LAB 3: Using the AXI Hardware Timer

Introduction

This lab introduces the LogiCORE IP AXI Timer. As its name suggests, this is an IP core containing two hardware counters. The timers can be used independently and in various different modes. This lab consists of three parts (two prelab portions which provide the necessary background information and one part (Part 3) which is to be performed during the lab session). Part 1 introduces the use of the AXI timer as a simple timer/counter. Part 2 explores the use of the timer in capture mode - a mode in which the timer captures its current counter value into an internal register. Part 3 investigates the pulse width modulation mode of the counter and uses this mode to control the brightness of a LED.

Part 1: Use of the AXI timer as a basic timer/counter

Setting up the user Linux environment:

1. Type the following on a terminal:

    source /CMC/tools/xilinx_14.7/14.7/ISE_DS/settings64_CMC_central_license.csh

    This sets up the Xilinx environment

Launching PlanAhead:

2. Launch the planAhead tool from your Lab3 directory and

    cd COEN317_LABS
    mkdir Lab3
    cd Lab3

    planAhead &

    A message may pop up:
    > A disk write failure occurred. There may be insufficient disk space or
    > you may not have write permission at the following directory.
    > /nfs/sw_cmc/linux-64/tools/xilinx_14.4/14.4/ISE_DS/.xinstall
    > Press Retry to try again, press Cancel to exit XilinxNotify.

    Press Cancel for now. The tool may be trying to write to the install directory.

Create a new Project from PlanAhead:

3.1 From the PlanAhead window, select:

File -> New Project

3.2 A window pops up, select. Next. Give your project a name (i.e. lab3)

3.3 RTL Project should be selected. Keep the selection.

Select Next

3.4 We do not need to specify any sources at this time
However, choose the target language as VHDL in the drop-down box

Target language: VHDL
Select Next

3.5 We do not need to specify any IPs either

Select Next

3.6 This system will make use of a User Constraints File (UCF) to specify that the GPIO output port signals are to be connected to specific pins of the Zynq chip which are connected to the LEDs. Prior to this step, use a text editor to create a file (called lab3.ucf) which contains the following:

```
# axi_gpio_for_C LEDS on ZC702
NET  axi_gpio_for_output_pins<0>  LOC = E15;
NET  axi_gpio_for_output_pins<1>  LOC = D15;
NET  axi_gpio_for_output_pins<2>  LOC = W17;
NET  axi_gpio_for_output_pins<3>  LOC = W5;
NET  axi_gpio_for_output_pins<4>  LOC = V7;
NET  axi_gpio_for_output_pins<5>  LOC = W10;
NET  axi_gpio_for_output_pins<6>  LOC = P18;
NET  axi_gpio_for_output_pins<7>  LOC = P17;
```

In the Add Constraints window, select Add files and navigate to select the lab3.ucf you had previously created. Select Next to continue.

3.7 Select the ZYNQ-7 ZC702 Evaluation Board for our project

Select Boards
Scroll down the list and choose ZYNQ-7 ZC702 Evaluation Board
Select Next

3.8 Review settings and select Finish

Create an embedded processor project with the Add Source wizard:

We will add sources to the newly created empty project.

4.1 On the left panel under Project Manager, Select Add Sources

4.2 Choose Add or Create Embedded Sources and select Next

4.3 Select Create Sub-design
A window should have popped up asking for a module name. Name it "system"
Module name: system
Select OK
Select Finish

Designing the system in XPS:

After step 4.3 is performed, XPS will be automatically started and will be used to build the system and add an instance of an AXI GPIO and an instance of an AXI Timer.

5.1 A window pops up and asks:
This project appears to be a blank zynq project.
Do you want to create a Base System using the BSB Wizard?

Click Yes

5.2 The BSB Wizard asks for additional settings
The AXI System should be selected by default with no other parameters.

Click OK

Verify the Zynq Processing System 7 is selected in "Select a System"
Click Next

Remove the GPIO_SW and LEDs_4Bits Peripherals because they are not needed
Click on "Select All", then "< Remove"
Click Finish

5.3 Add an instance of an AXI GPIO. Rename the Component Instance Name to axi_gpio_for_output. Specify the channel width as 8 bits.

5.4 We will now make the output pin of the GPIO external. This will cause it to appear as an port in the top-level system_stuv.vhd wrapper. This is required since we are applying constraints on this port (through the UCF file) and only top-level ports may appear in a UCF file.

Select the Ports tab in the System Assembly View
Expand axi_gpio_for_output and its (IO_IF) gpio_0
Right-click GPIO_IO and select No Connection

Right-click GPIO_IO_O and select Make External

In the External Ports tab rename the axi_gpio_for_output_GPIO_IO_O_pin to axi_gpio_for_output_pins (the name of the external port must be different from the name that was given to the instance of the AXI GPIO or elese an error during sysnthesis will occur).

5.5 Add the hardware timer

Expand DMA and Timer
Double-click AXI Timer/Counter
Click Yes to confirm the addition of the timer

Leave the default name (axi_timer_0) and settings
Click OK

The selected processor instance to connect the GPIO should be processing_system7_0

Click OK

5.6 Close XPS
File -> Exit

Exporting the Hardware to SDK:

You will notice in the top middle pane of Project Manager that there is a newly created Design Source: system (system.xmp)

6.1. Right-click on "system (system.xmp)" and choose Create Top HDL

You should now have a system_stub.vhd file.
This file is the top level VHDL code.

6.2. Synthesize the VHDL code.
Click on Run Synthesis on the left Panel

6.3. Implement the design.
Choose Run Implementation and click OK
There may be 3 constraint warnings, they do not affect our system and you can click OK
Wait for Implementation to finish.

6.4. Generate the Bitstream

Choose Generate Bitstream and click OK
Wait for bitstream generation to finish

6.5. Download the bitstream to the Zynq Board. Attach the power cable, the Platform Cable USB II,and the serial UART cable. Apply power the board, ensure the DIP switch settings are correct. check whether the Platform Cable USB II has a green light.

Choose Launch iMPACT and click OK.

A window may pop up asking you to save the file, click Cancel.

In iMPACT, right-click the Target and choose Program.

Make sure Device 2 (FPGA xc7z020) is highlighted and click OK

When you see Program Succeeded, close iMPACT:

File -> Exit

You do not need save the iMPACT project, select No when prompted to save the iMPACt project.

6.6 Go back to the PlanAhead tool and select:

File -> Export -> Export Hardware for SDK

Check “Launch SDK” and click OK on the pop up screen.

Using SDK to create a new application project:

7.1. From the main SDK window select:

File -> New -> Application Project
Name the project : lab3.1
Select Continue

Leave the Empty Application selected as the template. Select Finish

7.2. A sample C++ program which makes use of the Xilinx provided functions relating to the AXI timer is found in:

/home/t/ted/PUBLIC/COEN317/Lab3/Part1/main.cc_with_xilinx_functions

Copy this file into your

./COEN317_LABS/Lab3/lab3.1/lab3.1/lab3.1.sdk/SDK/SDK_Export/lab3.1/src

directory and name it main.cc. This program makes use of the Xilinx provided functions related to the AXI Timer:

```
XTmrCtr_Initialize(&TimerInstancePtr, XPAR_AXI_TIMER_0_DEVICE_ID)

XTmrCtr_SetResetValue(XTmrCtr * InstancePtr, u8 TmrCtrNumber,u32
ResetValue);

XTmrCtr_SetOptions(&TimerInstancePtr, XPAR_AXI_TIMER_0_DEVICE_ID,
XTC_CAPTURE_MODE_OPTION);

XTmrCtr_Start(&TimerInstancePtr, 0);

XTmrCtr_Stop(&TimerInstancePtr, 0);

u32 XTmrCtr_GetValue(XTmrCtr * InstancePtr, u8 TmrCtrNumber);
```

The header file xtmrctr.h contains the prototypes for the functions relating to the AXI Timer. You may open this header file into the SDK Editor pane by double clicking on the xtmrctr.h file in the Outline Pane. Once the header file has been loaded into the editor, use the mouse to highlight a function name and right-click and select Open Declaration.

The `XTmrCtr_Initialize()` function performs initialization of the `TimerInstancePtr` variable decrlared in `main()` as type `XTmrCtr`.

The `XTmrCtr_SetResetValue()` is used to load a reset value into the timer. In this sample program a reset value of 0 is loaded into the timer, although any value may be used if one so desires.

The AXI Timer is configurable in different modes depending upon how one wishes to use it, the function `XTmrCtr_SetOptions()` configures the counter to be in capture mode (more on the different modes later in this lab), as an up-counter (as opposed to a down counter).

The functions `XTmrCtr_Start()` and `XTmrCtr_Stop()` start and stop the counter. By calling these functions before and after a portion of code, one may count the number of clock cycles (FCLK0 cycles) needed to perform that code portion. The `XTmrCtr_GetValue()` is used to read the current value of the counter before and after the code snippet. The difference in the two values is the number of clock cycles required to execute the code snippet:

```
unsigned in before_value, after_value;

before_value = XTmrCtr_GetValue();
XTmrCtr_Start();
for ( i = i ; i < 10000; i++)
{
 // do something
}
XTmrCtr_Stop() ;
```

```
after_value = XTmrCtr_GetValue();
cout << “It took: “ << (after_value - before_value) << “ clocks to
do that loop “ << endl;
```

(Note: the above is meant as pseudocode as the required arguments to the various timer functions are not included).

The given C++ program uses the timer to count the number of clock cycles needed to perform a `XGpio_DiscreteWrite()` to the GPIO port connected to the LEDS on the ZC702 board. Save the file from within the SDK editor pane (CTRl-S) to recompile.

7.3 Connect the SDK terminal to the board in order to establish communication between the board and the host terminal window:

Window -> Show View -> Terminal

Click the green connect button
(looks like a green ‘N’ with a dot on each end)

On the popup, choose Serial as the Connection Type
The settings are:

Port: /dev/ttyUSB0
Baud Rate: 115200
Data Bits: 8
Stop Bits: 1
Parity: None
Flow Control: None
Timeout(sec): 5

Click OK

Make sure it says “CONNECTED”

7.4. Run the executable file on the board:

Right-click the lab3.1 folder
Choose Run As -> Run Configurations
Click Run

An alternative method of interfacing with the AXI Timer

As with the AXI GPIO, there is a more direct manner to use the timer/counter. The LogiCORE IP AXI Timer Product Specification (found in /home/t/ted/PUBLIC/COEN317/Lab3/ axi_timer_ds764.pdf) contains all the information needed to learn how to use the timer.

When we added the AXI Timer instance to our system with XPS, it was assigned some address and it was connected to the processor via the AXI INTERCONNECT instance. The AXI GPIO port was also connected to the processor through this AXI INTERCONNECT (albeit with a different base address obviously). The timer/counter contains hardware (status/control registers, data registers, control circuitry, address decoding circuitry, etc). All that is required is write the proper values into the control registers and read the values from the data registers. The manner in which to do so can be determined by careful reading of the datasheet (RTFM).

Table 1 gives the descriptions of the registers contained within the AXI Timer.

Table 1: AXI Timer Registers and addresses [1]

Base Address + Offset (hex)	Register Name	Description
C_BASEADDR + 0x00	TCSR0	Control/Status Register for Timer 0
C_BASEADDR + 0x04	TLR0	Load register for Timer 0. Used to hold the reset value and also to hold the capture value of the timer in capture mode
C_BASEADDR + 0x08	TCR0	Timer/Counter Register for Timer 0
C_BASEADDR + 0x10	TCSR1	Control/Status Register for Timer 1
C_BASEADDR + 0x14	TLR1	Load Register for Timer 1
C_BASEADDR + 0x18	TCR1	Timer/Counter Register for Timer 1

The timers may be configured in one of 4 modes of operation:

• **Cascade mode** in which each 32bit timer/counter register is cascaded together to form a 64bit counter. This mode is useful when it is necessary to have a count capability of greater than that provided by a single 32bit wide counter.

• **Capture mode** in which the current counter value is loaded into the load register when an external input is applied to the Trigger input of the counter.

• **Generate mode** in which the counter may produce a one clock cycle pulse on the Generate output of the counter when the a carry-out is generated from the counter. This mode is useful in situations requiring a periodic pulse to be generated.

• **PWM (pulse width modulation) mode** in which the two timers are used to produce a PWM signal. One timer (Timer 0) is used to specify the frequency of the pulse and the other timer (Timer1) is used to specify the pulse duty cycle.

The timers must first be configured by writing the appropriate values to the control/status registers. Table 7 on page 13 of the Timer Product Specification summarizes the control and status bits for timer 0. The general steps required to configure the timer are:

1. Set the mode of the counter to be one of the 4 available modes describe above. Specific bits in the control/status register are used to program a particular mode. For example, bit 0 is used to set the timer to be either in generate mode (bit 0 = 0) or capture mode (bit 0 = 1). Bit 9 is used to program the counter for PWM operation, and bit 11 is used to configure the counter for cascade operation.

2. Configure the counter to be either an up-counter or a down counter by setting bit 1 as appropriate (bit 1 = 0 for up counter, bit 1 = 1 for down counter)

3. Write a value into the timer load register (TLR) . This can be accomplished with a simple pointer dereference operation:

```
u32* Timer_Ptr =  (u32*) XPAR_TMRCTR_0_BASEADDR; // value defined
                                                 // in xparameters.h

// load 0 as the reset value into the load register of the counter

 *(Timer_Ptr + 1) = 0;
```

4. The value stored in the load register is then subsequently loaded into the timer/counter register (TCR) by asserting bit 5 of the control/status register. Careful reading of the datasheet reveals “setting this bit loads the timer/counter register (TCR) with a specified value in the timer/counter load register (TLR). *This bit prevents the running of the timer/counter*; hence this bit should be cleared alongside setting Enable Timer/Counter bit (bit7) in the control/status register “ [2]

5. To start the timer, set bit 7 of the status/control register to 1.

6. To stop the timer, set bit 7 of the status/control register to 0.

7. To read the counter value, simply use a pointer to dereference the counter register found at offset 8 bytes from the base address of the timer:

```
unsigned int count;
count =  *(Timer_Ptr + 2) ; // 2 ints = 8 bytes offset
```

Depending upon which mode is being used, other bits in the control/status register may be needed to be set or cleared as appropriate. The following program uses this alternative method of config-

uring and using the timer/counter (as opposed to the previous program which made use of the Xilinx provided functions). The program configures the timer counter in capture mode with a reset value of 0 as an up counter. It then uses the timer to measure the number of clocks required to perform a GPIO DiscreteWrite(). It then reloads the counter with the value stored in the load register and measures the time required to perform two memory write operations. You should study this program together with Table 7 of the timer datasheet to understand its operation:

```
#include “stdbool.h”

#include “xparameters.h”
#include “xil_types.h”
#include “xgpio.h”
#include “xil_io.h”
#include “xil_exception.h”
#include “xtmrctr.h”

#include <iostream>
using namespace std;

int main()
{

    static XGpio GPIOInstance_Ptr;

    u32* Timer_Ptr =  (u32*)XPAR_TMRCTR_0_BASEADDR;

 // base address = control/status reg
 // base address + 4 bytes = load register value (reset value)
 // base address + 8 bytes  = counter register

    int xStatus;

    cout << “####  Ted counter Application Starts ####” << endl;


    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    //Step-1: AXI GPIO Initialization
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    xStatus = XGpio_Initialize(&GPIOInstance_Ptr,
XPAR_AXI_GPIO_FOR_OUTPUT_DEVICE_ID);
    if(xStatus != XST_SUCCESS)
    {
     cout << “GPIO A Initialization FAILED” << endl;
     return 1;
```

```
    }

    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    //Step-2: AXI GPIO Set the Direction
    //~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    //XGpio_SetDataDirection(XGpio *InstancePtr, unsigned Chan-
nel, u32 DirectionMask);
    //we use only channel 1, and 0 is the the parameter for output

    XGpio_SetDataDirection(&GPIOInstance_Ptr, 1, 0);


// set mode to capture mode and specify up-count
// 0x001 = capture
// 0x000 = generate mode

 *(Timer_Ptr) = 0x001;

// load 0 as the reset value into the load register of the counter

 *(Timer_Ptr + 1) = 0;

// now load this value into the counter register by setting load =
1 in control register

*(Timer_Ptr) = 0x021;

// put the load bit off to allow the counter to be enabled when
the enable bit = 1
// since when load = 1 , the counter is prevented from running.

// start the timer by setting enable = 1 ;

*(Timer_Ptr) = 0x081;

// perform a write to the GPIO output port

 XGpio_DiscreteWrite(&GPIOInstance_Ptr, 1, 0xA3);

// stop the timer by setting enable = 0;

*(Timer_Ptr) = 0x001;

unsigned int count;

// read the value of the counter register
```

```
 count =  *(Timer_Ptr + 2) ;

 cout << "Count = " <<  count << endl;

    cout << "Timer reloaded with 0" << endl;
    *(Timer_Ptr) = 0x021;
    *(Timer_Ptr) = 0x001;
    count = *(Timer_Ptr + 2) ;
    cout << "Counter after reload = "<<  count << endl;

    // determine the overhead in starting and stopping the counter

    *(Timer_Ptr) = 0x081; // start the timer
     *(Timer_Ptr) = 0x001; // stop the timer
    count =  *(Timer_Ptr + 2);
    cout << "Time to access memory two times to start and stop
timer Counter = " << count <<  endl;

    return 0;

}
```

Questions:

1. Use the timer to measure how long (in terms of number of clock cycles) it takes to perform a **XGpio_DiscreteWrite()** and compare this to the to the time required to write to the GPIO port using the pointer dereference method. What conclusions can you draw from the two times?

2. If we wish are only interested in using the counter to read the counter register at certain points in time, does it make any difference whether the counter is programmed to operated in Generate mode or Capture mode ? Justify your answer.

PART 2: Capture Mode

Figure 1 shows the shows the timer's input and outputs. In capture mode, the timer can capture the current count value and save it inside the timer load register whenever the value on the timer CaptureTrig input is a certain value (either '0' or '1').

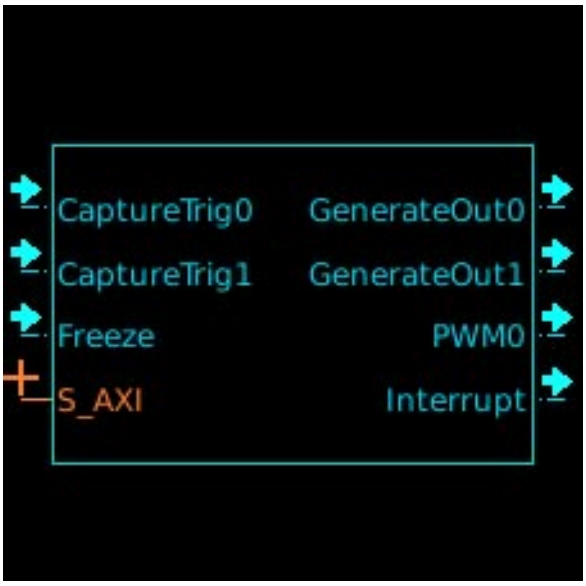

Figure 1: AXI Timer and its input/output ports.

In this part, we will use the timer in Capture mode to count the number of clocks between two successive presses of SW5. SW5 will be connected to the timers CaptureTrig0 input (by means of applying a constraint to this input via a UCF file). Open the lab3.ppr project in Planahead to modify the hardware.The required modifications to the system hardware are:

• In XPS, select the CaptureTrig0 port of the axi_timer_0 instance and specify: No Connection, followed by Make External. We will leave the External port name for this input as the default axi_timer_0_CaptureTrig0_pin

• In Planahead, generate a new top-level HDL by doubling clicking the system_I-system(system.xmp) source listed in the Design Sources area of the Project Manager pane. It is necessary to regenerate the system_stub.vhd since we have changed our hardware configuration (by making the capture trigger input as an external port). Once the system_stub.vhd has been regenerated, verify that the trigger input of the timer is listed as in input port in the entity:

```
 processing_system7_0_DDR_VRN : inout std_logic;
processing_system7_0_DDR_VRP : inout std_logic;
axi_gpio_for_output_pins : out std_logic_vector(7 downto 0);
axi_timer_0_CaptureTrig0_pin : in std_logic
 );
end system_stub;
```

• Open and modify the UCF file in the PlanAhead text editor so that the axi_timer_0_CaptureTrig0_pin input is constrained to location G19 of the chip which is wired to SW5 on the ZC702 development board. Add the following line to the UCF file:

```
NET axi_timer_0_CaptureTrig0_pin  LOC = G19 ;
```

Make sure you save the UCF file (if there is a small * above the name of the UCF file) within planAhead this indicates the file is not saved. Save it by selecting File -> Save File.

•Resynthesize, implement, generate a new bitstream, and program the FPGA with the new bit-stream using Impact.

• Export the new hardware to SDK.

Refer to the timer datasheet in order to set it up for Capture mode. Write a new version of main.cc (make a backup copy of your original) which performs the following pseudocode description:

Initialize the timer to be in capture mode with overwriting of each capture value and enable capture mode (HINT : write the hex value 0x19 to the timer control/status register);

load 0 as the initial reset value into the load register of the counter;

load the reset value into the counter (remember to clear the load bit to 0 afterwards so as to enable to counter);

start the timer by setting enable = 1 ;

```
while( the value of the load register is still equal to 0 )
{
  // wait until user presses SW5
}
```

```
capture_value = *(Timer_Ptr + 1) ; // read the captured value of the counter from the load
                                   // register
```

prompt the user to press SW% when they wish to capture a second counter value into the load register;

```
while( the value of the load register is still equal to capture_value)
{
  // keep on waiting until SW5 is pressed
}
```

```
new_capture_value =  *(Timer_Ptr + 1);
display the difference between new_capture_value and capture_value.
```

Switch Bounce.

Mechanical push button switches exhibit a characteristic known as switch bounce. Although we may depress a switch once and release it, due to bounce the internal switch contact may make contact with the switch terminal numerous times resulting in multiple pulses being generated instead of a clean logic ‘0’ ---> logic ‘1’ ----> logic ‘0’ transition. The timing diagram in Figure 2 illustrates a ‘clean’ switch (one with no bounce) and a ‘bouncy” switch. In reality, the actual waveform of a typical switch bounce is more complex as the duration of the pulses due to the bouncing of the mechanical switch contacts vary as indicated by Figure 3 and 4. There may even be bounce at the beginning of the transition (as well as at the end). Regardless, due to switch bounce, instead of a clean single pulse, several pulses may be generated even though the switch may have been pressed and released only one time.

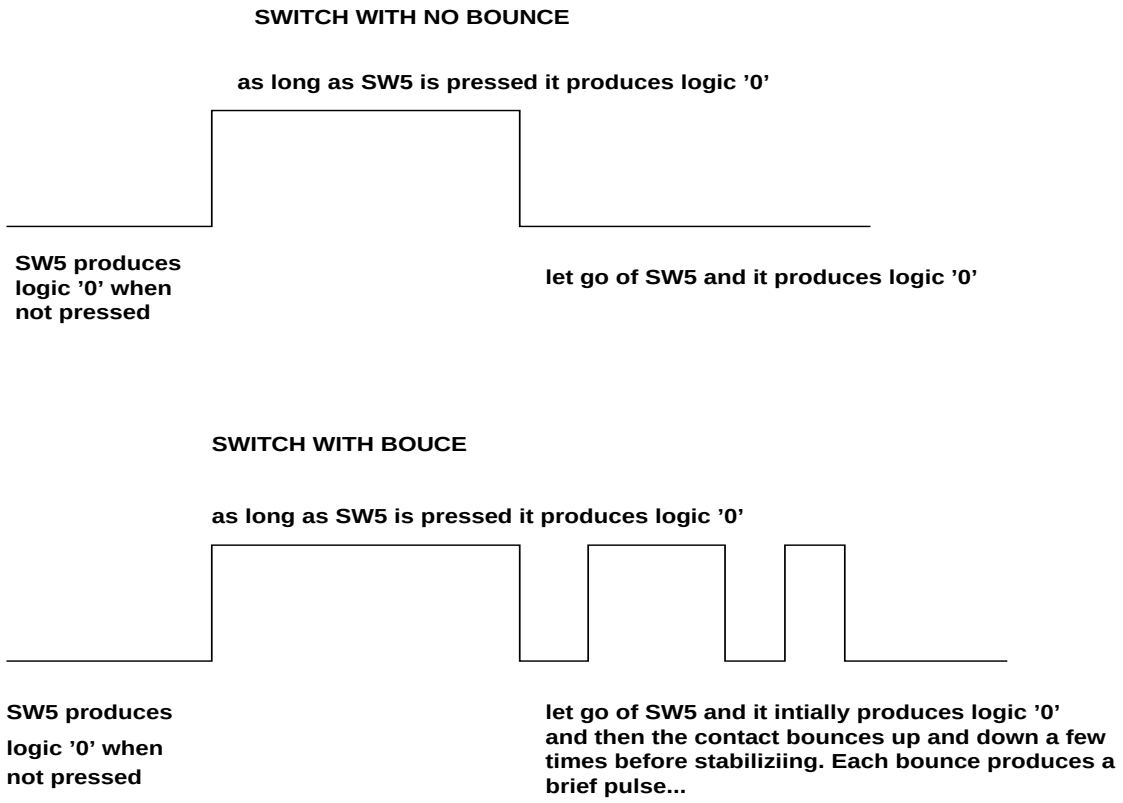

Figure 2: Switch bounce.

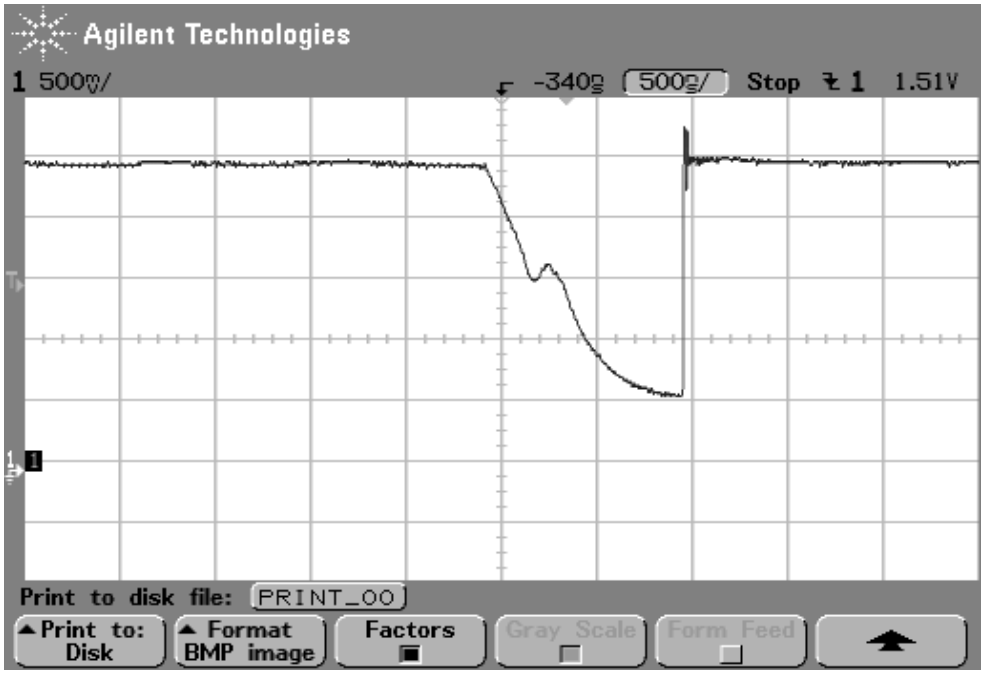

Figure 3: ZC702 Switch 5 bounce as measured with a digital storage oscilloscope.

Figure 4: Oscilloscope trace indicating actual switch bounce [3].

Bounce may be eliminated either through hardware or software. The push button switches on the ZC702 are not hardware debounced, therefore we have to account for switch bounce in our software. The pseudocode given earlier does not account for switch bounce. Hence, you may notice that even though you only press SW5 once, the program generates the final output without the user ever having physically pressed SW5 a second time. Typical bounce time for the pushbutton switches found on the ZC702 board are on the order of 0.1 - 5 ms [4].

Bonus Question: (total lab grade to not exceed 10/10 even if you answer the Bonus).

Rewrite the program so that it performs debouncing of SW5 in software.

Hint: Introduce a short delay between the two while loops in the previously given pseudocode to allow for any switch bounce to dissipate. You may have to experiment with the exact value of the delay.

Hint Hint: Use the current value of the timer's counter register to implement the above delay. Recall that the clock frequency to the counter is 50 MHz, this means that 25 000 000 clocks is equal to a delay of 0.5 seconds; ample time for any bounce to dissipate.

Part 3: Pulse Width Modulation

Pulse Width Modulation (PWM) is a technique which is used to generate a signal which repeatedly switches between a logic high and a logic low value. Both the period of the signal and the duty cycle (the portion of time that the signal of high) are controlled. [5]. Figure 5 illustrates a PWM signal with duty cycles of 80%, 50%, and 10 %.

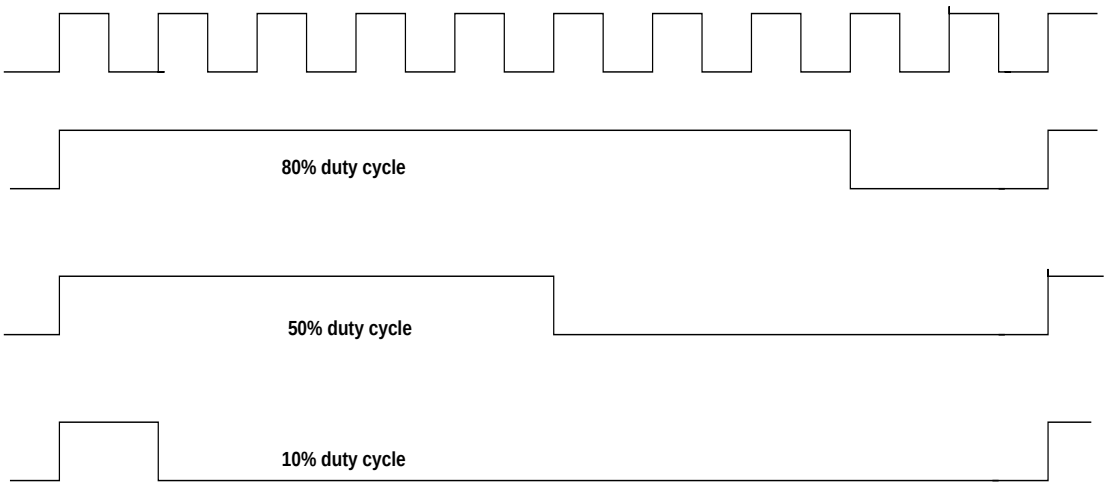

Figure 5: PWM signals with varying duty cycles.

The AXI Timer can be configured to provide a pulse-width modulated output (output port PWM0). When the timer is programmed in PWM mode, timer0 is used to control the period of

the PWM0 output and timer1 is used to specify the duty cycle according to the following formulas:

PWM_PERIOD (in seconds) = (value in Timer0 load register + 2) x AXI_CLOCK_PERIOD

PWM_HIGH_TIME (in seconds) = (value in Timer1 load register + 2) x AXI_CLOCK_PERIOD. *

(* with timers programmed to be in count down mode, refer to datasheet)

With a default clock frequency of 50MHz, if a period of 5 seconds with an on time of 2.5 seconds (corresponding to a 50% duty cycle) is desired, we would load the timer load registers with the:

TLR0 = (5 x 50 000 000) - 2 = 249 999 998

TLR1 = (2.5 x 50 000 000) -2 = 129 999 998

Questions:

1. Create a new project which uses the time in PWM mode to make one of the LEDs blink at some user-specified frequency with a user-specified on time. The required modifications to the capture-mode lab are:

• In XPS, make the PWM0 port of the AXI Timer as an External Port (select it and specify No Connection , followed by Make External).

• In the External Ports of XPS, rename the External Port associated to PWM.

• Constrain the PWM0 output to be connected to one of the user LEDs on the ZC702 board by adding it to a new UCF file:

NET "PWM" LOC = D15;

• Create the top HDL in PlanAhead.

• Synthesize, implement, and generate the bitstream.

• Program the FPGA with the bitstream using Impact.

• Export the hardware to SDK.

• In SDK, create a new application project.

• Refer to the AXI Timer Product Specification for the specifics of how to configure the timer in PWM mode.

• Write a new application program which performs the following:

configure the timer for PWM mode;

while(true)
{
Ask user to input the desired period of the PWM signal and the desired duty cycle (either in seconds or as a percentage of the period) ;

load the appropriate values into the TLR0 and TLR1 registers as per the formulas given on page 4 of the datasheet

start both timers;

}

References

1. LogiCORE IP AXI Timer Product Specification, DS764, July 25, 2012, Xilinx Inc. , p.11.

2. LogiCORE IP AXI Timer Product Specification, DS764, July 25, 2012, Xilinx Inc. , Table 7, p.14.

3. Photo by M. Segev, July 2014.

4. http://web.engr.oregonstate.edu/~traylor/ece473/lectures/debounce.pdf

5. Embedded System Design - A Unified Hardware/Software Introduction, Frank Vahid, Tony Givargis, John Wiley and Sons, Inc. 2002, p.92.

T. Obuchowicz/R. Lee
July 2014

Revision History:

- Sept. 8, 2016: Revised for Lab 3 Fall 2016. Part 1 and 2 revised as pre-lab. Part 3 to be performed in lab session. Revised for sourcing /CMC/tools/xilinx_14.7/14.7/ISE_DS/settings64_CMC_central_license.csh.