EECS 3216

Final Project Report: BLDC Motor Driver

March 30 - April 21

Table of Contents:

Abstract **3**
Analysis Correction **4**
Helper Modules **5**
debouncer.sv 5
InputDriver.sv 6
Driver.sv 7
Software Design **9**
Algorithm Explanation 9
Implementation 10
Hardware Design **17**
Previous Circuit Design 17
Improved Circuit Design 18
Components **19**
Simulations **20**
Testing **22**
Software Testing 22
Hardware Testing 24
Implemented Circuit **27**
Challenges **28**
Software Challenges 28
Hardware Challenges 28
Measurements **29**
Addressing the Damping Effect **30**
Conclusion **30**
References **31**

All design files and proposal reports can be found in this GitHub directory, and the demonstration video can be found HERE. (https://youtu.be/ajDVEePGOGo?si=b8xfoCtHZS__AAA1)

Abstract

Our project is about a driver for a three-phase BLDC motor. We will implement SVPWM (space vector PWM) to control the three phases of the motor. We will also implement an inverter with discrete components (capacitors, transistors, flyback diodes, etc) in order to demonstrate the driver working, using an RC low-pass filter and oscilloscope probes.

Our project operates on the software side, where we generate three distinct PWM signals using the SVPWM algorithm to maximize the power delivered to the load. These PWM signals would be fed to an inverter (we implemented that with discrete components), and the inverter would control a higher voltage source to deliver more power to the motor (our load) to drive it. As described before, the internal impedance of the motor behaves as a low-pass filter, so the output of the inverter is converted to a sinusoidal input to the motor.

Analysis Correction

We must address the analysis of our proposal. During testing, we noticed that the motor was moving when being driven by the system, but not completely rotating. This is because we neglected the stationary torque of the motor. Due to our approach being an open loop, the only way we could correct this was to increase the MOSFET's (switches) gate voltage to increase the current delivered. The corrected space-state equations are

(*Torque and Speed Controlling of a PMSM/BLDC Using Simulink and SOLO Blockset*, 2022).

$$\frac{d}{dt}\begin{bmatrix} i_a \\ i_b \\ i_c \\ \omega \end{bmatrix} = \begin{bmatrix} -R/L & 0 & 0 & -k_e\psi_a/L \\ 0 & -R/L & 0 & -k_e\psi_b/L \\ 0 & 0 & -R/L & -k_e\psi_c/L \\ k_e\psi_a/J & k_e\psi_b/J & k_e\psi_b/J & -B/J \end{bmatrix}\begin{bmatrix} i_a \\ i_b \\ i_c \\ \omega \end{bmatrix} + \begin{bmatrix} V_a/L \\ V_b/L \\ V_c/L \\ T_L/J \end{bmatrix}$$

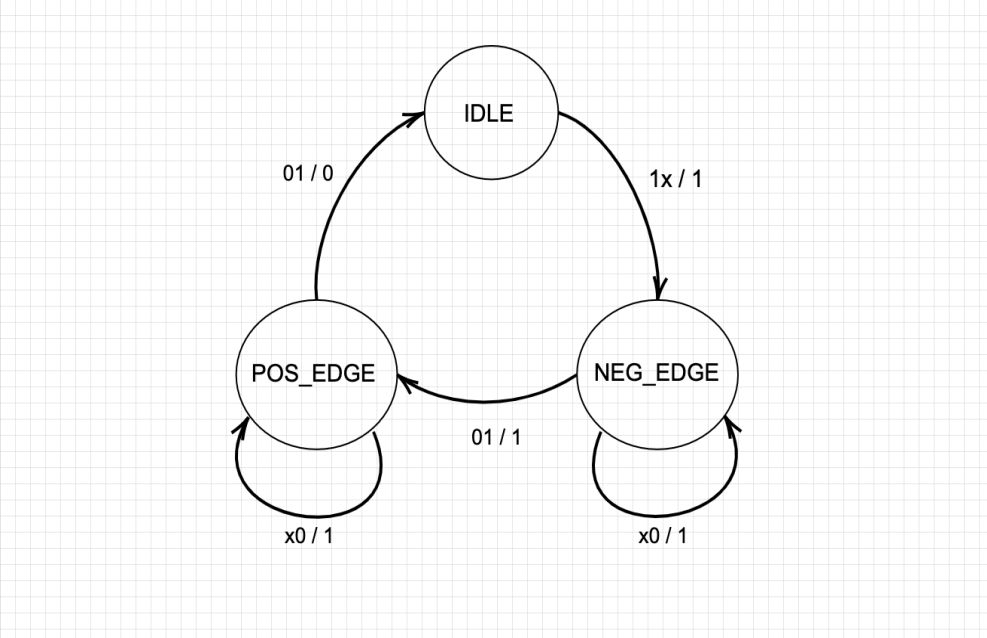
(Lee et al., 2023)

Please refer to our project proposal for the full analysis. Note that now there is an extra input dependency: the motor's load torque. We do not have the equipment to measure it, but we can remedy that in our open-loop approach by increasing the MOSFETs' gate voltage to increase the current delivered to the motor. We address the circuitry changes in the hardware design section.

Helper Modules

debouncer.sv

This module was used to debounce the input buttons Key1 and Key0, to be used as inputs in the next modules. For the following FSM, the inputs denote the button press and the completion of the debouncing period. The output is the stable signal of the button press; we denote an input that is not important as x.

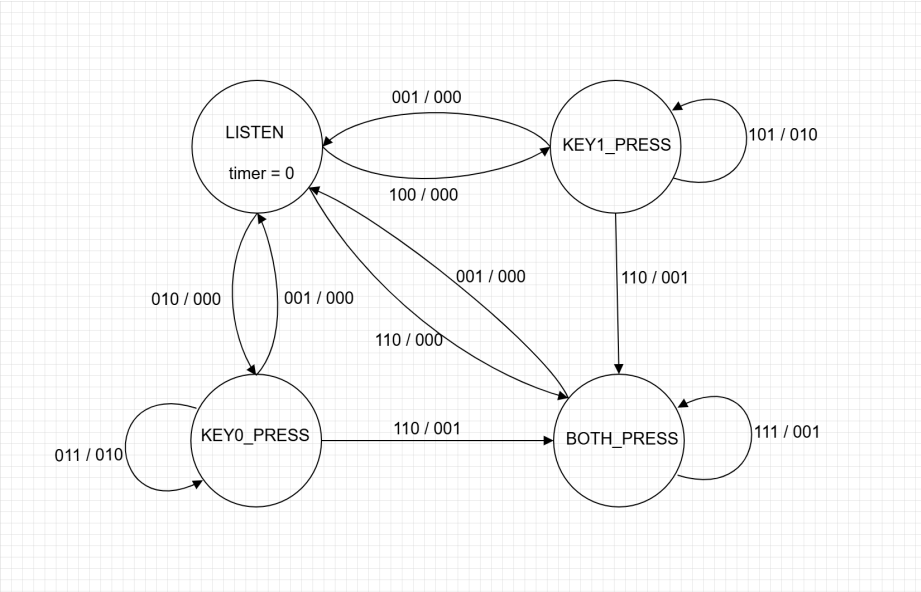


When a button is pressed, we go from IDLE to NEG_EDGE, drive the output high, and keep it in NEG_EDGE for the duration of the debouncing period. After completion of the debouncing period, and on the next (unstable) positive edge of the button, we go to POS_EDGE, and the functionality is the same. However, after the debouncing period is completed, we drive the output low and go back to IDLE.

InputDriver.sv

This module is responsible for synchronization of the button press signals (to avoid metastability) and for detecting when both buttons are pressed at the “same” time (given a synchronization window). It is composed of four states: LISTEN, KEY0_PRESSED, KEY1_PRESSED and BOTH_PRESSED. A SYNCH_DELAY is defined to synchronize the presses of the (debounced) inputs Key0 and Key1, we assume that if a user pressed Key0, and in the next SYNC_DELAY amount of time (we chose 50ms for our usecase) also presses Key1, then we assume that the user pretended to press both buttons at the same time.

Below is the Mealy FSM design for this module. For input bits, we use Key1 = third bit, Key0 = second bit, and for the first bit we define it as 1 if the internal timer counter equals the SYNC_DELAY, else if the timer is less than that we define it as 0; the timer can never be bigger than SYNC_DELAY, we ensure this in the interval logic. The output is organized as {Key1 Pressed, Key0 Pressed, Both keys Pressed} and is used in other modules.



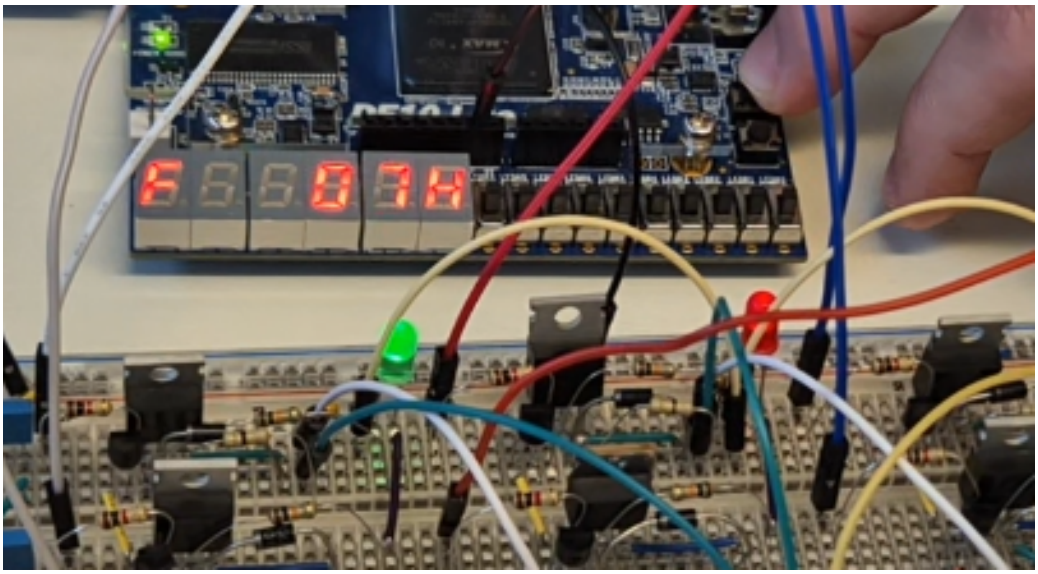
The initial state is always LISTEN, where we wait for input. Suppose the user pressed Key1, then we would go to state KEY1_PRESS and wait for the timer to equal SYNC_DELAY. If the user keeps pressing the same key until then, we stay in the current state KEY1_PRESS and update the output to 010, which represents that key1 has been pressed. Else, if before that, the user also presses key0, the FSM will transition to BOTH_PRESS, updating the output to 001, representing that both keys have been pressed. When the user releases any buttons, after some synchronization delay, the module will return to the LISTEN state, waiting for new inputs.

Driver.sv

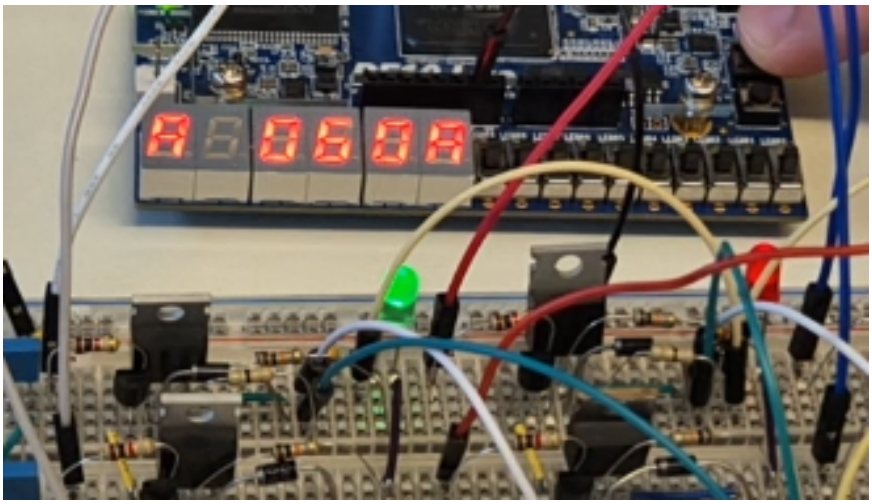
This module was used to interact with a user, get inputs, and modify the parameters of the generated signals, such as frequency, amplitude and dead time, as well as turn the signal on and off. There are three possible configurations: Frequency configuration, Amplitude configuration and Dead Time configuration, together with two input buttons Key1 and Key0.

When Key0 is pressed, the module changes the value according to the current configuration (i.e. if in Frequency configuration, it changes the signal's frequency), and Key1 is used to activate/deactivate the output signal. When both keys are pressed (for this we used the previously explained module inputDriver.sv), the module goes to the next configuration; from Frequency to Amplitude, Amplitude to Dead Time, and Dead Time back to Frequency.

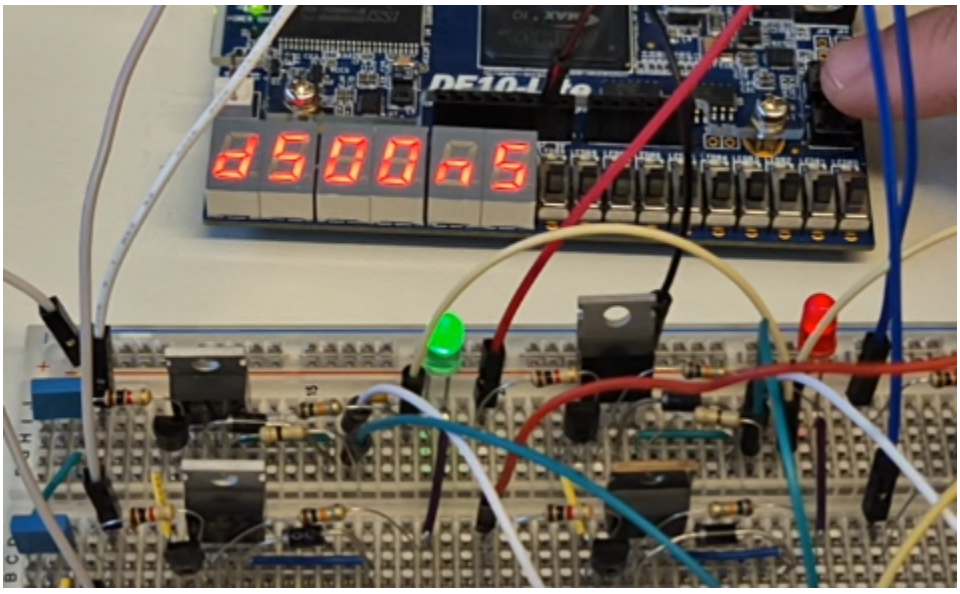
Frequency Configuration UI. Options are 3Hz, 5Hz, 6Hz, 7Hz and 8Hz.



Amplitude Configuration UI. Options are 20%, 40%, 60%, 80% and 100%.



Dead Time Configuration UI. Options are 300ns, 500ns, 700ns, 1us and 3us.

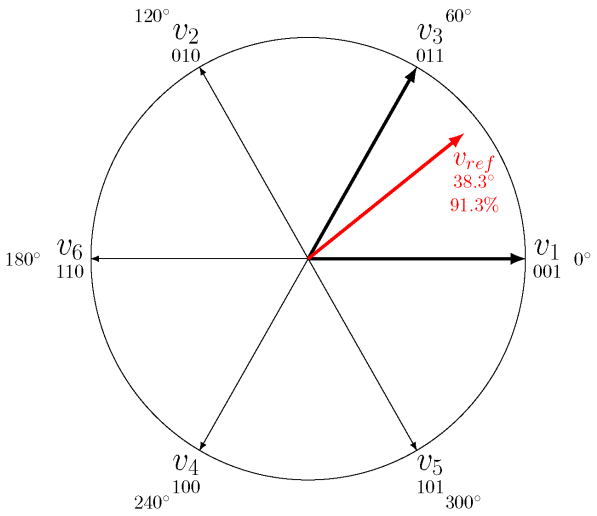


Software Design

Algorithm Explanation

We chose to implement the Space Vector Pulse Width Modulation (SVPWM) algorithm for this project. In SVPWM, instead of generating sinusoidal PWM signals offset by 120 degrees, the duty cycle and period of the outputs are controlled by a sector and a corresponding (preset) vector sequence.

Each sector corresponds to 6 sectors (of 60 degrees each) of a circle, the space vectors correspond to the codified output of the three phases (i.e. for a vector = 100, output 1 is high and outputs 2 and 3 are low).



(*Space Vector PWM Intro*, 2017)

Using a look-up table with sine wave values, we can use the SVPWM algorithm to represent any voltage vector in the above diagram (limited by the look-up table resolution)! The SVPWM algorithm was chosen (instead of SPWM) because it can maximize the power delivered to the load; SPWM can deliver 50% of power, while SVPWM, 86%.

The output of this module is labelled S; the first three LSBs (bits 0 - 2) are the top output of the inverter, and the last three MSBs (bits 5 - 3) are the reversed lower outputs of the inverter. The outputs are arranged so that the inverse of bit 0 is bit 3, bit 1 is bit 4, and bit 2 is bit 5.

Implementation

We created a module SVPWM.sv, which takes 5 inputs, 3 of which are amplitude, frequency and dead time configurations. This module was responsible for generating the 3 outputs and their inverse counterparts with dead time (so as not to short the power supply).

```
1     module SVPWM (
2             input wire clk,
3             input wire active,
4             input wire [6:0] AMPLITUDE,
5             input wire [13:0] FREQ,
6             input wire [13:0] DEAD_TIME_US,
7             output reg [5:0] S
8     );
```

We had to sample a base sine wave to implement an open-loop system. For that, we created a sine look-up table with 102 samples (the reason why 102 specifically will be explained shortly)

```
15          // Samples at 1 kHz. Normalized in units of clock cycle at 50MHz
16          localparam reg [15:0] SIN_LUT [0:SAMPLES-1] = '{
17                  16'd24999, 16'd26538, 16'd28071, 16'd29593, 16'd31097, 16'd32578, 16'd34030, 16'd35448, 16'd36826, 16'd38160, 16'd39443, 16'd40672,
18                  16'd41841, 16'd42946, 16'd43984, 16'd44949, 16'd45839, 16'd46649, 16'd47378, 16'd48021, 16'd48577, 16'd49044, 16'd49420, 16'd49703,
19                  16'd49892, 16'd49987, 16'd49987, 16'd49892, 16'd49703, 16'd49420, 16'd49044, 16'd48577, 16'd48021, 16'd47378, 16'd46649, 16'd45839,
20                  16'd44949, 16'd43984, 16'd42946, 16'd41841, 16'd40672, 16'd39443, 16'd38160, 16'd36826, 16'd35448, 16'd34030, 16'd32578, 16'd31097,
21                  16'd29593, 16'd28071, 16'd26538, 16'd24999, 16'd23460, 16'd21927, 16'd20405, 16'd18901, 16'd17420, 16'd15968, 16'd14550, 16'd13172,
22                  16'd11838, 16'd10555, 16'd9326 , 16'd8157 , 16'd7052 , 16'd6014 , 16'd5049 , 16'd4159 , 16'd3349 , 16'd2620 , 16'd1977 , 16'd1421 ,
23                  16'd954  , 16'd578  , 16'd295  , 16'd106  , 16'd11   , 16'd11   , 16'd106  , 16'd295  , 16'd578  , 16'd954  , 16'd1421 , 16'd1977 ,
24                  16'd2620 , 16'd3349 , 16'd4159 , 16'd5049 , 16'd6014 , 16'd7052 , 16'd8157 , 16'd9326 , 16'd10555, 16'd11838, 16'd13172, 16'd14550,
25                  16'd15968, 16'd17420, 16'd18901, 16'd20405, 16'd21927, 16'd23460
26          };
```

There were two FSMs in the SVPWM module, one FSM for each sector and another for each vector.

```
28              typedef enum logic [2:0] { S1, S2, S3, S4, S5, S6 } state_sector;
29
30              typedef enum logic [2:0] {
31                      V0 = 3'b000, V1 = 3'b100, V2 = 3'b110, V3 = 3'b010,
32                      V4 = 3'b011, V5 = 3'b001, V6 = 3'b101, V7 = 3'b111
33              } state_vector;
```

We defined a look-up table for the vector sequence for each sector. Each sequence was chosen, so we only changed 1 bit on each vector transition. This was needed to minimize switching losses on the MOSFETs and BJTs transistors.

```
35          localparam state_vector SECTOR_VECTORS_LUT [0:NUMBER_STATES-1][0:NUMBER_VECTORS-1] = '{
36                  '{V0, V1, V2, V7, V2, V1, V0}, '{V0, V3, V2, V7, V2, V3, V0},
37                  '{V0, V3, V4, V7, V4, V3, V0}, '{V0, V5, V4, V7, V4, V5, V0},
38                  '{V0, V5, V6, V7, V6, V5, V0}, '{V0, V1, V6, V7, V6, V1, V0}
39          };
```

Next is how we calculated each parameter. As mentioned in our proposal, the time allocated to each vector in a sequence goes as:

1. $T_1 = \frac{T_s}{\sqrt{3}} sin(\theta - \pi/3)$
2. $T_2 = \frac{T_s}{\sqrt{3}} sin(\theta)$
3. $T_0 = T_s - (T_1 + T_2)$

We implemented these values in the software as follows.

```
58          // SVPWM parameters calculations
59          reg [6:0] TIME = 0;
60          reg [17:0] AMP;
61          reg [32:0] T0, T1, T2;
62
63          // Normalizing by 100 and Dividing by sqrt{3}
64          // Note 2^19 */(100*sqrt{3}) = 0xBD2
65          assign AMP = (AMPLITUDE * SAMPLE_PERIOD * 40'hBD2) >> 19;
66
67          // Normalizing by max SIN_LUT value = 49999
68          assign T1 = AMP * SIN_LUT[TIME % SAMPLES] / 49999;
69          assign T2 = AMP * SIN_LUT[(TIME + (SAMPLES * 120 / 360)) % SAMPLES] / 49999;
70          assign T0 = SAMPLE_PERIOD - (T1 + T2);
```

Note that we defined the amplitude $AMP = \frac{T_s}{\sqrt{3}}$ where T_s is the Sample Period. To compute that in software, note that $2^{19} \cdot \frac{1}{100\sqrt{3}} = 0xBD2$ (we divided by 100 to account for the variable amplitude) we multiply the input 'AMPLITUDE' (which can be 20, 40, 60, 80 or 100) by this quantity and shift back by 19 (19 was chosen to get enough precision) to compute AMP. This was needed since the DE10-Lite board does not support floating-point operations. Also note that we shift the argument of 'T2' forwards by SAMPLES/6 since we cannot shift T1 back by the same amount; over time, this has the same effect as the original formulas.

Next, we'll go over how we implemented the state update. The following code is inside an 'always_comb' block. This block is called whenever the corresponding counters are incremented.

```
76                  case(CURR_SEC)
77                          S1: begin
78                                  NEXT_SEC = (sector_counter < SECTOR_PERIOD)? S1: S2;
79                                  NEXT_VEC = SECTOR_VECTORS_LUT[0][vector_index];
80                                  NEXT_PREDICT_VEC = SECTOR_VECTORS_LUT[0][(vector_index == 6)? 0 : vector_index+1];
81                          end
82                          S2: begin
83                                  NEXT_SEC = (sector_counter < SECTOR_PERIOD)? S2: S3;
84                                  NEXT_VEC = SECTOR_VECTORS_LUT[1][vector_index];
85                                  NEXT_PREDICT_VEC = SECTOR_VECTORS_LUT[1][(vector_index == 6)? 0 : vector_index+1];
86                          end
87                          S3: begin
88                                  NEXT_SEC = (sector_counter < SECTOR_PERIOD)? S3: S4;
89                                  NEXT_VEC = SECTOR_VECTORS_LUT[2][vector_index];
90                                  NEXT_PREDICT_VEC = SECTOR_VECTORS_LUT[2][(vector_index == 6)? 0 : vector_index+1];
91                          end
92                          S4: begin
93                                  NEXT_SEC = (sector_counter < SECTOR_PERIOD)? S4: S5;
94                                  NEXT_VEC = SECTOR_VECTORS_LUT[3][vector_index];
95                                  NEXT_PREDICT_VEC = SECTOR_VECTORS_LUT[3][(vector_index == 6)? 0 : vector_index+1];
96                          end
97                          S5: begin
98                                  NEXT_SEC = (sector_counter < SECTOR_PERIOD)? S5: S6;
99                                  NEXT_VEC = SECTOR_VECTORS_LUT[4][vector_index];
100                                 NEXT_PREDICT_VEC = SECTOR_VECTORS_LUT[4][(vector_index == 6)? 0 : vector_index+1];
101                         end
102                         S6: begin
103                                 NEXT_SEC = (sector_counter < SECTOR_PERIOD)? S6: S1;
104                                 NEXT_VEC = SECTOR_VECTORS_LUT[5][vector_index];
105                                 NEXT_PREDICT_VEC = SECTOR_VECTORS_LUT[5][(vector_index == 6)? 0 : vector_index+1];
106                         end
107                 endcase
```

We can see that the update of ‘NEXT_SEC’ and ‘NEXT_VEC’ is based on a straightforward type II state machine. However, it is not clear for what purpose we use ‘NEXT_PREDICT_VEC’ (it is used for the dead time implementation). This will be explained shortly. Next, we have the top output of the inverter, which is defined as

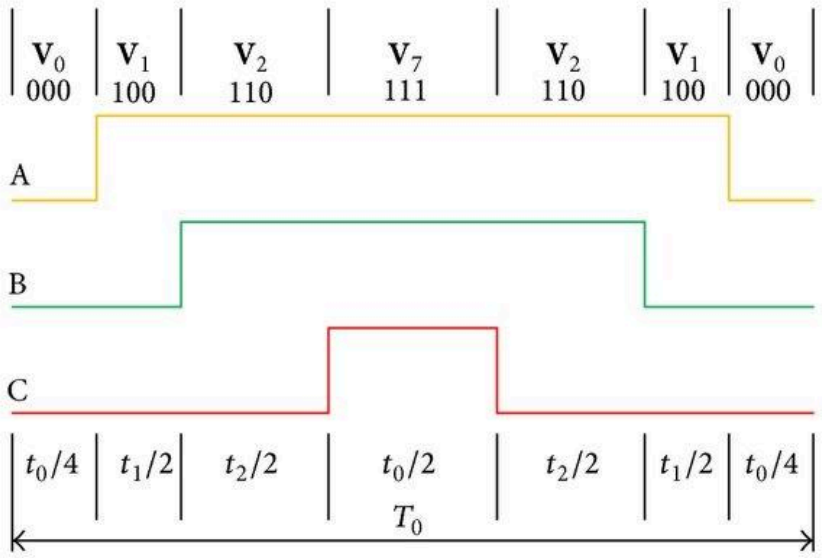
```
109                     // Inverting (top) Output
110                     // If not active (on an emergency stop for example) hard set top values to 0
111
112                     // CHANGE THIS TO 1 WHEN IMPLEMENTING NOT GATES!
113                     S[2:0] = active? ~CURR_VEC : 3'b111;
114             end
```

If we are in the active state (‘active’ is an input to the module), the top output is the inverted current vector; we inverted the logic after implementing a voltage shifter circuit, which will be covered in the hardware section.

We also had to update the period by which a vector is active, and this period must be variable to implement the SVPWM algorithm. The following code shows how we implemented it.

```
116             // Period Update
117             always_comb begin
118                     case(vector_index)
119                             0: VECTOR_PERIOD = T0 / 4;
120                             1: VECTOR_PERIOD = T1 / 2;
121                             2: VECTOR_PERIOD = T2 / 2;
122                             3: VECTOR_PERIOD = T0 / 2;
123                             4: VECTOR_PERIOD = T2 / 2;
124                             5: VECTOR_PERIOD = T1 / 2;
125                             6: VECTOR_PERIOD = T0 / 4;
126                             default: VECTOR_PERIOD = T0 / 4;
127                     endcase
128             end
```

In the above code, we allocated the first, middle and last vectors as zero vectors, which means that no current will be flowing in during the time allocated for T0. We can also see that the second, third, fifth and sixth vectors are active. This time allocation would allow us to generate the following voltage diagram (for the first sector, S1).



(*Switching Sequence of Traditional SVPWM.*, n.d.)

A similar voltage pattern would be implemented for each sector, however, the voltage distribution would change.

Next, we'll cover the clock block ('always_ff(@posedge clk)'). If the user presses the (debounced) reset button on its negative edge, we immediately deactivate the lower outputs of the inverter (bits 3- 5) by writing 1s to them (due to the inverted logic); this would turn off the whole operation. Else, while we are active, we keep incrementing the counters and other parameters normally.

We reset parameters when transitioning between samples and sectors. This answers why we chose an oddly specific number of samples at the beginning of the report (102 sine lut samples). Since we have 6 sectors, we'd like to go through an equal number of samples per sector. We noticed that having a total number of samples that was not a divisor of 6 introduced desynchronized behaviour to the waveforms; specifically, on vector transitions, the waves were losing synchronization. We fixed this by having a total number of samples divisible by 6; this way, we could go through an equal number of samples per sector and keep the synchronization.

```
130             // States Transitions
131             always_ff @(posedge clk or negedge active) begin
132                     if(~active) begin
133                             // If not active (on an emergency stop for example) hard set bottom values to 0
134
135                             // CHANGE THIS TO 1 WHEN IMPLEMENTING NOT GATES!
136                             S[5:3] <= 3'b111;
137                     end else begin
138                             sector_counter <= (sector_counter < SECTOR_PERIOD)? sector_counter + 1: 0;
139
140                             // Forcefully reset parameters
141                             // Output may be out of sync if frequency is not a divisor of system clock (50MHz)
142                             if((sample_counter >= SAMPLE_PERIOD) | (sector_counter >= SECTOR_PERIOD)) begin
143                                     vector_counter <= 0;
144                                     vector_index <= 0;
145                                     sample_counter <= 0;
146                                     TIME <= (TIME + 1) % SAMPLES;
```

Otherwise, we update the parameters normally. We increment ‘vector_counter’ until it reaches ‘VECTOR_PERIOD’, and we only increment the vector_index when we finish a vector

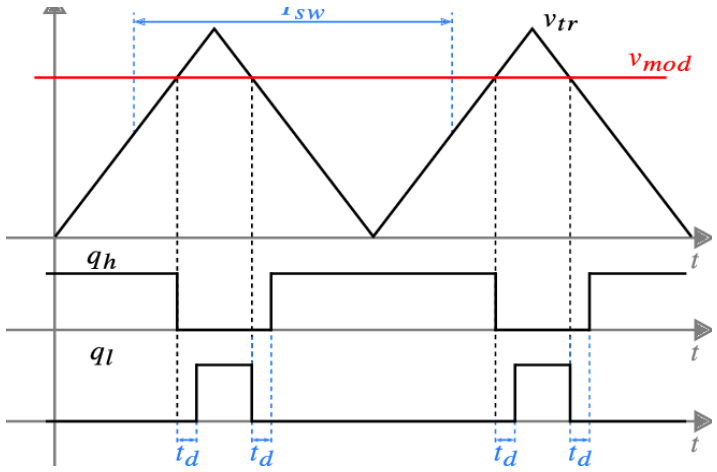
```
148                     // Incrementing parameters
149                     end else begin
150                             vector_counter <= (vector_counter < VECTOR_PERIOD)? vector_counter + 1: 0;
151                             sample_counter <= sample_counter + 1;
152
153                             // Only increment vector_index at the end of the vector period
154                             if(vector_counter >= VECTOR_PERIOD) vector_index <= (vector_index + 1) % (NUMBER_VECTORS);
155                     end
```

period and go to the next one.

The next section is the implementation of dead time (which was by far the most difficult). Dead time is the period in which both the top and bottom outputs of the inverter are off so that we may avoid shorting the power supply. To implement this, we must know what the next vector is in the sequence. If the next bit is a 1, and the current signal is also a 1, this signal must be flipped to a 0 **before the vector transition** so as not to short the power supply.

```
157                     // Dead Time Implementation for Non-Inverting (bottom) Output
158                     // Fist output is undefined, implement a pull down resistor.
159                     if(vector_counter >= (VECTOR_PERIOD - (DEAD_TIME_PERIOD))) begin
160                             // Initially both signals (top and bottom) are 0, and suppose the top output
161                             // goes to 1 at the beggining of a vector transition. The bottom signal waits for
162                             // DEAT_TIME_PERIOD / 2 at the end of the cycle and makes a prediction of what the next vector will be
163                             // (really we know what it is since the vector sequence is fixed).
164                             // If the prediction is correct we assign the vector value to the bottom output
165                             // (since the top output is inverted) else we set it to 0 for safety.
166
167                             S[3] <= (NEXT_PREDICT_VEC[0] == CURR_VEC[0])? NEXT_PREDICT_VEC[0]: 0;
168                             S[4] <= (NEXT_PREDICT_VEC[1] == CURR_VEC[1])? NEXT_PREDICT_VEC[1]: 0;
169                             S[5] <= (NEXT_PREDICT_VEC[2] == CURR_VEC[2])? NEXT_PREDICT_VEC[2]: 0;
170                     end else if(vector_counter >= DEAD_TIME_PERIOD) begin
171                             // After the initial DEAD_TIME_PERIOD we set the bottom output to the current vector value
172                             // (since top output is inverted).
173                             S[5:3] <= CURR_VEC;
174                     end
```

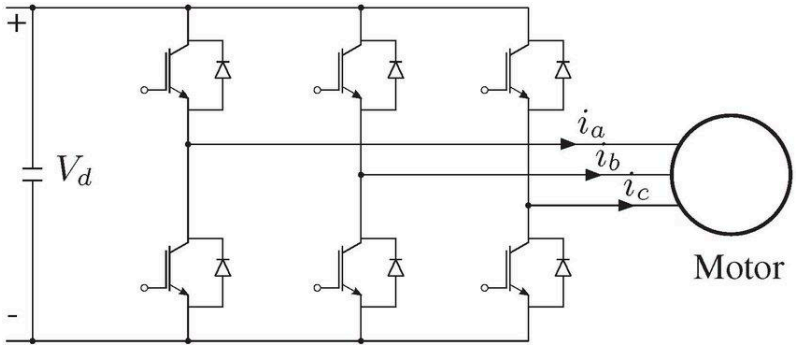
There are two important moments for the dead time signals. When the counter reaches ‘VECTOR_PERIOD - DEAD_TIME_PERIOD’, this is the end of the previous cycle, where we must ‘predict’ what the next output is going to be. If the prediction is equal to the current value of the signal, we make no changes, we reset to 0. The other important moment is the the counter reaches ‘DEADT_TIME_PERIOD’ in the beginning of the cycle, in which the dead time has completed, so we can invert the current value of the top signal and write it to the lower signal (since the output is inverted, the lower signals equals the bits of the current vector). The following figure best demonstrates the logic.



(*Fig. 6. Dead-Time Generation on the Inverter Control Signals by the PWM...*, n.d.)

Hardware Design

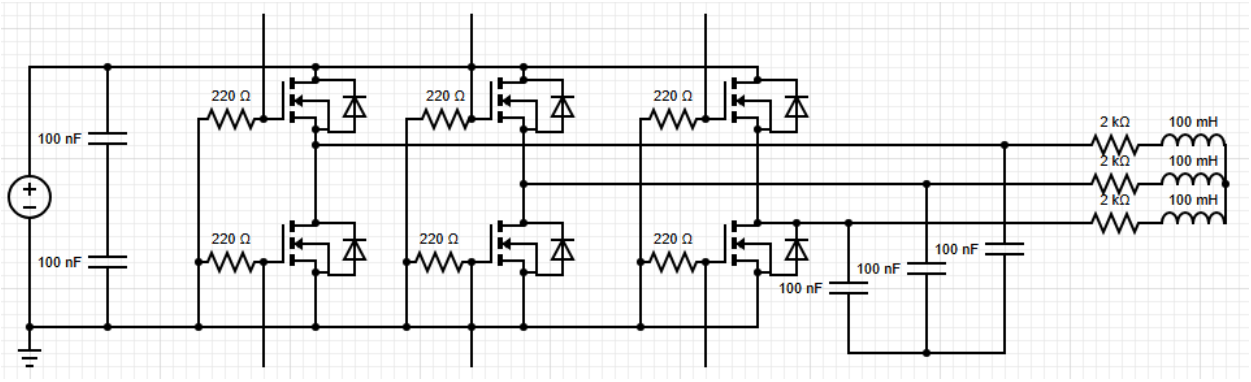
The hardware design was inspired by common inverter topologies, such as this one.



(*Fig. 2. Three Phase Inverter Topology*, n.d.)

We chose the IRF520 MOSFET to act as switches (in the saturation region) since the DE10-Lite can't provide enough current to operate BJTs. MOSFETs can handle higher currents, which are needed to drive the motor. The following was our initial circuit design.

Previous Circuit Design

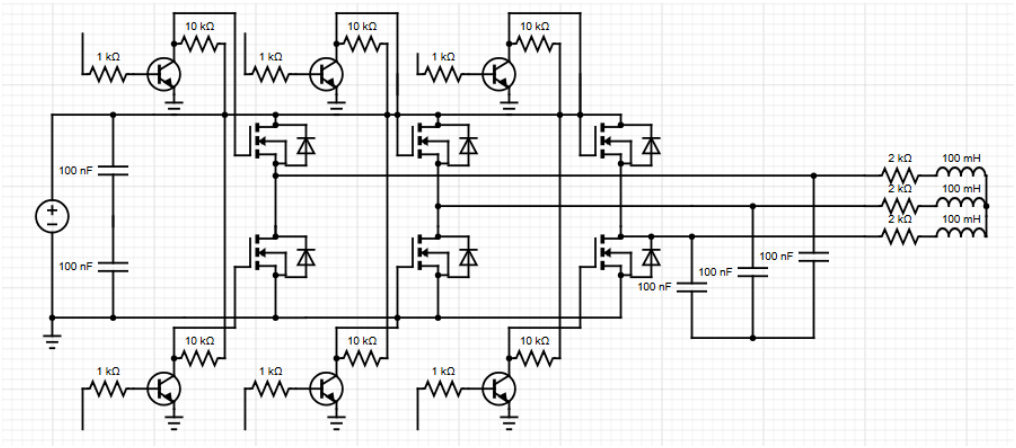


The hanging wires would be directly connected to the outputs of the DE10-Lite with the SVPWM software loaded. We implemented pull-down resistors at the gate of each MOSFET to ensure that all components would be initially off, and the three-phase resistors with inductors connection simulate the impedance of the motor, the diodes across the MOSFETs are to ensure that the back EMF of the motor do not damage the MOSFETs; even though they already have

built in reverse diodes, we did not know what the magnitude of the back EMF could be, so we added the diodes for redundancy. We also implemented the capacitors connected in a star configuration; those were mostly used for debugging and visualizing the raw output waves.

Improved Circuit Design

The circuitry mostly worked, however, we noticed that we were not getting enough current through the motor, and as a result, the motor did not fully move. This was the case because the voltage logic level of the DE10-Lite (3.3V) was too low to turn on the MOSFETs completely. For this reason, we decided to add extra circuitry to the MOSFETs gate input to shift up the voltage. Follows the improved (and final) circuitry.



The new circuitry design implements discrete NOT gates on the MOSFET inputs (hence why we reversed the logic on the software); now, there's no need for pull-down resistors since the BJTs internal diodes already act as pull-downs. This way, we can shift the logic voltage of the DE10-Lite from 3.3V to 5V, fully turning on the MOSFETs. We couldn't reuse any of the top BJTs due to the dead time implementation, else we could've reduced the amount of components used. The hanging wire would be connected to the output of the SVPWM module.

This new circuitry does technically consume more power due to the not gates being on when the system is off. However, the choice of 10kOhms collector and 1KOhms base resistors was done intentionally to make sure the BJTs remain in the saturation region with a collector current of 500uA (assuming a source of 5V for the BJTs), which implies a total current consumption of 3mA when the system is off.

Components

For the inverter switches, we used the IRF520 MOSFETs (*Irf520.Pdf*, n.d.), which have an on resistance of about 0.27 Ohms and a gate voltage threshold between 2V - 4V (this is why we had to introduce extra circuitry to shift the logic voltage level of the DE10-Lite of 3.3V) the on resistance seems to be high, but on testing we noticed the MOSFETs were not heating up at all. However, the IRF520 is commonly used for high-frequency switching applications, such as for this project.

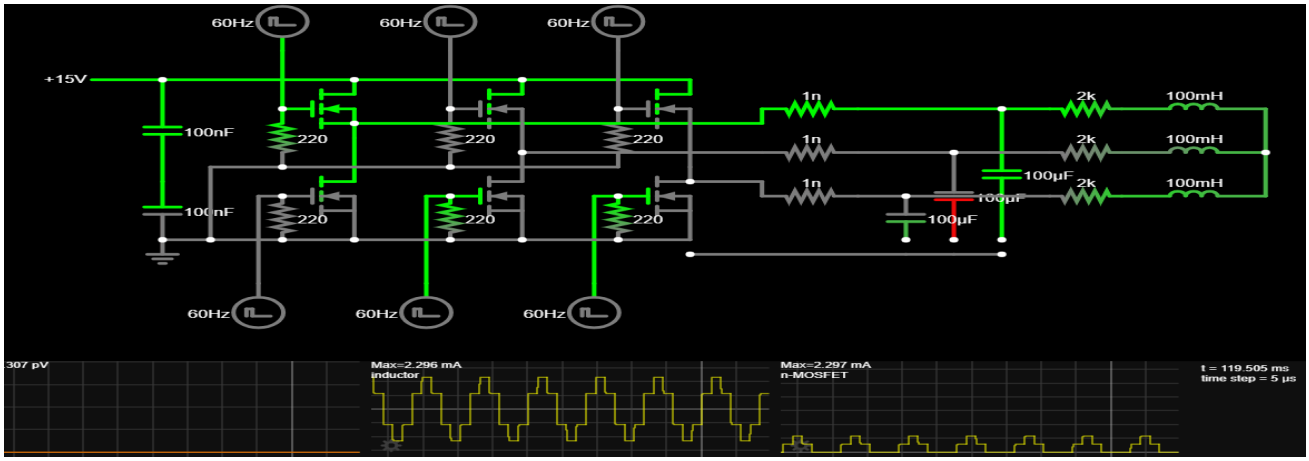
For the transistor used in the discrete not-gates, we used a combination of BC547B (alldatasheet.com, n.d.) with a beta value of around 290 (which is widely used for switching applications) and the 2A2222A-1726 (*P2N2222A - Amplifier Transistors NPN Silicon*, n.d.) with a beta value of around 200; the 2A2222 is mostly used for amplification applications, but tests showed if would suffice for our purposes.

For the motor load, we used the CrocSee Micro 3 AC mini BLDC generator (*CrocSee Micro 3 Phase AC Mini Hand Brushless Motor Generator Model Science Experiment Teaching Aid Kit DIY, Science - Amazon Canada*, n.d.). It is listed as a generator, but our tests showed that it can also be used as a 3-phase motor.

No specifications of the motor were given by the vendor. We measured the DC impedance across two legs to be around 50 ohms, which implies the DC resistance of each leg must be about 25 Ohms.

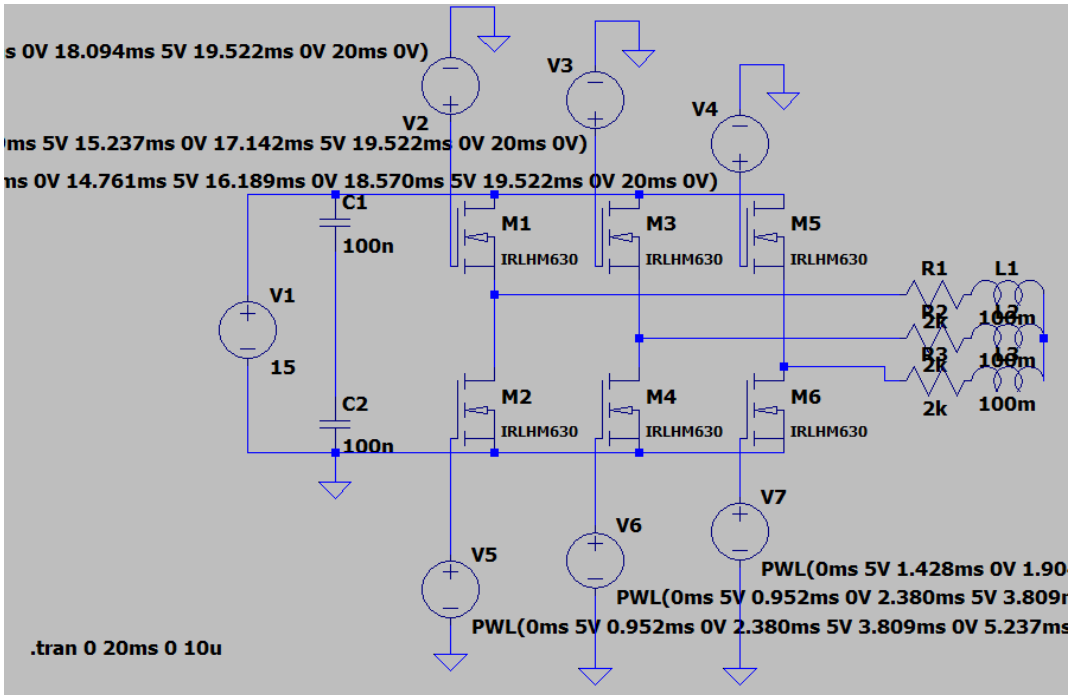
Simulations

Follows a hardware simulation on the online simulator falstad.com.

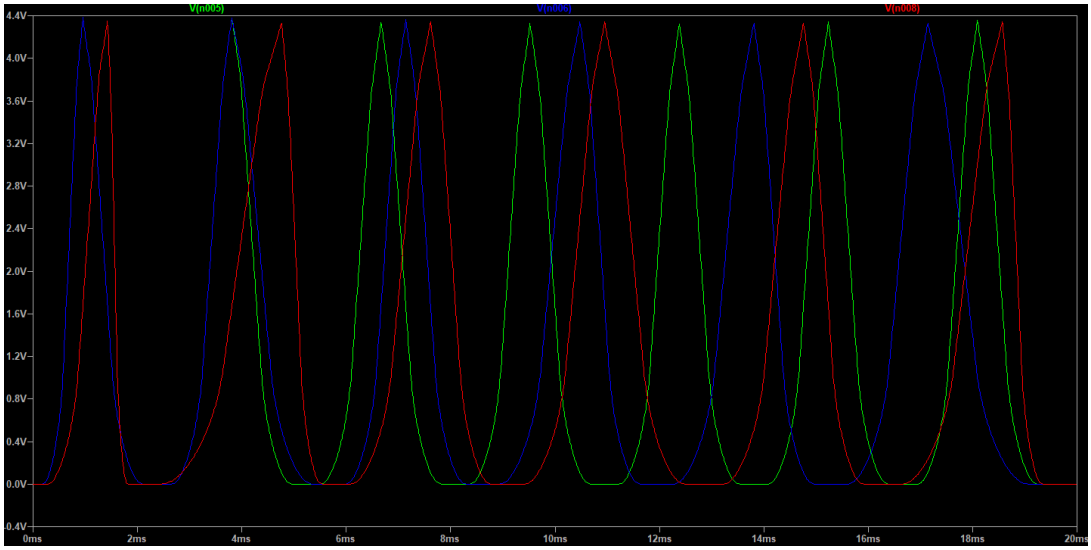


(https://tinyurl.com/26qqmcq9 n.d.)

An LTspice simulation was also made. We couldn't find the exact type of MOSFET used in our circuit; therefore, one with similar parameters was chosen. Follows the LTspice circuit and wave output.



LTspice circuit



LTspice simulation output voltage

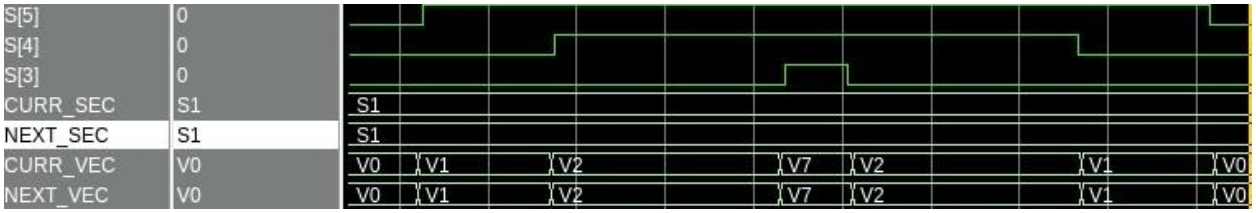
Testing

Several tests were performed on both our software and hardware throughout the semester to make sure all components were working correctly.

Software Testing

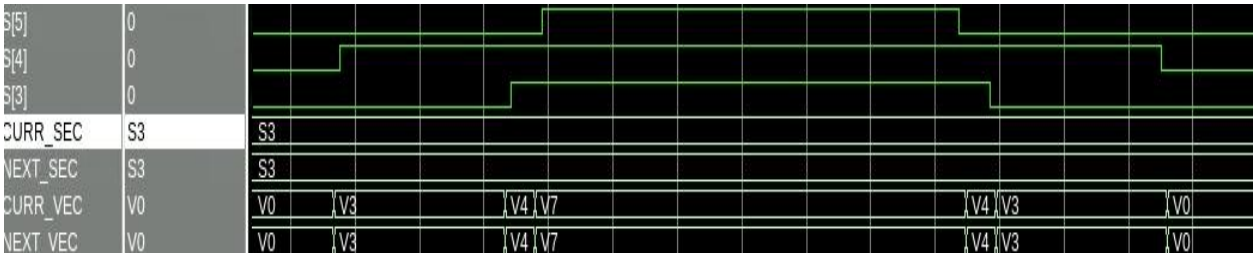
For the software tests, we used RTL simulation and GTK Wave to visualize our signals and make sure we were implementing the SVPWM algorithm correctly. Follow the screenshots of the outputs for each sector.

Sector 1



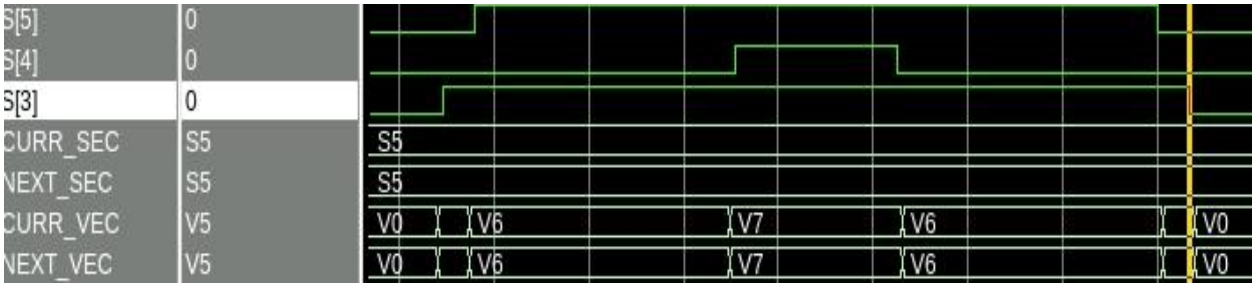
Sector 2

Sector 3

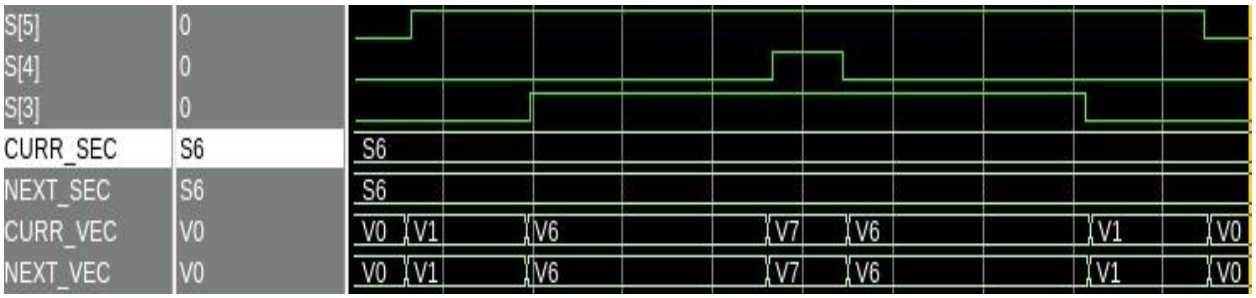


Sector 4

Sector 5



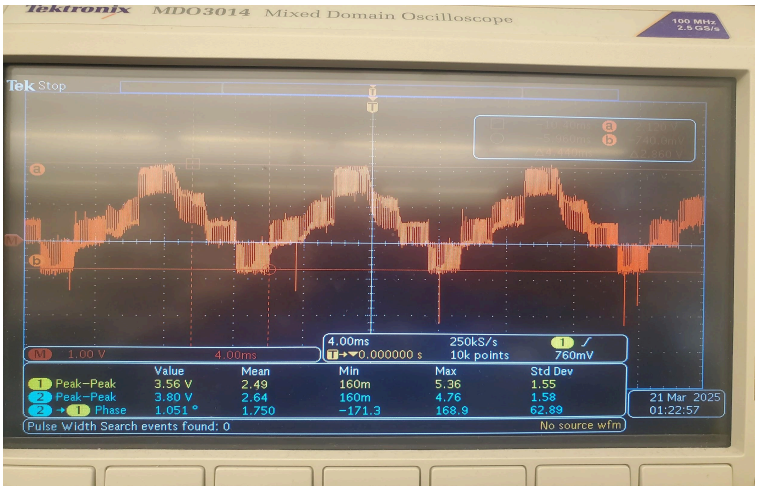
Sector 6

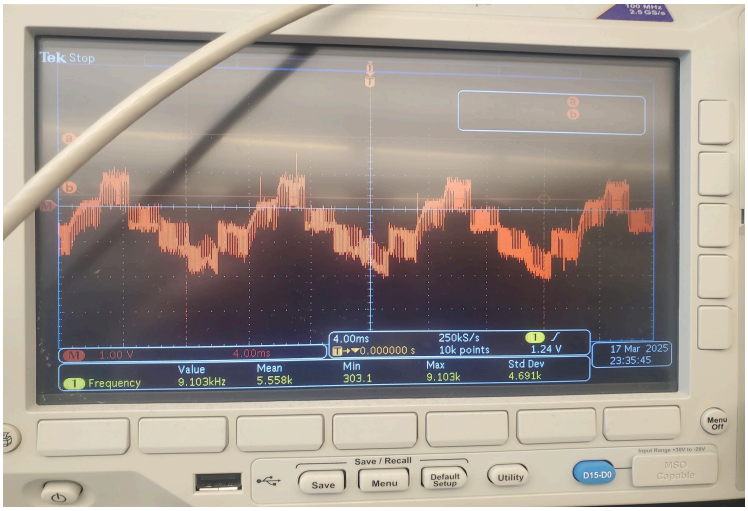


We see that all the vector sequences and signals are correct for each sector! Note that the previous waves were the reversed outputs (bits 3 - 5) for better visualization.

Hardware Testing

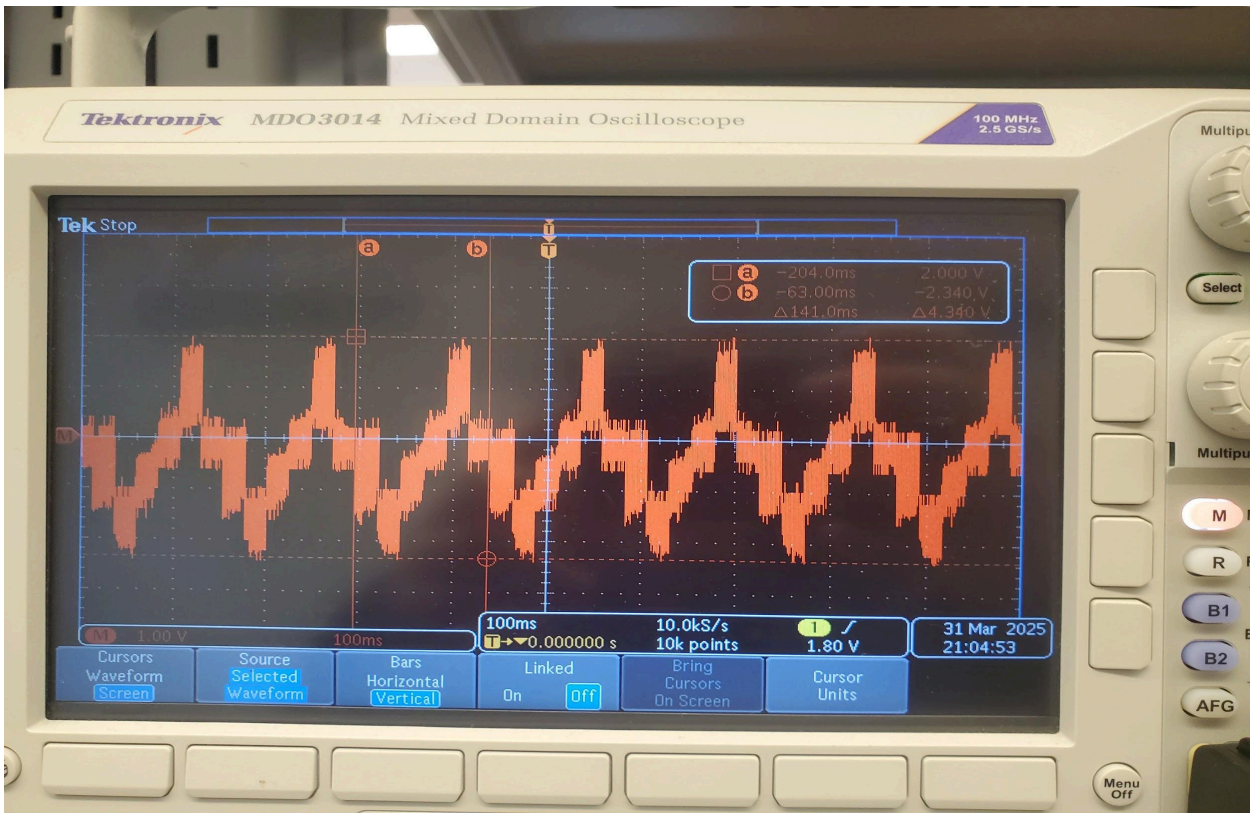
We used the Oscilloscopes available on the Lassonde 1006 labs to visualize the waveforms and validate our outputs. However, due to the limitations of the oscilloscopes (only one math channel available), we were only able to show one wave at a time; we could not show two waves 120 degrees offset. However, the oscilloscope's signal validated the outputs that we expected. Most signals are at 3Hz unless otherwise specified.

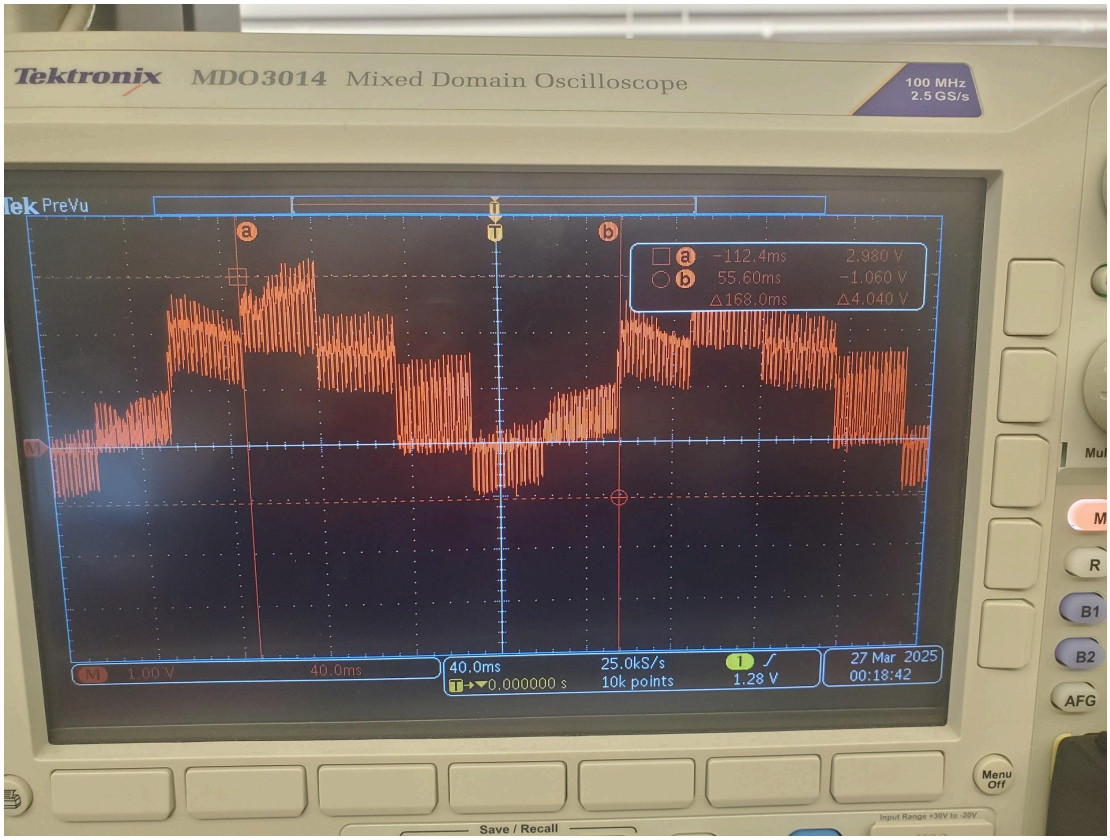


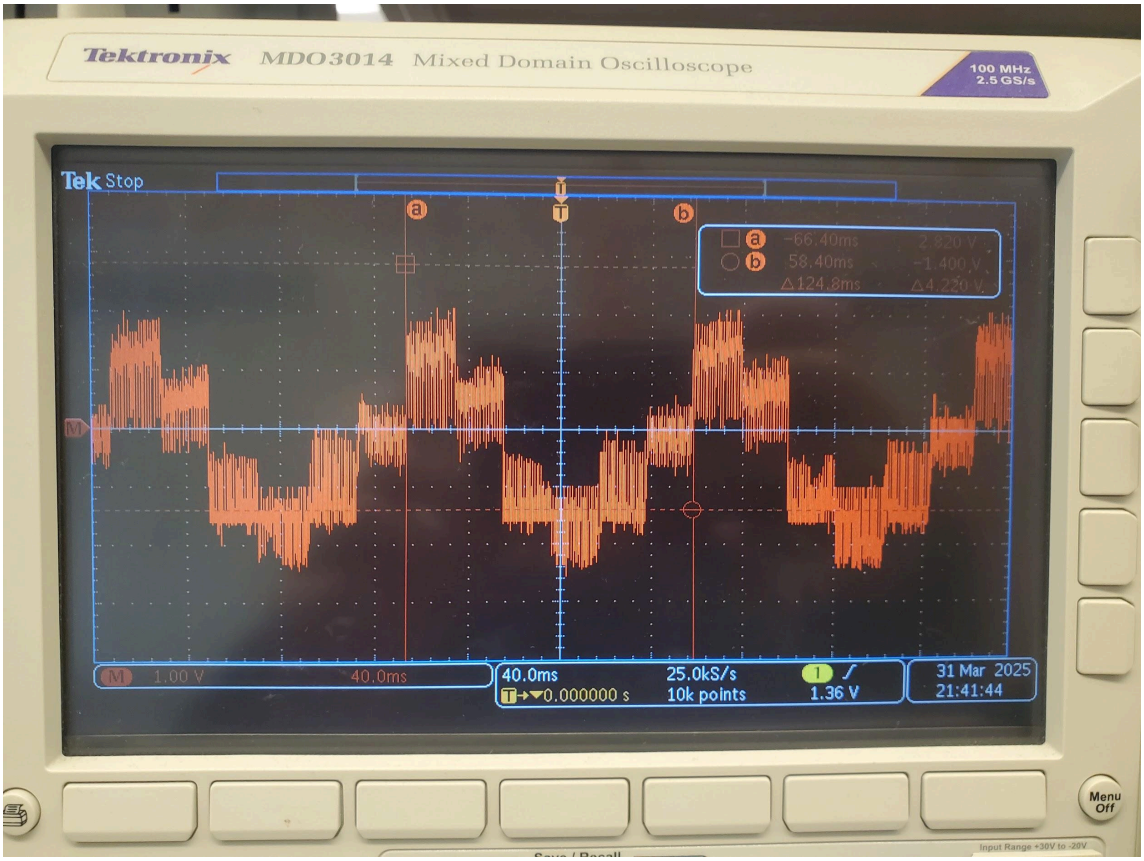


Note that the waveforms are supposed to be rectangular; this is normal for inverter outputs. When the output is connected through a motor, then the high-frequency digital waveforms are filtered out, and the signals become sinusoidal.


Run
Auto
a
b
2.00 V
100ms
100ms
10.0kS/s
−1.400000ms
10k points
7.40 V
Value
Mean
Min
Max
Std Dev
1 Frequency
831.9 Hz
Low resolution
17 Apr 2025
20:06:39
Cursors
Waveform
Screen
Source
Selected
Waveform
Bars
Horizontal
Vertical
Linked
On
Off
Bring
Cursors
On Screen
Cursor
Units


Tektronix
MDO3014
Mixed Domain Oscilloscope
100 MHz
2.5 GS/s
Tek Stop
a
b
100ms
0.000000 s
10.0kS/s
10k points
1.80 V
31 Mar 2025
21:04:53
Cursors
Waveform
Screen
Source
Selected
Waveform
Bars
Horizontal
Vertical
Linked
On
Off
Bring
Cursors
On Screen
Cursor
Units
Select
M
R
B1
B2
AFG
Menu
Off

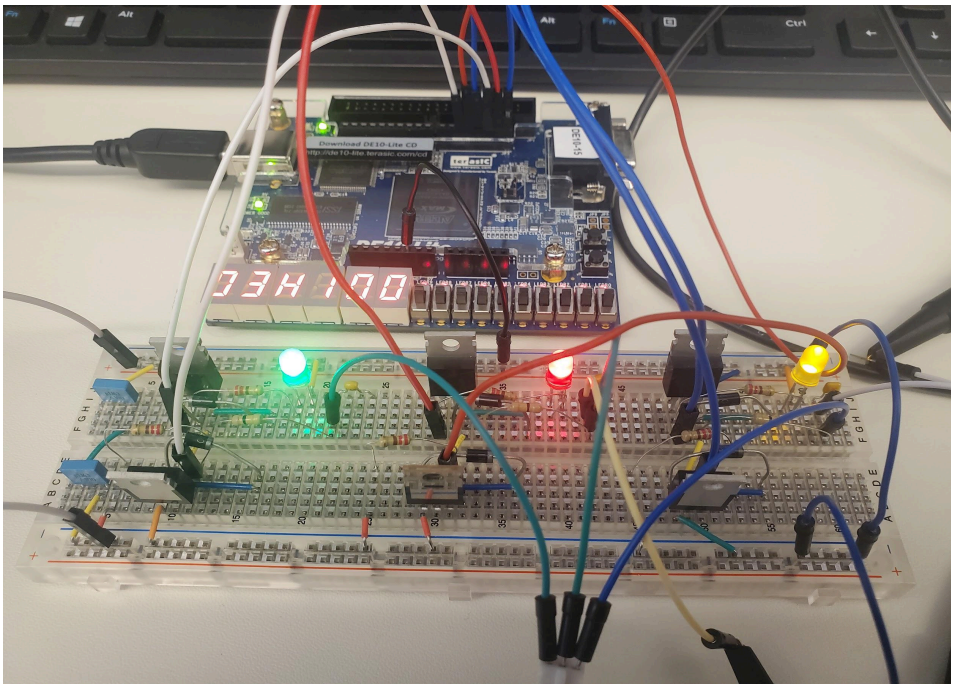

Tektronix MDO3014 Mixed Domain Oscilloscope
100 MHz
2.5 GS/s
Tek PreVu
a -112.4ms 2.980 V
b 55.60ms -1.060 V
Δ168.0ms Δ4.040 V
M 1.00 V 40.0ms
40.0ms 25.0kS/s
T→▼0.000000 s 10k points
1 1.28 V
27 Mar 2025
00:18:42
Menu Off
Save / Recall


Tektronix MDO3014 Mixed Domain Oscilloscope
100 MHz
2.5 GS/s
Tek Stop
a -66.40ms 2.820 V
b 58.40ms -1.400 V
Δ124.8ms Δ4.220 V
M 1.00 V 40.0ms
40.0ms 25.0kS/s
T→▼0.000000 s 10k points
1 1.36 V
31 Mar 2025
21:41:44
Menu Off

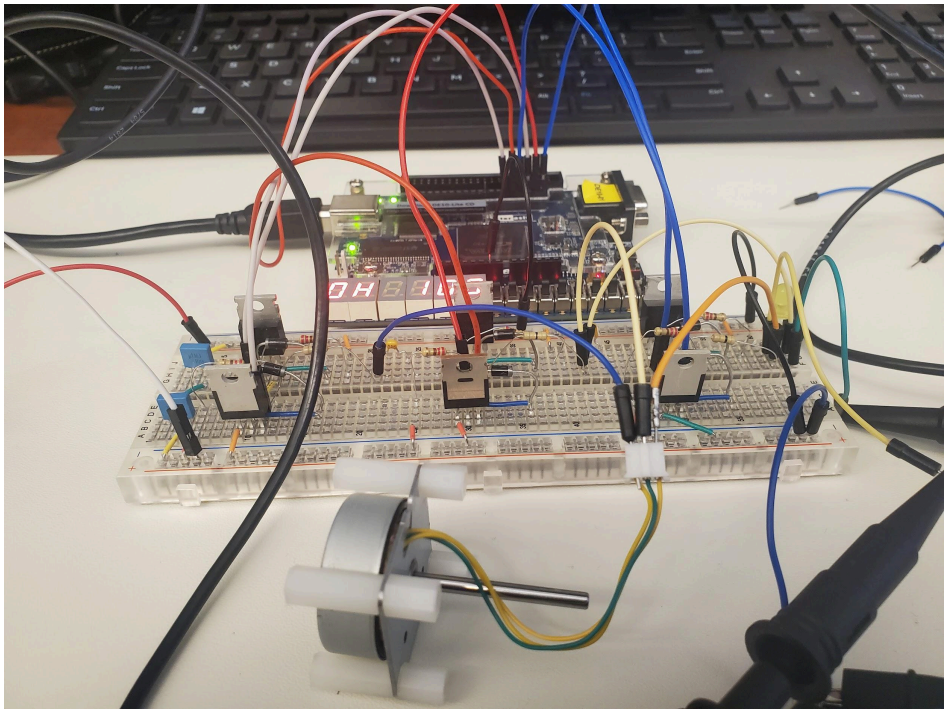
Implemented Circuit

Here, we show some pictures of the implemented hardware setup.

No load



With motor load



Challenges

Software Challenges

On the software side, the challenges were to research the SVPWM algorithm and read papers to understand it and implement it in System-Verilog. We checked various papers on the subject, which were very helpful for software development.

Another challenge in the software development was that we needed module parameters with huge sizes to account for the low frequencies used. We fixed this by compromising by reducing the length of other parameters that were not so crucial.

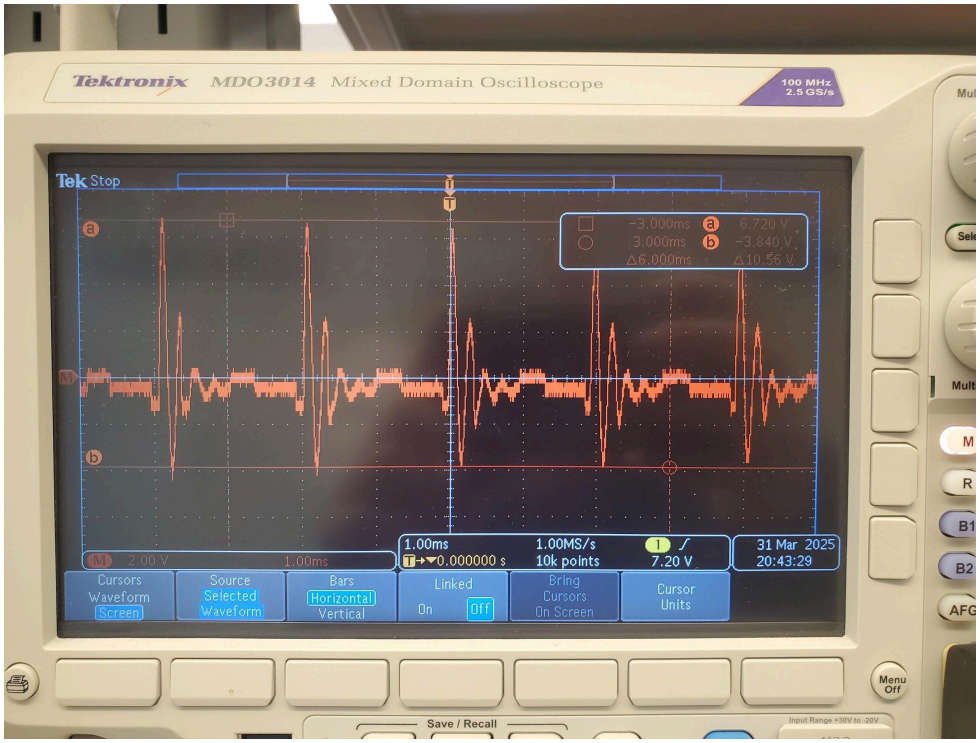
Hardware Challenges

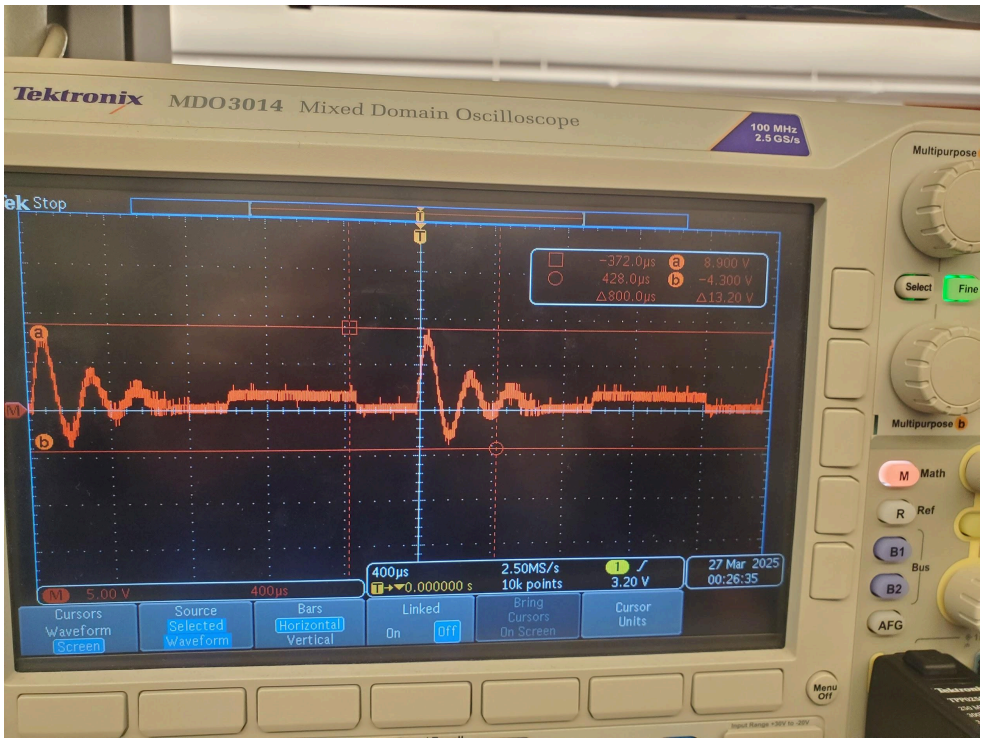
However, most of the challenges we experienced were on the hardware side. The first challenge was that the logic voltage level of the DE10-Lite is only 3.3V, which is quite low; we could not get sufficient power and current in the inverter output.

Another challenge we had was with one of the group members' personal DE10-Lite board; when none of us were touching it, it turned off and began making a sound. When checking with the lab technician, he mentioned it could've been an issue with the power adapter and that it would be hard to reverse engineer the actual issue; we continued working on the project with the lab boards.

Measurements

Several measurements were made during the development of the project. Follows the oscilloscope traces taken across the legs of the motor.





We can see that the voltage waves are sinusoidal as intended! Note also that the voltage amplitude is about 13.2V, which is quite close to 86% of the 15V supply (13V), which is the power that the SVPWM algorithm should supply!

Addressing the Damping Effect

We can observe on the waveforms and measurements that the sinusoidal waves are damped. This, at first, was due to the low MOSFETs gate voltage, which was not providing enough current to be applied through the motor. After the new circuit design, we were able to successfully drive the motor with more power, however, we damaged some of the MOSFETs while doing so. However, even with some damaged MOSFETs, we can see that the output waveforms are correct for the SVPWM algorithm, and we can even drive the motor to rotate sideways!

Conclusion

In conclusion, we were able to successfully implement the SVPWM algorithm in System-Verilog. The implemented software was uploaded to the DE10-Lite FPGA and connected to an inverter circuit, which was implemented by us using discrete components. We performed multiple analyses and tests throughout the development of the project to verify its functionality and outputs. We were able to successfully drive a BLDC motor (and LEDs) to demonstrate the functionality of the project. All design files and proposal reports can be found in this GitHub directory.

References

alldatasheet.com. (n.d.). *BC547B PDF*. Retrieved April 7, 2025, from http://www.alldatasheet.com/datasheet-pdf/view/2894/MOTOROLA/BC547B.html

CrocSee Micro 3 Phase AC Mini Hand Brushless Motor Generator Model Science Experiment Teaching Aid Kit DIY, Science—Amazon Canada. (n.d.). Retrieved April 7, 2025, from https://www.amazon.ca/CrocSee-Brushless-Generator-Experiment-Teaching/dp/B0722Z4PX5/ref=mp_s_a_1_16?crid=NR9Z957CI2PG&dib=eyJ2IjoiMSJ9.R3n7IAd5zG8fNZtGsPhm2kbe9qUnuM66vET_yN8wJz1qKeQshSJzvka-IN0B2Z8Y8jZZU9KFCaeagGRDtAZJ3dBrxZ7DRmsiwphBnW7khIIQFgXFqFoY2fR2CwZTW_kRUuhkva5mfzgSRn2cGpcgF0cEPrqXHdHNfJ5KR5zSvXLt2gEeCU5F01fhOWdq5x286jLiiduomr4Y00tyCQ1fAA.bFgRx9OeWBQ9_HXhzY4LBEzV_F7rqDmlzgoY_ZTUh_Q&dib_tag=se&keywords=3+phase+motor&qid=1740762280&sprefix=3+phase+motor%2Caps%2C90&sr=8-16

Falstad.com/circuit. (n.d.). Retrieved April 1, 2025, from https://tinyurl.com/26qqmcq9

Fig. 2. Three phase inverter topology. (n.d.). ResearchGate. Retrieved February 27, 2025, from https://www.researchgate.net/figure/Three-phase-inverter-topology_fig1_317297464

Fig. 6. Dead-time generation on the inverter control signals by the PWM... (n.d.). ResearchGate. Retrieved April 6, 2025, from https://www.researchgate.net/figure/Dead-time-generation-on-the-inverter-control-signals-by-the-PWM-technique_fig3_365902774

Irf520.pdf. (n.d.). Retrieved April 7, 2025, from https://www.vishay.com/docs/91017/irf520.pdf

Lee, U. H., Shepherd, T., Kim, S., De, A., Su, H., Gregg, R., Mooney, L., & Rouse, E. (2023). *How to Model Brushless Electric Motors for the Design of Lightweight Robotic Systems* (No. arXiv:2310.00080). arXiv. https://doi.org/10.48550/arXiv.2310.00080

P2N2222A - Amplifier Transistors NPN Silicon. (n.d.).

Space Vector PWM Intro. (2017, May 1). Switchcraft. https://www.switchcraft.org/learning/2017/3/15/space-vector-pwm-intro

Switching sequence of traditional SVPWM. (n.d.). ResearchGate. Retrieved April 6, 2025, from https://www.researchgate.net/figure/Switching-sequence-of-traditional-SVPWM_fig3_301718871

Torque and Speed Controlling of a PMSM/BLDC using Simulink and SOLO Blockset. (2022, July 8). https://www.solomotorcontrollers.com/blog/torque-speed-control-pmsm-bldc-simulink/