Maker Portfolio

Oliver Trevor

October 25, 2020

Contents

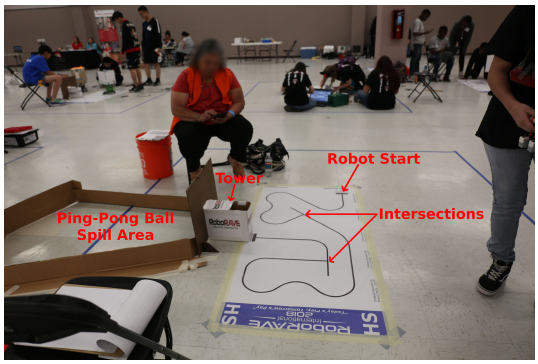
1 One-Page Project Summaries

Each of the following pages is a single-page summary of a project. For the more complex projects, full technical specifications are provided after the single-page summaries.

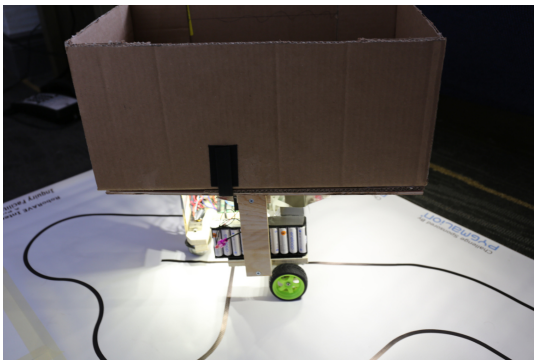
1.1 Solving the Impossible Line-Following Track: RoboRAVE 2018 International Competition

The RoboRAVE International robotics competition is an open, multinational robotics competition with a variety of challenges for different age groups. My teammate Chaitu Nookala (Dougherty Valley HS) and I competed and won 1st place in the High School division in 2018 (held in Albuquerque, New Mexico with over 30 countries participating) with our robot for the line-following challenge. Previous versions of our robot had won 1st place in California and USA National competitions in 2016, 2017, and 2018. Our robot also won in California in 2019 following a complete teardown and rebuild.

The task given to the robot was to follow a thin black line on a mat containing two T-shaped intersections, deliver one single ping-pong ball to a tower at the end, return to start, then deliver about 300 (number changed with each competition) ping-pong balls to the tower, all within a 3-minute time limit.

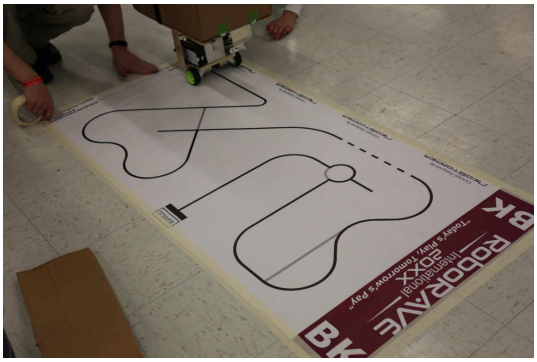


The Task



Our Robot

However, at the competition, RoboRAVE revealed a new "impossible" track intended for college students called the "Big Kids" track:



To this day, our robot remains the only one to ever have successfully completed the "Big Kids" track. The technical specifications detail how we designed a new computer-vision-based solution to the challenge that was flexible enough to complete even the impossible track and how we built a ping-pong ball delivery system that was (in our testing done at the competition) 100% reliable.

I built the hardware for the robot and wrote all of the image-processing and control code used in the software linked at the end of this PDF, as well as the RobotShell debugging utility we used to manually operate and test parts of the robot's automatic functions. My teammate was Chaitu Nookala from Dougherty Valley High School.

1.2 ORISC – Designing a Custom CPU Architecture and Toolchain for the Altera DE1 FPGA

To further my own understanding of how computers work at a fundamental level, and to create a teaching tool to explain those concepts to others, I designed my own processor architecture in Verilog, then wrote a full software stack to allow programming of the processor.

The only FPGA (board for prototyping custom digital logic) available to me was a relatively old Altera DE1, so I designed the processor to work within the constraints of that hardware. Additionally, I tried to make the architecture as simple as possible to make it more useful as a teaching aid.

The final result included the following hardware components, which I designed in Verilog (a hardware description language used to design digital logic) over about 1 year and 8 months:

- Central Processing Unit (CPU) with my custom Instruction Set Architecture (ISA)
  - implemented **26** instructions using a RISC Load/Store architecture
  - **10** 32-bit registers
  - capable of handling multiple interrupts
- Graphics Processing Unit (GPU) that can render textual output from programs to a VGA display
- PS/2 keyboard interface – digital logic to trigger CPU interrupts when keys are pressed on an external PS/2 keyboard

And the following software components:

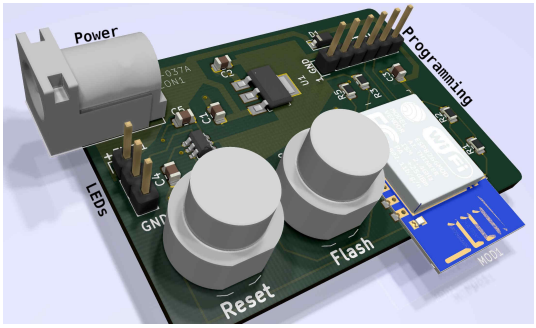
- Assembler – converts assembly code into machine code (binaries) to run on the CPU
- Disassembler – converts machine code (binaries) for the CPU back into assembly – used to check that the assembler is working correctly
- Emulator – software implementation of the CPU architecture – used to verify that the hardware implementation of the CPU is operating according to the specification
- **Compiler** – converts high-level code into assembly code for the CPU
  - implements the PL/0 language, a simple high-level programming language based loosely on Pascal
  - supports:
    - call stack for procedures (functions) – stores variables in an 8MB SDRAM
    - global and local variables
    - conditionals
    - while loops
    - procedures (functions)
    - integer arithmetic expressions
    - string literals
    - calling input/output instructions
  - capable of compiling a program that computes prime numbers and displays them using the GPU

1.3 Blinkstream – Custom Printed Circuit Board for Wirelessly Controlling LED Mood Lighting Strips

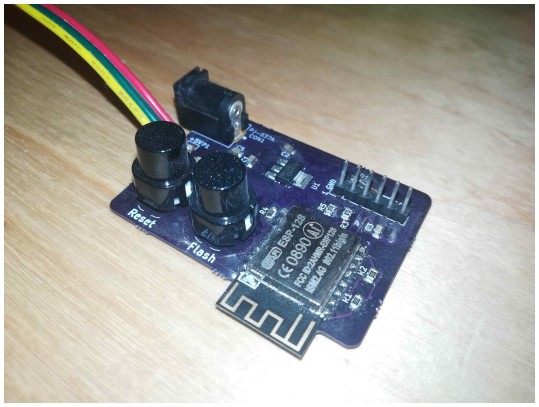
To teach myself how to design printed circuit boards, and to fill a gap in the market for a Wi-Fi-controlled, IoT-based mood lighting system that didn't require a proprietary app, I used KiCAD to design a PCB with a system for controlling NeoPixel LED lighting strips.

The device has just three connections: a 5V barrel jack connector for power, a 3-pin output connector for NeoPixel lighting strips, and a header for plugging in FTDI cables for reprogramming the board.

When a user receives a Blinkstream device, it creates a Wi-Fi network called "blinkstream" that they connect to, which provides a webpage where they can configure their home network settings and the number of LEDs connected to the controller. The controller calculates the power requirements for the LED strip based on the length and tells the user to check that their power supply is adequate (it has been tested with up to 300 LEDs). Once the user has configured their device, it reboots and connects to their home network, where it can be controlled from a web interface.



3D Rendering of the Board (Without Case)



Finished Actual Board (Without Case)

Manufacturing at Home - I used OSH Park (a circuit board prototyping service in Oregon) to manufacture my PCB design, which gave me a base PCB with no components attached. To actually build the board, I had to develop a home PCB assembly process that would work with small, surface-mounted components. Based on online resources and some experimenting, I developed a way of using an electric skillet (the $35 kind from Target used for cooking) to reflow solder paste in order to solder surface-mount components to the PCB.

The Blinkstream device showing a basic demo pattern that blends from magenta to purple. It can also show a variety of animated patterns.

1.4 Dual-Band "Copper Cactus" Ham Radio Antenna

I hold a General license to operate amateur (ham) radios in the United States and other countries with mutual-licensing agreements (my callsign is KM6WOX). Since my house is situated in a valley, it's a little difficult to get a radio signal to reach very far. To solve this problem, I built my own roof-mounted ham radio antenna for the 2-meter and 70-centimeter radio bands (ham radio frequencies are identified by their approximate wavelengths in a vacuum).

I based the antenna on specifications by Tony Petersen (callsign N7QVC) online, but added a long PVC pipe mast to raise the antenna above my roof. I also tuned the antenna by moving its connection to the radio (the feedpoint).

The main goal of the project was to build an effective antenna and learn the technical skill of **soldering copper pipe**. I was able to tune the antenna well enough to consistently reach a variety of distant ham radio systems on the 2-meter band:

- The "WIN System" ham radio repeater network via a repeater (callsign WA7G) in Napa Valley, CA, about 50 miles from my home.
- The Mt. Diablo Amateur Radio Club repeater (callsign W6CX) on the peak of a nearby mountain.
- The Woodside, CA packet radio Bulletin Board System (callsign N7ZX-4).
- The South Bay packet radio Bulletin Board System (callsign N0ARY-1, using N6ZX-4 to forward traffic).

1.5 Chicken Mister System

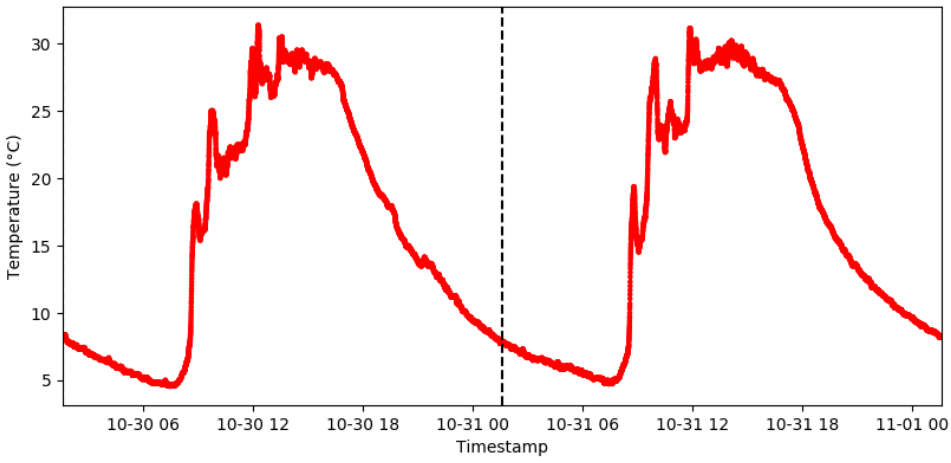
As I mention in my 900-character responses on SlideShare, one of the projects I'm most proud of for solving a pressing issue was the automatic misting system I built that turned on a water mister to keep my pet chickens cool when the temperature got too hot during the California summer.

The system grew over time to also include the following extra components for maintaining our garden and keeping the chickens happy:

- A system that automatically turned incandescent Christmas lights on in our lemon tree on frosty nights to prevent the frost killing the tree
- An outdoor temperature data-logging system that tracked the local temperature over many years to study the local microclimate
  - The SQLite 3 database I built on the Raspberry Pi to track historical temperature data currently holds 6.3 million data points, taken once every 10 seconds for years
- A remote camera system to watch the chickens while my family is on vacation

The system uses a waterproofed OneWire DS18B20 temperature sensor connected to a Raspberry Pi inside the chicken coop that logs the data and controls a relay for the water mister in summer and the Christmas lights in winter. An ordinary 24 VAC sprinkler power supply and sprinkler valve control the flow of water to the mister.

Here's some example data from the last 48 hours in the temperature database:



1.6 CardsToHumanity

I worked with Foothill Graphic Design Club to build a website that distributes virtual cards to front-line workers. As the backend developer, I was in charge of architecuturing the database, API, and layout of all the data. I chose to use a MySQL/MariaDB database to store user and card data, with Amazon S3 serving media (images) to reduce load on our servers (hosted through Heroku).

I wrote the entire backend in Java because it seemed to be the language most other programmers at the school know, which would mean that the club could continue maintaining the site in future (although I have committed to maintaining the backend myself for the forseeable future).

I designed a simple REST API that allowed the frontend (which was partially mine and partially written by another developer, Prem Giridhar, using Vue.JS) to fetch data from the backend asychronously without adding the load of generating templated content to the backend. Most data is sent between the frontend and backend using plain JSON serialized from Java objects using the Google Gson library.

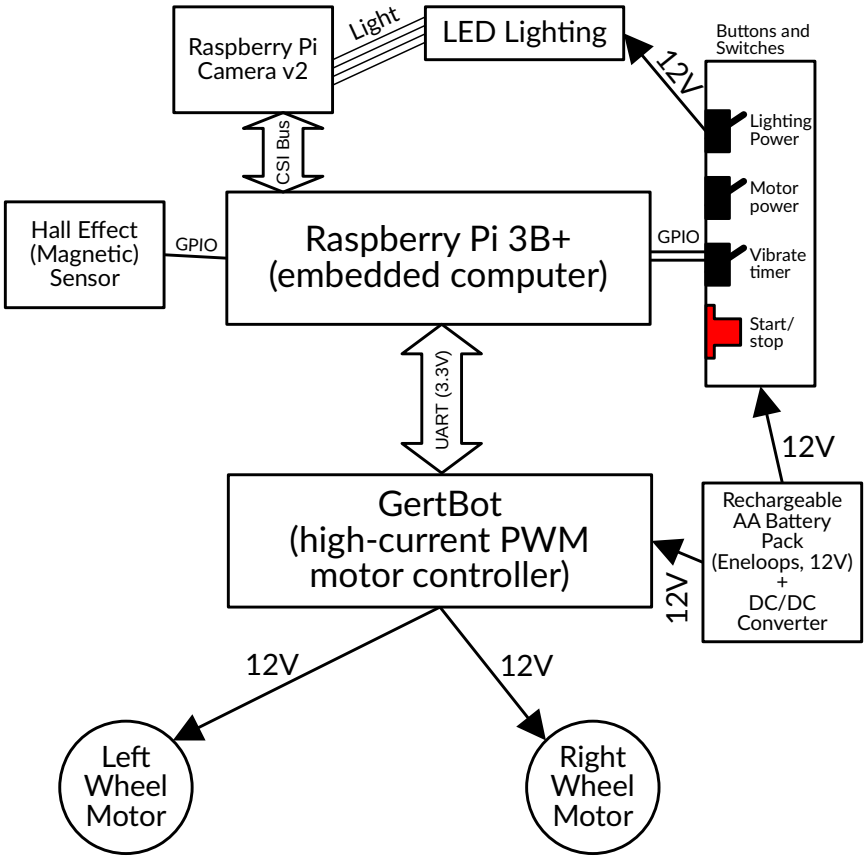
I also used the Javalin web framework for Java/Kotlin, the SLF4J logging library, the MariaDB client library for Java, AWS's SDK, SendGrid's SDK, the C3P0 database connection pooling library, the JavaFaker test data generation library, and the Selenium automatic testing framework.

The site can be viewed at https://www.cardstohumanity.com. We encourage everybody who can to submit a card or two for front-line workers.

2 Technical Specifications

2.1 Solving the Impossible Line-Following Track: RoboRAVE 2018 International Competition

My team's robot in its final revision. The orange bucket is used to load ping-pong balls into the hopper for delivery. See block diagram.



Broadly speaking, our robot performed faster and more reliably than many other designs because of two things:

1) Better error-handling ability

2) More raw sensor data

Previous robots for the line following challenge had used infrared color sensors to track the location of the line, but our testing found that these sensors were prone to errors caused by light reflected from the shiny surface of the track. We instead opted for a camera-based system that used computer vision to identify the location of the line in front of the robot.

The task given to all line followers was as follows:

1) Follow the line from start to finish, dropping one single ping-pong ball into the cardboard tower at the finish line.

2) Follow the line back to start.

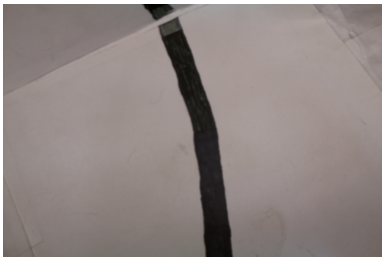
3) Load up the robot with 300 ping-pong balls (number varied between years). This task is done by the robot's operators.

4) Follow the line from start to finish once again, dropping all 300 ping-pong balls into the tower within the 3-minute time limit.

On the high school tracks, the line was made thinner, and two T-shaped intersections were added to confuse the robot. Points would be deducted for failing to follow the line, losing ping-pong balls, or other errors. This scoring system meant that the robot had to be able to follow the line at a relatively high speed without losing track of it at any time. We created the following computer vision algorithm using OpenCV to detect the location of the line:

2.1.1 Image Processing Algorithm

Step 1: Acquire Grayscale Image from Camera



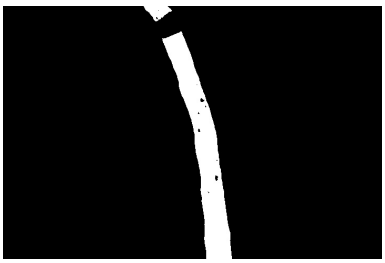
The first revision of the robot's software was a 300-line Python 3 program that captured and processed images at roughly 10 frames per second, which limited the sped at which the robot could react to sharp turns. We later rewrote the software from scratch into an approximately 2,000-line C++ system (broken into about 10 different classes) that captured and processed at **30 frames per second**.

The switch from Python to C++ was partially due to Python's inability to do true multi-threading because of the Global Interpreter Lock built into the CPython implementation of the Python interpreter. Our C++ version of the program used a separate thread to capture images from the camera to avoid the processing stage slowing down the capturing stage.

Step 2: Crop the Image

Many of the RoboRAVE International line-following tracks had logos printed on the side of the track near the line. These confused the image-processing system, so we added an option to the software to configure how much of the image to crop off.

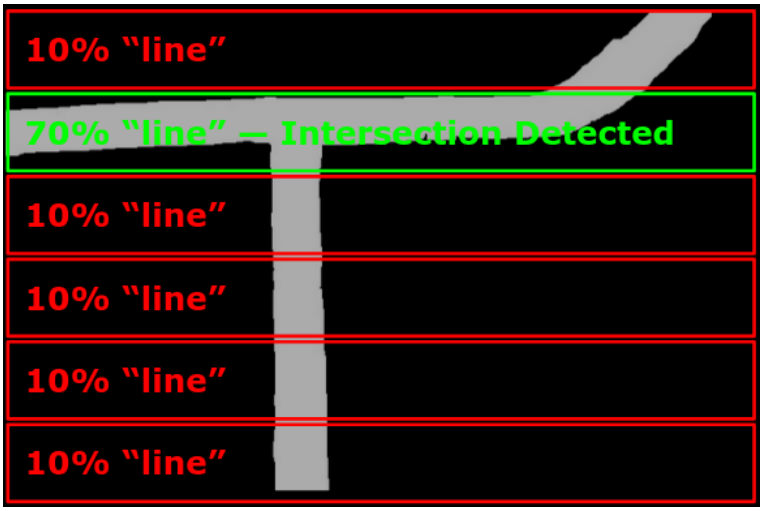
Step 3: Threshold the Image



We used OpenCV to apply an inverse binary threshold to the image, which used the intensity of pixels in the image to create black-and-white image of the line with shadows and other imperfections removed. An inverse threshold was used so that the object of interest (the line) would be given the highest pixel value (255), making it easier to calculate the location of the line.

Since our image threshold was configurable, we were able to tell the robot to ignore the gray "false paths" placed on the Big Kids track to confuse line followers.

Step 4: Check for an Intersection



Since the line follower's control systems (detailed in the next section) cannot tell which "fork" leading out of an intersection to take, we had to create a special case in the software to detect intersections and execute a 90° turn.

The intersection-detection routine breaks the image into horizontal segments, with each one being approximately as wide as the line. It computes the number of white pixels within each section. If a section has more than a certain number of white pixels, it recognizes the entire image as an intersection and tells the motor control system to turn 90°.

If the image was *not* an intersection, the algorithm continues on to step 5.

Step 5: Check if the Line Has Been Lost

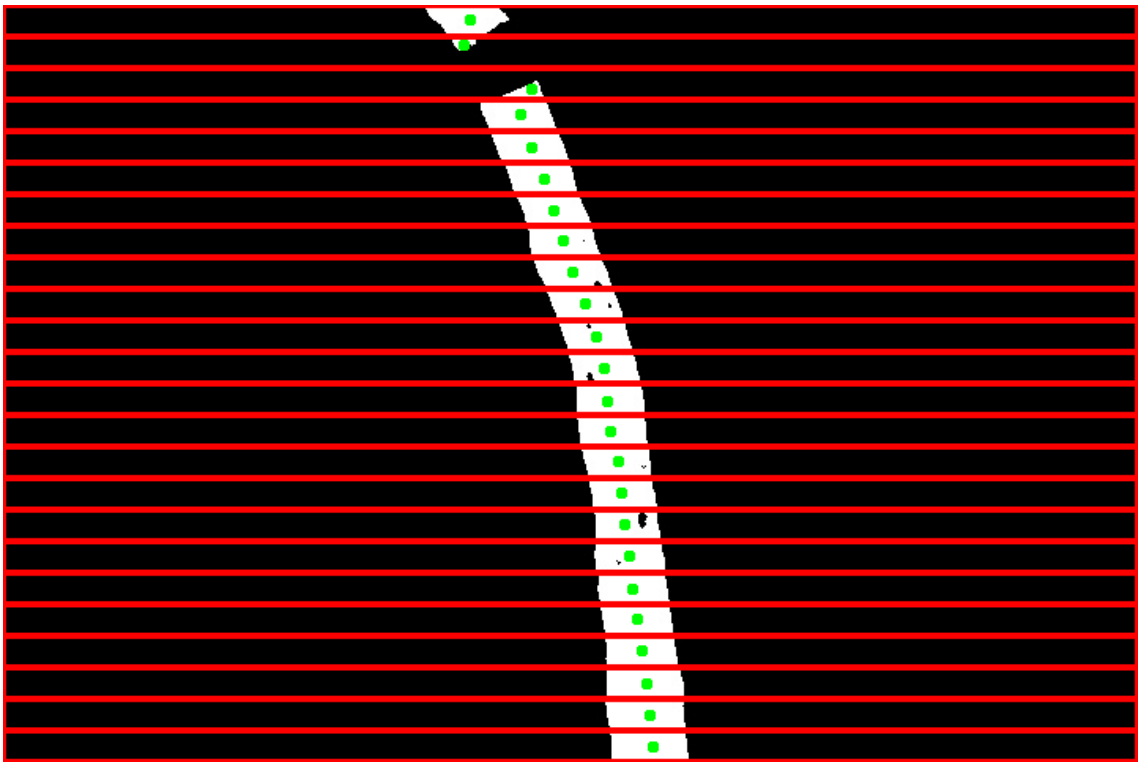
This step is arguably the most important one in our image-processing system. Unlike color-sensor-based line-followers, our camera system was able to recognize when it had driven too far and lost the line entirely.

It recognized this state by adding up the total number of white pixels in the thresholded image. If there were less than 50 white pixels in the image, the image processing system classified the image as "line lost" and told the motor control system to reverse slowly until a line was found.

This error-handling mechanism meant that the robot had the ability to recover effectively from what would otherwise be a game-ending scenario.

If the line was *not* lost, the algorithm continues on to step 6.

Step 6: Use "Center of Mass" to Find the Line



The algorithm breaks the image into segments, each one 20 rows of pixels tall (shown as red outlines in the image above). Starting with the bottommost segment, it computes the center of mass of the segment by treating white pixels as having a mass of 255 and black pixels as having a mass of 0. Using the center-of-mass calculation to find the line prevents small aberrations (like dirt or shadows on the track) from affecting where the image processor perceives the line to be. Segments containing less than 25 white pixels are ignored by the image processor, ensuring that breaks in the line do not confuse the line follower.

Additionally, scanning segments from the bottom of the image up ensures that, if there is a break in the line, the robot can still find a line to follow. *This part of the algorithm is why our robot was able to follow dashed lines on the Big Kids track; the image processing system simply "looked ahead" in the image to find where the next segment of the line was.*

To save on processing time, the algorithm stops scanning the image as soon as it finds one segment with a recognizable line. The diagram above shows the centers of mass for all segments as green dots for clarity only; in actual operation, the image processor would find the center of mass of the bottommost segment and stop.

The one-dimensional version of the equation for center of mass (m is "mass," which is pixel value):

$$x_{\text{com}} = \frac{\sum_{i=1}^{i=n} m_i x_i}{\sum_{i=1}^{i=n} m_i}$$

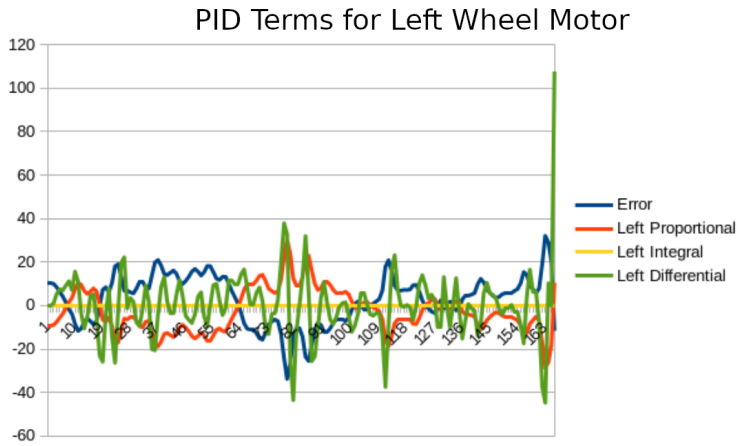
2.1.2 Motor Control System

Our robot used a PID (Proportional-Integral-Derivative or Proportional-Integral-Differential) control system to determine the speeds at which to run the motors based on the distance measured between the line and the center of the image by the image processing system. This ensured that the robot was constantly "trying" to get as close to the line as possible.

The PID system also meant that the robot's motions were smooth, rather than discrete stop-and-go motions that would damage the motors over time. This relatively short C++ function implemented the PID system at the core of our motor-control system:

```
/**
Method to compute the current value of a PID loop.
@param value The input value.
@param target_value The target value of the PID loop.
@returns The output value of the PID loop.
*/
int PID::compute(int value, int target_value) {
    int error = target_value - value;
    if (first_time) {
        previous_error = error;
        i_accumulator = 0;
        first_time = false;
    }
    i_accumulator += error;
    int output = (parameters.p * error) + (parameters.i * i_accumulator) + (parameters.d * (error - previous_error));
    previous_error = error;
    return output;
}
```

PID control systems rely on three constant parameters (P, I, and D) that must be determined for a given control system experimentally. We developed a system that plotted the values of each term in the PID equation for each image processed by the robot and used this data to fine-tune our PID loop parameters (x-axis is time, y-axis is unitless term values):



2.1.3 Electronics

Power

The onboard electronics of the robot were powered by two battery systems:

1) A garden-variety 5 V USB Lithium-Ion battery pack for the Raspberry Pi 3B+ (embedded computer) and GertBot (embedded motor controller).

2) Two 10-packs of off-the-shelf 1.2 V Eneloop AA nickel-metal hydride (NiMH) batteries (the kind usually bought by photographers for flash cameras), with the packs wired in parallel to allow

more current to be drawn (each individual pack was internally wired in series, making the entire battery system give a nominal voltage of 12 V).

We found that, as the batteries discharged, it was hard for the motor control system to effectively compensate for the slightly lowered voltage, so we added a DC/DC converter between the 12 V battery pack and the motor controller to ensure stable power.

Lighting

We found that the camera was able to operate faster (and give images with better contrast) with bright but diffuse lighting, so we added strips of white LED lights with diffusers above the camera. These lights were powered from the 12 V battery packs.

Actuators

The robot has exactly three motors, all of them ordinary 12 V DC motors. Two are used to move the wheels, and one is located inside the ping-pong ball hopper with a heavy off-center weight bonded to its axle. This third motor is used to vibrate the ping-pong ball hopper, preventing the ping-pong balls from jamming as they flow out of the hopper:

Ping-Pong Ball Hopper

Ping-Pong Ball Hopper With Ramp Removed, Showing Vibration Motor

Sensors

All our sensors were off-the-shelf components. We initially used a reed switch (and later a Hall Effect sensor) to detect a hard drive magnet attached to the sliding door underneath the ping-pong ball hopper. This meant that the robot would detect a magnetic field only when the door had been pushed open by hitting the tower at the end of the track (into which ping-pong balls were delivered).

Our main sensor was a Raspberry Pi Camera v2 (using the image-processing algorithm detailed previously). The robot also had a few manually-operated buttons and switches for enabling/disabling features of the firmware and switching modes (see block diagram detailed previously).

2.1.4 Challenges Encountered

Bugs were inevitable in a system that ended up having about 2,000 lines of C++ code. Here are a few of the problems we encountered and how we solved them:

Issue	**Solution**
Vibration from the robot moving would occasionally cause the reed switch (magnetic sensor) to trip too early.	We replaced the reed switch with a Hall Effect sensor, which is a fully solid-state magnetic sensor that does not suffer from false positives caused by vibration (i. e. it can only be triggered by a magnetic field).
At one specific competition, the back of the robot would get stuck on a wooden board that had been added to brace the tower where ping-pong balls were meant to be dumped.	We tore down and rebuilt the robot overnight with a piece of the chassis cut off, leaving an empty space below the ping-pong ball hopper on the robot just taller than the wooden board that allowed it to move past the board freely.
The ping-pong balls in the hopper would jam when the door opened due to friction between their surfaces.	We added a vibration motor to the cardboard back of the ping-pong ball hopper, which broke up the static friction between ping-pong balls.
Dirt or shadows on the track confused the image-processing algorithm, which would think that they were part of the line.	Initially, we used the OpenCV "erode" operation to remove such small aberrations. However, this was CPU-intensive on the relatively constrained Raspberry Pi (the Pi 4 did not exist at the time), so we eventually switched to the "center of mass" computation explained in the algorithm description, which is more resistant to outliers in the data.
Python, although it is a great language for robotics, incurs a lot of overhead from being an interpreted language, especially when doing repetitive calculations in loops, which are common in image processing. It also has the Global Interpreter Lock (GIL), which means that Python programs cannot be "truly multithreaded," limiting them to context-switching between multiple threads, rather than actually running them in parallel.	We initially tried to reduce the overhead from Python by using pre-made, optimized compiled routines from OpenCV and NumPy/SciPy, but we still hit a hard limit of about 10 FPS. We eventually rewrote the entire codebase in C++ with a 3rd-party unofficial C++ library for accessing the Raspberry Pi Camera from OpenCV. This allowed us to capture and process images at 30 FPS, speeding up the robot. C++ also allows true multithreading, so we used a mutex system to pass frames from a fast capture thread to a slower processing/control thread.
Part of the robot's task is to follow the track in reverse, returning *from* the end to the beginning to be reloaded with ping-pong balls.	We initially used a dual-camera setup with the IVPort multiplexer for Raspberry Pi Cameras, which allowed us to have one forwards-facing and one backwards-facing camera that could follow the line both forwards and backwards. However, it became difficult to give enough light to the back camera for it to follow the line effectively, so we eventually changed the robot to just do a 180° turn at the end of the line to return home.

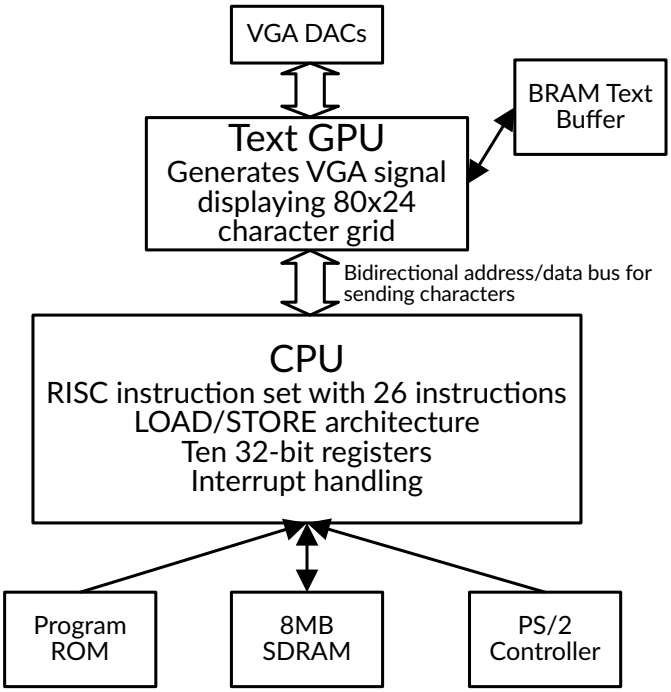
2.1.5 Tools and Libraries Used

The robot's software relied on a few open-source libraries and tools. Here they are:

- OpenCV - used for simple image-processing and display routines
- Raspicam - a third-party (unofficial) C++ library for accessing the Raspberry Pi Camera that allowed us to capture at much higher framerates than the official Python library
- GNU Readline - used to get text input from the user in the debugging shell we built for the robot
- NCurses - used for textual user interfaces for debugging the robot over SSH
- WiringPi - used to access the Raspberry Pi's GPIO pins from C++
- GertBot library - the official C library used to send command over UART to our motor controller
- CMake - the C/C++ build system we used to orchestrate the build process and dependency-finding of our complex program

2.2 ORISC - Custom CPU Architecture and Toolchain

To learn more about custom digital logic in FPGAs, I built my own CPU architecture in Verilog for the Altera DE1 FPGA development board. Over time, the project grew in scope to include creating a toolchain for programming the CPU. A conceptual layout of the device is shown below:



There were a few important architectural considerations I had to make early on in the project. Firstly, the entire CPU and GPU had to fit in the available logic elements of the FPGA chip on the Altera DE1 development board (a Cyclone II board), so the ISA had to be relatively simple. For that reason, I chose to make the instruction set RISC (Reduced Instruction Set Computer) and to have most instructions (with the exception of I/O and division instructions) execute in a single clock cycle.

A full listing of the instruction set follows. Skip the table for an overview of how the instruction set works.

2.2.1 Instruction Set Architecture

Opcode	**Mnemonic**	**Full Name**	**Function**	**Operand 1**	**Operand 2**
0	NOP	No Operation	Does nothing for one clock cycle. Useful for performing delays or busy-waiting for interrupts.	N/A	N/A
1	LOAD	Load	Loads a 32-bit value from RAM to a register.	Hard-coded address in RAM.	Destination register.

2	STORE	Store	Stores a the lowermost byte of a register into RAM.	Source register.	Hard-coded destination address in RAM.
3	ADD	Addition	Adds together the values stored in two registers, storing the result in register A.	First addend register.	Second addend register.
4	SUB	Subtraction	Subtracts the second operand register from the first one, storing the result in register A.	Minuend register.	Subtrahend register.
5	OUT	Output	Outputs the lowermost byte of the value stored in the second operand register to the port number stored in the first operand register. Used to access the GPU's text buffer and other external "addressable" I/O.	Register storing the port number to output to.	Register storing the value to output.
6	IN	Input	Fetches a value from the port number stored in the first operand register, storing it into the second operand register. Used to access the GPU's buffer, any data associated with interrupts (like a keyboard scancode), and any other "addressable" I/O.	Register storing the port number to get input from.	Register storing the value read from the input.
7	MOV	Move	Copy the value in one register to another.	Source register.	Destination register.
8	CMP	Compare	Compare the values in two registers. If the first is less than the second, set register A to 0. If they are equal, set register A to 1. If the first is greater than the second, set register A to 2. Used for conditional jumps. Comparing a register with itself can be used to trigger an unconditional jump.	First register to compare.	Second register to compare.
9	JMPL	Jump if less-than	If register A is zero (see the CMP instruction), then jump the instruction pointer register to the specified memory address.	Memory address to jump to.	N/A
10	JMPE	Jump if equal	If register A is 1 (see the CMP instruction), then jump the instruction pointer register to the specified memory address.	Memory address to jump to.	N/A

11	JMPG	Jump if greater-than	If register A is 2 (see the CMP instruction), then jump the instruction pointer register to the specified memory address.	Memory address to jump to.	N/A
12	RST	Soft reset	Set all registers (including the instruction pointer) back to zero, causing the program state to reset while preserving any data in RAM.	N/A	N/A
13	HALT	Halt	Stop all execution on the processor until a hard reset is performed. Used at the end of a program to prevent the CPU continuing to execute any garbage data in RAM beyond the last instruction.	N/A	N/A
14	ISR	Register Interrupt Service Routine	Set up an Interrupt Service Routine to be executed when a specific interrupt is triggered. ISRs are short functions that are automatically called when an interrupt occurs. Useful for handling keyboard scancodes from the PS/2 controller.	Register storing the interrupt number to set a handler on (8 interrupts are available in the default configuration).	Register storing the address of the first instruction of the Interrupt Service Handler.
15	INT	Trigger interrupt	Trigger an interrupt (and its associated Interrupt Service Routine) from software. Useful as a rudimentary way to do "system calls" by calling ISRs set up by the operating system. Triggering an interrupt causes all registers to be saved to a set of "shadow registers" which are restored when the ISR finishes executing (see ENDINT).	Register storing the interrupt number of the interrupt to trigger.	N/A
16	ENDINT	End interrupt	Used at the end of an Interrupt Service Routine to signal that the routine has finished and that the processor should return to the state it was in when the interrupt was triggered. Restores all values in "shadow registers" to the main registers.	N/A	N/A

17	RLOAD	Register-load	Load 32 bits from an address in RAM stored in a register. This allows accessing *calculated* addresses in RAM, rather than the fixed addresses that LOAD can access. Useful for stack variables, pointers, and anything else where the memory address is not known at compile-time.	Register storing the RAM address to load from.	Destination register to load the data into.
18	RSTORE	Register-store	Store the lowermost byte of a register to an address in RAM that is also stored in a register. This allows accessing *calculated* addresses in RAM, rather than the fixed addresses that STORE can access. Useful for stack variables, pointers, and anything else where the memory address is not known at compile-time.	Source register containing the value to be stored in RAM.	Destination register containing the address in RAM at which the value should be stored.
19	CLOAD	Constant-load	Load a constant value into a register. Not strictly necessary, but it makes writing assembly for the processor much easier and reduces the number of instructions to complete tasks like computing the offset to a stack variable.	Constant value to load.	Destination register to load into.
20	MULT	Multiply	Multiply the values in two registers together, storing the result in register A.	Register containing the first factor.	Register containing the second factor.
21	DIV	Divide	Perform an integer division of one register by another, storing the quotient in register A and the modulus in register B.	Register containing the dividend.	Register containing the divisor.
22	OR	Bitwise OR	Perform a bitwise OR operation between the values in two registers.	Register containing the first value to OR.	Register containing the second value to OR.
23	AND	Bitwise AND	Perform a bitwise AND operation between the values in two registers.	Register containing the first value to AND.	Register containing the second value to AND.
24	XOR	Bitwise exclusive-OR	Perform a bitwise XOR operation between the values in two registers.	Register containing the first value to XOR.	Register containing the second value to XOR.

25	NOT	Bitwise NOT	Flip all 32 bits in a register.	Register to flip the bits of.	N/A

The ORISC instruction set is a RISC instruction set with 26 instructions, using a LOAD/STORE architecture, meaning that only LOAD/STORE instructions can access memory. It has instructions for branching, basic arithmetic, and basic bitwise operations, as well as instructions that access special features of the processor like interrupt-driven I/O.

Each instruction is 9 bytes long, with the first 8 bits being the operation, the next 32 bits being the first operand, and the last 32 bits being the second operand. Instructions can have zero, one, or two operands, depending on the operation being performed. Each operand can be a number referring to a register (see registers table below), an address in memory, or a constant value.

The following registers are available on the CPU:

#	**Name**	**Type**	**Purpose**
0	IP	Instruction pointer	Stores the address in RAM of the first byte of the instruction currently being executed by the CPU. The IP register can be set by direct manipulation with LOAD/STORE/MOV/etc. or with JMP* instructions. It is also automatically set by the processor to perform Interrupt Service Routines.
1	A	Result register (also general-purpose)	Used to store the results of arithmetic operations and comparisons. Can also be used as a general-purpose register as long as the programmer is careful to not call an instruction that will overwrite the result value stored in the register.
2	B	General-purpose (also modulus register)	General-purpose register for any use. Also used by the DIV (divide) instruction to store the modulus of its operands.
3	C	General-purpose	General-purpose register for any use. The PL/0 compiler uses this register as an accumulator to for evaluating expressions, but this is not mandated by the CPU design.
4	D	General-purpose	General-purpose register for any use. The PL/0 compiler uses this register to store the offset for memory accesses in buffers (such as strings), but this is not mandated by the CPU design.
5	E	General-purpose	General-purpose register for any use. The PL/0 compiler uses this register to store the value being stored for memory accesses in buffers (such as strings), but this is not mandated by the CPU design.
6	F	General-purpose	General-purpose register for any use. The PL/0 compiler uses this register to store the stack pointer for the global scope, but this is not mandated by the CPU design.
7	G	General-purpose	General-purpose register for any use. The PL/0 compiler uses this register to store the stack pointer for the currently-running procedure's scope, but this is not mandated by the CPU design.

8	IE	Interrupt-enable	When set to 1, this register allows interrupts to be executed by the CPU as they arrive. When set to 0, no interrupts can execute. Useful for temporarily disabling interrupts during timing-sensitive portions of the code.
9	IR	Interrupt-running	When set to 1, this register indicates that an Interrupt Service Routine is being run. Used internally by the CPU to track interrupts.

2.2.2 Basic Toolchain

I also wrote a fairly comprehensive toolchain to allow for programming and testing the CPU.

Assembler

The assembler for the CPU converts assembly code into machine code (binaries) for the CPU. Those binaries are then fed through a ROM generator that creates a Verilog ROM file that can be synthesized into the CPU (a future version of the CPU will allow for loading programs from an SD card instead of a ROM).

Assembly programs are divided into a data section and a code section, with the first holding statically-allocated values loaded at boot-time and the second holding instructions. Variables in the data section are intended for use mainly as constants; stack variables should be allocated at run-time (see the section on the compiler).

The assembler was written in C with Flex and GNU Bison, which are utitilies for making it easier to write text parsers. It uses GLib for dynamic data structures, although most memory allocation is done manually for performance.

For certain instructions, an operand can be surrounded with square brackets to indicate that the assembler to store the statically-determined address of that operand, rather than the operand itself. This is useful for loading the memory offset to string literals stored in the data section of the binary.

An example assembly program for the CPU, with portions omitted for brevity (generated by the compiler):

```
.data:
    output_address = 0
    output_value = 0
    temp0 = 0 // General-purpose temporary storage location.
    string0 = "Hello World!"
.code:
    // Start root node.
    CLOAD 4000000,G // Start stack pointer at 4 MB, grows downwards (towards address 0).
    CLOAD 12,A
    SUB G,A
    MOV A,G
    MOV G,F
    CLOAD 4,A
    ADD G,A
    CLOAD [string0],B
    RSTORE B,A
    CLOAD 8,A
```

```
ADD G,A
CLOAD 0,B
RSTORE B,A
// Start variable assignment node.
// Start expression node.
CLOAD 0,C
CLOAD 8,A
ADD G,A
RLOAD A,D
ADD C,D
MOV A,C
CLOAD 1,D
ADD C,D
MOV A,C
// End expression node.
CLOAD 8,A
ADD G,A
RSTORE C,A
// End variable assignment node.
// Start buffer write node.
// Start expression node.
CLOAD 0,C
CLOAD 97,D
ADD C,D
MOV A,C
// End expression node.
STORE C,temp0
// Start expression node.
CLOAD 0,C
CLOAD 8,A
ADD G,A
RLOAD A,D
...
HALT
// End procedure call node.
// End root node.
```

Disassembler

To check that the assembler was generating the correct binary output, I also wrote a disassembler that converts a binary back into assembly code. The disassembler is written in plain C and produces output like this:

```
.data:
[0x000004] 0
[0x000008] 0
[0x00000c] 0
[0x000010] 1819043144
```

```
[0x000014] 1867980911
[0x000018] 560229490
[0x00001c] 150999808
.code:
[00000000] CLOAD 4000000, G
[0x000009] CLOAD 12, A
[0x000012] SUB G, A
[0x00001b] MOV A, G
[0x000024] MOV G, F
[0x00002d] CLOAD 4, A
[0x000036] ADD G, A
[0x00003f] CLOAD 16, B
[0x000048] RSTORE B, A
[0x000051] CLOAD 8, A
[0x00005a] ADD G, A
[0x000063] CLOAD 0, B
[0x00006c] RSTORE B, A
[0x000075] CLOAD 0, C
[0x00007e] CLOAD 8, A
[0x000087] ADD G, A
[0x000090] RLOAD A, D
    ...
[0x00019e] HALT
```

Emulator

To check that the CPU was executing programs in accordance with its specifications, I wrote a simple emulator in C for the CPU that reads in binaries produced by the assembler and gives the expected output of the CPU. This output can be compared with the debugging output of an IVerilog simulation of the CPU's digital logic with a simple diff to ensure that the CPU will not make any errors when running on the actual FPGA. The emulator also makes it easier to examine the internal state of a program on the CPU for debugging.

2.2.3 High-Level Toolchain: Compiler

To allow easy high-level programming of the CPU, I wrote a compiler for the PL/0 language (a stripped-down version of Pascal) that targets the ORISC instruction set and generates assembly that can be fed to the assembler to produce a runnable binary. The compiler is written in C++ using the Flex and GNU Bison utilities for parsing programs. It accepts in PL/0 such as this one, an example PL/0 program from Wikipedia that I used to test the correctness of the compiler:

```
/*******************************
Program for calculating and displaying prime numbers that uses procedures.
Based on an example from the Wikipedia article on PL/0.
*******************************/

VAR integer_to_measure = 0;
VAR integer_length = 0;
```

```
PROCEDURE get_integer_length;
BEGIN
    // Count how many digits are in the number.
    integer_length := 0;
    VAR number_shifter;
    number_shifter := integer_to_measure;
    WHILE number_shifter > 0 DO
    BEGIN
        number_shifter := number_shifter / 10;
        integer_length := integer_length + 1;
    END;
END;

VAR SCREEN_WIDTH = 80;
VAR integer = 0;
VAR line = 0;
PROCEDURE print_integer;
BEGIN
    // Get the number of digits in the integer.
    integer_to_measure := integer;
    CALL get_integer_length; // Stores return value in ‘integer_length‘.
    // Output the number one digit at a time, placing each digit at the correct position o
    VAR number_shifter;
    number_shifter := integer;
    WHILE number_shifter > 0 DO
    BEGIN
        VAR digit;
        digit := number_shifter % 10;
        output_address := line * SCREEN_WIDTH;
        output_address := output_address + integer_length - 1;
        output_value := digit + 48;
        CALL OUT;
        integer_length := integer_length - 1;
        number_shifter := number_shifter / 10;
    END;
END;

VAR arg;
VAR ret;
PROCEDURE is_prime;
BEGIN
    VAR i;
ret := 1;
i := 2;
WHILE i < arg DO
BEGIN
IF arg % i = 0 THEN
```

```
BEGIN
ret := 0;
i := arg;
END;
i := i + 1;
END;
END;

VAR max = 90;
VAR line_counter = 0;
PROCEDURE primes;
BEGIN
arg := 2;
WHILE arg < max DO
BEGIN
CALL is_prime;
IF ret = 1 THEN
        BEGIN
            integer := arg;
            line := line_counter;
            CALL print_integer;

            line_counter := line_counter + 1;
        END;
arg := arg + 1;
END;
END;

CALL primes;

CALL HALT;
```

The following features of the PL/0 language are supported by the compiler:

- Variable declarations - variables can be declared using the `VAR variable0_name = 7` syntax. Variables can be declared with no value (defaulting to an integer value of 0), with an integer value, or with a string literal.
- Variable assignments - the `:=` variable assignment operator assings a value to a variable. Variable assignments can use a value computed by an expression (see below).
  - For string variables or other buffers of bytes, square brackets can be used to assign to an offset from the start of the variable. For example, `my_string[i + 1] := 97;` would set the `i + 1`th character of the string `my_string` to the ASCII value for the character `'a'`.
- Expressions - sequences of variables and constants can be combined mathematically in expressions ranging from simple, such as `i + 1` to complex, such as `my_var0 * 7 % 5 - var2`.

- Conditionals - `IF ... THEN BEGIN ... END;` statements are used to execute a block of code only if a condition evaluates as true.
  - Conditions - a condition is two expressions with a comparison operator between them. The following comparison operators are available: `#, <, <=, >, >=` (where the pound sign means "not equal to").
- Procedures - are sequences of statements that can be called from another part of the program with a single line. Procedures have their own scope and stack frame, although they can access global variables as well.
  - Procedures are allowed to call themselves and recurse.
    - Procedures are allowed to call themselves and recurse.
      - Procedures are allowed to call themselves and recurse.
- Procedure calls - are statements that execute a procedure. See the section on the compiler's calling convention for how this works.
- While loops - repeatedly execute the same block of code as long as a condition (see above) remains true. They have the syntax `WHILE ... DO BEGIN ... END;`
  - `for` loops can be implemented using a `WHILE` loop and a counter.

Calling Convention

The compiler has full support for a variable stack, which grows downwards from address 4000000 (4 MB, halfway through the available SDRAM, which leaves another 4 MB for a future implementation of a heap).

The G register is used to store the stack pointer, which is an address in RAM at which the instruction pointer of the calling stack frame is stored. When a function exits, the value at the address stored in G is loaded into the instruction pointer register (IP) to return control flow to the calling stack frame.

Variables are allocated on the stack when a function is called by *subtracting* from G the total number of bytes taken by all variables in the function's stack frame. When the function returns, that same number is *added* back to G before G is used to find the value of IP to jump to.

Global variables have their own stack frame, which uses register F as a stack pointer register in the same manner as G. This makes it faster for procedures to access global variables by removing the need to unwind the stack.

The stack implementation is loosely based on the specification here.

2.2.4 Tools and Libraries Used

- Intel/Altera Quartus v13.0sp1 - the FPGA synthesis and simulation tools used to actually run the CPU design on the Altera DE1 board
  - Provides an SDRAM controller module, as well as clock generators using PLLs
- Icarus Verilog - another Verilog simular I used to debug the design
- CMake - C/C++ build system used to compile the toolchain for the CPU

- Flex and GNU Bison - implementations of the standard POSIX tools for writing text parsers in C and C++
- GLib 2.0 - open-source C library that provides basic memory management and dynamic data structures for the assembler

3 Links to Code and Design Files - All Placed Under Open-Source Licenses

The code and some historical data from the chicken mister and temperature logging system. *Fun fact: the BitBucket and GitHub usernames ("featherfeet") for these repositories are named after one of the chickens involved in the project.*

The code and KiCAD PCB design files for the Blinkstream LED strip controller.

The code for the toolchain, design, and specifications of the ORISC CPU architecture.

Most recent revision of the code for the RoboRAVE 2018 line-following robot. My contributions are under the aforementioned "featherfeet" account.

Active repository of the frontend and backend code for the CardsToHumanity site. My contributions are under the aforementioned "featherfeet" account.

Direct link to the running CardsToHumanity site.

4 Acknowledgements

Thank you to Suchin Ravi for teaching me Verilog, digital logic, and so much more.

Thank you to Shannon Sos (AP Computer Science) for teaching me Java.

Thank you to Joseph Friesen (Honors Principles of Engineering) for teaching me to be an engineer.