Final Project: PID Controller of 12V Fan

Sarah Hodges & Ryan Marcum

ENGC 401

Dr. Feng Wang

Liberty University

4 May 2023

Table of Contents

[Abstract 4](#_Toc133929190)

[Introduction – ABET Outcome #1 4](#_Toc133929191)

[Problem Identification 4](#_Toc133929192)

[Background – ABET Outcome #7 4](#_Toc133929193)

[PID 4](#_Toc133929194)

[Proportional Response 5](#_Toc133929195)

[Integral Response 5](#_Toc133929196)

[Derivative Response 6](#_Toc133929197)

[Applications 6](#_Toc133929198)

[Metacognition 6](#_Toc133929199)

[Methodology 7](#_Toc133929200)

[Principles of Engineering 7](#_Toc133929201)

[Principles of Science 7](#_Toc133929202)

[Principles of Mathematics 8](#_Toc133929203)

[Design 9](#_Toc133929204)

[Knowledge/Techniques/Skills/Tools Required 9](#_Toc133929205)

[Main Components 9](#_Toc133929206)

[Block Diagram 9](#_Toc133929207)

[ASMD Chart 9](#_Toc133929208)

[Fan\_control.c 9](#_Toc133929209)

[pid.c 9](#_Toc133929210)

[pwm.v 9](#_Toc133929211)

[Experimental Results 9](#_Toc133929212)

[Conclusion 9](#_Toc133929213)

[References 10](#_Toc133929214)

[Appendix 11](#_Toc133929215)

[Fan\_control.c 11](#_Toc133929216)

[pid.c 11](#_Toc133929217)

[pwm.v 11](#_Toc133929218)

# Abstract

The goal of this project was to develop a PID to control a 12V DC fan with a low and high speed. The change in these speeds were based off the read in voltage values from the LM60 temperature sensor which were converted to Celsius and Fahrenheit. Additionally, the duty cycle would be determined, printed to the MTL2 screen, and updated when a different threshold value was reached.

# Introduction – ABET Outcome #1

## Problem Identification

In this project, a PID controller of a 12V fan was developed for the purpose of cooling the ARM chip on the DE1-SoC board. Involved in this are a LM60 temperature sensor, 9V battery, battery connectors, a L9110 driver, 12V fan, DE1-SoC board, breadboard, jumper wires, ethernet, and external laptop. The way that this PID works is by first reading the temperature of the ARM chip and then determining whether the fan should be turned on in high or low mode. The temperature is read by the sensor as voltage that must be converted to Celsius ( C ) and Fahrenheit ( F ) which is an application of physics. Being able to combine the components provided with hardware developed in Quartus and C code is an application of engineering. Additionally, in order to understand how PID works requires multiple mathematical concepts. Therefore, since this project requires multiple steps and knowledge from engineering, physics, and mathematics; this project is considered to be a complex engineering problem.

# Background – ABET Outcome #7

## PID

PID stands for Proportional-Integral-Derivative and is used to control systems such as refrigerators or in this case a 12V DC fan. The process by which this is done is by reading the input data from the sensor, calculating the proportional/integral/derivative responses, and determining the output through summation of the PID computations (*The PID Controller & Theory explained*). Specifically, the PID acts as a feedback loop, taking the data from the sensor and comparing it to a set value and then reacts based off where that value falls in reference to the set value. In the case of a refrigeration PID controller, if the read value is above the set value, then the fan will blow cooler air until the read temperature reaches the desired set value which will cause the controller to turn off the fan. A similar cause and reaction happen when the read temperature is lower than the set value, but opposite (*Application of PID controller in controlling refrigerator temperature* 2009).

## Proportional Response

The first part of a PID controller is that of the proportional response which focuses of the deviation (aka the error term) between what value is read by the sensor and the set value(s) in the code. This component, “proportional gain determines the ratio of output response to the error signal” (*The PID Controller & Theory explained*) which directly corelates to the reaction of the controller. This means that as the proportional gain increases, so will the “speed of the control system response” (*The PID Controller & Theory explained*). However, if the proportional gain increases too much too fast, then the controller will become unstable.

## Integral Response

Second is the integral component which “sums the error term over time” (*The PID Controller & Theory explained*). With this component, unless the error is zero then even the smallest deviation from the set value will cause the component to increase (*The PID Controller & Theory explained*). This causes the steady-state error to decrease, however, this can be damaging because it can cause the system to be more oscillatory. This is due to the time that it takes the integrator to “unwind” from decreasing the steady-state error (*Introduction: Pid controller design*).

## Derivative Response

The third component of PID is that of the derivative response which handles the control signal and gives the system the ability to anticipate deviation. This means that with the derivative component, the control signal can increase when the deviation is relatively small in the case of the deviation beginning to slope upwards. It does so by anticipating that the error will continue to increase and thusly increase the control signal (*Introduction: Pid controller design*). However, this component’s anticipation of the deviation makes the system more sensitive to change and if there is too much noise, the derivative component will cause the system to become unstable. Therefore, the derivative component is generally kept small (*The PID Controller & Theory explained*).

## Applications

There are many applications for PIDs in today’s world from thermostats to robots. PIDs can be used to control the temperature of refrigerators can be used to control the output in a digital format (*Application of PID controller in controlling refrigerator temperature* 2009). Additionally, PIDs can be used to control servo motors as well and can even be modified to also be modified to use a neural network. This allows the controller to be able to “control different systems through quick learning process[es]” (Lin & Peng, 2010).

## Metacognition

In order to learn the background of how PIDs work and what they are used for, I applied research skills and metacognition. Specifically, we used the Jerry Falwell Library (JFL) and IEEE data bases to search for journal articles and books on PIDs and their applications. Additionally, we were also able to find credible websites, including those from other universities that explained different parts of PIDs to better understand how PIDs work and used knowledge gained through those sites to comprehend the technical details from the journal articles. From there, we applied the theoretical knowledge gained through research to the complex engineering problem at hand to complete the objectives and requirements of this project.

# Methodology

## Principles of Engineering

To complete this lab, both hardware and software needed to be created to communicate through the board and to the 12V DC fan. The hardware side is needed for PWM and PID control for the duty cycle. On the software side, the ADC and PID are created to control the high and low speeds of the fan as well as convert the voltage read by the LM60 temperature sensor to Celsius and Fahrenheit. To accomplish both the hardware and software tasks, we had to utilize the knowledge learned across our computer engineering and computer science classes.

## Principles of Science

When setting up and testing the LM60 temperature sensor, we discovered that it read the input value of the FPGA chip in volts instead of Celsius or Fahrenheit. Therefore, we found the formula that relates temperature and voltage which can be seen in Equation 1. In order to apply this equation to our ADC, we needed to shift it around to make TC the term on the left side for calculations. After doing this, the equation that we used to convert voltage to Celsius is Equation 2 and the code for it can be seen in Figure 1. Due to us being more familiar with Fahrenheit at a glance, we also converted the Celsius temperature to Fahrenheit with Equation 3 and the code for this can also be seen in Figure 1.

Text

Description automatically generated

Equation 1: Formula Relating Voltage and Temperature

A picture containing logo

Description automatically generated

Equation 2: Formula to Convert Voltage to Celsius

Text

Description automatically generated with medium confidence

Equation 3: Formula for Converting Celsius to Fahrenheit

Graphical user interface

Description automatically generated

Figure 1: Code for Converting Voltage to Celsius

## Principles of Mathematics

As previously discussed, a PID consists of three components: proportional response (KP), integral response (KI), and a derivative response (KD). First, KP provides the control action proportional to the derivation and is the proportion of the derivation times the process variable. Second, KI reduces steady-state derivations through the use of integration over time. Third, KD improves the transient response through differentiation with respect to time. Additionally, y(t) in Equation 4 is the process variable and e(τ) is the set value minus the process variable. The whole formula with these three components can be seen in Equation 4 (Kiam Heong Ang et al., 2005).

Diagram

Description automatically generated with medium confidence

Equation 4: Formula for PID

# Design

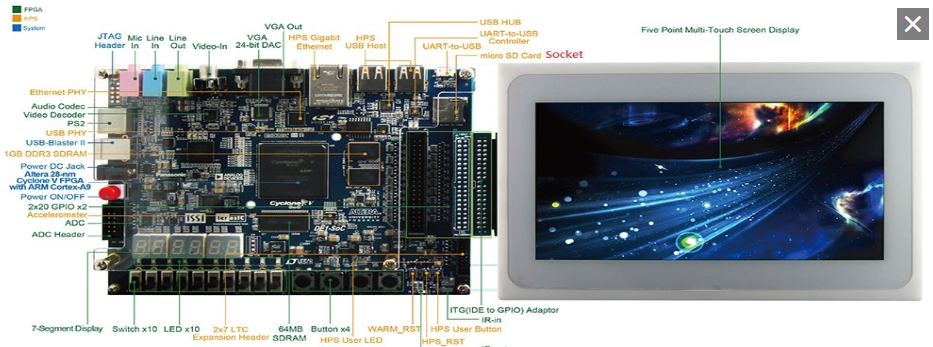
## Knowledge/Techniques/Skills/Tools Required

This project requires knowledge and skills in multiple fields to complete. Below is a list of these that are needed to be able to complete the project:

* Knowing how to code using an HDL(Hardware Description Language)
* Knowing how to code in c and using addressing to access hardware
* Coding and compiling code in a Linux environment
* Knowing how to use a DE1-SOC board
* Knowledge in electrical engineering for designing hardware connections
* Knowledge in knowing how to debug code
* Knowledge in advanced mathematics for using formulas and for PID design

## Main Components

The main components used in this project are as follows:

* DE1-SOC with MLT
* Bread board

Calendar

Description automatically generated

* LM60 Temperature Sensor

A picture containing piano

Description automatically generated

* L9110 Dual Channel Motor Driver Module

A close-up of a computer chip

Description automatically generated with low confidence

* 12v Fan

Text

Description automatically generated with medium confidence

* Jumper Wires

A picture containing seat, chair

Description automatically generated

* 9v Battery Connector

A picture containing diagram

Description automatically generated

* 9v Battery

Text

Description automatically generated with low confidence

## Block Diagram

Chart, box and whisker chart

Description automatically generated

Figure 2: Block Diagram

## Fan\_control.c

## This section is a breakdown of the main c code used in the project. The first figure here figure 3 is the make file used to compile the code for the project.

A screen shot of a computer

Description automatically generated with medium confidence

*Figure 3: Makefile*

Here in figure 4 the code get the current ADC value from the temperature sensor. It is converted then to degrees Celsius and sent to the PID update function and returns the updates duty cycle for the PWM. Then the temperature is converted to Fahrenheit and the duty and the 2 temperatures are printed on the terminal.

A picture containing text, screenshot, font

Description automatically generated

*Figure 4: ADC, Temperature, and Duty Calculations*

This next section in figure 5 takes the duty cycle calculated. This checks if it falls between certain range of values and then sets the duty cycle to 100%, 75%, 50%, 25%, or 0% duty and sends the updated duty cycle to the PWM.

A screenshot of a computer program

Description automatically generated with medium confidence

A picture containing text, screenshot, display, software

Description automatically generated

*Figure 5: Set Duty for PWM*

Next in figure 6 after the duty cycle is selected it is then printed on the on board monitor on the DE1-SOC board.

A picture containing text, screenshot

Description automatically generated

*Figure 6: Draw Duty Wave Form*

## pid.c

Pid.c here in figure 7 calculates the updated duty based on the inputs and returns the new duty cycle.

A picture containing text, electronics, screenshot, software

Description automatically generated

*Figure 7: PID Calculations*

## pwm.v

Below here in figure 8 is the wrapping circuit for the PWM. This section updates the DVSR and duty cycle for the PWM. This also sets the addressing for DVSR and Duty. DVSR is set to 0x0 and Duty is 0x10 up to 0x1f.

A screen shot of a computer code

Description automatically generated with low confidence

*Figure 8: PWM Wrapping Circuit*

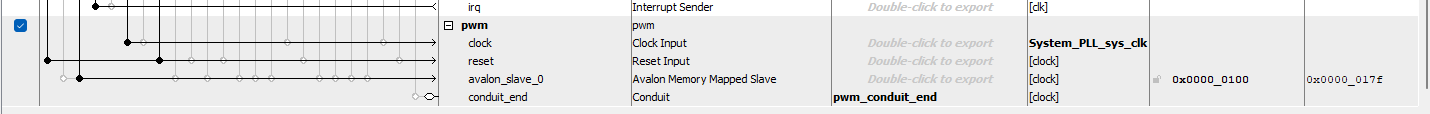
In figure 9 is the PWM hardware code. The first bit is a d flip-flop that updates the registers. Then in the bottom section calculates the tick and timing. Then lastly calculates the output for the PWM.

A screen shot of a computer code

Description automatically generated with low confidence

*Figure 9: PWM Hardware*

Then here in figure 10 is the platform design setup for the PWM. The clock and reset are connected to the main clock and reset for the system. The Avalon slave is connected to the lightweight bridge. Lastly the conduit end is exported and connected to pin GPIO\_0[1].



*Figure 10: PWM Platform Design*

# Experimental Results

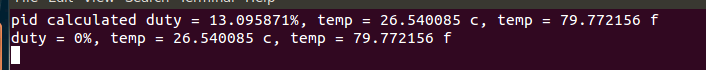
In order to run this project, the following setup (Figure 11) was required to connect the 12V fan, temperature sensor, 9V battery, and motor value to the DE1-SoC board and MTL2 screen.

A picture containing electronics

Description automatically generated

*Figure 11: Project Setup*

When run, the program updates the screen with the current calculated duty cycle and temperature in Celsius and Fahrenheit. Additionally, the duty cycle waveform is printed to the screen according to where the duty cycle falls in the threshold ranges. The 0%, 25%, 50%, 75%, and 100% duty cycles can be seen in Figures 12 – 16.



Text

Description automatically generated

*Figure 12: 0% Duty*



A picture containing text, indoor, oven, microwave

Description automatically generated

*Figure 13:25% Duty*



A screen shot of a computer

Description automatically generated with low confidence

*Figure 14: 50% Duty*



A screen shot of a computer

Description automatically generated with low confidence

*Figure 15: 75% Duty*



A picture containing text, display device, display, video

Description automatically generated

*Figure 16:100% Duty*

# Conclusion

In conclusion, this project allowed the students to answer ABET Outcomes #1 and #7 which ask to determine how this project is categorized as a complex engineering problem and whether the students can research and apply new knowledge. It is a complex engineering problem because of the different areas of engineering, science, and mathematics needed in order to complete the problem and the multiple steps required. This project utilized a PID with PWM and ADC to take in the voltage from the LM60 temperature sensor, converted it to Celsius and Fahrenheit, checked where the temperature value fell in respect to the set threshold values in the PID, and determined what duty cycle percentage then printed the waveform to the screen.

# References

4, E. G. said: J., & 6, J. M. said: J. (2014, February 23). *Temperature sensing with the LM60*. Geekily Interesting. Retrieved May 2, 2023, from <https://geekilyinteresting.wordpress.com/2014/02/23/temperature-sensing-with-the-lm60/>

Hamid, N. H., Kamal, M. M., & Yahaya, F. H. (2009). Application of PID controller in controlling refrigerator temperature. *2009 5th International Colloquium on Signal Processing & Its Applications*. <https://doi.org/10.1109/cspa.2009.5069255>

*Introduction: Pid controller design*. Control Tutorials for MATLAB and Simulink - Introduction: PID Controller Design. (n.d.). <https://ctms.engin.umich.edu/CTMS/index.php?example=Introduction§ion=ControlPID>

Kiam Heong Ang, Chong, G., & Yun Li. (2005). PID control system analysis, design, and Technology. *IEEE Transactions on Control Systems Technology*, *13*(4), 559–576. <https://doi.org/10.1109/tcst.2005.847331>

Lin, L., & Peng, X. (2010). A PID neural network control for permanent magnet synchronous motor servo system. *2010 5th International Conference on Computer Science & Education*. <https://doi.org/10.1109/iccse.2010.5593702>

*The PID Controller & Theory explained*. NI. (n.d.). <https://www.ni.com/en-us/shop/labview/pid-theory-explained.html>

# Appendix

## Fan\_control.c

## /\*

## PWM output is on GPIO0 D0 pin, like signal generator example.

## ADC is from A0

## \*/

## #include <stdio.h>

## #include <fcntl.h>

## #include <sys/mman.h>

## #include <signal.h>

## #include <time.h>

## #include <unistd.h>

## #include <stdlib.h>

## #include <stdbool.h>

## #include "physical.h"

## #include "address\_map\_arm.h"

## #include "PID.h"

## #include <math.h>

## // === FPGA side ===

## #define HW\_REGS\_BASE ( 0xff200000 )

## #define HW\_REGS\_SPAN ( 0x00200000 )

## #define HW\_REGS\_MASK ( HW\_REGS\_SPAN - 1 )

## /\* Controller parameters \*/

## #define PID\_PARAM\_KP 9

## #define PID\_PARAM\_KI 0.025

## #define PID\_PARAM\_KD 20

## 

## /\* address and field definition \*/

## #define PWM\_DVSR\_REG 0x0

## #define PWM\_DUTY\_REG 0x10

## 

## // character buffer

## volatile unsigned int \* vga\_char\_ptr = NULL ;

## void \*vga\_char\_virtual\_base;

## 

## /\* Prototypes for functions used for video display \*/

## void wait\_for\_vsync(volatile int \*);

## void get\_screen\_specs (volatile int \*);

## void clear\_screen(void);

## void plot\_pixel(int, int, int);

## void drawline (int, int, int, int, int);

## void VGA\_text (int, int, char \*);

## void VGA\_text\_clear();

## int pixel\_buffer\_start; // location of the pixel buffer video memory

## int main(void)

## {

## /\*\*\*\*\*\*\*\*\*\*

## \*Variables\*

## \*\*\*\*\*\*\*\*\*\*\*/

## int fd = -1; // used to open /dev/mem for access to physical addresses

## void \*LW\_virtual; // used to map physical addresses for the light-weight bridge

## void \*SDRAM\_virtual;

## volatile int \* pixel\_ctrl\_ptr; // virtual address of pixel buffer DMA controller

## volatile unsigned int \*adc;

## int res = 4;

## volatile bool change = true;

## volatile bool \_100 = true;

## volatile bool \_75 = true;

## volatile bool \_50 = true;

## volatile bool \_25 = true;

## volatile bool \_0 = true;

## volatile unsigned int ADC\_Value;

## 

## // === FPGA ===

## volatile unsigned int \*PWM\_addr = NULL;

## 

## /\* ARM PID Instance, float\_32 format \*/

## PIDController PID;

## 

## // Create virtual memory access to the FPGA light-weight bridge by calling fpga\_init() function

## if ((fd = open\_physical (fd)) == -1)

## return (-1);

## else if ((LW\_virtual = map\_physical (fd, LW\_BRIDGE\_BASE, LW\_BRIDGE\_SPAN)) == NULL)

## return (-1);

## 

## // Create virtual memory access to the SDRAM memory

## SDRAM\_virtual = mmap( NULL, SDRAM\_SPAN, ( PROT\_READ | PROT\_WRITE ), MAP\_SHARED, fd, SDRAM\_BASE);

## if (SDRAM\_virtual == NULL)

## return (0);

## 

## // === get VGA char addr =====================

## // get virtual addr that maps to physical

## vga\_char\_virtual\_base = mmap( NULL, FPGA\_CHAR\_SPAN, ( PROT\_READ | PROT\_WRITE ), MAP\_SHARED, fd, FPGA\_CHAR\_BASE );

## if( vga\_char\_virtual\_base == MAP\_FAILED ) {

## printf( "ERROR: mmap2() failed...\n" );

## close( fd );

## return(1);

## }

## 

## // Get the address that maps to the FPGA LED control

## vga\_char\_ptr =(unsigned int \*)(vga\_char\_virtual\_base);

## 

## PWM\_addr = (volatile unsigned int \*)(LW\_virtual + PWM\_BASE);

## 

## // get virtual address pointer to the pixel buffer DMA controller

## pixel\_ctrl\_ptr = (unsigned int \*) (LW\_virtual + PIXEL\_BUF\_CTRL\_BASE);

## /\* Initialize a virtual address pointer to the pixel buffer, used by drawing functions.

## \* Since our program uses the same address for both the front and back buffers, we only

## \* have to set this pointer to the pixel buffer memory once, and then use it throughout the

## \* program wherever we need to write to the pixels \*/

## pixel\_buffer\_start = (int) SDRAM\_virtual;

## clear\_screen ( ); // blank the VGA screen

## /\* Initialize the location of the back pixel buffer in the pixel buffer controller.

## \* This is the same address that we used for the front pixel buffer, which means that there

## \* is only one pixel buffer used in this program. We need to store the physical address

## \* of the pixel buffer in the pixel buffer DMA controller, as explained previously \*/

## \*(pixel\_ctrl\_ptr + 1) = SDRAM\_BASE;

## 

## /\* Set PID parameters \*/

## /\* Set this for your needs \*/

## PID.Kp = PID\_PARAM\_KP; /\* Proporcional \*/

## PID.Ki = PID\_PARAM\_KI; /\* Integral \*/

## PID.Kd = PID\_PARAM\_KD; /\* Derivative \*/

## 

## /\* Initialize PID system \*/

## PIDController\_Init(&PID, 1);

## 

## // get address for adc

## adc = ((unsigned int \*)(LW\_virtual + ADC\_BASE));

## 

## /\* Duty cycle for PWM \*/

## volatile int PWM\_Duty = 0;

## float temp\_c = 0;

## float REFERENCE\_TEMP = 25.0;

## float temp\_f = 0;

## volatile float duty = 0;

## 

## /\* create a message to be displayed on the VGA

## and LCD displays \*/

## char text\_top\_row[40] = "DE1-SoC ARM/FPGA\0";

## char text\_bottom\_row[40] = "Liberty ENGC401\0";

## char num\_string[20];

## VGA\_text (10, 1, text\_top\_row);

## VGA\_text (10, 2, text\_bottom\_row);

## 

## // turn on auto-update

## \*(adc+1) = 0x1;

## 

## \*(PWM\_addr + PWM\_DVSR\_REG) = 0x2;

## while (1){

## 

## 

## //manual control for testing

## //printf("Enter temp in c: ");

## //scanf("%f", &temp\_c);

## 

## //ADC to temp

## //calculate duty

## ADC\_Value = \*(adc) & 0xfff;

## temp\_c = (ADC\_Value - 482.3) / 7.11;

## duty = PIDController\_Update(&PID, REFERENCE\_TEMP, temp\_c);

## temp\_f =(temp\_c \* 9/5) + 32;

## printf("pid calculated duty = %f%%, temp = %f c, temp = %f f\n",duty, temp\_c, temp\_f);

## 

## 

## 

## /\* Check overflow, duty cycle in percent \*/

## //Sets any percentage above 87% to 100% duty

## if (duty >= 87){

## \_100 = true;

## if(\_75 || \_50 || \_25 || \_0){

## PWM\_Duty = 1 \* pow(2.0, (float)(res));

## change = true;

## \_75 = false;

## \_50 = false;

## \_25 = false;

## \_0 = false;

## \*(PWM\_addr + PWM\_DUTY\_REG) = (int)PWM\_Duty;

## }

## }

## //Sets any percentage above or equal to 63% and below 87% to 75% duty

## else if (duty >= 63 && duty <87 ){

## \_75 = true;

## if(\_100 || \_50 || \_25 || \_0){

## PWM\_Duty = 0.75 \* pow(2.0, (float)(res));

## change = true;

## \_100 = false;

## \_50 = false;

## \_25 = false;

## \_0 = false;

## \*(PWM\_addr + PWM\_DUTY\_REG) = (int)PWM\_Duty;

## }

## }

## //Sets any percentage above or equal to 37% and below 63% to 50% duty

## else if (duty >= 37 && duty < 63){

## \_50 = true;

## if(\_100 || \_75 || \_25 || \_0){

## PWM\_Duty = 0.50 \* pow(2.0, (float)(res));

## change = true;

## \_100 = false;

## \_75 = false;

## \_25 = false;

## \_0 = false;

## \*(PWM\_addr + PWM\_DUTY\_REG) = (int)PWM\_Duty;

## }

## }

## //Sets any percentage above or equal to 20% and below 37% to 25% duty

## else if (duty >= 20 && duty < 37){

## \_25 = true;

## if(\_100 || \_75 || \_50 || \_0){

## PWM\_Duty = 0.25 \* pow(2.0, (float)(res));

## change = true;

## \_100 = false;

## \_75 = false;

## \_50 = false;

## \_0 = false;

## \*(PWM\_addr + PWM\_DUTY\_REG) = (int)PWM\_Duty;

## }

## }

## //Sets any percentage below 20% to 0% duty

## else if (duty <20){

## \_0 = true;

## if(\_100 || \_75 || \_50 || \_25){

## PWM\_Duty = 0;

## change = true;

## \_100 = false;

## \_75 = false;

## \_50 = false;

## \_25 = false;

## \*(PWM\_addr + PWM\_DUTY\_REG) = (int)PWM\_Duty;

## }

## }//end of if/ else if statements

## 

## /\*

## \*Draws pwm\_out wave form

## \*/

## int color = 0x00ff00;

## if (change){

## //100% x = 800 y = high

## if (\_100){

## clear\_screen();

## drawline(0,150,800,150,color);

## sprintf(num\_string, "Duty Cycle: 100%%");

## PWM\_Duty = 100;

## }

## //75% x = 600 y = high after x = 601 y = low

## else if (\_75){

## clear\_screen();

## drawline(0,150,600,150,color);

## drawline(600,150,601,400,color);

## drawline(601,400,800,400,color);

## sprintf(num\_string, "Duty Cycle: 75%% ");

## PWM\_Duty = 75;

## }

## //50% x = 400 y = high after x = 401 y = low

## else if (\_50){

## clear\_screen();

## drawline(0,150,400,150,color);

## drawline(400,150,401,400,color);

## drawline(401,400,800,400,color);

## sprintf(num\_string, "Duty Cycle: 50%% ");

## PWM\_Duty = 50;

## }

## //25% x = 200 y = high after x = 201 y = low

## else if (\_25){

## clear\_screen();

## drawline(0,150,200,150,color);

## drawline(200,150,201,400,color);

## drawline(201,400,800,400,color);

## sprintf(num\_string, "Duty Cycle: 25%% ");

## PWM\_Duty = 25;

## }

## //0% x = 800 y = low

## else if (\_0){

## clear\_screen();

## drawline(0,400,800,400,color);

## sprintf(num\_string, "Duty Cycle: 0%% ");

## PWM\_Duty = 0;

## }

## VGA\_text (10, 3, num\_string);

## change = false;

## }

## 

## printf("duty = %d%%, temp = %f c, temp = %f f\n",PWM\_Duty, temp\_c, temp\_f);

## usleep(2000000);

## system("clear");

## 

## }

## // release the physical-memory mapping

## 

## // close /dev/mem

## 

## return 0;

## }

## /\* Function to blank the VGA screen \*/

## void clear\_screen( )

## {

## int i;

## int \*pixel\_ptr;

## pixel\_ptr = (int \*) pixel\_buffer\_start;

## 

## for (i = 0; i < 800\*480; i++)

## pixel\_ptr[i] = 0;

## }

## 

## void plot\_pixel(int x, int y, int line\_color)

## {

## int \*pixel\_ptr;

## pixel\_ptr = (int \*) pixel\_buffer\_start;

## 

## pixel\_ptr[y\*800 + x] = line\_color;

## }

## 

## void drawline(int x0, int y0, int x1, int y1, int color){

## 

## int dx, dy, err, x, y, sx, sy;

## dx = abs(x1-x0);

## dy = abs(y1-y0);

## sx = x0 < x1 ? 1 : -1;

## sy = y0 < y1 ? 1 : -1;

## x=x0;

## y=y0;

## err=dx-dy;

## while(x != x1 || y != y1){

## plot\_pixel(x,y,color);

## int e2 = 2\*err;

## if(e2 > -dy){

## err -= dy;

## x+= sx;

## }

## if(e2 < dx){

## err += dx;

## y += sy;

## }

## }

## plot\_pixel(x1,y1,color);// draw final point

## }

## /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## \* Subroutine to send a string of text to the VGA monitor

## \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

## void VGA\_text(int x, int y, char \* text\_ptr)

## {

## volatile char \* character\_buffer = (char \*) vga\_char\_ptr ; // VGA character buffer

## int offset;

## /\* assume that the text string fits on one line \*/

## offset = (y << 7) + x;

## while ( \*(text\_ptr) )

## {

## // write to the character buffer

## \*(character\_buffer + offset) = \*(text\_ptr);

## ++text\_ptr;

## ++offset;

## }

## }

## /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## \* Subroutine to clear text to the VGA monitor

## \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

## void VGA\_text\_clear()

## {

## volatile char \* character\_buffer = (char \*) vga\_char\_ptr ; // VGA character buffer

## int offset, x, y;

## for (x=0; x<70; x++){

## for (y=0; y<40; y++){

## /\* assume that the text string fits on one line \*/

## offset = (y << 7) + x;

## // write to the character buffer

## \*(character\_buffer + offset) = ' ';

## }

## }

## }

## pid.c

#include "PID.h"

#include <string.h>

void PIDController\_Init(PIDController \*pid, int resetStateFlag) {

/\* Derived coefficient A0 \*/

pid->A0 = pid->Kp + pid->Ki + pid->Kd;

/\* Derived coefficient A1 \*/

pid->A1 = (-pid->Kp) - ((float) 2.0 \* pid->Kd);

/\* Derived coefficient A2 \*/

pid->A2 = pid->Kd;

/\* Check whether state needs reset or not \*/

if(resetStateFlag)

{

/\* Clear the state buffer. The size will be always 3 samples \*/

memset(pid->state, 0, 3u \* sizeof(float));

}

}

float PIDController\_Update(PIDController \*pid, float setpoint, float measurement) {

/\*

\* Error signal

\*/

float error = measurement - setpoint;

float out;

/\* y[n] = y[n-1] + A0 \* x[n] + A1 \* x[n-1] + A2 \* x[n-2] \*/

out = (pid->A0 \* error) +

(pid->A1 \* pid->state[0]) + (pid->A2 \* pid->state[1]) + (pid->state[2]);

/\* Update state \*/

pid->state[1] = pid->state[0];

pid->state[0] = error;

pid->state[2] = out;

/\* return to application \*/

return (out);

}

## pwm.v

//==================================================================

// R: # reolution bit

// duty signal needs 1 extra bit

// \* e.g., 8- bit resolution need 2^8+1 values (0, 1, 2, ...,

// DIV: frequency Divider

// \* tick\_freq = pwm\_freq \* (2^resolution\_bit)

// \* DIV = system\_freq / tick\_freq

// \* use 32-bit freq divider

//==================================================================

// register map

// 0x10 to 0x1f for pwm duty cycles

// 0x00 for frequency divisor

//==================================================================

module pwm\_ip\_core

#(parameter W = 1, // width (# bits) of output port

R = 4 // # bits of PWM resolution (i.e., 2^R levels)

)

(

// to be connected to Avalon clock input interface

input wire clk, reset,

// to be connected to Avalon MM slave interface

input wire [4:0] addr,

input wire cs,

input wire read,

input wire write,

input wire [31:0] wr\_data,

output wire [31:0] rd\_data,

// external signal

output wire [W-1:0] pwm\_out

);

// signal declaration

reg [R:0] duty\_2d\_reg [W-1:0];

wire duty\_array\_en, dvsr\_en;

reg [31:0] q\_reg;

wire [31:0] q\_next;

reg [R-1:0] d\_reg;

wire [R-1:0] d\_next;

wire [R:0] d\_ext;

reg [W-1:0] pwm\_reg;

wire [W-1:0] pwm\_next;

wire tick;

reg [31:0] dvsr\_reg;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// wrapping circuit

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// decoding

assign duty\_array\_en = cs && write && addr[4];

assign dvsr\_en = cs && write && addr==5'b00000;

// register for divisor

always@(posedge clk or posedge reset)

if (reset)

dvsr\_reg <= 0;

else

if (dvsr\_en)

dvsr\_reg <= wr\_data;

// register file for duty cycles

always @(posedge clk)

if (duty\_array\_en)

duty\_2d\_reg[addr[3:0]] <= wr\_data[R:0];

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// multi-bit PWM

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

always @(posedge clk or posedge reset)

if (reset) begin

q\_reg <= 0;

d\_reg <= 0;

pwm\_reg <= 0;

end

else begin

q\_reg <= q\_next;

d\_reg <= d\_next;

pwm\_reg <= pwm\_next;

end

// "prescale" counter

assign q\_next = (q\_reg==dvsr\_reg) ? 0 : q\_reg + 1;

assign tick = q\_reg==0;

// duty cycle counter

assign d\_next = (tick) ? d\_reg + 1 : d\_reg;

assign d\_ext = {1'b0, d\_reg};

// comparison circuit

genvar i;

generate

for (i=0; i<W; i=i+1) begin: registers

assign pwm\_next[i] = d\_ext < duty\_2d\_reg[i];

end

endgenerate

assign pwm\_out = pwm\_reg;

// read data not used

assign rd\_data = pwm\_reg;

endmodule