivers and IP used in block design for the application to be developed.



Lab-2: Running a C application on the Processing system

Title: Running a C application on the processing system of the FPGA board.

Objective

- Make a block diagram in the Vivado.
- Generate XSA, import in Vitis.
- Write a basic “Hello world” application in the Vitis IDE, run it on the board.

Functional Description

The objective of this project is to go through the steps to run a software application on the FPGA processing system side which consists of ARM cortex A7 processor.

Block Diagram: Consists of only a Zynq processing system.

Solution

Vitis guideline

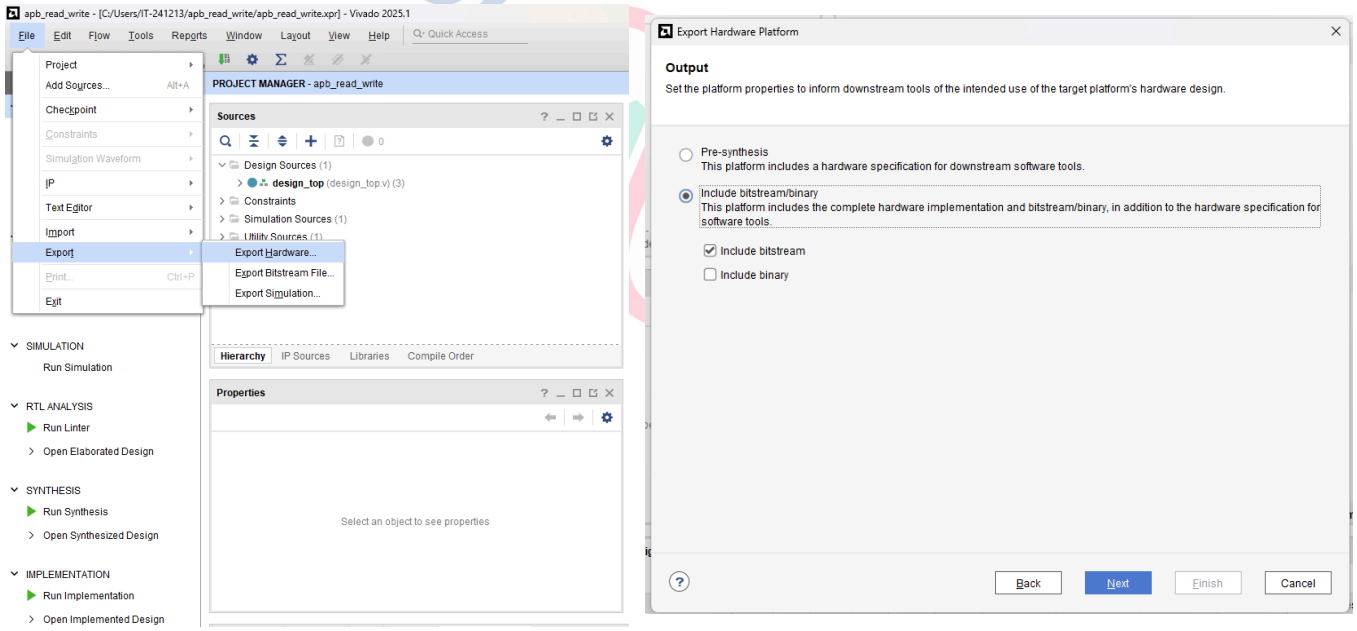
Developing application in Vitis has steps as follows:

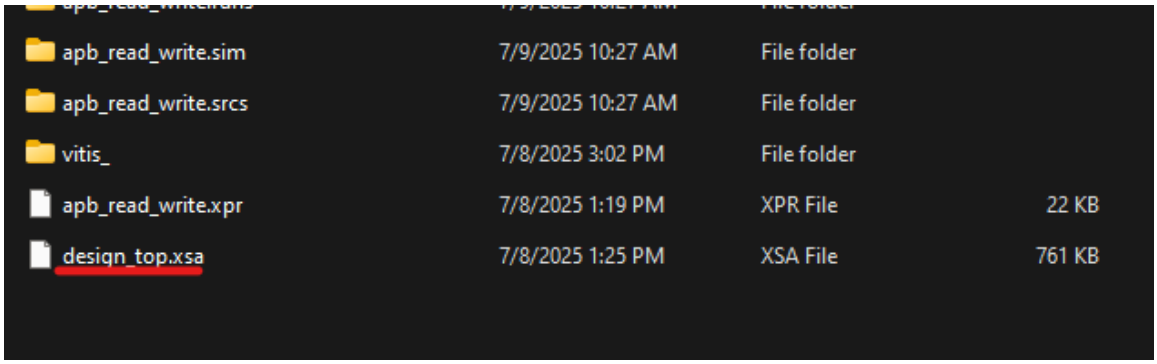
1. Creating a project using a hardware description file (.XSA) exported from Vivado.
2. Configuring the platform project.
3. Creating an application component.

We'll go through the steps for a "**Hello world**" project.

Step1: Exporting XSA file

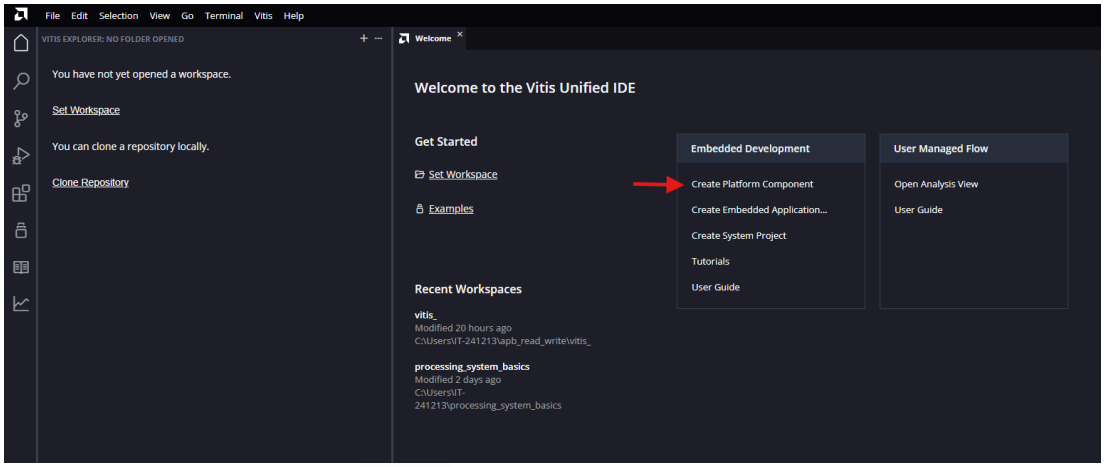
Selecting File > Export > Export Hardware creates a XSA file. Include the bitstream in the file when selecting options. This will create a XSA file in the project directory



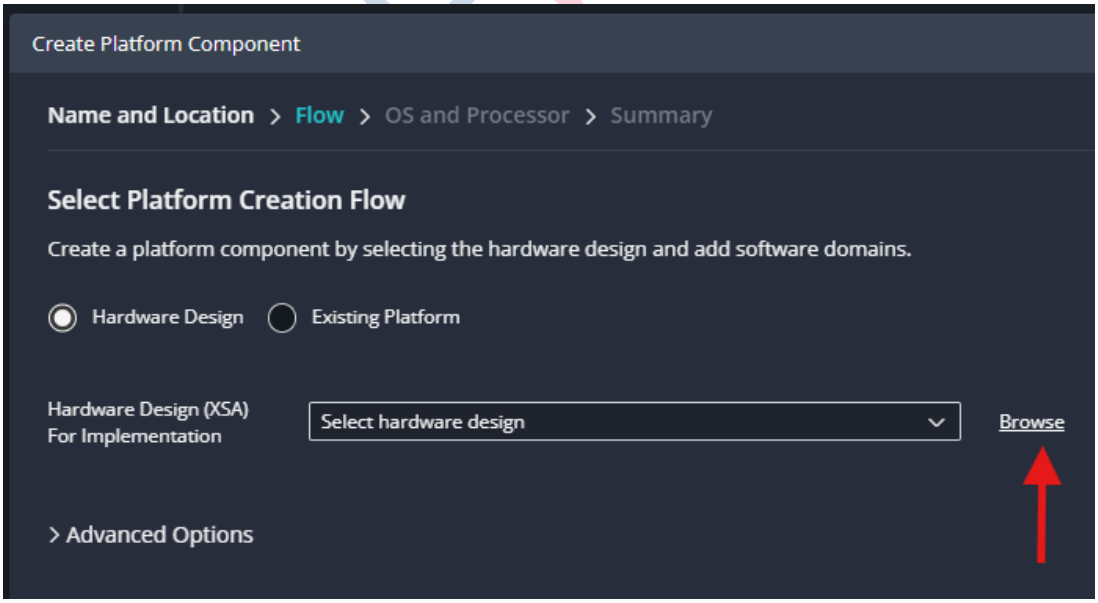


Step-2: Configuring platform application

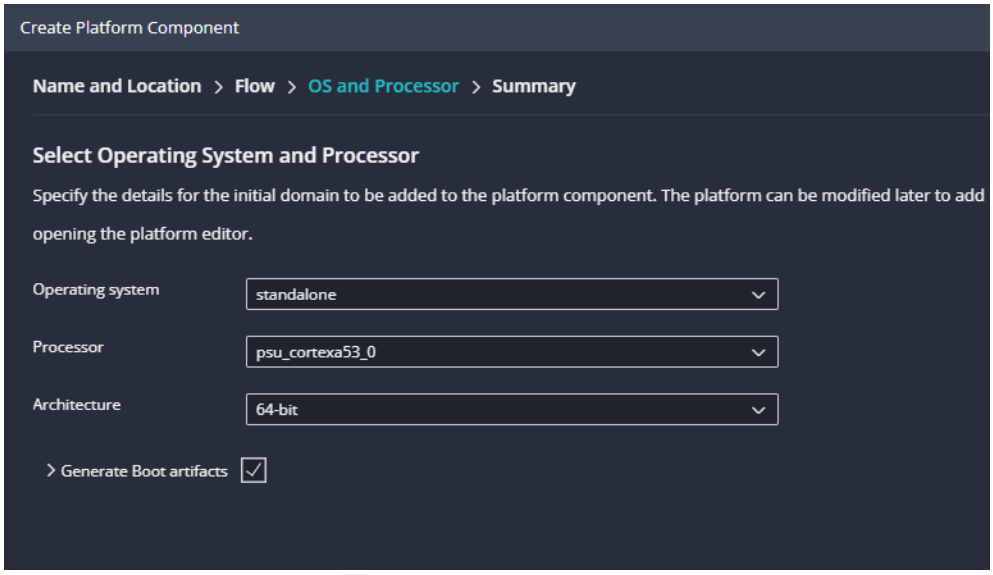
After opening Vitis, we select a directory for the project.
Then we create a platform component.



Then, we select the XSA file in the Flow tab

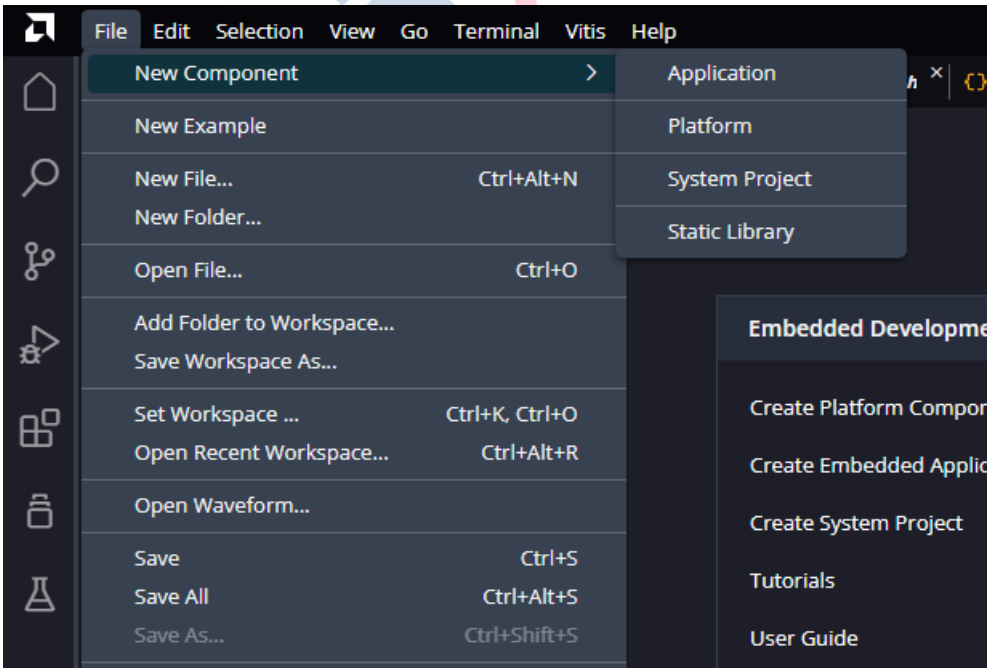


Next, we select the Operating system and the processor for our platform. For now, we choose standalone (bare metal) OS, A53_0 processor and 64-bit architecture and click finish.



Step-3: Creating application component

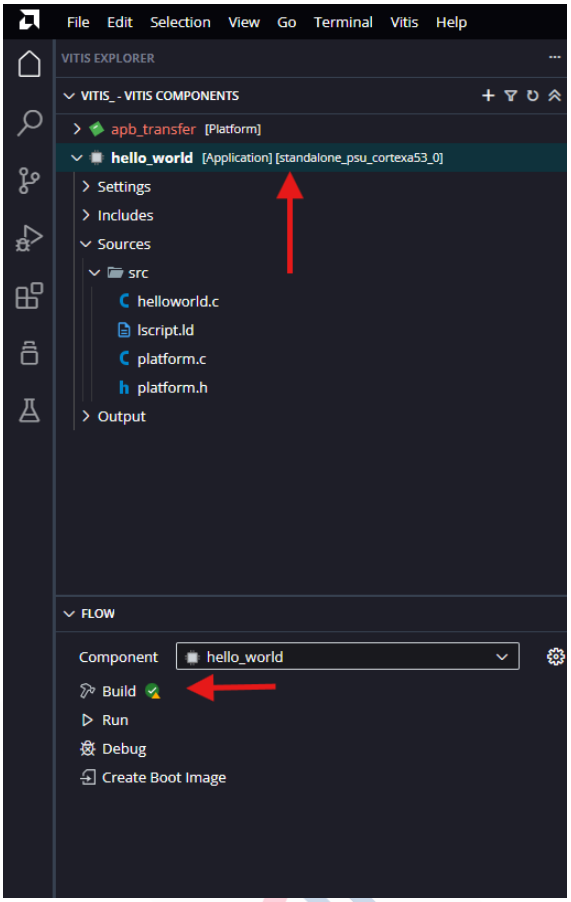
Next, we add an embedded application from the file tab.



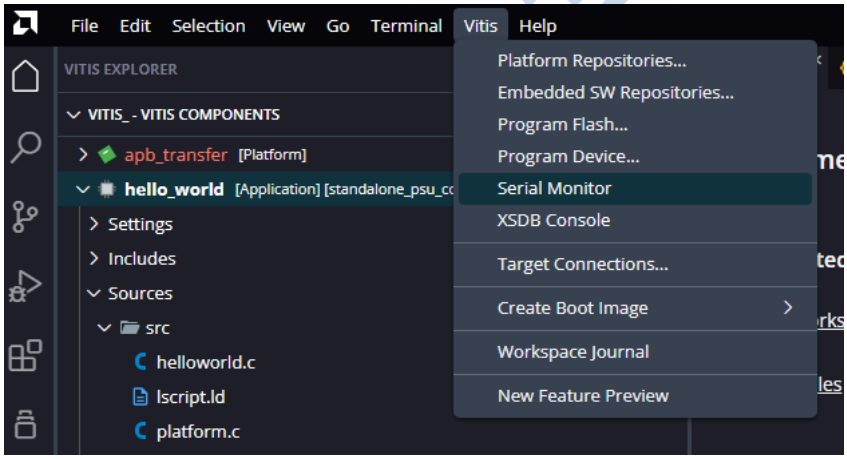
Or, we can choose from the examples. After finishing writing code, we build the application and run it after connecting the board to the computer.

Step-4: Writing application

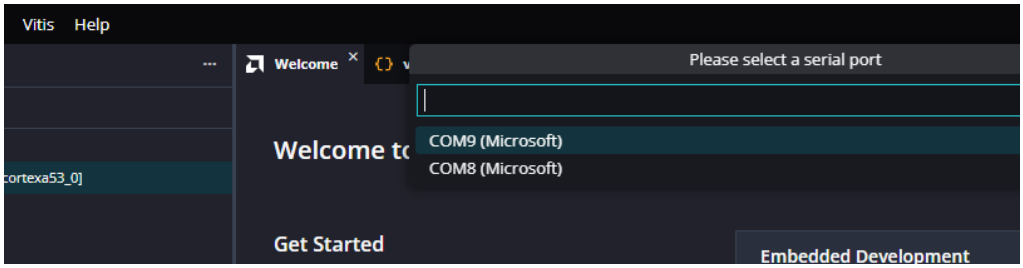
In the IDE, we can write an application in C that is compiled and built and then uploaded to the board. We can see the output in the COM port of the PC if the board is connected.

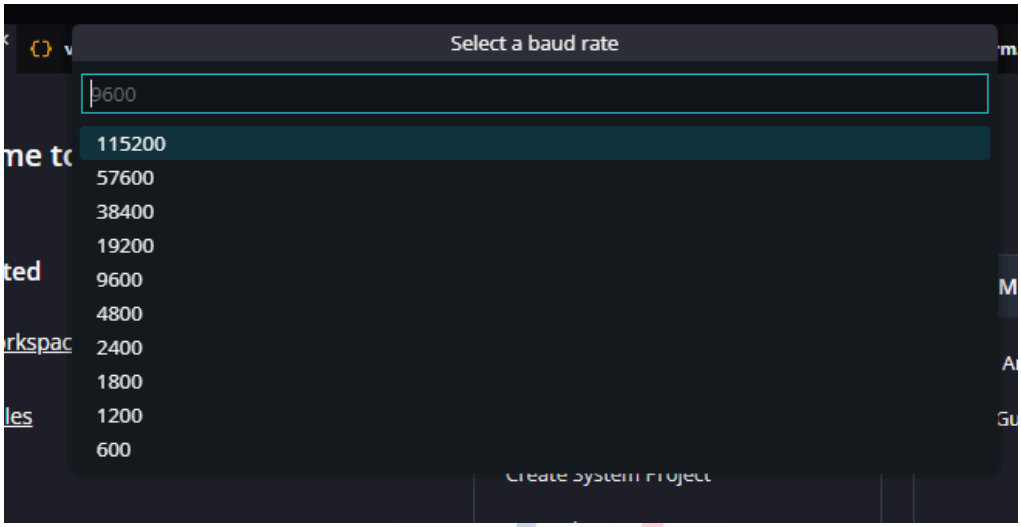


We can initiate the terminal with a specific baud rate if there's a serial connection configured (usually UART) in the PS.



Finally, select the appropriate COM port (the lowest value COM with FTDI) and the baud rate (default value of 115200).


Vitis
Help
Welcome
Please select a serial port
COM9 (Microsoft)
COM8 (Microsoft)
Welcome to
Get Started
Embedded Development


Select a baud rate
9600
115200
57600
38400
19200
9600
4800
2400
1800
1200
600

Lab-3: Logic controlled LEDs

Title: Implementing Logic controlled LEDs that are driven by push buttons.

Objective

- Control 4 LEDs present in the Zynq Ultrascale+ MPSoC using an FPGA board by controlling the logic circuit using push buttons.
- Design and implement a project in the Vivado IDE.
- Perform Block level integration between MPSoC in PS side and RTL block in PL side in Vivado design suite
- Develop XDC file to configure the IP to use external pins (LEDs).

Functional Description

The objective of this project is to control the LEDs using push buttons. The Push buttons control the inputs of logic gates, and the outputs of these gates drive the LEDs. **The logic is implemented in the PS.**
The Logic is as follows:
LED0 = PB0 | PB1
LED1 = PB0 & PB1
LED2 = PB0 ^ PB1
LED3 = ~(PB0 ^ PB1)

Block Diagram:

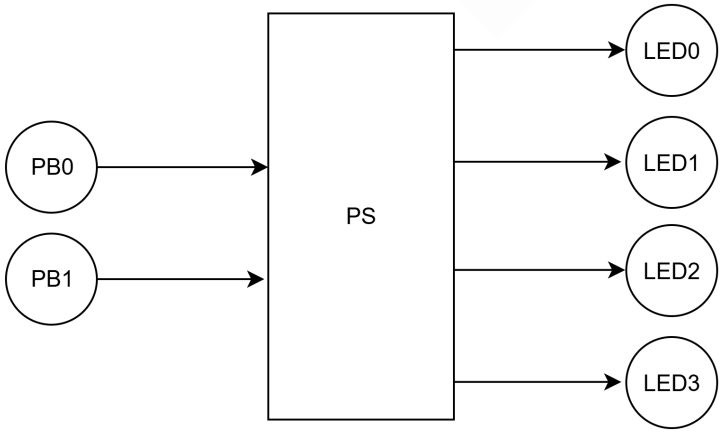
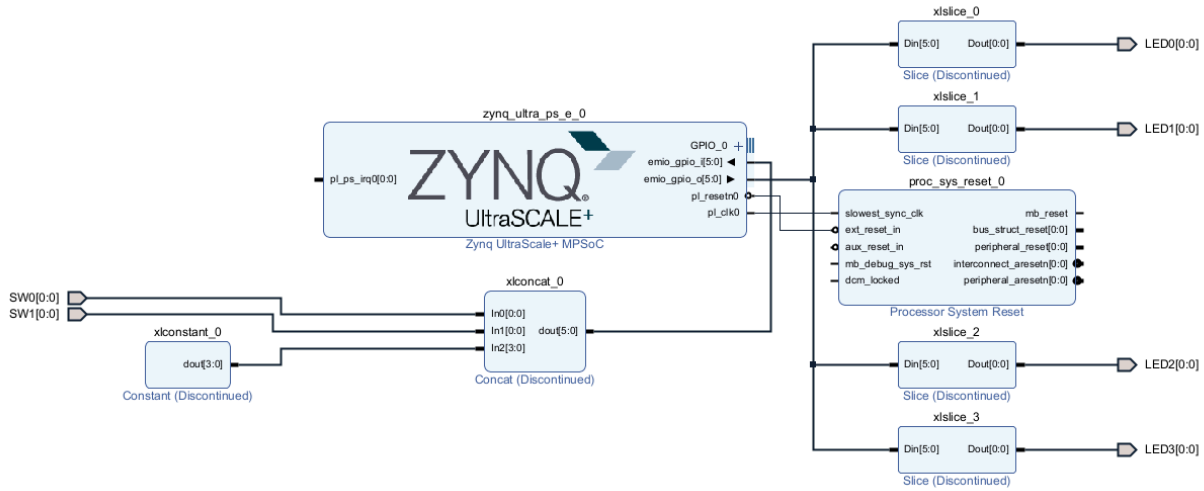


Figure 11: Block diagram of LED controller

Solution

Block diagram of the design



Vitis Application [code]

Lab-4: GPIO controlled LEDs

Title: Implementing Logic controlled LEDs that are driven by push buttons.

Objective

- Control 4 LEDs present in the Zynq Ultrascale+ MPSoC using an FPGA board by controlling AXI-GPIO in a cyclic manner.
- Design and implement a project in the Vivado IDE.
- Perform Block level integration between MPSoC in PS side and RTL block in PL side in Vivado design suite.
- Develop XDC file to configure the IP to use external pins (LEDs).

Functional Description

The objective of this project is to control the LEDs cyclicly. Each LED lights up for one second, stays off for the next 3 seconds. The timing is the same as the Lab1 additional task. **The logic is implemented in the PS.**

Block Diagram:

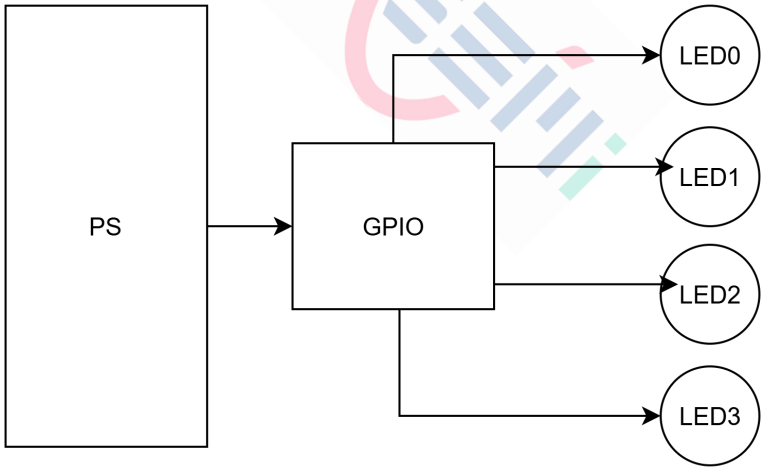
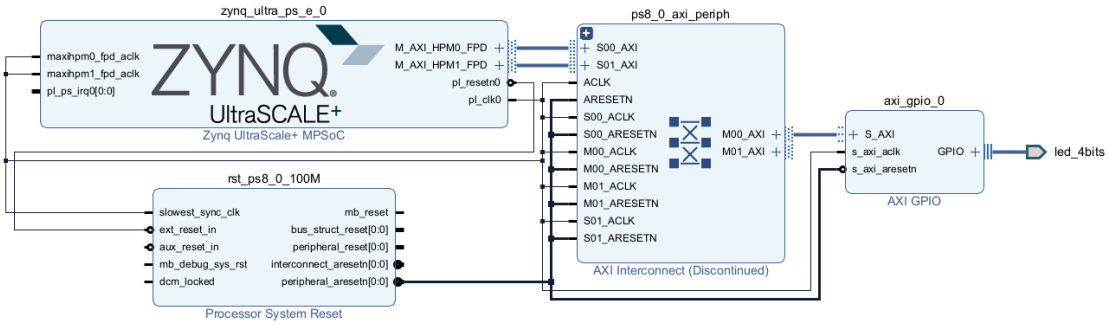


Figure 12: block diagram of GPIO based LED controller

Solution

Block diagram of the design:



Vitis application code:

```c
#include "xparameters.h"
#include <stdio.h>
#include "platform.h"
#include "xil_io.h"

int main()
{

    init_platform();
    u32 value = 0x01;
    while(1)
    {
        Xil_Out32(XPAR_GPIO_0_BASEADDR, value);
        msleep(1000);
        value = value << 1;
        if(value == 0x0)value = 0x1;
    }

    cleanup_platform();
    return 0;
}
```

Lab-5: APB implementation on FPGA Board

Title: LED controller using APB-Register Interface

Objective

- Control 4 LEDs present in the Zynq Ultrascale+ MPSoC using an FPGA board.
- Develop APB reg Interface based led controller RTL.
- Perform Block level integration between MPSoC in PS side and RTL block in PL side in Vivado design suite
- Develop Application to operate the controller through Vitis Platform.

Task Description

The APB register interface-based LED controller controls a total of 4 LEDs through an APB register interface consisting of 4 Register. Writing an odd value to a specific register makes the corresponding LED On. Similarly, writing an even number makes the LED go off.
An application developed in Vitis platform is responsible to perform write operations and operate the led controller through MPSoC integrated with it at the block level in Vivado design suite.

Block Diagram

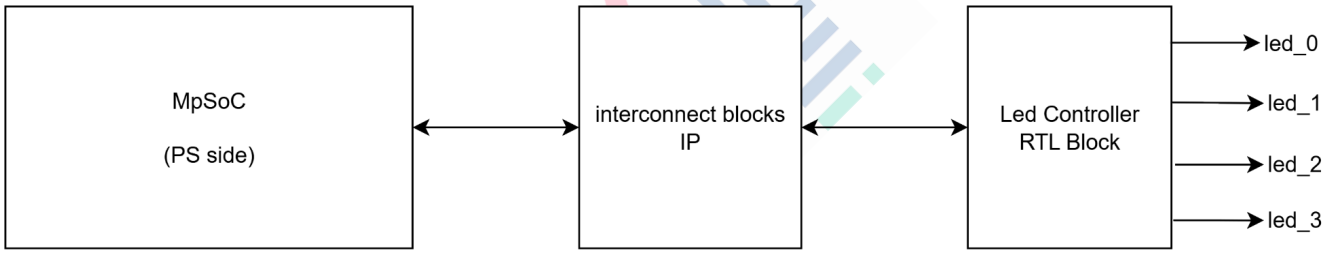


Figure 13:Conceptual View of the whole system implementations

Solution

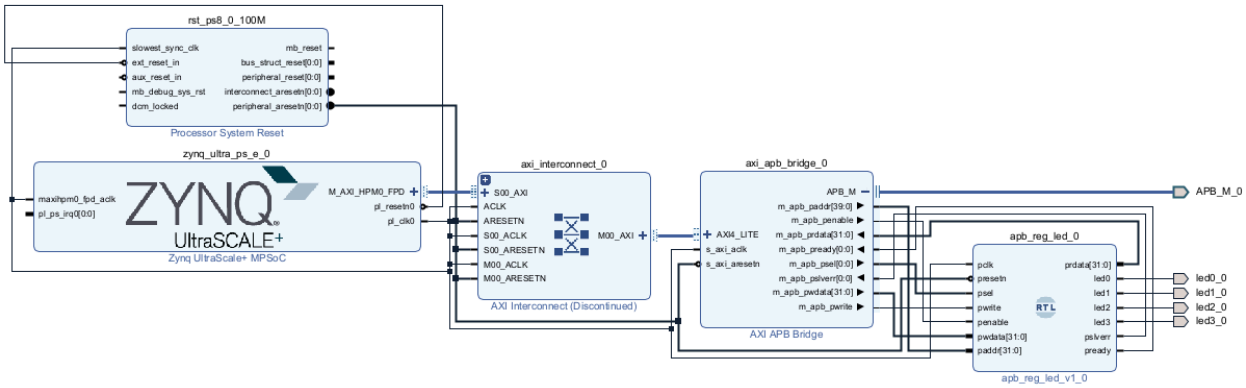


Figure: Block diagram of the design

Problems and solutions to Consider

- Validate design to automatically assign base address to APB. Always consider only offset addresses when implementing write or read logic to register.
- The Offset addresses must need to be byte addressable (multiple of 4).
- Configure Constraints by creating an XDC file to ensure external output pins are connected to LEDs present in the FPGA board.

Link for Constraints

Link For Verilog and Vitis Code

Lab-6: Implementing UART and interfacing with PS-UART

Title: UART Communication Interface Between PS and PL in Zynq MPSoC

Objective

- Enable reliable UART communication between the Processing System (PS) and the Programmable Logic (PL) in a Zynq MPSoC device.
- Implement an APB based UART interface in the PL to be configured by the APB master of the MPSoC.
- Configure the PS UART (e.g., UART1) for communication with the PL
- Develop software on the PS side to transmit and receive data via UART
- Test and validate the data exchange between PS and PL, ensuring integrity, correct timing, and protocol handling.
- Deliver a complete hardware-software design, including HDL, bitstream, software application.

Task Description

Design an APB based UART IP The Keystone architecture spec from TI. The task involves establishing reliable UART communication between the Processing System (PS) and the Programmable Logic (PL) in a Zynq Ultrascale+ MPSoC device. Specifically, the goal is to enable bidirectional data transfer between a PS UART peripheral (such as UART1) and a UART implemented within the PL fabric, which could be based on a custom UART logic module developed in Verilog.

Block Diagram

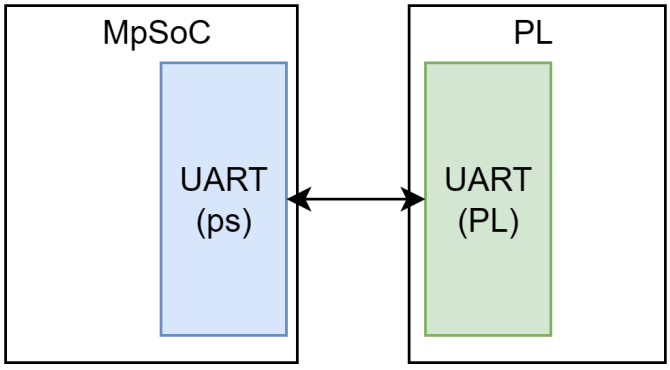
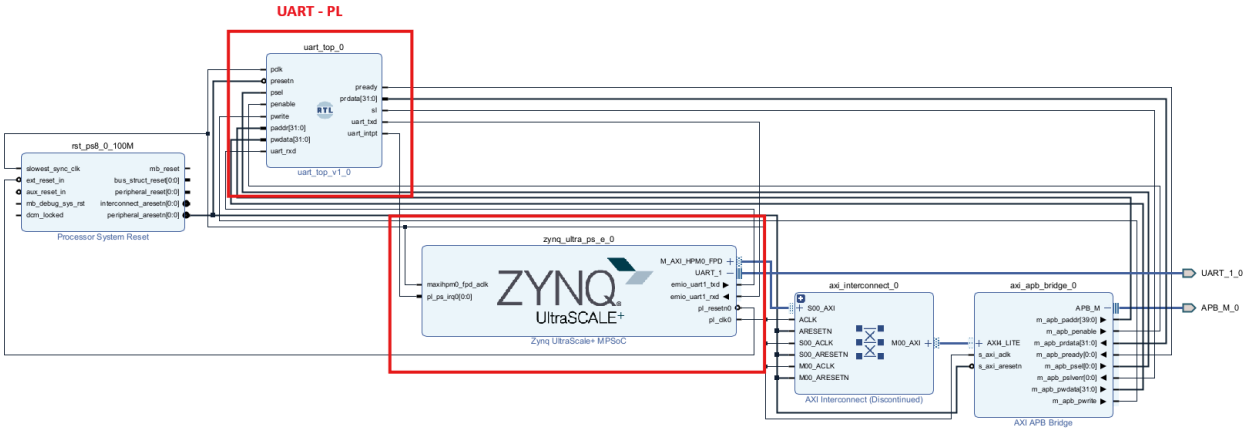


Figure 14: block diagram of UART (PS-PL) communication

Solution

Block Diagram of Vivado:



RTL Code of the UART Module

Design Description

The design includes Zynq Ultrascale+ MPSoC block (PS), AXI Interconnect, AXI-APB Bridge, UART Block (PL), System Reset Block.

The PS configures the PL side UART through APB by the AXI-APB bridge connection. The PL side UART is connected to the PS side UART through EMIO pins. PS UART TX would be routed to PL UART RX (uart_rxd) and PS UART RX would be routed to PL UART TX (uart_txd).

The PS UART is configured in Vitis through software. A UART (UART0) of PS is internally connected to the terminal of the PC through FTDI.

PS-UART Initialization

This function initializes PS side UART. It takes the driver instantiation pointer. Configuration can be altered are- Baud rate, 8-bit format, number of stop bits, parity.

```
int UartPS_Init(XUartPs *uartPS)
{
   XUartPs_Config *uartConfig;
   XUartPsFormat format;

   printf("UART-PS: Initializing\n");
   uart_pl_recv_count = 0;
   //configure data format, baud rate, parity
   format.DataBits = XUARTPS_FORMAT_8_BITS;
   format.BaudRate = XUARTPS_DFT_BAUDRATE;
   format.StopBits = XUARTPS_FORMAT_1_STOP_BIT;
   format.Parity   = XUARTPS_FORMAT_EVEN_PARITY;
   //Load UART configuration
   uartConfig = XUartPs_LookupConfig(XPAR_XUARTPS_1_BASEADDR);
   printf("UART-PS : Lookup configuration complete\n");
   //Initialize UART based on that configuration
   XUartPs_CfgInitialize(uartPS, uartConfig, XPAR_XUARTPS_1_BASEADDR);
   printf("UART-PS : Initialization complete\n");
   //Set data format for the driver
   XUartPs_SetDataFormat(uartPS, &format);
   printf("UART-PS : Setting data format complete\n");
   XUartPs_SetBaudRate(uartPS, 115200);
   XUartPs_EnableUart(uartPS);
   printf("UART-PS : Enabling complete\n");
   return 0;
}
```

Full Application Code

Typical Issues & Solutions:

1. APB Slave address does not get assigned by Vivado.
   Solution: The interface signal of the AXI-APB bridge must be made external. (APB_M_0 in the figure)

2. Vivado tries to optimize the design and flattens the hierarchy of the modules, so the synthesized circuit looks all messed up. Input and output go to blocks they are not supposed to.
   Solution: Force Vivado to keep hierarchy by adding (* keep_hierarchy = "true" *) in front of the module declaration. For example:

   (* keep_hierarchy = "true" *) module clock_gen (..)
   // your code
   Endmodule

3. Baud rate mismatch: If the baud rate is not the same (or varies by a large margin), communication does not happen.
   Solution: In order to measure the baud rate, we used counters that count the number of cycles the data lines (uart_txd and rxd) are low. We send 0x00 with even parity, 8 bit data. So there are 10 bits sent with logic 0. We can then count the number of cycles and figure out the actual baud rate of the system.

4. Giving enough time to receive: The peripheral communication happens at a much slower rate. If the baud rate is for example 115200, it's much slower than 100MHz MPSoC.
   Solutions: So, just after sending data from one UART to another, we wait a bit (using a blank while loop or sleep function) to let the receiving UART get enough time to receive them. Otherwise, we'll get zeros.

Important Notes:

- When exporting an XSA file from Vivado, delete the existing platform and app. Copy the app code beforehand. Then with the new XSA, create a new platform and app component. Otherwise, re-reading the XSA file will not reflect the change.

- Baud rate is never equal to desired value because of the limitation of using integer baud rate divisors. Use a value as close as possible.

- To use interrupt, we need an interrupt ID. The SCUGIC has a built-in function for that. It requires the PL interrupt pin (from technical reference manual), trigger (level or edge), priority. Then we pass a function.
- Run block automation on the Zynq Ultrascale+ block. It automatically configures UART, I2C and DDR which is expected by the Vitis bootloader.

Lab-7: SPI Implementation on FPGA board

Title: SPI Communication Between PS and PL in Zynq Ultrascale+ MPSoC FPGA board.

Objective

- To develop a SPI RTL module operating in both master and slave mode.
- Perform Block level integration to establish communication between SPI present in MPSoC of PS side and SPI RTL module of PL side.
- Develop Application in Vitis platform to perform communication between the two SPI where one is in manager mode and other in subordinate.

Task Description

Develop a FIFO based SPI RTL module that operates in both Manager and Subordinate mode. The SPI present in the MPSoC is brought out externally through the EMIO port. The SPI RTL module is configured and controlled by the APB-Register interface. Through the AXI-interconnect IP integration the MPSoC performs the necessary write operation to operate the SPI RTL in the mode necessary.
Both the SPI operations are performed through the application developed in Vitis Platform.

Block Diagram

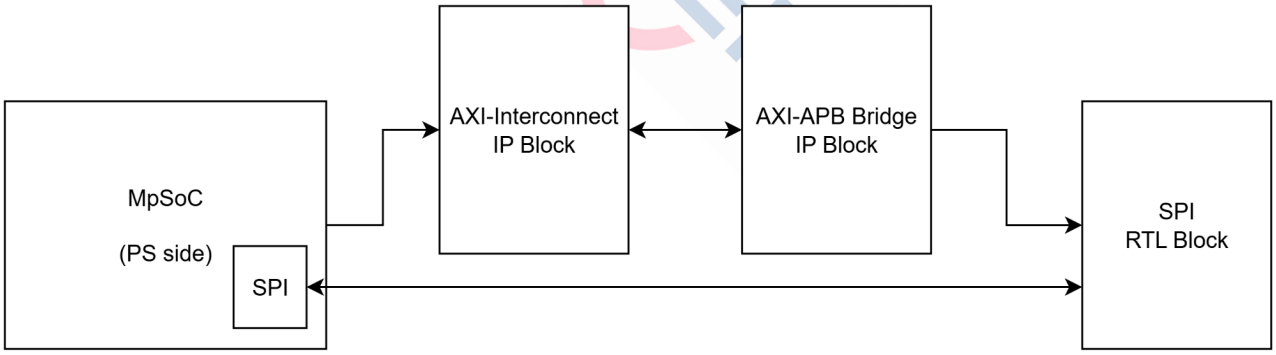
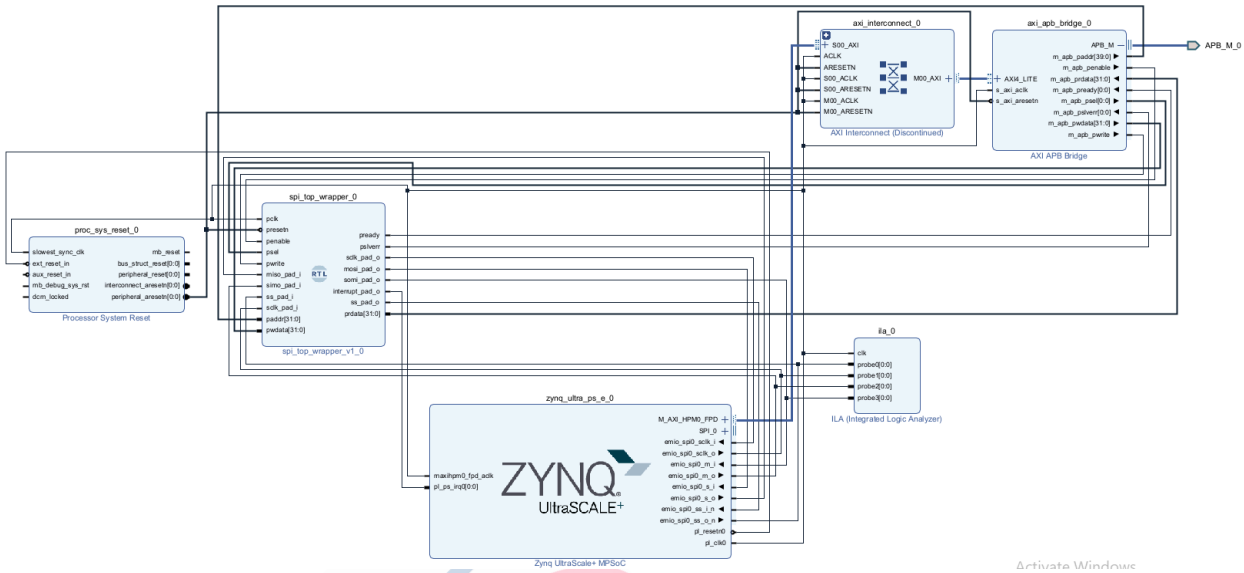


Figure 15: Conceptual View of SPI Communication Interface

Solution

Block diagram of the design:



Problems and Solutions to Consider

1. The PS SPI has a fixed transfer width of 16 bit, on the other hand the SPI RTL module on the PL side accepts a world length of 8 (max). But each FIFO has a width of 8. So, when transmission starts as ***slave_select*** signal goes low, after 16 bits of transmission ss goes high. As a result, the SPI misses the first 8 bits because its shift-register is 8-bit wide and the first 8 bits are shifted away by the next 8 bits.

   Solution:

   We modified our SPI RTL module to accept only the first 8 bits of the transmission and wrote the software for the PS side SPI so that it only sends 8-bits of data, the remaining 8 bits are zeros.

2. Generic PS-SPI initialization:

This function initializes PS side SPI. For this function to work, declare the driver instantiation and configuration as global variables in the file, name them as spips and config1 (or change them in the function)

```
s32 SETUP_SPIPS()
{
  // Configure SPI0 as Master
  printf("SPI-PS: STARTING SETUP\n");
  config1 = XSpiPs_LookupConfig(XPAR_XSPIPS_0_BASEADDR);
  if (config1 == NULL) {
    printf("SPI-PS: LookupConfig for SPI0 failed\n");
    return XST_FAILURE;
```

```
}
XSpiPs_CfgInitialize(&spips, config1, config1->BaseAddress);
XSpiPs_SetOptions(&spips, XSPIPS_MASTER_OPTION);
XSpiPs_SetClkPrescaler(&spips, XSPIPS_CLK_PRESCALE_8);
XSpiPs_Enable(&spips);
printf("SPI-PS: SETUP COMPLETE\n\n");
return XST_SUCCESS;
```

Link for Vitis Code

Link for System Verilog Code

Lab-8: I²C Slave Implementation on FPGA Board

Title: I2C Communication Between PS and PL in Zynq Ultrascale+ MPSoC FPGA board.

Objective

- To develop an I2C subordinate RTL module.
- Perform Block level integration to establish communication between I2C present in MPSoC of PS side operating as a manager and I2C subordinate RTL block of PL side.
- Develop application in Vitis platform to perform communication between the two I2C.

Task Description

Develop a I2C RTL module that operates only as a subordinate. The I2C present in the MPSoC is brought out externally through the EMIO port. The I2C RTL module is configured and controlled by the APB-Register interface. Through the AXI-interconnect IP integration the MPSoC performs the necessary write operation to operate the I2C RTL module.
Both the I2C operations are performed through the application developed in Vitis Platform.

Block Diagram

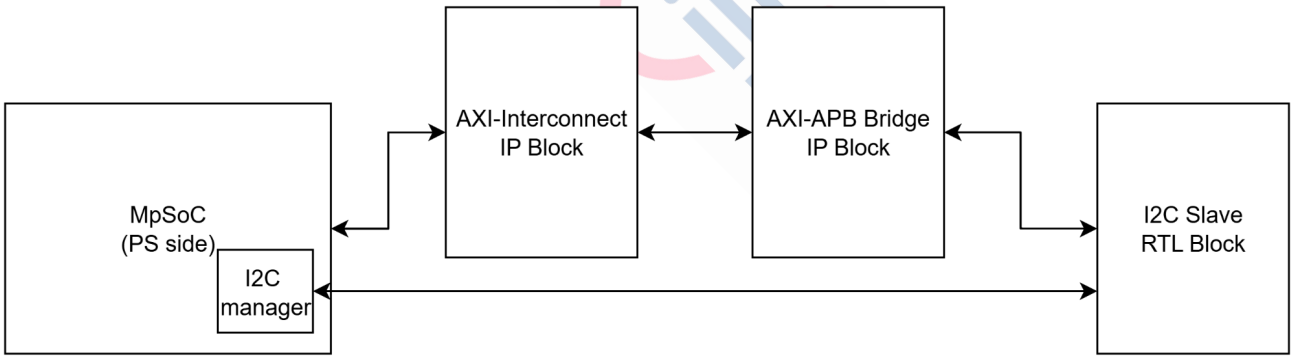
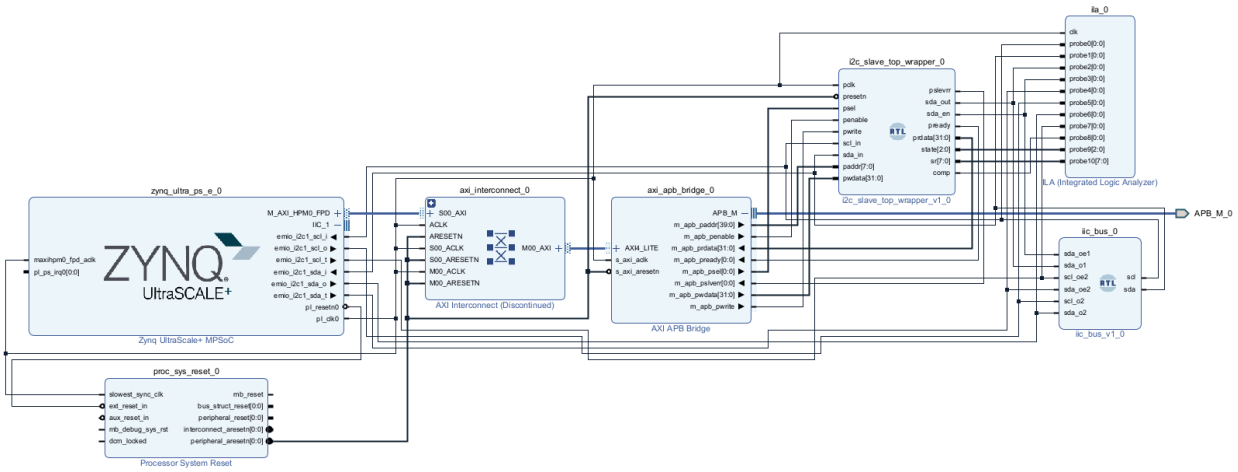


Figure 16:Conceptual View of I2C Communication Interface

Solution

Block diagram of the design:



Problems and solutions to Consider:

1. The PS side I2C controller shows significant issues while implemented as a subordinate. It shows irregularities in its operation. Avoid configuring and operating it as a Slave and rather operate it as a master.

2. Replicate the tristate buffered-wired AND bus sharing mechanism between I2C Manager and Subordinate in a separate “bus” RTL block. This RTL block should establish the interconnection between the PS side I2C and PL side I2C.

3. The bus enable/trigger signal of the PS side I2C is Active low. So, establish the bus logic in the “bus” RTL block accordingly.

4. The PS I2C bus, when enabled, will go into operation automatically if it transmits FIFO containing any kind of data unknowingly or from its previous operation. So, it is a norm to initialize the I2C controller after an I2C reset to avoid such issues.

5. The PS side I2C during its SLA+R/W transmit operation, the last bit indicates the R/W bit where,
    a. 0, indicates initiating Read operation (receiving data from Subordinate)
    b. 1, indicates initiating write operation (transmitting data to Subordinate)

Generic PS-I2C initialization:

This function configures the PS side IIC to operate in master mode. Set the SCLK value, address option (7 or 10 bit) accordingly.

```
s32 SETUP_IICPS()
{
   s32 status;
   printf("IIC-PS : Initializing\n");
   iic_config = XIicPs_LookupConfig(XPAR_I2C0_BASEADDR);
   status = XIicPs_CfgInitialize(&iicps, iic_config, iic_config->BaseAddress);
   if(status == XST_SUCCESS)printf("IIC-PS : Configuration lookup done\n");
   else printf("IIC-PS : Configuration lookup failed\n");

   printf("IIC-PS : Resetting device\n");
   XIicPs_Reset(&iicps);

   status = XIicPs_SelfTest(&iicps);
   if(status == XST_SUCCESS)printf("IIC-PS : Self testsuccessful\n");
   else printf("IIC-PS : Selftest falied\n");

   //XIicPs_SetupSlave(&iicps, IIC_SLAVE_ADDR);
   XIicPs_SetSClk(&iicps, 50000000);
   XIicPs_SetOptions(&iicps, XIICPS_7_BIT_ADDR_OPTION);
   printf("IIC-PS : SETUP COMPLETE\n");
   return XST_FAILURE;
}
```

System Verilog Code.

Vitis Code - IIC Application.

Lab-9: Watchdog Timer Implementation on FPGA board

Title: Implement of Watchdog timer module in the PL side of FPGA board

Objective:

- To develop a WDT RTL module.
- Perform Block level integration where the Interrupt generated by WDT received by the Interrupt request port in MPSoC and MPSoC performs necessary configuration to WDT.
- Develop Application in Vitis platform to perform necessary interrupt service routine.

Task Description

To implement a watchdog timer in the FPGA board and generate interrupt service routine to server the interrupt.

Block Diagram:

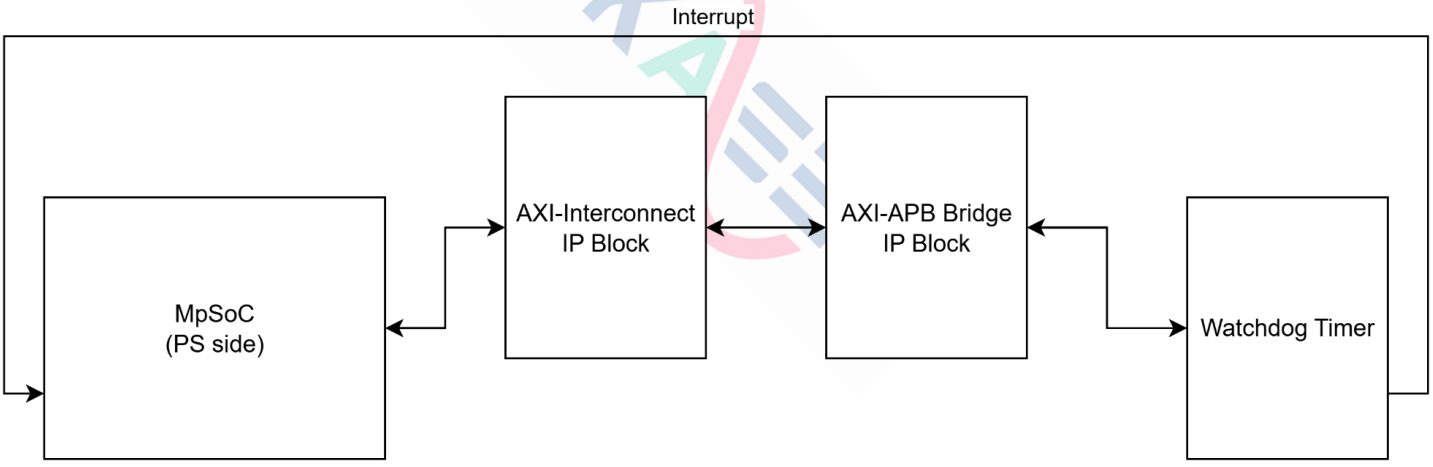
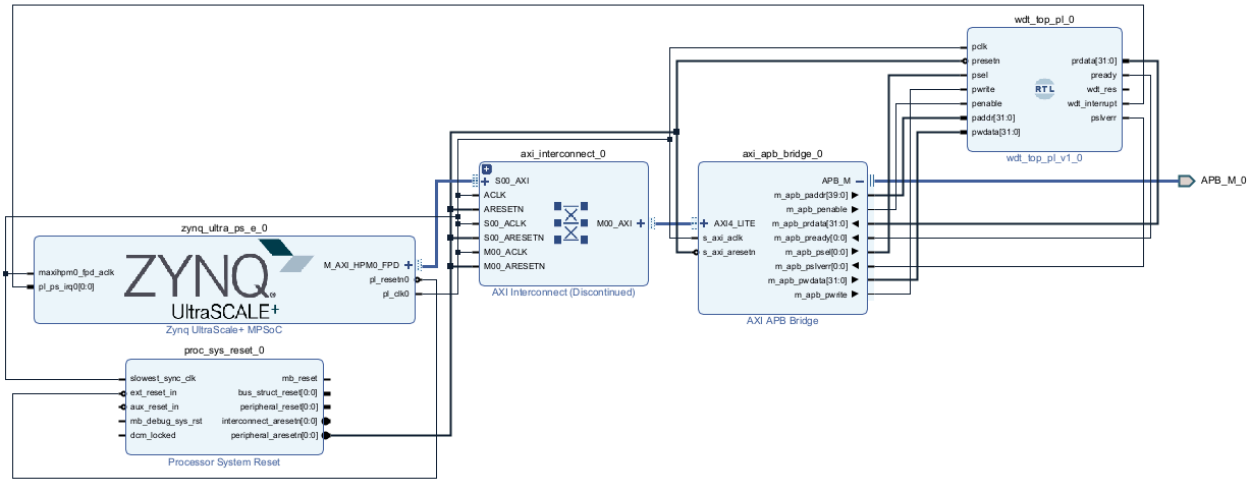


Figure 17:Conceptual View of MPSoC-WDT interface

Solution

Block diagram of the design:



System Verilog code

Application code

Lab-10: Final - Implementing Temperature-humidity controller

Title: Design and Implementation of a Temperature and Humidity sensor-based System using custom IPs (APB, I^2C, SPI, Watchdog Timer, UART).

Features

- Integrates a PS software application with custom PL hardware controllers on a Zynq MPSoC.
- Uses timer and GPIO interrupts for efficient, event-driven processing without polling.
- Features custom AXI slave peripherals to simulate sensors and directly control actuators.
- Provides automated closed-loop control by reacting to thresholds and restoring defaults.
- Manages system logic and user input with a robust software state machine.
- Utilizes multiple protocols (AXI, APB, UART) for internal control and status reporting

Block Diagram

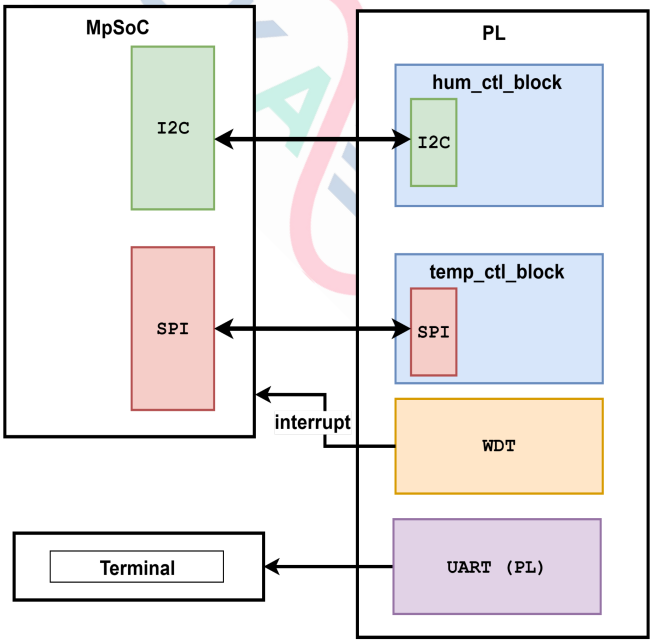


Figure 18: Block diagram for temperature-humidity controller

Signal Description

Name	Width	Direction	Description
pb_temp_inc	1	Input	Push button to increase temperature
pb_temp_dec	1	Input	Push button to decrease temperature
pb_hum_inc	1	Input	Push button to increase humidity
pb_hum_dec	1	Input	Push button to decrease humidity
fan_led	1	Output	LED to indicate the fan is ON.
heater_led	1	Output	LED to indicate the heater is ON.
hum_led	1	Output	LED to indicate the humidifier is ON.
dehum_led	1	Output	LED to indicate the dehumidifier is ON.

Functional Description

The module contains two independent controllers. One is for temperature and the other is for humidity. The temperature controller is a SPI slave module which holds the data of real time temperature. The humidity controller is a I2C slave module which holds the data of real time humidity. Both take push button signals as input to increase or decrease the value of temperature or humidity respectively. Push button functions are as follows:

pb_temp_inc → Increase Temperature by 1°C (Push button 1 of the board)
pb_temp_dec → Decrease Temperature by 1°C (Push button 3 of the board)
pb_hum_inc → Increase Humidity by 1°C (Push button 2 of the board)
pb_hum_dec → Increase Humidity by 1°C (Push button 4 of the board)

There is also a Watchdog timer used as a timer to generate an interrupt every 5 seconds. The processor service that interrupts by taking the value of temperature and humidity by its own SPI and I2C. After taking the value of temperature and humidity the processor checks whether the value has breached the threshold value or not. The default value of temperature is **25°C** and humidity is **50%**. The threshold value of the temperature is **40°C** and **15°C**. The threshold value of the humidity is **70%** and **30%**.

If the sampled temperature is above 40°C then the fan (fan LED) turns on.
If the sampled temperature is below 15°C then the heater (heater LED) turns on.
If the sampled temperature is above 70% then the dehumidifier (dehumidifier LED) turns on.
If the sampled temperature is below 30% then the humidifier (humidifier LED) turns on.

Description of LEDs:
LED 0 → Fan (fast blinking)
LED 1 → Dehumidifier (3 sec on 1 sec off)
LED 2 → Heater (slow blinking)
LED 3 → Humidifier (3 sec off 1 sec on)

All the actuators increase or decrease the respective parameter at every 1 second. For example, if the fan is turned on then the temperature will decrease by 1°C at every 1s. After reaching the default value the respective LED will turn off.

If any of the LEDs gets turned on the respective push buttons become inactive. For example, if the temperature sampled by PS is **50°C** then the fan (fan LED) will be turned on and the push button to increase or decrease the temperature will be inactive until the fan turns off again.

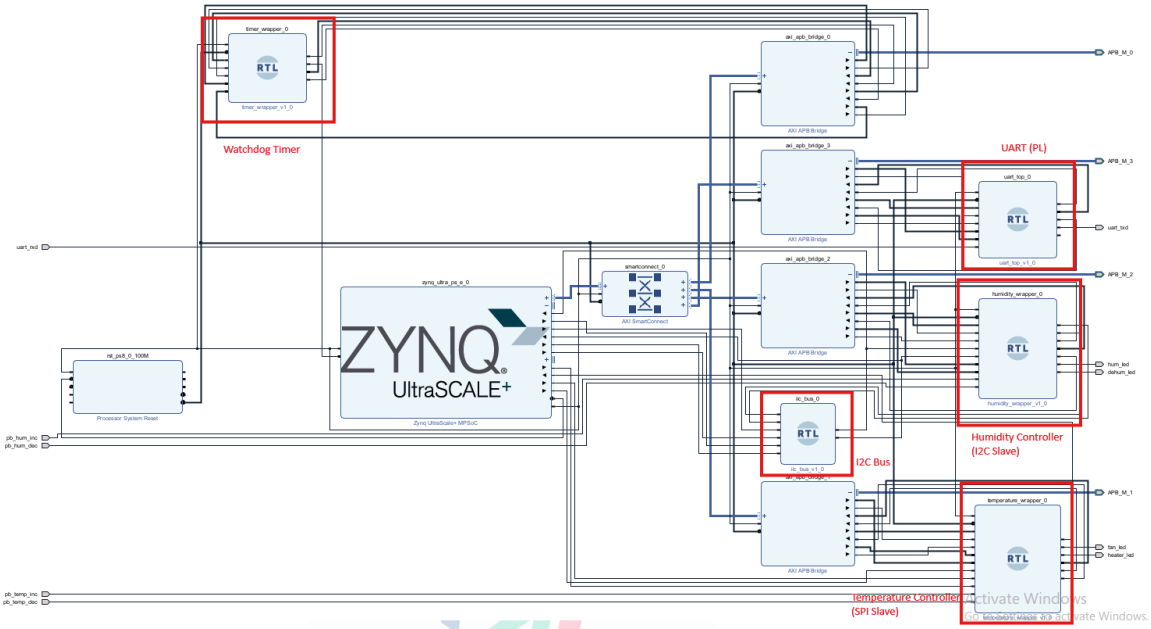
The values will be read by the PS from the corresponding IPs for the temp and humidity data. The values are then written to UART (PL) which will send it to the terminal directly for constant monitoring.

Added Task (Optional)

- Make the LEDs blinking timings configurable.
- Make the LEDs of increasing or decreasing capability configurable.
- Make the interrupt time configurable.
- Make the push button's effect configurable.

Solution

Block Diagram of Vivado



Design Description

The design is centered around the **Zynq Ultrascale+ MPSoC** and includes components such as the **Smart AXI Interconnect**, **AXI-APB Bridge**, **System Reset Block**, and the following key modules:

- **Watchdog Timer (timer_wrapper_0):**
  This peripheral is configured as a general-purpose timer. It is programmed to generate a periodic interrupt every five seconds, which serves as the primary trigger for the Processing System (PS) to read sensor data and update system logic.

- **Temperature Controller (temperature_wrapper_0):**
  This is an SPI-based sensor. It periodically updates the temperature value to the CPU via SPI. Based on this data, the CPU controls the fan or heater by asserting an enable signal via the APB interface.

- **Humidity Controller (humidity_wrapper_0):**
  This is an I2C-based sensor. It periodically updates the humidity value to the CPU via I2C. The CPU then triggers the humidifier or dehumidifier via the APB interface.

- **PL UART (uart_top_0):**
  This peripheral provides a standard serial communication interface used for displaying system status information.

RTL Code of the UART Module

Application Code

Typical Issues & Solutions

Since there are multiple peripherals connected via the APB interface, the standard AXI Interconnect will not work here.

Solution: Have to use Smart AXI Interconnect

Important Notes

Address space: Address space means all the possible addresses in a particular addressing system. For example, if we're running a CPU that has a 4-bit address bus, then the possible number of addresses is 2^4, which is 16. It can be 0-15. This is the address space.

Loading memory: CPU can store (From CPU registers to Memory) and load (Memory to CPU registers) using LOAD/STORE instructions. Loading and storing data requires the address of the piece of memory that is to be used.
In a typical ARM architecture, The CPU has its own set of general-purpose registers (r0, r1 etc.). If it has to store more than that, it uses Memory (RAM).

Memory mapped I/O: If we want to configure peripherals, we map the address of the I/O ports of the peripherals to the Memory address space. This way, when the CPU executes an instruction to store/load from that memory address, what it's actually doing is that it's communicating with the peripheral. As the I/O is mapped to the memory space, this is known as Memory mapped I/O.

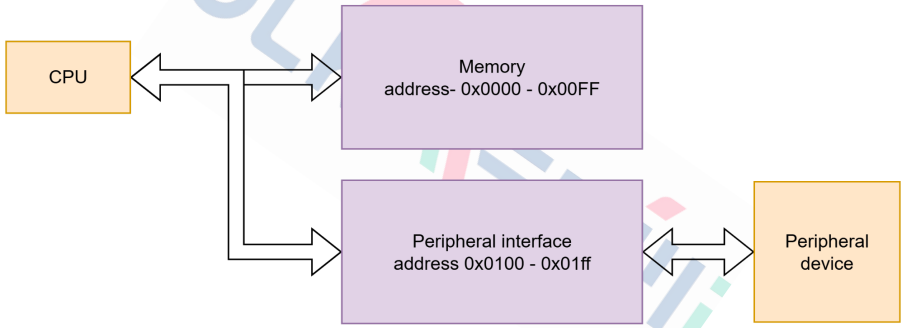


Figure 19: Memory mapping

MIO: Multiplexed input/output. It is used to control the I/O pins at the PS side

EMIO: Extended multiplexed input/output. This I/O pin goes through the PL side, communicating with the IPs in the configurable logic blocks.

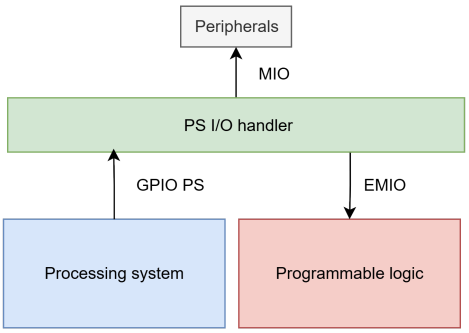


Figure 20: MIO and EMIO in MPSoC

Common pitfalls

1. Allocate memory carefully. The memory addresses should be a multiple of **4**. In the RTL, you should only compare digits that change value, not the whole address.
2. Run block automation on the Zynq PS, otherwise it will not run any application from Vitis. Running this configures UART, DDR and other necessary peripherals and internal hardware.
3. Avoid re-reading the XSA file. Instead, delete the application and create a new one. Vitis doesn't recognize the new XSA file, throws errors.

PL UART-terminal communication:

It is possible to send the data from the PL through a custom UART interface to the FTDI chip which then sends the data to the terminal using the USB interface. Here is how FTDI works:

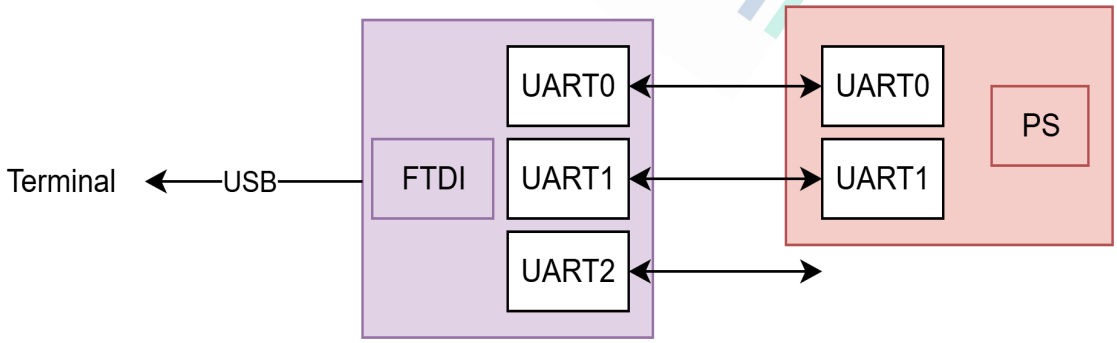


Figure 21: UART-USB-Terminal communication

FTDI has 3 channels, two of them connected to UART0 and UART1 on the PS side, The UART2 is not connected to any interface. Users can interface with this UART and send data directly to the terminal if needed. See **TRM FT4232HL USB UART Interface** for more information. To implement this,

1. Implement an UART in the PL side with correct baud rate at both sides.

2. Connect the FDTI and PL UART correctly. PS-UART-TX connects to PL-UART-RX and vice versa.
3. Configure the Constraint file (.XDC). In the ZCU 104 MPSoC, PL-UART-RX is A20 and PL-UART-TX is C19.
4. Initialize and configure the PL UART in Vitis. Writing value in the transmit buffer will print an ASCII equivalent character in the terminal. 48 prints '0'.
5. It is recommended that you have enough FIFO depth to accommodate a reasonable number of characters to print.
6. You can use VSPRINTF to format output. You should write your own code, but in case you're in a hurry, here it is:

```
void print_new(const char* string, ...)
{
   int i=0;
   va_list args;
   va_start(args, string);
   char buffer[256];
   vsprintf(buffer, string, args);
   printf("%s", buffer);
   while(i < 255 && buffer[i] != '\0')
   {
            //Write the buffer to the PL UART FIFO
            //You should check for FIFO overflow.
   }
          va_end(args);
}
```

Considerations when starting new project:

1. Address check: Always specify the range of Address in the RTL code with respect to offset. Don't compare all 32 bits.
2. Set the AXI-APB bridge output port as external (otherwise Vivado will not allocate a memory address to this component).
3. Try to avoid ILA. It slows down the process.
4. When using ILA, include major signals such as *pstate*.
5. When using ILA, if there is a problem reading/writing a register, program the device using the generated bitstream in Vivado then try again.
6. Sometimes, when using ILA, Vitis compiler throws an error like **"*** can not write memory address 0xaaff**"**. Don't panic, just restart the board.
7. Instead of using hardcoded address in Vitis, try to use macros such as #define.

8. Push-buttons on the board are not equipped with a de-bouncing circuit. Which will cause problems if you are using button input as edge triggered.

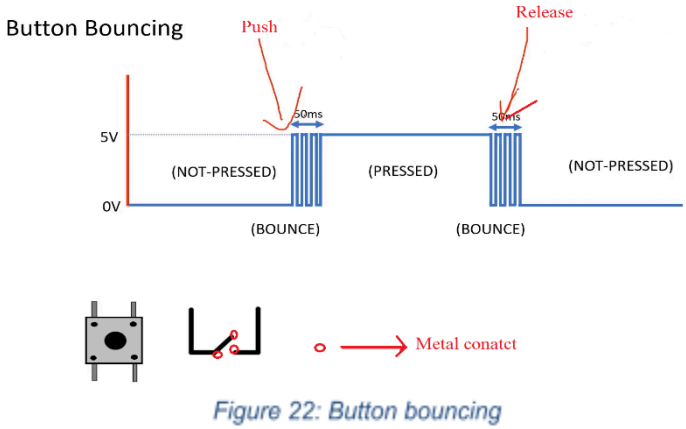


Figure 22: Button bouncing

9. Setting up interrupt service routine:

```
XGetEncodedIntrId(GIC_ID, TRIGGER TYPE, 0x00, 0x00, &ID);
Get GIC_ID from Xilinx technical reference manual
Trigger type - 0x01 - 0 -> 1 Edge
                    0x02 - 1 -> 0 Edge
                    0x04 - 0 Level trigger
                    0x08 - 1 Level trigger
ID                  the actual ID you use to setup interrupt

XSetupInterruptSystem(GIC,(XInterruptHandler)ISR,
        ID, XPAR_XSCUGIC_0_BASEADDR, XINTERRUPT_DEFAULT_PRIORITY);

GIC is the address of the GIC driver instance.
ISR is the interrupt service routing function
ID is the ID you got from the previous function
```

NOTE: Try to implement the application in poll mode first. Test the functionality and then go for implementing it in interrupt mode.

References

Important reference documents to go through white working with Zynq Ultrascale+ MPSoC FPGA board -

- Zynq Ultrascale+ MPSoC Data Sheet : Overview
    - Link- Zynq UltraScale+ MPSoC Data Sheet: Overview (DS891)
    - Use Case-
        - Contains overview of Zynq Ultrascale+ MPSoC FPGA board.
        - Major Elements present inside the PS (Processing System) and PL (Programmable Logic) side.

- Zynq Ultrascale+ Device Technical reference Manual (UG1085)
    - Link- Zynq UltraScale+ Device Technical Reference Manual (UG1085) • Viewer • AMD Technical Information Portal
    - Use case of the Document:
        - Brief Information about Zynq Ultrascale+ device architecture.
        - Detailed Information about peripherals and IPs present within the PS side such as UART, I2C etc.
        - Includes their features, functional description, block level diagram and register overview.
        - Also includes their programming flowchart.

- Vitis Tutorials: Embedded Software (XD260)
    - Link - Vitis Embedded Software Tutorials • Vitis Tutorials: Embedded Software (XD260) • Reader • AMD Technical Information Portal
    - Use case of this documents:
        - Tutorials on How to get started with Vitis IDE.

- Zynq Ultrascale+ MPSoC Embedded Design Tutorial (UG1209)
    - Link -Zynq UltraScale+ MPSoC Embedded Design Tutorial (UG1209) • Zynq UltraScale+ MPSoC Embedded Design Tutorial (UG1209) • Reader • AMD Technical Information Portal
    - Use case-

- Standalone Library Documentation (UG643)
    - Link- BSP and Libraries Overview • Standalone Library Documentation: BSP and Libraries Document Collection (UG643) • Reader • AMD Technical Information Portal
    - Use case of this document- To learn about the Xilinx standard C libraries