B.I.T.S.H.I.F.T.E.D. D.A.T.A.

Bits **I**nto a **T**imed **S**ystem in **H**DL using **I**nverse kinematics, **F**inite state machines, and **T**ransceivers in **E**mbedde**d** **D**igital **A**rchitecture **T**o perform **A**ctuation

Topic	Active Learning Design Project
Course	ENEL 453: Digital Systems Design
Date	December 2nd, 2024
Name	Aleks Berezowski
UCID	30112220

Table of Contents

Introduction

This project takes a target pair of x- and y-coordinates meant for a 2-degree-of-freedom robot arm, calculates the needed servo angles to move the arm to the coordinates via inverse kinematics, and performs the arm movement.

The project had the following targets:

1. Calculation of inverse kinematics via a Python generated 2-key lookup table
2. Control of servos using PWM signals without burning them out
3. UART communication with a computer to receive commands and display a simulation
4. Reading a distance sensor's PWM output

Of these targets, three of four were achieved. Reading the distance sensor's PWM output was not achieved due to issues reading the incoming PWM signal, even though the simulation of the ultrasonic sensor worked as intended. Further, the mechanical setup did not work due to the chosen servos lacking the torque needed to control the arm.

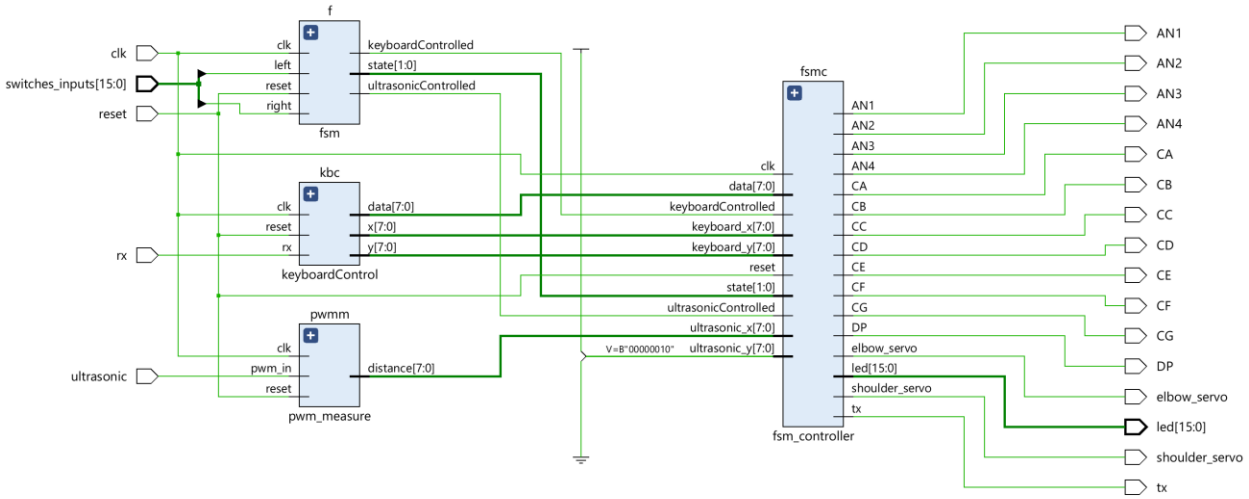
This report is split into the following sections:

1. Architecture: Explores the high-level design of this project, which includes the top-level RTL schematic, the physical setup of the Basys3 and peripherals, a schematic diagram of all external circuitry, and a brief description of top-level blocks.
2. Detailed Design: Each module will be covered in depth, which includes RTL schematics, a state diagram if appropriate, the purpose of the module, inputs, outputs, details about how the module works, possible improvements, and credits if outside sources were used.
3. Implementation Results: Shows the utilization and timing report summaries.
4. Testing: An overview of testbenches used, in addition to manual testing done.
5. Python: Scripts developed in Python were used to generate one of the modules and run the arm simulation is covered in this section.
6. Learnings: Notable lessons learned and interesting discoveries discussed here. This a more informal section and was elected to be included for personal reflection.
7. Conclusion: A summary of all sections.
8. Appendix 1: All SystemVerilog code and the constraints file. Please note Python files will not be included as the entire project is available on GitHub here: https://github.com/thesixtium/FPGA-Final-Project

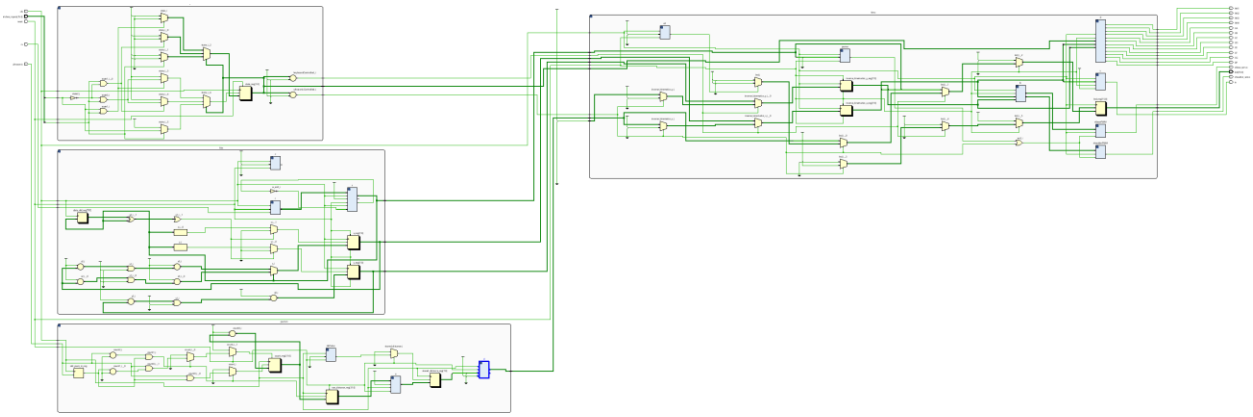
Please note "Appendix 2: AI Transcripts" is not included due to no AI generated code making it into the final iteration of this project (AI was only used for the generation of the project name), even though AI generation for code was attempted. Further, "Appendix 3: Team Members' Contributions" is not included due to this being a solo project.

Architecture

RTL Schematic for Top Level



RTL Schematic for Top Level

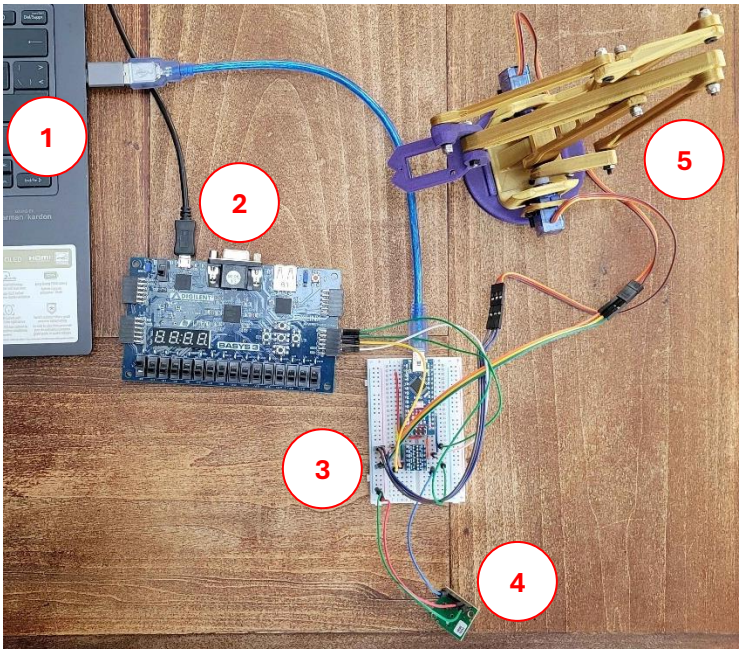


RTL Schematic with One Level Expanded

Basys3 Physical Setup

The physical setup has five components:

1. Laptop: Sends UART commands, receives UART data, and powers the Basys3. Connects to component 2.
2. Basys3: Controls the project and interfacing with all components. Connects to component 1 and component 3.
3. Breadboard: Contains a level shifter to convert the servos' and ultrasonic sensor's 5V0 logic to Basys3's 3V3 logic. Also contains an Arduino Nano that is only used as a convenient 5V0 and 3V3 power supply. Connects to component 2, component 4, and component 5.
4. Ultrasonic Sensor: Gets the distance of an object. Connects to component 3.
5. Servo Controlled Arm: Contains two servos. Connects to component 3, everything except the servos were eventually disconnected due to problems with servo torque.



Top-down view of the physical project

Throughout the project, the joint labelled 1 is referred to as the elbow joint and is controlled by the elbow servo. Likewise, joint 2 is referred to as the shoulder joint and controlled by the shoulder servo.

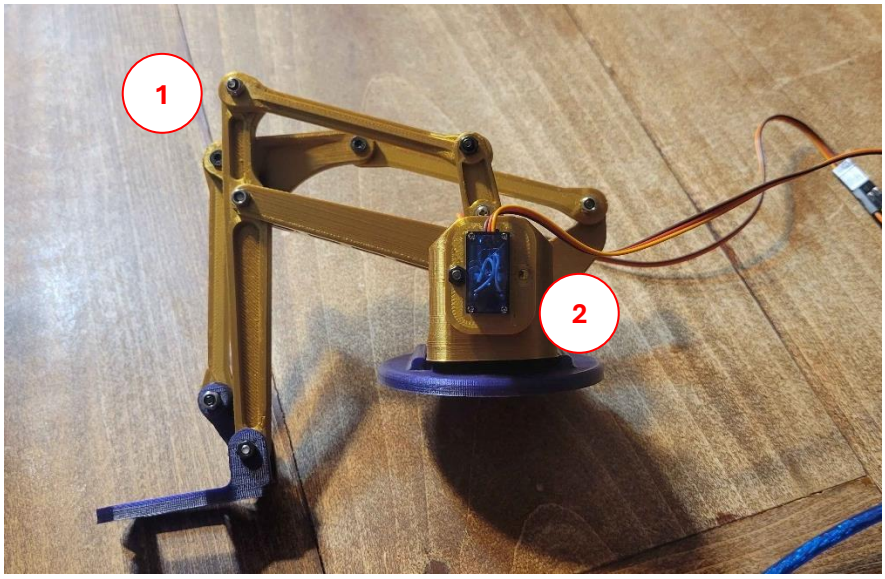
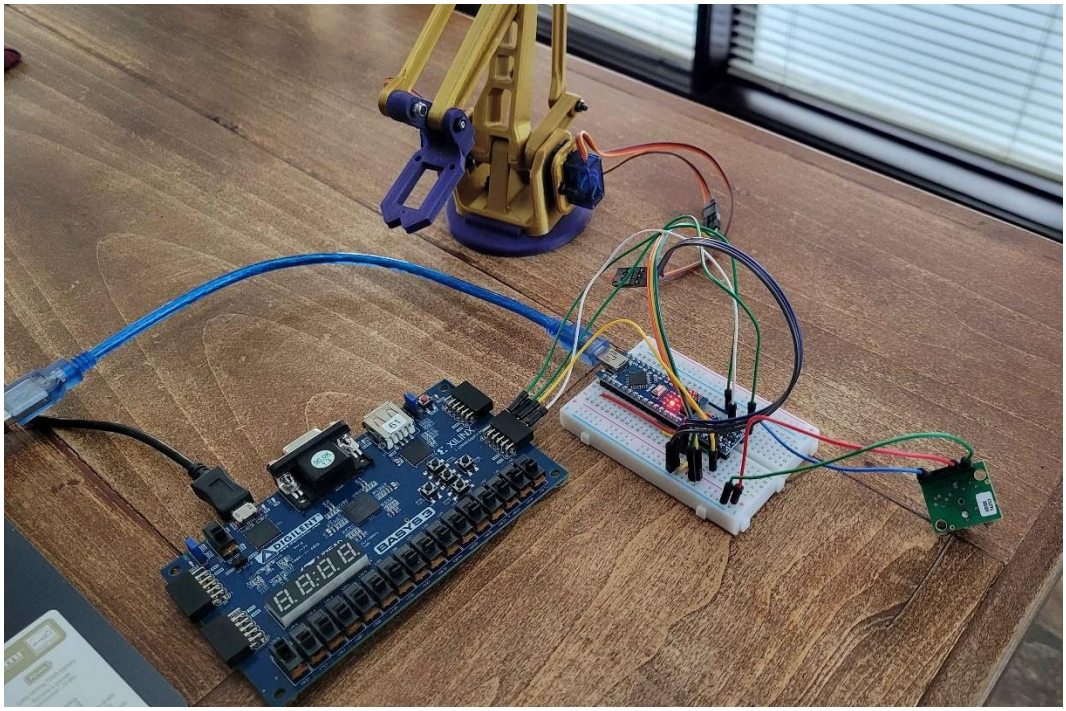
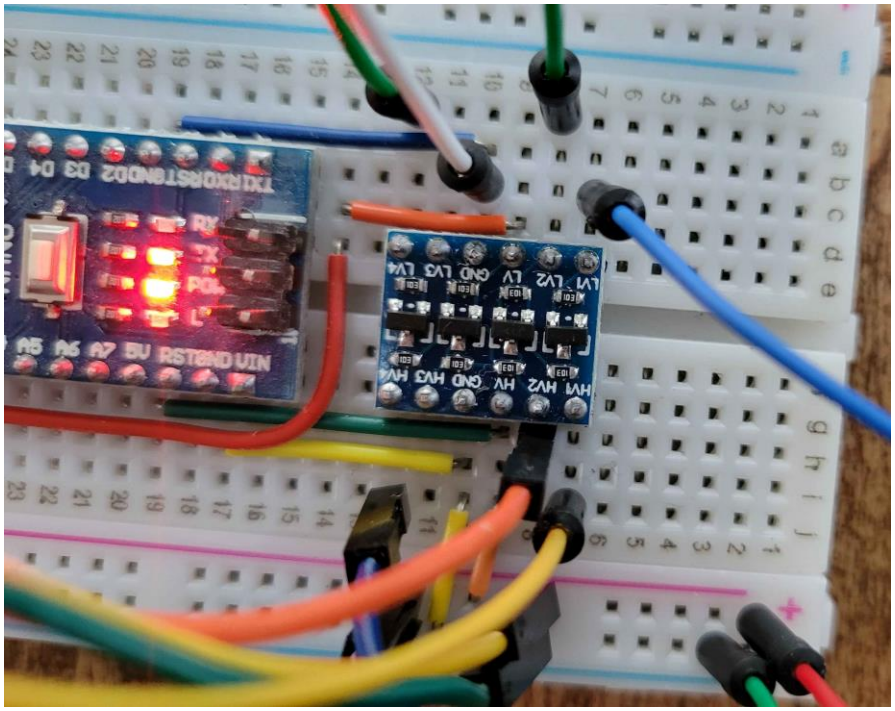


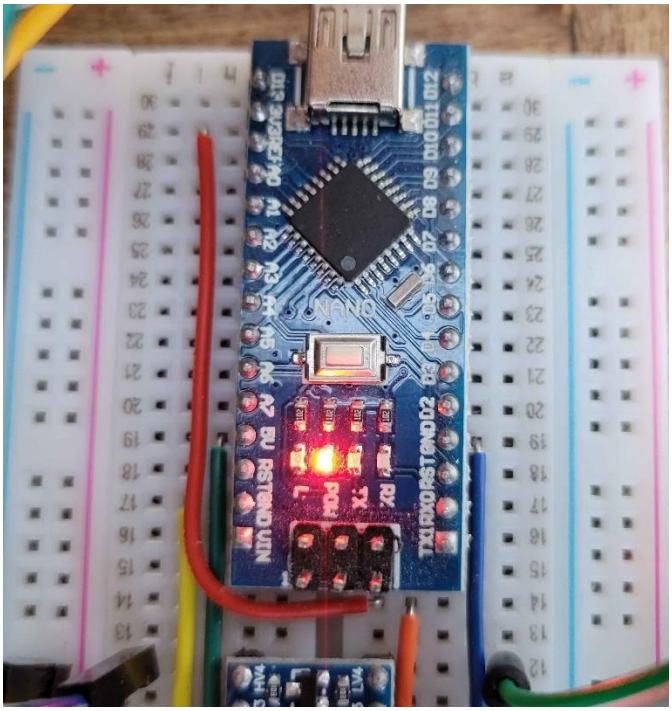
Image of the arm to be controlled



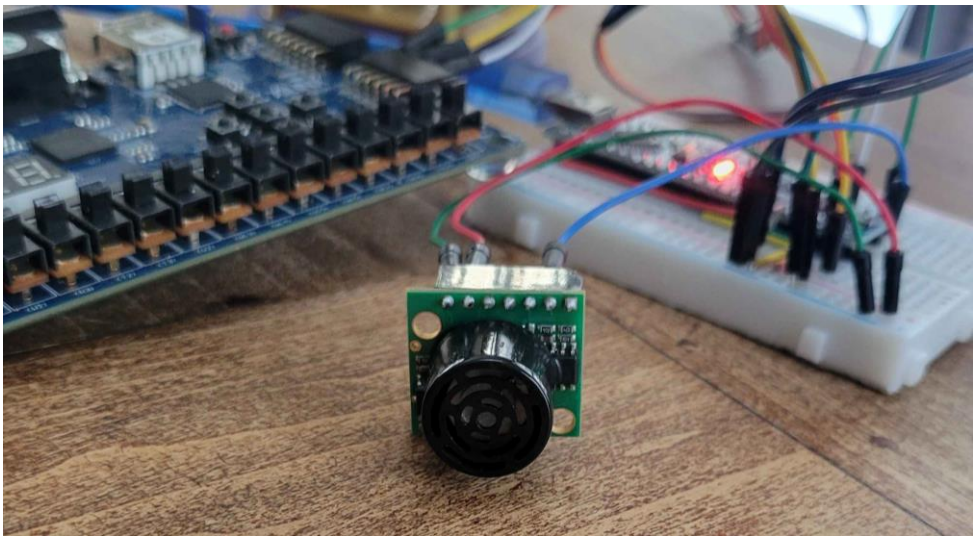
Isometric view of the project



Top-down view of the level shifter



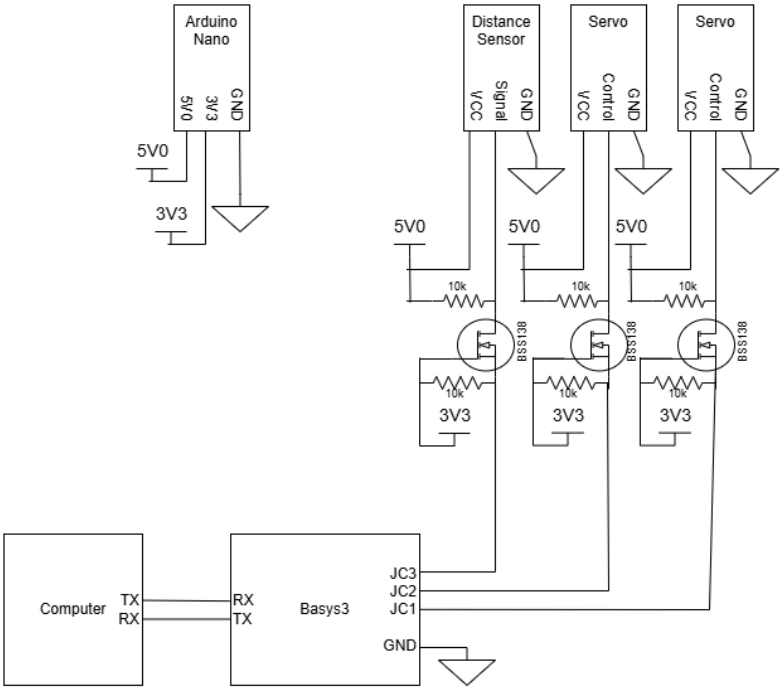
Top-down view of the Arduino Nano



Better view of the ultrasonic sensor

Schematic Diagram of External Circuitry

All the MOSFETs and resistors are integrated into an off-the-shelf bi-directional logic level shifter. Further, all 3V3 and 5V0 power supplies are from an Arduino Nano, and the ground of the Arduino Nano is connected to the ground pin on the JC connector.



Schematic Diagram

Description of Top-Level Blocks

FSM Module

This project is a Moore state machine, and the FSM Module is used to determine the state.

The inputs are clock, reset, switch 15, and switch 0. It outputs a state and two signals that determines if the arm is controlled by the keyboard or the ultrasonic sensor, which are all used by the FSM Controller Module.

PWM Measure Module

This module measures the input PWM signal from the ultrasonic sensor and translates it into a distance. This distance is set as the desired x-coordinate for when the arm is controlled by the ultrasonic sensor and is input into the FSM Controller Module.

The inputs are clock, reset, and the level-shifted input PWM signal from the ultrasonic sensor. The output is the distance, which is the desired x-coordinate for the arm.

Keyboard Control Module

This module interprets the computer's keyboard by reading the Basys3's UART rx port. The incoming keyboard button is decoded into modifications to the desired x- and y-coordinates for the arm. The incoming keyboard button is also output for debugging and demonstration purposes.

This module has two parameters: the maximum desired x- and y-coordinates. The inverse kinematics module uses a lookup table that has maximum x- and y-coordinates, thus maximums on the output coordinates ensure the desired coordinates do not exceed those bounds. The inputs are clock, reset, and the incoming rx line from the computer. The outputs are the data received, the desired x-coordinate, and the desired y-coordinate. All outputs are used by the FSM Controller module.

FSM Controller Module

The FSM Controller Module enacts the state-based logic of the finite state machine. It takes in the state produced by the FSM Module and changes the Basys3's outputs accordingly.

The inputs are the clock, reset, a logic line indicating if the arm should be keyboard controlled, the desired x- and y-coordinates if the arm is keyboard controlled, the incoming data from the keyboard, a logic line indicating if the arm should be ultrasonic sensor controlled, and the desired x- and y-coordinates if the arm is ultrasonic sensor controlled. The outputs are the Bayas3 LEDs, the PWM signal for the shoulder servo, the PWM signal for the elbow servo, the tx line of the UART connection, and all the display cathodes and anodes for the 7-segment displays.

Level Shifter Peripheral

Due to the servos and ultrasonic sensor having 5-volt logic levels, a bi-directional logic level shifter was needed. The level shifter preserves PWM signals going from 3V3 to 5V0 (used to control servos from the Basys3) and signals going from 5V0 to 3V3 (used when the ultrasonic transmits data to the Basys3).

Each channel of the level shifter is a MOSFET and two pull-up resistors. Three of the five channels are used: two going from 3V3 to 5V0 for servo control, and one going from 5V0 to 3V3 for ultrasonic sensor data. The level shifter can be found here: https://www.amazon.ca/KeeYees-Channels-Converter-Bi-Directional-Shifter/dp/B07LG646VS

Servos Peripheral

The servos used for this project are SG90 servos. They are controlled by a 20ms period PWM signal where the duty cycle of the PWM signal controls the position of the servo. According to manufacturer specifications, a 5% duty cycle should position the servo at 0 degrees and 10% should position the servo at 180 degrees.

The servos operate at 5 volts and have a 5-volt logic level. The datasheet can be found here: http://www.ee.ic.ac.uk/pcheung/teaching/DE1_EE/stores/sg90_datasheet.pdf

Ultrasonic Sensor Peripheral

The ultrasonic sensor used was an MB1000 from MaxBotix, with the goal of allowing the robotic arm to touch an object place in front of it by going to the position specified by the ultrasonic sensor. Its 20 Hz PWM signal was chosen as the data output.

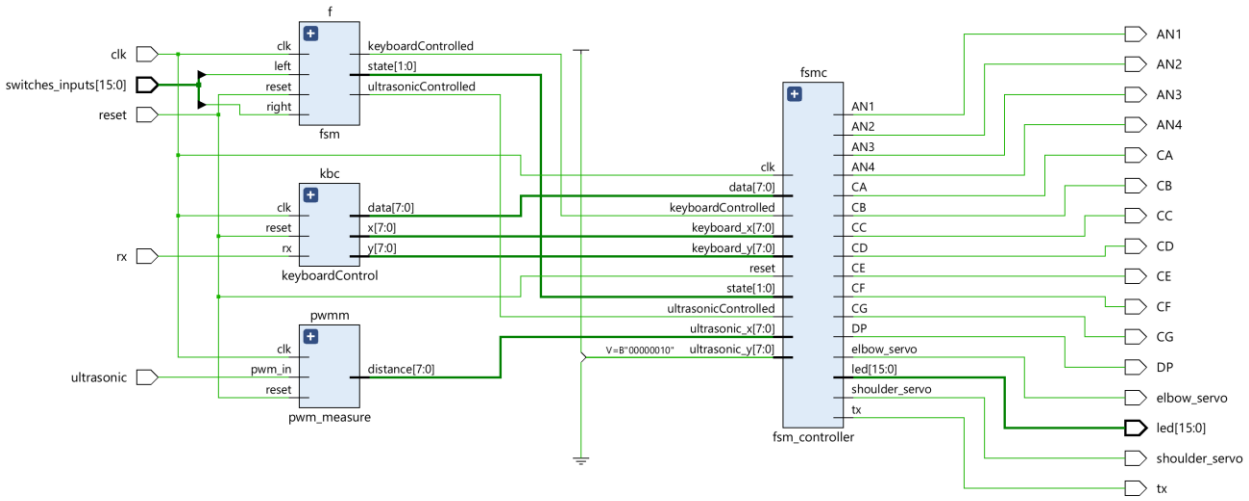
The ultrasonic sensor can be found here: https://maxbotix.com/products/ultrasonic_sensors_mb1000

Detailed Design

For all modules, assume that unless otherwise stated any input labelled “clk” is an input for the main 10 MHz clock signal, and “reset” is a synchronous, active high reset to reset the project to a known state.

It is also important to note that the Basys3 operates at 10 MHz in this project. It was originally designed around a 100 MHz operating frequency; however, the Worst Negative Slack was very close to negative or zero on several implementations. Thus, the clock speed was lowered to ensure the project met timing; it was lowered by a factor of 10 because it made redoing math relating to the PWM signals easy to do.

Top Level



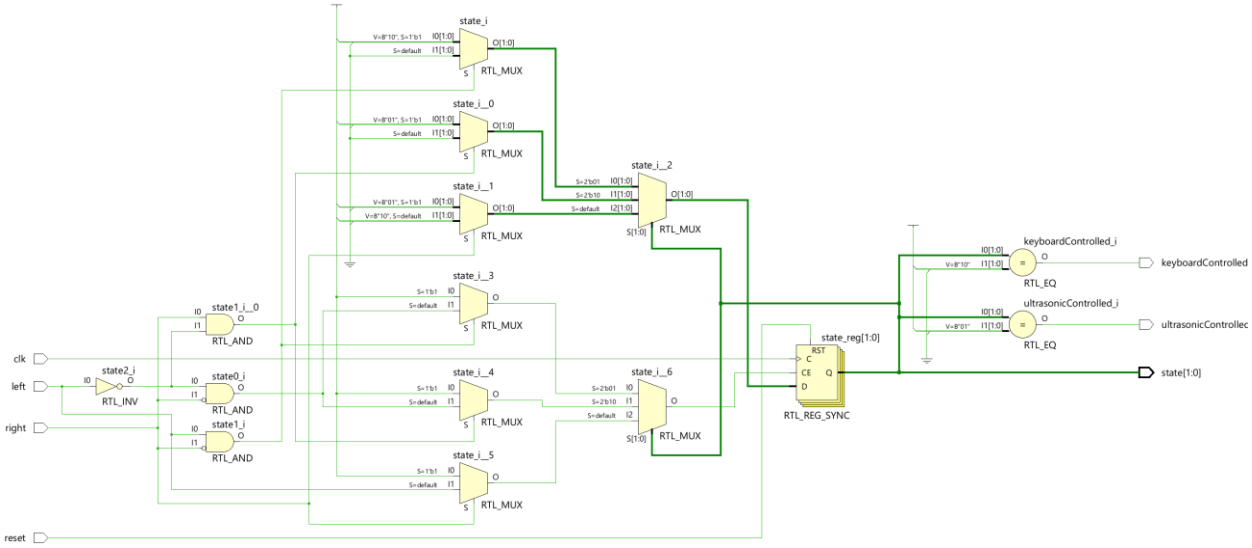
RTL Schematic for Top Level

Purpose: Connects the main sub modules to external inputs and outputs, as well as connects main sub modules together.

Inputs: clk, reset, rx is the receiving line of the Basys3’s UART connection, ultrasonic is the ultrasonic sensor’s PWM signal, and switches_inputs is the Basys3’s switches.

Outputs: tx is the transmitting line of the Basys3’s UART connection, shoulder_servo is the PWM signal to the shoulder servo, elbow_servo is the PWM signal to the elbow servo, and led is the Basys3’s LEDs. CA, CB, CC, CD, CE, CF, CG, and DP are all cathodes for the 7-segment displays, while AN1, AN2, AN3, and AN4 are all anodes for the 7-segment displays.

FSM



RTL Schematic

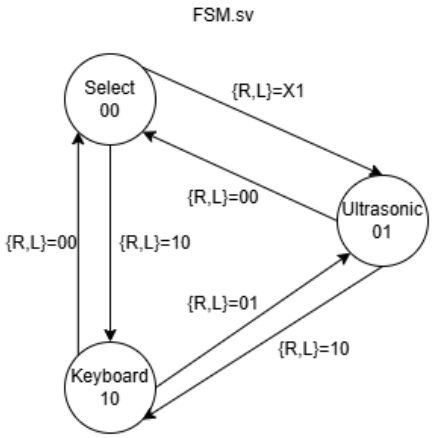


Diagram of the Finite State Machine

Purpose: Determine the state of the Basys3 finite state machine.

Inputs: clk, reset, right is the right-most switch, and left is the left-most switch.

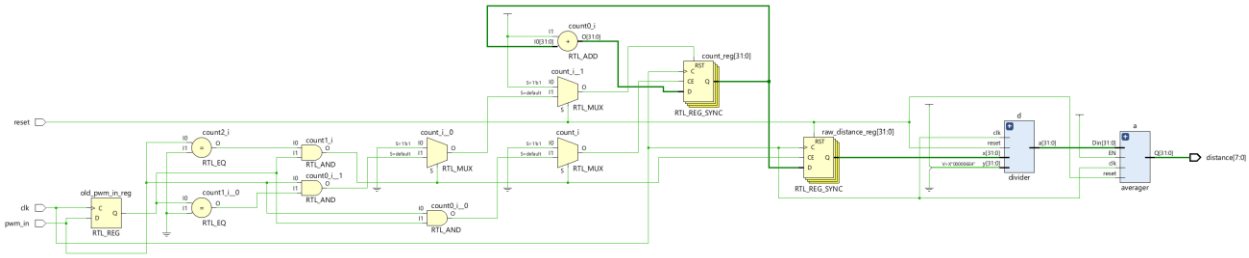
Output: state is the state of the finite state machine, keyboardControlled is a boolean stating if the arm should be keyboard controlled, and ultrasonicControlled is a boolean stating if the arm should be keyboard controlled.

Details: This module implements the logic for changing states of the Moore state machine. It is a Moore machine because the outputs depend only on the current state. Originally, the state

dependant logic (like changing the LEDs based on the state) was in this module, but it was cleaner to move that logic to a separate module.

Possible Improvements: A state machine may not have been needed to control this project, and it may have been better to use muxes to implement the functionality of changing the outputs. I found setting it up as a state machine helped my design organization, however it may have led to requiring more logic. Finally, the keyboardControlled and ultrasonicControlled signals may removable, in favor of decoding the state in the FSM controller.

PWM Measure



RTL Schematic

Purpose: Determine the length of the ultrasonic sensor's PWM signal and translate it into a distance reading.

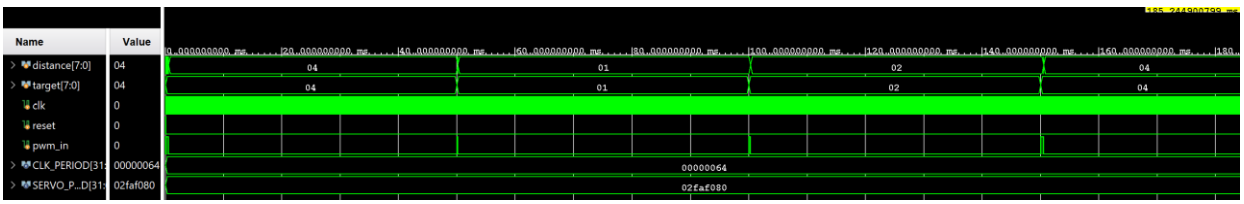
Parameters: DIVISION_AMOUNT is the scaling factor of the PWM count according to simulation testing.

Inputs: clk, reset, and pwm_in is the ultrasonic sensor's PWM signal.

Output: distance is the distance reading from the ultrasonic sensor.

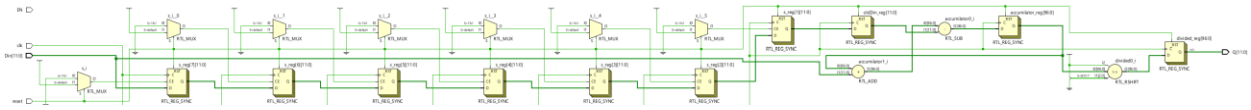
Details: When the PWM signal goes from low to high, the module starts counting how many clock cycles have passed. When the PWM signal then goes from high to low, the number of clock cycles passed into a moving average filter and scaled according to manufacturer specifications. Finally, when the PWM signal goes from low to high the count is restarted.

Possible Improvements: The biggest improvement would be making this module work in implementation, instead of just in simulation. As shown below, when performing a post-synthesis simulation on the module with the ultrasonic sensor's PWM signal, the output is equal to the desired target. However, when manually testing, the values output are incorrect.



Result of testbench simulation

Averager



RTL Schematic

Purpose: Provides a moving average of incoming samples.

Parameters: power specifies the number of samples in the moving average's window (2^N samples in the window), and N specifies the width of the input data.

Inputs: clk, reset, EN is the enable pin, and Din is the data in.

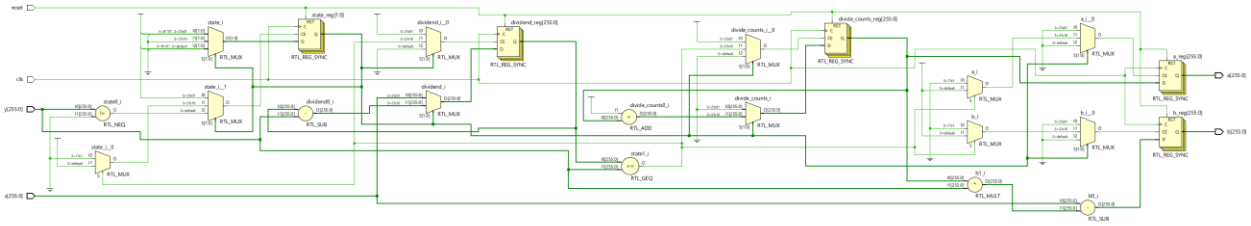
Output: Q is the data out.

Details: This average works as an accumulator with a shift register. When a new data is input, it is added to the accumulator variable and the shift register. The data output from the shift register is then subtracted from the accumulator variable and discarded. Thus, the accumulator variable is equal to the sum of all data in the shift register, and the average of all the data can be calculated by the accumulator variable divided by the length of the shift register.

Possible Improvements: Unsure if there is a better way to average data in a moving window, however an algorithm that does not require a big shift register for big moving average windows and no division would be an improvement.

Credit: The framework and reset of the module are taken from the provided Lab 5 code.

Divider



RTL Schematic

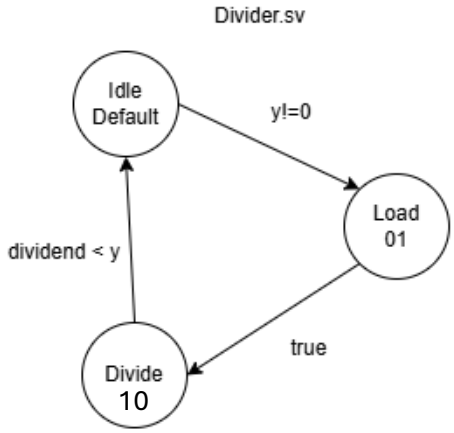


Diagram of the Finite State Machine

Purpose: Divide a 32-bit dividend by a 32-bit divisor, returning a quotient and a remainder. This is used to scale the output of the ultrasonic sensor.

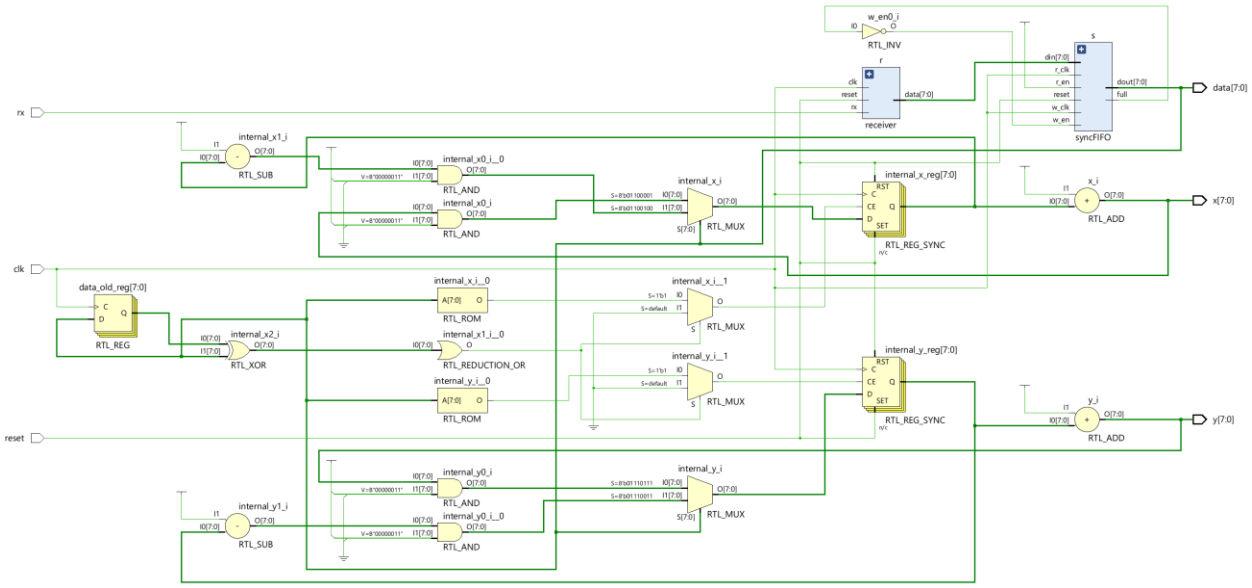
Inputs: clk, reset, x is the 32-bit dividend, and y is the 32-bit divisor.

Output: a is the 32-bit quotient, and b is the 32-bit remainder.

Details: A simple algorithm of repetitive subtraction was implemented for division. The divisor is subtracted from the dividend until the dividend is smaller than the divisor. The number of subtractions it took to get to that state is then returned as the quotient.

Possible Improvements: The chosen division algorithm is slow and could be replaced with a faster one, or all of this could be replaced by IP from the Vivado IP catalog. Further, a ready signal could be implemented to show that division is done.

Keyboard Control



RTL Schematic

Purpose: Change the desired x- and y-coordinates for the arm based on received UART commands.

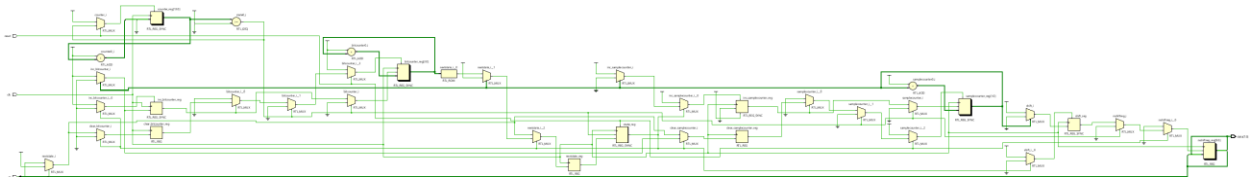
Inputs: clk, reset, and rx is the receiving line of the Bayas3's UART.

Output: data is the UART command received as bits, x is the desired x-coordinate, and y is the desired y-coordinate.

Details: The UART commands are translated by the receiving module, and then put into a first-in first-out queue. The data is put into the queue because when it was not, the keyboard presses were not always interpreted correctly. Data is then taken out of this queue and interpreted to change the x- or y-coordinates of the arm.

Possible Improvements: Instead of adding a queue as a buffer, it would be better to fix the underlying problem that the queue patched.

Receiver



RTL Schematic

Reciever.sv

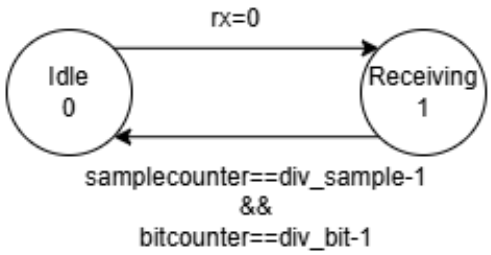


Diagram of the Finite State Machine

Purpose: Receive bits from the Basys3's UART rx line and turn it into usable data.

Inputs: clk, reset, and rx is the receiving line of the Basys3's UART line.

Output: data is the 8-bit data translated from the rx line.

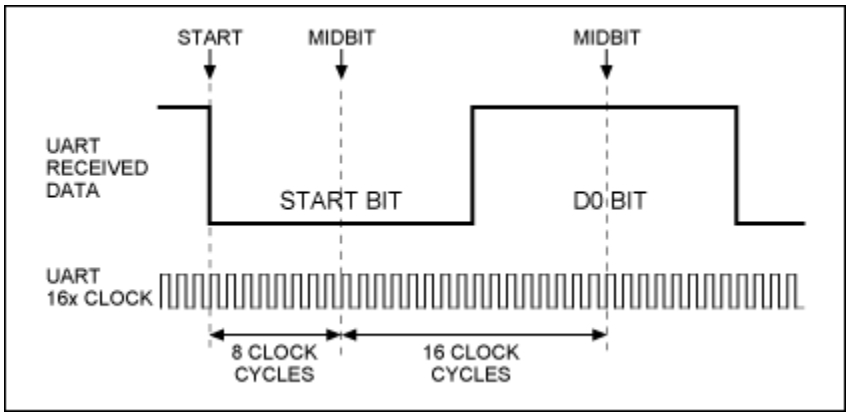
Details: The receiver works by parallelizing the incoming sequential bits, as data comes in on a single line and is outputted on eight lines. This is done via a state machine with two states: idle and receiving.

In receiving, the module waits until the rx input is pulled low, which signals the start of a UART packet, and switches to the receiving state. In the receiving state, data is added to a shift register as it is received to parallelize it. However, this is done by two separate counters: bitcounter and samplecounter.

Bitcounter keeps track of the number of bits received, which is important as UART packages are 10-bits long. This counter is used to return to an idle state when the last data bit is received.

While UART is a protocol that needs to decide on a clock frequency between both devices (known as the baud rate), a clock frequency or a clock synchronization signal is never sent between devices. To address this, the incoming rx line is heavily oversampled. The Basys3 will sample the

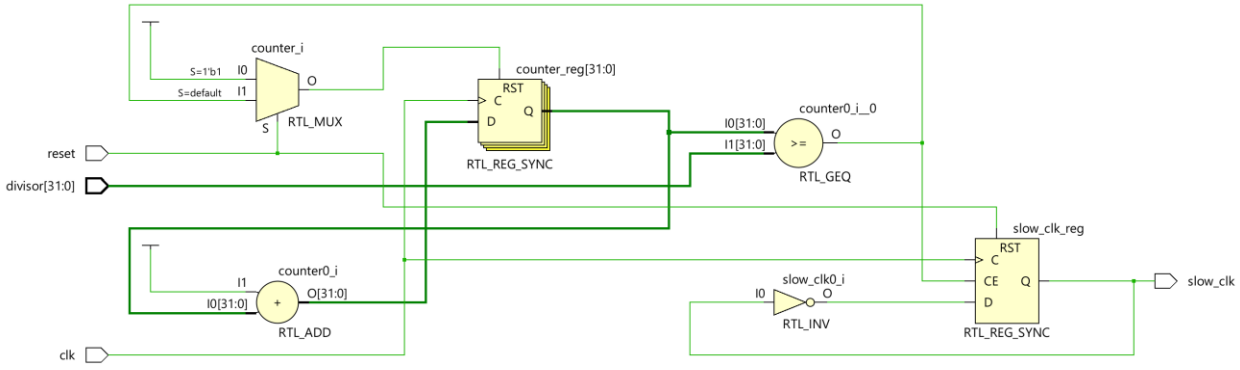
9600 Hz rx line at the board's clock frequency of 10 MHz. However, only the midbit sample of each rx bit is considered the intended value of the UART packet's bit and recorded. This is done by pre-calculating how many Basys3 clock cycles should occur per UART bit, and then dividing by two to get how many samples should happen before the midbit. Once the receiving state is reached, this module waits until the midbit, samples it, and repeats this until all the bits have been sampled. Due to taking the middle of the rx bit, if what the module considers the middle of the bit and the actual middle of the bit are not perfectly aligned the sample will still be a correct representation of the packet's intended bit.



A figure showing a UART frequency (top), a clock 16 times faster than the UART frequency (analogous to the Basys3), and where midbits are.

Credit: This code was taken from here, with comments added to show understanding:
https://www.instructables.com/UART-Communication-on-Basys-3-FPGA-Dev-Board-Power-1/

CLK Divider



RTL Schematic

Purpose: Generate a lower frequency clock signal from a higher frequency input clock signal.

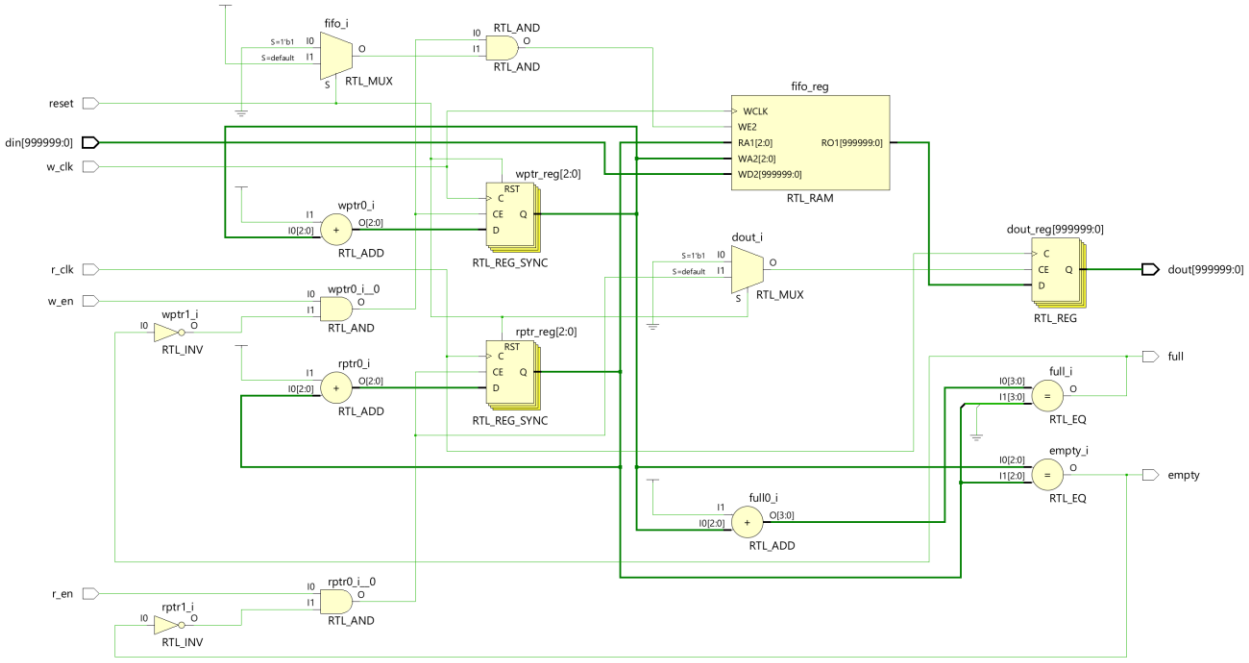
Inputs: clk, reset, and divisor as a 32-bit divisor.

Output: slow_clk as the generated clock signal.

Details: This module counts up until the count is equal to divisor, then inverses the output slow_clk signal and resets the count. Because every clk signal the count is increased by one, the output slow_clk has a frequency of the input clock frequency divided by the divisor.

Possible Improvements: This module should not be used because manually generating a slower clock frequency could break timing, thus this module should be substituted for a properly generated slower Basys3 clock.

Sync FIFO



RTL Schematic

Purpose: This synchronous first-in first-out queue (FIFO) stores incoming data, and then produces data in the order it was received. It is used to hold keyboard commands temporarily.

Inputs: reset, w_clk is the clock for writing to the queue, r_clk is the clock for reading from the queue, w_en is the write enable, r_en is read enable, and din is the data to be written to the queue.

Output: dout is the data output of the queue, empty is a flag indicating if the queue is empty, and full is a flag indicating if the queue is full.

Details: This module has a pointer for writing to memory and a pointer for reading from memory. The writing pointer is at the end of the queue, and the reading pointer is at the front of the queue. As the queue is written to every positive edge of w_clk, the writing pointer moves forward one spot. It eventually loops to the first queue address when the address of the writing pointer overflows to a value of 0.

```
always @ (posedge w_clk) begin
    if(reset) begin
      wptr <= 0;
    end else begin
      // If writing and not full, then write incoming data
      // Overflow makes the write pointer effectively a circular buffer
      if (w_en & !full) begin
        fifo[wptr] <= din;
        wptr <= wptr + 1;
      end
    end
  end
```

SystemVerilog taken from syncFIFO.sv showing writing to the queue

The read pointer works the same way, except reading data from memory and working on the positive edge of r_clk.

```
  always @ (posedge r_clk) begin
    if (reset) begin
      rptr <= 0;
    end else begin
      // If reading and not empty, output the data and increase pointer
      if (r_en & !empty) begin
        dout <= fifo[rptr];
        rptr <= rptr + 1;
      end
    end
  end
```

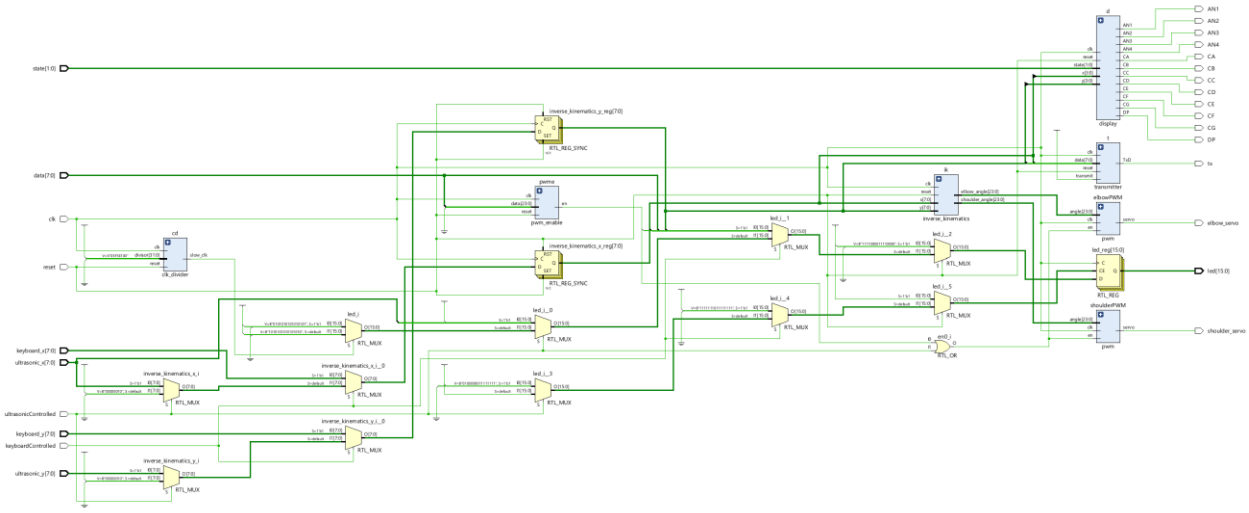
SystemVerilog taken from syncFIFO.sv showing reading from the queue

Due to the overflow on the writing pointer and reading pointer, this queue works as a circular buffer. The queue is full if the next spot that the writing pointer wants to go to is where the reading pointer is, and empty if the reading point is where the writing pointer is.

Possible Improvements: Due to this module working as intended and being fairly efficient, I am unsure what can be improved.

Credit: The base verilog that was modified was taken from
https://www.chipverify.com/verilog/synchronous-fifo

FSM Controller



RTL Schematic

Purpose: Enacts the state-based logic of the finite state machine, which includes switching if the keyboard or ultrasonic sensor is controlling the arm and changing the display.

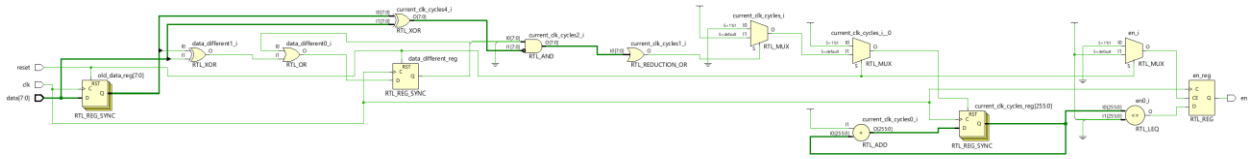
Inputs: clk, reset, state is the state of the FSM, keyboardControlled indicates if the arm should be keyboard controlled, keyboard_x is the desired x-coordinate if the arm is keyboard controlled, keyboard_y is the desired y-coordinate if the arm is keyboard controlled, data is the data from the input UART, ultrasonicControlled indicates if the arm should be ultrasonic controlled, ultrasonic _x is the desired x-coordinate if the arm is ultrasonic controlled, and ultrasonic _y is the desired y-coordinate if the arm if ultrasonic controlled.

Output: led is the output to the Basys3 LEDs, shoulder_servo is the PWM signal to the shoulder servo, elbow_servo is the PWM signal to the elbow servo, and tx is the data to be sent to the computer over UART. CA, CB, CC, CD, CE, CF, CG, and DP are all cathodes for the 7-segment displays, while AN1, AN2, AN3, and AN4 are all anodes for the 7-segment displays.

Details: This module handles changing the LEDs and what is controlling the arm via multiplexors based on the state. It also houses the PWM Enable, Inverse Kinematics, PWM, Transmitter, and Display sub-modules as they are all partially controlled by the current state.

Possible Improvements: The keyboardControlled and ultrasonicControlled flags could probably be removed in favor of decoding the state.

PWM Enable



RTL Schematic

Purpose: When the data to the servo changes, this module enables the servo PWM for a length of time to allow the servo to get to the desired position before turning the servo PWM off. This is needed because the servos kept burning out when data was constantly fed to them as they would keep attempting to readjust.

Parameters: N is the input width of the data to monitor, CLK_CYCLES is how many clock cycles to keep the enable high before turning off.

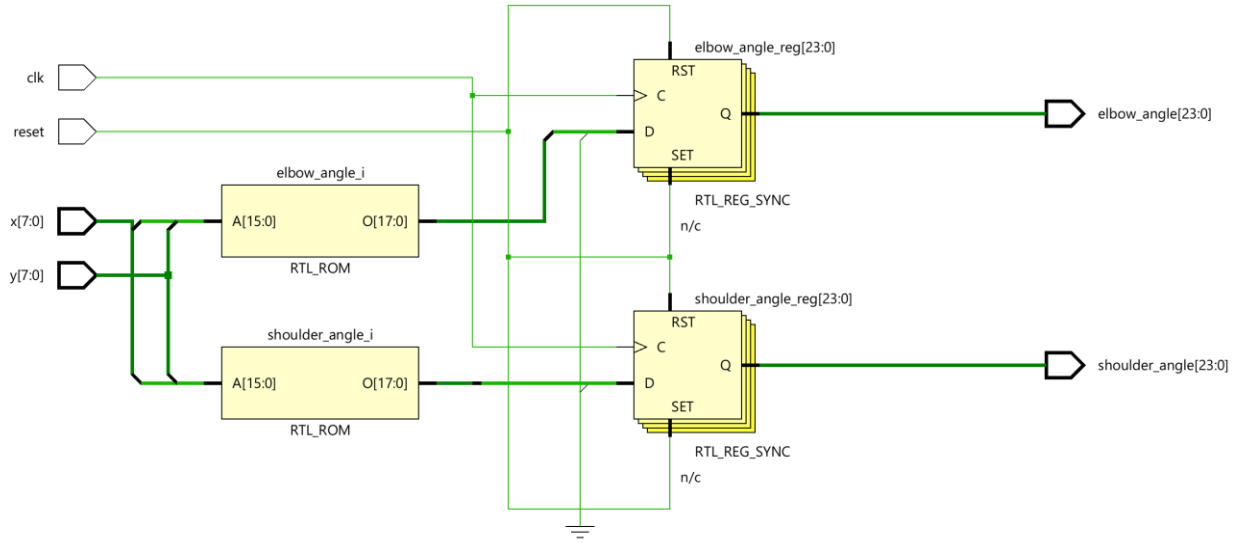
Inputs: data is an N-bit input from the keyboard control module, clk, and reset.

Output: en is the enable for the servo PWM signal.

Details: This module has a counter that starts when the current input data is different from the input data from last clock cycle. While the counter is active the enable signal is turned on. When the counter is turned off due to it reaching the desired number of clock cycles the enable signal is turned off.

Possible Improvements: The internal variable current_clk_cycles should be scaled with the CLK_CYCLES parameter. Additionally, the logic for putting enable high or low may be able to be simplified.

Inverse Kinematics



RTL Schematic

Purpose: Translate the desired x- and y-coordinates to positions for the servo. It was decided not to perform the inverse kinematics calculations and instead look them up because doing the calculations would require a lookup table for trigonometry functions, so it seemed better design-wise to do a lookup table of the output of inverse kinematics instead.

Inputs: clk, reset, x is the desired x-coordinate of the arm, and y is the desired y-coordinate of the arm.

Output: shoulder_angle and elbow_angle are the required angle of the servos.

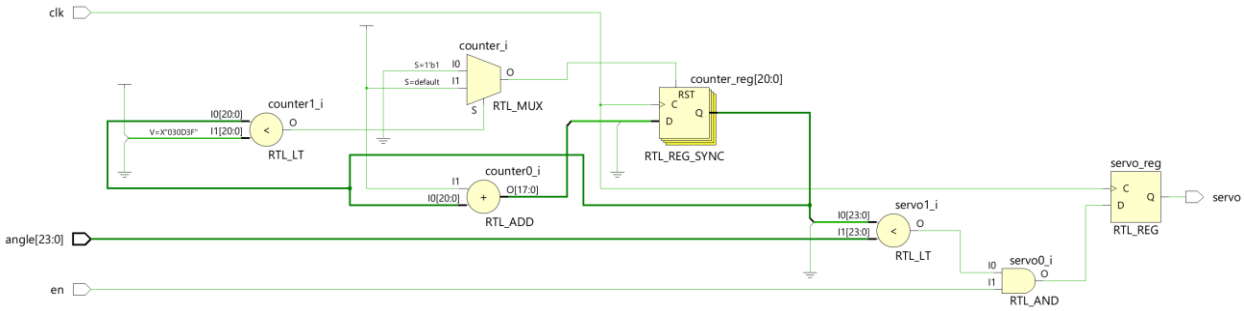
Details: The logic of this module was generated by a Python script, which will be covered in a later section. It works as a big lookup table based with keys of the desired x- and y-coordinates, which are appended together for the case statement. Based on the case statement, different precalculated servo positions are returned.

```
        16'b00000001_00000010 : begin  // x=1 y=2
          // elbow = 131.6
          // shoulder = 136.2
          elbow_angle <= 86605;
          shoulder_angle <= 89988;
        end
```

SystemVerilog taken from inverse_kinematics.sv showing a sample case

Possible Improvements: The x and y inputs could be converted from integers to fixed point numbers to allow more nuance in the movement. Further, the width of the x and y inputs could be scaled via a parameter based on the maximum value possible.

PWM



RTL Schematic

Purpose: Produce a PWM signal to drive SG90 servos.

Inputs: clk, en is an enable, angle is a 23-bit angle value.

Output: servo as the output PWM signal for a servo.

Details: This module is very similar to the clock divider module in that it counts to a specific number (angle, in this case). However, when count equals angle the PWM module changes the signal from high to low and continues counting instead of resetting the count and flipping the output signal. When count equals the period of the PWM signal the output is set to high and the count is reset. It is important to note that angle is not in a unit such as radians or degrees, it is the required count to set the servo to the desired position. After testing, it was determined that the angle input needed and the output angle of the servo were not linearly correlated, as shown here:

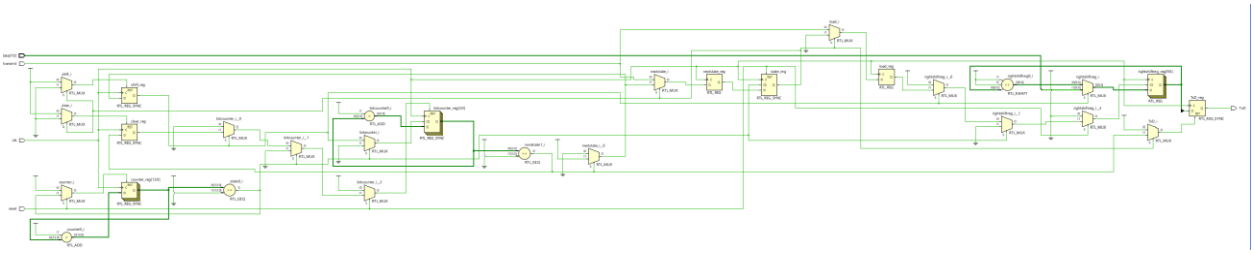
Because the servos often needed to go under 50 degrees, the lack of linear correlation was very problematic.

Possible Improvements: More testing should be done to figure out a better way to convert the desired angle to bits than with a lookup table. Additionally, testing of the servos did not show a

linear relationship between PWM signal length and servo position, which should be explored further. Finally, there is a known bug that some servos will get stuck at either 0 degrees or 180 degrees and be unable to recover unless physically pushed, regardless of load on the servo or requested angle. Additionally, a reset signal should be added.

Credit: I used this video to learn how to initially generate PWM signals because I started this project before we did it in class: https://www.youtube.com/watch?v=zNln9hJ5J78. I also used this video to learn how to apply PWM signals to servos and setup servo-specific parts of the project: https://www.youtube.com/watch?v=JpzeoQI2MII. Finally, the manufacturer's datasheet gave good information on the length of PWM signal needed: http://www.ee.ic.ac.uk/pcheung/teaching/DE1_EE/stores/sg90_datasheet.pdf

Transmitter



RTL Schematic

Transmitter.sv

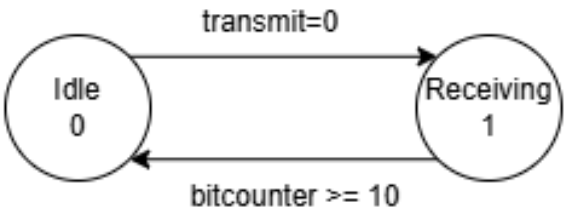


Diagram of the Finite State Machine

Purpose: Transmit bits over the Basys3's UART tx line.

Inputs: clk, reset, transmit is a boolean to start transmitting data, and data is the data to transmit.

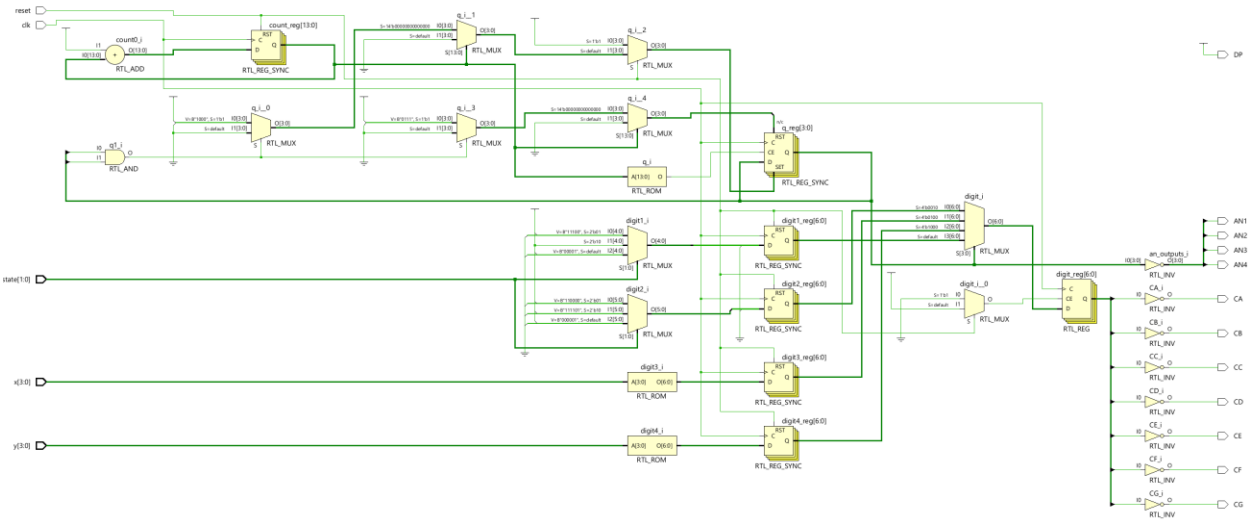
Output: TxD is the output bits to be sent over the Basys3 UART tx port.

Details: The transmitter takes parallel data and makes it sequential so it can be sent over the tx line. It does this by putting the data and UART protocol bits into a shift register and transmitting them one at a time.

Possible Improvements: The code should be upgraded from Verilog to SystemVerilog. It would also be nice to replace the magic number of 10415 with a way to calculate it via clock frequency and baud rate parameters. Further, it may be better to use Vivado to make a second clock that runs at the needed baud rate and use it as the transmission clock instead of using a counter.

Credit: This code was taken from here, comments were added in to show my understanding: https://www.instructables.com/UART-Communication-on-Basys-3-FPGA-Dev-Board-Power/

Display



RTL Schematic

Purpose: Display values to the four 7-segment displays.

Inputs: clk, reset, state is the state of the FSM, x is the desired x-coordinate of the arm, and y is the desired y-coordinate of the arm.

Output: CA, CB, CC, CD, CD, CE, CF, CG, and DP are all outputs to the cathodes of the four 7-segment displays and used to choose which segments are illuminated. AN1, AN2, AN3, and AN4 are all outputs to the anodes of the 7-segment display and are used to choose which of the four 7-segment displays are active.

Details: The module displays the current state of the FSM in the first two 7-segment displays (ul for ultrasonic controlled, bd for keyboard controlled, and -- for the default state), the desired x-coordinate in the third 7-segment, and the desired y-coordinate in the fourth 7-segment. It is

implemented similarly to how it was in lab sessions; however multiple lab modules are combined into this one module due to needing less flexibility in what is displayed.

Possible Improvements: It might have been better to keep the display module split into smaller submodules for clarity. Additionally, having the same decoder written twice and only switching the input (x or y) is not phenomenal and should be replaced by a module to encourage code reuse.

Credit: Large parts of this code are taken from code provided for lab sessions.

Implementation Results

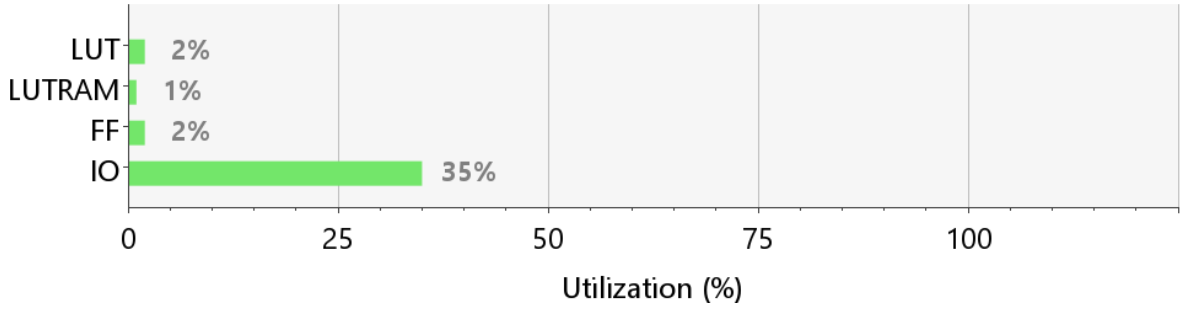
Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	90.847 ns	Worst Hold Slack (WHS):	0.052 ns	Worst Pulse Width Slack (WPWS):	3.750 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	1294	Total Number of Endpoints:	1294	Total Number of Endpoints:	725

All user specified timing constraints are met.

Design Timing Summary

Resource	Utilization	Available	Utilization %
LUT	461	20800	2.22
LUTRAM	17	9600	0.18
FF	701	41600	1.69
IO	37	106	34.91



Design Utilization Summary

Testing

All modules except averager, receiver, transmitter, and display had testbenches written.

Testbenches

While the final project runs at 10MHz, several modules were tested at 100MHz for two reasons. First, if they work at 100MHz then they should meet timing at 10MHz. Second, the project was originally designed to run at 100MHz, thus early modules were tested at 100MHz.

All test benches followed the same format:

- Set timescale
- Declare and initialize all input and output signals
- Instantiate the unit under test
- Start oscillating the clock
- Reset the unit under test
- Stimulate the input signals
- Use $stop to stop the simulation while not closing it

All testbench waveforms were then visually inspected to ensure the output signals were correct.

Manual Testing

Manual testing focused on ensuring the entire system works, with extra time spent on ensuring the modules without testbenches were confirmed to be working.

Receiver was tested by sending serial data from a computer to the Basys3 and displaying the received bits on the LEDs.

Transmitter was tested by sending known patterns from the Basys3 to the serial interface on a computer and checking the results.

Display was visually tested by checking the data output by the transmitter module against the display.

The entire project was tested by connecting to the Python simulation, inputting UART commands and switching states via onboard switches, and checking outputs. By checking the inputs against the outputs of the physical servos and the simulation, it was confirmed that the project works as intended.

Test Record

Item Tested	Method	Result
clk_divider module	Testbench and Waveform Inspection	Passed
civider module	Testbench and Waveform Inspection	Passed
fsm module	Testbench and Waveform Inspection	Passed
inverse_kinematics module	Testbench and Waveform Inspection	Passed
pwm_measure module	Testbench and Waveform Inspection	Passed
pwm_enable module	Testbench and Waveform Inspection	Passed
pwm module	Testbench and Waveform Inspection	Passed
syncFIFO module	Testbench and Waveform Inspection	Passed
display module	Manual testing via stimulating switches	Passed
receiver module	Manual testing via computer serial terminal	Passed
transmitter module	Manual testing via computer serial terminal	Passed
Idle mode	Manual testing via stimulating switches	Passed
Keyboard Controlled Mode	Manual testing via Python script	Passed
Ultrasonic Controlled Mode	Manual testing via moving ultrasonic sensor	Failed

Python

Two Python scripts were used: a Jupyter Notebook to generate the SystemVerilog code to implement the inverse kinematics lookup table and a Python script to receive UART data and simulate what the arm should do.

lookup_generator.ipynb

Purpose: Automatically generates the SystemVerilog required to implement the inverse kinematics lookup table. It was cumbersome to manually input values into the inverse kinematics module, and automating it made development much easier.

Details: The notebook starts with results of manually testing servo angle inputs and observed servo positions, such as an input of 145,000 resulting in 90 degrees. Based off a linear interpretation of these values, the radians_to_bits function takes a desired radians and converts it into the angle value to be input into the PWM module.

Next, the beginning of the always block is printed with the reset statement and the start of the case statement. For each x- and y-coordinate in the action space defined (the arm can move to any coordinate where x and y are 1, 2, 3, or 4) the inverse kinematics are calculated. If the x- and y-coordinates are reachable by the arm, the case statement for that x and y is printed out with comments. Finally, the end of the case statement and end statements are printed, and the code is manually copied into the SystemVerilog file.

Possible Improvements: The first improvement would be implementing the non-linearity observed in the servos. The code could also automatically modify the SystemVerilog file and may be able to

be automatically ran when the synthesizer is run. Further, this could be modified to work with fixed point values instead of integers, giving the arm more freedom.

Credit: The Python math was taken from https://stackoverflow.com/questions/76970645/calculate-inverse-kinematics-of-2-dof-planar-robot-in-python

simulation.py

Purpose: Simulates how an arm controlled by this project would operate as a visual demonstration.

Details: This code takes the x- and y- coordinates provided by the Basys3 over UART and provides a visual simulation of how the arm would look. If in keyboard control mode, the user can send commands to the Basys3 to change the angle of the arm.

Credit: The Python math was taken from https://stackoverflow.com/questions/76970645/calculate-inverse-kinematics-of-2-dof-planar-robot-in-python

Learnings

First, I learnt more about state machines due to how much they were used in this project. I had a vague understanding of them before this project, but never used them due to not fully understanding them or appreciating how much simpler and efficient they can make a design. Due to my increased confidence in the design and implementation of them, I have started to implement one in my capstone project, which has led to a much easier to program system than the previous iteration.

Second, I now understand UART communication a lot better. Before this project I would use it via libraries, never thought about how it worked, and had some incorrect assumptions about it. However, this project taught me how it works at the bit level, which made me confident in my understanding and dispelled some of my assumptions about UART. Further, learning about UART taught me one method of crossing clock domains, which was oversampling and taking the mid bit. I found it really cool that I was able to interpret a synchronous digital signal without having its clock as an input.

I learnt about another way of crossing clock domains, which was through a first-in first-out queue. I first learnt how to make the queue and use overflow to create a circular buffer, which was something I would have thought of before. I then added in a way to have the read be on one clock frequency and the write be on another clock. Especially because of how often queues with two clocks are used in the application of FPGAs, I really enjoyed learning how they work and the implementation.

I loved learning how to generate a lookup table and communicate with the Basys3 via Python. I liked using my previous knowledge of Python and my new knowledge of Basys3 to speed up generating the lookup table, especially because with very few modifications I could generate a lookup table

that takes any number of input variables and outputs any number of values, using any formulas or other logic to specify what the values should be. Further, a lookup table has instant access compared to an algorithm that needs to compute the outputs at runtime, which could help if I ever had an extremely complex formula or logic with a definable set of inputs and outputs, a lot of storage, and a bottleneck of processing power.

Finally, I learnt about how to integrate different modules together on a structural level, and designing the modules around being able to integrate easily. This project used a lot of modules compared to what I have previously experienced, and I had to design most of the modules myself. This led to me thinking more about the inputs and outputs and where the module should be in the hierarchy. This is something I still need to learn a lot more about, as shown by me having some redundant inputs and outputs between modules currently, and there probably being a way to further clean up the structure of the project. As a software engineering student, I think this learning is probably the most important to me, as when I design big software systems I will have a better understanding of how to structure my code and the dataflow between independent processes.

Conclusion

This project implemented the inverse kinematics for a 2-degree-of-freedom robot arm that is controlled by servos. It did this with state machines, lookup tables, and user inputs into the system.

Of the four project targets, the calculation of inverse kinematics, the safe control of servos using PWM signals, and UART communication were achieved. The last target, reading the ultrasonic sensor's PWM output, was successfully simulated but not achieved in implementation.

This report covered the high-level architecture of the project, a detailed breakdown of each module, results of implementation, the testing procedure, how Python was used to assist this project, and learnings from this project.

Appendix 1: Code and XDC

averager.sv

```
// File info:
// - File from Lab 5 when we designed our own averager
// - Applies a moving average filter to incoming data

module averager
  #(parameter int
    power = 8, // 2**N samples, default is 2**8 = 256 samples
    N = 32)   // # of bits to take the average of
  (
  input logic clk,   // 10 MHz clock
  input logic reset,  // Active high reset
  input logic EN,    // Enable pin

  input logic [(N-1):0] Din,  // Data input

  output logic [(N-1):0] Q   // Averaged data
);
  localparam MAX_SIZE = power * N;

  logic [(N-1):0] s [0:(power-1)];
  int i;
  logic [(N-1):0] oldDin;

  logic [MAX_SIZE:0] accumilator;
  logic [MAX_SIZE:0] divided;
  assign Q = divided[(N-1):0];

  always @(posedge clk) begin
    if(reset) begin
      accumilator <= 0;
      oldDin <= 0;
      for (int i = 0; i < power; i=i+1)
        s[i] <= 0;
      divided <= 0;
    end else begin
      accumilator <= accumilator + Din - oldDin;

      s = {s[1:(power-1)], Din};
      oldDin <= s[0];
      divided <=  accumilator / 'd8;
    end
  end
```

```
endmodule
```

clk_divider.sv

```
// File info:
// - Original file
// - Produces a clock frequency that is the input frequency / divisor

module clk_divider (
   input  logic clk,          // Input clock
   input  logic reset,        // Active high reset
   input  logic [31:0] divisor,  // How much to divide the clock by
   output logic slow_clk       // Output slower clock
);

   logic [31:0] counter;

   always_ff @(posedge clk) begin
      if (reset) begin
         counter <= 0;
         slow_clk <= 0;
      end else begin
         counter <= counter + 1;

         if (counter >= divisor) begin
            counter <= 9'd0;
            slow_clk <= ~slow_clk;
         end
      end
   end
endmodule
```

constraints.xdc

```
## This file is a general .xdc for the Basys3 rev B board
## To use it in a project:
## - uncomment the lines corresponding to used pins
## - rename the used ports (in each line, after get_ports) according to the top level signal names
in the project
## Xilinx part number XC7A35T-1CPG236C (from Reference Manual)
## Xilinx part number xc7a35tcpg236-1 (from Xilinx Vivado)

## Clock signal
```

```
set_property -dict { PACKAGE_PIN W5   IOSTANDARD LVCMOS33 } [get_ports clk]
create_clock -add -name sys_clk_pin -period 100.00 -waveform {0 5} [get_ports clk]


## Switches
set_property -dict { PACKAGE_PIN V17   IOSTANDARD LVCMOS33 } [get_ports
{switches_inputs[0]}]
set_property -dict { PACKAGE_PIN V16   IOSTANDARD LVCMOS33 } [get_ports
{switches_inputs[1]}]
set_property -dict { PACKAGE_PIN W16   IOSTANDARD LVCMOS33 } [get_ports
{switches_inputs[2]}]
set_property -dict { PACKAGE_PIN W17   IOSTANDARD LVCMOS33 } [get_ports
{switches_inputs[3]}]
set_property -dict { PACKAGE_PIN W15   IOSTANDARD LVCMOS33 } [get_ports
{switches_inputs[4]}]
set_property -dict { PACKAGE_PIN V15   IOSTANDARD LVCMOS33 } [get_ports
{switches_inputs[5]}]
set_property -dict { PACKAGE_PIN W14   IOSTANDARD LVCMOS33 } [get_ports
{switches_inputs[6]}]
set_property -dict { PACKAGE_PIN W13   IOSTANDARD LVCMOS33 } [get_ports
{switches_inputs[7]}]
set_property -dict { PACKAGE_PIN V2    IOSTANDARD LVCMOS33 } [get_ports
{switches_inputs[8]}]
set_property -dict { PACKAGE_PIN T3    IOSTANDARD LVCMOS33 } [get_ports
{switches_inputs[9]}]
set_property -dict { PACKAGE_PIN T2    IOSTANDARD LVCMOS33 } [get_ports
{switches_inputs[10]}]
set_property -dict { PACKAGE_PIN R3    IOSTANDARD LVCMOS33 } [get_ports
{switches_inputs[11]}]
set_property -dict { PACKAGE_PIN W2    IOSTANDARD LVCMOS33 } [get_ports
{switches_inputs[12]}]
set_property -dict { PACKAGE_PIN U1    IOSTANDARD LVCMOS33 } [get_ports
{switches_inputs[13]}]
set_property -dict { PACKAGE_PIN T1    IOSTANDARD LVCMOS33 } [get_ports
{switches_inputs[14]}]
set_property -dict { PACKAGE_PIN R2    IOSTANDARD LVCMOS33 } [get_ports
{switches_inputs[15]}]


## LEDs
set_property -dict { PACKAGE_PIN U16   IOSTANDARD LVCMOS33 } [get_ports {led[0]}]
set_property -dict { PACKAGE_PIN E19   IOSTANDARD LVCMOS33 } [get_ports {led[1]}]
set_property -dict { PACKAGE_PIN U19   IOSTANDARD LVCMOS33 } [get_ports {led[2]}]
set_property -dict { PACKAGE_PIN V19   IOSTANDARD LVCMOS33 } [get_ports {led[3]}]
set_property -dict { PACKAGE_PIN W18   IOSTANDARD LVCMOS33 } [get_ports {led[4]}]
set_property -dict { PACKAGE_PIN U15   IOSTANDARD LVCMOS33 } [get_ports {led[5]}]
set_property -dict { PACKAGE_PIN U14   IOSTANDARD LVCMOS33 } [get_ports {led[6]}]
set_property -dict { PACKAGE_PIN V14   IOSTANDARD LVCMOS33 } [get_ports {led[7]}]
```

```
set_property -dict { PACKAGE_PIN V13   IOSTANDARD LVCMOS33 } [get_ports {led[8]}]
set_property -dict { PACKAGE_PIN V3    IOSTANDARD LVCMOS33 } [get_ports {led[9]}]
set_property -dict { PACKAGE_PIN W3    IOSTANDARD LVCMOS33 } [get_ports {led[10]}]
set_property -dict { PACKAGE_PIN U3    IOSTANDARD LVCMOS33 } [get_ports {led[11]}]
set_property -dict { PACKAGE_PIN P3    IOSTANDARD LVCMOS33 } [get_ports {led[12]}]
set_property -dict { PACKAGE_PIN N3    IOSTANDARD LVCMOS33 } [get_ports {led[13]}]
set_property -dict { PACKAGE_PIN P1    IOSTANDARD LVCMOS33 } [get_ports {led[14]}]
set_property -dict { PACKAGE_PIN L1    IOSTANDARD LVCMOS33 } [get_ports {led[15]}]


##7 Segment Display
set_property -dict { PACKAGE_PIN W7   IOSTANDARD LVCMOS33 } [get_ports {CA}]
set_property -dict { PACKAGE_PIN W6   IOSTANDARD LVCMOS33 } [get_ports {CB}]
set_property -dict { PACKAGE_PIN U8   IOSTANDARD LVCMOS33 } [get_ports {CC}]
set_property -dict { PACKAGE_PIN V8   IOSTANDARD LVCMOS33 } [get_ports {CD}]
set_property -dict { PACKAGE_PIN U5   IOSTANDARD LVCMOS33 } [get_ports {CE}]
set_property -dict { PACKAGE_PIN V5   IOSTANDARD LVCMOS33 } [get_ports {CF}]
set_property -dict { PACKAGE_PIN U7   IOSTANDARD LVCMOS33 } [get_ports {CG}]

set_property -dict { PACKAGE_PIN V7   IOSTANDARD LVCMOS33 } [get_ports DP]

set_property -dict { PACKAGE_PIN U2   IOSTANDARD LVCMOS33 } [get_ports {AN1}]
set_property -dict { PACKAGE_PIN U4   IOSTANDARD LVCMOS33 } [get_ports {AN2}]
set_property -dict { PACKAGE_PIN V4   IOSTANDARD LVCMOS33 } [get_ports {AN3}]
set_property -dict { PACKAGE_PIN W4   IOSTANDARD LVCMOS33 } [get_ports {AN4}]


##Buttons
set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports reset]
# set_property -dict { PACKAGE_PIN T18   IOSTANDARD LVCMOS33 } [get_ports btnU]
# set_property -dict { PACKAGE_PIN W19   IOSTANDARD LVCMOS33 } [get_ports btnL]
# set_property -dict { PACKAGE_PIN T17   IOSTANDARD LVCMOS33 } [get_ports btnR]
# set_property -dict { PACKAGE_PIN U17   IOSTANDARD LVCMOS33 } [get_ports btnD]


##Pmod Header JA
# set_property -dict { PACKAGE_PIN J1   IOSTANDARD LVCMOS33 } [get_ports {ultrasonic}];#Sch
name = JA1
# set_property -dict { PACKAGE_PIN L2   IOSTANDARD LVCMOS33 } [get_ports
{elbow_servo}];#Sch name = JA2
# set_property -dict { PACKAGE_PIN J2   IOSTANDARD LVCMOS33 } [get_ports {base_servo}];#Sch
name = JA3
# set_property -dict { PACKAGE_PIN G2   IOSTANDARD LVCMOS33 } [get_ports {JA[3]}];#Sch name
= JA4
# set_property -dict { PACKAGE_PIN H1   IOSTANDARD LVCMOS33 } [get_ports {JA[4]}];#Sch name
= JA7
# set_property -dict { PACKAGE_PIN K2   IOSTANDARD LVCMOS33 } [get_ports {JA[5]}];#Sch name
= JA8
```

```
# set_property -dict { PACKAGE_PIN H2   IOSTANDARD LVCMOS33 } [get_ports {echo}];#Sch
name = JA9
# set_property -dict { PACKAGE_PIN G3   IOSTANDARD LVCMOS33 } [get_ports {trig}];#Sch name
= JA10

##Pmod Header JB
# set_property -dict { PACKAGE_PIN A14   IOSTANDARD LVCMOS33 } [get_ports
{shoulder_servo}];#Sch name = JB1
#set_property -dict { PACKAGE_PIN A16   IOSTANDARD LVCMOS33 } [get_ports {JB[1]}];#Sch
name = JB2
#set_property -dict { PACKAGE_PIN B15   IOSTANDARD LVCMOS33 } [get_ports {JB[2]}];#Sch
name = JB3
#set_property -dict { PACKAGE_PIN B16   IOSTANDARD LVCMOS33 } [get_ports {JB[3]}];#Sch
name = JB4
#set_property -dict { PACKAGE_PIN A15   IOSTANDARD LVCMOS33 } [get_ports {JB[4]}];#Sch
name = JB7
#set_property -dict { PACKAGE_PIN A17   IOSTANDARD LVCMOS33 } [get_ports {JB[5]}];#Sch
name = JB8
#set_property -dict { PACKAGE_PIN C15   IOSTANDARD LVCMOS33 } [get_ports {JB[6]}];#Sch
name = JB9
#set_property -dict { PACKAGE_PIN C16   IOSTANDARD LVCMOS33 } [get_ports {JB[7]}];#Sch
name = JB10

##Pmod Header JC
set_property SEVERITY {Warning} [get_drc_checks NSTD-1]
set_property SEVERITY {Warning} [get_drc_checks UCIO-1]
set_property -dict { PACKAGE_PIN K17   IOSTANDARD LVCMOS33 } [get_ports
{elbow_servo}];#Sch name = JC1
set_property -dict { PACKAGE_PIN M18   IOSTANDARD LVCMOS33 } [get_ports
{shoulder_servo}];#Sch name = JC2
set_property -dict { PACKAGE_PIN N17   IOSTANDARD LVCMOS33 } [get_ports {ultrasonic}];#Sch
name = JC3
#set_property -dict { PACKAGE_PIN P18   IOSTANDARD LVCMOS33 } [get_ports {JC[3]}];#Sch
name = JC4
#set_property -dict { PACKAGE_PIN L17   IOSTANDARD LVCMOS33 } [get_ports {JC[4]}];#Sch
name = JC7
#set_property -dict { PACKAGE_PIN M19   IOSTANDARD LVCMOS33 } [get_ports {JC[5]}];#Sch
name = JC8
#set_property -dict { PACKAGE_PIN P17   IOSTANDARD LVCMOS33 } [get_ports {JC[6]}];#Sch
name = JC9
#set_property -dict { PACKAGE_PIN R18   IOSTANDARD LVCMOS33 } [get_ports {JC[7]}];#Sch
name = JC10

##Pmod Header JXADC
#set_property -dict { PACKAGE_PIN J3   IOSTANDARD LVCMOS33 } [get_ports {JXADC[0]}];#Sch
name = XA1_P
#set_property -dict { PACKAGE_PIN L3   IOSTANDARD LVCMOS33 } [get_ports {JXADC[1]}];#Sch
name = XA2_P
```

```
#set_property -dict { PACKAGE_PIN M2   IOSTANDARD LVCMOS33 } [get_ports {JXADC[2]}];#Sch
name = XA3_P
#set_property -dict { PACKAGE_PIN N2   IOSTANDARD LVCMOS33 } [get_ports {JXADC[3]}];#Sch
name = XA4_P
#set_property -dict { PACKAGE_PIN K3   IOSTANDARD LVCMOS33 } [get_ports {JXADC[4]}];#Sch
name = XA1_N
#set_property -dict { PACKAGE_PIN M3   IOSTANDARD LVCMOS33 } [get_ports {JXADC[5]}];#Sch
name = XA2_N
#set_property -dict { PACKAGE_PIN M1   IOSTANDARD LVCMOS33 } [get_ports {JXADC[6]}];#Sch
name = XA3_N
#set_property -dict { PACKAGE_PIN N1   IOSTANDARD LVCMOS33 } [get_ports {JXADC[7]}];#Sch
name = XA4_N


##VGA Connector
#set_property -dict { PACKAGE_PIN G19   IOSTANDARD LVCMOS33 } [get_ports {vgaRed[0]}]
#set_property -dict { PACKAGE_PIN H19   IOSTANDARD LVCMOS33 } [get_ports {vgaRed[1]}]
#set_property -dict { PACKAGE_PIN J19   IOSTANDARD LVCMOS33 } [get_ports {vgaRed[2]}]
#set_property -dict { PACKAGE_PIN N19   IOSTANDARD LVCMOS33 } [get_ports {vgaRed[3]}]
#set_property -dict { PACKAGE_PIN N18   IOSTANDARD LVCMOS33 } [get_ports {vgaBlue[0]}]
#set_property -dict { PACKAGE_PIN L18   IOSTANDARD LVCMOS33 } [get_ports {vgaBlue[1]}]
#set_property -dict { PACKAGE_PIN K18   IOSTANDARD LVCMOS33 } [get_ports {vgaBlue[2]}]
#set_property -dict { PACKAGE_PIN J18   IOSTANDARD LVCMOS33 } [get_ports {vgaBlue[3]}]
#set_property -dict { PACKAGE_PIN J17   IOSTANDARD LVCMOS33 } [get_ports {vgaGreen[0]}]
#set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports {vgaGreen[1]}]
#set_property -dict { PACKAGE_PIN G17   IOSTANDARD LVCMOS33 } [get_ports {vgaGreen[2]}]
#set_property -dict { PACKAGE_PIN D17   IOSTANDARD LVCMOS33 } [get_ports {vgaGreen[3]}]
#set_property -dict { PACKAGE_PIN P19   IOSTANDARD LVCMOS33 } [get_ports Hsync]
#set_property -dict { PACKAGE_PIN R19   IOSTANDARD LVCMOS33 } [get_ports Vsync]


##USB-RS232 Interface
set_property -dict { PACKAGE_PIN B18   IOSTANDARD LVCMOS33 } [get_ports rx]
set_property -dict { PACKAGE_PIN A18   IOSTANDARD LVCMOS33 } [get_ports tx]


##USB HID (PS/2)
# set_property -dict { PACKAGE_PIN C17   IOSTANDARD LVCMOS33   PULLUP true } [get_ports
PS2Clk]
# set_property -dict { PACKAGE_PIN B17   IOSTANDARD LVCMOS33   PULLUP true } [get_ports
PS2Data]


##Quad SPI Flash
##Note that CCLK_0 cannot be placed in 7 series devices. You can access it using the
##STARTUPE2 primitive.
#set_property -dict { PACKAGE_PIN D18   IOSTANDARD LVCMOS33 } [get_ports {QspiDB[0]}]
#set_property -dict { PACKAGE_PIN D19   IOSTANDARD LVCMOS33 } [get_ports {QspiDB[1]}]
```

```
#set_property -dict { PACKAGE_PIN G18   IOSTANDARD LVCMOS33 } [get_ports {QspiDB[2]}]
#set_property -dict { PACKAGE_PIN F18   IOSTANDARD LVCMOS33 } [get_ports {QspiDB[3]}]
#set_property -dict { PACKAGE_PIN K19   IOSTANDARD LVCMOS33 } [get_ports QspiCSn]


## Configuration options, can be used for all designs
set_property CONFIG_VOLTAGE 3.3 [current_design]
set_property CFGBVS VCCO [current_design]

## SPI configuration mode options for QSPI boot, can be used for all designs
set_property BITSTREAM.GENERAL.COMPRESS TRUE [current_design]
set_property BITSTREAM.CONFIG.CONFIGRATE 33 [current_design]
set_property CONFIG_MODE SPIx4 [current_design]
```

display.sv

```
// File info:
// - Based on parts of given SystemVerilog from labs
// - Combined files together
// - Displays FSM state and desired x- and y-coordinates

module display (
   input  logic      clk,   // 10 MHz clock
   input  logic      reset,  // Active high reset
   input  logic [1:0] state,  // FSM state

   input  logic [3:0] x,  // Desired x-coordinate
   input  logic [3:0] y,  // Desired y-coordinate

   // Display cathodes
   output logic      CA,
   output logic      CB,
   output logic      CC,
   output logic      CD,
   output logic      CE,
   output logic      CF,
   output logic      CG,
   output logic      DP,

   // Display anodes
   output logic       AN1, AN2, AN3, AN4
);

   logic [6:0] digit1;
   logic [6:0] digit2;
   logic [6:0] digit3;
   logic [6:0] digit4;
```

```
logic [3:0] d, q;
logic [13:0] count;
logic [3:0] digit_select;
logic [3:0] an_outputs;
logic [6:0] digit;

// Which digit is getting shown (anode active low)
assign digit_select = q;
assign an_outputs = ~q;

// Never use the decimal point
assign DP = 1;

always @(posedge clk) begin
  if ( reset ) begin
    digit1 <= 7'b000_0000;
    digit2 <= 7'b000_0000;
    digit3 <= 7'b000_0000;
    digit4 <= 7'b000_0000;
  end else begin

    // This case statement selects what digits to display
    case ( state )
      //      A
      //    F   B
      //      G
      //    E   C
      //      D  DP
      // abc_defg

      // 00: Selection of ultrasonic or keyboard (default)
      // 01: Ultrasonic
      // 10: Keyboard

      default : begin
        digit1 <= 7'b000_0001;
        digit2 <= 7'b000_0001;
      end

      2'b01 : begin
        digit1 <= 7'b001_1100;
        digit2 <= 7'b011_0000;
      end

      2'b10 : begin
        digit1 <= 7'b001_1111;
        digit2 <= 7'b011_1101;
      end
```

```
endcase


case (x)             // 6543210
   'd0: digit3 <= 7'b1111110; // 0     A-6
   'd1: digit3 <= 7'b0110000; // 1   F-1    B-5
   'd2: digit3 <= 7'b1101101; // 2     G-0
   'd3: digit3 <= 7'b1111001; // 3   E-2    C-4
   'd4: digit3 <= 7'b0110011; // 4     D-3     DP
   'd5: digit3 <= 7'b1011011; // 5
   'd6: digit3 <= 7'b1011111; // 6
   'd7: digit3 <= 7'b1110000; // 7
   'd8: digit3 <= 7'b1111111; // 8
   'd9: digit3 <= 7'b1111011; // 9
   default: digit3 <= 7'b0000001;
endcase


case (y)             // 6543210
   'd0: digit4 <= 7'b1111110; // 0     A-6
   'd1: digit4 <= 7'b0110000; // 1   F-1    B-5
   'd2: digit4 <= 7'b1101101; // 2     G-0
   'd3: digit4 <= 7'b1111001; // 3   E-2    C-4
   'd4: digit4 <= 7'b0110011; // 4     D-3     DP
   'd5: digit4 <= 7'b1011011; // 5
   'd6: digit4 <= 7'b1011111; // 6
   'd7: digit4 <= 7'b1110000; // 7
   'd8: digit4 <= 7'b1111111; // 8
   'd9: digit4 <= 7'b1111011; // 9
   default: digit4 <= 7'b0000001;
endcase


// This case statement multiplexes the output digit
case ( digit_select )

   default : begin
      digit <= digit1;
   end

   4'b0010 : begin
      digit <= digit2;
   end

   4'b0100 : begin
      digit <= digit3;
   end
```

```
        4'b1000 : begin
          digit <= digit4;
        end

      endcase
    end
end


// Counter
always_ff @(posedge clk) begin
  if (reset) begin
    count <= 14'b0;
  end else begin
    count <= count + 1;
  end
end

// There is one high bit in q (one hot encoding), thus to make
//    a multiplexer you can use a circular shift to pass the high
//    bit around to each position
always_ff @(posedge clk) begin
  if (reset) begin
    // Reset state values for q
    q <= 4'b1111;
  end else if (count == 14'b0) begin
    // Propagate signals through the DFF
    if (q[0] && q[1]) begin
      q <= 4'b1000;
    end else begin
      q <= d;
    end
  end
end

// Connect the DFFs into a chain/loop
assign d[0] = q[3];
assign d[1] = q[0];
assign d[2] = q[1];
assign d[3] = q[2];

// All the outputs
assign CA = ~digit[6];
assign CB = ~digit[5];
assign CC = ~digit[4];
assign CD = ~digit[3];
assign CE = ~digit[2];
```

```
   assign CF = ~digit[1];
   assign CG = ~digit[0];
   assign AN1 = an_outputs[3];
   assign AN2 = an_outputs[2];
   assign AN3 = an_outputs[1];
   assign AN4 = an_outputs[0];

endmodule
```

divider.sv

```
// File info:
// - Original file

// Divider works by subtracting divisor from dividend until the
//    divisor is bigger than the dividen, then returning how
//    many times that occured as the quotient
module divider (
   input  logic clk,   // 10 MHz clock
   input  logic reset, // Active high reset
   input  logic [31:0] x,  // Dividend
   input  logic [31:0] y,  // Divisor
   output logic [31:0] a,  // Quotient
   output logic [31:0] b   // Remainder
);

   logic [1:0] state;
   // 00: Idle (default)
   // 01: Loading
   // 10: Dividing

   logic [31:0] dividend;
   logic [31:0] divide_counts;

   always @(posedge clk) begin

      if ( reset ) begin
         state <= 0;
         dividend <= 0;
         divide_counts <= 0;
         a <= 0;
         b <= 0;
      end else begin
         case ( state )

            default : begin
               if ( y != 0 ) begin // No divide by 0
```

```
            state <= 2'b01;
          end
        end

        2'b01 : begin
          dividend <= x;
          state <= 2'b10;
          divide_counts <= 0;
        end

        2'b10 : begin
          if ( dividend >= y ) begin
            dividend <= dividend - y;
            divide_counts <= divide_counts + 1;
          end else begin
            a <= divide_counts;
            b <= x - ( divide_counts * y );
            state <= 2'b00;
          end
        end

      endcase
    end

  end

endmodule
```

fsm.sv

```
// File info
//  - Original
//  - Decides states of the FSM

module fsm (
  input logic clk,    // 10 MHz clock
  input logic reset,  // Active high reset

  input logic right,  // Switch on the right side of the board
  input logic left,   // Switch on the left side of the board

  output logic [1:0] state,        // FSM state
  output logic keyboardControlled,   // If the arm is keyboard controlled
  output logic ultrasonicControlled  // If the arm is ultrasonic controlled
);
  // Possible states:
  // 00: Selection of ultrasonic or keyboard (default)
```

```
  // 01: Ultrasonic
  // 10: Keyboard

  always @(posedge clk) begin
    if ( reset ) begin
      state <= 0;
    end else begin
      case(state)

        default : begin
          if ( right ) begin
            state <= 2'b01;
          end else if ( left ) begin
            state <= 2'b10;
          end
        end

        2'b01 : begin
          if ( left && ~right ) begin
            state <= 2'b10;
          end else if (~left && ~right ) begin
            state <= 2'b00;
          end
        end

        2'b10 : begin
          if ( right && ~left ) begin
            state <= 2'b01;
          end else if (~left && ~right ) begin
            state <= 2'b00;
          end
        end

      endcase
    end
  end

  assign keyboardControlled = ( state == 2'b10);
  assign ultrasonicControlled = ( state == 2'b01);
endmodule
```

fsm_controller.sv

```
// File info
// - Original
// - Enacts the state-based logic of the finite state machine
```

```
module fsm_controller (
  input logic clk,        // 10 MHz clock
  input logic reset,      // Active high reset
  input logic [1:0] state, // FSM State

  input logic keyboardControlled, // If the arm is keyboard controlled
  input logic [7:0] keyboard_x,   // Desired arm x-coordinate if keyboard controlled
  input logic [7:0] keyboard_y,   // Desired arm y-coordinate if keyboard controlled
  input logic [7:0] data,         // Recieved UART data

  input logic ultrasonicControlled, // If the arm is ultrasonic controlled
  input logic [7:0] ultrasonic_x,   // Desired arm x-coordinate if ultrasonic controlled
  input logic [7:0] ultrasonic_y,   // Desired arm y-coordinate if ultrasonic controlled

  output logic [15:0] led,    // LEDs
  output logic shoulder_servo, // PWM for the shoulder servo
  output logic elbow_servo,    // PWM for the elbow servo
  output logic tx,             // Transmitting UART data

  // Display cathodes
  output logic CA,
  output logic CB,
  output logic CC,
  output logic CD,
  output logic CE,
  output logic CF,
  output logic CG,
  output logic DP,

  // Display anodes
  output logic AN1,
  output logic AN2,
  output logic AN3,
  output logic AN4
);

  logic pwm_en;
  logic [7:0] inverse_kinematics_x;
  logic [7:0] inverse_kinematics_y;

  // Clock divider for the dancing lights
  logic slow_clk;
  clk_divider cd (
    .clk(clk),
    .reset(reset),
    .divisor('d100000000),
    .slow_clk(slow_clk)
  );
```

```
always @( posedge clk ) begin
  if ( reset ) begin
    // Show a distinct bit pattern for reset
    led <= 16'b1111_0000_1111_0000;

    inverse_kinematics_x <= 2;
    inverse_kinematics_y <= 2;
  end else begin
    if ( keyboardControlled ) begin
      inverse_kinematics_x <= keyboard_x;
      inverse_kinematics_y <= keyboard_y;

      // Change LEDs if in keyboard mode
      led[15] <= pwm_en;
      led[14:12] <= inverse_kinematics_x[2:0];
      led[11:9] <= inverse_kinematics_y[2:0];
      led[7:0] <= data;

    end else if ( ultrasonicControlled ) begin
      inverse_kinematics_x <= ultrasonic_x[7:0];
      inverse_kinematics_y <= ultrasonic_y;

      // Change LEDs if in ultrasonic mode
      led[14] <= 1;
      led[7:0] <= ultrasonic_x[7:0];
    end else begin
      // Show a distinct bit pattern for default state
      // - Helped worlds for debugging
      if ( slow_clk )
        led <= 16'b0101_0101_0101_0101;
      else
        led <= 16'b1010_1010_1010_1010;

      inverse_kinematics_x <= 2;
      inverse_kinematics_y <= 2;
    end
  end
end

// Solves the problem of servos burning out by detecting the last
//   time they were assigned a different value, and only enabling
//   the servos for a short time after the change. Else, rely on
//   the internal friction of the servo to keep it in place
pwm_enable #(24, 200000 * 700) pwme (
  .data(data),
  .clk(clk),
  .reset(reset),
```

```
    .en(pwm_en)
);

// Figure out the needed angle for the servos based on requested
//    (x,y) values. Implemented as a 2D lookup table.
logic [23:0] shoulder_angle;
logic [23:0] elbow_angle;
inverse_kinematics ik (
    .clk(clk),
    .reset(reset),
    .x(inverse_kinematics_x),
    .y(inverse_kinematics_y),
    .shoulder_angle(shoulder_angle),
    .elbow_angle(elbow_angle)
);

// PWMs for the output servos, always enable when ultrasonic controlled
pwm shoulderPWM (clk, pwm_en | ultrasonicControlled, shoulder_angle, shoulder_servo);
pwm elbowPWM    (clk, pwm_en | ultrasonicControlled, elbow_angle, elbow_servo);

// Always transmit data back to my laptop for the simulation
transmitter t (
    .clk(clk),
    .reset(reset),
    .transmit(1),
    .data({inverse_kinematics_x[3:0], inverse_kinematics_y[3:0]}),
    .TxD(tx)
);

// Display to the seven segment display
display d (
    .clk(clk),
    .reset(reset),
    .state(state),

    .x(inverse_kinematics_x),
    .y(inverse_kinematics_y),

    .CA(CA),
    .CB(CB),
    .CC(CC),
    .CD(CD),
    .CE(CE),
    .CF(CF),
    .CG(CG),
    .DP(DP),

    .AN1(AN1),
```

```
    .AN2(AN2),
    .AN3(AN3),
    .AN4(AN4)
  );
endmodule
```

inverse_kinematics.sv

```
// File info:
// - Python generated!
// - Python script is my own script
// - Lookup servo angles based on desired x- and y-coordinates

module inverse_kinematics (
  input  logic clk,              // 10 MHz clock
  input  logic reset,             // Active high reset
  input  logic [7:0]  x,          // Desired x-coordinate
  input  logic [7:0]  y,          // Desired y-coordinate
  output logic [23:0] shoulder_angle,  // Output shoulder angle (Calibrated value, not degrees or
radians)
  output logic [23:0] elbow_angle     // Output elbow angle (Calibrated value, not degrees or
radians)
);

always @(posedge clk) begin
  if (reset) begin
    elbow_angle <= 74692;
    shoulder_angle <= 97346;
  end else begin
    case({x, y})
      default : begin  // x=1 y=1
        // elbow = 121.4
        // shoulder = 152.7
        elbow_angle <= 74692;
        shoulder_angle <= 97346;
      end
      16'b00000001_00000001 : begin  // x=1 y=1
        // elbow = 121.4
        // shoulder = 152.7
        elbow_angle <= 74692;
        shoulder_angle <= 97346;
      end
      16'b00000001_00000010 : begin  // x=1 y=2
        // elbow = 131.6
        // shoulder = 136.2
        elbow_angle <= 86605;
        shoulder_angle <= 89988;
```

```
        end
        16'b00000001_00000011 : begin  // x=1 y=3
          // elbow = 129.8
          // shoulder = 116.4
          elbow_angle <= 100942;
          shoulder_angle <= 91285;
        end
        16'b00000001_00000100 : begin  // x=1 y=4
          // elbow = 122.6
          // shoulder = 93.2
          elbow_angle <= 117699;
          shoulder_angle <= 96487;
        end
        16'b00000010_00000001 : begin  // x=2 y=1
          // elbow = 94.7
          // shoulder = 136.2
          elbow_angle <= 86605;
          shoulder_angle <= 116616;
        end
        16'b00000010_00000010 : begin  // x=2 y=2
          // elbow = 106.9
          // shoulder = 123.7
          elbow_angle <= 95625;
          shoulder_angle <= 107812;
        end
        16'b00000010_00000011 : begin  // x=2 y=3
          // elbow = 109.4
          // shoulder = 106.1
          elbow_angle <= 108352;
          shoulder_angle <= 106007;
        end
        16'b00000010_00000100 : begin  // x=2 y=4
          // elbow = 105.2
          // shoulder = 83.6
          elbow_angle <= 124607;
          shoulder_angle <= 108989;
        end
        16'b00000011_00000001 : begin  // x=3 y=1
          // elbow = 76.6
          // shoulder = 116.4
          elbow_angle <= 100942;
          shoulder_angle <= 129656;
        end
        16'b00000011_00000010 : begin  // x=3 y=2
          // elbow = 86.8
          // shoulder = 106.1
          elbow_angle <= 108352;
          shoulder_angle <= 122344;
```

```
        end
        16'b00000011_00000011 : begin  // x=3 y=3
          // elbow = 90.0
          // shoulder = 90.0
          elbow_angle <= 120000;
          shoulder_angle <= 120000;
        end
        16'b00000011_00000100 : begin  // x=3 y=4
          // elbow = 86.7
          // shoulder = 67.1
          elbow_angle <= 136528;
          shoulder_angle <= 122392;
        end
        16'b00000100_00000001 : begin  // x=4 y=1
          // elbow = 60.6
          // shoulder = 93.2
          elbow_angle <= 117699;
          shoulder_angle <= 141212;
        end
        16'b00000100_00000010 : begin  // x=4 y=2
          // elbow = 68.4
          // shoulder = 83.6
          elbow_angle <= 124607;
          shoulder_angle <= 135617;
        end
        16'b00000100_00000011 : begin  // x=4 y=3
          // elbow = 70.4
          // shoulder = 67.1
          elbow_angle <= 136528;
          shoulder_angle <= 134135;
        end
        16'b00000100_00000100 : begin  // x=4 y=4
          // elbow = 64.5
          // shoulder = 38.9
          elbow_angle <= 156874;
          shoulder_angle <= 138437;
        end
      endcase
    end
end

endmodule
```

keyboardControl.sv

```
// File info:
// - Original file
```

```
// - Change the desired x- and y-coordinates for the arm based on received rx commands

module keyboardControl #(parameter X_MAX=4, Y_MAX=4) (
  input  logic clk,       // 10 MHz clock
  input  logic reset,     // Active high reset
  input  logic rx,        // UART recieve line

  output logic [7:0] data,  // RX command recieved
  output logic [7:0] x,     // Desired x-coordinate
  output logic [7:0] y      // Desired y-coordinate
);

  logic [7:0] data_in;
  logic [7:0] data_old;
  logic [7:0] internal_x;
  logic [7:0] internal_y;

  // Ensure output (x,y) range is [1,4] instead of [0,3]
  assign x = internal_x + 1;
  assign y = internal_y + 1;

  // Translate UART RX into 8 bits:
  //  - Remove start bit, parity, stop bits
  //  - Parallelize
  receiver r ( .clk(clk), .reset(reset), .rx(rx), .data(data_in) );

  // Fifo Queue to store the keyboard value
  logic fifoEmpty;
  logic fifoFull;
  syncFIFO #( 2, 8 ) s (
    .reset(reset),
    .w_clk(clk),
    .r_clk(clk),
    .w_en(~fifoFull),  // Write when not full
    .r_en('b1),  // Always read
    .din(data_in),

    .dout(data),
    .empty(fifoEmpty),
    .full(fifoFull)
  );

  // Store the old data like in the button debouncer so can see
  //    if the data has changed later in the controller
  always_ff @(posedge clk) begin
    data_old <= data;
  end
```

```
  // Very simple decoder to increase/decrease target (x,y)
  //    of the servo arm
  // - w: y++
  // - a: x++
  // - s: y--
  // - d: x--
  always @(posedge clk) begin
    if (reset) begin
      internal_x <= 2;
      internal_y <= 3;
    end else if ( data_old ^ data ) begin
      case(data)
        8'b0111_0111 :  begin  // W
          internal_y <= ( (internal_y + 1) % Y_MAX );
        end

        8'b0110_0001 :  begin  // A
          internal_x <= ( (internal_x + 1) % X_MAX );
        end

        8'b0111_0011 :  begin  // S
          internal_y <= ( (internal_y - 1) % Y_MAX );
        end

        8'b0110_0100 :  begin  // D
          internal_x <= ( (internal_x - 1) % X_MAX );
        end
      endcase
    end
  end
endmodule
```

pwm.sv

```
// File info:
// - Math and SV based on https://www.youtube.com/watch?v=zNln9hJ5J78
// - Math based on http://www.ee.ic.ac.uk/pcheung/teaching/DE1_EE/stores/sg90_datasheet.pdf
// - Math and SV based on https://www.youtube.com/watch?v=JpzeoQI2MII
// - Output a PWM signal to control a servo

// Math:
// 1. Clock Frequency = 10 MHz
// 2. Servo Period = 20 ms
// 3. Clock Edges = 1 / Clock Frequency = 100 ns
// 4. Counter Length = Servo Period / Clock Edges = 200 000

module pwm (
```

```
  input  logic clk,         // 10 MHz clock
  input  logic en,          // Enable the PWM signal
  input  logic [23:0] angle,  // Calibrated value, not degrees or radians

  output logic servo        // Output PWM signal
);
  localparam SERVO_PERIOD = 'd199999;

  logic [20:0] counter;

  always @(posedge clk) begin
    // Counter that resets every servo period
    if(counter < SERVO_PERIOD) begin
      counter <= counter + 1;
    end else begin
      counter <= 0;
    end

    // Generate duty cycle
    if( ( counter < angle ) & en ) begin
      // Start of the signal high
      servo <= 1;
    end else begin
      servo <= 0;
    end
  end
endmodule
```

pwm_enable.sv

```
// File info:
// - Original file
// - Enables the servo PWM for a short length of time on data change

module pwm_enable#(parameter N=8, CLK_CYCLES=10)(
  input logic [N-1:0] data,  // Input data
  input logic clk,         // 10 MHz clock
  input logic reset,       // Active high reset
  output logic en          // Servo enable
);
  logic [N-1:0] old_data;
  logic data_different;
  logic [255:0] current_clk_cycles;

  always_ff @(posedge clk) begin
    if (reset) begin
      old_data <= 'b0;
```

```
        data_different <= 0;
        current_clk_cycles <= 'b0;
      end else begin
        // When data changes, set en high for CLK_CYCLES
        old_data <= data;
        data_different <= data_different | (old_data ^ data);
        en <= ( current_clk_cycles <= CLK_CYCLES );

        if (data_different & ~(old_data ^ data)) begin
          current_clk_cycles <= current_clk_cycles + 1;
        end else begin
          current_clk_cycles <= 0;
        end
      end
    end

endmodule
```

pwm_measure.sv

```
// File info:
// - Original
// - Converts the ultrasonic PWM into a distance int

module pwm_measure #(parameter int DIVISION_AMOUNT=1664)(
    input logic clk,     // 10 MHz clock
    input logic reset,   // Active high reset
    input logic pwm_in,  // Ultrasonic PWM input

    output logic [7:0] distance  // Translated distance
);

    logic [31:0] count;
    logic [31:0] raw_distance;
    logic old_pwm_in;

    // Division!! I know because I'm dividing by a constant I could use
    //    the method to divide shown in our labs that uses approximation
    //    and bit shifting, but I wanted to try making a dividor
    logic [31:0] divided_distance;
    divider d (
      .clk(clk),
      .reset(reset),
      .x(raw_distance),
      .y(DIVISION_AMOUNT),
      .a(divided_distance)
    );
```

```
  // I found the ultrasonic sensor to give off random noisy values,
  //    both in FPGA's and using the same ultrasonic sensor with the
  //    same constants and same communication method with an Arduino
  //    for capstone. Thus, I needed to use an averager to smooth it out.
  logic [31:0] averaged_distance;
  averager a (
     .clk(clk),
     .reset(reset),
     .EN(1),
     .Din(divided_distance),
     .Q(averaged_distance)
  );

  assign distance = averaged_distance + 1;

  // Store old value so can detect changes in the signal
  always_ff @(posedge clk) begin
    old_pwm_in <= pwm_in;
  end

  always @(posedge clk) begin
    if ( reset ) begin
       count <= 0;
       raw_distance <= 0;
    end else begin
       if ( pwm_in == 0 && old_pwm_in == 1) begin
          // If signal goes from high to low, assume done measuring
          raw_distance <= count;
       end else if ( pwm_in == 1 && old_pwm_in == 0) begin
          // If signal goes low to high, reset the count so you can start measuring
          count <= 0;
       end else if ( pwm_in == 1 && old_pwm_in == 1) begin
          // If signal is high and stays high, increase count
          count <= count + 1;
       end
    end
  end

endmodule
```

receiver.sv

```
// File info:
// - Verilog copied from: https://www.instructables.com/UART-Communication-on-Basys-3-FPGA-Dev-Board-Power-1/
// - Comments added to show understanding
```

```
// - Receive bits from the Basys3's UART rx line and turn it into usable data

module receiver (
   input  logic clk,       // 10 MHz clock
   input  logic reset,     // Active high reset
   input  logic rx,        // UART recieve line

   output logic [7:0] data  // RX command recieved
);

   logic shift;
   logic state;
   logic nextstate;
   logic clear_bitcounter;
   logic inc_bitcounter;
   logic inc_samplecounter;
   logic clear_samplecounter;

   logic [3:0]  bitcounter;
   logic [1:0]  samplecounter;
   logic [13:0] counter;
   logic [9:0]  rxshiftreg;

   localparam clk_freq = 100_000_000;
   localparam baud_rate = 9_600;
   localparam div_sample = 4;
   localparam div_counter = clk_freq / (baud_rate * div_sample);
   localparam mid_sample = (div_sample / 2);
   localparam div_bit = 10;

   assign data = rxshiftreg[8:1];

   // Handles flags thrown by the finite state machine
   always @(posedge clk) begin
      if (reset) begin
         state <= 0;
         bitcounter <= 0;
         counter <= 0;
         samplecounter <= 0;
      end else begin
         counter <= counter + 1;
         if (counter >= div_counter - 1) begin
            counter <= 0;
            state <= nextstate;
            if (shift) rxshiftreg <= {rx, rxshiftreg[9:1]};
            if (clear_samplecounter) samplecounter <= 0;
            if (inc_samplecounter)   samplecounter <= samplecounter + 1;
            if (clear_bitcounter)    bitcounter <= 0;
```

```
        if (inc_bitcounter)     bitcounter<= bitcounter+ 1;
      end
    end
end

// Finite state machine!!
always@(posedge clk) begin
    shift <= 0;
    clear_samplecounter <= 0;
    inc_samplecounter <= 0;
    clear_bitcounter <= 0;
    inc_bitcounter <= 0;
    nextstate <= 0;

    case(state)

      // Idle until 0 is send over the line
      // First 0 just shows that transmission is starting, so don't
      //    need to store it
      0: begin
        if (rx) begin
          // If line is kept high, stay in idle
          nextstate <= 0;
        end else begin
          // If line pulled low, start to recieve
          nextstate <= 1;         // Go to recieve state
          clear_bitcounter <= 1;    // Reset bit counter
          clear_samplecounter <= 1;  // Reset sample counter
        end
      end

      // Receiving!
      1: begin
        // Unless finished, keep recieving
        nextstate <= 1;

        // If in the middle of a bit, shift the rx register and
        //    the incoming data as the newest bit
        if (samplecounter == mid_sample - 1) begin
          shift <= 1;
        end

        // If at the end of a transmitted bit, either go
        //    to the next bit or finish transmission
        if (samplecounter == div_sample - 1) begin
          // UART send 10 bit long packets, if at the
          //    end of a packet then go back to idle
          if(bitcounter == div_bit - 1 ) begin
```

```
            nextstate <= 0;
          end

          // Step up by counter by one bit and reset
          //    the sample counter
          inc_bitcounter <= 1;
          clear_samplecounter <= 1;
        end else begin
          // Increase the sample counter
          inc_samplecounter <= 1;
        end
      end

      default: nextstate <= 0;
    endcase
  end
endmodule
```

top_level.sv

```
// File info:
//  - Original file
//  - Note all physical units are in inches
//  - Project inspired by https://www.youtube.com/watch?v=JpzeoQI2MII

`timescale 1ns / 1ps

module top_level(
  // Inputs
  input logic clk,       // 10 MHz clock
  input logic reset,     // Middle button on Basys3
  input logic rx,        // UART recieve
  input logic ultrasonic,  // Ultrasonic sensor PWM
  input logic [15:0] switches_inputs,  // Switches

  // Outputs
  output logic tx,          // UART transmit
  output logic shoulder_servo,  // Shoulder Servo PWM
  output logic elbow_servo,    // Elbow Servo PWM
  output logic [15:0] led,     // LEDs

  // Display cathodes
  output logic CA,
  output logic CB,
  output logic CC,
  output logic CD,
  output logic CE,
```

```
    output logic CF,
    output logic CG,
    output logic DP,

    // Display anodes
    output logic AN1,
    output logic AN2,
    output logic AN3,
    output logic AN4
    );

    // State of the FSM
    logic [1:0] state;

    // Whether arm is keyboard controlled or ultrasonic controlled
    logic keyboardControlled;
    logic ultrasonicControlled;

    // Finite State Machine
    // Possible states:
    //  - Start state
    //  - Keyboard Controlled
    //  - Ultrasonic Controlled
    fsm f (
      .clk(clk),
      .reset(reset),

      .right(switches_inputs[0]),
      .left(switches_inputs[15]),

      .state(state),
      .keyboardControlled(keyboardControlled),
      .ultrasonicControlled(ultrasonicControlled)
    );

    // The (x,y) coordinates of the arm if controlled by the ultrasonic
    //   - Only used one ultasonic, so hardcode the y to be 2"
    logic [7:0] ultrasonic_x;
    logic [7:0] ultrasonic_y = 2;

    // Measure the ultrasonic sensor's PWM signal
    //  - Sensor also has UART and Analog outputs
    //  - Measuring PWM seemed the coolest
    pwm_measure pwmm (
      .clk(clk),
      .reset(reset),
      .pwm_in(ultrasonic),
      .distance(ultrasonic_x)
```

```
);

// The (x,y) coordinates of the arm if controlled by keyboard
//   - Also return the data sent over UART for display
logic [7:0] data;
logic [7:0] keyboard_x;
logic [7:0] keyboard_y;
keyboardControl #(4, 4) kbc (
   .clk(clk),
   .reset(reset),
   .rx(rx),
   .data(data),
   .x(keyboard_x),
   .y(keyboard_y)
);

// Wrapper to do all the finite state machine controll as the FSM
//    above only decides the state, it doesn't do much with it
fsm_controller fsmc (
   .clk(clk),
   .reset(reset),
   .state(state),

   // Keyboard Controlled Information
   .keyboardControlled(keyboardControlled),
   .keyboard_x(keyboard_x),
   .keyboard_y(keyboard_y),
   .data(data),

   // Ultrasonic Controlled Information
   .ultrasonicControlled(ultrasonicControlled),
   .ultrasonic_x(ultrasonic_x),
   .ultrasonic_y(ultrasonic_y),

   // Outputs
   .led(led),
   .shoulder_servo(shoulder_servo),
   .elbow_servo(elbow_servo),
   .tx(tx),

   .CA(CA),
   .CB(CB),
   .CC(CC),
   .CD(CD),
   .CE(CE),
   .CF(CF),
   .CG(CG),
   .DP(DP),
```

```
    .AN1(AN1),
    .AN2(AN2),
    .AN3(AN3),
    .AN4(AN4)
  );


endmodule
```

syncFIFO.sv

```
// File info:
// - Verilog taken from https://www.chipverify.com/verilog/synchronous-fifo
// - Upgraded from V to SV
// - Changes from active low to active high reset
// - Added a write clock and read clock (so it can go across clock domains)
// - Added comments to show understanding
// - Implements a synchronous first-in first-out queue

module syncFIFO #(parameter DEPTH=8, DWIDTH=1000000)
(
  input logic reset,   // Active high reset
  input logic w_clk,  // Write clock
  input logic r_clk,  // Read clock
  input logic w_en,   // Write enable
  input logic r_en,   // Read enable

  input  logic [DWIDTH-1:0] din,  // Data into FIFO
  output logic [DWIDTH-1:0] dout,  // Data out of FIFO

  output logic empty,  // High when FIFO is full
  output logic full    // Low when FIFO is full
);

  logic [$clog2(DEPTH)-1:0] wptr;  // Write pointer
  logic[$clog2(DEPTH)-1:0] rptr;   // Read pointer
  logic [DWIDTH-1:0] fifo[DEPTH];  // The actual storage

  // All the writing happens here
  always @ (posedge w_clk) begin
    if(reset) begin
      wptr <= 0;
    end else begin
      // If writing and not full, then write incoming data
      // Overflow makes the write pointer effectivley a circular buffer
      if (w_en & !full) begin
```

```
            fifo[wptr] <= din;
            wptr <= wptr + 1;
        end
    end
end

// This was super helping for debugging
initial begin
    $monitor(
        "[%0t] [FIFO] w_en=%0b din=0x%0h rd_en=%0b dout=0x%0h empty=%0b full=%0b",
        $time,
        w_en,
        din,
        r_en,
        dout,
        empty,
        full);
end

// All the reading stuff
always @ (posedge r_clk) begin
    if (reset) begin
        rptr <= 0;
    end else begin
        // If reading and not empty, output the data and increase pointer
        if (r_en & !empty) begin
            dout <= fifo[rptr];
            rptr <= rptr + 1;
        end
    end
end

// Because it's a circular queue with different read and write
//    pointers, full is when you're about the write to the
//    the front of the queue. However, it's fully empty when
//    you're reading where the write pointer is

// Full if write pointer is about to be at read pointer
assign full = ((wptr +1) == rptr);

// Empty if read pointer is at write pointer
assign empty = (wptr == rptr);


endmodule
```

transmitter.sv

```
// File info:
// - Verilog copied from: https://www.instructables.com/UART-Communication-on-Basys-3-FPGA-Dev-Board-Power/
// - Comments added to show understanding
// - Transmits one packet worth of data over UART transmit line

module transmitter(
  input clk,       // 10 MHz clock
  input reset,     // Active high reset
  input transmit,  // Whether or not to transmit
  input [7:0] data,  // Data to transmit
  output reg TxD    // UART transmit line
);

  reg [3:0] bitcounter;
  reg [13:0] counter;
  reg state,nextstate;
  reg [9:0] rightshiftreg;
  reg shift;
  reg load;
  reg clear;

  // Does stuff based on flags changed by the FSM
  always @ (posedge clk) begin
    if (reset) begin
      state <=0;
      counter <=0;
      bitcounter <=0;
    end else begin
      counter <= counter + 1;

      // 10415 = master clock frequency / baud rate of 9600
      // Bayas3 is going WAY too fast to be at 9600 baud, so need
      //    to do stuff only on 9600 Hz
      if (counter >= 10415) begin
        state <= nextstate;
        counter <=0;

        // Load data with needed UART start/stop bits
        if (load) begin
          rightshiftreg <= {1'b1,data,1'b0};
        end

        if (clear) begin
          bitcounter <=0;
        end
```

```
        if (shift) begin
          rightshiftreg <= rightshiftreg >> 1;
          bitcounter <= bitcounter + 1;
        end
      end
    end
end

// Finite state machine!
always @ (posedge clk) begin
    load <=0;
    shift <=0;
    clear <=0;
    TxD <=1;  // Default held high because when UART is idle, the line is high

    case (state)
      // Idle
      0: begin
        if (transmit) begin
          nextstate <= 1;
          load <=1;
          shift <=0;
          clear <=0;
        end else begin
          nextstate <= 0;
          TxD <= 1;
        end
      end

      // Transmitting
      1: begin
        // UART is a 10 bit protocol, so if 10 bits have been
        //    sent then go back to idle
        if (bitcounter >=10) begin
          nextstate <= 0;
          clear <=1;
        end else begin
          // Send the next bit in the shift register and shift it
          // Shift register works like a queue for this
          nextstate <= 1;
          TxD <= rightshiftreg[0];
          shift <=1;
        end
      end

      default: nextstate <= 0;
```

```
    endcase
  end


endmodule
```

clk_divider_tb.sv

```
`timescale 1ns / 1ps

module clk_divider_tb();

  // Parameters
  parameter CLK_PERIOD = 10; // 10ns for 100MHz clock

  // Signals
  logic clk = 0;
  logic reset = 0;
  logic [31:0] divisor = 0;
  logic slow_clk = 0;

  // Instantiate the Unit Under Test (UUT)
  clk_divider uut ( clk, reset, divisor, slow_clk );

  always #(CLK_PERIOD / 2) clk = ~clk;

  // Test stimulus
  initial begin
    reset = 0;
    #(10 * CLK_PERIOD);
    reset = 1;
    #(10 * CLK_PERIOD);
    reset = 0;

    divisor = 8'd1;
    #(50 * CLK_PERIOD);

    divisor = 8'd2;
    #(50 * CLK_PERIOD);

    divisor = 8'd3;
    #(50 * CLK_PERIOD);

    divisor = 8'd4;
    #(50 * CLK_PERIOD);
```

```
    $stop;
  end

endmodule
```

divider_tb.sv

```
`timescale 1ns / 1ps

module divider_tb();

  // Parameters
  parameter CLK_PERIOD = 10; // 10ns for 100MHz clock

  // Signals
  logic clk = 0;
  logic reset = 0;
  logic [31:0] number = 0;
  logic [31:0] divisor = 0;
  logic [31:0] quotient = 0;
  logic [31:0] remainder = 0;

  // Instantiate the Unit Under Test (UUT)
  divider uut (
    clk,
    reset,
    number,    // Dividend
    divisor,    // Divisor
    quotient,    // Quotient
    remainder     // Remainder
  );

  always #(CLK_PERIOD / 2) clk = ~clk;

  // Test stimulus
  initial begin
    #(100 * CLK_PERIOD);

    number = 9;
    divisor = 1;
    #(50 * CLK_PERIOD);

    number = 6;
    divisor = 2;
    #(50 * CLK_PERIOD);

    number = 8;
```

```
    divisor = 5;
    #(50 * CLK_PERIOD);

    number = 60_001;
    divisor = 2;
    #(60000 * CLK_PERIOD);
    $stop;
  end

endmodule
```

fsm_tb.sv

```
`timescale 1ns / 1ps

module fsm_tb();

  // Parameters
  parameter CLK_PERIOD = 10; // 10ns for 100MHz clock

  // Signals
  logic clk = 0;
  logic reset = 0;
  logic right = 0;
  logic left = 0;

  logic [1:0] state = 0;
  logic keyboardControlled = 0;
  logic ultrasonicControlled = 0;

  // Instantiate the Unit Under Test (UUT)
  fsm uut (
    clk,
    reset,

    right,
    left,

    state,
    keyboardControlled,
    ultrasonicControlled
  );

  always #(CLK_PERIOD / 2) clk = ~clk;

  // Test stimulus
  initial begin
```

```
    reset = 1;
    #(5 * CLK_PERIOD);

    reset = 0;
    right = 0;
    left = 0;
    #(20 * CLK_PERIOD);

    right = 0;
    left = 1;
    #(20 * CLK_PERIOD);

    right = 1;
    left = 0;
    #(20 * CLK_PERIOD);

    right = 0;
    left = 0;
    #(20 * CLK_PERIOD);
    $stop;
  end

endmodule
```

inverse_kinematics_tb.sv

```
`timescale 1ns / 1ps

module inverse_kinematics_tb();

  // Parameters
  parameter CLK_PERIOD = 10; // 10ns for 100MHz clock

  // Signals
  logic clk = 0;
  logic reset = 0;
  logic [7:0]  x = 1;
  logic [7:0]  y = 1;
  logic [23:0] shoulder_angle = 0;
  logic [23:0] elbow_angle = 0;

  // Instantiate the Unit Under Test (UUT)
  inverse_kinematics uut (
    clk,
    reset,
    x,
    y,
```

```
    shoulder_angle,
    elbow_angle
  );

  always #(CLK_PERIOD / 2) clk = ~clk;

  // Test stimulus
  initial begin
    // End simulation
    reset = 1;
    #(10 * CLK_PERIOD);
    reset = 0;

    x = 1;
    y = 2;
    #(10 * CLK_PERIOD);

    x = 1;
    y = 2;
    #(10 * CLK_PERIOD);

    x = 2;
    y = 3;
    #(10 * CLK_PERIOD);

    x = 4;
    y = 4;
    #(10 * CLK_PERIOD);

    x = 4;
    y = 2;
    #(10 * CLK_PERIOD);

    $stop;
  end

endmodule
```

pwm_enable_tb.sv

```
`timescale 1ns / 1ps

module pwm_enable_tb();

  // Parameters
  parameter CLK_PERIOD = 10; // 10ns for 100MHz clock
```

```
  // Signals
  logic [7:0] data = 0;
  logic clk = 0;
  logic reset = 0;
  logic en = 0;

  // Instantiate the Unit Under Test (UUT)
  pwm_enable uut (
    .data(data),
    .clk(clk),
    .reset(reset),
    .en(en)
  );

  always #(CLK_PERIOD / 2) clk = ~clk;

  // Test stimulus
  initial begin
    // End simulation
    reset = 1;
    #(10 * CLK_PERIOD);

    data = 1;
    reset = 0;
    #(20 * CLK_PERIOD);

    data = 2;
    #(20 * CLK_PERIOD);

    // data = 2;
    // #(20 * CLK_PERIOD);

    // data = 1;
    // #(20 * CLK_PERIOD);
    $stop;
  end

endmodule
```

pwm_measure_tb.sv

```
`timescale 1ns / 1ps

module pwm_measure_tb();

  // Parameters
  parameter CLK_PERIOD = 100; // 100ns for 10MHz clock
```

```
parameter SERVO_PERIOD = 50_000_000;
//parameter SERVO_PERIOD = 600_000;

// Signals
logic [7:0] distance = 0;
logic [7:0] target = 0;
logic clk = 0;
logic reset = 0;
logic pwm_in = 0;

// Instantiate the Unit Under Test (UUT)
pwm_measure #(1664) uut  ( clk, reset, pwm_in, distance );

always #(CLK_PERIOD / 2) clk = ~clk;

// Test stimulus
initial begin
  // End simulation
  reset = 1;
  #(10 * CLK_PERIOD);

  reset = 0;
  #(10 * CLK_PERIOD);

                pwm_in = 1;
                target = 4;
  #(147_000 * target);
                pwm_in = 0;
  #(SERVO_PERIOD - (147_000 * target));

                pwm_in = 1;
                target = 1;
  #(147_000 * target);
                pwm_in = 0;
  #(SERVO_PERIOD - (147_000 * target));

                pwm_in = 1;
                target = 2;
  #(147_000 * target);
                pwm_in = 0;
  #(SERVO_PERIOD - (147_000 * target));

  pwm_in = 1;
  target = 4;
  #(147_000 * target);
                pwm_in = 0;
  #(SERVO_PERIOD - (147_000 * target));
```

```
    pwm_in = 1;
    target = 3;
    #(147_000 * target);
                pwm_in = 0;
    #(SERVO_PERIOD - (147_000 * target));

    pwm_in = 1;
    target = 4;
    #(147_000 * target);
                pwm_in = 0;
    #(SERVO_PERIOD - (147_000 * target));

    $stop;
  end

endmodule
```

pwm_tb.sv

```
`timescale 1ns / 1ps

module pwm_tb();

  // Parameters
  parameter CLK_PERIOD = 10; // 10ns for 100MHz clock

  // Signals
  logic clk = 0;
  logic signal = 0;
  logic en = 0;
  logic [23:0] angle = 0;

  // Instantiate the Unit Under Test (UUT)
  pwm uut ( .clk(clk), .en(en), .angle(angle), .servo(signal));

  always #(CLK_PERIOD / 2) clk = ~clk;

  // Test stimulus
  initial begin
    en = 0;
    #(100 * CLK_PERIOD);
    en = 1;

    angle = 50_000;
    #(400_000 * CLK_PERIOD);

    angle = 100_000;
```

```
    #(400_000 * CLK_PERIOD);
    $stop;
  end

endmodule
```

syncFIFO_tb.sv

```
// File info:
// - Verilog taken from https://www.chipverify.com/verilog/synchronous-fifo

`timescale 1ns / 1ps

module syncFIFO_tb();
  // Parameters
  parameter CLK_PERIOD = 10; // 10ns for 100MHz clock

  // Signals
  reg w_clk;
  logic r_clk;
  logic [3:0] din;
  logic [3:0] dout;
  logic [3:0] rdata;
  logic empty;
  logic r_en;
  logic w_en;
  logic full;
  logic reset;
  logic stop;

  // Instantiate the Unit Under Test (UUT)
  syncFIFO uut (
    .reset(reset),
    .w_en(w_en),
    .r_en(r_en),
    .w_clk(w_clk),
    .r_clk(r_clk),
    .din(din),
    .dout(dout),
    .empty(empty),
    .full(full)
  );

  always #(CLK_PERIOD) w_clk <= ~w_clk;
  assign r_clk = w_clk;

  initial begin
```

```
    w_clk <= 0;
    reset <= 1;
    w_en <= 0;
    r_en <= 0;
    stop <= 0;

    #(CLK_PERIOD * 5) reset <= 0;
  end

  // Write data into FIFO
  initial begin
    @(posedge w_clk);

    for (int i = 0; i < 50; i = i+1) begin
      // Wait until space
      while (full) begin
        @(posedge w_clk);
        $display("[%0t] FIFO full", $time);
      end;

      // Put new values
      w_en <= $random;
      din <= $random;
      $display(
        "[%0t] w_clk i=%0d w_en=%0d din=0x%0h ",
        $time,
        i,
        w_en,
        din
      );

      // Wait for posedge on clock
      @(posedge w_clk);
    end

    stop = 1;
  end

  initial begin
    @(posedge r_clk);

    while (!stop) begin
      // Wait until data
      while (empty) begin
        r_en <= 0;
        $display("[%0t] FIFO empty", $time);
        @(posedge r_clk);
      end;
```

```
        // Sample new values from FIFo at random pace
        r_en <= $random;
        @(posedge r_clk);
        rdata <= dout;
        $display(
          "[%0t] r_clk r_en=%0d rdata=0x%0h ",
          $time,
          r_en,
          rdata
        );
      end

      #(CLK_PERIOD * 50) $finish;
    end
endmodule
```