Leveraging High-Level Synthesis to Migrate Motor Control Algorithms from Microcontroller to FPGA

Emily Belovich
NASA Glenn Research Center
Cleveland, OH, USA
emily.c.belovich@nasa.gov

Julie Blystone
NASA Glenn Research Center
Cleveland, OH, USA
julie.a.blystone@nasa.gov

John Maroli
NASA Glenn Research Center
Cleveland, OH, USA
john.m.maroli@nasa.gov

Matthew Granger
NASA Glenn Research Center
Cleveland, OH, USA
matthew.granger@nasa.gov

***Abstract*—As motor control algorithms become increasingly complex, traditional microcontroller-based implementations are reaching computational limits that prevent the controller from operating at the required speed. This paper presents a novel workflow leveraging High-Level Synthesis (HLS) to migrate motor control algorithms from a microcontroller implementation to a Field-Programmable Gate Array (FPGA) implementation. The proposed approach utilizes the free Vitis HLS software to automatically convert Embedded Coder-generated C code from a Simulink model into Hardware Description Language (HDL) code suitable for FPGA deployment.**

***Index Terms*—Motor Drive, Electrified Aircraft, Motor Control Algorithm, High-Level Synthesis, FPGA**

I. Introduction

Motor control algorithms for aeronautics applications are becoming increasingly sophisticated, requiring advanced algorithms that reach the computational limits of traditional microcontroller-based implementations. The need for faster, more efficient control has become critical. Increased complexity also drives the need for streamlined development processes for flight software [1].

Cryogenic motor drives are an area of active research that require much more complex control algorithms. These drives operate at very low temperatures and must deliver nearly pure sinusoidal stator currents to minimize eddy-current heating; even small current ripple can produce enough loss to quench superconducting windings. In practice, high-frequency (switching-frequency) harmonics can be mitigated by increasing the inverter's switching frequency and using passive output filters, albeit with a trade-off in switching losses and filter size [2]. By contrast, low-frequency "spatial" harmonics (for example, those arising from motor saliency or winding layout) occur near the fundamental frequency and cannot be easily removed by conventional LC filters. A high-bandwidth drive can instead include dedicated control loops (such as resonant or repetitive controllers tuned to the problematic frequencies) to force those current components to zero. Each added controller loop increases the real-time computational load. On a single-core processor, the loop execution time may become too long to meet the required bandwidth. To overcome this, a parallel computation architecture can be used so that additional current regulators can run simultaneously.

Field-Programmable Gate Arrays (FPGAs) offer a compelling alternative to microcontrollers, providing parallel processing capabilities and significantly higher operating frequencies [3]. However, the transition from C-based algorithms running on a microcontroller to algorithms written in a hardware description language (HDL) and running on an FPGA traditionally requires extensive manual code conversion and hardware design expertise.

This paper presents a novel workflow to convert a visual Simulink controller model into VHDL (Very High Speed Integrated Circuit Hardware Description Language) to run on an FPGA. The Simulink model is converted to C code using the Embedded Coder toolbox and then converted to VHDL using High-Level Synthesis (HLS). The controller model running on the FPGA is compared to the same model running on the microcontroller, as both were derived from the Simulink model.

Auto-generating HDL code using HLS represents a sustainable path forward [4], [5], enabling engineers to meet demanding performance requirements without sacrificing development efficiency. This approach positions organizations to effectively address the computational challenges of next-generation motor control algorithms, particularly in demanding environments such as superconducting motor systems where precision and speed are paramount [6].

II. Microcontroller Workflow and Limitations

This section describes the microcontroller-based workflow for algorithm development. The motor control algorithm is first implemented as a Simulink model. Simulink provides excellent visualization and simulation capabilities for complex control systems. The Embedded Coder toolbox is then used to generate C code from the Simulink model, which is deployed on a microcontroller.

As control algorithms become increasingly complex, the limits of traditional microcontrollers are being reached. More complex control calculations generally take more CPU clock cycles and therefore more time to execute the control loop. This can be offset by performing computations in parallel, but control calculations often require sequential computation. The time it takes to perform all control calculations directly determines the maximum speed of a control loop.

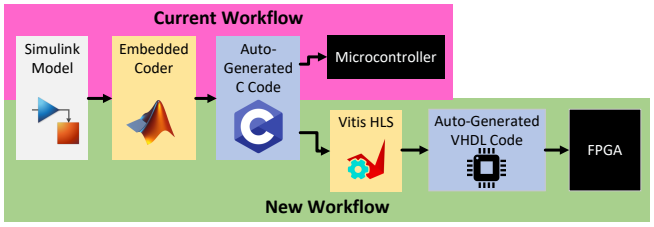

Fig. 1. Pictorial representation of the microcontroller workflow (top, in pink) and FPGA workflow (bottom, in green).

III. FPGA Workflow

FPGAs offer several advantages over microcontrollers for motor control applications. They can perform complex calculations in less time with proper implementation, and their parallel processing architecture allows multiple operations to execute simultaneously where possible. Both of these advantages reduce overall control loop computation time. The barrier to using FPGAs over microcontrollers has typically been increased implementation complexity. Traditional FPGA development requires expertise in hardware description languages such as Verilog or VHDL. This is at a much lower level than the theoretical problem of controlling a motor.

To address these challenges, this paper proposes a new workflow that leverages Vitis High-Level Synthesis (HLS), a free tool from AMD (Advanced Micro Devices) that automatically converts C code into optimized HDL code suitable for FPGA implementation. The proposed workflow maintains the high-level Simulink development environment where engineers continue to create and refine motor control algorithms using established modeling practices. The Embedded Coder toolbox generates C code from the Simulink model exactly as in the microcontroller workflow. Instead of deploying the C code directly onto a microcontroller, it is used as an input to Vitis HLS for automated conversion to VHDL. The VHDL code generated by Vitis HLS then runs on an FPGA. This approach preserves the existing Simulink-based development methodology while enabling deployment to FPGA hardware. A pictorial representation of the two workflows is shown in Fig. 1.

The process is mostly automated, but some manual intervention is required. A wrapper file must be written in C for both workflows to interface with the generated C code, though the contents of the wrapper will differ slightly depending on the workflow. In addition, the FPGA workflow requires a VHDL wrapper file to interface with the generated VHDL code. These wrapper files serve the purpose of interfacing the motor control algorithm with the physical hardware inputs and outputs on the target device.

The presented approach has advantages over two other approaches evaluated for generating VHDL from a Simulink model. The HDL Coder toolbox from MathWorks [7] was evaluated for converting Simulink models to VHDL, however delays and timing had to be handled at the Simulink model level. While this gave the developers a high level of control over computation path timing, it weighed down the model and made modifications more complex. AMD Vitis Model Composer [8] was also evaluated for generating VHDL. This worked within the Simulink model, but required model modifications, similar to those required by HDL Coder, that added complexity. The proposed approach required no implementation-specific modifications to the control model, keeping the model concise and control-focused. Additionally, Vitis HLS is freely available and does not require the purchase of a license, unlike HDL Coder and Vitis Model Composer.

IV. Implementation

The feasibility of the proposed approach was demonstrated using a motor control algorithm. This algorithm was converted from Simulink to C using Simulink's Embedded Coder toolbox, then from C to VHDL using Vitis HLS. The VHDL code generated by Vitis HLS is packaged in an Intellectual Property (IP) core. This reduces design complexity because the IP core contains only top-level inputs and outputs instead of the underlying VHDL source code. From here on, references to the "controller IP" or "controller module" refer to the IP core implementing the motor control algorithm that was generated using Vitis HLS. This IP core is added to a Vivado project. Vivado, another free tool from AMD, is a comprehensive software environment for the analysis, synthesis, place-and-route, and verification of FPGA designs.

A Vivado project was created to instantiate the controller IP and test it on FPGA hardware. The Nexys Video, an FPGA development board manufactured by Digilent, was used for this test. The implemented algorithm on the FPGA board was compared to simulation output from the Simulink model to verify that numeric computation was accurate. The implementation was not used to control a motor since this was a preliminary test.

Known input values were provided to the simulation and the controller IP and the outputs of both were compared. The controller module generated using Vitis HLS uses the ap_memory interface. With this interface, the module inputs are stored in memory and the module reads values from specific memory addresses as it needs them. There were several memory components instantiated in the Vivado project. These memory components store the values that are provided to the controller module on each control loop execution. The controller module was run for 2,100 iterations.

The purpose of this test was to confirm the viability of the workflow and verify that the controller IP running on the FPGA produces the correct output values. With this intention, the Vivado project consists of a top-level wrapper file that contains four components:

1) Vivado's Virtual Input Output (VIO) IP core: this allows the user to run and reset controller during hardware testing
2) Controller IP core: the IP core generated using Vitis HLS that implements the motor control algorithm
3) Vivado's Block Memory Generator IP core(s): Block Random Access Memory (BRAM). Multiple BRAMs

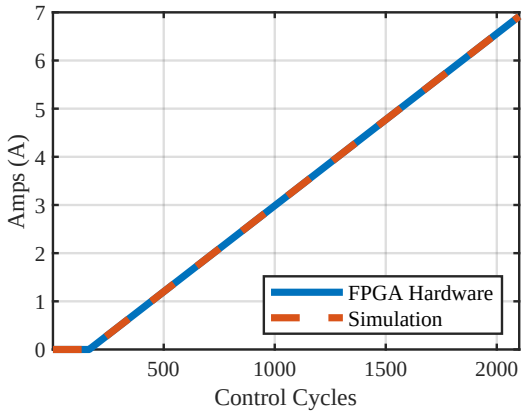

Fig. 2. Commanded q-axis Current (Iq*)

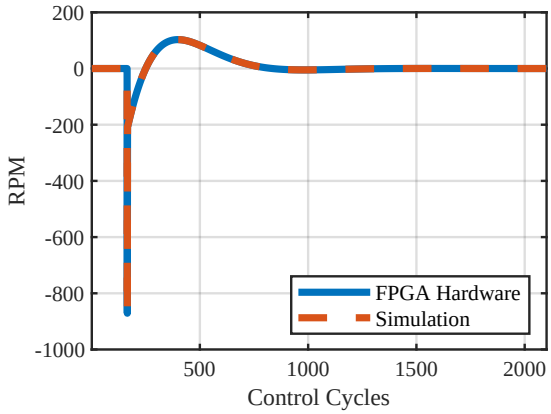

Fig. 3. Speed (in RPM) Measured by the Observer

are instantiated to hold the input values provided to the controller IP

4) Vivado's Integrated Logic Analyzer (ILA) IP core: similar to an oscilloscope for digital signals. Used to view and save the output values from the controller IP.

The top-level wrapper file in the Vivado project was synthesized, implemented, and tested on FPGA hardware in the laboratory. As stated above, the ILA IP core was used to capture the outputs from the controller. The controller FPGA outputs were compared to the controller Simulink simulation outputs.

V. Results

The results below validate this approach for complex, multi-component control algorithms. Hardware testing confirmed that the FPGA implementation of the controller maintained the functional behavior of the original Simulink model.

A. Data Accuracy

This section compares data collected during a Simulink simulation of the motor controller to data collected when the controller was running on the FPGA. Fig. 2 shows the commanded q-axis current in amps. It shows the current ramp sequence for starting up the motor with sensorless control. Fig. 3 shows the speed (in RPM) measured by the observer. It can be seen that the angle/speed observer was reset (the transient) and then settles to 0. The x-axis of each graph denotes control cycles (controller iterations). Each plot spans 2,100 cycles.

In both figures, the FPGA hardware and simulation data look identical. Further analysis confirmed all values are within 0.001 units (amps or RPM, as appropriate) of each other. This shows Vitis HLS correctly converts the motor control algorithm from C to VHDL and it runs on the FPGA with a high degree of accuracy.

B. HLS Optimizations

The Vitis HLS software provides many optimization pragmas. Pragmas are compiler directives embedded in the C/C++ code that guide the tool's hardware micro-architecture without changing the algorithm's functional behavior. For example, the 'allocation' pragma is used to tell Vitis HLS how many hardware instances it may create for a given operator or function, thereby enforcing resource sharing. Limiting the number of instances of an operator or function reduces hardware utilization but can increase latency.

The next sections of the paper refer to two motor controller models, the 'base model' and the 'optimized model'. The 'base model' is the model without any HLS pragmas applied, while HLS pragmas are used in the 'optimized model'. The intention was to use pragmas to decrease FPGA resource utilization, accepting that doing so would likely increase the controller's latency.

The following HLS pragmas were used:

- #pragma HLS allocation function instances=sinf limit=1
- #pragma HLS allocation function instances=cosf limit=1
- #pragma HLS allocation function instances=atan2f limit=1
- #pragma HLS allocation function instances=rt_hypotf limit=2

The functions sinf(), cosf(), and atan2f() compute the sine, cosine, and arctangent, respectively. The rt_hypotf() function computes the Euclidean norm of two single-precision floats. The pragmas listed above limit the number of instances of the sinf(), cosf(), atan2f() and rt_hypotf() functions in the optimized model to 1, 1, 1, and 2, respectively. For comparison, in the base model, sinf() and cosf() combined had a total of 4 instances. There were also 5 instances of atan2f() and 4 instances of rt_hypotf(). The following sections will discuss the impact of these pragmas on the latency and FPGA resource utilization of the base model and optimized model.

C. Latency

This section compares the latencies of the base model and the optimized model for 2,100 control cycles. Latency is measured in FPGA clock cycles. The model's latency is the number of FPGA clock cycles required for the model to complete one control cycle. For simplicity, HLS pipelining was intentionally avoided. The controller operates by accepting

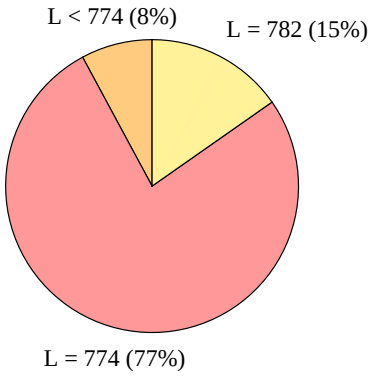

Fig. 4. Latency, L, in FPGA clock cycles, for the base model

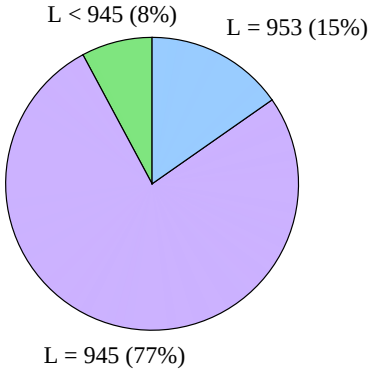

Fig. 5. Latency, L, in FPGA clock cycles, for the optimized model

input values, processing them to completion, producing output values, and only then accepting new inputs. Consequently, computations do not overlap and controller latency translates directly into execution speed. Both controller models run on the FPGA using a 100MHz clock, which has a 10ns clock period. Fig. 4 shows the measured latency distribution for the base model, and Fig. 5 shows the measured latency distribution for the optimized model.

As can be seen in the graphs, the latency of each model is variable. This is because the controller module only reads input values from memory as it needs them. Some iterations require the control algorithm to read more input values than other iterations. This requires additional clock cycles which increases the latency for that iteration.

The longest recorded latency of the base model is 782 clock cycles, which occurred 15% of time. This latency corresponds to a control frequency of 128.8kHz. The longest recorded latency of the base model is 953 clock cycles, which occurred 15% of the time. This latency corresponds to a control frequency of 104.9kHz. The latency of the optimized model is approximately 22% greater than that of the base model. This is expected, because the optimized model limits the number of hardware instances of several functions which forces operations to execute serially instead of in parallel.

TABLE I
FPGA RESOURCE UTILIZATION

	Base Model	Optimized Model	Resource Change
LUT	51,579	38.398	-26%
LUTRAM	751	539	-28%
Flip-Flops	43,367	32,010	-26%
Block RAM	9.5	6.5	-32%
DSP Slices	128	80	-38%

D. FPGA Resource Utilization

This section compares the FPGA resource utilization of the base model and the optimized model. After running place-and-route on a Vivado project, Vivado generates a hardware utilization report that explains how many FPGA hardware resources are required to implement the design. In the report, device resources are summarized using five categories: LUTs, LUTRAM, Flip-Flops, Block RAM, and DSP slices.

- Look-Up Tables (LUTs) are the fundamental combinational logic elements of an FPGA. They implement arbitrary Boolean functions and are the primary resource used for control logic.
- LUTRAM refers to LUTs configured as small distributed memory rather than logic, allowing designers to implement compact memories, low-latency buffers, or shift registers directly within the logic fabric.
- Flip-Flops (FFs) are sequential storage elements that hold binary states (0 or 1) and synchronize logic operations between clock cycles. They are used for pipelining, state machines, and data registers.
- Block RAM (BRAM) consists of dedicated on-chip memory blocks that provide larger and more efficient storage than LUTRAM, commonly used for buffers, lookup tables, and data storage structures.
- Digital Signal Processing (DSP) slices are specialized arithmetic units optimized for high-speed multiply, multiply-accumulate, and other arithmetic operations. They are essential for signal processing, filtering, and computationally intensive algorithms.

Together, these categories describe how a design maps onto the FPGA's logic, memory, storage, and arithmetic resources, providing a measure of implementation cost and scalability. Table I summarizes the resource utilization of the base model and the optimized model when the design is implemented on the Nexys Video board.

The table shows it is possible to decrease the FPGA resource utilization by 26% to 38% by simply adding HLS optimization pragmas to the C/C++ code in the Vitis HLS project.

E. Comparison with Microcontroller

The generated C code was compiled for a single CPU core of an AMD Zynq 7015 SoC, running directly on the core

without an operating system. The SoC contains a dual-core ARM Cortex-A9 CPU running at 667MHz. The maximum achievable control frequency of the control loop alone was 169.5kHz, however this does not include the time required to interface with inputs and outputs. Reading user commands and ADC values decreases the maximum loop frequency to 86.5kHz. The second CPU core was dedicated to telemetry.

The maximum achievable control frequency of the FPGA controller module was 128.8kHz, with the FPGA clock running at 100MHz. This was not diminished, however, by reading user commands, reading ADC values, or by sending telemetry. Those tasks can all be done in parallel on the FPGA, resulting in a significant performance increase over the microcontroller.

Parallelization can be achieved on the microcontroller, however it is limited by the number of cores. The FPGA can perform significantly more parallel computations with enough device resources. Microcontroller telemetry was implemented on the second CPU core in this instance, so it did not effect controller performance. This did, however, result in the ADC reads needing to take place on the same core as the control algorithm, which reduced maximum control frequency.

While it is difficult to draw direct comparisons between two very different devices, it becomes clear that the parallelization of input and output tasks is an advantage of the FPGA that could yield higher aggregate control loop frequencies. The number of parallel tasks is not locked to the number of CPU cores either, since many parallel tasks can be implemented in the FPGA fabric.

VI. Conclusion and Future Work

This work demonstrates that HLS tools provide a practical and cost-effective pathway for migrating motor control algorithms from a microcontroller implementation to an FPGA implementation. The proposed workflow preserves the existing Simulink-based development processes while unlocking the performance benefits of FPGA hardware. By leveraging existing free vendor tools, like Vitis HLS, this approach improves efficiency, increases development speed, and decreases costs associated with advanced motor control system development. The successful migration of the complete motor control algorithm validates the scalability and reliability of this methodology.

The next step in this work is to integrate the FPGA controller with power electronics to control a physical motor, verifying that our control implementation is practically sound as indicated by comparison with simulation.

References

[1] J. M. Maroli, B. A. Morris, J. A. Blystone, and A. M. Oconnor, "Utilizing code generation from models for electric aircraft motor controller flight software," in *AIAA AVIATION 2023 Forum*, p. 4274, 2023.

[2] M. G. Granger, T. F. Tallerico, A. D. Anderson, J. J. Scheidler, P. Kascak, A. Leary, and R. Jansen, "Concept design of a 1.4 mw drive for rotor loss minimization in a partially superconducting motor," in *2022 IEEE/AIAA Transportation Electrification Conference and Electric Aircraft Technologies Symposium (ITEC+EATS)*, pp. 714–719, 2022.

[3] J. Schneider, "Field programmable gate arrays (fpgas) vs. microcontrollers: What's the difference?," 2025. Available: https://www.ibm.com/think/topics/fpga-vs-microcontroller. Accessed: 2025-11-19.

[4] S. Lahti and T. D. Hämäläinen, "High-level synthesis for fpgas—a hardware engineer's perspective," *IEEE Access*, vol. 13, pp. 28574–28593, 2025.

[5] E. Lee, M. Parker, and K. Ober, "Benefits of hls for dod asic development," in *NAECON 2024 - IEEE National Aerospace and Electronics Conference*, pp. 119–124, 2024.

[6] E. Belovich, M. Carbone, M. Granger, and S. Kowalewski, "Resonant motor drive for electrified aircraft: Algorithms and controls," in *2025 IEEE/AIAA Transportation Electrification Conference and Electric Aircraft Technologies Symposium (ITEC+EATS)*, pp. 1–5, 2025.

[7] MathWorks, "HDL Coder," 2025. Available: https://www.mathworks.com/products/hdl-coder.html. Accessed: 2026-03-11.

[8] AMD, "Vitis Model Composer," 2025. Available: https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis/vitis-model-composer.html. Accessed: 2026-03-11.