California State University - Fresno
Lyles College of Engineering
Department of Electrical and Computer Engineering

Technical Report

Experiment Title: DE2-115 Music Player Project
Course Title: ECE 178 Embedded Systems
Instructor: Dr. Reza Raeisi
Date Submitted: 11 December 2022

Prepared By:	Sections Written:
Jason Lawrence Alsola	Section 4.2, 6 Appendices A, B, D
Puya Fard	Section 2, 3.1, 3.2, 3.3
Zoe Statzer	Section 1, 4.1, 5 Appendices A, B, C, D

INSTRUCTOR SECTION

Comments:

Final Grade: Team Member 1: Jason Lawrence Alsola
Team Member 2: Puya Fard
Team Member 3: Zoe Statzer

TABLE OF CONTENTS

1. STATEMENT OF OBJECTIVES

This project was performed to create a program that would make the DE2-115 board into a music player that utilized the following functionalities: read in wav files from an SD card, skip to the next song, go to the previous song, play or pause the current song, display a song time progress bar on the LEDRs, display the time that has elapsed in the song on the 7 segment displays, and output the audio of the audio CODEC.

2. THEORETICAL BACKGROUND

Eclipse IDE

The Eclipse Integrated Development Environment is a source navigator and editor, a source debugger and profiler, a compiler, linker, and assembler for C and C++.

The Nios II EDS provides a consistent software development environment that works for all Nios II processor systems. With the Nios II EDS running on a host computer, an Intel FPGA, and a JTAG download cable (such as an Intel FPGA USB-Blaster download cable), you can write programs for and communicate with any Nios II processor system. The Nios II SBT includes proprietary and open-source tools (such as the GNU C/C++ tool chain) for creating Nios II programs.

The Nios II EDS automates board support package (BSP) creation for Nios II processor-based systems, eliminating the need to spend time manually creating BSPs. The BSP provides a C/C++ runtime environment, insulating you from the hardware in your embedded system. Intel FPGA BSPs contain the Intel FPGA hardware abstraction layer (HAL), an optional RTOS, and device drivers. Each Nios II program you develop consists of an application project, optional user library projects, and a BSP project. You build your Nios II program to create an Executable and Linking Format File (.elf) which runs on a Nios II processor. The Nios II SBT creates software projects for you. Each project is based on a make file.

C is a high-level programming language.

Intel DE-series FPGA Boards

For this project we assume that the reader has access to an Intel DE-series board, such as the one shown in Figure 1. The figure depicts the DE3-115 board, which features an Intel Cyclone IV FPGA chip. The board provides a lot of other resources, such as memory chips, slider switches, push button keys, LEDs, audio input/output, video input (NTSC/PAL decoder) and video output (VGA). It also provides several types of serial input/output connections, including a USB port for connecting the board to a personal computer. In this tutorial we will make use of only a few of the resources: the FPGA chip, slider switches, LEDs, and the USB port that connects to a computer. Although we have chosen the DE2-115 board as an example, the tutorial is pertinent for other DE-series boards that are described in the University Program section of Intel's website.

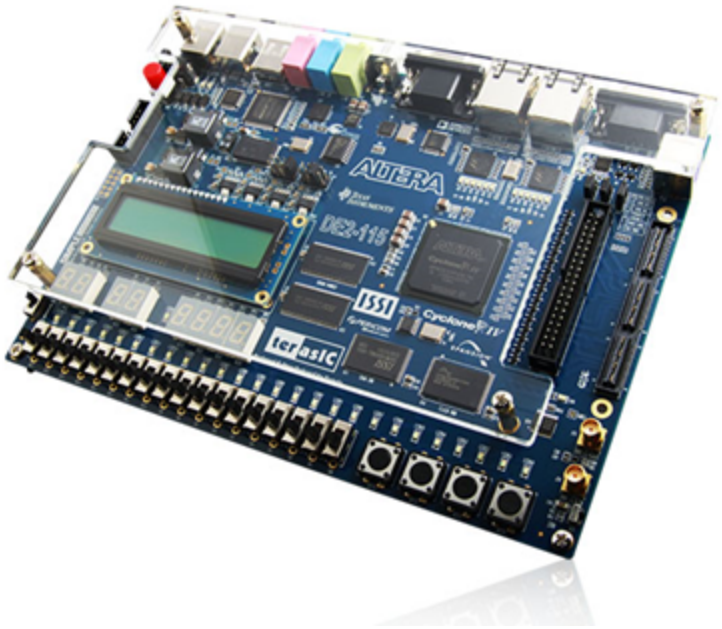

Figure 2.0: An Intel DE2-115 Board

Intel Nios II Embedded processor

We will use a simple hardware system that is shown in **Figure 2.1** It includes the Intel Nios® II embedded processor, which is a soft processor module defined as code in a hardware-description language. A Nios II module can be included as part of a larger system, and then that system can be implemented in an Intel FPGA chip by using the Quartus Prime software.

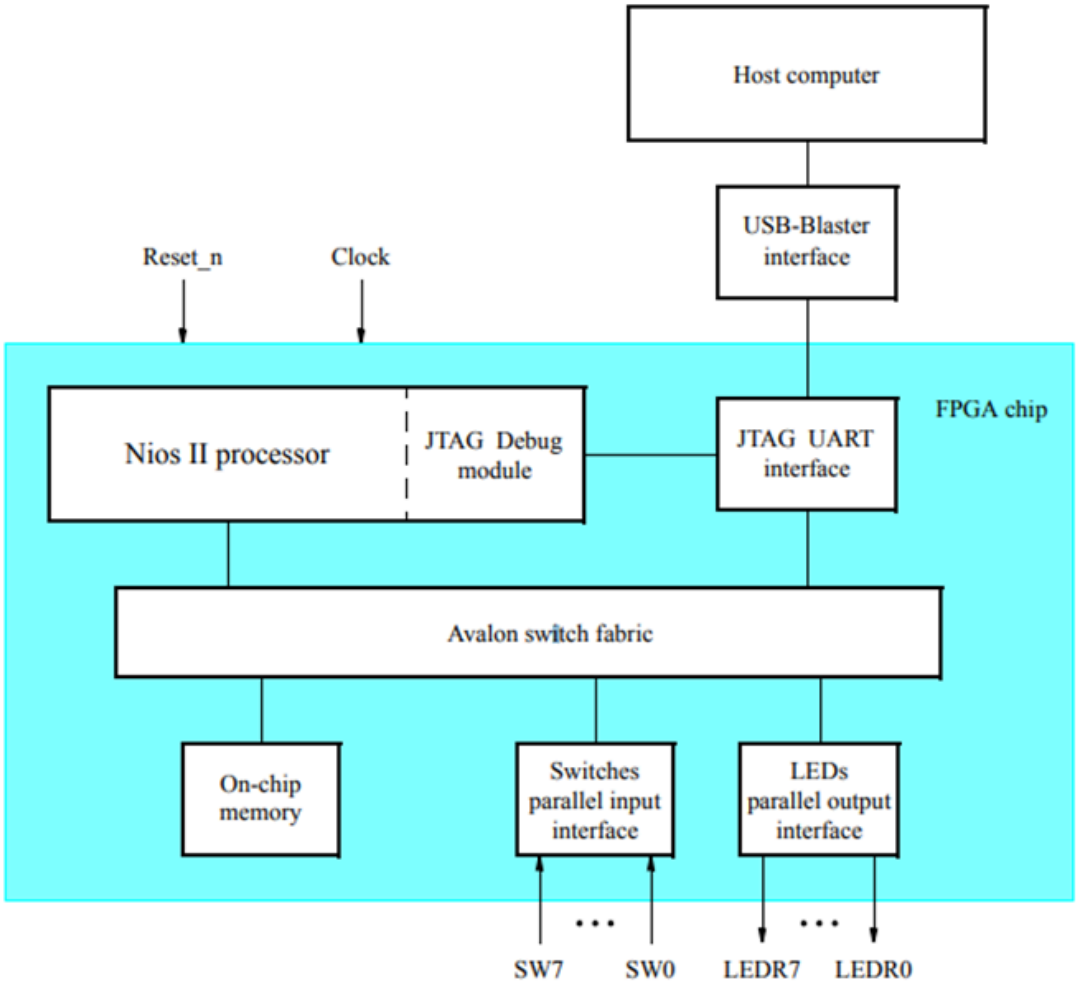

Figure 2.1: A Simple example of a Nios II system

The Nios II EDS provides a consistent software development environment that works for all Nios II processor systems. With the Nios II EDS running on a host computer, an Intel FPGA FPGA, and a JTAG download cable (such as an Intel FPGA USB-Blaster download cable), you can write programs for and communicate with any Nios II processor system.

The Nios II SBT includes proprietary and open-source tools (such as the GNU C/C++ tool chain) for creating Nios II programs. The Nios II EDS automates board support package (BSP) creation for Nios II processor-based systems, eliminating the need to spend time manually creating BSPs. The BSP provides a C/C++ runtime environment, insulating you from the hardware in your embedded system. Intel FPGA BSPs contain the Intel FPGA hardware abstraction layer (HAL), an optional RTOS, and device drivers.

Each Nios II program you develop consists of an application project, optional user library projects, and a BSP project. You build your Nios II program to create an Executable and Linking Format File (.elf) which runs on a Nios II processor. The Nios II SBT creates software projects for you. Each project is based on a makefile.

An interrupt sends the Program Counter (PC) to the Interrupt Handling Routine (IHR) to identify the priority of the interrupts and to perform their Interrupt Service Routine (ISR), or what a program that each peripheral has related to its interrupt. To perform an interrupt, three conditions must be met:

1. The PIE-bit in CTL0 is set to 1 (The Interrupt Handler is accepting interrupts)
2. An IRQ (IRQ_k) is asserted (A peripheral sends an interrupt)
3. The corresponding interrupt-enable bit $CTL3_k$ is set to 1

The Seven-Segment Display, as set in a previous experiment, is memory-mapped at a specific address, with specific bits matching a certain segment on each of the displays. The first Display, HEX0, has its segments mapped to bits 0-6 at the first base address. The second Display, HEX1, has its segments mapped to bits 8-14 on the same address. This can be seen in **Figure 2.2**.

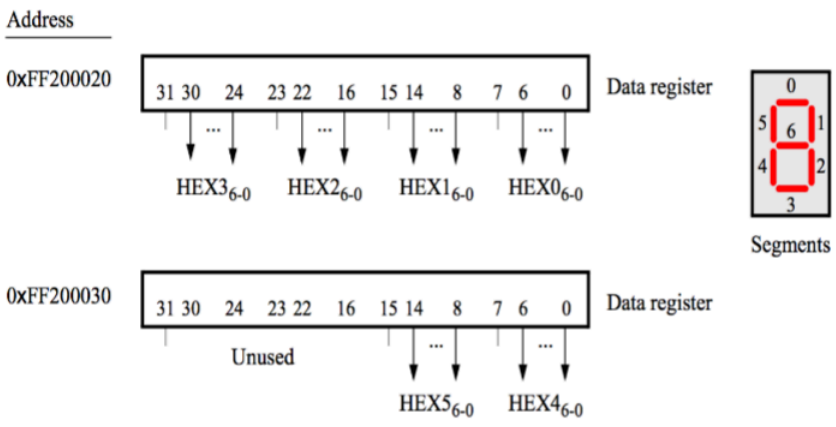

Figure 2.2: DE2-115 Memory-Mapped example of the Seven-Segment Display.

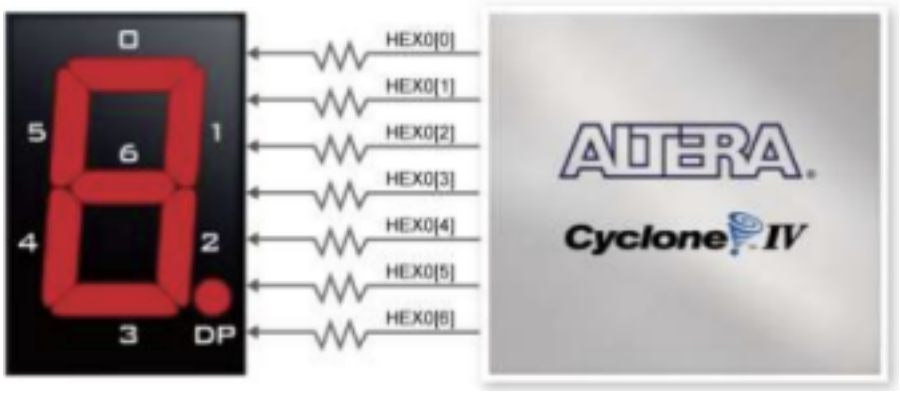

Figure 2.3: Schematic of 7-segment display

Schematic of 7 segment display is given above on **Figure 2.3**. The DE2-115 development board has eight 7-segment displays, each containing seven pins, HEX0-HEX6, in a common anode (active low) configuration. **Figure 2.4** shows the pin configuration for the 7-segment display, this is needed when creating the custom system in Qsys.

Signal Name	*FPGA Pin No.*	*Description*	*I/O Standard*
HEX0[0]	PIN_G18	Seven Segment Digit 0[0]	2.5V
HEX0[1]	PIN_F22	Seven Segment Digit 0[1]	2.5V
HEX0[2]	PIN_E17	Seven Segment Digit 0[2]	2.5V

HEX0[3]	PIN_L26	Seven Segment Digit 0[3]	Depending on JP7
HEX0[4]	PIN_L25	Seven Segment Digit 0[4]	Depending on JP7
HEX0[5]	PIN_J22	Seven Segment Digit 0[5]	Depending on JP7
HEX0[6]	PIN_H22	Seven Segment Digit 0[6]	Depending on JP7
HEX1[0]	PIN_M24	Seven Segment Digit 1[0]	Depending on JP7
HEX1[1]	PIN_Y22	Seven Segment Digit 1[1]	Depending on JP7
HEX1[2]	PIN_W21	Seven Segment Digit 1[2]	Depending on JP7
HEX1[3]	PIN_W22	Seven Segment Digit 1[3]	Depending on JP7
HEX1[4]	PIN_W25	Seven Segment Digit 1[4]	Depending on JP7
HEX1[5]	PIN_U23	Seven Segment Digit 1[5]	Depending on JP7
HEX1[6]	PIN_U24	Seven Segment Digit 1[6]	Depending on JP7
HEX2[0]	PIN_AA25	Seven Segment Digit 2[0]	Depending on JP7
HEX2[1]	PIN_AA26	Seven Segment Digit 2[1]	Depending on JP7
HEX2[2]	PIN_Y25	Seven Segment Digit 2[2]	Depending on JP7
HEX2[3]	PIN_W26	Seven Segment Digit 2[3]	Depending on JP7
HEX2[4]	PIN_Y26	Seven Segment Digit 2[4]	Depending on JP7
HEX2[5]	PIN_W27	Seven Segment Digit 2[5]	Depending on JP7
HEX2[6]	PIN_W28	Seven Segment Digit 2[6]	Depending on JP7
HEX3[0]	PIN_V21	Seven Segment Digit 3[0]	Depending on JP7
HEX3[1]	PIN_U21	Seven Segment Digit 3[1]	Depending on JP7
HEX3[2]	PIN_AB20	Seven Segment Digit 3[2]	Depending on JP6
HEX3[3]	PIN_AA21	Seven Segment Digit 3[3]	Depending on JP6
HEX3[4]	PIN_AD24	Seven Segment Digit 3[4]	Depending on JP6
HEX3[5]	PIN_AF23	Seven Segment Digit 3[5]	Depending on JP6
HEX3[6]	PIN_Y19	Seven Segment Digit 3[6]	Depending on JP6
HEX4[0]	PIN_AB19	Seven Segment Digit 4[0]	Depending on JP6
HEX4[1]	PIN_AA19	Seven Segment Digit 4[1]	Depending on JP6
HEX4[2]	PIN_AG21	Seven Segment Digit 4[2]	Depending on JP6
HEX4[3]	PIN_AH21	Seven Segment Digit 4[3]	Depending on JP6
HEX4[4]	PIN_AE19	Seven Segment Digit 4[4]	Depending on JP6
HEX4[5]	PIN_AF19	Seven Segment Digit 4[5]	Depending on JP6
HEX4[6]	PIN_AE18	Seven Segment Digit 4[6]	Depending on JP6
HEX5[0]	PIN_AD18	Seven Segment Digit 5[0]	Depending on JP6
HEX5[1]	PIN_AC18	Seven Segment Digit 5[1]	Depending on JP6
HEX5[2]	PIN_AB18	Seven Segment Digit 5[2]	Depending on JP6
HEX5[3]	PIN_AH19	Seven Segment Digit 5[3]	Depending on JP6
HEX5[4]	PIN_AG19	Seven Segment Digit 5[4]	Depending on JP6
HEX5[5]	PIN_AF18	Seven Segment Digit 5[5]	Depending on JP6
HEX5[6]	PIN_AH18	Seven Segment Digit 5[6]	Depending on JP6
HEX6[0]	PIN_AA17	Seven Segment Digit 6[0]	Depending on JP6
HEX6[1]	PIN_AB16	Seven Segment Digit 6[1]	Depending on JP6
HEX6[2]	PIN_AA16	Seven Segment Digit 6[2]	Depending on JP6
HEX6[3]	PIN_AB17	Seven Segment Digit 6[3]	Depending on JP6
HEX6[4]	PIN_AB15	Seven Segment Digit 6[4]	Depending on JP6
HEX6[5]	PIN_AA15	Seven Segment Digit 6[5]	Depending on JP6

Figure 2.4: Schematic of 7-segment display

The 27 red LEDs (LEDRs) were also implemented in this project. **Figure 2.5** shows the connection specifications for the LEDRs, these are shown by the red diodes respectively. Each LED is driven by a pin on the DE2 board on **Figure 2.6**

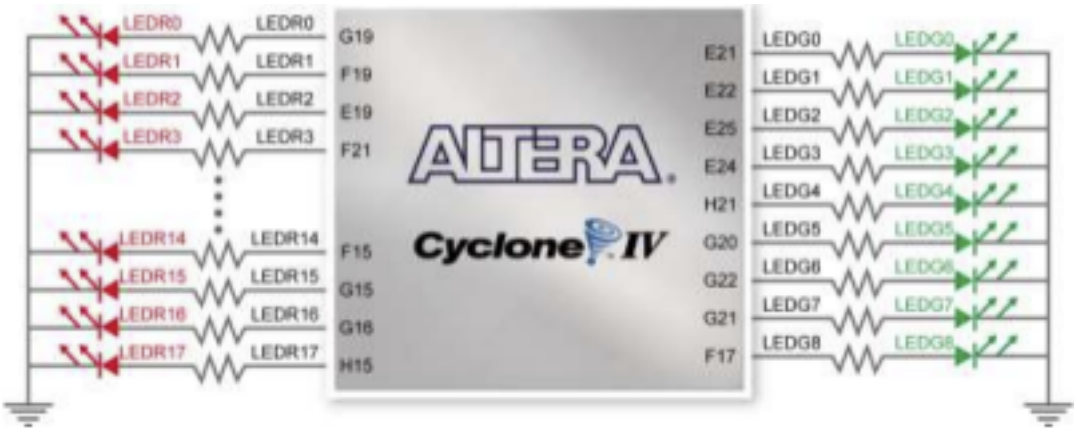

Figure 2.5: LEDRs configuration

Signal Name	FPGA Pin No.	Description	I/O Standard
LEDR[0]	PIN_G19	LED Red[0]	2.5V
LEDR[1]	PIN_F19	LED Red[1]	2.5V
LEDR[2]	PIN_E19	LED Red[2]	2.5V
LEDR[3]	PIN_F21	LED Red[3]	2.5V
LEDR[4]	PIN_F18	LED Red[4]	2.5V
LEDR[5]	PIN_E18	LED Red[5]	2.5V
LEDR[6]	PIN_J19	LED Red[6]	2.5V
LEDR[7]	PIN_H19	LED Red[7]	2.5V
LEDR[8]	PIN_J17	LED Red[8]	2.5V
LEDR[9]	PIN_G17	LED Red[9]	2.5V
LEDR[10]	PIN_J15	LED Red[10]	2.5V
LEDR[11]	PIN_H16	LED Red[11]	2.5V
LEDR[12]	PIN_J16	LED Red[12]	2.5V
LEDR[13]	PIN_H17	LED Red[13]	2.5V

Figure 2.6: Pin locations of LEDRs

The DE2-115 board contains 4 Push-Buttons, KEY0, KEY1, KEY2, KEY3. **Figure 2.7** shows the connections between the Push-Buttons and the FPGA. Pressing a push-button provides a low logic level, and high logic level when not pressed. The keys are debounced using Schmitt Trigger, as seen in **Figure 2.8**. **Figure 2.9** shows the pin configuration for the Push-Buttons.

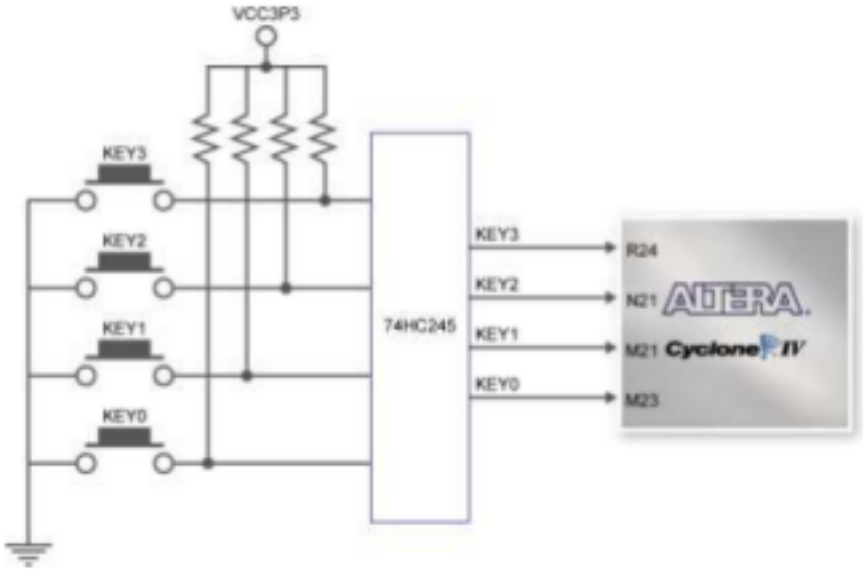

Figure 2.7: Push-Button schematic

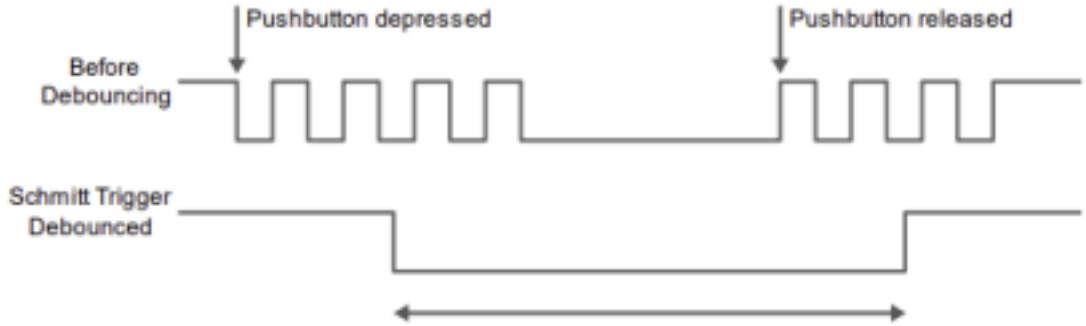

Figure 2.8: Schmitt-Trigger

Signal Name	*FPGA Pin No.*	*Description*	*I/O Standard*
KEY[0]	PIN_M23	Push-button[0]	Depending on JP7
KEY[1]	PIN_M21	Push-button[1]	Depending on JP7
KEY[2]	PIN_N21	Push-button[2]	Depending on JP7
KEY[3]	PIN_R24	Push-button[3]	Depending on JP7

Figure 2.9: Pin configuration for Push-Buttons

development board. To turn on the LEDs a logic level 1 must be written to it, writing a logic level 0 turns off the LEDs. **Figure 2.6** shows the pin configuration for the LEDs.
The DE2-115 Board has two internal timers with a clock of 50 MHz, meaning that it is able to read or execute $5.0 * 10^7$ instructions before a full second will have elapsed or decrement its

counter at such a rate. This is necessary to know when calculating the desired period of the timer and changing it via its low start and high start values.

The RUN bit is set when the timer is found to be running. The TO bit is set when the counter has reached 0 and can be reset by having anything written into the Status register.

In the Control register, the STOP bit can be written to pause the operations of the timer. Writing into the START bit will continue this. The CONT bit determines whether or not the timer will repeat the countdown after the counter will have elapsed in its entirety. The ITO bit determines whether or not the timer will send an interrupt signal when the timer elapses. The register layout of the internal timers is shown below in **Figure 2.10**.

Offset	Name	R/W	Description of Bits						
			15	...	4	3	2	1	0
0	status	RW	*(1)*					RUN	TO
1	control	RW	*(1)*			STOP	START	CONT	ITO
2	periodl	RW	Timeout Period – 1 (bits [15:0])						
3	periodh	RW	Timeout Period – 1 (bits [31:16])						
4	snapl	RW	Counter Snapshot (bits [15:0])						
5	snaph	RW	Counter Snapshot (bits [31:16])						

Figure 2.10: DE2-115 Internal Timer Registers.

The Status register contains two defined bits. Bit 0 is the TO (timeout) bit. This bit is 1 when the internal counter reaches 0. This bit stays at 1 until it is overwritten by a master peripheral. Writing 0 to the status register clears the TO bit. Bit 1 is the RUN bit.The RUN bit is set to 1 when the internal counter is running, otherwise it is set to 0. Writing to the status register does not alter the RUN bit. **Figure 2.11** shows the contents of the Status register.

Bit	Name	R/W/C	Description
0	TO	RC	The TO (timeout) bit is set to 1 when the internal counter reaches zero. Once set by a timeout event, the TO bit stays set until explicitly cleared by a master peripheral. Write zero to the status register to clear the TO bit.
1	RUN	R	The RUN bit reads as 1 when the internal counter is running; otherwise this bit reads as 0. The RUN bit is not changed by a write operation to the status register.

Figure 2.11: Timer Status register

The Control register contains 4 defined bits. Bit 0 is the ITO bit. Writing 1 to this bit enables interrupts to be generated by the timer. Bit 1 is the CONT bit, writing 1 to this bit restarts the timer once it's done, or if the TO bit is 1. Bit 2 is the START bit, writing one to this bit starts

the timer. Bit 3 is the STOP bit, writing one to this bit stops the counter. **Figure 2.12** shows the contents of the Control register.

Bit	Name	R/W/C	Description
0	ITO	RW	If the ITO bit is 1, the interval timer core generates an IRQ when the status register's TO bit is 1. When the ITO bit is 0, the timer does not generate IRQs.
1	CONT	RW	The CONT (continuous) bit determines how the internal counter behaves when it reaches zero. If the CONT bit is 1, the counter runs continuously until it is stopped by the STOP bit. If CONT is 0, the counter stops after it reaches zero. When the counter reaches zero, it reloads with the value stored in the period registers, regardless of the CONT bit.
2	START *(1)*	W	Writing a 1 to the START bit starts the internal counter running (counting down). The START bit is an event bit that enables the counter when a write operation is performed. If the timer is stopped, writing a 1 to the START bit causes the timer to restart counting from the number currently stored in its counter. If the timer is already running, writing a 1 to START has no effect. Writing 0 to the START bit has no effect.

Figure 2.12: Timer Control register

The Period low period period high registers hold the value of the counter. Period low contains the lower 16 bits of the timer interval and Period high contains the upper 16 bits of the timer interval. For example, if a one second timer is to be created using a 50 mHZ clock, the value of 0x2FAF080 must be written to the Period high and Period low registers. The Period high will contain the value of 0x2FA and the Period low will contain the value of 0xF080. The Snap low and Snap high registers provide the value of the counter during runtime, this is done by writing 1 to these registers.

SD card interface

A HAL is provided that supports a FAT16 filesystem, though we found it had trouble with modern cards where the cluster size is large. We had success using an old 512 MB card formatted to FAT16 using Windows. Conversely, using a 2 GB card formatted with Windows used clusters too large for the filesystem, and the HAL complained about an incompatible file system.

The function short alt_up_sd_card_read(short file_handle) only accurately reads ASCII text from files. According to the documentation, the return value is the next character read from the file, or a negative number upon error. However, for characters in the file outside of the range of ASCII text (for example, a binary file containing a byte '0xf1') the byte is read, then cast as a char, but assigned to a short. The result is a sign extension for bytes with the eighth-bit set that makes the short appear negative, and thus looks like a file read error. The fix is to modify the HAL code to not cast bytes read from the card as a char. This can be accomplished in the HAL source file "altera_up_sd_card_avalon_interface.c" by removing the (char) cast on line 1733.

Upon building the syslib project, the NIOS II IDE will generate and compile a "alt_sys_init.c" under "your_project_syslib->Debug->system_description" that initializes all necessary hardware. We found that for this SD core, the appropriate initialization macros did not get called as part of this process, and so it was necessary to manually invoke the macros ALTERA_UP_SD_CARD_AVALON_INTERFACE_INSTANCE(name, device) and ALTERA_UP_SD_CARD_AVALON_INTERFACE_INIT(name, device) that are defined in "altera_up_sd_card_avalon_interface.h" (be sure to #include this header as well). The name is the name of the interface in SOPC, but in all caps (e.g. SD_CARD_INTERFACE), and the device is the same, but in the same casing as SOPC (e.g. sd_card_interface).

If your core is called "sd_card_interface" in SOPC, your top level VHDL mappings in the system port map (with all the appropriate signals and pins declared elsewhere) will look something like the one given below in **Figure 2.13.**

```
b_SD_cmd_to_and_from_the_sd_card_interface      => SD_CMD,
b_SD_dat3_to_and_from_the_sd_card_interface     => SD_DAT3,
b_SD_dat_to_and_from_the_sd_card_interface      => SD_DAT,
o_SD_clock_from_the_sd_card_interface           => SD_CLK
```

Figure 2.13: SD card interface

Audio CODEC

The DE2-115 Board has an audio encoder/decoder (CODEC) that provides an interface for audio input and output. By default, the sample rate of the audio CODEC is 8000 samples/sec. Data is transmitted serially between the board and the CODEC, but data written to or read from the audio core is transmitted in parallel. The audio core also contains four FIFOs to buffer the input and output data, both of which have left and right audio channels. An example configuration of the DE2-115 Board's audio registers' layout is shown below in **Figure 2.14**.

Address	31 . . . 24	23 . . . 16	15 . . . 10	9	8	7 . . . 4	3	2	1	0	
0xFF203040	Unused			WI	RI		CW	CR	WE	RE	Control
0xFF203044	WSLC	WSRC	RALC			RARC					Fifospace
0xFF203048	Left data										Leftdata
0xFF20304C	Right data										Rightdata

Figure 2.14: DE2-115 Board Audio CODEC Default Register Configuration.

The Control Register, offset 0, is mainly used for generating interrupts and is irrelevant to the current experiment. The Fifospace Register, offset 4, is read only and contains the number of bytes currently stored in the left and right audio-input FIFOs (via RALC and RARC respectively); WSLC and WSRC contain the number of bytes for the respective audio-output FIFOs.

To functionally operate the Audio CODEC, data automatically read from the red Microphone line, visible in the top-left corner of the board in **Figure 2.0**, must be written into both the Left Data Register and Right Data Register; this loads the data into the Write FIFOs. Then, provided that the Fifospace Register indicates that there is still data available in the FIFOs, the Left Data Register and Rightdata Register must both be read; this takes the data from the head of the FIFOs. Once this is done, data is transmitted to the green Audio Out line.

3. EXPERIMENTAL PROCEDURE

3.1 Equipment Used

- Quartus Prime 16.1 Lite Edition
    - Qsys
    - Nios II 16.1 Software Build Tools for Eclipse
- DE2-115 Board
    - Included USB Blaster 2.0 A (Male) to B (Male)
    - Included AC Power Adapter
- 2GB SD card
- A Personal Computer with required specifications to run the system

3.2 Project Procedure Description

3.2.1 Creating a new system for our project in Quartus and Qsys

a. First, we created a new project in Quartus
b. Called this project “project_top”
c. Then we created a new HDL file inside the project file to be synthesized later
d. Then we created a new Qsys system with the required PIOs and hardware components. These components are:
    i. Nios 2 processor
    ii. On chip memory
    iii. SDRAM
    iv. PIO: LEDr
    v. PIO: KEYs

vi. PIO: 7-segment display
vii. Audio CODEC
viii. SD card interface
ix. JTAG_UART
x. Interval Timer
xi. System ID

e. Then we connected the corresponding pins within the system to get it working
f. Then we enable Key's ,and Time's, and UART interrupt handling routine
g. Then we assigned their base addresses and integrated their corresponding interrupt bit number
h. Finally, we compiled everything and got the .qip file ready to be used in our Quartus top module
i. And lastly, we wrote the Verilog connectivity file in our top module and synthesized everything to create our working system

3.2.2 Creating a new project in Eclipse Nios 2 platform

2. We first created a new project with BSP folder in eclipse
3. We assigned a name
4. We selected the corresponding .sopcinfo file that Quartus generated for us
5. We selected the Hello_World template to start with
6. Starting fresh, we deleted everything in the Hello_World.c file and started coding for our project in C language.

3.2.3 Getting started with coding the project in C language

7. We first start with adding required #include files for our project. These are:
    a. #include <stdio.h>
    b. #include "system.h"
    c. #include "altera_avalon_pio_regs.h"
    d. #include "altera_avalon_timer.h"
    e. #include "altera_avalon_timer_regs.h"
    f. Then we start initializing necessary variables for our project. These variables are used throughout the project for keys, hexes, timers, sd card, and audio interface.

3.2.4 Creating a lookup table for Hex display

8. We created a look up table for hex display
9. We bitmapped the corresponding decimal and used a offset to display the number
10. Our Minutes goes up to decimal 5, and our seconds goes up to 60.
11. We used two variables to be allocated for our array.

3.2.5 Getting our Interval Timer initialized

12. We used interval timer to keep track of delays and operations for this project
13. To initialize, we set the corresponding period into its corresponding register
14. We first needed to calculate the period in order to have a functioning 1 second timer.
15. We set up the timer as it was described in the background section of this report.

3.2.6 Coding the Timer Progress bar

16. We used a while loop in order to calculate the period of increment on our LEDr.
17. We converted the total time into calculated partitions to be incremented every certain seconds to match the total time of the song
18. This will also be configured with the Hex incrementing timer.

3.2.7 Coding the Hex Timer display

19. We used a if statement to increment the song runtime with the similar calculations we made for our progress bar.
20. We used our interval timer to increment the hex display by 1 every 1 second up until it equals the total time of the song.
21. This will be synchronized with the progress bar and work together when the song starts.

3.2.8 Key interrupt

22. We used push button keys polling interrupt method to do several functions with it
23. KEY3 will be responsible for resetting our progress bar and hex display and playing the previous song stored in our memory.
24. KEY2 will be responsible for Pause and Play operations which will also trigger our hex display and progress bar.
25. KEY1 will be responsible for the Next song function as well as resetting the hex display and progress bar.
26. KEY0 is just a reset button that restarts the device.

3.2.9 SD card interface

27. SD card interface initialization by following the directions provided under Background information
28. The SD card will contain 3 .wav music files that will be imported to our main SDRAM memory of our FPGA device.
29. SD card files will also include some header files and information about the songs that will be displayed at our JTAG_UART terminal.

3.2.10 Audio CODEC initialization

30. Audio CODEC must be accessed.

31. Audio must be written into the Left Data and Right data registers from the Microphone line.
32. Audio must be read from the Left Data and Right Data registers into the Audio Out line.
33. The speakers will output the data from the Audio Out line.

3.3 Procedure Execution

1. Creating a new system for our project in Quartus and Qsys

First we need to create a new project wizard from files>new project wizard.

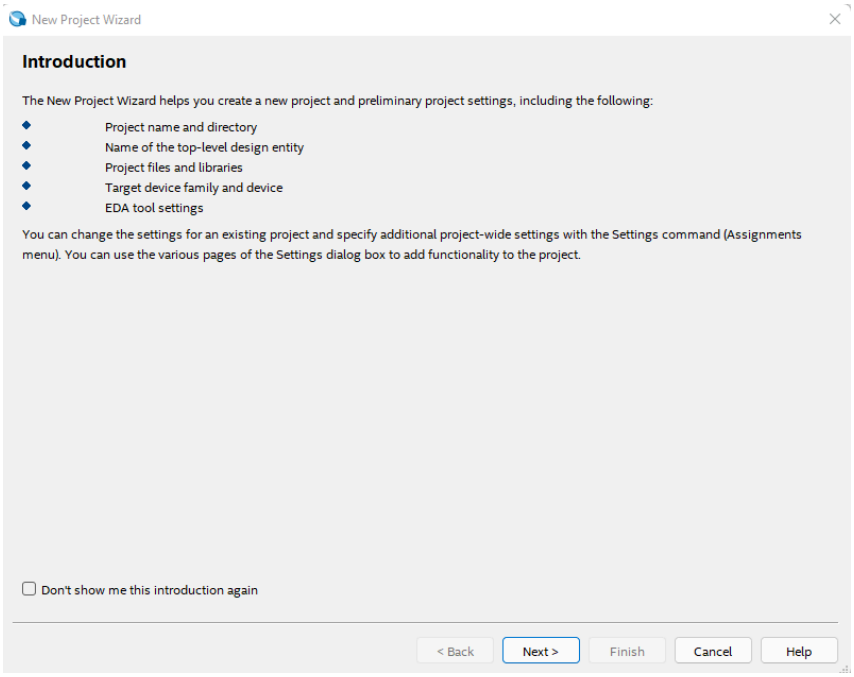

Figure 3.0: New project wizard

We then assign a project directory, name, and name of the top level design entity for this project.

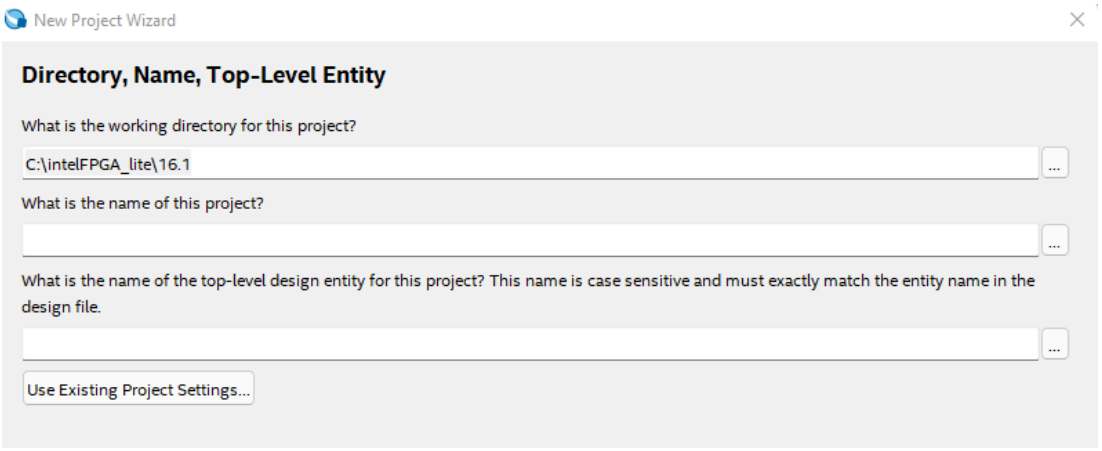

Figure 3.1: Assigning names

Then there will be a window that will ask us several questions about the device we are using. We are going to select the Cyclone IV E series and using the name filter, we will identify the corresponding device we have, Which will be EP4CE115F29.

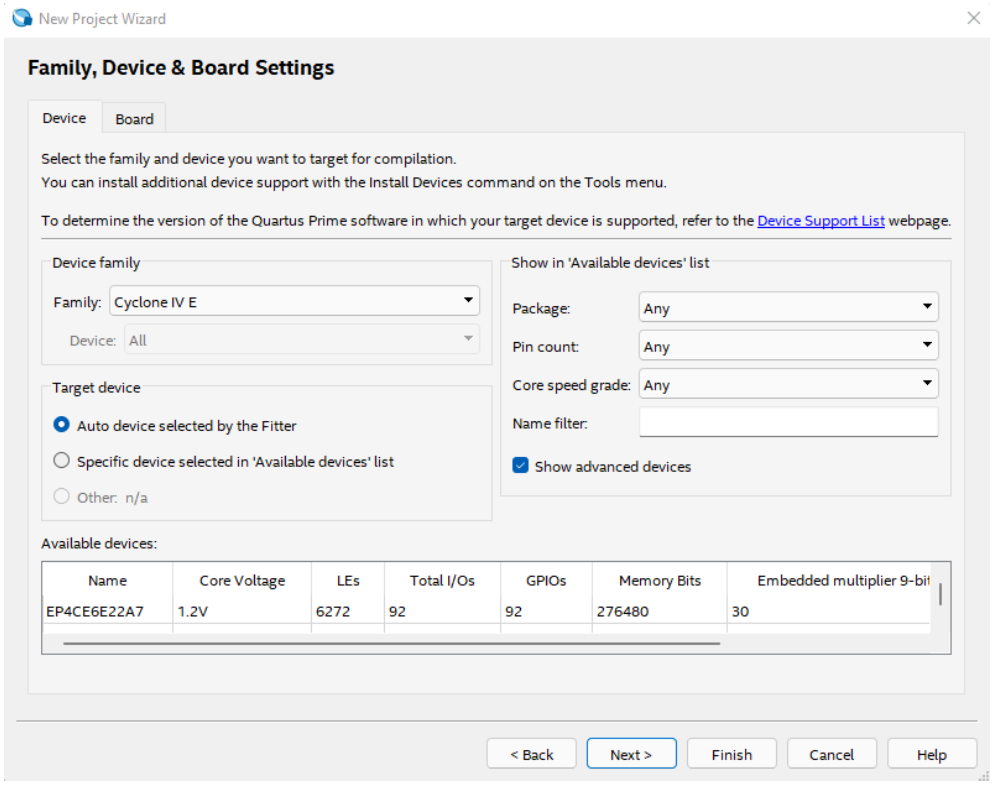

Figure 3.2: Registering device

We will then stop here, and open Qsys from the Tools bar on the top. Wait for Qsys to open.

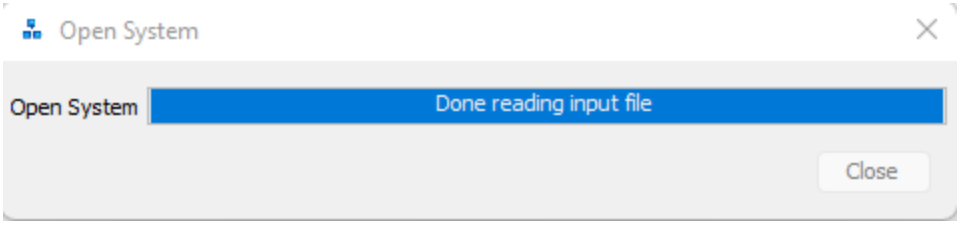

Figure 3.3: Waiting for Qsys

We now will start adding the peripherals we need for this system. We will start with Nios 2 processor. We will use the Nios 2/f model for this project.

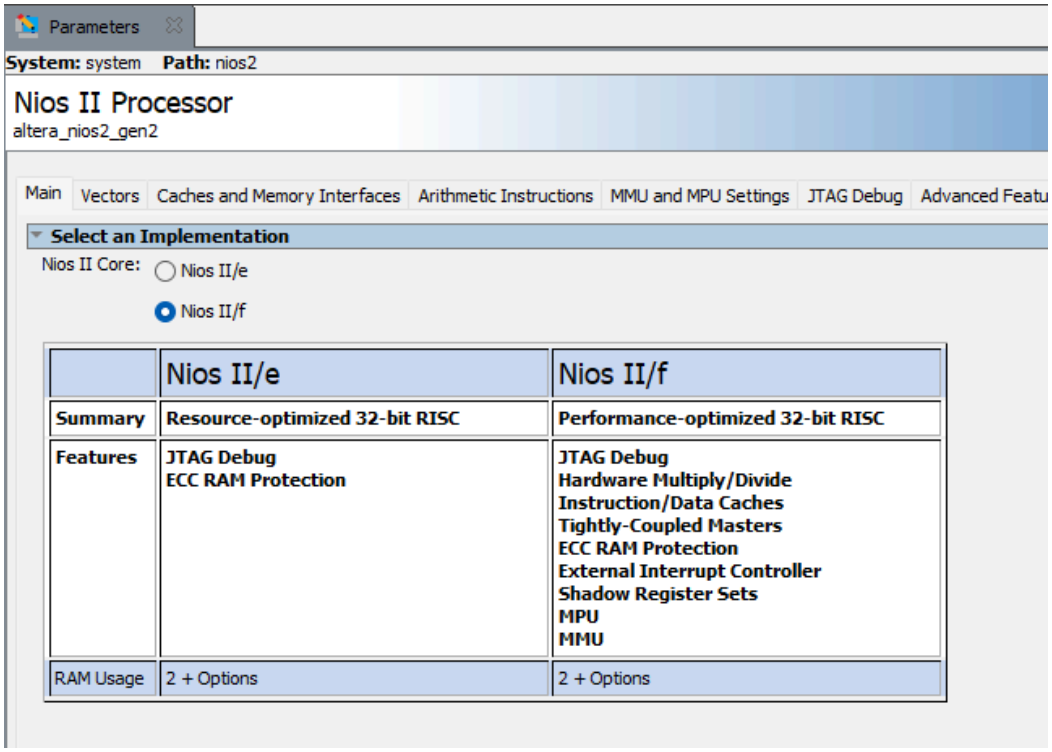

Figure 3.4: Selecting Nios 2/f model

We will now select our on chip memory. It will be called RAM. The only thing we change here will be the total memory size to 8192 bytes.

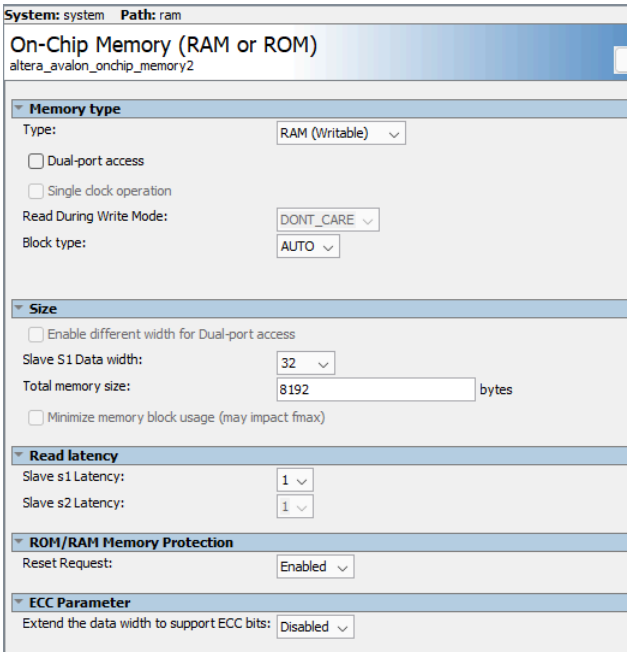

Figure 3.5: Selecting RAM

We will now select our SDRAM and configure its timing so that it will work with our board. These settings must be matching to the image provided below on **Figure 3.6**

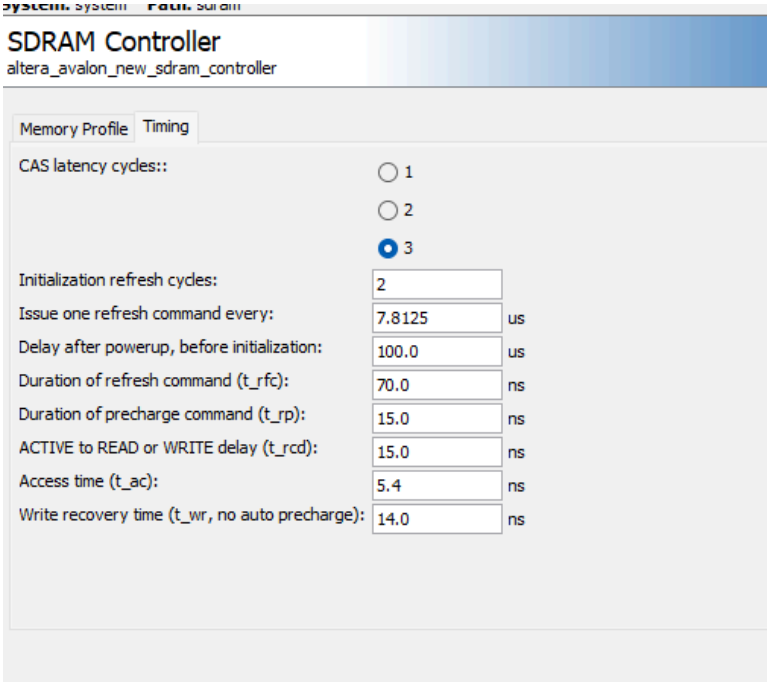

Figure 3.6: Selecting SDRAM

Now we will select our interval timer device that will handle all the timings in our project.

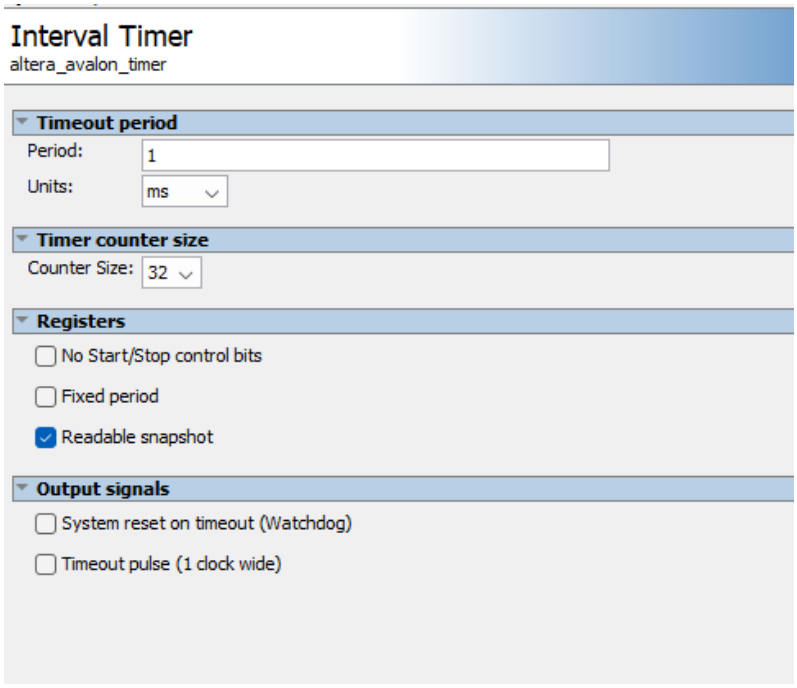

Figure 3.7: Selecting Interval Timer

We will now move onto the part that will select different kinds of PIOs. Our first PIO will be for LED red. In this configuration we will set the width to 18 bits since there are 18 leds and select it to be Output.

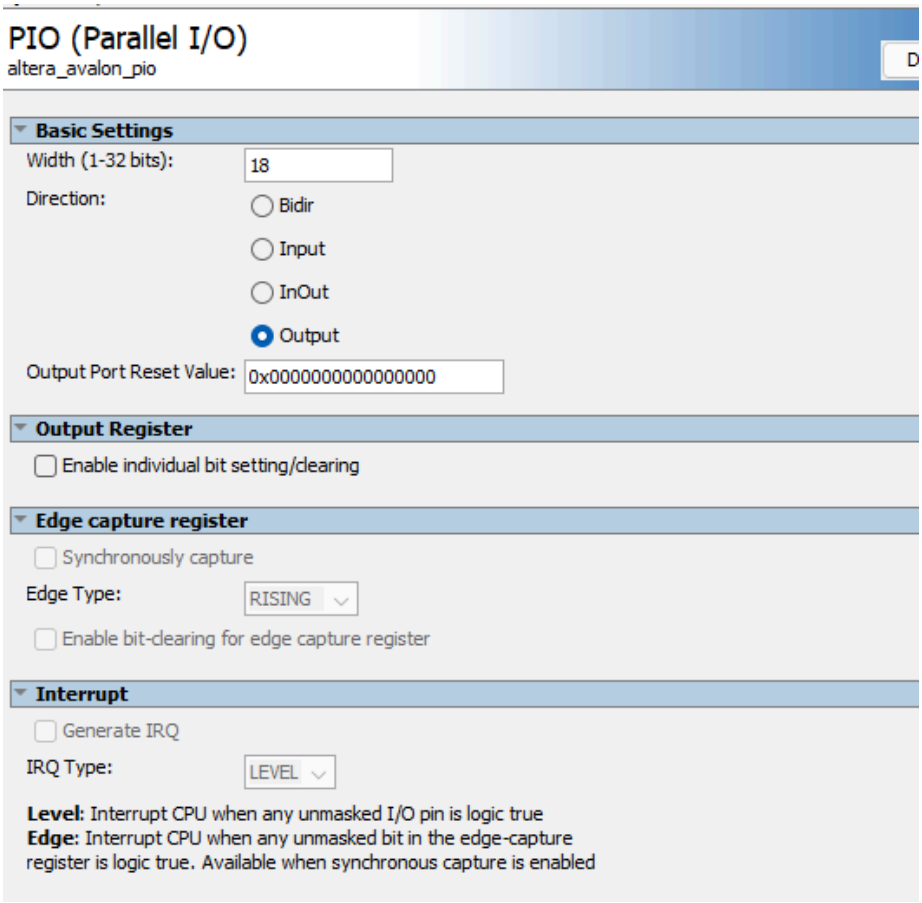

Figure 3.8: Selecting LEDr PIO

We will now select our KEYs PIO. We will change the width to 4 bit and select Input direction. We will also enable interrupt by Generate IRQ and select level. We will also need to select a synchronously captured Edge capture register and make it a falling edge.

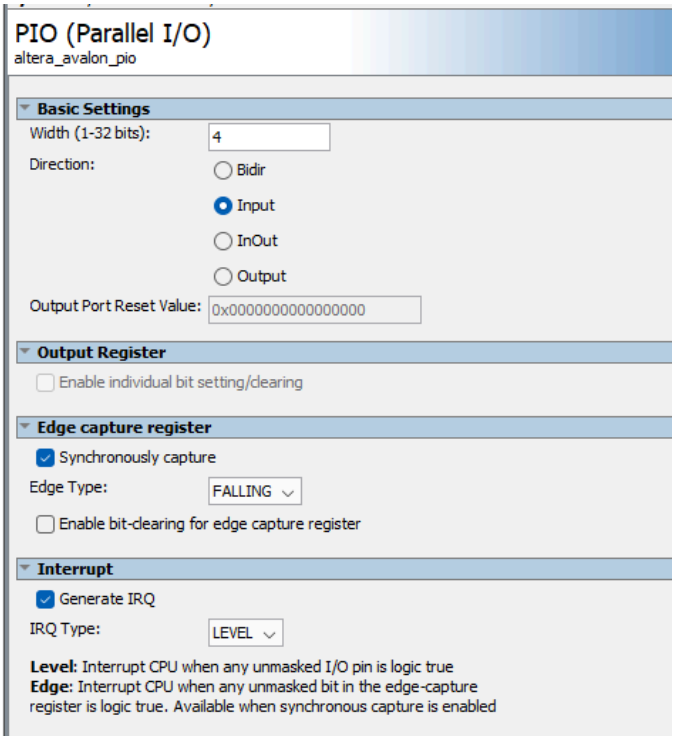

Figure 3.9: Selecting KEYs PIO

We will now select our Hex display Pio for HEX display given for HEX0 to HEX3. We need to select a 32 bit width and make it an output. We will add another HEX display PIO for HEX 4-7 and do the same configuration we did for the first one.

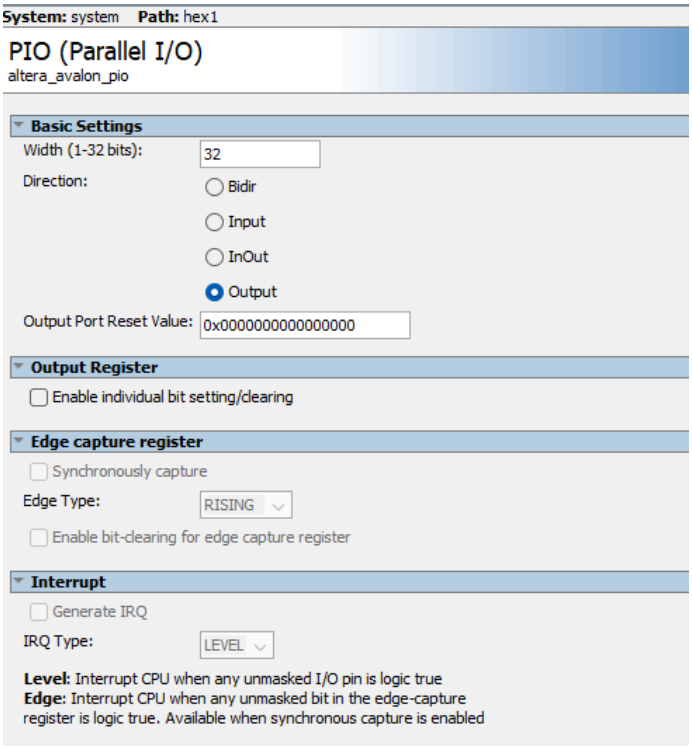

Figure 3.10: Selecting HEX display PIO

We will now select our SD card interface. We don't need to do any changes to its configuration and settings.

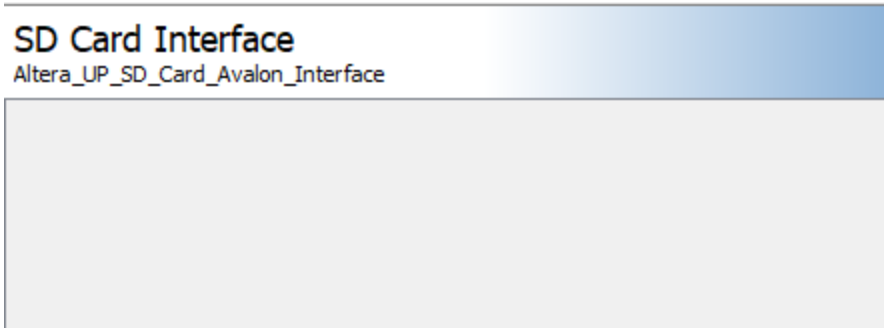

Figure 3.11: Selecting SD card interface

We will need to select the Audio CODEC interface to output music from our device. We don't need to change and play with its settings.

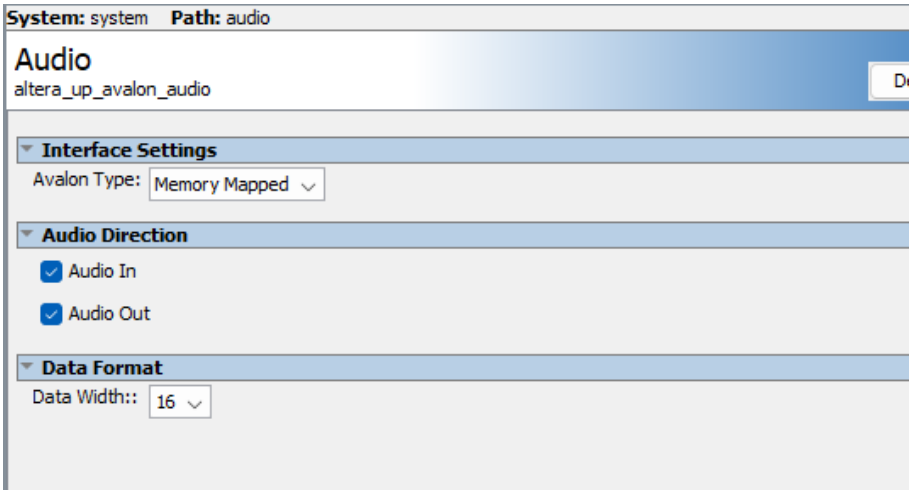

Figure 3.12: Selecting Audio CODEC

Finally we will add a system_id peripheral to be used during our configuration and identification and JTAG_UART system.

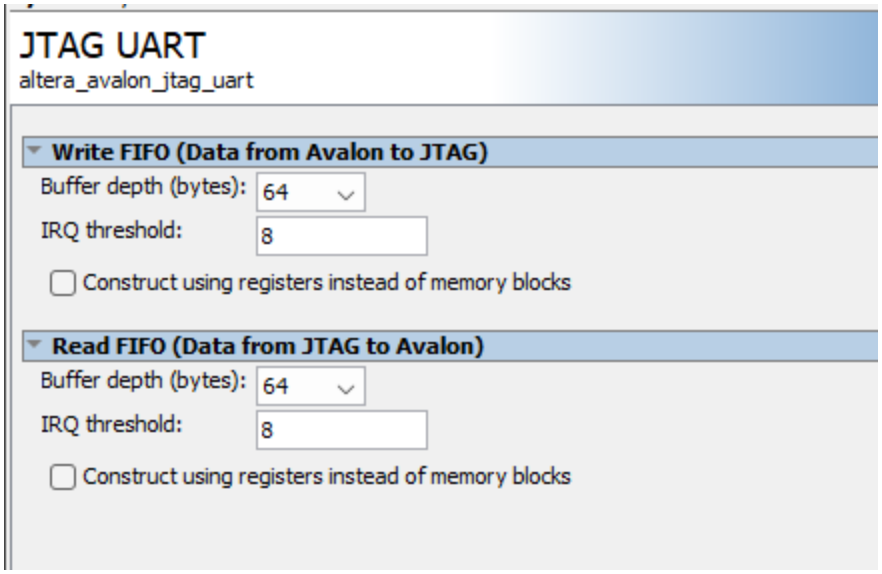

Figure 3.13: Selecting JTAG

Once we are done adding these devices into our system. We now need to generate addressing assignments for every device we added. This will be automatically assigned to each device once we select Automatic assignment. After that we need to do correct wiring and connections so everything is working properly. Some important things to be connected are **reset** and **clk** wires.

The figure below shows all the connections made for our system to work properly.

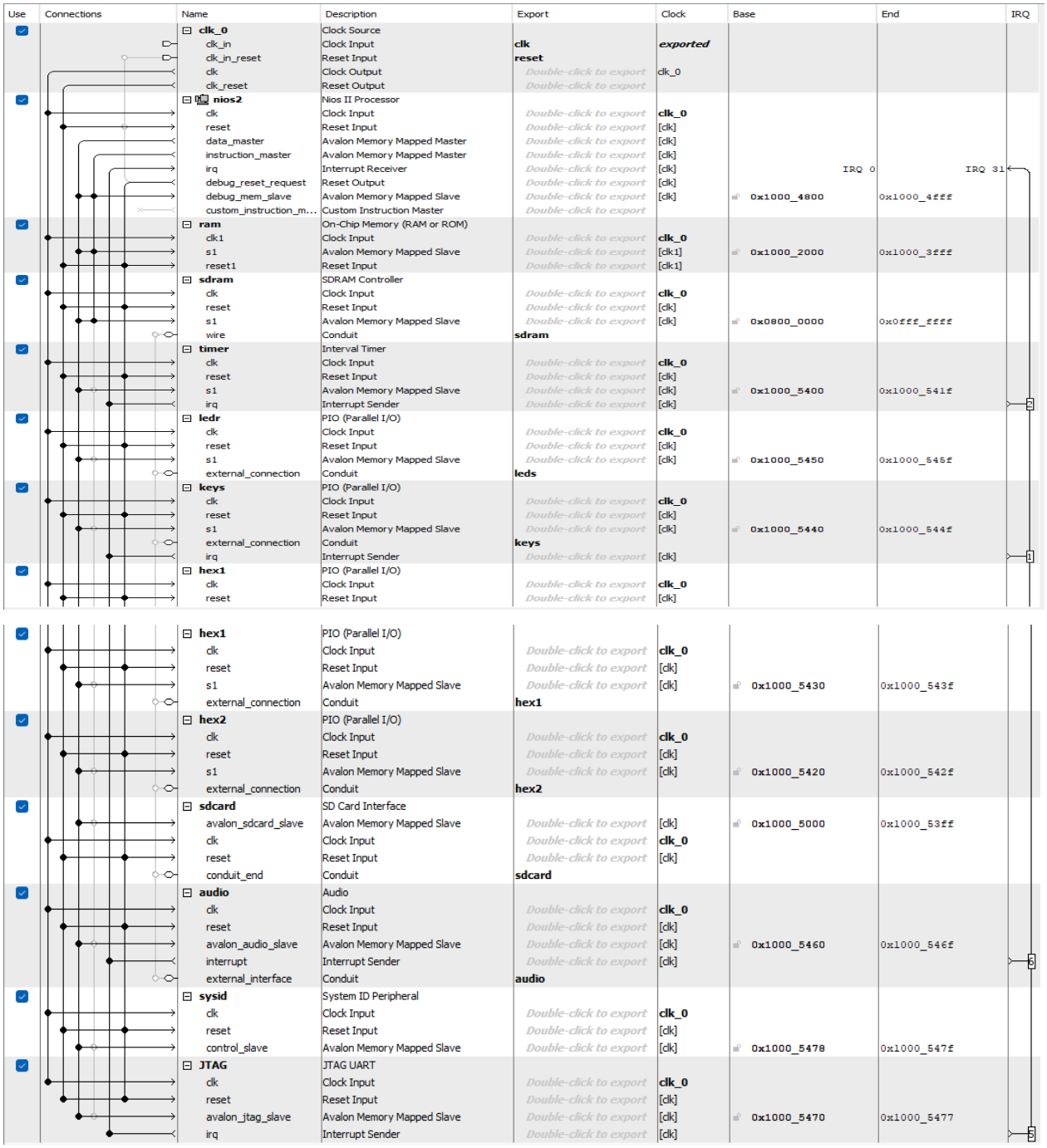

Figure 3.14: Entire system connections

Once done, we will generate HDL and let it finish and close Qsys.

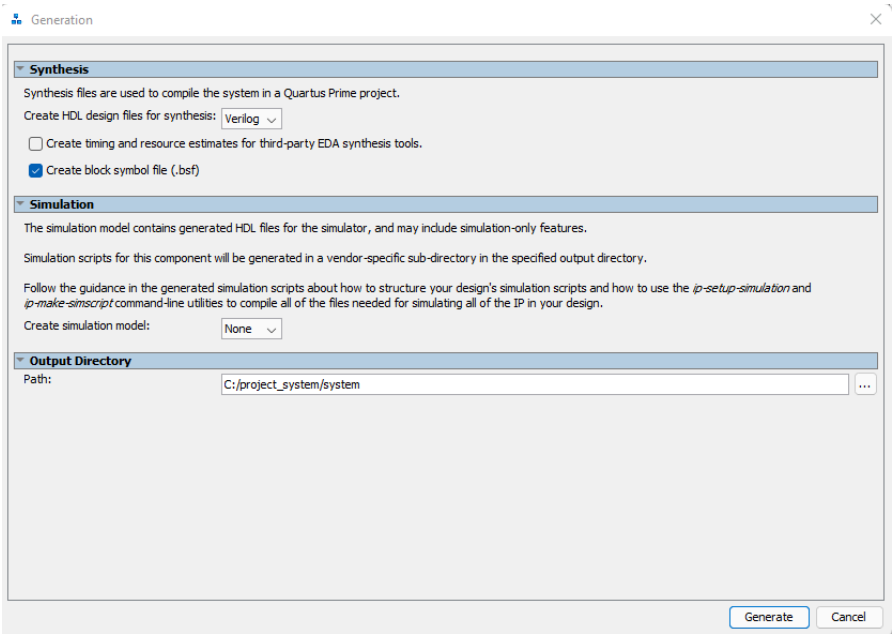

Figure 3.15: Generate HDL

Once we get the generation complete message from Qsys, we will close this window and move back to Quartus to finish the rest of our configuration.

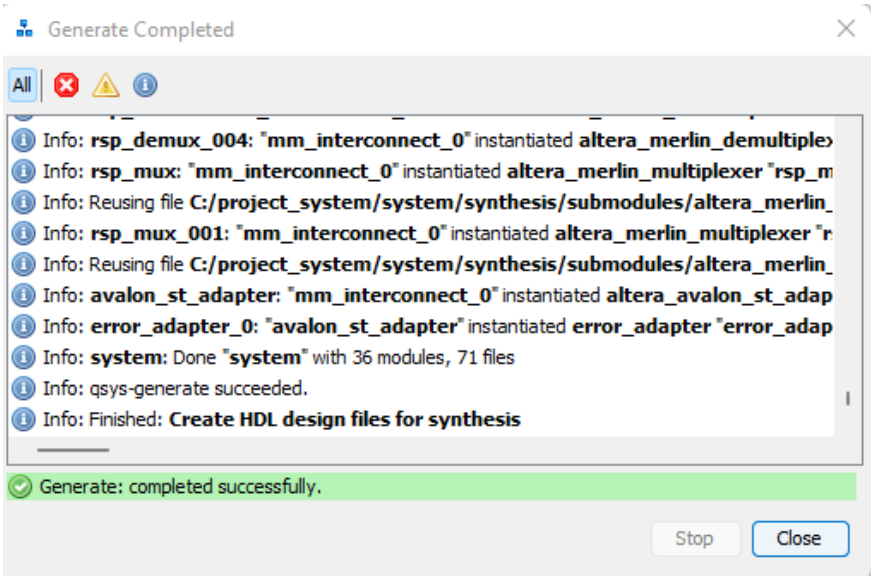

Figure 3.16: Generate complete

On Quartus we first need to add the Qsys file we generated to our project files. For that we will go to Project > add files > select .qip file generated and add. We then will create an VHDL file to write our Verilog code for our connectivity and configuration. The following input and output connections must be made for our system to work.

```
module topModule(SD_CMD, SD_DAT, SD_DAT3, SD_CLK, AUDIO_DACRLCK, AUDIO_DAC, AU
input [3:0] KEY;
input CLOCK_50;
output [31:0] HEX3_0;
output [31:0] HEX4_7;
output [17:0] LEDR;
output [12:0] DRAM_ADDR;
output [1:0] DRAM_BA;
output DRAM_CAS_N,DRAM_RAS_N,DRAM_CLK;
output DRAM_CKE,DRAM_CS_N,DRAM_WE_N;
output [3:0] DRAM_DQM;
inout  [31:0] DRAM_DQ;
input AUDIO_ADC;
input AUDIO_ADCLRCK;
input AUDIO_BCLK;
output AUDIO_DAC;
input AUDIO_DACRLCK;
output SD_CLK;
inout SD_DAT3;
inout SD_DAT;
inout SD_CMD;



system nios2(
.clk_clk (CLOCK_50),
.reset_reset_n(KEY[0]),
.keys_export(KEY),
.leds_export(LEDR),
.sdram_addr(DRAM_ADDR),
.sdram_ba(DRAM_BA),
.sdram_cas_n(DRAM_CAS_N),
.sdram_cke(DRAM_CKE),
.sdram_cs_n(DRAM_CS_N),
.sdram_dq(DRAM_DQ),
.sdram_dqm(DRAM_DQM),
.sdram_ras_n(DRAM_RAS_N),
.sdram_we_n(DRAM_WE_N),
.hex1_export(HEX3_0),
.hex2_export(HEX4_7),
.audio_ADCDAT(AUDIO_ADC),
.audio_ADCLRCK(AUDIO_ADCLRCK),
.audio_BCLK(AUDIO_BCLK),
.audio_DACDAT(AUDIO_DAC),
.audio_DACLRCK(AUDIO_DACRLCK),
.sdcard_b_SD_cmd(SD_CMD),
.sdcard_b_SD_dat(SD_DAT),
.sdcard_b_SD_dat3(SD_DAT3),
.sdcard_o_SD_clock(SD_CLK)

);
assign DRAM_CLK = CLOCK_50;
endmodule
```

Figure 3.17: Quartus Top module

Once finished we will generate synthesis and wait until it finishes. If problems occur, we will go back and fix them until synthesis is completed properly.

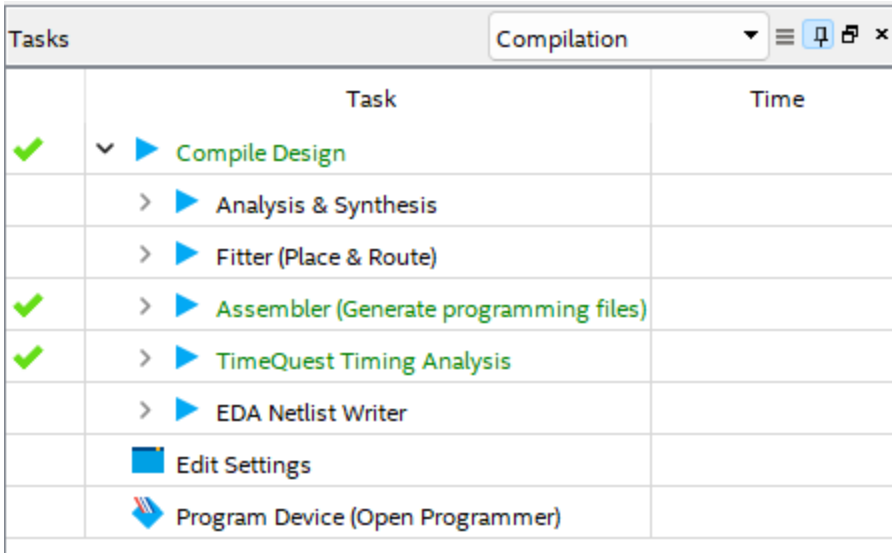

Figure 3.18: Quartus synthesis complete

Now, we must have a project file that will contain .sopcinfo file and .sop file to be used for our Eclipse IDE. We will open Eclipse IDE under Tools in Quartus and start a new project inside a fresh new folder.

Quartus Prime was launched, and, in the Tools bar, the Nios II Software Build Tools for Eclipse was selected. A folder that included a previous lab's corrected Nios II system was created and named "Project_system", and that folder was selected to be the workspace for the Eclipse software.

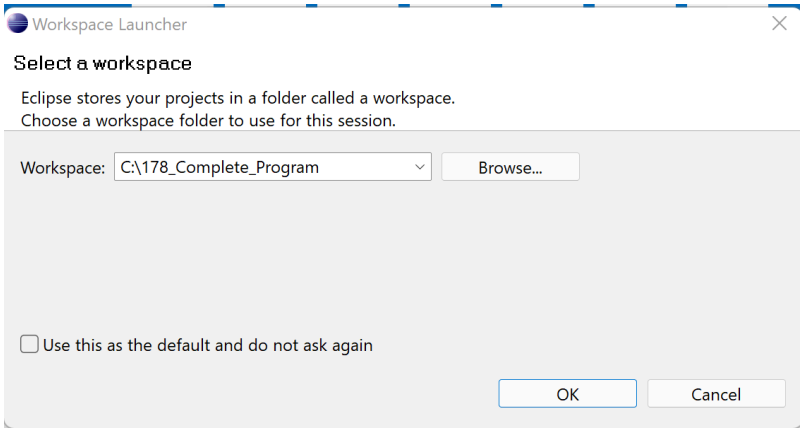

Figure 3.19: Eclipse IDE opened

In the Nios II bar of the Eclipse menu, the Quartus Prime Programmer was selected. The Board was connected to the host computer, and the Hardware Detection button was pressed, as shown in **Figure 3.1**. After the Board was set as the hardware, the "Add File…" button was pressed, and the previous lab's time-limited .sof file was added to the setup. Once the EP4CE115F29 chip was confirmed to be in the setup, it was selected, and its "Program/Configure" checkbox was checked. This setup, as visible in **Figure 3.2**, was saved as "Chain1.cdf". The chip was then selected again, and the Start button was pressed.

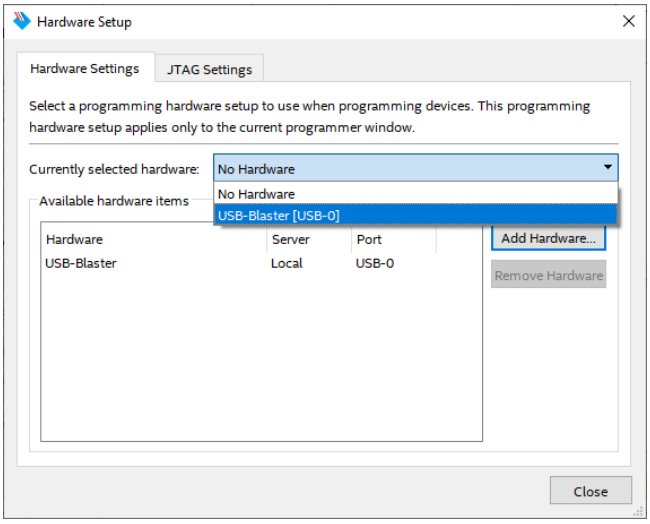

Figure 3.20: Hardware setup for Eclipse IDE

Figure : Hardware Setup menu of the Quartus Prime Programmer. The DE2-115 Board is connected to the host computer, hence the USB-Blaster [USB-0] option.

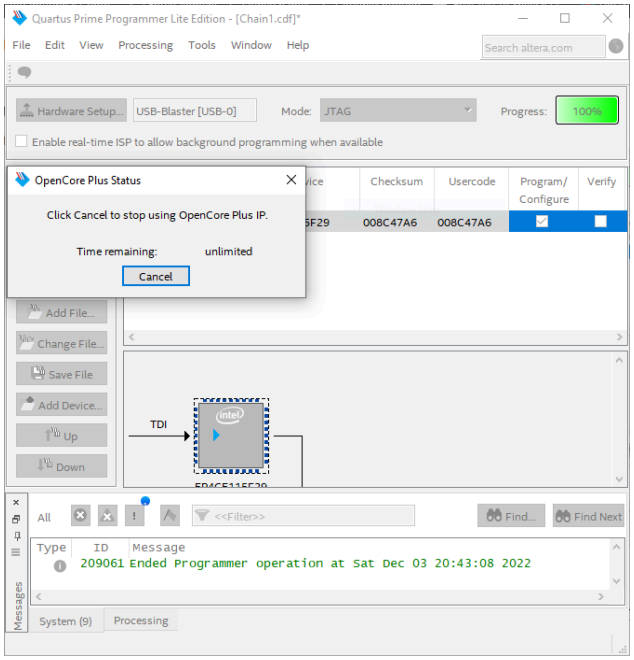

Figure 3.21 : Chain1.cdf setup with the EP4CE115F29 chip used for the DE2-115 Board.

Back in the Eclipse menu, the File tab was pressed, then, in New, a new Nios II Application and BSP from Template was selected. The aforementioned corrected Nios II system's .sopc file was selected as the project's system, and the project was given the name "Project" using a "Hello World" template as a program base, as shown in **Figure 3.21**.

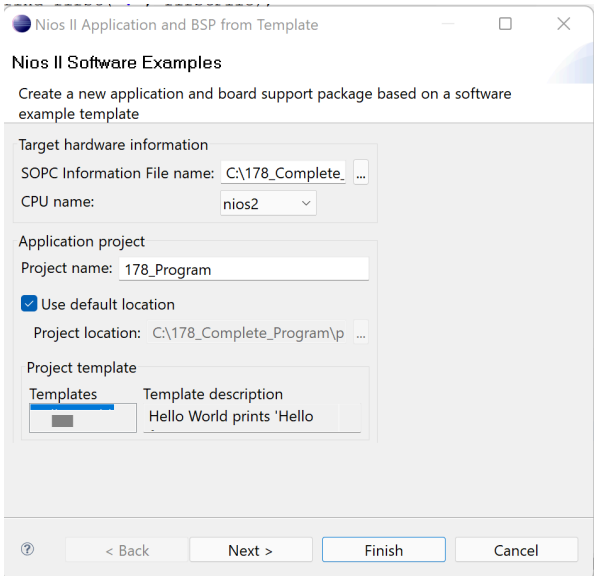

Figure 3.22: Project Application and BSP from Template Wizard Menu.

The project folder Project was right-clicked, and the project was built then subsequently run as a Nios II Hardware in order to verify its functionality. The existing C code in the Hello World.c file, save for the include statements, was erased. In the Project BSP folder, the “system.h” and “altera_avalon_pio_regs.h” header files were included in the C program. Additionally, as shown in **Figure 3.22**, several more header files called “alt_irq.h”, “altera_avalon_timer_regs.h”, “altera_avalon_timer.h”, and “altera_up_sd_card_avalon_interface.h” were found necessary to include in order to facilitate the desired program.

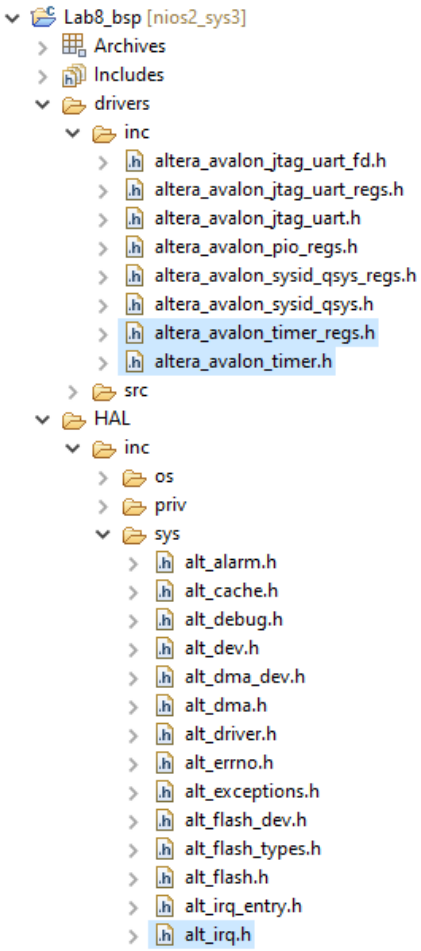

Figure 3.23: The necessarily included header file to register an Interrupt Handler found in the project BSP folder.

We first start with initializing your header files needed for this project. We will need the "system.h" and "altera_avalon_pio_regs.h" header files to be included in the C program. Additionally, as shown in **Figure 3.22**, several more header files called "alt_irq.h", "altera_avalon_timer_regs.h", "altera_avalon_timer.h", and "altera_up_sd_card_avalon_interface.h" were found necessary to include in order to facilitate the desired program.

hello_world.c | alt_main.c

```
#include <stdio.h>
#include "system.h"
#include "altera_avalon_pio_regs.h"
#include "altera_avalon_timer.h"
#include "altera_avalon_timer_regs.h"
#include <altera_up_sd_card_avalon_interface.h>

// Necessary variables //
int songMins[] = {1, 0, 2, 3}; // Change this and the one below for testing
int songSecs[] = {18, 36, 23, 2};
int tracknumber = 0;
int timertrack = 0;
int totalSecs;
int partitions;
int scrub = 0;
int progress = 0;

// Pushbutton variables //
volatile int edge_capture;
int paused = 0;
```

Figure 3.24: Initializing our header files and variables

Our variables are initialized as global type variables which will be used throughout the project. We have SongMins [] array to initialize the minutes of our song for test purposes, and we have SongSecs[] array to initialize the seconds of our song for test purposes. We have a track number variable for tracking the minutes we are currently playing. We have a total secs variable to calculate the partition we need for our LED red bar display as well as partitions variable for calculating progress bar as well.

Volatile int Edge_capture variable will be used for Key's interrupt method to capture the falling edge of the pressed button that will do a desired function in our system.
Int paused is used to detect if a pause has occurred during our Key's interrupt method.

```
int main()
{
    volatile int *edge_capture_ptr = (volatile int*) edge_capture;

    // Song Run Time Variables //
    int min[]={2139062080,2139062137,2139062052,2139062064,2139062041,2139062034}; //minutes 0-5

    int sec[]={1077968767,1081704319,1076133759,1076920191,1075412863,1074954111,1073905535,
            1081638783,1073774463,1074823039,2034270079,2038005631,2032435071,2033221503,
            2031714175,2031255423,2030206847,2037940095,2030075775,2031124351,608206719,
            611942271,606371711,607158143,605650815, 605192063,604143487,611876735,604012415,
            605060991,809533311,813268863,807698303,808484735,806977407,806518655,805470079,
            813203327,805339007,806387583,423657343,427392895,421822335,422608767,421101439,
            420642687,419594111,427327359,419463039,420511615,306216831,309952383,304381823,
            305168255,303660927,303202175,302153599,309886847,302022527,303071103,75530111}; //seconds 00-60
    int a = 0;
    int b = 0;

    // Shut off 7 segment displays to start //
    IOWR_ALTERA_AVALON_PIO_DATA(HEX1_BASE, 2139062143);
    IOWR_ALTERA_AVALON_PIO_DATA(HEX2_BASE, 2139062143);
```

Figure 3.25: Initializing our arrays

We initialized our arrays in **Figure 3.25.** In this array we mapped the corresponding decimal number to be displayed on our HEX display. We have two arrays, one is corresponding to our seconds, which is from 0 to 60. Two, is corresponding to our Minutes, which is 0 to 5.

```
// Shut off 7 segment displays to start //
IOWR_ALTERA_AVALON_PIO_DATA(HEX1_BASE, 2139062143);
IOWR_ALTERA_AVALON_PIO_DATA(HEX2_BASE, 2139062143);


// Timer configuration //
IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_BASE, 0b1000); // Initial stop
IOWR_ALTERA_AVALON_TIMER_PERIODH(TIMER_BASE, 0x02FA); // Top half of 50,000,000
IOWR_ALTERA_AVALON_TIMER_PERIODL(TIMER_BASE, 0xF080); // Bottom half of 50,000,000
IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_BASE, 0b0110); // Start, Continuous


// May want to export the following code to a function //
```

Figure 3.26: Timer configuration

We configured our Timer at the given code above, we first initialized and calculated our period. We load our period to the top half and bottom half of our system timer and start the interval timer by loading in 0b0110 to its control register.

```
// Loop to continuously check if a partition is passed //
int TOBit;
while(1) // Change this condition later to something while unpaused
{
    totalSecs = (songMins[tracknumber] * 60) + songSecs[tracknumber];
    partitions = totalSecs / 18;
    *edge_capture_ptr = IORD_ALTERA_AVALON_PIO_EDGE_CAP(KEYS_BASE) & 0b1111;
    if (*edge_capture_ptr == 0b0000) // Default; assuming a song is playing
    {
        TOBit = IORD_ALTERA_AVALON_TIMER_STATUS(TIMER_BASE) & 0b0001;
        if (TOBit == 0b0001) // Meaning a whole second has passed
        {
```

Figure 3.27: Initialize while loop

In this part of our code at figure **Figure 3.27** we will create a loop to continuously check if a partition is passed. We will start off with a while(1) and code it with if statements that will check if a Key is pressed or not. If the key is not pressed, we will read Interval Timer's status register and do the required work with another if statement.

```
if (TOBit == 0b0001) // Meaning a whole second has passed
{

        scrub++;
        if (scrub < totalSecs) // As long as the timer is still within the song playing
        {
                if (scrub >= partitions) // If a threshold has passed and an LED needs to light up.
                {
                                scrub = 0; // Resets scrub so as to keep partitions static
                                progress = (progress * 2) + 1; // Shifts left, then keeps the previous LEDs lit
                                IOWR_ALTERA_AVALON_PIO_DATA(LEDR_BASE, progress); // Updates LEDs
                }
                //IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER_BASE, 0b00); // Resets TO bit to continue operations; IMPORTANT
        }
        else // When a song has completely elapsed
        {
                break; // Exit loop, change later
                // Will probably want to reset all the variables used after the song is finished
        }
```

Figure 3.28: Progress bar code

In this part of the code at **Figure 3.28** we will use a variable named scrub to keep track of shifting our leds. We will calculate the total time of the progress and shift it to the left as long as our condition is satisfied.

```
// Song runtime code //
if (sec[a] == 75530111) // If sec=60 increase minutes and reset seconds
{
            a=0; //reset second mask counter
            b++; //increase minute mask counter
            timertrack--;
            IOWR_ALTERA_AVALON_PIO_DATA(HEX1_BASE, sec[a]);
            IOWR_ALTERA_AVALON_PIO_DATA(HEX2_BASE, min[b]);
}
else if(timertrack > totalSecs)
{
IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_BASE, 0x8);
}
else //seconds is not yet equal to 60, continue counting
{
            IOWR_ALTERA_AVALON_PIO_DATA(HEX1_BASE, sec[a]);
            IOWR_ALTERA_AVALON_PIO_DATA(HEX2_BASE, min[b]);
            a++; //increase second mask counter
}
// end of Song runtime code //


IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER_BASE, 0b00); // Resets TO bit to continue operations; IMPORTANT
timertrack++;
```

Figure 3.29: Progress bar code

In **Figure 3.29** we will initialize the code required to start the HEX display timer. We will use IOWR functions to write the corresponding decimal to our HEX display along with a second period timer. This will be synchronized along with the Progress bar.

```
else if (*edge_capture_ptr == 0b0100) // Assuming the Play/Pause button is KEY2
    // Add Play functionality later
{
    if (paused == 0) // If not paused, pauses
    {
            paused = 1;

            // Stops Timer, thereby stopping normal operations but not the PC from checking Pushbuttons again
            IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_BASE, 0b1000);
            IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER_BASE, 0b00);

            // !!! Insert whatever other code needs to execute and pause here !!! //
            printf("Song PAUSED \n");
            printf("------------------------------ \n");
            IOWR_ALTERA_AVALON_PIO_EDGE_CAP(KEYS_BASE, 0x00); // Resets the Pushbutton context; IMPORTANT
    }
    else // If paused, plays
    {
            paused = 0;

            IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_BASE, 0b0110);

            // !!! Insert whatever other code needs to execute and play here !!! //
            printf("Song RESUMED \n");
            printf("------------------------------ \n");
            IOWR_ALTERA_AVALON_PIO_EDGE_CAP(KEYS_BASE, 0x00); // Resets the Pushbutton context; IMPORTANT
    }
}
```

Figure 3.30: Key 2 initialized

We used KEY2 to do the following functions coded in **Figure 3.30.** We first start off by stopping the timer, thereby stopping normal operations but not the PC from checking Push Buttons again. We then printf to the console that we are either paused or resumed the song. We will then use a variable to keep track of pause and play operations. Depending on the Pause or Play, we will stop or play back the timer to run the progress and hex display bar as well as music player.

```
else if (*edge_capture_ptr == 0b1000) // Previous song KEY3
            // Add Play functionality later
{
    int WriteLED = 0;
    progress = 0;
    scrub = 0;
    progress = 0;
    a = 0;
    b = 0;
    timertrack = 0;
    if(tracknumber == 0)
        {tracknumber = 3;}
    else
            {tracknumber--;}

    // Shut off 7 segment displays to start //
    IOWR_ALTERA_AVALON_PIO_DATA(HEX1_BASE, 2139062143);
    IOWR_ALTERA_AVALON_PIO_DATA(HEX2_BASE, 2139062143);

    //Play previous song code
    printf("Previous song selected from SD Card \n");
    printf("------------------------------ \n");
    IOWR_ALTERA_AVALON_PIO_DATA(LEDR_BASE,WriteLED); // Shut off LEDR
    IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_BASE, 0b0110); // Start, Continuous
    IOWR_ALTERA_AVALON_PIO_EDGE_CAP(KEYS_BASE, 0x00); // Resets the Pushbutton context; IMPORTANT
}
```

Figure 3.30: Key 3 initialized

In **Figure 3.30** we will initialize the KEY3 to do several things for our system. We will first initialize the global variables back to 0. Then we will Shut off 7 segment displays to start again when the previous song is playing. We will also restart the timer for our new song playing in the system. And finally we will printf all the operations done to our Nios 2 console for our user to read.

```
  else if (*edge_capture_ptr == 0b0010) // Next song KEY1
              // Add Play functionality later
  {
      int WriteLED = 0;
      progress = 0;
      scrub = 0;
      progress = 0;
      a = 0;
      b = 0;
      timertrack = 0;
      if(tracknumber == 3)
          {tracknumber = 0;}
      else
          {tracknumber++;}

      // Shut off 7 segment displays to start //
      IOWR_ALTERA_AVALON_PIO_DATA(HEX1_BASE, 2139062143);
      IOWR_ALTERA_AVALON_PIO_DATA(HEX2_BASE, 2139062143);
      IOWR_ALTERA_AVALON_PIO_DATA(LEDR_BASE,WriteLED); // Shut off LEDR
      IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_BASE, 0b0110); // Start, Continuous
      //Play NEXT song code
      printf("Next song selected from SD Card \n");
      printf("------------------------------ \n");

      IOWR_ALTERA_AVALON_PIO_EDGE_CAP(KEYS_BASE, 0x00); // Resets the Pushbutton context; IMPORTANT
  }
}
  return 0;
```

Figure 3.31: Key 1 initialized

In **Figure 3.31** we will initialize the KEY3 to do several things for our system. We will first initialize the global variables back to 0. Then we will Shut off 7 segment displays to start again when the next song is playing. We will also restart the timer for our new song playing in the system. And finally we will printf all the operations done to our Nios 2 console for our user to read.

4. ANALYSIS

4.1 Experimental Results

Results for KEY1 are seen in **Figure 4.0**, when the button is pressed the time variables are reset and the next song variables are called to run the progress bar and timer.

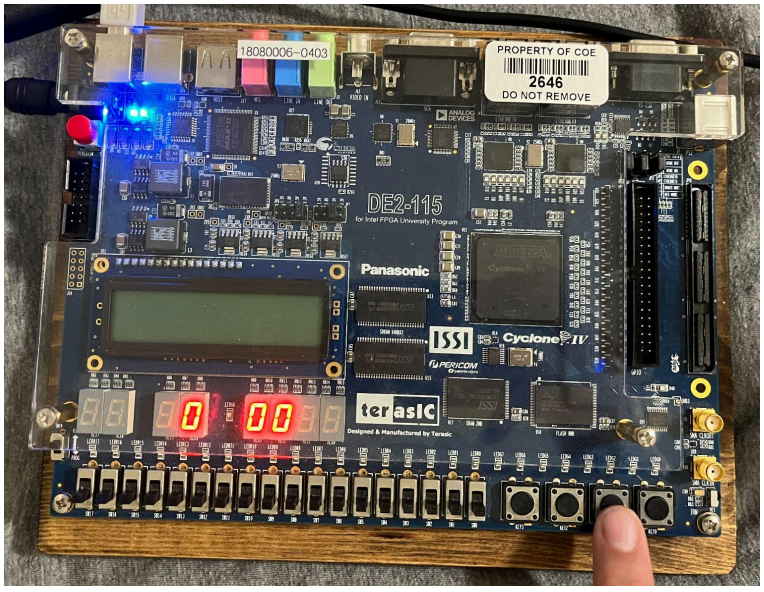

Figure 4.0: KEY1 pressed to skip to next song

Results for KEY3 are seen in **Figure 4.1**, when the button is pressed the time variables are reset and the previous song variables are called to run the progress bar and timer.

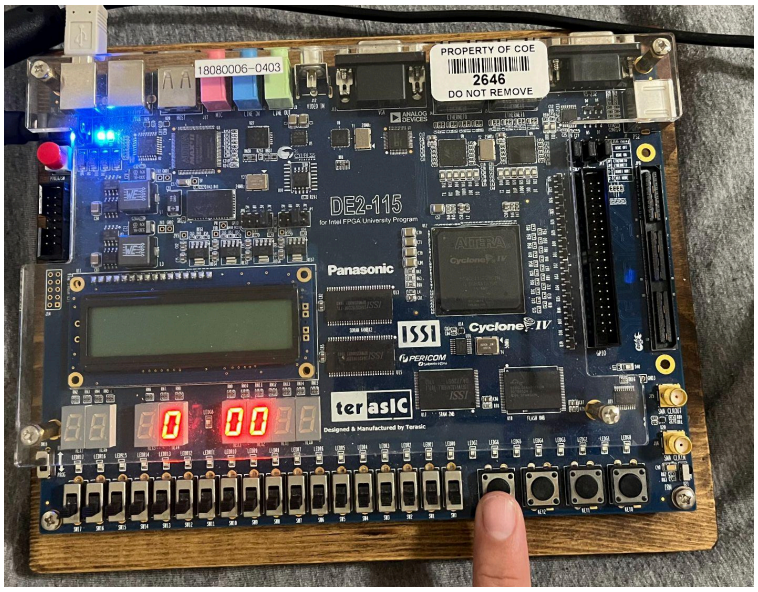

Figure 4.1: KEY3 pressed to go back to previous song

Results for KEY2 are seen in **Figure 4.2** and **Figure 4.3**, when the button is pressed the program checks to see if the song is currently playing, if it is the progress bar and timer are paused however if the song is not currently playing, the progress bar and timer resume.

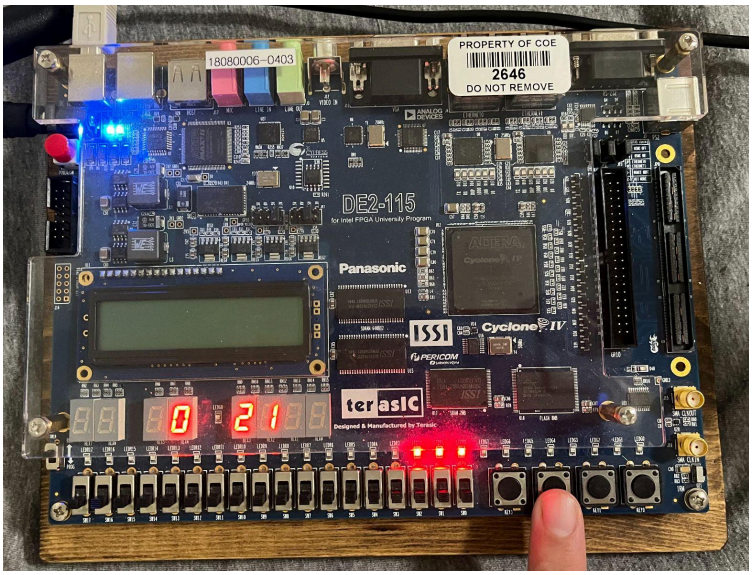

Figure 4.2: KEY2 pressed to pause current song

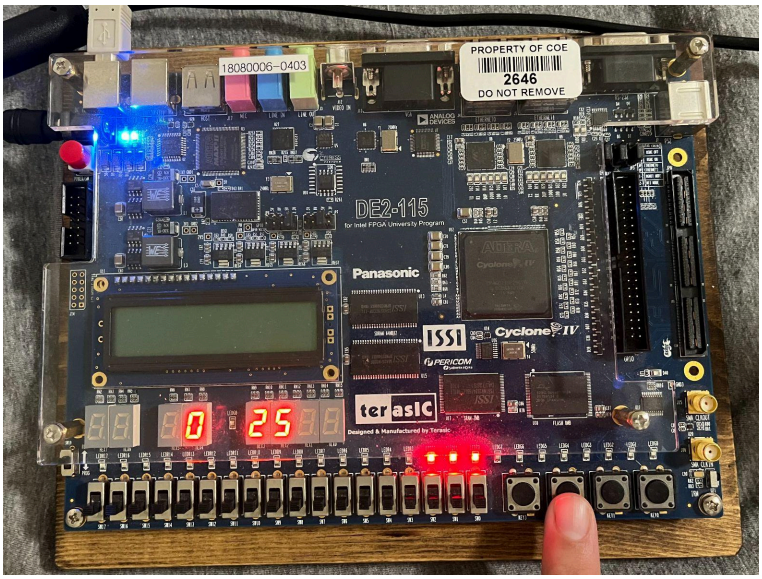

Figure 4.3: KEY2 pressed to resume playing current song

When a song has ended, the progress bar is full and timer stops displaying the final song run time until either KEY1 or KEY3 are pressed, results for this are seen in **Figure 4.4**.

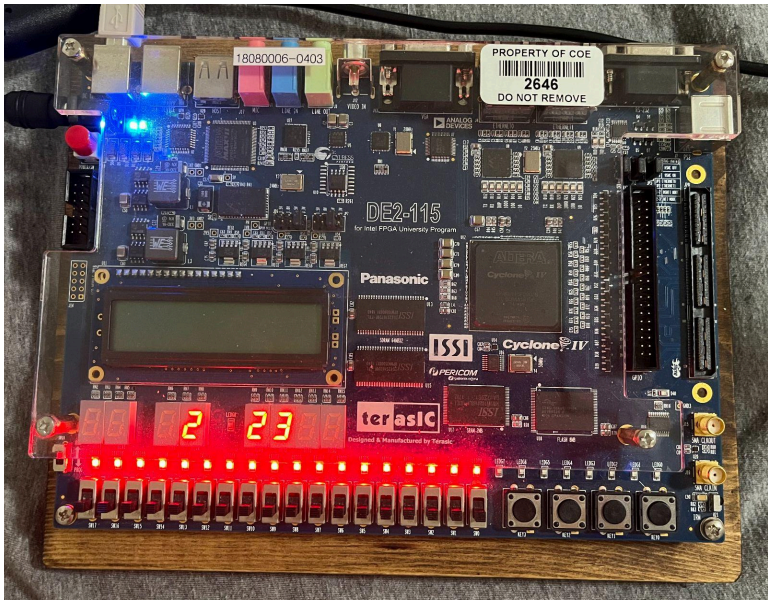

Figure 4.4: Song over, progress bar full and timer stopped

The SD card program waits for a card to be inserted, when the SD card is inserted the program would first check to make sure it was Fat16 configured, if it is the program reads the files one by one and displays the contents and well as the size, it is is not FAT16 configured the program fails to initialize.

Problems Tasks Console Nios II Console Properties

178_Complete_Program Nios II Hardware configuration - cable: USB-Blaster on localhost [USB-0] device ID: 1 instance ID: 0 name: jtaguart_0

```
Initialized. Waiting for SD card...

Card Connected

FAT16 file system detected

============================

Found file: Song1.wav

Contents:

Inside this .wav file is the period and length of the song

Content size: 58

============================


Card Disconnected
```

Figure 4.5: SD card readings

4.2 Data Analysis

The program in **Appendix B** was mostly based on the code on page 11 of [1] and modified to fit the Nios II system used in the experiment. A quick debugging session of the code revealed that the program was able to successfully access the audio CODEC and enter the while loop, but nothing else would notably occur; the former point was made evident during demonstration, where the board's system was reset and the speakers used finally produced an output in the form of a single tone, likely a result of improperly closing access to the audio CODEC. This means that the condition of the if statement—checking if the fifospace variable, or what is stored in the Fifospace Register—continuously fails and returns false. The only way to affect what is in the Fifospace Register known to the team is to write to it, and that only occurs when the if statement returns true. Thus, when the system is unable to write to the FIFOs, it is unable to read from them then subsequently write to them and echo the input audio. The point of contention becomes when the data should be written to the FIFOs in order to start the while loop.

The program in **Appendix C** accesses the SD Card core and enters a while loop that continuously detects if an SD Card is present in the slot. Once an SD Card is detected, it scans the card to check if it is FAT16 formatted then for data, namely, the first file accessible first then subsequent data afterward. It outputs the files' names and contents, as shown above in **Figure 4.5**. This program works fine with test text (.txt) files inserted into the SD Card. Curiously, when the program was tested with actual audio (.wav) files, the contents output were a large block of rectangular symbols. Similar to how the contents of an image file may be a string of R, G, and B integer values and potentially some compression algorithm, the contents of an audio file may be strings of binary resembling some method of waveform digitization that do not necessarily align with any internationally recognized character such as ASCII. Thus, the resulting "characters" in the JTAG output are similarly unrecognizable.

The team was unsuccessful in merging the program with that in **Appendix D**. This was found to be caused by the inherent use of the outermost while loop; its condition statement is set to 1, which means it should always run true. When integrated into the main program, the Program Counter would simply remain trapped inside, leaving the rest of the program to be discontinued since both the Timer and Push Buttons do not rely on interrupts and cannot preempt the task at hand. This unfortunately meant that, along with the inability to integrate the audio CODEC as well, the sample audio files would be unable to be played.

What was able to be successfully demonstrated was the program in **Appendix D**. This code first initializes several things, namely the display codes for certain numbers on the Seven-Segment Displays, or HEXes, as well as the Timer. Then, it continuously reads the Timer's TO bit in its Status Register and checks it in an if statement. When the TO bit is set, it updates the global

variables and updates the HEX and LED peripherals as necessary and appropriately to the function of the experiment. All this code was placed in the main function without the use of interrupts, since, during debugging, attempting to register the Pushbuttons as an interrupt would cause the Program Counter to get stuck at a HAL function called NIOS2_WRITE_STATUS (context), found within alt_irq_enable_all(status) function, which is found within the necessary alt_irq_register() function. Though while the Program Counter gets stuck at that function, the Pushbuttons have already been registered as an interrupt at that point and can thus execute its ISR as normal. This, however, does mean that regular operations within the main() function would never come to be processed since the Program Counter is taken hostage, just as above with **Appendix C**. This means that it was found necessary to facilitate the Pushbuttons as well as the Timer through polling rather than through interrupts. However, this should have little effect on the actual program itself since the Board's internal Timers are, by default, set to 50 MHz. And if, for example, one instruction is executed every 1 period, or 20 ns, then there would be almost 50000 instructions executed before the Timer should assert one interrupt every 1 ms as needed by the program.

5. CONCLUSIONS

Overall, our project was mostly successful, as demonstrated we were able to implement a working SD card initialization and reading text from files, all push button functionalities (play/pause, skip, previous), an LEDR progress bar, and a seven segment display of the time elapsed. The parts of the project that we would like to fully complete and improve upon in the future would be reading specific .wav file data from the SD card and being able to actually play the sound via the audio CODEC.

6. REFERENCES

[1] *Audio core for Altera DE-Series Boards*, Altera Corporation, San Jose, CA, 2016. Accessed on: Dec. 5, 2022. [Online]. Available: https://www.cl.cam.ac.uk/teaching/1617/ECAD+Arch/files/docs/Audio_core.pdf

APPENDIX A

Headers and Global Variables

```
#include <stdio.h>
#include "system.h"
#include "altera_avalon_pio_regs.h"
#include "altera_avalon_timer.h"
#include "altera_avalon_timer_regs.h"
#include "altera_up_avalon_audio_regs.h"
#include "altera_up_avalon_audio.h"
#include <altera_up_sd_card_avalon_interface.h>
#define BUF_SIZE 80000 // about 10 seconds of buffer (@ 8K samples/sec)

// Necessary variables //
int songMins[] = {1, 0, 2, 3}; // Change this and the one below for testing
int songSecs[] = {18, 36, 23, 2};
int tracknumber = 0;
int timertrack = 0;
int totalSecs;
int partitions;
int scrub = 0;
int progress = 0;

// Pushbutton variables //
volatile int edge_capture;
int paused = 0;
```

APPENDIX B

C Program for Audio CODEC

```
int main(void)
{
alt_up_audio_dev * audio_dev;
/* used for audio record/playback */
unsigned int l_buf;
unsigned int r_buf;
// open the Audio port
audio_dev = alt_up_audio_open_dev (AUDIO_NAME);
if ( audio_dev == NULL)
alt_printf ("Error: could not open audio device \n");
else
alt_printf ("Opened audio device \n");
/* read and echo audio data */
while(1)
{
int fifospace = alt_up_audio_read_fifo_avail (audio_dev, ALT_UP_AUDIO_RIGHT);
if ( fifospace > 0 ) // check if data is available
{
// read audio buffer
alt_up_audio_read_fifo (audio_dev, &(r_buf), 1, ALT_UP_AUDIO_RIGHT);
alt_up_audio_read_fifo (audio_dev, &(l_buf), 1, ALT_UP_AUDIO_LEFT);
// write audio buffer
alt_up_audio_write_fifo (audio_dev, &(r_buf), 1, ALT_UP_AUDIO_RIGHT);
alt_up_audio_write_fifo (audio_dev, &(l_buf), 1, ALT_UP_AUDIO_LEFT);
}
}
// What if len (the 1s in the HAL functions) gets changed to 10?
}
```

APPENDIX C

C Program for SD Card Reader

```
int main()
{
  alt_up_sd_card_dev *device = NULL;
  int connected = 0;

  device = alt_up_sd_card_open_dev(SDCARD_NAME);
  if (device != NULL)
  {
  printf("Initialized. Waiting for SD card...\n");
  while(1)
  {
      if ((connected == 0) && (alt_up_sd_card_is_Present()))
      {
          printf("Card connected.\n");
          if (alt_up_sd_card_is_FAT16())
          {
          printf("FAT16 file system detected.\n");

          printf("Looking for first file.\n");
          char * firstFile = "filenameunchanged";
          alt_up_sd_card_find_first(".", firstFile);
          printf("Volume Name: '%s'\n\n", firstFile);

          short file;
          while((file = alt_up_sd_card_find_next(firstFile)) != -1)
          {
              int contentCount = 0;
              printf("============================\n");
              printf("Found file: '%s'\n", firstFile);

              short fileHandle = alt_up_sd_card_fopen(firstFile,false);
              printf("File handle: %i\n", fileHandle);

              printf("Contents:\n");
              short int readCharacter;
```

```
                        while ((readCharacter = alt_up_sd_card_read(fileHandle)) != -1)
                        {
                                printf("%c", readCharacter);
                                ++contentCount;
                        }

                        printf("\nContent size: %i", contentCount);
                        printf("\n============================\n\n");
                }
                }
                else
                {
                printf("Unknown file system.\n");
                }

                connected = 1;
        }
        else if ((connected == 1) && (alt_up_sd_card_is_Present() == false))
        {
                printf("Card disconnected.\n");
                connected = 0;
        }
   }
   }
   else
   {
   printf("Initialization failed.\n");
   }
return 0;
}
```

APPENDIX D

C Program for Song Progress Bar and Run Time

```
int main()
{
	volatile int *edge_capture_ptr = (volatile int*) edge_capture;

	// Song Run Time Variables //
	int
min[]={2139062080,2139062137,2139062052,2139062064,2139062041,2139062034};
//minutes 0-5

	int sec[]={1077968767,1081704319,1076133759,1076920191,1075412863,1074954111,
1073905535,1081638783,1073774463,1074823039,2034270079,2038005631,2032435071,
2033221503,2031714175,2031255423,2030206847,2037940095,2030075775,2031124351,
608206719,611942271,606371711,607158143,605650815,605192063,604143487,611876735,
604012415,605060991,809533311,813268863,807698303,808484735,806977407,806518655,
805470079,813203327,805339007,806387583,423657343,427392895,421822335,422608767,
421101439,420642687,419594111,427327359,419463039,420511615,306216831,309952383,
304381823,305168255,303660927,303202175,302153599,309886847,302022527,303071103,
75530111}; //seconds 00-60
	int a = 0;
	int b = 0;

	// Shut off 7 segment displays to start //
	IOWR_ALTERA_AVALON_PIO_DATA(HEX1_BASE, 2139062143);
	IOWR_ALTERA_AVALON_PIO_DATA(HEX2_BASE, 2139062143);



	// Timer configuration //
	IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_BASE, 0b1000); // Initial
stop
	IOWR_ALTERA_AVALON_TIMER_PERIODH(TIMER_BASE, 0x02FA); // Top half
of 50,000,000
	IOWR_ALTERA_AVALON_TIMER_PERIODL(TIMER_BASE, 0xF080); // Bottom
half of 50,000,000
	IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_BASE, 0b0110); // Start,
Continuous
```

```
	// Loop to continuously check if a partition is passed //
	int TOBit;
	while(1) // Change this condition later to something while unpaused
	{
	totalSecs = (songMins[tracknumber] * 60) + songSecs[tracknumber];
	partitions = totalSecs / 18;
	*edge_capture_ptr = IORD_ALTERA_AVALON_PIO_EDGE_CAP(KEYS_BASE) &
0b1111;
	if (*edge_capture_ptr == 0b0000) // Default; assuming a song is playing
	{
	TOBit = IORD_ALTERA_AVALON_TIMER_STATUS(TIMER_BASE) & 0b0001;
	if (TOBit == 0b0001) // Meaning a whole second has passed
	{

		scrub++;
		if (scrub < totalSecs) // As long as the timer is still within the song playing
		{
			if (scrub >= partitions) // If a threshold has passed and an LED needs to
light up.
			{
					scrub = 0; // Resets scrub so as to keep partitions static
					progress = (progress * 2) + 1; // Shifts left, then keeps the
previous LEDs lit
					IOWR_ALTERA_AVALON_PIO_DATA(LEDR_BASE,
progress); // Updates LEDs
			}
			//IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER_BASE, 0b00);
// Resets TO bit to continue operations; IMPORTANT
		}
		else // When a song has completely elapsed
		{
			break; // Exit loop, change later
			// Will probably want to reset all the variables used after the song is
finished
		}
```

```
		// Song runtime code //
		if (sec[a] == 75530111) // If sec=60 increase minutes and reset seconds
		{
			a=0; //reset second mask counter
			b++; //increase minute mask counter
			timertrack--;
			IOWR_ALTERA_AVALON_PIO_DATA(HEX1_BASE, sec[a]);
			IOWR_ALTERA_AVALON_PIO_DATA(HEX2_BASE, min[b]);
		}
		else if(timertrack > totalSecs)
		{
		IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_BASE, 0x8);
		}
		else //seconds is not yet equal to 60, continue counting
		{
			IOWR_ALTERA_AVALON_PIO_DATA(HEX1_BASE, sec[a]);
			IOWR_ALTERA_AVALON_PIO_DATA(HEX2_BASE, min[b]);
			a++; //increase second mask counter
		}
		// end of Song runtime code //


		IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER_BASE, 0b00); // Resets
TO bit to continue operations; IMPORTANT
		timertrack++;
	}
	}
	else if (*edge_capture_ptr == 0b0100) // Assuming the Play/Pause button is KEY2
	// Add Play functionality later
	{
	if (paused == 0) // If not paused, pauses
	{
		paused = 1;

		// Stops Timer, thereby stopping normal operations but not the PC from checking
Pushbuttons again
		IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_BASE, 0b1000);
		IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER_BASE, 0b00);
```

```
		// !!! Insert whatever other code needs to execute and pause here !!! //
		printf("Song Paused\n");
		IOWR_ALTERA_AVALON_PIO_EDGE_CAP(KEYS_BASE, 0x00); // Resets
the Pushbutton context; IMPORTANT
	}
	else // If paused, plays
	{
		paused = 0;

		IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_BASE, 0b0110);

		// !!! Insert whatever other code needs to execute and play here !!! //
		printf("Song Resumed\n");
		IOWR_ALTERA_AVALON_PIO_EDGE_CAP(KEYS_BASE, 0x00); // Resets
the Pushbutton context; IMPORTANT
	}
	}
	else if (*edge_capture_ptr == 0b1000) // Previous song KEY3
		// Add Play functionality later
	{
	int WriteLED = 0;
	progress = 0;
	scrub = 0;
	progress = 0;
	a = 0;
	b = 0;
	timertrack = 0;
	if(tracknumber == 0)
	{tracknumber = 3;}
	else
		{tracknumber--;}

	// Shut off 7 segment displays to start //
	IOWR_ALTERA_AVALON_PIO_DATA(HEX1_BASE, 2139062143);
	IOWR_ALTERA_AVALON_PIO_DATA(HEX2_BASE, 2139062143);

	//Play previous song code
	printf("Previous song loaded from SD card\n");
```

```
        IOWR_ALTERA_AVALON_PIO_DATA(LEDR_BASE,WriteLED); // Shut off LEDR
        IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_BASE, 0b0110); // Start,
Continuous
        IOWR_ALTERA_AVALON_PIO_EDGE_CAP(KEYS_BASE, 0x00); // Resets the
Pushbutton context; IMPORTANT
        }

        else if (*edge_capture_ptr == 0b0010) // Next song KEY1
                // Add Play functionality later
        {
        int WriteLED = 0;
        progress = 0;
        scrub = 0;
        progress = 0;
        a = 0;
        b = 0;
        timertrack = 0;
        if(tracknumber == 3)
        {tracknumber = 0;}
        else
        {tracknumber++;}

        // Shut off 7 segment displays to start //
        IOWR_ALTERA_AVALON_PIO_DATA(HEX1_BASE, 2139062143);
        IOWR_ALTERA_AVALON_PIO_DATA(HEX2_BASE, 2139062143);
        IOWR_ALTERA_AVALON_PIO_DATA(LEDR_BASE,WriteLED); // Shut off LEDR
        IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_BASE, 0b0110); // Start,
Continuous
        //Play NEXT song code
        printf("Next song loaded from SD card\n");
        IOWR_ALTERA_AVALON_PIO_EDGE_CAP(KEYS_BASE, 0x00); // Resets the
Pushbutton context; IMPORTANT
        }
  }
        return 0;
}
```