PRE-LAB 2: General Purpose IO (GPIO)

Introduction

This pre-lab will create a system which contains an instance of an AXI GPIO (General Purpose Input/Output) port. The processor will send information to the programmable logic fabric through the GPIO. The GPIO is configured as an output port and will be connected to the 8 user LEDS on the ZC702 board. An AXI Interconnect will be used to connect the AXI GPIO to the processor via one of the available General Purpose AXI Master Ports contained within the processor section of the XC7020 chip. Figure 1 shows a block diagram of the system.

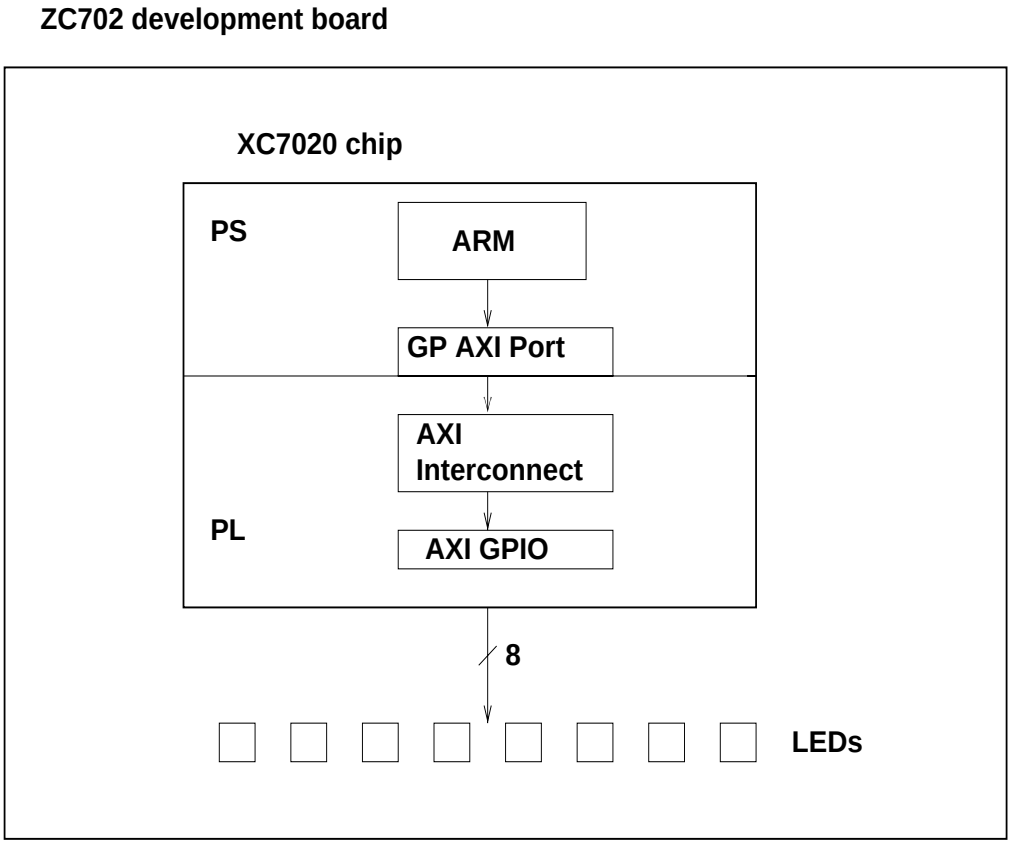

Figure 1: System with processor and GPIO.

Procedure

Setting up the user Linux environment:

1. Setup the Linux environment to run the Xilinx tools by entering the following command from the Linux prompt:

source /CMC/tools/xilinx_14.7/14.7/ISE_DS/settings64_CMC_central_license.csh

2. Create a subdirectory to hold the files related to this lab and launch the PlanAhead software.

cd COEN317_LABS
mkdir Lab2_Pre
cd Lab2_Pre

ted@happy Lab2_Pre 3:32pm > planAhead &

Create a new project from PlanAhead:

3.1 From the PlanAhead window, select:

File -> New Project

3.2 A window pops up, select. Next Give your project a name (i.e. lab2)

3.3 RTL Project should be selected. Keep the selection.

Select Next

3.4 We do not need to specify any sources at this time, however, choose the target language as VHDL in the drop-down box:

Target language: VHDL
Select Next

3.5 We do not need to specify any IPs either

Select Next

3.6 This system will make use of a User Constraints File (UCF) to specify that the GPIO output port signals are to be connected to specific pins of the Zynq chip which are connected to the LEDs. Prior to this step, use a text editor to create a file (called lab2_pre.ucf) which contains the following:

```
# This a comment line.
# axi_gpio_for_C LEDS on ZC702
NET  axi_gpio_for_C_pins<0>    LOC = E15;
NET  axi_gpio_for_C_pins<1>    LOC = D15;
NET  axi_gpio_for_C_pins<2>    LOC = W17;
NET  axi_gpio_for_C_pins<3>    LOC = W5;
NET  axi_gpio_for_C_pins<4>    LOC = V7;
NET  axi_gpio_for_C_pins<5>    LOC = W10;
NET  axi_gpio_for_C_pins<6>    LOC = P18;
NET  axi_gpio_for_C_pins<7>    LOC = P17;
```

In the **Add Constraints** window, select **Add files** and navigate to select the lab2.ucf you had previously created. Select Next to continue.

3.7 Select the ZYNQ-7 ZC702 Evaluation Board for our project:

Select Boards
Scroll down the list and choose ZYNQ-7 ZC702 Evaluation Board
Select Next

3.8 Review the settings and select Finish

Create an embedded processor project with the Add Source wizard:

We will add sources to the newly created empty project.

4.1 On the left panel under Project Manager, Select Add Sources.

4.2 Choose Add or Create Embedded Sources and select Next.

4.3 Select Create Sub-design
A window should have popped up asking for a module name. Name it "system".

Module name: system
Select OK
Select Finish

Designing the system in XPS:

After step 4.3 is performed, XPS will be automatically started and will be used to build the system and add an instance of an AXI GPIO.

5.1 A window pops up and asks:
This project appears to be a blank zynq project.
Do you want to create a Base System using the BSB Wizard?

Click Yes

5.2 The BSB Wizard asks for additional settings. The AXI System should be selected by default with no other parameters.

Click OK

Verify the Zynq Processing System 7 is selected in "Select a System".

Click Next

Remove the GPIO_SW and LEDs_4Bits Peripherals because they are not needed.

Click on "Select All", then "Remove"
Click Finish

5.3 An instance of an AXI GPIO will be added to the system. Expand the **General Purpose IO** tab on the left hand side under **IP Catalog** (refer to Figure 2) and double click the **AXI General Purpose IO**

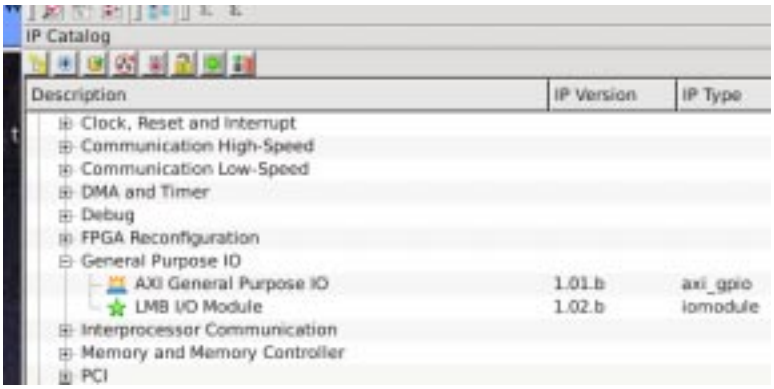

Figure 2: Adding an AXI GPIO.

Click Yes when prompted for:

Do you want to add one axi_gpio 1.01.b IP instance to your design?

Rename the Component Instance Name to axi_gpio_for_output in the XPS Core config window (not required, but it is helpful to give relevant names when there are more than one component of a given type). Note further that in the XPS Core config window there is an option to Enable Channel 2 of the GPIO but this lab makes use of only Channel 1. We will specify the bit width of Channel 1 as 8 bits by expanding Channel 1 and specifying the GPIO Data Channel Width to 8. Select OK. Refer to Figure 3.

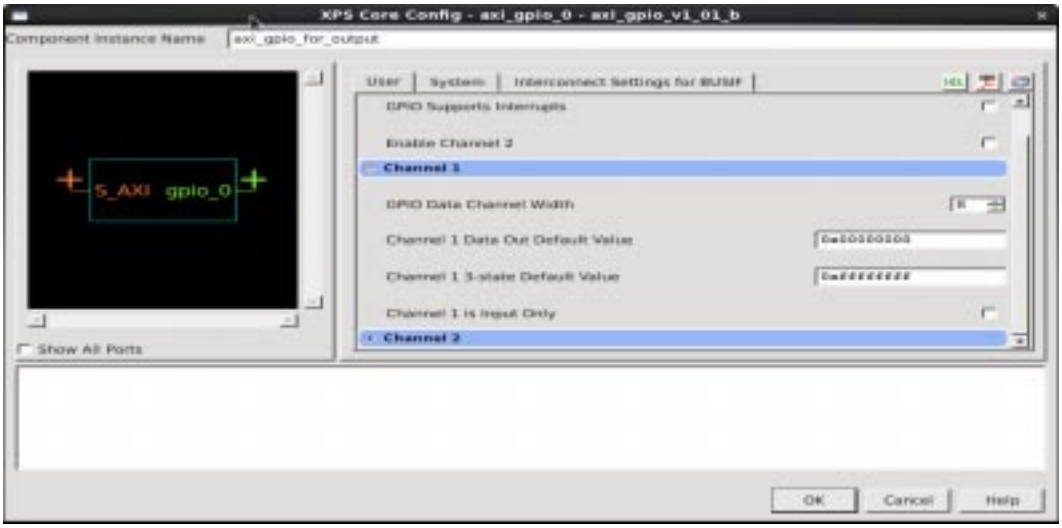

Figure 3: Specifying the data width.

5.4 In the Instantiate and Connect IP window which will appear, the choice:

Select processor instance to connect to; XPS will make the Bus Interface connection, assign the address, and make IO ports external

will be selected as the default choice. Leave it as the default choice. Ensure also that processing_system_7 is the listed processor. Select OK.

XPS allows also for user-defined connections which are useful when designing more complex systems involving many different IP cores. You will note that the following messages will be displayed in the Console window of XPS:

Address Map for Processor processing_system7_0
(0x41200000-0x4120ffff) axi_gpio_for_output axi_interconnect_1
INFO:EDK - Do connection by refering to bus interface of a processing_system7_0 in design
INFO:EDK - Create new axi_interconnect IP instance axi_interconnect_1
INFO:EDK - Connect clock port M_AXI_GP0_ACLK to processing_system7_0_FCLK_CLK0
INFO:EDK - Connect bus interface S_AXI to axi_interconnect_1
INFO:EDK - Connect clock port S_AXI_ACLK to processing_system7_0_FCLK_CLK0
INFO:EDK - Successfully did connection by refering to M_AXI_GP0 bus interface in processing_system7_0
INFO:EDK - External IO port grouping doneINFO:EDK - Generate address successfully
INFO:EDK - Successfully finished auto bus connection for IP instance: axi_gpio_for_output

XPS has automatically connected the AXI GPIO port which is instantiated inside the programmable logic fabric to an instance of AXI Interconnect (also within the fabric). The processor is act-

ing as the Master of the GPIO (which is the Slave). The processor will communicate with the GPIO through one of the General Purpose AXI Master ports (referred to as M_AXI_GP0) via an AXI Interconnect. XPS provides a graphical representation of these Master-Slave AXI connections.

To view these graphical connections, select the **Bus Interfaces** tab in the top middle portion of the XPS window indicated in Figure 4.

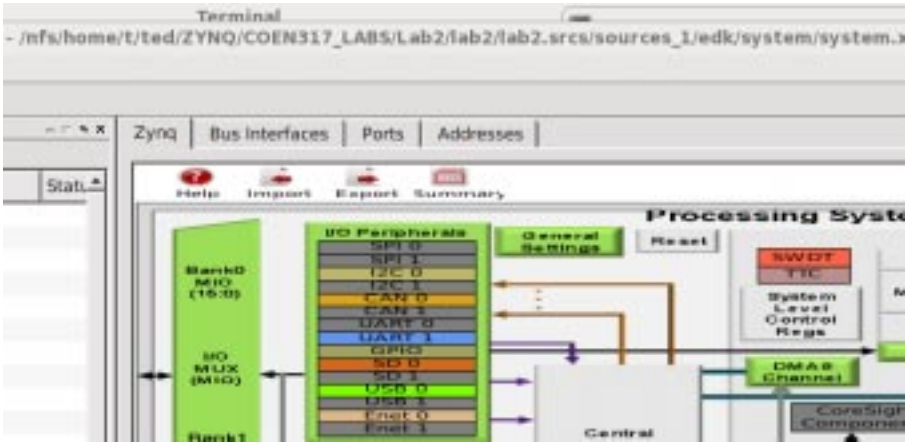

Figure 4: Bus Interfaces Tab.

XPS will now display the Bus Interface view of the designed system as shown in Figure 5.

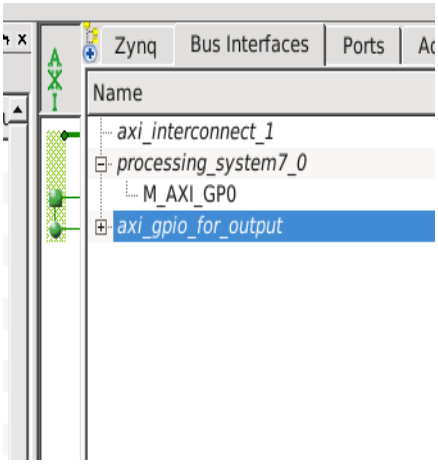

Figure 5: Bus Interface connections.

Figure 5 indicates that our system consists of three instances:

- an axi_interconnect (named axi_interconnect1)
- the ARM processor (named processing_system7_0)

• an AXI GPIO (which we explicitly added and renamed as axi_gpio_for_output)

The processors M_AXI_GP0 port (one of the two general purpose master AXI ports available) is connected to the GPIO instance. In this figure, a square indicates the instance which is acting as the Master, and a circle represents a slave device.

Selecting the **Ports** tab (instead of the Bus Interfaces) lists the actual connection by signal names. Refer to Figure 6 (it may be necessary to select the small square (with the +/- inside of it) to expand the list of ports associated with the instance).

5.5 We will now make the output pin of the GPIO external. This will cause it to appear as an port in the top-level system_stub.vhd wrapper. This is required since we are applying constraints on this port (through the UCF file) and only top-level ports may appear in a UCF file.

Select the **Ports** tab in the System Assembly View.

Expand axi_gpio_for_output and its (IO_IF) gpio_0
Right-click GPIO_IO and select No Connection
Right-click GPIO_IO_O and select Make External

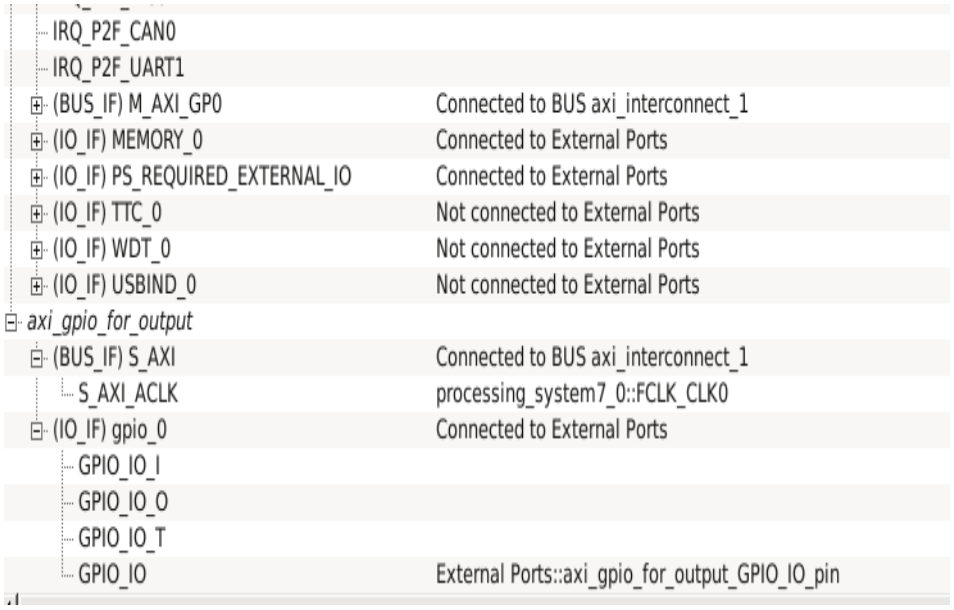

Figure 6: Port connections.

As indicated in Figure 6, the (BUS_IF) M_AXI_GP0 port of the processor is connected to the AXI bus through the axi_interconnect1 instance. The S_AXI (Slave_AXI) port of the axi_gpio_for_output instance is also connected to the bus through this axi_interconnect1. Scroll up through the list to view the connections for the FCLK_FCLK0 port of the processing_system7 instance as shown in Figure 7:

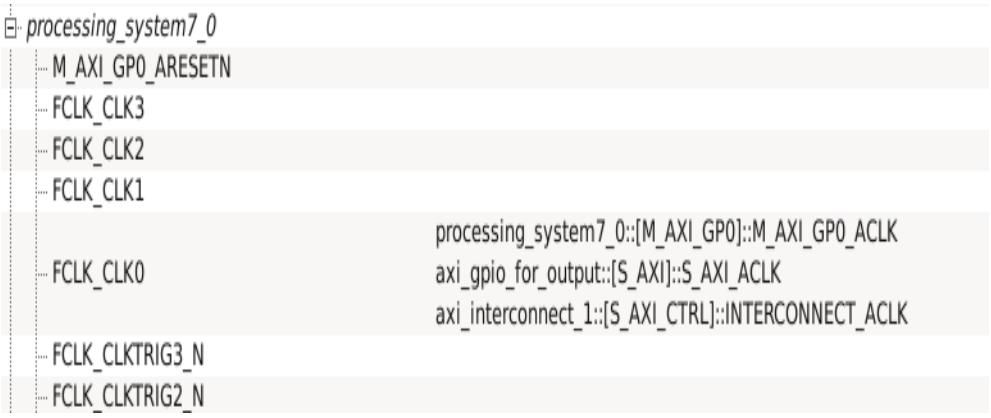

Figure 7: FCLK0 port connections.

AXI is a synchronous bus protocol, it requires a clock input. XPS has automatically made the necessary clock connections in the three instances of our tools to one of the 4 clocks (FCLK_CLK0, FCLK_CLK1, CLK_CLK2,FCLK_CLK3) available in the processor.

5.6 Collapse axi_gpio_for_output and expand the **External Ports** and rename the axi_gpio_for_output_GPIO_IO_O_pin to axi_gpio_for_C_pins. Refer to Figure 8.

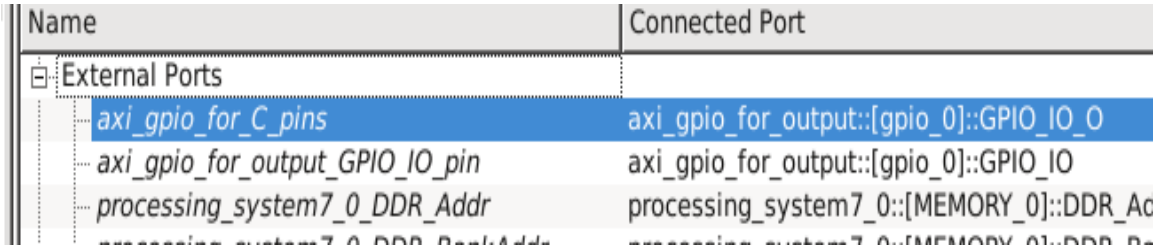

Figure 8: Renaming the pin.

Although not required, it is convenient to rename the pin to some appropriate name (relevant to the system being designed).
Since the GPIO_IO_O pin was made external and since we renamed it to axi_gpio_for_C_pins, this name (axi_gpio_for_C_pins) will appear as on OUT port in the top-level VHDL wrapper (system_stub.vhd).

5.7 We have now completed building our project in XPS and are ready create the top-level VHDL code from Project Manager in PlanAhead. Before closing the XPS project, it is a good design practice to perform a Design Rule Check from XPS to ensure that there are no errors. From XPS select Project -> Design Rule Check. Observe the messages listed in the Console window of XPS:

```
Running system level DRCs...

Performing System level DRCs on properties...

Running DRC Tcl procedures for OPTION SYSLEVEL_DRC_PROC...
ZynqConfig: Terminated for tcl mode
Done!
```

5.8 Close the XPS tool by selecting:

File -> Exit

Closing XPS restores the PlanAhead window to the foreground.

Exporting the Hardware to SDK:

6.1 Right click the system(system.xmp) design source in the Project Manager pane of the main PlanAhead window and select **Create Top HDL**.

6.2 Synthesize the design by selecting **Run Synthesis** from the **Synthesis** tab in the Project Manager of PlanAhead. Wait until Synthesis is complete.

6.3 When the Synthesis Completed window appears, implement the design by selecting OK (make sure that **Run Implementation** has been selected). There may be 3 warnings concerning constraint locations of PS_PORB_IBUF which may be ignored as they do not affect the system. Click OK to close the warnings.

6.4 Upon completion of implementation, a window will appear prompting Generate Bitstream. Select OK. If the following error is reported during the Generate Bitstream procedure:

ERROR:Bitgen:342 - This design contains pins which have locations (LOC) that are not user-assigned or I/O Standards (IOSTANDARD) that are not user-assigned. This may cause I/O contention or incompatibility with the board power or connectivity affecting performance, signal integrity or in extreme cases cause damage to the device or the components to which it is connected. To prevent this error, it is highly suggested to specify all pin locations and I/O standards to avoid potential contention or conflicts and allow proper bitstream creation. To demote this error to a warning and allow bitstream creation with unspecified I/O location or standards, you may apply the following bitgen switch: -g UnconstrainedPins:Allow

then it will be necessary to set the -g switch. Under the **Program and Debug -> Bitstream Settings** and in the More Options field type in:

-g UnconstrainedPins:Allow

Press enter after you type in the option and then select Apply followed by OK. Refer to Figure 9.

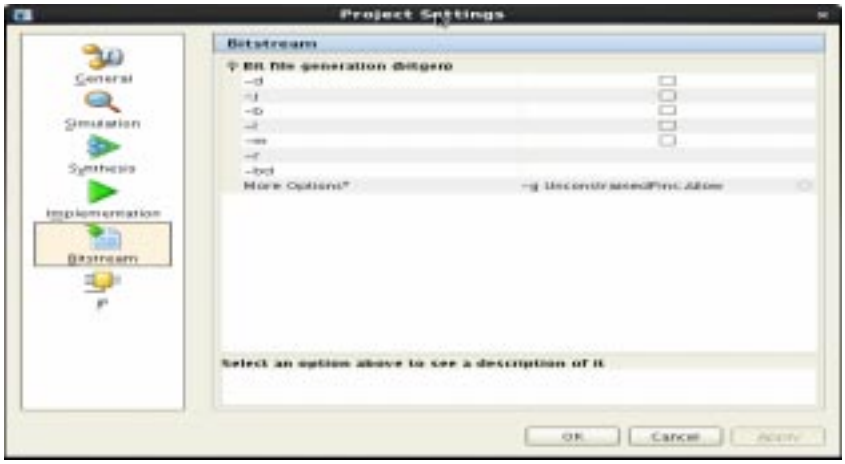

Figure 9: Setting Bitgen options.

6.5 Make sure that the power cable, the Platform Cable USB II and USB cable are attached to the board. Ask your TA to verify that the cables are properly connected. Turn on the power switch. Ensure that the status LED on the PLatform Cable USB II is green once power has been applied to the board (without power, it should be amber). Ensure that the 5 DIP switches on the blue DIP switch SW16 are all in the down position (the number printed on the switch defines the "top" position).

Choose **Launch Impact** and select OK. A window may appear prompting you to save the file. Select Cancel to close this window.

In the Impact window (Figure 10) right click the Target (xc7z020) and select **Program** to program the device with the system_stub.bit file.

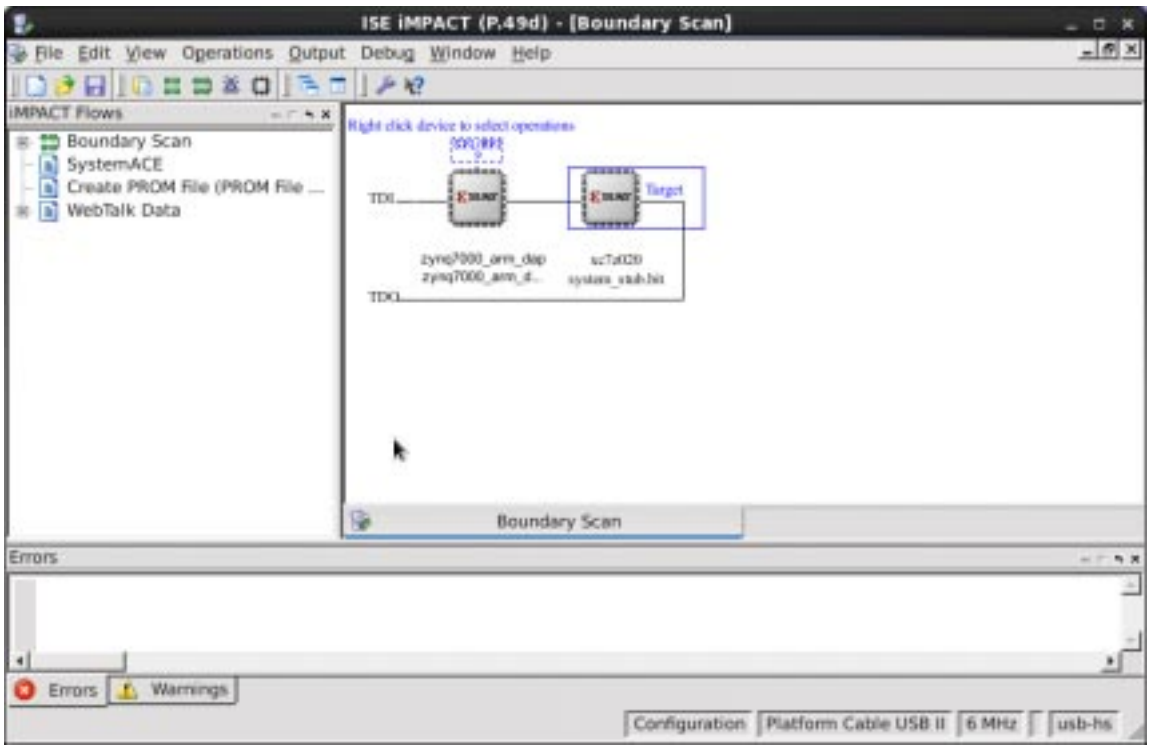

Figure 10: Selecting the device to program.

6.6 Export the hardware to SDK, from the PlanAhead window, select File -> Export -> Export Hardware for SDK. Enable the **Launch SDK** check box and select OK. SDK will open and we will now create a new application project.

Using SDK to create an application project:

7.1 From SDK, select File -> New Application Project.

7.2 In the New Project window, specify a project name (prelab2), C++ as the programming language and standalone as the OS platform. Select Finish to continue. The New Project Template window will appear with Empty Application selected as the available template. Leave it selected and select Finish to complete the project.

7.3 Edit the main.cc program so that it contains the following:

```
#include “xparameters.h”
#include “xil_types.h”
#include “xgpio.h”
#include “xil_io.h”
#include “xil_exception.h”
```

```
#include <iostream>
using namespace std;

int main()
{

   static XGpio GPIOInstance_Ptr;
   int xStatus;

   cout << “#### Application Starts ###” << endl;


   //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   //Step-1: AXI GPIO Initialization
   //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 xStatus = XGpio_Initialize(&GPIOInstance_Ptr,
XPAR_AXI_GPIO_FOR_OUTPUT_DEVICE_ID);

if(xStatus != XST_SUCCESS)
{
   cout << “GPIO A Initialization FAILED” << endl;
   return 1;
}


   //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   //Step-2: AXI GPIO Set the Direction
   //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   //XGpio_SetDataDirection(XGpio *InstancePtr, unsigned Channel,
   // u32 DirectionMask);
   //we use only channel 1, and 0 is the  parameter for output
   // 1 is the  parameter for input

XGpio_SetDataDirection(&GPIOInstance_Ptr, 1, 0);

int ted_data; // use int since cin does not know how to
              // handle u8 , this is the data which will be sent
              // to the GPIO using the XGpio_DiscreteWrite
              //  function call

for(int i = 0 ; i < 9 ; i++)
{
 cout << “Enter the data to be sent to the GPIO” << endl;
 cin >> ted_data;
 // cast the data to u8 and send i to the GPIO
```

```
 XGpio_DiscreteWrite(&GPIOInstance_Ptr, 1, (u8) ted_data);
}

cout << “End of program” << endl;

return 0;
}
```

Save the modifications made to main.cc by entering Control-S while the cursor is inside the editor pane, or by entering File -> Save. SDK will compile the program and create the executable file (main.elf).

7.4 Establish a serial terminal connection with the host computer and board:

Window -> Show View -> Terminal

Select the green connect button (it resembles a green ’N’ with a dot on each end)

In the popup window, specify the settings as:

Connection Type: Serial
Port: /dev/ttyUSB0
Baud Rate: 115200
Data Bits: 8
Stop Bits: 1
Parity: None
Flow Control: None
Timeout(sec): 5

Select OK.

7.5 Run the executable file on the board. Right-click the **lab2** folder and choose Run As -> Run Configurations. In the popup window, right click Xilinx C/C++ ELF and choose **New** (This only needs to be created once). Leave the default settings and click **Run**. Open the SDK Terminal to view the output.

Comments on the program:

The program does the following :

• it initializes the GPIO port with a call to a Xilinx provided function called `XGpio_Initialize()`.

•it sets the GPIO port to be an output port by calling another Xilinx provided function called `Gpio_SetDataDirection();`

• it enters a for loop in which a user-entered value is sent to the GPIO port through a call to the `XGpio_DiscreteWrite()` function.

The prototypes for these functions are found in the xgpio.h file. The actual functions are defined in the file xgpio.c which is part of the board support package created by SDK when the project was initially created. The board support package is part of the project directory and is located in:

./prelab2/lab2.sdk/SDK/SDK_Export/lab2_bsp/ps7_cortexa9_0/libsrc/gpio_v3_00_a/src/

directory of the project.

These three functions all make use of a pointer to argument of data type XGpio.
The definition of this date type is found in xgpio.h file and is given as:

```
typedef struct {
        u32 BaseAddress;        /* Device base address */
        u32 IsReady;            /* Device is initialized and ready */
        int InterruptPresent;   /* Are interrupts supported in h/w */
        int IsDual;             /* Are 2 channels supported in h/w */
} XGpio;
```

The file xparameters.h contains #define compiler directives for some of the arguments used in these three function calls. For example, the `XGpioInitialize()` function has as its second argument XPAR_AXI_GPIO_FOR_OUTPUT_DEVICE_ID:

```
 xStatus = XGpio_Initialize(&GPIOInstance_Ptr,
XPAR_AXI_GPIO_FOR_OUTPUT_DEVICE_ID);
```

If you open the xparameters.h file from within SDK by double clicking on it in the Outline portion of the SDK window (located towards the right hand portion of the SDK window as shown in Figure 11) you will see that it contains the following #define directives:

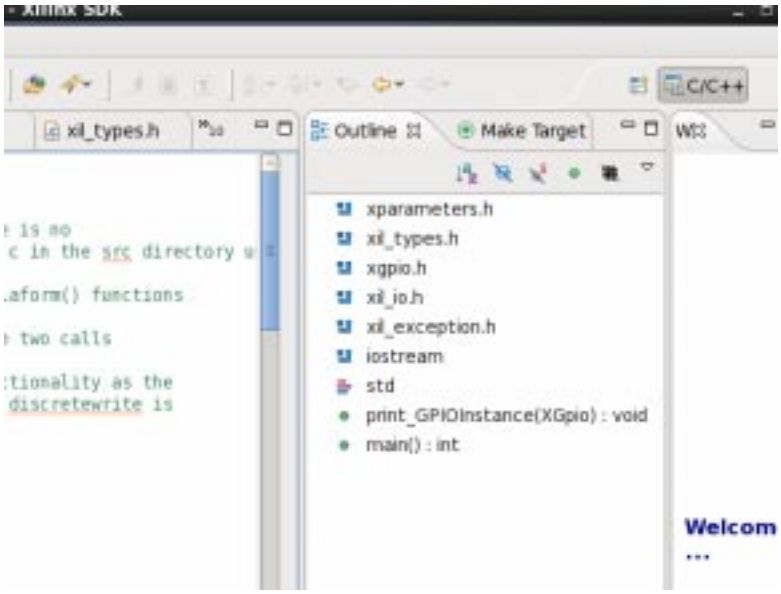

Figure 11: Outline pane of SDK.

```
/* Definitions for peripheral AXI_GPIO_FOR_OUTPUT */
#define XPAR_AXI_GPIO_FOR_OUTPUT_BASEADDR 0x41200000
#define XPAR_AXI_GPIO_FOR_OUTPUT_HIGHADDR 0x4120FFFF
#define XPAR_AXI_GPIO_FOR_OUTPUT_DEVICE_ID 0
#define XPAR_AXI_GPIO_FOR_OUTPUT_INTERRUPT_PRESENT 0
#define XPAR_AXI_GPIO_FOR_OUTPUT_IS_DUAL 0
```

The values in the #define statements originate from the values assigned to the GPIO device when it was instantiated into the system from within XPS. To verify this, open XPS from PlanAhead by doubling clicking on the system.xmp Design Source in the Project Manager pane of PlanAhead. When XPS opens, select the **Addresses** tab and examine the value of the Base Address associated with the axi_gpio_for_output instance - it is the same value as that given in the #define XPAR_AXI_GPIO_FOR_OUTPUT_BASEADDR. Refer to Figure 12.

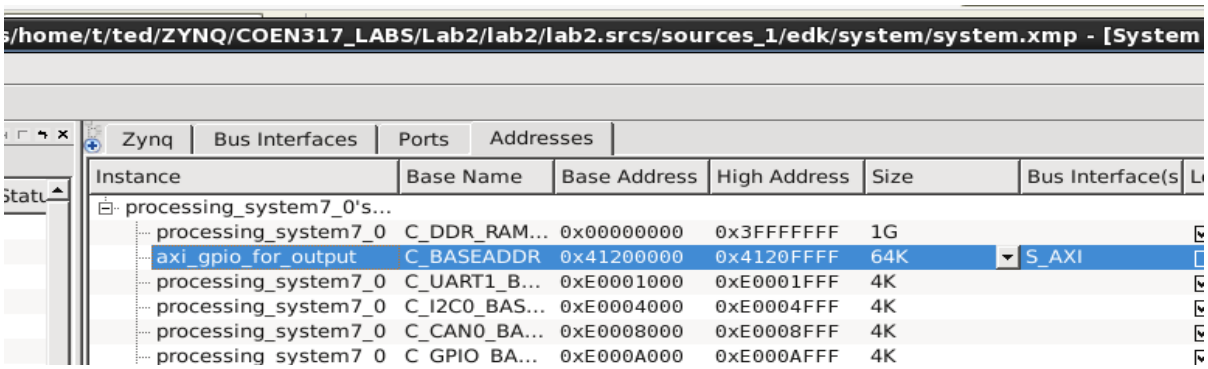

Figure 12: Address Tab in XPS.

The embedded systems we are designing with XPS are of the “memory-mapped IO” type. In systems where the IO devices are memory mapped, every device is assigned an address (or a range of addresses). These addresses are distinct from the range of addresses assigned to the RAM memory. Each device has as part of it’s internal hardware address decoding logic. This address decoding logic will “activate” the device whenever an address associated with it is presented on the address bus.

An alternative method of communicating with a GPIO port

While one may always make use of the three Xilinx provided functions to interface with a GPIO port, there is a more direct manner in which to configure and read/write to a GPIO. This method makes use of the addresses assigned to the GPIO and knowledge of the GPIO obtained by consulting the datasheet for the AXI GPIO. According to the Xilinx provided LogiCORE IP AXI GPIO datasheet (DS744 July 25, 2012) “there are four internal registers in the AXI_GPIO design. The internal registers of the AXI GPIO are at a fixed offset from the base address and are byte accessible”[1]. Table 1 summarizes the addresses of these 4 registers.

Table 1: GPIO Registers and addresses [1]

Base Address + Offset (hex)	Register Name	Description
C_BASEADDR + 0x00	GPIO_DATA	Channel 1 AXI GPIO Data Register
C_BASEADDR + 0x04	GPIO_TRI	Channel 1 AXI GPIO 3-state register (used to control the direction of each bit in the corresponding data register)
C_BASEADDR + 0x08	GPIO2_DATA	Channel 2 AXI GPIO Data Register

Table 1: GPIO Registers and addresses [1]

Base Address + Offset (hex)	Register Name	Description
C_BASEADDR + 0x0c	GPIO2_TRI	Channel 2 AXI GPIO 3-state register (used to control the direction of each bit in the corresponding data register)

The data sheet explains that “the AXI GPIO data register is used to read the input ports and write to the output ports. There are two GPIO data registers (GPIO_DATA and GPIO2_DATA), one corresponding to each channel. The channel 1 data register (GPIO_DATA) is always present, the channel 2 data register (GPIO2_DATA) is present only if the core is configured for dual channel” [1]. Recall that when we added the GPIO to our system in XPS, it was configured with only 1 channel. Thus our GPIO port has only one data register. To configure the channel as either an input or output channel (this is what the Xilinx function /XGpio_SetDataDirection() ultimately performs), the data sheet specifies “the AXI GPIO 3-state register is used to configure the ports dynamically as input or output. When a bit within this register is set, the corresponding I/O port is configured as an input port. When the bit is reset, the corresponding I/O port is configured as an output port” [2] . The data sheet provides the following useful “User Application Tips” :

“For input ports use the following steps:

a. Configure the port as input by writing the corresponding bit in the GPIOx_TRI register with the value 1.
b. Read the corresponding bit in the GPIOx_DATA register.

For output ports use the following steps:

a. Configure the port as output by writing the corresponding bit in the GPIOx_TRI register with the value 0.
b. Write the corresponding bit in the GPIOx_DATA register” [3] .

The following C++ program uses this more direct approach to perform the same task as the first program:

```
#include “xparameters.h”
#include “xil_types.h”
#include “xgpio.h”
#include “xil_io.h”
#include “xil_exception.h”

#include <iostream>
using namespace std;

int main()
```

```
{

// declare a pointer to the base address of the GPIO instance
// the value of XPAR_AXI_GPIO_FOR_OUTPUT_BASEADDR
// is defined through a #define directive in the
// xparameters.h file. The value is determined by the
// address generated by XPS when the GPIO was added (0x41200000 )

u32* gpio_ptr =  (u32*) XPAR_AXI_GPIO_FOR_OUTPUT_BASEADDR  ;

// configure the port as output by writing a 0 into the
// port data direction register found at offset 4 bytes from
// the base address ( 4 bytes = 1 u32 offset in pointer arithmetic)

*(gpio_ptr + 1 )  = 0;


int ted_data ; // use int to input data as
               // cin does not know how to
               // handle u8 datatype

for(int i = 0 ; i < 9 ; i++)
{
 cout << “Enter the data to be sent to the GPIO port  “ << endl;
 cin >> ted_data;
 cout << “The data is “ << ted_data << endl;

// write the data to the GPIO port  (at offset 0)
// cast the data as a u8

*gpio_ptr =  (u8) ted_data;
}


 cout << “End of program” << endl;
// cleanup_platform();
    return 0;
}
```

To verify that this program works, rename the original main.cc file to some other name such as main.cc.backup and then enter the above code and save it as main.cc . Compile,download and run this new version to verify it functions in the same manner as the first version.

The author would like to thank Michael Segev for discovering this alternative method of accessing I/O devices.

Questions:

1. The following data types are defined in the xgpio.h file:

```
typedef unsigned char u8;
typedef unsigned short u16;
typedef unsigned long u32;

typedef struct {
u32 BaseAddress; /* Device base address */
u32 IsReady; /* Device is initialized and ready */
int InterruptPresent /* Are interrupts supported in h/w */
int IsDual; /* Are 2 channels supported in h/w */
} XGpio;
```

Write a function which receives an argument of type `XGpio` and prints out the values of the fields of the argument. Invoke your function before and after the call to `XGpio_Initialize()` in the version of the program which makes use of the three Xilinx functions. From the values of the output produced by the print function, what can you conclude that the `XGpio_Initialize()` function performs?

2. Redesign your system to include a second instance of an AXI GPIO which will be used as an input port connected to pushbutton switch 5 (SW5) on the ZC702 board. The software program will continuously poll the state of this switch, and when the user presses the switch, the software will then ask the user to enter a number which will then be sent to the GPIO output port to drive the LEDs. The steps which need to be performed are:

• add an input GPIO for SW5 and select its width to be 1 bit, rename is axi_gpio_for_SW5

• select the **Ports** tab in the System Assembly view of XPS and expand axi_gpio_for_SW5 and its (IO_IF) gpio_O.

Right click GPIO_IO and select No connection.
Right click GPIO_I and select Make External

• collapse axi_gpio_for_SW5 and expand **External Ports**. Rename axi_gpio_for_SW5_GPIO_IO_I_pin to axi_gpio_for_SW5_pin.

• Add the following entry to the .ucf file:

NET axi_gpio_for_SW5_pin<0> LOC = G19;

Note that when the project was created in PlanAhead, the lab2.ucf was copied into a directory within the project. You will have to modify the lab2.ucf which was copied into the project directory found in:

./lab2/lab2.srcs/constrs_1/imports/Lab2/lab2.ucf
.
• Resynthesize, reimplement, generate a new bitstream, program the FPGA with the new bitstream using Impact, and export hardware to SDK.

• In SDK create a new application project for lab5_sw5 (this will create a new BSP with the updated xparameters.h file).

• Write a new C++ program to :

```
configure the output GPIO port
configure the input GPIO port

 while ( the input port == 0 )  // when  SW5 is not pressed it
                                // produces a logic '0'
 {
   // wait for the user to press SW5
 }

   enter the data to be sent to the LEDs via the output GPIO
   send the data to the output GPIO port
```

Create a New Run Configuration and download and run the program.

References

1. LogiCORE IP AXI GPIO (v1.10.b) Product Specification, Xilinx Inc. DS744 July 25, 2012, p 8.

2. LogiCORE IP AXI GPIO (v1.10.b) Product Specification, Xilinx Inc. DS744 July 25, 2012, p. 9.

3. LogiCORE IP AXI GPIO (v1.10.b) Product Specification, Xilinx Inc. DS744 July 25, 2012, p. 12-13.

T. Obuchowicz/R. Lee
July 2014

Revision history:

- Sept. 8, 2016 : modifed to become a pre-lab for Lab 2 (background information on using GPIO ports) , modified for sourcing /CMC/tools/xilinx_14.7/14.7/ISE_DS/ settings64_CMC_central_license.csh