E155 Final Project: Magic See-Saw

Henry Limm and Alex Moody

Abstract

An electromechanical system is implemented to balance a ball on an actuated platform with one degree of freedom. The rectangular platform rotates about its short axis while the ball is free to roll along the platform's long axis. A soft potentiometer is used to provide ball position feedback to a ATSAM4S4B microcontroller that handles the main loop. A discrete time PID controller was implemented in hardware on the Cyclone IV FPGA. The microcontroller communicates with the Cyclone IV FPGA over an SPI connection to send error values and receive control values. The control values are calibrated on the microcontroller and converted into a PWM signal to drive the servo motor and actuate the platform. The system robustly responds to disturbances on the ball with small overshoot and small oscillations before arriving at a stationary steady state.

Introduction

The final project for E155 tasked students with using the ATSAM4S4B microcontroller and the Cyclone IV E FPGA to create a useful or interesting system. The HMC engineering curriculum introduces its students to traditional and modern control theory. All engineers learn about the transient response of systems and how different control laws can affect them. Although students work through many simulations implementing traditional and modern control in both discrete and continuous time, they often have less experience in working with physical systems. Even the physical systems students work with, though, are highly damped and do not require too much additional control beyond simple proportional control. Thus, the motivation behind this project is to practice implementing feedback control systems on a physical system that is more complicated to model. The final design was chosen such that the system's step response could be easily observed.

Overall System

The overall system consists of a ball track on a platform with a linear potentiometer mounted inside the track. A servo motor actuates the platform via a cam connecting the servo arm to an offset axis on the platform.

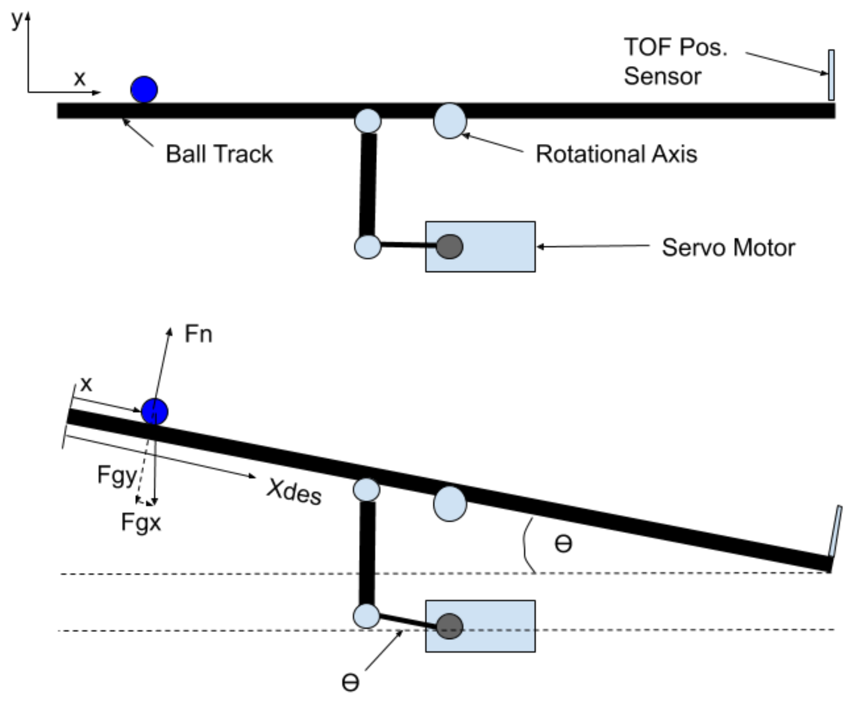
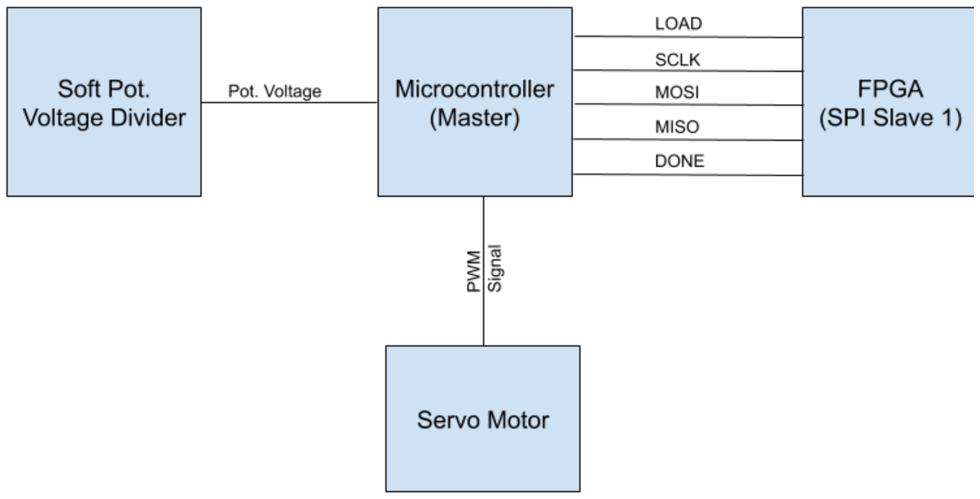


Figure 1: Physical System

The microcontroller manages the main tasks of the system by interfacing with the servo motor, potentiometer and FPGA as shown in the following figure:



Feedback System

For the ball to balance at a desired location, the motor must be controlled by a feedback loop. The following section describes each component of the feedback system in detail.

Position Sensor

The ball's position is tracked by a thin, flat rheostat that is adhered to the bottom of the ball's track. The <part no> is comprised of three layers: a base layer with a linear resistivity, a conductive layer, and an insulating layer with a cutout along the length of the sensor that separates the other two. When the ball (or anything else) presses on the sensor with enough force <F>, it completes a connection between the conductive and resistive layers. The resistance of the completed circuit will vary with where the sensor is pressed.

The variable resistor is integrated with a $10\ k\Omega$ resistor to create a voltage divider. This setup yields the following relationship between the voltage into the circuit and out of the intermediate node.

$$Vout = Vin * \frac{Rpot}{Rpot+10k\Omega} \qquad (1)$$

Additionally, since the resistance across the potentiometer is proportional to the position of contact, equation (1) can be rearranged to express the position of the ball in terms of Vin, Vout, trackLength, max potentiometer resistance (Rmax) and the 10kΩ resistor. All of these parameters are fixed or available to the microcontroller at each time step.

$$xball = \frac{trackLength * V\,out * 10k\Omega}{(3.3 - V\,out) * Rmax} \quad (2)$$

Where Vin = 3.3V, Vout is the output from the ADC, Rmax is 18.1kΩ and the track length is 500mm. This function solves for position based on the measurement of the voltage divider. The expression outputs the position of the ball in meters. Therefore, 0.25m corresponds to the center of the track.

Correction Calculation

Calculating the angle at which the track needs to be tilted in order to return the ball to its desired position is done primarily on the FPGA. However, some of the calculation is completed on the microcontroller itself.

MCU Passthrough

Before sending any information to the primary calculator on the FPGA, the microcontroller must first process the ball's position. Once the microcontroller measures the voltage across the rheostat, it converts that voltage to the ball's position using equation (2). Since the PID controller on the FPGA requires an error input, the position of the ball is compared to the hard coded desired position. The difference is then sent to the FPGA over the SPI connection as a 16 bit signed integer. Since the error value is initially a floating-point decimal ranging from 0-0.5, this requires the error to be scaled in magnitude before truncating to an integer. The error is multiplied by a factor of 1000 to provide the FPGA with three significant figures. Once the FPGA returns the proper control angle based on the error and previous errors, the microcontroller converts the angle to the proper range of the physical system. This requires limiting the angle such that the motor is not set to an angle that the mechanical system is not designed for. The parameters of the physical system will be discussed further in the mechanical design.

FPGA PID Controller

A proportional and derivative controller was implemented on the FPGA. A PID controller was chosen over state space alternatives because no plant model is required nor is an observer required to obtain the necessary states for control. Since the only accessible states are the angle and the position of the ball, creating an observer and plant model for the mechanical system is more time consuming than the traditional PID alternative. As discussed later, the mechanical system does not match its model perfectly which may have caused further difficulties with the state space approach. Finally, one of the most important factors in choosing a control design alternative was the ease of implementation on an FPGA. Since discrete PID control can be expressed as a difference equation, it was an appealing approach to implement on the FPGA. The standard expression for a PID controller output is represented by the sum of the error, integral of past errors, and the derivative of the error.

$$u(t) = Kp * e(t) + Ki\int_0^t e()d + Kd * \frac{d}{dt}e(t) \quad (3)$$

Equation (3) can be converted to discrete time by finding its z-transform.

$$U(z) = [\frac{(Kp+Ki+Kd)+(-Kp-2Kd)z^{-1}+Kdz^{-2}}{1-z^{-1}}]E(z) \quad (4)$$

This expression can then be converted to the following difference equation.

$$u[k] = u[k-1] + (Kp + Ki + Kd) * e[k] + (-Kp - 2Kd) * e[k-1] + Kd * e[k-2]$$

Where Kp, Ki and Kd are the gains for their proportional integral and derivative control respectively.

Given the structure of the difference equation, the FPGA could be easily configured to compute PID output values in hardware. The gain values for each type of control are hardcoded into the FPGA hardware and are multiplied by the correct error. The PID control is initialized by the microcontroller when reset is pulled high prior to any SPI communication in the main control loop. Once the errors and output is initialized at zero, the PID controller updates every time it receives an error value over SPI. In one clock cycle, the controller calculates the output with the difference equation represented by combinational logic while also cycling the errors through. The current error becomes the previous error and the previous error becomes the previous previous error and so forth. Additionally, an external signal is set high during the same clock cycle to signify that the computation has occurred. The error values are stored in the registers until another error value is sent to the FPGA by the microcontroller. The hardware implemented can be found in Appendix C.

One of the largest challenges with the PID controller on the FPGA is handling the error values with the 16 bit signed arithmetic. Recall the transformation from floats to integers described in the MCU passthrough section. In order for higher resolution error values to be used by the FPGA larger error values need to be sent to the FPGA logic. This process sometimes caused internal error values to exceed the maximum values of the 16 bit integer type after the gains were applied. Thus, improper control outputs were calculated by the logic. Ultimately, the problem was solved by reducing the size of the input error values and setting integral control gain to zero. Since integral control adds to the control output for every time step the error is nonzero, the value quickly grows beyond the max value of 16 bit integer. These issues can be fixed if floating point values are implemented in the logic. Floating point values allow all of the computations to function off of the original error that is calculated by the microcontroller.

Ultimately, the implementation of a PID controller on the FPGA is an elegant design that does not require excessive hardware. It is a fast way to calculate the proper control value for a system.

MCU Peripherals/Communication Protocols

Analog/Digital Converter

The analog to digital converter is used to convert the voltage corresponding to the position of the ball to a digital value. This peripheral is handled by a provided library that initializes the correct registers. The 10 bit resolution was used on the ADC channel 0 with a gain of 1. The voltage from the voltage divider is in the range of 0-2.3V so it does not need to be amplified. Additionally, the offset is turned off because the voltage does not go below zero and can be used directly from the voltage divider.

Serial Peripheral Interface

The microcontroller and the FPGA communicate with each other via an adaptation of the Serial Peripheral Interface (SPI) specification, taking advantage of a peripheral on the microcontroller built for that purpose. In addition to following the SPI communication format, the system also implements several additional status lines.

General SPI Peripheral

Data transfer between the microcontroller and the FPGA is done using Channel 0 of the SPI module on the microcontroller and a module on the FPGA adapted from the module in E155 Lab 7.

The microcontroller uses the library `fpga_controller.h` at the highest level to manage communications. This library is little more than a wrapper for the SPI library, but also manages the additional communication lines that will be detailed in the following section.

The SPI library, `SAM4S4B_spi.h`, is not actually the library provided for later E155 labs. This library adds functionality to disable the write protection on the peripheral using the Write Protection Mode Register (`SPI_WPMR`) register and the associated password (0x535049). While the write protection should be disabled by default, forcing the disable on each initialization has proven to ensure that the peripheral works.

In addition to manually disabling the peripheral's write protection on startup, this SPI module can also be configured to send up to sixteen bits of data at a time rather than eight. When the peripheral is initialized in `fpga_controller.h`, the Chip Select Register for channel 0

(SPI_CSR0) is given the value 8 for its BITS field, which configures the peripheral to send and receive 16 bits in a single transmission.

The fpga_controller.h library must also configure the SPI clock parameters, which are also written to the channel 0 Chip Select Register by SAM4S4B_spi.h. These parameters define the SPI clock (SPCK) polarity and phase as well as the frequency of the SPI clock itself.

To work in conjunction with the FPGA's SPI module, the SPCK polarity is set to be inactive at logic level zero (CPOL = 0) and the phase is such that data is changed on the clock's leading edge and captured on the falling edge (NCPHA = 0). To set the frequency of SPCK, the libraries define the SPI bit rate. The bit rate is defined as $SPCk\ Bit\ Rate\ =\ f_{peripheral\ clock}/SCBR$, where SCBR is a field within SPI_CSR0. The microcontroller's clock is approximately $40\ MHz$, so SCBR is set to 10 to achieve a bit rate of $4\ MHz$. All other fields are left at default

Extra Communication Lines

The SPI communication between the microcontroller and the FPGA does not actually use the standard Chip Enable line at all. Instead, there are two status lines between the microcontroller and the FPGA, one driven by each. These lines are named GIVE_ERROR and GET_ANGLE, and load and done on the microcontroller and FPGA, respectively. When the microcontroller is ready to send the ball's current error to the FPGA, it raises GIVE_ERROR (load on the FPGA), which activates the SPI module within the FPGA to read in the microcontroller's transmission. Once the microcontroller finishes transmitting, it lowers GIVE_ERROR and waits. Once the FPGA finishes its round of correction calculations, it raises GET_ANGLE on the microcontroller (done internally). This prompts the microcontroller to read in the lever's new angle from the FPGA, writing garbage bits at the same time. Once the FPGA finishes its transmission, it lowers GET_ANGLE (done) and waits for the next transmission from the microcontroller.

Pulse Width Modulation

The servo motor is actuated by a PWM signal where the width of the pulse represents the angle to set the axel. The DS3218 servo has an operating frequency of 50-330Hz. Therefore the period of the PWM signal had to be large enough to encompass the full range of pulse widths to control the motor but still be within the frequency range. Since 0.5ms represents 0˚on the servo and 2.5ms represents 270˚on the servo, a total period of 4.5ms was chosen. This value is not the only functional period but it did allow the servo to operate on the upper end of its frequency and receive motor angles faster. The PWM peripheral on the microcontroller is used to produce the waveform.

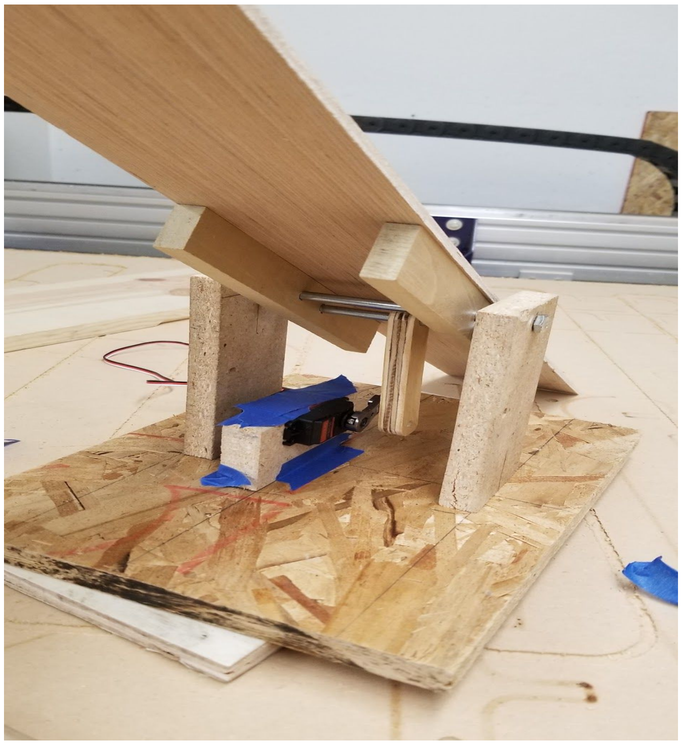
The PWM peripheral produces a waveform by incrementing a counter every clock cycle. Once the counter reaches a defined value, it resets and starts counting from zero again. Each time the counter resets, a new pulse is sent high. Thus, the period of the waveform depends on the clock speed and the counter reset value. The clock speed used is the main clock divided by eight. Since the main clock speed is known, the counter value to achieve the desired period could be calculated.

The PWM peripheral similarly defines the duty cycle of the waveform. By setting the max duty cycle in the peripheral, the PWM peripheral sets the pulse low once the counter reaches the max duty cycle value. Thus, the duty cycle waveform can range from a constant low logic level all the way to a constant high logic level depending on the max counter values set. The desired max duty cycle value can be calculated similarly to the max period value. Given the desired width of the pulse, and the speed of the clock, the max value of the counter can be calculated. Using the relationship between the angle and the width of the PWM pulse, an expression for the max duty cycle value can be written in terms of the desired angle of the motor.

Once the max duty cycle and max period values are calculated, they can be written to the PWM update registers that will adjust the PWM signal accordingly at the end of the next PWM period.

Mechanical Design

The mechanical design of the system consists of a wood base with a wood platform rotating on a suspended axis.



The servo actuates a cam that adjusts the tilt of the platform. The cam connection point on the platform has the same radius as the connection point on the servo. Therefore, the motor angle approximately matches the platform angle.

Calibration

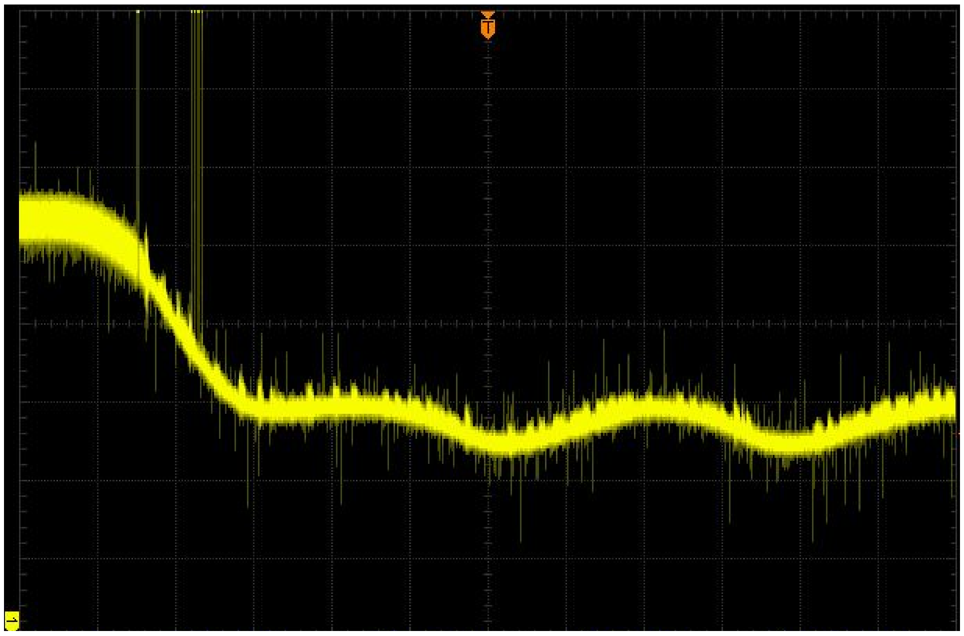
Although the servo PWM signals were configured on the microcontroller such that it would have a neutral point of 0˚, the mechanical system was not perfectly constructed to match this value. Therefore, the platform had to be calibrated with the motor values such that the neutral position of the platform is level. Further calibration on the position of the ball was necessary because the potentiometer was not perfectly centered about the neutral axis.

Tuning

The advantage of PID control is that there are only three tuning parameters, the proportional, integral and derivative control. The proportional control was not as large as the derivative control value because the system is extremely underdamped so it is important to limit the oscillations of the system. It is important to note that the gain on the derivative control value depends on the sampling time. Thus, a larger sampling time was implemented to achieve better derivative control. The integral control gain is set to zero on the FPGA for reasons described in the PID section. Integral control, however is implemented on the microcontroller in software and has a gain of 1 since the PD controller only produces a steady state error of about 3cm.

Current Implementation

The current implementation responds well to large and small disturbances with a small amount of overshoot and small oscillations before reaching its steady state.



References

Toochinda, Varodom. Digital PID Controllers. PDF 2011.

Chasnov, Ben; Norfleet, Caleb. Lab 7. PDF 2015, 2019.

Appendix A: Code

```c
/**
 * motorControl.c
 *
 * Top level code for microcontroller in the ball balancing system.
 * Reads a ball's position and feeds it to an FPGA with a PD control
accelerator
 * over SPI.
 *
 * H Limm, A Moody
 * Fall 2019
 * HMC ENGR 155
 */

#include "SAM4S4B.h"
#include "fpga_controller.h"
#include <stdio.h>

#define XDES .25  //desired ball position



int main() {
        float xdOffset = 0.011;
        float xd = XDES + xdOffset;
        double angleOffset = 151.5;
        double maxAngle = 10;

        //initialize controller
        samInit();
        //initialize pins
        pioInit();
        //initialize delay
        tcInit();
        tcDelayInit();
        //initialize SPI
        uint32_t csActiveHigh = 1;
        spiInit(SPCK_POL, SPCK_PHA, SPI_CSR_BITS_16, SCBR_DIV,
csActiveHigh);
        //initialize PWM/motor position
        pwmInit(0, 300, 22479, 14986);
        servoSet(angleOffset);
        //initialize ADC for feedback
        adcInit(ADC_MR_LOWRES_BITS_10);
        adcChannelInit(0,ADC_CGR_GAIN_X1,ADC_COR_OFFSET_OFF);
        //initialize PID controller
        initController();
        //initialize integral control
	double netError=0;
	float pos = 0;

	// set a xdes changer
	int maxtime = 200;
	int loopcount = 0;
	float offset = 0.10;

	// start the loop
	while(1) {
		//Find Error
		int ki = 1;
		float voltage = adcRead(0);
		if (voltage < 2.2) {
			pos = voltage*.276/(3.3-voltage);
		}
		float error = pos - (xd+offset);

		//beginning of control (input error)
		int16_t nangle = getNewAngle(error);
		double angle = (double) nangle/12;

		//Integral Control to Remove Steady State
		angle = angle + ki*netError;
		if (netError > 2) {
			netError = 2;
		}
		else if (netError < -2) {
			netError = -2;
		}
		netError = netError + error;
		//end of control (output angle)

		//Threshold angle
		if (angle > maxAngle) {
			angle = maxAngle;
		}
		else if (angle < -maxAngle-2) {
			angle = -maxAngle-2;
		}

		//Convert angle to proper servo values
		angle = angle+angleOffset;
		servoSet(angle);
		tcDelayMillis(40);

		// position changer
		if (loopcount >= maxtime) {
```

```
                        offset = -1 * offset;
                        loopcount = 0;
                    }
                    loopcount++;


        }

}

```

```c
/**
 * fpga_controller.h
 *
 * Library for interfacing with FPGA configured as a feedback controller.
 * Essentially an SPI wrapper that exclusively talks to the FPGA.
 *
 * H Limm, A Moody
 * Fall 2019
 * HMC ENGR 155
 */

#ifndef FPGA_CONTROLLER_H
#define FPGA_CONTROLLER_H
#include <stdint.h>
#include "SAM4S4B.h"

// SPI PARAMETERS
#define SPCK_POL 0              // inactive state of clock 0
#define SPCK_PHA 0              // data changed on leading edge, captured on following
#define SCBR_DIV 10             // on 40 MHz clock -> f_SPCK = 40/10 MHz = 4 MHz

// pins for FPGA comm
#define GIVE_ERROR PIO_PA18     // SOME PIN
#define GET_ANGLE PIO_PA19      // ANOTHER PIN
#define FPGA_RESET PIO_PA20     // arrivederci

#define QUANT_FACTOR 1000       // (x swing is 0 - 0.5; multiply to get to scalars)

void initController() {
        // initialize spi
        spiInit(SPCK_POL, SPCK_PHA, SPI_CSR_BITS_16, SCBR_DIV, 1);

        // initialize additional pins for the communication
        pioPinMode(GIVE_ERROR, PIO_OUTPUT);
        pioPinMode(GET_ANGLE, PIO_INPUT);
        pioPinMode(FPGA_RESET, PIO_OUTPUT);

        // pulse reset on the FPGA so that the control registers are zeroed
        pioDigitalWrite(FPGA_RESET, 1);
        tcDelayMillis(20);
        pioDigitalWrite(FPGA_RESET, 0);
}

// gives error to the FPGA, waits, then reads the new angle from FPGA
int16_t getNewAngle(float error) {
        pioDigitalWrite(GIVE_ERROR, 1);         // tell FPGA that MCU is sending error
        int16_t error_q = (int16_t) (error*QUANT_FACTOR);       // quantize (scale+round)
        spiTxRx(error_q, 1);                            // send error MCU -> FPGA
        pioDigitalWrite(GIVE_ERROR, 0);         // tell FPGA that MCU is done sending

        // wait for the FPGA to finish calculating the new angle
        while(!pioDigitalRead(GET_ANGLE));

        return spiTxRx(0x4148, 1);                      // get error MCU <- FPGA
}

#endif

```

```
/**
 * SAM4S4B_spi.h
 *
 * Contains base address locations, register structs, definitions, and
functions
 * for the SPI (Serial Peripheral Interface) peripheral of the SAM4S4B
 * microcontroller.
 *
 * Henry Limm, 2020
 * hlimm@hmc.edu
 * Fall 2019
 */

#ifndef SAM4S4B_SPI_H
#define SAM4S4B_SPI_H
#include <stdint.h>
#include "SAM4S4B_pmc.h"
#include "SAM4S4B_pio.h"

// SPI BASE ADDRESS

#define SPI_BASE (0x40008000U)



// SPI REGISTERS

// SPI Control Register
typedef struct {
        volatile uint32_t SPIEN         : 1;
        volatile uint32_t SPIDIS        : 1;
        volatile uint32_t                       : 5;
        volatile uint32_t SWRST         : 1;
        volatile uint32_t                       : 16;
        volatile uint32_t LASTXFER      : 1;
        volatile uint32_t                       : 7;
} SPI_CR_bits;

// SPI Mode Register
typedef struct {
        volatile uint32_t MSTR          : 1;
        volatile uint32_t PS            : 1;
        volatile uint32_t PCSDEC        : 1;
        volatile uint32_t                       : 1;
        volatile uint32_t MODFDIS       : 1;
        volatile uint32_t WDRBT         : 1;
        volatile uint32_t                       : 1;
```

```c
        volatile uint32_t LLB           : 1;
        volatile uint32_t                       : 8;
        volatile uint32_t PCS           : 4;
        volatile uint32_t                       : 4;
        volatile uint32_t DLYBCS        : 8;
} SPI_MR_bits;

// SPI Receive Data Register
typedef struct {
        volatile uint32_t RD    : 16;
        volatile uint32_t PCS   : 4;
        volatile uint32_t               : 12;
} SPI_RDR_bits;

// SPI Transmit Data Register
typedef struct {
        volatile uint32_t TD            : 16;
        volatile uint32_t PCS           : 4;
        volatile uint32_t                       : 4;
        volatile uint32_t LASTXFER      : 1;
        volatile uint32_t                       : 7;
} SPI_TDR_bits;

// SPI Status Register
typedef struct {
        volatile uint32_t RDRF          : 1;
        volatile uint32_t TDRE          : 1;
        volatile uint32_t MODF          : 1;
        volatile uint32_t OVRES         : 1;
        volatile uint32_t ENDRX         : 1;
        volatile uint32_t ENDTX         : 1;
        volatile uint32_t RXBUFF        : 1;
        volatile uint32_t TXBUFE        : 1;
        volatile uint32_t NSSR          : 1;
        volatile uint32_t TXEMPTY       : 1;
        volatile uint32_t UNDES         : 1;
        volatile uint32_t                       : 5;
        volatile uint32_t SPIENS        : 1;
        volatile uint32_t                       : 15;
} SPI_SR_bits;

// SPI Interrupt Enable Register
typedef struct {
        volatile uint32_t RDRF          : 1;
        volatile uint32_t TDRE          : 1;
        volatile uint32_t MODF          : 1;
        volatile uint32_t OVRES         : 1;
        volatile uint32_t ENDRX         : 1;
        volatile uint32_t ENDTX         : 1;
        volatile uint32_t RXBUFF        : 1;
        volatile uint32_t TXBUFE        : 1;
        volatile uint32_t NSSR          : 1;
        volatile uint32_t TXEMPTY       : 1;
        volatile uint32_t UNDES         : 1;
        volatile uint32_t                       : 21;
} SPI_IER_bits;

// SPI Interrupt Disable Register
typedef struct {
        volatile uint32_t RDRF          : 1;
        volatile uint32_t TDRE          : 1;
        volatile uint32_t MODF          : 1;
        volatile uint32_t OVRES         : 1;
        volatile uint32_t ENDRX         : 1;
        volatile uint32_t ENDTX         : 1;
        volatile uint32_t RXBUFF        : 1;
        volatile uint32_t TXBUFE        : 1;
        volatile uint32_t NSSR          : 1;
        volatile uint32_t TXEMPTY       : 1;
        volatile uint32_t UNDES         : 1;
        volatile uint32_t                       : 21;
} SPI_IDR_bits;

// SPI Interrupt Mask Register
typedef struct {
        volatile uint32_t RDRF          : 1;
        volatile uint32_t TDRE          : 1;
        volatile uint32_t MODF          : 1;
        volatile uint32_t OVRES         : 1;
        volatile uint32_t ENDRX         : 1;
        volatile uint32_t ENDTX         : 1;
        volatile uint32_t RXBUFF        : 1;
        volatile uint32_t TXBUFE        : 1;
        volatile uint32_t NSSR          : 1;
        volatile uint32_t TXEMPTY       : 1;
        volatile uint32_t UNDES         : 1;
        volatile uint32_t                       : 21;
} SPI_IMR_bits;

// SPI Chip Select Register
typedef struct {
        volatile uint32_t CPOL          : 1;
        volatile uint32_t NCPHA         : 1;
        volatile uint32_t CSNAAT        : 1;
        volatile uint32_t CSAAT         : 1;
        volatile uint32_t BITS          : 4;
```

```c
        volatile uint32_t SCBR          : 8;
        volatile uint32_t DLYBS         : 8;
        volatile uint32_t DLYBCT        : 8;
} SPI_CSRx_bits;                                    // There are three of these

// SPI Write Protection Mode Register
typedef struct {
        volatile uint32_t WPEN  : 1;
        volatile uint32_t               : 7;
        volatile uint32_t WPKEY :24;
} SPI_WPMR_bits;

// SPI Write Protection Status Register
typedef struct {
        volatile uint32_t WPVS          : 1;
        volatile uint32_t                       : 7;
        volatile uint32_t WPVSRC        : 8;
        volatile uint32_t                       : 16;
} SPI_WPSR_bits;



// OVERALL PERIPHERAL STRUCT
typedef struct {
        volatile SPI_CR_bits    SPI_CR;                 // (offset: 0x00)
        volatile SPI_MR_bits    SPI_MR;                 // (offset: 0x04)
        volatile SPI_RDR_bits   SPI_RDR;                // (offset: 0x08)
        volatile SPI_TDR_bits   SPI_TDR;                // (offset: 0x0C)
        volatile SPI_SR_bits    SPI_SR;                 // (offset: 0x10)
        volatile SPI_IER_bits   SPI_IER;                // (offset: 0x14)
        volatile SPI_IDR_bits   SPI_IDR;                // (offset: 0x18)
        volatile SPI_IMR_bits   SPI_IMR;                // (offset: 0x1C)
        volatile uint32_t               Reserved1[4];   // (offset: 
0x20-0x2C) 4× 4 bytes res
        volatile SPI_CSRx_bits  SPI_CSR0;               // (offset: 0x30)
        volatile SPI_CSRx_bits  SPI_CSR1;               // (offset: 0x34)
        volatile SPI_CSRx_bits  SPI_CSR2;               // (offset: 0x38)
        volatile SPI_CSRx_bits  SPI_CSR3;               // (offset: 0x3C)
        volatile uint32_t               Reserved2[41];  // (offset: 
0x40-0xE0) 41× 4 bytes
        volatile SPI_WPMR_bits  SPI_WPMR;               // (offset: 0xE4)
        volatile SPI_WPSR_bits  SPI_WPSR;               // (offset: 0xE8)
        volatile uint32_t               Reserved3[5];   // (offset: 
0xEC-0xFC) 5× 4 bytes res
        volatile uint32_t               Reserved4[10];  // (offset: 
0x100-0x124) PDC Regs
} spi;

```

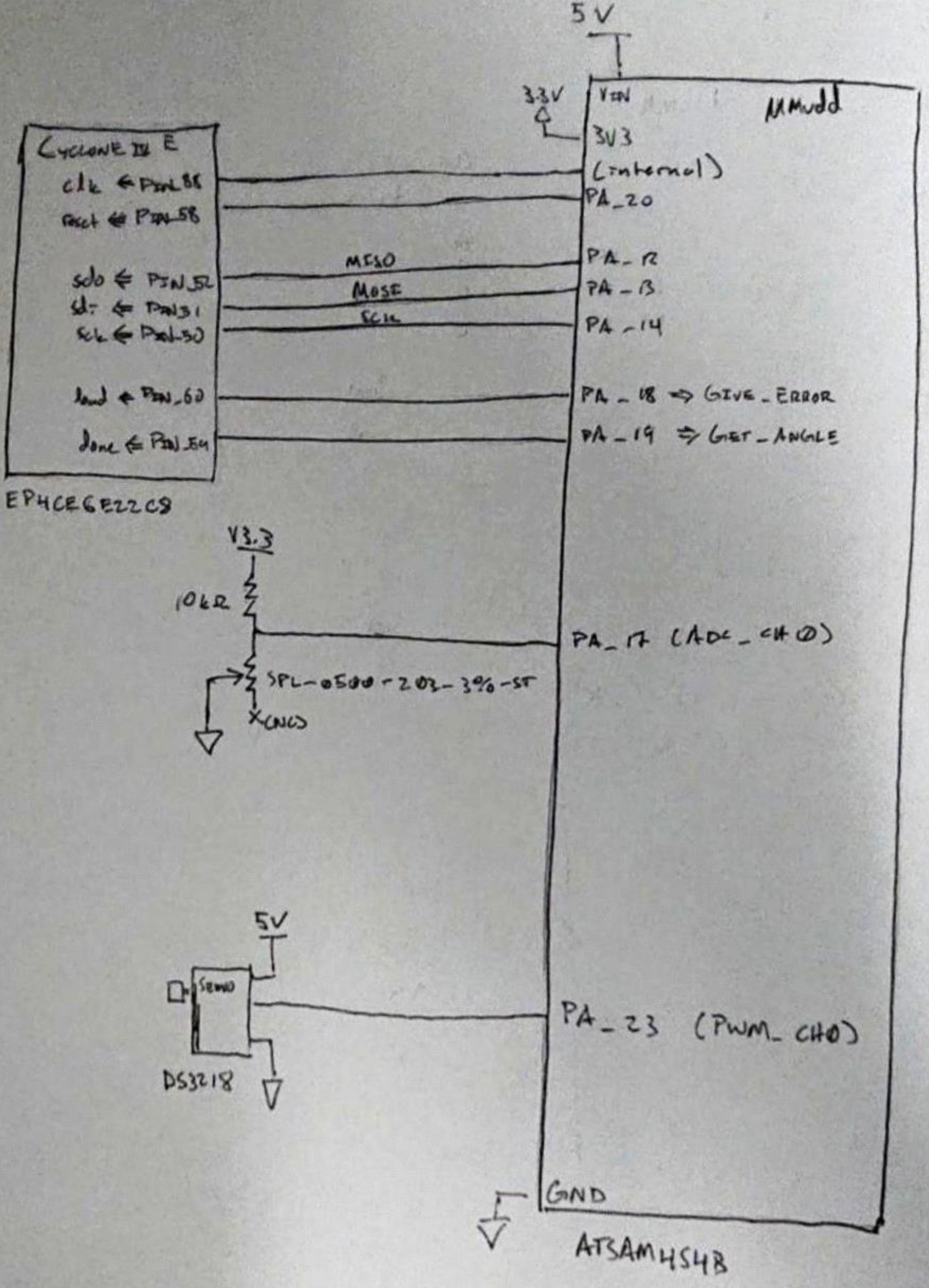
```c
// POINTER TO A spi-SIZED PIECE OF MEMORY AT THE SPI PERIPHERAL BASE ADDRESS
#define SPI ((spi*) SPI_BASE)



// VALUES FOR SEVERAL OF THE SPI REGISTERS
#define SPI_CSR_BITS_8          0
#define SPI_CSR_BITS_9          1
#define SPI_CSR_BITS_10         2
#define SPI_CSR_BITS_11         3
#define SPI_CSR_BITS_12         4
#define SPI_CSR_BITS_13         5
#define SPI_CSR_BITS_14         6
#define SPI_CSR_BITS_15         7
#define SPI_CSR_BITS_16         8

// SPECIFIC PIO PINS AND PERIPHERAL FUNCTIONS THAT SPI USES (set in spiInit())
#define SPI_NPCS0               PIO_PA11        // Peripheral A
#define SPI_NPCSQ               PIO_PA15        // gpio
#define SPI_MISO_PIN    PIO_PA12        // Peripheral A
#define SPI_MOSI_PIN    PIO_PA13        // Peripheral A
#define SPI_SPCK                PIO_PA14        // Peripheral A
#define SPI_FUNC                PIO_PERIPH_A

// password to enable write protection
// WPEN defaults to 0 in SPI_WPMR, so using this is not entirely necessary
#define SPI_WPMR_WPKEY_PASSWD (0x535049 << 8);



// SPI USER FUNCTIONS
void spiInit(uint32_t clkpol, uint32_t ncpha, uint32_t bits,
                                uint32_t scbr, uint32_t csActiveHigh) {
        pmcEnablePeriph(PMC_ID_SPI);
        pioInit();

        // set pins to SPI functionality
        pioPinMode(SPI_MISO_PIN, SPI_FUNC);
        pioPinMode(SPI_MOSI_PIN, SPI_FUNC);
        pioPinMode(SPI_SPCK, SPI_FUNC);
        if (!csActiveHigh)
                pioPinMode(SPI_NPCS0, SPI_FUNC);        // use SPI_FUNC if not active high
        else if (csActiveHigh) {
                pioPinMode(SPI_NPCSQ, PIO_OUTPUT);      // manually control
                pioDigitalWrite(SPI_NPCSQ, 0);
        }

        // configure control and mode registers

        SPI->SPI_CR.SPIEN = 1;  // enable SPI
        SPI->SPI_WPMR.WPKEY = SPI_WPMR_WPKEY_PASSWD; // set WP password
        SPI->SPI_WPMR.WPEN = 0; // disable write protection
        SPI->SPI_MR.MSTR = 1;   // set to Master mode

        // configure Chip Select Register
        SPI->SPI_CSR0.CPOL = clkpol;
        SPI->SPI_CSR0.NCPHA = ncpha;
        SPI->SPI_CSR0.BITS = (uint8_t) bits;    // bits per transfer
        SPI->SPI_CSR0.SCBR = scbr;
}

// void spiTx(uint16_t data) {
//      // use # of bits as initialized
//      while(!(SPI->SPI_SR.TDRE));     // wait for old data in TDR to go
to shift reg
//      // once the TDR is clear for new data:
//      SPI->SPI_TDR.TD = data;
// }

// int spiRxReady() {
//      // return 1 when the Receive Data Register is full
//      return SPI->SPI_SR.RDRF;
// }

// int spiRx() {
//      if(spiRxReady())
//              return (uint16_t) SPI->SPI_RDR.RD;
//      else
//              return (uint16_t) 0;
// }

int16_t spiTxRx(int16_t txData, uint32_t csActiveHigh) {
        // wait for old data in TDR to go to shift register and sent out
        // while(!(SPI->SPI_SR.TDRE));
        if (csActiveHigh) pioDigitalWrite(SPI_NPCSQ, 1); // for active high
CS
        // once the TDR has been cleared for new data:
        SPI->SPI_TDR.TD = txData;
        // wait for all new RX data is transferred from shift register to
RDR
        while(!(SPI->SPI_SR.RDRF));
        if(csActiveHigh) pioDigitalWrite(SPI_NPCSQ, 0); // for active high
CS
```

```
        // return the RX data in the RDR
        return (int16_t) SPI->SPI_RDR.RD;
}

#endif

```

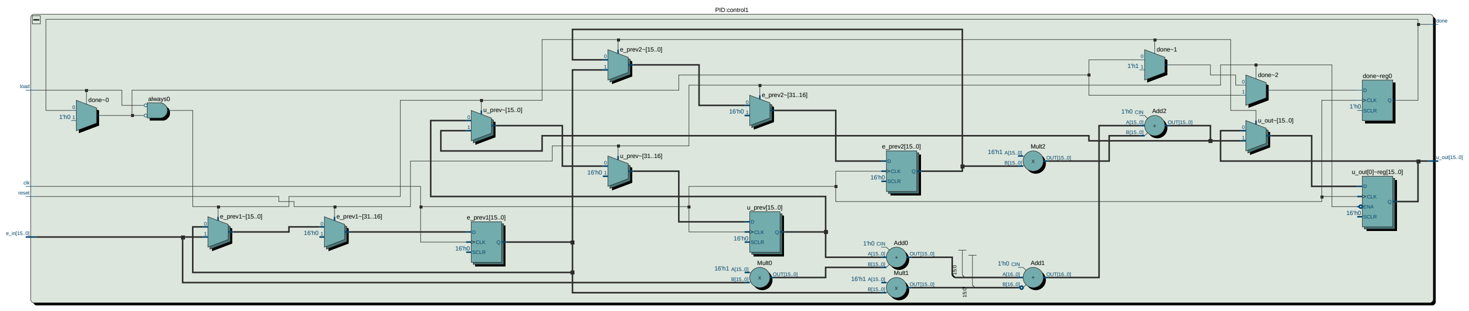
```
//////////////////////////////////////////////
// controller.sv
// HMC E155 20 November 2019
// amoody@hmc.edu
// spi module modeled after E155 lab7 starter code
// code was modeled after an example Verilog pid controller:
// https://openlab.citytech.cuny.edu/?get_group_doc=4784/1393258757-
DigitalPIDControllers_2011.pdf
//////////////////////////////////////////////

//////////////////////////////////////////////
// controller
//   Top level module with SPI interface and SPI core
//////////////////////////////////////////////

module controller(input  logic clk, reset,
           input  logic sck,
           input  logic sdi,
           output logic sdo,
           input  logic load,
           output logic done);

    logic [15:0] error, angle;

    controller_spi spi(sck, sdi, sdo, done, error, angle);
    controller_core core(clk, reset, load, error, done, angle);
endmodule

//////////////////////////////////////////////
// controller_spi
//   SPI interface.  Shifts in error values
//   Captures angle when done, then shifts it out
//   Tricky cases to properly change sdo on negedge clk
//////////////////////////////////////////////

module controller_spi(input  logic sck,
               input  logic sdi,
               output logic sdo,
               input  logic done,
               output logic [15:0] error,
               input  logic [15:0] angle);

    logic         sdodelayed, wasdone;
    logic [15:0] angleCaptured;

    // assert load
    // apply 8 sclks to shift in error, starting with error[0]
    // then deassert load, wait until done
    // then apply 8 sclks to shift out angle, starting with cyphertext[0]
    always_ff @(posedge sck)
        if (!wasdone)  {angleCaptured, error} = {angle, error[14:0], sdi};
        else           {angleCaptured, error} = {angleCaptured[14:0], 
error, sdi};

    // sdo should change on the negative edge of sck
    always_ff @(negedge sck) begin
        wasdone = done;
        sdodelayed = angleCaptured[14];
    end

    // when done is first asserted, shift out msb before clock edge
    assign sdo = (done & !wasdone) ? angle[15] : sdodelayed;
endmodule

/////////////////////////////////////////////
// controller_core
//   top level ceontroller
//   when load is asserted, takes the current error and
//   calculates angle
// difference eqution:
//  u_out =  u_prev + k1*e_in - k2*e_prev[1] + k3*e_prev[2];
/////////////////////////////////////////////

module controller_core(input  logic  clk, reset,
                input  logic         load,
                input  logic [15:0] error,
                output logic         done,
                output logic [15:0] angle);
    PID control1(clk, load, reset, error, angle, done);


endmodule

/////////////////////////////////////////////
// PID Controller
// k1 = kp + ki + kd
// k2 = -kp - 2*kd
// k3 = kd
/////////////////////////////////////////////

module PID #(parameter WIDTH = 16)
            (input  logic clk, load, reset,
            input  logic signed [WIDTH-1:0] e_in,
            output logic signed [WIDTH-1:0] u_out,
            output logic done);
```

```

	logic signed [WIDTH-1:0] k_1 = 16'd1;
	logic signed [WIDTH-1:0] k_2 = 16'd1;
	logic signed [WIDTH-1:0] k_3 = 16'd1;

    logic signed [WIDTH-1:0] u_prev;
    logic signed [WIDTH-1:0] e_prev1;
    logic signed [WIDTH-1:0] e_prev2;



    always @(posedge clk) begin
		if (load)
			done = 0;
        if (reset==1) begin
            u_prev = 0;
            e_prev1 = 0;
            e_prev2 = 0;
        end
        else if ((~load) && (~done)) begin
            assign u_out = u_prev + (k_1 * e_in) - (k_2 * e_prev1) + (k_3 *
e_prev2);
            e_prev2 <= e_prev1;
            e_prev1 <= e_in;
            u_prev <= u_out;
			done = 1;
        end
	end

endmodule

```

Appendix B: Schematic


5V
3.3V
VIN
3V3
(internal)
µMudd
Cyclone IV E
clk ⇐ PIN_88
reset ⇐ PIN_58
sdo ⇐ PIN_52
sdi ⇐ PIN_31
sck ⇐ PIN_30
load ⇐ PIN_60
done ⇐ PIN_54
EP4CE6E22C8
PA_20
MISO
MOSI
SCK
PA_12
PA_13
PA_14
PA_18 ⇒ GIVE_ERROR
PA_19 ⇒ GET_ANGLE
V3.3
10kΩ
PA_17 (ADC_CH0)
SPL-0500-203-3%-ST
X(NC)
5V
Servo
DS3218
PA_23 (PWM_CH0)
GND
ATSAM4S4B

Appendix C: FPGA Schematic


PID:control1
load
clk
reset
e_in[15..0]
done
u_out[15..0]
done~0
1'h0
always0
e_prev1~[15..0]
e_prev1~[31..16]
16'h0
e_prev1[15..0]
D
CLK
SCLR
Q
u_prev~[15..0]
e_prev2~[15..0]
e_prev2~[31..16]
u_prev~[31..16]
e_prev2[15..0]
u_prev[15..0]
Mult0
Mult1
Mult2
16'h1
A[15..0]
B[15..0]
OUT[15..0]
Add0
Add1
Add2
1'h0 CIN
A[16..0]
B[16..0]
OUT[16..0]
15:0
done~1
1'h1
done~2
u_out~[15..0]
done~reg0
u_out[0]~reg[15..0]
ENA