Introduction

In this lab, you will design a Nios2-based System-on-Chip (SoC) platform and develop associated software that will use Pulse Width Modulation (PWM) to modulate the intensity of the 26 LEDs (18 red and 8 green) that run across the bottom left side of the DE2-115 board.

Your project will be graded based on a demo to one of the TAs or the instructor.

Your Platform

Use Quartus Prime and Platform Designer to design a system consisting of the following components:

- A **Clock Source**, which accepts an external clock and reset signal (already added by default)
- A Nios2/f processor
- A **System and SDRAM Clocks for DE-Series Boards**, which accepts the external clock as a reference and generates a 50 MHz system clock and a phase-adjusted SDRAM clock
- An **SDRAM Controller Intel FPGA** with 13 row bits, 10 column bits, 4 banks, and a 32-bit width (i.e. a 8192x1024x4x4 = 128 MB RAM, which allows the Nios2 processor to access instructions and data from external SDRAM memory
- A 26-bit **PIO (Parallel I/O) Intel FPGA IP** I/O interface to the 26 LEDs
- A second **Clock Source**, which outputs the SDRAM clock to an external pin (external to both the platform design and the FPGA as a whole
- A **Performance Counter Unit Intel FPGA IP**, which we will use as a hardware cycle counter to use as a reference against which software will generate a PWM signal
- A **JTAG UART Intel FPGA IP** for printing messages to the console

Once you've added these components, make the following connections:

- From the first **Clock Source**, connect the *clk* output to the *ref_clk* input on the **System and SDRAM Clocks for DE-Series Boards**
- From the first **Clock Source**, connect the *clk_reset* output to the *reset* (or *ref_reset*) inputs on all the other components
- From the **System and SDRAM Clocks for DE-Series Boards**, connect the *sys_clk* output to the *clk* input on all the other components **except for the SDRAM Controller Intel FPGA** and the second **Clock Source**
- From the **System and SDRAM Clocks for DE-Series Boards**, connect the *sdram_clk* output to the *clk* input on the **SDRAM Controller Intel FPGA** and the second **Clock Source**
- From the **SDRAM Controller Intel FPGA**, double click the "Export" column foir the "Conduit" and then press Enter

- From the Nios2 processor, connect the *Instruction Master* to the *s1* port on the **SDRAM Controller Intel FPGA**
- From the Nios2 processor, connect the *Data Master* to the *s1* port on the **SDRAM Controller Intel FPGA**, as well as the *Avalon Memory Mapped Slave* on the *JTAG UART*, the *control_slace* on the **Performance Counter Unit**, and the *s1* port on the **Parallel IO**
- From the **Parallel IO**, double click the "Export" column for the "Conduit", type "leds", and then press Enter
- Rename the **Parallel IO** as "LEDS"

Make sure that you verify your connections against both the lecture slides and the tutorial given in lecture and lab, posted on Youtube.

Your Code

Your software will perform two tasks:

1. Generate a PWM signal that will control the brightness of all 26 LEDs on the DE2-115 board
2. Linearly vary the PWM duty cycle to vary the brightness of the LEDs according to a fixed period, as given by the value set to a variable

To do this, your code should run in an infinite loop that simultaneously controls the PWM signal as well as the duty cycle of the PWM signal.

Your code should have two constant global variables whose values can be modified by a user and reflected in the operation after the code is recompiled and re-deployed on the platform running on the DE-115.

The variables should be declared and set with example values as shown below:

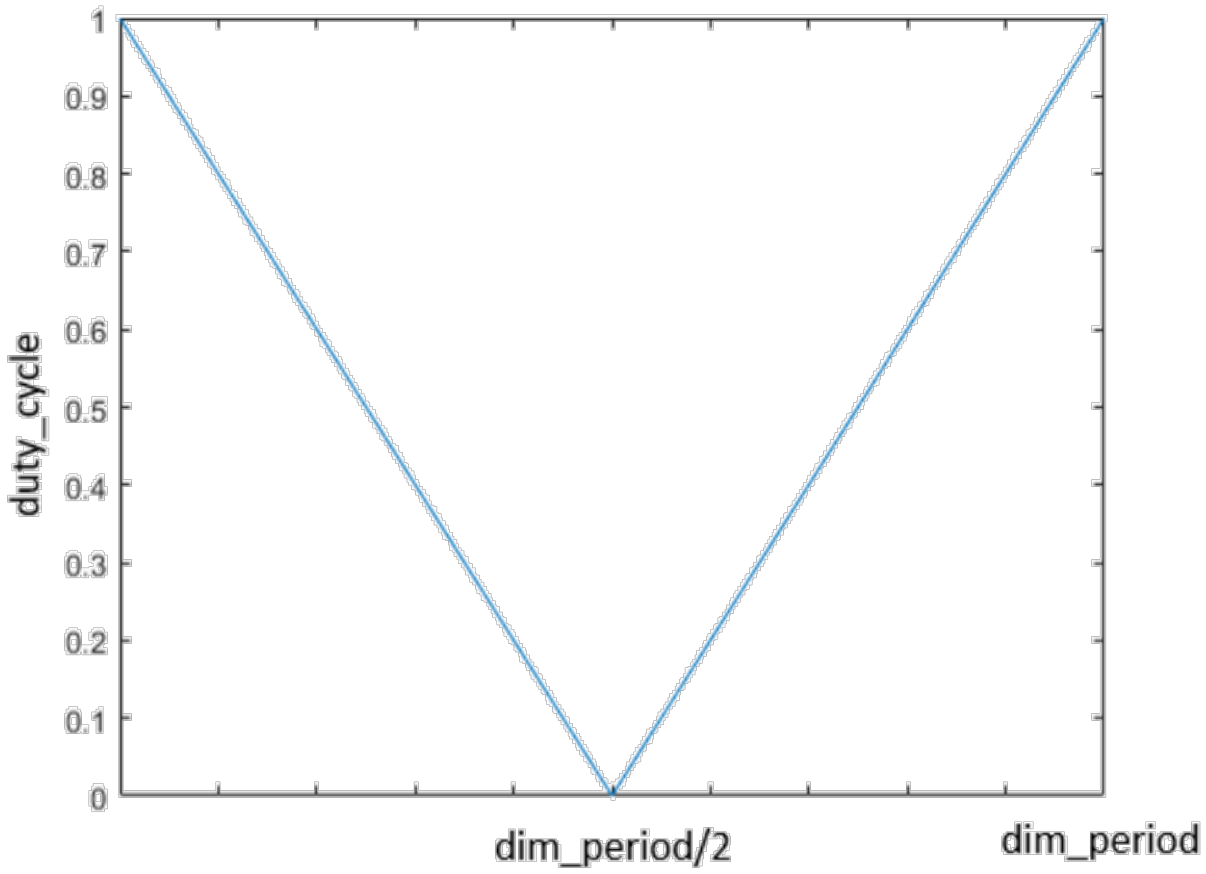
```
1 float pwm_frequency = 10e3f;
2 float dim_period = 2.0f;
```

The `pwm_frequency` variable defines the PWM frequency in Hz; that is, the inverse of which is the period of the PWM signal in seconds. From `pwm_frequency` your code can compute the `pwm_period` as `1.0f/pwm_frequency`.

The pwm_period can be expressed in terms of clock cycles as:

```
1 alt_u64 pwm_period_in_cycles = (alt_u64)(pwm_period * (float)
      ALT_CPU_FREQ);
```

The `dim_period` variable defines the length of time required for the PWM duty cycle to be swept from 100% to 0% and back to 100%.



The dim_period can be converted to cycles as:

```
alt_u64 dim_period_in_cycles = (alt_u64)(dim_period * (float)
    ALT_CPU_FREQ);
```

The needed PWM duty cycle, which determines the brightness of the LEDs, can be computed as:

```
float current_duty_cycle = fabs((cycle_counter % dim_period_in_cycles)
    *(-2.f/dim_period_in_cycles) + 1.0f);
```

Additional Requirements

Be sure to include the following header files:

```
#include <stdio.h>
// Provides printf().

#include <system.h>
```



```
// Provides symbols for the base addresses of all peripherals (e.g.
// PERFORMANCE_COUNTER_0_BASE and LEDS_BASE and the CPU frequency
// ALT_CPU_FREQ.  It also includes io.h, which provides IO
// intrinsics IORD() and IOWR().

#include <alt_types.h>
// Provides the standard types, such as alt_u32.

#include <io.h>
// Provides the IORD() and IOWR() instrinsics for the ldwio and stwio
    instructions.
```

Parallel I/O Module

The output value of the parallel I/O module can be changed using the IOWR() instrinsic with its base address. For example, the following code will turn on all the LEDs, assuming all 26 LEDs are wired to a single 26-bit parallel I/O:

```
IOWR(LEDS_BASE,0,0x3FFFFFF); // note the second argument should be 0
                             // because there is only one control
                                register
```

Performance Counter Module

The performance counter module contains several control and status registers that can similarly be accessed with IORD() and IOWR().

- When read, register 0 contains the lower 32 bits of the cycle counter
- When written, register 0 stops the counter when the value written is 0 and resets the counter when the value written is 1
- When read, register 1 contains the upper 32 buts on the cycle counter
- When written, register 1 starts the counter when the value written is 0

You can use the following code to access the performance counter:

```
// Start the counter; do this only once at the beginning of your main()
     function
// Note that this writes the value 0 to register 1 on the performance
    counter module.
```



```
// This assumes you used the default name for the performance counter, which is
// "performance_counter_0".

IOWR(PERFORMANCE_COUNTER_0_BASE,1,0);

// Read the counter.  Begin with declaring a 64-bit value that can be split between
// high and lo halves.
union {
  struct {
    alt_u32 lo;
    alt_u32 hi;
  } parts;
  alt_u64 raw;
} cnt;

// Read upper 32 bits
cnt.parts.hi = IORD(PERFORMANCE_COUNTER_0_BASE,1);

// Read lower 32 bits
cnt.parts.lo = IORD(PERFORMANCE_COUNTER_0_BASE,0);

// Refer to both halves as "cnts.raw"
```

Requirements

During your demo, we will ask you to change both the PWM frequency and the dim period to demonstrate that your code works. We will verify the correctness of your code by visual inspection of the DE2-115 board while in operation. We will also ask you questions about how your code works.

Submitting Your Code

When you are ready to turn in your code for grading, each student should submit their C code to the submission box on Dropbox. You do not need to submit your hardware design.

Rubric

- Correct operation - 50 points
- Use of performance counter as clock - 20 points
- Demonstration of modification of PWM frequency and dim period - 30 points

Total points possible: 100.

Getting Started

This project requires several steps to set up your environment before you get started:

1. Create a subdirectory for your project with a name of, for example, “lights”
2. Create a new Quartus project with a name of, for example, “lights”
3. Within that project, import the assignments from the file “DE2_115_pin_assignments.qsf” available on the course Dropbox
4. Add the file “lights.sv” to the project, also available on the course Dropbox
5. Add the file SDC1.sdc file to project using the file contents shown below
6. Create a new Platform Designer project and add the modules listed above, save the file as “nios_system.qsys”
7. Create a “software” directory in your project directory, in that directory create your skeleton software project and BSP using the commands below
8. In the “software/lights” directory, edit the “hello_world.c” file with your project code
9. To test, start the jtag daemon using the command shown below

List of Common Commands

Contents of SDC1.sdc:

```
create_clock -name CLOCK_50 -period 20 [get_ports CLOCK_50]
derive_pll_clocks -create_base_clocks
```

Create app and BSP skeletons:

```
cd lights
mkdir software
cd software
```



```
nios2-swexample-create --name=lights --sopc-file=../nios_system.sopcinfo --type=hello_world --cpuname=nios2_gen2_0 --app-dir=lights --bsp-dir=lights_bsp
cd lights
./create-this-app
```

Start the JTAG daemon (do this before configuring the FPGA, downloading your software, opening a JTAG UART terminal, or starting the GDB debug server:

```
LD_LIBRARY_PATH="/usr/local/3rdparty/altera/18.1/quartus/linux64" jtagd --config /usr/local/3rdparty/altera/18.1/quartus/linux64/pgm_parts.txt
```

Program the FPGA (creates the platform):

```
nios2-configure-sof ~/lights/output_files/lights.sof
```

Download and run the software onto your CPU:

```
# change to your app directory (e.g ~/lights/software/lights)
nios2-download -g lights.elf
```

Open the JTAG UART console:

```
nios2-terminal
```

Start the debugger:

```
nios2-gdb-server --tcpport 8888 --tcppersist

# change to your app directory (e.g ~/lights/software/lights)
nios2-elf-gdb lights.elf
remote target localhost:8888
load
b main
c
```

Change to app directory

```
nios2-download -g lights.elf
```

