Lab 8: FPGA/DAC

UC Davis Physics 116B
Rev 3/7/2019.2

Introduction

In this lab, you will use the skills you've acquired with VHDL and the Spartan 3E development board to experiment with the on-board digital to analog converter (DAC).

Pre-lab Exercises

For this lab, we will use the quad DAC on the Spartan 3E development board. This DAC has four 12-bit DACs, two of which have V_{ref}=3.3V and two of which have V_{ref}=2.5V. The output voltage of each DAC will be

$$\begin{aligned} V_{out} &= \frac{D}{2^{12}} V_{ref} \\ &= \frac{D}{4096} V_{ref} \end{aligned}$$

For each reference voltage, calculate the voltage change associated with the least significant bit (LSB), as well as the voltage change associated with each of the four most significant bits [11-8].

Files Provided for this Lab

You will be provided with the "DAC_SET.vhd" module to convert parallel words into the serial data to program the on-board DAC, which we discussed in detail in class. You will also be provided with the "pins.ucf" file to constrain the pin positions[1]. These files are located in the `Files/Labs/Lab_8 Files` area of the course Canvas page, and they will also be available on USB stick.

Listings of these two files are included in the Appendix of this document. Be sure you understand them.

Spartan 3E On Board Quad DAC

Figure 1 shows the location of the LT2624 quad DAC, its associated output header, and a schematic representation of its operation. The chip contains 4 12-bit DACs, two of which have V_{ref}=3.3V

[1]There are lots of pins to connect, and you've already demonstrated you know how to do this.

and two of which have V_{ref}=2.5V. Each DAC may be set individually to a different value of D, or all DACs may be setting simultaneously to the same value of D.

The DAC chip is configured by means of an active low chip select (DAC_ CS), a serial data line (SPI_ MOSI), and a serial data clock (SPI_SCK). In addition, an active low reset (DAC_CLR) must be kept high for normal operation! The capability exists to read back the configuration data through the SPI_ MISO pin, but we will not do this.

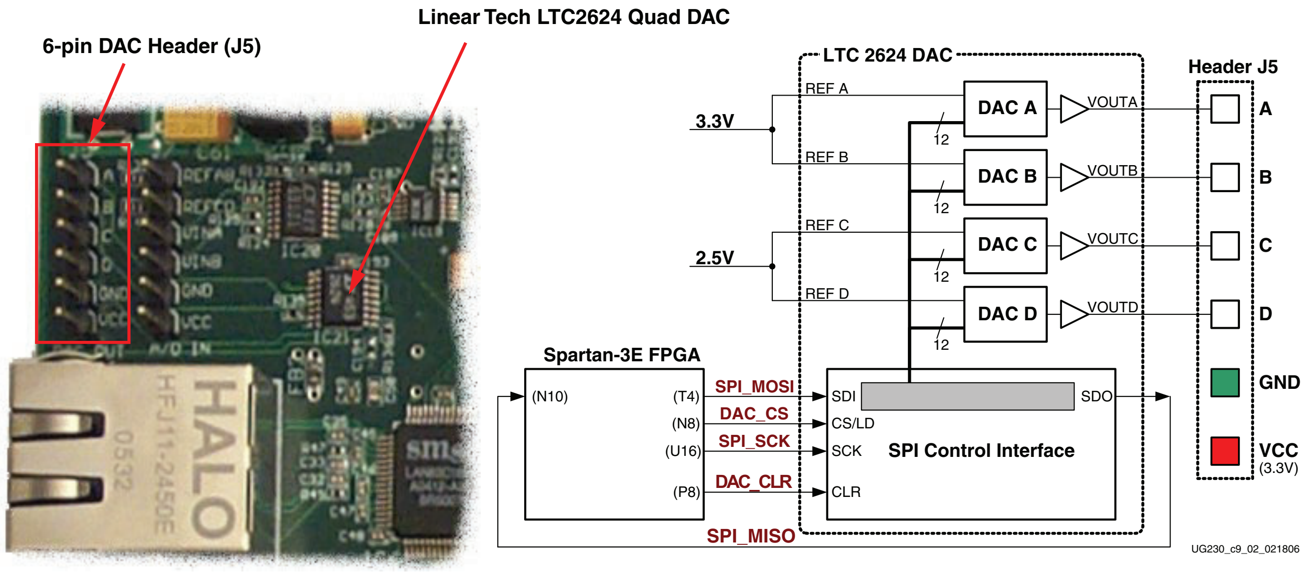


Figure 1: Location of on-board DAC and output header on Spartan 3E Starter Board, including schematic representation their operation.

You are being provided with the "DAC_SET.vhd" module to perform the actual configuration of the DAC, but it's important to understand its operation, which we discussed in class.

Table 1: Bit definitions within the 32-bit DAC configuration word.

bits	**Use**	**Comment**
3-0	–	Don't care
15-4	DAC	12-bit DAC setting
19-16	ADDRESS	"0000" $\rightarrow$ set DAC A "0001" $\rightarrow$ set DAC B "0010" $\rightarrow$ set DAC C "0011" $\rightarrow$ set DAC D "1111" $\rightarrow$ set ALL
23-20	COMMAND	We will use COMMAND="0011", which will promptly set the DAC and enable it at the end of the write cycle
31-24	–	Don't care

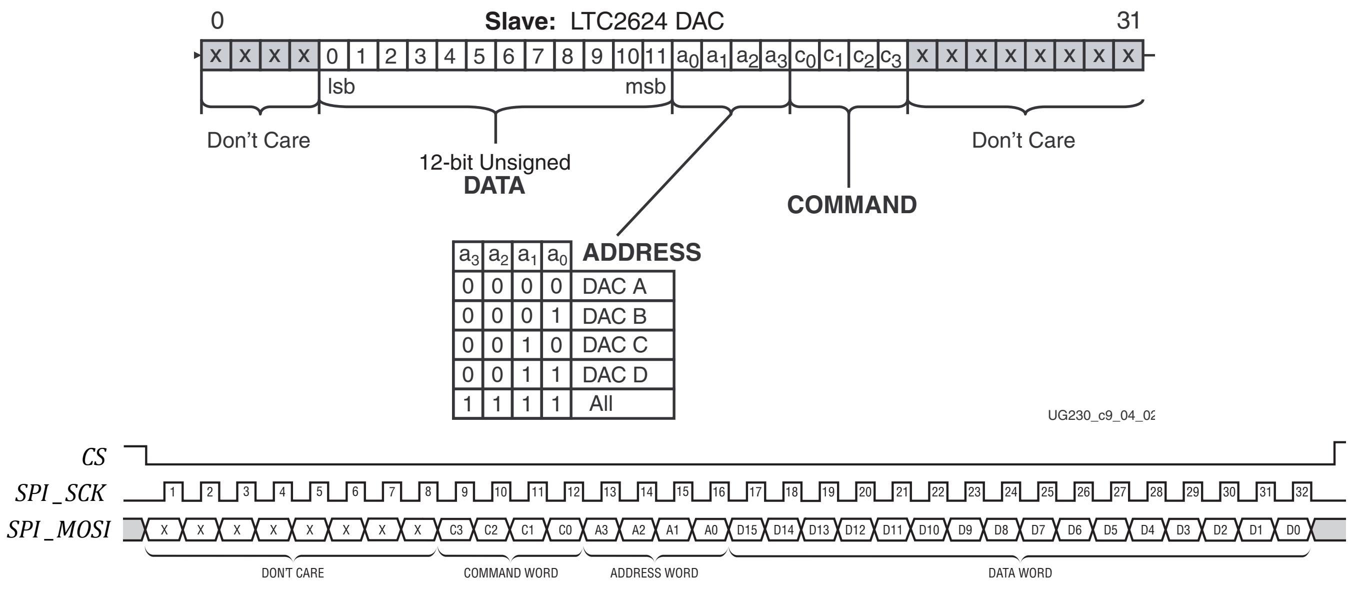


Figure 2: DAC configuration encoding and time line.

The DAC is configured by means of a 32-bit configuration word, defined in Table 1. Figure 2 illustrates this packed word, as well as the programming sequence. DAC_CS goes low, and the 32 bits of the configuration word are asserted onto SPI_ MOSI, starting with the MSB. While each bit is valid, a rising edge on SPI_SCK clocks it into the DAC chip. After 32-bits have been loaded into the DAC, DAC_CS goes high again. We will always use COMMAND="0011", which will cause the selected DAC to immediately go to the set value at this point.

VHDL Code

Start a new project and define a new VHDL module with the following inputs and outputs:

```
entity LAB8 is
Port (CLK : in  STD_LOGIC;                          -- On board 50 MHz clock
      SW  : in  STD_LOGIC_VECTOR (3 downto 0);  -- On Board slide switches
      LED : out  STD_LOGIC_VECTOR (7 downto 0); -- On Board LEDs
      DAC_CS : out  STD_LOGIC;                      -- Chip select to DAC
      SPI_MOSI : out  STD_LOGIC;                    -- Serial Data bit to DAC
      SPI_SCK : out  STD_LOGIC;                     -- Serial Clock to DAC
      DAC_CLR : out STD_LOGIC;                      -- DAC clear.  MUST BE SET TO '1'!
      J1 : out STD_LOGIC_VECTOR (3 downto 0));  -- General purpose diagnostic pins
end LAB8;
```

You have been provided with the "DAC_SET.vhd" module, but you must prototype it with the following component specification before the “begin” line of the architecture block[2]:

```
component DAC_SET is
    Port ( CLK : in  STD_LOGIC;
           DAC : in  STD_LOGIC_VECTOR (11 downto 0);
           ADDRESS: in  STD_LOGIC_VECTOR (3 downto 0);
           COMMAND : in  STD_LOGIC_VECTOR (3 downto 0);
           SET : in STD_LOGIC;
           BUSY : out  STD_LOGIC;
           DAC_CS : out  STD_LOGIC;
           MOSI : out  STD_LOGIC;
           SCK : out  STD_LOGIC);
end component;
```

In this area, you must also define the DAC, ADDRESS, COMMAND, SET, and BUSY signals, with the appropriate sizes. The remaining signals from DAC_SET *could* be tied directly to corresponding pins of the main module, but as we’ll discuss below, we want to tie them to two different outputs, so you should also create the inermediate STD_LOGIC signals CS,MOSI, and SCK.

In the concurrent section of the code (after “begin”), you can instantiate the DAC_SET module with

```
D: DAC_SET port map(CLK,DAC,ADDRESS,COMMAND,SET,BUSY,CS,MOSI,SCK);
```

You’ll then need to tie CS, MOSI, and SCK to DAC_CS, SPI_MOSI, and SPI_SCK, respecively.

Because the pins on the DAC chip are very small, it’s difficult to look directly at the configuration signals with an oscilloscope, so in addition to the connections above we’ll tie the following four signals to the four elements of the J1 vector as well: SET, CS, MOSI, and SCK. This will allow us to monitor them at the J1 connector on the board, shown in Figure 3.

Now make the following connections:

- Set DAC_CLR to ’1’ to prevent the DAC from going into a permanently reset state[3].
- Set COMMAND to “0011”. This will cause the DAC output to go to the set value immediately when DAC_CS goes high. Other values would allow it to be, for example, set in one cycle, and then turned on later.
- Tie LED[3:0] to the ADDRESS[3:0].
- The LED[7:4] to DAC[11:8] (four most significant bits) of DAC setting.

[2]It might save time to copy the port() part of this directly from the “DAC_SET.vhd” file.

[3]The author learned this the hard way.

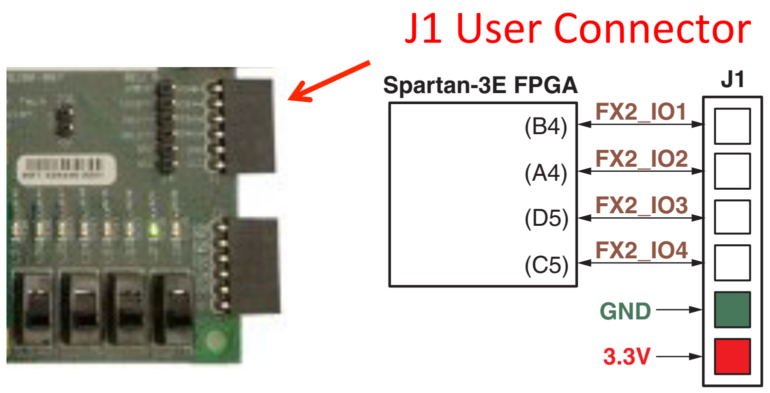


Figure 3: Location, pinouts, and FPGA mapping of the J1 general purpose header.

Once you have created your project and written the outline of your code, you should copy the "DAC_SET.vhd" and "pins.ucf" files to the project directory and "Add Source..." them to your main VHDL file.

We will now set the DAC in two different ways. In the first part, we will set all the DACs to a constant value by using the switches to set the highest order bits. In the second part, we will generate a sawtooth wave by incrementing the DAC setting by one each time we set the DAC.

As an optional third part, you may modify the code to generate a triangle wave instead of a sawtooth wave.

Setting the DACs to a Constant Value

For this section, tie DAC[7:0] to "00000000", and DAC[11:8] to the four slider switches. This effectively turns our 12-bit DACs into 4-bit DACs.

Next, create a process which will test on the rising edge of CLK, and inside that IF block, test on the BUSY and SET signals. If BUSY='0' and SET='0', assert the SET bit. In other words, set the DAC as often as you can.

Compile the code, generate the bit file, and program it to the FPGA. Verify that the four lower LEDs are all ON (ADDRESS="1111"), and that the slide switches are reflected in the LED[7:4] (DAC[11:8]).

Now use the scope to look at the configuration signals on the J1 header (see Figure 3). You can ground the probes to the "gnd" pins provided on the board. First trigger on the DAC_CS pin going high (i.e. the end of the configuration cycle), and also look at the SET pin. If you wrote your code correctly, you should see the DAC_CS pin going high for two 50 MHz clock cycles, with the SET pin going high for the second. Verify that this is the case, and for your writeup, draw a timeline of the (expected) CLK, SET, BUSY, and DAC_CS signals to explain why this is so. Your writeup should also include an image of the scope traces.

Now trigger on the DAC_CS pin going low, and set the time scale so that you can see its entire length. The configuration should take 1 clock cycle to set up, followed by 32 serial bits, each of which takes 2 clock cycles, for a total of 65 50 MHz clock cycles. Verify that this is the length of time the DAC_CS signal remains low.

Using a second scope probe, look at the SPI_MOSI data pin. Because COMMAND="0011" and ADDRESS="1111", you should see six sequential bits always high, while manipulating the switches should change the state of the four bits immediately after those, and all other bits should be 0. Start with all switches set to '0', then turn them on one at a time and observe whether the SPI_MOSI bit behaves as expected. Include pictures of scope traces with at least two different switches set.

If all of this looks correct, use the DVM to measure the voltage between ground and the DAC output pins at header J5 (see Figure 1). Because ADDRESS="1111", changing any of the switch settings should change the output of all four DACs. Verify that this is so.

Now, looking just at the output of DAC A ($V_{ref} = 3.3$V), measure the amount the voltage changes when you change the state of each of the four switches, and confirm that the change matches your predictions from the pre-lab exercises. Your lab write-up should include your predicted and observed values.

Generating a Sawtooth Wave

In this section, we'll replace the constant DAC setting with a setting that ramps through all 2^{12} possible settings, and then loops around, generating a sawtooth wave.

First, make the following changes in the concurrent section of your code:

- Eliminate the assignment of the lower 8 DAC bits to "000000".
- Eliminate the connection of the higher 4 DAC bits to the switches.
- Eliminate the assignment of ADDRESS to "1111" and instead tie the ADDRESS bits to the switches.

In the process section, create a new variable (between "process" and "begin") called "dac_value" that is an integer with a range of 0 to 4095, which is initialized to 0. Tie this to the DAC bits with

```
DAC <= std_logic_vector(to_unsigned(dac_value,12));
```

and increment it each time you assert the SET signal. Compile the code, generate the bit file and program it to the FPGA.

Turn all switches ON and verify that all four DACs are producing sawtooth waves. Verify that the maximum values are what you expect for the AB and CD pairs, based on their respective values for V_{ref}.

Next, make just one DAC at a time ramp but setting the switches to the values indicated in Table 1, and verify that none of the others change when you do this.

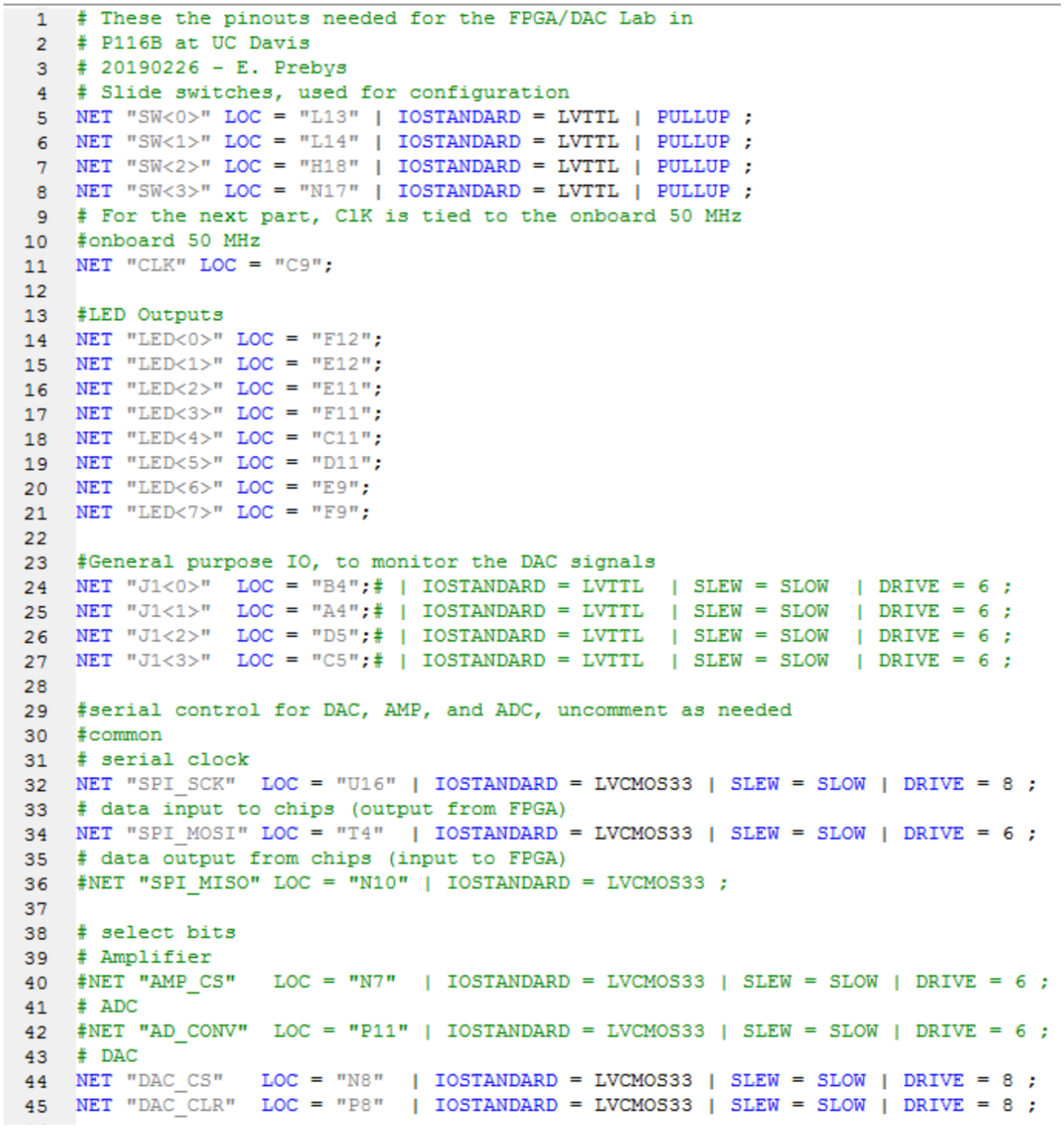
In the last section, you (hopefully) showed that each SET cycle took 65+2=67 cycles of the 50 MHz clock. Based on this, calculate the expected period of the sawtooth wave; i.e. the time it takes to go through all 4096 values. Verify this matches the period you observe on the scope.

For your lab writeup, include your code and the scope trace of the sawtooth wave.

OPTIONAL: Triangle Wave

If you have time, modify your code to produce a triangle wave instead of a sawtooth wave, by counting up through all values, then counting down through all values.

Appendix: Pin-outs and DAC_SET.vhd Routine



```
# These the pinouts needed for the FPGA/DAC Lab in
# P116B at UC Davis
# 20190226 - E. Prebys
# Slide switches, used for configuration
NET "SW<0>" LOC = "L13" | IOSTANDARD = LVTTL | PULLUP ;
NET "SW<1>" LOC = "L14" | IOSTANDARD = LVTTL | PULLUP ;
NET "SW<2>" LOC = "H18" | IOSTANDARD = LVTTL | PULLUP ;
NET "SW<3>" LOC = "N17" | IOSTANDARD = LVTTL | PULLUP ;
# For the next part, ClK is tied to the onboard 50 MHz
#onboard 50 MHz
NET "CLK" LOC = "C9";

#LED Outputs
NET "LED<0>" LOC = "F12";
NET "LED<1>" LOC = "E12";
NET "LED<2>" LOC = "E11";
NET "LED<3>" LOC = "F11";
NET "LED<4>" LOC = "C11";
NET "LED<5>" LOC = "D11";
NET "LED<6>" LOC = "E9";
NET "LED<7>" LOC = "F9";

#General purpose IO, to monitor the DAC signals
NET "J1<0>"  LOC = "B4";# | IOSTANDARD = LVTTL  | SLEW = SLOW  | DRIVE = 6 ;
NET "J1<1>"  LOC = "A4";# | IOSTANDARD = LVTTL  | SLEW = SLOW  | DRIVE = 6 ;
NET "J1<2>"  LOC = "D5";# | IOSTANDARD = LVTTL  | SLEW = SLOW  | DRIVE = 6 ;
NET "J1<3>"  LOC = "C5";# | IOSTANDARD = LVTTL  | SLEW = SLOW  | DRIVE = 6 ;

#serial control for DAC, AMP, and ADC, uncomment as needed
#common
# serial clock
NET "SPI_SCK"  LOC = "U16" | IOSTANDARD = LVCMOS33 | SLEW = SLOW | DRIVE = 8 ;
# data input to chips (output from FPGA)
NET "SPI_MOSI" LOC = "T4"  | IOSTANDARD = LVCMOS33 | SLEW = SLOW | DRIVE = 6 ;
# data output from chips (input to FPGA)
#NET "SPI_MISO" LOC = "N10" | IOSTANDARD = LVCMOS33 ;

# select bits
# Amplifier
#NET "AMP_CS"   LOC = "N7"  | IOSTANDARD = LVCMOS33 | SLEW = SLOW | DRIVE = 6 ;
# ADC
#NET "AD_CONV"  LOC = "P11" | IOSTANDARD = LVCMOS33 | SLEW = SLOW | DRIVE = 6 ;
# DAC
NET "DAC_CS"   LOC = "N8"  | IOSTANDARD = LVCMOS33 | SLEW = SLOW | DRIVE = 8 ;
NET "DAC_CLR"  LOC = "P8"  | IOSTANDARD = LVCMOS33 | SLEW = SLOW | DRIVE = 8 ;
```

```vhdl
-----------------------------------------------------------------------------------
-- Company: UC Davis
-- Engineer: Eric Prebys
--
-- Create Date:    10:55:47 02/06/2019
-- Design Name:    DAC_SET
-- Module Name:    DAC_SET - Behavioral
-- Project Name:   UC Davis P116B FPGA Lab 3
-- Target Devices: xc3s500e-4fg320
-- Tool versions:  ISE 14.7
-- Description:
--   Will set the Linear Tech LTC2624 Quad DAC 12-bit DAC on the
--   Spartan 3E starter board.  It will take 65 clock cycles, following
--   the synchronous assertion of the SET pin.
--
--   IMPORTANT NOTE: In addition to the pins driven by this component,
--   the DAC_CLR pin (pin P8 on Xilinx Chip) must be driven HI to
--   prevent the DAC from remaining in a RESET state!
--
-- Inputs:
--   CLK            - 50MHz onboard clock
--   DAC        - 12-bit DAC setting
--   ADDRESS    - 4-bit address word (0-3-> DAC#, 15-> ALL)
--   COMMAND    - 4-bit command word (usually "0011")
--   SET        - begin set process
-- Ouputs:
--   BUSY   - setting in progress (active HI)
--  (tie the following directly to the DAC)
--   DAC_CS - amp chip select     (active LOW)
--   MOSI   - serial data bits
--   SCK    - serial clock, clocked at 1/2 of CLK speed. Ambiently LOW, so
--            it can be ORed with other SCK drivers
-- Dependencies: NONE
--
-- Revision:
-- Revision 0.01 - File Created 2/06/2019 - E.Prebys
--                 Based on AMP_SET
--          1.00   2/07/2019 - E.Prebys
--                       Working version.  Cleaned up and commented.
-- Additional Comments:
--
-----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
```

```
entity DAC_SET is
    Port ( CLK : in  STD_LOGIC;
           DAC : in  STD_LOGIC_VECTOR (11 downto 0);
           ADDRESS: in  STD_LOGIC_VECTOR (3 downto 0);
           COMMAND : in  STD_LOGIC_VECTOR (3 downto 0);
           SET : in STD_LOGIC;
           BUSY : out  STD_LOGIC;
           DAC_CS : out  STD_LOGIC;
           MOSI : out  STD_LOGIC;
           SCK : out  STD_LOGIC);
end DAC_SET;

architecture Behavioral of DAC_SET is

-- This is a 32 bit word that encodes the DAC, ADDRESS, and COMMAND words
signal SETWORD: STD_LOGIC_VECTOR (31 downto 0);

begin

-- Pack the individual words into one 32 bit setword (see manual)
SETWORD(3 downto 0) <= "0000";           -- Don't care
SETWORD(15 downto 4) <= DAC;                -- 12-bit DAC setting
SETWORD(19 downto 16) <= ADDRESS;      -- 4-bit ADDRESS (0-3=DAC#, 15=ALL)
SETWORD(23 downto 20) <= COMMAND;       -- 4-bit COMMAND (usually "0011")
SETWORD(31 downto 24) <= "00000000";    -- Don't care

process(CLK,SET)
-- This state machine is ambiently in the IDLE state, with
-- DAC_CS HI, BUSY LOW, and SCK LOW
--
-- In response to a SET, it will assert a BUSY HI and DAC_CS LOW
-- It will then go through 32 2-CLK cycles of LOW,HI,
-- referring to the SCK state.  the appropriate DAC
-- bit will be asserted on MOSI at state LOW and held through
-- HI, starting with the MSB. After 32 bits, it will return to IDLE.
--
type state_type is (IDLE,LOW,HI);
variable state: state_type;

variable bit_index: integer range 0 to 31 := 31;
begin
  if rising_edge(CLK) then
     case state is
          when IDLE =>              -- Ambiently keep SCK and BUSY low, DAC_C
             SCK <= '0';
```

```
              BUSY <= '0';
              DAC_CS <= '1';
             if SET = '1' then     -- Set BUSY high, DAC_CS low, initialize b
                  BUSY <= '1';
                  DAC_CS <= '0';
                bit_index := 31;    -- MSB first
                 state := LOW;
              end if;
          when LOW =>                    -- SCK low
             SCK <= '0';
              state := HI;
             MOSI <= SETWORD(bit_index);
          when HI =>                     -- SCK edge clocks in data
             SCK <= '1';
              if bit_index > 0 then
                 state := LOW;
                 bit_index := bit_index -1;
              else
                 state := IDLE;
              end if;
     end case;
  end if;

end process;


end Behavioral;


```