Migrating Motor Controller C++ Software from a Microcontroller to a PolarFire FPGA with LegUp High-Level Synthesis

White Paper

June 2020

INTRODUCTION

This white paper is aimed at engineers designing embedded motor controllers for Industrial applications. Specifically, industrial control applications that involve the precise control and coordination of several motors using a software-based motor controller. Embedded motor controller software typically targets an ARM Cortex-M microcontroller running a real-time operating system. However, certain industrial control applications require higher motor performance and precision than what is possible to achieve using a software-based motor controller.

In closed-loop motor controllers, low latency and low jitter is critical because the motor current needs to be changed at regular intervals based on real-time sensor feedback from motor encoders. Software running on a microcontroller, even running a real-time operating system, may struggle to achieve low latencies (sub 10 microseconds) due to jitter, interrupt latency, and time taken to compute. Instead, we propose to migrate the software-based motor controller to a hardware-based motor controller implemented on a Microsemi PolarFire FPGA. An FPGA implementation offers deterministic latency and jitter, which will enable your industrial design to achieve the best possible motor performance and precision.

Designing a new hardware-based motor controller from scratch for an FPGA, using Verilog/VHDL, can be time consuming. Typically, an engineer already has an existing motor controller designed in C/C++ that works on a microcontroller. In this situation, the ideal solution is to automatically convert the existing C++ software code into an equivalent hardware implementation targeting a Microsemi PolarFire FPGA. This is made easy by the LegUp High-Level Synthesis (HLS) tool and integrated development environment, which can compile C++ software into a hardware block targeting a PolarFire FPGA.

By migrating from a microcontroller to an FPGA, the motor controller will have a deterministic latency, with a fixed number of clock cycles between receiving motor encoder feedback and driving current to the motors. In this whitepaper, we give a case-study on how an engineer migrated an existing C++ motor control application working on a microcontroller to a Microsemi PolarFire FPGA using the LegUp HLS tool.

> LegUp HLS allowed us to easily port our existing C++ motor controller code to a Microsemi FPGA with minimal modifications. The generated hardware core met our latency, Fmax, and area requirements. The LegUp team has also provided us with great support!
>
> Koji Yoneda
> CEO,
> Sodick America

MOTOR CONTROLLER SYSTEM OVERVIEW

The target system for the motor controller is shown in Figure 1 and consists of two boards. The microcontroller board has an ARM microcontroller with an attached DDR memory, which is connected over PCIe with the FPGA board. The FPGA board has a PolarFire FPGA (MPF300TS_ES-1FCG1152E) connected to 8 motors and has a DDR memory holding data needed by the motor control algorithm.

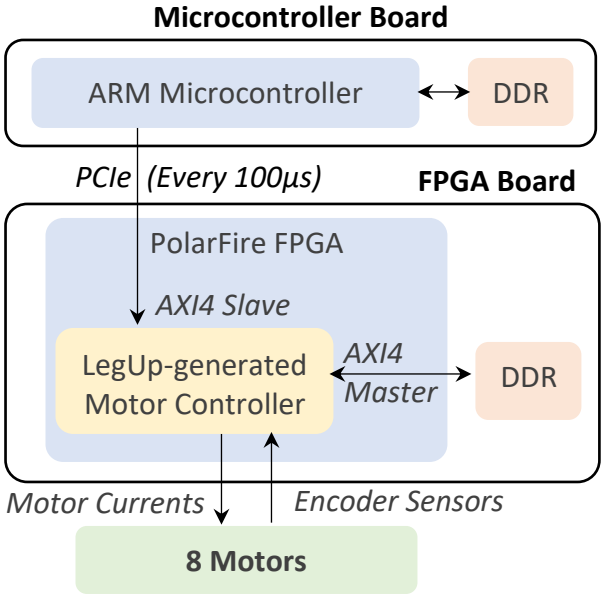

Figure 1: Motor Controller System Diagram

In Figure 2, we show how the coarse-grained motor positions are calculated by the microcontroller every 100 microseconds (labeled A, B, C). Every 100 microseconds, the ARM microcontroller sends the updated position of the motor over the PCIe interface, updates the FPGA motor controller control status registers, and starts the fine-grained motor controller running on the FPGA.

On the FPGA, the LegUp-generated motor controller hardware core is controlled via an AXI slave interface. The fine-grained motor positions are calculated by the FPGA every 10 microseconds or less (labeled 1, 2, 3, 4 in Figure 2) using input feedback from the motor encoder sensors. The FPGA outputs the motor currents every 10 microseconds to adjust the motor positions.

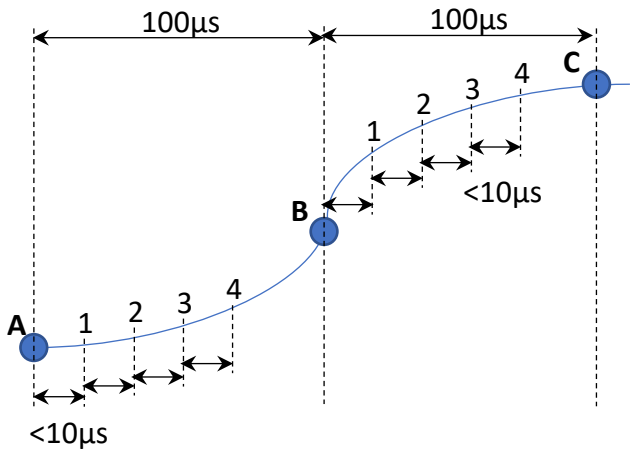

Figure 2: Coarse and Fine-grained Motor Path Calculation

The FPGA motor algorithm in C++ consists of two loops iterating over 8 motors: the path generator loop and the motor output current loop. There is also an initial stage to perform a DMA request for the compensation data from DDR memory using an AXI master interface. LegUp synthesizes the C++ code into a hardware core, including the AXI slave and master interfaces.

LEGUP HLS DESIGN METHODOLOGY

An engineer can use the LegUp HLS eclipse-based IDE to implement their motor controller software in C++. First, the engineer verifies their application's functionality by compiling and running the software on their local computer with a C++ software testbench. Next, the engineer specifies circuit Fmax constraints, and uses LegUp HLS to automatically convert the C++ software into an equivalent Verilog module in a few minutes.

For hardware verification, LegUp HLS supports an automated cycle-accurate co-simulation flow using Modelsim. The co-simulation flow will run the C++ software testbench to gather input test vectors and expected outputs, then will simulate the generated Verilog module with the input test vectors, while verifying the outputs. The co-simulation flow also allows the user to accurately measure the cycle latency of the motor controller and includes support for AXI interfaces. Finally, the user can click a button in the LegUp HLS IDE to synthesize the design using Microsemi Libero, generate an FPGA bitstream and report the FMax and area.

Figure 3 shows a block diagram of the LegUp-generated motor controller hardware block. The input to the controller is the motor position from the encoder sensor (encoder_pos). For each of the 8 motors there are two current outputs (current_iq, and current_id), which are represented in the C++ as arrays. The motor controller includes an AXI4 slave interface, to allow the ARM microcontroller to burst write data to the control status registers in memory-mapped slave memory. The memory-mapped slave memory is represented in C++ as a global struct. The motor controller also contains an AXI4 master interface to burst read from DDR off-chip memory. The core motor controller algorithm consists of two loops that iterate over the 8 motors: the path generator loop and the motor output current loop. We apply the LegUp HLS loop pipelining user-constraint so that the loops can be pipelined with overlapping execution of iterations for better latency.

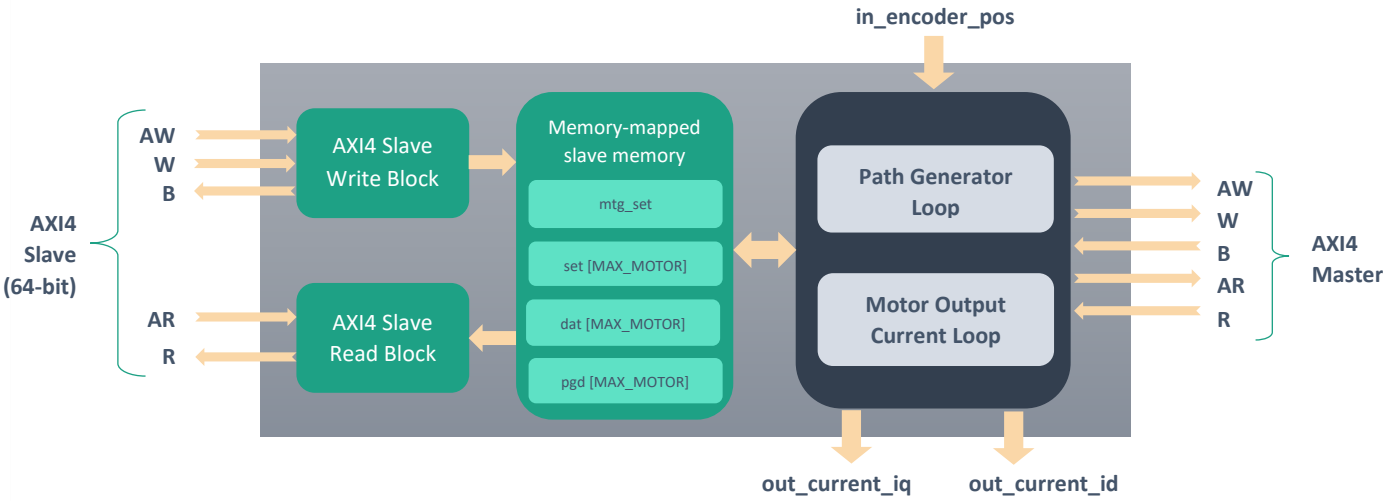

Figure 3: LegUp-generated Motor Controller FPGA Hardware Block

LEGUP HLS HARDWARE INTERFACES

In LegUp HLS, the user can specify the top-level hardware interfaces using function arguments with standard C++ types and LegUp-specific C++ types for AXI interfaces. These types will be compiled into the corresponding hardware interface in Verilog.

```
void __attribute__((noinline)) MotorControl
(
    uint32 in_encoder_pos[MOTORS],
    int16  out_current_iq[MOTORS],
    int16  out_current_id[MOTORS],
    AxiInterface<uint32, uint64, uint8> &master
) {
```

Figure 4: C++ Top-Level for Motor Controller Block

Figure 4 shows the motor controller C++ top-level level interfaces for the motor controller hardware block in C++. For each motor, the block has two 16-bit outputs (out_current_iq and out_current_id) and a 32-bit input (in_encoder_pos). These inputs and outputs are represented as an array in C++, which is split into individual ports when generating the Verilog module. The AXI master interface is specified as a reference to the LegUp-provided "AxiInterface" C++ class, with template arguments specifying an address width of 32 bits, data width of 64 bits, and byte enable of 8 bits. This can be used to DMA data from DDR memory using burst requests.

```
struct SlaveMemoryT {
    HardwareMtmGlobalSettings mtg_set;
    HardwareMtmGlobalData     mtg_dat;
    HardwareMotorSettings     set[MOTORS];
    HardwareMtmCommonData     dat[MOTORS];
    PathGenData               pgd[MOTORS];
    ...
}
```

Figure 5: AXI4 Slave Memory-Mapped C++ Struct

The AXI slave memory-mapped memory is represented in C++ as a global struct as shown in Figure 5. LegUp has a user-constraint to specify that a struct should be accessible from an AXI slave interface. LegUp will then automatically generate the 64-bit AXI slave interface ports and assign addresses to the struct elements. These memory-mapped addresses can be found in a generated header file.

Figure 6 shows the Verilog module interface generated by LegUp HLS for the 8 motor inputs/outputs. Input ports are labeled with "_readdata" and output ports are labeled with "_writedata". Whether a port is an input or output is inferred automatically by LegUp based on how the C++ array function arguments are accessed in the top-level function. Arrays that are only read from become input ports. In Figure 7, we show the generated Verilog for the AXI master and slave interfaces. These are generated automatically from the C++ software and the user-specified constraints.

```
input      [31:0] in_enc_pos_a0_readdata;
input      [31:0] in_enc_pos_a1_readdata;
...
input      [31:0] in_enc_pos_a7_readdata;
output reg [15:0] out_current_iq_a0_writedata;
output reg [15:0] out_current_id_a0_writedata;
output reg [15:0] out_current_iq_a1_writedata;
output reg [15:0] out_current_id_a1_writedata;
...
output reg [15:0] out_current_iq_a7_writedata;
output reg [15:0] out_current_id_a7_writedata;
```

Figure 6: LegUp-generated Verilog for Motor Inputs/Outputs

```
output reg [31:0] master_AW_AWADDR_data_to_sink;
input             master_AW_ready_from_sink;
output reg        master_AW_valid_to_sink;
output reg [7:0]  master_AW_AWLEN_data_to_sink;
output reg [63:0] master_W_WDATA_data_to_sink;
input             master_W_ready_from_sink;
...
input      [31:0] axi_s_AW_AWADDR_data_from_source;
output reg        axi_s_AW_ready_to_source;
input             axi_s_AW_valid_from_source;
input      [7:0]  axi_s_AW_AWLEN_data_from_source;
input      [63:0] axi_s_W_WDATA_data_from_source;
output reg        axi_s_W_ready_to_source;
```

Figure 7: LegUp-generated Verilog for AXI4 Master/Slave Interface

FLOATING-POINT AND FIXED-POINT SUPPORT

The motor controller C++ software algorithm used standard floating-point operations. LegUp HLS includes support for floating-point IP cores that are optimized for the PolarFire FPGA DSP architecture both in terms of latency and FMax. LegUp also supports constraining the number of floating-point hardware units used for each operation type. These floating-point units will be shared among the required floating-point operations, which saves FPGA area but can reduce performance.

Figure 8 shows an example of the C++ template arguments for the LegUp floating-point multiplier core. By default, the floating-point cores will be configured automatically by LegUp based on the C++ floating point types. But if the user wishes to specify custom mantissa and exponent widths, for example in an AI application, they can directly instantiate these floating-point classes in their C++.

```
template <
    // Configure floating-point format:
    unsigned M_W,     // Mantissa width
    unsigned E_W,     // Exponent width
    class FPType = ap_fp<M_W, E_W>,

    // Configure DSP multiplier width: A x B
    unsigned MULT_W_A,     // Input A width
    unsigned MULT_W_B,     // Input B width
    >
FPType fmult(FPType A, FPType B) { ... }
```

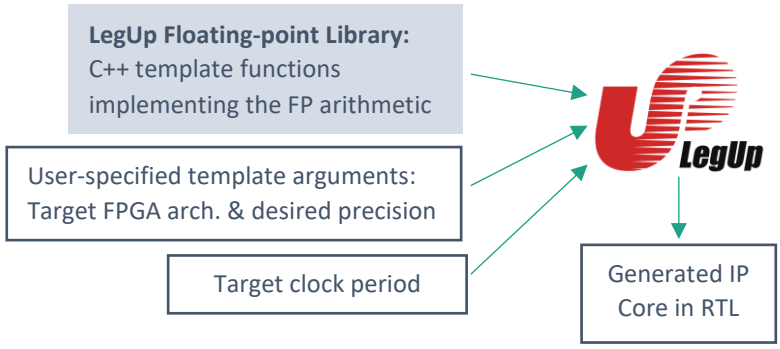

Figure 8: LegUp floating-point library. Example of floating-point multiplier template arguments

LegUp HLS also provides an easy way for engineers to specify fixed-point operations using the C++ fixed-point library (ap_fixed). LegUp HLS can support all major C++ operations in fixed-point. The fixed-point library allows an engineer to start with a floating-point implementation of the motor controller algorithm to match the microcontroller implementation, then migrate to fixed-point math if required to reduce the area on the FPGA. Fixed-point hardware units will consume significantly less area than floating point hardware units, while offering higher performance but more upfront design effort. Table 1 shows some examples of the LegUp ap_fixed types with their corresponding ranges.

Table 1: Example LegUp fixed-point types and ranges

Type	Quantum	Range
ap_fixpt<8, 4>	0.0625	-8 to 7.9375
ap_ufixpt<4, 12>	256	0 to 3840
ap_ufixpt<4, -2>	0.015625	0 to 0.234375

FPGA PERFORMANCE

The initial implementation of the motor controller C++ was using a push button approach to LegUp HLS. Next, the engineer spent a few weeks performing design space exploration using LegUp HLS user-constraints and tuning the Microsemi Libero synthesis options.

Table 2 shows the hardware quality of results for the motor controller hardware targeting a PolarFire FPGA (MPF300TS_ES-1FCG1152E). The cycle latency was almost halved and the Fmax was improved to meet the 200MHz target clock frequency. The final deterministic latency for the motor controller was approximately 2µs. The area of the motor controller was also reduced to fit inside the 300K LUT PolarFire FPGA.

Table 2: Motor Controller FPGA hardware quality of results

	Cycle Latency	Clock Frequency	Deterministic Latency	LUT Count
Initial	800	150 MHz	5.33 µs	320K
Optimized	**450**	**210 MHz**	**2.14 µs**	**200K**

We performed experiments comparing the original C++ motor controller to the improved FPGA version. We found that the FPGA-based motor controller had a 2.5-6X speedup in terms of latency compared to the ARM microcontroller, depending on the jitter of the microcontroller and real-time operating system.

CONCLUSION

In conclusion, LegUp HLS can offer an easy path to migrate existing C/C++ code targeting a micro-controller to a Microsemi PolarFire FPGA. For industrial control applications, an FPGA can offer motor controllers with significantly better deterministic latency and jitter compared to a microcontroller.

Key Takeaways

- LegUp HLS simplifies FPGA design by allowing you to program the FPGA using C/C++ software
- C/C++ code for microcontrollers can be migrated to a Microsemi FPGA using LegUp HLS
- Microsemi FPGAs offer 2.5-6X better latency/jitter for motor control vs. a microcontroller