EE 5143 – M9 Lab – Dr Rahman

You will notice that your DE0-Nano board is densely studded with surface mount components. In addition to the large black centrally-placed Cyclone IV FPGA IC there are a great many other components there for various support functions. These include voltage regulators, crystal oscillator, switch de-bouncing IC etc. In addition to these, there are also a number of other chips present (both on the front and back of the board) that further extend the capabilities of the board as an application development platform. These include a set of synchronous dynamic random access memory (SDRAM) chips for (volatile) data storage, an electrically-erasable programmable read only memory (EEPROM) chip for (non-volatile) data storage, an analog-to-digital converter (ADC) chip for digitization of analog voltage signals and a micro-electromechanical system (MEMS) 3-axis digital-output accelerometer for acceleration sensing in one, two or three mutually orthogonal axes. These devices present interesting possibilities for more advanced system designs utilizing the DE0-Nano board. In this week's lab, we learn how to use one of the above-mentioned devices – the NS ADC128S022, 8-channel, 12-bit analog-to-digital converter. We are going to try out some VHDL code that will make use of this IC to digitize analog voltages and display the digital data on the eight LEDs available on the board. The kind of VHDL code needed to accomplish this task is significantly more complex than what we have encountered so far in this course. Writing this type of code parallels writing device driver programs in software programming languages. Coding device drivers is considered an advanced task because not only one needs a good command over the syntax and semantics of the coding language but also a good understanding of the architecture and functioning of the device for which the driver is being written. While software device drivers are written by specialist programmers, when it comes to doing the same thing in VHDL, we as hardware engineers, cannot escape that easily. The ability to write VHDL device drivers is almost taken for granted with all professional HDL coders. Thus, we must learn how to do this important task and that is the purpose of this lab. While it may be somewhat involved, the ability to write device drivers opens the doors to using the various chips listed above in useful and interesting applications. With this capability you move beyond just being able to use the switches and LEDs on the DE0-Nano board. More expensive FPGA development boards come with an even bigger assortment of useful ICs, using which requires being able to write device drivers. You can also add your own external hardware to your board using the expansion headers present on the DE0-Nano PCB and then you will need to write some drivers for whatever you are adding to the board externally.

While writing VHDL code to control an IC, such as the ADC chip on your board, may be somewhat complicated, by now you know sufficient VHDL that you can be exposed to such code. The main reason for the complexity of device driver code is the requirement for a number of separate synchronized signals to drive functional ICs. In order to reduce pin count, these devices make use of serial data transfer which requires all commands and data to be exchanged with the device using properly timed logic transitions. These ICs are very unforgiving when it comes to signal stream timing requirements – every bit stream must be perfectly ordered and timed in accordance with chip manufacturer's specifications for it to make sense to the device. This means that all processes have to be coded very carefully and these usually involve elaborate nested if-then-else statements to construct bit streams that are fed to the target IC. However, once you understand how to write one device driver for a serial interface device, it becomes easy to write other similar drivers for other devices.

The starting point for this kind of coding exercise is always to begin by studying the datasheet of the device for which the driver is to be written. The relevant datasheet accompanies this lab document but note that any IC's datasheet can be easily obtained for free from the manufacturer's website. These datasheets can be quite long. Our ADC's datasheet, for example, is 20 pages long. In other cases, the length can exceed 100 pages. Luckily, for writing device drivers you needn't go through the entire document as you are only concerned with the programming model for the device. To begin, read pages 1, 2 and 3 of the datasheet now to gain a high-level understanding of this device. Also, take a look at figure 8 on page 16 to see how the ADC is used with a controlling (master) device. Figure 8 shows a microprocessor or a digital signal processor (DSP) but in our case it is an FPGA. A Useful tip: As you go through this lab, keep this document, the ADC datasheet and the DE0-Nano manual all three open on your computer so you can switch back and forth, as needed below.

On the DE0-Nano board, the ADC's pins are already connected to specific FPGA pins which we'll use in our pin assignments. The details of these connections are in section 3.6 on pages 20, 21 and 22 of the DE0-Nano user manual. You should read this section immediately after reading the first three pages of the ADC datasheet. Figure 3-9 in the user manual shows a block diagram of interconnections between the ADC and the FPGA. The two are connected with four single-bit digital lines: CS_n (chip select, active low), DIN (serial data in), DOUT (serial data out) and SCLK (serial clock). The signal directions here are specified from the viewpoint of the ADC. CS_n goes into the ADC to select (make active) the device, DIN is the serial data stream that needs to be fed to the ADC (the only purpose here is to tell the ADC which one of the possible eight analog input pin's voltage is to be digitized), DOUT is the serial 12-bit digital data that comes out of the ADC. SCLK is a clock signal that the ADC needs to work (this is used for timing the DIN and DOUT serial bit streams). The master device (FPGA in this case) generates and provides SCLK signal to the ADC. There are eight input pins on the ADC to receive an analog voltage for conversion. These pins are connected to a small 2x13 pin header on the back side of your board. The eight input channels are available on pins on this header as shown in figure 3-8 on page 21 of the manual. The channel pins are named as Analog_In0 to Analog_In7. The ADC can digitize signal present on only one of these pins at a time and you can tell it the channel number (0 to 7) through the serial data you feed to the DIN input. You can set your channel of choice by setting three of the four slide switches on the DE0-Nano board. After each conversion, 12-bit digital data comes out of the DOUT output serially. If you only need 8-bit precision then just throw away the 4 least significant digits of this data. This then is basically the idea behind using the ADC device on your board.

To use the ADC in practice, multiple signals have to be exchanged in a set order. This is indicated in the all-important timing diagram on page 7 of the datasheet. The timing diagram is the most important information you need to write your VHDL code. In fact, your VHDL code is a translation of the timing sequence shown in the diagram to VHDL code. This particular way of exchanging serial data between devices is known as the Serial Peripheral Interface (SPI), which was originally developed by Motorola. This four-wire (CS_n, SCLK, DIN, DOUT) interface can connect chips together so that they can communicate data with other in a serial fashion. In this kind of interface, one device is always a master whereas one or more devices connected to it are slaves. In our case, the FPGA acts as the master device while the ADC is a slave.

Let's go through figure 1 timing diagram in the ADC datasheet from the top to bottom. As the diagram shows, in order to use the ADC you start by activating the chip select pin (by pulling it

down from the resting state of logic high to active state of logic low). Next, you start feeding a clock signal to the ADC (notice that in the absence of the clock the logic level at the clock pin rests at logic high). The clock in this case will be a 2.5 MHz square waveform so you need a PLL to generate this ADC clock. Then you send in serial data through DIN to specify which input channel to digitize. Once the ADC begins to receive its clock you can send in three bits to specify which of the eight channels (0 to 7) will be digitized in the current conversion cycle (in subsequent conversion cycles you can keep the input channel the same or switch to a different channel as you wish). Immediately after that, the ADC begins to send 12 bits out through its DOUT output (most significant bit, bit 11 first and least significant bit, bit 0 last). You need to accept it and do something with this data. This is how the ADC is used during one conversion cycle. Note that in our code the value for channel-to-be-digitized comes from the settings of three slide switches on the DE0-Nano board but in some other case this value can come from any internal or external source. You can also arrange things so that some or all channels are automatically scanned at a set rate etc. It is all in the code you write.

What you need to actually use the converter is VHDL code for ADC-controller which will provide the ADC with all the signals that it needs and will also accept the serial output data from it. This lab demonstrates such an ADC-controller core. Additionally, our core will also read three of the four slide switches on the board through which we can specify the channel we want digitized. Furthermore, it will also read the fourth slide switch to get an indication of what to do with the digitized data. In this case, we'll either dump 8-bit ADC data (we get this by chopping off 4 least significant bits of the ADC output data) on the eight LEDs to get a visual indication of the binary data produced by the ADC or show the data in a more intuitive way as a bar graph display on the LEDs (so that more LEDs are lit for higher voltages than for lower voltages). Our code here contains both ADC-control and display functions. More generally, an ADC-control core will not have the display part in it as it will simply pass on the ADC output data to another code block for further processing. You can easily remove the display parts of the code to write yourself such an ADC-control-only core, if you desire. You could then cascade your ADC-controller core to other blocks such as one to appropriately scale the data, to filter it digitally, or to display it on an LCD or multiplexed 7-segment display etc.

At this point, it is instructive to take a look at the entity declaration for our ADC-controller.

```
entity ADC is
   port(
      clk_50MHz : in std_logic;  -- External 50 MHz clock
      channel_val : in std_logic_vector(2 downto 0); -- SW0,SW1, SW2 select analog-in channel
      display_mode : in std_logic; -- Select binary (1) or bar-graph (0) LED display from SW3
      data_in : in std_logic; -- Accept serial data from the ADC. For DIN
      channel : out std_logic; -- Send analog-in channel number serially to the ADC. For DOUT
      clk_signal : out std_logic := '1'; -- Serial clock to the ADC. For SCLK
      cs_signal : out std_logic := '1'; -- Chip select to the ADC. For CS_n (CS bar)
      LED_out : out std_logic_vector(7 downto 0) -- 8 LED display outputs to the DE0-Nano board
           );
end entity;
```

With the description given earlier and the in-line comments above, it should be easy to see what the different ports in this entity do. Note that the data_in signal will connect to the DOUT pin on

the ADC (data coming 'in' to the controller core is actually the data that comes 'out' of the ADC thus its mode is given as 'in'). channel signal will connect to the DIN pin of the ADC. Clk_signal signal will connect to the SCLK pin on the ADC (our controller will act as the clock master – it will generate and provide a clock to the ADC). cs_signal will connect to the CS bar pin on the ADC. The other signals connect to the DE0-Nano clock oscillator and the various switches and LEDs on the board. Notice that clk_signal and cs_signal both have been declared as having an initial value of logic high. This has been done so that the ADC sees the proper (resting) logic levels when it is first started. This only matters in the beginning. Afterwards, the code takes care of the appropriate logic levels on all pins at all times.

To understand the architecture of the VHDL code refer to the accompanying .vhd text file which contains the entire code and can be opened using Notepad (or Word). After the entity declaration given above, comes the architecture. Here there are a number of declarations in the architecture declarative region (the part of architecture before the architecture 'begin' keyword). Some internal signals are declared here. Notice, in particular, that a 'shared' variable called 'flag' of type Boolean (possible values: true/false) is declared here and initially set to 'false'. The keyword 'shared' makes sure that this variable is shared (visible) with all processes in the architecture. 'flag' is a very important variable here as it indicates whether we are actively transferring data or are in a rest phase between consecutive data transfers. After these data objects there is a declaration for a component called 'MyPLL'. You will obtain it in exactly the same way as the PLL IP variation you got in module 8 lab. The IP manager will create the definition file for MyPLL – just make sure that your PLL reduces the frequency from 50 MHz to 2.5 MHz and specify M9 Lab project folder for it where ALL files for this lab will reside. Note that this week we are using this PLL IP as a component in a VHDL structural manner (using port mapping) rather than using it as a block in a schematic editor. Any IP block can be used in either way, as was described in the previous lab. Next the architecture begins after the keyword 'begin'. First you create ('instantiate' in VHDL terminology) one My_PLL component for use in your system. Note that this instantiation statement is considered a concurrent statement in VHDL. The rest of the code comprises the main design.

There are three processes here. A good way of controlling a set of multiple synchronous signals is to dedicate a separate process to each signal. Each process is sensitive to the system clock (to maintain synchronism) and possibly to other (asynchronous) signals as well, such as reset etc. The system clock here is the 2.5 MHz clock that we get out of our My_PLL component – not the 50 MHz external clock. The order of these processes in VHDL code does not matter at all because each process is considered a single concurrent statement whose order in VHDL is of no importance.

The first process is for generating the SCLK clock signal for the ADC. Study the process code to see how it works. Note that 'if (flag = true) then' can be written simply as 'if (flag) then'.

The second process is for generating the chip select signal CS_n or CS bar. Again, study its code to see how it works.

Next comes a large process that carries the label 'dout_process' (it could be broken into smaller processes, if desired). It is for interfacing with the DIN and DOUT connections on the ADC as well as for connecting to switches and LEDs on the DE0-Nano board. Much of the timing logic is contained in this process (in the first part, before it gets to the display operations). Spend

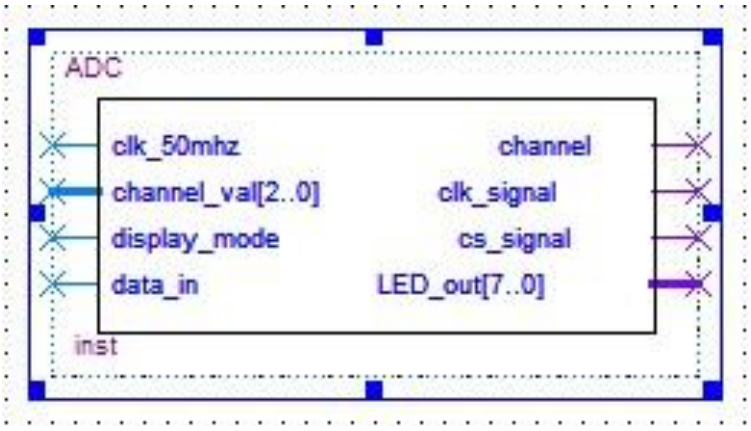
considerable time on making sure that you understand how the timing logic is coded here (refer to the timing diagram as you study this). Multiple nested if – then structures are typical of this type of code. The very first part of the code simply inserts a delay before each data read operation.

The next part sends in the channel number to the ADC as a serial three bit value through the DIN pin. This action precedes all data acquisition cycles as the ADC needs to know which of the eight analog voltage input pins to sample. Voltage at the selected pin will be obtained ('track' and 'hold' in ADC terminology) before digitization commences. Local variable 'kntr' is used to keep track of clock pulses so that the signal generation/transfer is in accordance with the timing diagram (study the logic here with reference to the timing diagram). Note that the channel number data is sent to the ADC and is acquired by it on falling edges of the clock so this part works during falling edge transition of the internal 2.5 MHz clock. Again, this is in accordance with the timing diagram.

Proceeding further, you read in 12 successive digits from the DOUT pin of the ADC. This happens while data transfer operation is taking place ('flag' is true) and at each rising clock edge. After data read operation the data is re-ordered (and truncated to eight significant bits).

The last part of this process displays the acquired data (held in array data_out of eight std_logic values) on the eight LEDs on the DE0-Nano board. This can be done in one of two ways, as can be seen in the code. Note that 'to_integer' is a function contained in the ieee.numeric_std library (included at the top of the code listing) which converts its std_logic argument to an integer value so that numeric range comparisons can be carried out.

To do this week's lab, create a new folder (M9Lab). First create an IP variation of ALTPLL called My_PLL inside this project folder. Make your PLL generate a 2.5 MHz clock. Next open a new project inside this folder, calling it 'ADC'. Create a new VHDL code entry file and enter the code given in the ADC.vhd file. Compile and correct any errors. After compilation, with code visible, generate a block symbol for your compiled code. Once this is done and you double click on the resulting adc.bsf file, you should see the schematic layout editor open and the block appearing as below.



You can demonstrate use of your ADC code in one of two ways. You can use Pin Planner to connect the ports of the ADC entity to the ADC, oscillator, switches and LEDs on the DE0-Nano

board, followed by compiling again (then configure your FPGA using the ADC.sof file). With this approach you do not need to generate a schematic block symbol for your code. You can also start a new project called top_level (or something of your choice), open a new schematic editor window in it and wire up your ADC block shown above to various DE0-Nano board resources. For this lab, do the former but you will most likely handle things using the schematic approach if you were making something where ADC was just one part of a larger design.

The selected channel pin (say, Analog_In0) will now become a voltage pickup – any voltage present at this pin, with respect to the DE0-Nano ground, will be continuously digitized and displayed on the LEDs. Some combination of LEDs will be on – depending on what voltage is being induced on the pin. This combination will likely change if you touch the analog input pin on the 2x13 connector. In noisy environments you may see the LEDs flickering because of random noise pickup. This may be reduced by placing a small capacitor between the input pin and ground, as seen in the ADC application circuit. Note that the reference voltage for the ADC is 3.3 volts. This means that voltages in the range of 0 to 3.3 V are digitized. Take care not to apply a voltage higher than 3.3 V to the active input pin. A 3.3 V input will produce binary 11111111 (maximum value) and a 0 v input (connecting the input pin to the ground pin) will produce binary 00000000 (minimum value). This is easy to verify because both these voltages are available on pins on the 2x13 pin header. If you use a single AA battery (observe polarity) then its 1.5 v should result in about half of the LEDs lit in the bar graph display mode. Finally, note that while in this block we use the data_out value internally in the code to drive LEDs, we can also simply take it out as an 8-bit std_logic_vector and use it as an input to other blocks in any desired application. Such an application may be a voltmeter. In this case, the binary data needs to be scaled (so that it is mapped from 0-255 to 0-3.3), converted from binary to binary-coded decimal (BCD) format and then used to drive an LCD or a seven segment display. These other functions can be performed in separately coded blocks which are wired at the end in a top-level schematic. In any case, having an ADC control core is very useful. Notice, that you can give away your core (with all associated files) to someone else, with information on what each port expects and they can simply use it without knowing much about the ADC itself! This feature makes it possible for collaborative groups to break up large tasks into simpler components that can be worked on by individual members of the collaboration. Extremely useful for large projects. You will get a taste of this in your project although you will have to develop all blocks yourself.

I strongly urge you to spend time and effort in understanding how the code in this lab actually works (with reference to the ADC timing diagram). If you understand the logic then it will become much easier to write similar device drivers for other ICs. The ADC is probably the easiest one to write code for, of all the peripheral chips on your DE0-Nano board, so it was chosen as an example to introduce writing of FPGA interfacing code. Other devices are slightly harder to handle but now that you know how things work in general you may want to give them a try. For instance, writing VHDL code to write a byte at a given address in the DE0-Nano EEPROM or to read a byte from a given address. Being able to use some non-volatile storage is often useful in many applications. If you are comfortable with this lab then that may be the next challenge.

As usual, submit this lab assignment by placing screen shots of code window, RTL window and Pin Planner window in a Word file and submitting this as a Word or pdf file through the Blackboard.