FPGA Field Oriented Control Jackson Gray, Aaron Yeiser 2019/12/13 jackmaxg@mit.edu, aveiser@mit.edu



Our mess of a test bench

We designed an FPGA implementation of Field Oriented Control (FOC) for brushless motors. An FPGA is perfect for controlling a brushless motors because it allows complex control loops with substantial amounts of digital processing to execute at high frequency and low latency. Our motor controller implemented field-oriented control---an advanced digital control strategy for driving various types of multiphase motors. FOC is designed to efficiently control the current through a multiphase motor without having to run the control loop extremely aggressively. Our FPGA implementation of FOC required us to run the FOC control loop at least at several kilohertz, and possibly faster for better performance. Our software design had to implement coordinate transforms to translate the measured phase currents into the rotor's reference frame, as well as providing a UART for communication with a computer, switching logic to control the inverter, and modules to interpret data from the peripheral ADCs and encoder. On the hardware side, we designed PCBs to provide a clean interface between the CMOD A7 3.3V logic and the Prius inverter 12V logic, as well as providing current sensors and a variable reluctance resolver decoder. The combined challenges of designing a good hardware interface as well as good software interfaces for peripheral hardware and a clean control loop made our motor controller a very difficult project. With the UART interface working, we were able to demonstrate that our design could spin the motor with a very substantial amount of torque, as well as effectively field weakening the motor to make it spin faster at a low bus voltage.

Brushless motors are three phase synchronous permanent magnet motors that require electronic commutation. The lack of mechanical contacts is advantageous for weight and longevity, but they require complex control electronics. There are many different methods for controlling brushless motors, and one of the most useful methods is field oriented control, or FOC. FOC is a control strategy that controls the phase and magnitude of the currents through the three phases of the motor. Specifically, the three observed motor phase currents are mathematically transformed into currents directly in phase with the motor's electrical angle (D current) and in quadrature with the motor's electrical angle (Q current). In an ideal electric motor with no rotor reluctance, we only want Q current to be nonzero. By changing D and Q voltages, we can do an inverse transform to get the target voltages for each motor phase. These voltages are used to generate PWM signals on each motor phase, with these PWM signals optimized to avoid excessive switching.

FOC is an extremely powerful motor control strategy. One advantage of FOC is that the control loop for FOC can be run substantially slower than other similar control strategies because the Q and D currents are relatively slow to change, even with a rapidly spinning motor. FOC can be easily adapted to induction motors, as well as motors with nonzero reluctance torque. With knowledge of motor parameters, sensorless control of the motor can be integrated into FOC. FOC also gives accurate torque control of the motor, enabling a brushless motor to be used as a servomotor.

Hardware Details

To have a physical platform to test on, we needed some hardware. We decided to use the inverter module out of a Gen2 Toyota Prius and an interior permanent magnet (IPM) motor from a Hyundai Sonata hybrid. Both of these parts greatly simplified the hardware work we would need to do, as both are quite simple to use, with minimal external components required on the

custom PCB motherboard to interface with our FPGA. As for our FPGA, we opted to the CMOD A7-35T breadboardable Artix-7 breakout board. It wasn't particularly expensive, had enough of the required pins, and importantly was small enough to mount on a pcb.

The Prius inverter has a simple pinout, with three 12v digital lines to control the fully isolated gate drivers, and two -12v to 12v analog lines representing two phase currents, along with a variety of other miscellaneous lines for things like enable, temperature feedback, etc. We used a 3.3v to 12v logic level shifter to drive the gate control lines from the FPGA. For converting the +-12v analog current sense signals to something the FPGA could read, we designed a differential amplifier circuit which scaled the +-12v to a +3.3v to 0.0v signal, for our 3.3v high frequency SAR ADC's to convert. Our ADC's read out the converted signals onto a three wire SPI interface. To make the most out of our current sense ADC's bandwidth, we gave each ADC their own SPI bus. The rest of the ADC's (for converting temperature and bus voltage signals) shared a common SPI bus, as they were less critical than those used to control the motor.



Left: Prius power electronics module Right: Inverter module with isolated gate drive and current sense

The IPM motor had it's three phase power connections which bolted directly to the inverter, and a 10 pin connector, six for the variable reluctance resolver for rotor position measurement, two for the motor winding thermistor, and two redundant connections for chassis grounding. We reterminated this connector into a DB9 connector, and layed out a smaller sub-board for a \$20 IC for resolver-to-digital conversion. It worked by exciting a 20khz signal across one coil of the variable reluctance resolver, and then observing the resulting signals across the two other coils, who's amplitudes would vary depending on the rotary position of the motor, due to the position dependant inductive coupling between the excitation coil and either of the observer coils. The IC handled all of these functions, producing either a position or velocity

measurement over the SPI bus, depending on the state of a control line. Unfortunately this IC only operates using 5v logic, so this chip also required a logic level converter. We ended up using two level shifters, one for outgoing signals and another for incoming signals.



Left : the backside of the motor, with the variable reluctance resolver shown Right: The Resolver to Digital converter board we designed

To implement all the various circuits to run the device, we designed two separate boards, one for the resolver-to-digital IC and it's peripheral circuitry, and one main board (motherboard) which supports the FPGA, and implements the power regulators and communication level converters required to operate the rest of the systems. We designed them in EAGLE PCB, on a two layer board. We then had them fabricated by 3PCB, a chinese boardhouse with extremely fast turnaround times.



Left: Motherboard board render Right: Resolver to digital converter board render





U U

Unfortunately, once we had the resolver to digital board built up and started testing it, we discovered that the IC didn't seem to be operating. Probing the various signals like the excitation pins and the crystal oscillator revealed that the IC wasn't doing anything. We did some sifting through the datasheet, but we weren't able to determine what was wrong. On the edge connector pinout, the 5v and 12v supplies were right next to each other, and we suspect that we might have accidentally tapped the 12v against the 5v rail, which would likely have killed the IC.



Resolver decoder board schematic

To get our project running, we really needed positional feedback. In theory, it is possible to operate the motor without sensor feedback, but it is certainly not an easy thing to do with your very first motor controller, and is particularly difficult with the motor we are using. Lucky for us, Austin Brown, a friend who had done some previous work developing FOC based motor controls, offered to let us use one of his old motor + encoder rigs, which we promptly bent a piece of sheet metal for and mounted to the side of the motor, using duct-tape to couple the output shaft of our motor to the smaller motor on his testing rig.



Austin's radial flux magnetic encoder breakout board

The encoder he was using was a radial flux magnetic encoder, which could detect the fields of a nearby magnet and deduce the direction of the magnetic field lines running in plane with the IC. The IC was in quadurature mode, meaning that it was spitting an A and a B signal, two square waves offset by 90 phase degrees. The frequency of this signal was proportional to the RPM, every falling or rising edge of either signal encoded a 1 unit move either forwards or backwards. This system has no way to represent absolute position, so the encoder included a third "i" signal, which would transition to a high state at a fixed position once per rotation.

Software Details

In order to implement the field oriented control algorithm, we needed to implement the Clarke and Park coordinate transforms in order to convert between three phase currents and voltages, and two phase voltages and currents in the stator and rotor reference frames. The FOC control loops control Q and D current in the rotor reference frame, but Q and D currents and voltages are mathematical abstractions. The Clarke (and inverse Clarke) transforms are implementable as matrix multiplication. The structure of the Clarke transform matrix allows it to be optimized on an FPGA by substituting bit shift operations for multiplies, and the inverse Clarke transform was similarly optimized. We were able to use only two multiply operations per Clarke (or inverse Clarke) transform.

The Park transform maps voltages and currents from the stator reference frame to the rotor reference frame. As such, applying the Park transform is equivalent to multiplying by a rotation matrix. We were able to avoid generating this matrix in real life by using CORDIC. CORDIC is an algorithm for generating trigonometric functions with minimal hardware resources, using only bit shifts and addition. In order to generate sines and cosines, CORDIC rotates the unit vector by angles $\arctan(1)$, $\arctan(\frac{1}{2})$, $\arctan(\frac{1}{4})$, etc.. These rotation matrices can be scaled so that all elements are powers of two, allowing bit shifts to be substituted for multiplication. CORDIC can also be used for rotating an entire vector, rather than just calculating sines and cosines. This is achievable by substituting the unit vector for the target vector to be rotated. Xilinx provides an IP block that implements CORDIC, so we wrote a wrapper around the Xilinx CORDIC IP block.

One very useful feature we wrote for implementing our designs was a fixed point multiplication block. Although this block was not very complicated, it allowed us to input integer and fractional bit depth for both the inputs and the output of the multiplier. Having a module that elegantly handled fixed point multiplication greatly simplified the process for designing the rest of the controller, because we did not have to think about scaling integer multiplication every time we needed to perform a multiplication.

We also implemented a noise-resistant UART transmitter and receiver in order to give commands to the motor controller from the computer and read internal data from the motor controller on the computer. The noise-resistant receiver averaged the received values over one bit time interval in order to avoid flipping bits. This noise rejection was especially important due to the electrical noise from the inverter, which would frequently result in receiver errors. At a baud rate of 115200, one bit was approximately 868 clock cycles, so the UART receiver would average 868 samples to determine whether a one or zero was transmitted. This averaging

technique was also applied to detect the start bit of the transmission in order to prevent noise from falsely triggering the UART receiver. With noise rejection, we were able to reliably command Q and D currents through the motor, which allowed us to effectively demonstrate field weakening in action.

The UART transmitter was substantially simpler than the receiver, as it did not have to perform any noise filtering. However, both ends of the UART transmitter and receiver had to function in a human--readable manner. We wrote a module that would convert registers to their hexadecimal equivalent to transmit over the UART, as well a module that would string together multiple bytes and send them over the UART. The UART receiver decoder had to perform the reverse function--inputting human commands and converting them into motor controller instructions. We built a frontend for the motor controller that allowed us to input an operation select character followed by hexadecimal payload bits. For instance, sending Q0400 over the UART would command a current of 15 amps (1 unit of current was roughly 14.6 mA) 90 degrees ahead of the rotor position, and IF800 would command -30 amps of D current (directly in phase with the rotor) for field weakening. Invalid strings were automatically rejected by the receiver to avoid potentially dangerous outcomes, especially since the motor controller had the potential to output very large amounts of power.

Field oriented control requires inputs to measure phase currents and rotor position. The ADCs and resolver board communicated via a serial interface, which had to be implemented according to the datasheet timing specifications. We used SAR ADCs for this project, which meant that the ADC would operate like a serial device by default, outputting the most significant bit of measurements first after a couple clock cycles. Unfortunately, the resolver IC never worked properly, possibly due to damage while soldering or ESD damage, so we had to retrofit a quadrature encoder onto the motor with sheet metal and duct tape. The encoder had a third channel that fired once per revolution to achieve absolute position. In order to get absolute position from the encoder, we had to manually rotate the motor until the third channel of the encoder fired, at which point we could accurately fix the position. We also disabled the motor until we had an absolute position fix from the encoder in order to prevent unpredictable behavior from the motor. Unfortunately for our sanity, the encoder was a 4000-count encoder, rather than a 4096-count encoder, so our encoder count would not automatically roll over when we passed the zero point again. In addition, our angles were all scaled such that a rotation of 360 degrees would not change the angle register, due to integer overflow. In order to scale the 4000-count encoder to a 4096-count encoder, we added 3 extra steps per 128 measured steps. This method

was quite effective and only resulted in about two bits of accuracy lost, which was not a big deal in the grand scheme of things.

To actually apply a three phase voltage to the motor, we used a special type of PWM generator called SVPWM. The first important part of SVPWM is that it synchronizes the switching of all three phases so that they are always symmetric and as overlapped as possible. It achieves this by using the three voltage setpoints (one per phase) as individual thresholds superimposed onto a triangle wave, and switches a given phase whenever the triangle wave crosses the corresponding threshold level. The other important part of SVPWM is that it can adjust the dc component of the three phase output to increase the peak-to-peak voltage of the output sinusoids higher than the DC bus voltage. This only works because the number of phases is odd. Imagining three sinusoids, at no point in time are any two sinusoids at their peak, and the inverter only ever needs to produce a differential voltage of ³/₄ of whatever the peak-to-peak voltages of the sinusoids is. Therefore, the inverter is capable of producing a three phases which are 4/3rds the voltage of vbus. To be able to do this, it is important that the three phases are switched synchronously, as we established SVPWM is able to do earlier.

Our implementation of SVPWM uses an up/down counter to generate the triangle waves, which is driven by a clock multiply/divider that we wrote. Because the rate at which the triangle counter needs to be incremented is not evenly divisible by the clock frequency, we needed to be able to generate a clock which is capable of producing something like 5 output pulses every 6 input clock edges (a factor of 5/6), for example. This was achieved by using a counter which is incremented by the numerator on every input clock pulse, and produces an output clock edge every time the counter counts up to the denominator. Once the counter has passed the denominator, a bit of logic executes which rolls over the counter, while retaining however much it has overcounted by.



The fractional clock divider configured for, 2/3rds dividing input clock on top, output clock on bottom

This clocking method was a reasonably elegant solution (so long as your only concerned with average frequency), though the implementation isn't perfectly idiotproof. We should have implemented this functionality directly into the triangle wave generator, instead of using implementing a clock divider module and incrementing the triangle wave counter every clock edge of the clock divider, as it would have allowed for higher resolution and higher counter frequency. If we forget how the triangle wave generation mechanism worked and tried to increase the switching frequency later, we think we may run into a bug or two, but it works as is for now.

The SVPWM module itself was designed to take three signed voltage levels, and output three high/low states for each output phase. The module was implemented about as you'd expect, utilizing a counter with the same bit width as the inputs, with combinatorial logic which switches direction when it reaches the endpoints. We did have to implement a couple important protections, the first being that the input voltages are only sampled at the top and bottom of the switching cycle. This way we don't run into issues with the output being intermittently on and off if it's updated in the middle of a pwm cycle. Unfortunately, we decided not to implement the DC offset shifting for SVPWM, as it was additional complexity that wasn't required for the controller to be operational, and which would make the output waveforms more difficult to interpret.



Two cycles of an SVPWM output in testbench, displaying the triangle wave generation and output pin states.

Implementation details

Our implementation was relatively lightweight and did not require the full area of the FPGA. This makes sense because our application is not particularly parallelized--we mostly care about having very low latency for our feedback loop. As such, we could have increased the

clock frequency from 100 MHz to around 200 MHz without issue. Power consumption was estimated to be around 300 mW maximum.



The FPGA space utilization was fairly lightweight

Challenges and Setbacks

One major time sink was the actual PCB layout. While we have experience designing and laying out PCBs for projects in a short period of time, this board posed a few unforeseen and underestimated challenges. Firstly, while we mostly knew the pinout and behavior of the various Prius inverter signals, there were still a few unknown signals that we had to design the circuit to be flexible around. For example, the analog Vbus feedback line exhibited this very strange 0.5 volt to 2 volt linear relationship to a vbus of 0v to 250v. Similarly, we had to design a flexible circuit for translating the digital signals from the inverter, as we weren't sure what digital output standard the inverter implemented.

The second was just the sheer number of signals that had to be organized and individually considered, and many more decisions had to be made than expected. Trying to work on a board and move things forward can be difficult with a handful of difficult decisions haven't been made. A related great annoyance was that all of the devices we needed the FPGA to interface with ran different logic standards than the FPGA, so every interface needed logic level shifting.

The third issue was that the resolver decoder pcb wasn't going to be as easy as taking a known good board sending out for a couple copies like we thought it would be. Unfortunately, the original version of the board had quite a few important signals not routed that we hoped would be. This meant some amount of editing had to be done, and given that we didn't have any

of the libraries for the original parts and how messy the original schematic was, we ended up just rebuilding the board from scratch.

One major annoyance with designing power electronics is the sheer amount of electrical noise generated by switching large amounts of current. The Prius inverted connector came with a large ferrite bead, and we were able to successfully mitigate noise on the encoder leads by threading them through a ferrite bead. However, even with using a fairly nice (presumably shielded) USB cable for the serial interface, we had to deal with substantial amounts of noise on the UART--enough that well over half of the signals we sent were corrupted. This was objectively bad and somewhat unsafe, as having a reliable serial interface is important for controlling the motor. We were able to solve the issue of UART noise by implementing an averaging filter over each bit, which drastically reduced the BER.

Future plans

One of the primary reasons we chose to build a sophisticated brushless motor controller for our project is that we have many practical uses for a powerful brushless motor controller that implements field oriented control. We intend on continuing development of the motor controller into the future. As is, our motor controller implements core functionality of a brushless motor controller running FOC--we can efficiently control the Q and D currents through the motor. However, we would like to make the motor controller more practical and usable for high power electric vehicles and other applications, as well as cleaning up the code for greater configurability and extensibility.

Our current UART interface to the motor controller is effective, but somewhat primitive. It is capable of reliably sending simple commands to the motor, but it does not have any logging functionality. We would like to be able to modify more operating parameters, such as PI controller gains, switching frequency, as well as additional motor functionality. We would like to add a velocity control loop that wraps the current control loop, as well as built in lookup tables for optimal Q and D currents for any given speed-torque combination. This will allow us to put the motor controller on a vehicle and run as efficiently as possible while still performing well. This increased functionality will require more serial interface options, so that we can configure the functionality on the fly. We could also set defaults through the serial interface by accessing non-volatile memory. Additional motor features also include safety features like overcurrent, undervoltage, and overtemperature lockouts. Another use for a high speed UART connection is logging data. We should be able to trigger a logging interval, which would take high-speed measurements of motor parameters, store them in DRAM, and then output those measurements over the serial bus. By changing logging parameters, we could easily harvest data about the motor performance to optimize the motor controller performance. We could also add measurement modes where the controller automatically steps through different parameters and logs and outputs serial data to a computer. An automatic logger would greatly simplify characterizing the motor performance.

On a similar note, we want to write a flexible telemetry manager system for streaming various important state machine inputs (like modes, sensor readings, motor currents, positions and velocities) to a receiver, likely our UART interface back to the computer. This telemetry information would be broadcast repeatedly, allowing the operator to easily monitor the performance of the controller in real time. The CMOD A7 board does in fact have some non-volatile memory on board, which we may eventually want to use to implement some sort of telemetry cache which we can read out of on startup.

One feature we would like to add relatively soon is to start utilizing other signals from the inverter. Currently we are ignoring many of the signals coming from the inverter, as they were not critical for functionality of the controller. The bus voltage feedback line is likely going to be implemented soon here so that our control loop can scale the SVPWM voltages appropriately. This will require writing a more flexible SPI interface, for use with sampling all of the low speed ADC's in an iterative fashion.

From a more hardware perspective, we would like to eventually improve the inverter itself. The Prius inverter isn't particularly impressive in terms of switching frequency or high speed performance. It's gate driver board doesn't realistically allow for switching much faster than ~20khz, and we think it would be a fun and interesting task to try and design a more compact inverter module which is able to perform up past 100khz, potentially making use of GaNFETs or other new technologies. Being able to control low inductance motors by making use of extremely high speed switching would be a great application for our hardware implementation of Field Oriented Control, because of the low latency of our control loop.

Appendix - Schematic:

