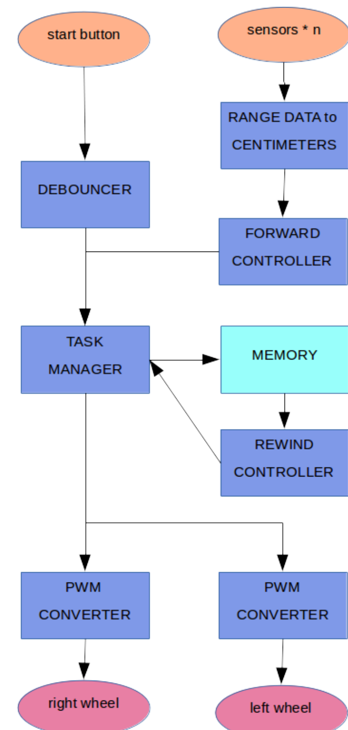
Obstacle Avoiding Mobile Vehicle

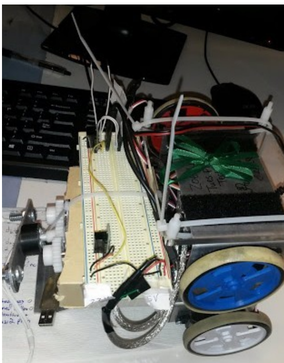
Jonathan Garcia-Mallen (jogama)



The goal was to have a robot go forward and avoid any obstacles in the way. After a certain period of time has passed, the robot would stop. Then the robot would rewind for the same amount of time. When rewinds, it returns to its start pose via the same path it used to reach its end pose. Any turns it makes on the forward trip are done in reverse for the return trip. While rewinding, it performs no obstacle avoidance. To avoid obstacles, two proximity sensors with ten centimeter range were mounted on the front.

To the right are the verilog modules planned for this project. Not all of them could be implemented in time. I explain bellow the process and challenges behind implementing each of them.

Modules used in the presentation



My presentation utilized the debounce, synchronize, forward controller, and pwm converter modules. The system had two inputs and two outputs: proximity sensors and motors. My robot was able to react to move away from obstacles in front of it. Wherever possible, I used modules made by the staff and reused my own code from previous labs.

System Inputs and outputs

Inputs were Sharp GP2Y0D810Z0F Digital Distance Sensors.

I soldered them to included pins and stuck them onto a breadboard. They were quite straightforward. They went high if they detected an object within 10cm, and were low otherwise. I tied two nexys LEDs to them for sanity checking. A third was intended for wall following, but not used.

The outputs were SM_S4303R continuous rotation servos. I ran them with a 440Hz PWM, and found their zero to be around 60% duty cycle. They're pretty common servos and inexpensive servos. But documentation is very sparse, and it would have been nice to spend more time finding their true zero.

Using the wavefunction generator was crucial for learning how the servos worked. Without this tool, I would never have gotten the servos working. For about a week I thought all my servos were fried because no wave would work. I later learned from staff that oscilloscope probes are inappropriate for the wavefunction generator. The servos likely weren't working due to the probe's high resistance. Once given banana plugs, I was able to control the motor with this tool.

While what I learned was valuable, I lost a decent amount of time servos were bad when I was just using the wrong probes. I stopped working with the motors the moment I saw the nexys drive them with a pwm. However, it would have been good to stick with the motors to further test varying the pwm duty cycle.

In general, it would have been better to work with simpler things instead of jumping into bigger things, despite my confidence in my ability to do the simpler things. See passthrough controller.

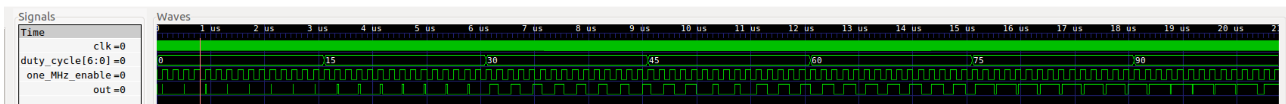
Debouncer and Synchronizer
Both of these modules were provided by the staff for lab4. The debouncer makes sure a single button press is not register red multiple times. The synchronize prevents metastability issues with switches.

PWM converter
The final pwm pipeline consisted of two modules; pass2pwm and pwm.

module pwm
Module pwm took a duty cycle and and gave a pwm wave. I tried testing the pwm module directly with the oscilloscope with unclear results. pwm requires a megaherz enable. This is most easily simulated with multiple clocks. I had difficulties with this, as the second clock did nothing. This was the final solution:

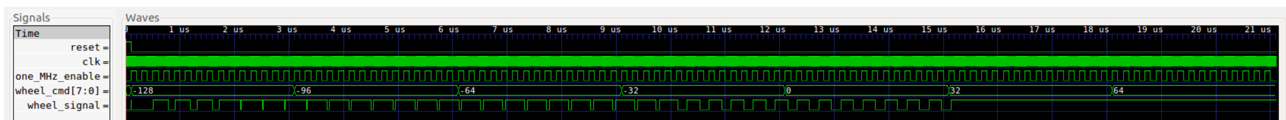
```
initial forever #1 clk = ~clk;
initial forever #100 one_MHz_enable = ~one_Mhz_enable;
```

Which allowed us the below waveform:



modules pwm_converter and pass2pwm
The pwm module receives a duty cycle in the range [0, 100], but the controllers all output wheel commands in a different range. A conversion is required. pwm_converter and pass2pwm were different approaches to this simple operation.

pwm_converter tried to convert between wheel commands plus motor zero, and the duty cycle for the pwm wave. I thought to perform the operation (duty_cycle = wheel_cmd*(50/127)+60). (I had not yet parametarized wheel command width, and the commands were always in the range [-128, 127]). This was authored in a rush. The result was overcomplicated and did not function. Difficulties came from both signage of wires and in working too hard to prevent overflow in the final duty cycle. Viewing waveforms in simulation did not provide more insight.



an easier way to prevent overflow in duty cycle. Signage was easily used in this module to "flip" one of the motors. A flipped motor reverses wheel commands. This is useful for cars, as one motor is physically flipped compared to the other.

There were still issues with pass2wm. Most glaring was that wheel speed could be varied between zero and the maximum in physical tests. Anything in between could change the direction of rotation as well as the magnitude. This remained unsolved.

The pwm_converter problems was an integration problem. This could have been avoided by putting more thought into the wires that feed into different modules. Rather than simply define conversion, defining the range of values one could receive from another. pass2pwm was a refactor; the solution

required changing other modules as well as the conversion module. The chances of making a mistake here are high.

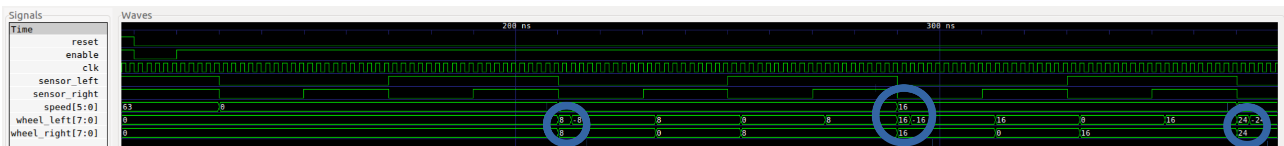
Forward controller

The forward controller took input from the proximity sensors and produced one wheel command for each motor. It seers the vehicle around obstacles. Its first iteration, still incomplete in the codebase, was designed to handle arbitrary quantities of sensors:

$$wheel_l = speed * \left(\sum_{i=0}^{\lfloor n/2 \rfloor} p_i r_i \right)$$

$$wheel_r = speed * \left(\sum_{i=\lceil n/2 \rceil}^{n} p_i r_i \right)$$

Since the car would only have two sensors for obstacle avoidance, this was abandoned in lieu of the bangbang controller. An example of bangbang control would be if you could only turn your bicycle handlebar by -45°, 0, and +45°. Bangbang control has nothing in between. Its simplicity was attractive; I quickly got it to work in simulation:



The circles highlight a glitch in the state transition. I left it, since a few nanoseconds of incorrect motor commands would not impact the controller's overall performance. Its final form was a six-state state machine to more intelligently handle both sensors going high. However, it still , I was not able to make a motor spin counterclockwise. The robot would only spin. To debug this issue, I wrote an even simpler controller.

The passthrough controller passed the speed to the pwm directly to facilitate debugging motor direction. Once it worked, this module was split into modules passthrough and pass2pwm. The latter was described above.

It would have been much better to start with the simplest possible controller by continuing to study the servo motors and pwm. This would have prevented, or at least reversed, the above progression. It is valuable to start simple, even if the simple is far away from the goal.

Modules that did not make it to the presentation

These are modules that I began implementing, but could not bring to working order in time for the presentation. These include the task_manager and the rewind controller

module rewind_controller

We used the mybram module from lab5. Left and right wheel commands were concatenated and stored in one address.

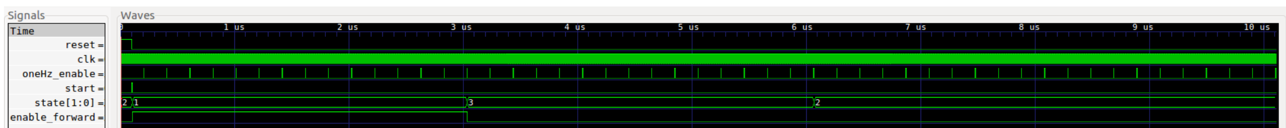
The maximum refresh rate for the sensors is 390 Hz. Using the lab4 clock divider, we made a 390Hz sampling clock. The memory address pointer was updated on this clocks positive edge. This protects against one Hertz enable signal staying high for multiple cycles.

The reversed the wheel commands are the original commands multiplied by negative one. The address pointer is decremented rather than incremented. This is another situation where signed wheel commands simplify code.

This module uses idle, forward, and reverse states. However, these states are received as input from the task manager. This simplified the rewind controller. Unfortunately, we did not get to test this

module as the task manager was taking priority. A nice way to test it would be to generate a dataset of monotonically increasing wheel commands. The unit under test should then produce monotonically decreasing wheel commands.

module task_manager
This is a simple 3-state state machine. It sent its state to the rewind controller and sent enables to other controllers. It was tested to spend fifteen seconds in the forward state, fifteen in rewind, and then return to idle to wait for a start button press. When tested on the nexys, it failed to count fifteen seconds and appeared to remain stuck in idle. It can count to fifteen in simulation, however:



IDLE = 2, FORWARD = 1, REWIND = 3

Hooking the nexus up to the oscilloscope, the one megahertz enable displayed clearly. Knowing the one hertz enable would not be visible, I connected it to the oscilloscope hoping it would measure a maximum voltage of 3.3. But the oscilloscope did not trigger. All clocks, including the one in this simulation, were made with the divider module from lab 4. If I were to continue debugging it, I would take the antitheft_fsm module nd start stripping it down. I was at a loss for how to deal with this one.

Tools I used and work not in verilog for this

inout in verilog
The two inputs and two outputs all went through the JD port on the nexys. To reconcile this in the top level nexys.v file, I declared the JD port as an inout. This, in turn, required that I uncomment some lines in the .xcd file.

Cypress lab

One Nexys 4 was lost in this project. There was some velcro separating the metal robot chassis and Nexys 4. However, it is probably that it was not enough, and that part of the Nexys still made contact with the chassis, causing a short through its capacitors.

This motivated my learning how to use a laser cutter. The Cypress lab was quite usefull, and I was impressed at the resources available there. Regardless, paranoia when a board must be close to metal is warranted.

Test benches
Initially, I used ISE to generate test benches, and would then move them into the correct project directory. After tiring of this, I macros within emacs to ease manual test bench writing.

For nearly all of my tests, I used gtkwave to visualize the simulation. This is an open source waveform viewer that can be used with verilog dumpfiles. It exists for windows, linux, and mac. It's a bit quirky, but it reliably well, and outside of lab. Still, I found it quite slow to run the sequence of commands necessary to run a simulation. I wrote the script /tests/runtest.sh. With this, running

```
$ ./runtest.sh foo_tb.v
```

from the terminal would run the simulation for the module foo. I found this to be a quicker workflow than having to add new files in vivado or ISE.

Version control
I used git and github for version control. This became crucial the day of the presentation. I made many changes to the code with the peace of mind that I could roll back if need be. When my time to present came, my most current codebase was decidedly not functional. However, I was able to roll back to a version that was working, and presented that.

In addition, git worked somewhat like a log book to me. Writing this report was made easier by looking through my git logs.