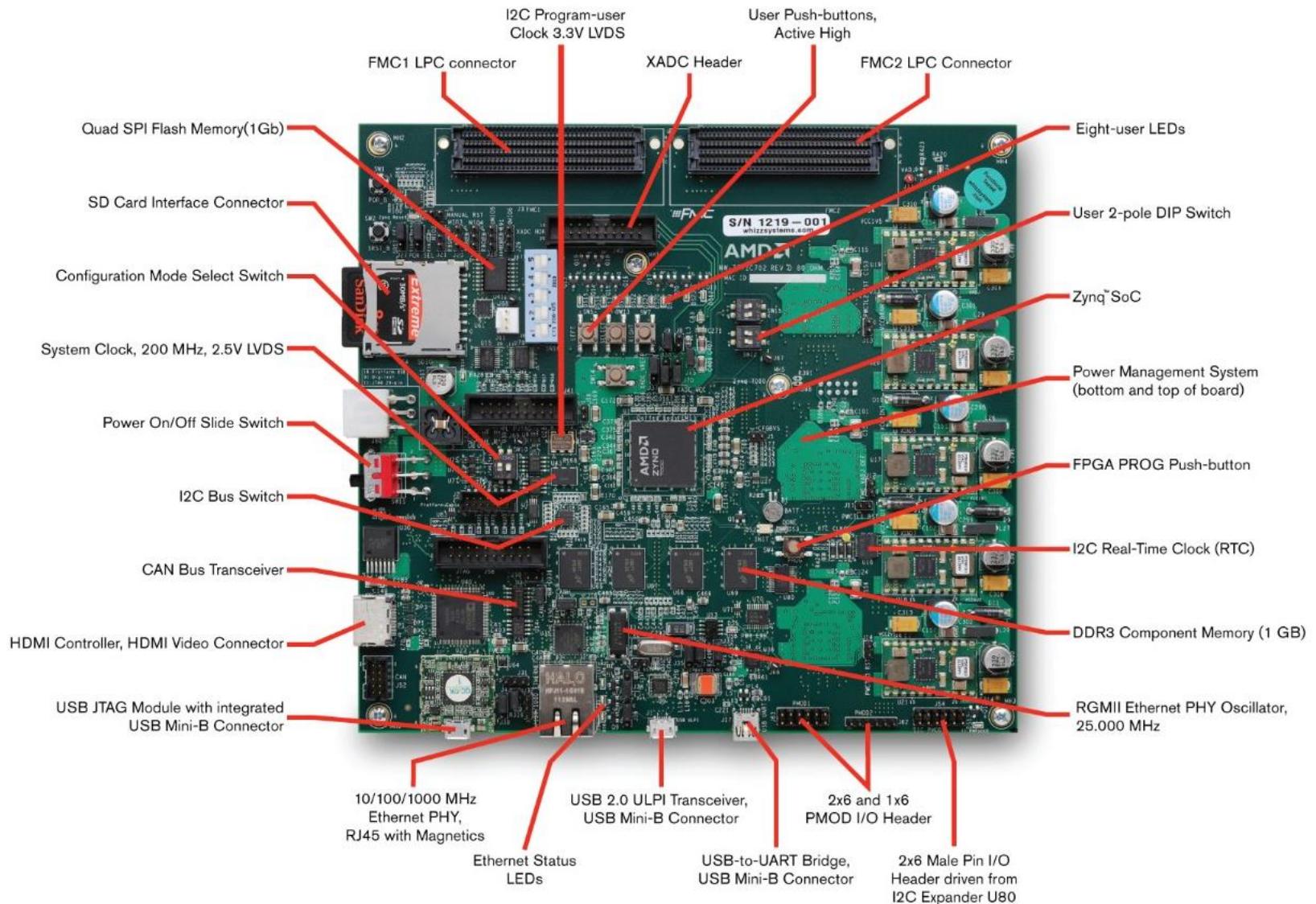




Zynq™ 7000 SoC & Vitis Lab

AMD
together we advance_

Zynq 7000 SoC ZC702 Evaluation Kit

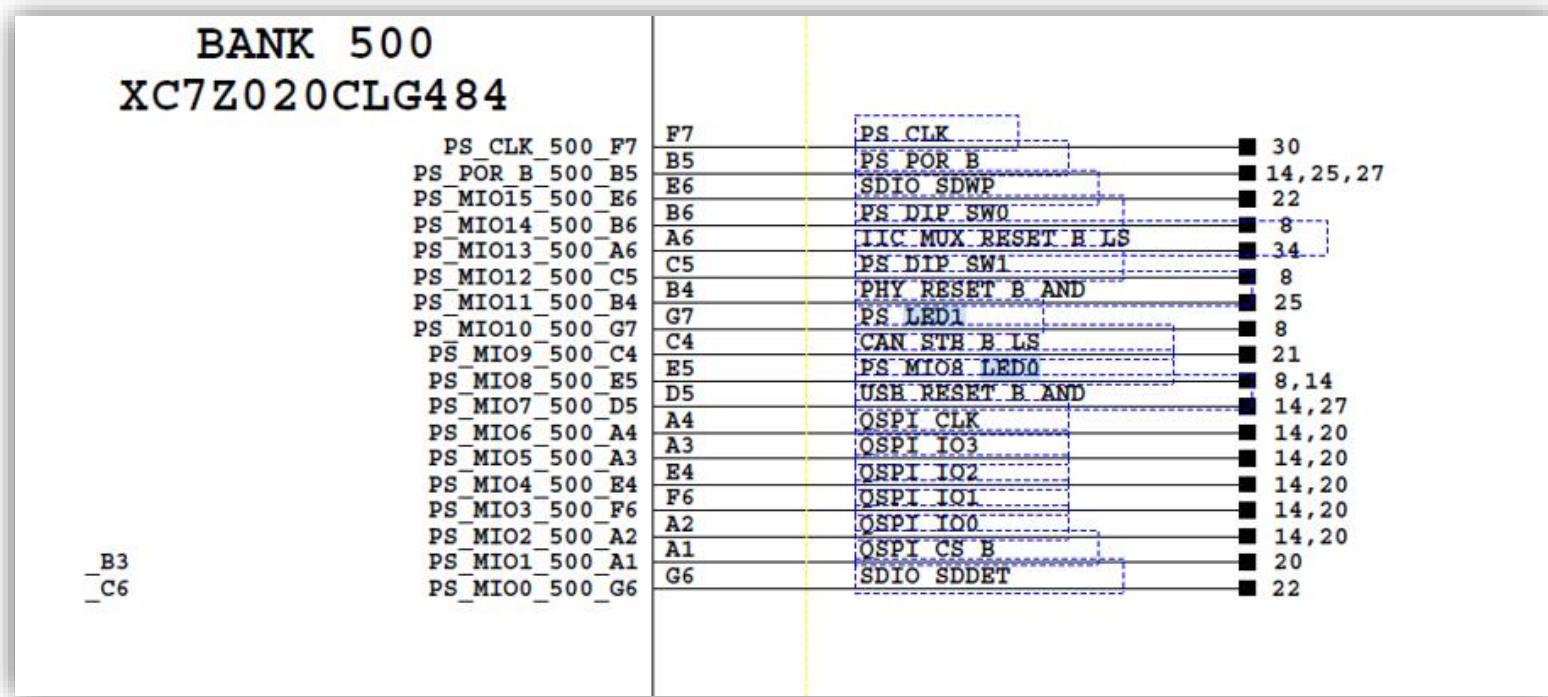


GPIO Read & Write (MIO&EMIO&AXI GPIO)

MIO Configuration

Both MIO and EMIO are pins directly controlled by the CPU, and Vivado offers default interface configurations for MIO, making it easily configurable through the GUI.

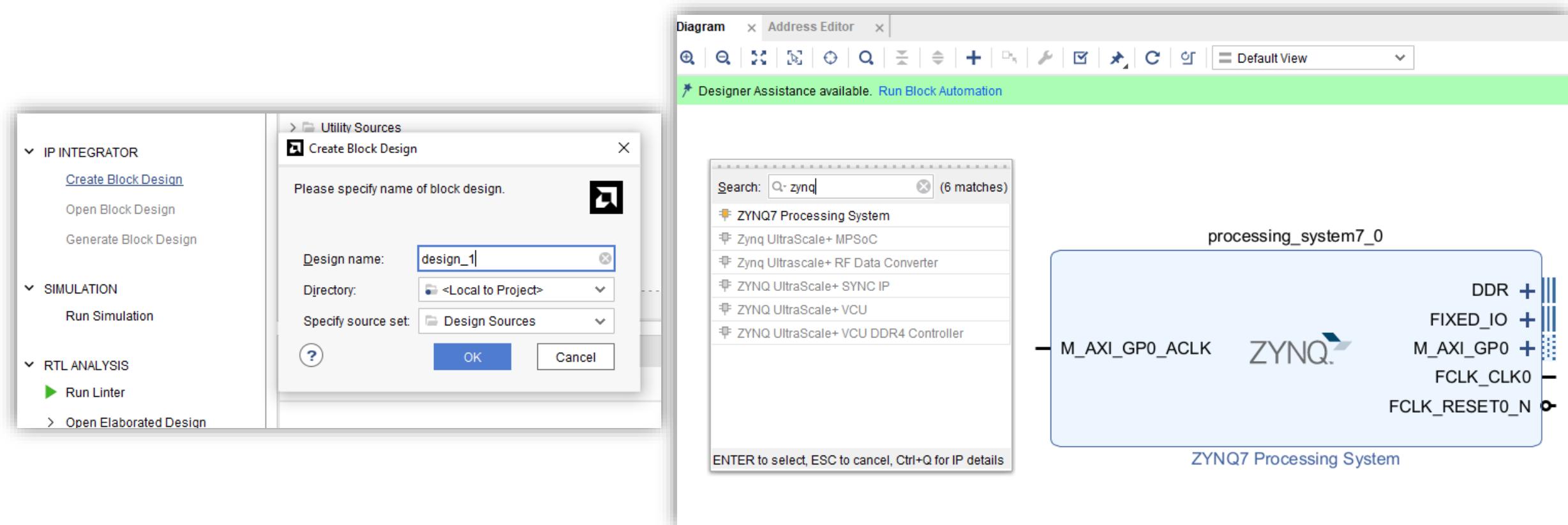
As we are using the development board for learning purposes, some of the MIO pins have been pre-configured. In this LAB, We will be utilizing an LED connected to MIO10 and a Button connected to MIO12.



Vivado Block Design

We will accomplish the hardware design of the Zynq through the Block Design (BD) functionality in Vivado.

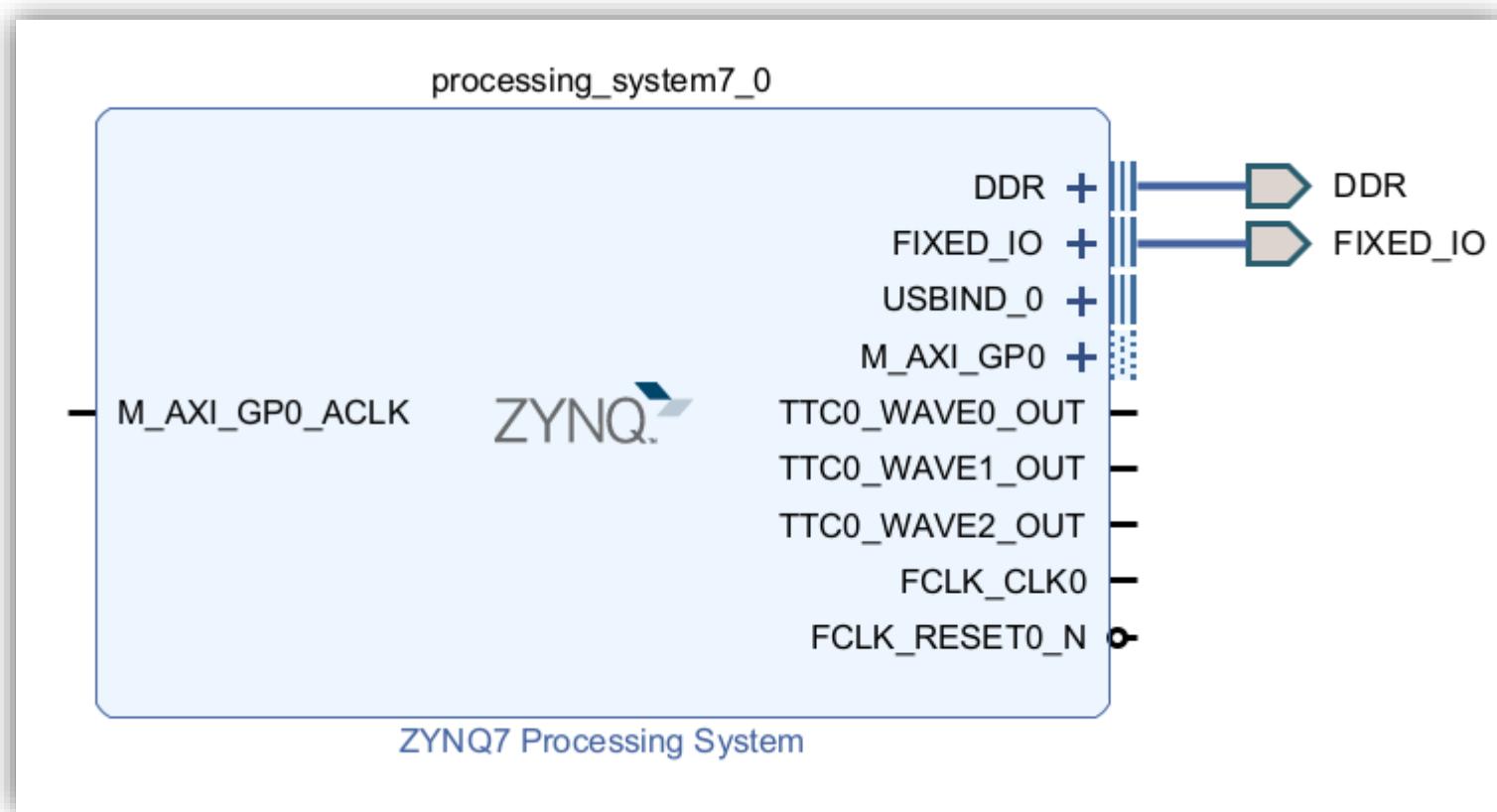
Vivado can easily instantiate the ZYNQ IP Core via GUI..



Vivado Block Design

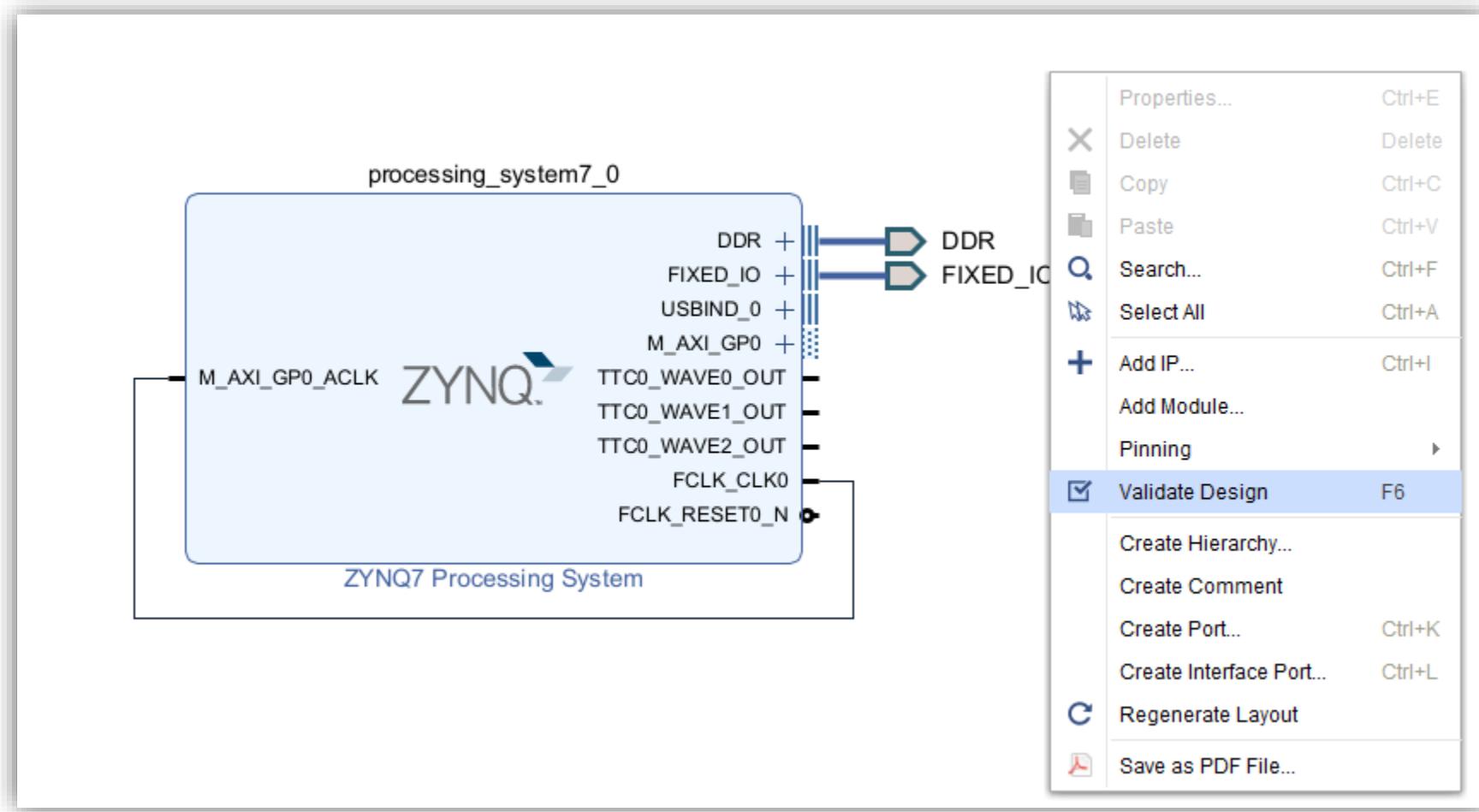
Pressing the “Run Block Automation” will automatically handle the most basic configuration in the software.

A fundamental Zynq design should minimally include a DDR and external I/O connectivity.



Vivado Block Design

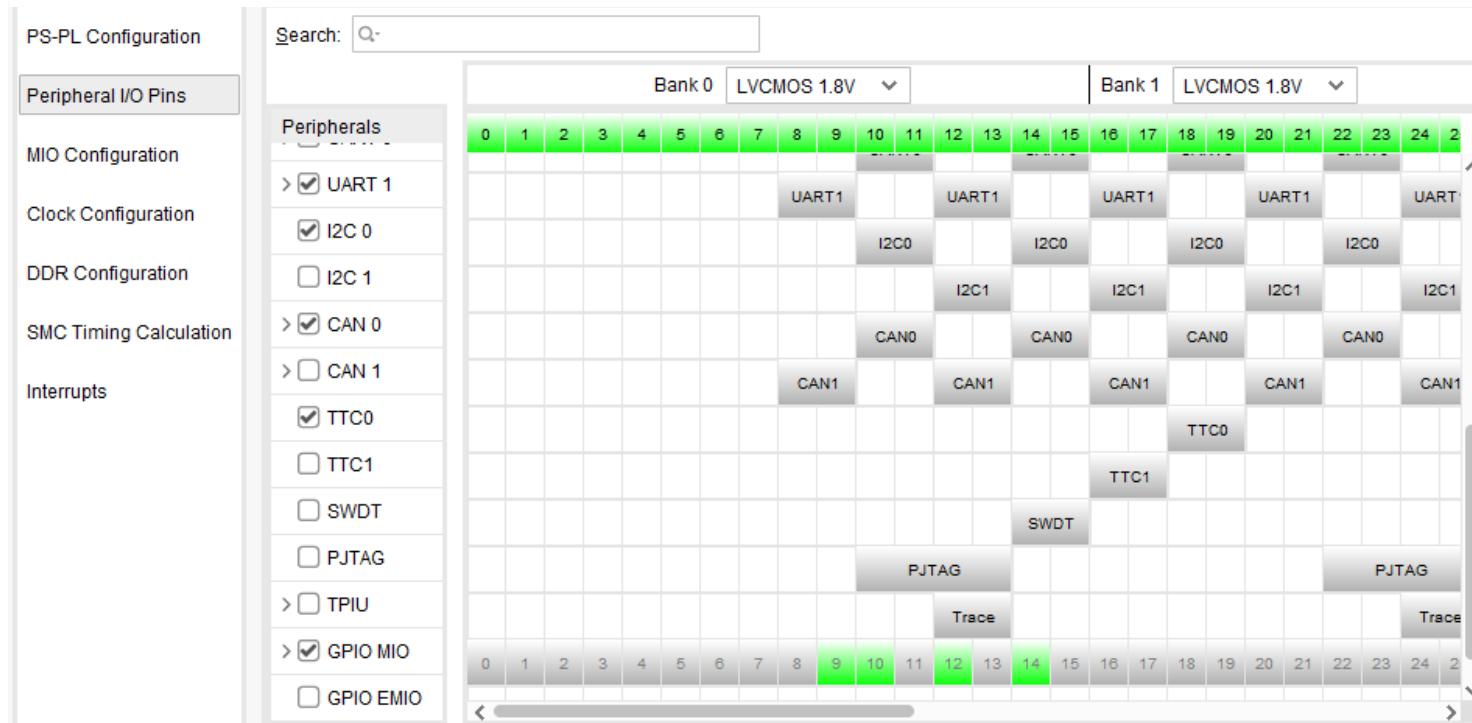
Upon pressing the check button, the software will perform a Design Rule Check (DRC) once.



MIO Configuration

As mentioned earlier, ensure that MIO10 and MIO12 are configured as GPIO.

(Option) Additionally, for ease of design, consider enabling UART1 for use.



EMIO Configuration

Each pin in EMIO is configured as a GPIO and utilizes resources from the Programmable Logic (PL) side.

Therefore, EMIO needs to be configured manually. In this LAB, we will need to allocate one EMIO for a BUTTON on the Programmable Logic (PL) side.

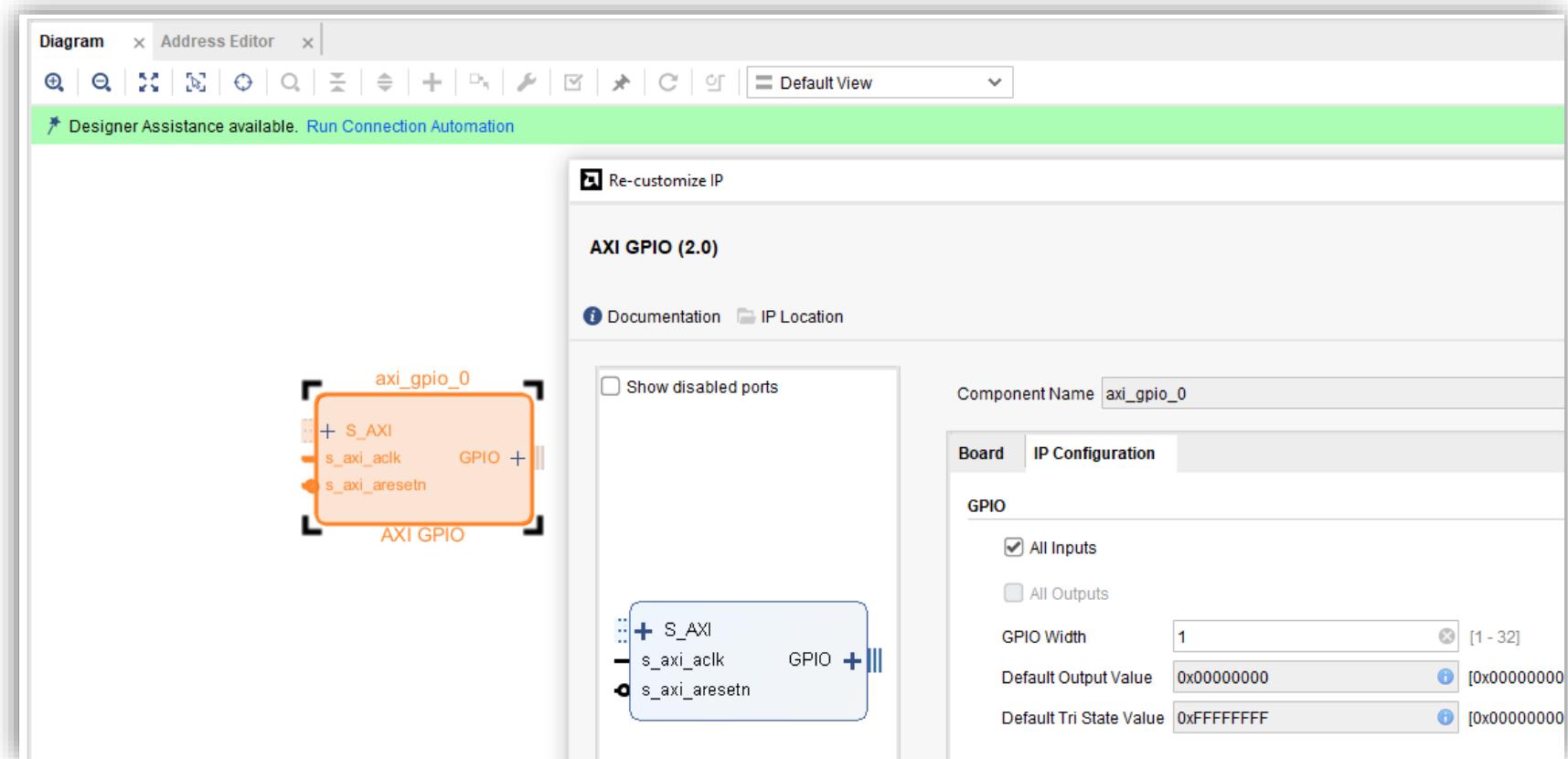
The screenshot shows the Xilinx Vivado Design Suite interface. On the left, the 'MIO Configuration' tab is selected, displaying a table for peripheral assignments. The table has columns for Peripheral, IO, Signal, and IO Type. Under the Peripheral column, checkboxes are present for I2C 1, SPI 0, SPI 1, CAN 0, CAN 1, and GPIO. For CAN 0, the IO dropdown is set to 'MIO 46 .. 47'. Under the GPIO section, 'GPIO MIO' is checked, and 'EMIO GPIO (Width)' is set to 1. Other checkboxes for ENET Reset, USB Reset, and I2C Reset are also present. Below the table, there are sections for Application Processor Unit and Programmable Logic Test and Debug.

On the right, the 'Block Diagram' view shows a ZYNQ7 Processing System. A blue box labeled 'processing_system7_0' contains several IP cores: M_AXI_GP0_ACLK, TTC0_WAV, TTC0_WAV, TTC0_WAV, FCLK, and FCLK_RESET. A context menu is open over the 'GPIO' block, listing options such as Block Interface Properties, Highlight, Unhighlight, Delete, Copy, Paste, Search, Select All, Add IP, Add Module, Make External (which is highlighted in blue), Pinning, and Validate Design.

AXI GPIO Configuration

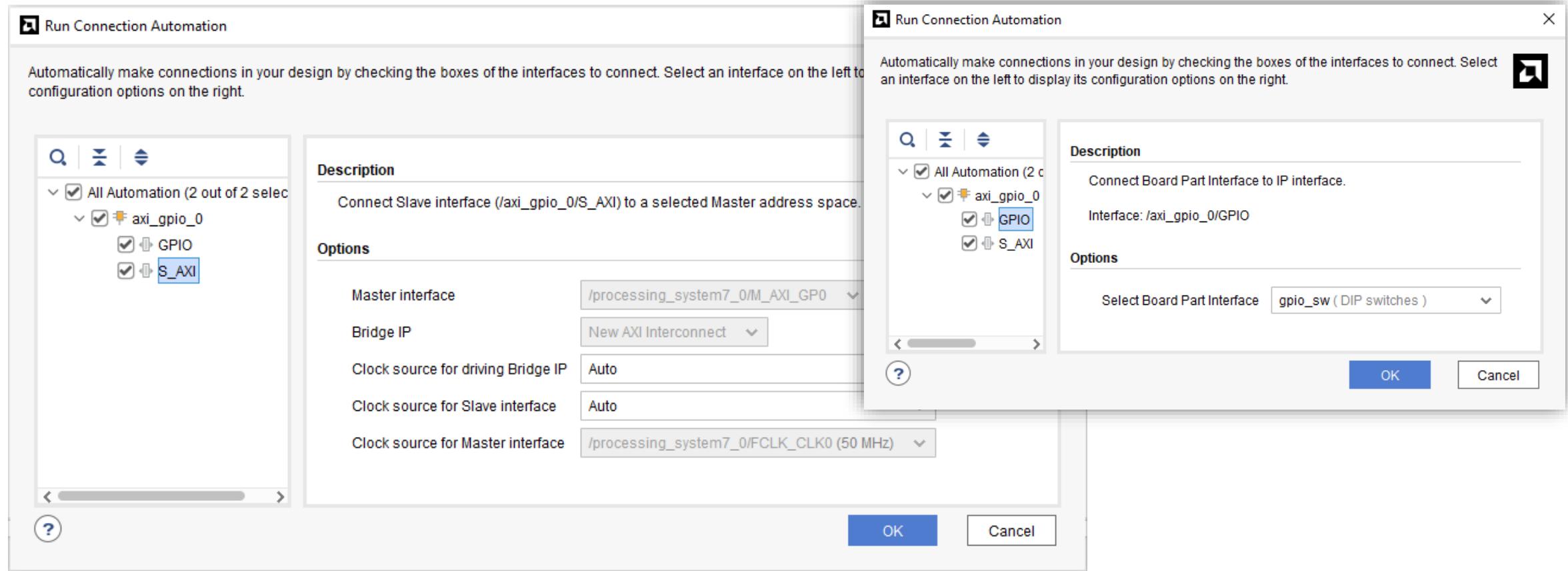
Next, utilize the button on the Programmable Logic (PL) side.

Invoke the AXI GPIO IP and set the width to one bit.



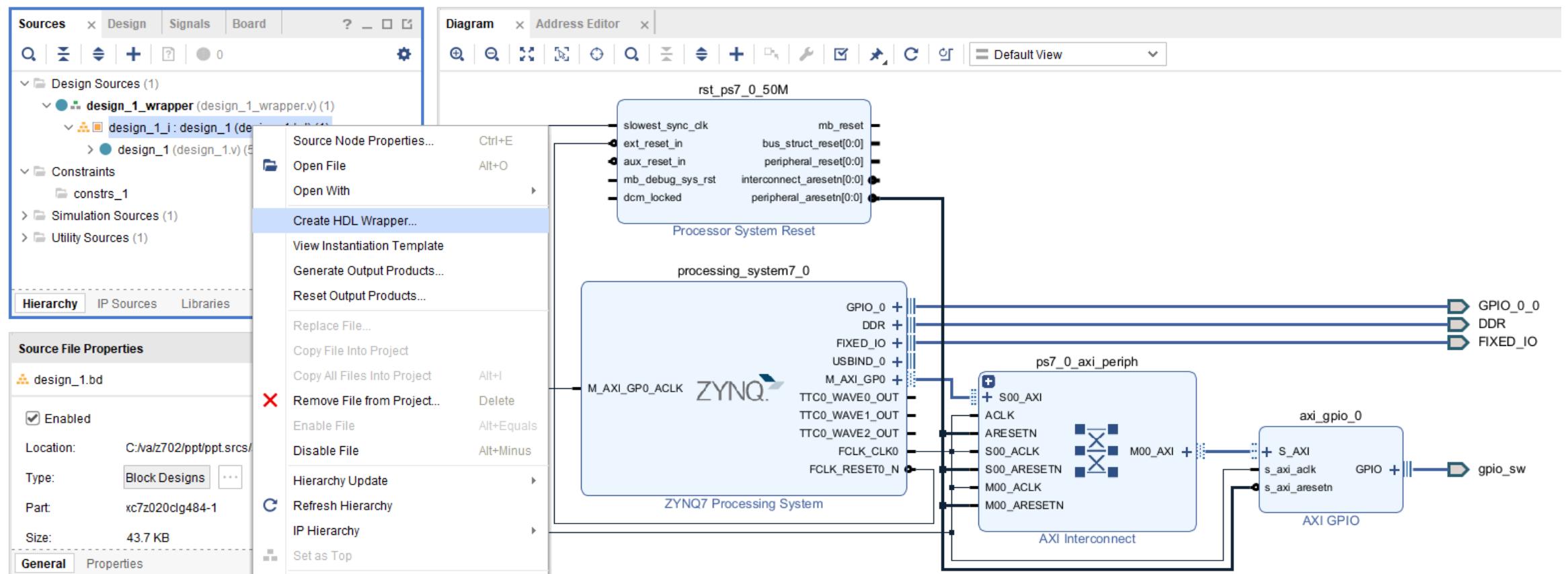
Run Connection Automation

Pressing the “Run Connection Automation” will automatically connect the selected object.



Run Connection Automation

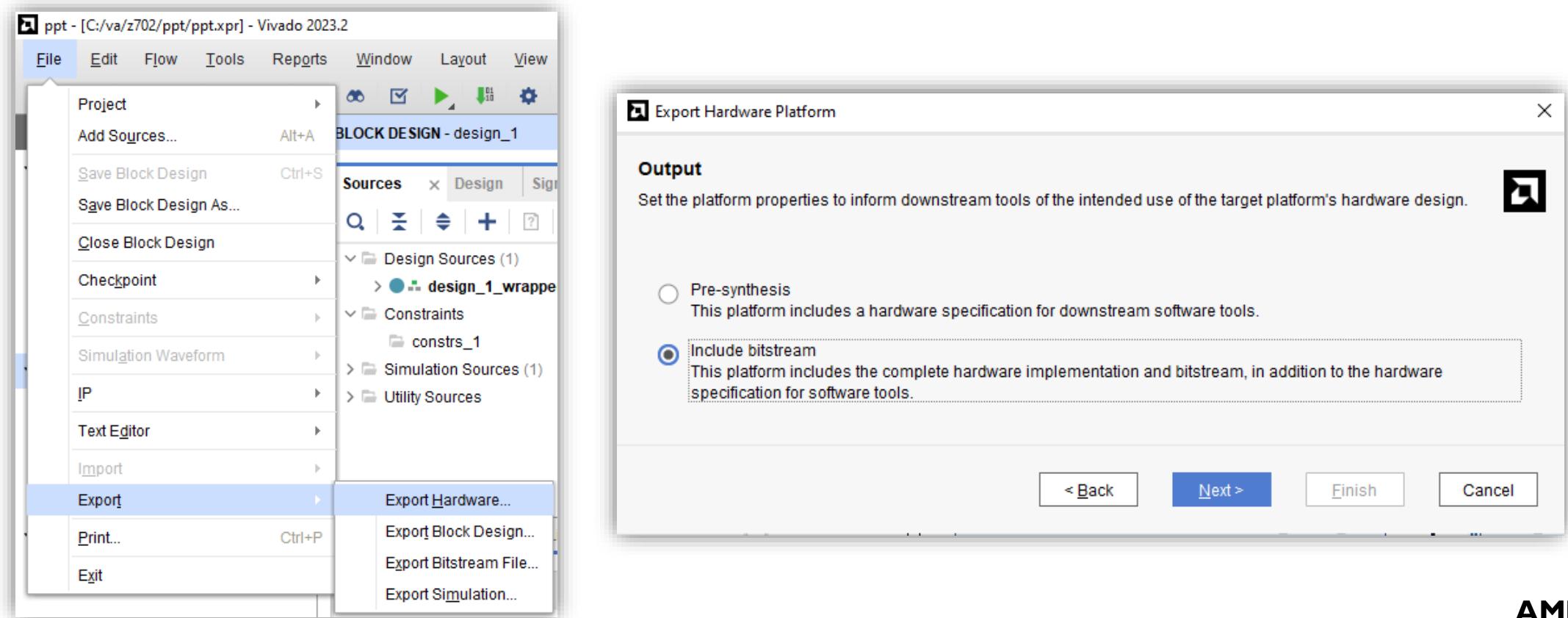
Subsequently, the software will complete the connection-related design, and then you can generate the top-level file output as an XSA file.



Output xsa file to Vitis

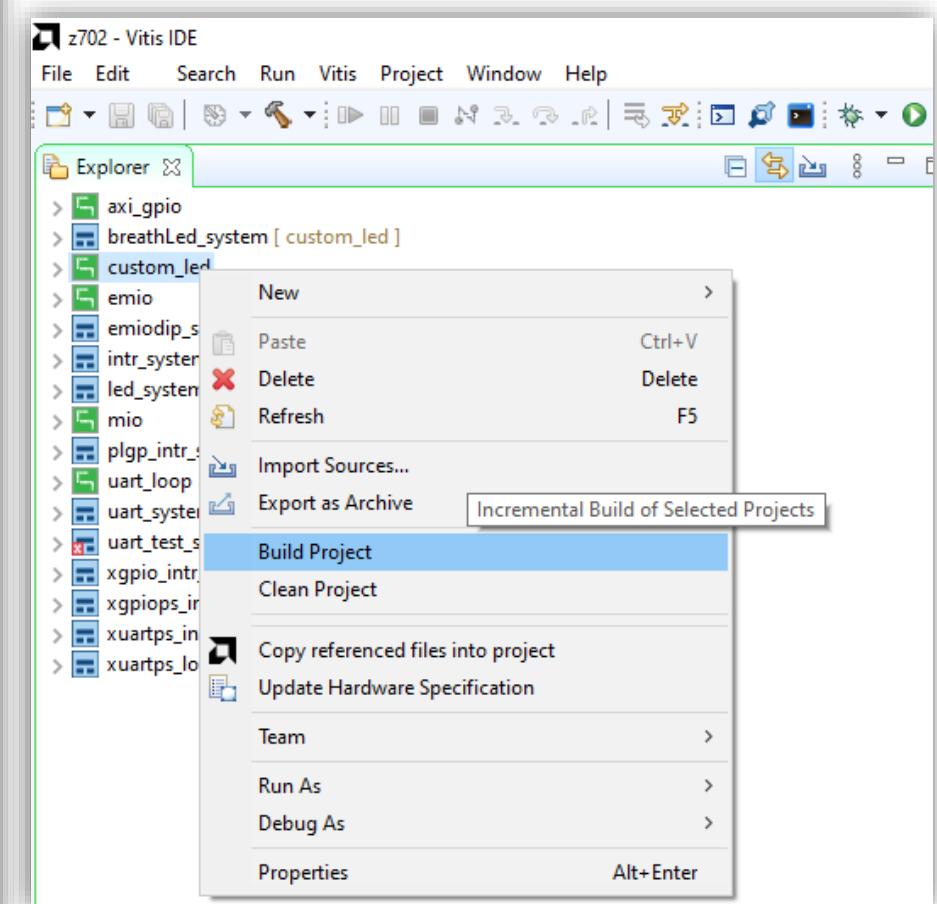
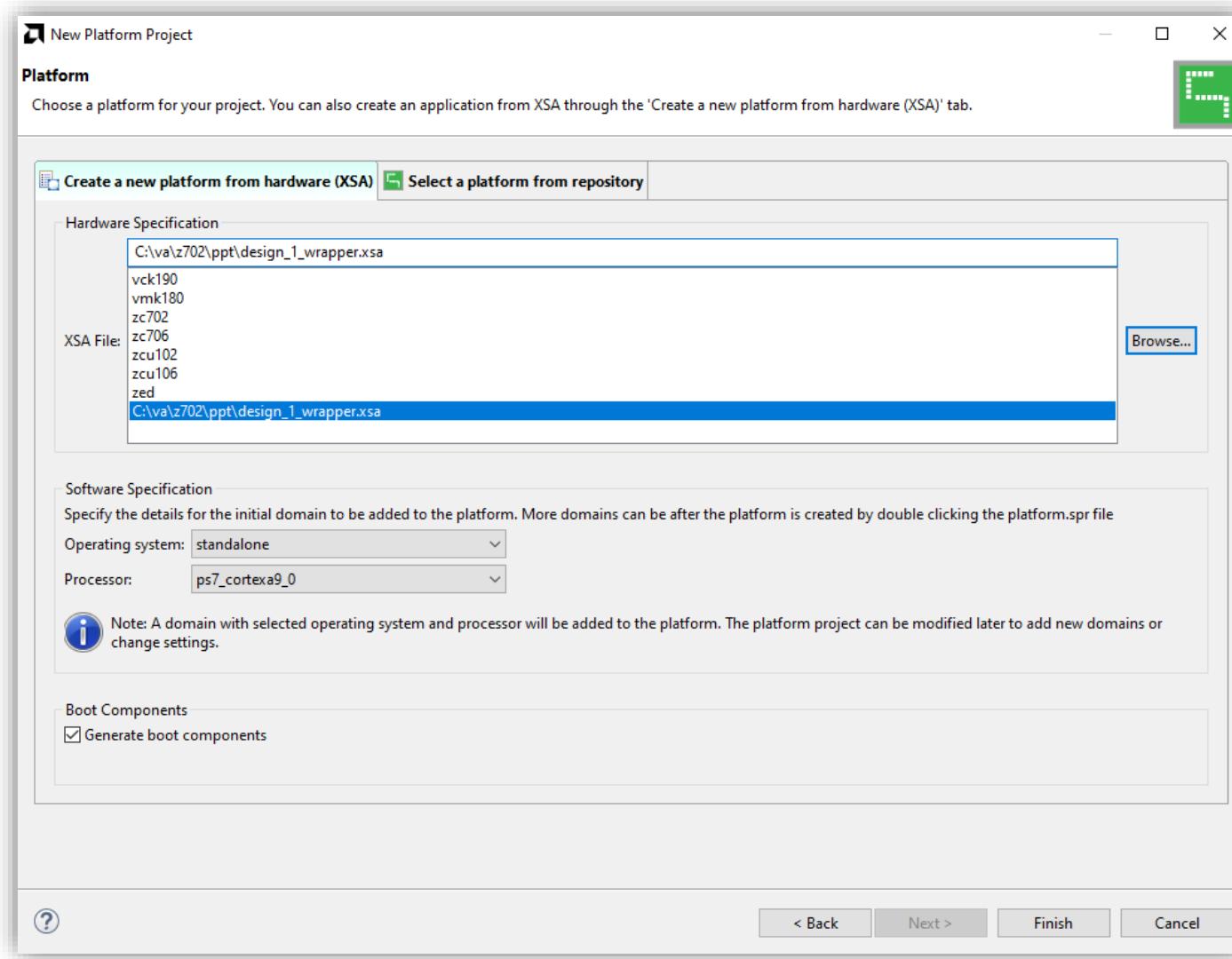
If there are uses of resources on the Programmable Logic (PL) side, you need to generate the bitstream to output the XSA file,

On the other hand, also ensure that the constraints for the Programmable Logic (PL) side are correct.



Build the Platform

Establish a virtual platform in Vitis based on the hardware information in the XSA file.



Build the Application

Sequentially select the platform and CPU for the application, choose the working domain, and finally, select the example APP to apply.

Platform

Choose a platform for your project. You can also create an application from XSA through the 'Create a new platform from hardware (XSA)' tab.

Name	Board	Flow	Vendor	Path
axi_gpio [custom]	zc702	Embedded SW Dev	xilinx	C:\vi\z702\axi_gpio\export\axi_gpio
custom_led [custom]	zc702	Embedded SW Dev	xilinx	C:\vi\z702\custom_led\export\custom_led
emio [custom]	zc702	Embedded SW Dev	xilinx	C:\vi\z702\emio\export\emio
mio [custom]	zc702	Embedded SW Dev	xilinx	C:\vi\z702\mio\export\mio
uart_loop [custom]	zc702	Embedded SW Dev	xilinx	C:\vi\z702\uart_loop\export\uart_loop
vilino_vck100_hardware_202201	vcl	Embedded Accel	xilinx.com	C:\Vilino\Vilino_2022_01\hardware

Platform Info

General Info

- Name: custom_led
- Part: xc7z020clg484-1
- Family: zynq

Acceleration Resources

The selected platform does not have application acceleration capabilities

Domain Details

Domains

- Domain name: standalone on ps7_cortexa9_0

Application Project Details

Specify the application project name and its system project properties

Application project name: led_test

System Project

Create a new system project for the application or select an existing one from the workspace

Select a system project: breathLed_system

+ Create new...

System project details

System project name: led_test_system

Target processor

Select target processor for the Application project.

Processor	Associated applications
ps7_cortexa9_0	led_test

Show all processors in the hardware specification

[? < Back](#) [Next >](#) [Finish](#) [Cancel](#)

Build the Application

Sequentially select the platform and CPU for the application, choose the working domain, and finally, select the example APP to apply.

The image shows two side-by-side screenshots of a software application's "New Application Project" wizard.

Left Screenshot (Domain Step):

- Title:** New Application Project
- Section:** Domain
- Text:** Select a domain for your project or create a new domain
- Sub-section:** Select the domain that the application would link to or create a new domain
- Note:** New domain created by this wizard will have all the requirements of the application template selected in the next step.
- UI Elements:**
 - A list box labeled "Select a domain" containing "standalone on ps7_cortexa9_0".
 - A button labeled "+ Create new...".
 - A "Domain details" section with fields:
 - Name: standalone_domain
 - Display Name: standalone on ps7_cortexa9_0
 - Operating System: standalone
 - Processor: ps7_cortexa9_0
- Buttons:** ? < Back Next > Finish

Right Screenshot (Templates Step):

- Title:** New Application Project
- Section:** Templates
- Text:** Select a template to create your project.
- Available Templates:**
 - Find: [Search bar]
 - Embedded software development templates
 - Dhrystone
 - Empty Application (C++)
 - Empty Application(C)
 - Hello World** (highlighted)
 - IwlIP Echo Server
 - IwlIP TCP Perf Client
 - IwlIP TCP Perf Server
 - IwlIP UDP Perf Client
 - IwlIP UDP Perf Server
 - Memory Tests
 - OpenAMP echo-test
 - OpenAMP matrix multiplication Demo
 - OpenAMP RPC Demo
 - Peripheral Tests
 - RSA Authentication App
 - Zynq DRAM tests
 - Zynq FSBL
- Sub-section:** Hello World
- Description:** Let's say 'Hello World' in C.
- Buttons:** ? < Back Next > Finish Cancel

Build the Application

We can learn through the documents and examples available in the BSP directory.

For ease of explanation, we will demonstrate the control of the three types of GPIO separately.

Board Support Package

View current BSP settings, or configure settings like STUDIO peripheral selection, compiler flags, SW intrusive profiling, add/remove libraries, assign drivers to peripherals, change versions of OS/libraries/drivers etc.

[Modify BSP Settings...](#) [Reset BSP Sources](#)

A BSP settings file is generated with the user options selected in the settings dialog. To use existing settings, click the below link. This operation clears any existing modifications done. All the subsequent changes are applied on top of the loaded settings.

[Load BSP settings from file](#)

Operating System

Name: standalone
Version: 8.0
Description: Standalone is a simple, low-level software layer. It provides access to basic processor features such as caches, interrupts and exceptions as well as the basic features of a hosted environment, such as standard input and output, profiling, abort and exit.
Documentation: -

Drivers **Libraries**

Name	Driver	Documentation	Examples
ps7_ddr_0	ddrps	Documentation Link	-
ps7_ddrc_0	generic	-	-
ps7_dev_cfg_0	devcfg	Documentation Link	Import Examples
ps7_dma_ns	dmaps	Documentation Link	Import Examples
ps7_dma_s	dmaps	Documentation Link	Import Examples
ps7_ethernet_0	emacps	Documentation Link	Import Examples
ps7_globaltimer_0	generic	-	-
ps7_gpio_0	gpio_ps	Documentation Link	Import Examples
ps7_gpv_0	generic	-	-
ps7_i2c_0	iicps	Documentation Link	Import Examples

MIO_GPIO(Write)

1. Searching for configurations (define ID)
different devices may have more than one set of GPIOs.
2. Initialize (declare & GpioPs structure)
GpioPs structure based on the declared number.
3. Set direction
in this structure, set pin 10 as output, 1 for output, 0 for input
4. Enable
you can choose to enable a specific Gpio Pin or entire Bank
5. Write function
in this structure, write a value to pin 10

```

1 #include <stdio.h>
2 #include "platform.h"
3 #include "xil_printf.h"
4 #include "xgpiops.h"
5 #include "xparameters.h"
6 #include "xbasic_types.h"
7 #include "sleep.h"
8 #define GPIO_DEVICE_ID      XPAR_XGPIOPS_0_DEVICE_ID
9
10
11 XGpioPs Gpio;
12 XGpioPs_Config *ConfigPtr;
13
14 int main()
15 {
16     ConfigPtr = XGpioPs_LookupConfig(GPIO_DEVICE_ID);
17
18     XGpioPs_CfgInitialize(&Gpio, ConfigPtr , ConfigPtr->BaseAddr);
19
20     XGpioPs_SetDirectionPin(&Gpio, 10, 1);
21
22     XGpioPs_SetOutputEnablePin(&Gpio, 10, 1);|
23
24     while(1){
25         XGpioPs_WritePin(&Gpio, 10, 1);
26
27         sleep(1);
28
29         XGpioPs_WritePin(&Gpio, 10, 0);
30
31         sleep(1);
32     }
33     return 0;
34 }
```

MIO_GPIO(Write)

```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xgpiops.h"
#include "xparameters.h"
#include "xbasic_types.h"
#include "sleep.h"

#define GPIO_DEVICE_ID XPAR_XGPIOPS_0_DEVICE_ID

XGpioPs Gpio;
XGpioPs_Config *ConfigPtr;
```

MIO_GPIO(Write)

```
int main()
{
    ConfigPtr = XGpioPs_LookupConfig(GPIO_DEVICE_ID);

    XGpioPs_CfgInitialize(&Gpio, ConfigPtr, ConfigPtr->BaseAddr);

    XGpioPs_SetDirectionPin(&Gpio, 10 , 1);

    XGpioPs_SetOutputEnablePin(&Gpio, 10 , 1);

    while(1) {
        XGpioPs_WritePin(&Gpio, 10 , 1);

        sleep(1);

        XGpioPs_WritePin(&Gpio, 10 , 0);

        sleep(1);
    }
    return 0;
}
```

MIO_GPIO(Read)

1. Searching for configurations (define ID)
different devices may have more than one set of GPIOs.

2. Initialize (declare & GpioPs structure)
GpioPs structure based on the declared number.

3. Set direction
in this structure, set pin 12 as input, 1 for output, 0 for input

4. Read function
in this structure, Read a value to pin 12

GPIO configuration and initialization only need to be done once.

Reading does not require enable, but when reading,
it is necessary to use another variable to store the value.

```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xgpiops.h"
#include "xparameters.h"
#include "xbasic_types.h"
#include "sleep.h"
#define GPIO_DEVICE_ID      XPAR_XGPIOPS_0_DEVICE_ID

XGpioPs Gpio;
XGpioPs_Config *ConfigPtr;

int main()
{
    u32 data;

    ConfigPtr = XGpioPs_LookupConfig(GPIO_DEVICE_ID);
    XGpioPs_CfgInitialize(&Gpio, ConfigPtr, ConfigPtr->BaseAddr);
    XGpioPs_SetDirectionPin(&Gpio, 10, 1);
    XGpioPs_SetDirectionPin(&Gpio, 12, 0);

    XGpioPs_SetOutputEnablePin(&Gpio, 10, 1);

    while(1){
        data = XGpioPs_ReadPin(&Gpio, 12);

        XGpioPs_WritePin(&Gpio, 10, data);
    }
    return 0;
}
```

MIO_GPIO(Read)

```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xgpiops.h"
#include "xparameters.h"
#include "xbasic_types.h"
#include "sleep.h"

#define GPIO_DEVICE_ID XPAR_XGPIOPS_0_DEVICE_ID

XGpioPs Gpio;
XGpioPs_Config *ConfigPtr;
```

MIO_GPIO(Read)

```
int main()
{
    u32 data ;

    ConfigPtr = XGpioPs_LookupConfig(GPIO_DEVICE_ID);

    XGpioPs_CfgInitialize(&Gpio, ConfigPtr, ConfigPtr->BaseAddr);

    XGpioPs_SetDirectionPin(&Gpio, 10 , 1);
    XGpioPs_SetDirectionPin(&Gpio, 12 , 0);

    XGpioPs_SetOutputEnablePin(&Gpio, 10 , 1);

    while(1) {
        data = XGpioPs_ReadPin(&Gpio, 12);

        XGpioPs_WritePin(&Gpio, 10 , data);
    }
    return 0;
}
```

EMIO_GPIO

Since EMIO is considered an extension of MIO,
the first EMIO pin is labeled as pin 54
(MIO pins are numbered from 0 to 53).

```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xgpiops.h"
#include "xparameters.h"
#include "xbasic_types.h"
#include "sleep.h"
#define GPIO_DEVICE_ID      XPAR_XGPIOPS_0_DEVICE_ID

XGpioPs Gpio;
XGpioPs_Config *ConfigPtr;

int main()
{
    u32 data;

    printf("Test");

    ConfigPtr = XGpioPs_LookupConfig(GPIO_DEVICE_ID);
    XGpioPs_CfgInitialize(&Gpio, ConfigPtr , ConfigPtr->BaseAddr);

    XGpioPs_SetDirectionPin(&Gpio, 10, 1);
    XGpioPs_SetDirectionPin(&Gpio, 54, 0);

    XGpioPs_SetOutputEnablePin(&Gpio, 10, 1);

    while(1){
        data = XGpioPs_ReadPin(&Gpio, 54);
        XGpioPs_WritePin(&Gpio, 10, data);
    }
    return 0;
}
```

Block Diagram

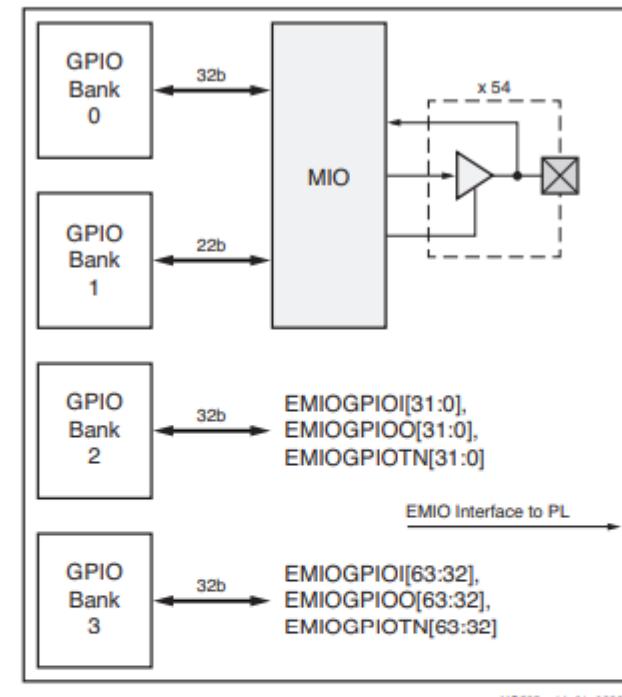


Figure 14-1: GPIO Block Diagram

As shown in [Figure 14-1](#), the GPIO module is divided into four banks:

- Bank0: 32-bit bank controlling MIO pins[31:0]
- Bank1: 22-bit bank controlling MIO pins[53:32]

Note: Bank1 is limited to 22 bits because the MIO has a total of 54 pins.
- Bank2: 32-bit bank controlling EMIO signals[31:0]
- Bank3: 32-bit bank controlling EMIO signals[63:32]

AXI_GPIO

Unlike MIO, which uses `xgpiops.h`,
AXI GPIO requires the use of `xgpio.h`.

Additionally, the default AXI GPIO ID is the same as PS GPIO.

Therefore, we need to change AXI GPIO ID name.

Since we have connected only one set of AXI GPIO,
set the channel to 1 here.

The structure should also be present, with the difference that
AXI GPIO configuration does not need to be declared in
advance (it is determined by the PL side).

```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xgpiops.h"
#include "xgpio.h"
#include "xparameters.h"
#include "xbasic_types.h"
#include "xscugic.h"
#include "sleep.h"

#define GPIO_DEVICE_ID XPAR_XGPIOPS_0_DEVICE_ID //PS GPIO ID
#define AXI_GPIO_ID XPAR_GPIO_0_DEVICE_ID //AXI GPIO ID
#define GPIO_CHANNEL1 1 //FIRST Gpio channel

XGpioPs Gpio;
XGpioPs_Config *ConfigPtr;

XGpio AXI_Gpio;

int main()
{
    u32 data = 0;

    printf("AXI_GPIO Test");

    // PS gpio Config and init
    ConfigPtr = XGpioPs_LookupConfig(GPIO_DEVICE_ID);
    XGpioPs_CfgInitialize(&Gpio, ConfigPtr, ConfigPtr->BaseAddr);

    // Include AXI_GPIO Config and init
    XGpio_Initialize(&AXI_Gpio, AXI_GPIO_ID);

    // PS Enable and Set direction
    XGpioPs_SetDirectionPin(&Gpio, 10, 1);
    XGpioPs_SetOutputEnablePin(&Gpio, 10, 1);

    XGpio_SetDataDirection(&AXI_Gpio, GPIO_CHANNEL1, 0x00000001);

    while(1){
        if(XGpio_DiscreteRead(&AXI_Gpio,GPIO_CHANNEL1) == 0)
            data = ~data;
        XGpioPs_WritePin(&Gpio, 10, data);
    }
    return 0;
}
```

MIO_GPIO(AXI_GPIO)

```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xgpiops.h"
#include "xgpio.h"
#include "xparameters.h"
#include "xbasic_types.h"
#include "xscugic.h"
#include "sleep.h"

#define GPIO_DEVICE_ID XPAR_XGPIOPS_0_DEVICE_ID
#define AXI_GPIO_ID XPAR_GPIO_0_DEVICE_ID
#define GPIO_CHANNEL11

XGpioPs Gpio;
XGpioPs_Config *ConfigPtr;

XGpio AXI_Gpio;
```

MIO_GPIO(AXI_GPIO)

```
int main()
{
    u32 data = 0;
    printf("AXI_GPIO Test");

    ConfigPtr = XGpioPs_LookupConfig(GPIO_DEVICE_ID);
    XGpioPs_CfgInitialize(&Gpio, ConfigPtr, ConfigPtr->BaseAddr);

    XGpio_Initialize(&AXI_Gpio, AXI_GPIO__ID);

    XGpioPs_SetDirectionPin(&Gpio, 10 , 1);
    XGpioPs_SetOutputEnablePin(&Gpio, 10 , 1);

    XGpio_SetDataDirection(&AXI_Gpio, GPIO_CHANNEL1, 0x00000001);

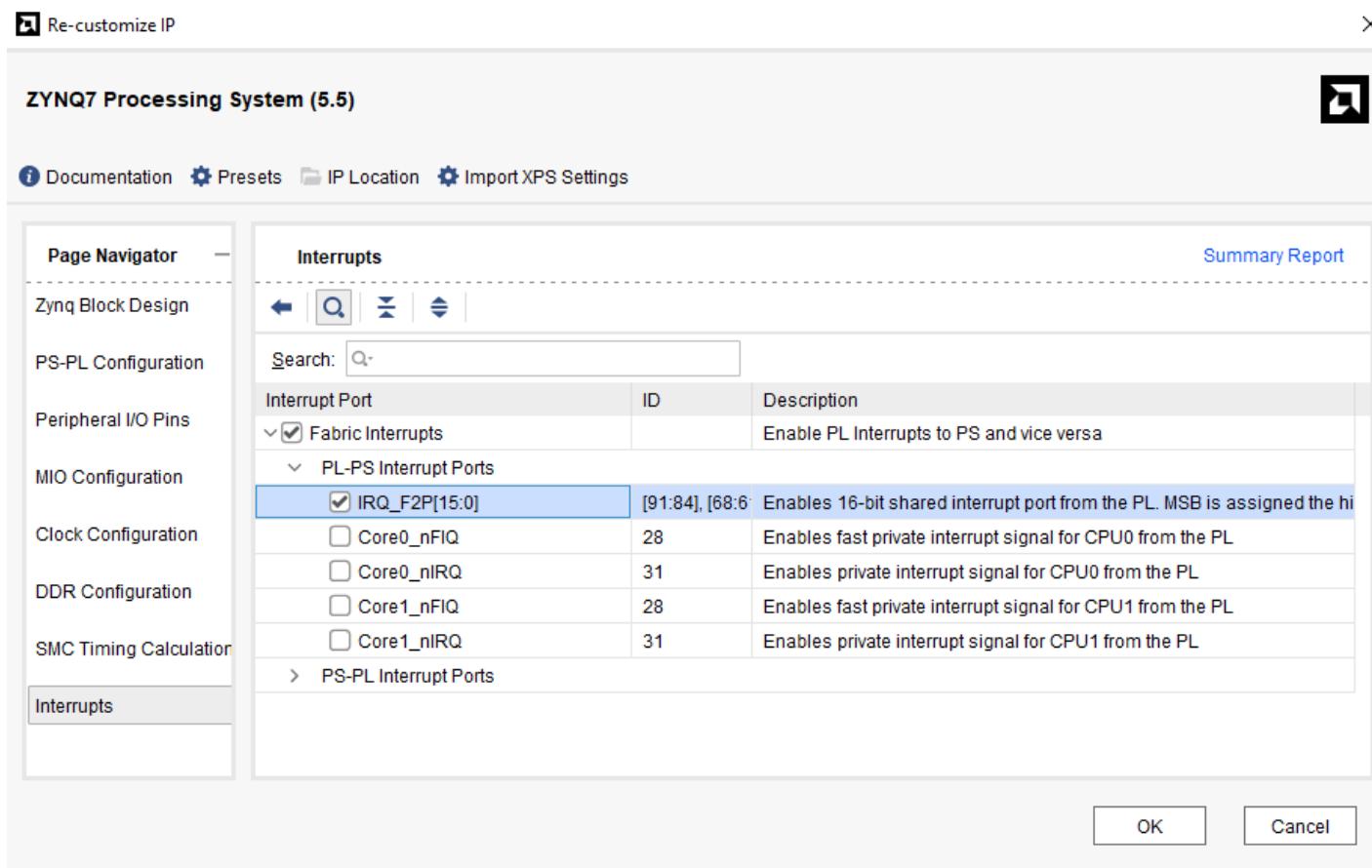
    while(1) {
        if(XGpio_DiscreteRead(&AXI_Gpio, GPIO_CHANNEL1) == 0);
        data = ~data;
        XGpioPs_WritePin(&Gpio, 10 , data);
    }
    return 0;
}
```

PS or PL Interrupt

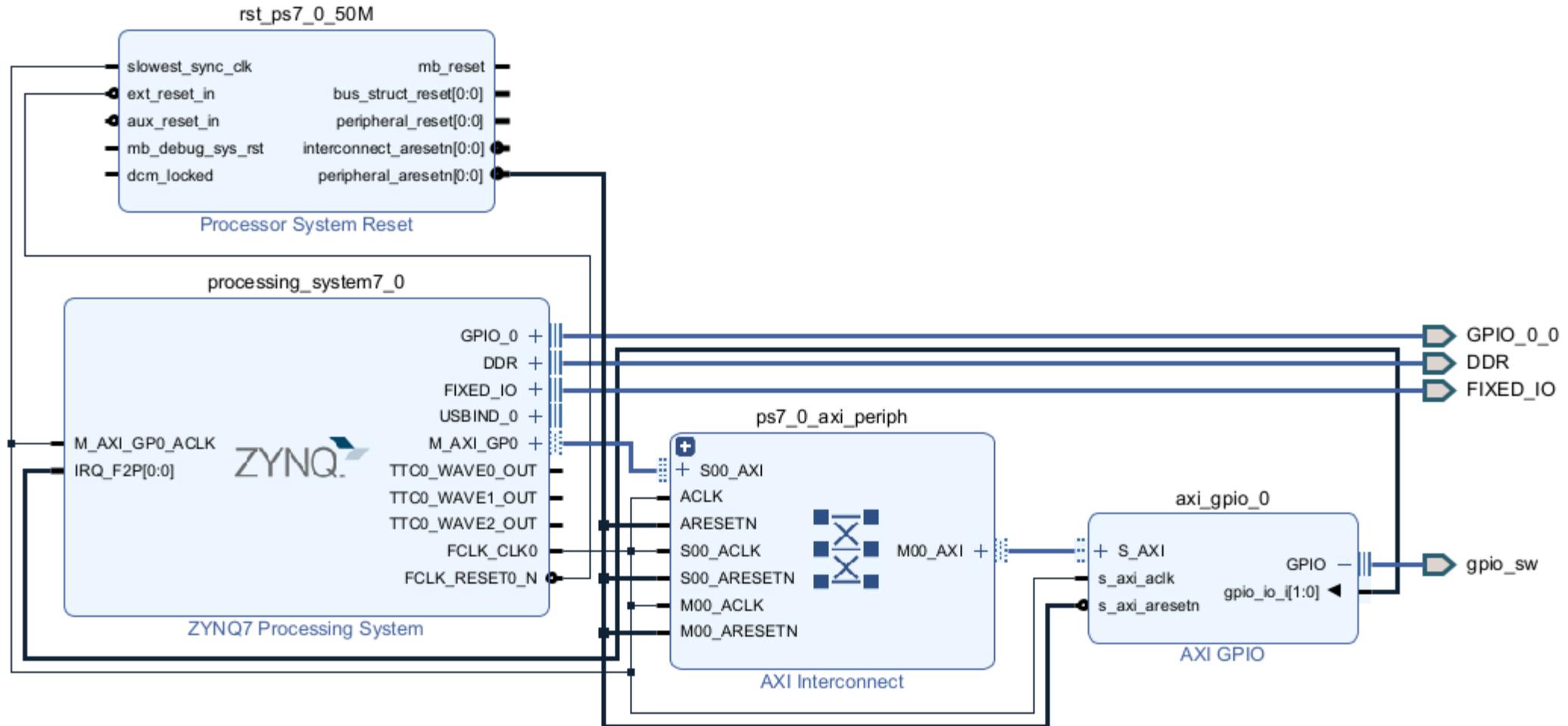
Block Design

For this Lab, you need to add an interrupt triggered by the PL side.

So, in addition to enabling the interrupt port for AXI GPIO, you also need to go to the Zynq settings, enable IRQ, and make the connection.



Connect IRQ Manually



PS Intr_include&define

1. When using GPIO on the PS side and utilizing the GIC,
it is necessary to **configure and instantiate** both the GPIO and GIC.

2. Additionally, besides declaring their respective IDs,
there needs to be an ID match for the **connection**.

3. In the IntrHandler, we determine the actions to be taken
when the interrupt occurs,
and this is where we need to write our own control logic.

```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xgpiops.h"
#include "xparameters.h"
#include "xbasic_types.h"
#include "xscugic.h"
#include "sleep.h"

#define GPIO_DEVICE_ID      XPAR_XGPIOPS_0_DEVICE_ID
#define GPIO_INTERRUPT_ID   XPAR_XGPIOPS_0_INTR
#define INTC_DEVICE_ID       XPAR_SCUGIC_SINGLE_DEVICE_ID

XGpioPs Gpio;
XGpioPs_Config *ConfigPtr;

XScuGic Intc;
XScuGic_Config *IntcConfig;

void SetupInterruptSystem(XScuGic *Intc, XGpioPs *Gpio, u16 GpioIntrId);
void IntrHandler();

u32 key = 0 ;
```

Intr_system

- 1.No modifications are needed in the first half, we only need to adjust the connected sections.
- 2.Next is choosing the enabled interrupt type and actually enabling the interrupt functionality for a specific pin.

```

void SetupInterruptSystem(XScuGic *GicInstancePtr, XGpioPs *Gpio,
                         u16 GpioIntrId)
{
    XScuGic_Config *IntcConfig;

    IntcConfig = XScuGic_LookupConfig(INTC_DEVICE_ID);
    XScuGic_CfgInitialize(GicInstancePtr, IntcConfig,
                          IntcConfig->CpuBaseAddress);

    Xil_ExceptionInit();
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
                                (Xil_ExceptionHandler)XScuGic_InterruptHandler,
                                GicInstancePtr);
    Xil_ExceptionEnableMask(XIL_EXCEPTION_IRQ);

    XScuGic_Connect(GicInstancePtr, GpioIntrId,
                    (Xil_ExceptionHandler)IntrHandler,
                    (void *)Gpio);

    XScuGic_Enable(GicInstancePtr, GpioIntrId);

    // XGpioPs_SetCallbackHandler(Gpio, (void *)Gpio, IntrHandler);

    // XGpioPs_SetIntrType(Gpio, GPIO_BANK, 0x00, 0xFFFFFFFF, 0x00);
    XGpioPs_SetIntrTypePin(Gpio, 54 , XGPIOPS_IRQ_TYPE_EDGE_FALLING);

    // XGpioPs_IntrEnable(Gpio, GPIO_BANK, (1 << Input_Bank_Pin));
    XGpioPs_IntrEnablePin(Gpio, 54);

}

```

IntrHandler & Main

The handler sets a flag and temporarily disables the interrupt functionality.

```
void IntrHandler(){
    printf("Detect");
    key = 1;
    XGpioPs_IntrDisablePin(&Gpio, 54);
}
```

In the main function, besides clearing the original interrupt register status, remember to re-enable it for a repeated loop.

```
int main()
{
    u32 data = 0;

    printf("Test");

    ConfigPtr = XGpioPs_LookupConfig(GPIO_DEVICE_ID);

    XGpioPs_CfgInitialize(&Gpio, ConfigPtr , ConfigPtr->BaseAddr);

    XGpioPs_SetDirectionPin(&Gpio, 10, 1);
    XGpioPs_SetDirectionPin(&Gpio, 54, 0);

    XGpioPs_SetOutputEnablePin(&Gpio, 10, 1);

    SetupInterruptSystem(&Intc, &Gpio, GPIO_INTERRUPT_ID);

    while(1){
        if(key){
            data = ~data;

            key = 0;

            XGpioPs_IntrClearPin(&Gpio,54);

            XGpioPs_WritePin(&Gpio, 10, data);

            usleep(200000);

            XGpioPs_IntrEnablePin(&Gpio, 54);
        }
    }
    return 0;
}
```

PS_intr

```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xgpiops.h"
#include "xparameters.h"
#include "xbasic_types.h"
#include "xscugic.h"
#include "sleep.h"

#define GPIO_DEVICE_ID XPAR_XGPIOPS_0_DEVICE_ID
#define GPIO_INTERRUPT_ID XPAR_XGPIOPS_0_INTR
#define INTC_DEVICE_ID XPAR_SCUGIC_SINGLE_DEVICE_ID

XGpioPs Gpio;
XGpioPs_Config *ConfigPtr;

XScuGic Intc;
XScuGic_Config *IntcConfig;

void SetupInterruptSystem(XScuGic *Intc, XGpioPs *Gpio, u16 GpioIntrId);
void IntrHandler();

u32 key = 0 ;
```

PS_Intr

```
int main() {
    u32 data = 0;
    printf("Test");

    ConfigPtr = XGpioPs_LookupConfig(GPIO_DEVICE_ID);
    XGpioPs_CfgInitialize(&Gpio, ConfigPtr, ConfigPtr->BaseAddr);
    XGpioPs_SetDirectionPin(&Gpio, 10, 1);
    XGpioPs_SetDirectionPin(&Gpio, 54, 0);
    XGpioPs_SetOutputEnablePin(&Gpio, 10, 1);

    SetupInterruptSystem(&Intc, &Gpio, GPIO_INTERRUPT_ID);
    while(1) {
        if(key) {
            data = ~data;
            key = 0;
            XGpioPs_IntrClearPin(&Gpio, 54);
            XGpioPs_WritePin(&Gpio, 10, data);
            usleep(200000);
            XGpioPs_IntrEnablePin(&Gpio, 54);
        }
    }
    return 0;
}
```

PS_Intr

```

void SetupInterruptSystem(XScuGic *GicInstancePtr, XGpioPs *Gpio, u16 GpioIntrId) {
    XScuGic_Config *IntcConfig;
    IntcConfig = XScuGic_LookupConfig(INTC_DEVICE_ID);
    XScuGic_CfgInitialize(GicInstancePtr, IntcConfig, IntcConfig->CpuBaseAddress);

    Xil_ExceptionInit();
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
        (Xil_ExceptionHandler)XScuGic_InterruptHandler, GicInstancePtr);
    Xil_ExceptionEnableMask(XIL_EXCEPTION_IRQ);

    XScuGic_Connect(GicInstancePtr, GpioIntrId, (Xil_ExceptionHandler)IntrHandler, (void *)Gpio);
    XScuGic_Enable(GicInstancePtr, GpioIntrId);

    //XGpioPs_SetCallbackHandler(Gpio, (void *)Gpio, IntrHandler);
    //XGpioPs_SetIntrType(Gpio, GPIO_BANK, 0x00, 0xFFFFFFFF, 0x00);
    XGpioPs_SetIntrTypePin(Gpio, 54, XGPIOPS_IRQ_TYPE_EDGE_FALLING);

    //XGpioPs_IntrEnable(Gpio, GPIO_BANK, (1 << Input_Bank_Pin));
    XGpioPs_IntrEnablePin(Gpio, 54);
}

void IntrHandler() {
    printf("Detect");
    key = 1;
    XGpioPs_IntrDisablePin(&Gpio, 54);
}

```

PL Intr_include&define

PL interrupts are connected to the GIC.

According to the manual, the first PL interrupt is at bit 61.

Table 7-4: PS and PL Shared Peripheral Interrupts (SPI) (Cont'd)

Source	Interrupt Name	IRQ ID#	Status Bits (mpcore Registers)	Required Type	PS-PL Signal Name	I/O
IOP	GPIO	52	spi_status_0[20]	High level	IRQP2F[16]	Output
	USB 0	53	spi_status_0[21]	High level	IRQP2F[15]	Output
	Ethernet 0	54	spi_status_0[22]	High level	IRQP2F[14]	Output
	Ethernet 0 Wake-up	55	spi_status_0[23]	Rising edge	IRQP2F[13]	Output
	SDIO 0	56	spi_status_0[24]	High level	IRQP2F[12]	Output
	I2C 0	57	spi_status_0[25]	High level	IRQP2F[11]	Output
	SPI 0	58	spi_status_0[26]	High level	IRQP2F[10]	Output
	UART 0	59	spi_status_0[27]	High level	IRQP2F[9]	Output
	CAN 0	60	spi_status_0[28]	High level	IRQP2F[8]	Output
PL	PL [2:0]	63:61	spi_status_0[31:29]	Rising edge/ High level	IRQF2P[2:0]	Input
	PL [7:3]	68:64	spi_status_1[4:0]	Rising edge/ High level	IRQF2P[7:3]	Input

```

370 /* **** */
371
372 /* Definitions for Fabric interrupts connected to ps7_scugic_0 */
373 #define XPAR_FABRIC_AXI_GPIO_0_IP2INTC_IRPT_INTR 61U
374
375 /* **** */
376

```

PL Intr_include&define

Essentially, it is similar to the PS setup, with the difference being that we are enabling PL AXI-related pins.

```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xgpiops.h"
#include "xgpio.h"
#include "xparameters.h"
#include "xbasic_types.h"
#include "xscugic.h"
#include "sleep.h"

#define GPIO_DEVICE_ID      XPAR_XGPIOPS_0_DEVICE_ID //PS  GPIO ID
#define AXI_GPIO_ID          XPAR_GPIO_0_DEVICE_ID    //AXI GPIO ID
#define INTc_DEVICE_ID        XPAR_SCUGIC_SINGLE_DEVICE_ID

#define AXI_GPIO_INTERRUPT_ID XPAR_FABRIC_AXI_GPIO_0_IP2INTC_IRPT_INTR

#define GPIO_CHANNEL1         1 //FIRST Gpio channel

XGpioPs Gpio;
XGpioPs_Config *ConfigPtr;

XScuGic Intc;
XScuGic_Config *IntcConfig;

XGpio AXI_Gpio;

void SetupInterruptSystem(XScuGic *GicInstancePtr, XGpio *AXI_Gpio,
                         u16 AXI_GpioIntrId);
void IntrHandler();

u32 key = 0 ;

int main()
```

Intr_system

- 1.Initially, we only needed to modify the enabled interrupt types. Now, we are using **priority** to determine the interrupt priority between PS and PL.
- 2.The last parameter in the priority function specifies the interrupt type, which **0x01** indicates a high-level trigger.
- 3.Because this interrupt affects both the PS and PL sides, both the **global** and the **individual interrupt** need to be enabled.
- 4.Additionally, the mask determines which bit in the 32-bit word is to enable the interrupt.

```

void SetupInterruptSystem(XScuGic *GicInstancePtr, XGpio *AXI_Gpio,
                        u16 AXI_GpioIntrId)
{
    XScuGic_Config *IntcConfig;

    IntcConfig = XScuGic_LookupConfig(INTC_DEVICE_ID);
    XScuGic_CfgInitialize(GicInstancePtr, IntcConfig,
                          IntcConfig->CpuBaseAddress);

    Xil_ExceptionInit();
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
                                (Xil_ExceptionHandler)XScuGic_InterruptHandler,
                                GicInstancePtr);
    Xil_ExceptionEnableMask(XIL_EXCEPTION_IRQ);

    XScuGic_Connect(GicInstancePtr, AXI_GpioIntrId,
                    (Xil_ExceptionHandler)IntrHandler,
                    (void *)AXI_Gpio);

    XScuGic_Enable(GicInstancePtr, AXI_GpioIntrId);

    // Priority A0 = Highest 0x1 = High Activity
    XScuGic_SetPriorityTriggerType(GicInstancePtr, AXI_GpioIntrId,
                                   0xA0, 0x1);

    //Enable AXI_GPIO IP Intr
    XGpio InterruptGlobalEnable(AXI_Gpio);
    XGpio InterruptEnable(AXI_Gpio, 0x00000001);
}

void IntrHandler(){
    printf("Detect");
    key = 1;
    XGpio InterruptDisable(&AXI_Gpio, 0x00000001);
}

```

Main

Apart from the name, the rest of the sections have remained almost unchanged..

```
int main()
{
    u32 data = 0;

    printf("AXI_GPIO Test");

    // PS gpio Config and init
    ConfigPtr = XGpioPs_LookupConfig(GPIO_DEVICE_ID);
    XGpioPs_CfgInitialize(&Gpio, ConfigPtr , ConfigPtr->BaseAddr);

    // Include AXI_GPIO Config and init
    XGpio_Initialize(&AXI_Gpio, AXI_GPIO__ID);

    // PS Enable and Set direction
    XGpioPs_SetDirectionPin(&Gpio, 10, 1);
    XGpioPs_SetOutputEnablePin(&Gpio, 10, 1);

    XGpio_SetDataDirection(&AXI_Gpio, GPIO_CHANNEL1,0x00000001);

    SetupInterruptSystem(&Intc, &AXI_Gpio, AXI_GPIO_INTERRUPT_ID);

    while(1){
        if(key){
            if(XGpio_DiscreteRead(&AXI_Gpio,GPIO_CHANNEL1) == 0)
                data = ~data;

            key = 0;

            XGpio InterruptClear(&AXI_Gpio,0x00000001);

            XGpioPs_WritePin(&Gpio, 10, data);

            usleep(200000);

            XGpio_InterruptEnable(&AXI_Gpio, 0x00000001);
        }
    }
    return 0;
}
```

PL_intr

```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xgpiops.h"
#include "xgpio.h"
#include "xparameters.h"
#include "xbasic_types.h"
#include "xscugic.h"
#include "sleep.h"

#define GPIO_DEVICE_ID XPAR_XGPIOPS_0_DEVICE_ID
#define AXI_GPIO_ID XPAR_GPIO_0_DEVICE_ID
#define INTC_DEVICE_ID XPAR_SCUGIC_SINGLE_DEVICE_ID

#define AXI_GPIO_INTERRUPT_ID XPAR_FABRIC_AXI_GPIO_0_IP2INTC_IRPT_INTR
#define GPIO_CHANNEL

XGpioPs Gpio;
XGpioPs_Config *ConfigPtr;
XScuGic Intc;
XScuGic_Config *IntcConfig;
XGpio AXI_Gpio;

void SetupInterruptSystem(XScuGic *Intc, XGpioPs *Gpio, u16 GpioIntrId);
void IntrHandler();

u32 key = 0 ;
```

PL_intr

```

int main(){
    u32 data = 0;
    printf("AXI_GPIO Test");

    ConfigPtr = XGpioPs_LookupConfig(GPIO_DEVICE_ID);
    XGpioPs_CfgInitialize(&Gpio, ConfigPtr, ConfigPtr->BaseAddr);
    XGpio_Initialize(&AXI_Gpio, AXI_GPIO_ID);

    XGpioPs_SetDirectionPin(&Gpio, 10, 1);
    XGpioPs_SetOutputEnablePin(&Gpio, 10, 1);
    XGpio_SetDataDirection(&AXI_Gpio, GPIO_CHANNEL1, 0x00000001);
    SetupInterruptSystem(&Intc, &AXI_Gpio, AXI_GPIO_INTERRUPT_ID);

while(1){
    if(key){
        if(XGpio_DiscreteRead(&AXI_Gpio, GPIO_CHANNEL1) == 0)
            data = ~data;
        key = 0;

        XGpio_InterruptClear(&AXI_Gpio, 0x00000001);
        XGpioPs_WritePin(&Gpio, 10, data);
        usleep(200000);
        XGpio_InterruptEnable(&AXI_Gpio, 0x00000001);
    }
}
return 0;
}

```

PL_intr

```

void SetupInterruptSystem(XScuGic *GicInstancePtr, XGpioPs *AXI_Gpio, u16 AXI_GpioIntrId) {
    XScuGic_Config *IntcConfig;
    IntcConfig = XScuGic_LookupConfig(INTC_DEVICE_ID);
    XScuGic_CfgInitialize(GicInstancePtr, IntcConfig, IntcConfig->CpuBaseAddress);

    Xil_ExceptionInit();
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
        (Xil_ExceptionHandler)XScuGic_InterruptHandler, GicInstancePtr);
    Xil_ExceptionEnableMask(XIL_EXCEPTION_IRQ);

    XScuGic_Connect(GicInstancePtr, AXI_GpioIntrId,
        (Xil_ExceptionHandler)IntrHandler, (void *)AXI_Gpio);
    XScuGic_Enable(GicInstancePtr, AXI_GpioIntrId);
    XScuGic_SetPriorityTriggerType(GicInstancePtr, AXI_GpioIntrId, 0xA0, 0x1);

    XGpio InterruptEnable(AXI_Gpio, 0x00000001);
    XGpio InterruptGlobalEnable(AXI_Gpio);
}

void IntrHandler() {
    printf("Detect");
    key = 1;
    XGpio_IntrDisable(&AXI_Gpio, 0x00000001);
}

```

PL_AXI IP Control

AXI inst

```

`timescale 1 ns / 1 ps

module axiled_v1_0_S00_AXI #
(
    // Users to add parameters here
    parameter START_FREQ_STEP = 10'd100,
    // User parameters ends
    // Do not modify the parameters beyond this line

    // Width of S_AXI data bus
    parameter integer C_S_AXI_DATA_WIDTH      = 32,
    // Width of S_AXI address bus
    parameter integer C_S_AXI_ADDR_WIDTH       = 4
)
(
    // Users to add ports here
    output led,
    // User ports ends
    // Do not modify the ports beyond this line
)

```

```

        // Add user logic here
breath_led #(
    .START_FREQ_STEP(START_FREQ_STEP)
)
u_breath_led(
    .sys_clk          (S_AXI_ACLK),
    .sys_rst_n        (S_AXI_ARESETN),
    .sw_ctrl          (slv_reg0[0]),
    .set_en           (slv_reg1[31]),
    .set_freq_step    (slv_reg1[9:0]),
    .led               (led)
);
// User logic ends

endmodule

```

```

// Instantiation of Axi Bus Interface S00_AXI
axiled_v1_0_S00_AXI #((
    .C_S_AXI_DATA_WIDTH(C_S00_AXI_DATA_WIDTH),
    .C_S_AXI_ADDR_WIDTH(C_S00_AXI_ADDR_WIDTH)
) ) axiled_v1_0_S00_AXI_inst (
    .S_AXI_ACLK(s00_axi_aclk),
    .S_AXI_ARESETN(s00_axi_aresetn),
    .S_AXI_AWADDR(s00_axi_awaddr),
    .S_AXI_AWPROT(s00_axi_awprot),
    .S_AXI_AWVALID(s00_axi_awvalid),
    .S_AXI_AWREADY(s00_axi_awready),
    .S_AXI_WDATA(s00_axi_wdata),
    .S_AXI_WSTRB(s00_axi_wstrb),
    .S_AXI_WVALID(s00_axi_wvalid),
    .S_AXI_WREADY(s00_axi_wready),
    .S_AXI_BRESP(s00_axi_bresp),
    .S_AXI_BVALID(s00_axi_bvalid),
    .S_AXI_BREADY(s00_axi_bready),
    .S_AXI_ARADDR(s00_axi_araddr),
    .S_AXI_ARPROT(s00_axi_arprot),
    .S_AXI_ARVALID(s00_axi_arvalid),
    .S_AXI_ARREADY(s00_axi_arready),
    .S_AXI_RDATA(s00_axi_rdata),
    .S_AXI_RRESP(s00_axi_rresp),
    .S_AXI_RVALID(s00_axi_rvalid),
    .S_AXI_RREADY(s00_axi_rready)
);

```

AXI Wrappwe

```

`timescale 1 ns / 1 ps

module axiled_v1_0 #
(
    // Users to add parameters here
    parameter START_FREQ_STEP = 10'd100,
    // User parameters ends
    // Do not modify the parameters beyond this line
)

// Parameters of Axi Slave Bus Interface S00_AXI
parameter integer C_S00_AXI_DATA_WIDTH = 32,
parameter integer C_S00_AXI_ADDR_WIDTH = 4
)
(
    // Users to add ports here
    output led,
    // User ports ends
    // Do not modify the ports beyond this line
)

```

```

    "
    // Instantiation of Axi Bus Interface S00_AXI
    axiled_v1_0_S00_AXI #(
        .START_FREQ_STEP(START_FREQ_STEP),
        .C_S_AXI_DATA_WIDTH(C_S00_AXI_DATA_WIDTH),
        .C_S_AXI_ADDR_WIDTH(C_S00_AXI_ADDR_WIDTH)
    ) axiled_v1_0_S00_AXI_inst (
        .led(led),
        .S_AXI_ACLK(s00_axi_aclk),
        .S_AXI_ARESETN(s00_axi_aresetn),
        .S_AXI_AWADDR(s00_axi_awaddr),
        .S_AXI_AWPROT(s00_axi_awprot),
        .S_AXI_AWVALID(s00_axi_awvalid),
        .S_AXI_AWREADY(s00_axi_awready),
        .S_AXI_WDATA(s00_axi_wdata),
        .S_AXI_WSTRB(s00_axi_wstrb),
        .S_AXI_WVALID(s00_axi_wvalid),
        .S_AXI_WREADY(s00_axi_wready),
        .S_AXI_BRESP(s00_axi_bresp),
        .S_AXI_BVALID(s00_axi_bvalid),
        .S_AXI_BREADY(s00_axi_bready)
    );

```

Packager ip

Clicking on the XML inside the source will bring you back to this page.

The screenshot shows the Xilinx IP Packager interface with the following details:

Sources:

- Design Sources (2):
 - axiled_v1_0 (axiled_v1_0.v) (1)
 - axiled_v1_0_S00_AXI_inst: axiled_v1_0_S00_AXI (axiled_v1_0_S00_AXI.v) (1)
 - u_breath_led: breath_led (breath_led.v)- IP-XACT (1): component.xml
- Constraints
- Simulation Sources (1)
- Utility Sources

Hierarchy, Libraries, Compile Order: Buttons at the bottom of the Sources panel.

Source File Properties:

- component.xml
- Enabled:
- Location: c:/va/custom_ip/ip_repo/axiled_1_0
- Type: IP-XACT
- Size: 33.5 KB
- Modified: Today at 09:10:00 AM
- Read-only: No

General, Properties: Buttons at the bottom of the Source File Properties panel.

Project Summary: Package IP - axiled

Packaging Steps:

- Identification (selected)
- Compatibility
- File Groups
- Customization Parameters
- Ports and Interfaces
- Addressing and Memory
- Customization GUI
- Review and Package

Identification:

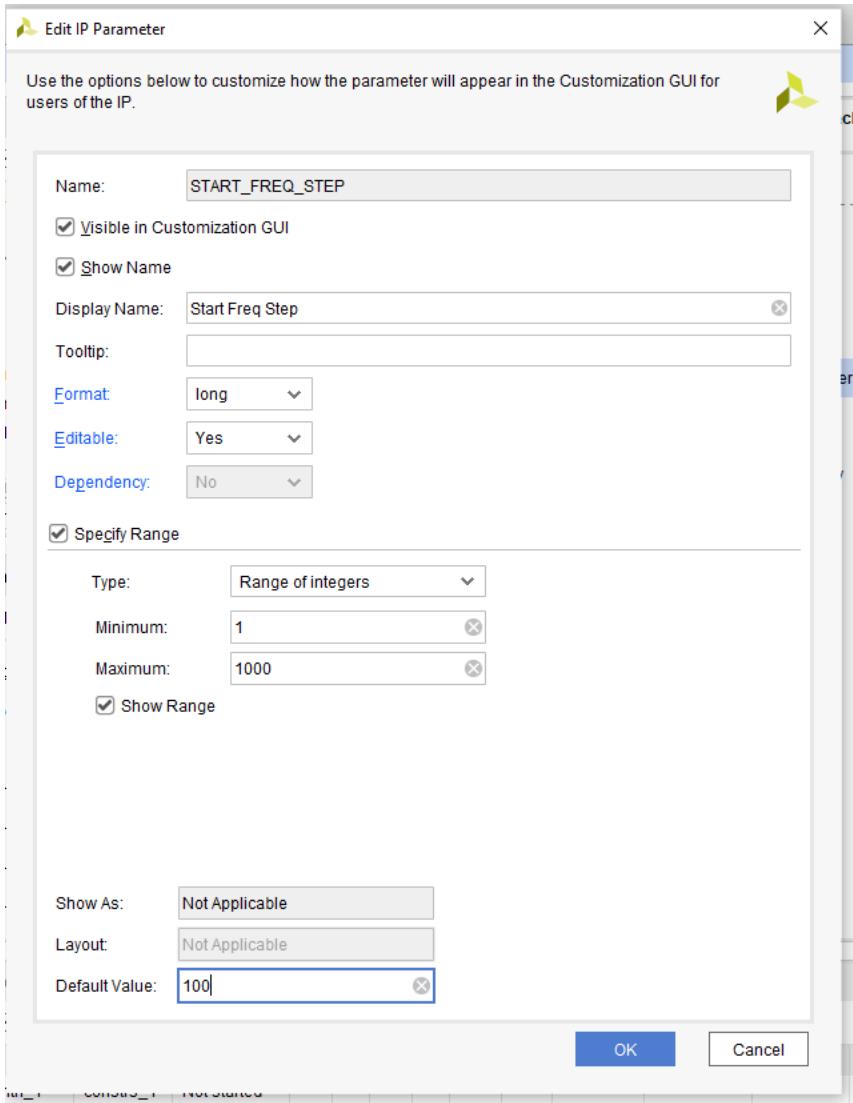
Vendor:	xilinx.com
Library:	user
Name:	axiled
Version:	1.0
Display name:	axiled_v1_0
Description:	axi_led_ip
Vendor display name:	
Company url:	
Root directory:	c:/va/custom_ip/ip_repo/axiled_1_0
Xml file name:	c:/va/custom_ip/ip_repo/axiled_1_0/component.xml

Categories:

- + AXI_Peripheral

Packager ip parameter

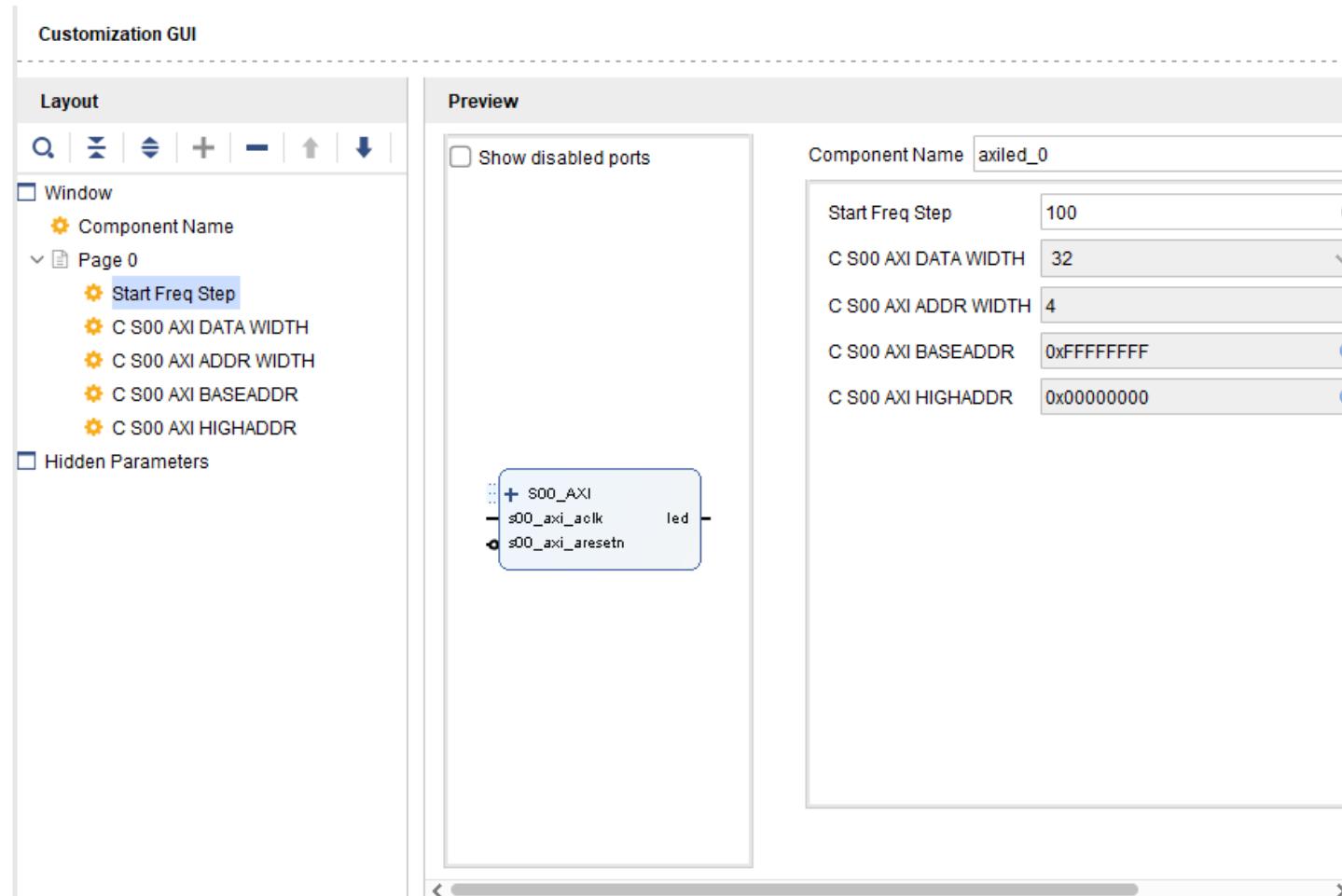
This will impact subsequent issues regarding function invocation in Vivado and Vitis viewing.



Name	Description	Display Name	Value
Customization Parameters			
C_S00_AXI_DATA_WIDTH	Width of S_AXI data bus	C S00 AXI DATA WIDTH	32
C_S00_AXI_ADDR_WIDTH	Width of S_AXI address bus	C S00 AXI ADDR WIDTH	4
C_S00_AXI_BASEADDR		C S00 AXI BASEADDR	0xFFFFFFFF
C_S00_AXI_HIGHADDR		C S00 AXI HIGHADDR	0x00000000
START_FREQ_STEP		Start Freq Step	100

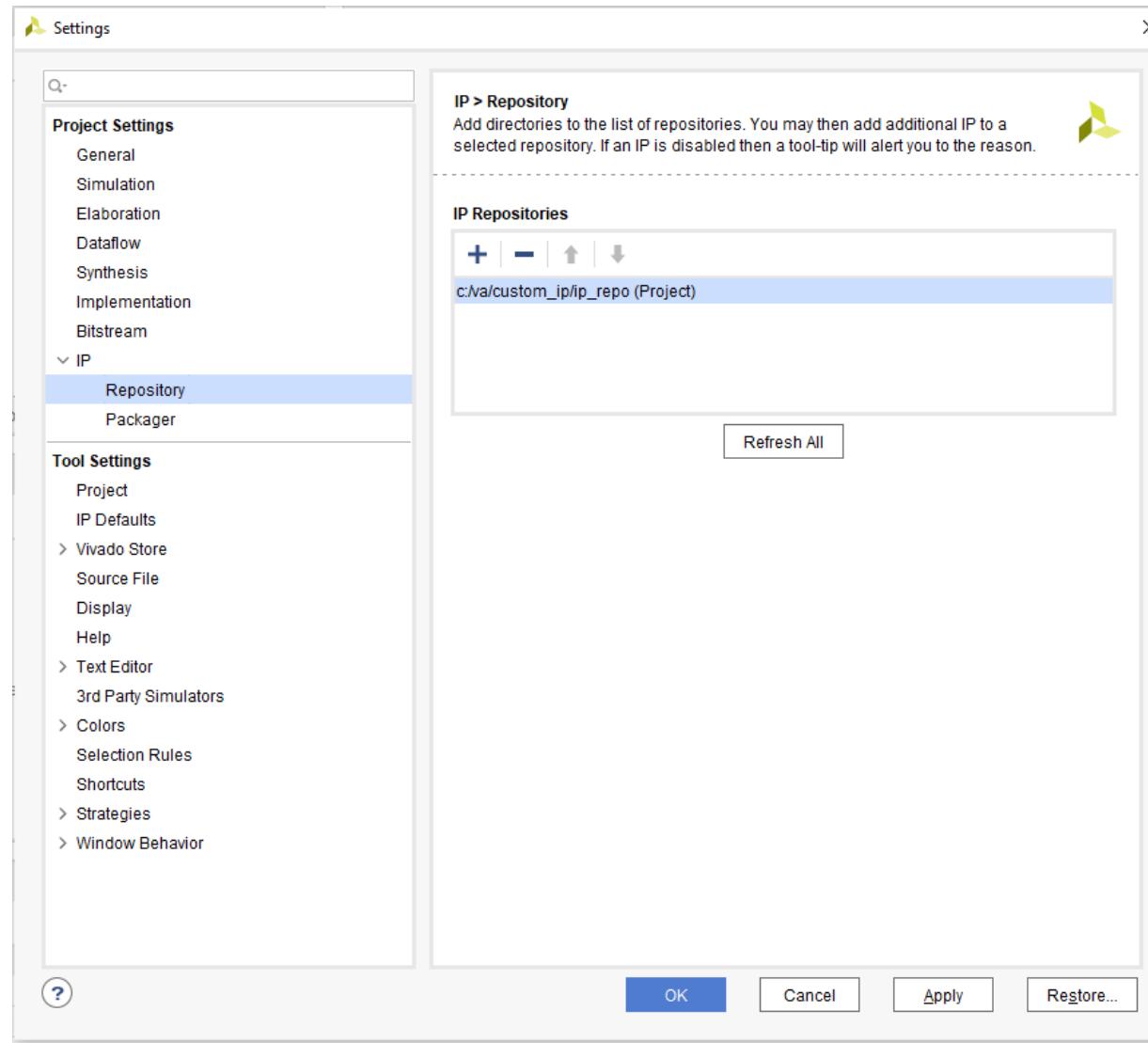
IP GUI Setting

You can review the previously configured interfaces and parameters.



IP Repository

You need to set the IP path to use the IP in our design.



Makefile modify

Name	Date modified	Type	Size
axiled.c	11/23/2023 9:10 AM	C 來源檔案	1 KB
axiled.h	11/23/2023 9:10 AM	C Header 來源檔案	3 KB
axiled_selftest.c	11/23/2023 9:10 AM	C 來源檔案	2 KB
Makefile	11/23/2023 11:59 AM	File	1 KB

INCLUDEFILES=*.h
LIBSOURCES=*.c
OUTS = *.o



INCLUDEFILES=\$(wildcard *.h)
LIBSOURCES=\$(wildcard *.c)
OUTS = \$(wildcard *.o)

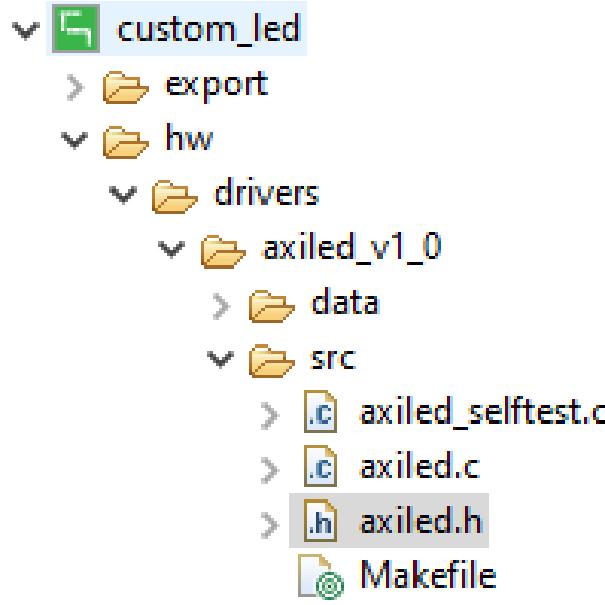
```

1 COMPILER=
2 ARCHIVER=
3 CP=cp
4 COMPILER_FLAGS=
5 EXTRA_COMPILER_FLAGS=
6 LIB=libxil.a
7
8 RELEASEDIR=../../../../lib
9 INCLUDEDIR=../../../../include
10 INCLUDES=-I.. -I$(INCLUDEDIR)
11
12 INCLUDEFILES=$(wildcard *.h)
13 LIBSOURCES=$(wildcard *.c)
14 OUTS = $(wildcard *.o)
15
16 libs:
17     echo "Compiling axiled..."
18     $(COMPILER) $(COMPILER_FLAGS) $(EXTRA_COMPILER_FLAGS) $(INCLUDES) $(LIBSOURCES)
19     $(ARCHIVER) -r $(RELEASEDIR)/$(LIB) $(OUTS)
20     make clean
21
22 include:
23     $(CP) $(INCLUDEFILES) $(INCLUDEDIR)
24
25 clean:
26     rm -rf $(OUTS)
27

```

Vitis Design Flow

Vitis will generate corresponding APIs based on the XSA file for usage.



```
#include <stdio.h>
#include "axiled.h"
#include "xparameters.h"
#include "platform.h"
#include "xil_printf.h"
#include "xil_io.h"
#include "sleep.h"

#define LED_IP_BASEADDR XPAR_AXILED_0_S00_AXI_BASEADDR
#define LED_IP_REG0    AXILED_S00_AXI_SLV_REG0_OFFSET
#define LED_IP_REG1    AXILED_S00_AXI_SLV_REG1_OFFSET

int main()
{
    init_platform();

    int freq_flag = 0;
    while(1){
        if(freq_flag == 0){
            AXILED_mWriteReg(LED_IP_BASEADDR, LED_IP_REG1, 0x000000ff);
            freq_flag = 1;
        }
        else{
            AXILED_mWriteReg(LED_IP_BASEADDR, LED_IP_REG1, 0x0000002f);
            freq_flag = 0;
        }

        AXILED_mWriteReg(LED_IP_BASEADDR, LED_IP_REG0, 0x00000001);
        sleep(5);
        AXILED_mWriteReg(LED_IP_BASEADDR, LED_IP_REG0, 0x00000000);
        sleep(1);
    }
    return 0;
}
```

Breath_LED

```
#include <stdio.h>
#include "axiled.h"
#include "xparameters.h"
#include "platform.h"
#include "xil_printf.h"
#include "xil_io.h"
#include "sleep.h"

#define LED_IP_BASEADDR XPAR_AXILED_0_S00_AXI_BASEADDR
#define LED_IP_REG0 AXILED_S00_AXI_SLV_REG0_OFFSET
#define LED_IP_REG1 AXILED_S00_AXI_SLV_REG1_OFFSET
```

Breath_LED

```
int main()
{
    init_platform();

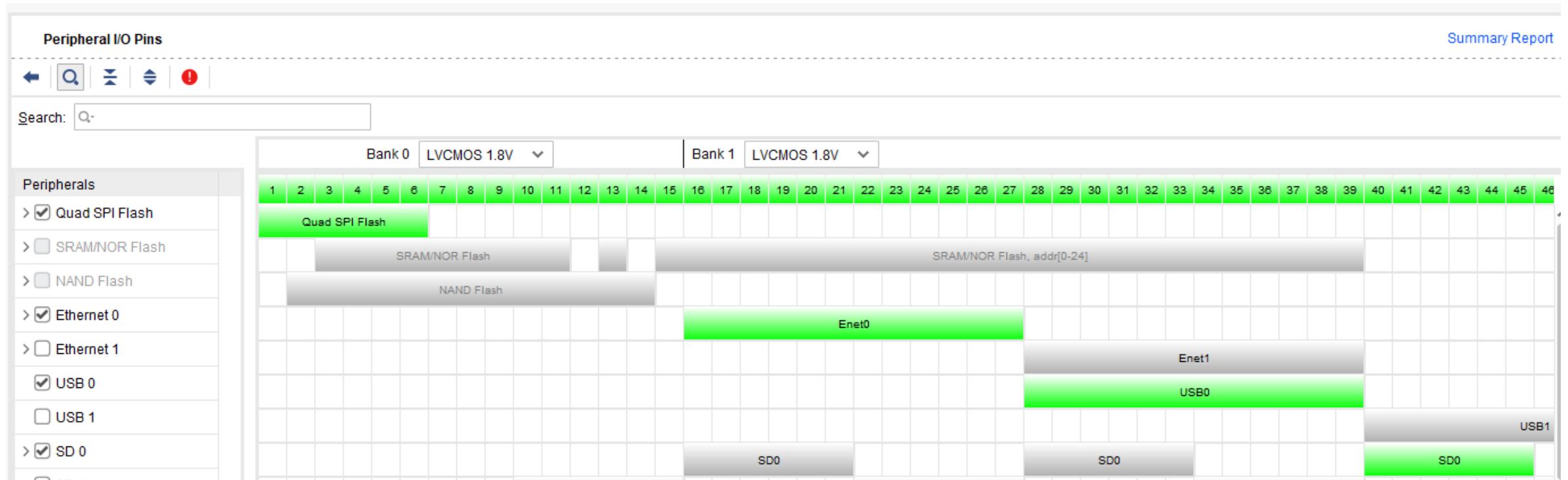
    int freq_flag = 0;
    while(1){
        if(freq_flag == 0){
            AXILED_mWriteReg(LED_IP_BASEADDR, LED_IP_REG1, 0x000000ff);
            freq_flag = 1;
        }
        else{
            AXILED_mWriteReg(LED_IP_BASEADDR, LED_IP_REG1, 0x0000002f);
            freq_flag = 0;
        }

        AXILED_mWriteReg(LED_IP_BASEADDR, LED_IP_REG0, 0x00000001);
        sleep(5);
        AXILED_mWriteReg(LED_IP_BASEADDR, LED_IP_REG0, 0x00000000);
        sleep(1);
    }
    return 0;
}
```

SD Card or Flash Boot

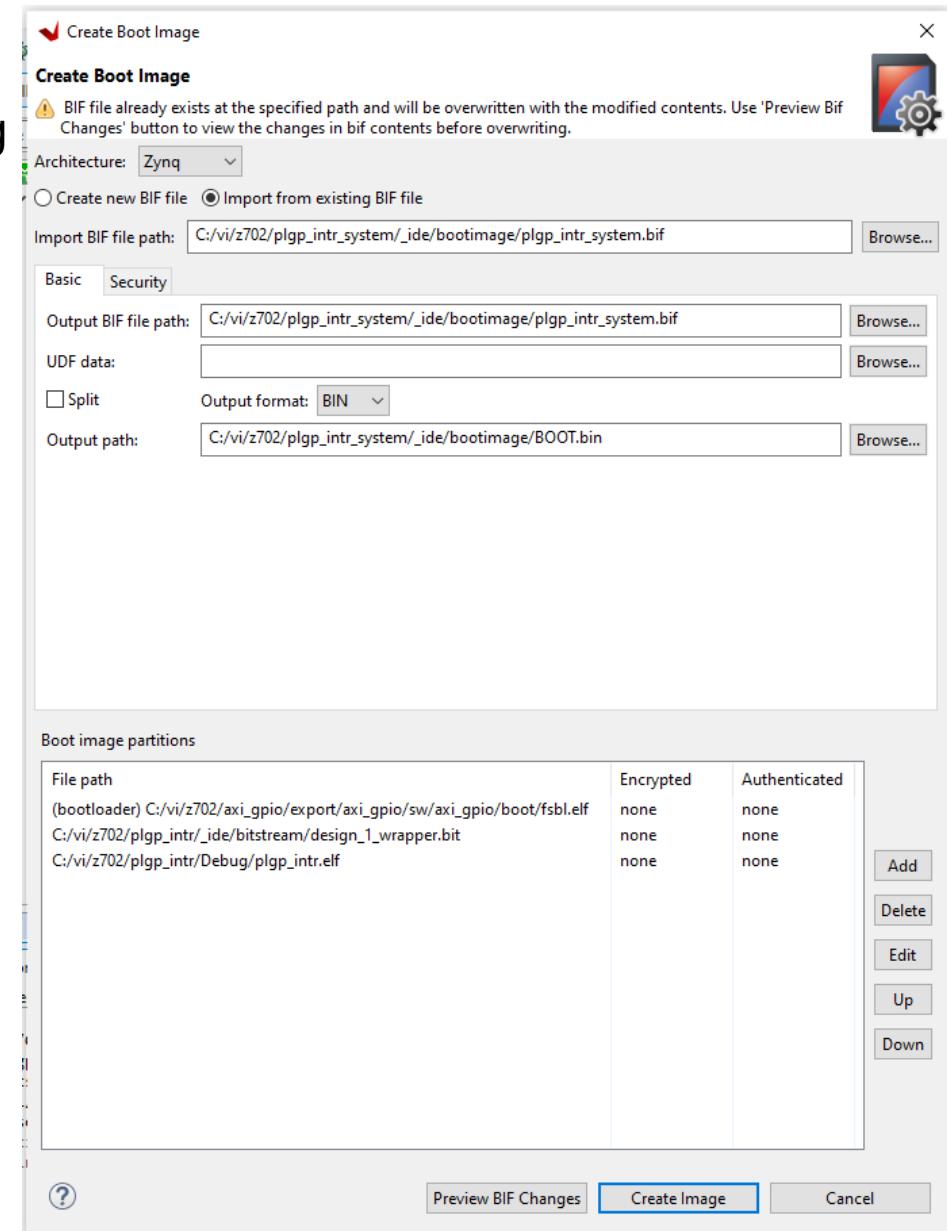
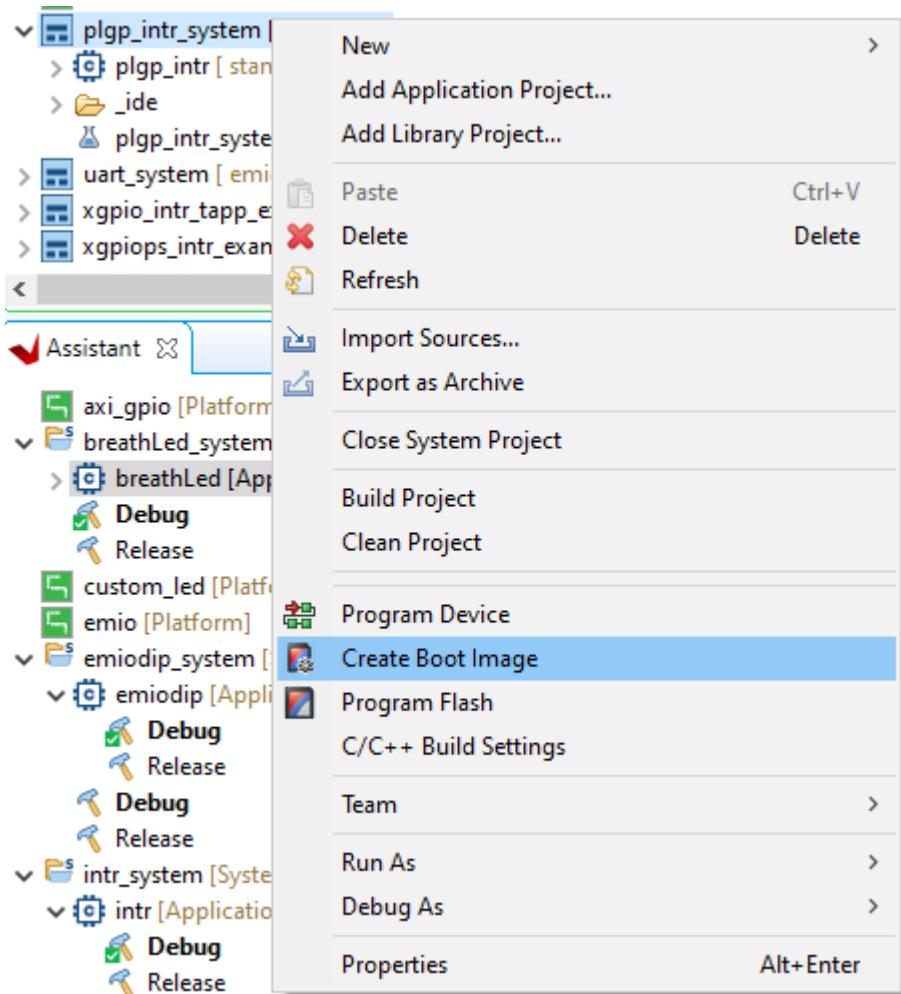
MIO Configuration

Ensure that the SD card and SPI flash are enabled.



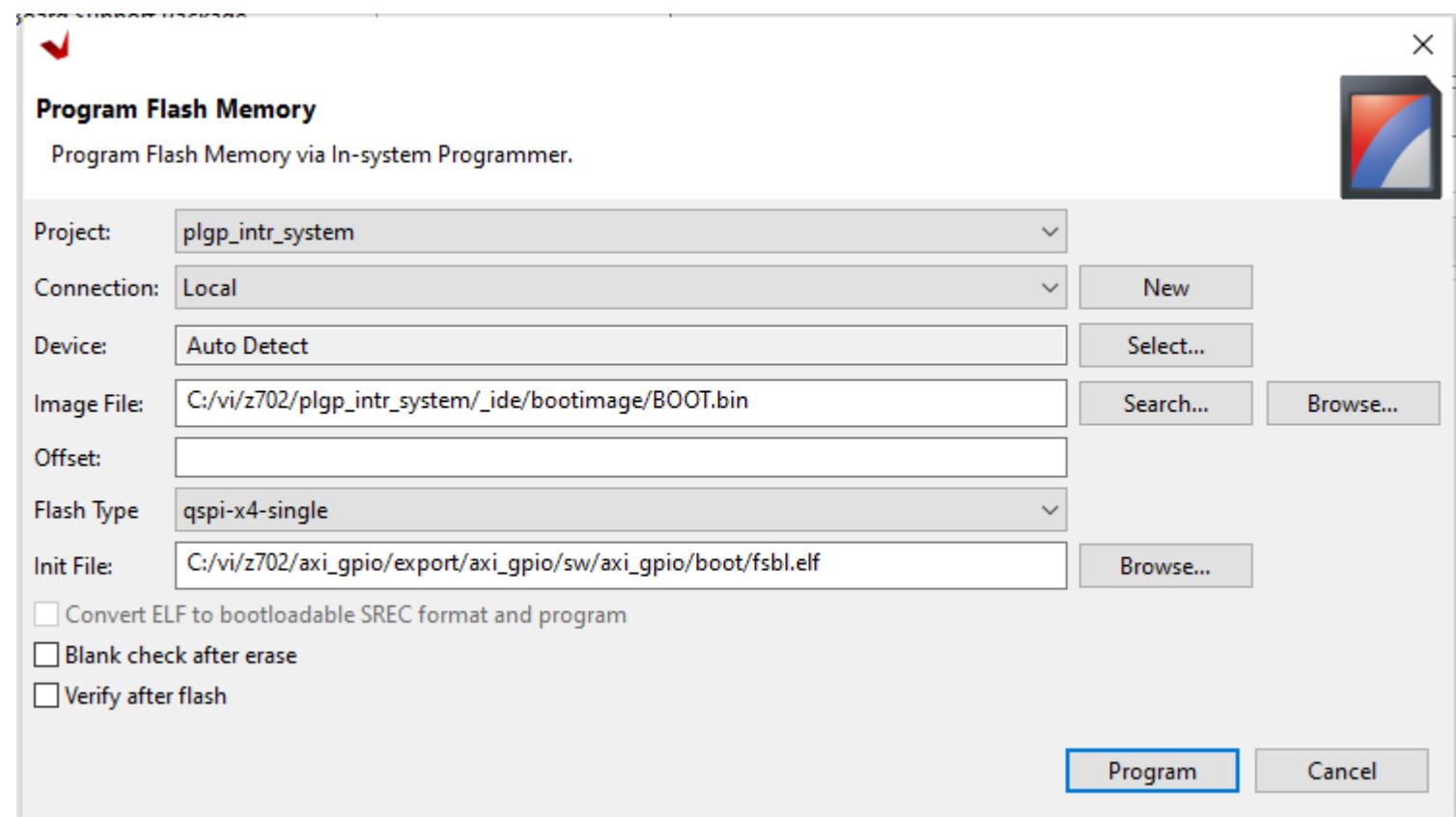
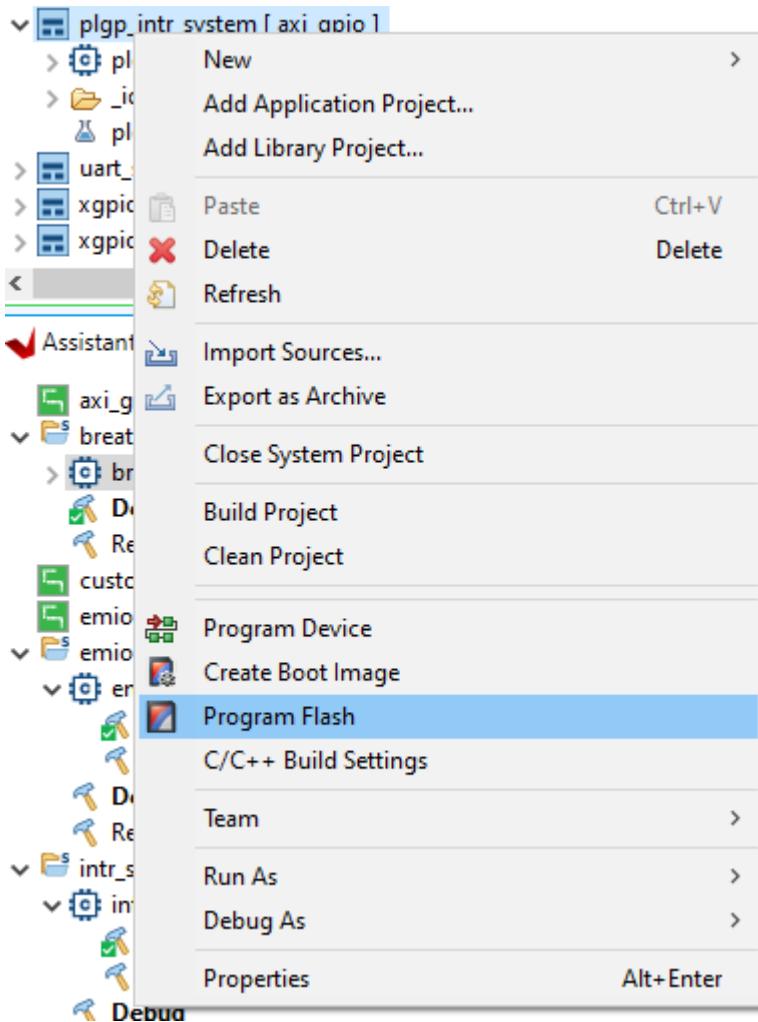
Create Boot Image

The software will automatically detect the corresponding files and package them into a bootable image.



Program Flash

The software will display the corresponding boot flash based on the linked device.



UART Controller

Mio Configuration

UART is a form of asynchronous communication.

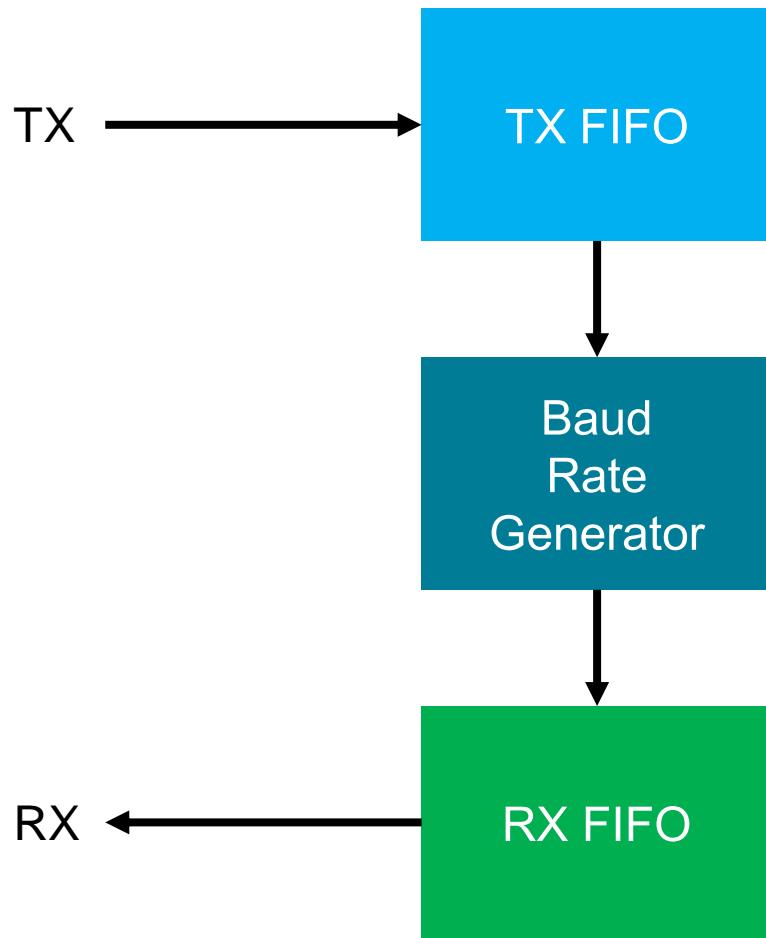
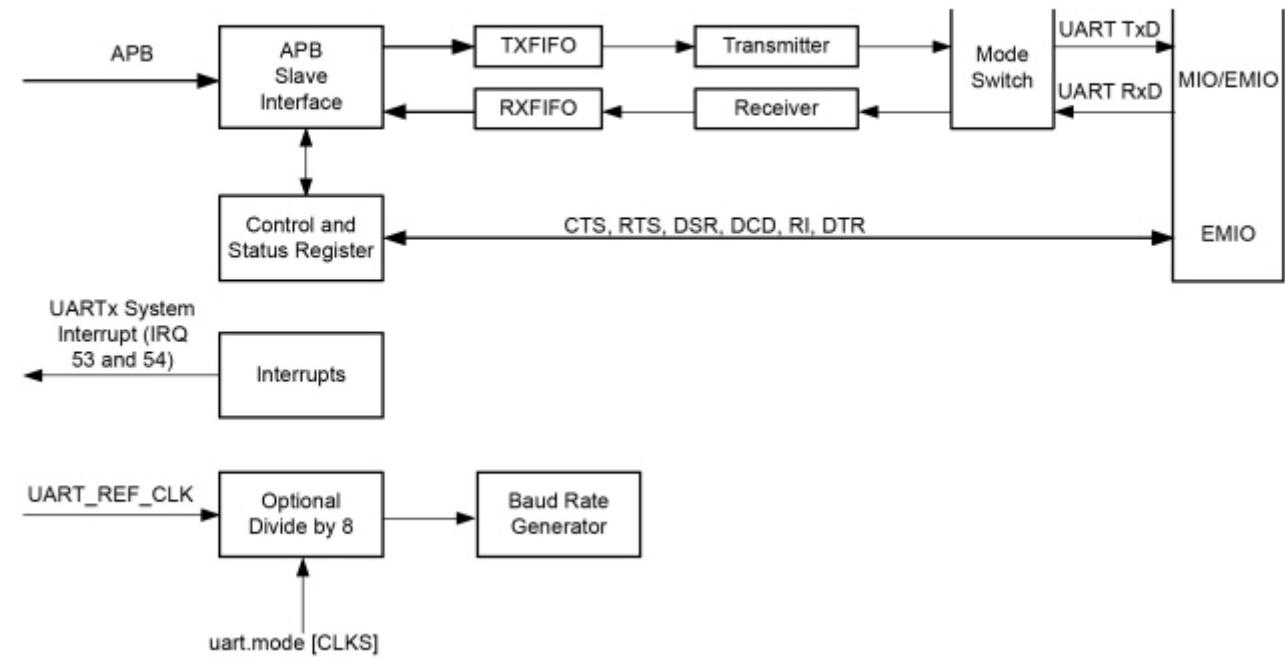


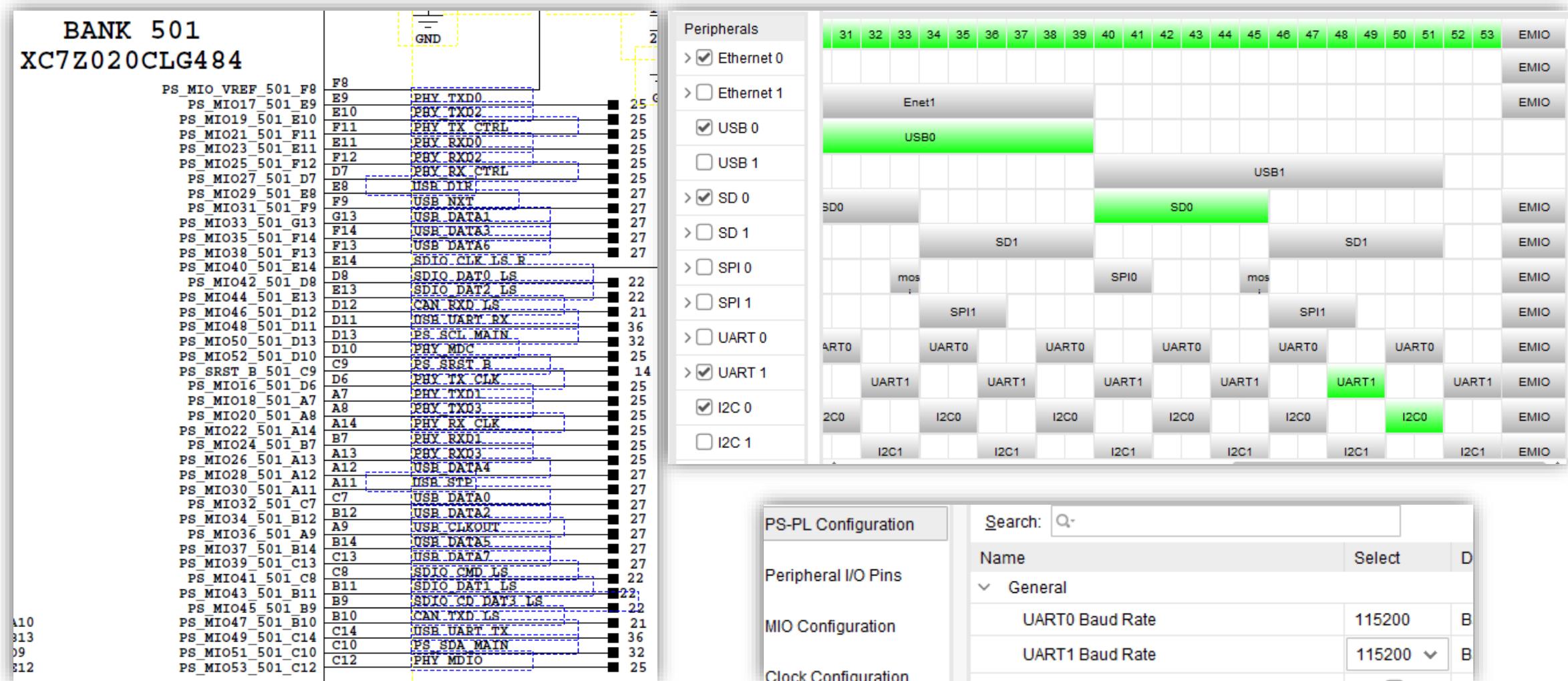
Figure: UART Controller



X15379-12051E

Mio Configuration

Zynq 702 uses UART1 by default.



UART Controller Header

In this lab, we attempt to independently create a UART header file and use a single main function to call it. The purpose of the first two lines is to avoid errors due to repeated declarations. You can centralize modifications for the required header files, declared definitions, and self-written functions in this section.

```
1 #ifndef __USER_UART_H
2 #define __USER_UART_H
3
4 #include "xparameters.h"
5 #include "xuartps.h"
6 #include "xil_printf.h"
7 #include "sleep.h"
8
9 #define UART_DEVICE_ID      XPAR_XUARTPS_0_DEVICE_ID
10 #define TEST_BUFFER_SIZE    100
11
12 int Uart_Send(XUartPs *Uart_Ps,u8 *sendbuf,int length);
13 int Uart_Init(XUartPs *Uart_Ps,u16 DeviceID);
14
15 #endif
16
```

UART Controller Init

Next is the writing of the controller itself.

First, naturally, include the header file we just created.

The second step is similar to previous practices – configuration and initialization.

The third step using Vitis' provided API for self-testing.

At this point, the general initialization steps have been completed. Now, we can make modifications for some commonly used settings. These settings are usually generated when creating the platform. For ease of modification, we will move some of these settings to the outer layer.

You can modify the operation mode and BaudRate settings using a similar approach as the following two functions. Alternatively, you can declare a structure to handle the modifications:

```
1 #include "user_uart.h"
2
3 //XUartPsFormat uart_format =
4 //{
5 //    115200,
6 //    XUARTPS_FORMAT_8_BITS,
7 //    XUARTPS_FORMAT_NO_PARITY,
8 //    XUARTPS_FORMAT_1_STOP_BIT,
9 //};
10
11 int Uart_Init(XUartPs *Uart_Ps , u16 DeviceId){
12     int Status;
13     XUartPs_Config *Config;
14
15     Config = XUartPs_LookupConfig(DeviceId);
16     if (NULL == Config) {
17         return XST_FAILURE;
18     }
19
20     Status = XUartPs_CfgInitialize(Uart_Ps, Config, Config->BaseAddress);
21     if (Status != XST_SUCCESS) {
22         return XST_FAILURE;
23     }
24
25     /* Check hardware build */
26     Status = XUartPs_SelfTest(Uart_Ps);
27     if (Status != XST_SUCCESS) {
28         return XST_FAILURE;
29     }
30
31     XUartPs_SetOperMode(Uart_Ps, XUARTPS_OPER_MODE_NORMAL);
32 //    XUartPs_SetBaudRate(Uart_Ps, 9600);
33
34 //    XUartPs_SetDataFormat(Uart_Ps, &uart_format);
35
36     return XST_SUCCESS;
37 }
```

UART Operation Mode

Normal Mode

Normal mode is used for standard UART operations.

Automatic Echo Mode

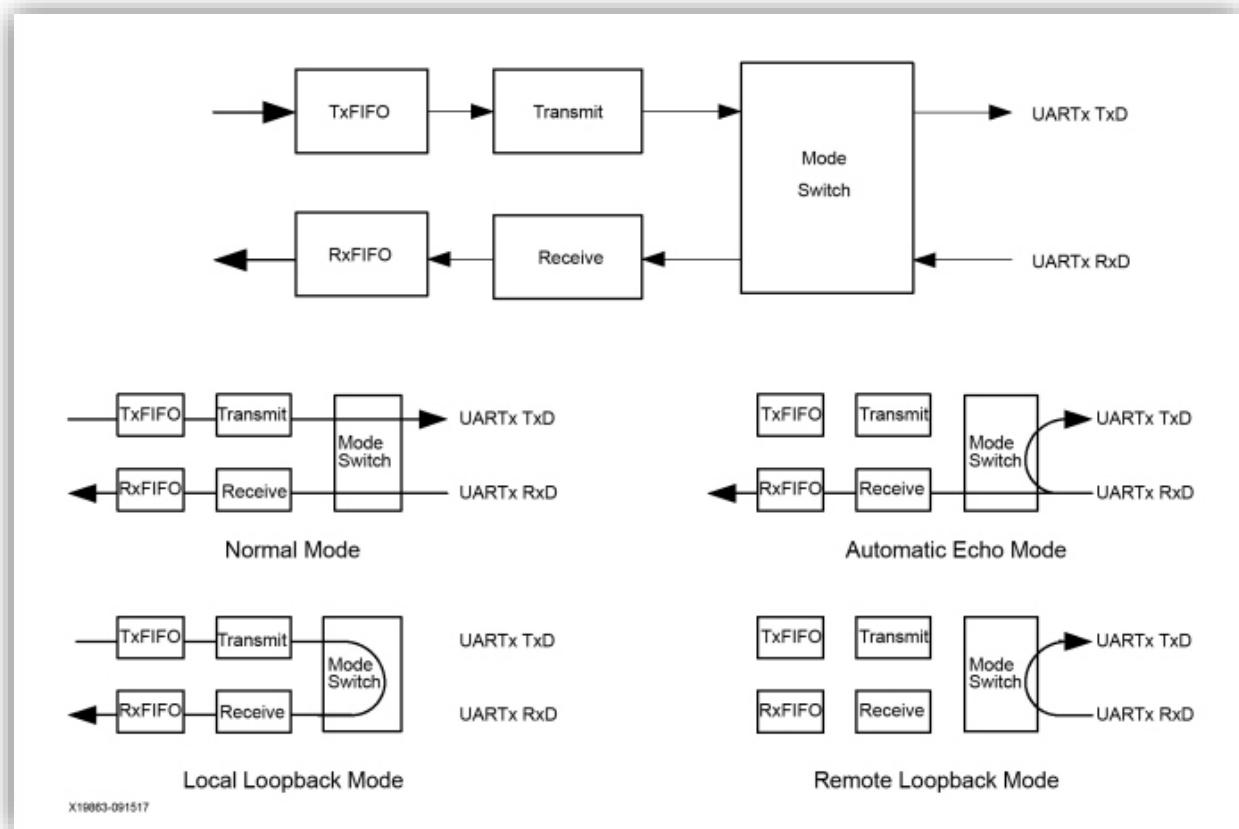
Echo mode receives data on RxD and the mode switch routes the data to both the receiver and the TxD pin. **Data from the transmitter cannot be sent out from the controller.**

Local Loopback Mode

Local loopback mode does **not connect to the RxD or TxD pins**. Instead, the transmitted data is looped back around to the receiver.

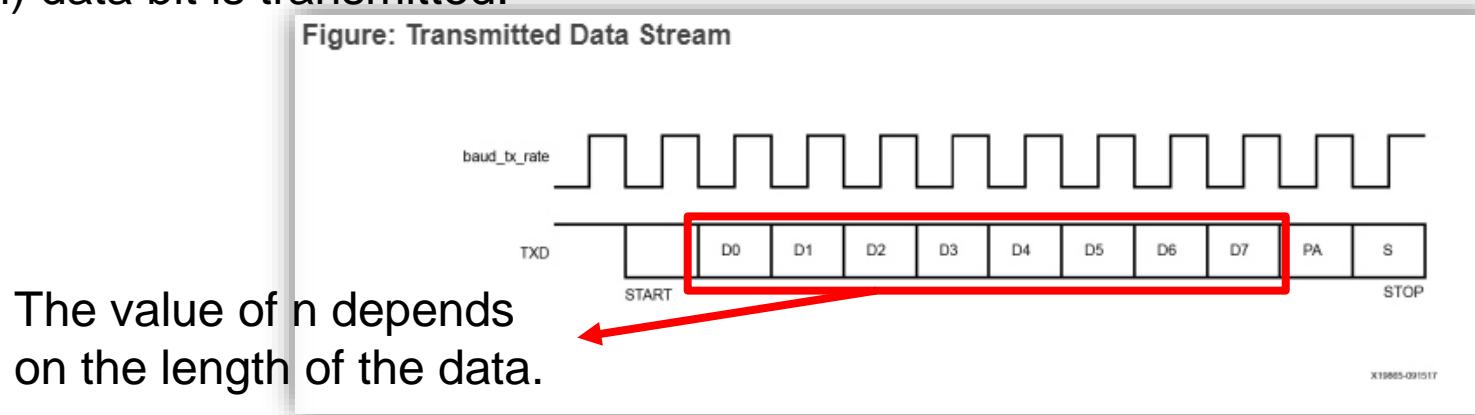
Remote Loopback Mode

Remote loopback mode connects the RxD signal to the TxD signal. In this mode, **the controller cannot send anything on TxD and the controller does not receive anything on RxD**.



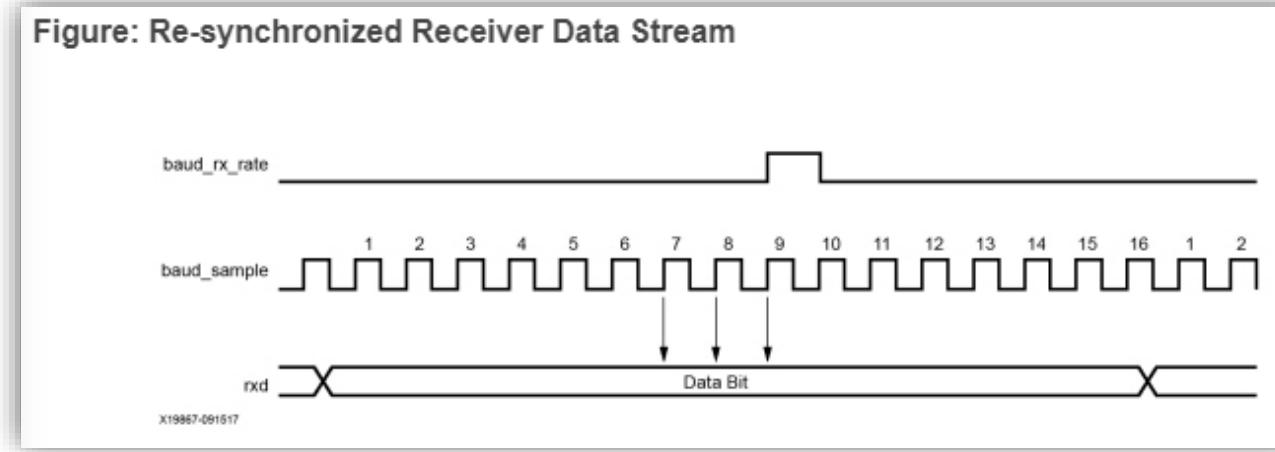
UART Controller Send & Main Function

As data is stored in the FIFO, we will first define a Receive Buffer area to represent it. Each rising edge of the clock triggers the transmission of one data bit, continuing until the nth(Length) data bit is transmitted.



During reception, the sampling is done bit by bit.

To avoid sampling errors, each bit is sampled 16 times. EX: Baud Rate **9600** need **153.6KHz** Clock(**9600 x 16**)



UART Controller Send & Main Function

You need to define a transmission buffer and use `uart_send` to transmit the data received from the terminal into the buffer.

As the buffer is essentially a matrix, if you want to completely receive the data within the buffer, you need to use a loop to receive it.

```
1 #include "user_uart.h"
2
3 static u8 RecvBuffer[TEST_BUFFER_SIZE];
4
5 XUartPs Uart_Ps;
6
7 int main(void){
8     int Status;
9 //    u8 sendbuf[] = "Hello World!\r\n";
10
11     Status = Uart_Init(&Uart_Ps, UART_DEVICE_ID);
12     if(Status == XST_FAILURE){
13         xil_printf("Uartps Failed\r\n");
14         return XST_FAILURE;
15     }
16
17     while(1){
18         Uart_Send(&Uart_Ps, RecvBuffer, 6);
19         print("UART Test !");
20     }
21     return XST_SUCCESS;
22 }
```

```
int Uart_Send(XUartPs *Uart_Ps, u8 *RecvBuffer, int length){
    u16 RC,TC;

    RC = 0;
    while(RC < length){
        RC += XUartPs_Recv(Uart_Ps, &RecvBuffer[RC], (length - RC));
    }
    TC = XUartPs_Send(Uart_Ps, RecvBuffer, length);
    if (TC != length){
        return XST_FAILURE;
    }
}
```

Uart.h

```
#ifndef __USER_UART_H
#define __USER_UART_H

#include "xparameters.h"
#include "xuartps.h"
#include "xil_printf.h"
#include "sleep.h"

#define UART_DEVICE_ID XPAR_XUARTPS_0_DEVICE_ID
#define TEST_BUFFER_SIZE100

int Uart_Send(XUartPs *Uart_Ps,u8 *sendbuf,int length);
int Uart_Init(XUartPs *Uart_Ps,u16 DeviceID);

#endif
```

Main.c

```
#include "user_uart.h"

static u8 RecvBuffer[TEST_BUFFER_SIZE];

XUartPs Uart_Ps;

int main(void){
    int Status;
    //u8 sendbuf[] = "Hello World!\r\n";

    Status = Uart_Init(&Uart_Ps, UART_DEVICE_ID);
    if(Status == XST_FAILURE){
        xil_printf("Uartps Failed\r\n");
        return XST_FAILURE;
    }

    while(1){
        Uart_Send(&Uart_Ps, RecvBuffer, 6);
        print("UART Test !");
    }
    return XST_SUCCESS;
}
```

User uart.h

```
#include "user_uart.h"
//XUartPsFormat uart_format =
//{
//115200,
//XUARTPS_FORMAT_8_BITS,
//XUARTPS_FORMAT_NO_PARITY,
//XUARTPS_FORMAT_1_STOP_BIT,
//};

int Uart_Init(XUartPs *Uart_Ps , u16 DeviceId){
    int Status;
    XUartPs_Config *Config;
    Config = XUartPs_LookupConfig(DeviceId);
    if (NULL == Config) {
        return XST_FAILURE;
    }
    Status = XUartPs_CfgInitialize(Uart_Ps, Config, Config->BaseAddress);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }
/* Check hardware build */
    Status = XUartPs_SelfTest(Uart_Ps);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }
    XUartPs_SetOperMode(Uart_Ps, XUARTPS_OPER_MODE_NORMAL);
    return XST_SUCCESS;
}
```

User uart.h

```
int Uart_Send(XUartPs *Uart_Ps,u8 *RecvBuffer, int length){  
    u16 RC,TC;  
  
    RC = 0;  
    while(RC < length){  
        RC += XUartPs_Recv(Uart_Ps, &RecvBuffer[RC], (length - RC));  
    }  
    TC = XUartPs_Send(Uart_Ps, RecvBuffer, length);  
    if (TC != length){  
        return XST_FAILURE;  
    }  
}
```

Timer

Timer

The screenshot shows the XPS Clock Configuration interface. The left sidebar has a 'Clock Configuration' tab selected. The main area displays clock settings for the PS Hardcore Timer and the Timer to PL.

PS Hardcore Timer: The CPU is configured with an ARM PLL at 666.666666 MHz, resulting in an actual frequency of 666.666687 MHz. The DDR is configured with a DDR PLL at 533.333333 MHz, resulting in an actual frequency of 533.333374 MHz.

Component	Clock Source	Requested Freq...	Actual Frequency(...)	Range(MHz)
CPU	ARM PLL	666.666666	666.666687	50.0 : 667.0
DDR	DDR PLL	533.333333	533.333374	200.000000 : 534.000000

Timer to PL: The WDT is configured with a CPU_1X source at 133.333333 MHz, resulting in an actual frequency of 111.111115 MHz. The TTC0 CLKIN0 is also configured with a CPU 1X source at 133.333333 MHz, resulting in an actual frequency of 111.111115 MHz.

Component	Clock Source	Requested Freq...	Actual Frequency(...)	Range(MHz)
WDT	CPU_1X	133.333333	111.111115	0.100000 : 200.000000
TTC0 CLKIN0	CPU 1X	133.333333	111.111115	0.100000 : 200.000000

Annotations with arrows point from the text labels to the corresponding rows in the table.

Timer

In this experiment, we will modify the "mio_lab" by replacing the sleep function used inside with a timer calculation.

```
1 #include <stdio.h>
2 #include "platform.h"
3 #include "xil_printf.h"
4 #include "xgpiops.h"
5 #include "xparameters.h"
6 #include "xbasic_types.h"
7 #include "sleep.h"
8 #define GPIO_DEVICE_ID      XPAR_XGPIOPS_0_DEVICE_ID
9
10 XGpioPs Gpio;
11 XGpioPs_Config *ConfigPtr;
12
13 int main()
14 {
15     ConfigPtr = XGpioPs_LookupConfig(GPIO_DEVICE_ID);
16
17     XGpioPs_CfgInitialize(&Gpio, ConfigPtr , ConfigPtr->BaseAddr);
18
19     XGpioPs_SetDirectionPin(&Gpio, 10, 1);
20
21     XGpioPs_SetOutputEnablePin(&Gpio, 10, 1);|
22
23     while(1){
24         XGpioPs_WritePin(&Gpio, 10, 1);
25
26         sleep(1);
27
28         XGpioPs_WritePin(&Gpio, 10, 0);
29
30         sleep(1);
31     }
32
33     return 0;
34 }
```

Timer

Key Function:

- EnableAutoReload

Automatically reset to zero and restart when the timer completes.

- LoadTimer

Set the timer duration.

If we want to achieve a 100ms timer:

$$0.1 \times 666.666\text{MHz}(\text{PS Clock}) = 66666666.6(\text{DEC}) \Rightarrow 3F94067(\text{HEX})$$

```

1 #include <stdio.h>
2 #include "platform.h"
3 #include "xil_printf.h"
4 #include "xgpiops.h"
5 #include "xparameters.h"
6 #include "xbasic_types.h"
7 #include "xscutimer.h"
8 #include "xscugic.h"
9
10 #define GPIO_DEVICE_ID XPAR_XGPIOPS_0_DEVICE_ID
11 #define TIMER_DEVICE_ID XPAR_XSCUTIMER_0_DEVICE_ID
12 #define INTC_DEVICE_ID XPAR_SCUGIC_SINGLE_DEVICE_ID
13 #define TIMER_IRQT_INTR XPAR_SCUTIMER_INTR
14 #define TIMER_LOAD_VALUE 0x3F94067
15
16 XGpioPs Gpio;
17 XScuTimer TimerInstance;
18 XScuGic IntcInstance;
19 XScuTimer TimerInstancePtr;
20
21 void mio_init(){
22
23     XGpioPs_Config *ConfigPtr;
24     ConfigPtr = XGpioPs_LookupConfig(GPIO_DEVICE_ID);
25     XGpioPs_CfgInitialize(&Gpio, ConfigPtr, ConfigPtr->BaseAddr);
26
27     XGpioPs_SetDirectionPin(&Gpio, 10, 1);
28
29     XGpioPs_SetOutputEnablePin(&Gpio, 10, 1);
30 }
31
32 void timer_init(){
33
34     XScuTimer_Config *ConfigPtr;
35     ConfigPtr = XScuTimer_LookupConfig(TIMER_DEVICE_ID);
36     XScuTimer_CfgInitialize(&TimerInstance, ConfigPtr, ConfigPtr->BaseAddr);
37
38     XScuTimer_EnableAutoReload(&TimerInstance);
39
40     XScuTimer_LoadTimer(&TimerInstance, TIMER_LOAD_VALUE);
41
42 }
43
44 }
```

Timer

Connect the interrupt to the timer,
triggering the interrupt when the timer completes.

Finally, make sure to start the timer for it to begin counting.

```

46④ static void TimerIntrHandler(void *CallBackRef)
47 {
48     int led_status = 0;
49     XScuTimer *TimerInstancePtr = (XScuTimer *) CallBackRef;
50     print("led = %d \r\n");
51     if (XScuTimer_IsExpired(TimerInstancePtr)) {
52         XScuTimer_ClearInterruptStatus(TimerInstancePtr);
53         XGpioPs_WritePin(&Gpio, 10, led_status);
54         led_status = ~led_status;
55     }
56 }
57
58④ void timerIntrSystem()
59 {
60     XScuGic_Config *IntcConfig;
61     IntcConfig = XScuGic_LookupConfig(INTC_DEVICE_ID);
62     XScuGic_CfgInitialize(&IntcInstance, IntcConfig, IntcConfig->CpuBaseAddress);
63
64     Xil_ExceptionInit();
65     Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_IRQ_INT,
66                                 (Xil_ExceptionHandler)XScuGic_InterruptHandler,
67                                 &IntcInstance);
68     Xil_ExceptionEnable();
69
70     XScuGic_Connect(&IntcInstance, TIMER_IRPT_INTR,
71                     (Xil_ExceptionHandler)TimerIntrHandler,
72                     (void *)&TimerInstancePtr);
73
74     XScuGic_Enable(&IntcInstance, TIMER_IRPT_INTR);
75     XScuTimer_EnableInterrupt(&TimerInstancePtr);
76
77 }
78
79
80
81④ int main()
82 {
83     mio_init();
84
85     timer_init();
86
87     timerIntrSystem();
88
89     XScuTimer_Start(&TimerInstancePtr);
90
91     return XST_SUCCESS;
92 }
93

```



Timer_test

```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xgpiops.h"
#include "xparameters.h"
#include "xbasic_types.h"
#include "xscutimer.h"
#include "xscugic.h"

#define GPIO_DEVICE_ID XPAR_XGPIOPS_0_DEVICE_ID
#define TIMER_DEVICE_ID XPAR_XSCUTIMER_0_DEVICE_ID
#define INTC_DEVICE_ID XPAR_SCUGIC_SINGLE_DEVICE_ID
#define TIMER_IRPT_INTR XPAR_SCUTIMER_INTR
#define TIMER_LOAD_VALUE 0x3F94067

XGpioPs Gpio;
XScuTimer TimerInstance;
XScuGic IntcInstance;
XScuTimer TimerInstancePtr;
```

Timer_test

```
void mio_init(){
    XGpioPs_Config *ConfigPtr;
    ConfigPtr = XGpioPs_LookupConfig(GPIO_DEVICE_ID);
    XGpioPs_CfgInitialize(&Gpio, ConfigPtr, ConfigPtr->BaseAddr);

    XGpioPs_SetDirectionPin(&Gpio, 10, 1);

    XGpioPs_SetOutputEnablePin(&Gpio, 10, 1);

    print("mio_init success\r\n");
}

void timer_init(){

    XScuTimer_Config *ConfigPtr;
    ConfigPtr = XScuTimer_LookupConfig(TIMER_DEVICE_ID);
    XScuTimer_CfgInitialize(&TimerInstance, ConfigPtr, ConfigPtr->BaseAddr);

    XScuTimer_EnableAutoReload(&TimerInstance);

    XScuTimer_LoadTimer(&TimerInstance, TIMER_LOAD_VALUE);

    print("timer_init success\r\n");
}
```

Timer_test

```
static void TimerIntrHandler(void *CallBackRef) {
    int led_status = 0;
    XScuTimer *TimerInstancePtr = (XScuTimer *) CallBackRef;
    print("led = 0 \r\n");
    if (XScuTimer_IsExpired(TimerInstancePtr)) {
        XScuTimer_ClearInterruptStatus(TimerInstancePtr);
        XGpioPs_WritePin(&Gpio, 10, led_status);
        led_status = ~led_status;
        print("led change\r\n");
    }
}
```

Timer_test

```
void timerIntrSystem() {
    XScuGic_Config *IntcConfig;
    IntcConfig = XScuGic_LookupConfig(INTC_DEVICE_ID);
    XScuGic_CfgInitialize(&IntcInstance, IntcConfig, IntcConfig->CpuBaseAddress);

    Xil_ExceptionInit();
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_IRQ_INT,
        (Xil_ExceptionHandler)XScuGic_InterruptHandler, &IntcInstance);
    Xil_ExceptionEnable();

    print("exception enable\r\n");

    XScuGic_Connect(&IntcInstance, TIMER_IRPT_INTR,
        (Xil_ExceptionHandler)TimerIntrHandler,
        (void *)&TimerInstancePtr);

    print("connect\r\n");

    XScuGic_Enable(&IntcInstance, TIMER_IRPT_INTR);
    XScuTimer_EnableInterrupt(&TimerInstancePtr);

    print("intr finish\r\n");
    XScuTimer_Start(&TimerInstancePtr);

}
```

Timer_test

```
int main() {
    mio_init();

    timer_init();

    timerIntrSystem();

    print("timer start\r\n");

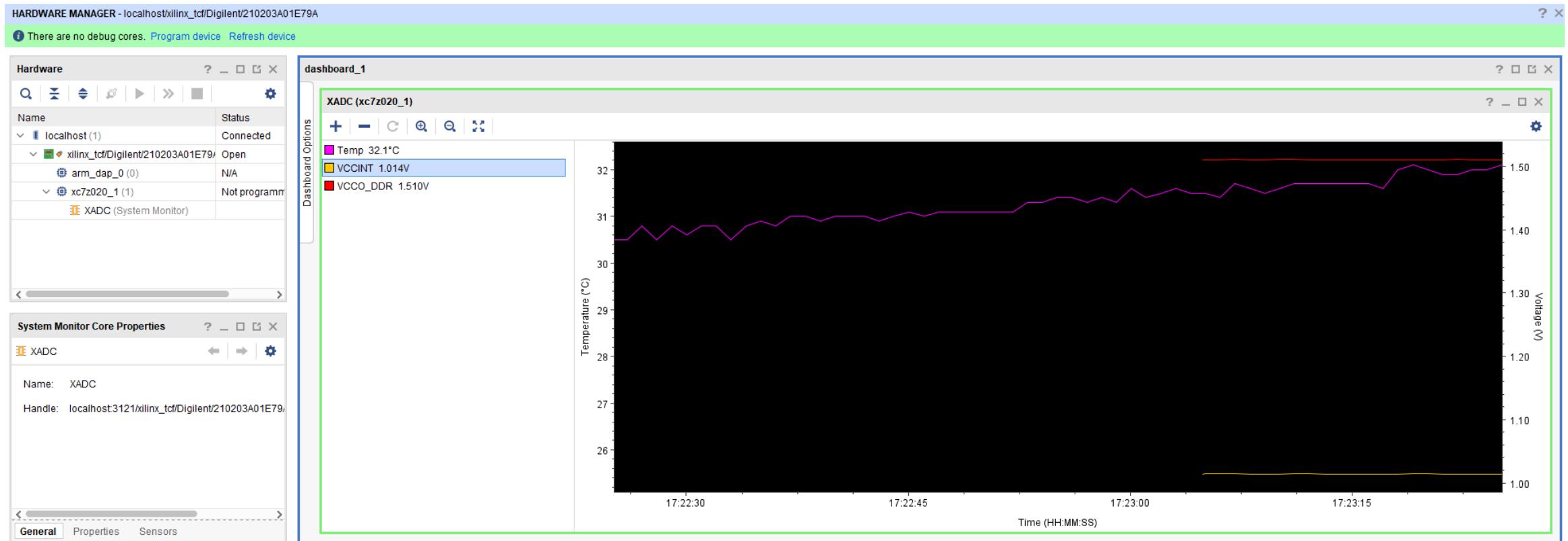
    print("finish\r\n");

    return XST_SUCCESS;
}
```

XADC

XADC

The XADC (Xilinx Analog-to-Digital Converter) is a hard core, and there is no need to program a bit file to obtain chip temperature and voltage through the XADC.



XADC

You can also pass the values detected by XADC to the PS (Processing System) for further use.

Configure the numerical format and initialize.

```
int XAdcPolledPrintfExample(u16 XAdcDeviceId)
{
    int Status;
    XAdcPs_Config *ConfigPtr;
    u32 TempRawData;
    u32 VccPintRawData;
    u32 VccPauxRawData;
    u32 VccPdroRawData;
    float TempData;
    float VccPintData;
    float VccPauxData;
    float MaxData;
    float MinData;
    XAdcPs *XAdcInstPtr = &XAdcInst;

    printf("\r\nEntering the XAdc Polled Example. \r\n");

    /*
     * Initialize the XAdc driver.
     */
    ConfigPtr = XAdcPs_LookupConfig(XAdcDeviceId);
    if (ConfigPtr == NULL) {
        return XST_FAILURE;
    }
    XAdcPs_CfgInitialize(XAdcInstPtr, ConfigPtr,
                         ConfigPtr->BaseAddress);
```

Set channel mode & Get XADC values.

```
* This function sets the specified Channel Sequencer Mode in the Configuration
* Register 1 :
*   - Default safe mode (XADCPS_SEQ_MODE_SAFE)
*   - One pass through sequence (XADCPS_SEQ_MODE_ONEPASS)
*   - Continuous channel sequencing (XADCPS_SEQ_MODE_CONTINPASS)
*   - Single Channel/Sequencer off (XADCPS_SEQ_MODE_SINGCHAN)
*   - Simultaneous sampling mode (XADCPS_SEQ_MODE_SIMUL_SAMPLING)
*   - Independent mode (XADCPS_SEQ_MODE_INDEPENDENT)
*
XAdcPs_SetSequencerMode(XAdcInstPtr, XADCPS_SEQ_MODE_SAFE);
/*
 * Read the on-chip Temperature Data (Current/Maximum/Minimum)
 * from the ADC data registers.
 */
TempRawData = XAdcPs_GetAdcData(XAdcInstPtr, XADCPS_CH_TEMP);
TempData = XAdcPs_RawToTemperature(TempRawData);
printf("\r\nThe Current Temperature is %0d.%#03d Centigrades.\r\n",
       (int)(TempData), XAdcFractionToInt(TempData));

TempRawData = XAdcPs_GetMinMaxMeasurement(XAdcInstPtr, XADCPS_MAX_TEMP);
MaxData = XAdcPs_RawToTemperature(TempRawData);
printf("The Maximum Temperature is %0d.%#03d Centigrades. \r\n",
       (int)(MaxData), XAdcFractionToInt(MaxData));

TempRawData = XAdcPs_GetMinMaxMeasurement(XAdcInstPtr, XADCPS_MIN_TEMP);
MinData = XAdcPs_RawToTemperature(TempRawData & 0xFFFF0);
printf("The Minimum Temperature is %0d.%#03d Centigrades. \r\n",
       (int)(MinData), XAdcFractionToInt(MinData));
```

QSPI Read&Write

QSPI_MIO Configuration

The screenshot shows the MIO Configuration interface for a ZYNQ7 Processing System (5.5). The left sidebar has a 'Page Navigator' with options like Zynq Block Design, PS-PL Configuration, Peripheral I/O Pins, MIO Configuration (selected), Clock Configuration, DDR Configuration, SMC Timing Calculation, and Interrupts. The main area is titled 'MIO Configuration' and shows two dropdowns for Bank 0 I/O Voltage (LVCMS 1.8V) and Bank 1 I/O Voltage (LVCMS 1.8V). Below these are search and filter tools, and a table with columns: Peripheral, IO, Signal, IO Type, Speed, Pullup, and Direction. A summary report button is also present. On the right, there's another 'Peripheral I/O Pins' section with a table for Bank 0 and Bank 1, showing pin assignments for Quad SPI Flash, SRAM/NOR Flash, and NAND Flash. A detailed view of the Quad SPI Flash assignment is shown in a separate window.

MIO Configuration

Bank 0 I/O Voltage: LVCMS 1.8V Bank 1 I/O Voltage: LVCMS 1.8V

Peripheral I/O Pins

Peripheral	IO	Signal	IO Type	Speed	Pullup	Direction
Memory Interfaces						
> <input checked="" type="checkbox"/> Quad SPI Flash	MIO 1 .. 6					
> <input type="checkbox"/> SRAM/NOR Flash						
> <input type="checkbox"/> NAND Flash						
I/O Peripherals						
> Application Processor Unit						
> Programmable Logic Test and Debug						

Peripheral I/O Pins

Peripheral	Bank 0	Bank 1
Quad SPI Flash	LVCMS 1.8V	LVCMS 1.8V
SRAM/NOR Flash		
NAND Flash		

Peripherals

Peripheral	Bank 0	Bank 1
Quad SPI Flash	LVCMS 1.8V	LVCMS 1.8V
SRAM/NOR Flash		
NAND Flash		

QSPI read/write verification steps

Referring to the example provided by Vitis:

1. Initialize the QSPI driver so that it's ready to use
2. Initialize the write buffer for a pattern to write to the FLASH
3. Set Manual Start and Manual Chip select options and drive HOLD_B pin high.
4. Read Flash ID
5. Enable Flash and Erase Flash
6. Write the data in the write buffer to the serial FLASH.
7. Read the contents of the FLASH from TEST_ADDRESS.
8. Verify the data read is the data that was written.

QSPI_init

Xilinx will generate corresponding flash parameters and command based on the development board. You can use them directly here.

```

28/* 
29 * The following constants define the offsets within a FlashBuffer data
30 * type for each kind of data. Note that the read data offset is not the
31 * same as the write data because the QSPI driver is designed to allow full
32 * duplex transfers such that the number of bytes received is the number
33 * sent and received.
34 */
35#define COMMAND_OFFSET      0 /* FLASH instruction */
36#define ADDRESS_1_OFFSET    1 /* MSB byte of address to read or write */
37#define ADDRESS_2_OFFSET    2 /* Middle byte of address to read or write */
38#define ADDRESS_3_OFFSET    3 /* LSB byte of address to read or write */
39#define DATA_OFFSET         4 /* Start of Data for Read/Write */
40#define DUMMY_OFFSET        4 /* Dummy byte offset for fast, dual and quad
41 * reads
42 */
43#define DUMMY_SIZE          1 /* Number of dummy bytes for fast, dual and
44 * quad reads
45 */
46#define RD_ID_SIZE          4 /* Read ID command + 3 bytes ID response */
47#define BULK_ERASE_SIZE     1 /* Bulk Erase command size */
48#define SEC_ERASE_SIZE      4 /* Sector Erase command + Sector address */
49 */
50/*
51 * The following constants specify the extra bytes which are sent to the
52 * FLASH on the QSPI interface, that are not data, but control information
53 * which includes the command and address
54 */
55#define OVERHEAD_SIZE       4
56*/

```

```

56/*
57 * The following constants specify the page size, sector size, and number of
58 * pages and sectors for the FLASH. The page size specifies a max number of
59 * bytes that can be written to the FLASH with a single transfer.
60 */
61#define SECTOR_SIZE         0x10000
62#define NUM_SECTORS         0x100
63#define NUM_PAGES           0x10000
64#define PAGE_SIZE            256
65
66/* Number of flash pages to be written.*/
67#define PAGE_COUNT          16
68
69/* Flash address to which data is to be written.*/
70#define TEST_ADDRESS         0x000055000
71#define UNIQUE_VALUE         0x05
72/*
73 * The following constants specify the max amount of data and the size of the
74 * the buffer required to hold the data and overhead to transfer the data to
75 * and from the FLASH.
76 */
77#define MAX_DATA             (PAGE_COUNT * PAGE_SIZE)

```

QSPI_Init

Declare the data to be written to the flash. This section can be modified by the user.

```
***** Variable Definitions *****/
/*
 * The instances to support the device drivers are global such that they
 * are initialized to zero each time the program runs. They could be local
 * but should at least be static so they are zeroed.
 */
static XQspiPs QspiInstance;

/*
 * The following variable allows a test value to be added to the values that
 * are written to the FLASH such that unique values can be generated to
 * guarantee the writes to the FLASH were successful
 */
int Test = 5;

/*
 * The following variables are used to read and write to the flash and they
 * are global to avoid having large buffers on the stack
 */
u8 ReadBuffer[MAX_DATA + DATA_OFFSET + DUMMY_SIZE];
u8 WriteBuffer[PAGE_SIZE + DATA_OFFSET];

*****
```

QSPI Init

1. Initialize the QSPI driver so that it's ready to use
2. Initialize the write buffer for a pattern to write to the FLASH

```

331 void qspi_init(){
332
333     u8 UniqueValue;
334     int Count;
335
336     /* Initialize the QSPI driver so that it's ready to use*/
337     XQspiPs_Config *QspiConfig;
338     QspiConfig = XQspiPs_LookupConfig(QSPI_DEVICE_ID);
339     XQspiPs_CfgInitialize(&QspiInstance, QspiConfig,
340                           QspiConfig->BaseAddress);
341
342     /*
343      * Initialize the write buffer for a pattern to write to the FLASH
344      * and the read buffer to zero so it can be verified after the read,
345      * the test value that is added to the unique value allows the value
346      * to be changed in a debug environment to guarantee
347      */
348     for (UniqueValue = UNIQUE_VALUE, Count = 0; Count < PAGE_SIZE;
349          Count++, UniqueValue++) {
350         WriteBuffer[DATA_OFFSET + Count] = (u8)(UniqueValue + Test);
351     }
352     memset(ReadBuffer, 0x00, sizeof(ReadBuffer));

```

3. Set Manual Start and Manual Chip select options and drive HOLD_B pin high.
4. Read Flash ID
5. Enable Flash and Erase Flash

```

354 /*
355  * Set Manual Start and Manual Chip select options and drive HOLD_B
356  * pin high.
357  */
358 XQspiPs_SetOptions(&QspiInstance, XQSPIPS_MANUAL_START_OPTION |
359                     XQSPIPS_FORCE_SSELECT_OPTION |
360                     XQSPIPS_HOLD_B_DRIVE_OPTION);
361
362 /* Set the prescaler for QSPI clock*/
363 XQspiPs_SetClkPrescaler(&QspiInstance, XQSPIPS_CLK_PRESCALE_8);
364
365 /* Assert the FLASH chip select.*/
366 XQspiPs_SetSlaveSelect(&QspiInstance);
367
368
369 FlashReadID();
370
371 FlashQuadEnable(&QspiInstance);
372
373 /* Erase the flash.*/
374 FlashErase(&QspiInstance, TEST_ADDRESS, MAX_DATA);
375 }

```

QSPI_FLASH API

```
*****
/**
*
* This function reads from the serial FLASH connected to the
* QSPI interface.
*
* @param QspiPtr is a pointer to the QSPI driver component to use.
* @param Address contains the address to read data from in the FLASH.
* @param ByteCount contains the number of bytes to read.
* @param Command is the command used to read data from the flash. QSPI
* device supports one of the Read, Fast Read, Dual Read and Fast
* Read commands to read data from the flash.
*
* @return None.
*
* @note None.
*
*****
void FlashRead(XQspiPs *QspiPtr, u32 Address, u32 ByteCount, u8 Command)[]

*****
/**
*
* This function erases the sectors in the serial FLASH connected to the
* QSPI interface.
*
* @param QspiPtr is a pointer to the QSPI driver component to use.
* @param Address contains the address of the first sector which needs to
* be erased.
* @param ByteCount contains the total size to be erased.
*
* @return None.
*
* @note None.
*
*****
void FlashErase(XQspiPs *QspiPtr, u32 Address, u32 ByteCount)[]
```

```
*****
/**
*
* This function reads serial FLASH ID connected to the SPI interface.
*
* @return XST_SUCCESS if read id, otherwise XST_FAILURE.
*
* @note None.
*
*****
int FlashReadID(void)[]

*****
/**
*
* This function enables quad mode in the serial flash connected to the
* SPI interface.
*
* @param QspiPtr is a pointer to the QSPI driver component to use.
*
* @return None.
*
* @note None.
*
*****
void FlashQuadEnable(XQspiPs *QspiPtr)[]
```

QSPI_Write&Read

Write data to QSPI Flash.

```
/*
 * Write the data in the write buffer to the serial FLASH a page at a
 * time, starting from TEST_ADDRESS
 */
for (Page = 0; Page < PAGE_COUNT; Page++) {
    FlashWrite(QspiInstancePtr, (Page * PAGE_SIZE) + TEST_ADDRESS,
               PAGE_SIZE, WRITE_CMD);
}
```

```
/* Flash address to which data is to be written.*/
#define TEST_ADDRESS          0x00055000
#define UNIQUE_VALUE          0x05
/*
```

Read data from QSPI Flash.

```
/*
 * Read the contents of the FLASH from TEST_ADDRESS, using Normal Read
 * command. Change the prescaler as the READ command operates at a
 * lower frequency.
 */
FlashRead(QspiInstancePtr, TEST_ADDRESS, MAX_DATA, READ_CMD);
```

Verify the data

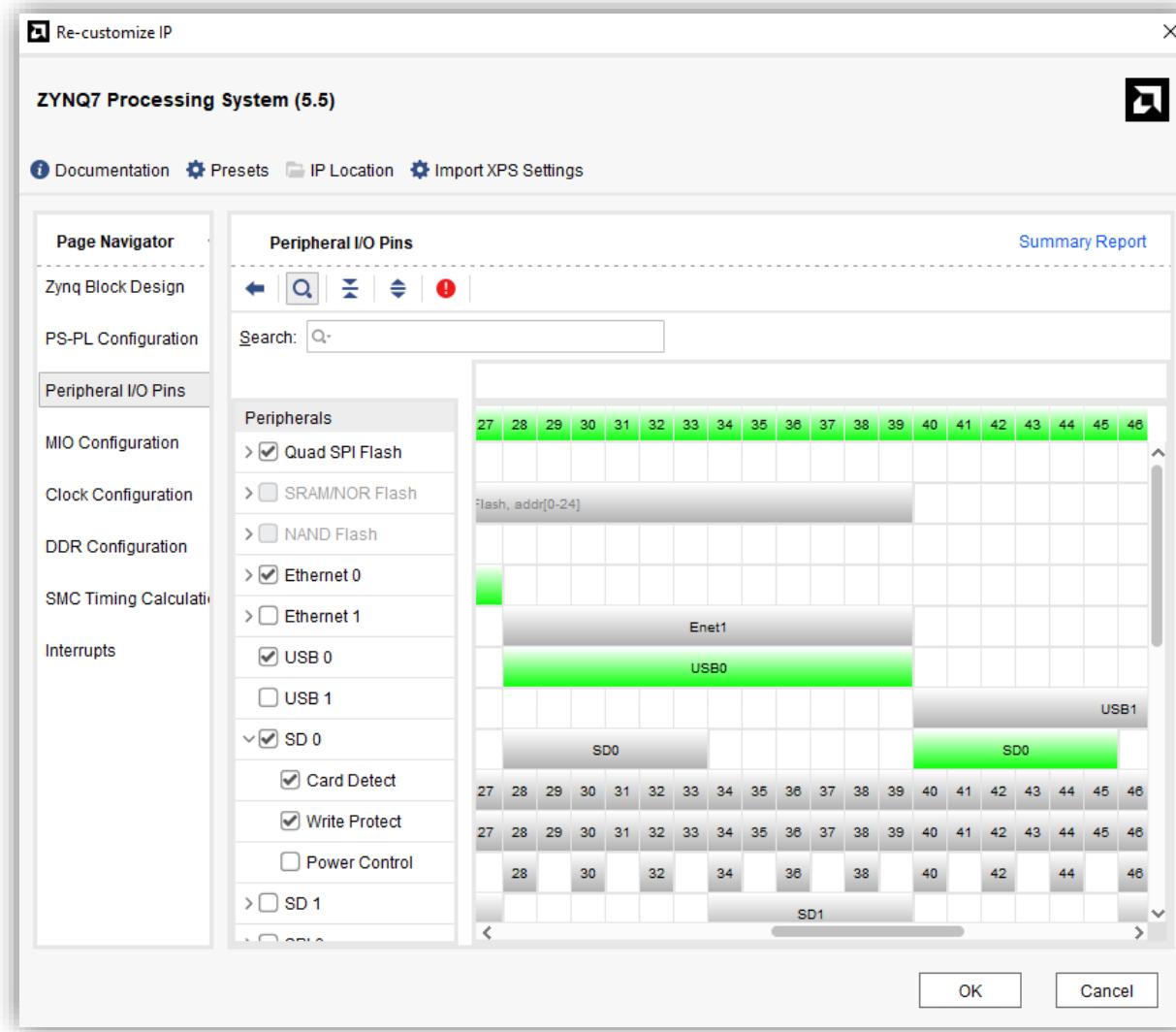
```
/*
 * Setup a pointer to the start of the data that was read into the read
 * buffer and verify the data read is the data that was written
 */
BufferPtr = &ReadBuffer[DATA_OFFSET];

for (UniqueValue = UNIQUE_VALUE, Count = 0; Count < MAX_DATA;
     Count++, UniqueValue++) {
    if (BufferPtr[Count] != (u8)(UniqueValue + Test)) {
        return XST_FAILURE;
    }
}
```

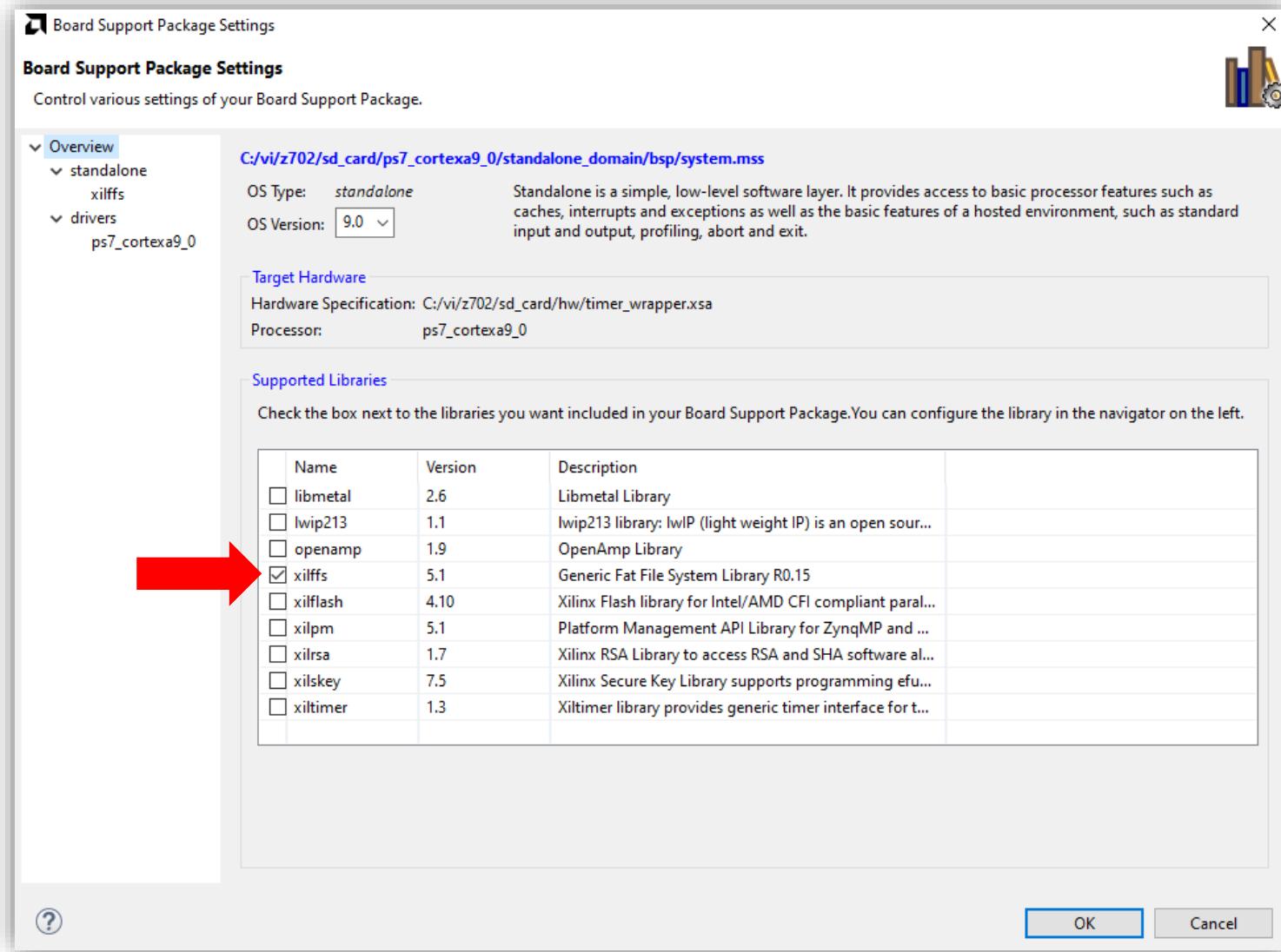
```
QSPI FLASH Polled Example Test
FlashID=0x20 0xBB 0x18
Successfully ran QSPI FLASH Polled Example Test
```

SD Card Read&Write

SD Card_MIO Configuration



FS Library



Application Layer

FATFS

Storage
Interface

RTC



FS Library

Board Support Package Settings

Control various settings of your Board Support Package.

Overview

- standalone
 - xilffs
- drivers
 - ps7_cortexa9_0

Configuration for library: [xilffs](#)

Name	Value	Default	Type	Description
enable_exfat	false	false	boolean	0:Disable exFAT, 1:Enable exFAT(Also Enables LFN)
enable_multi_partition	false	false	boolean	0:Single partition, 1:Enable multiple partition
fs_interface	1	1	integer	Enables file system with selected interface. Enter 1 for SD. Enter 2 for RAM
num_logical_vol	2	2	integer	Number of volumes (logical drives, from 1 to 10) to be used.
read_only	false	false	boolean	Enables the file system in Read_Only mode if true. ZynqMP fsbl will set this to true
set_fs_rpath	0	0	integer	Configures relative path feature (valid values 0 to 2).
use_chmod	false	false	boolean	Enables use of CHMOD functionality for changing attributes (valid only with read_only set to false)
use_lfn	0	0	integer	Enables the Long File Name(LFN) support if non-zero. Disabled by default: 0, LFN with static working buffer: 1, Dynamic working buffer: 2 ...
use_mkfs	true	true	boolean	Disable(0) or Enable(1) f_mkfs function. ZynqMP fsbl will set this to false
use_strfunc	0	0	integer	Enables the string functions (valid values 0 to 2).
use_trim	false	false	boolean	Disable(0) or Enable(1) TRIM function. ZynqMP fsbl will set this to false
word_access	true	true	boolean	Enables word access for misaligned memory access platform
ramfs_size	3145728	3145728	integer	RAM FS size

[?](#) OK Cancel

FS mount

The example in the file system (fs) is designed for dynamic addresses, but for our Zynq, the addresses we read are always the same. Therefore, we can modify it here to use a fixed address for writing.

Example

```
int main (void)
{
    FATFS *fs;      /* Pointer to the filesystem object */

    fs = malloc(sizeof (FATFS)); /* Get work area for the volume */
    f_mount(fs, "", 0);        /* Mount the default drive */
```

```
1 #include <stdio.h>
2 #include "platform.h"
3 #include "xil_printf.h"
4 #include "ff.h"
5
6 int main()
7 {
8     FATFS fs;
9     f_mount(&fs, "", 0);
10}
```

Path option. The string without drive number means the default drive.

Mounting option. 0: Do not mount now (to be mounted on the first access to the volume),
 1: Force mounted the volume to check if it is ready to work.

```
FRESULT f_mount (
    FATFS*   fs,      /* [IN] Filesystem object */
    const TCHAR* path, /* [IN] Logical drive number */
    BYTE      opt     /* [IN] Initialization option */
);
```

```
FRESULT f_unmount (
    const TCHAR* path /* [IN] Logical drive number */
);
```

SD_Write

The f_open function is not only used to open files but can also be modified with parameters to create a file.

mode

Mode flags that specifies the type of access and open method for the file. It is specified by a combination of following flags.

Flags	Meaning
FA_READ	Specifies read access to the file. Data can be read from the file.
FA_WRITE	Specifies write access to the file. Data can be written to the file. Combine with FA_READ for read-write access.
FA_OPEN_EXISTING	Opens a file. The function fails if the file is not existing. (Default)
FA_CREATE_NEW	Creates a new file. The function fails with FR_EXIST if the file is existing.
FA_CREATE_ALWAYS	Creates a new file. If the file is existing, it will be truncated and overwritten.
FA_OPEN_ALWAYS	Opens the file if it is existing. If not, a new file will be created.
FA_OPEN_APPEND	Same as FA_OPEN_ALWAYS except the read/write pointer is set end of the file.

Mode flags in POSIX fopen() function corresponds to FatFs mode flags as follows:

POSIX	FatFs
"r"	FA_READ
"r+"	FA_READ FA_WRITE
"w"	FA_CREATE_ALWAYS FA_WRITE
"w+"	FA_CREATE_ALWAYS FA_WRITE FA_READ
"a"	FA_OPEN_APPEND FA_WRITE
"a+"	FA_OPEN_APPEND FA_WRITE FA_READ
"wx"	FA_CREATE_NEW FA_WRITE
"wx+"	FA_CREATE_NEW FA_WRITE FA_READ

```
#define FILE_NAME "SD_Test"
char user_write_data[50] = "This is Write Data";

void sd_write_data(char wr_data[],u32 wr_len){

    FIL fil;
    UINT bw;

    /* Open(Create) a text file */
    f_open(&fil, FILE_NAME , FA_CREATE_ALWAYS | FA_WRITE | FA_READ);
    /* Set read/write pointer to 0 */
    f_lseek(&fil, 0);
    f_write (&fil,wr_data,wr_len,&bw);
    f_close (&fil);

}

int main()
{
    FATFS fs;
    u32 user_len = 0;
    f_mount(&fs, "", 0);

    user_len = strlen(user_write_data);
    sd_write_data(user_write_data,user_len);
}
```

SD_Write

The f_lseek function moves the file read/write pointer of an open file object.
It can also be used to expand the file size (cluster pre-allocation).

```
FRESULT f_lseek (
    FIL*   fp, /* [IN] File object */
    FSIZE_t ofs /* [IN] Offset of file read/write pointer to be set */
);
```

The f_write writes data to a file.
Expose the variable for easier modification by the user and
use the strlen function to obtain the length of the string.

```
FRESULT f_write (
    FIL* fp,          /* [IN] Pointer to the file object structure */
    const void* buff, /* [IN] Pointer to the data to be written */
    UINT btw,         /* [IN] Number of bytes to write */
    UINT* bw         /* [OUT] Pointer to the variable to return number of bytes written */
);
```

The f_close function closes an open file.

```
FRESULT f_close (
    FIL* fp /* [IN] Pointer to the file object */
);
```

```
#define FILE_NAME "SD_Test"
char user_write_data[50] = "This is Write Data";

void sd_write_data(char wr_data[],u32 wr_len){

    FIL fil;
    UINT bw;

    /* Open(Create) a text file */
    f_open(&fil, FILE_NAME , FA_CREATE_ALWAYS | FA_WRITE | FA_READ);
    /* Set read/write pointer to 0 */
    f_lseek(&fil, 0);
    f_write (&fil,wr_data,wr_len,&bw);
    f_close (&fil);

}

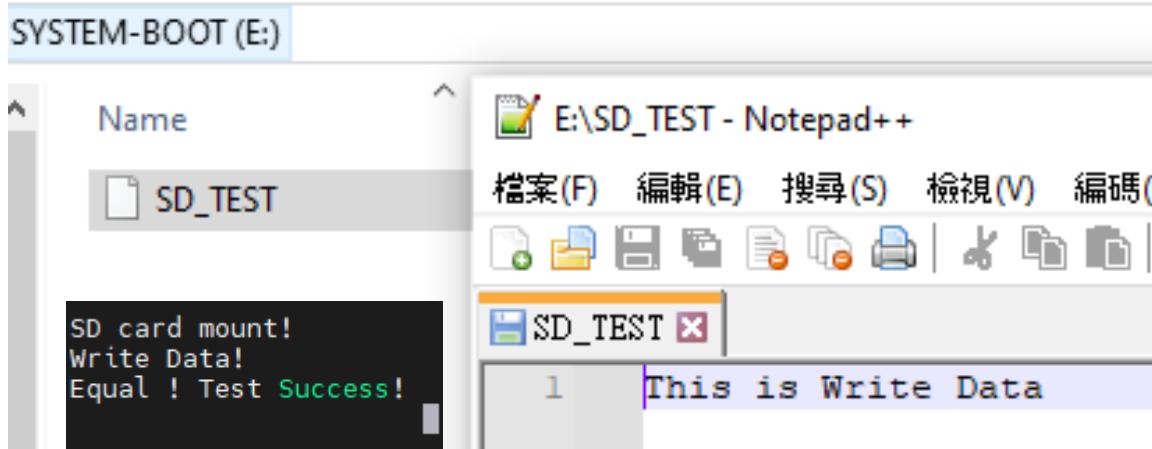
int main()
{
    FATFS fs;
    u32 user_len = 0;
    f_mount(&fs, "", 0);

    user_len = strlen(user_write_data);
    sd_write_data(user_write_data,user_len);
}
```

SD_Read

The f_read function reads data from a file.

```
FRESULT f_read (
    FIL* fp,      /* [IN] File object */
    void* buff,   /* [OUT] Buffer to store read data */
    UINT btr,     /* [IN] Number of bytes to read */
    UINT* br      /* [OUT] Number of bytes read */
);
```



```
void sd_read_data(char rd_data[], u32 rd_len){
    FIL fil;
    UINT br;

    /* Open(Create) a text file */
    f_open(&fil, FILE_NAME, FA_READ);
    /* Set read/write pointer to 0 */
    f_lseek(&fil, 0);
    f_read(&fil, rd_data, rd_len, &br);
    f_close(&fil);
}

int main()
{
    u32 user_len = 0;
    FATFS fs;
    char user_read_data[50] = "";
    f_mount(&fs, "", 1);
    xil_printf("SD card mount! \r\n");

    user_len = strlen(user_write_data);
    sd_write_data(user_write_data, user_len);
    xil_printf("Write Data! \r\n");

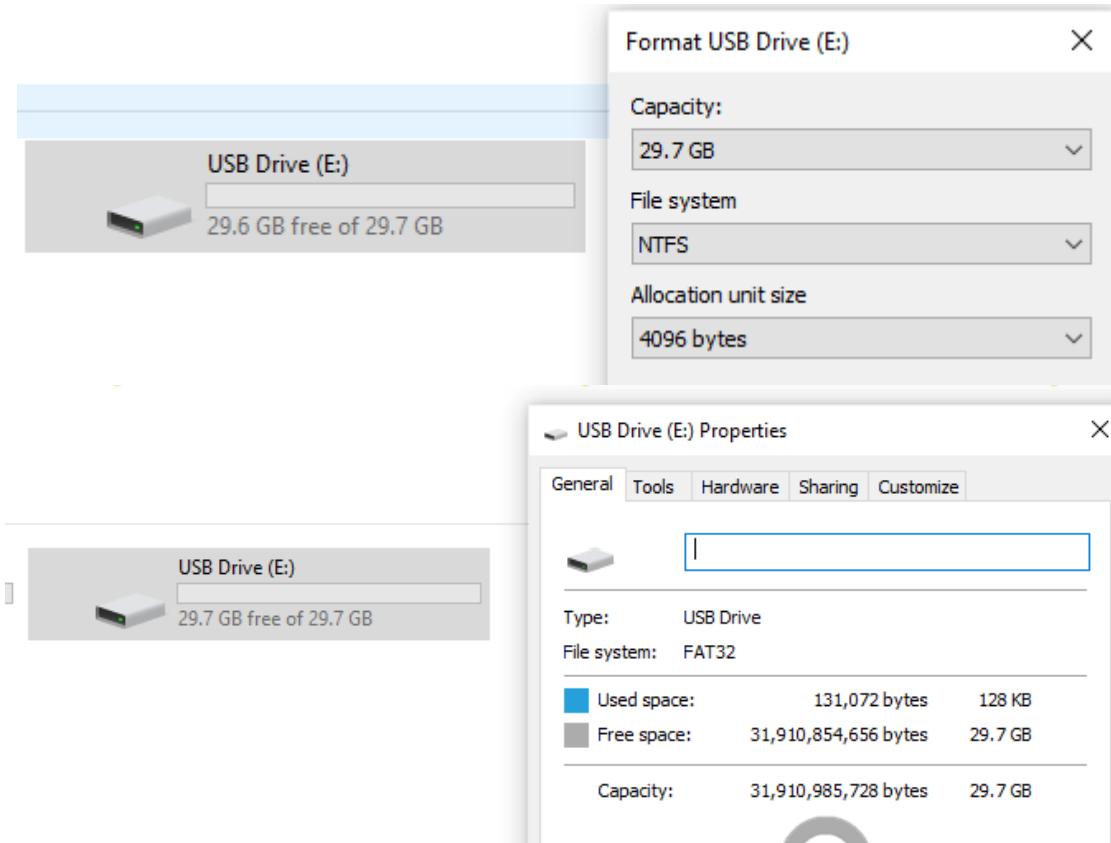
    sd_read_data(user_read_data, user_len);

    if(strcmp(user_write_data, user_read_data) == 0)
        xil_printf("Equal ! Test Success! \n");
    else
        xil_printf("Not Equal ! Test Failed! \n");

    return 0;
}
```

SD_Format

The f_mkfs function creates an FAT/exFAT volume on the logical drive.



```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "ff.h"

#define FILE_NAME "SD_Test_Format"
char user_write_data[50] = "This is Write Data";
FATFS fs;

void sd_mount(){

    HRESULT status;
    BYTE work[FF_MAX_SS]; /* Work area (larger is better for processing time) */
    status = f_mount(&fs, "", 1);
    if (status != FR_OK){
        xil_printf("SD card format! \r\n");
        f_mkfs("", 0, work, sizeof work);
        f_mount(&fs, "", 1);
    }
    return 0;
}

HRESULT f_mkfs (
    const TCHAR* path, /* [IN] Logical drive number */
    const MKFS_PARM* opt,/* [IN] Format options */
    void* work, /* [-] Working buffer */
    UINT len /* [IN] Size of working buffer */
);
```

SD_RW

```
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "ff.h"

#define FILE_NAME "SD_Test_Format"
char user_write_data[50] = "This is Write Data";
FATFS fs;

void sd_mount(){
    FRESULT status;
    BYTE work[FF_MAX_SS]; /* Work area (larger is better for processing time) */
    status = f_mount(&fs, "", 1);
    if (status != FR_OK){
        xil_printf("SD card format! \r\n");
        f_mkfs("", 0, work, sizeof work);
        f_mount(&fs, "", 1);
    }
    return 0;
}
```

SD_RW

```
void sd_write_data(char wr_data[], u32 wr_len) {  
  
    FIL fil;  
    UINT bw;  
  
    /* Open(Create) a text file */  
    f_open(&fil, FILE_NAME, FA_CREATE_ALWAYS | FA_WRITE | FA_READ);  
    /* Set read/write pointer to 0 */  
    f_lseek(&fil, 0);  
    f_write(&fil, wr_data, wr_len, &bw);  
    f_close(&fil);  
  
}  
void sd_read_data(char rd_data[], u32 rd_len) {  
  
    FIL fil;  
    UINT br;  
  
    /* Open(Create) a text file */  
    f_open(&fil, FILE_NAME, FA_READ);  
    /* Set read/write pointer to 0 */  
    f_lseek(&fil, 0);  
    f_read(&fil, rd_data, rd_len, &br);  
    f_close(&fil);  
  
}
```

SD_RW

```
int main(){
    u32 user_len = 0;
    char user_read_data[50] = "";

    sd_mount();
    xil_printf("SD card mount! \r\n");

    user_len = strlen(user_write_data);
    sd_write_data(user_write_data,user_len);
    xil_printf("Write Data! \r\n");

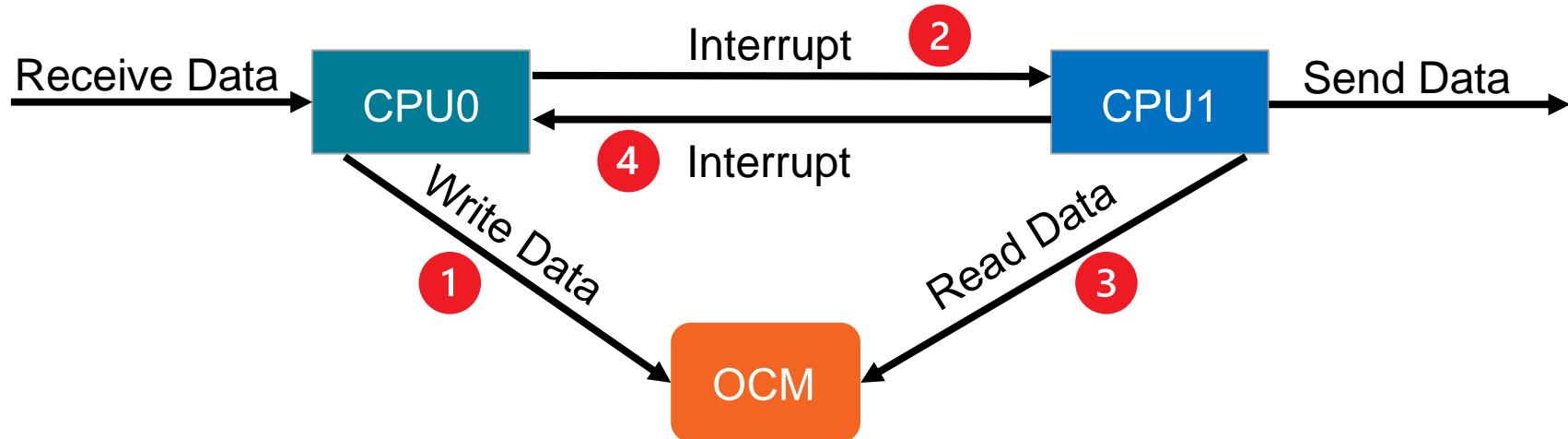
    sd_read_data(user_read_data,user_len);

    if(strcmp(user_write_data,user_read_data) == 0)
        xil_printf("Equal ! Test Success! \n");
    else
        xil_printf("Not Equal ! Test Failed! \n");
return 0;
}
```

ZYNQ dual-core AMP

Lab Flow

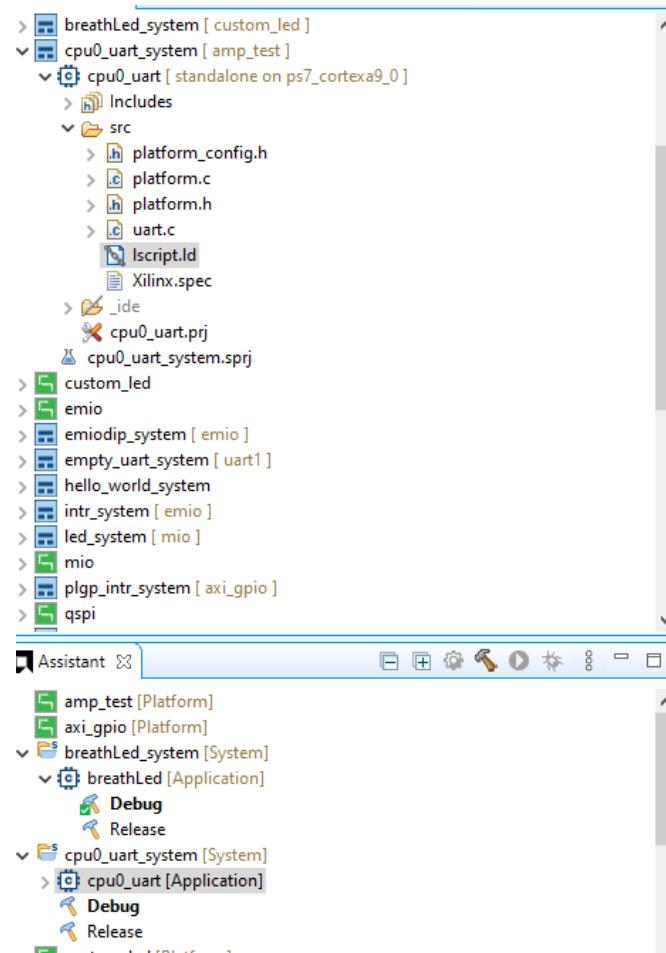
In this dual-core communication, data sharing between the two CPUs will occur through OCM. Additionally, software interrupts will be utilized to enable each core to access OCM separately.



SD_Write

Allocate DDR memory space for the two CPUs.

Since the two CPUs share OCM, there is no need for separate allocation.



CPU0

Linker Script: lscript.ld

A linker script is used to control where different sections of an executable are placed in memory. In this page, you can define new memory regions, and change the assignment of sections to memory regions.

Available Memory Regions

Name	Base Address	Size
ps7_ddr_0	0x100000	0x3FF00000
ps7_qspi_linear_0	0xFC000000	0x1000000
ps7_ram_0	0x0	0x30000
ps7_ram_1	0xFFFFF0000	0xFE00

Stack and Heap Sizes

Stack Size: 0x2000
Heap Size: 0x2000

Section to Memory Region Mapping

Section Name	Memory Region
.text	ps7_ddr_0
.init	ps7_ddr_0
.fini	ps7_ddr_0
.rodata	ps7_ddr_0
.rodata1	ps7_ddr_0
.sdata2	ps7_ddr_0
.sbss2	ps7_ddr_0
.data	ps7_ddr_0
.data1	ps7_ddr_0
.got	ps7_ddr_0
.note.gnu.build-id	ps7_ddr_0
.ctors	ps7_ddr_0
.dtors	ps7_ddr_0
.fixup	ps7_ddr_0
.eh_frame	ps7_ddr_0

CPU1

Linker Script: lscript.ld

A linker script is used to control where different sections of an executable are placed in memory. In this page, you can define new memory regions, and change the assignment of sections to memory regions.

Available Memory Regions

Name	Base Address	Size
ps7_ddr_0	0x100000	0xOFF00000
ps7_qspi_linear_0	0xFC000000	0x1000000
ps7_ram_0	0x0	0x30000
ps7_ram_1	0xFFFFF0000	0xFE00

Stack and Heap Sizes

Stack Size: 0x2000
Heap Size: 0x2000

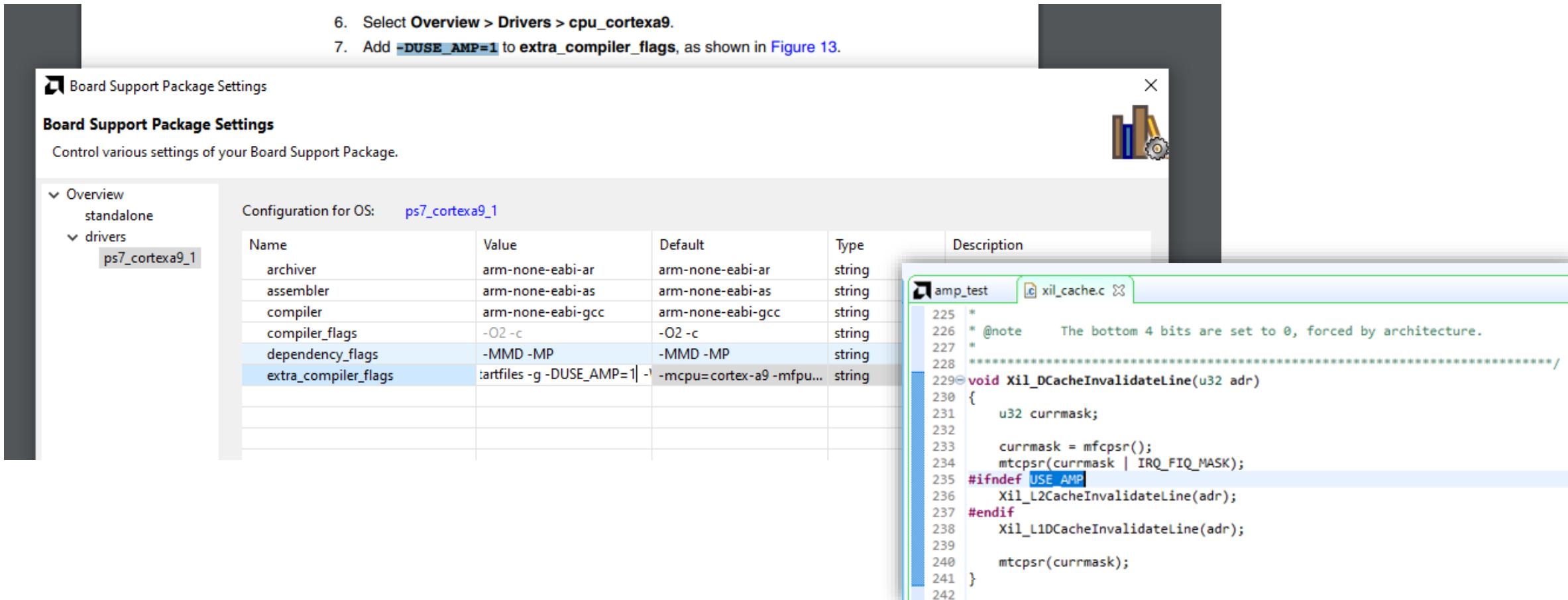
Section to Memory Region Mapping

Section Name	Memory Region
.text	ps7_ddr_0
.init	ps7_ddr_0
.fini	ps7_ddr_0
.rodata	ps7_ddr_0
.rodata1	ps7_ddr_0
.sdata2	ps7_ddr_0
.sbss2	ps7_ddr_0
.data	ps7_ddr_0
.data1	ps7_ddr_0
.got	ps7_ddr_0
.note.gnu.build-id	ps7_ddr_0
.ctors	ps7_ddr_0
.dtors	ps7_ddr_0
.fixup	ps7_ddr_0
.eh_frame	ps7_ddr_0

SD_Write

Modify the library settings for CPU1 domain. The effect is to enable the shared cache setting.

6. Select Overview > Drivers > `cpu_cortexa9`.
7. Add `-DUSE_AMP=1` to `extra_compiler_flags`, as shown in Figure 13.



Name	Value	Default	Type	Description
archiver	arm-none-eabi-ar	arm-none-eabi-ar	string	
assembler	arm-none-eabi-as	arm-none-eabi-as	string	
compiler	arm-none-eabi-gcc	arm-none-eabi-gcc	string	
compiler_flags	<code>-O2 -c</code>	<code>-O2 -c</code>	string	
dependency_flags	<code>-MMD -MP</code>	<code>-MMD -MP</code>	string	
<code>extra_compiler_flags</code>	<code>tartfiles -g -DUSE_AMP=1</code>	<code>-mcpu=cortex-a9 -mfpu...</code>	string	

`xil_cache.c`

```
225 * @note The bottom 4 bits are set to 0, forced by architecture.
226 ****
227 void Xil_DCacheInvalidateLine(u32 adr)
228 {
229     u32 currmask;
230
231     currmask = mfcpsr();
232     mtcpsr(currmask | IRQ_FIQ_MASK);
233 #ifndef USE_AMP
234     Xil_L2CacheInvalidateLine(adr);
235 #endif
236     Xil_L1DCacheInvalidateLine(adr);
237
238     mtcpsr(currmask);
239 }
240
241 }
242 }
```

CPU0_Uart

Receive data from UART and transfer it to the shared OCM. At the same time, wake up CPU1 to access the data.

Because we wake up CPU1 through CPU0 to perform tasks,
we need to define the addresses related to CPU1 in CPU0 first.

```
#include "xparameters.h"
#include "xuartps.h"
#include "xil_exception.h"
#include "xscugic.h"
#include "xil_printf.h"
#include "xil_mm.h"
#include "stdio.h"

#define SHARE_BASE          0xfffff0000           //OCM First Address
#define CPU1_COPY_ADDR      0xfffffffff0          //Storage CPU1 Start Address(fixed)
#define CPU1_START_ADDR     0x100000000          //CPU1 Start Address

#define INTC_DEVICE_ID      XPAR_SCUGIC_SINGLE_DEVICE_ID //intr ID
#define CPU1_ID              XSCUGIC_SPI_CPU1_MASK        //CPU1 ID
#define SOFT_INTR_ID_TO_CPU0 0                         //intr to CPU0 ID
#define SOFT_INTR_ID_TO_CPU1 1                         //intr to CPU1 ID

#define sev()                __asm__ ("sev")           //Awake CPU1 & Run Application(Send Event)

void start_cpu1();
void cpu0_intr_init(XScuGic *intc_ptr);
void soft_intr_handler(void *CallbackRef);

XScuGic Intc;
int rec_freq_flag = 0;
int freq_gear;
```



CPU0_Main

```

int main()
{
    // disable cache access
    Xil_SetTlbAttributes(SHARE_BASE,0x14de2);
    Xil_SetTlbAttributes(CPU1_COPY_ADDR,0x14de2);

    start_cpu1();
    cpu0_intr_init(&Intc);
    while(1){
        if(rec_freq_flag == 0){
            xil_printf("CPU0 : Input 1~5 to change freqenct\r\n");
            scanf("%d",&freq_gear);
            if(freq_gear > 1 && freq_gear <= 5){
                xil_printf("CPU0 : Input is %d\r\n" , &freq_gear);
                Xil_Out8(SHARE_BASE,freq_gear);
                XScuGic_SoftwareIntr(&Intc,SOFT_INTR_ID_TO_CPU1,CPU1_ID);
                rec_freq_flag = 1;
            }
            else{
                xil_printf("CPU0 : input error");
            }
        }
    }
    return 0 ;
}

```

Temporarily disable access to OCM and CPU1 for cache.
(mmu.h)

Refer to the Cortex-A9 Memory Management Unit (MMU)
register values table.(e.g. 0x14de2)

Writing the 8 bit Value to the the specified address.

CPU0

```
void start_cpu1()
{
    Xil_Out32(CPU1_COPY_ADDR,CPU1_START_ADDR);
    dmb(); → Pre-check if there are other instructions in memory.
    sev(); → Awake CPU1 & Run Application(Send Event)
}

void cpu0_intr_init(XScuGic *intc_ptr){

    XScuGic_Config *intc_cfg_ptr;
    intc_cfg_ptr = XScuGic_LookupConfig(INTC_DEVICE_ID);
    XScuGic_CfgInitialize(intc_ptr, intc_cfg_ptr,intc_cfg_ptr->CpuBaseAddress);

    Xil_ExceptionInit();
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
        (Xil_ExceptionHandler)XScuGic_InterruptHandler,
        intc_ptr);
    Xil_ExceptionEnable();

    XScuGic_Connect(intc_ptr, SOFT_INTR_ID_TO_CPU0,
        (Xil_ExceptionHandler)soft_intr_handler,
        (void *)intc_ptr);

    XScuGic_Enable(intc_ptr, SOFT_INTR_ID_TO_CPU0);

}

void soft_intr_handler(void *CallbackRef){

    xil_printf("CPU0 : soft Intr from CPU1\r\n");
    rec_freq_flag = 0;
}
```

Pre-check if there are other instructions in memory.
Awake CPU1 & Run Application(Send Event)

CPU1_LED

Read data from OCM to modify the frequency of the breathing light.

```
#include "xparameters.h"
#include "xil_exception.h"
#include "xscugic.h"
#include "xil_printf.h"
#include "xil_mmu.h"
#include "stdio.h"
#include "axiled.h"

#define SHARE_BASE      0xfffff0000          //OCM First Address

#define INTC_DEVICE_ID  XPAR_SCUGIC_SINGLE_DEVICE_ID //intr ID
#define CPU0_ID         XSCUGIC_SPI_CPU0_MASK        //CPU0 ID
#define SOFT_INTR_ID_TO_CPU0 0                  //intr to CPU0 ID
#define SOFT_INTR_ID_TO_CPU1 1                  //intr to CPU1 ID

#define LED_IP_BASEADDR XPAR_AXILED_0_DEVICE_ID
#define LED_IP_REG0    AXILED_S00_AXI_SLV_REG0_OFFSET
#define LED_IP_REG1    AXILED_S00_AXI_SLV_REG1_OFFSET

void cpu1_intr_init(XScuGic *intc_ptr);
void soft_intr_handler(void *CallbackRef);

XScuGic Intc;
int soft_intr_flag = 0;
int freq_gear;
```

```
int main()
{
    int freq_step = 0;

    Xil_SetTlbAttributes(SHARE_BASE,0x14de2);
    cpu1_intr_init(&Intc);

    AXILED_mWriteReg(LED_IP_BASEADDR, LED_IP_REG1, 1);

    while(1){
        if(soft_intr_flag){
            freq_gear = Xil_In8(SHARE_BASE); ----->
            xil_printf("CPU1 : Receive Data is %d",freq_gear);
            switch(freq_gear){
                case 1 : freq_step = 20;break;
                case 2 : freq_step = 50;break;
                case 3 : freq_step = 100;break;
                case 4 : freq_step = 200;break;
                case 5 : freq_step = 500;break;
                default: freq_step = 50;break;
            }
            AXILED_mWriteReg(LED_IP_BASEADDR, LED_IP_REG1,(0x80000000|freq_step));
            XScuGic_SoftwareIntr(&Intc,SOFT_INTR_ID_TO_CPU0,CPU0_ID);
            soft_intr_flag = 0;
        }
    }
    return 0;
}
```

Read Data from OCM.

CPU1_LED

Almost the same as CPU0.

```
void cpu1_intr_init(XScuGic *intc_ptr){

    XScuGic_Config *intc_cfg_ptr;
    intc_cfg_ptr = XScuGic_LookupConfig(INTC_DEVICE_ID);
    XScuGic_CfgInitialize(intc_ptr, intc_cfg_ptr,intc_cfg_ptr->CpuBaseAddress);

    Xil_ExceptionInit();
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
                                (Xil_ExceptionHandler)XScuGic_InterruptHandler,
                                intc_ptr);
    Xil_ExceptionEnable();

    XScuGic_Connect(intc_ptr, SOFT_INTR_ID_TO_CPU1,
                    (Xil_ExceptionHandler)soft_intr_handler,
                    (void *)intc_ptr);

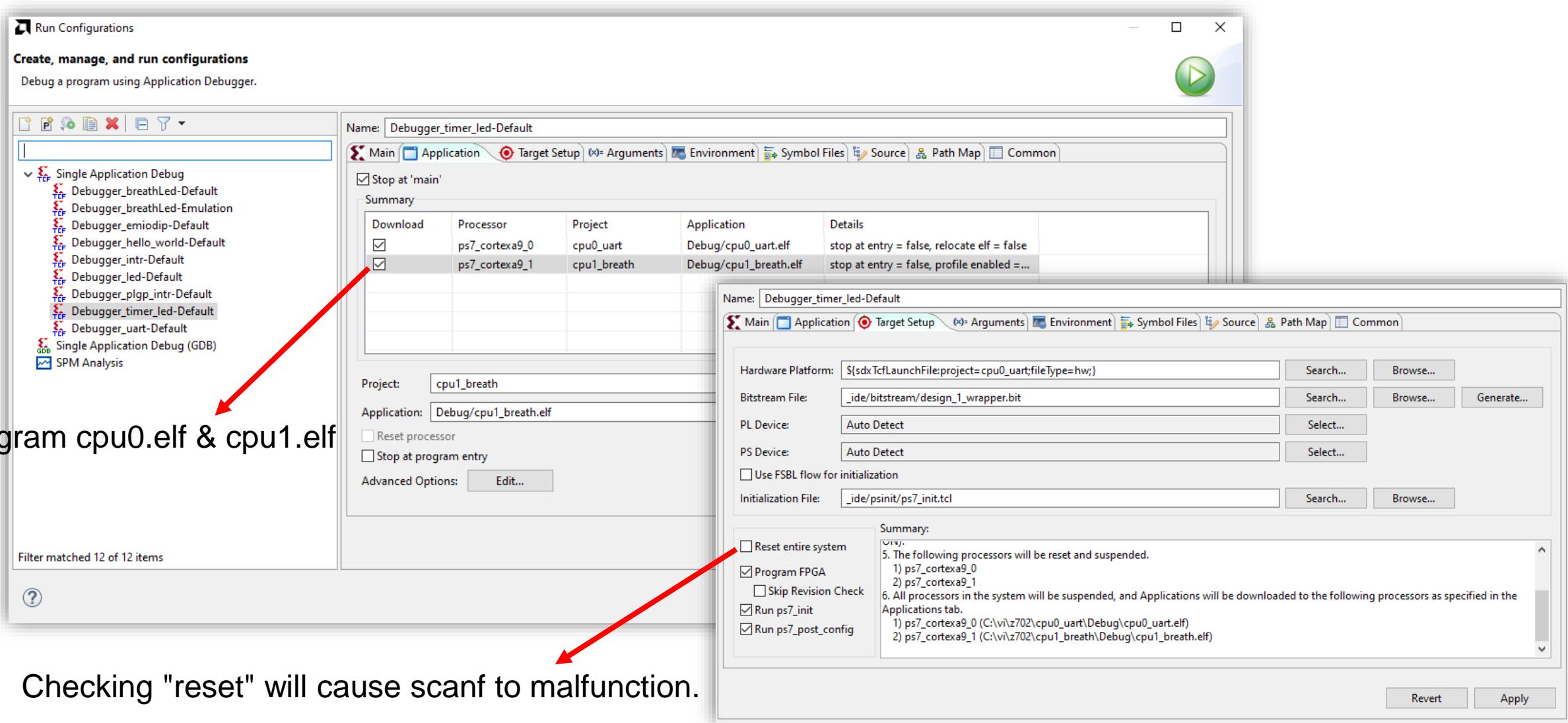
    XScuGic_Enable(intc_ptr, SOFT_INTR_ID_TO_CPU1);

}

void soft_intr_handler(void *CallbackRef){

    xil_printf("CPU1 : soft Intr from CPU0\r\n");
    soft_intr_flag = 0;
}
```

Run_configuration



Create FSBL to Boot

As we modified the default CPU0 address, we need to generate another FSBL file to boot.

New Application Project

Application Project Details

Specify the application project name and its system project properties

Application project name:

System Project

Create a new system project for the application or select an existing one from the workspace

Select a system project

- cpu0_uart_system
- cpu1_breath_system
- xuartps_intr_example_2_system
- + Create new...

System project details

System project name:

Target processor

Select target processor for the Application project.

Processor	Associated applications
ps7_cortexa9_0	FSBL
ps7_cortexa9_1	

Show all processors in the hardware specification

< Back Finish Cancel

New Application Project

Templates

This application requires xilfss library in the Board Support Package. You can go back to the previous pages to select a different platform and domain or create a new one with a suitable hardware and software.

Available Templates:

Find:

Zynq FSBL

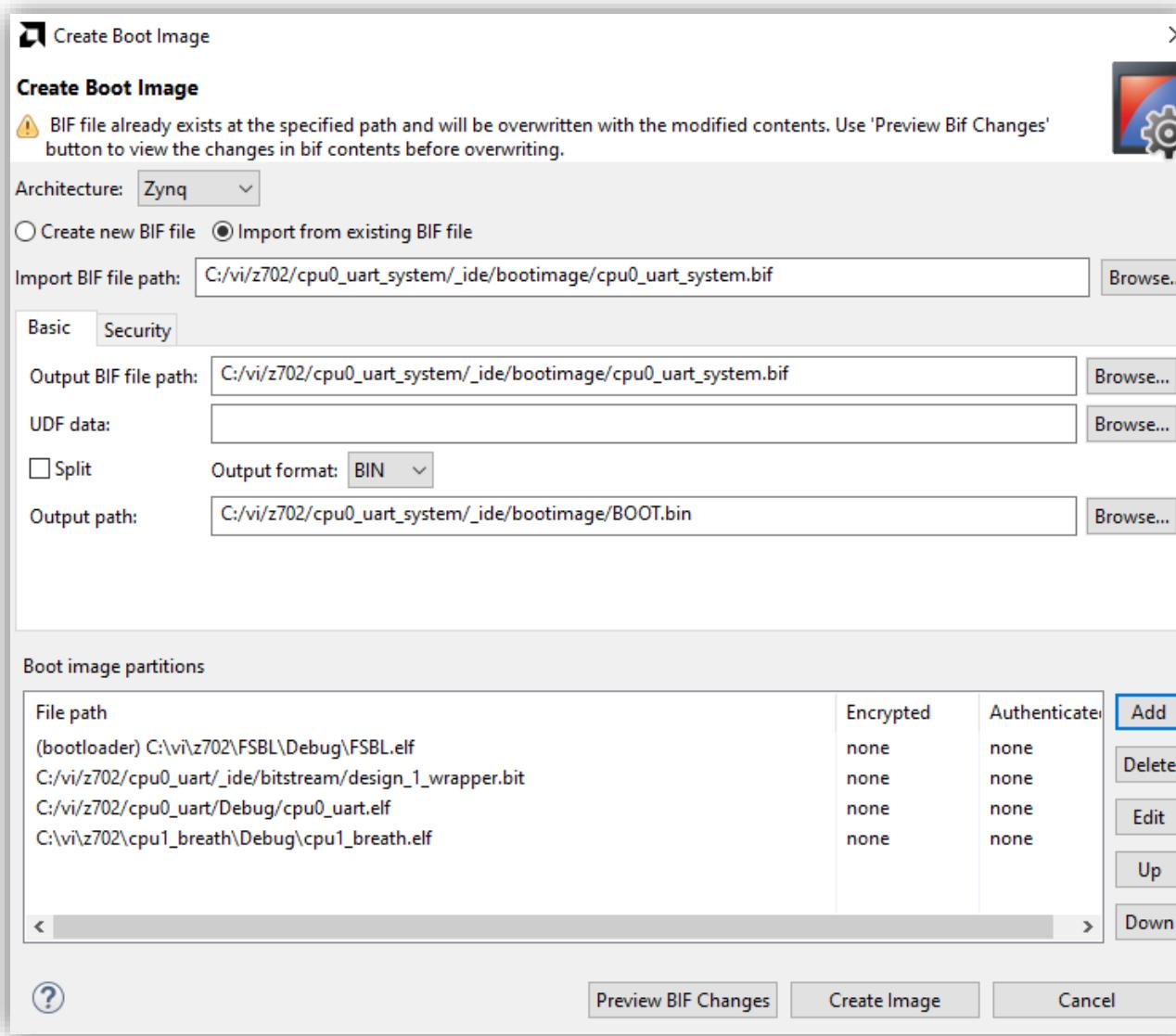
First Stage Bootloader (FSBL) for Zynq. The FSBL configures the FPGA with HW bit stream (if it exists) and loads the Operating System (OS) Image or Standalone (SA) Image or 2nd Stage Boot Loader image from the non-volatile memory (NAND/NOR/QSPI) to RAM (DDR) and starts executing it. It supports multiple partitions, and each partition can be a code image or a bit stream.

Embedded software development templates

- Dhrystone
- Empty Application (C++)
- Empty Application(C)
- Hello World
- IwIP Echo Server
- IwIP TCP Perf Client
- IwIP TCP Perf Server
- IwIP UDP Perf Client
- IwIP UDP Perf Server
- Memory Tests
- OpenAMP echo-test
- OpenAMP matrix multiplication Demo
- OpenAMP RPC Demo
- Peripheral Tests
- RSA Authentication App
- Zynq DRAM tests
- Zynq FSBL**

< Back Finish Cancel

Create Boot Image



The order of the files below must be consistent
(related to the boot order).

CPU0

```

#include "xparameters.h"
#include "xuartbps.h"
#include "xil_exception.h"
#include "xscugic.h"
#include "xil_printf.h"
#include "xil_mmu.h"
#include "stdio.h"

#define SHARE_BASE 0xfffff0000 //OCM First Address
#define CPU1_COPY_ADDR 0xfffffffff0 //Storage CPU1 Start Address(fixed)
#define CPU1_START_ADDR 0x10000000 //CPU1 Start Address

#define INTC_DEVICE_ID XPAR_SCUGIC_SINGLE_DEVICE_ID //intr ID
#define CPU1_ID XSCUGIC_SPI_CPU1_MASK //CPU1 ID
#define SOFT_INTR_ID_TO_CPU0 0 //intr to CPU0 ID
#define SOFT_INTR_ID_TO_CPU1 1 //intr to CPU1 ID

#define sev() __asm__ ("sev") //Awake CPU1 & Run Application(Send Event)

void start_cpu1();
void cpu0_intr_init(XScuGic *intc_ptr);
void soft_intr_handler(void *CallbackRef);

XScuGic Intc;
int rec_freq_flag = 0;
int freq_gear;

```

CPU0

```

void start_cpu1() {
    Xil_Out32(CPU1_COPY_ADDR,CPU1_START_ADDR) ;
    dmb() ;
    sev() ;
}

void cpu0_intr_init(XScuGic *intc_ptr) {
    XScuGic_Config *intc_cfg_ptr;
    intc_cfg_ptr = XScuGic_LookupConfig(INTC_DEVICE_ID);
    XScuGic_CfgInitialize(intc_ptr, intc_cfg_ptr,intc_cfg_ptr->CpuBaseAddress);

    Xil_ExceptionInit();
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
        (Xil_ExceptionHandler)XScuGic_InterruptHandler,intc_ptr);
    Xil_ExceptionEnable();

    XScuGic_Connect(intc_ptr, SOFT_INTR_ID_TO_CPU0,
        (Xil_ExceptionHandler)soft_intr_handler,(void *)intc_ptr);

    XScuGic_Enable(intc_ptr, SOFT_INTR_ID_TO_CPU0);
}

void soft_intr_handler(void *CallbackRef) {

    xil_printf("CPU0 : soft Intr from CPU1\r\n");
    rec_freq_flag = 0;
}

```

CPU1

```
#include "xparameters.h"
#include "xil_exception.h"
#include "xscugic.h"
#include "xil_printf.h"
#include "xil_mmu.h"
#include "stdio.h"
#include "axiled.h"

#define SHARE_BASE 0xfffff0000 //OCM First Address

#define INTC_DEVICE_ID XPAR_SCUGIC_SINGLE_DEVICE_ID //intr ID
#define CPU0_ID XSCUGIC_SPI_CPU0_MASK //CPU0 ID
#define SOFT_INTR_ID_TO_CPU0 0 //intr to CPU0 ID
#define SOFT_INTR_ID_TO_CPU1 1 //intr to CPU1 ID

#define LED_IP_BASEADDR_XPAR_AXILED_0_DEVICE_ID
#define LED_IP_REG0 AXILED_S00_AXI_SLV_REG0_OFFSET
#define LED_IP_REG1 AXILED_S00_AXI_SLV_REG1_OFFSET

void cpu1_intr_init(XScuGic *intc_ptr);
void soft_intr_handler(void *CallbackRef);

XScuGic Intc;
int soft_intr_flag = 0;
int freq_gear;
```

CPU1

```
int main(){
    int freq_step = 0;

Xil_SetTlbAttributes(SHARE_BASE, 0x14de2);
cpu1_intr_init(&Intc);

AXILED_mWriteReg(LED_IP_BASEADDR, LED_IP_REG1, 1);

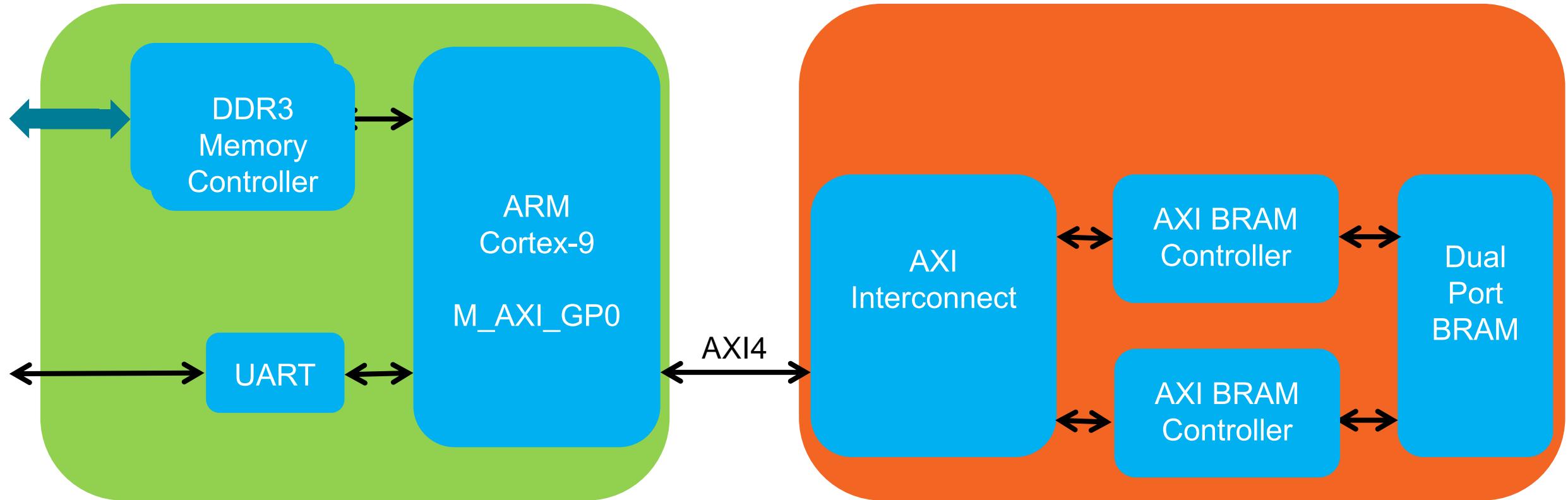
while(1){
    if(soft_intr_flag){
        freq_gear = Xil_In8(SHARE_BASE);
        xil_printf("CPU1 : Receive Data is %d", freq_gear);
        switch(freq_gear){
            case 1 : freq_step = 20;break;
            case 2 : freq_step = 50;break;
            case 3 : freq_step = 100;break;
            case 4 : freq_step = 200;break;
            case 5 : freq_step = 500;break;
            default: freq_step = 50;break;
        }
        AXILED_mWriteReg(LED_IP_BASEADDR, LED_IP_REG1, (0x80000000|freq_step));
        XScuGic_SoftwareIntr(&Intc, SOFT_INTR_ID_TO_CPU0, CPU0_ID);
        soft_intr_flag = 0;
    }
    return 0;
}
```

CPU1

```
void cpu1_intr_init(XScuGic *intc_ptr){  
  
    XScuGic_Config *intc_cfg_ptr;  
    intc_cfg_ptr = XScuGic_LookupConfig(INTC_DEVICE_ID);  
    XScuGic_CfgInitialize(intc_ptr, intc_cfg_ptr,intc_cfg_ptr->CpuBaseAddress);  
  
    Xil_ExceptionInit();  
    Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,  
        (Xil_ExceptionHandler)XScuGic_InterruptHandler,intc_ptr);  
    Xil_ExceptionEnable();  
  
    XScuGic_Connect(intc_ptr, SOFT_INTR_ID_TO_CPU1,  
        (Xil_ExceptionHandler)soft_intr_handler,(void *)intc_ptr);  
  
    XScuGic_Enable(intc_ptr, SOFT_INTR_ID_TO_CPU1);  
  
}  
  
void soft_intr_handler(void *CallbackRef){  
  
    xil_printf("CPU1 : soft Intr from CPU0\r\n");  
    soft_intr_flag = 0;  
}
```

PS to PL Data Transfer

Lab Architecture



ZYNQ IP Core Configuration

The screenshot shows the ZYNQ IP Core Configuration interface for the ZYNQ7 Processing System (5.5). The main window title is "PS-PL Configuration".

PS-PL Configuration Tab:

- Search:** A search bar at the top.
- Name:** A column header for the table.
- Select:** A column header for the table.
- Description:** A column header for the table.
- AXI Non Secure Enablement:** A section expanded to show:
 - GP Master AXI Interface:**
 - M AXI GP0 interface:** Enables General purpose AXI master interface 0
 - M AXI GP1 interface:** Enables General purpose AXI master interface 1
 - GP Slave AXI Interface:**
 - HP Slave AXI Interface:**
 - ACP Slave AXI Interface:**
 - DMA Controller:**
 - PS-PL Cross Trigger interface:** Enables PL cross trigger interface

Clock Configuration Tab:

Component	Clock Source	Requested Freq...	Actual Frequency(...)	Range(MHz)
> Processor/Memory Clocks				
> IO Peripheral Clocks				
PL Fabric Clocks				
> FCLK_CLK0	IO PLL	50	50.000000	0.100000 : 250.000000
> FCLK_CLK1	IO PLL	50	10.000000	0.100000 : 250.000000
> FCLK_CLK2	IO PLL	50	10.000000	0.100000 : 250.000000
> FCLK_CLK3	IO PLL	50	10.000000	0.100000 : 250.000000
> System Debug Clocks				
> Timers				

Buttons: OK and Cancel.

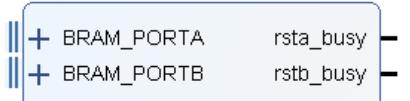
BRAM Configuration

Block Memory Generator (8.4)

[Documentation](#) [IP Location](#)

IP Symbol **Power Estimation**

Show disabled ports



Component Name: blk_mem_gen_0

Basic **Port A Options** **Port B Options** **Other Options** **Summary**

Mode: BRAM Controller Generate address interface with 32 bits
Memory Type: True Dual Port RAM Common Clock

ECC Options:
ECC Type: No ECC
 Error Injection Pins: Single Bit Error Injection

Write Enable:
 Byte Write Enable
Byte Size (bits): 8

Algorithm Options:
 Defines the algorithm used to concatenate the block RAM primitives.
 Refer datasheet for more information.
Algorithm: Minimum Area Primitive: 8lx2

Summary

Load Init File

Coe File: no_coe_file_loaded [Browse](#) [Edit](#)

Mem File: no_mem_loaded [Browse](#) [Edit](#)

Fill Remaining Memory Locations
Remaining Memory Locations (Hex): 0

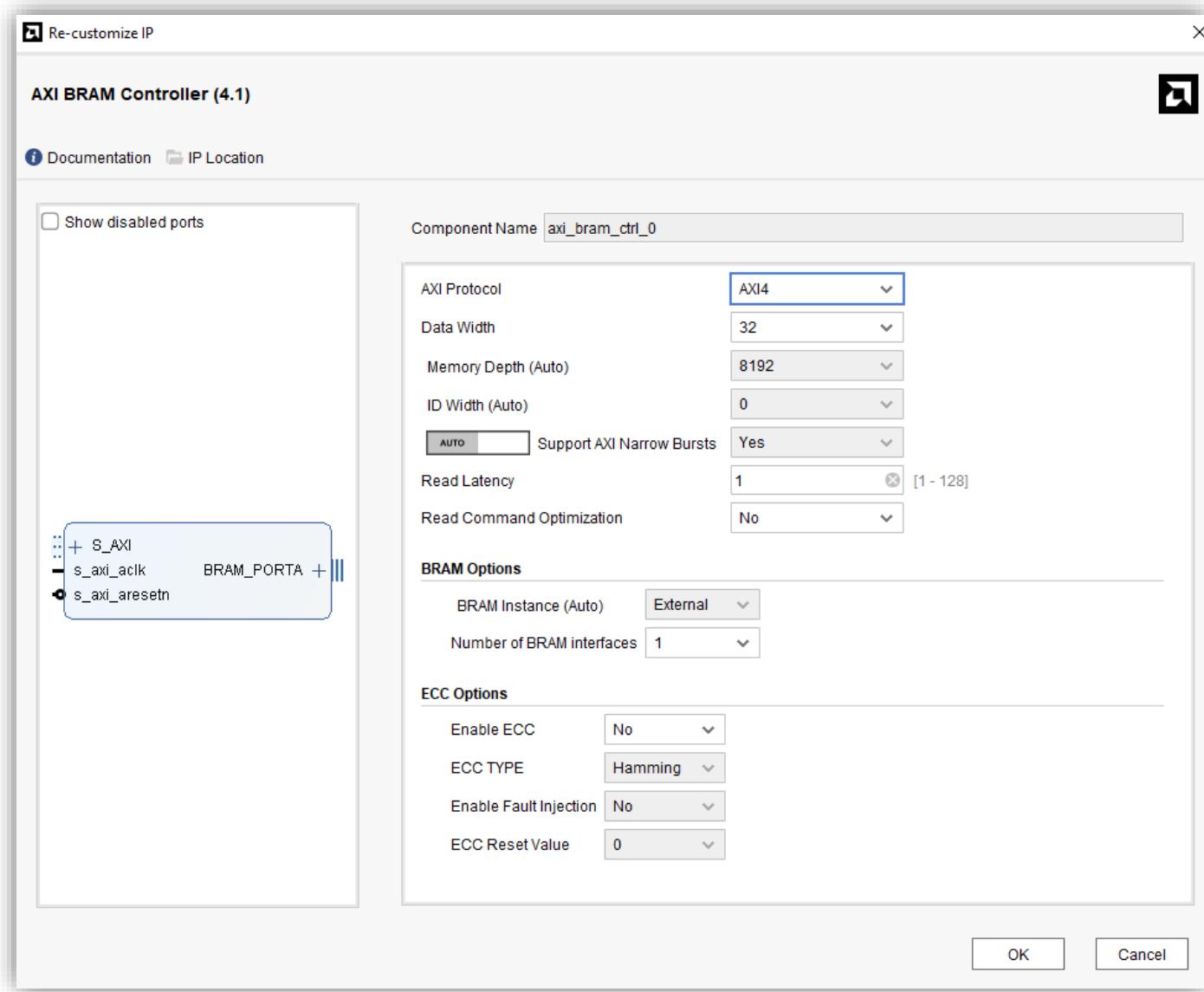
Structural/UniSim Simulation Model Options:
 Defines the type of warnings and outputs are generated when a read-write or write-write collision occurs.
Collision Warnings: All

Behavioral Simulation Model Options:
 Disable Collision Warnings Disable Out of Range Warnings

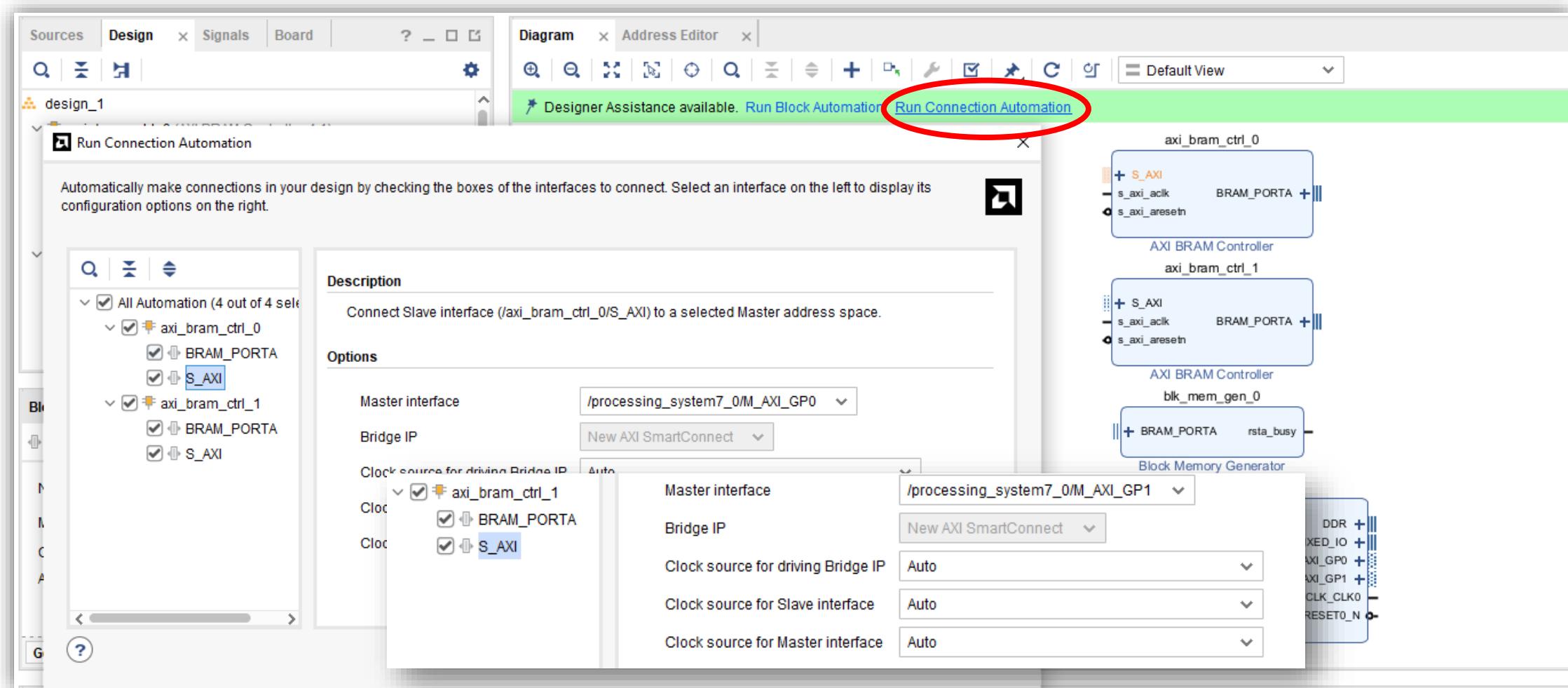
Safety logic to minimize BRAM data corruption:
 Enable Safety Circuit

OK **Cancel**

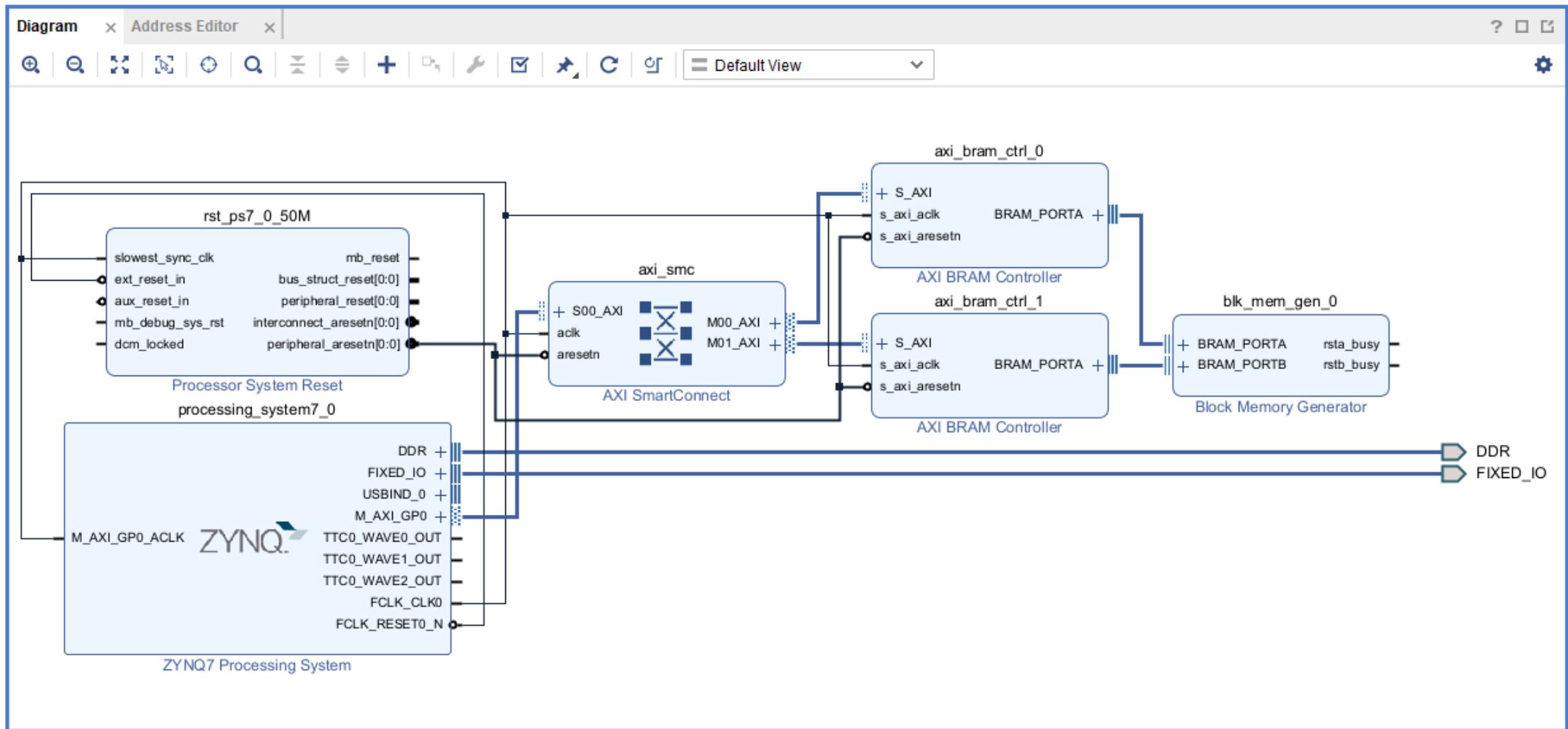
BRAM Controller



Run Connection Automation



Block Design



Address Editor

Diagram x Address Editor x

Assigned (1) Unassigned (0) Excluded (1) Hide All

Name	Interface	Slave Segment	Master Base Address	Range	Master High Address
Network 0					
/processing_system7_0					
/processing_system7_0/Data (32 address bits : 0x40000000 [1G])					
/axi_bram_ctrl_0/S_AXI	S_AXI	Mem0	0x4000_0000	8K	0x4000_1FFF
Excluded (1)					
/axi_bram_ctrl_1/S_AXI	S_AXI	Mem0			

Diagram x Address Editor x

Assigned (2) Unassigned (0) Excluded (0) Hide All

Name	Interface	Slave Segment	Master Base Address	Range	Master High Address
Network 0					
/processing_system7_0					
/processing_system7_0/Data (32 address bits : 0x40000000 [1G])					
/axi_bram_ctrl_0/S_AXI	S_AXI	Mem0	0x4000_0000	4K	0x4000_0FFF
/axi_bram_ctrl_1/S_AXI	S_AXI	Mem0	0x4000_1000	4K	0x4000_1FFF

Bram Read&Write

bram rw_test_system rw_test lscript.ld

Linker Script: lscript.ld

A linker script is used to control where different sections of an executable are placed in memory. In this page, you can define new memory regions, and change the assignment of sections to memory regions.

Available Memory Regions

Name	Base Address	Size
axi_bram_ctrl_0_Mem0	0x40000000	0x1000
axi_bram_ctrl_1_Mem0	0x40001000	0x1000
ps7_ddr_0	0x100000	0x3FF00000
ps7_qspi_linear_0	0xFC000000	0x1000000
ps7_ram_0	0x0	0x30000
ps7_ram_1	0xFFFF0000	0xFE00

Base Address column is highlighted with a red box.

```

1 #include <stdio.h>
2 #include "platform.h"
3 #include "xil_printf.h"
4 #include <sleep.h>
5 #include "xil_io.h"
6 #include "xparameters.h"
7
8 int main()
9 {
10     int a[16] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
11     int b[16] = {0};
12     init_platform();
13
14     print("-----The test is start.....\n\r");
15     xil_printf( "Write data:\n\r");
16
17     memcpy((void *)0x40000000, a, 16*4);
18     for(int i = 0;i<16;i++){
19         xil_printf( "%x ",a[i]);
20     }
21     xil_printf( "\n\r");
22
23     sleep(1);
24     xil_printf( "Read data:\n\r");
25
26     memcpy(b, (void*)0x40001000, 4*16);
27     for(int i = 0;i<16;i++){
28         xil_printf( "%x ",b[i]);
29     }
30
31     xil_printf( "\n\r");
32     xil_printf("-----The test is end!-----\n\r");
33     cleanup_platform();
34
35 }
```

Output window:

```

-----The test is start.....
Write data:
0 1 2 3 4 5 6 7 8 9 A B C D E F
Read data:
0 1 2 3 4 5 6 7 8 9 A B C D E F
-----The test is end!-----

```

BRAM r&w

```
int main(){
    int a[16] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
    int b[16] = {0};
    init_platform();

    print("-----The test is start...-----\n\r");
    xil_printf( "Write data:\n\r");

    memcpy((void *)0x40000000, a, 16*4);
    for(int i = 0;i<16;i++){
        xil_printf( "%x ",a[i]);
    }
    xil_printf( "\n\r");

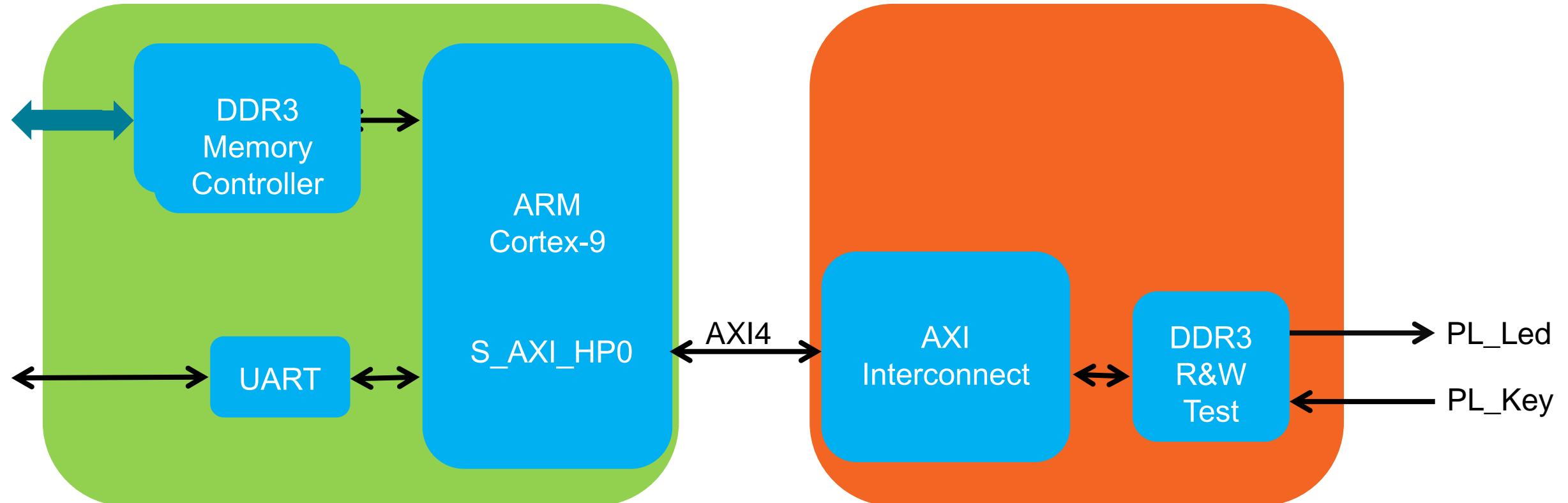
    sleep(1);
    xil_printf( "Read data:\n\r");

    memcpy(b, (void*)0x40001000, 4*16);
    for(int i = 0;i<16;i++){
        xil_printf( "%x ",b[i]);
    }

    xil_printf( "\n\r");
    xil_printf("-----The test is end!-----\n\r");
    cleanup_platform();
    return 0;
}
```

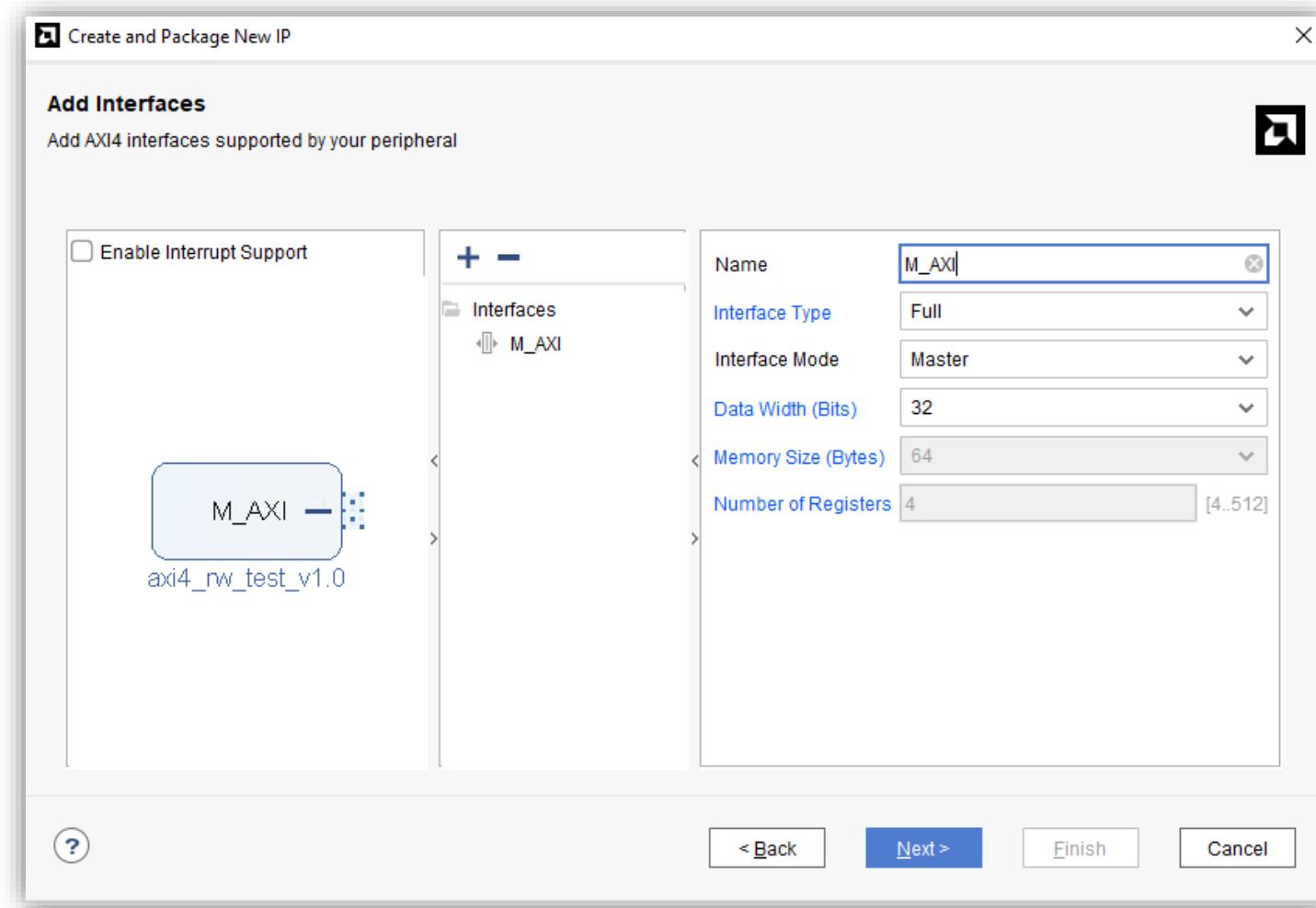
PL Read&Write DDR

Lab Architecture



Xilinx AXI4_Wrapper IP

The AXI4 interface IP provided by the official source comes with a built-in self-test program for read and write operations.



Zynq Core IP Configuration

ZYNQ7 Processing System (5.5)

Documentation Presets IP Location Import XPS Settings

PS-PL Configuration

Search:

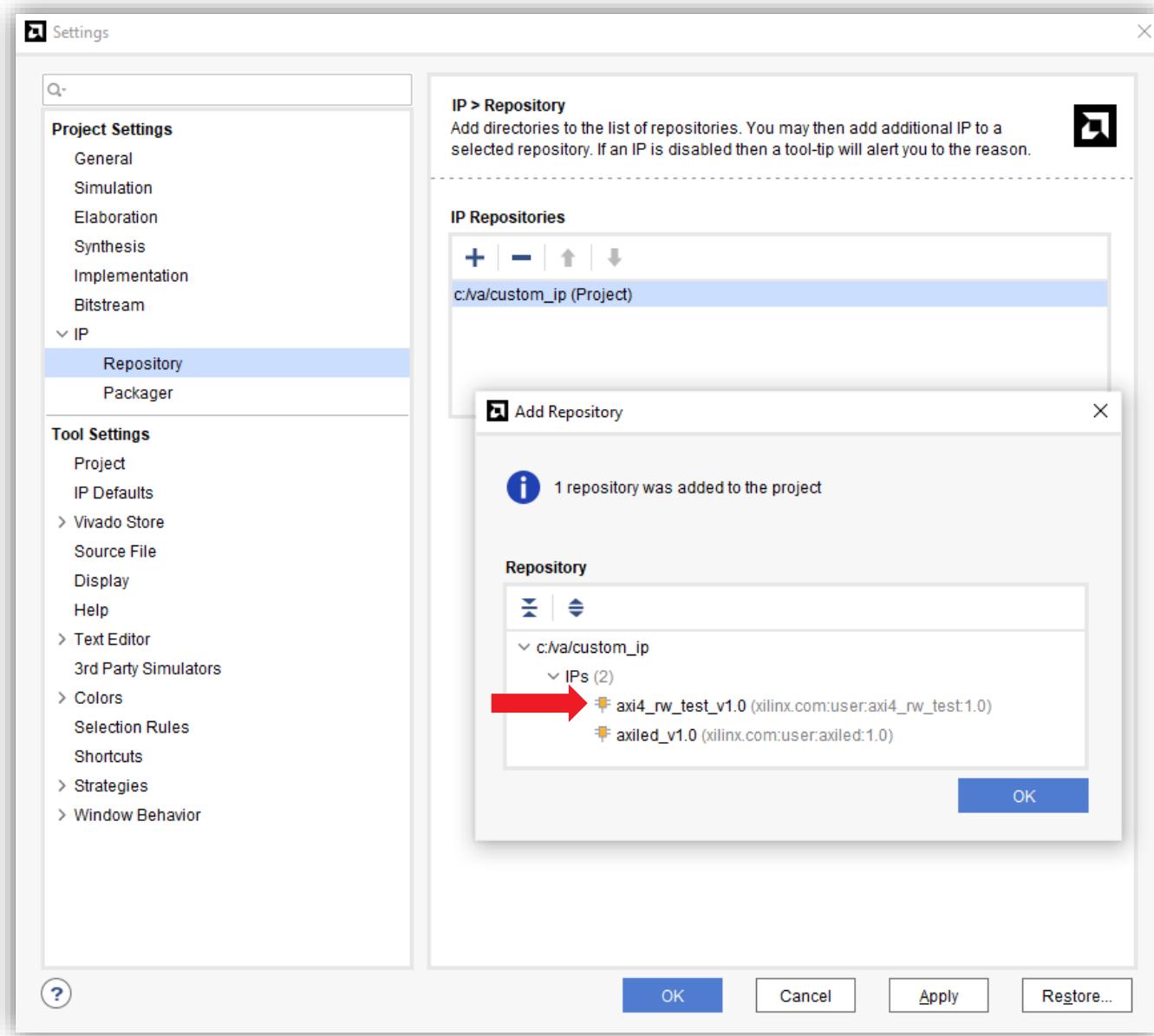
Name	Select	Description
> General		
AXI Non Secure Enablement	0	Enable AXI Non Secure Transaction
GP Master AXI Interface		
M AXI GP0 interface	<input type="checkbox"/>	Enables General purpose AXI master interface 0
M AXI GP1 interface	<input type="checkbox"/>	Enables General purpose AXI master interface 1
GP Slave AXI Interface		
S AXI GP0 interface	<input type="checkbox"/>	Enables General purpose 32-bit AXI Slave interface 0
S AXI GP1 interface	<input type="checkbox"/>	Enables General purpose 32-bit AXI Slave interface 1
HP Slave AXI Interface		
S AXI HP0 interface	<input checked="" type="checkbox"/>	Enables AXI high performance slave interface 0
S AXI HP1 interface	<input type="checkbox"/>	Enables AXI high performance slave interface 1
S AXI HP2 interface	<input type="checkbox"/>	Enables AXI high performance slave interface 2
S AXI HP3 interface	<input type="checkbox"/>	Enables AXI high performance slave interface 3
ACP Slave AXI Interface		
DMA Controller		
> PS-PL Cross Trigger interface	<input type="checkbox"/>	Enables PL cross trigger signals to PS and vice-versa

Clock Configuration

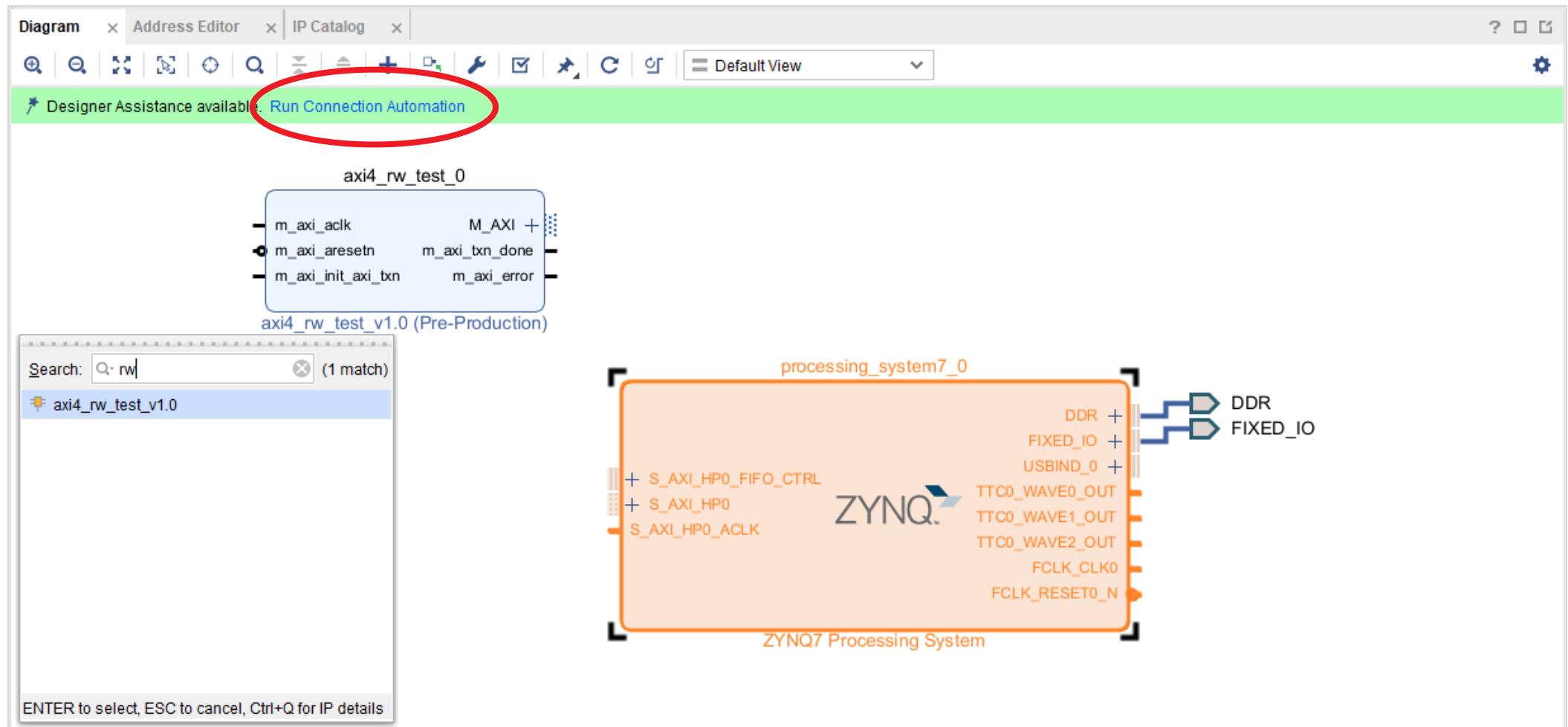
Component	Clock Source	Requested Freq...	Actual Frequency...
> Processor/Memory Clocks			
> IO Peripheral Clocks			
> PL Fabric Clocks			
FCLK_CLK0	IO PLL	50	50.000000
FCLK_CLK1	IO PLL	50	10.000000
FCLK_CLK2	IO PLL	50	10.000000
FCLK_CLK3	IO PLL	50	10.000000
> System Debug Clocks			
> Timers			

OK Cancel

IP Repository

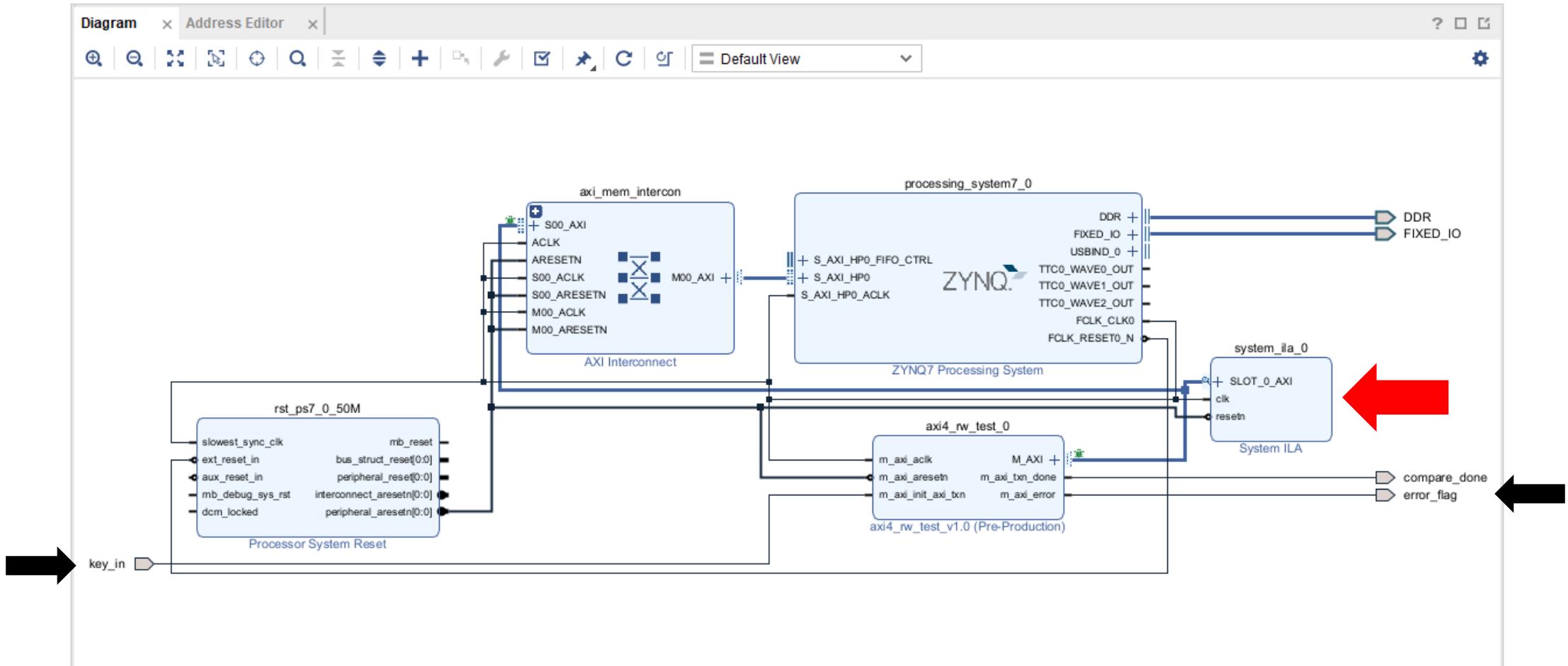


Block Design

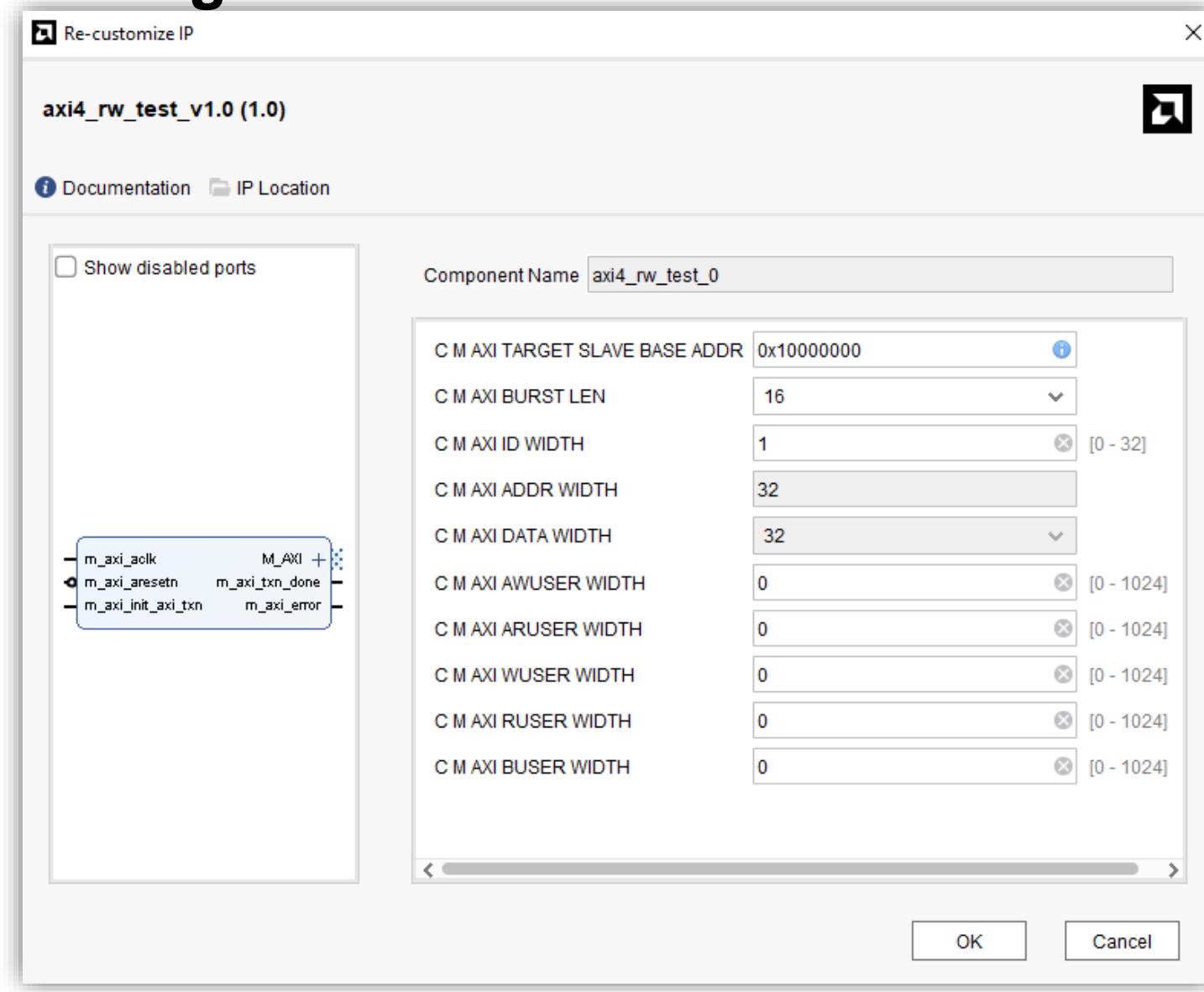


Block Design

Assign the control signals for read and write operations to external buttons, and assign the comparison results to LED lights. Additionally, add ILA to observe internal signals.



AXI4_Wrapper Configuration



PS Read Data

The PS side is only responsible for reading data from DDR.

```

1 #include "stdio.h"
2 #include "xil_cache.h"
3 #include "xil_printf.h"
4 #include "xil_io.h"
5
6 int main(){
7
8     int i;
9     char c;
10    //Disable Cache Access DDR
11    Xil_DCacheDisable();
12    printf("Test Start!");
13
14    //Read DDR3|
15    while(1){
16        scanf("%c",&c);
17        if(c == 'c'){
18            printf("start\r\n");
19            for(i=0;i<4096;i+=4){
20                printf("%d is %d\n",i,(int)(Xil_In32(0x10000000+i)));
21            }
22        }
23    }
24    return 0;
25}
26

```

Each data occupies 4 bits.

Start reading from the address set in PL side.

```

78 ****
79 /**
80 * @brief  Disable the Data cache.
81 *
82 * @return None.
83 */
84 ****
85 void Xil_DCacheDisable(void)
86 {
87     u32 CtrlReg;
88     /* clean and invalidate the Data cache */
89     Xil_DCacheFlush();
90     CtrlReg = mfcp(XREG_CP15_SYS_CONTROL);
91
92     CtrlReg &= ~(XREG_CONTROL_DCACHE_BIT);
93     /* disable the Data cache */
94     mtcp(XREG_CP15_SYS_CONTROL, CtrlReg);
95 }
96

```

DDR Write&Read

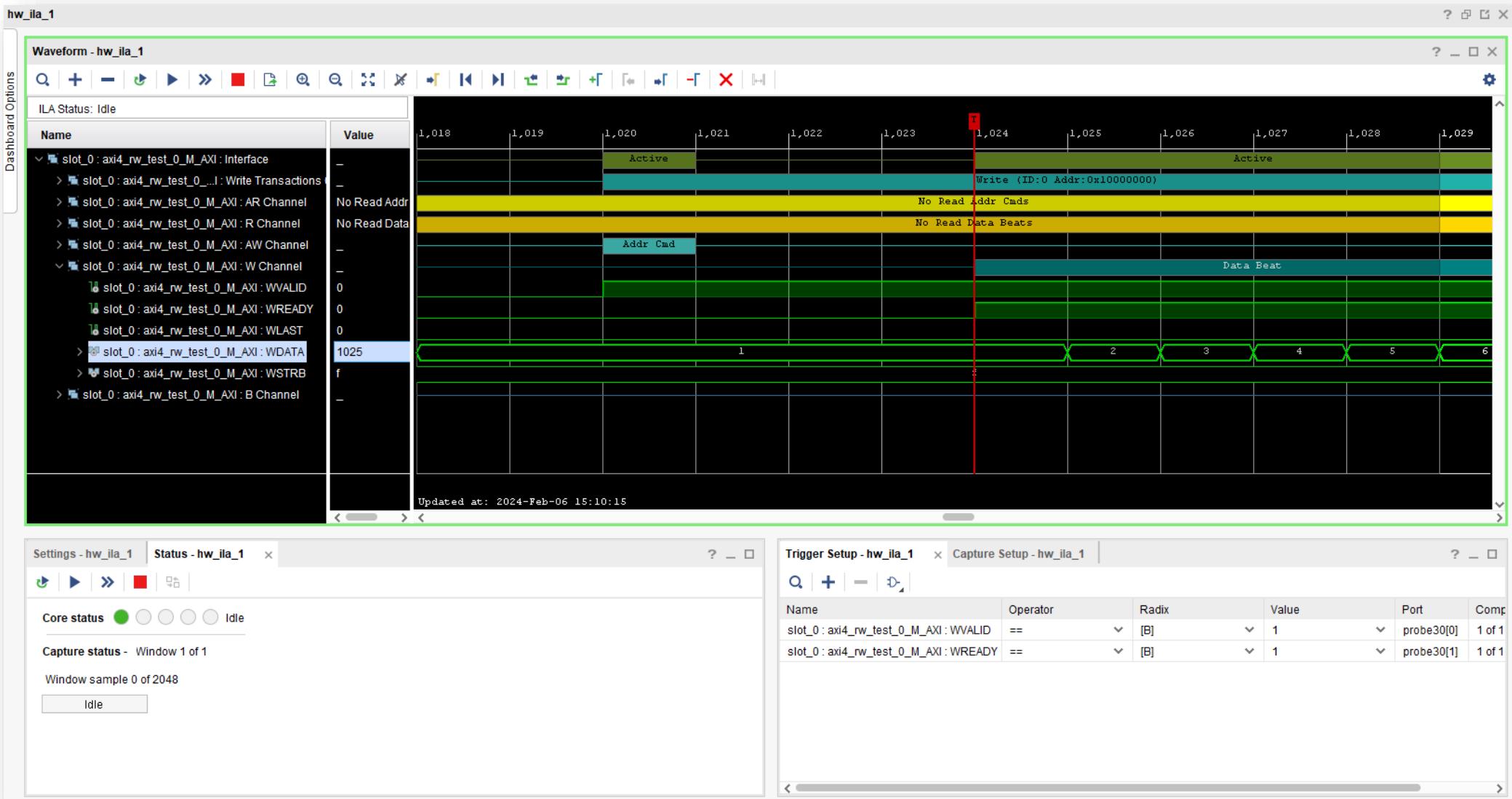
Before pressing the button, it's a random number.

```
Test Start!start
0 is 1364266066
4 is -1347157726
8 is 218303946
12 is 975180548
16 is 235032326
20 is 542706312
24 is 1378357250
28 is -1862136752
32 is -2113810327
36 is -1607169468
40 is 236333329
44 is 1612804934
48 is 445679679
52 is 541495313
56 is 403512332
60 is 577999825
64 is -1601958750
68 is 464127264
72 is 433082565
76 is 1781877526
80 is 353927328
84 is 1147216962
```

After pressing the button, the displayed data is the data written after pressing.

```
start
0 is 1
4 is 2
8 is 3
12 is 4
16 is 5
20 is 6
24 is 7
28 is 8
32 is 9
36 is 10
40 is 11
44 is 12
48 is 13
52 is 14
56 is 15
60 is 16
64 is 17
68 is 18
72 is 19
76 is 20
80 is 21
```

ILA



DDR_test

```
#include "stdio.h"
#include "xil_cache.h"
#include "xil_printf.h"
#include "xil_io.h"

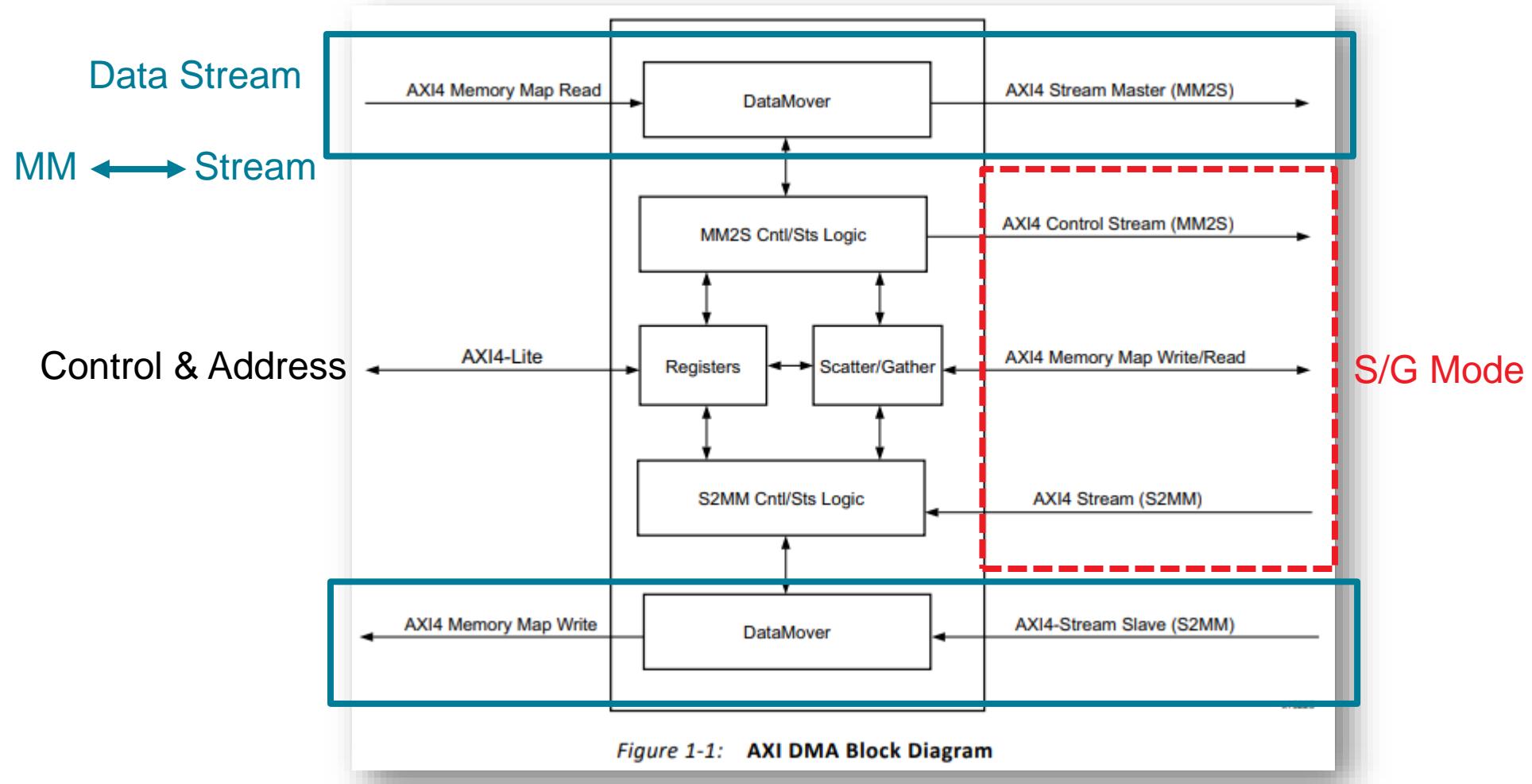
int main(){

    int i;
    char c;
    //Disable Cache Access DDR
    Xil_DCacheDisable();
    printf("Test Start!");

    //Read DDR3
    while(1){
        scanf("%c",&c);
        if(c == 'c'){
            printf("start\r\n");
            for(i=0;i<4096;i+=4){
                printf("%d is %d\r\n",i,(int)(Xil_In32(0x10000000+i)));
            }
        }
    }
    return 0;
}
```

DMA

AXI DMA Block Diagram



Direct Register Mode (Simple DMA)

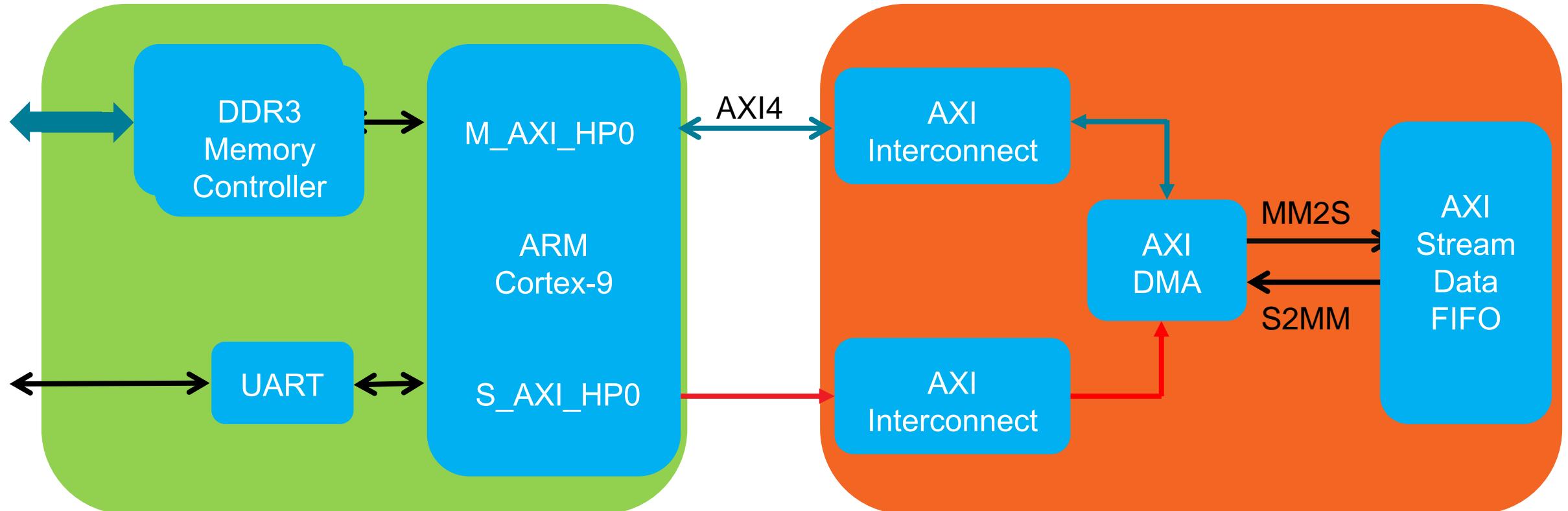
A DMA operation for the **MM2S** channel is set up and started by the following sequence:

1. Start the MM2S channel running by setting the run/stop bit to 1 (MM2S_DMACR.RS =1).
2. If desired, enable interrupts by writing a 1 to MM2S_DMACR.IOC_IrqEn and MM2S_DMACR.Err_IrqEn.
3. Write a valid source address to the MM2S_SA register.
4. Write the number of bytes to transfer in the MM2S_LENGTH register.

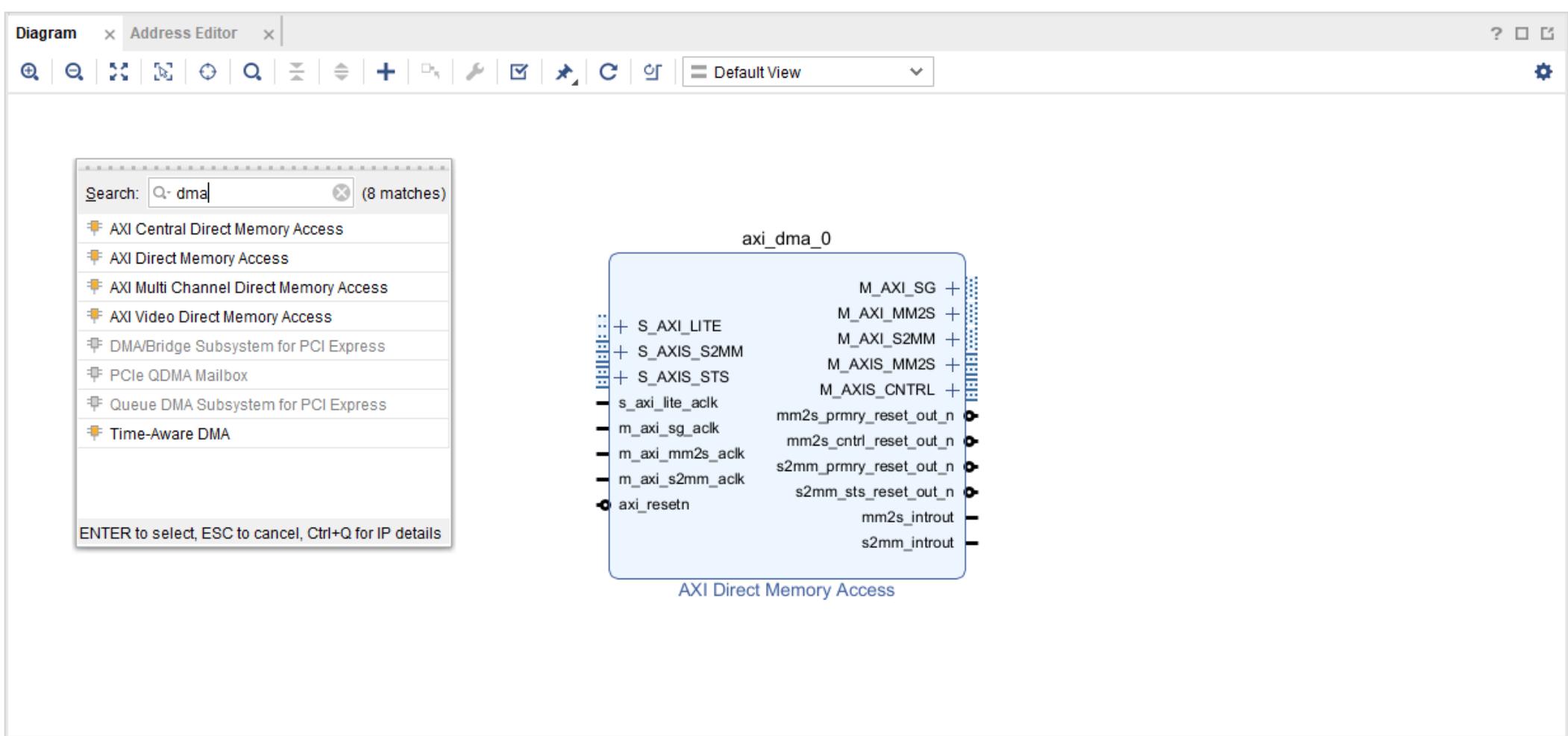
A DMA operation for the **S2MM** channel is set up and started by the following sequence:

1. Start the S2MM channel running by setting the run/stop bit to 1 (S2MM_DMACR.RS =1).
2. If desired, enable interrupts by writing a 1 to S2MM_DMACR.IOC_IrqEn and S2MM_DMACR.Err_IrqEn.
3. Write a valid destination address to the S2MM_DA register.
4. If the AXI DMA is not configured for Data Re-Alignment then a valid address must be aligned or undefined results occur.
5. Write the length in bytes of the receive buffer in the S2MM_LENGTH register.

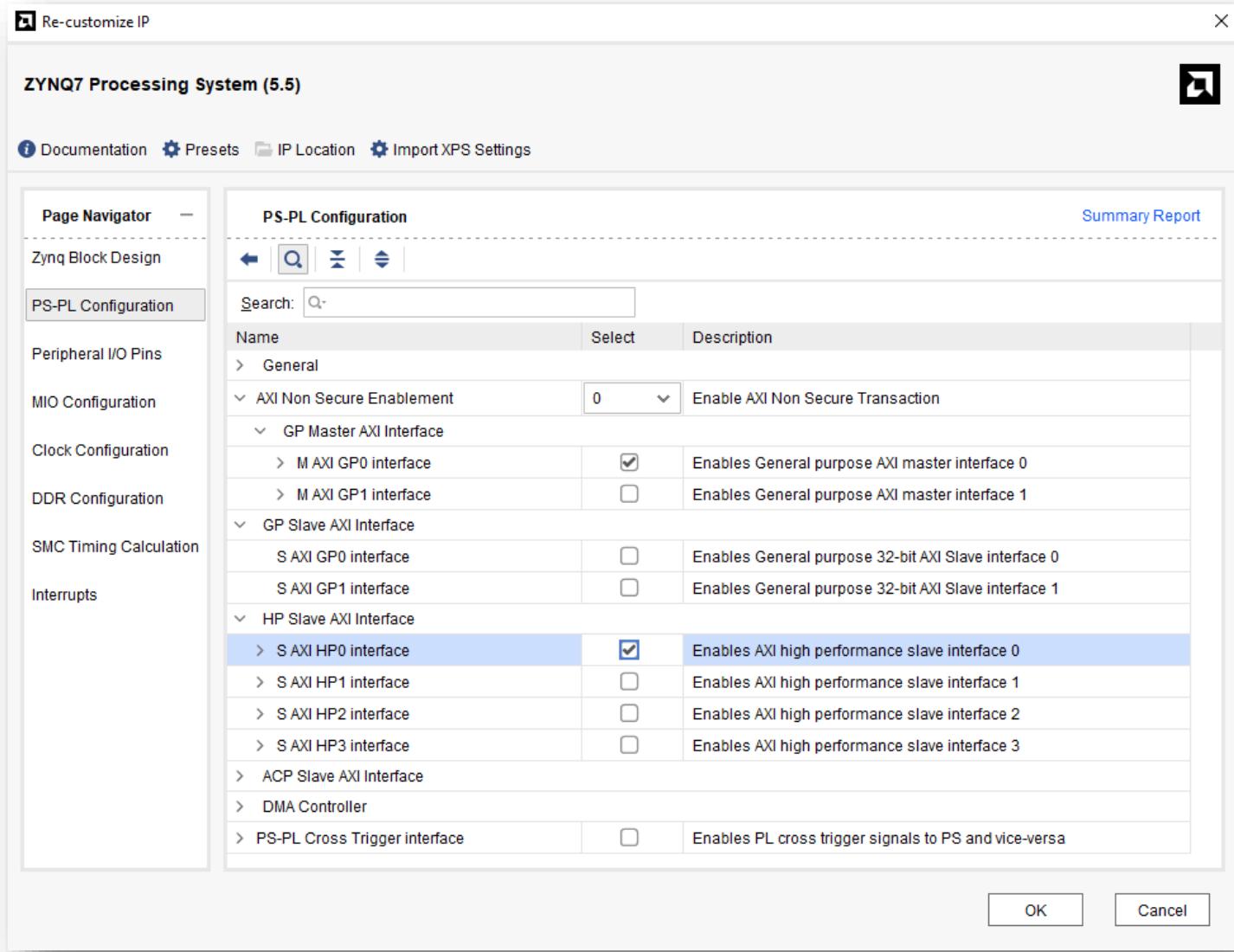
Lab Architecture



ILA



ZYNQ Core IP Configuration



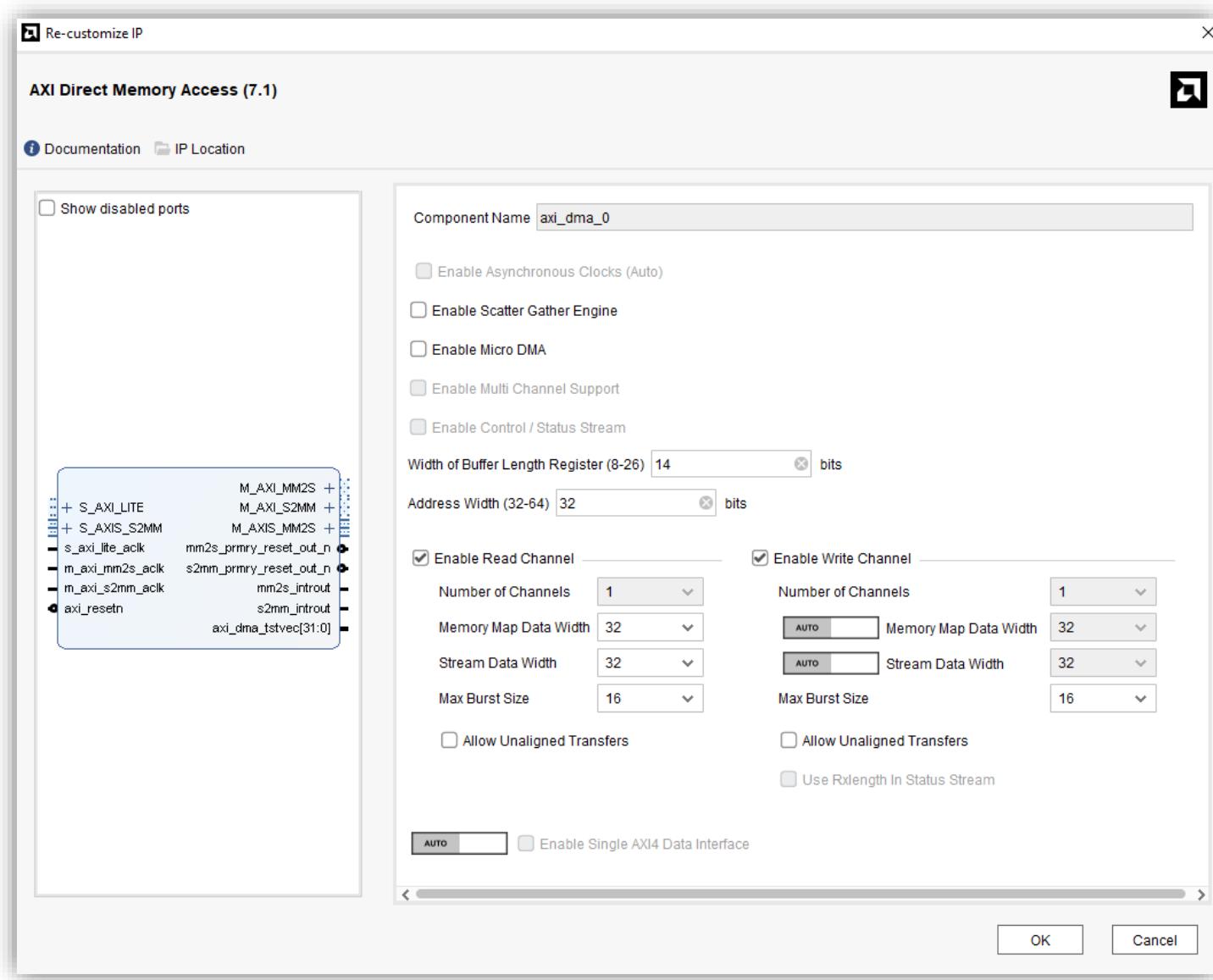
ZYNQ Core IP Configuration

The screenshot shows the ZYNQ Core IP Configuration interface for a ZYNQ7 Processing System (5.5). The main window displays the **Clock Configuration** tab under the **Zynq Block Design** category. In the **Basic Clocking** section, the Input Frequency (MHz) is set to 33.333333 and the CPU Clock Ratio is 6:2:1. The **Interrupts** tab is open, showing a list of interrupt ports. The **PL-PS Interrupt Ports** section includes the following entries:

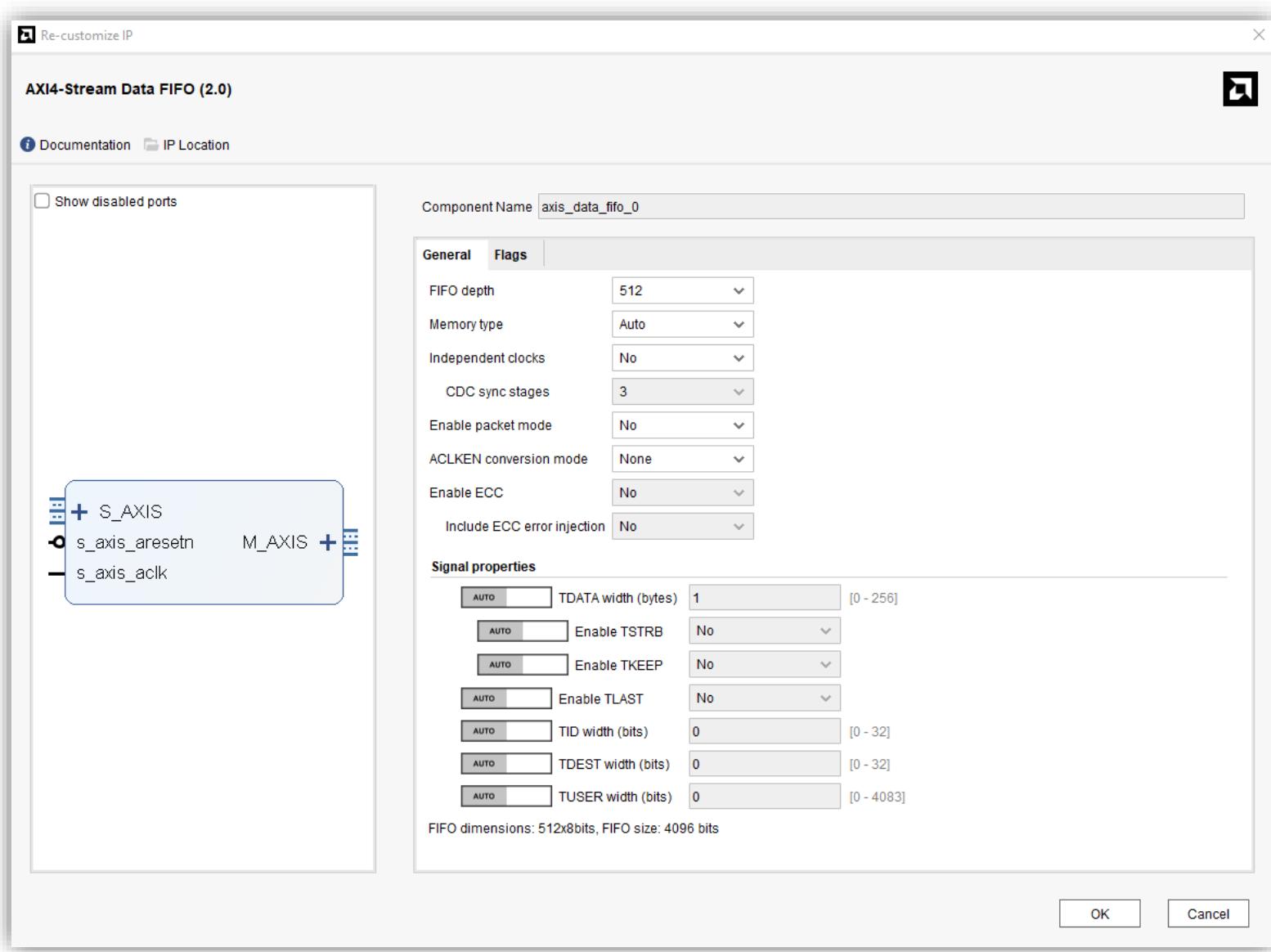
Interrupt Port	ID	Description
<input checked="" type="checkbox"/> IRQ_F2P[15:0]	[91:84], [68:6]	Enables 16-bit shared interrupt port from the PL. MSB is assigned the hi
<input type="checkbox"/> Core0_nFIQ	28	Enables fast private interrupt signal for CPU0 from the PL
<input type="checkbox"/> Core0_nIRQ	31	Enables private interrupt signal for CPU0 from the PL
<input type="checkbox"/> Core1_nFIQ	28	Enables fast private interrupt signal for CPU1 from the PL
<input type="checkbox"/> Core1_nIRQ	31	Enables private interrupt signal for CPU1 from the PL

At the bottom of the interface are **OK** and **Cancel** buttons.

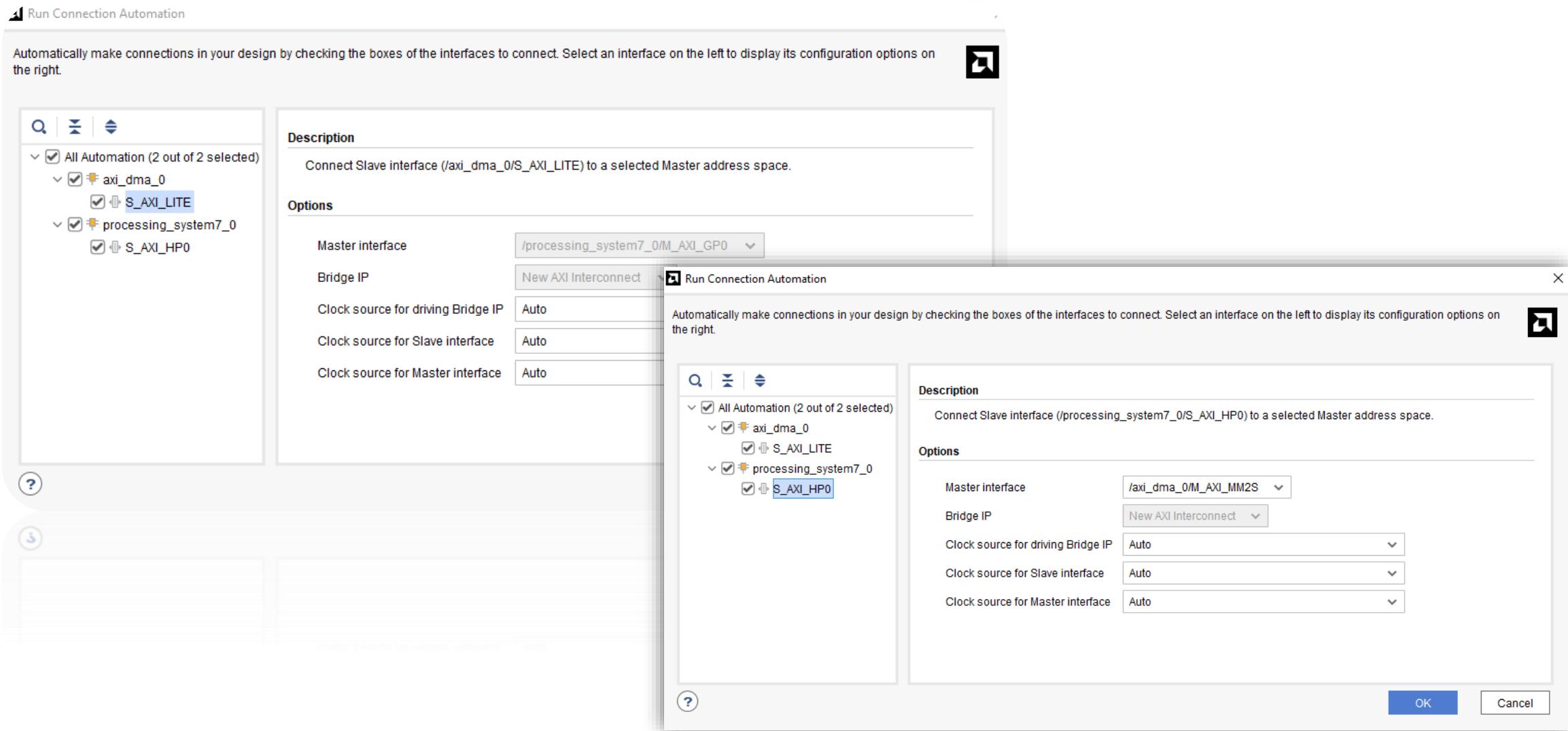
AXI DMA IP Configuration



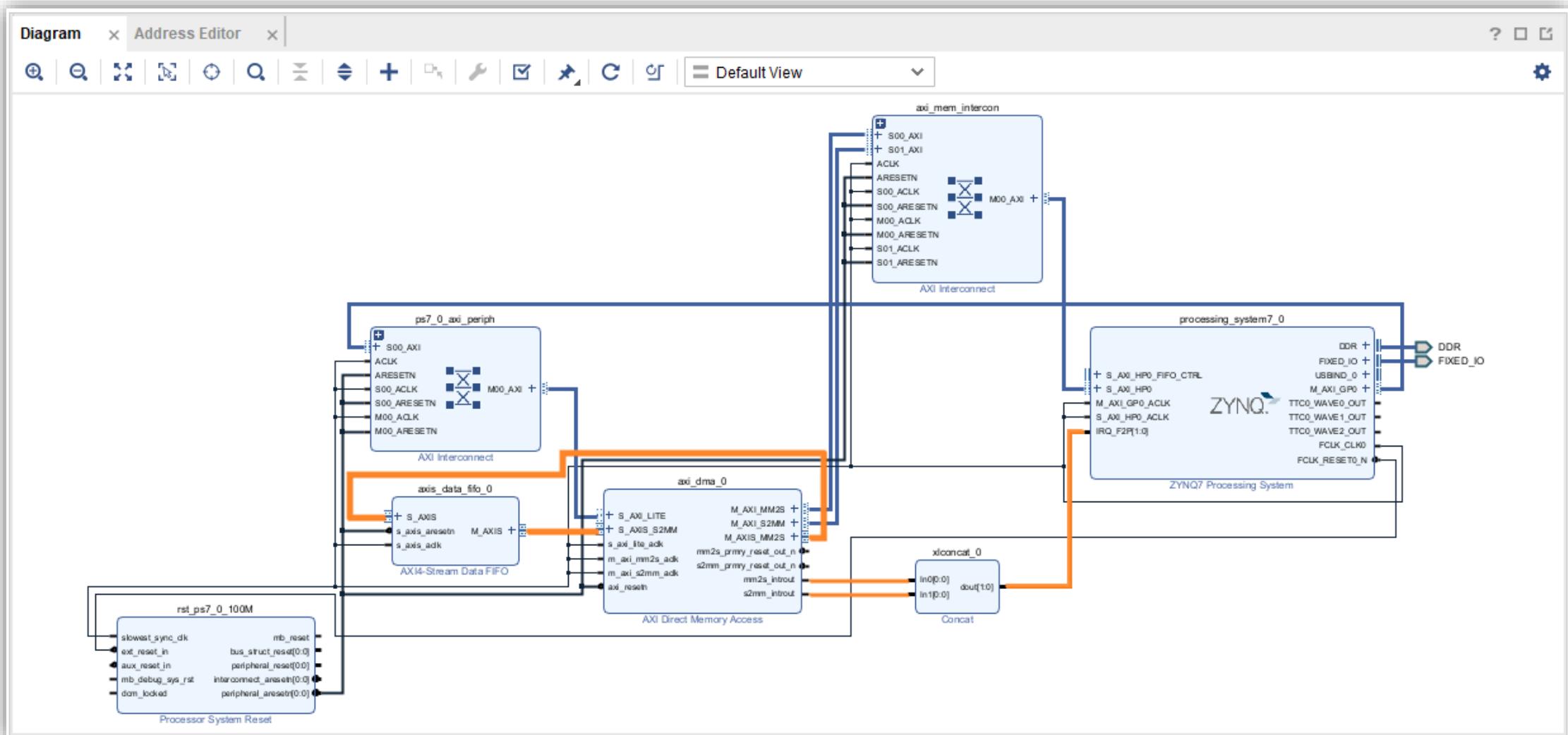
AXI Stream Data FIFO



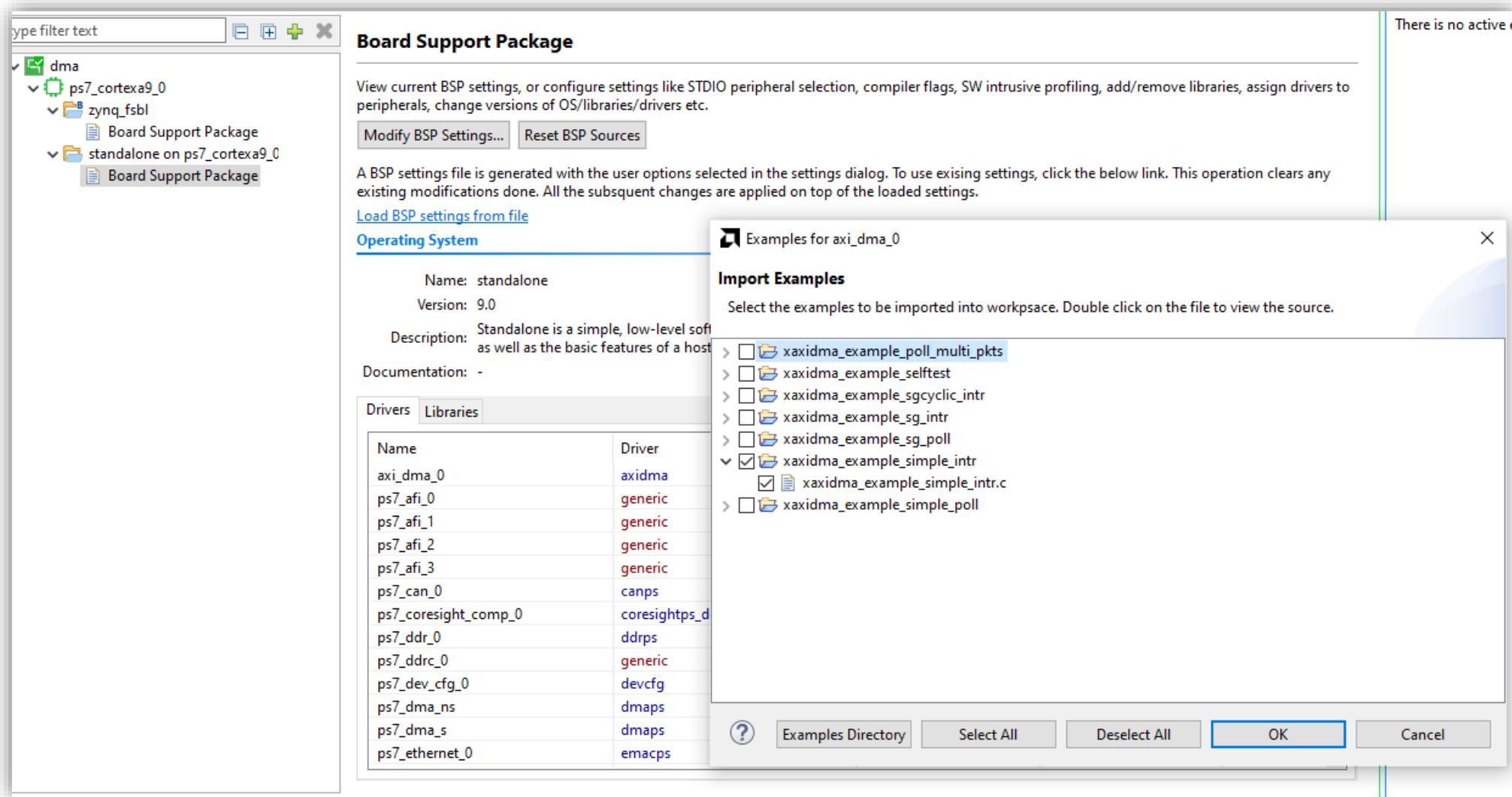
Connection



Connection



DMA Example Design



DMA Test Lab

Linker Script: lscript.ld

A linker script is used to control where different sections of an executable are placed in memory.
In this page, you can define new memory regions, and change the assignment of sections to memory regions.

Available Memory Regions

Name	Base Address	Size
ps7_ddr_0	0x100000	0x3FF00000
ps7_qspi_linear_0	0xFC000000	0x1000000
ps7_ram_0	0x0	0x30000
ps7_ram_1	0xFFFFF000	0xFE00

```

.c dma_rw.c    .c xavidma_example_simple_intr.c    .c xscugic.c    .h xparameters.h
104
105
106 /* Definitions for peripheral PS7_DDR_0 */
107 #define XPAR_PS7_DDR_0_S_AXI_BASEADDR 0x00100000
108 #define XPAR_PS7_DDR_0_S_AXI_HIGHADDR 0x3FFFFFFF
109
110

```

```

1 **** Include Files ****
2 #include "xavidma.h"
3 #include "xparameters.h"
4 #include "xil_exception.h"
5 #include "xscugic.h"
6
7 **** Constant Definitions ****
8 #define DMA_DEV_ID      XPAR_AXIDMA_0_DEVICE_ID
9 #define RX_INTR_ID      XPAR_FABRIC_AXIDMA_0_S2MM_INTROUT_VEC_ID
10 #define TX_INTR_ID      XPAR_FABRIC_AXIDMA_0_MM2S_INTROUT_VEC_ID
11 #define INTC_DEVICE_ID  XPAR_SCUGIC_SINGLE_DEVICE_ID
12 #define INTC_HANDLER    XScuGic InterruptHandler
13
14 #define DDR_BASE_ADDR   XPAR_PS7_DDR_0_S_AXI_BASEADDR          //0x00100000
15 #define MEM_BASE_ADDR   (DDR_BASE_ADDR + 0x1000000)           //0x01100000
16 #define TX_BUFFER_BASE (MEM_BASE_ADDR + 0x00100000)           //0x01200000
17 #define RX_BUFFER_BASE (MEM_BASE_ADDR + 0x00300000)           //0x01400000
18
19 /* Timeout loop counter for reset */
20 #define RESET_TIMEOUT_COUNTER 1000
21 #define TEST_START_VALUE 0xC
22
23 /* Buffer and Buffer Descriptor related constant definition */
24 #define MAX_PKT_LEN 0x100
25
26 **** Function Prototypes ****
27 static int CheckData(int Length, u8 StartValue);
28 static void TxIntrHandler(void *Callback);
29 static void RxIntrHandler(void *Callback);
30 static int SetupIntrSystem(XScuGic *IntcInstancePtr,XAxiDma *AxiDmaPtr, u16 TxIntrId, u16 RxIntrId);
31 static void DisableIntrSystem(XScuGic *IntcInstancePtr,u16 TxIntrId, u16 RxIntrId);
32
33 **** Variable Definitions ****
34 /* Device instance definitions */
35 static XAxiDma AxiDma; /* Instance of the XAxiDma */
36 static XScuGic Intc; /* Instance of the Interrupt Controller */
37
38 /* Flags interrupt handlers use to notify the application context the events. */
39 volatile u32 TxDone;
40 volatile u32 RxDone;
41 volatile u32 Error;

```

DMA Test Lab

```
192 int main(void){  
193     XAxiDma_Config *Config;  
194     int Index;  
195     int Status;  
196     u8 *TxBufferPtr;  
197     u8 *RxBufferPtr;  
198     u8 Value;  
199     TxBufferPtr = (u8 *)TX_BUFFER_BASE ;  
200     RxBufferPtr = (u8 *)RX_BUFFER_BASE;  
201  
202     xil_printf("\r\n--- Entering main() --- \r\n");  
203  
204     /* Initialize DMA engine */  
205  
206     Config = XAxiDma_LookupConfig(DMA_DEV_ID);  
207     if (!Config) {  
208         xil_printf("No config found for %d\r\n", DMA_DEV_ID);  
209         return XST_FAILURE;  
210     }  
211     Status = XAxiDma_CfgInitialize(&AxiDma, Config);  
212  
213     if (Status != XST_SUCCESS) {  
214         xil_printf("Initialization failed %d\r\n", Status);  
215         return XST_FAILURE;  
216     }  
217  
218     if (XAxiDma_HasSg(&AxiDma)) {  
219         xil_printf("Device configured as SG mode \r\n");  
220         return XST_FAILURE;  
221     }  
222 }
```



Decide on the DMA mode as simple mode.

DMA Test Lab

```

/* Set up Interrupt system */

Status = SetupIntrSystem(&Intc, &AxiDma, TX_INTR_ID, RX_INTR_ID);
if (Status != XST_SUCCESS) {
    xil_printf("Failed intr setup\r\n");
    return XST_FAILURE;
}
/* Disable all interrupts before setup */
XAxIDma_IntrDisable(&AxiDma, XAXIDMA_IRQ_ALL_MASK,XAXIDMA_DMA_TO_DEVICE);
XAxIDma_IntrDisable(&AxiDma, XAXIDMA_IRQ_ALL_MASK,XAXIDMA_DEVICE_TO_DMA);

/* Enable all interrupts */
XAxIDma_IntrEnable(&AxiDma, XAXIDMA_IRQ_ALL_MASK,XAXIDMA_DMA_TO_DEVICE);
XAxIDma_IntrEnable(&AxiDma, XAXIDMA_IRQ_ALL_MASK,XAXIDMA_DEVICE_TO_DMA);

/* Initialize flags before start transfer test */
TxDone = 0;
RxDone = 0;
Error = 0;

Value = TEST_START_VALUE;
for (Index = 0; Index < MAX_PKT_LEN; Index++) {
    TxBufferPtr[Index] = Value;
    Value = (Value + 1) & 0xFF;
}

/* Flush the buffers before the DMA transfer, in case the Data Cache is enabled */
Xil_DCacheFlushRange((UINTPTR)TxBufferPtr, MAX_PKT_LEN);

```

Disable cache access to DDR in advance.

```

/* Start Transfer Channel */
Status = XAxIDma_SimpleTransfer(&AxiDma, (UINTPTR) TxBufferPtr,MAX_PKT_LEN,
                                XAXIDMA_DMA_TO_DEVICE);
if (Status != XST_SUCCESS) {
    return XST_FAILURE;
}
Status = XAxIDma_SimpleTransfer(&AxiDma, (UINTPTR) RxBufferPtr,MAX_PKT_LEN,
                                XAXIDMA_DEVICE_TO_DMA);
if (Status != XST_SUCCESS) {
    return XST_FAILURE;
}

/* Flush the buffers before the DMA transfer, in case the Data Cache is enabled */
Xil_DCacheFlushRange((UINTPTR)RxBufferPtr, MAX_PKT_LEN);

while(!TxDone && !RxDone && !Error);
if(Error){
    xil_printf("Testing Failed.");
    goto Done;
}
/* Test finished, check data */
Status = CheckData(MAX_PKT_LEN, 0xC);
if (Status != XST_SUCCESS) {
    xil_printf("Data check failed\r\n");
    goto Done;
}

xil_printf("Successfully Ran AXI DMA Loop\r\n");
/* Disable TX and RX Ring interrupts and return success */
DisableIntrSystem(&Intc, TX_INTR_ID, RX_INTR_ID);

Done:xil_printf("--- Exiting main() --- \r\n");
return XST_SUCCESS;

```

Enable the DMA transfer channel.

DMA Interrupt Subsystem

```

static int SetupIntrSystem(XScuGic *IntcInstancePtr,
    XAxiDma *AxiDmaPtr, u16 TxIntrId, u16 RxIntrId){

    XScuGic_Config *IntcConfig;
    int Status;

    /* Initialize the interrupt controller driver so that it is ready to use. */
    IntcConfig = XScuGic_LookupConfig(INTC_DEVICE_ID);
    if (NULL == IntcConfig) {
        return XST_FAILURE;
    }
    Status = XScuGic_CfgInitialize(IntcInstancePtr, IntcConfig,
        IntcConfig->CpuBaseAddress);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    /* Set Interrupt Priority and TriggerType */
    XScuGic_SetPriorityTriggerType(IntcInstancePtr, TxIntrId, 0xA0, 0x3);
    XScuGic_SetPriorityTriggerType(IntcInstancePtr, RxIntrId, 0xA0, 0x3);

    /* Connect the device driver handler */
    Status = XScuGic_Connect(IntcInstancePtr, TxIntrId,
        (Xil_InterruptHandler)TxIntrHandler,
        AxiDmaPtr);
    if (Status != XST_SUCCESS) {
        return Status;
    }
    Status = XScuGic_Connect(IntcInstancePtr, RxIntrId,
        (Xil_InterruptHandler)RxIntrHandler,
        AxiDmaPtr);
    if (Status != XST_SUCCESS) {
        return Status;
    }
}

```

```

/* Enable interrupts from the hardware */

XScuGic_Enable(IntcInstancePtr, TxIntrId);
XScuGic_Enable(IntcInstancePtr, RxIntrId);

Xil_ExceptionInit();
Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
    (Xil_ExceptionHandler)INTC_HANDLER,
    (void *)IntcInstancePtr);

Xil_ExceptionEnable();

return XST_SUCCESS;

```

DMA Interrupt Subsystem

```
static void TxIntrHandler(void *Callback)
{
    u32 IrqStatus;
    int TimeOut;
    XAxiDma *AxiDmaInst = (XAxiDma *)Callback;

    /* Read pending interrupts */
    IrqStatus = XAxiDma_IntrGetIrq(AxiDmaInst, XAXIDMA_DMA_TO_DEVICE);
    /* Acknowledge pending interrupts */
    XAxiDma_IntrAckIrq(AxiDmaInst, IrqStatus, XAXIDMA_DMA_TO_DEVICE);
    /* If no interrupt is asserted, we do not do anything */
    if ((IrqStatus & XAXIDMA_IRQ_ERROR_MASK)) {
        Error = 1;
        /* Reset should never fail for transmit channel */
        XAxiDma_Reset(AxiDmaInst);
        TimeOut = RESET_TIMEOUT_COUNTER;
        while (TimeOut) {
            if (XAxiDma_ResetIsDone(AxiDmaInst)) {
                break;
            }
            TimeOut -= 1;
        }
        return;
    }
    /* If Completion interrupt is asserted, then set the TxDone flag */
    if ((IrqStatus & XAXIDMA_IRQ_IOC_MASK)) {
        TxDone = 1;
    }
}
```

```
static void RxIntrHandler(void *Callback)
{
    u32 IrqStatus;
    int TimeOut;
    XAxiDma *AxiDmaInst = (XAxiDma *)Callback;

    /* Read pending interrupts */
    IrqStatus = XAxiDma_IntrGetIrq(AxiDmaInst, XAXIDMA_DEVICE_TO_DMA);
    /* Acknowledge pending interrupts */
    XAxiDma_IntrAckIrq(AxiDmaInst, IrqStatus, XAXIDMA_DEVICE_TO_DMA);

    /* If no interrupt is asserted, we do not do anything */
    if ((IrqStatus & XAXIDMA_IRQ_ERROR_MASK)) {
        Error = 1;
        /* Reset could fail and hang
         * NEED a way to handle this or do not call it??
         */
        XAxiDma_Reset(AxiDmaInst);
        TimeOut = RESET_TIMEOUT_COUNTER;
        while (TimeOut) {
            if (XAxiDma_ResetIsDone(AxiDmaInst)) {
                break;
            }
            TimeOut -= 1;
        }
        return;
    }
    /* If completion interrupt is asserted, then set RxDone flag */
    if ((IrqStatus & XAXIDMA_IRQ_IOC_MASK)) {
        RxDone = 1;
    }
}
```

CheckData & Disable

```
static int CheckData(int Length, u8 StartValue)
{
    u8 *RxPacket;
    int Index = 0;
    u8 Value;

    RxPacket = (u8 *) RX_BUFFER_BASE;
    Value = StartValue;

    /* Invalidate the DestBuffer before receiving the data, in case the
     * Data Cache is enabled
     */
    Xil_DCacheInvalidateRange((UINTPTR)RxPacket, Length);

    for (Index = 0; Index < Length; Index++) {
        if (RxPacket[Index] != Value) {
            xil_printf("Data error %d: %x/%x\r\n",
                      Index, RxPacket[Index], Value);
            return XST_FAILURE;
        }
        Value = (Value + 1) & 0xFF;
    }
    return XST_SUCCESS;
}
```

```
static void DisableIntrSystem(XScuGic *IntcInstancePtr,
                               u16 TxIntrId, u16 RxIntrId){
    /* Disconnect the interrupts for the DMA TX and RX channels */
    XScuGic_Disconnect(IntcInstancePtr, TxIntrId);
    XScuGic_Disconnect(IntcInstancePtr, RxIntrId);
}
```

Dma_test

```
***** Include Files *****
#include "xaxidma.h"
#include "xparameters.h"
#include "xil_exception.h"
#include "xscugic.h"

***** Constant Definitions *****
#define DMA_DEV_ID XPAR_AXIDMA_0_DEVICE_ID
#define RX_INTR_ID XPAR_FABRIC_AXIDMA_0_S2MM_INTROUT_VEC_ID
#define TX_INTR_ID XPAR_FABRIC_AXIDMA_0_MM2S_INTROUT_VEC_ID
#define INTC_DEVICE_ID XPAR_SCUGIC_SINGLE_DEVICE_ID
#define INTC_HANDLERXScuGic_InterruptHandler

#define DDR_BASE_ADDR XPAR_PS7_DDR_0_S_AXI_BASEADDR//0x00100000
#define MEM_BASE_ADDR (DDR_BASE_ADDR + 0x1000000)//0x01100000
#define TX_BUFFER_BASE (MEM_BASE_ADDR + 0x00100000)//0x01200000
#define RX_BUFFER_BASE (MEM_BASE_ADDR + 0x00300000)//0x01400000

/* Timeout loop counter for reset */
#define RESET_TIMEOUT_COUNTER10000
#define TEST_START_VALUE0xC

/* Buffer and Buffer Descriptor related constant definition */
#define MAX_PKT_LEN0x100
```

Dma_test

```
***** Function Prototypes *****
static int CheckData(int Length, u8 StartValue);
static void TxIntrHandler(void *Callback);
static void RxIntrHandler(void *Callback);
static int SetupIntrSystem(XScuGic *IntcInstancePtr,XAxiDma *AxiDmaPtr, u16 TxIntrId, u16 RxIntrId);
static void DisableIntrSystem(XScuGic *IntcInstancePtr,u16 TxIntrId, u16 RxIntrId);

***** Variable Definitions *****
/* Device instance definitions */
static XAxiDma AxiDma; /* Instance of the XAxiDma */
static XScuGic Intc; /* Instance of the Interrupt Controller */

/* Flags interrupt handlers use to notify the application context the events. */
volatile u32 TxDone;
volatile u32 RxDone;
volatile u32 Error;
```

Dma_test

```

***** Function Definitions *****
static void TxIntrHandler(void *Callback) {
    u32 IrqStatus;
    int TimeOut;
    XAxiDma *AxiDmaInst = (XAxiDma *)Callback;
    /* Read pending interrupts */
    IrqStatus = XAxiDma_IntrGetIrq(AxiDmaInst, XAXIDMA_DMA_TO_DEVICE);
    /* Acknowledge pending interrupts */
    XAxiDma_IntrAckIrq(AxiDmaInst, IrqStatus, XAXIDMA_DMA_TO_DEVICE);
    /* If no interrupt is asserted, we do not do anything */
    if ((IrqStatus & XAXIDMA_IRQ_ERROR_MASK)) {
        Error = 1;
    /* Reset should never fail for transmit channel */
        XAxiDma_Reset(AxiDmaInst);
        TimeOut = RESET_TIMEOUT_COUNTER;
        while (TimeOut) {
            if (XAxiDma_ResetIsDone(AxiDmaInst)) {
                break;
            }
            TimeOut -= 1;
        }
    return;
}
/* If Completion interrupt is asserted, then set the TxDone flag */
    if ((IrqStatus & XAXIDMA_IRQ_IOC_MASK)) {
        TxDone = 1;
    }
}

```

Dma_test

```

static void RxIntrHandler(void *Callback) {
    u32 IrqStatus;
    int TimeOut;
    XAxiDma *AxiDmaInst = (XAxiDma *)Callback;
    /* Read pending interrupts */
    IrqStatus = XAxiDma_IntrGetIrq(AxiDmaInst, XAXIDMA_DEVICE_TO_DMA);
    /* Acknowledge pending interrupts */
    XAxiDma_IntrAckIrq(AxiDmaInst, IrqStatus, XAXIDMA_DEVICE_TO_DMA);
    /* If no interrupt is asserted, we do not do anything */
    if ((IrqStatus & XAXIDMA_IRQ_ERROR_MASK)) {
        Error = 1;
        /* Reset could fail and hang
         * NEED a way to handle this or do not call it??
         */
        XAxiDma_Reset(AxiDmaInst);
        TimeOut = RESET_TIMEOUT_COUNTER;
        while (TimeOut) {
            if (XAxiDma_ResetIsDone(AxiDmaInst)) {
                break;
            }
            TimeOut -= 1;
        }
    return;
}
/* If completion interrupt is asserted, then set RxDone flag */
if ((IrqStatus & XAXIDMA_IRQ_IOC_MASK)) {
    RxDone = 1;
}
}

```

Dma_test

```

static int CheckData(int Length, u8 StartValue) {
    u8 *RxPacket;
    int Index = 0;
    u8 Value;

    RxPacket = (u8 *) RX_BUFFER_BASE;
    Value = StartValue;
    /* Invalidate the DestBuffer before receiving the data, in case the
     * Data Cache is enabled
    */
    Xil_DCacheInvalidateRange((UINTPTR)RxPacket, Length);
    for (Index = 0; Index < Length; Index++) {
        if (RxPacket[Index] != Value) {
            xil_printf("Data error %d: %x/%x\r\n",
                      Index, RxPacket[Index], Value);
            return XST_FAILURE;
        }
        Value = (Value + 1) & 0xFF;
    }
    return XST_SUCCESS;
}

static void DisableIntrSystem(XScuGic *IntcInstancePtr,
    u16 TxIntrId, u16 RxIntrId) {
    /* Disconnect the interrupts for the DMA TX and RX channels */
    XScuGic_Disconnect(IntcInstancePtr, TxIntrId);
    XScuGic_Disconnect(IntcInstancePtr, RxIntrId);
}

```

Dma_test

```
int main(void) {
    XAxiDma_Config *Config;
    int Index;
    int Status;

    u8 *TxBufferPtr;
    u8 *RxBufferPtr;
    u8 Value;
    TxBufferPtr = (u8 *)TX_BUFFER_BASE ;
    RxBufferPtr = (u8 *)RX_BUFFER_BASE;

    xil_printf("\r\n--- Entering main() --- \r\n");
    /* Initialize DMA engine */
    Config = XAxiDma_LookupConfig(DMA_DEV_ID);
    if (!Config) {
        xil_printf("No config found for %d\r\n", DMA_DEV_ID);
        return XST_FAILURE;
    }
    Status = XAxiDma_CfgInitialize(&AxiDma, Config);
    if (Status != XST_SUCCESS) {
        xil_printf("Initialization failed %d\r\n", Status);
        return XST_FAILURE;
    }

    if (XAxiDma_HasSg(&AxiDma)) {
        xil_printf("Device configured as SG mode \r\n");
    }
    return XST_FAILURE;
}
```

Dma_test

```
/* Set up Interrupt system */

Status = SetupIntrSystem(&Intc, &AxiDma, TX_INTR_ID, RX_INTR_ID);
if (Status != XST_SUCCESS) {
    xil_printf("Failed intr setup\r\n");
    return XST_FAILURE;
}
/* Disable all interrupts before setup */
XAxiDma_IntrDisable(&AxiDma, XAXIDMA_IRQ_ALL_MASK,XAXIDMA_DMA_TO_DEVICE);
XAxiDma_IntrDisable(&AxiDma, XAXIDMA_IRQ_ALL_MASK,XAXIDMA_DEVICE_TO_DMA);

/* Enable all interrupts */
XAxiDma_IntrEnable(&AxiDma, XAXIDMA_IRQ_ALL_MASK,XAXIDMA_DMA_TO_DEVICE);
XAxiDma_IntrEnable(&AxiDma, XAXIDMA_IRQ_ALL_MASK,XAXIDMA_DEVICE_TO_DMA);

/* Initialize flags before start transfer test */
TxDone = 0;
RxDone = 0;
Error = 0;

Value = TEST_START_VALUE;
for (Index = 0; Index < MAX_PKT_LEN; Index ++) {
    TxBUFFERPTR[Index] = Value;
    Value = (Value + 1) & 0xFF;
}
```

Dma_test

```

/* Flush the buffers before the DMA transfer, in case the Data Cache is enabled */
Xil_DCacheFlushRange((UINTPTR)TxBufferPtr, MAX_PKT_LEN);
/* Start Transfer Channel */
Status = XAxiDma_SimpleTransfer(&AxiDma, (UINTPTR) TxBufferPtr,MAX_PKT_LEN,
XAXIDMA_DMA_TO_DEVICE);
if (Status != XST_SUCCESS) {
    return XST_FAILURE;
}
Status = XAxiDma_SimpleTransfer(&AxiDma, (UINTPTR) RxBufferPtr,MAX_PKT_LEN,
XAXIDMA_DEVICE_TO_DMA);
if (Status != XST_SUCCESS) {
    return XST_FAILURE;
}
/* Flush the buffers before the DMA transfer, in case the Data Cache is enabled */
Xil_DCacheFlushRange((UINTPTR)RxBufferPtr, MAX_PKT_LEN);
while(!TxDone && !RxDone && !Error);
    if(Error){
        xil_printf("Testing Failed.");
        goto Done;
}
/* Test finished, check data */
Status = CheckData(MAX_PKT_LEN, 0xC);
if (Status != XST_SUCCESS) {
    xil_printf("Data check failed\r\n");
    goto Done;
}
xil_printf("Successfully Ran AXI DMA Loop\r\n");
/* Disable TX and RX Ring interrupts and return success */
DisableIntrSystem(&Intc, TX_INTR_ID, RX_INTR_ID);
Done:xil_printf("--- Exiting main() --- \r\n");
return XST_SUCCESS;

```