COEN 317 Lab 4 (UJ-X)

by

Mamadou Diao Kaba

William Picard

Performed on

March 19th, 2024.

"I certify that this submission is my original work and meets the Faculty’s Expectations of Originality."

Introduction

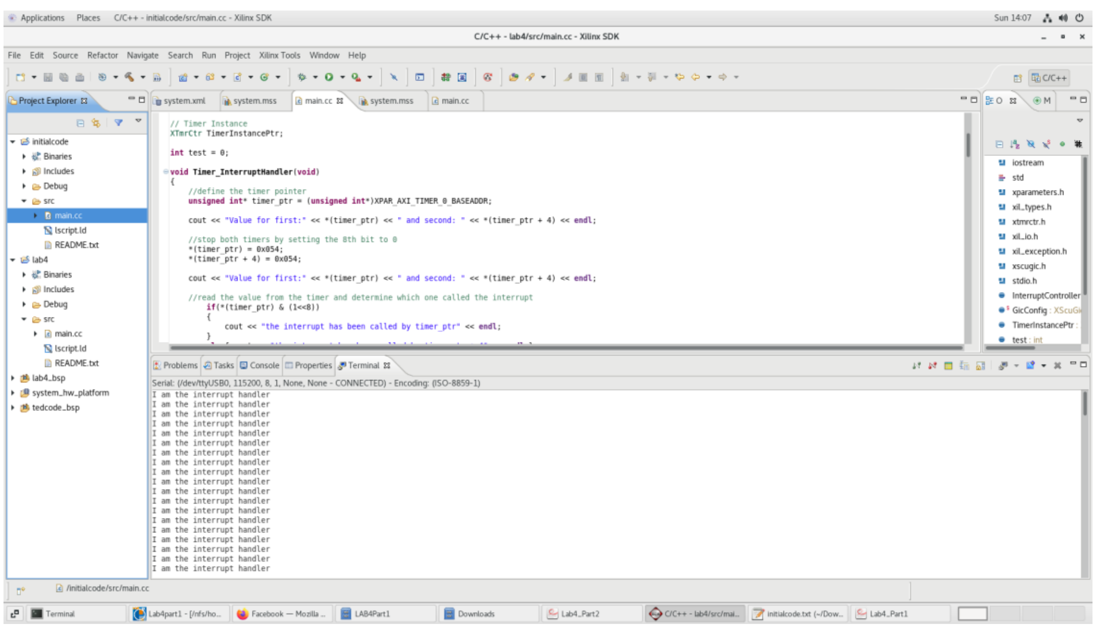
In this initial part of the laboratory session, we delve into understanding the interrupt structure of the ARM processor. Interrupts play a crucial role in processor functionality, enabling the processor to pause its ongoing tasks to address incoming interrupt requests, typically from higher-priority services. Thus, prompt servicing of these requests is paramount. In the preceding lab session, we explored the AXI Timer, a module primarily tasked with clock sequence counting. We reviewed three distinct operational modes for the timer. In this session, our focus shifts to understanding how the timer's two registers facilitate interrupt handling based on timer intervals. Our practical tasks include generating a pulse on the external Interrupt port and establishing a connection between this port and the Interrupt request pin of the ARM processor.

The second part of the lab introduces us to the AXI CDMA (Direct Memory Access). This component facilitates high-speed data transfers from a memory source to a memory destination. Our objective is to configure the CDMA as a master device to oversee two high-performance slave ports, specifically the Master General Purpose ports. These ports will be interconnected using the AXI Interconnect module, thereby ensuring efficient data throughput between the AXI master and the DDR memory system of the processor. Furthermore, one of the lab inquiries involves determining the duration required for data transfer. To accomplish this, we'll employ the AXI Timer, a familiar module introduced in the third lab session, which counts the clock sequence and offers various operational modes.

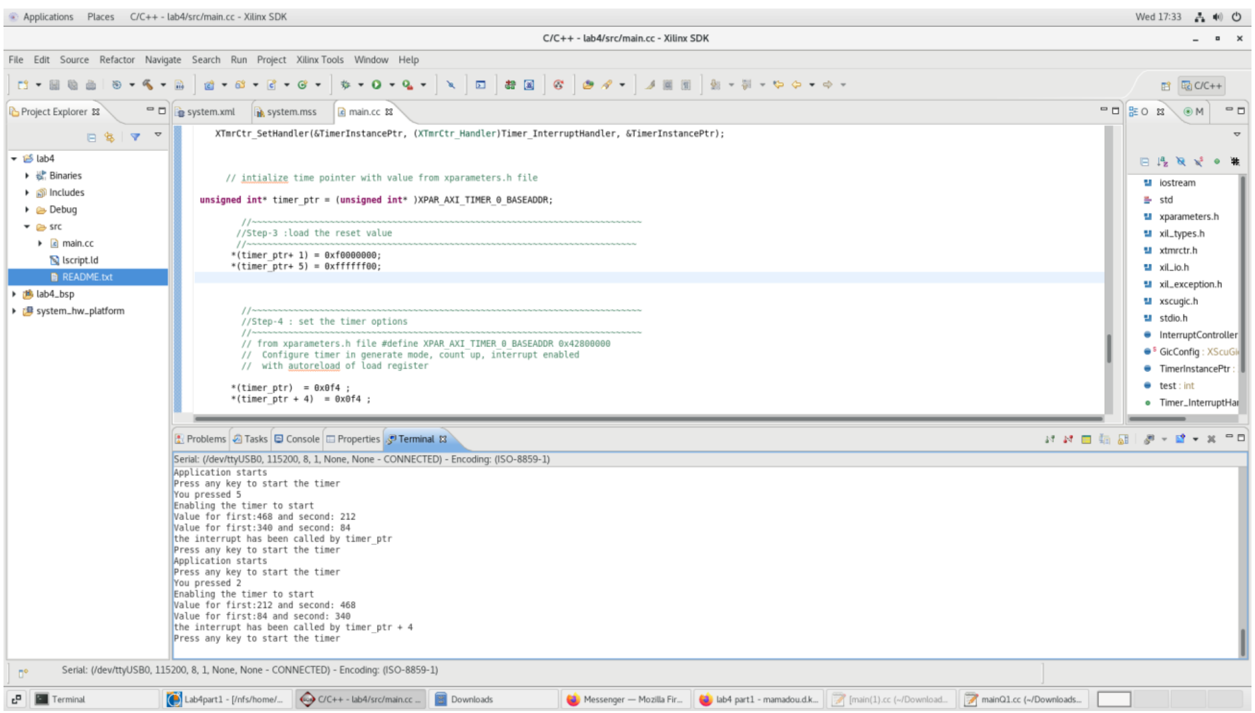
Results and Questions

Part 1

We can examine the main.cc code provided in Appendix A, Code 1 to gain insight into how the software configures the interrupt controller and manages interrupts. Initially, there are three global timer variables: an instance for the interrupt controller, configuration for the controller, and a timer instance. These variables are declared globally to enable access from functions outside of the main function. Furthermore, there exist three functions responsible for handling interrupts, initializing the interrupt controller driver, and connecting the interrupt controller. While I won't delve deeply into the specifics of these functions, we'll focus on the significant sections of the code worth highlighting. In steps 3 and 4, the timer pointer is set to specific values, which are determined from the AXI Timer manual referenced in the previous lab session. To determine the correct value, it's essential to identify which bit needs to be set to 1. Additionally, consulting the manual is necessary to ascertain the deassert value for the timer pointer. The provided output below showcases the result obtained from executing the code:



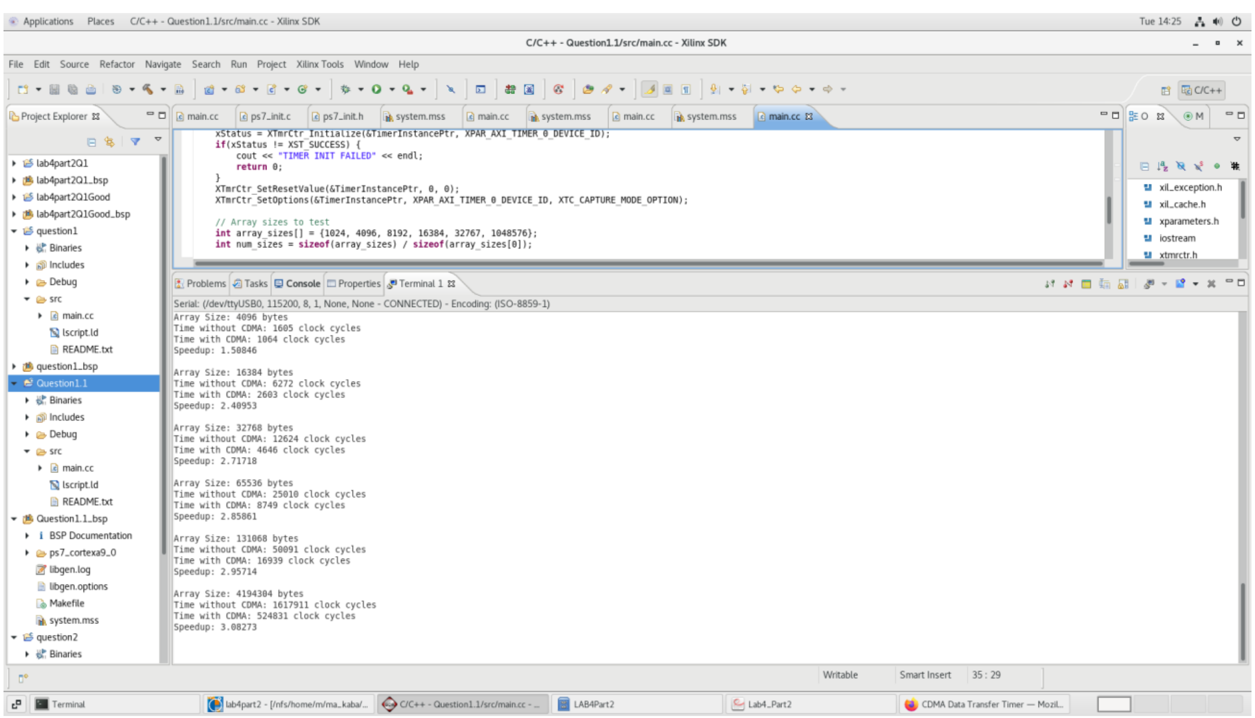
The AXI timer features only one interrupt signal despite containing two timers. Hence, it's crucial to discern which timer triggers the interrupt. Initially, the interrupt signal is determined as the logical OR of interrupts from both timers, identified using the TINT bit. To implement these necessary changes, adjustments are required in two sections of the previous code. Firstly, within the main function, the setup for the second timer needs to be established. Secondly, in the interrupt handler, modifications are made to halt the timers by resetting the 8th bit to 0, followed by identifying the timer generating the interrupt. The modified code is detailed in Appendix A, Code 2, and the ensuing output demonstrates the result obtained upon executing the modified code:



Part 2

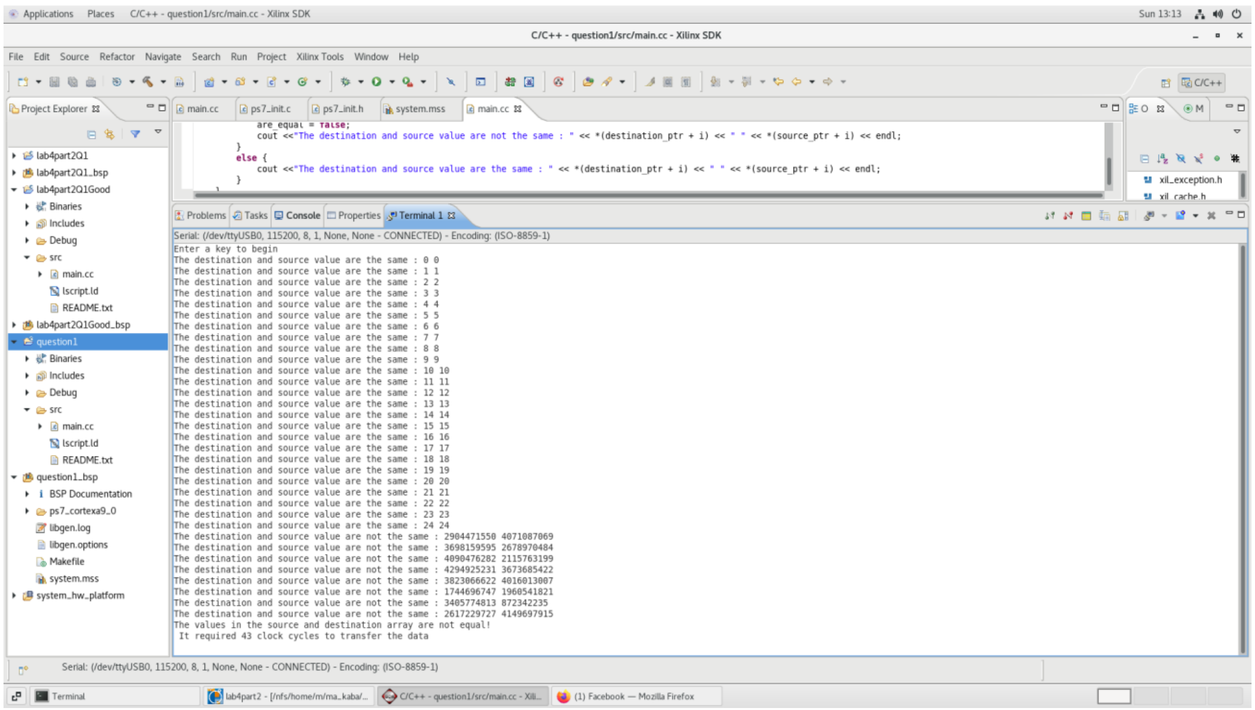
To set up the hardware for the first part, we follow the steps in the lab manual. We add a CDMA instance and two AXI interconnect instances. We manually connect them instead of using the processor system 7 manual connection. The lab manual explains different ways to make these connections, but we can simply click some boxes in the bus tab to designate ports as master or slave. Once the hardware is set up, we write a program in Xilinx SDK to transfer data between the source and destination. The lab manual gives us a step-by-step guide in pseudocode for this program. Question 1 asks us to use an AXI timer to find out how long it takes to transfer the data. So, we enhance the hardware by adding an AXI Timer/Counter instance. We can connect it manually or let the processor's system 7 handle it. To modify the program in the SDK, we refer to lab 3 where we used the timer to count cycles for a discrete write. Instead of a discrete write, we're now doing a data transfer with the CDMA. We modify the code from the first part by adding a timer variable, initializing it, setting it in capture mode, starting the timer and then stopping it. The code for question 1 is shown in Appendix A, Code 3.

This is the output we got:



For question 2, we investigate what happens if we try to write data beyond the maximum allowed bits in the BBT register. We can modify the code to write data beyond the limit and compare the source with the destination to see what happens. The code for question 2 is shown in Appendix A, Code 4.

When we run the code, we see that the destination data doesn't change when we write beyond the maximum allowed bits. This is because bits 25 to 32 are reserved and shouldn't be modified. Therefore, it showcases garbage values in those destinations:



Conclusion

In conclusion, through part 1 and 2 of lab 4, we successfully configured hardware components such as the CDMA and AXI interconnects to facilitate data transfer operations. By integrating these hardware elements into our programming environment using Xilinx SDK, we were able to develop a program that effectively transferred data between a source and destination. Leveraging an AXI timer allowed us to accurately measure the time required for data transfer, while further experimentation revealed the consequences of attempting to write data beyond the permissible limits in the BBT register. These hands-on activities not only enhanced our understanding of hardware configuration and programming techniques but also provided valuable insights into the robustness and limitations of the implemented hardware systems.

APPENDIX A

Code 1

```
#include <iostream>
using namespace std;
#include "xparameters.h"
#include "xil_types.h"
#include "xtmrctr.h"
#include "xil_io.h"
#include "xil_exception.h"
#include "xscugic.h"
#include <stdio.h>

/* Instance of the Interrupt Controller */
XScuGic InterruptController;

/* The configuration parameters of the controller */
static XScuGic_Config *GicConfig;

// Timer Instance
XTmrCtr TimerInstancePtr;

int test = 0;

void Timer_InterruptHandler(void)
{
    cout << "I am the interrupt handler" << endl;
}

int SetUpInterruptSystem(XScuGic *XScuGicInstancePtr)
{
        /*
        * Connect the interrupt controller interrupt handler to the hardware
        * interrupt handling logic in the ARM processor.
        */
        Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
        (Xil_ExceptionHandler) XScuGic_InterruptHandler,
        XScuGicInstancePtr);
        /*
        * Enable interrupts in the ARM
        */
        Xil_ExceptionEnable();
        return XST_SUCCESS;
}
```

```
int ScuGicInterrupt_Init(u16 DeviceId,XTmrCtr *TimerInstancePtr)
{
	int Status;
	/*
	* Initialize the interrupt controller driver so that it is ready to
	* use.
	* */

	GicConfig = XScuGic_LookupConfig(DeviceId);
	if (NULL == GicConfig)
	{
		return XST_FAILURE;
	}
	Status = XScuGic_CfgInitialize(&InterruptController, GicConfig,
	GicConfig->CpuBaseAddress);

	if (Status != XST_SUCCESS)
	{
		return XST_FAILURE;
	}
	/*
	* Setup the Interrupt System
	* */
	Status = SetUpInterruptSystem(&InterruptController);
	if (Status != XST_SUCCESS)
	{
		return XST_FAILURE;
	}

	// add this line to fix the interrupt only running once
      // Nov. 15, 2017
      // solution found by Anthony Fiorito on Xilinx Forum

	XScuGic_CPUWriteReg(&InterruptController, XSCUGIC_EOI_OFFSET,
XPAR_FABRIC_AXI_TIMER_0_INTERRUPT_INTR);

	/*
	* Connect a device driver handler that will be called when an
	* interrupt for the device occurs, the device driver handler performs
	* the specific interrupt processing for the device
	*/
	Status = XScuGic_Connect(&InterruptController,

XPAR_FABRIC_AXI_TIMER_0_INTERRUPT_INTR,

(Xil_ExceptionHandler)XTmrCtr_InterruptHandler,
	(void *)TimerInstancePtr);
	if (Status != XST_SUCCESS)
	{
		return XST_FAILURE;
	}
	/*
	* Enable the interrupt for the device and then cause (simulate) an
	* interrupt so the handlers will be called
	*/
	XScuGic_Enable(&InterruptController, XPAR_FABRIC_AXI_TIMER_0_INTERRUPT_INTR);
```

```
        return XST_SUCCESS;
}
int main()
{
  cout << "Application starts " << endl;
  int xStatus;

    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    //Step-1 :AXI Timer Initialization
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    xStatus = XTmrCtr_Initialize(&TimerInstancePtr, XPAR_AXI_TIMER_0_DEVICE_ID);
    if(XST_SUCCESS != xStatus)
    {
        cout << "TIMER INIT FAILED " << endl;
        if(xStatus == XST_DEVICE_IS_STARTED)
        {
            cout << "TIMER has already started" << endl;
            cout << "Please power cycle your board, and re-program the bitstream" << endl;
        }
        return 1;
    }

        //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        //Step-2 :Set Timer Handler
        //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    //
    // cast second argument to data type XTmrCtr_Handler since in gcc it gave a warning
    // and with g++ for the C++ version it resulted in an error

        XTmrCtr_SetHandler(&TimerInstancePtr, (XTmrCtr_Handler)Timer_InterruptHandler,
&TimerInstancePtr);

   // intialize time pointer with value from xparameters.h file

 unsigned int* timer_ptr = (unsigned int* )XPAR_AXI_TIMER_0_BASEADDR;

     //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    //Step-3 :load the reset value
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    *(timer_ptr+ 1) = 0xffffff00;

     //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     //Step-4 : set the timer options
     //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
     // from xparameters.h file #define XPAR_AXI_TIMER_0_BASEADDR 0x42800000
     //  Configure timer in generate mode, count up, interrupt enabled
     //  with autoreload of load register
    *(timer_ptr)  = 0x0f4 ;

        //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        //Step-5 : SCUGIC interrupt controller Initialization
        //Registration of the Timer ISR
        //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
        xStatus=
        ScuGicInterrupt_Init(XPAR_PS7_SCUGIC_0_DEVICE_ID, &TimerInstancePtr);
        if(XST_SUCCESS != xStatus)
        {
```

```
                cout << " :( SCUGIC INIT FAILED )" << endl;
                return 1;
        }

    // Beginning of our main code
        //We want to control when the timer starts
        char input;
        cout << "Press any key to start the timer" << endl;
        cin >> input ;
        cout << "You pressed "<<  input << endl;
  cout << "Enabling the timer to start" << endl;
    *(timer_ptr) = 0x0d4 ;   // deassert the load 5 to allow the timer to start counting
    // let timer run forever generating periodic interrupts
        while( 1)
    {
     //  // wait forever and let the timer generate periodic interrupts
    }
  return 0;
}
```

Code 2

```
#include "xparameters.h"
#include "xil_types.h"
#include "xtmrctr.h"
#include "xil_io.h"
#include "xil_exception.h"
#include "xscugic.h"
#include <stdio.h>
#include <iostream>
using namespace std;

/* Instance of the Interrupt Controller */
XScuGic InterruptController;

/* The configuration parameters of the controller */
static XScuGic_Config *GicConfig;

// Timer Instance
XTmrCtr TimerInstancePtr;

int test = 0;

void Timer_InterruptHandler(void)
{
        //define the timer pointer
        unsigned int* timer_ptr = (unsigned int*)XPAR_AXI_TIMER_0_BASEADDR;

        cout << "Value for first:" << *(timer_ptr) << " and second: " << *(timer_ptr + 4) << endl;

        //stop both timers by setting the 8th bit to 0
        *(timer_ptr) = 0x054;
        *(timer_ptr + 4) = 0x054;

        cout << "Value for first:" << *(timer_ptr) << " and second: " << *(timer_ptr + 4) << endl;
```

```
	//read the value from the timer and determine which one called the interrupt
	if(*(timer_ptr) & (1<<8))
	{
		cout << "the interrupt has been called by timer_ptr" << endl;
	}
	else{ cout << "the interrupt has been called by timer_ptr + 4" << endl;}

	//prompt user to enter a value to begin the timer
	char input;
	cout << "Press any key to start the timer" << endl;
	cin >> input ;
	cout << "You pressed "<<  input << endl;
	cout << "Enabling the timer to start" << endl;

	*(timer_ptr) = 0x0d4 ;
	*(timer_ptr + 4) = 0x0d4 ;

}

int SetUpInterruptSystem(XScuGic *XScuGicInstancePtr)
{
	/*
	* Connect the interrupt controller interrupt handler to the hardware
	* interrupt handling logic in the ARM processor.
	*/
	Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
	(Xil_ExceptionHandler) XScuGic_InterruptHandler,
	XScuGicInstancePtr);
	/*
	* Enable interrupts in the ARM
	*/
	Xil_ExceptionEnable();
	return XST_SUCCESS;
}

int ScuGicInterrupt_Init(u16 DeviceId,XTmrCtr *TimerInstancePtr)
{
	int Status;
	/*
	* Initialize the interrupt controller driver so that it is ready to
	* use.
	* */

	GicConfig = XScuGic_LookupConfig(DeviceId);
	if (NULL == GicConfig)
	{
		return XST_FAILURE;
	}
	Status = XScuGic_CfgInitialize(&InterruptController, GicConfig,
	GicConfig->CpuBaseAddress);

	if (Status != XST_SUCCESS)
	{
		return XST_FAILURE;
	}
	/*
	* Setup the Interrupt System
```

```
                    * */
                    Status = SetUpInterruptSystem(&InterruptController);
                    if (Status != XST_SUCCESS)
                    {
                            return XST_FAILURE;
                    }

                    XScuGic_CPUWriteReg(&InterruptController, XSCUGIC_EOI_OFFSET,
XPAR_FABRIC_AXI_TIMER_0_INTERRUPT_INTR);

                    /*
                    * Connect a device driver handler that will be called when an
                    * interrupt for the device occurs, the device driver handler performs
                    * the specific interrupt processing for the device
                    */
                    Status = XScuGic_Connect(&InterruptController,

XPAR_FABRIC_AXI_TIMER_0_INTERRUPT_INTR,

(Xil_ExceptionHandler)XTmrCtr_InterruptHandler,
                                                                    (void *)TimerInstancePtr);

                    if (Status != XST_SUCCESS)
                    {
                            return XST_FAILURE;
                    }
                    /*
                    * Enable the interrupt for the device and then cause (simulate) an
                    * interrupt so the handlers will be called
                    */
                    XScuGic_Enable(&InterruptController, XPAR_FABRIC_AXI_TIMER_0_INTERRUPT_INTR);

                    return XST_SUCCESS;
            }

            int main()
            {
              cout << "Application starts " << endl;
              int xStatus;

                //~~~~~~~~~~
                //Step-1 :AXI Timer Initialization
                //~~~~~~~~~~
                xStatus = XTmrCtr_Initialize(&TimerInstancePtr, XPAR_AXI_TIMER_0_DEVICE_ID);
                if(XST_SUCCESS != xStatus)
                {
                    cout << "TIMER INIT FAILED " << endl;
                    if(xStatus == XST_DEVICE_IS_STARTED)
                    {
                        cout << "TIMER has already started" << endl;
                        cout << "Please power cycle your board, and re-program the bitstream" << endl;
                    }
                    return 1;
                }

                    //~~~~~~~~~~
                    //Step-2 :Set Timer Handler
```

```
        //~~~~~~~~~
    //
    // cast second argument to data type XTmrCtr_Handler since in gcc it gave a warning
    // and with g++ for the C++ version it resulted in an error

        XTmrCtr_SetHandler(&TimerInstancePtr, (XTmrCtr_Handler)Timer_InterruptHandler,
&TimerInstancePtr);

   // intialize time pointer with value from xparameters.h file

 unsigned int* timer_ptr = (unsigned int* )XPAR_AXI_TIMER_0_BASEADDR;

     //~~~~~~~~~
    //Step-3 :load the reset value
    //~~~~~~~~~
    *(timer_ptr+ 1) = 0xf0000000;
    *(timer_ptr+ 5) = 0xffffff00;

     //~~~~~~~~~
     //Step-4 : set the timer options
     //~~~~~~~~~
     // from xparameters.h file #define XPAR_AXI_TIMER_0_BASEADDR 0x42800000
     //  Configure timer in generate mode, count up, interrupt enabled
     //  with autoreload of load register

    *(timer_ptr)  = 0x0f4 ;
    *(timer_ptr + 4)  = 0x0f4 ;

        //~~~~~~~~~
        //Step-5 : SCUGIC interrupt controller Initialization
        //Registration of the Timer ISR
        //~~~~~~~~~
        xStatus=
        ScuGicInterrupt_Init(XPAR_PS7_SCUGIC_0_DEVICE_ID, &TimerInstancePtr);
        if(XST_SUCCESS != xStatus)
        {
                cout << " :( SCUGIC INIT FAILED )" << endl;
                return 1;
        }

    // Beginning of our main code

        //We want to control when the timer starts
        char input;
        cout << "Press any key to start the timer" << endl;
        cin >> input ;
        cout << "You pressed "<<  input << endl;
  cout << "Enabling the timer to start" << endl;

    *(timer_ptr) = 0x0d4 ;   // deassert the load 5 to allow the timer to start counting
    *(timer_ptr + 4) = 0x0d4 ;

    // let timer run forever generating periodic interrupts

        while( 1)
    {
     //  // wait forever and let the timer generate periodic interrupts
```

```
    }

  return 0;
}
```

Code 3

```
#include "xil_exception.h"
#include "xil_cache.h"
#include "xparameters.h"
#include <iostream>
#include "xtmrctr.h"

using namespace std;

// Function to transfer data without CDMA
unsigned int transferWithoutCDMA(u32* source_ptr, u32* destination_ptr, int num_integers, XTmrCtr&
TimerInstancePtr) {
  XTmrCtr_Start(&TimerInstancePtr, 0);
  for(int i = 0; i < num_integers; i++) {
    *(destination_ptr + i) = *(source_ptr + i);
  }
  XTmrCtr_Stop(&TimerInstancePtr, 0);
  return XTmrCtr_GetValue(&TimerInstancePtr, 0);
}

// Function to transfer data with CDMA
unsigned int transferWithCDMA(u32* cdma_ptr, u32* source_ptr, u32* destination_ptr, int num_bytes,
XTmrCtr& TimerInstancePtr) {
  // Reset CDMA
  *(cdma_ptr) = 0x00000004;
  // Configure CDMA
  *(cdma_ptr) = 0x00000020;
  // Load source and destination addresses
  *(cdma_ptr + 6) = (u32)source_ptr;
  *(cdma_ptr + 8) = (u32)destination_ptr;
  // Flush cache
  Xil_DCacheFlush();
  // Set number of bytes to transfer
  *(cdma_ptr + 10) = num_bytes;

  XTmrCtr_Start(&TimerInstancePtr, 0);
  // Wait for transfer to complete
  while(!(*(cdma_ptr + 1) & 2)) {}
  XTmrCtr_Stop(&TimerInstancePtr, 0);
  return XTmrCtr_GetValue(&TimerInstancePtr, 0);
}

int main() {
  // Declare variables
  u32* source_ptr = (u32*)XPAR_PS7_DDR_0_S_AXI_HP0_BASEADDR;
  u32* destination_ptr = (u32*)XPAR_PS7_DDR_0_S_AXI_HP2_BASEADDR;
  u32* cdma_ptr = (u32*)XPAR_AXI_CDMA_0_BASEADDR;
  XTmrCtr TimerInstancePtr;
```

```
    int xStatus;

    // Initialize AXI Timer
    xStatus = XTmrCtr_Initialize(&TimerInstancePtr, XPAR_AXI_TIMER_0_DEVICE_ID);
    if(xStatus != XST_SUCCESS) {
      cout << "TIMER INIT FAILED" << endl;
      return 0;
    }
    XTmrCtr_SetResetValue(&TimerInstancePtr, 0, 0);
    XTmrCtr_SetOptions(&TimerInstancePtr, XPAR_AXI_TIMER_0_DEVICE_ID,
XTC_CAPTURE_MODE_OPTION);

    // Array sizes to test
    int array_sizes[] = {1024, 4096, 8192, 16384, 32767, 1048576};
    int num_sizes = sizeof(array_sizes) / sizeof(array_sizes[0]);

    // Measure transfer time for each array size
    for(int i = 0; i < num_sizes; i++) {
      int num_integers = array_sizes[i];
      int num_bytes = num_integers * sizeof(u32);

      // Without CDMA
      unsigned int cycles_without_cdma = transferWithoutCDMA(source_ptr, destination_ptr,
num_integers, TimerInstancePtr);

      // With CDMA
      unsigned int cycles_with_cdma = transferWithCDMA(cdma_ptr, source_ptr, destination_ptr,
num_bytes, TimerInstancePtr);

      // Output results
      cout << "Array Size: " << num_bytes << " bytes" << endl;
      cout << "Time without CDMA: " << cycles_without_cdma << " clock cycles" << endl;
      cout << "Time with CDMA: " << cycles_with_cdma << " clock cycles" << endl;

      // Compute speedup
      float speedup = static_cast<float>(cycles_without_cdma) / cycles_with_cdma;
      cout << "Speedup: " << speedup << endl << endl;
    }

    return 0;
  }
```

Code 4

```
#include "xil_exception.h"
#include "xil_cache.h"
#include "xparameters.h"
#include <iostream>
#include "xtmrctr.h"

using namespace std;

int main()
{
```

```
//---------------------------------------------------Declare variables-----------------------------------
//declare the three pointers
u32* cdma_ptr = (u32*) XPAR_AXI_CDMA_0_BASEADDR;
u32* source_ptr = (u32*) XPAR_PS7_DDR_0_S_AXI_HP0_BASEADDR;
u32* destination_ptr = (u32*) XPAR_PS7_DDR_0_S_AXI_HP2_BASEADDR;

//declare the timer pointer
XTmrCtr TimerInstancePtr;
int xStatus;

//---------------------------------------------------Initialize the variables----------------------------------------
-------
//initializing the contents of the source array within the maximum size
for(int i =0; i <= 24; i++)
{
        *(source_ptr + i) = i;
}

//initializing the contents of the destination array
for(int i =0; i <= 24; i++)
{
        *(destination_ptr + i) = -i;
}

//----------initialize the axi timer
xStatus = XTmrCtr_Initialize(&TimerInstancePtr, XPAR_AXI_TIMER_0_DEVICE_ID);
if(xStatus != XST_SUCCESS)
    {
  cout <<  "TIMER INIT FAILED" << endl;
  return 0;
}

//setting the timer reset value
XTmrCtr_SetResetValue(&TimerInstancePtr, 0, 0);

//setting the timer option
XTmrCtr_SetOptions(&TimerInstancePtr, XPAR_AXI_TIMER_0_DEVICE_ID,
XTC_CAPTURE_MODE_OPTION);


//reset the cdma
*(cdma_ptr) = 0x00000004;

//configure the cdma
*(cdma_ptr) = 0x00000020;

//load address of source array
*(cdma_ptr + 6) = 0x20000000;

//load address of destination array
*(cdma_ptr + 8) = 0x30000000;

//flush the cash
Xil_DCacheFlush();

//number of bytes to transfer
*(cdma_ptr + 10) = 0x64;
```

```
//--------------------------------------------------main operations-----------------------------------------
//take in a value to start the process
char dummy;
cout << "Enter a key to begin" << endl;
cin >> dummy;

//start the timer
XTmrCtr_Start(&TimerInstancePtr, 0);

//begin transfer
int status_reg_val = *(cdma_ptr + 1) & 2;
while(status_reg_val == 0)
{
	status_reg_val = *(cdma_ptr + 1) & 2; //isolate the idle bit
	if(status_reg_val == 2){ break; } //break when idle bit is 1
}

//transfer ends, stop timer and get the count
XTmrCtr_Stop(&TimerInstancePtr, 0);
unsigned int count = XTmrCtr_GetValue(&TimerInstancePtr, 0);

bool are_equal = true;
//compare the contents after the transfer
for(int i=0 ; i <= 32; i++)
{
	if(*(destination_ptr + i) != *(source_ptr + i))
	{
		are_equal = false;
		cout <<"The destination and source value are not the same : " <<
*(destination_ptr + i) << " " << *(source_ptr + i) << endl;
	}
	else {
		cout <<"The destination and source value are the same : " << *(destination_ptr
+ i) << " " << *(source_ptr + i) << endl;
	}
}
//output the results
if (are_equal){ cout << "The values in the source and destination array are equal!" << endl;}
else { cout << "The values in the source and destination array are not equal!" << endl;}
cout << " It required " <<  count << " clock cycles to transfer the data " <<  endl;
return 0;
}
```