EE 5143 – M8 Lab – Dr Rahman

This week we look at some further concepts that will prepare us for working on the course project later. Mainly, we are going to see how digital systems can be designed by interconnecting code blocks (often also called intellectual property blocks or IP blocks). This type of design is similar to what you did in your M6 lab when you constructed an AND-OR-INVERT (AOI) circuit by interconnecting gates in a schematic diagram in QP. The difference in this lab is that instead of using primitive logic gates you will interconnect blocks that may themselves contain designs of any complexity. This is an extremely powerful concept as it allows you to design very complicated systems in an orderly fashion. You start by designing small sub-systems as separate entity/architecture pairs. After compiling each such pair (as an individual project), you ask QP to generate a block diagram for your entity/architecture pair. The block diagram appears as a black box with just the I/O interfaces visible. The block diagram also appears as a file in your project folder, with the extension .bsf which stands for block symbol or block schematic file. A number of these blocks (existing in the same project folder) can be wired together in the QP schematic editor. Finally, input and output pins can be placed on the schematic editor (as was done in the M6 lab) and connected to the rest of the system. This way, a large complex system can be broken into small manageable sub-systems that are individually designed, coded, debugged and tested. Once, all subsystems have been created these are just interconnected with each other to generate the complete system.

The blocks we use are usually called intellectual property (IP) blocks because (i) coding them is an ‘intellectual’ exercise as it requires knowledge and creativity on your part (ii) once you create a working block, it is essentially your ‘property’ which you can use in any way you see fit (even sell it to others). We’ll see later that this idea of IP blocks is also prevalent in VLSI design, although in a different form. IP blocks (or simply, IP) can also be obtained from Intel, Xilinx etc. – sometimes free and at other times after making a payment. Third parties also sell IPs for useful functions and algorithms, especially in video and signal processing. In the particular context of FPGAs and QP, Altera/Intel sometimes calls its IPs Megafunctions. You can take a look at the collection of free IP available with QP if you look at the right side of the QP main window. Under Library you can see the various classes of IP that are available. Note that IP block is also called IP core. Go ahead and explore which IPs are available as built-in features in QP.

Each time you need to use one of these built-in IPs (these are requested simply by double clicking on them), you have to answer a series of questions. QP uses your responses to generate an IP block that is customized to your needs. The code QP generates by following your responses is called an IP variation. As we are doing our work in VHDL so we should let QP generate associated files in VHDL. The files that are generated by QP when you start customizing an IP include an IP variation file (.qip for Quartus IP and also a .vhd file containing IP’s VHDL code that we do not use directly but must exist for QP’s use during compilation), a VHDL component definition file (.cmp), a VHDL file which contains a template for component instantiation (for use in structural VHDL code, if you choose to do your coding that way rather than through a schematic editor). The idea is that you will take a look at the instantiation .vhd text file and then copy and paste it into your own structural VHDL code in order to instantiate (create) an instance of that IP in your code. Of course, you can instantiate any number of instances of the same IP in your structural VHDL code – using different instance labels. That way you can have multiple copies of the same component, if you need more than one of them. You can also ask for a schematic block diagram file to be created with your IP variation. This will be used if you do a schematic design (as we'll do in this lab). Finally, it is important that all files generated by QP during IP customization reside in your current project folder so that any needed files can be accessed by QP during the compilation process.

To get a taste of how a QP-provided IP could be customized for your own use, do the following in order to create a RAM memory block controller. Create a temporary folder in your EE 5143 projects folder – you can call it something like Tempo. Launch QP but do not start a project. Go to the IP Catalog pane on the right side of the QP window. Under library you will see Basic Functions, double click on it and then go down to On Chip Memory and double click on it. Go down and double click on RAM: 1-PORT (a 1-port memory is one where the same data port is used for both memory read and memory write). A Save IP Variation pop-up message box comes up. First, for IP variation file type select VHDL. After that, for IP variation file name navigate to your Tempo folder and name your file IP_Test. Click OK. The MegaWizard Plug-In Manager appears to start customizing the IP to suit your needs. On the first page, make sure that the currently selected device family says Cyclone IV. The IP variation is specific to a given device family as different families have different amounts and types of memories available. The block diagram of the IP shown there with its signal ports will give you a good idea of how this IP block works. Select the 'q' output bus width to 8 bits as we are making an IP for storing 8 bit (1 byte) words in each individual memory location. For number of words of memory select 1024 (1K bytes). This will generate a 1K byte memory block addressable in the form of 1 byte words. Leave the memory block type as Auto. Clocking method should be left at single clock. Before you click on Next to move to the next page, notice a couple of other things. There is a Documentation button at the top right of the window. This button gives you access to two user guides. One is a manual for embedded memory blocks that are available in Cyclone IV and other similar device families. This particular IP core uses one of the so-called M9K memory blocks on your FPGA chip (look at the bottom left where it says Resource Usage 1 M9K) . As you would guess, an M9K block is a memory block integrated on the FPGA chip with the capacity to hold a total of 9K bits. There are several of these memory blocks scattered throughout the chip such that the total memory capacity is 6M bits. Obviously, these blocks are very useful if you want to temporarily (remember these are SRAMs so are volatile) store some data during processing inside the FPGA. The other documentation available is a user guide for the IP core that you are using (all commercial IP cores come with user guides that explain their function and how to use them in your own system i.e. what I/Os are available, the maximum clock frequency possible etc.). You should access both of these documents just to have a quick look at them (these are quite long so don't read them now). Now press Next to move to the next page. Leave everything as it is on this page and move to the next page. Again, leave everything as it is on this page and move to the next page. Do the same thing two more times until you are on the last page (6 of 6). On this page, put checks in boxes to generate these files: .cmp (component declaration file), .bsf (block symbol file – this is the symbol that will be generated for use in the schematic layout), .vhd file (there will be two: one is the actual IP component definition file which is always generated and the other is a component instantiation template file which, as explained above is for copy and paste in your own code if you want to write structural VHDL code). Once you click Finish, all these files will be generated and placed in the Tempo folder. You can now go directly to the Tempo folder and examine each file (except the .qip – Quartus IP file which is a wizard-generated system file). All .vhd files are simple text files and can be read using either Notepad or Word. The .bsf file holds a customized block symbol for the IP block (I tend to say IP block whereas Altera calls the same thing IP core). If you double click on it then QP opens up in the schematic editing mode with the IP block symbol positioned in the schematic area. Thin lines show single bit connections whereas thicker lines are multi-bit connections (buses). The little ‘x’ symbols show where external connections will be made. Note that all I/O’s are registered with the use of clocked D flip-flops. This completes our first look at IP cores and the core wizard. We will not use this core in this lab – this was just to serve as an example of how you configure an IP core in QP.

Now we come to this week’s lab project. It is a simple project where we cascade two blocks in the QP schematic editor to build a simple 1 second time period LED blinker. One of the blocks will be an IP core from Altera that configures a phase locked loop (PLL) inside the Cyclone IV FPGA chip. The other will be a block that is a counter that we code in VHDL. The PLL block will bring down the clock frequency from 50 MHz to 10 MHz by dividing the clock by a factor of 5. The other block will count rising edges of the 10 MHz clock output from the PLL to create a 1 Hz square (50% duty cycle) waveform which will be fed to an LED on the DE0-Nano board. These blocks can be created in any order but here we’ll first create the PLL block.

In addition to logic elements (LEs), interconnection wiring and I/O buffers, an FPGA chip usually contains other useful circuitry that is said to exist as ‘hardened’ blocks. A hard block or hard IP is a specific circuit that has a fixed purpose which has been fabricated on the FPGA chip. For example, at the four corners of your FPGA die there are four individual PLL blocks. Each of these could be used to generate one or more clock signals to use as you desire. The PLLs allow the 50 MHz input clock to be either increased or decreased in frequency, generate two synchronized clocks with a known frequency and/or phase difference between them etc. Altera provides PLL megafunction (ALTPLL) to configure any PLL as you wish. Similarly, there are embedded SRAM memory blocks which can be configured using their own megafunction (as we saw above). There are also quite a few 18 bit x 18 bit hardware multipliers that can be used to quickly multiply two numbers that are up to 18 bits wide. These hardware multipliers are also configurable using similar techniques as described above. More advanced FPGAs (Arria, Stratix etc. from Intel) have other embedded hardware on them so that implementing common functions doesn’t require synthesizing circuits using FPGA LEs which should be kept for your own custom algorithms. The hard blocks are also highly optimized so they will function better and quicker than circuits synthesized to perform the same function using VHDL code. Note that QP is intelligent enough that if it notices something in your VHDL code that can be implemented with a hardened feature available on the targeted FPGA then it will use that hard block instead of synthesizing that circuit (and thus consuming LEs). Thus, if you are multiplying numbers then it will use embedded multipliers, if possible, rather than constructing a multiplier using raw logic. Note that if it is a simple multiplication such as multiplication by a power of 2 then it will be implemented as a shifting logic and an entire hard multiplier will not be wasted on it.

To start this week’s project, first save any existing files in QP and close any existing project. At this time do not start a new project. Let’s first create a PLL IP variation – following steps very similar to those followed above for the memory block IP. In this case go to Library -> Basic Functions -> Clocks; PLLs and Resets -> PLL -> ALTPLL. Double click on ALTPLL to get started. First, make sure that IP variation file type is selected as VHDL. Navigate to M8 folder and call your IP variation MyPLL. Click OK. After a slight pause, IP variation manager window appears. On page 1 of 12 of IP manager, select device speed grade as 6, make sure the family is selected as Cyclone IV, enter frequency of inclk0 input as 50MHz – this is the frequency that our PLL will be fed, from the external oscillator. Leave the rest as it is, just making sure that at the bottom ‘which output clock will be compensated for’ says ‘c0’. Click Next. On the next page (2 of 12) uncheck creation of ‘arest’ input and ‘locked’ output. Notice how connections disappear on the block diagram to show the IP is being modified as you want it. Go on clicking Next until you reach page 6 of 12. Make sure that ‘use this clock’ is checked. Enter 5 as the clock division factor; leaving 1 as the multiplication factor. This will make the PLL convert 50 MHz clock into a 10 MHz clock. Go on clicking Next until you reach the last page of the manager (12 of 12). Here check to generate .cmp, .bsf and .vhd files. In fact, the only one you really need here (other than those that are automatically generated by default and listed as grayed-out at the top) is the .bsf symbol file for use in the schematic editor. If you check now then all of these files will be found inside your M8 folder. These files will appear as MyPLL.bsf, MyPLL.cmp etc. There will also be a file called MyPLL.ppf with ppf standing for pin planner file. This file is for internal use by the Pin Planner and does not concern us.

The next thing to do is to write some VHDL code to divide the 10 MHz output from the PLL to obtain a 1 Hz output. This time you will also convert it into a schematic block. To begin, perform the same actions as in the previous labs. Save all files and close any open project then start a new project using the New Project Wizard. Call the project BlinkerTest. It will hold your VHDL code for clock division. This project can be placed in a folder (directory) inside your EE 5143 projects folder. You can call this project’s folder M8 (for module 8). This folder can either be created separately inside the EE 5143 projects folder or you can navigate inside the EE 5143 projects folder from the wizard and then create a new M8 folder inside it. Either way works. One you have named the project BlinkerTest, this same name will be used for the entity of this design. Complete the wizard, as usual. Remember, if QP says there is already a project or files in the M8 folder then disregard that by clicking on ‘No’. Create a new VHDL file and save it as BlinkerTest.vhd. Then type in the following code in the text entry area of BlinkerTest.vhd:

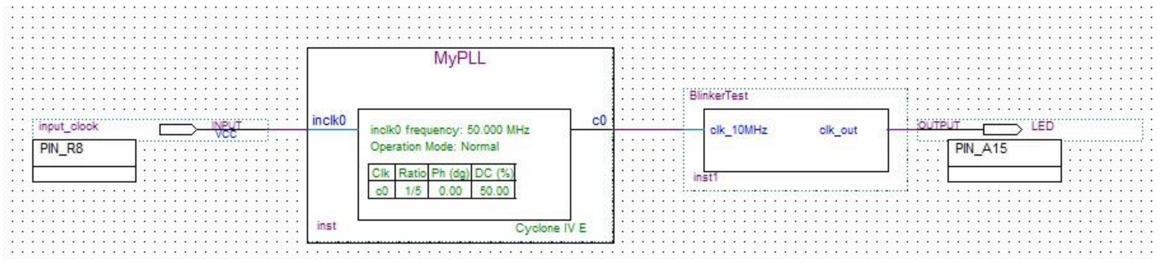
```
---------------------------------
--                  Library Declaration        --
---------------------------------
-- Like any other programming language, we should declare libraries

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

---------------------------------
--                  Entity Declaration                --
---------------------------------
-- Here we specify all input/output ports

entity BlinkerTest is
        port(
                clk_10MHz : in std_logic ;
                clk_out : out std_logic
        );
end entity;

---------------------------------
```

```
--          Architecture Declaration            --
----------------------------------
--          here we put the description code of the design

architecture behave of BlinkerTest is

-- signal declaration

signal clk_1Hz : std_logic := '0' ;
signal scaler : integer range 0 to 5000000 ;

begin

-- this process is used to scale down the 10 MHz frequency from the PLL to a 1 Hz rate

clk_1Hz_process : process( clk_10Mhz )
begin
     if(rising_edge(clk_10MHz)) then
                                    if (scaler < 5000000) then
                                                scaler <= scaler + 1 ;
                                    else
                                                scaler <= 0;
                                                clk_1Hz <= NOT(clk_1Hz);
                                    end if;
    end if;
end process clk_1Hz_process;

clk_out <= clk_1Hz;

end behave;
```

By now, you should be able to understand exactly what this piece of VHDL code does so I'll not go into much detail. Notice that it counts 5 million rising edges on the 10 MHz clock coming from the PLL output (which is the input to the entity). After counting 5 million rising edges it toggles the internal signal clk_1Hz. Thus, every half a second the std_logic value of clk_1Hz changes from 0 to 1 and then from 1 to 0, effectively generating a symmetric (50% duty cycle) 1 Hz waveform. This is taken out as signal clk_out.

Save and compile your code. Then, with the code window visible, go to File -> Create / Update -> Create Symbol Files for Current File. This will execute quickly and result in the creation of a BlinkerTest.bsf file inside your M8 folder. You may want to check that this file is present in the M8 folder before proceeding further.

Next, save all files and close your existing project. Start a new project which will also be placed inside the M8 folder. Call it BlinkerTest_TopLevel. This will hold the interconnection between the two blocks that you have created so far. Complete the wizard as usual. This time, go to the file menu and using New create a Block Diagram/Schematic File. This will open up in the editing area as a window full of grid points. Now the procedure is very similar to the M6 lab. Right click anywhere in the schematic editing area and click insert -> symbol. Then expand 'Project' to see MyPLL and BlinkerTest blocks. Place one instance of each on to the schematic area and wire them up as shown below (refer to M6 Lab instructions in case of any difficulty). Enlarge this document to see everything more clearly.



After the wiring process you will have everything looking as above except the labels PIN_R8 and PIN_A15 (those appear after pin assignment which comes later).

Once you are satisfied with your schematic circuit, save all files and then compile BlinkerTest_TopLevel. Here, I want to mention a helpful point. This circuit is made of only two blocks but other circuits may consist of many blocks connected together. If your overall circuit does not work as intended and you figure out that the fault lies in a certain block then you can double click on that block and ask to be taken to the code for that block. After that, you can examine the code to see what may need fixing. In order to fix that code you will have to close the top level project and open the project for the block that needs attention. Then fix the code and compile it. Once that is done, you need to go back and open the top level project again which will show the schematic diagram. Then you need to compile the top level itself again. Thus, every time you need to fix a component block you need to compile both that block and after that the top level.

Next, make pin assignments using the pin planner, as usual. Connect the input pin you have named input_clock to FPGA pin R8 and the output pin LED to A15. Then compile BlinkerTest_TopLevel. Now you are ready to configure your FPGA. Use the BlinkerTest_TopLevel.sof file from the output folder in M8 to configure your FPGA on the DE0_Nano board. LED0 should start blinking at a 1 Hz rate.

This week you should also learn how to download your configuration code into the configuration memory on your board so that the FPGA gets to work after you turn the power off and then on again. This requires a file different from the SRAM object file (.sof) that we have been using so far. The file we need here is called a JTAG indirect configuration (.jic) file. We always first have to have a .sof file then, if needed, we can create a .jic file by using the .sof file. This is usually called file conversion and can be accomplished from the file menu in QP, using the option Convert Programming Files. I leave it to you to do this by following instructions in the DE0-Nano user manual. Go to Chapter 9 (page 145) and follow the instructions there to generate a BlinkerTest_TopLevel.jic file. Follow further instructions there to load it into the configuration flash memory device. Notice that writing the configuration bit stream to the flash chip takes considerably more time than simple FPGA configuration using a .sof file. Because of all the additional steps needed in file conversion etc. we usually do not bother with configuration memory programming. But if you want your board to actually hold your code in a non-volatile fashion then you must generate and use a .jic file. After .jic file programming your code will not execute automatically; you need to disconnect your USB connection and then rec-connect it to power cycle the board. Then you should see your LED blinking. This time you can disconnect your board and re-connect it as many times as you like and the LED will always blink when power is applied. You do not actually need your board connected to a computer as you can just connect the USB connector to a 5V USB power supply as used for charging phones etc. and that will make the board work. If your board flash has been programmed then you can take your board anywhere without your computer and power it to make it work. One final point. The bit stream resides inside the flash memory and the FPGA loads it automatically every time power is applied to the board. You need to erase it if you want to use the board for other projects (using either .sof or .jic files). Follow instructions on pages 151 and 152 of DE0-Nano user manual to erase your flash after you have tested your board so it will be ready for future projects.

To submit your M8 lab assignment, place screen shots of your BlinkerTest code, BlinkerTest_TopLevel schematic and pin planner window into a single file and submit it as either a Word or pdf file through the Blackboard. Completing this assignment successfully will get you 20 out of 25 points this week. To get full 25 points you need to do the following bonus activity.

Modify your BlinkerTest code so that it reads one of the slide switched on the DE0-Nano board and depending on the switch setting makes the LED blink at 1 Hz or 2 Hz rate. This does not require any change to MyPLL but will require changes to both BlinkerTest code and to BlinkerTest_TopLevel schematic. You should send the BlinkerTest code, BlinkerTest_TopLevel schematic and pin planner screen shots to support your bonus activity. These can be in a separate file or in the same file as the main project content.