CALIFORNIA STATE UNIVERSITY

NORTHRIDGE

CSUN

Lab Report

ECE-520 SYSTEM ON CHIP DESIGN

(SOC Design)

By:

Kumar Sai Reddy Guntaka

202616615

Introduction

This project demonstrates how to use the Zybo Z7-20's Audio Codec and RAM to record samples of audio and play them back. Vivado is used to build the demo's hardware platform, and Xilinx SDK is used to program the bitstream onto the board and to build and deploy a C application.

To use this demo, the Zybo Z7-20 must be connected to a computer over MicroUSB, which must be running a serial terminal. For more information on how to set up and use a serial terminal.

The audio demo records a 5 second sample from microphone(J6) or line in (J7) and plays it back on headphone out(J5). Recording and playback are controlled by push buttons from a usb uart serial menu as shown below.

This tutorial will guide you through the process of creating a Zynq design using the Vivado Integrated Development Environment (IDE) and introduce the IP Integrator environment for the generation of a simple Zynq processor design to be implemented on a Zynq development board. The Software Development Kit (SDK) will then be used to create a simple software application which will run on the Zynq's ARM Processing System (PS) to control the hardware that is implemented in the Programmable Logic (PL).

The tutorial is split into three exercises, and is organized as follows:

Exercise IA - This exercise will guide you through the process of launching Vivado IDE and creating a project. The various stages of the New Project Wizard will be introduced.

Exercise 1B - In this exercise, we will use the project that was created in Exercise IA to build a simple Zynq embedded system with the graphical tool, IP Integrator, and incorporating existing IP from the Vivado IP Catalog and created IP. A number of design aids will be used throughout this exercise, such as the Board Automation feature which automates the customization of IP modules for a specified device or board. The Designer Assistance feature, which assists with the connections between the Zynq PS and the IP modules in the PL will also be demonstrated.

Once the design is finished, a number of stages will be undertaken to complete the hardware system and generate a bitstream for implementation in the PL. The completed hardware design will then be exported to the Software Development Kit (SDK) for the development of a simple software application in Exercise IC.

Exercise IC - In this short third exercise, the SDK will be introduced, and a simple software application will be created to allow the Zynq processor to interact with the IP implemented in the PL. A connection to the hardware server that allows the SDK to communicate with the Zynq processors will be established. The software drivers that are automatically created by the Vivado IDE for IP modules will be explored and integrated into the software application, before finally building and executing the software application on the Zynq.

NOTE: Throughout all the practical tutorials exercise we will be using one single working directory If this is not suitable, you can substitute it for a directory of your choice, but you should be aware that you will be required to make alterations to some source files in order to complete exercises successfully.

Exercise IA: Creating a IP Integrator Design

In this exercise we will create a new project in Vivado IDE by moving through the stages of the Vivado IDE New Project Wizard.

The Zybo requires a onetime additional set-up procedure in order to set the Default Part correctly. This is necessary as Vivado 2015.1 does not contain a board part for the Zybo development board. If you have already configured Vivado 2015.1 with the Zybo board part.

Open windows explorer and navigate to the following location

C:\Xilinx\sources\darta\boardfiles

In this directory you will see a file named board files. This contains the board files Zybo development board. Updated board parts can be retrieved from the Digilent Website using the following link:

https://reference.digilentinc.com/vivado:boardfiles

Copy the Zybo file by right clicking on the file and selecting copy.

We will start by launching the Vivado IDE.

(a) Launch Vivado by double-clicking on the Vivado desktop icon: or by navigating to Start > All Programs > Xilinx Design Tools > Vivado 2018.2 > Vivado 2018.2

When Vivado loads, you will be presented with the Getting Started.

(b) Select the option to Create New Project and the New Project Wizard will open and click next.

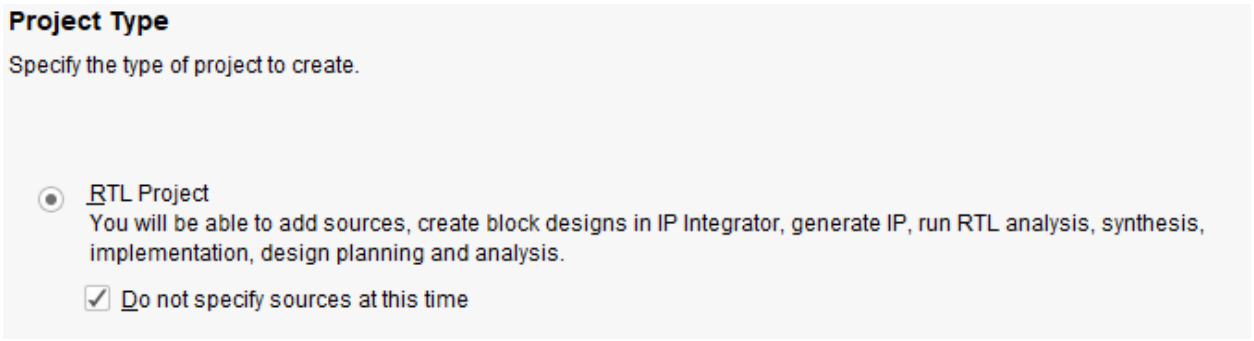
(c) At the Project Name dialogue, enter the Project name and choose the location.

Make sure that you select the option to Create project subdirectory. All options should be the same as shown below:

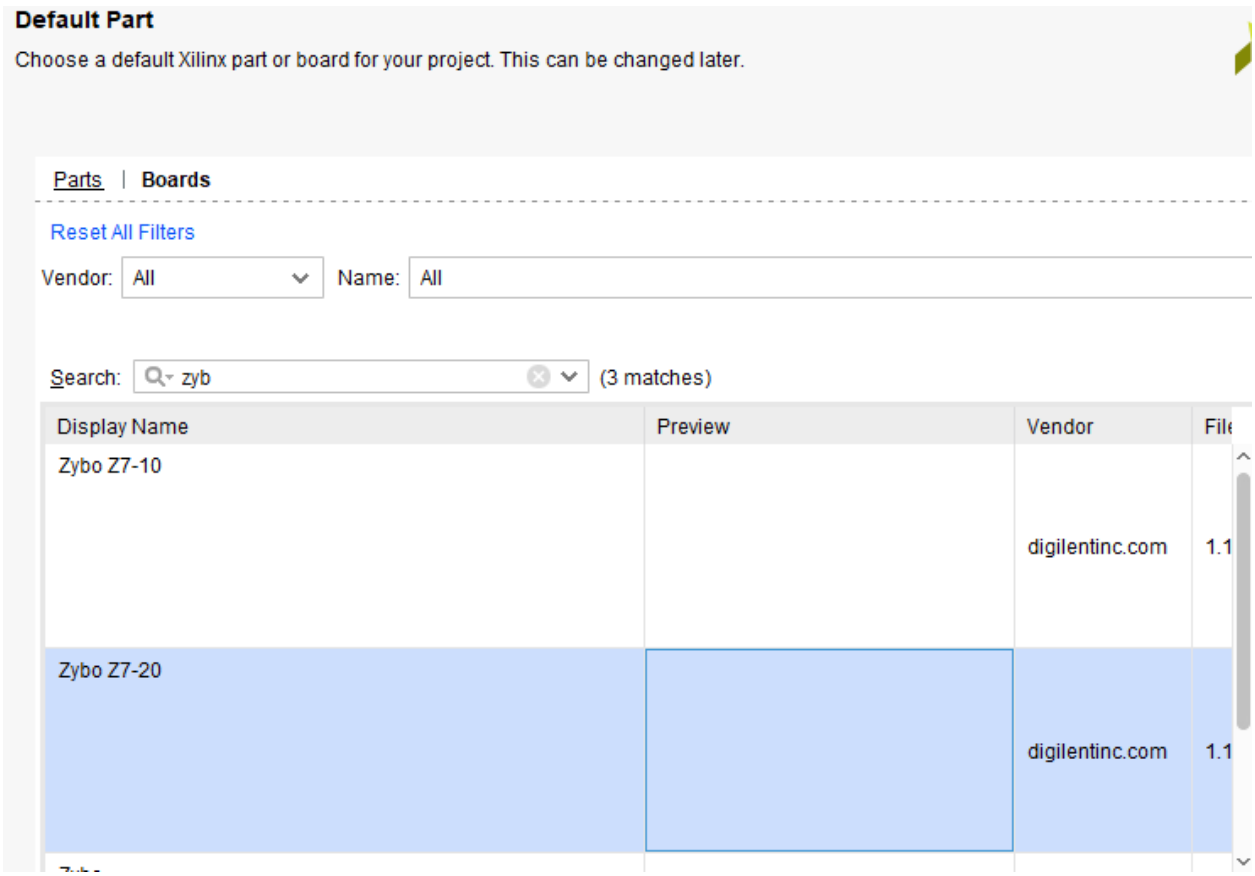
Click Next.

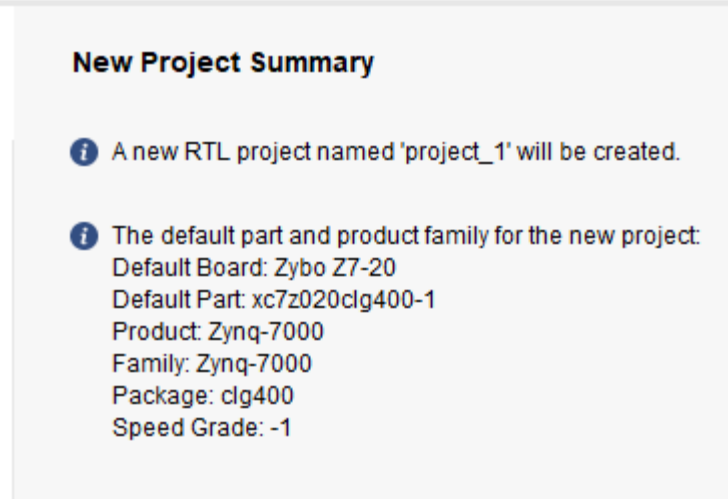
A directory name will be created on your location if it did not already exist.

(d) At the Project Type dialogue, select RTL Project and ensure that the option Do not specify sources at this time is selected:



(e) Board Rev list, as shown Select the appropriate revision for your board.



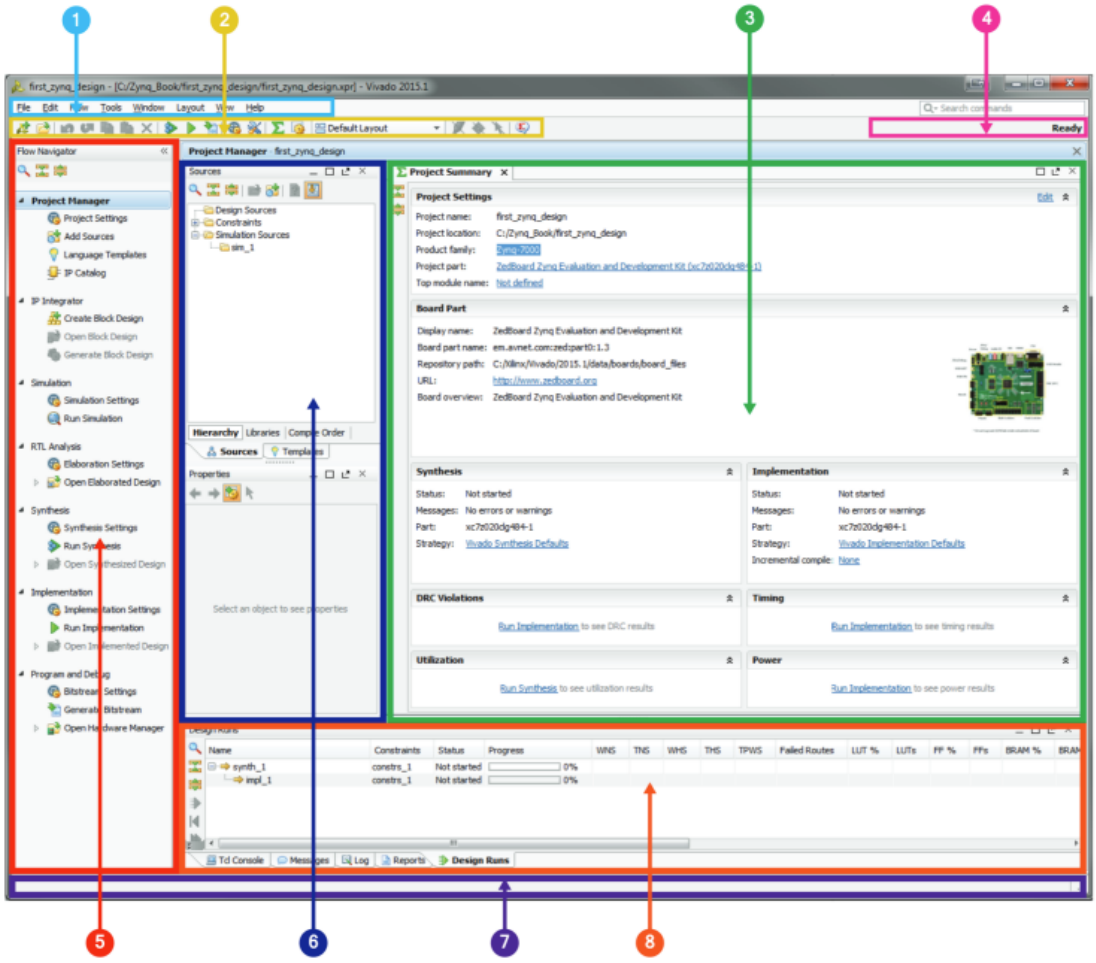


Now that we have created our first project in Vivado IDE, we can now move on to creating our first Zynq embedded system design.

Before doing that, the Vivado IDE tool layout should be introduced. The default Vivado IDE environment layout (other layouts can be chosen by selecting different perspectives). This layout is specifically targeted for If you are using the Zybo.

With reference to the numbered labels, the main components of the Vivado IDE environment are:

1. Menu Bar - The main access bar gives access to the Vivado IDE commands.
2. Main Toolbar - The main toolbar provides easy access to the most used Vivado IDE commands. Tooltips provide information about each command on the toolbar, and these can be viewed by hovering the mouse pointer over the buttons.



3. Workspace - The workspace provides a larger area for panels which require a greater screen space and those with a graphical interface, such as: Schematic Panel, Device Panel, Package Panel, Text Editor Panel

4. Project Status Bar - The project status bar displays the status of the currently active design.

5. Flow Navigator - The Flow Navigator provides easy access to the tools and commands that are necessary to guide your design from start to finish, starting in the Project Manager section with design entry and ending with bitstream generation in the Program and Debug section. Run commands are available in the Simulation, Synthesis and Implementation sections to simulate, synthesize and implement the active design.

6. Data Windows Pane -The Data Windows pane, by default, displays information that relates to design data and sources, including:

- Properties window - Shows information about selected logic objects or device resources.
- Netlist window - Provides a hierarchical view of the synthesized or elaborated logic design.

- Sources window - Shows IP Sources, Hierarchy, Libraries and Compile Order views.

7. Status Bar - The status bar displays a variety of information, including:

    - Detailed information regarding menu bar and toolbar commands will be shown in the lower left side of the status bar when the command is accessed.
    - When hovering over an object in the Schematic window with the mouse pointer, the object details appear in the status bar.
    - During constraint and placement creation in the Device and Package windows, validity and constraint type will be shown on the left side of the status bar. Site coordinates and type will be shown in the right side.
    - The task progress of a running task will be relocated to the right side of the status bar when the Background button is selected.

8. Results Window Area -The Results Window displays the status and results of commands in a set of windows grouped in the bottom of the Vivado IDE environment. As commands progress, messages are generated, and log files and reports are created. The related information is shown here. The default windows are:

    - Messages - Displays all messages for the active design.
    - TCI Console - TCI commands can be entered here, and a history of previous commands and outputs are also available.
    - Reports - Quick access is provided to the reports generated throughout the design flow.
    - Log -Displays the log files generated by the simulation, synthesis and implementation processes.
    -
    -
    - Design Runs -Manages runs for the current project.

    Additional windows that can appear in this area as required are: Find Results window, Timing Results window and Package Pins window.

With the layout of the Vivado IDE environment introduced, we can now move on to creating the Zynq system.

Exercise: 1B Creating a Zynq system in Vivado

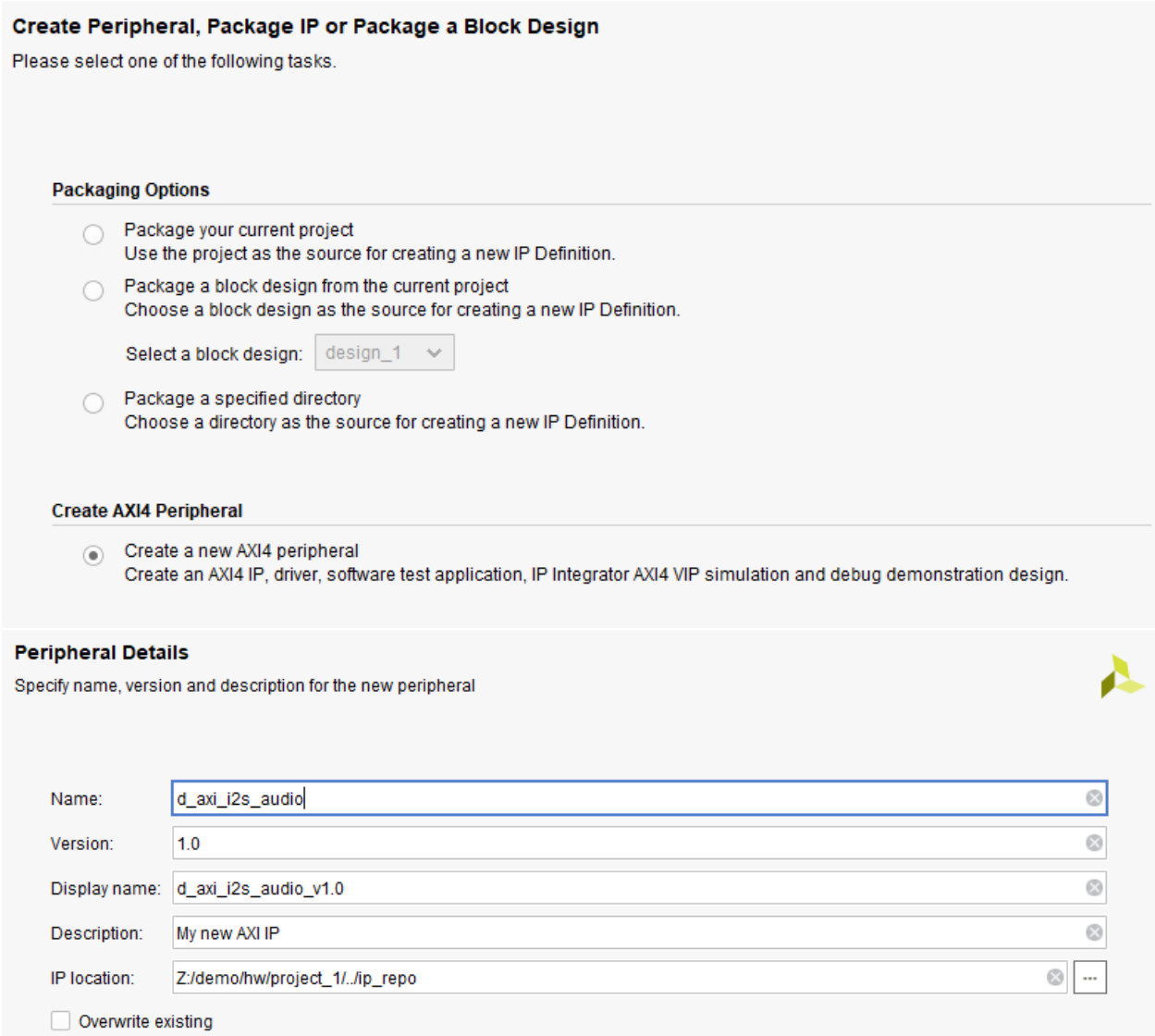
In this exercise we will create a simple Zynq embedded system which implements a General-Purpose Input/Output (GPIO) controller in the PL of the Zynq device. The GPIO controller will connect to the Buttons. It will also be connected to the Zynq processor via an AXI bus connection, allowing the buttons to be controlled by a software application which we will create.

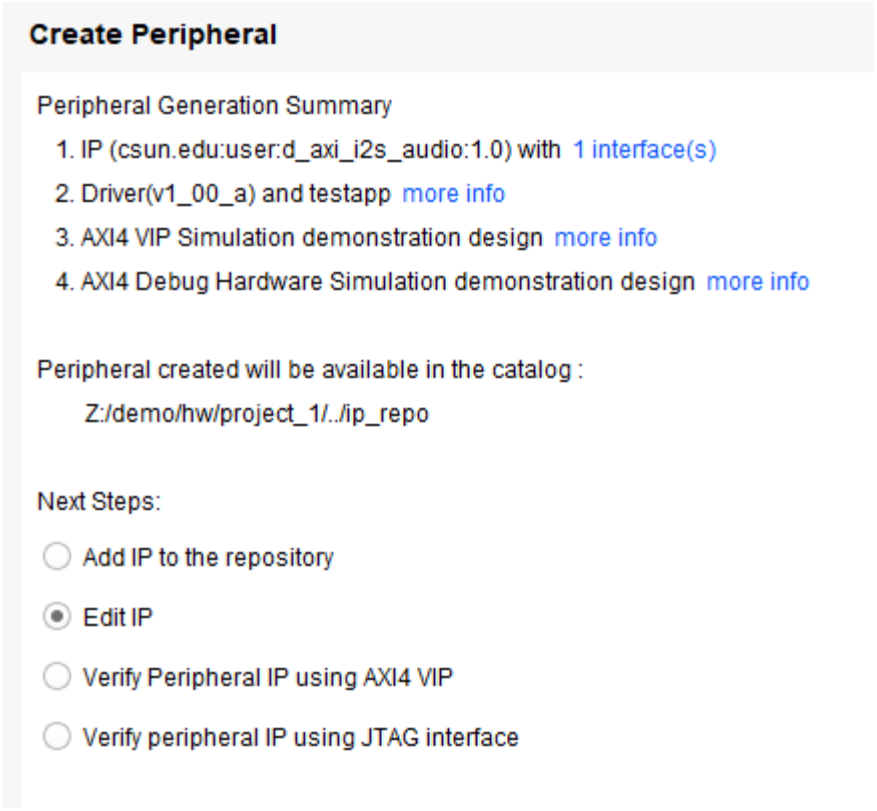
We will begin by creating a new Block Design in Vivado IDE.

(a) Before that in the menu toolbar, select Create customized IP.

Enter project name in the Design name box Click OK. The Vivado IP Integrator Diagram canvas will open in the Workspace.

Create the IP d_axi_i2s_audio_v2_0 and use that Ip in the main project.





After creating a IP add the IP in main project by selecting ->settings ->IP ->Repository -> add the path and Click OK.

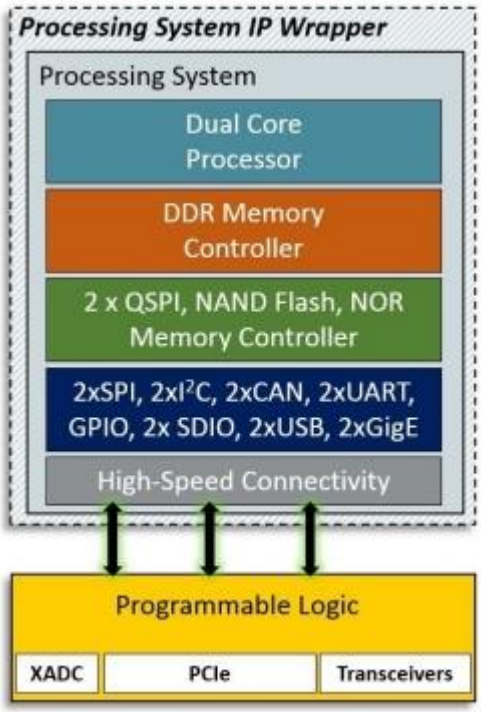
(b) The first block that we will add to our design will be a Zynq Processing System. To view Summary report Double click on the Zynq block.

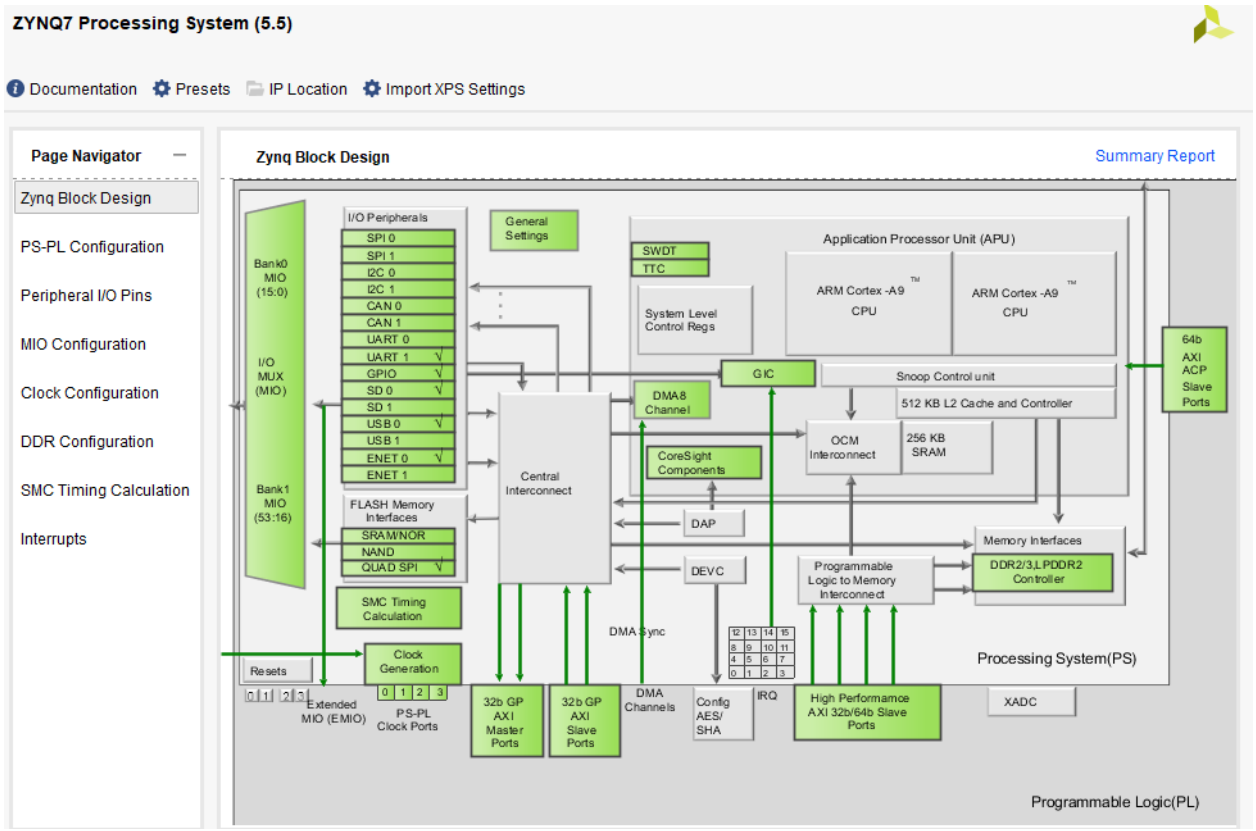
Xilinx provides the Processing System IP Wrapper for the Zynq®-7000 to accelerate your design and its configuration for your embedded products

The Processing System IP is the software interface around the Zynq-7000 Processing System. the Zynq-7000 family consists of a system-on-chip (SoC) style integrated processing system (PS) and a Programmable Logic (PL) unit, providing an extensible and flexible SoC solution on a single die. The Processing System IP Wrapper acts as a logic connection between the PS and the PL while assisting you to integrate custom and embedded IPs with the processing system using the Vivado® IP integrator.

Key Features and Benefits

- Enable/Disable I/O Peripherals (IOP)
- Enable/Disable AXI I/O ports (AIO)
- MIO Configuration
- Extended MULTIPLE USE I/O's (EMIO)
- DDR Configuration
- Security and Isolation Configuration
- Interconnect Logic for Vivado IP - PS interface
- PL Clocks and Interrupts





(c) In the Vivado lP integrator Diagram canvas, right-click anywhere and select AddlP.

(d) Alternatively, select the Add IP button in the toolbar at the left of the canvas Enter zynq in the search field and select the ZYNQ7 Processing System, as shown in Be careful not to select the BFM version and press the Enter key on your keyboard.

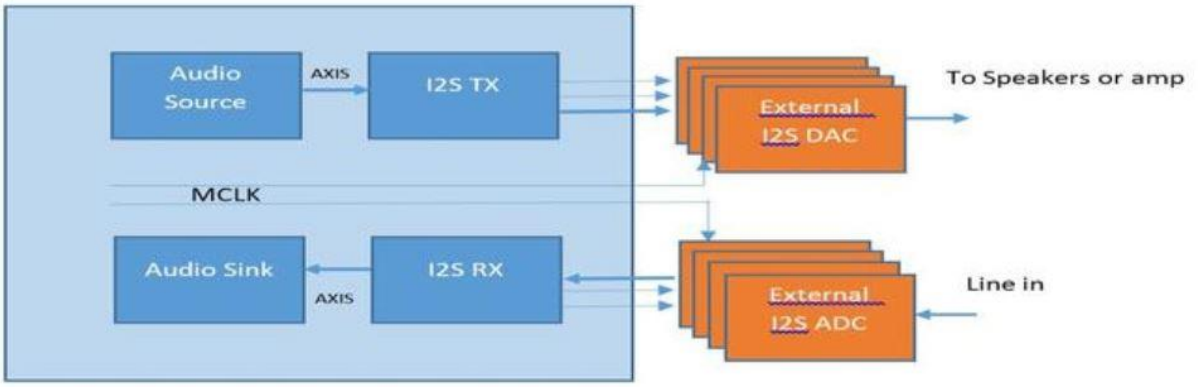
(e) Similarly add the other IP's.

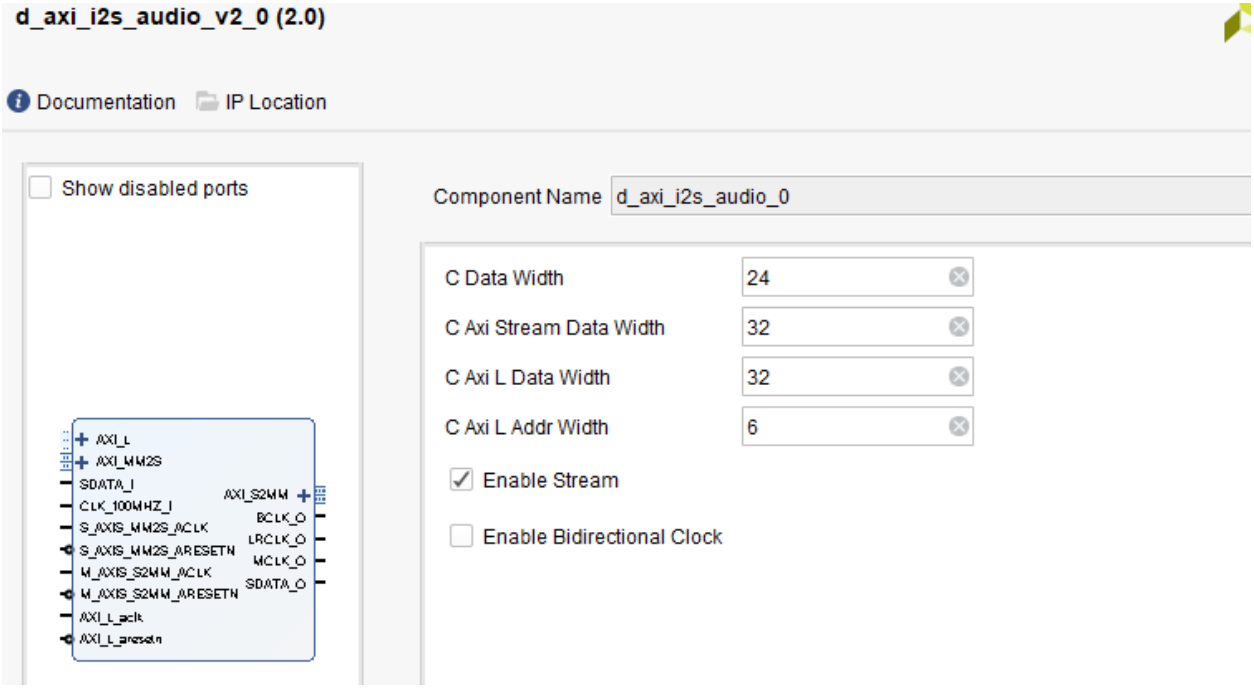
The Xilinx® LogiCORE™ IP I2S Transmitter and Receiver cores are soft IP cores in Xilinx Vivado design suite which make it easy to implement Inter-IC-Sound (I2S) interface used to connect audio devices for transmitting and receiving PCM audio.

Key Features and Benefits

- AXI4S Compliant
- Can be configured up to 4 I2S interfaces, each channel supporting 2 audio channels
- Can be configured up to 4 stereo or 8 independent channels
- 16/24-bit data support
- Master I2S mode

- Configurable FIFO depth
- AES Channel Status Extraction/Insertion
- License free IPs



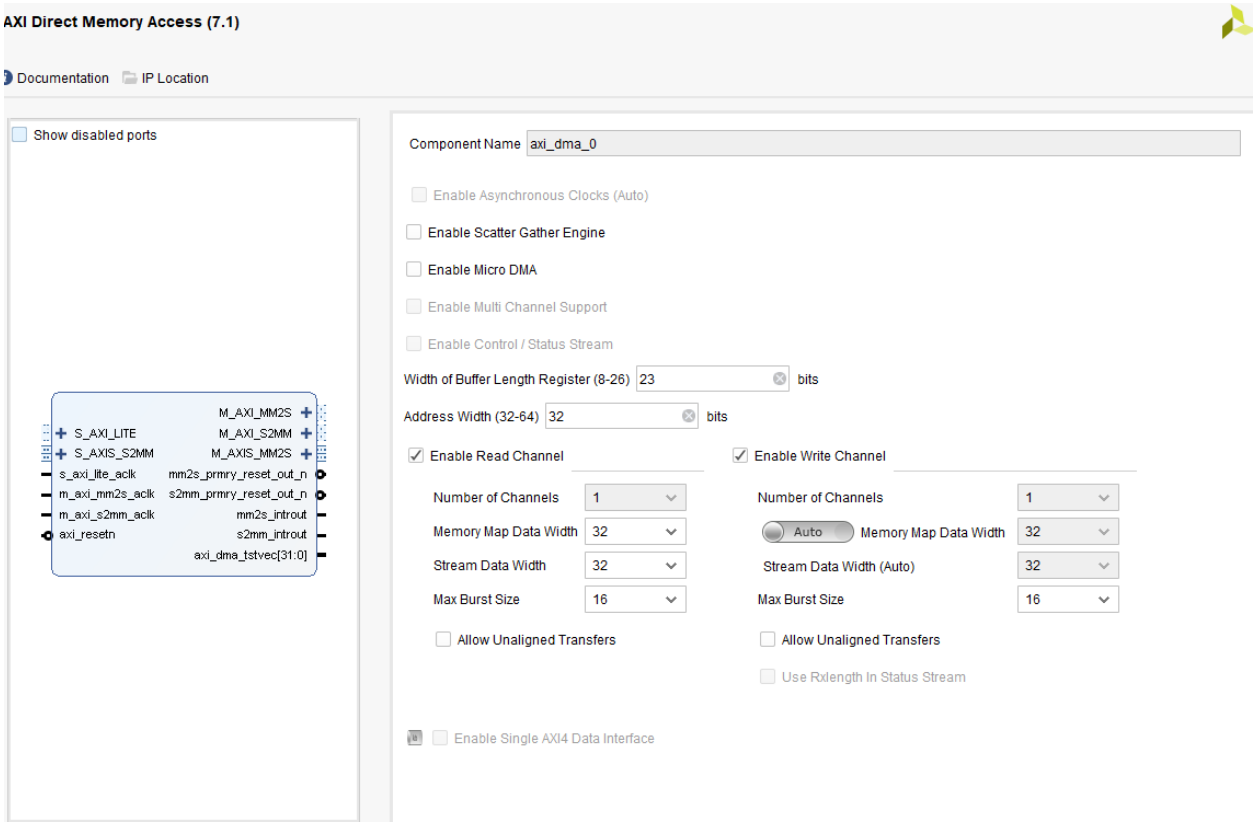


(f) Add DMA

The AXI Direct Memory Access (AXI DMA) IP provides high-bandwidth direct memory access between memory and AXI4-Stream-type target peripherals. Its optional scatter gather capabilities also offload data movement tasks from the Central Processing Unit (CPU) in processor based systems. Initialization, status, and management registers are accessed through an AXI4-Lite slave interface.

Key Features and Benefits

- AXI4 compliant
- Optional Scatter/Gather (SG) DMA support. When Scatter/gather mode is not selected the IP operates in Simple DMA mode.
- Primary AXI4 Memory Map and AXI4-Stream data width support of 32, 64, 128, 256, 512, and 1024 bits
- Optional Data Re-Alignment Engine
- Optional AXI Control and Status Streams
- Optional Keyhole support
- Optional Data Re-Alignment support
- Optional Micro DMA support

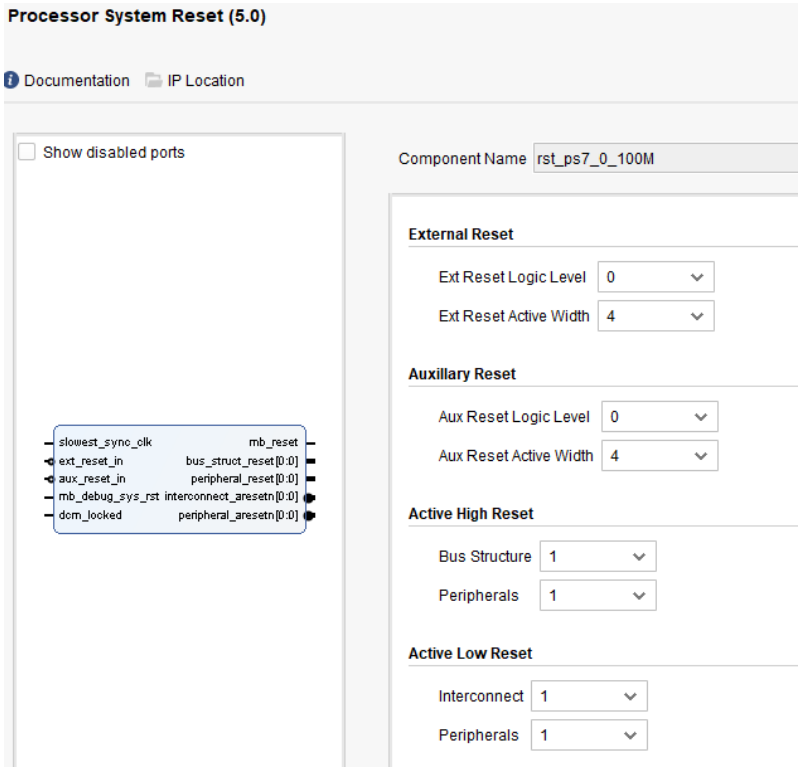


(g) Add Processor System Reset

The Xilinx Processor System Reset Module design allows the customer to tailor the design to suit their application by setting certain parameters to enable/disable features. The parameterizable features of the design are discussed in Processor System Reset Module Design Parameters.

Key Features and Benefits

- Asynchronous external reset input is synchronized with clock.
- Asynchronous auxiliary external reset input is synchronized with clock.
- Both the external and auxiliary reset inputs are selectable active high or active low.
- Selectable minimum pulse width for reset inputs to be recognized.
- Selectable load equalizing.
- DCM Locked input.
- Power On Reset generation.

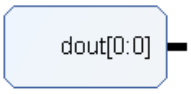


(h) And add a constant connected to mute enable.

The Constant IP core is used to drive a constant value on a bus. Often there is a need to drive a signal or a bus to a predetermined value. The Constant IP core provides this capability in the block design.

Key Features and Benefits

- Parameterizable bus width
- Value to be driven in decimal format

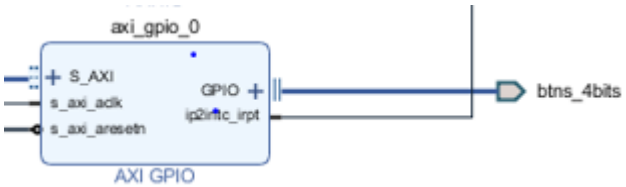


(i) Connect the GPIO to buttons to control.

The AXI GPIO provides a general-purpose input/output interface to the AXI (Advanced extensible Interface) interface. This 32-bit soft IP core is designed to interface with the AXI4-Lite interface.

Key Features and Benefits

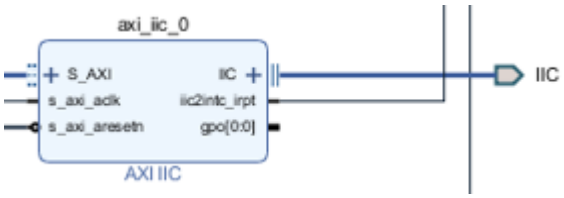
- Supports the AXI4-Lite interface specification
- Supports configurable single or dual GPIO channel (s)
- Supports configurable channel width for GPIO pins from 1 to 32 bits
- Supports dynamic programming of each GPIO bit as input or output
- Supports individual configuration of each channel
- Supports independent reset values for each bit of all registers
- Supports optional interrupt request generation



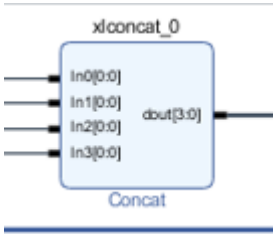
(j) Add the AXI IIC connected to zynq.

This defines the architecture, hardware (signal) interface, software (register) interface and parameterization options for the LogiCORE™ IP AXI IIC Bus Interface module. This module connects to the Advanced Microcontroller Bus Architecture (AMBA®) specification's Advanced extensible Interface (AXI) and provides a low-speed, two-wire, serial bus interface to a large number

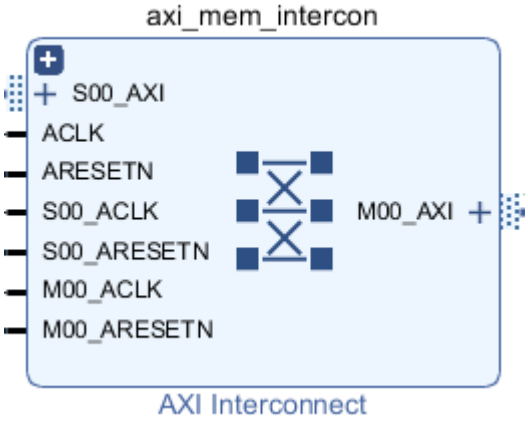
of popular devices. AXI IIC supports all features, except high-speed mode, of the Philips I2C-Bus Specification.



(k) The Concat IP core is used for concatenating bus signals of varying widths.



(l) AXI Interconnect



Included at no additional charge with Vivado and ISE Design Suite

The AXI Interconnect IP connects one or more AXI memory-mapped Master devices to one or more memory-mapped Slave devices. The AXI interfaces conform to the AMBA® AXI version 4 specifications from ARM®, including the AXI4-Lite control register interface subset. The Interconnect IP is intended for memory-mapped transfers only; AXI4-Stream transfers are not applicable. The AXI Interconnect IP can be used from the Vivado® IP catalog as a core from the Embedded Development Toolkit (EDK) or as a standalone core from the CORE Generator™ IP catalog.

Key Features and Benefits

EDK

- Selectable interconnect architecture
    - Crossbar mode (Performance optimized): Shared-Address, Multiple-Data (SAMD) crossbar architecture with parallel pathways for write and read data channels
    - Shared Access mode (Area optimized): Shared write data, shared read data, and single shared address pathways.
- AXI protocol compliant (AXI3, AXI4, and AXI4-Lite) includes:
    - Burst lengths up to 256 for incremental (INCR) bursts
    - Converts AXI4 bursts > 16 beats when targeting AXI3 slave devices by splitting transactions
    - Generates REGION outputs for use by slave devices with multiple address decode ranges
    - Propagates USER signals on each channel, if any; independent USER signal width per channel (optional)
    - Propagates Quality of Service (QoS) signals, if any; not used by the AXI Interconnect core (optional)
- Interface data widths:
    - AXI4: 32, 64, 128, 256, 512, or 1024 bits
    - AXI4-Lite: 32 bits
- 32-bit address width
- Connects to 1-16 master devices and to 1-16 slave devices
- Built-in data-width conversion, synchronous/ asynchronous clock-rate conversion and AXI4-Lite/AXI3 protocol conversion
- Optional register-slice pipelining and Datapath FIFO buffering

- Optional packet-FIFO capability
    - Delays issuing AWVALID until the complete burst is stored in the write data FIFO
    - Delays issuing ARVALID until the read data FIFO has enough vacancy to store the entire burst length
- Supports multiple outstanding transactions in crossbar mode
- “Single-Slave per ID” method of cyclic dependency (deadlock) avoidance
- Fixed priority and round-robin arbitration
- Supports Trust Zone security for each connected slave as a whole
- Support for Read-only and Write-only masters and slaves, resulting in reduced resource utilization.

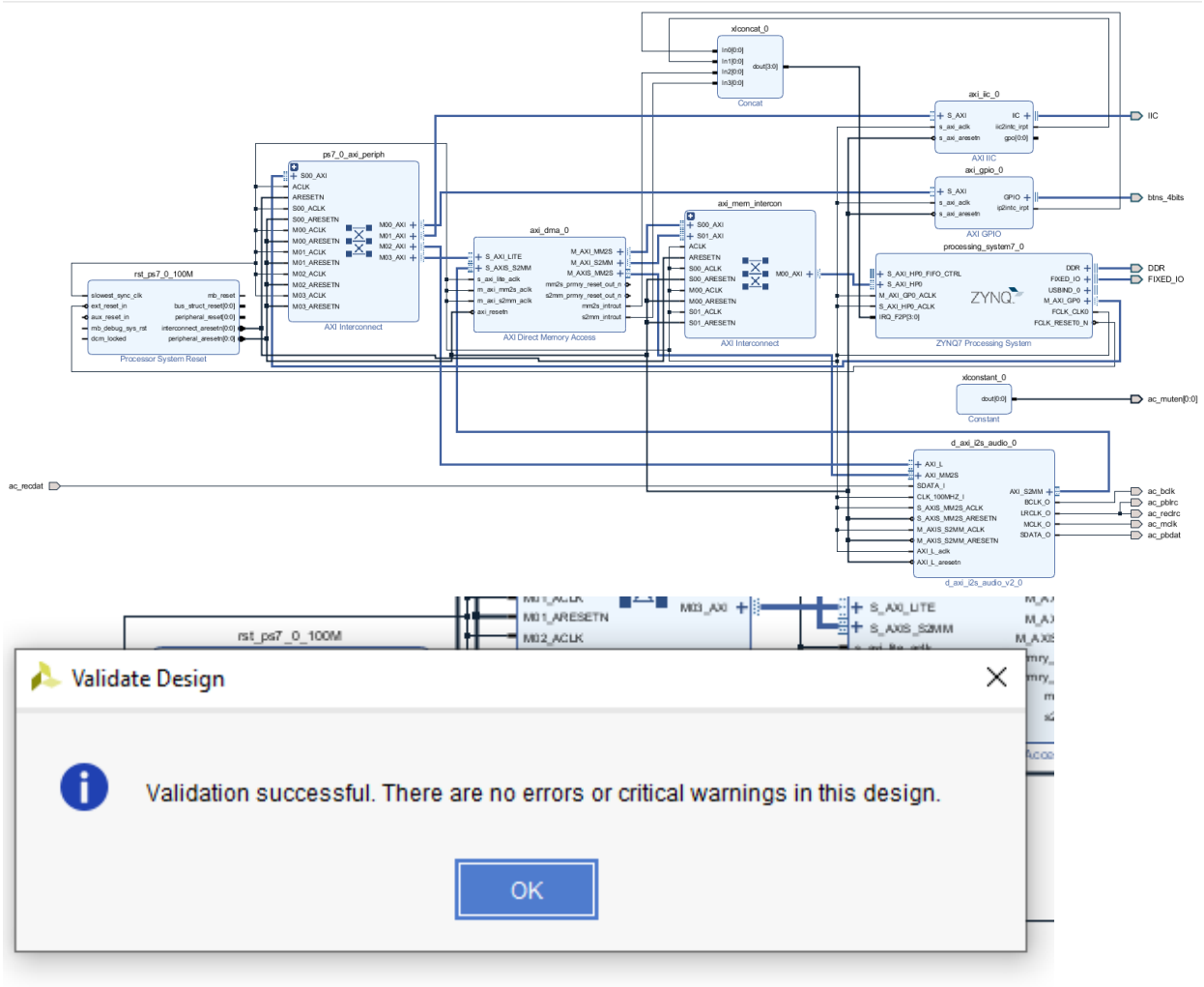
CORE Generator

- AXI protocol compliant (AXI4 only), including:
    - Burst lengths up to 256 for incremental (INCR) bursts
    - Propagates Quality of Service (QoS) signals, if any; not used by the AXI Interconnect core (optional)
- Interface data widths:32, 64, 128, 256, 512, or 1024 bits
- Address width: 12 to 64 bits
- Connects to 1-16 master devices and to one slave device
- Built-in data-width conversion and synchronous /asynchronous clock-rate conversion
- Optional register-slice pipelining and Datapath FIFO buffering
- Optional packet-FIFO capability
    - Delays issuing AWVALID until the complete burst is stored in the write data FIFO

- Delays issuing ARVALID until the read data FIFO has enough vacancy to store the entire burst length

- Supports multiple outstanding transactions
- Fixed priority and round-robin arbitration
- Support for Read-only and Write-only master devices, resulting in reduced resource utilization.

(m) Click the Run Block Automation option from the Designer Assistance message at the top of the Diagram window



Exercise:2

a) After validation create a HDL wrapper and generate a bitstream.

b) After bitstream export the hardware file which describes the hardware.

Source codes:

3.1 Demo.c

```
#include "demo.h"
#include "audio.h"
#include "dma.h"
#include "intc.h"
#include "userio.h"
#include "iic.h"

/****************************** Include
Files ******************************/

#include "xaxidma.h"
#include "xparameters.h"
#include "xil_exception.h"
#include "xdebug.h"
#include "xiic.h"
#include "xaxidma.h"
#include "xtime_l.h"


#ifdef XPAR_INTC_0_DEVICE_ID
 #include "xintc.h"
 #include "microblaze_sleep.h"
#else
 #include "xscugic.h"
#include "sleep.h"
#include "xil_cache.h"
#endif
```

```
/************************** Constant
Definitions *****************************/

/*
 * Device hardware build related constants.
 */

// Audio constants
// Number of seconds to record/playback
#define NR_SEC_TO_REC_PLAY     5

// ADC/DAC sampling rate in Hz
//#define AUDIO_SAMPLING_RATE      1000
#define AUDIO_SAMPLING_RATE   96000

// Number of samples to record/playback
#define
NR_AUDIO_SAMPLES       (NR_SEC_TO_R
EC_PLAY*AUDIO_SAMPLING_RATE)

/* Timeout loop counter for reset
 */
#define
RESET_TIMEOUT_COUNTER   10000

#define TEST_START_VALUE    0x0
```

```
/************************** Type
Definitions
*****************************/


/****************** Macros (Inline Functions)
Definitions *********************/


/************************** Function
Prototypes
*****************************/
#if (!defined(DEBUG))
extern void xil_printf(const char *format, ...);
#endif


/************************** Variable
Definitions *****************************/
/*
 * Device instance definitions
 */

static XIic sIic;
static XAxiDma sAxiDma;     /* Instance of the
XAxiDma */

static XGpio sUserIO;

#ifdef XPAR_INTC_0_DEVICE_ID
 static XIntc sIntc;
#else
 static XScuGic sIntc;
#endif

//
// Interrupt vector table
#ifdef XPAR_INTC_0_DEVICE_ID
const ivt_t ivt[] = {
   //IIC
   {XPAR_AXI_INTC_0_AXI_IIC_0_IIC2INT
C_IRPT_INTR,
(XInterruptHandler)XIic_InterruptHandler,
&sIic},
   //DMA Stream to MemoryMap Interrupt
handler
   {XPAR_AXI_INTC_0_AXI_DMA_0_S2M
M_INTROUT_INTR,
(XInterruptHandler)fnS2MMInterruptHandler,
&sAxiDma},
   //DMA MemoryMap to Stream Interrupt
handler
   {XPAR_AXI_INTC_0_AXI_DMA_0_MM2
S_INTROUT_INTR,
(XInterruptHandler)fnMM2SInterruptHandler,
&sAxiDma},
   //User I/O (buttons, switches, LEDs)
   {XPAR_AXI_INTC_0_AXI_GPIO_0_IP2IN
TC_IRPT_INTR,
(XInterruptHandler)fnUserIOIsr, &sUserIO}
};
#else
const ivt_t ivt[] = {
   //IIC
   {XPAR_FABRIC_AXI_IIC_0_IIC2INTC_IR
PT_INTR,
(Xil_ExceptionHandler)XIic_InterruptHandler,
&sIic},
   //DMA Stream to MemoryMap Interrupt
handler
   {XPAR_FABRIC_AXI_DMA_0_S2MM_IN
TROUT_INTR,
(Xil_ExceptionHandler)fnS2MMInterruptHandl
er, &sAxiDma},
   //DMA MemoryMap to Stream Interrupt
handler
   {XPAR_FABRIC_AXI_DMA_0_MM2S_IN
TROUT_INTR,
(Xil_ExceptionHandler)fnMM2SInterruptHandl
er, &sAxiDma},
   //User I/O (buttons, switches, LEDs)
   {XPAR_FABRIC_AXI_GPIO_0_IP2INTC_I
RPT_INTR,
(Xil_ExceptionHandler)fnUserIOIsr,
&sUserIO}
};
#endif
```

```
/*****************************************
*****************************************
/
/**
*
* Main function
*
* This function is the main entry of the interrupt
test. It does the following:
*   Initialize the interrupt controller
*   Initialize the IIC controller
*   Initialize the User I/O driver
*   Initialize the DMA engine
*   Initialize the Audio I2S controller
*   Enable the interrupts
*   Wait for a button event then start selected
task
*   Wait for task to complete
*
* @param    None
*
* @return
*       - XST_SUCCESS if example finishes
successfully
*       - XST_FAILURE if example fails.
*
* @note     None.
*
*****************************************
*****************************************
/
int main(void)
{
   int Status;

   Demo.u8Verbose = 0;

   //Xil_DCacheDisable();

   xil_printf("\r\n--- Entering main() --- \r\n");


   //
   //Initialize the interrupt controller
   Status = fnInitInterruptController(&sIntc);
   if(Status != XST_SUCCESS) {
      xil_printf("Error initializing interrupts");
      return XST_FAILURE;
   }


   // Initialize IIC controller
   Status = fnInitIic(&sIic);
   if(Status != XST_SUCCESS) {
      xil_printf("Error initializing I2C
controller");
      return XST_FAILURE;
   }

   // Initialize User I/O driver
   Status = fnInitUserIO(&sUserIO);
   if(Status != XST_SUCCESS) {
      xil_printf("User I/O ERROR");
      return XST_FAILURE;
   }


   //Initialize DMA
   Status = fnConfigDma(&sAxiDma);
   if(Status != XST_SUCCESS) {
      xil_printf("DMA configuration ERROR");
      return XST_FAILURE;
   }


   //Initialize Audio I2S
   Status = fnInitAudio();
   if(Status != XST_SUCCESS) {
      xil_printf("Audio initializing ERROR");
      return XST_FAILURE;
   }

   {
      XTime  tStart, tEnd;

      XTime_GetTime(&tStart);
      do {
         XTime_GetTime(&tEnd);
      }
```

```
    while((tEnd-
tStart)/(COUNTS_PER_SECOND/10) < 20);
  }
  //Initialize Audio I2S
  Status = fnInitAudio();
  if(Status != XST_SUCCESS) {
    xil_printf("Audio initializing ERROR");
    return XST_FAILURE;
  }


  // Enable all interrupts in our interrupt vector
table
  // Make sure all driver instances using
interrupts are initialized first
  fnEnableInterrupts(&sIntc, &ivt[0],
sizeof(ivt)/sizeof(ivt[0]));



  xil_printf("-----------------------------------------
---------------\r\n");
  xil_printf("Zybo Z7-20 DMA Audio
Demo\r\n");
  xil_printf("-----------------------------------------
---------------\r\n");
  xil_printf("  Controls:\r\n");
  xil_printf("  BTN1: Record from MIC
IN\r\n");
  xil_printf("  BTN2: Play on HPH OUT\r\n");
  xil_printf("  BTN3: Record from LINE
IN\r\n");
  xil_printf("-----------------------------------------
---------------\r\n");

  //main loop

  while(1) {

    // Checking the DMA S2MM event flag
    if (Demo.fDmaS2MMEvent)
    {
      xil_printf("\r\nRecording Done...");
```

```
      // Disable Stream function to send data
(S2MM)
      Xil_Out32(I2S_STREAM_CONTROL_
REG, 0x00000000);
      Xil_Out32(I2S_TRANSFER_CONTRO
L_REG, 0x00000000);

      Xil_DCacheInvalidateRange((u32)
MEM_BASE_ADDR,
5*NR_AUDIO_SAMPLES);
      //microblaze_invalidate_dcache();
      // Reset S2MM event and record flag
      Demo.fDmaS2MMEvent = 0;
      Demo.fAudioRecord = 0;
    }

    // Checking the DMA MM2S event flag
    if (Demo.fDmaMM2SEvent)
    {
      xil_printf("\r\nPlayback Done...");

      // Disable Stream function to send data
(S2MM)
      Xil_Out32(I2S_STREAM_CONTROL_
REG, 0x00000000);
      Xil_Out32(I2S_TRANSFER_CONTRO
L_REG, 0x00000000);
      //Flush cache
      Xil_DCacheFlushRange((u32)
MEM_BASE_ADDR,
5*NR_AUDIO_SAMPLES);
      //Reset MM2S event and playback flag
      Demo.fDmaMM2SEvent = 0;
      Demo.fAudioPlayback = 0;
    }

    // Checking the DMA Error event flag
    if (Demo.fDmaError)
    {
      xil_printf("\r\nDma Error...");
      xil_printf("\r\nDma Reset...");


      Demo.fDmaError = 0;
      Demo.fAudioPlayback = 0;
```

```
        Demo.fAudioRecord = 0;
    }

    // Checking the btn change event
    if(Demo.fUserIOEvent) {

        switch(Demo.chBtn) {
            case 'u':
                if (!Demo.fAudioRecord &&
!Demo.fAudioPlayback)
                {
                    xil_printf("\r\nStart
Recording...\r\n");
                    fnSetMicInput();

                    fnAudioRecord(sAxiDma,NR_A
UDIO_SAMPLES);
                    Demo.fAudioRecord = 1;
                }
                else
                {
                    if (Demo.fAudioRecord)
                    {
                        xil_printf("\r\nStill
Recording...\r\n");
                    }
                    else
                    {
                        xil_printf("\r\nStill Playing
back...\r\n");
                    }
                }
                break;
            case 'd':
                if (!Demo.fAudioRecord &&
!Demo.fAudioPlayback)
                {
                    xil_printf("\r\nStart
Playback...\r\n");
                    fnSetHpOutput();
                    fnAudioPlay(sAxiDma,NR_AUD
IO_SAMPLES);
                    Demo.fAudioPlayback = 1;
                }
                else
                {
                    if (Demo.fAudioRecord)
                    {
                        xil_printf("\r\nStill
Recording...\r\n");
                    }
                    else
                    {
                        xil_printf("\r\nStill Playing
back...\r\n");
                    }
                }
                break;
            case 'r':
                if (!Demo.fAudioRecord &&
!Demo.fAudioPlayback)
                {
                    xil_printf("\r\nStart
Recording...\r\n");
                    fnSetLineInput();
                    fnAudioRecord(sAxiDma,NR_A
UDIO_SAMPLES);
                    Demo.fAudioRecord = 1;
                }
                else
                {
                    if (Demo.fAudioRecord)
                    {
                        xil_printf("\r\nStill
Recording...\r\n");
                    }
                    else
                    {
                        xil_printf("\r\nStill Playing
back...\r\n");
                    }
                }
                break;
            case 'l':
                if (!Demo.fAudioRecord &&
!Demo.fAudioPlayback)
                {
                    xil_printf("\r\nStart Playback...");
                    fnSetLineOutput();
```

```
                    fnAudioPlay(sAxiDma,NR_AUDIO_SAMPLES);
                    Demo.fAudioPlayback = 1;
                }
                else
                {
                    if (Demo.fAudioRecord)
                    {
                        xil_printf("\r\nStill Recording...\r\n");
                    }
                    else
                    {
                        xil_printf("\r\nStill Playing back...\r\n");
                    }
                }
                break;
            default:
                break;
        }

        // Reset the user I/O flag
        Demo.chBtn = 0;
        Demo.fUserIOEvent = 0;


    }

  }

  xil_printf("\r\n--- Exiting main() --- \r\n");


  return XST_SUCCESS;

}
```

3.1.2 Demo.h:

```
#ifndef MAIN_H_
#define MAIN_H_

/***************************** Include Files *********************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "xil_io.h"
#include "xstatus.h"
#include "xparameters.h"
#include "xil_cache.h"


/************************** Constant Definitions *****************************/
#define RETURN_ON_FAILURE(x) if ((x) != XST_SUCCESS) return XST_FAILURE;

#define
DMA_DEV_ID      XPAR_AXIDMA_0_DEVICE_ID

#ifdef XPAR_V6DDR_0_S_AXI_BASEADDR
#define
DDR_BASE_ADDR      XPAR_V6DDR_0_S_AXI_BASEADDR
#elif XPAR_S6DDR_0_S0_AXI_BASEADDR
#define
DDR_BASE_ADDR      XPAR_S6DDR_0_S0_AXI_BASEADDR
#elif
XPAR_AXI_7SDDR_0_S_AXI_BASEADDR
#define
DDR_BASE_ADDR      XPAR_AXI_7SDDR_0_S_AXI_BASEADDR
#elif XPAR_MIG7SERIES_0_BASEADDR
```

```
#define
DDR_BASE_ADDR      XPAR_MIG7SERIES
_0_BASEADDR
#elif
XPAR_PS7_DDR_0_S_AXI_BASEADDR
#define
DDR_BASE_ADDR      XPAR_PS7_DDR_0_
S_AXI_BASEADDR
#endif



#ifndef DDR_BASE_ADDR
#warning CHECK FOR THE VALID DDR
ADDRESS IN XPARAMETERS.H, DEFAULT
SET TO 0x010000000
#define MEM_BASE_ADDR      0x010000000
#else
#define
MEM_BASE_ADDR      (DDR_BASE_ADDR
+ 0x10000000)
#endif

#ifdef XPAR_INTC_0_DEVICE_ID
#define
RX_INTR_ID     XPAR_INTC_0_AXIDMA_0
_S2MM_INTROUT_VEC_ID
#define
TX_INTR_ID     XPAR_INTC_0_AXIDMA_0
_MM2S_INTROUT_VEC_ID
#else
#define
RX_INTR_ID     XPAR_FABRIC_AXI_DMA
_0_S2MM_INTROUT_INTR
#define
TX_INTR_ID     XPAR_FABRIC_AXI_DMA_
0_MM2S_INTROUT_INTR
#endif

#define
TX_BUFFER_BASE     (MEM_BASE_ADDR
+ 0x00100000)
#define
RX_BUFFER_BASE     (MEM_BASE_ADDR
+ 0x00300000)
```

```
#define
RX_BUFFER_HIGH     (MEM_BASE_ADDR
+ 0x004FFFFF)

#ifdef XPAR_INTC_0_DEVICE_ID
#define
INTC_DEVICE_ID         XPAR_INTC_0_DEV
ICE_ID
#else
//#define
INTC_DEVICE_ID         XPAR_SCUGIC_SIN
GLE_DEVICE_ID
#endif

#ifdef XPAR_INTC_0_DEVICE_ID
 #define INTC       XIntc
 #define
INTC_HANDLER   XIntc_InterruptHandler
#else
 #define INTC       XScuGic
 #define
INTC_HANDLER   XScuGic_InterruptHandler
#endif


/***************************** Type
Definitions
*****************************/
typedef struct {
    u8 u8Verbose;
    u8 fUserIOEvent;
    u8 fVideoEvent;
    u8 fAudioRecord;
    u8 fAudioPlayback;
    u8 fDmaError;
    u8 fDmaS2MMEvent;
    u8 fDmaMM2SEvent;
    int fDVIClockLock;
    char chBtn;
    u8 fLinkEvent;
    u8 fLinkStatus;
    int linkSpeed;
    int mac;
    XStatus fMacStatus;
} sDemo_t;
```

```
/************************** Function
Prototypes
*****************************/


// This variable holds the demo related settings
volatile sDemo_t Demo;

#endif /* MAIN_H_ */
```

3.2.1 Audio.h

```
#ifndef AUDIO_H_
#define AUDIO_H_

#include "xparameters.h"
#include "xil_io.h"
#include "xiic.h"
#include "xil_printf.h"
#include "xil_cache.h"
#include "xstatus.h"
#include "sleep.h"
#include "dma.h"
#include "demo.h"

/************************** Constant
Definitions *****************************/
#define DDR_OFFSET            0x07F00000

// Base Addresses
#define
SW_ADDR               XPAR_SWITCHES_0_B
ASEADDR
#define
AUDIO_CTL_ADDR        XPAR_D_AXI_I2S
_AUDIO_0_AXI_L_BASEADDR

//SLave address of the ADAU audio controller
#define IIC_SLAVE_ADDR        0x1A //for
Zybo 0b0011010
```

```
#define
DDR_BASEADDR          XPAR_MIG_7SERI
ES_0_BASEADDR


//Bit field construction
struct bits {
   u32 u32bit0:1;
   u32 u32bit1:1;
   u32 u32bit2:1;
   u32 u32bit3:1;
   u32 u32bit4:1;
   u32 u32bit5:1;
   u32 u32bit6:1;
   u32 u32bit7:1;
   u32 u32bit8:1;
   u32 u32bit9:1;
   u32 u32bit10:1;
   u32 u32bit11:1;
   u32 u32bit12:1;
   u32 u32bit13:1;
   u32 u32bit14:1;
   u32 u32bit15:1;
   u32 u32bit16:1;
   u32 u32bit17:1;
   u32 u32bit18:1;
   u32 u32bit19:1;
   u32 u32bit20:1;
   u32 u32bit21:1;
   u32 u32bit22:1;
   u32 u32bit23:1;
   u32 u32bit24:1;
   u32 u32bit25:1;
   u32 u32bit26:1;
   u32 u32bit27:1;
   u32 u32bit28:1;
   u32 u32bit29:1;
   u32 u32bit30:1;
   u32 u32bit31:1;
};

union ubitField{
   u8 rgu8[4];
   u32 l;
   struct bits bit;
```

```
};


// I2S Status Register Flags
enum i2sStatusFlags {
   TX_FIFO_EMPTY              = 0,
   TX_FIFO_FULL               = 1,
   RX_FIFO_EMPTY              = 16,
   RX_FIFO_FULL               = 17
};

// I2S Fifo Control Register Bits
enum i2sFifoControlBits {
   TX_FIFO_WR_EN              = 0,
   RX_FIFO_RD_EN              = 1,
   TX_FIFO_RST                = 30,
   RX_FIFO_RST                = 31
};

// I2S Fifo Transfer Control Register Bits
enum i2sFifoTransferControlBits {
   TX_RS                      = 0,
   RX_RS                      = 1
};

// I2S CLK control register
enum i2sClockControlBits {
   SAMPLING_RATE_BIT0         = 0,
   SAMPLING_RATE_BIT1         = 1,
   SAMPLING_RATE_BIT2         = 2,
   SAMPLING_RATE_BIT3         = 3,
   MASTER_MODE_ENABLE         = 16,
};

//Audio controller registers

//Audio controller registers
enum i2sRegisters {
   I2S_RESET_REG              =
AUDIO_CTL_ADDR,
   I2S_TRANSFER_CONTROL_REG   =
AUDIO_CTL_ADDR + 0x04,
   I2S_FIFO_CONTROL_REG       =
AUDIO_CTL_ADDR + 0x08,
   I2S_DATA_IN_REG            =
AUDIO_CTL_ADDR + 0x0c,
   I2S_DATA_OUT_REG           =
AUDIO_CTL_ADDR + 0x10,
   I2S_STATUS_REG             =
AUDIO_CTL_ADDR + 0x14,
   I2S_CLOCK_CONTROL_REG      =
AUDIO_CTL_ADDR + 0x18,
   I2S_PERIOD_COUNT_REG       =
AUDIO_CTL_ADDR + 0x1C,
   I2S_STREAM_CONTROL_REG     =
AUDIO_CTL_ADDR + 0x20
};


//ADAU internal register addresses
enum adauRegisterAdresses {
   R0_LEFT_ADC_VOL                    =
0x00,
   R1_RIGHT_ADC_VOL                   =
0x01,
   R2_LEFT_DAC_VOL                    =
0x02,
   R3_RIGHT_DAC_VOL                   =
0x03,
   R4_ANALOG_PATH                     =
0x04,
   R5_DIGITAL_PATH                    =
0x05,
   R6_POWER_MGMT                      =
0x06,
   R7_DIGITAL_IF                      =
0x07,
   R8_SAMPLE_RATE                     =
0x08,
   R9_ACTIVE                          = 0x09,
   R15_SOFTWARE_RESET
= 0x0F,
   R16_ALC_CONTROL_1
= 0x10,
   R17_ALC_CONTROL_2
= 0x11,
   R18_ALC_CONTROL_2
= 0x12
};
```

```
/************************** Variable Definitions ****************************/

// general reg
extern u8 u8Verbose;

/************************** Function Definitions ****************************/

XStatus fnAudioWriteToReg(u8 u8RegAddr, u16 u8Data);
XStatus fnAudioReadFromReg(u8 u8RegAddr, u8 *u8RxData);
//XStatus fnAudioPllConfig();
XStatus fnAudioStartupConfig ();
XStatus fnInitAudio();
void fnAudioRecord(XAxiDma AxiDma, u32 u32NrSamples);
void fnAudioPlay(XAxiDma AxiDma, u32 u32NrSamples);
void fnSetLineInput();
void fnSetLineOutput();
void fnSetMicInput();
void fnSetHpOutput();

#endif /* AUDIO_H_ */
```

3.2.2 Audio.c

```
#include "audio.h"
#include "demo.h"

/************************** Variable Definitions ****************************/

extern volatile sDemo_t Demo;

/*****************************************************************************
*
 * Function to write one byte (8-bits) to one of the registers from the audio
 * controller.
 *
 * @param   u8RegAddr is the LSB part of the register address (0x40xx).
 * @param   u8Data is the data byte to write.
 *
 * @return  XST_SUCCESS if all the bytes have been sent to Controller.
 *          XST_FAILURE otherwise.
 ******************************************************************************/
XStatus fnAudioWriteToReg(u8 u8RegAddr, u16 u8Data) {

   u8 u8TxData[2];
   u8 u8BytesSent;

   u8TxData[0] = u8RegAddr << 1;
   u8TxData[0] = u8TxData[0] | ((u8Data>>8) & 0b1);

   u8TxData[1] = u8Data & 0xFF;

   u8BytesSent = XIic_Send(XPAR_IIC_0_BASEADDR, IIC_SLAVE_ADDR, u8TxData, 2, XIIC_STOP);

   //check if all the bytes where sent
   if (u8BytesSent != 3)
   {
      //return XST_FAILURE;
   }

   return XST_SUCCESS;
}

/*****************************************************************************
*
 * Function to read one byte (8-bits) from the register space of audio controller.
 *
 * @param   u8RegAddr is the LSB part of the register address (0x40xx).
```

```
 * @param   u8RxData is the returned value
 *
 * @return  XST_SUCCESS if the desired
number of bytes have been read from the
controller
 *          XST_FAILURE otherwise
 ****************************************
****************************************
/
XStatus fnAudioReadFromReg(u8 u8RegAddr,
u8 *u8RxData) {

    u8 u8TxData[2];
    u8 u8BytesSent, u8BytesReceived;

    u8TxData[0] = u8RegAddr;
    u8TxData[1] = IIC_SLAVE_ADDR;

    u8BytesSent =
XIic_Send(XPAR_IIC_0_BASEADDR,
IIC_SLAVE_ADDR, u8TxData, 2,
XIIC_STOP);
    //check if all the bytes where sent
    if (u8BytesSent != 2)
    {
        return XST_FAILURE;
    }

    u8BytesReceived =
XIic_Recv(XPAR_IIC_0_BASEADDR,
IIC_SLAVE_ADDR, u8RxData, 1,
XIIC_STOP);
    //check if there are missing bytes
    if (u8BytesReceived != 1)
    {
        return XST_FAILURE;
    }

    return XST_SUCCESS;
}

/****************************************
****************************************
*
```

```
 * Configures audio codes's internal PLL. With
MCLK = 12.288 MHz it configures the
 * PLL for a VCO frequency = 49.152 MHz.
 *
 * @param   none.
 *
 * @return  XST_SUCCESS if PLL is locked
 ****************************************
****************************************
/
//XStatus fnAudioPllConfig() {
//
//  u8 u8TxData[8], u8RxData[6];
//  int Status;
//
//  Status =
fnAudioWriteToReg(R0_CLOCK_CONTROL,
0x0E);
//  if (Status == XST_FAILURE)
//  {
//      if (Demo.u8Verbose)
//      {
//          xil_printf("\r\nError: could not write
R0_CLOCK_CONTROL (0x0E)");
//      }
//      return XST_FAILURE;
//  }
//
//  // Write 6 bytes to R1
//  // For setting the PLL with a MCLK = 12.288
MHz the datasheet suggests the
//  // following configuration 0xXXXXXX2001
//  u8TxData[0] = 0x40;
//  u8TxData[1] = 0x02;
//  u8TxData[2] = 0x00; // byte 1
//  u8TxData[3] = 0x7D; // byte 2
//  u8TxData[4] = 0x00; // byte 3
//  u8TxData[5] = 0x0C; // byte 4
//  u8TxData[6] = 0x20; // byte 5
//  u8TxData[7] = 0x01; // byte 6
//
//  Status =
XIic_Send(XPAR_IIC_0_BASEADDR,
IIC_SLAVE_ADDR, u8TxData, 8,
XIIC_STOP);
```

```
//  if (Status != 8)
//  {
//      if (Demo.u8Verbose)
//      {
//          xil_printf("\r\nError: could not send data
to R1_PLL_CONTROL (0xXXXXXX2001)");
//      }
//      return XST_FAILURE;
//  }
//  // Poll PLL Lock bit
//  u8TxData[0] = 0x40;
//  u8TxData[1] = 0x02;
//
//  //Wait for the PLL to lock
//  do {
//      XIic_Send(XPAR_IIC_0_BASEADDR,
IIC_SLAVE_ADDR, u8TxData, 2,
XIIC_STOP);
//
//      XIic_Recv(XPAR_IIC_0_BASEADDR,
IIC_SLAVE_ADDR, u8RxData, 6,
XIIC_STOP);
//      if(Demo.u8Verbose) {
//          xil_printf("\nAudio PLL R1 =
0x%x%x%x%x%x%x", u8RxData[0],
u8RxData[1],
//              u8RxData[2], u8RxData[3],
u8RxData[4], u8RxData[5]);
//      }
//  }
//  while((u8RxData[5] & 0x02) == 0);
//
//  //Set COREN
//  Status =
fnAudioWriteToReg(R0_CLOCK_CONTROL,
0x0F);
//  if (Status == XST_FAILURE)
//  {
//      if (Demo.u8Verbose)
//      {
//          xil_printf("\r\nError: could not write
R0_CLOCK_CONTROL (0x0F)");
//      }
//      return XST_FAILURE;
//  }
```

```
//
//  return XST_SUCCESS;
//}

/*************************************
*************************************
*
 * Configure the initial settings of the audio
controller, the majority of
 * these will remain unchanged during the
normal functioning of the code.
 * In order to generate a correct BCLK and
LRCK, which are crucial for the
 * correct operating of the controller, the
sampling rate must me set in the
 * I2S_TRANSFER_CONTROL_REG. The
sampling rate options are:
 *    "000" -  8 KHz
 *    "001" - 12 KHz
 *    "010" - 16 KHz
 *    "011" - 24 KHz
 *    "100" - 32 KHz
 *    "101" - 48 KHz
 *    "110" - 96 KHz
 * These options are valid only if the I2S
controller is in slave mode.
 * When In master mode the ADAU will
generate the appropriate BCLK and LRCLK
 * internally, and the sampling rates which will
be set in the
I2S_TRANSFER_CONTROL_REG
 * are ignored.
 *
 * @param    none.
 *
 * @return  XST_SUCCESS if the configuration
is successful
 *************************************
*************************************
/
XStatus fnAudioStartupConfig ()
{

   union ubitField uConfigurationVariable;
   int Status;
```

```
    // Configure the I2S controller for generating a valid sampling rate
    uConfigurationVariable.l = Xil_In32(I2S_CLOCK_CONTROL_REG);
    uConfigurationVariable.bit.u32bit0 = 1;
    uConfigurationVariable.bit.u32bit1 = 0;
    uConfigurationVariable.bit.u32bit2 = 1;
    Xil_Out32(I2S_CLOCK_CONTROL_REG, uConfigurationVariable.l);

    uConfigurationVariable.l = 0x00000000;

    //STOP_TRANSACTION
    uConfigurationVariable.bit.u32bit1 = 1;
    Xil_Out32(I2S_TRANSFER_CONTROL_REG, uConfigurationVariable.l);

    //STOP_TRANSACTION
    uConfigurationVariable.bit.u32bit1 = 0;
    Xil_Out32(I2S_TRANSFER_CONTROL_REG, uConfigurationVariable.l);

    //slave: I2S
    Status = fnAudioWriteToReg(R15_SOFTWARE_RESET, 0b000000000);
    Status = XST_SUCCESS;
    if (Status == XST_FAILURE)
    {
        if (Demo.u8Verbose)
        {
            xil_printf("\r\nError: could not write R15_SOFTWARE_RESET (0x00)");
        }
        return XST_FAILURE;
    }
    usleep(1000);
    Status = fnAudioWriteToReg(R6_POWER_MGMT, 0b000110000);
    if (Status == XST_FAILURE)
    {
        if (Demo.u8Verbose)
        {
            xil_printf("\r\nError: could not write R6_POWER_MGMT (0b000110000)");
        }
        return XST_FAILURE;
    }
    Status = fnAudioWriteToReg(R0_LEFT_ADC_VOL, 0b000010111);
    if (Status == XST_FAILURE)
    {
        if (Demo.u8Verbose)
        {
            xil_printf("\r\nError: could not write R0_LEFT_ADC_VOL (0b000010111)");
        }
        return XST_FAILURE;
    }
    Status = fnAudioWriteToReg(R1_RIGHT_ADC_VOL, 0b000010111);
    if (Status == XST_FAILURE)
    {
        if (Demo.u8Verbose)
        {
            xil_printf("\r\nError: could not write R0_LEFT_ADC_VOL (0b000010111)");
        }
        return XST_FAILURE;
    }
    Status = fnAudioWriteToReg(R2_LEFT_DAC_VOL, 0b101111001);
    if (Status == XST_FAILURE)
    {
        if (Demo.u8Verbose)
        {
            xil_printf("\r\nError: could not write R0_LEFT_ADC_VOL (0b000010111)");
        }
        return XST_FAILURE;
    }
    Status = fnAudioWriteToReg(R3_RIGHT_DAC_VOL, 0b101111001);
    if (Status == XST_FAILURE)
```

```
    {
        if (Demo.u8Verbose)
        {
            xil_printf("\r\nError: could not write
R0_LEFT_ADC_VOL (0b000010111)");
        }
        return XST_FAILURE;
    }
    Status =
fnAudioWriteToReg(R4_ANALOG_PATH,
0b000000000);
    if (Status == XST_FAILURE)
    {
        if (Demo.u8Verbose)
        {
            xil_printf("\r\nError: could not write
R0_LEFT_ADC_VOL (0b000010111)");
        }
        return XST_FAILURE;
    }
    fnAudioWriteToReg(R5_DIGITAL_PATH,
0b000000000);
    fnAudioWriteToReg(R7_DIGITAL_IF,
0b000001010);
    fnAudioWriteToReg(R8_SAMPLE_RATE,
0b000000000);
    usleep(1000);
    fnAudioWriteToReg(R9_ACTIVE,
0b000000001);
    fnAudioWriteToReg(R6_POWER_MGMT,
0b000100000);


    return XST_SUCCESS;
}

/*****************************************
*****************************************
*
 * Initialize PLL and Audio controller over the
I2C bus
 *
 * @param   none
 *
 * @return  none.
 *****************************************
*****************************************
/
XStatus fnInitAudio()
{
    int Status;

    //Set the PLL and wait for Lock
    //Status = fnAudioPllConfig();
//  if (Status != XST_SUCCESS)
//  {
//      if (Demo.u8Verbose)
//      {
//          xil_printf("\r\nError: Could not lock
PLL");
//      }
//  }

    //Configure the ADAU registers
    Status = fnAudioStartupConfig();
    if (Status != XST_SUCCESS)
    {
        if (Demo.u8Verbose)
        {
            xil_printf("\r\nError: Failed I2C
Configuration");
        }
    }

    Demo.fAudioPlayback = 0;
    Demo.fAudioRecord = 0;

    return XST_SUCCESS;
}

/*****************************************
*****************************************
*
 * Configure the the I2S controller to receive
data, which will be stored locally
 * in a vector. (Mem)
 *
 * @param   u32NrSamples is the number of
samples to store.
 *
```

```
 * @return  none.
 ******************************************
*****************************************
/
void fnAudioRecord(XAxiDma AxiDma, u32
u32NrSamples)
{
   union ubitField uTransferVariable;

   if (Demo.u8Verbose)
   {
      xil_printf("\r\nEnter Record function");
   }

   uTransferVariable.l =
XAxiDma_SimpleTransfer(&AxiDma,(u32)
MEM_BASE_ADDR, 5*u32NrSamples,
XAXIDMA_DEVICE_TO_DMA);
   if (uTransferVariable.l != XST_SUCCESS)
   {
      if (Demo.u8Verbose)
         xil_printf("\n fail @ rec; ERROR: %d",
uTransferVariable.l);
   }

   // Send number of samples to recorde
   Xil_Out32(I2S_PERIOD_COUNT_REG,
u32NrSamples);

   // Start i2s initialization sequence
   uTransferVariable.l = 0x00000000;
   Xil_Out32(I2S_TRANSFER_CONTROL_RE
G, uTransferVariable.l);
   uTransferVariable.bit.u32bit1 = 1;
   Xil_Out32(I2S_TRANSFER_CONTROL_RE
G, uTransferVariable.l);

   // Enable Stream function to send data
(S2MM)
   Xil_Out32(I2S_STREAM_CONTROL_REG,
0x00000001);

   if (Demo.u8Verbose)
   {
      xil_printf("\r\nRecording function done");
   }
}

/******************************************
*****************************************
*
 * Configure the I2S controller to transmit data,
which will be read out from
 * the local memory vector (Mem)
 *
 * @param   u32NrSamples is the number of
samples to store.
 *
 * @return  none.
 ******************************************
*****************************************
/
void fnAudioPlay(XAxiDma AxiDma, u32
u32NrSamples)
{
   union ubitField uTransferVariable;

   if (Demo.u8Verbose)
   {
      xil_printf("\r\nEnter Playback function");
   }

   // Send number of samples to record
   Xil_Out32(I2S_PERIOD_COUNT_REG,
u32NrSamples);
   // Start i2s initialization sequence
   uTransferVariable.l = 0x00000000;
   Xil_Out32(I2S_TRANSFER_CONTROL_RE
G, uTransferVariable.l);
   uTransferVariable.bit.u32bit0 = 1;
   Xil_Out32(I2S_TRANSFER_CONTROL_RE
G, uTransferVariable.l);


   uTransferVariable.l =
XAxiDma_SimpleTransfer(&AxiDma,(u32)
MEM_BASE_ADDR, 5*u32NrSamples,
XAXIDMA_DMA_TO_DEVICE);
   if (uTransferVariable.l != XST_SUCCESS)
   {
```

```
        if (Demo.u8Verbose)
            xil_printf("\n fail @ play; ERROR: %d",
uTransferVariable.l);
    }

    // Enable Stream function to send data
(MM2S)
        Xil_Out32(I2S_STREAM_CONTROL_RE
G, 0x00000002);
    if (Demo.u8Verbose)
    {
        xil_printf("\r\nPlayback function done");
    }
}

/*****************************************
*****************************************
*
 * Configure the input path to MIC and disables
all other input paths.
 * For additional information pleas refer to the
ADAU1761 datasheet
 *
 * @param   none
 *
 * @return  none.
 *****************************************
*****************************************
/
void fnSetMicInput()
{
    //MX1AUXG = MUTE; MX2AUXG =
MUTE; LDBOOST = 0dB; RDBOOST = 0dB
    fnAudioWriteToReg(R4_ANALOG_PATH,
0b000010100);
    if (Demo.u8Verbose)
    {
        xil_printf("\r\nInput set to MIC");
    }
}

/*****************************************
*****************************************
*
 * Configure the input path to Line and disables
all other input paths
 * For additional information pleas refer to the
ADAU1761 datasheet
 *
 * @param   none
 *
 * @return  none.
 *****************************************
*****************************************
/
void fnSetLineInput()
{
    //MX1AUXG = 0dB; MX2AUXG = 0dB;
LDBOOST = MUTE; RDBOOST = MUTE
    fnAudioWriteToReg(R4_ANALOG_PATH,
0b000010010);
    fnAudioWriteToReg(R5_DIGITAL_PATH,
0b000000000);
    if (Demo.u8Verbose)
    {
        xil_printf("\r\nInput set to LineIn");
    }
}

/*****************************************
*****************************************
*
 * Configure the output path to Line and disables
all other output paths
 * For additional information pleas refer to the
ADAU1761 datasheet
 *
 * @param   none
 *
 * @return  none.
 *****************************************
*****************************************
/
void fnSetLineOutput()
{
    //zybo does not have a line output
    //MX3G1 = mute; MX3G2 = mute; MX4G1 =
mute; MX4G2 = mute;
```

```
   //fnAudioWriteToReg(R4_ANALOG_PATH,
0x00);

   if (Demo.u8Verbose)
   {
      xil_printf("\r\nOutput set to LineOut");
   }
}

/*************************************
*************************************
*
 * Configure the output path to Headphone and
disables all other output paths
 * For additional information pleas refer to the
ADAU1761 datasheet
 *
 * @param   none
 *
 * @return  none.
 *************************************
*************************************
/
void fnSetHpOutput()
{
   //MX5G3 = MUTE; MX5EN = MUTE;
MX6G4 = MUTE; MX6EN = MUTE
   fnAudioWriteToReg(R4_ANALOG_PATH,
0b000010110);
   fnAudioWriteToReg(R5_DIGITAL_PATH,
0b000000000);
   if (Demo.u8Verbose)
   {
      xil_printf("\r\nOutput set to HeadPhones");
   }
}
```

DMA.h

```
#ifndef DMA_H_
#define DMA_H_

#include "xparameters.h"
#include "xil_printf.h"
#include "xaxidma.h"

/************************** Variable
Definitions ****************************/


/************************** Function
Definitions ***************************/

void fnS2MMInterruptHandler (void
*Callback);
void fnMM2SInterruptHandler (void
*Callback);
```

```
XStatus fnConfigDma(XAxiDma *AxiDma);

#endif /* DMA_H_ */
```

DMA.c

```
#include "dma.h"
#include "demo.h"

/************************** Variable Definitions *****************************/

extern volatile sDemo_t Demo;
extern XAxiDma_Config *pCfgPtr;

/*****************************************************************************
 * This is the Interrupt Handler from the Stream to the MemoryMap. It is called
 * when an interrupt is trigger by the DMA
 *
 * @param   Callback is a pointer to S2MM channel of the DMA engine.
 *
 * @return  none
 *
 ******************************************************************************/
void fnS2MMInterruptHandler (void *Callback)
{
   u32 IrqStatus;
   int TimeOut;
   XAxiDma *AxiDmaInst = (XAxiDma *)Callback;
   //Read all the pending DMA interrupts
   IrqStatus = XAxiDma_IntrGetIrq(AxiDmaInst, XAXIDMA_DEVICE_TO_DMA);

   //Acknowledge pending interrupts
   XAxiDma_IntrAckIrq(AxiDmaInst, IrqStatus, XAXIDMA_DEVICE_TO_DMA);

   //If there are no interrupts we exit the Handler
   if (!(IrqStatus & XAXIDMA_IRQ_ALL_MASK))
   {
      return;
   }

   // If error interrupt is asserted, raise error flag, reset the
   // hardware to recover from the error, and return with no further
   // processing.
   if (IrqStatus & XAXIDMA_IRQ_ERROR_MASK)
   {
      Demo.fDmaError = 1;
      XAxiDma_Reset(AxiDmaInst);
      TimeOut = 1000;
      while (TimeOut)
      {
         if(XAxiDma_ResetIsDone(AxiDmaInst))
         {
            break;
         }
         TimeOut -= 1;
      }
      return;
   }

   if ((IrqStatus & XAXIDMA_IRQ_IOC_MASK))
   {
      Demo.fDmaS2MMEvent = 1;
   }
}

/*****************************************************************************
 * This is the Interrupt Handler from the MemoryMap to the Stream. It is called
```

```
 * when an interrupt is trigger by the DMA
 *
 * @param   Callback is a pointer to MM2S
channel of the DMA engine.
 *
 * @return  none
 *
 ************************************
************************************
/
void fnMM2SInterruptHandler (void *Callback)
{

   u32 IrqStatus;
   int TimeOut;
   XAxiDma *AxiDmaInst = (XAxiDma
*)Callback;

   //Read all the pending DMA interrupts
   IrqStatus =
XAxiDma_IntrGetIrq(AxiDmaInst,
XAXIDMA_DMA_TO_DEVICE);
   //Acknowledge pending interrupts
   XAxiDma_IntrAckIrq(AxiDmaInst, IrqStatus,
XAXIDMA_DMA_TO_DEVICE);
   //If there are no interrupts we exit the Handler
   if (!(IrqStatus &
XAXIDMA_IRQ_ALL_MASK))
   {
      return;
   }

   // If error interrupt is asserted, raise error flag,
reset the
   // hardware to recover from the error, and
return with no further
   // processing.
   if (IrqStatus &
XAXIDMA_IRQ_ERROR_MASK){
      Demo.fDmaError = 1;
      XAxiDma_Reset(AxiDmaInst);
      TimeOut = 1000;
      while (TimeOut)
      {
         if(XAxiDma_ResetIsDone(AxiDmaInst)
)
         {
            break;
         }
         TimeOut -= 1;
      }
      return;
   }
   if ((IrqStatus &
XAXIDMA_IRQ_IOC_MASK))
   {
      Demo.fDmaMM2SEvent = 1;
   }
}

/************************************
************************************
*
 * Function to configure the DMA in Interrupt
mode, this implies that the scatter
 * gather function is disabled. Prior to calling
this function, the user must
 * make sure that the Interrupts and the Interrupt
Handlers have been configured
 *
 * @return  XST_SUCCESS - if configuration
was successful
 *          XST_FAILURE - when the
specification are not met
 ************************************
************************************
/
XStatus fnConfigDma(XAxiDma *AxiDma)
{
   int Status;
   XAxiDma_Config *pCfgPtr;

   //Make sure the DMA hardware is present in
the project
   //Ensures that the DMA hardware has been
loaded
   pCfgPtr =
XAxiDma_LookupConfig(XPAR_AXIDMA_0
_DEVICE_ID);
```

```
    if (!pCfgPtr)
    {
        if (Demo.u8Verbose)
        {
            xil_printf("\r\nNo config found for %d",
XPAR_AXIDMA_0_DEVICE_ID);
        }
        return XST_FAILURE;
    }

    //Initialize DMA
    //Reads and sets all the available information
    //about the DMA to the AxiDma variable
    Status = XAxiDma_CfgInitialize(AxiDma,
pCfgPtr);
    if (Status != XST_SUCCESS)
    {
        if (Demo.u8Verbose)
        {
            xil_printf("\r\nInitialization failed %d");
        }
        return XST_FAILURE;
    }

    //Ensures that the Scatter Gather mode is not
active
    if(XAxiDma_HasSg(AxiDma))
    {
        if (Demo.u8Verbose)
        {

            xil_printf("\r\nDevice configured as SG
mode");
        }
        return XST_FAILURE;
    }

    //Disable all the DMA related Interrupts
    XAxiDma_IntrDisable(AxiDma,
XAXIDMA_IRQ_ALL_MASK,
XAXIDMA_DEVICE_TO_DMA);
    XAxiDma_IntrDisable(AxiDma,
XAXIDMA_IRQ_ALL_MASK,
XAXIDMA_DMA_TO_DEVICE);
```

```
    //Enable all the DMA Interrupts
    XAxiDma_IntrEnable(AxiDma,
XAXIDMA_IRQ_ALL_MASK,
XAXIDMA_DEVICE_TO_DMA);
    XAxiDma_IntrEnable(AxiDma,
XAXIDMA_IRQ_ALL_MASK,
XAXIDMA_DMA_TO_DEVICE);

    return XST_SUCCESS;
}
```

iic.h

```
#ifndef IIC_H_
#define IIC_H_

#include "xiic.h"

#define RETURN_ON_FAILURE(x) if ((x) !=
XST_SUCCESS) return XST_FAILURE;

typedef struct
{
    u8 rgbMac[6];
} macAddress_t;

XStatus fnInitIic(XIic *psIic);
XStatus fnReadMACAsync(XIic *psIic,
macAddress_t *pMac, XStatus *pfMacReady);

#endif /* IIC_H_ */
```

iic.c

```
#include "xparameters.h"
#include "iic.h"
#include <string.h>

#define IIC_DEVICE_ID
XPAR_AXI_IIC_0_DEVICE_ID
#define EEPROM_ADDRESS 0x57 //0xAE as
8-bit
#define MAC_MEM_ADDRESS 0xFA
```

```
typedef u8 memAddress_t; //Change to u16, if EEPROM uses 16-bit register address

static u8 rgbWriteBuf[sizeof(memAddress_t)];

static void ReadMACSendHandler(XIic *psIic, int ByteCount);
static void ReadMACReceiveHandler(XIic *psIic, int ByteCount);
static void StatusHandler(XIic *InstancePtr, int Event);

static macAddress_t *pgMac;
static XStatus *pgfMacStatus;
/*
 * IIC controller init function. Uses interrupts which have to be intialized and enabled
 * outside of this function.
 */
XStatus fnInitIic(XIic *psIic)
{
   XIic_Config *psConfig;

   // Initialize the IIC driver so that it is ready to use.
   psConfig = XIic_LookupConfig(IIC_DEVICE_ID);
   if (psConfig == NULL) {
      return XST_FAILURE;
   }


   RETURN_ON_FAILURE(XIic_CfgInitialize(psIic, psConfig,
         psConfig->BaseAddress));

   RETURN_ON_FAILURE(XIic_DynamicInitialize(psIic));

   return XST_SUCCESS;
}

XStatus fnReadMACAsync(XIic *psIic, macAddress_t *pMac, XStatus *pfMacStatus)
{
   memAddress_t memAddress = MAC_MEM_ADDRESS;

   pgMac = pMac; pgfMacStatus = pfMacStatus;

   memset(pgMac, 0, sizeof(*pgMac));
   *pgfMacStatus = XST_DEVICE_BUSY;

   psIic->Stats.TxErrors = 0;

   // Set the Handlers for transmit and reception.
   XIic_SetSendHandler(psIic, psIic,
         (XIic_Handler) ReadMACSendHandler);
   XIic_SetRecvHandler(psIic, psIic,
         (XIic_Handler) ReadMACReceiveHandler);
   XIic_SetStatusHandler(psIic, psIic,
         (XIic_StatusHandler) StatusHandler);

   // Use repeated start when sending the register address
   XIic_SetOptions(psIic, XIic_GetOptions(psIic) | XII_REPEATED_START_OPTION);

   // Start the IIC device.
   RETURN_ON_FAILURE(XIic_Start(psIic));

   // Set the EEPROM slave address
   XIic_SetAddress(psIic, XII_ADDR_TO_SEND_TYPE, EEPROM_ADDRESS);

   // 8/16-bit register addressing
   if (sizeof(memAddress_t) == 2)
   {
      rgbWriteBuf[0] = (u8) (memAddress >> 8);
      rgbWriteBuf[1] = (u8) memAddress ;
   }
   else
   {
      rgbWriteBuf[0] = (u8) memAddress ;
   }
```

```
    // Send register address
    RETURN_ON_FAILURE(XIic_DynMasterSend(psIic, &rgbWriteBuf[0], sizeof(memAddress_t)));

    return XST_SUCCESS;
}

// This will be called when the Register Address is sent
static void ReadMACSendHandler(XIic *psIic, int ByteCount)
{
    // Turn off repeated start for the read part
    XIic_SetOptions(psIic, XIic_GetOptions(psIic) & ~XII_REPEATED_START_OPTION);

    // Read MAC address
    if (XST_SUCCESS != XIic_DynMasterRecv(psIic, (u8*)pgMac, sizeof(*pgMac)))
    {
        *pgfMacStatus = XST_RECV_ERROR;
        XIic_Stop(psIic);
    }
}

// This will be called when the MAC Address is read
static void ReadMACReceiveHandler(XIic *psIic, int ByteCount)
{
    *pgfMacStatus = XST_SUCCESS;
    //We have finished the transfer
    XIic_Stop(psIic);
}

/*****************************************************************************/
/**
* This Status handler is called asynchronously from an interrupt
* context and indicates the events that have occurred.
*
* @param    InstancePtr is a pointer to the IIC driver instance for which
*       the handler is being called for.
* @param    Event indicates the condition that has occurred.
*
* @return   None.
*
* @note     None.
*
*****************************************************************************/
static void StatusHandler(XIic *psIic, int Event)
{
    switch (Event)
    {
        case XII_BUS_NOT_BUSY_EVENT:
            //If the bus was busy when we tried a send and now it seems to be free
            if (pgfMacStatus && *pgfMacStatus == XST_SEND_ERROR)
            {
                fnReadMACAsync(psIic, pgMac, pgfMacStatus);
            }
            break;

        case XII_ARB_LOST_EVENT:
        case XII_SLAVE_NO_ACK_EVENT:
            if (pgfMacStatus) *pgfMacStatus = XST_SEND_ERROR;
            break;
    }
}
```

intc.c:

```
#include "intc.h"
#include "xparameters.h"
```

```
XStatus fnInitInterruptController(intc *psIntc)
{
    int result = 0;
#ifdef XPAR_XINTC_NUM_INSTANCES
    // Init driver instance
    RETURN_ON_FAILURE(XIntc_Initialize(ps
Intc, INTC_DEVICE_ID));

    // Start interrupt controller
    RETURN_ON_FAILURE(XIntc_Start(psIntc
, XIN_REAL_MODE));

    Xil_ExceptionInit();
    // Register the interrupt controller handler
with the exception table.
    // This is in fact the ISR dispatch routine,
which calls our ISRs
    Xil_ExceptionRegisterHandler(XIL_EXCEP
TION_ID_INT,
                (Xil_ExceptionHandler)XIntc_Interru
ptHandler,
                psIntc);

#endif
#ifdef XPAR_SCUGIC_0_DEVICE_ID
    XScuGic_Config *IntcConfig;

    /*
     * Initialize the interrupt controller driver so
that it is ready to
     * use.
     */
    IntcConfig =
XScuGic_LookupConfig(INTC_DEVICE_ID);
    if (NULL == IntcConfig) {
        return XST_FAILURE;
    }

    result = XScuGic_CfgInitialize(psIntc,
IntcConfig, IntcConfig->CpuBaseAddress);
    if (result != XST_SUCCESS) {
        return XST_FAILURE;
    }



#endif
    //Xil_ExceptionEnable();

    return XST_SUCCESS;
}

/*
 * This function enables interrupts and connects
interrupt service routines declared in
 * an interrupt vector table
 */
void fnEnableInterrupts(intc *psIntc, const ivt_t
*prgsIvt, unsigned int csIVectors)
{
    unsigned int isIVector;

    Xil_AssertVoid(psIntc != NULL);
    Xil_AssertVoid(psIntc->IsReady ==
XIL_COMPONENT_IS_READY);



    /* Hook up interrupt service routines from
IVT */
    for (isIVector = 0; isIVector < csIVectors;
isIVector++)
    {
#ifdef __MICROBLAZE__
        XIntc_Connect(psIntc,
prgsIvt[isIVector].id, prgsIvt[isIVector].handler,
prgsIvt[isIVector].pvCallbackRef);

        /* Enable the interrupt vector at the
interrupt controller */
        XIntc_Enable(psIntc,
prgsIvt[isIVector].id);
#else
        XScuGic_SetPriorityTriggerType(psIntc,
prgsIvt[isIVector].id, 0xA0, 0x3);
```

```
        XScuGic_Connect(psIntc,
prgsIvt[isIVector].id, prgsIvt[isIVector].handler,
prgsIvt[isIVector].pvCallbackRef);
        XScuGic_Enable(psIntc,
prgsIvt[isIVector].id);

#endif
    }
    Xil_ExceptionInit();
    Xil_ExceptionRegisterHandler(XIL_EXCEP
TION_ID_INT,
(Xil_ExceptionHandler)INTC_HANDLER,
psIntc);
    Xil_ExceptionEnable();

}
```

Intc.h

```
#ifndef INTC_H_
#define INTC_H_

#include "xstatus.h"
#ifdef XPAR_INTC_0_DEVICE_ID
 #include "xintc.h"
#else
 #include "xscugic.h"
#endif

#define RETURN_ON_FAILURE(x) if ((x) !=
XST_SUCCESS) return XST_FAILURE;

/*
 * Structure for interrupt id, handler and callback
reference
 */
typedef struct {
    u8 id;
    XInterruptHandler handler;
    void *pvCallbackRef;
} ivt_t;

#ifdef XPAR_INTC_0_DEVICE_ID
 XStatus fnInitInterruptController(XIntc
*psIntc);
 void fnEnableInterrupts(XIntc *psIntc, const
ivt_t *prgsIvt, unsigned int csIVectors);
#define intc XIntc
#define INTC_DEVICE_ID
XPAR_INTC_0_DEVICE_ID
#else
 XStatus fnInitInterruptController(XScuGic
*psIntc);
 void fnEnableInterrupts(XScuGic *psIntc, const
ivt_t *prgsIvt, unsigned int csIVectors);
#define intc XScuGic
#define INTC_DEVICE_ID
XPAR_PS7_SCUGIC_0_DEVICE_ID
#define
INTC_HANDLER    XScuGic_InterruptHandler
#endif


#endif /* INTC_H_ */
```

Userio.h

```
#include <stdio.h>
#include "xparameters.h"
#include "userio.h"
#include "demo.h"

#define USERIO_DEVICE_ID    0

extern volatile sDemo_t Demo;

void fnUpdateLedsFromSwitches(XGpio
*psGpio);

XStatus fnInitUserIO(XGpio *psGpio)
{
    /* Initialize the GPIO driver. If an error
occurs then exit */
    RETURN_ON_FAILURE(XGpio_Initialize(p
sGpio, USERIO_DEVICE_ID));

    /*
     * Perform a self-test on the GPIO.  This is a
minimal test and only
```

```
    * verifies that there is not any bus error when reading the data
    * register
    */
   RETURN_ON_FAILURE(XGpio_SelfTest(psGpio));

   /*
    * Setup direction register so the switches and buttons are inputs and the LED is
    * an output of the GPIO
    */
   XGpio_SetDataDirection(psGpio, BTN_SW_CHANNEL, BTNS_SWS_MASK);

   fnUpdateLedsFromSwitches(psGpio);

   /*
    * Enable the GPIO channel interrupts so that push button can be
    * detected and enable interrupts for the GPIO device
    */
   XGpio_InterruptEnable(psGpio, BTN_SW_INTERRUPT);
   XGpio_InterruptGlobalEnable(psGpio);

   return XST_SUCCESS;
}

void fnUpdateLedsFromSwitches(XGpio *psGpio)
{
   static u32 dwPrevButtons = 0;
   u32  dwBtn;
   u32 dwBtnSw;

   dwBtnSw = XGpio_DiscreteRead(psGpio, BTN_SW_CHANNEL);
   dwBtn = dwBtnSw & (BTNU_MASK|BTNR_MASK|BTND_MASK|BTNL_MASK|BTNC_MASK);
   if (dwBtn==0){//No buttons pressed?
      Demo.fUserIOEvent = 0;
      dwPrevButtons = dwBtn;
      return;
   }
   // Has anything changed?
   if ((dwBtn ^ dwPrevButtons))
   {

      u32 dwChanges = 0;


      dwChanges = dwBtn ^ dwPrevButtons;
      if (dwChanges & BTNU_MASK) {
         Demo.chBtn = 'u';
         if(Demo.u8Verbose) {
            xil_printf("\r\nBTNU");
         }
      }
      if (dwChanges & BTNR_MASK) {
         Demo.chBtn = 'r';
         if(Demo.u8Verbose) {
            xil_printf("\r\nBTNR");
         }
      }
      if (dwChanges & BTND_MASK) {
         Demo.chBtn = 'd';
         if(Demo.u8Verbose) {
            xil_printf("\r\nBTND");
         }
      }
      if (dwChanges & BTNL_MASK) {
         Demo.chBtn = 'l';
         if(Demo.u8Verbose) {
            xil_printf("\r\nBTNL");
         }
      }
      if (dwChanges & BTNC_MASK) {
         Demo.chBtn = 'c';
         if(Demo.u8Verbose) {
            xil_printf("\r\nBTNC");
         }
      }

      // Keep values in mind
      //dwPrevSwitches = dwSw;
      Demo.fUserIOEvent = 1;
      dwPrevButtons = dwBtn;
```

```
    }
}

/*
 * Default interrupt service routine
 * Lights up LEDs above active switches.
Pressing any of the buttons inverts LEDs.
 */
void fnUserIOIsr(void *pvInst)
{
    XGpio *psGpio = (XGpio*)pvInst;

    /*
     * Disable the interrupt
     */
    XGpio_InterruptGlobalDisable(psGpio);




    /*
     * Check if the interrupt interests us
     */
    if ((XGpio_InterruptGetStatus(psGpio) &
BTN_SW_INTERRUPT) !=
            BTN_SW_INTERRUPT) {
        XGpio_InterruptGlobalEnable(psGpio);
        return;
    }

    fnUpdateLedsFromSwitches(psGpio);



     /* Clear the interrupt such that it is no longer
pending in the GPIO */

    XGpio_InterruptClear(psGpio,
BTN_SW_INTERRUPT);

    /*
    * Enable the interrupt
    */
    XGpio_InterruptGlobalEnable(psGpio);
}
```

Userio.c:

```
#include <stdio.h>
#include "xparameters.h"
#include "userio.h"
#include "demo.h"

#define USERIO_DEVICE_ID    0

extern volatile sDemo_t Demo;

void fnUpdateLedsFromSwitches(XGpio
*psGpio);

XStatus fnInitUserIO(XGpio *psGpio)
{
    /* Initialize the GPIO driver. If an error
occurs then exit */
    RETURN_ON_FAILURE(XGpio_Initialize(p
sGpio, USERIO_DEVICE_ID));

    /*
     * Perform a self-test on the GPIO.  This is a
minimal test and only
     * verifies that there is not any bus error when
reading the data
     * register
     */
    RETURN_ON_FAILURE(XGpio_SelfTest(p
sGpio));

    /*
     * Setup direction register so the switches and
buttons are inputs and the LED is
     * an output of the GPIO
     */
    XGpio_SetDataDirection(psGpio,
BTN_SW_CHANNEL, BTNS_SWS_MASK);

    fnUpdateLedsFromSwitches(psGpio);

    /*
```

```
    * Enable the GPIO channel interrupts so that push button can be
    * detected and enable interrupts for the GPIO device
    */
   XGpio_InterruptEnable(psGpio, BTN_SW_INTERRUPT);
   XGpio_InterruptGlobalEnable(psGpio);

   return XST_SUCCESS;
}

void fnUpdateLedsFromSwitches(XGpio *psGpio)
{
   static u32 dwPrevButtons = 0;
   u32  dwBtn;
   u32 dwBtnSw;

   dwBtnSw = XGpio_DiscreteRead(psGpio, BTN_SW_CHANNEL);
   dwBtn = dwBtnSw & (BTNU_MASK|BTNR_MASK|BTND_MASK|BTNL_MASK|BTNC_MASK);
   if (dwBtn==0){//No buttons pressed?
      Demo.fUserIOEvent = 0;
      dwPrevButtons = dwBtn;
      return;
   }
   // Has anything changed?
   if ((dwBtn ^ dwPrevButtons))
   {

      u32 dwChanges = 0;


      dwChanges = dwBtn ^ dwPrevButtons;
      if (dwChanges & BTNU_MASK) {
         Demo.chBtn = 'u';
         if(Demo.u8Verbose) {
            xil_printf("\r\nBTNU");
         }
      }
      if (dwChanges & BTNR_MASK) {
         Demo.chBtn = 'r';
         if(Demo.u8Verbose) {
            xil_printf("\r\nBTNR");
         }
      }
      if (dwChanges & BTND_MASK) {
         Demo.chBtn = 'd';
         if(Demo.u8Verbose) {
            xil_printf("\r\nBTND");
         }
      }
      if (dwChanges & BTNL_MASK) {
         Demo.chBtn = 'l';
         if(Demo.u8Verbose) {
            xil_printf("\r\nBTNL");
         }
      }
      if (dwChanges & BTNC_MASK) {
         Demo.chBtn = 'c';
         if(Demo.u8Verbose) {
            xil_printf("\r\nBTNC");
         }
      }

      // Keep values in mind
      //dwPrevSwitches = dwSw;
      Demo.fUserIOEvent = 1;
      dwPrevButtons = dwBtn;
   }
}

/*
 * Default interrupt service routine
 * Lights up LEDs above active switches. Pressing any of the buttons inverts LEDs.
 */
void fnUserIOIsr(void *pvInst)
{
   XGpio *psGpio = (XGpio*)pvInst;

   /*
    * Disable the interrupt
    */
   XGpio_InterruptGlobalDisable(psGpio);
```

```
   /*
    * Check if the interrupt interests us
    */
   if ((XGpio_InterruptGetStatus(psGpio) &
BTN_SW_INTERRUPT) !=
         BTN_SW_INTERRUPT) {
      XGpio_InterruptGlobalEnable(psGpio);
      return;
   }

   fnUpdateLedsFromSwitches(psGpio);
```

```
    /* Clear the interrupt such that it is no longer
pending in the GPIO */

   XGpio_InterruptClear(psGpio,
BTN_SW_INTERRUPT);

   /*
   * Enable the interrupt
   */
   XGpio_InterruptGlobalEnable(psGpio);
}
```