Technical University of Munich

Project Course in Power Electronics and Drive Systems

Implementation of SPI protocol for DAC7716 on Zedboard

Author:
Aeishwarya Baviskar
Enrollment No: 03693192
Abhinav Anand
Enrollment No: 03693977

Supervisor:
Mr. Eyke Liegmann

December 28, 2018

Index

List of Figures

Motivation

Efficient functioning of any power converter demands precision of the implemented control technique, which is compromised by the delays in measurement and computation. Often the whole setup becomes bulky and importable due to the amount of measuring and monitoring devices connected in the peripheral of the main converter.
This project was an attempt to implement a digital to analog converter on an FPGA which is comparatively smaller in size than the current measuring equipment being used. It reduces the size as well as the cost of the converter. FPGA also allows us to monitor and generate signals from the on board processor. Since, all the signals are readily available, signal processing becomes very feasible. The FPGA device used in this project is the Zedboard from Xilinx.

1 Introduction to the document

This document is designed as a guide to implement the SPI protocol on the Xilinx Zedboard.The aim of this project was to make a custom AXI IP with the SPI protocol for DAC7716. DAC7716 is a digital to analog converter by Texas Instruments. The programming for the protocol is done in VHDL (VLSI hardware description language).
The document is divided into FIVE sections. The first section is the introduction; the second section talks about the theoretical overview of Zedboard and its most important features that can be exploited while designing any logic with the Zedboard. The third section explains the SPI communication protocol in general and the specific requirements for the DAC7716 chip. This section also highlights details that should be kept in mind while designing such protocol for the DAC7716 chip. Section four tackles the implementation procedure of the Design flow in Vivado. The fifth section explains the procedure to create a custom AXI and IP for the protocol developed in section four.

2 Zedboard

Zedboard is a complete Field Programmable Gate Array (FPGAs) based development board which contains everything required to create any OS/RTOS (Linux, Android, Windows) based design. A Xilinx Zynq-7000 all programmable SoC tightly coupled with ARM processing system, 7 series Programmable logic, and expansion connectors exposing the Processing System (PS) and Programmable Logic (PL) Input/Outputs are also a part of the Zedboard [3].
Figure 1 shows the functional block diagram of the major component of the Zedboard and figure 2 shows their corresponding positions on the Zedboard. The biggest advantage of the Zynq-7000 processing system is that, we have the scalability and flexibility of an FPGA and the performance standards of a typical Application Specific Standard Circuit (ASIC).

2.1 Processing System (PS)

The processing system side of the Zynq-7000 executes the program code of the system and/or manages the operating system. It consists of ARM based dual Cortex A9 processing system which follows a 32 bit addressing system and is divided into three major sections-

1. Application Processor Unit (APU)
2. Fully Integrated Memory Controller
3. I/O peripherals

The PS side carries out its function with the help of central interconnects, memory interfaces, timers, on-chip memory, interrupt controllers, PS Master/Slaves and Peripheral Master/Slaves. The detailed block diagram of the Zynq Architecture is shown in figure [5].

2.2 Programmable Logic (PL)

The Programmable Logic side of the Zynq 7000 Zedboard helps offloading tasks on the Processing system side. This accelerates the tasks and reclaims processor bandwidth for additional tasks. Most importantly, the PS side is always in control of the operations performed by the PL side in a System on chip application manner.
The PL side consists of configurable logic and input/output blocks, Random Access memory (RAMs), and Digital Signal Processors (DSPs) [3]. The configurable logic blocks is a combination of a number of basic logic blocks consisting of arrays of logic gates, flip-flops and look-up-tables which can be combined as per designer's wish (Field Programmable Gate Arrays) to form a certain combinational/sequential circuit which can implement particular functionality [8].

2.3 Advanced Extensible Interface (AXI)

Utilizing the synergic effect of the PS and PL side of the Zynq SoC involves communication between the two. This is achieved by the Advanced Extensible Interface (AXI). AXI is a burst-oriented

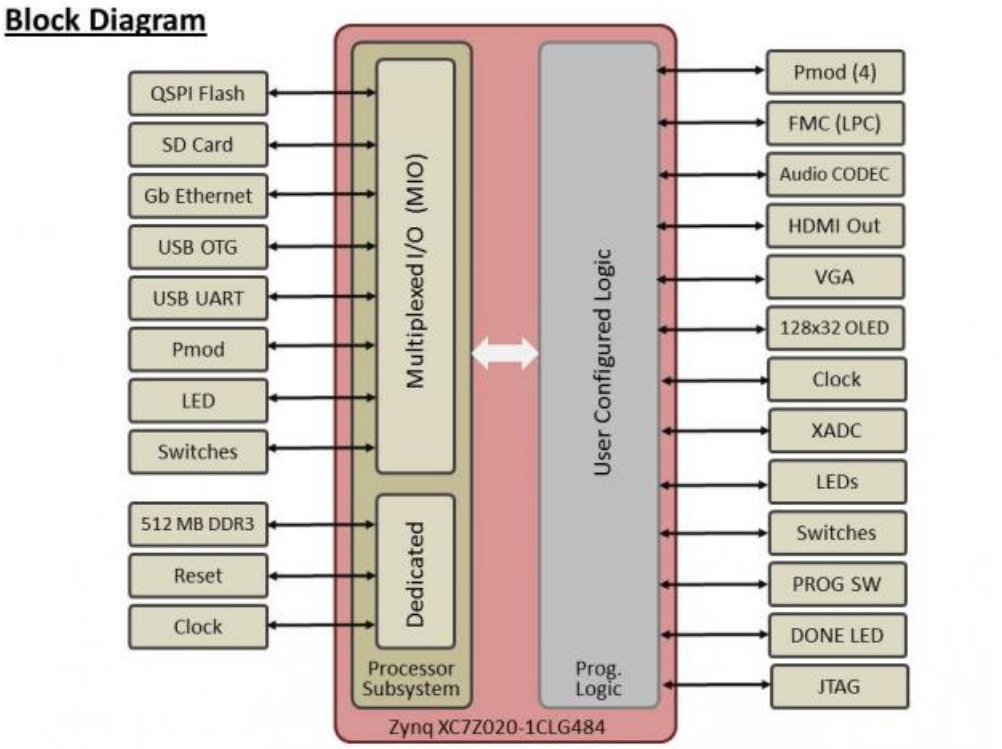

Figure 1: Block Diagram Zedboard [3]

protocol intended for high bandwidth while providing low latency. Each AXI ports contains independent read and write channels. The Zynq SoC contains three different types of AXI transfer methods that can be used to interface the PS to the PL side as follows,

1. AXI4 : For high performance Memory Map Burst Transfer
2. AXI4-Lite : For simple control interfaces
3. AXI4-Stream : For high speed unidirectional Data Transfer

With the AXI interface, communication happens between AXI-Master and AXI-Slave. AXI-Interconnect helps create transaction between two or more master/slave modules. Data is sent simultaneously in both direction between master and slave on the AXI Bus system.

2.4 FMC Connector

In addition to the PMOD headers, Zedboard also contains a Mezzanine Card (FMC) connector as an expansion module. This connector provides a unified interfacing platform for any kind of Input/Output logic (e.g- Digital to Analog Conversion chip in our case). Decoupling I/O interfaces from FPGA in this manner thus simplifies the I/O interface module design while maximizing Zedboard usage[4].

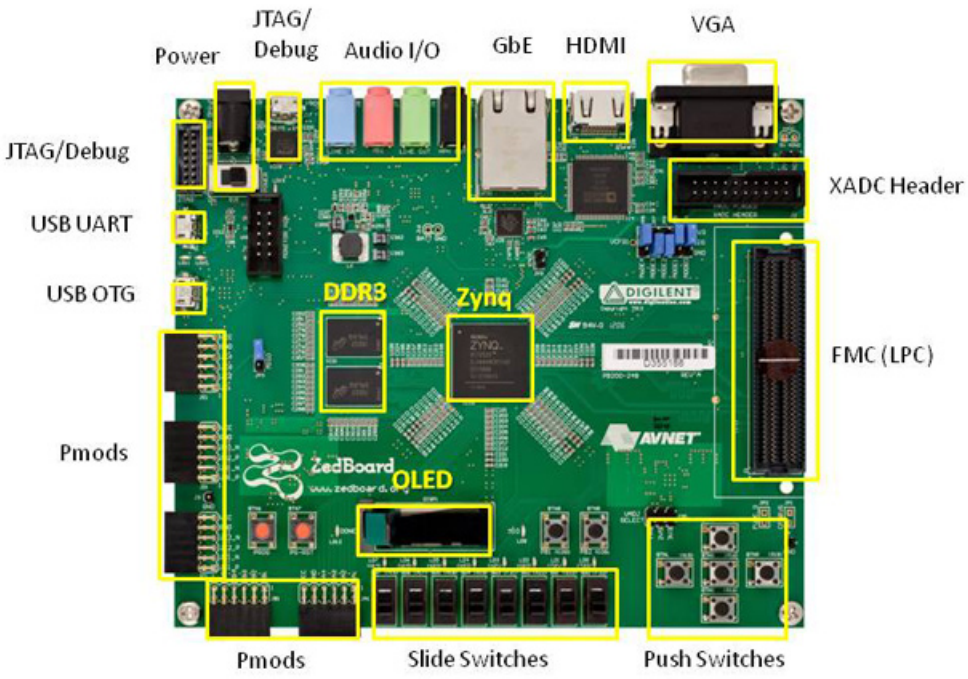

Figure 2: Zedboard Structural Block [3]

3 Serial Peripheral Interface (SPI)

In a serial interface all data bits are transferred one bit at a time. This is unlike a parallel interface where all data bits are transferred at once. It sacrifices speed of data transfer and saves a lot of interface pins on an embedded device. Manz rules are thus associated with different serial communication protocols. SPI in particular is a synchronous serial data transfer protocol [1]. Which means that every data bit is transferred with the falling edge of the clock. SPI works on four signal lines which are named as follows,

1. Chip Select [CS]
2. Clock [SCLK]
3. Master In Slave Out [MISO]
4. Master Out Slave In [MOSI]

SPI protocol has only one master [1]. The master chooses which slave to communicate with. The master initiates the communication by pulling the chip select low. The communication channel is duplex, i.e. master can send data to the slave or receive data from it. Then a clock signal is setup on which both the master and slave can communicate. Master generates data on the MOSI line and samples from the MISO line [1]. Figure 3 shows single slave connections, and figure 4 shows the multi slave connections. Several configurations can be made with a single master and multiple slaves such as the daisy chain. Since we are using only one master and slave in this work, multiple slave configurations are out of context of this text.

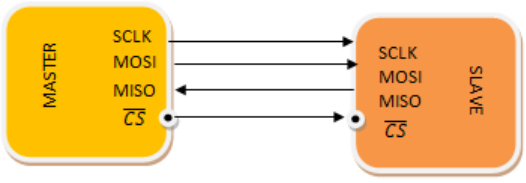

Figure 3: SPI Single Slave

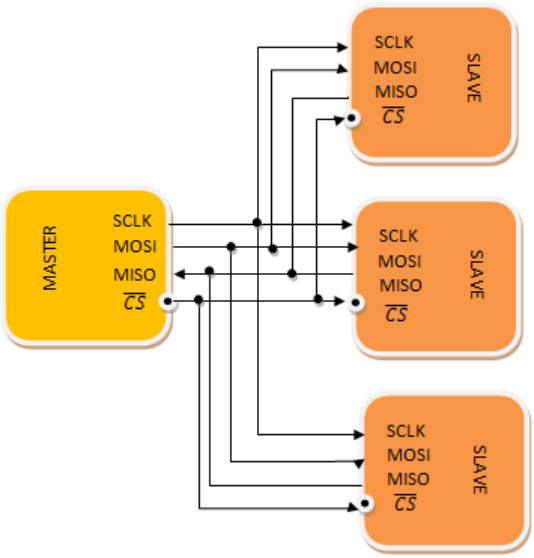

Figure 4: SPI Multiple Slave

3.1 DAC7716

Characteristic features of DAC7716:

- Quad channel : 4 analog outputs
- Clock frequency: 50MHz maximum
- Voltage levels supported: 1.8V, 3V, 5V
- Four auxiliary digital to analog control (DAC) registers
- Asynchronous reset

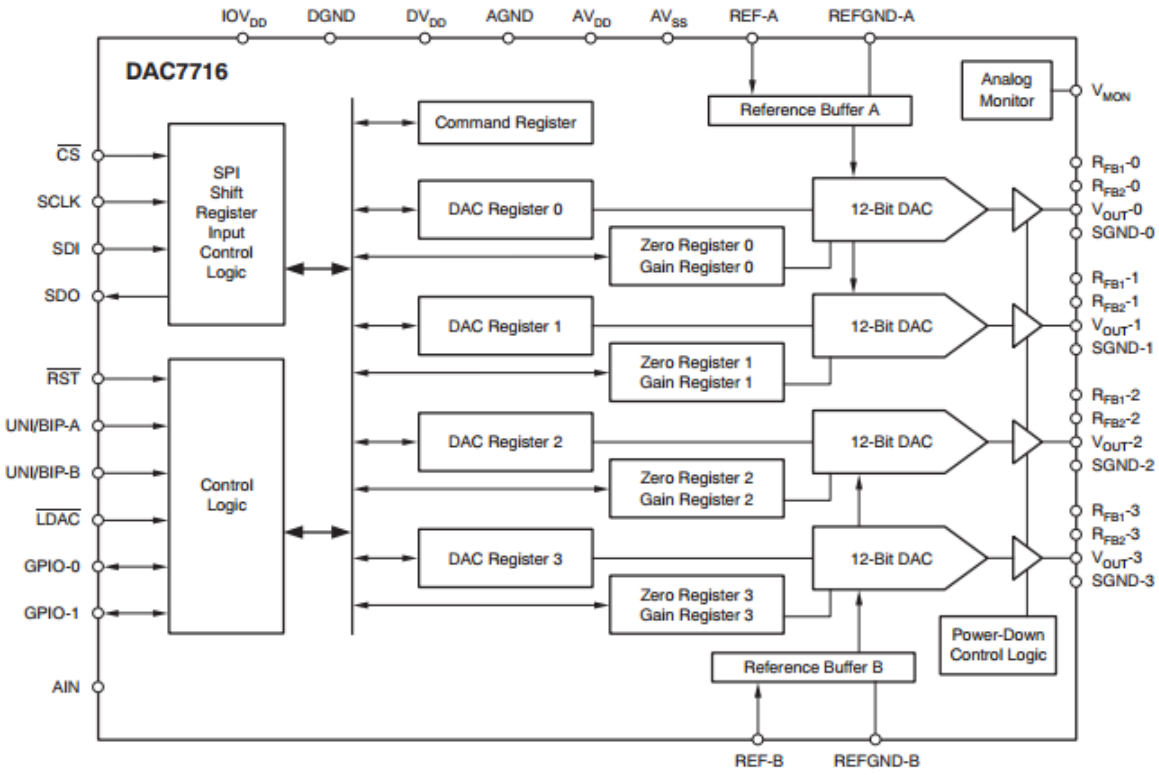

Figure 5: Block Diagram of DAC7716 [2]

As shown in figure 5 the DAC7716 has four SPI pins, namely chip select (active low) $\overline{CS}$, SPI clock pin ($SCLK$), data input SDI, and data output SDO. This IC is the SPI slave. Figure 5 also shows the four auxiliary registers in DAC7716. The SPI input register takes a 24 bit input. The 24 bit input contains a 4 bit DAC register address, 12 bit data input and other reserved bits as shown in figure 6. It should be noted that the address of the DAC register also defines the output channel. The first bit (MSB) as shown in the figure 6 defines whether the SPI master writes or reads from the slave ($\overline{W}/R$). In our case the MSB is always '0' as we want to write to the DAC register. Data is loaded in the SPI register starting from MSB. The falling edge of the chip select

23	22-20	19 16	15 4	3-0
$R/\overline{W}$	Reserved	Address	Data	Reserved

Figure 6: 24 Bit input to SPI Register [2]

signal $\overline{CS}$ is the indicator to start communication. After 24 bits are fully transferred the latch signal is pulled low to transfer the data from SPI input register to respective DAC register. If chip select $\overline{CS}$ is turned high before 24 bits are fully transferred, the batch of data is ignored. Latch can also be tied low during the entire operation. Rising edge of chip select $\overline{CS}$ indicates end of transfer. For details regarding the timings of different signals please refer to the data sheet [2].

4 Implementation in Vivado

4.1 Logic Flow

Figure 7 shows that the input clock from the processor is divided by 4 in the code. This can be divided by any desired value as long as SPI clock stays withing the range specified by the data sheet of DAC7716. Figure 8 shows the logic used to generate the chip select signal and figure 9 shows the logic used to generate the latch signal. Note that both the chip select and latch are generated independently from one another and are only connected through the edge counter. Since, it is required that the latch signal turns low only after chip select signal has turned high, latch signal is controlled with the negative edge. It ensures that latch is turned low (on for an active low signal) after exactly half cycle after chip select is turned high (off for active low signal) and vice-a-versa.

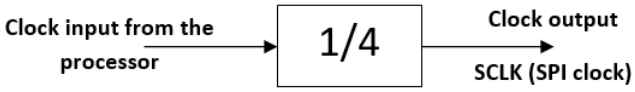

Figure 7: Processor Clock Division

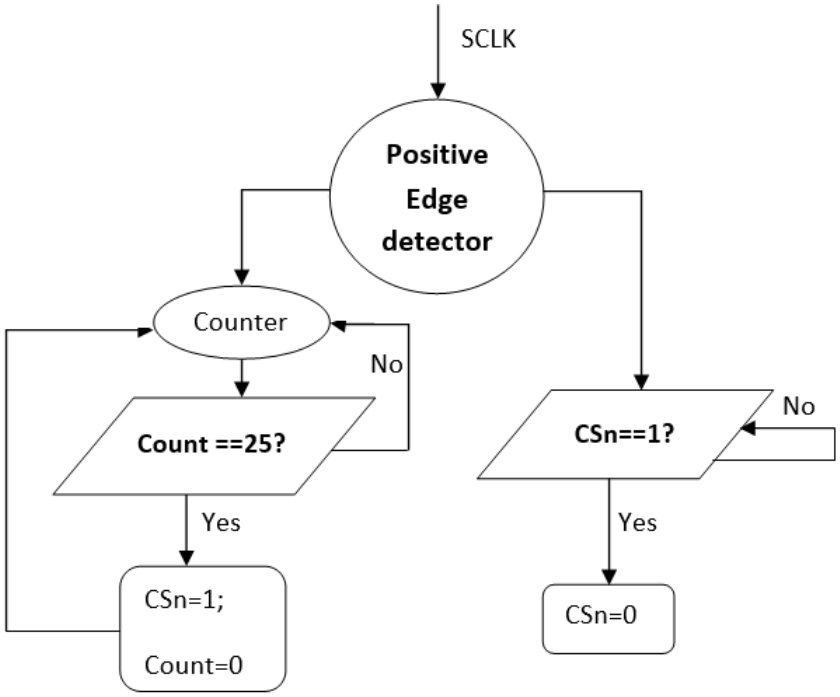

Figure 8: Chip select logic (CSn indicated active low signal $\overline{CS}$)

At every positive edge of the chip select signal, new 24 bit data is entered into the output data register. This data is then transmitted through the SDO (SPI data out) pin to the DAC7716 at

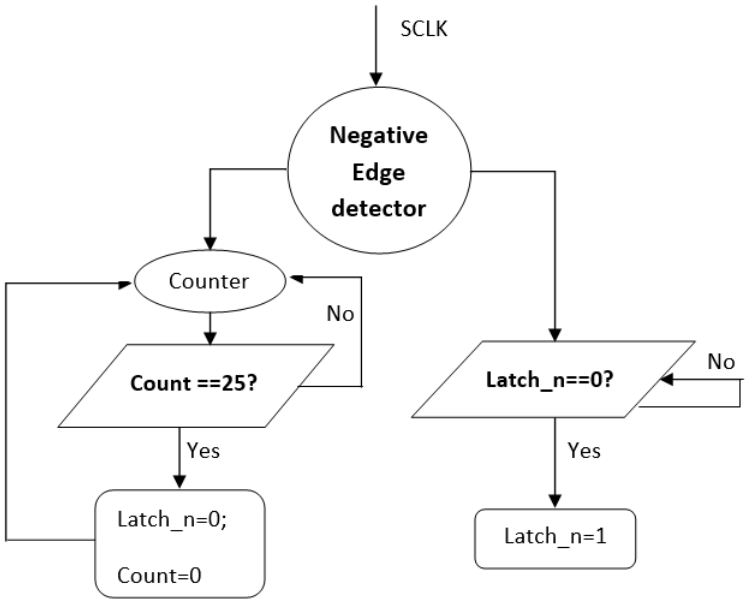

Figure 9: Latch signal logic (Latch_n indicated active low signal $\overline{Latch}$)

every positive edge of the SPI clock as shown in figure 10. The green signals in figure 10 are input signals, red represent the output signals and blue is the internal data bus. (Refer to the appendix for the complete code)

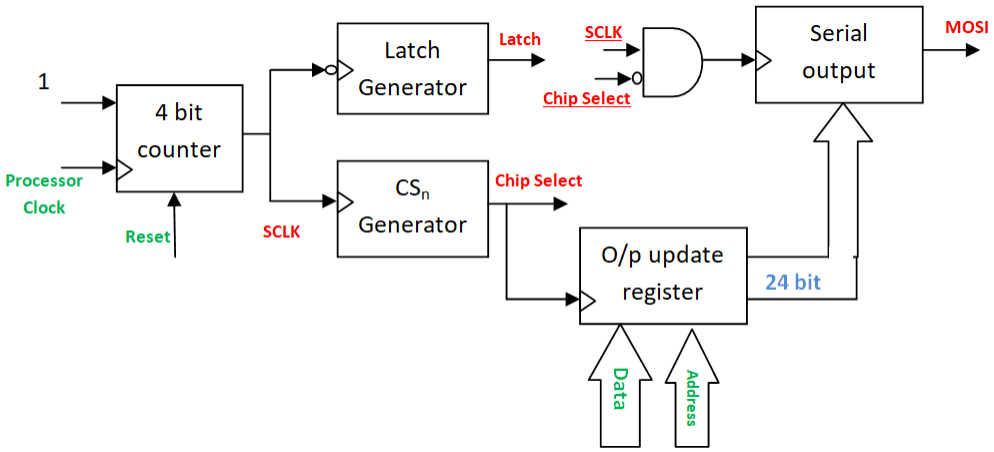

Figure 10: Complete Logic

4.1.1 Using PS clock in PL

After the VHDL code is written, the functioning can be checked by performing a simulation. During simulation a self generated clock can be used. But this clock will not work in a sequential logic on the FPGA. Hence, the programmable logic (PL) fabric clock from the processor must be used. To use the processor clock in the programmable logic, we can create a block diagram and use the VHDL logic as an RTL block as shown in figure 11. FCLK_CLK0 is the PL fabric clock used.

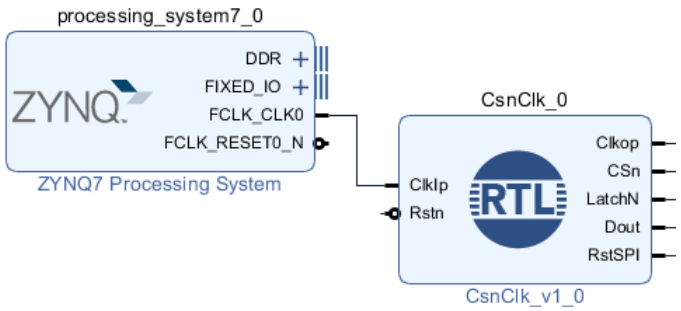

Figure 11: VHDL Logic as RTL block with Zynq processor

Customize the Zynq processor IP to configure the clock as shown in figure 12 (double click on the Zynq IP block to open this window).

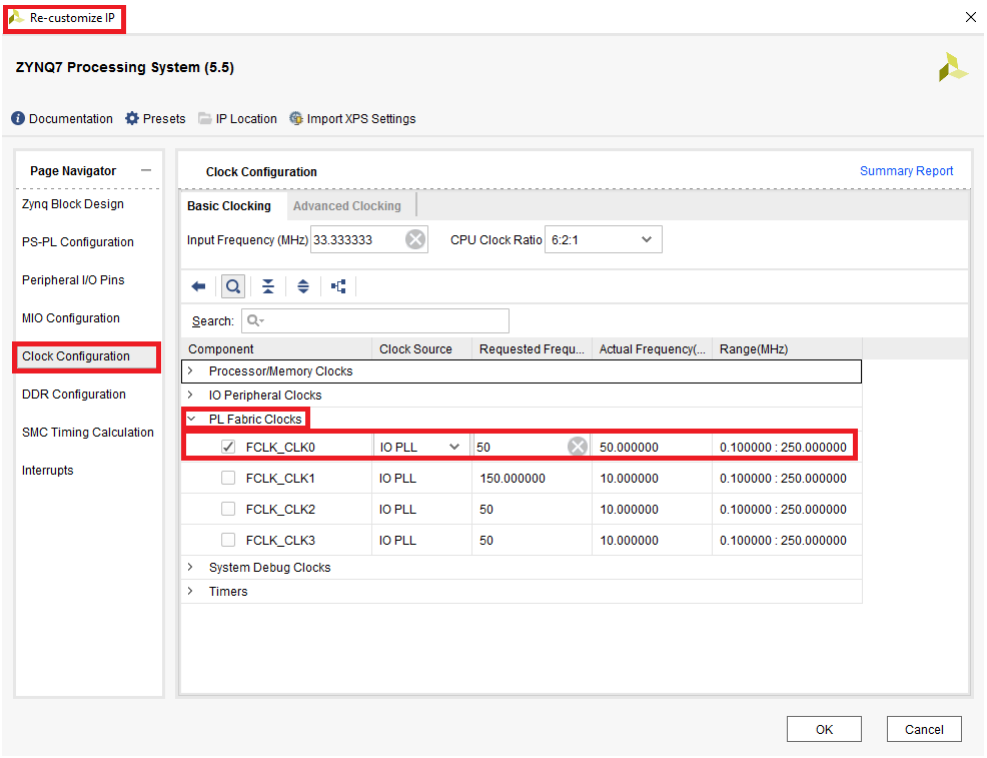

Figure 12: Customizing Zynq IP

Next step is to create a VHDL wrapper for the block diagram (Refer to figure 13). After the HDL wrapper is successfully created the source window in Vivado will look like in figure 14. The project contains the HDL wrapper created by Vivado as the top level entity, next is the block diagram and then in the end the user defined VHDL logic.

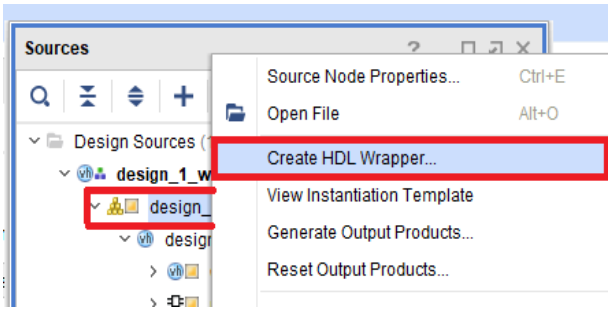

Figure 13: Creating HDL wrapper for block diagram

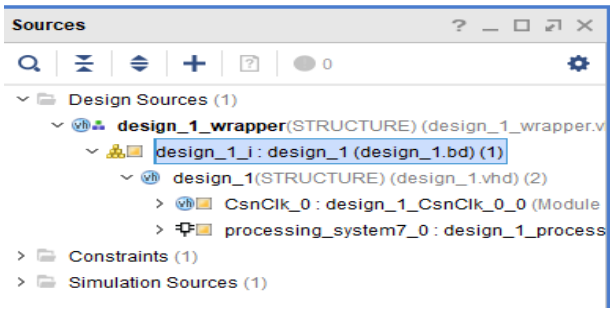

Figure 14: Source window in Vivado

Follow the steps in the process flow navigator window on the left hand side and generate a bit-stream. After the bit stream is successfully generated, follow the following steps to burn the programmable logic on the FPGA and switch the PS clock on.
Step 1: Exporting hardware: click on the file menu and then export the hardware as shown in the figure 15. Make sure that the include bit-stream tick box is checked while exporting the hardware.

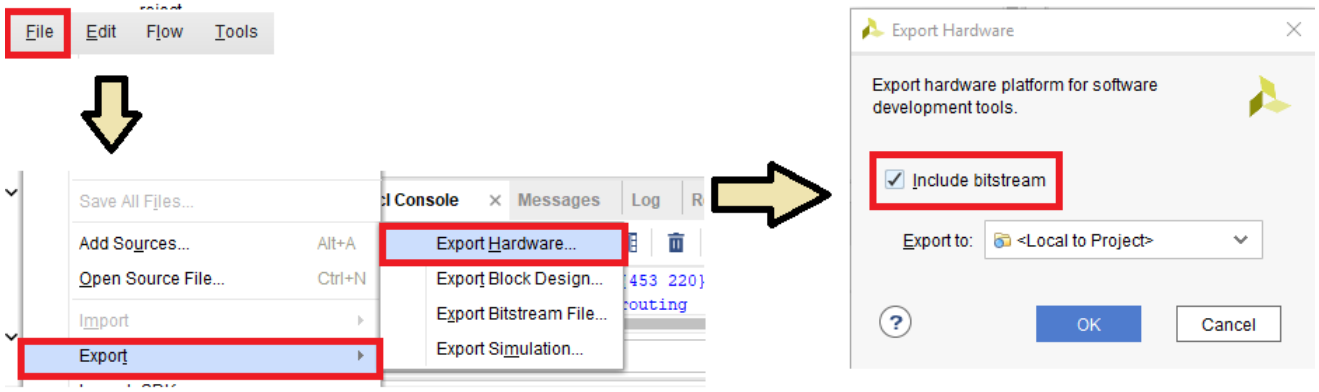

Figure 15: Exporting Hardware

Step 2: Launching SDK (refer to figure 16)

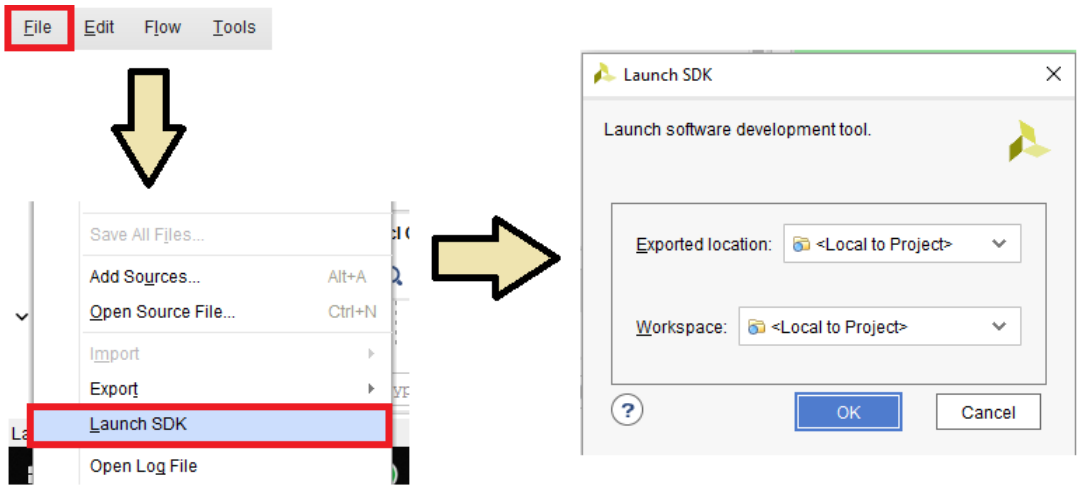

Figure 16: Launching SDK from file menu

Step 3: After the SDK is launched we need to make an empty application project in the SDK. Running this empty application will start the PS clock and thus it can be used in the programmable logic. To make an empty application in the SDK refer to figure 17. After creating the application project you can view it in the project explorer tab on the left hand side.

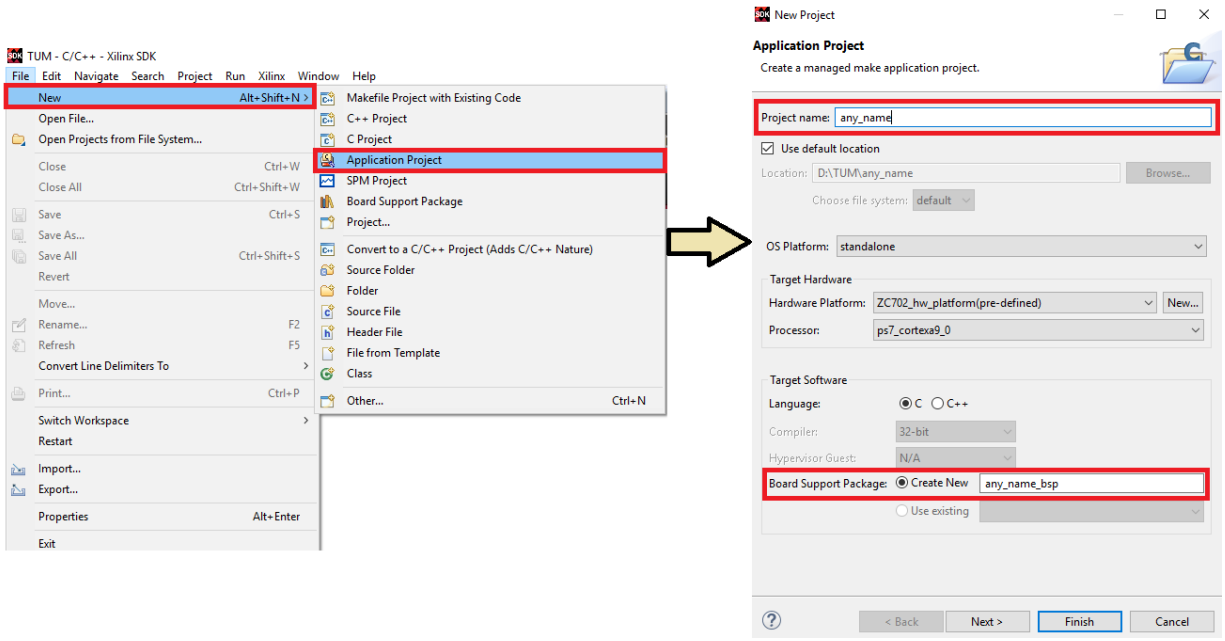

Figure 17: Creating application project

Step 4: Running the application project and programming the FPGA: Run the application project by right clicking on the project name in the project explorer (refer to figure 18).

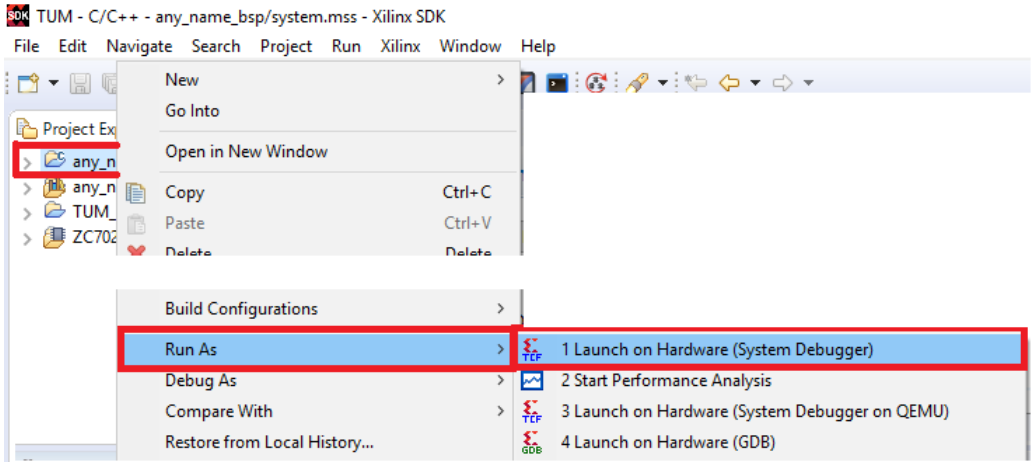

Figure 18: Running application project

Step 5: The final step is to program the FPGA as shown in figure 19.

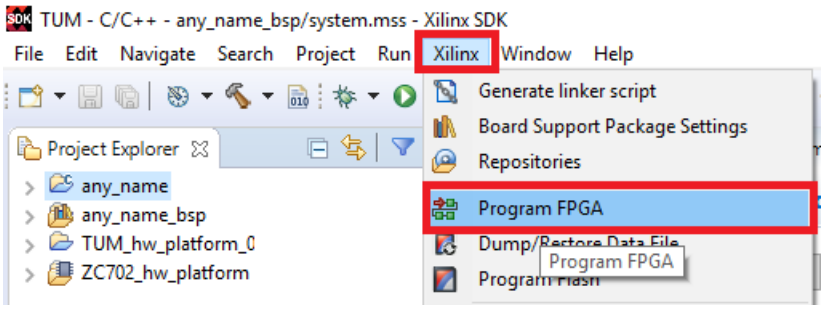

Figure 19: Programming FPGA

4.1.2 Results

The results in this section show the following input signals to the DAC7716 from the FPGA,

1. SPI input clock
2. Data input to the DAC7716
3. Chip select signal
4. Latch signal

Please note that, the auxiliary register address is set to ′0100′ and the data is ′1111111111111111′ for testing the logic.

1) Simulation in Vivado

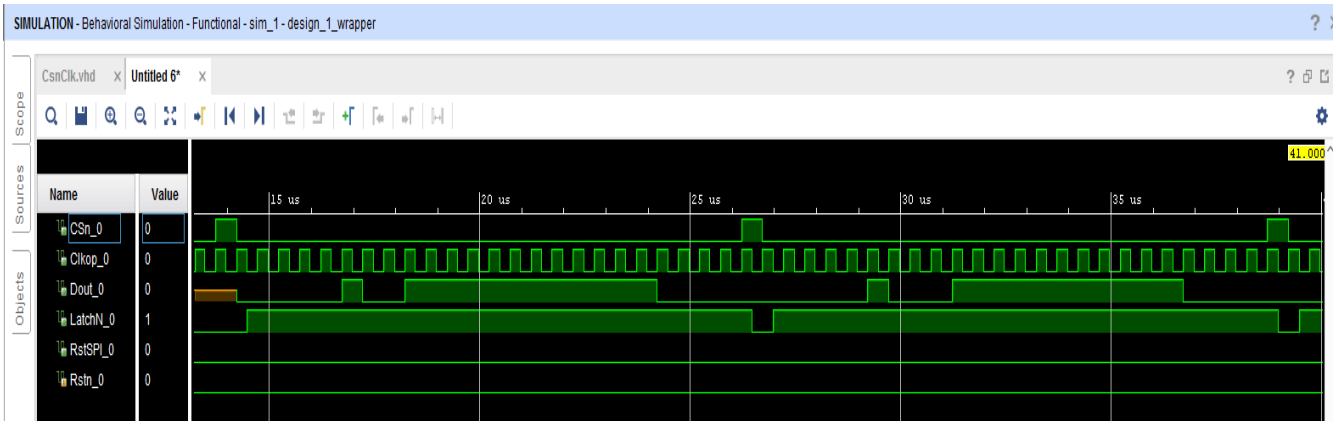

Figure 20: Signals from simulation results in Vivado

2) Result from oscilloscope.

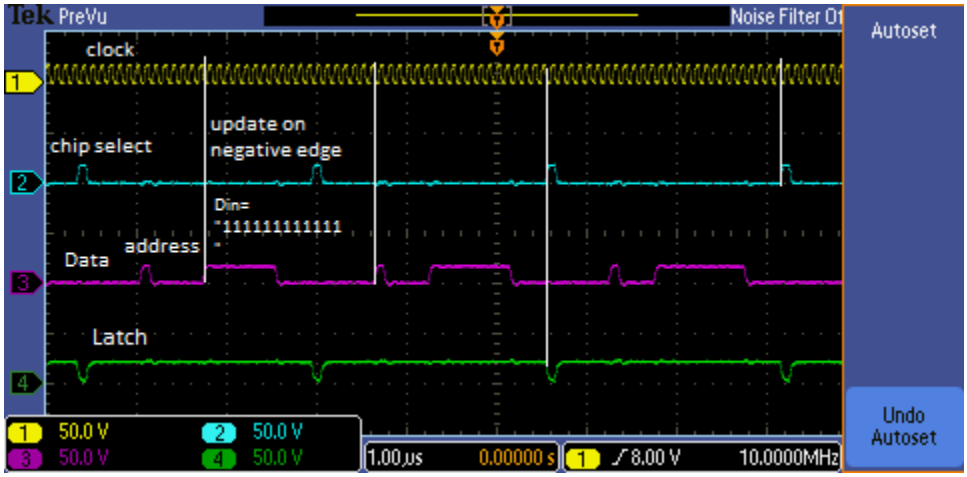

Figure 21: Signals from oscilloscope

4.2 AXI IP Customization

The Intellectual Property (IP) block refers to pre-configured logic blocks that perform a particular function in the overall design flow. Multiple IPs can be used for a particular block design which can be connected using an IP integrator provided by Vivado. It enables rapid interconnection of IPs through a common AXI protocol. The Vivado IP Packager also allows modifying the created IP and repackaging the whole configuration again as single IP block. This repackaged customized IP can then be either added to the Vivado IP catalog to be used in a local Vivado block design or can be delivered to an end user in a repository directory or in an archive. Custom Vivado IPs can be tied to the respective peripheral pins situated at different banks of the Zedboard using a single user-defined constraint file. The design flow steps can be seen in figure 22. IP customization can be done in following two manner

1. Creating an RTL module from the functional code/source file
2. Creating and Packaging using Vivado IP Packager

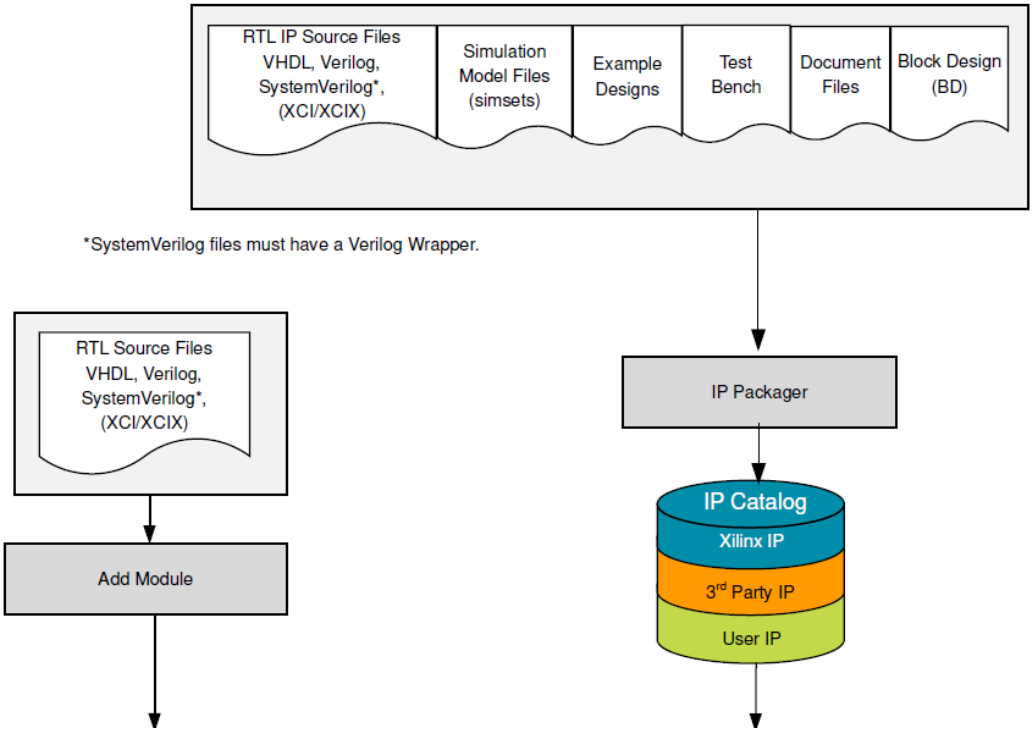

Figure 22: AXI IP Customization Design Flow [9]

With reference to this project, the second method has been followed and hence will be discussed now on. The creation, modification and repackaging of an IP block that can be used to replicate the functionality of DAC in a Vivado design block has been explained below.

4.2.1 Creation of a user-defined IP

The first step involves creating and packaging an IP using "Vivado create and package IP" tool. The idea is to create a general IP block with parameters as desired by the complexity of final functionality that the customized IP will be serving. This means choosing-

1. AXI protocol interface type[AXI4/-Lite/-Stream]
2. AXI Interface Mode [Master/Slave]
3. Number of interfaces
4. Number of registers in each of the interfaces
5. Number of data width (bits) in each register

As per our source code for SPI communication protocol, a single slave interface of four registers based on AXI4 Lite protocol is sufficient to implement in the design flow. Once the IP has been created it can be packaged and then added to the user defined repository.

4.2.2 Editing/Modifying the IP function

Step 1: Right click on the created IP and select the 'Edit IP' option. This opens a new Vivado window containing the VHDL source codes of the created IP. The IP packager window has been shown in figure 23.

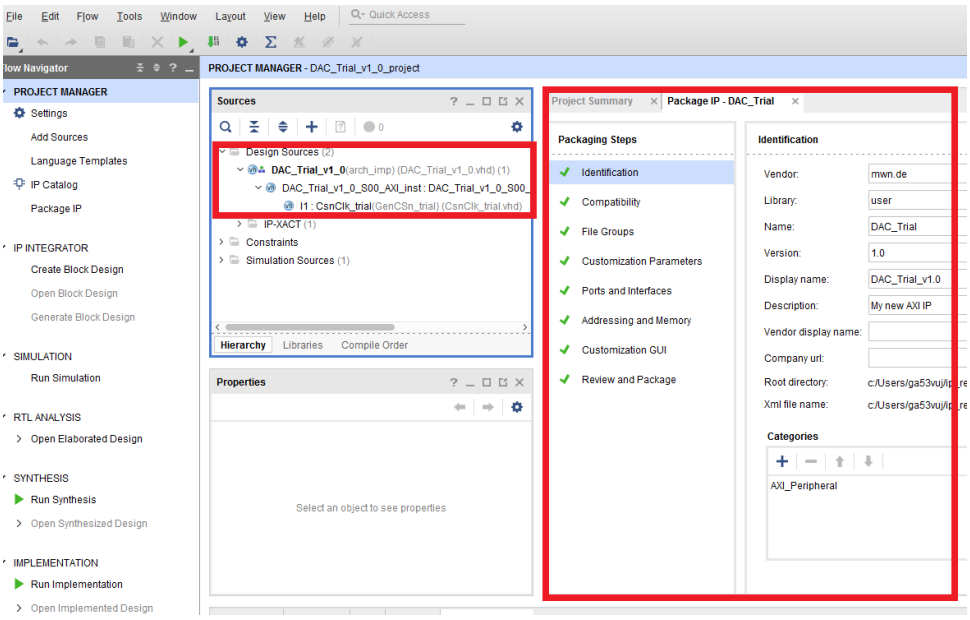

Figure 23: IP customization and packaging wizard

Step 2: Import the SPI protocol source code described in section 3 within the source code situated at lowest hierarchical level. In our case it is the interface level code (titled '_S00_AXI_'). This

process instantiates the original user defined source code within the interface level.

Step 3: Open the interface level code and make following changes-

1. create user defined ports needed for external mapping of the IP signals
2. include the new ports in VHDL code architecture while declaring component
3. add/modify user based logic: port mapping (tying up logically) the internal signals of user defined code with internal signals at interface level

Figure 24 shows the aforementioned changes.

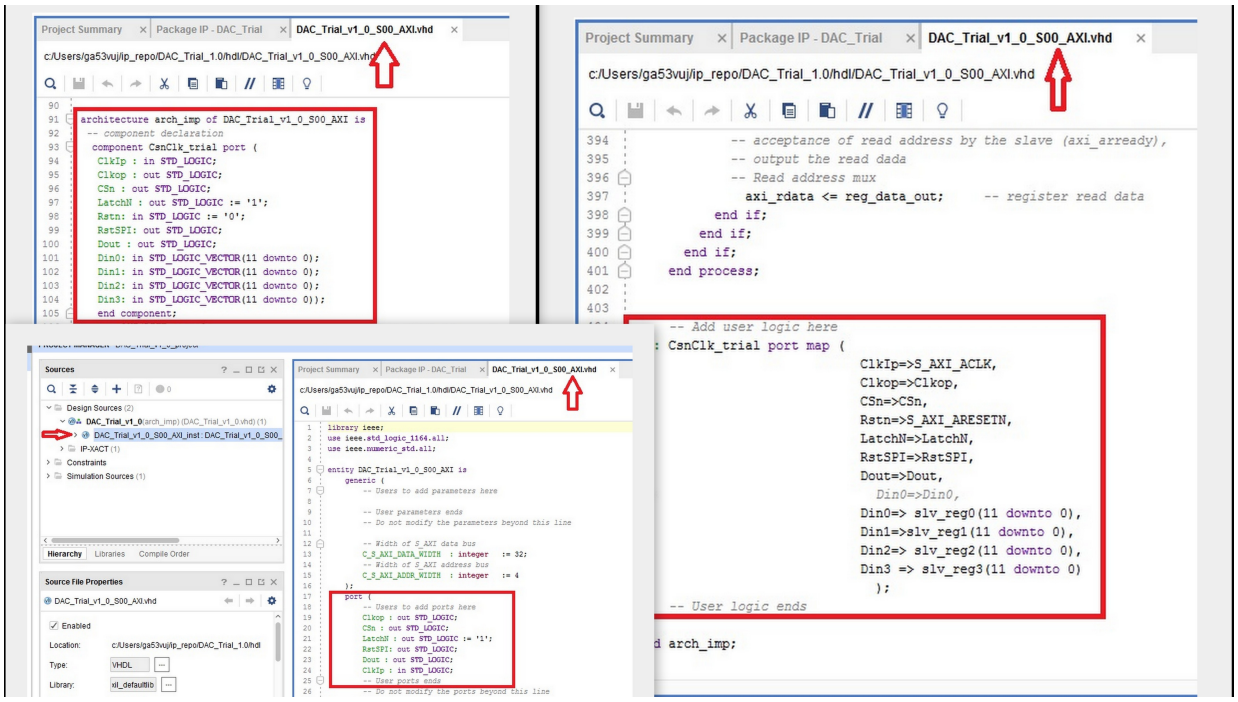

Figure 24: Slave interface level modifications

Step 4: Open the top module level code and make following changes-

1. create user defined ports (same as interface level module)
2. include the new ports in VHDL code architecture while declaring component
3. port mapping the internal signals of interface level code with internal signals at top module level

Figure 25 shows the aforementioned changes.

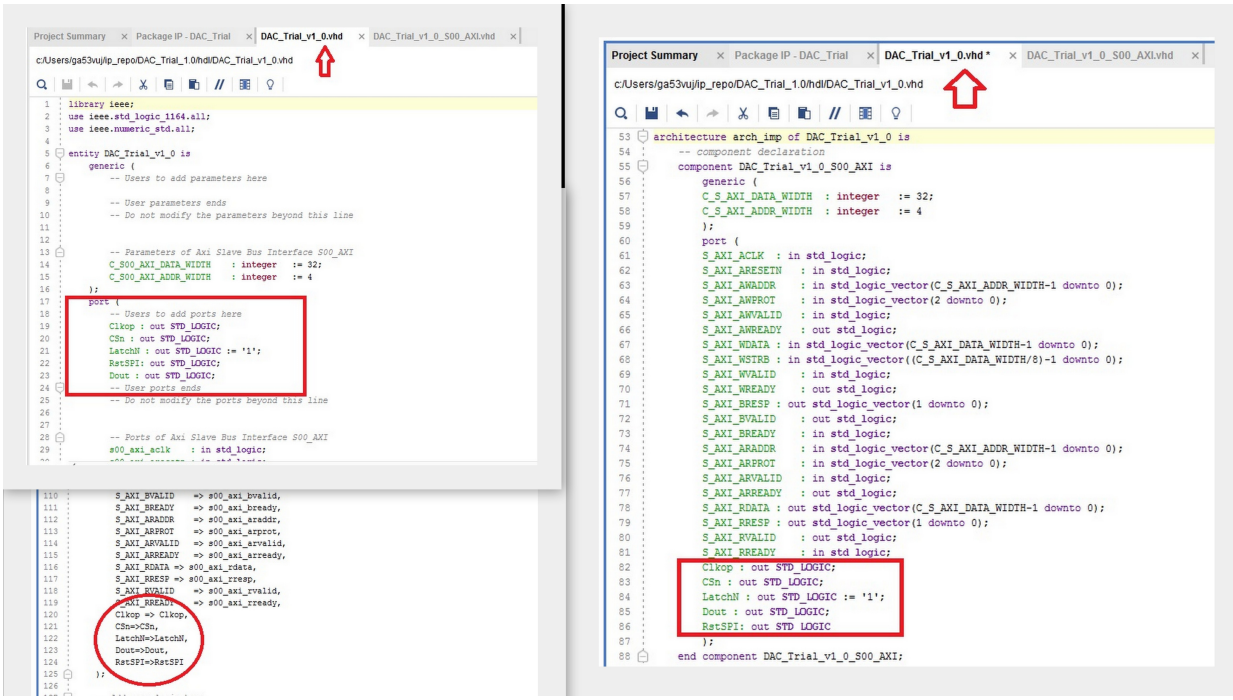

Figure 25: Top module level modifications

Step 5: Run synthesis and implementation of the customized top level source code file in order to validate avoiding any implementation level problems in future designs.

Step 6: Go to 'Package IP' tab and repackage the IP. This includes merging all the changes in file groups and updating all other categories (viz. Customization Parameters, Ports & Interfaces etc) again, as shown in figure 23.

4.3 Final System

Once the modified IP block has been added to the repository, we are ready to create the final IP based block design. Vivado provides the feature of running block addition and connection automation, which includes adding all the necessary IP to our block design on single click. The final block design is shown in figure 26. The process flow of Digital to Analog conversion implemented by this block design can be described as follows

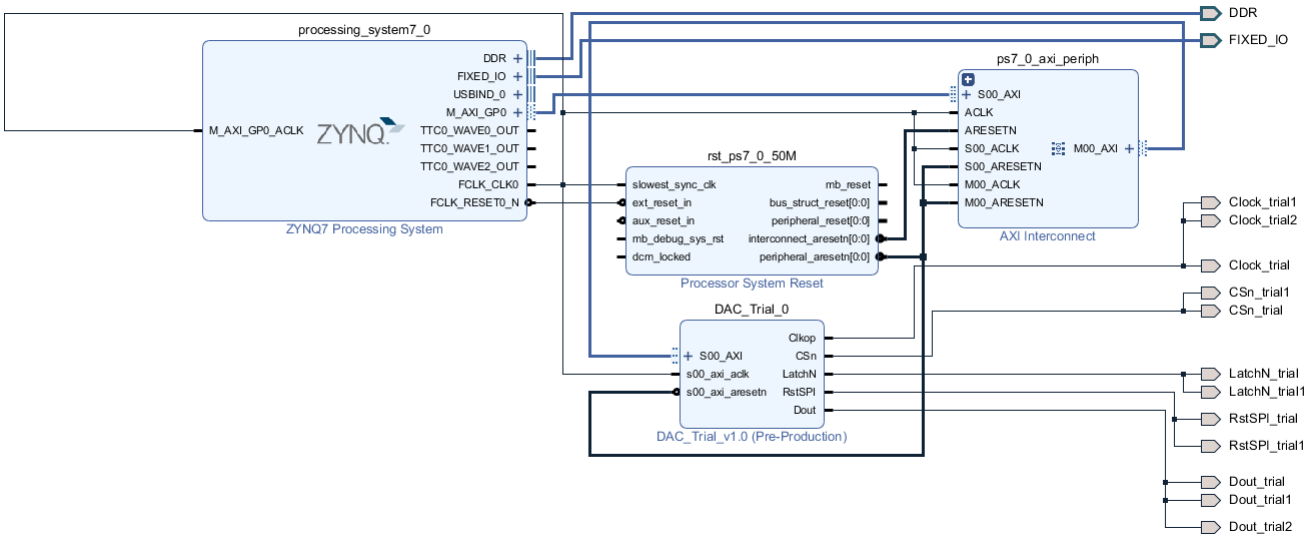

Figure 26: Final Block Design

The digital equivalent of a sinusoidal analog signal was generated on the PS side of Zynq board using look-up tables. These digital codes are sent to the DAC chip physically mounted on the Zedboard using FMC Connectors. Since, the DAC chip follows SPI based communication, the customized DAC IP encodes the input digital data in the format desired by the DAC chip. The output of the DAC chip is observed using an oscilloscope.

4.3.1 Components of Block Design

The final block design, shown in figure 26 has the following IP block as components

1. Zynq Processing system block
   This IP core represents the functionality of Zynq Processing system of the Zedboard. It acts as a logic connection between the PS and the PL while assisting the user to integrate custom and embedded IP cores within the processing system using Vivado Design Suite [5]. Although it offers the user myriad features, in this design its role is rather limited. The Zynq IP core generates the clock signal for the entire process flow and acts as a Master to the slave DAC IP. Figure 27 shows the features of Zynq IP core.

2. Processor System reset
   This logic core IP provides customized resets for an entire processor system, including the

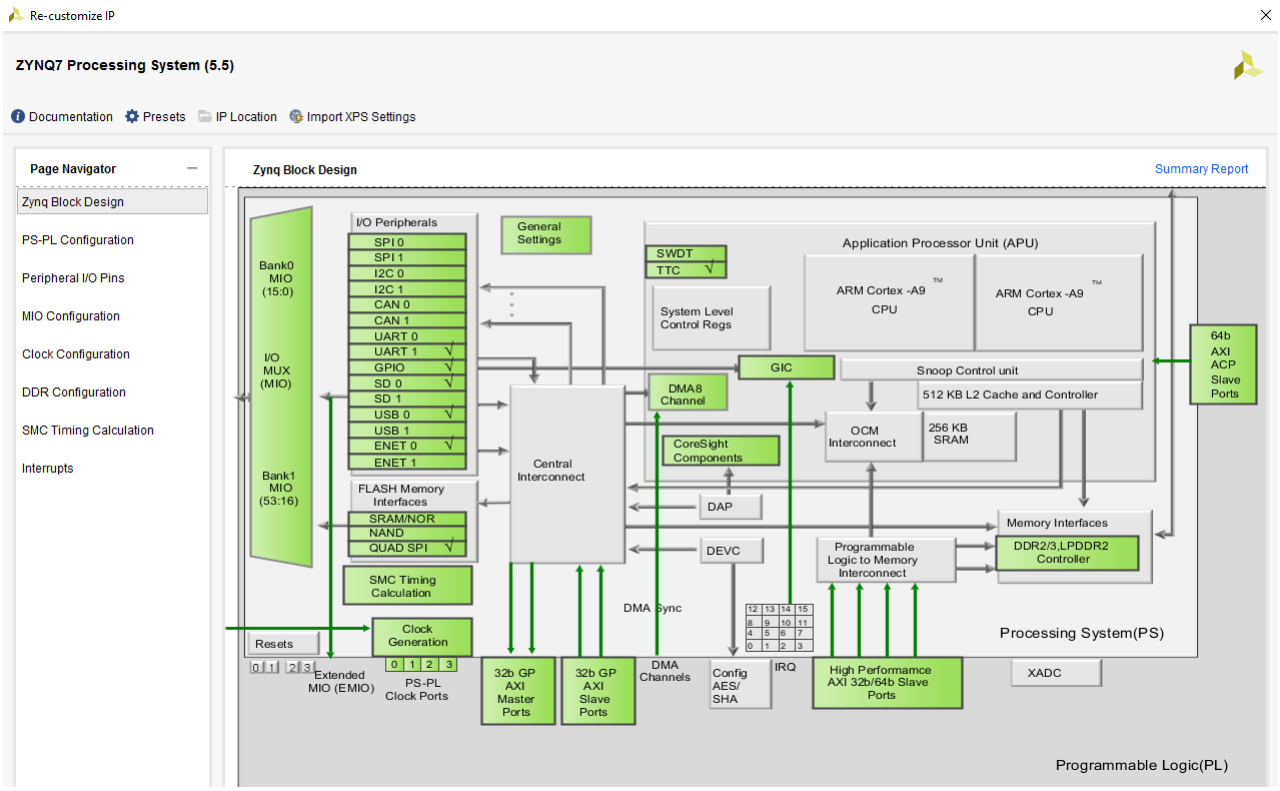

Figure 27: Zynq Processor Logic IP core

processor, the interconnect and the peripherals. It allows users to tailor their design as per their application by setting certain parameters to enable/disable features i.e it provides a mechanism to handle the reset condition for a given system. The core handles numerous external/internal reset conditions at the input, and generates appropriate resets at the output [6].

3. Customized DAC IP
   This IP core acts as Slave to the Zynq Processor IP core. Thus it takes input from the PS (through AXI interconnect IP core) and generates desired output signals. These output signals are mapped to the FMC connector pins (connected with the DAC chip) and thus are made external to the design.

4. AXI Interconnect
   The AXI Interconnect core connects one or more AXI based memory mapped Master devices to one or more AXI based memory mapped Slave devices. This means that any mixture of Master and Slave devices, varying in terms of data widths, clock domain and AXI sub protocol (AXI4, AXI3, AXI-Lite), can be connected to it [7].

4.3.2 Constraints File

According to the design flow, the output ports of the DAC IP needs to be connected to the respective input ports of the DAC chip (refer figure 28) through FMC connector.

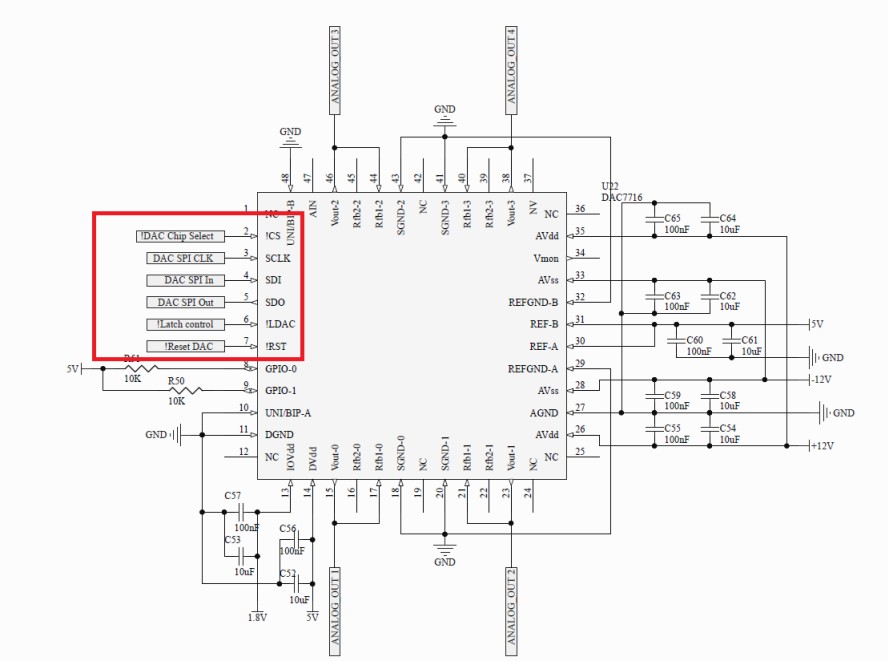

Figure 28: Layout of DAC chip [10]

The complete description of Zedboard banks and pins is available in the master constraint file available at Zedboard website, which can be modified as per requirement. The desired external PIN number on the Zedboard FMC connector is obtained from the FMC pin mapping (refer figure 29) data sheet of the DAC chip.

The customized constraint file can then be added to the main Vivado project as shown in figure 30.

4.3.3 Results

After creating the block design, it is validated using the "Validate Block Design" button. Vivado lets us automatically generate output products from the block design and create an HDL wrapper for it. Once a wrapper is created and the output ports have been correctly mapped the design is ready to move to synthesis, implementation stages and bit stream generation stages as described in details in section 4.1.1.

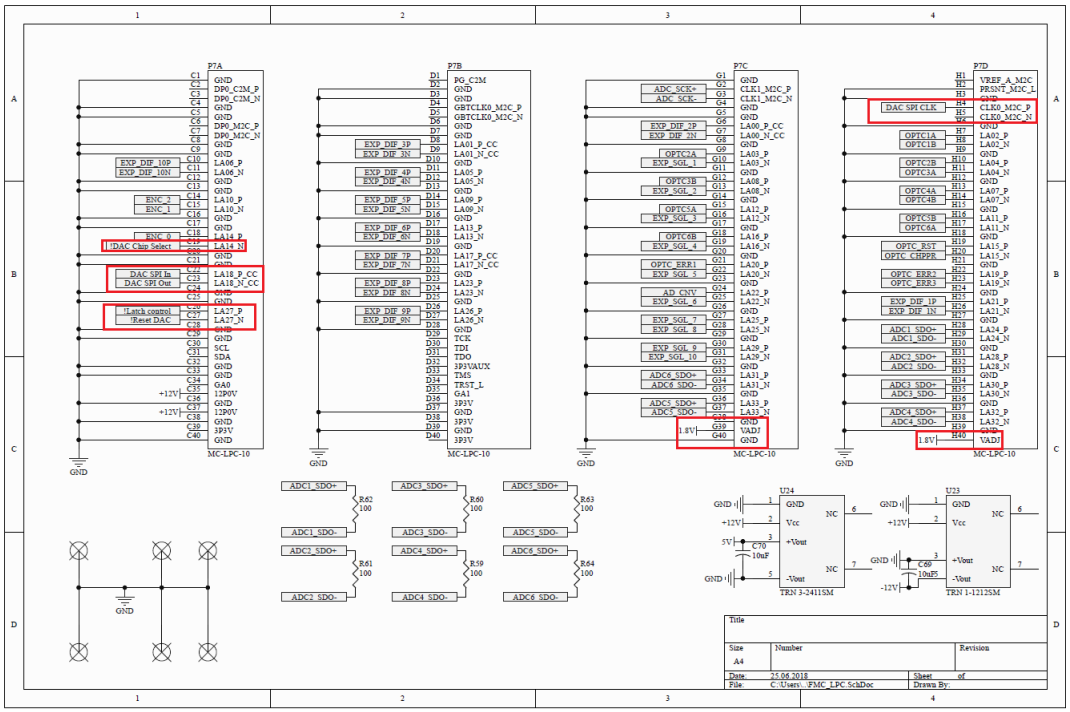

Figure 29: FMC pin layout of DAC chip [10]

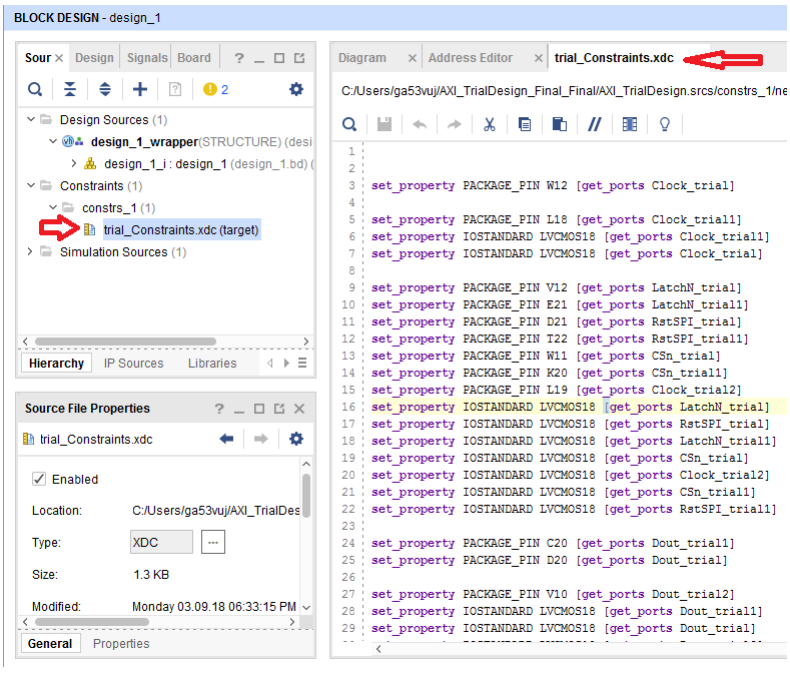

Figure 30: Constraints for mapping output Ports externally

5 Getting the Analog output

After following the design flow discussed in the previous section, the block design is ready to be implemented and run on the Zedboard. The associated steps involve programming the Zedboard using generated bit stream, writing to the PS side of memory (to provide a digital input to the design flow), and then observing the analog output on an oscilloscope. Xylinx provides a software development platform (SDK) which can conveniently perform the first two steps (Refer section 4.1.1 for details). The third step is done using an oscilloscope probe.

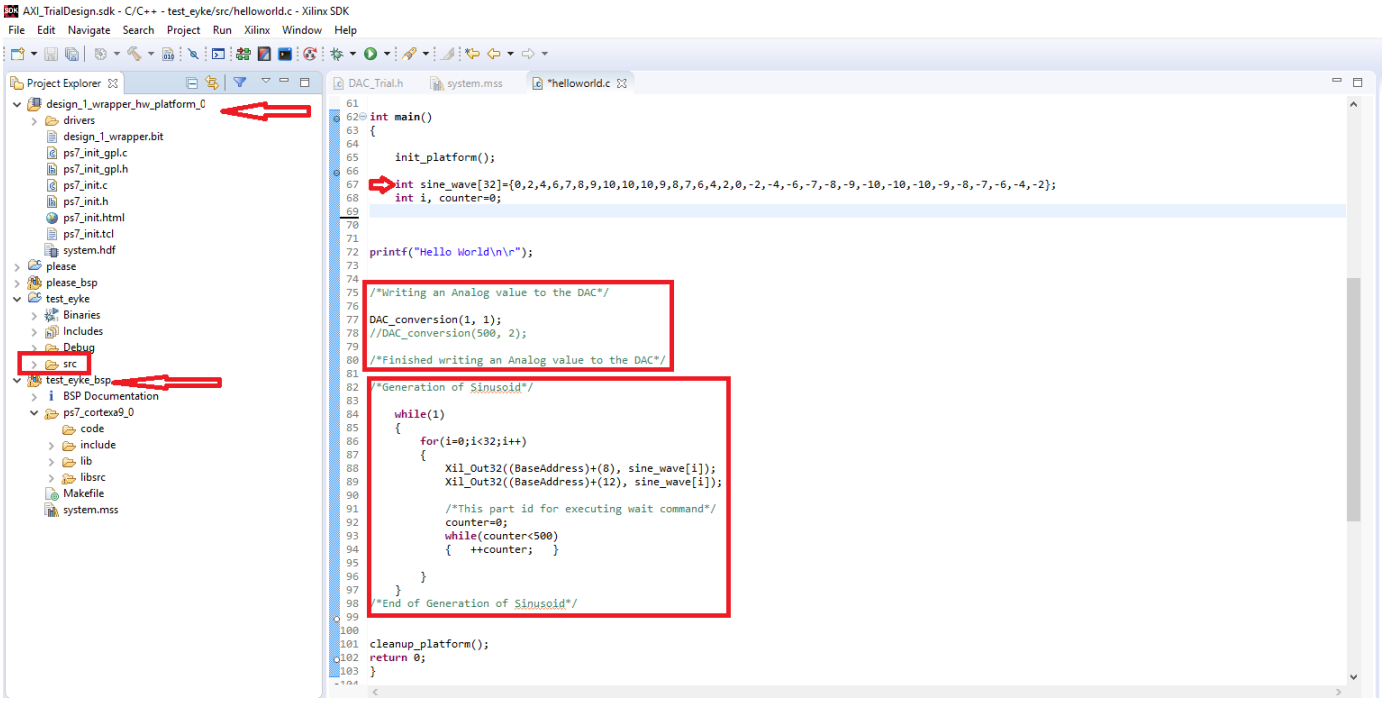

Figure 31: Source Code for Writing to the Memory

5.1 Writing to the Memory

After exporting the hardware from Vivado project window, SDK can be launched from there. This opens a new window as shown in figure 31. The default window, contains the desired configuration files of the Vivado project and the board support packages (BSPs). In order to write to the PS side of Zedboard, the user needs to create a new Application Project, and write the required instructions to be given to the board as source codes (C Language). Creating a new project automatically creates the associated board support packages.

The source code in figure 31 shows the instructions being given to the Zedboard. The functions "XylinxOut32" and "XylinxIn32" are used to write a certain value to or read from a specified address in the memory respectively. The code implements two functionality

1. Generating a sinusoidal wave
   The digital sequence for a sinusoidal wave can be generated using look-up table method.

This involves generating the digital values for different analog values of sine wave and then writing those values sequentially to the memory address of the DAC IP slave register. The digital input sequence is stored in an array and then written to the memory location using a for loop. It should be noted that to continuously generate the sine wave to be observed at oscilloscope, the sequence needs to be given to the DAC chip continuously, thus the code is placed inside an infinite loop.

2. Obtaining an equivalent Analog voltage for any digital/input.
   The function "DAC_Conversion" takes the desired analog input value and DAC output channel number as its argument and then write that value to the address location of the slave register corresponding to the entered channel number. It should be noted that although the slave registers are 32 bits wide, the desired analog value need not necessarily be entered in the hexadecimal form, but can also be provided as an integer. Also, an internal calibration factor (200 in this case) is needed to compensate for the internal attenuation of voltages.

This source code is stored in a "helloworld.c" file created inside "src" section of the new Application project.

5.2 Programming the FPGA

This essentially means configuring the gate arrays of the Zedboard using the bit stream generated as per our block design. SDK provides the user this with functionality using a single button click as highlighted in figure 31. The details can be seen in section 4.1.1. Once the FPGAs have been programmed, the application project needs to be launched/run on the board in "System debugger mode", which finally completes our design flow and generates the output.

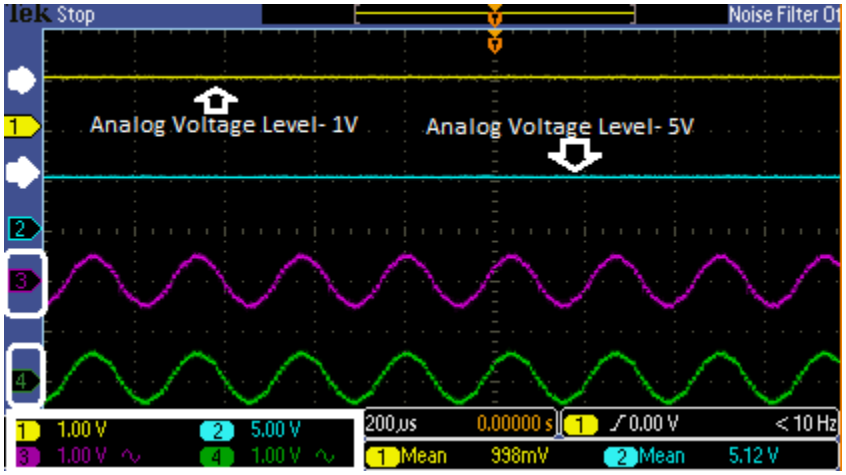

Figure 32: Analog Outputs

5.3 Results

The final result as per the steps described in 5.1 is obtained using an oscilloscope and is shown in figure 32. The function "DAC_Conversion" mentioned in Section 5.1 is called twice to write

analog integer values one and five to the first and second channels of the DAC chip respectively. The output analog voltage can be seen on the first and second oscilloscope output channel shown in figure 32.

The third and fourth channel of oscilloscope shows respectively the sinusoidal signal generated on the Zedboard using look-up table approach shown in figure 31. It should be noted that the generated sinusoidal signals are centered around origin and thus the customized AXI IP works with positive and negative values as well.

6 Conclusion and future work

The DAC protocol was successfully implemented on the FPGA and tested by generating signals in the processor. The implementation was also tested at different clock speeds and it gave satisfactory results. The idea of dividing the computational load on the processor/controller can be taken further to compute and implement switching signals for the converter. Likewise, most of the measuring tasks can be transferred to the FPGA, making more space for complex computations online.

References

[1] Leens, Frédéric. "An introduction to I2C and SPI protocols." IEEE Instrumentation & Measurement Magazine 12.1 (2009): 8-13.

[2] "Quad, 12-Bit, High-Accuracy, ±16V Output, Serial Input DIGITAL-TO-ANALOG CONVERTER." Texas Instruments

[3] "Zynq Architecture - Zynq 7000 Architecture" Xilinx

[4] "I/O Design Flexibility with the FPGA Mezzanine Card (FMC)" Xilinx, Aug 19, 2009

[5] "Processing System 7 Logic Core IP" Xilinx, May 10, 2017

[6] "Processor System Reset Module Logic Core IP" Xilinx, Nov 18, 2015

[7] "AXI Interconnect Logic Core IP" Xilinx, Dec 20, 2017

[8] "Digital System Design using VHDL" Charles H. Roth Jr., Lizzy Kurian John

[9] "Vivado Design Suite User Guide - Creating and Packaging Custom IP" Xilinx

[10] "Design and Implementation of a Mezzanine Card for an FPGA-based Real-Time Control-System", Thomas Kreppel, EAL, TUM, 2018

7 Appendix

CODE

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

--// Entity declarations
entity CsnClk is
    Port ( ClkIp : in STD_LOGIC;
           Clkop : out STD_LOGIC;
           CSn : out STD_LOGIC;
           LatchN : out STD_LOGIC := '1';
           Dout : out STD_LOGIC;
           Rstn: in STD_LOGIC := '0';
           RstSPI: out STD_LOGIC);
end CsnClk;

--// Architecture declarations
architecture GenCSn of CsnClk is

type address is array(0 to 3) of STD_LOGIC_VECTOR(3 downto 0);
type Datasig is array(0 to 16) of STD_LOGIC_VECTOR(11 downto 0);

--// Signal declarations
signal addrs: address;
signal Din: Datasig;
signal Clkop_sig : STD_LOGIC :='0';
signal Csn_sig : STD_LOGIC := '0';
signal LatchN_sig : STD_LOGIC := '0';
signal Do_sig : STD_LOGIC;
signal D2trns : STD_LOGIC_VECTOR(23 downto 0):= (others=> '0');
shared variable CSnLowFlag : INTEGER := 0;

begin
-- // Address of the auxiliary registers
addrs(0)<="0100";
addrs(1)<="0101";
addrs(2)<="0110";
addrs(3)<="0111";

--//Channel 1 output
Din(0) <= "000001010001";
Din(1) <= "000001100100";
Din(2) <= "000001110111";
Din(3) <= "000010000111";
```

```
Din(4) <= "000010010010";
Din(5) <= "000010010110";
Din(6) <= "000010010010";
Din(7) <= "000010000111";
Din(8) <= "000001110111";
Din(9) <= "000001100100";
Din(10) <="000001010001";
Din(11) <="000001000001";
Din(12) <="000000110110";
Din(13) <="000000110010";
Din(14) <="000000110110";
Din(15) <="000001000001";
Din(16) <="000001010001";
--// Channel 2 Output
Din1(0) <= "000000110010";
Din1(1) <= "000000110010";
Din1(2) <= "000000110010";
Din1(3) <= "000000110010";
Din1(4) <= "000000000000";
Din1(5) <= "000000000000";
Din1(6) <= "000000000000";
Din1(7) <= "000000000000";
Din1(8) <= "000000110010";
Din1(9) <= "000000110010";
Din1(10) <="000000110010";
Din1(11) <="000000110010";
Din1(12) <="000000000000";
Din1(13) <="000000000000";
Din1(14) <="000000000000";
Din1(15) <="000000000000";
Din1(16) <="000000110010";
--// Channel 3 Output
Din2(0) <= "000000110010";
Din2(1) <= "000000000000";
Din2(2) <= "000000110010";
Din2(3) <= "000000000000";
Din2(4) <= "000000110010";
Din2(5) <= "000000000000";
Din2(6) <= "000000110010";
Din2(7) <= "000000000000";
Din2(8) <= "000000110010";
Din2(9) <= "000000000000";
Din2(10) <="000000110010";
Din2(11) <="000000000000";
Din2(12) <="000000110010";
Din2(13) <="000000000000";
Din2(14) <="000000110010";
```

```
Din2(15) <="000000000000";
Din2(16) <="000000110010";

--// Channel 4 output
Din3(0) <= "111111111111";
Din3(1) <= "011111111111";
Din3(2) <= "001111111111";
Din3(3) <= "000111111111";
Din3(4) <= "000011111111";
Din3(5) <= "000111111111";
Din3(6) <= "001111111111";
Din3(7) <= "011111111111";
Din3(8) <= "000011111111";
Din3(9) <= "000001111111";
Din3(10) <="000000111111";
Din3(11) <="000000011111";
Din3(12) <="000000111111";
Din3(13) <="000001111111";
Din3(14) <="000011111111";
Din3(15) <="000111111111";
Din3(16) <="001111111111";
-- // Matching signals to corresponding pin outs
Clkop<=Clkop_sig;
CSn <= Csn_sig;
LatchN<= LatchN_sig;
Dout <= Do_sig;
RstSPI<= Rstn;

--// Clock generation process
GenClock: process(ClkIp)
    constant div_by : integer := 4;
    variable count_clk : integer:= 0;
begin
    if (Rstn = '0' AND ClkIp'event AND ClkIp = '1') then
            count_clk := count_clk+1;
            if count_clk = div_by then
                Clkop_sig <= Clkop_sig XOR '1' ;
                count_clk := 0;
            end if;
     end if;
end process;

-- // Chip select generation
SwitchCSn : process(Clkop_sig)
    constant bit24 : integer :=25;
    variable count_bit : integer:= 0;
    variable Csn_flag : integer :=0;
```

```
begin
    if (Clkop_sig'event AND Clkop_sig = '1') then
            if Csn_flag = 1 then
                Csn_sig<= '0';
                CSnLowFlag := 1;
            end if;
            if count_bit = bit24 then
                Csn_sig <= '1' ;
                CSnLowFlag:= 0;
                count_bit := 0;
                Csn_flag := 1;
            end if;
             count_bit := count_bit+1;
     end if;
end process;

--// Latch generation process
GenLATCH : process(Clkop_sig)
    constant RegNo : integer :=1;
    constant bit24 : integer :=25;
    variable count_reg : integer:= 0;
    variable LatchN_flag : integer :=0;
begin
    if (Clkop_sig'event AND Clkop_sig = '0') then
            if LatchN_flag = 1 then
                LatchN_sig<= '1';
            end if;
            if count_reg = RegNo*bit24 then
                LatchN_sig <= '0' ;
                count_reg := 0;
                LatchN_flag := 1;
            end if;
            count_reg := count_reg+1;
     end if;
end process;

--//updating the data out register
updateD2Trns: process(Csn_sig)
     constant ChannelNo : integer := 4;
    constant DataStrLen : integer:=16;
    variable count_channel : integer := 0;
    variable count_datastr: integer:= 0;
    variable count_datastr1: integer:= 0;
    variable count_datastr2: integer:= 0;
    variable count_datastr3: integer:= 0;
begin
    if (Csn_sig'event AND Csn_sig = '1' ) then
```

```
            D2trns(20 downto 17) <= addrs(count_channel);
            SEL := addrs(count_channel);
            case SEL is
                when "0100" => D2trns(16 downto 5) <= Din(count_datastr);
                       count_datastr := count_datastr +1;
                when "0101" => D2trns(16 downto 5) <= Din1(count_datastr1);
                       count_datastr1 := count_datastr1 +1;
                when "0110" => D2trns(16 downto 5) <= Din2(count_datastr2);
                       count_datastr2 := count_datastr2 +1;
                when "0111" => D2trns(16 downto 5) <= Din3(count_datastr3);
                       count_datastr3 := count_datastr3 +1;
                when others => NULL;
            end case;
            count_channel := count_channel+1;
            if count_channel = ChannelNo then
                count_channel := 0;
            end if;
            if count_datastr = DataStrLen then
                count_datastr:= 0;
            end if;
            if count_datastr1 = DataStrLen then
                 count_datastr1:= 0;
            end if;
            if count_datastr2 = DataStrLen then
                  count_datastr2:= 0;
            end if;
            if count_datastr3 = DataStrLen then
                  count_datastr3:= 0;
            end if;
        end if;
    end process;
    --// Data transmission
    DataOut: process (Clkop_sig)
        constant data_length : integer :=23;
        variable count_bit : integer :=0;
    begin
        if (Clkop_sig'event AND Clkop_sig = '1' AND CSnLowFlag =1) then
            Do_sig <= D2trns(data_length-count_bit);
            count_bit:= count_bit+1;
            if count_bit = data_length+1 then
                count_bit := 0;
            end if;
        end if;
    end process;
    end GenCSn;
```