Lab 4 – DC Motor Control

Vincent “Styxx” Chang, Enrique Gutierrez, Sahil Bissessur
TA: Carrie Segal || Wednesdays, 4 PM – 7 PM

Overview

In the final lab, we work with Verilog and motors to obtain full control of a motor's speed and direction.

- In the first part of the lab, we manually move the motor to obtain inputs from the optical encoders within the motor to see if the motor is moving clockwise or counterclockwise.
- In the second part of the lab, we drive the motor with an h-bridge, which we use to change the direction it's rotating, and modulate the motor's speed with a pulse width modulator.
- In the final part of the lab, we create a C interface to make the motor move in a windshield wiper motion – sixty degrees one direction from the starting point then all the way to sixty degrees in the other direction from the starting point and back.

Methodology

For part 1 of the lab, we want to interact with the motor by manually turning it clockwise (CW) or counter-clockwise (CCW) and having it reflect its relative position from its initial spot on the screen. In our Verilog, we code in a finite state machine that changes states depending on the values of the channel A and channel B optical encoders inside of the motor. Every time the motor turns, an interrupt is sent to the board. As channel A and channel B change relative to their previous values, the motor is either turning – clockwise or counter clockwise – or not turning at all. In the cases the motor is turning, we change the value of a counter – increasing or decreasing respectively – then send the most recent spin direction of the motor to the least significant bit on the data line of the CPLD for the DSP to read into `0x10f000` – 1 and 0 respectively.

Once running, `e4.out` is able to see the direction the motor is running. It also counts the number of “ticks” the motor is away from its initial starting position; the value is negative if it's CCW from its initial position or positive if it's CW from its initial position. The interrupts per second increase the faster the motor is turned and decrease the slower it is turned.

For part 2 of the lab, we actively drive the motor with an h-bridge: the LMD18201 chip. This chip has a PWM input to vary the speed the motor is running and a direction input to change the direction the motor is running. In our Verilog, we add in another finite state machine that handles handshaking with the DSP, since we are working with addresses. To handle addresses, we have a case statement that interacts with the handshaking finite state machine. If a certain address pops up on the address bar, we take the following actions:

- `0x10cXXX` – Set the motor to run CW
- `0x10dXXX` – Set the motor to run CCW
- `0x10eXXX` – Left shift a 0 into the PWM register
- `0x10fXXX` – Left shift a 1 into the PWM register OR send the direction the motor is going to the least significant data bit.

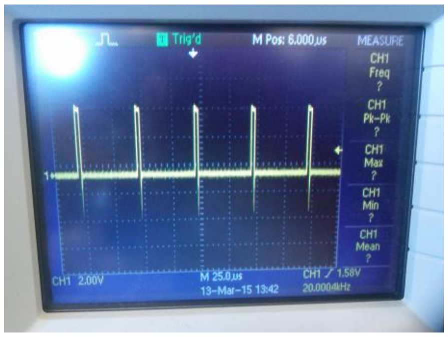
Pressing F while running e4.out writes to an address in `0x10c`, which sends a signal from the CPLD to the h-bridge's direction input to set the motor to run clockwise. Pressing B writes to an address in `0x10d` which sends the opposite signal to the h-bridge's direction input to set the motor to run counter-clockwise. While pressing P allows the user to input a value between 0 and 499 to set the duty cycle of the PWM accordingly. This accesses addresses `0x10e` and `0x10f` to left shift in 0-bits and 1-bits into the 9-bit register we have for the PWM. We then have PWM code that outputs the according signal to the PWM input on the h-bridge to accurately modify the speed at which the motor is running.

After this, we wire the CPLD to the h-bridge and the motor accordingly in order to get the motor to run as intended.

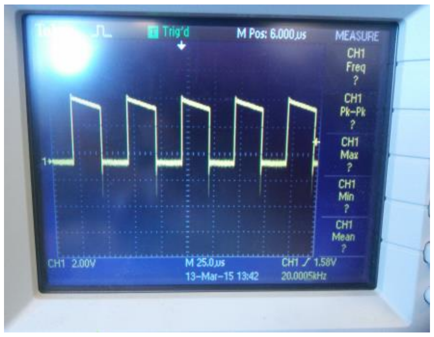
In the last part of the lab, we create a C interface that works with our Verilog code to get the motor to run in a windshield wiper motion. This C interface handles interrupts similar to `e4.out`; however, this C interface works with the Verilog to keep track of the motor's position – with assistance to the counter created in part 1. Within a specific positive and negative position – which the counter is able to check – we keep the motor running back and forth between those two positional values. This lets the motor run as if it were a windshield wiper – between two equal points in both directions.

Results

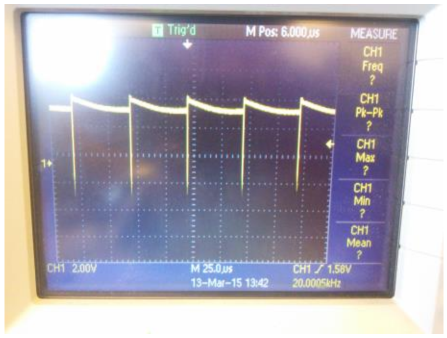
We were able to finish Part 1 and Part 2. However, our pulse width modulator never worked fully as intended: while we were able to vary the speed of the motor – and we were able to achieve all sorts of speeds, from low to high – it never directly correlated with the PWM number input into the program (i.e. a duty cycle of 499/500 was slower than a duty cycle of 50/500). Our motor was able to figure out its position accordingly, as well as change direction upon the press of a button, but the speed never directly correlated with the PWM.



Picture 1: Low duty cycle



Picture 2: 50% duty cycle



Picture 3: High duty cycle

Because we never finished fixing our PWM, we did not finish the final part of the lab to compile a C interface and make the motor work in a fashion similar to a windshield wiper, nor were we able to create a PID controller to interact with the motor.

Road Blocks

In the beginning, we definitely had problems attempting to figure out how to get the counter to work with the channel A and B inputs from the motor. While we understood we needed to implement a state machine, we did not really anticipate the need for multiple state machines running concurrently. Since we had only previously implemented at most one state machine for our labs – and handled addresses WITHIN that singular machine – the concept of using multiple ones was strange. But eventually, we figured out how to implement our code to interface correctly with the motor.