KU LEUVEN

FACULTY OF ENGINEERING TECHNOLOGY
CAMPUS GEEL

SYSTEM ON CHIP

Lab 1: Zynq I/O and interrupts

Arnab Saha

Kodchakorn Thaboot

Supervisor: Levi Mariën

Electronic Circuits and Systems (ESAT)
Advanced Integrated Sensing Lab (ADVISE)

Radiation and its Effects on Microelectronics and Photonics (RADMEP)

Academic year 2022-2024

Contents

List of Tables

List of Figures

List of Acronyms

SoC	System on Chip
FPGA	Field Programmable Gate Array
JTAG	Joint Test Action Group
UART	Universal Asynchronous Receiver/Transmitter
PS	Processing System
PL	Programmable Logic
AXI	Advanced eXtensible Interface
SDK	Software Developement Kit
IDE	Integrated Development Environment
IP	Intellectual Property
GPIO	General Purpose Input/Output
GIC	Generic Interrupt Controller
NREE	Non-Recuring Engineering Effort
HIL	Harware in the Loop
BSP	Board Support Package
ILA	Integrated Logic Analyzer
GIER	Global Interrupt Enable Register
TOW	Toggle on Write
IER	Interrupt Enable Register
ISR	Interrupt Status Register

Chapter 1

Introduction

Embedded systems are pervasive in our modern technological landscape, ranging from automobiles and appliances to medical equipment and industrial control systems. These systems are designed to perform specific tasks with a high degree of reliability, efficiency, and performance. The heart of many embedded systems is a microcontroller or a microprocessor, which is responsible for executing the program instructions and interfacing with the system's peripherals.

In this lab report, we focus on one type of embedded system that uses a Field-Programmable Gate Array (FPGA) board called the Zynq, which combines a dual-core ARM processor with programmable logic. The Zynq board provides an ideal platform for developing embedded systems that require real-time processing, high-speed data acquisition, and customizable interfaces.

1 Lab Overview

In this lab, we explore the concepts of input/output (I/O) and interrupts in the context of embedded systems. The I/O refers to the communication between the embedded system and the external world, such as sensors, actuators, and human-machine interfaces. Interrupts are signals that are triggered by external events, such as a button press or a sensor reading, that cause the microcontroller to pause its current operation and execute a specific routine. The PYNQ-Z2 board was used for this lab experiment.

Chapter 2

System Description

1 System-on-Chip (SoC)

A System on Chip (SoC) is a complete electronic system that integrates all components of a computer or other electronic system onto a single microchip. A central processing unit (CPU), memory, input/output interfaces, and several other peripheral components including sensors, power management, and communication interfaces are all generally integrated into the SoC. As they limit the need for external components and reduce the total size and power consumption of a device, SoCs are made to be extremely efficient and small. As chip sizes get smaller, so do production costs, energy waste, and the amount of space occupied by large systems. Moreover, the electronics can be made portable as a single chip that controls all system device functionality [1].

2 Field Programmable Gate Arrays (FPGA)

The SoC designs frequently employ FPGAs to offer flexibility and customization in the implementation of the system's functions. A programmable logic device called an FPGA may be set up to carry out a wide range of tasks, from fundamental logic gates to sophisticated digital circuits. Custom logic or processing capabilities that are not offered by the conventional processor or peripheral components can be implemented in an SoC architecture using the FPGA. These can involve activities like high-speed data processing, signal filtering, or specialized input/output (I/O) interfaces. By incorporating an FPGA, the SoC may be tailored for certain uses or applications and is quickly modified as required [2]. For this lab, the PYNQ-Z2 development board was used to perform the required tasks.

3 PYNQ-Z2

The PYNQ-Z2 is a development board developed by Xilinx that is meant specifically for developing embedded systems with the Python programming language.

The board is built around the Xilinx Zynq-7000 system-on-chip (SoC), which combines an FPGA fabric with an Arm Cortex-A9 dual-core CPU. With this special CPU and FPGA setup, users may create high-performance, low-latency embedded systems by combining the processing capabilities of both devices. The FPGA fabric contains 1.3M re-configurable gates with 512 MB of DDR3/Flash memory. The PYNQ-Z2 board features a range of interfaces, including HDMI, Ethernet, USB, and PMOD, which enable developers to interface with a variety of sensors, actuators, and other peripherals. The board also includes a MicroSD slot, which provides additional storage space for data and program code [3]. Figure 2.1 shows the key features of the PYNQ-Z2 board.

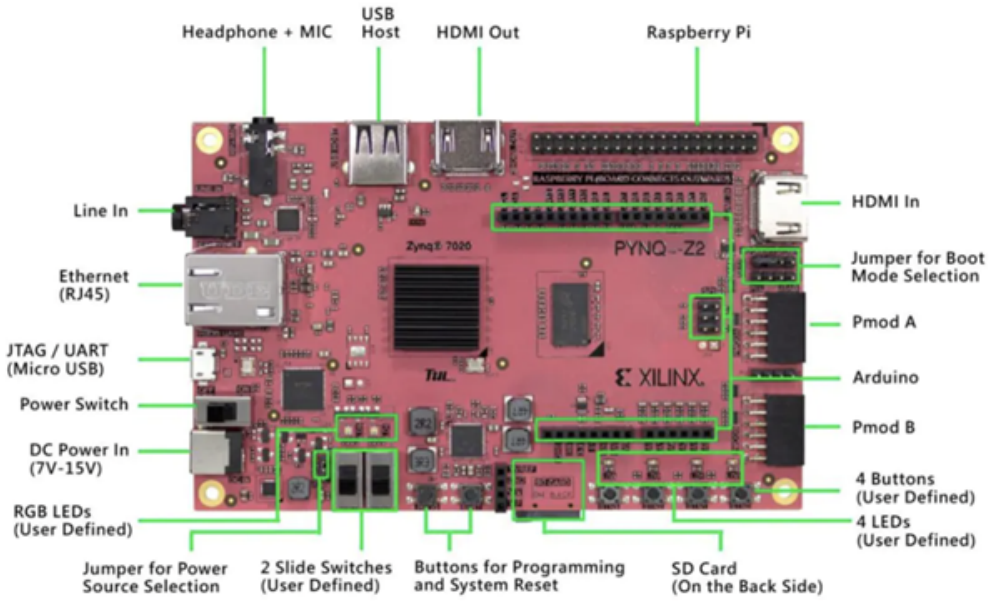

Figure 2.1: PYNQ-Z2 key features [4]

The JTAG/UART port was used in this lab to connect the board to the computer. In order to program and debug the FPGA on the PYNQ-Z2 board, the JTAG (Joint Test Action Group) interface is generally utilized. The UART (Universal Asynchronous Receiver/Transmitter) port, often known as a serial port, is a standard communication interface used for serial communication. For serial connection with a computer or other devices, the PYNQ-Z2 board's UART port can be utilized. When the hardware UART is absent or inoperable, the Xilinx® System Debugger Command-line Interface (XSDB) enables virtual UART using JTAG [5].

3.1 ZYNQ 7000 Architecture

The Zynq-7000 is a Xilinx SoC device series that combines a dual-core ARM Cortex-A9 CPU with FPGA technology. The TUL PYNQ-Z2 is developed based on the

Zynq XC7Z020-1CLG400C, which is a specific device within the Zynq-7000 family of SoCs. Figure 2.2 shows the Zynq-7000 architecture's functional building blocks [6].

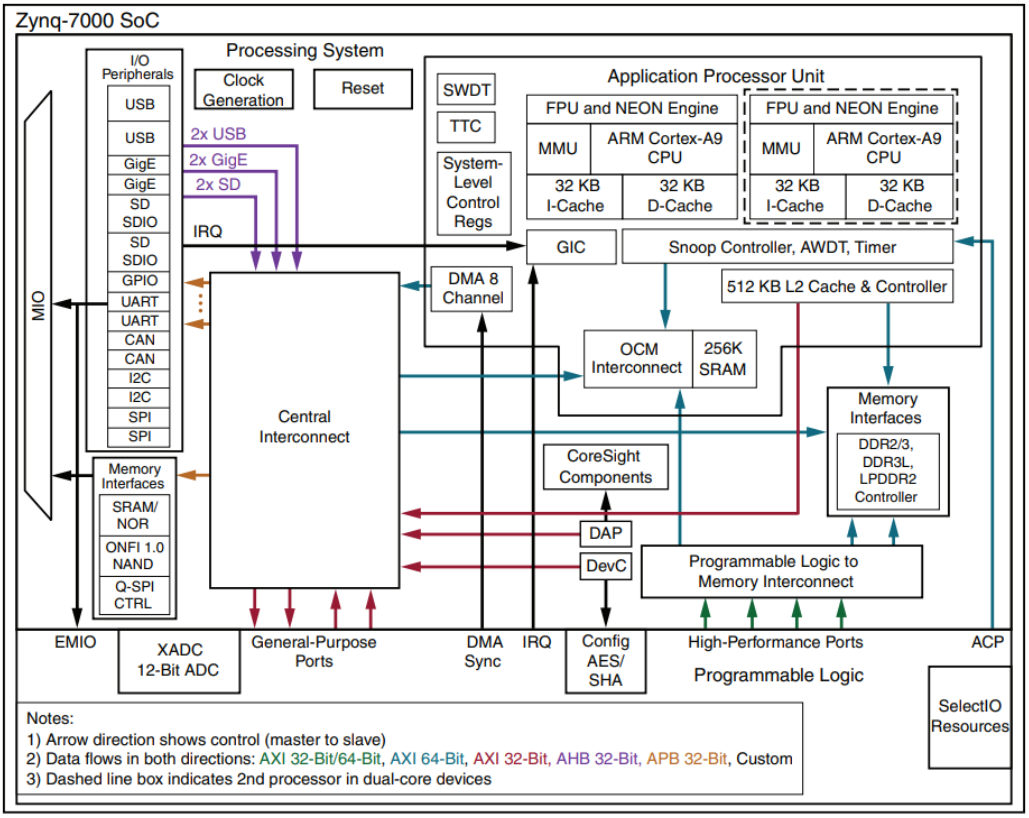

Figure 2.2: Architectural Overview of ZYNQ 7000 family [6]

The complex architecture in figure 2.2 can be simplified as shown in figure 2.3. The processing system (PS) and the programable logic (PL) are the two primary structural components of the Zynq architecture. They both are linked by Advanced eXtensible Interface (AXI) connections which control the transfer of data between PS and PL. PS has a fixed architecture, which means the user cannot adjust or design it. In addition to supporting the operating system and software routines, it hosts the dual-core ARM cortex-A9 processor and system memory. The PL functions as an FPGA, capable of implementing high-speed logic, arithmetic, and data flow subsystems. The peripherals, or functional components, in PL can be customized by the user. There are three major tasks that the peripherals typically complete. Coprocessor is the first one on the list. It serves as a supplement for the main processor. Consequently, the peripherals can improve some operations. The second serves as the cores for connecting to other components like lights and switches. Lastly, extra memory components are stored, including data and instructions [7].

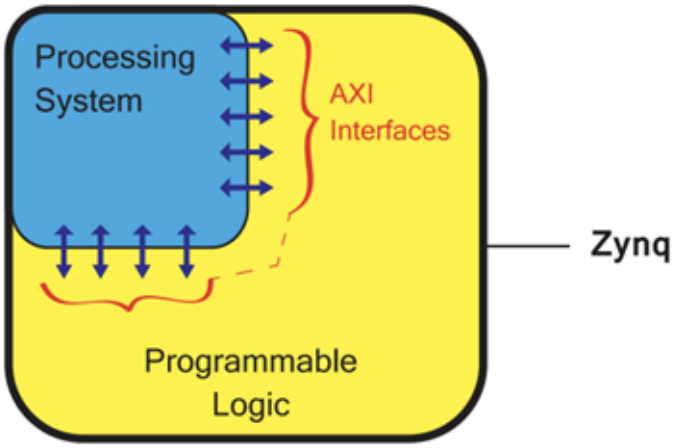

Figure 2.3: The Basic Zynq architecture [7]

4 Vivado IDE

Vivado is an Integrated Development Environment (IDE) for the development and execution of digital circuits and systems. FPGA and SoC devices' designs are the main use for this Xilinx product. A graphical interface for designing, testing, and debugging digital circuits is offered by the Vivado IDE. It has a variety of tools and capabilities, including a design editor, a simulator, and a place-and-route tool, for creating and validating digital circuits. It enables us to create the hardware system, which involves designing the peripheral blocks and other logic in PL before establishing the link to the PS. The peripheral parts of the system design correspond to the Intellectual Property (IP) functional blocks, which are available in the Xilinx libraries in the Vivado IDE. It is a functional block of logic or data that is designed by one party and is reusable.

The PS and PL are connected by the AXI, as well as the link inside the PS and PL itself, such as the link between two peripherals. A single AXI master and a single AXI slave are the two endpoints of the AXI interface where data can go both forward and backward simultaneously [8].

5 Xilinx SDK

The Xilinx SDK is a software development kit (SDK) that Xilinx offers for creating applications for their programmable logic devices, such as FPGAs and SoCs. A variety of tools and libraries are available in the Xilinx SDK for designing, developing, testing, and debugging software programs on Xilinx hardware. It has the IDE based on Eclipse. The Xlinix SDK has support for the Xilinx Vivado design suite for hardware development. It supports C/C++, Java, and Python as the programming

environment [9].

6 SoC Design Flow

The SoC design flow is the process of developing, validating, and testing a sophisticated integrated circuit (IC) that incorporates several functional blocks and complex interconnects on a single chip. A simplified design flow of the SoC is shown in figure 2.4.

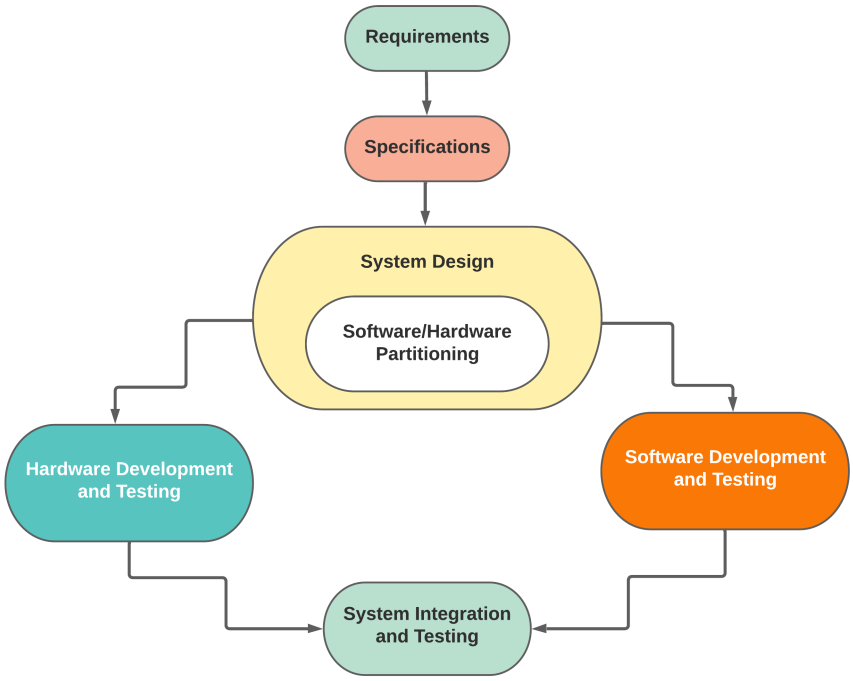

Figure 2.4: SoC Design Flow-chart

From figure 2.4, we can see that to design the SoC, we first have to get the requirements, and then based on the requirements, we will specify the functionality of the design, interfaces, and performance criteria. In the first part of this lab, our goal is to expand the general-purpose input/output (GPIO) functionality of the PYNQ-Z2 board, and in the second part, we will try to trigger interrupt with rude GPIO. So, based on these requirements, the total system design can be approached from a top-down perspective with partitioning into Hardware development and Software development segments. In the following sections, the working principle of the hardware blocks and the idea of implementation will be discussed in detail.

Chapter 3

Methodology

1 Lab 1.1 - Expand the GPIO Functionality

In this part, we will try to control the LEDs in the PYNQ-Z2 board by reading the state of Switch0 and the Push-buttons on the board. The idea is to perform the following functionalities -

1. If **Switch0** is **low** i.e., **'0'**
    - Pushing button 0, LED 0 turns on
    - Pushing button 1, LED 1 turns on
    - Pushing button 2, LED 2 turns on
    - Pushing button 3, LED 3 turns on
2. Else if **Switch0** is **high** i.e., **'1'**
    - All LEDs are blinking with an interval of 500 ms
    - Pushing the buttons has no effect on the LEDs when Switch0 is high

The project workflow of the lab is shown as follows:

- Step 1: Create a Vivado project
- Step 2: Design a block diagram
- Step 3: Generate the bitstream and export the hardware
- Step 4: Create the application using the SDK
- Step 5: Program the Board

We will first discuss the hardware part created by using the Vivado IDE and then we will discuss the software development using the Xilinx SDK based on the hardware specifications. Finally, for the system integration, the combined testing of hardware and software elements will be done while co-debugging with hardware/software cross-triggers.

1.1 Hardware Development

To develop the hardware elements, the Vivado IDE was used. The hardware development generally includes the design of the peripheral and other logic that is implemented in the PL and connecting the peripherals to the PS. The general idea of the connection among the hardware components is shown in figure 3.1 with a simplified block diagram.

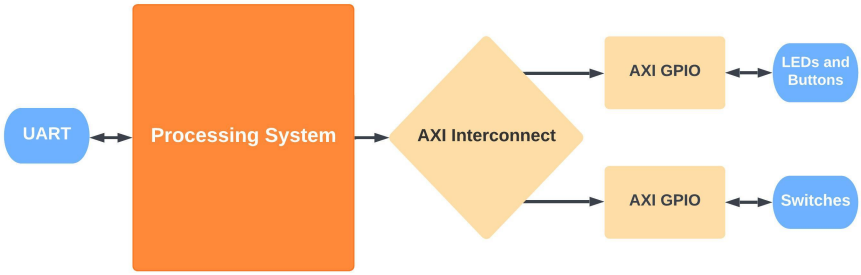

Figure 3.1: Basic Block Diagram of the connections among the Hardware components

The idea here is to create two GPIO components and connect them to the AXI bus to control the operation of the LEDs, Pushbuttons, and Switches. This part is introduced in the PL section of the board. One of the GPIO blocks is connected to the LEDs and Pushbuttons and the other one is connected to the Switches. The two AXI GPIO blocks are connected to the PS of the ZYNQ through AXI interconnect. The AXI Interconnect is a high-performance, adjustable interconnect developed for use in the ZYNQ board's PL component. Inside the PL segment, it is utilized to link several AXI master and slave IP blocks. The AXI Interconnect allows efficient communication across the various IP blocks by offering a high-bandwidth, low-latency communication route. The PS of the Zynq contains the Dual-Core ARM Cortex-A9 Processor, Operating System, Memory Controller, Interrupt Controller, System Timer, DMA Controller, Interconnect, AXI Bus Interface, Peripherals (UART, I2C, SPI, USB, Ethernet). In order to establish communication between the computer and the PS, the UART port is utilized.

1.1.1 AXI GPIO block

The Zynq AXI GPIO block is a module that offers a straightforward interface for handling digital input/output signals in an SoC architecture. The AXI GPIO block is made up of a collection of input/output pins and a programmable set of registers that let the user regulate how these pins function. The PL or the PS of the Zynq SoC can be linked to the AXI GPIO block of the Zynq SoC through the AXI interface. When the AXI GPIO block is linked to the PS, the PS may read from and write to

the AXI registers to access the GPIO block.

The AXI GPIO block may be set up to function in either input mode or output mode. The block reads the input value from the GPIO pins and saves it in a register while it is in input mode. The block drives the output value onto the GPIO pins while in output mode. By writing to the control registers, the GPIO block may be configured using the software. By setting the ”Direction” parameter to either ”Input” or ”Output” in the design block in vivado, the GPIO block can be configured for the desired direction. Reading the input value from the GPIO data register is required while in input mode. In output mode, the GPIO data register must be written with the desired output value [10].

1.1.2 Registers within AXI GPIO block

There are a number of configurable registers in the AXI GPIO block that regulate different elements of its functionality. These registers are accessible by memory-mapped I/O, which means that other system modules can read from or write to them as if they were memory addresses. The different configuration registers within the AXI GPIO block are given below:

AXI GPIO Data Register: There are two AXI GPIO data registers (GPIO DATA and GPIO2 DATA) in the system, one for each channel. The GPIO ports are read from and written to using the AXI GPIO data register.

AXI GPIO 3-State Control Register: There are two AXI GPIO 3-state control registers (GPIO_TRI and GPIO2_TRI), one for each channel. The dynamic configuration of each GPIO pin as an input or an output is managed by this register.

Global Interrupt Enable Register: The Global Interrupt Enable register controls the master enable/disable of the processor’s interrupt output. If the global interrupt is enabled, all interrupt events produced by the GPIO pins will result in the assertion of an interrupt signal on the AXI bus. The system can then utilize this interrupt signal to initiate an interrupt service routine (ISR) in the CPU.

IP Interrupt Enable Register: The IP Interrupt Enable Register (IPIER) controls which GPIO pins can generate interrupts. Each bit in the register corresponds to a distinct GPIO pin, and turning that bit to 1 enables interrupts for that pin.

IP Status Register: The IP Status Register (IPISR) shows which GPIO pins have caused interrupts. When a bit in the register is set to 1, it means that an interrupt has happened on the specific GPIO pin that the bit in the register corresponds to.

Table 3.1 lists the different AXI GPIO configuration registers and their addresses [10].

Address Space Offset	Register Name	Access Type	Default Value	Description
0x0000	GPIO_DATA	R/W	0x0	Channel 1 AXI GPIO Data Register
0x0004	GPIO_TRI	R/W	0x0	Channel 1 AXI GPIO 3-state Control Register
0x0008	GPIO2_DATA	R/W	0x0	Channel 2 AXI GPIO Data Register
0x000C	GPIO2_TRI	R/W	0x0	Channel 2 AXI GPIO 3-state Control Register
0x011C	GIER	R/W	0x0	Global Interrupt Enable Register
0x0128	IP IER	R/W	0x0	IP Interrupt Enable Register (IP IER)
0x0120	IP ISR	R/TOW	0x0	IP Interrupt Status Register

Table 3.1: AXI GPIO Registers and their addresses

1.1.3 Hardware Implementation

We started by creating a Vivado project by adding the board PYNQ-Z2 to the program. Following that, we added the ZYNQ7 Processing System to the block diagram, as shown in Fig 3.2. The block was then set up to operate two blocks of GPIOs. DDR and FIXED_IO must first be linked to the block. DDR serves as the board's main memory, where items like the working file are stored. FIXED_IO provides the connection between the laptop and the UART port. The M_AXI_GPIO is another important port that connects to the AXI interconnection block to enable the peripherals.

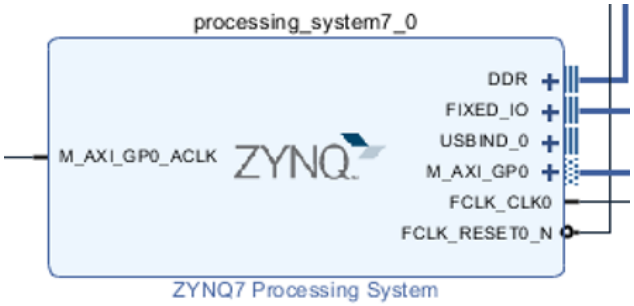

Figure 3.2: Processing System block

Then, in order to expand the functions of the PS, we added two AXI GPIO blocks to the PL section, as illustrated in Fig 3.3. Reading, writing, and receiving interrupts from peripherals are all features of AXI GPIO. It is frequently controlled by software and can be used as an input or an output. The first block provides access to the buttons and LEDs. The Board Interface allows for the selection of LEDs as leds

4bits. On the AXI GPIO core, activate the second channel, then select btns 4bits to connect the buttons. By choosing the Board Interface as sws_2bits, the second block provides access to the switches. Two further blocks are shown in Fig 3.4, Process System Reset Module, and AXI Interconnect will appear after clicking on Run Connection Automation. The Process System Reset Module offers both the block's own reset and the reset for another block.

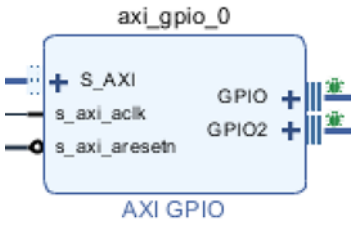

Figure 3.3: AXI GPIO block

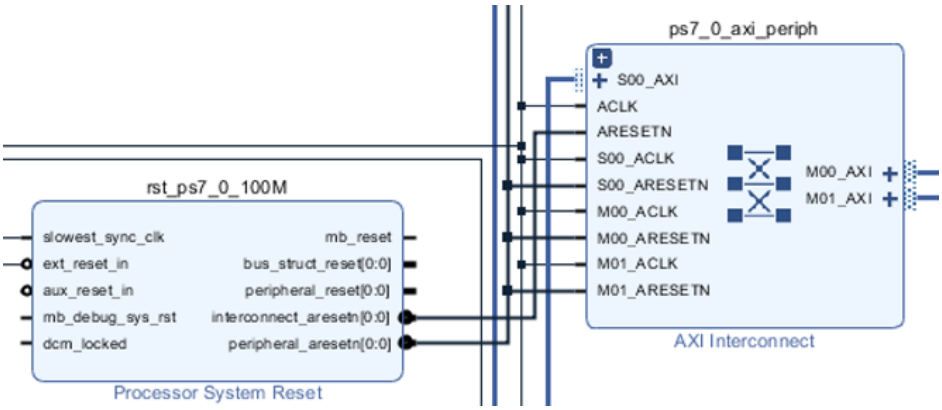

Figure 3.4: Process System Reset Module and AXI Interconnect block

Debugging the GPIO is the next step. This action is optional. This makes it possible for programs to operate properly and display errors. After Run Connection Automation, the System ILA block will be automatically added to the design as seen in Fig 3.5. It is used to monitor the internal signals and interface designs. It offers a wide range of functionality, including triggers and interface debugging. In this lab, it is used to debug the output interfaces from GPIO with the data and trigger option, which allows data to be captured when an event occurs.

To ensure that the block is uninterrupted, we must add the free running clock to the ILA. As a result, the clock pin is linked to the Clocking Wizard instead of the processing system clock. We can build our own clocking network using Clocking Wizard. The System Clock block is connected to the Clocking Wizard as the interface board to adjust the clock speed in the system clock. Overall, the entire block diagram should be shown in Fig 3.6.

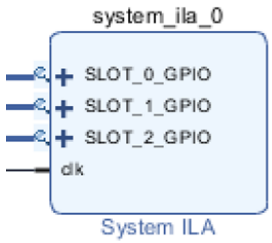

Figure 3.5: System ILA block

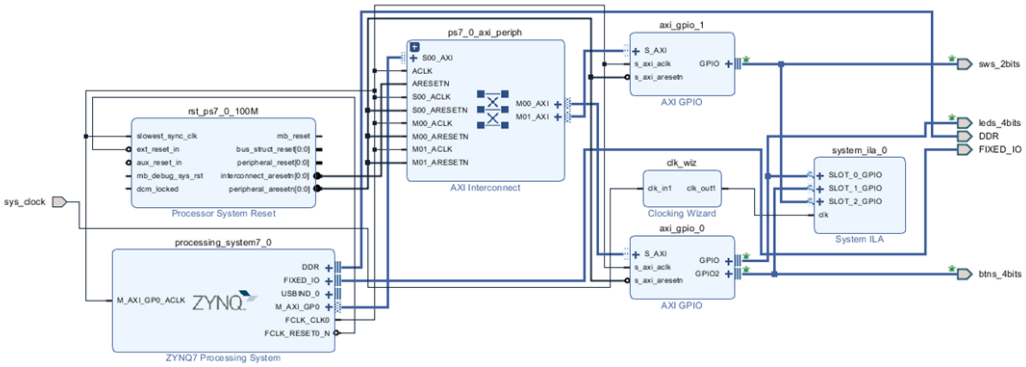

Figure 3.6: Complete Block Diagram

The design must then be validated in order to identify any errors or issues currently on the block diagram. The user can correct any problems or warnings from the validate design before building an HDL wrapper. The HDL wrapper is a top-level entity required for the block design, in which the block design is translated into a source file.

To program the PL, the following step is to generate a bitstream file. The bitstream file is a binary file that contains the configuration information for the board. The board is therefore programmed using a particular bitstream. The last step before developing the software part is to export the hardware. It will export all hardware specifications needed by the SDK.

1.2 Software Development

After exporting the hardware specifications, we move to the software development part using Xlinix SDK. The SDK uses C as its primary language. After selecting Export Hardware in the Vivado IDE section, the Vivado IDE has exported the system.hdf file to the SDK, which contains the Hardware Platform Specification of our design along with an IP block list and the system's address map. The general idea of the program flow to generate the required functionalities is shown in the flowchart in figure 3.7.

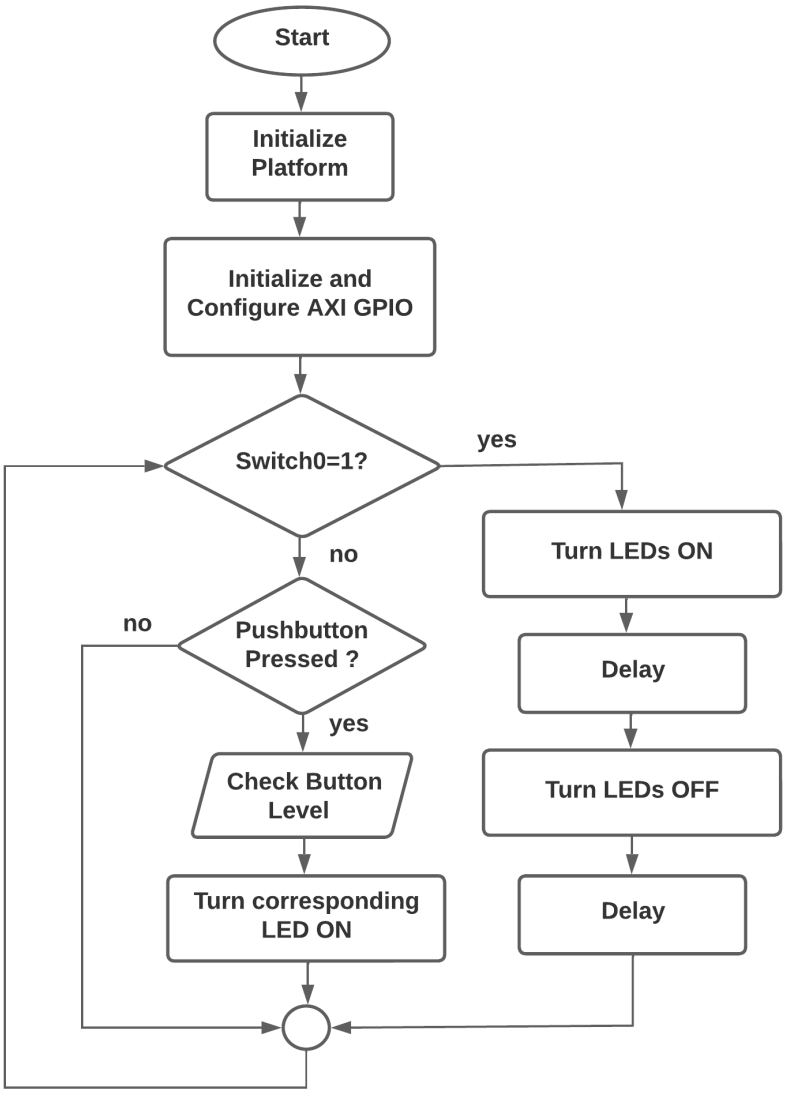

Figure 3.7: Program flowchart for expanding GPIO functionalities

1.2.1 Software Implementation

We start with the helloworld.c template given in SDK. This contains our main() function. To initialize and clean up our platform, we import the init_platform() and cleanup_platform() from the platform.h file. The ARM cortex-A9 CPU's cache memory is activated via the init_platform() function. Cache clearing is done via the cleanup_platform() function. To perform the required functionalities, we added a source file called Gpio_control.c and defined our functions in it. We included the source file in the helloworld.c with Gpio_control.h header file which is used to store the variables and function prototypes.

Helloworld.c

For helloworld.c, we started by including the required header files for the required functions. This main source file contains the stdio.h, platform.h, xil_printf.h, xil_types.h, and gpio_control.h header files. The description of each file is explained as follows:

- **Stdio.h** contains the functions for standard input/output in the C language.
- **Platform.h** defines the hardware platform for a specific design. It typically contains definitions for memory map addresses, interrupt numbers, and other platform-specific information.
- **xil_printf.h** is a header file in the Xilinx SDK that contains the declaration of the xil_printf() function, which is used to print formatted text to the console or serial port of a Xilinx FPGA device during the execution of a program.
- **Xil_types.h** provides definitions for various data types used in Xilinx software development.
- **Gpio_control.h** stores the variables and function prototypes defined to perform the lab module.

Gpio_control.h

As we have mentioned before, Gpio_control.h stores the variables, and functions prototypes that are defined in Gpio_control.c. It also includes the following header files:

- **Xgpio.h** file includes function prototypes and definitions for utilizing the Xilinx GPIO driver in C or C++ programs. It offers functions for reading from and writing to the GPIO pins as well as for initializing and configuring the pins.
- **Xparameters.h** is a header file produced by the Xilinx Vivado toolchain. It includes macro definitions for hardware resources like memory locations, interrupt numbers, and device IDs for the configuration in a certain project.

- **Sleep.h** is a header file in the standard C library that contains routines for temporarily halting program execution.

After initializing the platform, we start by configuring, initializing, and setting the data direction for the AXI GPIO in the main function by using configure_gpio() which is defined in gpio_control.c file. The detailed working principle of this function is given below:

configure_gpio()

It performs the configuration and initialization of those two AXI GPIO peripherals. The XGpio_LookupConfig(GPIO_DEVICE_ID) configures the Gpio with the GPIO_DEVICE_ID specified in the xparameters.h file. The function accepts the device ID as an input parameter and returns a pointer to XGpio_Config struct that includes the GPIO controller's configuration data.

XGpio_CfgInitialize(&Gpio, XGpio_Config, Base Address) function takes three parameters as input. The &Gpio contains the GPIO controller instance to be initialized. XGpio_Config contains the GPIO controller's configuration data obtained from the previous function. The Base Address defines the GPIO controller's base address within the system's address space. Based on these input parameters, the function initializes the GPIO controller in the FPGA fabric.

XGpio_SetDataDirection(&Gpio, channel, bits) is used to configure the GPIO pins. By setting bits set to 0, the &Gpio instance is set as output and by 1 as input. With the channel, it can be set up to the channel to work. In this case, the LEDs and buttons, which are in the same AXI GPIO block, are assigned to input channels 1 and 2, respectively. However, LEDs are the output, as they demonstrate the output light, and buttons are the input, as they are toggled to drive the LEDs. The switches are connected to the other AXI GPIO block and assigned to channel 1 as input as it will control which function to perform on the LEDs.

After we have configured and initialized our GPIO pins as required, we go to our required functionalities, which in this case is toggling the LEDs based on the state of Switch0. The toggle_led() function is used to perform this operation, which is called inside a while loop in the main function so that it continues to run indefinitely. The toggle_led() function is defined in Gpio_control.c which performs the following functionality:

toggle_led()

It is used to control the LEDs. We first check the state of Switche0 with an if-else condition; if Switch0 is low, the pushbutton status will be checked. If the pushbutton is pressed, the LED is turned on, and its location follows the button location. However, if Switch0 is high, all the LEDs will blink with a time interval of 500 ms.

2 Lab 1.2 - The rude GPIO tries to Interrupt

In this part, we will expand the previous lab to trigger an interrupt in the PYNQ-Z2 board by reading the state of Switch1. The idea is to perform the following functionality along with the previous one -

1. If **Switch1** is **high**, an interrupt is triggered.
    - The LEDs will flash an SOS signal using Morse Code
        - ... — ...
        - short, short, short, long, long, long, short, short, short
        - Short pulse is 500 ms
        - Long pulse is 1500 ms

The project workflow of the lab is shown as follows:

- Step 1: Open the Vivado project from lab 1.1
- Step 2: Configure the block diagram to introduce the interrupt
- Step 3: Generate the bitstream and export the hardware
- Step 4: Create the application using the SDK
- Step 5: Program the Board

We will first discuss the hardware configuration done using the Vivado IDE and then we will move to the configuration in the software part based on the new hardware specifications.

2.1 Hardware Development

To get the processor's attention, interruptions can be applied at any moment. It halts the processor's ongoing tasks. When an interrupt occurs, the CPU will store its current state before running the interrupt service routine. After executing the interrupt routine, the processor will restore the context and return to the main flow again.

2.1.1 Generic Interrupt Controller (GIC)

A Generic Interrupt Controller (GIC), which is included in the processing system, is used by Zynq to execute an interrupt. In addition to disabling, masking, and prioritizing the interrupt sources, the GIC can function as an enabler, allowing PL to interrupt PS. The GIC accepts interrupts from peripherals and external devices including timers, UARTs, and other system parts. To identify the source of the interrupt, each

interrupt source is given a distinct Interrupt ID (IID) number. The GIC notifies the ARM processor of an interrupt by sending it a signal whenever it happens. The processor pauses the job it is working on and launches the ISR for the relevant interrupt source. The ISR takes the appropriate steps to handle the interrupt after reading the IID from the GIC. This can entail changing the system state, reading data from the peripheral that caused the interrupt, or carrying out some other action. After that, the ARM processor picks up where it left off, and the system keeps running normally up until the next interrupt [7].

To perform the operation on GIC we need to modify the block diagram from lab 1.1 as shown in figure 3.8. As indicated by the red line in Fig 3.8, the interrupt is carried out between the peripheral switches block and the Generic Interrupt Controller (GIC) located inside the processing system. The AXI GPIO block will alert the GIC when the interrupt occurs. The GIC then locates, masks, and activates the interrupt. After executing the interrupt function, the processor resumes its previous function.

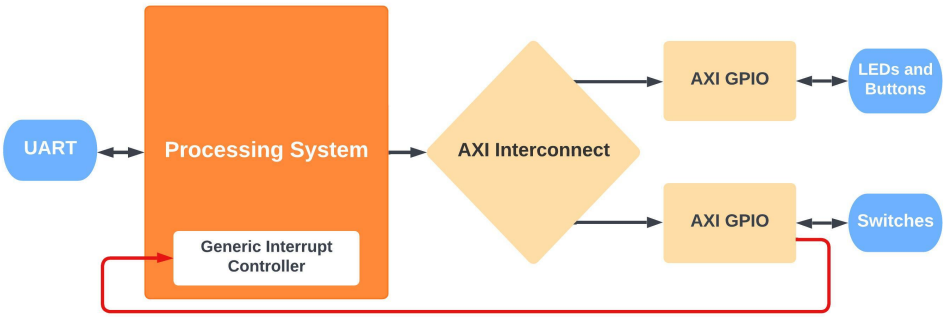

Figure 3.8: Basic Block Diagram to introduce interrupts in the system

2.1.2 Hardware Implementation

In the Vivado IDE, we modified one AXI GPIO block (for switches) to be a dual-channel device because the interrupt is enabled. The interrupt will be sent out by the output port named ip2intc_irpt, as shown in Fig 3.9.

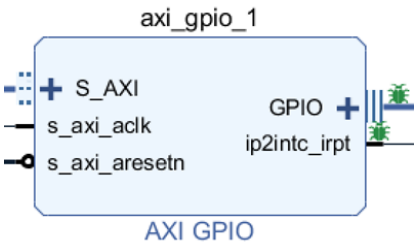

Figure 3.9: AXI GPIO block with interrupts

We also need to enable the interrupts on the processing system block so that it can receive the interrupt from the peripherals through the input port named IRQ_F2P[0:0], as shown in Fig 3.10.

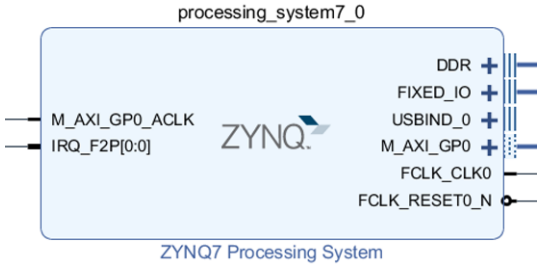

Figure 3.10: Processing System block with interrupts

The debugging is performed for the interrupt. Also, the free running clock is connected to the system ILA for the interrupt. The complete block diagram with interrupts is shown in Fig 3.11.

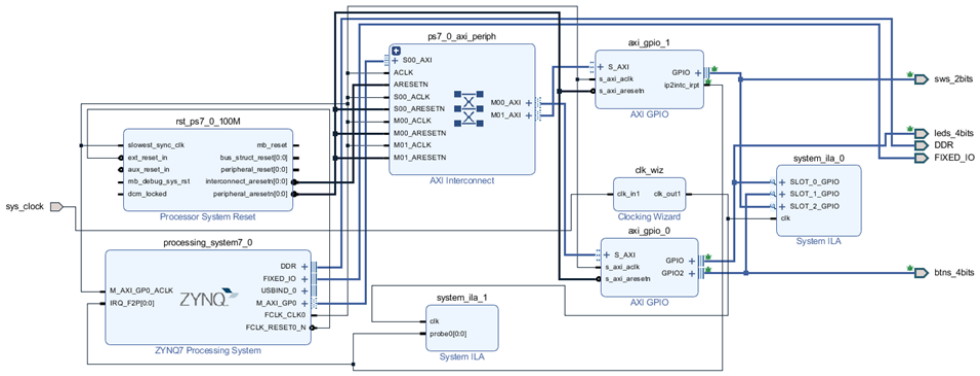

Figure 3.11: Complete Block Diagram with interrupts

2.2 Software Development

After exporting the new hardware specifications, we move to Xlinix SDK to program our board as required to enable the interrupt function in our system. The general idea of the program flow to include the interrupt in the program developed in lab 1.1 is shown in figure 3.12.

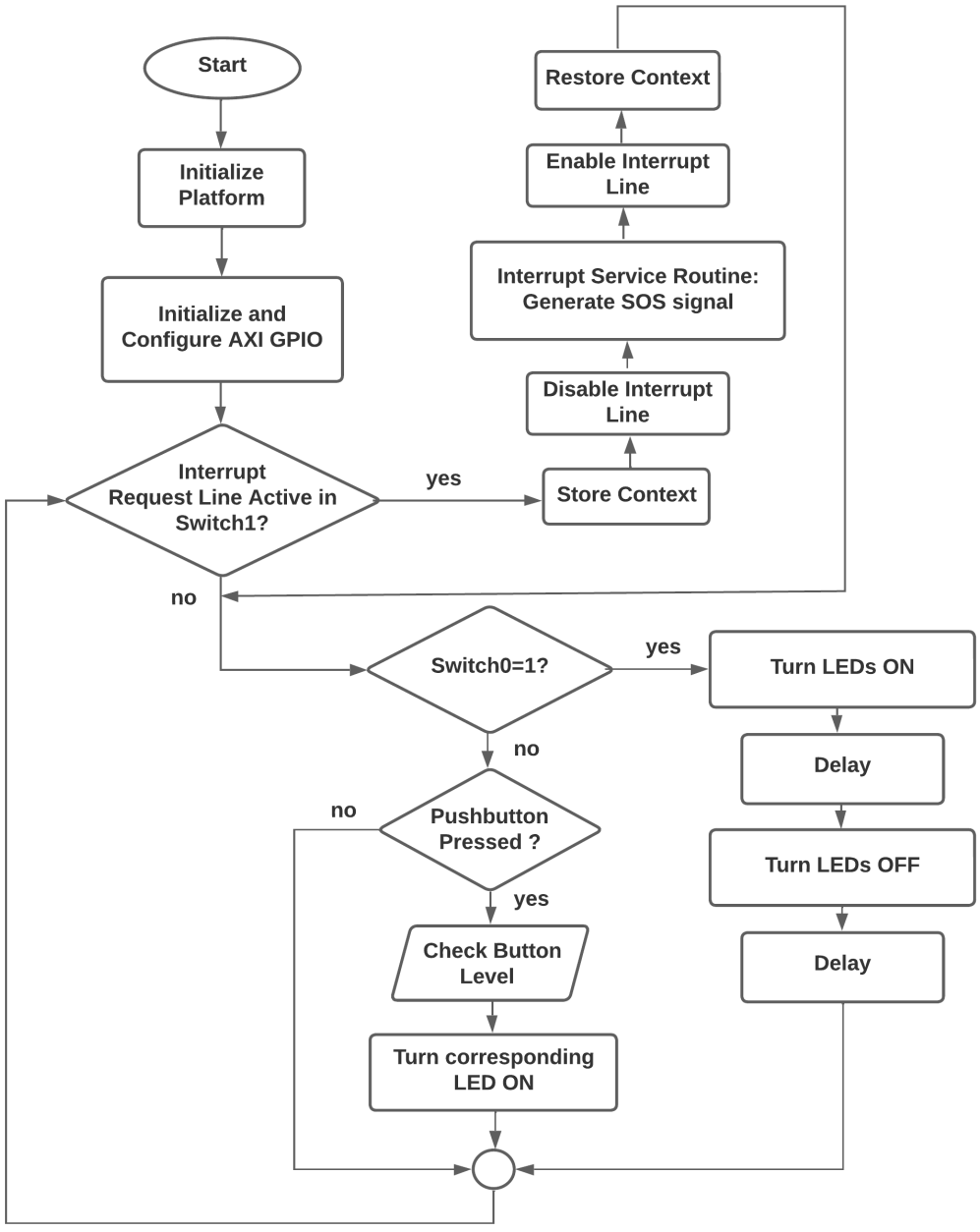

Figure 3.12: Program flowchart to introduce interrupt

From the program flow chart, we can see that the system will trigger an interrupt when Switch1 is high. Then it will store the context and disable the interrupt line until the interrupt routine has been carried out, which is in this case generating the SOS signal. After executing the routine, it will enable the interrupt line again and restore the context. The program flow to generate the SOS signal is given in figure 3.13.

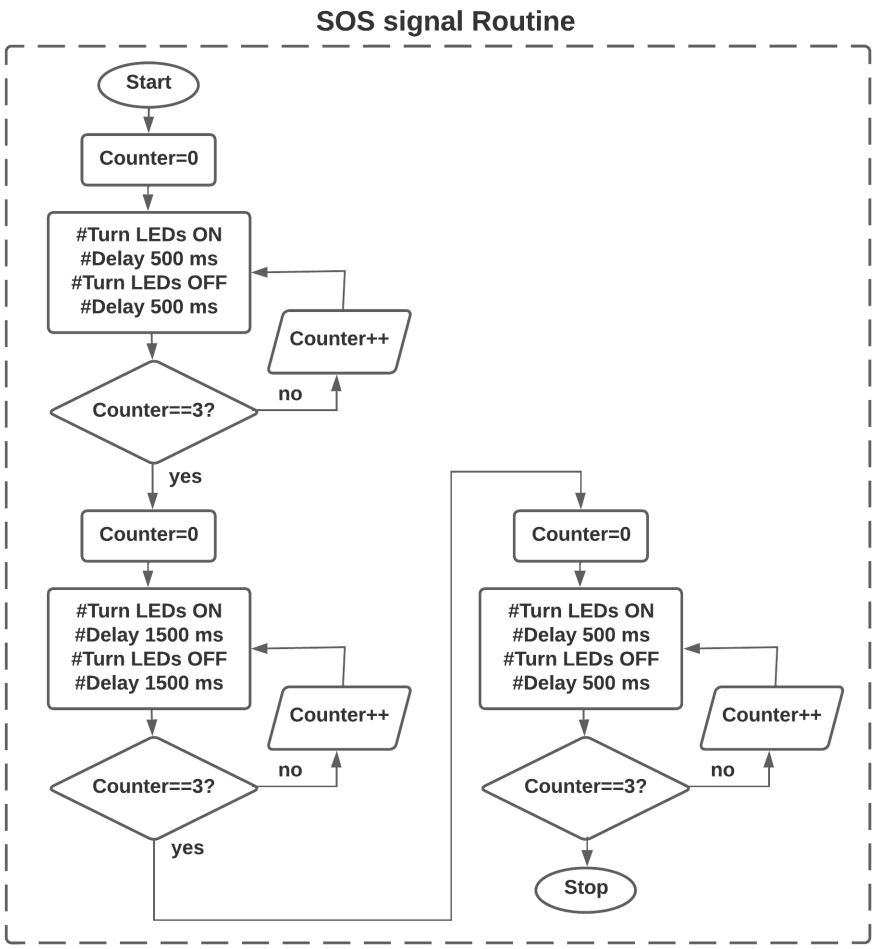

Figure 3.13: Program flow chart to generate SOS signal

2.2.1 Software Implementation

We start with the previously configured helloworld.c file again in SDK. We will add the function intr_setup() outside the while loop to configure and initialize the GIC. The function is defined in Gpio_control.c.

Gpio_control.h

To carry out the functionality of interrupt in the system, we need to include some new header files in Gpio_control.h like xcugic.h and xil_exception.h, that contains the functions required to configure and initialize the GIC.

- **Xscugic.h** provides the function prototypes and data types required to utilize the GIC.
- **Xil_exception.h** provides a set of functions and macros that can be utilized to handle exceptions, like interrupt.

After that, we start by configuring, initializing, and setting the AXI GPIO and GIC for interrupt. We defined the intr_setup() and intr_handler() functions to carry out this operation in the Gpio_control.c. The detailed working principle of these functions is given below:

Intr_setup()

XScuGic_LookupConfig(XPAR_PS7_SCUGIC_0_DEVICE_ID) provides a pointer to SuGic_config to the configuration structure of the ScuGic device driver, which provides details about the device's base address and the number of interrupt inputs.

XScuGic_CfgInitialize(&Inst, SuGic_config, Base Address) function takes three parameters as input. The &Inst is a pointer to the ScuGic instance to be initialized, SuGic_config contains the GIC configuration data obtained from the previous function, and Base Address is the base address of the CPU that the ScuGic is connected to. Based on these parameters, the function initializes the GIC in the FPGA fabric.

XGpio_InterruptEnable(&Gpio, XGPIO_IR_CH1_MASK) requires two arguments, a pointer &Gpio to the GPIO instance for which interrupts are to be enabled and a mask XGPIO_IR_CH1_MASK, that determines which interrupt channels to activate.

XGpio_InterruptGlobalEnable(&Gpio) function is used to activate interrupts for a GPIO device at the global level, allowing the device to create interrupts when specified events occur at the Gpio instance specified to trigger the interrupt.

Xil_ExceptionRegisterHandler() function is called with three arguments. The first argument is the ID for an interrupt exception, the second one is the custom function

to handle the interrupt event, and the third one is a pointer to the SuGic instance that was initialized.

Xil_ExceptionEnable() enables handling exceptions in the system.

XScuGic_Connect() takes four arguments as input, a pointer to the interrupt controller instance, the ID for the interrupt signal generated by the GPIO device driver, the custom function that will be executed during an interrupt, and a pointer to the GPIO instance that checks for the interrupt signal.

XScuGic_Enable() takes two arguments as input, a pointer to the interrupt controller instance and the ID for the interrupt signal generated by the GPIO device driver. This function activates the interrupt signal produced by the GPIO device driver, and the corresponding interrupt handler function is called by the interrupt controller, who then forwards the interrupt event to the processor.

intr_handler()

The XGpio_InterruptDisable() function first disables the interrupt line in AXI GPIO, so that the interrupt can't be triggered again before the routine is executed. Then we check the status of Switch1 and if it is high, the LEDs will generate an SOS signal as shown in figure 3.13. After that, the interrupt status in the specific channel of the GPIO device is cleared with XGpio_InterruptClear(). The interrupt line is enabled again to trigger the next interrupt with the XGpio_InterruptEnable() function.

The function we defined, in this case, will generate an SOS signal only for one cycle when we trigger the interrupt by changing the state of Switch1 from low to high. After that, it returns to its original routine, even if Switch1 is kept high. If we want to generate a continuous SOS signal as long as Switch1 is kept high, we can do that by introducing a loop in our interrupt handler function code. In that case, the SOS signal will be continued to generate until the state of Switch1 is changed back to low.

2.3 ILA Debugging

Next, we perform the hardware debugging using the logic debugging IP, which is Integrated Logic Analyzer (ILA) debugging. Before going through Vivado, we debug the software by selecting Launch on Hardware (System Debugger). The ILA debugging is to inspect and capture the internal signal in the design and verify that it functions correctly. It enables us to see the current status of the FPGA and the internal signal while the board is running.

For the system_ila_0 block. There are three interface slots.

- SLOT_0_GPIO is connected to the LEDs.
- SLOT_1_GPIO is connected to the buttons.
- SLOT_2_GPIO is connected to the switches.

For the system_ila_1 block, it connects to the interrupt between the AXI GPIO switch block and PS.

The trigger position in the window is automatically set to 512, as shown in Fig 3.14. It shows the amount of data that has been captured before an event occurs. In this case, 512 data are captured before the event occurs. The Window data depth represents the amount of data captured in one capture window. It is set to 1024. As a result, the window captures 512 data before the event and another 512 data after the event. So that we can observe the data behavior both before and after the event happens. The red line with the T alphabet as shown in Fig 3.14 points to the position where the condition happens.

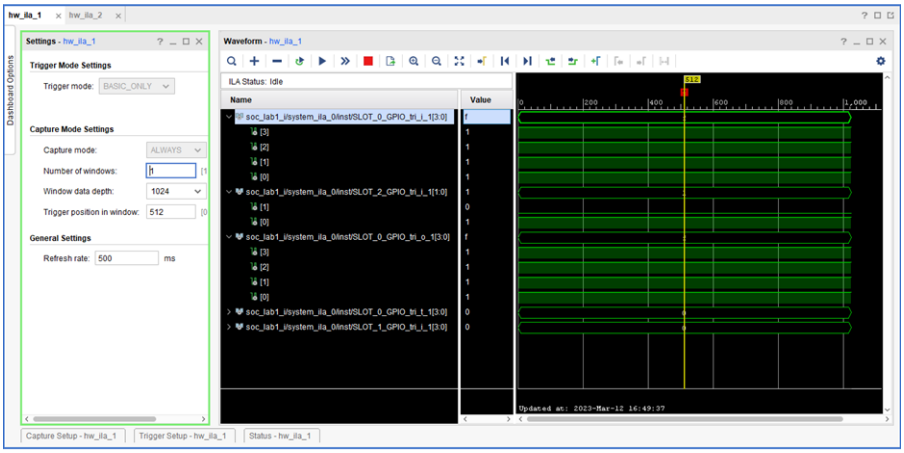

Figure 3.14: HARDWARE MANAGER with data captured

Without any specific condition, when we click on the Run Trigger, the results are shown as in Fig 3.14. We set the switch 0 as high (1); thus, SLOT_2_GPIO is high. Then all four LEDs are on.

2.3.1 ILA Triggers with Specific Behavior

In the Trigger setup windows, we can capture the data by choosing the specific signal. When the event in the signal occurs, the program starts capturing the data. We choose SLOT_2_GPIO for the trigger setup and set the value to 1 as shown in Fig 3.15. Therefore when the switches are set to 1, the data starts to be captured.

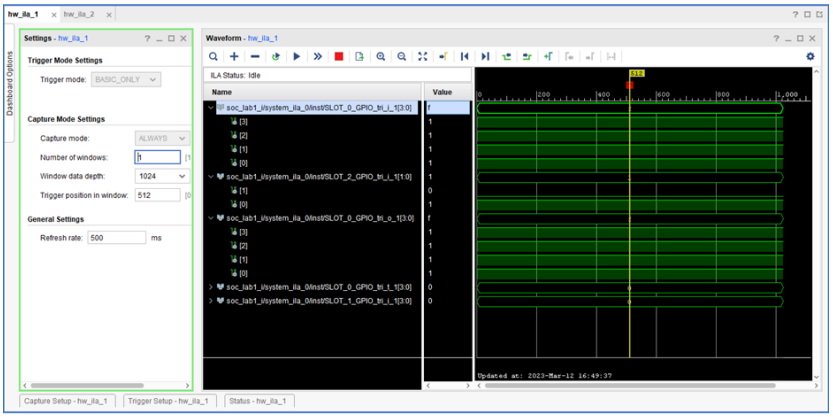

Figure 3.15: ILA debugging results

On the Status window, as shown in Fig 3.15, the Core status shows as Idle. After running the trigger for the ILA core, the status will be displayed as Waiting for Trigger. So, we need to trigger the condition by switching the switch to start capturing the data. The internal signal in Fig 3.15 can be simply explained. When the switch in SLOT_2_GPIO is triggered, the LEDs are on.

2.3.2 ILA Debugging with Interrupts

The ILA debugging with interrupts shows only one internal signal, which represents the interrupt from the AXI GPIO switch block. As shown in Fig 3.16, when the signal is 1, the interrupt is triggered.

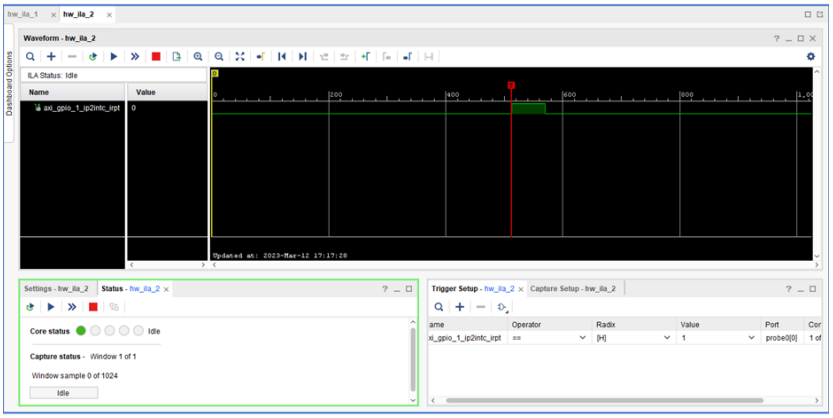

Figure 3.16: ILA debugging for interrupt part

Chapter 4

Conclusion

In this lab, we have studied some of the fundamental concepts of SoC and embedded systems. The PYNQ-Z2 board is a versatile and powerful platform that can be used for a wide range of applications. The Zynq-7000 Architecture in the PYNQ-Z2 board (SoC device), developed by Xilinx, was also studied. Using the PYNQ-Z2 board we performed some GPIO control functionalities and triggered interrupt signals. The Vivado IDE is used to design the hardware component, while the Xilinx SDK is used to develop the program code, which is given in the Appendix. We demonstrated how to use the board to toggle LEDs based on the state of switches and push-buttons. We also showed how to generate an SOS signal using interrupt signals.

References

[1] What is a system on chip (soc)? [Online]. Available: https://anysilicon.com/what-is-a-system-on-chip-soc/, (accessed: 28.02.2023).

[2] What is an fpga? field programmable gate array, https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html, (Accessed on 03/11/2023).

[3] "Xup pynq-z2." (), [Online]. Available: https://www.xilinx.com/support/university/xup-boards/XUPPYNQ-Z2.html. (accessed: 28.02.2023).

[4] "Dfrobot pynq-z2 development board." (), [Online]. Available: https://www.mouser.be/new/dfrobot/dfrobot-pynqz2-dev-board/. (accessed: 01.03.2023).

[5] Using jtag uart, https://www.xilinx.com/htmldocs/xilinx2018_1/SDK_Doc/xsct/use_cases/xsdb_using_jtag_uart.html, (Accessed on 03/11/2023).

[6] Ds190-zynq-7000-overview.pdf • viewer • documentation portal, https://docs.xilinx.com/v/u/en-US/ds190-Zynq-7000-Overview, (Accessed on 03/11/2023).

[7] M. A. E. L. H. Crockett R. A. Elliot and R. W. Stewart, The Zynq Book. Strathclyde Academic Media, 2014.

[8] "Axi reference guide." (), [Online]. Available: https://docs.xilinx.com/v/u/en-US/ug1037-vivado-axi-reference-guide. (accessed: 01.03.2023).

[9] Software development kit (sdk), https://www.xilinx.com/products/design-tools/legacy-tools/sdk.html, (Accessed on 03/11/2023).

[10] Pg144-axi-gpio.pdf • viewer • documentation portal, https://docs.xilinx.com/v/u/en-US/pg144-axi-gpio, (Accessed on 03/11/2023).

Appendix

1 main.c

```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xil_types.h"
#include "Gpio_control.h"


int main() {

        xil_printf("initialization of the platform\n\r");
    init_platform();

    // Configure GPIO 0 and 1
    xil_printf("initialization and configuration of the AXI GPIO\n\r");
    // function call to configure the GPIO
    configure_gpio();

    // setup the interrupt for AXI GPIO 1
    intr_setup();

    xil_printf("Blink says the LED\n\r");
    while (1) {
        // function call to toggle the LEDs
        toggle_leds();
    }


    // Clean the code
    xil_printf("cleanup of the platform\n\r");
    cleanup_platform();
    return 0;
}
```

2 gpio_control.c

```
/*
 * Gpio_control.c
 *
 *  Created on: Mar 2, 2023
 *      Author: user
 */


#include "Gpio_control.h"

/**
 * Function to configure the AXI GPIO blocks
 */
void configure_gpio() {
        //objects that holds the configuration for the GPIO (based on the device ID)
        cfg_ptr = XGpio_LookupConfig(XPAR_AXI_GPIO_0_DEVICE_ID);
        cfg_ptr2 = XGpio_LookupConfig(XPAR_AXI_GPIO_1_DEVICE_ID);
        // initialize the AXI GPIO 0 based on the configuration
        XGpio_CfgInitialize(&axi_gpio_0, cfg_ptr, cfg_ptr->BaseAddress);
        // initialize the AXI GPIO 1 based on the configuration
        XGpio_CfgInitialize(&axi_gpio_1, cfg_ptr2, cfg_ptr2->BaseAddress);
        // set data direction
        // bit 0 --> output, bit 1 --> input
        // set the LEDs on input channel 1 as output (0)
        XGpio_SetDataDirection(&axi_gpio_0, 1, 0);
        // set the buttons on input channel 2 as input (1)
        XGpio_SetDataDirection(&axi_gpio_0, 2, 1);
        //set switches on input channel 1 as input (1)
        XGpio_SetDataDirection(&axi_gpio_1, 1, 1);
}

/**
 * Function to toggle the LEDs
 */
void toggle_leds() {

        // GPIO read on switches
        // XGpio_SetDataDirection(instance_of_GPIO, channel)
        switches_check = XGpio_DiscreteRead(&axi_gpio_1, 1);

        // Turn on switch 0 as low
        if(switches_check == 0b00000000 || switches_check == 0b00000010){
                buttons_check = XGpio_DiscreteRead(&axi_gpio_0, 2);
                // When the buttons are checked (pressed) --> write value to LED
            XGpio_DiscreteWrite(&axi_gpio_0, 1, buttons_check);
            usleep(500); // 500 s

         // Else turn on switch 0 as high
         }
         else{
```

```
                XGpio_DiscreteWrite(&axi_gpio_0, 1, 0b00001111);
                usleep(500000); // 500 ms
                XGpio_DiscreteWrite(&axi_gpio_0, 1, 0b00000000);
                usleep(500000); // 500 ms
        }

}

/**
 * Setup function for the interrupt
 */
void intr_setup() {
        // initialize the interrupt controller
        intr_controller_cfg_ptr = XScuGic_LookupConfig(XPAR_PS7_SCUGIC_0_DEVICE_ID);
        XScuGic_CfgInitialize(&intr_controller, intr_controller_cfg_ptr,
                        intr_controller_cfg_ptr->CpuBaseAddress);

        // enable the interrupt on the AXI GPIO 1
        XGpio_InterruptEnable(&axi_gpio_1, XGPIO_IR_CH1_MASK);
        XGpio_InterruptGlobalEnable(&axi_gpio_1);

        // connect the interrupt controller to the interrupt-handling hardware
        Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
                        (Xil_ExceptionHandler) XScuGic_InterruptHandler, &intr_controller);
        Xil_ExceptionEnable();

        // connect the AXI GPIO interrupt to the interrupt controller
        XScuGic_Connect(&intr_controller, XPAR_FABRIC_AXI_GPIO_1_IP2INTC_IRPT_INTR,
                        (Xil_ExceptionHandler) intr_handler, &axi_gpio_1);

        // enable the AXI GPIO interrupt at the interrupt controller
        XScuGic_Enable(&intr_controller, XPAR_FABRIC_AXI_GPIO_1_IP2INTC_IRPT_INTR);

}

/**
 * Interrupt handler
 */
void intr_handler() {

        // Disable the AXI GPIO interrupts
        XGpio_InterruptDisable(&axi_gpio_1, XGPIO_IR_CH1_MASK);

        // PUT CODE HERE FOR INTERRUPT HANDLER
        // PERFORM THE FUNCTIONALITY
        switches_check = XGpio_DiscreteRead(&axi_gpio_1, 1);

        if(switches_check == 0b00000010 || switches_check == 0b00000011) {
                for (idx = 0; idx <3; idx++) {
                        XGpio_DiscreteWrite(&axi_gpio_0, 1, 0b00001111);
                        usleep(500000); // 500 ms
                        XGpio_DiscreteWrite(&axi_gpio_0, 1, 0b00000000);
                        usleep(500000); // 500 ms
```

```
                }
                for (idx = 0; idx <3; idx++) {
                        XGpio_DiscreteWrite(&axi_gpio_0, 1, 0b00001111);
                        usleep(1500000); // 1500 ms
                        XGpio_DiscreteWrite(&axi_gpio_0, 1, 0b00000000);
                        usleep(1500000); // 1500 ms
                }
                for (idx = 0; idx <3; idx++) {
                        XGpio_DiscreteWrite(&axi_gpio_0, 1, 0b00001111);
                        usleep(500000); // 500 ms
                        XGpio_DiscreteWrite(&axi_gpio_0, 1, 0b00000000);
                        usleep(500000); // 500 ms
                }

        }

        // clear the interrupt
        (void) XGpio_InterruptClear(&axi_gpio_1, XGPIO_IR_CH1_MASK);

        //Re-enable the AXI GPIO interrupt
        XGpio_InterruptEnable(&axi_gpio_1, XGPIO_IR_CH1_MASK);
}
```

3 gpio_control.h

```
/*
 * Gpio_control.h

 *
 *  Created on: Mar 2, 2023
 *      Author: user
 */

#ifndef SRC_GPIO_CONTROL_H_
#define SRC_GPIO_CONTROL_H_

#include "xgpio.h"
#include "xil_types.h"
#include "xscugic.h"
#include "xparameters.h"
#include "sleep.h"
#include "xil_exception.h"

// variables
// AXI GPIO
XGpio_Config *cfg_ptr; // contain GPIO configuration 1
XGpio_Config *cfg_ptr2; // contain GPIO configuration 2
XGpio axi_gpio_0; // LEDs and buttons
XGpio axi_gpio_1; // switches
u8 switches_check;
u8 idx;
u8 buttons_check;

// interrupt
XScuGic_Config *intr_controller_cfg_ptr;
XScuGic intr_controller;

// function prototypes
void configure_gpio();
void toggle_leds();

// prototype functions for the interrupts
void intr_setup();
void intr_handler();

#endif /* SRC_GPIO_CONTROL_H_ */
```