Translating Modelica to HDL: An Automated Design Flow for FPGA-based Real-Time Hardware-in-the-Loop Simulations

Christian Köllner
FZI Forschungszentrum Informatik
Haid-und-Neu-Str. 10-14
76131 Karlsruhe
koellner@fzi.de

Torsten Blochwitz
ITI GmbH
Webergasse 1
01067 Dresden
blochwitz@itisim.com

Thomas Hodrius
SET GmbH
August-Braun-Straße 1
88239 Wangen/Allgäu
hodrius@smart-e-tech.de

Abstract

Advances in the development of electric vehicles challenge existing test methodologies and tools. In particular, hardware-in-the-loop test rigs to verify electric motor controllers require real-time drivetrain emulation with response times in the order of one microsecond. Field-programmable gate arrays can fulfill these requirements due to their high parallelism and the possibility to realize efficient and predictable I/O interfaces. We present an integrated methodology which translates Modelica models to VHDL hardware designs. Our methodology combines well-engineered algorithms from Modelica compilation and high-level synthesis for hardware. We demonstrate its capabilities using the example of a DC motor which was synthesized and implemented on a Xilinx Virtex-5 device.

Keywords: FPGA; High-level synthesis; VHDL; Hardware-in-the-Loop; Real-time

1 Introduction

Recent movement towards electric vehicles imposes new challenges on the development of drivetrains. Especially the verification of electric motor controllers (EMCs) using the hardware-in-the-loop (HiL) test methodology requires real-time simulation of the functional environment with low latencies. An EMC is an integrated device, consisting of an electronic control unit (ECU) and a power stage. The ECU implements current, acceleration and/or speed control and safety functions whereas the power stage generates the motor currents. The test rig wires the EMC to an emulator, as shown in Figure 1. An electric motor emulator (EME) emulates an electrical motor under real conditions, including position feedback and other sensor signals. If needed, a power stage recreates the original currents and voltages.

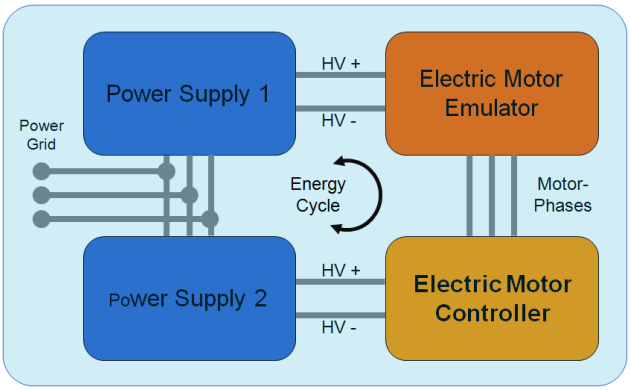

Figure 1: EMC test bed schematic

Due to the dynamic electric behavior of the motor, the model iteration rate has to be in the order of one microsecond. Since such real-time requirements are hard to meet using software solutions, HiL emulators of electric machines typically involve a field-programmable gate array (FPGA) which carries out time-critical computations. FPGAs are highly parallel reconfigurable hardware circuits which are well-suited for high-performance real-time computations. However, their programming model is fundamentally different from general-purpose computing. This fact makes current modeling environments lack an integrated flow from model to hardware. Although Modelica has proven to be an effective language for describing electric hybrid drivetrains [1], there is currently no tool support for compiling Modelica to FPGAs.

Our contribution tries to close this gap. We propose an integrated methodology for compiling Modelica models to an FPGA configuration. The implementation is realized and validated using SimulationX. Our approach combines well-known methodologies from both differential-algebraic equation (DAE) processing and high-level synthesis (HLS). We employ inline integration to obtain a compact calculation rule which can be efficiently mapped to

hardware. Moreover, we incorporate parametrizable circuit templates (so-called IP cores) to solve common subproblems during the mapping process.

Our paper is organized as follows: Section 2 investigates related work from commercial and academic perspective. Section 3 gives a short explanation of FPGA functionality and the programming model. Based on the specifics of FPGA operation, section 4 states the requirements to achieve an integrated, automated design flow from model to hardware. Section 5 explains these implications on model entry. In section 6, we discuss the overall design flow from Modelica to hardware. Section 7 presents the characteristics of an exemplary direct current (DC) motor model which was translated to hardware. Finally, section 8 concludes the paper and gives an outlook to future work.

2 State of the Art

Electric motor controllers used in automation and automotive applications combine controller and power stages in one device. Testing and verifying EMCs in an HiL environment is challenging, since the behavior of the electric motor must be rebuilt true to original. Otherwise, the EMC would diagnose a malfunction and enter failure mode. The interface between the EMC and HiL system can be realized on a mechanical, electric power, or signal level [2].

On the mechanical level, the original electric motor is connected to the EMC. Another motor is flanged and applies the mechanical load, computed online by a simulation model. Such dynamometer test stands (as shown in Figure 2) are expensive to build, hard to control, and not flexible in usage.

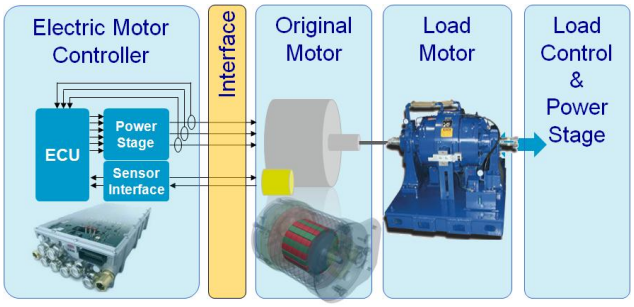

Figure 2: Dynamometer test stand

Interfacing on the signal level requires cutting the connection between the controller and power stage. This “cracked ECU” approach requires knowledge of controller internals. The behavior of the electric motor and its load is computed by a fast microprocessor or an FPGA device. The computed current-sensing signals are fed back to the ECU along with other simulated sensor signals (shown in Figure 3). This approach excludes the power stage from test and verification.

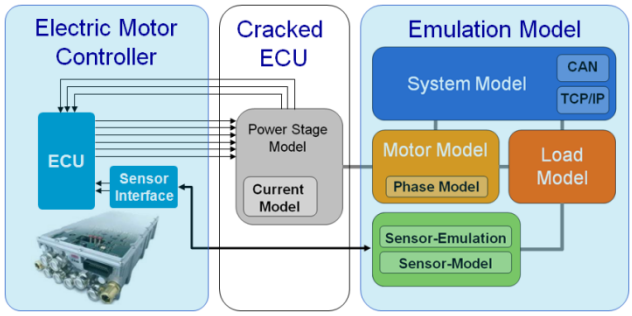

Figure 3: Cracked ECU test bed

When interfacing at the electric power level, the electric current is generated by special power electronics and fed back to the power stage of the unit under test. This methodology is referred to as Power Hardware-in-the-Loop (P-HiL). The SET EME realizes this methodology, reproducing proper power loads [3] without rotating parts (see Figure 4). The interface to the EMC is identical to the real motor. It consists of the motor phases and position sensor signals (e.g. resolver), if needed. Its applications vary from small servo controls with less than 100 W to electric power trains with several 100 kW. A wide range of motor types and rotor position interfaces is supported.

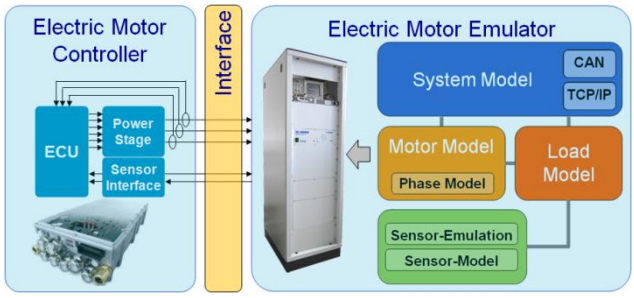

Figure 4: Electric motor emulator test bed

To achieve realistic emulation behavior, high switching frequencies of the EME power amplifiers are needed. This is especially important when operating at high rotational speeds and to emulate dynamic behavior, such as speed ramps. Hence, for these use cases special power amplifiers with application-dependent switching frequencies up to 800 kHz are deployed. Controlling the power amplifier requires input/computation/output latencies of 1.25 µs.

Both the cracked ECU approach and P-HiL typically rely on FPGA-based implementations of the motor simulation. In absence of a suitable toolchain these models are commonly coded by hand, using a hardware description language (HDL). Examples

DOI 10.3384/ecp12076355

include a commercial model of inverter and permanent magnet synchronous machine (PMSM) [4], a DC motor [5], a squirrel-cage induction machine [6] and a generic implementation which covers an exhaustive set of AC motor types [7]. Yet, there is no general agreement on the type of arithmetic: most models incorporate fixed point arithmetic [5-7] whereas one contribution relies on floating point [4]. The development of such models is generally error-prone and time-consuming, especially if complex models (e.g. a nonlinear model of synchronous motors) or detailed drivetrains, including clutches and rigid end stops, must be realized.

In reference [8], HDL Coder from The Mathworks was used to implement a Simulink DC motor model on an FPGA. This toolchain is restricted to Simulink models without continuous states. User interactions and reformulation of the model are necessary to achieve a fast and synthesizable FPGA design. A similar approach is presented in [9]. The authors create a Matlab/Simulink model of a permanent magnet synchronous machine using the Xilinx System Generator (XSG) blockset. Again, the methodology requires the engineer to model at the hardware level. Reference [10] presents an approach to generate fixed point code from Modelica. It is capable of exporting Mitrion-C code for FPGA applications, but no details are given on how the transformation towards an FPGA design works, and no FPGA implementation is presented.

3 FPGA Fundamentals

3.1 Overview

An FPGA is an integrated digital circuit whose functionality is programmable after manufacturing. To achieve programmability, FPGAs generally provide configurable combinatorial logic blocks and memory elements. These can be wired in a large variety of ways. By combining both primitives – logic and memory – it is theoretically possible to recreate any digital circuit. Recent FPGAs are computationally equivalent to roughly 20 million logic gates. Most devices provide additional built-in macro cells for frequent tasks, such as hardware multipliers and static RAM.

3.2 Programming FPGAs

In most cases, a hardware description language (HDL), such as VHDL and Verilog is used to describe the intended digital circuit. Vendor-specific toolchains transform the described design into a netlist representation, map it to device primitives, optimize the geometric placement of that mapping and finally produce a programming file which configures the FPGA.

HDLs also define control-flow statements, which in fact turn them into general-purpose programming languages. However, these constructs are primarily intended for simulation/verification purposes and are mostly not supported for circuit modeling. A HDL description is said to be synthesizable, if it is possible to represent it by a functionally equivalent netlist. Therefore, synthesizability is a mandatory prerequisite to FPGA configuration. Particularly, analog-mixed signal extensions of VHDL (VHDL-AMS [11]) are generally not synthesizable.

3.3 Example

The following example is kept in VHDL and illustrates the impact of a specific notation on the synthesized circuit. Assume that we want to transform the following computation into a digital circuit:

$$r \coloneqq a \cdot b + c \cdot d$$

If we encode all operands using a fixed point representation, there is a straightforward VHDL translation of the given calculation rule:

```
r <= a * b + c * d;
```

This implementation implicitly prescribes a combinatorial, fully-spatial realization. Synthesis infers a circuit which consists of two multipliers and one adder. Although this is the fastest possible realization, it may miss a design goal: Embedded in a synchronous design, this circuit may drop the achievable clock rate because of its combinatorial path. This can be avoided by buffering multiplication results in intermediate registers. If we need to save FPGA resources, a longer computation time might be acceptable. In this case, the calculation can be described as finite state machine (FSM):

```
Compute: process(Clk)
begin
  if (rising_edge(Clk)) then
    case state is
      when Mul1 => tmp1 <= A * B;
                   state <= Mul2;
      when Mul2 => tmp2 <= tmp1;
                   tmp1 <= C * D;
                   state <= Add;
      when Add =>  R <= tmp1 + tmp2;
                   state <= Mul1;
    end case;
  end if;
end process;
```

This implementation spreads the computation across three clock cycles. Since at most one multiplication happens per clock step, synthesis will share

resources: the novel circuit requires only one multiplier instead of two.

Changing the computation to floating point arithmetic requires the designer to use either special libraries or to interface the design with an IP core. IP cores are pre-built circuit templates with well-defined functionality which are either supplied by the device manufacturer or third-party vendors. This option usually provides better performance and detailed hardware tuning parameters. IP cores are also available for advanced mathematical operators, such as division, square-root and trigonometry.

High-level synthesis (HLS) is a field of research which addresses automated transformation of formal behavioral descriptions (mostly C/C-like programming languages) to hardware [12]. The transformation is constrained by requirements, such as resource consumption and time. Despite commercial tools are available, their success is limited. This is not only due to their high asset costs but also due to the user's uncertainty with respect to the quality of results [13]. Their effectiveness varies strongly with problem domain and coding style. Our contribution exploits the ideas of high-level synthesis. By tailoring its methodologies to the specific area of physics simulation we get a domain-specific approach which is able to meet our resource and timing requirements.

4 Requirements

The intended application imposes several implications on the chosen approach and equation processing. The following subsections discuss them in more detail.

4.1 Inline integration

Typical code generation from Modelica relies on a software infrastructure which distinguishes solver and model. The solver is in control of the overall simulation and employs callback functions to transfer control to the model-specific evaluation of derivatives. A tight interaction with strong data dependencies connects the solver and model components. This interaction is entirely time-multiplexed, exposing only little potential to parallelize [14]. Establishing a spatial distinction between solver and model on the FPGA would produce hardly any benefit. Thus, it is preferable to synthesize a self-contained calculation rule which encompasses the overall computation to carry out one integration step. This technique is called inline integration [15].

4.2 Real-time execution

During real-time computation, two conditions must be fulfilled: First, the computation time to perform a single integration step must be bounded and predictable. Second, the integration step size must have a lower bound. Since data acquisition and output of an HiL emulator usually happen at a fixed sample rate, it is even desirable to employ a fixed-step integration method.

Moreover, Modelica events must be used with care. Due to the fixed step size, the precise time instance of state events cannot be localized. Events are shifted to the end of the current integration step. In our case, this should not lead to problems because the step size used on a FPGA device is small compared to common processor-based HiL systems.

At event instances, a Modelica simulator carries out event iteration. The model is recomputed at the same time instance until discrete variables do not change anymore. The number of necessary event iteration steps cannot be predicted. Hence, the real-time condition might be violated. For that reason the model should be built in such a way that avoids event iterations. The Modelica compiler should recognize if the model requires event iterations (e.g. due to algebraic loops over discrete variables) and inform the user.

Implicit integration methods as well as algebraic constraints can necessitate the solution of non-linear systems of equations during simulation. Since such systems are usually solved by numerical methods, it is not guaranteed that the solution algorithm converges within a bounded number of iterations. Therefore, non-linear systems of equations should be avoided by the model. Ultimately, Modelica allows for embedding arbitrarily complex algorithms into any computation. It is the designer's duty to ensure that they have bounded execution times.

4.3 Choice of arithmetic

PC-based simulations usually rely on IEEE 754 floating point data types. Although this type of arithmetic can be implemented on an FPGA, it has weaker performance and higher resource consumption compared to equally-sized fixed point data. The situation changes if an adequate fixed point representation would require disproportionately large word sizes. FPGAs support "uncommon" word lengths (which are not powers of two). An appropriate synthesis flow should exploit these facts and support both – possibly mixed – floating point and fixed point arithmetic operators.

DOI 10.3384/ecp12076355

4.4 Sustaining domain-specific knowledge

A key challenge is to identify the level of abstraction at which a preprocessed model should be handed over to the hardware-centric synthesis flow. Physical computations involve many subproblems which can be directly mapped to IP cores. Examples are mathematical operators, such as sine/cosine, square-root and the absolute value function. Calls to such functions should be preserved in order to give the synthesis flow a chance to adopt dedicated hardware components. Another example is the solution of linear equation systems, which is necessary to simulate models with algebraic loops. In the past, numerous high performance linear solvers for FPGAs were developed [16-19]. To enable their usage, model preprocessing should keep linear systems instead of inserting a specific solver algorithm.

4.5 Minimizing computation effort

Compiler optimizations, such as common subexpression elimination and exploiting algebraic identities are particularly important when targeting FPGAs. Device resources are limited, and each additional operation will affect either performance or area. Conversely, the slimmer design will fit on the smaller and cheaper device. Although it is possible to generate FPGA solvers for linear or nonlinear equation systems, avoiding such systems helps to keep the design compact.

5 FPGA-Aware Modeling

As implied by the special capabilities and limitations of FPGAs, the user should adhere to certain modeling guidelines when designing models for FPGA execution. Violating them can cause the translation to fail or lead to bloated hardware designs. We implemented a Modelica library prototype which contains frequently used elements for modeling electrically driven drivetrains and takes these aspects into account. Using this library and considering some modeling guidelines will lead to synthesizable designs faster than using the general purpose Modelica Standard Library or the SimulationX libraries. Figure 5 shows the structure of the library.

Special considerations were necessary for the dry friction model. Real-time motor emulation requires a robust friction model that reproduces correct stiction behavior. Usage of the friction element should neither result in a combined discrete continuous system

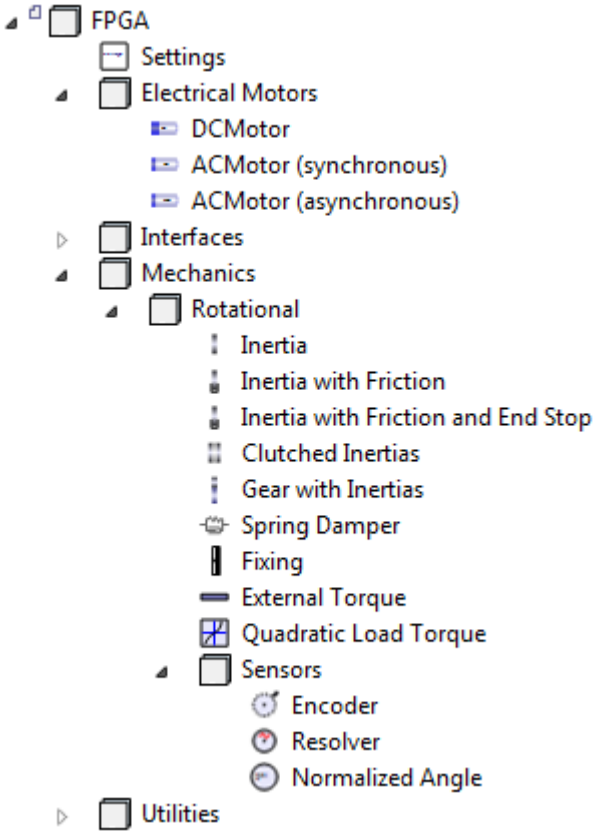

Figure 5: Screenshot of the library structure

of equations nor cause event iteration. By combining friction behavior with inertia, the resulting friction torque and the new discrete state can be computed explicitly. The solution of a system of equations and event iteration become obsolete. This approach is used by the library elements "Inertia with Friction" and "Clutched Inertias."

Further systems of equations can be avoided, if some modeling guidelines are obeyed. For example, an inertia element should be placed between elements which introduce a torque to the system (spring dampers, motors, loads). Inertia elements should not be strung together. These rules do not restrict the model features which can be represented by the library. Only the way in which models are to be built up is slightly constrained. If the rules are violated and systems of equations persist, the Modelica compiler generates appropriate warnings.

6 Compilation and Synthesis

Figure 6 illustrates the overall design flow which is implemented by our software prototype. The following subsections explain the procedure step-by-step.

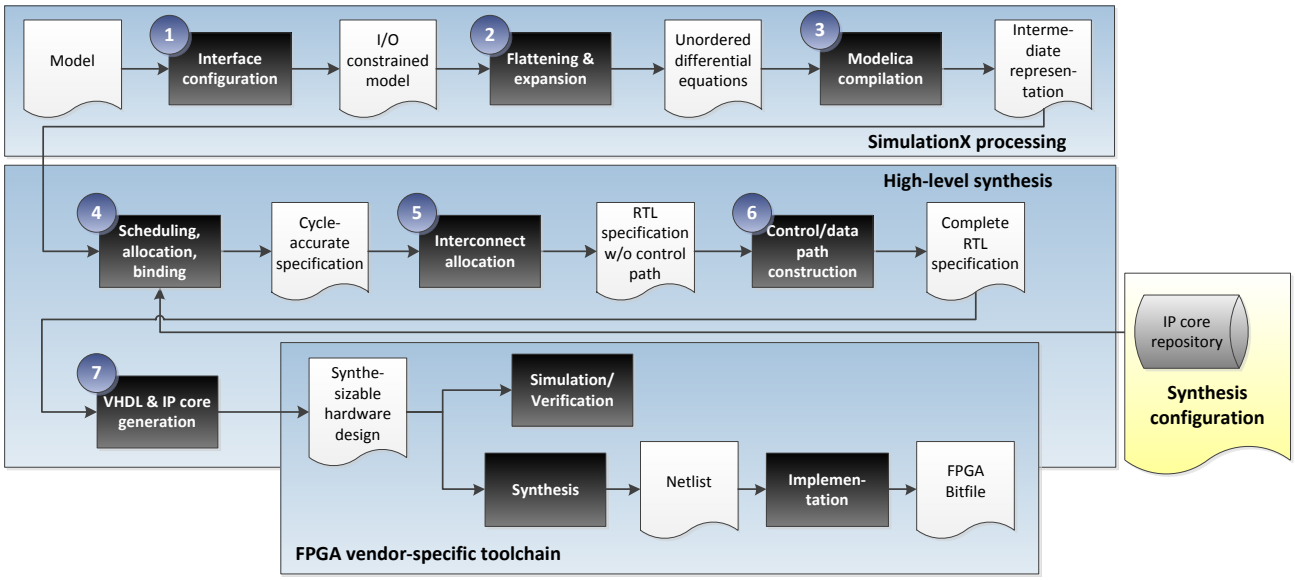

Figure 6: Overall model compilation and synthesis flow

6.1 Preparation of the model

First, the interface of the model is to be specified. The user selects inputs, outputs and parameters which shall be available on the FPGA. Inputs, outputs and parameters will become VHDL ports of the generated hardware design unit.

6.2 Modelica compilation

Most stages of the compilation process are not specific to FPGA code generation. Some steps after flattening (step 2) of the Modelica model are specific according to the requirements of Section 4. In order to reduce the complexity of the resulting VHDL code, loops of known and constant range are unrolled, and equations of higher dimension are expanded. Furthermore, equations and variables which do not influence the selected model outputs are removed. Functions are inlined since function calls would bloat the hardware by requiring an execution stack.

Since state events cannot be precisely located anyway, all conditions are covered implicitly by the `noEvent(…)` function. Algebraic loops containing discrete variables would require event iteration. This case is detected by the SimulationX Modelica compiler which displays an appropriate message. The integration formulas for computing the values of continuous states from their derivatives are introduced in an early stage of the compilation process. This enables symbolic simplifications on these parts of the algorithm too. We use Euler's forward integrations method, which is a good compromise between computational effort and stability.

The SimulationX compiler produces either C code or a bytecode representation for simulation. We extended its capabilities to generate an XML-based assembler-like intermediate representation to be processed by the FPGA-centric tooling. The instruction set was chosen to match hardware capabilities. For example, op-codes for common mathematical operators exist which allow fixed point and floating point operands of arbitrary sizes. The resulting behavioral description basically contains two procedures:

- Initialization part
- Iteration part

The initialization part is an algorithm which computes initial variable values from all model parameters. It may also perform some non-trivial computation, such as iteration to find consistent state values. Since it is executed only once (at the beginning of the simulation), it is not time critical. The iteration part contains the actual computation which is performed during simulation. It is a function of model inputs and state, transforming those quantities into output and new state. This algorithm gets iterated for each time step and therefore must have a predictable and bounded execution time.

6.3 Scheduling

When mapping an algorithm to hardware, three fundamental tasks need to be distinguished:

- Scheduling assigns execution time (i.e. clock tick) to each instruction.
- Allocation determines which hardware functional units (FUs) to instantiate and in which quantities. For each instruction there must be at least one FU which can execute it.

DOI 10.3384/ecp12076355

- Binding assigns each instruction to a FU. It must ensure that no two instructions are assigned to the same FU at the same time. It should also account for interconnection costs which are induced by its choice.

Superscalar processors perform scheduling and binding dynamically (allocation is determined by manufacturing). They analyze the incoming instruction stream for data dependencies and schedule them automatically. A tremendous amount of logic is required to achieve such functionality. Recreating superscalarity on an FPGA is not a viable option. Instead, a static schedule is pre-computed. Another advantage is that execution time is completely predictable.

Our prototype employs the force-directed scheduling algorithm (FDS, [20]). FDS is a time-constrained approach which exploits instruction-level parallelism. Its input is a control-/data-flow graph (CDFG) and a time constraint. Upon success, it returns a schedule which heuristically minimizes the amount of required FUs. Generous time constraints lead to fewer FUs and therefore reduce resource consumption. Figure 7 shows the scheduled CDFG of a DC motor model. The model itself will be introduced in Section 7. Each rectangle depicts a variable/constant load/store instruction whereas each circle depicts an arithmetic operation. In the given example, multiplication was configured to last three cycles, addition/subtraction two cycles.

6.4 Allocation and binding

Allocation and binding are downstream stages to scheduling. The schedule determines the minimum amount of FU instances of each kind which are required. It does not prescribe which instance will actually execute a specific instruction. Binding multiple staggered instructions to the same FU is called resource sharing. Obviously, sharing is desirable, since it helps to reduce the area of the overall hardware design. On the negative, it can lead to performance degradation. Input multiplexers will be necessary to select from different operands. They increase the combinatorial delay and may affect the clock rate. If the operand sources get placed at far-off chip locations, routing delays will further drop the clock rate.

We employ a heuristic to tackle the problem. Our algorithm sequentially assigns each instruction to an FU by either allocating a new FU instance or reusing a previously allocated one. In case of reuse, assignments that reuse existing interconnect are preferred. If reusing any previously allocated FU would require overly large multiplexers, a new FU is allocated instead.

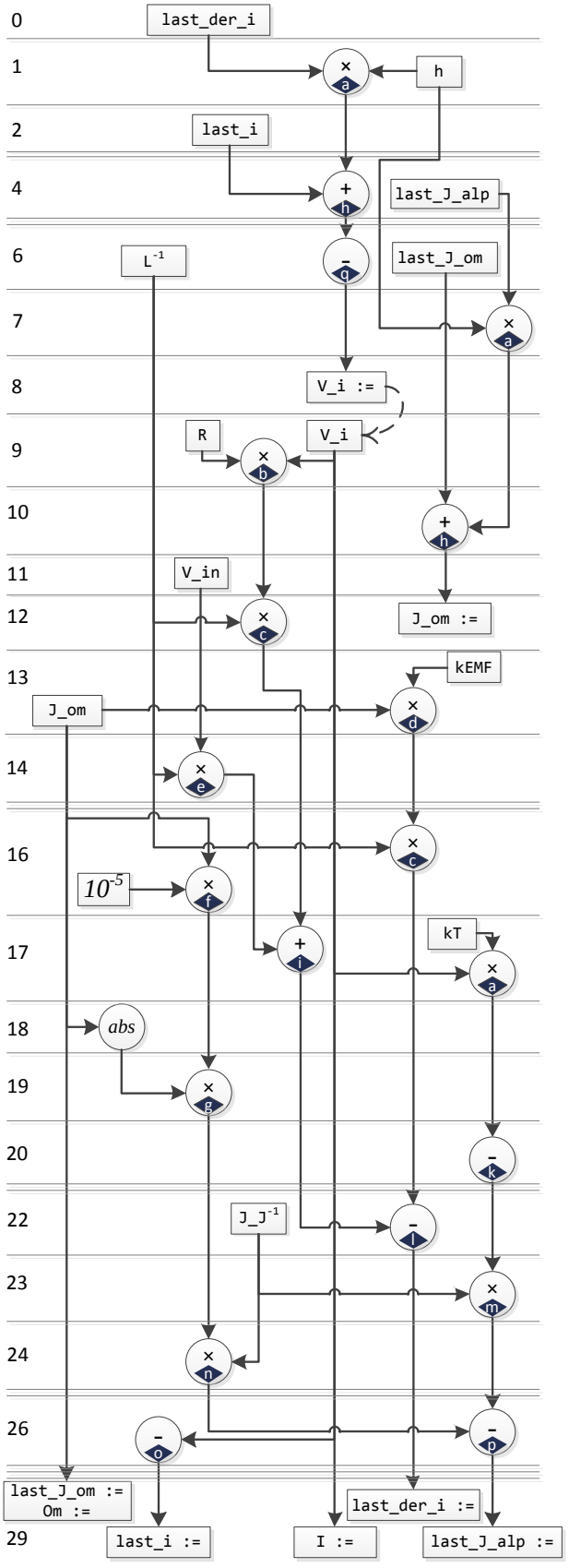

Figure 7: Scheduled and bound CDFG of a DC motor with quadratic friction

The result of allocation/binding the DC motor CDFG is shown in Figure 7: Characters inside diamonds enumerate the FU instances which the operations were mapped to. The operating point was set to spare resource sharing in favor of performance. Moreover, the outcome suggests that the binding procedure was able to identify the most economic candidates for resource sharing: The multiplications in control steps 1 and 7 are mapped to the same hardware multiplier. This is reasonable, since both operations share the common operand h.

The set of instantiable FUs is provided by an IP core repository. It must hold an according FU type for each kind of instruction. The repository is assembled from hand-written cores as well as vendor-specific IP cores. The latter are shipped with the FPGA toolchain and provide off-the-shelf implementations of complex arithmetic units, such as floating point operators, trigonometric operators and square-root.

6.5 Interconnect allocation

Once the complete instruction stream is scheduled and bound to appropriate FU instances, an interconnect network is constructed. It is responsible for routing operational results to their target FUs. The schedule may also require the network to buffer intermediate results. This happens if a result is not processed within the same clock step it was produced. Thus, the interconnect network is composed of multiplexers and flip-flops.

We developed an incremental merging heuristic which considers both register count and multiplexer size. An initial solution is constructed by assigning each instruction outcome to an individual storage register. Afterwards, register pairs are iteratively selected and merged whereby the merging decisions try to balance the multiplexer sizes of the overall interconnect structure.

6.6 Control path construction

The control path is a hardware unit which conducts the temporal interaction of all data path components. This includes asserting handshake signals and setting an input selection for each multiplexer.
After the scheduling, allocation/binding and interconnect allocation steps have been completed, the control path is completely specified in its behavior. It just needs to be expressed by an explicit implementation. In the scope of this contribution, an FSM representation was chosen. Each control step of the schedule constitutes one state. A VHDL process steps the state forward with each rising clock edge. Another combinatorial process computes appropriate settings for handshake signals and multiplexers, based on the current state. FSM descriptions are recognized by FPGA synthesis tools. These try to infer an optimal hardware representation for the given FSM. To support optimal inference, we represent the state variable using a VHDL enumeration data type. This gives VHDL synthesis a chance to choose an optimal state encoding [21].

6.7 Source code generation

The generated design involves VHDL source code, but also parameterization scripts for vendor-specific IP cores which were instantiated from the IP core repository. Although our approach is conceptually independent of device technology, the generated design is technology-dependent if it involves vendor-specific IP cores. So far, Xilinx FPGAs are supported.

7 Results

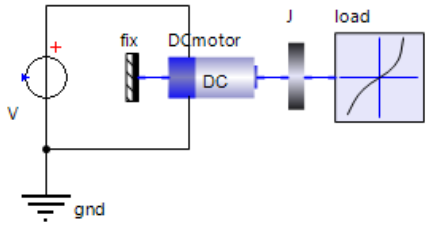

Figure 8: Sample model

We demonstrate the transformation process using the model of a DC motor (Figure 8). The motor is connected to an inertia and a load torque with quadratic dependency on speed. This is the typical behavior of a fan. The voltage at the voltage source (`V.v`) is used as input, current (`V.i`) and motor speed (`J.om`) are the outputs.

The generated VHDL code is synthesizable on an FPGA. All `Real` variables of the Modelica model are represented by fixed point numbers with 32 bits precision at inputs and outputs. Intermediate results are processed at higher precision. The proportioning into integral and fractional part was done individually for each quantity, with respect to its range of values. Figure 9 compares the output values of the VHDL code to the simulation results, using the Euler forward method and a step size of 1 µs. The motor is fed by a voltage jump of 12 V. The simulation results are reproduced with sufficient accuracy. Minor deviations are caused by the fixed point representation of the variables in VHDL.

DOI 10.3384/ecp12076355

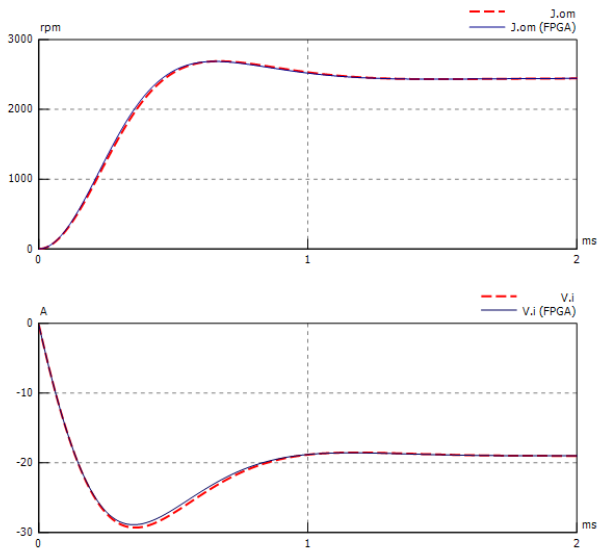

Figure 9: Simulation results (red) and FPGA results (blue)

To achieve synchronized data transfer, the design unit is equipped with additional handshake signals. These signals control initialization and model evaluation. Figure 10 shows the basic structure of the resulting hardware design unit. Model initialization and evaluation are separated into two individual FSMs which share a register bank. Asserting the `Init` signal causes the initialization procedure to capture and preprocess all parameters. This includes precomputing the reciprocals of moment of inertia (`J_J`) and rotor coil inductivity (`L`). Since division is a costly hardware operation, this step improves runtime performance.

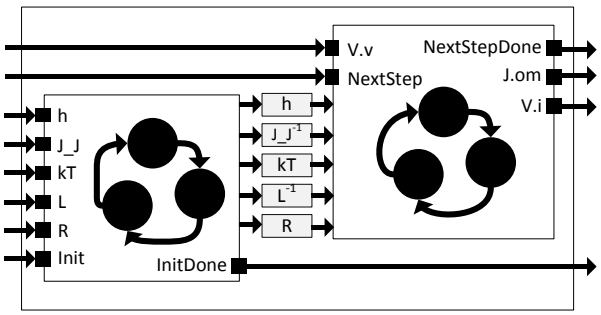

Figure 10: Architectural overview of the generated hardware design unit

Figure 11 shows the interplay of all handshake signals. Once the initialization is complete, model evaluation is controlled by the signals `NextStep` and `NextStepDone`. As noted in Section 3, the latencies of arithmetic operators are design parameters and affect computation time, clock rate and chip area. Although low latencies reduce the overall computation time, this usually comes at the cost of clock rate.

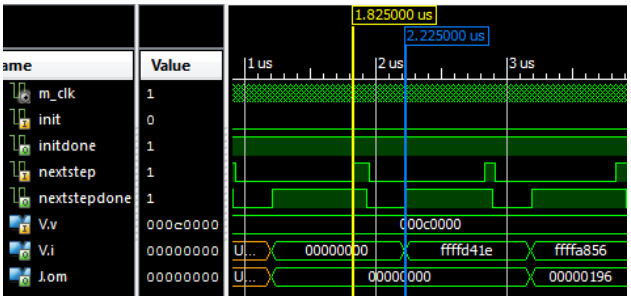

Figure 11: Initialization and runtime behavior of the design unit

The goal was integrate the generated design into SET's EME hardware. Due to the hardware requirements, the design must achieve a clock rate of 100 MHz on a Virtex-5 LX110 device and complete any model evaluation within 1 µs. Consequently, the schedule of the overall computation (an example is given in Figure 7) must not exceed 100 clock cycles. Using three different configurations, we generated corresponding design variants.

Table 1: Characteristics of the generated designs

L_{mul}	L_{add}	L_{tot}	*Slice usage*	F_{max} *(MHz)*
1	1	17	5%	89
3	2	30	6%	105
9	3	43	6%	102

Table 1 summarizes the characteristics of the generated designs. The columns depict, from left to right: 32×32 bit multiplication latency, 32 bit addition latency, schedule length of model evaluation, slice usage and maximum achievable clock rate after placing and routing the design on the target device. Slice usage is an approximate measure of the chip area which is consumed by the hardware design.

Although the first variant provides the fastest computation time, it does not reach the target frequency of 100 MHz. The remaining two alternatives are both viable. However, the second option is superior compared to the third one. It provides an overall input/output latency of 400 ns at 100 MHz, including handshake-induced wait cycles. This is more than sufficient to meet the requirement of 1 µs.

8 Conclusions and Outlook

The toolchain approach described in this document will allow the efficient realization of flexible electric motor emulators. The combined model of motor and drivetrain is built using the FPGA-aware Modelica library. The resulting model is automatically transformed to an FPGA design. The FPGA controls the

EME hardware. Although the computation needed to accomplish a DC motor simulation is manageable, its hardware implementation introduces many new degrees of freedom: architecture, scheduling, resource allocation and binding, parameterization of arithmetic data types and corresponding hardware operators. Designing such hardware manually is a complex and time-consuming task. If the first draft does not meet the design goals, alternative implementations need to be explored, multiplying the effort. This contribution will allow an EME operator to model an application using SimulationX and link it directly to the hardware – even with moderate FPGA knowledge.

One of the next steps in our joint research project is the semi-automatic determination of the optimum fixed point representation for the model variables. A compromise between accuracy and occupied FPGA resources is to be found. It is also conceivable to realize a hybrid approach which combines fixed and floating point arithmetic in a single model, based on cost/accuracy tradeoffs.

Another field is the convenient subdivision and numerically robust reconnection of sub models. This becomes eminent as soon as a complex model exceeds FPGA resources. In this case, slow sub models could be computed on a microprocessor, and only the fast parts run on the FPGA.

The presented work is not restricted to electric motor emulation. It would be highly interesting to evaluate it for implementing sophisticated control algorithms on FPGA devices, based on Modelica models.

9 Acknowledgment

The presented work was accomplished within the project SimCelerate, which is funded by the Federal German Ministry of Education and Research (grant no. 01M3196C).

References

[1] Winkler D., Gühmann C. Hardware-in-the-Loop simulation of a hybrid electric vehicle using Modelica/Dymola. Yokohama, Japan: The 22nd International Battery, Hybrid and Fuel Cell Electric Vehicle Symposium & Exposition, Japan Automobile Research Institute, 2006

[2] Köhl, S., Himmler, A.: Anwendungen und Trends bei der HIL-Simulation. Simulation und Test in der Funktions- und Softwareentwicklung für die Automobilelektronik II, expert verlag, Berlin, 2008, pp. 203-217

[3] SET GmbH Echtzeit-Emulation beschleunigt die Entwicklung, Funktions- und Leistungstests von E-Motor-Steuergeräten, Makt&Technik Vol. 27, 2010-05

[4] Liebau H., Jakoby H., Crepin, J.: HiL-Simulation elektrischer Fahrzeugantriebe. Automotive Engineering Partners, Vol. 2011-05

[5] Zhou Y. J., Mei T. X., FPGA based real time simulation of electrical machines, Proc. 16th IFAC World Congress, 2005

[6] Matar M., Iravani R., Massively parallel implementation of AC machine models for FPGA-based real-time simulation of electromagnetic transients, IEEE Transactions on Power Delivery, Vol. 26, No. 2, pp. 830-840 2011

[7] Chen H., Sun S., Aliprantis D., Zambreno J., Dynamic simulation of electric machines on FPGA boards, Electric Machines and Drives Conference, 2009

[8] Köllner C., Yao H., Müller-Glaser K. D.: Entwurfsmethodiken zur Echtzeitsimulation physikalisch motivierter Modelle auf FPGAs: Eine Fallstudie. Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV), 2011.

[9] Dufour C., Belanger J., Lapointe V., and Abourida S., "Real-time simulation on FPGA of a permanent magnet synchronous machine drive using a finite-element based model," Symposium on Power Electronics, Electrical Drives, Automation and Motion (SPEEDAM), 2008

[10] Nordström U., López J. D., Elmqvist H., Automatic Fixed-point Code Generation for Modelica using Dymola, Proc. Intl. Modelica Conf., 2006

[11] VHDL Analog and Mixed-Signal Extensions, IEEE Std. 1076.1-1999

[12] Coussy P., Morawiec A. High-Level Synthesis: from Algorithm to Digital Circuit. Springer Netherlands, 2010.

[13] Grant M., Smith G. High-Level Synthesis: Past, Present, and Future. Journal: IEEE Design and Test of Computers. Vol. 26, pp. 18-25, 2009.

[14] Nyström K., Aronsson P., Fritzson P., Parallelization in Modelica, Proc. 4th Intl. Modelica Conf., 2005

[15] Elmqvist H., Otter M. and Cellier F.E.: Inline Integration: A New Mixed Symbolic/Numeric Approach for Solving DAE Systems. Proc. ESM'95, European Simulation Multiconf., 1995.

[16] Johnson J., Chagnon T., Vachranukunkiet P., Nagvajara P., Nwankpa C., Sparse LU Decomposition using FPGA, International Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA), 2008

[17] Daga V., Govindu G., Prasanna V., Gangadharpalli S., Sridhar V., Floating-point based block LU decomposition on FPGAs, Proc. Intl. Conf. on Engineering Reconfigurable Systems, 2004

[18] Gonzalez J., Núñez R. C. LAPACKrc: Fast linear algebra kernels/solvers for FPGA accelerators. Journal of Physics: Conference Series. 2009

[19] Fischer T., Entwurf eines FPGA-Cores zur Simulationsbeschleunigung zeitkontinuierlicher Modelle im HiL Kontext. GI Fachtagung Echtzeit 2011 - Herausforderungen durch Echtzeitbetrieb, 2011

[20] Paulin P. G., Knight J. P. Force-directed scheduling in automatic data path synthesis. Proc. 24th ACM/IEEE Design Automation Conf. (DAC), 1987

[21] Xilinx, Inc. Synthesis and Simulation Design Guide. UG626 (v13.4), 2012

 DOI 10.3384/ecp12076355