# PROJECT -3
# Evaluation of Altera DE1 SoC and Xilinx PYNQ

For

ECEN 5863 -PROGRAMMABLE LOGIC DESIGN

By

SANJU PRAKASH KANNIOTH

SRINATH SHANMUGANADHAN

RAJ LAVINGIA

Date: 13TH DECEMBER 2018

## Table of Contents:

## Executive Summary

The DE1 SoC contains a Cyclone V Altera FPGA and an ARM based Hard processor system (HPS) which provides a lot of flexibility in offloading the non-logic synthesis parts onto the HPS and thus the logic intensive parts can be configured on the FPGA fabric. Modules 1, 3 and 4 of the project use this SoC.

PYNQ is an open-source project that was created by Xilinx to make it easier for users to program with their Zynq Systems on Chips. This uses python libraries to program the processor and the programmable logic available on the board and thus provides a great deal of flexibility. We used this board for module 2 of our project.

In **module 1**, DE1 SoC board was used to program the FPGA fabric for various designs. Lab 1 involved MUX utilization, connecting IO peripherals such as switches, 7 segment display, LEDs and Keys. Lab 5 involved construction of counters, BCD counters, Real time clock and a Morse code generator.

In **module 2**, we created a Morse code generator on the Xilinx PYNQ board and configured 2 switches to act as asynchronous inputs to the board. We also had to configure the input events as interrupt events.

In **module 3,** we created a project that would use both the Cyclone V FPGA and the HPS on the DE1 SoC. The HPS communicates to the FPGA via the AXI bus and controls the LEDs on the board.

In **module 4,** we chose to use the DE1 SoC to create a game console. We programmed 3 games onto the HPS and got user inputs from the board. We used the VGA port available on the board to hook it up to a VGA monitor and also used the audio ports on the board to play some music.

**Results summary for the modules:**

| Module Number | | %utilization | Fmax |
|---|---|---|---|
| Module1 | Lab1_part1 | <1%(1) | N/A |
| | Lab1_part2 | <1%(3) | N/A |
| | Lab1_part3 | <1%(2) | N/A |
| | Lab1_part4 | <1%(1) | N/A |
| | Lab1_part5 | <1%(6) | N/A |
| | Lab1_part6 | <1%(18) | N/A |
| | Lab5_part1 | <1%(7) | 446.43MHz |
| | Lab5_part2 | <1%(48) | 154.56 MHz |
| | Lab5_part3 | <1%(82) | 140.85 MHz |
| | Lab5_part4 | <1%(63) | 53.58 MHz |
| | | | |
| Module2 | | N/A | N/A |
| | | | |
| Module 3 | | 7%(2213) | 51.23 MHz(altera_reserved) |
| | | | 92.58 MHz(clock_50) |
| | | | 717.36 MHz(HPS) |
| | | | |
| Module 4 | | 3%(1017) | 48.78 MHz(vga_pll) |
| | | | 98.53 MHz(clock_50_1) |
| | | | 717.36 MHz(HPS) |

# Module 1

**Objectives:**

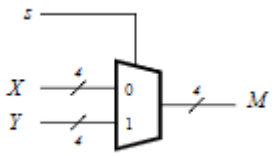
- To become familiar with SoC devices like the DE-1 Soc and appreciate the capabilities of the Altera Cyclone V System on Chip in terms of Hardware peripherals and the amount of available configurable fabric.
- To integrate software and hardware designs of the circuits and the timing systems using both the Altera Monitor Program and the SoC EDS
- Become familiar with timed circuits and how to implement them for a Morse Code Encoder and a Real Time Clock. Moreover, we also learn how to connect input and output devices to the FPGA fabric and finally implement the circuit using Verilog and VHDL code that can control the LEDs, Buttons and Switches

**Procedure:**

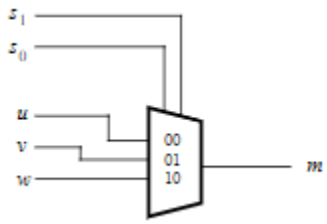
**Lab1:**

There are total 5 parts in Lab1.

- In part1, we connect each switch to the respective LEDs with the help of a given Verilog code. We also must map the pins of the board with the help of the pin planner tool in Quartus. The code is designed in such a way that on turning ON the switch the respective LEDs will turn ON.
- In part2, we implemented a mux that is used to control the LEDs with the help of different switches. If SW9 is low, then LEDR0 to LEDR3 are being controlled by switches SW0 to SW3. If SW9 is high, then LEDR0 to LEDR3 are being controlled by SW4 to SW7. Here we just have 1 select line and the inputs and outputs are 4 bits.




- In part3, we implement a 3-1 mux. Here we have 2 select lines each 2 bits wide. The inputs and outputs are 2 bits. If SW9 is 1, LED0 and LED1 are lighted up by switches 4 and 5 and if SW9 is 0 and SW8 is 1 LED0 and LED1 are lighted up by switches 2 and 3. LED9 and LED8 will also light up when SW9 and SW8 is turned ON.




- In part4, we write a Verilog code for displaying different characters and numbers on the 7-segment display. We use 2 switches SW0 and SW1 and respective letters are numbers are displayed on the 7-segment display.

| $c_1c_0$ | DE1-SoC Characters |
|---|---|
| 00 | d |
| 01 | E |
| 10 | 1 |
| 11 | |

- In part5, we are displaying on 3 7 segment displays instead of 1. Now we have taken switches 8 and 9(select lines) and switches 0 to 5 to control the letters and numbers D, E and 1. Switches 0 to 5 provide 2 bit codes to select characters. We followed the following table given to us and made the Verilog code accordingly and rotated 2 characters and 1 number accordingly.

| $SW_{9-8}$ | Characters | | |
|---|---|---|---|
| 00 | d | E | 1 |
| 01 | E | 1 | d |
| 10 | 1 | d | E |

- In part6, we used the same code as the previous part but in this case, we used all the 7 segment displays and on changing the value of 2 select lines we were able to change the contents of 7 segment display from characters D, E and number "1".

**Lab5:**

- In part1, we wrote a Verilog code of a modulo counter. Here we have put k=20 and n=8. This counter is implemented with pushbutton 0 as asynchronous reset and pushbutton 1 as clock input. When we press pushbutton 1, counter starts from 0 and increments by 1. The outputs are displayed on the LEDs.
- In part2, we designed BCD counter (3 Digit) by utilizing the code from part1. For this counter we set the k value as 10. Here we displayed each digit of the counter on the 7-segment display. We triggered the nth place digit only when n-1th digit generates roll over signal. Pushbutton 0 is used for asynchronous reset and 50 MHz clock was used on DE-1 SoC board.
- In part3, we made a real time clock with the help of the previously used counter module. Here we have partition for minutes and seconds. There is a 2-digit seconds counter which will count till 60, 2-digit millisecond counter which also counts till 60. We also have configured Keys for reset function, stop function and for pre-setting the minute part of the time.
- In part4, we made a Morse code generator with the help of the previously used code. With the help of Keys[2..0] and Switches[7..0] we are able to display character. We have set the dot and dash duration as 0.5 and 1.5 sec. Array contains dot and dash represented by one 1 and 3 1s in array. The code is designed such that Key 0 is used for resetting and Key 1 for displaying the Morse code.

**Observations:**

The whole FPGA board has a lot of fabric in it. We could implement a lot of things at the same time. According to the observations from all the Labs and its respective parts we came to know that none of our project is using more than 1 % of the fabric. The total ALMs are 32000 in this chip 5CEMA5F31C6.

**Questions:**

1) In step 2 do the displays work as expected? **Yes**
2) In step 3, is your timer accurate? How would you confirm this? **Yes, the timer on the board is very accurate. We started the stop watch on our phone and at the same time the clock on the board was started. Cross referencing the 2 values we were able to confirm that the timing in the board is accurate.**
3) Record the fMAX for each lab. **The Fmax is not displayed for any of the parts in the first lab as they do not contain an sequential logic in them. The Fmax for lab 5 are pasted below:**

| | |
|---|---|
| Lab5_part1 | 446.43MHz |
| Lab5_part2 | 154.56 MHz |
| Lab5_part3 | 140.85 MHz |
| Lab5_part4 | 53.58 MHz |

4) Estimate the % utilization of the FPGA logic for each lab at completion.
**The % utilizations for all the lab parts are listed below.**

| | |
|---|---|
| Lab1_part1 | <1%(1) |
| Lab1_part2 | <1%(3) |
| Lab1_part3 | <1%(2) |
| Lab1_part4 | <1%(1) |
| Lab1_part5 | <1%(6) |
| Lab1_part6 | <1%(18) |
| Lab5_part1 | <1%(7) |
| Lab5_part2 | <1%(48) |
| Lab5_part3 | <1%(82) |
| Lab5_part4 | <1%(63) |

**Lessons Learned:**

For Module1 Part1, initially we programmed the given code on to the board but were not able to light up the LEDs with the help of the switches. Then we realized the problem was with the pin planning. After doing proper pin assignments we were able to get the correct output. Assigning each pin manually is a tedious task and is not too efficient. Using Tcl script instead of that makes the larger designs more efficient.

The buttons and 7 segment displays on the board are active low. Moreover, the buttons are also debounced which indicates that they can be used for clock and reset.

# Module 2

**Objective:**

- Learn how to program the Xilinx Pynq board using python

**Procedure:**

- Downloaded the PYNQ-Z1 v2.3 image from http://www.pynq.io/board and created a bootable SD card using win32DiskImager.
- Inserted the SD card onto the board and booted it up by putting the jumper wires in the right position to boot device from SD card.
- One end of an Ethernet cable was connected to the board and the other end was connected to the PC. The PC was then configured with a static IP to facilitate communication with the board. The IP of the board is https://192.168.2.99 and thus the static IP assigned to the PC should be in the form 192.168.2.x where 'x' is any value from 0 to 255 except 99. We configured the IP to be 192.168.2.95.
- Once the static IP was assigned, typing the boards IP onto a web browser prompted the Jupyter notebook login screen. Once logged in the all the example files were found inside the Getting Started folder in the Jupyter notebook.
- We also found that there was a file sharing service called Samba running on the board which helped upload files from our PC to the board directly. This was achieved by typing \\192.168.2.99\xilinx in the windows navigation bar.
- To program the Morse code project using Asynchronous interrupt handlers we took the help of the Asyncio example program available on the Jupyter notebook.
- Button 3 was configured for dots (.) and button 2 was configured for dashes (-). Button1 was used to assert the user selection. We stored the Morse code values in a Dictionary (a python data structure). The user input was compared with the dictionary and the corresponding alphabet was displayed at the output.

**Lessons learned:**

The PYNQ-Z1 image already programmed on the SD card was not functioning properly initially and led to a lot of frustration as we were not able to figure out why typing the IP on the web browser was not opening the Jupyter notebook. Once a new image was burnt onto the SD card the IP was working as expected. So it is a good practice to always start by erasing the already existing image and burn the latest version available on the PYNQ website.

Also, we figured that a debouncing logic was required for the button presses as a single press of the button gave out multiple values. The debouncing logic was created by using a simple delay. This delay was created using the asyncio.sleep() function.

# Module 3

**Objectives:**

- To learn how to use HPS/ARM to communicate with FPGA.
- To know how to use the FPGA-HPS bridge and AXI bus to interface with PIO's.
- Use Qsys to connect the port of HPS component to memory mapped slave components
- Design ARM C program to control the LED's from the FPGA fabric.

**Procedure:**

- Loaded the my_first_hps-fpga project which includes all the required pins declarations for HPS and FPGA.
- We added a PIO block on Qsys named pio_led with width as 10 and changed output reset value to 0x3ff.
- The connections from h2f_lw_axi_master from hps_0 was connected to all the slave pins of all the PIO's. Also, connected the system clock and reset. Then, we exported the external connection to connect our LED's in Verilog.
- Header file was generated using the batch file and this header file can be used in the C program to reference the addresses of the LEDs.
- We write a makefile to execute this program and a executable file gets created.
- We then boot the HPS with the given Linux console using SD card.
- To copy the executable file onto the linux, we configured static IP address using "ifconfig eth0 <IP-addr>" on linux and then copied the file from the host PC using the secure copy command.
- Executing the executable file we can see the LED's blinking on the board.

**Questions:**

1) Record the fMAX for each lab. What is the highest speed clock used in the design?
   **The highest speed clock used in our design was for the HPS which is 717.36 MHz.**

| | |
|---|---|
| Altera_reserved_tck | 51.23 MHz |
| Clock_50_1 | 92.58 MHz(clock_50) |
| HPS Clocks | 717.36 MHz(HPS) |

2) Estimate the % utilization of the FPGA logic for each lab at completion.
   **The % utilization was found to be 7% (2213/32070)**

Once the FPGA was programmed we could see the LED's blinking for 60 times. Thus, the FPGA board behaves as expected.

**Lessons learned:**

To send the executable file we used secured copy method through ethernet. Also, we learnt how to configure static IP address for the board.

# Module 4

**Objectives:**

- To apply the skills garnered in the course to develop a project of our own
- To understand how all the components of an SoC blend together to form an application and to understand the capabilities of an SoC.

**Project Description:**

We created 3 games on the DE1 SoC – Quiz, Tic Tac Toe and Hangman. The initial menu displays the game choices on the terminal emulator and the user chooses the game using the host PC keyboard. The in game inputs are configured to be controlled through the on-board keys and switches. The VGA monitor displays outputs screens based on the game chosen. The audio output is always running in the background while the user plays the games.

**Procedure:**

- The designed was started by creating a new project on Quartus. A new Qsys file was created. The HPS block was first added to the Qsys design and all the required configurations were added to the HPS component.
- Then the IO blocks for the Keys, LEDs, Switches, VGA data, VGA address and VGA write enable were added and were set as inputs or outputs depending on their functionalities in the project.
- The slave ports of all the IO blocks were connected to the master AXI bus peripheral of the HPS. The required components were exported, and base addresses were assigned to each of them.
- Next the required Verilog code was written for the VGA and Audio portion of our project. We also had a synopsis design constraint file (sdc) to set the timings of our design.
- The instantiations for the Qsys components were copied from Qsys which is available under the generate tab. Once the Verilog code was written, the tcl script was run to assign all the pins on the board.
- The code was then compiled without any errors and the SRAM object file (soc) was created. Programming this onto the board enables the FPGA and the HPS.
- To configure the HPS an SD card running a yocto linux image was inserted and a terminal emulator (putty) was opened on the host PC to access the linux system. The linux system running on the HPS does not know anything about the underlying FPGA interconnects and thus needs a form of virtual addressing to use the FPGA components.
- After successfully creating the .sof file a script can be run in the project directory using the SoC Embedded Design Suite (EDS) to generate a header file that contains all the base address for the exported peripherals from Qsys. This header file can be included in the program written on the HPS and the peripherals can be invoked from the HPS by just adding these base addresses to the virtual address base.
- The 3 games were programmed on the HPS and the user inputs were given through the on-board keys and switches. The pixel and color values for the VGA display were sent from the HPS to the FPGA through 3 PIO blocks that we had configured. One PIO block was used to receive the ‘X’ and ‘Y’ pixel co-ordinates, another one to receive the color hex code and the last one to enable the write to the VGA. The audio code runs continuously in parallel with the games.

**Lessons learned:**

The makefile given by Altera in their example projects had a few bugs. The paths were not correct as they may have been written for an older version of Quartus. We had to edit these paths to make it work. Also, the name of the executable can be changed in the Makefile.

## Conclusion

We got to implement a lot of concepts that we learned in the class in this final project. We understood how everything comes together as one whole application. The DE1 SoC is a very powerful board packed with a lot of peripherals and a lot of logic. This combined with the flexibility of having a Hard Processor ARM core makes it a very versatile combination. The HPS allows us to off load all the non-logic parts onto the HPS and lets the FPGA part take care of all the logic synthesis.

## References


https://github.com/tintinmj/Hangman-game/blob/master/game.c

https://docplayer.net/22632645-De1-soc-fpga-independent-study.html

https://forums.intel.com/s/question/0D50P00003yyT5zSAE/sopccreateheaderfiles-command-not-found-?language=en_US

https://github.com/ayk33/my_first_hps-fpga

https://www.digikey.com/eewiki/pages/viewpage.action?pageId=15925278

https://www.youtube.com/watch?v=2WUkEt4-Q7Q

http://www.dejazzer.com/eece4740/lectures/lec07_HPS_FPGA_howto.pdf

DE1-SoC_User_manual.pdf

# Appendix 1 – Audio

Source: https://www.youtube.com/watch?v=zzIi7ErWhAA

On De-1 SoC Board, audio codec WM8731 is present which has an integrated headphone driver and an in built 2 ADCs and 2 DACs. In order to configure this codec first of all we used the I2C interface. This interface requires 2 signals. 1 for clock and another one for data. When the state is idle, both the signals are kept high.

The start condition is generated by pulling the data line LOW while the clock line is still HIGH. Now in order to talk, Master(FPGA) sends the address of device (0x34 for audio codec in our case).

After a 7-bit address and a R/W bit are sent, the master releases the data line. Now after sending the data the master waits for the ACK from the slave. As soon as it gets the ACK from slave, the master sends next data. Here the data line also has the pull up resistors which pulls the line HIGH when its state is not actively controlled. A STOP condition is generated by pulling the clock line HIGH while the data line is kept LOW. After receiving the STOP condition, the data is not sent further.

In total 16 bits are sent out of which 9 bits are data and 7 bits are address bits.

In codec, there is a slave mode which will provide necessary signals like main clock, interface clock, DAC synchronization and actual audio. There is also a USB clock which will run at 12 MHz and sampling rate of 48KSps and 16-bit depth.

If we make an audio of 30 seconds, it will occupy (30 second x 8KSps x 16 bit depth rate) = 3.84Mbit of memory.

We had configured KEY0 for changing the volume up and down.


Limitations: -

We tried saving the audio from a source into the buffer and then change the sampling rate of that audio accordingly and play it at the end from MIC OUT connector, but we were not able to do it.

For doing that we need to convert the audio file into .WAV format. After opening that .WAV file on a hex editor we need to keep the sound and remove everything else from that. Now that resulting hex file needs to be converted into the. mif file. This resulting .mif file has to be given to Qsys and then we can play the recorded music.