Lab #8
Adding a Custom Hardware VHDL PWM controller to the Altera NIOS II Processor and Memory Mapping it onto the Avalon Interconnect Fabric

EELE 475
HARDWARE AND SOFTWARE ENGINEERING
FOR EMBEDDED SYSTEMS

Assignment Date: 11/10/15

Lab Report Due Date: 11/24/15

Lab Description

Radio controlled (RC) planes, vehicles, and quadcopters are controlled by Pulse Width Modulation (PWM) signals. These signals control servo motors (i.e. steering) and speed control units for DC motors used in propulsion. A custom Pulse Width Modulation (PWM) component, created in VHDL, has been developed to create a PWM signal. We will make use of this component in today's lab where you will learn how to attach the component to the NIOS II processor so that the NIOS II will see it as a memory mapped peripheral. In addition, we will set it up in the Qsys IDE so that it appears just like all the other components you have used. You will be able to easily create three PWM controllers on the Avalon bus (interconnect fabric). Like last week, you will need to use the following Altera tools:

- **Quartus II** - To compile the VHDL that creates your processor hardware.
- **Programmer** - To download your bit stream to the DE2 board. (QuartusII -> Tools -> Programmer)
- **Qsys** - To create your processor system. (QuartusII -> Tools -> Qsys)
- **NIOS II IDE** - To write, compile, and download your C program. (Qsys -> Tools -> Nios II Software build tools)

Your starting point will be the system you created last time (i.e. GPS packet parsing). Create a new lab8 directory and copy all your files from lab7 into this directory.

Part 1 – Creating the Avalon Bus Wrapper

Any custom hardware that you create for the NIOS II processor needs to interface to the Avalon interconnect fabric.   The interface supports both memory-mapped peripherals as well as streaming interfaces. The streaming interfaces are useful for data processing tasks such as digital filtering. We will just use the memory-mapped interface. A memory-mapped interface can function as either a *master* or a *slave*. A *master* can initiate data transfers while a *slave* only responds to data transfers. We will only implement a memory mapped slave interface, which is the easiest to create. The Avalon Interface Specifications can be found at (and on the D2L site):

http://www.altera.com/literature/manual/mnl_avalon_spec.pdf

The Avalon slave that we will create will look like figure 3-1 and figure 3-2 found on page 3-1 & 3-2 (section 3.1) in the Avalon spec. The slave must be able to handle complicated transfers, so there are additional signals available for a slave. We, however, will only be writing data to several registers to control the PWM peripheral. This means that our interface is relatively simple. The only signals we need are the clock, reset, address, data, and a write enable.

The VHDL entity of the VHDL PWM controller is (The VHDL code is on the D2L web site):

```
entity pwm_control is
   Port ( clk                    : in  STD_LOGIC;  -- PWM timing is based on the number of clock cycles from clk
        reset                    : in  STD_LOGIC;  -- reset
        enable                   : in  STD_LOGIC;  -- enable PWM (1 = pulse, 0 = no pulses)
        control                  : in  STD_LOGIC_VECTOR (7 downto 0);  -- control word [-128:+127], signed char, controls pulse
        pulse_period             : in  STD_LOGIC_VECTOR (23 downto 0); -- number of clock cycles (100 MHz) between pulses
        pulse_neutral            : in  STD_LOGIC_VECTOR (23 downto 0); -- pulse width in clock cycles when control word is 0
        pulse_largest            : in  STD_LOGIC_VECTOR (23 downto 0); -- pulse width in clock cycles when control word is +127
        pulse_smallest           : in  STD_LOGIC_VECTOR (23 downto 0); -- pulse width in clock cycles when control word is -128
        pwm_signal               : out  STD_LOGIC);  -- the PWM signal
end pwm_control;
```

Notice that the information that we will pass to the controller will be based on the number of clock cycles of the system clock.

> **Note:** Don't assume that the clock will be 100 MHz. We will target 100 MHz, but it may be less than this value (it could be 50 MHz) due to a particular device being targeted, the NIOS II processor type, and timing closure issues. The maximum speed needs to be derived from the timing report and this has to be checked after you compile your design and before you move on to software development. If you need a slower clock, you will need to change the clock value in the following likely locations and recompile your design.

1. PLL (Edit with MegaWizard tool)
2. .sdc file
3. Frequency of Nios clock in the Qsys IDE.

We will pass six data words to the controller. Four of the words need to be initialized at the beginning of your program.   The control word will change frequently as you "drive" the vehicle (the PWM could be controlling steering and/or speed) and the enable control word will gate the output pwm signal (something to check if you are not seeing the PWM signal).

The data bus of the NIOS II is 32-bits. This means we will create six 32-bit registers. The control words that are smaller than 32 bits will be right aligned and the extra, more significant bits, will be ignored on a write. They should be sign extended to the full 32-bits on a read.   To connect to the Altera bus, we need to create a bus wrapper for the pwm_control block. This wrapper is called pwm_avalon and is implemented as a slave interface. The slave interface entity is shown below.

```
entity pwm_ is
   port (
      clk                  : in std_logic;
      reset_n              : in std_logic;
      avs_s1_write         : in std_logic;
      avs_s1_address       : in std_logic_vector(2 downto 0);
      avs_s1_writedata     : in std_logic_vector(31 downto 0);
      pwm_signal           : out std_logic
   );
end pwm;
```

Notice the timing diagram for writes on the Avalon interface that is shown in Figure 3-3 on page 3-9 (section 3.5.1) of the Avalon spec.   If you look in the wrapper VHDL code, you will see that the data is latched on the rising edge of the clock (and when the write enable is asserted). Notice that the address bus that connects to the Avalon interface (avs_s1_address) gives the offset address of the registers in words.   This means that Qsys does the full address decoding for you.

In your code you will need to know the following register offsets that are relative to the base address that Qsys assigns to the PWM peripheral. These are, for example:

```
#define PWM1_BASE_ADDRESS          0x01001020
#define PWM1_CTRL                  ((volatile int *) PWM1_BASE_ADDRESS)
#define PWM1_PERIOD                ((volatile int *) (PWM1_BASE_ADDRESS + 4))
#define PWM1_NEUTRAL               ((volatile int *)( PWM1_BASE_ADDRESS + 8))
#define PWM1_LARGEST               ((volatile int *)( PWM1_BASE_ADDRESS + 12))
#define PWM1_SMALLEST              ((volatile int *)( PWM1_BASE_ADDRESS + 16))
```

```
#define PWM1_ENABLE                ((volatile int *) (PWM1_BASE_ADDRESS + 20))
```

where PWM1_CTRL controls the pulse width. It is an 8-bit signed integer where a value of +128 sets the pulse width to PWM_LARGEST (in clock cycles), 0 sets it to PWM1_NEUTRAL (in clock cycles), and -128 sets it to PWM1_SMALLEST (in clock cycles).

> **Note:** In the define statements above, your Eclipse C preprocessor probably won't like a #define using a #define as part of the definition (other C compilers will work with this). You may need to compute what these address values are directly and plug them into the define statement.

Part 2 – Register Modifications

In this part you will modify the Avalon Registers. The registers given in pwm_avalon.vhd are write only, which makes debugging a challenge if the PWM peripherals don't work at first (Note: if it they don't work at first, double check the assigned base address values). You now need to make the registers both read and writable. In addition, you need to make an extra set of read-only registers (i.e. shadow registers) that are aliased meaning that registers are offset by 64 bytes (16 32-bit words) that can also be read and reflect the same register values controlling the PWM. In order for you to read your registers, your entity in the pwm_avalon.vhd should look like:

```
entity pwm_avalon is
        port (
                clk                     : in std_logic;
                reset_n                 : in std_logic;
                avs_s1_address          : in std_logic_vector(3 downto 0);
                avs_s1_write            : in std_logic;
                avs_s1_writedata        : in std_logic_vector(31 downto 0);
                avs_s1_read             : in std_logic;
                avs_s1_readdata         : out std_logic_vector(31 downto 0);
                pwm_signal              : out std_logic
        );
end pwm_avalon;
```

Part 1 – Qsys Modification

Note: In the following instructions, only the Avalon *write* signals are described. Treat the *read* signals in your modified entity in a similar fashion.

1. Copy the following files from the web (D2L) and put them into your project directory.

   a. pwm_avalon.vhd (which you have modified to make the registers readable)
   b. pwm_control.vhd

2. Create new peripheral by clicking *new* at the bottom left of Qsys (at the bottom of the Component Library tab).

3. Click on the tab *HDL Files*, click on *add* and add pwm_avalon.vhd. (you will also need pwm_control.vhd in your directory). Don't worry if you get an error at this point. We will fix them next.

4. Click on the tab *Signals*.

   a. Set clk to the Interface *clock* and signal type *clk*
   b. reset_n to the Interface *reset* and signal type *reset_n*
   c. avs_s1_write to the Interface *s1* and the signal type *write*
   d. avs_s1_address to the Interface *s1* and the signal type *address*
   e. avs_s1_writedata to the Interface *s1* and the signal type *writedata*
   f. pwm_signal to the Interface *new Conduit* and the signal type *export* (this brings the signal out of the nios_system entity). It will show up as *conduit_end*.

5. Click on the *Interfaces* tab. Click on the button that says "Remove Interfaces with no signals". Go to the "s1" (Avalon Memory Mapped Slave) section and set Associated Reset to reset.

6. Click on the *Library Info* tab and enter the following

   a. Display Name : pwm_controller
   b. Group : Peripherals/FPGA Peripherals

   Click finish. The pwm peripheral should now show up under *Library* in the component library tab.

7. Add three pwm peripherals to your system. Name these peripherals pwm1, pwm2, and pwm3 and *Auto-Assign Base Addresses* again. Add the clocks, resets, data, and export using the name pwm1, pwm2, and pwm3.

8. Regenerate your Qsys nios_system.

9. You will now have three signals in the nios_system component that needs to be connected to the external world.

```
-- the_pwm1
signal pwm_signal_from_the_pwm1   :   OUT  STD_LOGIC;
-- the_pwm2
signal pwm_signal_from_the_pwm2   :   OUT  STD_LOGIC;
-- the_pwm3
signal pwm_signal_from_the_pwm3   :   OUT  STD_LOGIC;
```

10. Connect these three signal in the port map to:

```
pwm_signal_from_the_pwm1          => GPIO_1(1),
pwm_signal_from_the_pwm2          => GPIO_1(3),
pwm_signal_from_the_pwm3          => GPIO_1(5)
```

These signals are located as the top three I/O pins in JP2 of the Expansion Headers, on the right side (See page 35 in the DE2 user's manual).

11. Compile your hardware design. Check your timing report to make sure that the design can run at the desired clock speed. If not, you will need to change the following clock settings and recompile:

a. PLL (Edit with MegaWizard tool)
b. .sdc file
c. Frequency of Nios clock setting in the Qsys IDE.

12. Create a couple header post connectors to connect the output pwm signals to an oscilloscope. Connectors, wire, shrink wrap tubing, etc. can be acquired from the ECE stockroom.

<u>Part 4 – Software</u>

13. Open up the NOIS II IDE

14. Select File->New->NIOS II C/C++ Application

15. Select the Project Template *Blank Project*.

16. Write a program that will create the following PWM signals coming from pwm1 (GPIO_1(1)).

a. If SW0 is on (LEDR0 on), the control word will be set to -128 and the pulse width shown on the oscilloscope will be 1 msec.

b. If SW1 is on (LEDR1 on), the control word will be set to 0 and the pulse width shown on the oscilloscope will be 1.5 msec.

c. If SW2 is on (LEDR2 on), the control word will be set to +127 and the pulse width shown on the oscilloscope will be 2 msec.

d. The pulses should repeat themselves every 20 msec.

e. Ignore the switches if several of them are on.

17. The values to set PWM1_PERIOD, PWM1_NEUTRAL, PWM1_LARGEST, and PWM1_SMALLEST can be computed by the Matlab script pwm.m (which may need to be modified), which is found on the D2L web site.

18. Write code to show that the register values can be read back both from both their original writable locations and also from their new aliased read locations.

Nios II Cache Considerations for Software Programmers

If you select a cache for the Nios II as one of the CPU options, you need to be aware that the Nios II cores have no hardware cache coherency mechanism. This means that writing to cache will not show up in memory (and thus in your memory mapped registers) unless you either bypass the cache or flush the cache.

You have the following options:
1. Don’t use a cache
2. Flush the cache after writing to it.
3. Use the Bit-31 Cache Bypass (Nios II/f core only)

See Altera’s Documentation on Cache and Tightly-Coupled Memory (n2sw_nii52007.pdf)

For HAL Users

The HAL provides the C-language macros IORD and IOWR that expand to the appropriate assembly instructions to bypass the data cache.

Table 9–1. HAL I/O Macros to Bypass the Data Cache (page 9-4)

IORD(BASE, REGNUM) - Read the value of the register at offset REGNUM in a device with base address BASE. Registers are assumed to be offset by the address width of the bus.

IOWR(BASE, REGNUM, DATA) - Write the value DATA to the register at offset REGNUM in a device with base address BASE. Registers are assumed to be offset by the address width of the bus.

Bit-31 Cache Bypass

The ldio/stio family of instructions explicitly bypass the data cache. Bit-31 provides an alternate method to bypass the data cache. Using the bit-31 cache bypass, the normal ld/st family of instructions can be used to bypass the data cache if the most significant bit of the address (bit 31) is set to one. The value of bit 31 is only used internally to the processor; bit 31 is forced to zero in the actual address accessed. This limits the maximum byte address space to 31 bits.

Using bit 31 to bypass the data cache is a convenient mechanism for software because the cacheability of the associated address is contained in the address. This usage allows the address to be passed to code that uses the normal ld/st family of instructions, while still guaranteeing that all accesses to that address consistently bypass the data cache.

Bit-31 cache bypass is only provided in the Nios II/f core, and must not be used with other Nios II cores. The other Nios II cores limit their maximum byte address space to 31 bits to ease migration of code from one implementation to another. They effectively ignore the value of bit 31, which allows code written for a Nios II/f core using bit 31 cache bypass to run correctly on other current Nios II implementations. In general, this feature depends on the Nios II core implementation.

Part 4 – Run the Software

19. Compile and run your program like you did in the previous lab

Have the instructor verification sheet (given below) signed off when you get the pwm signals displayed on the oscilloscope (Demo #1) and when you can read the shadow registers (Demo #2).

Please return any oscilloscope probes that you use and put them back where you found them. Please leave your workstation clean and neat. Turn off boards and oscilloscopes.

Instructor Verification Sheet

Include this as the back page of your lab report
to get credit for Lab #8

Lab #8
Adding Custom Hardware to the NIOS II Soft Processor
and Memory Mapping it on the Avalon Interconnect Fabric

EELE 475
HARDWARE AND SOFTWARE ENGINEERING
FOR EMBEDDED SYSTEMS

Name : ______________________

Demo #1: Show that turning on the following switches will create the following effects where the pulse period is 20 msec.

1. SW0 will create a pwm signal with a pulse width of 1 msec.
2. SW1 will create a pwm signal with a pulse width of 1.5 msec.
3. SW2 will create a pwm signal with a pulse width of 2 msec.

Show the oscilloscope measuring these parameters.

Verified: ________________________________ Date:__________________

Demo #2: Show that the register values can be read back both from both their original writable locations and from their new aliased read locations (offset by 64 bytes).

Verified: ________________________________ Date:__________________