Circuit Solver C++ Code Generator Command Line Interface Tool

User Guide

Open Real-Time Simulation Framework

https://github.com/OpenRealTimeSimulation

Matthew Milton

December 23, 2021

Courtesy to Michele Difronzo, Dhiman Chowdhury, Mark Vygoder, and Andrea Benigni for their contributions to the project and some figures presented in this document.

Contents

Chapter 1

Introduction

1.1 Overview

As part of the Open Real-Time Simulation (ORTiS) framework, the C++ Circuit Solver Code Generator Command Line Interface (CLI) tool, shortened to solver codegen tools, generates C++ source code for solvers of multi-physics networked systems such as electrical, power electronic, and energy conversion systems. These systems are defined with a netlist file which is input to the tool. The solver algorithm used in the generated solvers is the Latency-Based Linear Multi-step Compound (LB-LMC) method.

The solver codegen tools create simulation solvers of given systems as C++ function definitions tailored to the given system expressed by a plain-text netlist. The function definitions are created by the tools through first parsing the parameters and component listings in the given netlist. Then, the tools search for and customize model code taken from a built-in or user provided library of component model definitions using given parameters. Next, the tools generate C++ code to solve system equations and read/write I/O signals. Finally, the tools concatenate the resultant model and system solver code into a C++ solver function definition. These C++ solver definitions can then be passed to High-Level Synthesis (HLS) tools, such as Xilinx Vivado HLS, to be systematically converted to FPGA execution cores described in Hardware Description Languages (HDL) such as VHDL or Verilog. These FPGA execution cores of the solvers can then be incorporated in FPGA-based real-time simulator designs to simulate the system of interest. Moreover, the generated solver definitions can also be used for high performance offline simulations or CPU-based real-time simulation as well.

This document serves as a basic user guide for how to use the CLI tool and to define netlists of systems. Information on the generated solver function definitions and how to utilize them in offline testbenches, and to synthesize them for FPGA execution, are also provided.

1.2 Where to Download

The Solver Code Generator CLI tool (solver codegen tool) is part of the Open Real-Time Simulation (ORTiS) framework. Tools and development libraries of the ORTiS framework are found here on Github:
`https://github.com/OpenRealTimeSimulation`

Presently, the solver codegen tools are publicly distributed only as C++ source code, with binaries occasionally released. C++ source code of the tools can be compiled with C++14 compliant compiler suites such as GCC or Mingw-w64, among others. The tools may have static dependencies on third-party libraries which must be downloaded from their respective sources and configured within your C++ build environment. See Chapter 8 for instructions on how to build the sources.

1.3 Licensing

The LB-LMC Solver C++ Code Generator CLI tool is licensed under the GNU General Public License (GPL) v3.0 (`https://www.gnu.org/licenses/`).

Librar(y/ies) used by the tool are licensed under their own terms by their respective owners.

1.4 Literature

The LB-LMC solver method, and the codegen solver tools generating solvers using this method, has been presented in several publications over the years. The following are papers in which the LB-LMC method, the codegen tools, and their applications have been proposed.

The main papers on the solver codegen tools:

M. Milton, A. Benigni, "ORTiS solver codegen: C++ code generation tools for high performance, FPGA-based, real-time simulation of power electronic systems," *SoftwareX*, vol. 13, Jan. 2021. Available: [`https://www.sciencedirect.com/science/article/pii/S2352711021000054`]

M. Milton, A. Benigni, "Software and Synthesis Development Libraries for Power Electronic System Real-Time Simulation," *2019 IEEE Electric Ship Technologies Symposium (ESTS)*, Arlington VA, 2019, Aug. 2019.
Available: [`https://ieeexplore.ieee.org/document/8847940`]

The LB-LMC solver method was first proposed in this paper:

A. Benigni and A. Monti, "A Parallel Approach to Real-Time Simulation of Power Electronic Systems," *IEEE Trans. Power Electronics*, vol. 30, no. 9, pp. 5192–5206, Sept. 2015.
Available: [`https://ieeexplore.ieee.org/document/6918539`]

The use of the LB-LMC method and the solver codegen tools for power electronic system FPGA-based real-time simulation are presented in the following papers (earliest to latest):

M. Milton, "A Comparison of FPGA Implementation of Latency-Based Solvers for Power Electronic System Real-Time Simulation," M.S. Thesis. University of South Carolina, Columbia, SC, 2016. Available: [`http://www.proquest.com/`], [`https://scholarcommons.sc.edu/etd/3903/`]

M. Milton, A. Benigni, and J. Bakos, "System-Level, FPGA-Based, Real-Time Simulation of Ship Power Systems," *IEEE Transactions on Energy Conversion*, vol. 32, no. 2, pp. 737-747, June 2017.
Available: [`https://ieeexplore.ieee.org/document/7894204`]

M. Difronzo, M. Milton, M. Davidson, and A. Benigni, “Hardware–in–the–loop testing of high switching frequency power electronics converters,” *2017 IEEE Electric Ship Technologies Symposium (ESTS)*, Arlington VA, 2017, pp. 299–304.
Available: [https://ieeexplore.ieee.org/document/8069297]

M. Milton, M. Vygoder, J. Gudex, R. Cuzner, A. Benigni, “Power Electronic System Real-Time Simulation on National Instruments FPGA Platforms,” *2019 IEEE Electric Ship Technologies Symposium (ESTS)*, Arlington VA, 2019, Aug. 2019.
Available: [https://ieeexplore.ieee.org/document/8847934]

M. Milton, A. Benigni, and A. Monti, “Real-Time Multi-FPGA Simulation of Energy Conversion Systems,” *IEEE Transactions on Energy Conversion*, Dec. 2019.
Available: [https://ieeexplore.ieee.org/document/8822485]

M. Difronzo, Md. M. Biswas, M. Milton, H. Ginn III, and A. Benigni, “System Level Real-Time Simulation and Hardware-in-the-Loop Testing of MMCs,” *Energies*, vol. 14, no. 11, p. 3046, May 2021.
Available: [https://www.mdpi.com/1996-1073/14/11/3046]

M. Vygoder, M. Milton, J. Gudex, R. Cuzner, A. Benigni, “A Hardware-in-the-Loop Platform for DC Protection,” *IEEE Journal of Emerging and Selected Topics in Power Electronics*, vol. 9, no. 3, pp. 2605–2619, June 2021.
Available: [https://ieeexplore.ieee.org/document/9171341]

Chapter 2

Usage

This chapter presents how to use the solver codegen CLI tool in your command shell.

The basic usage of the Solver C++ Code Generator CLI tool to generate C++ source code from a netlist file is this in a command shell/console:

```
codegen netlist_file
```

where *netlist_file* is the name of the netlist file. The codegen tool will parse the netlist file and then produce a single C++ solver function definition header file for the network system described in the netlist. The header file will be stored in current directory in the console. Examples of using the tool include:

```
codegen rlc_circuit.netlist
codegen X:\some_dir\system_models\ShipZonalSystem.netlist
./codegen /projects/models/inverter_model.netlist
```

No real restrictions are put on the name of the netlist file itself, though in general, the file can be postfixed with *.netlist* for easier user identification of the type of file. If no argument is given to the tool, it will print out information about itself. Should one want to get help from the tool, or learn more about the tool, one can pass the options *-help* and *-about* respectively to the tool, like so:

```
codegen -help
codegen -about
```

Chapter 3

Netlist Format

This chapter discusses the format for netlist files or strings used to define system networks that will have a solver generated for by the codegen tools.

The netlist file that the codegen CLI tool can read is merely a plain-text file. Each line of the file specifies **only one** of either a command, a comment, or a component definition. Commands are started by a # sign, comments are started with a %, and component definitions are started by the name of the component.

The following is a specification for netlist file format.

```
% some comment goes here.
```

Lines starting with % are comments used to document the netlist. These lines are ignored by the codegen tool. For multi-line comments, merely start each line with the % character.

```
#name system_model_label
```

The #*name* command defines the name/label of the system model which the netlist defines. The label must only start with or contain letters (a-z, A-Z) or underscore (_). Digits (0-9) can also be in the label but the label cannot start with digits. Other characters and white space are not allowed in the label. **All netlists must contain this command to be valid (mandatory)**.

```
#const constant_label constant_value
```

The #*const* command defines a constant value which can be applied in the component definitions. The *constant_label* item defines the label of the constant, while *constant_value* is the numerical value assigned to the constant. Only one constant definition per given label is allowed. The label must only start with or contain letters (a-z, A-Z) or underscore (_). Digits (0-9) can also be in the label but the label cannot start with digits. Other characters and white space are not allowed in the label. The value must be a decimal value defined in base 10 (i.e. 5.5, 3, 100.0e-6, etc.) and **cannot** be defined as a math expression nor in terms of another constant. Constants are used in component definitions by merely refering to the constant label. During parsing of the netlist, the constant labels will be replaced by the corresponding value.

```
ComponentType component_label (param1, ..., paramP) {node_index1, ..., node_indexN}
```

The above is a component definition (listing). All component listings start with the type of the component *ComponentType* (Resistor, VoltageSource, etc.), followed by the component's **unique** label. The label must only start with or contain letters (a-z, A-Z) or underscore (_). Digits (0-9) can also be in the label but the label cannot start with digits. Other characters and white space are not allowed in the label. After the label, the component's parameters are defined in parentheses

(), such as time step, capacitance, converter levels, etc. Each component type can have its own number of parameters. The parameters must be decimal values defined in base 10 (i.e. 5.5, 3, 100.0e-6, etc.) and **cannot** be defined as a math expression. Parameters can be set by constants defined by the *#const* command; merely place constant label into where parameter can go.

The node/terminal connections of the component into the system are defined in curly brackets {}. Each index must be a positive integer (0 and up) and cannot contain exponents (such as 25e3). The indices must increment sequentially from 0 onwards for system to be solvable. The index value 0 refers to the grounding/common node in the system network. All system networks must have at least 1 common node (node 0) to be nonsingular and uniquely solvable. In general, all components will have at least 2 or more node/terminal connections. Constants can be used for the node index values, but the constant value must follow the rules for index values.

Note that each line of a netlist must contain only 1 statement: comment, command, or component definition. Multiple statements in a line will cause errors and/or malformed generated code. All labels should be unique in the netlist (one instance) and must only start with or contain letters (a-z, A-Z) or underscore (_). Digits (0-9) can also be in the labels but the labels cannot start with digits. Other characters and white space are not allowed in any label. Currently, no math expressions are supported in the netlist format.

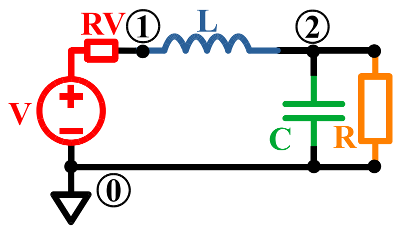

Figure 3.1: RLC Circuit Example with Nodes Indexed

An example of a netlist file for a RLC circuit system, seen in Figure. 3.1 is shown below:

```
% an example of a netlist: example.netlist
#name RLC_Circuit
#const DT 50.0e-9
#const V  100.0
#const RV 0.001
#const R  10.0
#const L  1e-3
#const C  10.0e-3

% components of model
VoltageSource vs (V,RV) {1, 0}
Resistor res (R) {2,0}
Inductor ind (DT, L) {1,2}
Capacitor cap (DT,C) {2,0}
```

Chapter 4

Supported Components

This chapter lists all of the component types currently supported by the codegen tool. All components listed in the netlist definition format.

4.1 Sources

4.1.1 Basic Sources

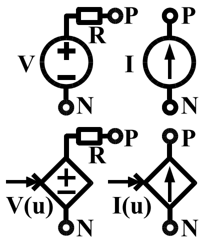

Figure 4.1: Basic Sources; top fixed sources, bottom functional sources

```
VoltageSource label (voltage, series_resistance) {P,N}

CurrentSource label (current) {P,N}

FunctionalVoltageSource label (series_resistance) {P,N}

FunctionalCurrentSource label () {P,N}
```

The *series_resistance* parameter cannot be zero. Indices *P* and *N* are the respective positive and negative terminals.

The functional sources will be generated to take as input each time step a real-valued signal which is the magnitude of the source.

4.1.2 Ideal Voltage Sources

```
IdealVoltageSource label (voltage, solution_id) {P,N}

IdealFunctionalVoltageSource label (solution_id) {P,N}
```

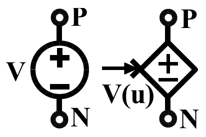

Figure 4.2: Ideal Voltage Sources; left fixed, right functional

Indices *P* and *N* are the respective positive and negative terminals.

The functional source will be generated to take as input each time step a real-valued signal which is the magnitude of the source.

The use of ideal voltage sources induces the use of modified nodal analysis within the generated LB-LMC solver, where instead of the ideal voltage sources' across terms are computed as solution, their through terms are computed as solution (think of super nodes in nodal analysis). With modified nodal analysis, the ideal voltage source introduces new equations into Gx=b at the end of the system. Due to this setup, each ideal voltage source is given an unique solution ID number which is a value greater than the largest node index in the netlist and must be integrally sequential to this node index (in other words, if a system has 6 non-zero nodes, the ideal voltage sources should have solution ids 7, 8, 9, etc., without any integers skipped). The use of solution ID numbers ensures the ideal voltage source equations are inserted into last rows of the system.

Note that the use of ideal voltage sources, due to modified nodal analysis, can make the LB-LMC solver less efficient. Consider using *VoltageSource* and *FunctionalVoltageSource* components instead, which are non-ideal from series resistance, where ideal voltage sources are not absolutely necessary. The series resistance of the voltage sources can be set small ($\leq$1.0e-6) to approximate being ideal.

4.1.3 Dependent Sources

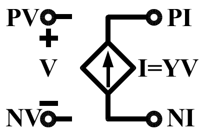

Figure 4.3: Voltage Controlled Current Source

```
VoltageControlledCurrentSource label (transconductance) {PV,NV, PI, NI}
```

Indices *PV* and *NV* are the respective positive and negative terminals where the voltage input is measured from. Indices *PI* and *NI* are the respective positive and negative terminals of the controlled current source. *Transconductance* is the voltage-to-current gain Y of the controlled source ($I = YV$).

Note that *VoltageControlledCurrentSource* can be used to model transconductance between two ports of a system.

Dependent sources are different from functional sources in that dependent sources' magnitudes are directly affected by measured voltages and currents within the given system without delays. Whereas with functional sources, their magnitude is based on an input signal given to simulation solver. Functional sources can be setup to be like dependent sources but can insert 1 unit delay in the measurement inputs. Use dependent sources where the magnitude of the source is based on voltage and currents measured in the system and no inserted delays are allowed for modeling (i.e., op-amp modeling).

4.2 Basic Elements

```
Resistor label (resistance) {P,N}

Capacitor label (time_step, capacitance) {P,N}

Inductor label (time_step, inductance) {P,N}
```

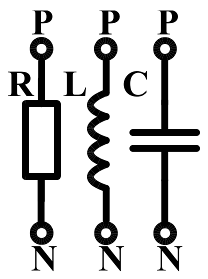

Figure 4.4: Basic Elements

The parameters cannot be zero. Indices *P* and *N* are the respective positive and negative terminals. The *time_step* parameter is the simulation time step length (in seconds) for discretization of the components state equations. All components that have this parameter should use same value for the time step length. Capacitor and Inductor component state equations are implicitly discretized with Trapezoidal method and solved using nodal methods.

4.3 Power Electronics

This section covers the power electronics components presently supported natively in the code-gen tools. All of these components are considered non-linear and therefore their models are explicitly integrated as specified in LB-LMC method.

4.3.1 Three-Leg Bridge Converter with Ideal Switches

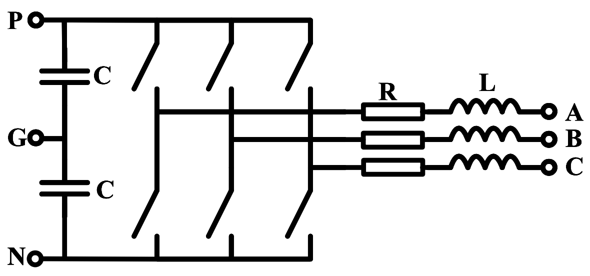

Figure 4.5: 3-Leg Bridge Converter with Ideal Switches

```
BridgeConverter3LegIdealSwitches label (time_step, dc_cap, leg_ind, leg_res) {P,G,N,A,B,C}
```

Models a 3-leg bridge converter with split-bus (bipolar) DC side. Inductors with series resistance are in series of each leg terminated at *A*-*C*; capacitors are on the DC side (*P*,*N* terminals), with center point at terminal *G*. The converter can be used for DC/DC, DC/AC, and AC/DC configuration. Depending on how converter is connected into system network, it can be of half-bridge, full-bridge, or "delta"-bridge configuration. State equations of the converter are discretized explicitly with a first order method, with time step given by *time_step*. Switching elements are modeled as purely ideal switches without anti-parallel diodes. Either the upper or lower switch can be turned on per leg. If the converter is disabled, the switches are replaced with anti-parallel diodes for diode bridge behavior.

No parameter, sans *leg_res* can be zero. The *time_step* parameter is the simulation time step length (in seconds) for discretization of the components state equations. All components that have this parameter should use same value for the time step length. The *dc_cap* is capacitance of each capacitor on split-bus DC side of converter. Inductance and series resistance of each inductor on the leg side is set with the *leg_ind* and *leg_res* parameters. Indices *P, G, N* refer to respective positive, center-point, and negative terminals of split-bus DC side of converter. Indices *A,B,C* refer to the DC or AC leg side of the converter.

This component has four boolean signal inputs, where three are gate signals and one is a disable signal. The gate signals each control a phase leg, deciding if upper arm is conducting (high, 1) or lower arm is conducting (low, 0). If the disable signal is high, then the switches are no longer controlled by the gates, instead acting as diodes whose conduction is dependent solely on phase leg inductor current.

4.3.2 One-Leg Bridge Converter with Ideal Switches and Anti-parallel Diodes

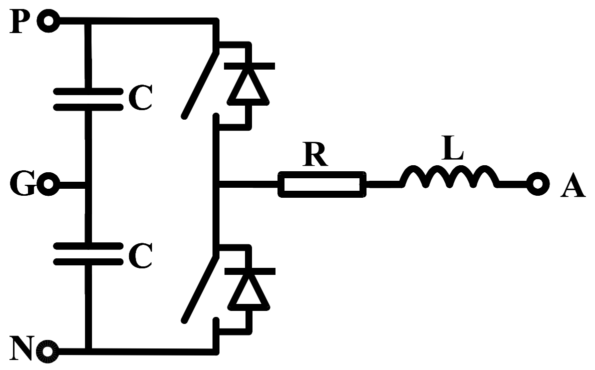

Figure 4.6: 1-Leg Bridge Converter with Ideal Switches and Anti-parallel Diodes

```
BridgeConverter_1LegIdealSwitchesAntiParallelDiodes label
        (time_step, dc_cap, leg_ind, leg_res, diode_threshold_voltage) {P,G,N,A}
```

Models a 1-leg bridge converter with split-bus (bipolar) DC side and anti-parallel diodes across each switch element. Inductors with series resistance are in series of the 1 leg terminated at *A*; capacitors are on the DC side (*P,N* terminals), with center point at terminal *G*. The converter can be used for DC/DC, DC/AC, and AC/DC configuration. State equations of the converter are discretized with Euler Forward method, with time step given by *time_step*. Switching elements are modeled as purely ideal switches with ideal anti-parallel diodes. The upper and lower switches can be turned on or off independently per leg.

Note that this component definition must be done in a single line within a netlist; the definition is shown on multiple lines here to fit on the page.

No parameter, sans *leg_res* can be zero. The *time_step* parameter is the simulation time step length (in seconds) for discretization of the components state equations. All components that have this parameter should use same value for the time step length. The *dc_cap* is capacitance of each capacitor on split-bus DC side of converter. Inductance and series resistance of each inductor on the leg side is set with the *leg_ind* and *leg_res* parameters. The threshold voltage across the anti-parallel diodes for when they conduct is set with *diode_threshold_voltage* parameter. Note that the threshold voltage is not the forward bias voltage of the diodes since the diodes are purely ideal short or open with no series source; the threshold voltage is merely the level that must be exceeded for a diode to conduct. Indices *P, G, N* refer to respective positive, center-point, and negative terminals of split-bus DC side of converter. Index *A* refer to the DC or AC leg side of the converter.

This component can be used with a unipolar DC bus by tying the G to a high resistance to common and setting dc_cap to double the capacitance expected across the DC P and N terminals.

This component has two boolean signal inputs acting as gate signals for the switches. If first gate signal is high, then upper arm conducts; same for second gate signal and lower arm. Should both gate signals be high, the converter will short out. Otherwise, if both gate signals are low, conduction of the converter is based on the diode conduction, determined from their across voltage and through currents.

4.3.3 Three-Leg Bridge Converter with Ideal Switches and Anti-parallel Diodes

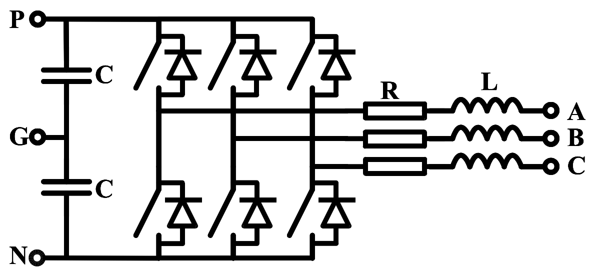

Figure 4.7: 3-Leg Bridge Converter with Ideal Switches and Anti-parallel Diodes

```
BridgeConverter_3LegIdealSwitchesAntiParallelDiodes label
        (time_step , dc_cap , leg_ind , leg_res , diode_threshold_voltage) {P,G,N,A,B,C}
```

The same as *BridgeConverter_1LegIdealSwitchesAntiParallelDiodes* above but the 3 leg version. Terminals B and C are the terminals for the additional legs.

This component can be used with a unipolar DC bus by tying the G to a high resistance to common and setting dc_cap to double the capacitance expected across the DC P and N terminals.

This component has six boolean signal inputs acting as gate signals for the switches. The first two signals control phase leg A, next control leg B, and last control leg C. If first gate signal is high, then upper arm of phase leg A conducts; same for second gate signal and lower arm of phase leg A. The remaining gate signals control the other arms of the converter. Should both gate signals of a phase leg be high, the converter will short out. Otherwise, if both gate signals are low, conduction of the converter is based on the diode conduction, determined from their across voltage and through currents.

This component model was used in this journal paper:
(M. Vygoder, et al.) [https://ieeexplore.ieee.org/document/9171341].

4.3.4 N-Level, 3-Leg Modular Multilevel Converter with Ideal Switch Half-Bridge Modules

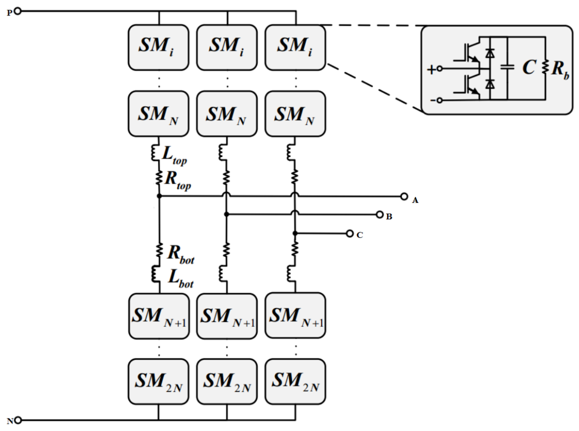

Figure 4.8: N-Level, 3-Leg Modular Multilevel Converter with Half Bridge Modules using only ideal switches

```
ModularMultilevelConverter_HalfBridgeModules label
(
        time_step,
        bleeding_resistance,
        arm_inductance,
        arm_resistance,
        sw_mod_capacitance,
        init_cap_voltage,
        num_levels
) {P,N,A,B,C}
```

Models a 3-leg Modular Multilevel Converter (MMC) with half-bridge switching modules each containing a capacitor and two ideal switches (diodes not fully modeled). The component type name may be altered to *ModularMultilevelConverter_3LegHalfBridgeModules* in future versions of the tools. Note that this component definition must be done in a single line within a netlist; the definition is shown on multiple lines here to fit on the page.

- time_step is the time step length for the discretization of the converter model, in seconds; cannot be zero.
- bleeding_resistance is the bleeding parallel resistance value across the switching module capacitors; cannot be zero.
- arm_inductance is the series inductance value of the converter arms; cannot be zero.
- arm_resistance is the series resistance value of the converter arms; can be zero.
- sw_mod_capacitance is the switching module capacitor value; cannot be zero.
- init_cap_voltage is the initial voltage of each of the switching module capacitors at simulation startup.
- num_levels is the number of voltage levels for the MMC; cannot be less than 2.

This model takes as signal inputs boolean gate signals per switching module (one signal per module). If a gate signal is high, the corresponding module capacitor is inserted into arm; else, the module is bypassed. This model also outputs capacitor voltages and arm inductor currents as real-valued signals for use in measurements and control feedback.

This MMC component can exhibit some numerical instability if the legs do not have sufficient current draw during operation. To remedy this issue, either ensure the component is sufficiently loaded during simulation, utilize linear filtering circuits on terminals in netlist, or insert linear parasitic components on the MMC terminals (like small capacitance) in netlist as seen in real systems.

This component model was used in this journal paper:
(M. Difronzo, et al.) [`https://www.mdpi.com/1996-1073/14/11/3046`].

4.3.5 N-Level, 1-Leg Modular Multilevel Converter with Ideal Switch + Anti-parallel Diode Half-Bridge Modules

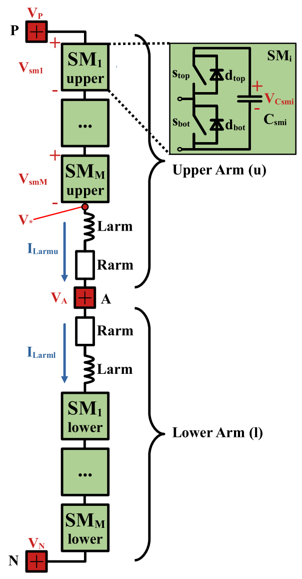

Figure 4.9: N-Level, 1-Leg Modular Multilevel Converter with Half Bridge Modules using ideal switches + anti-parallel diodes

```
ModularMultilevelConverter_1LegHalfBridgeAntiParallelDiodes  label
(
        time_step ,
        num_sm_per_arm ,
        sm_capacitance ,
        arm_inductance ,
        arm_resistance ,
        init_cap_voltage ,
        init_ind_current ,
        diode_voltage_threshold ,
        diode_current_threshold
)  {P,N,A}
```

Models a 1-leg Modular Multilevel Converter (MMC) with half-bridge switching modules each containing a capacitor with two ideal switches and their ideal anti-parallel diodes. Unlike the other component model BridgeConverter_3LegIdealSwitchesAntiParallelDiodes, this model uses fully modeled ideal anti-parallel diodes across the switches and can handle all conduction modes of the modules (inserted, bypassed, open, short). Note that this component definition must be done in a single line within a netlist; the definition is shown on multiple lines here to fit on the page.

- time_step is the time step length for the discretization of the converter model, in seconds; cannot be zero.
- num_sm_per_arm is the integer number of switching modules per arm of the converter. Must be one or greater.
- sm_capacitance is the switching module capacitor value; cannot be zero.
- arm_inductance is the series inductance value of the converter arms; cannot be zero.
- arm_resistance is the series resistance value of the converter arms; can be zero.
- init_cap_voltage is the initial voltage of each of the switching module capacitors at simulation startup; should be greater than 0.
- init_ind_current is the initial current of each of arm going down the arms from top at simulation startup.
- diode_voltage_threshold is the conduction voltage threshold for diodes to start conducting; NOT necessarily related to forward bias voltage. Should be greater than 0.
- diode_current_threshold is the conduction current threshold for diodes to continue conducting; NOT necessarily related to forward bias current. Should be $>=$ 0.
- P node index of positive DC terminal of converter (upper arm).
- N node index of negative DC terminal of converter (lower arm).
- A node index of leg/phase A terminal of converter (between arms).

This model takes as signal inputs boolean gate signals per switching module (two signals per module). If a switch module's gate signals are:

- Stop=1, Sbot=0, then it is inserted
- Stop=0, Sbot=1, then it is bypassed
- Stop=1, Sbot=1, then it is shorted
- Stop=0, Sbot=0, then it's conduction is dependent on diode conduction

This model also outputs capacitor voltages and arm inductor currents as real-valued signals for use in measurements and control feedback.

The model is discretized using an explicit first order method. To build a 3-Leg MMC using this component, use three of these components in parallel, where the P terminals are connected, same for N terminals.

This MMC component can exhibit some numerical instability if the legs do not have sufficient current draw during operation. To remedy this issue, either ensure the component is sufficiently loaded during simulation, utilize linear filtering circuits on terminals in netlist, or insert linear parasitic components on the MMC terminals (like small capacitance) in netlist as seen in real systems.

4.3.6 Dual Active Bridge Converter with Ideal Switches and Transformer Equivalency

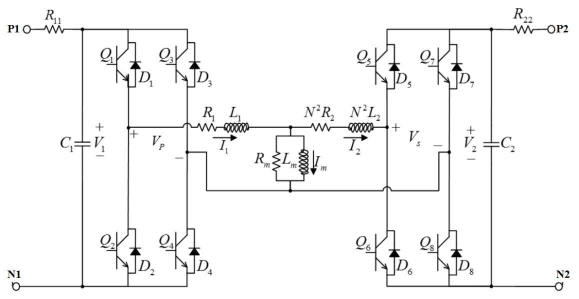

Figure 4.10: Dual Active Bridge Converter with Ideal Switches and Transformer Equivalency

```
DualActiveBridgeConverter_IdealSwitches label
(
        time_step,
        r_input1,
        r_input2,
        r_transformer1,
        r_transformer2,
        r_magnetizing,
        l_magnetizing,
        c_filter1,
        c_filter2,
        l_transformer1,
        l_transformer2,
        turns_ratio
) {P1, N1, P2, N2}
```

Models a Dual Active Bridge (DAB) converter with ideal switches and transformer equivalency. Note that this component definition must be done in a single line within a netlist; the definition is shown on multiple lines here to fit on the page.

- time_step is the time step length for the discretization of the converter model, in seconds; cannot be zero.
- The r_input1 and r_input2 parameters are input terminal resistances R_{11}, R_{22} of the converter sides. Cannot be zero and should be significantly small ($\leq 1m\Omega$).
- The r_transformer1 and r_transformer2 parameters are primary and secondary side series resistances R_1, R_2 of the transformer. Cannot be zero.
- The r_magnetizing and l_magnetizing parameters are transformer equivalency magnetizing resistance R_m and inductance L_m. Cannot be zero.
- The c_filter1 and c_filter2 parameters are primary and secondary side filter capacitances C_1, C_2. Cannot be zero.
- The l_transformer1 and l_transformer2 parameters are primary and secondary side series inductances L_1, L_2 of the transformer. Cannot be zero.
- turns_ratio is the turns ratio N of the transformer; Cannot be zero.

4.4 Port Modeling

4.4.1 Norton Equivalent Port

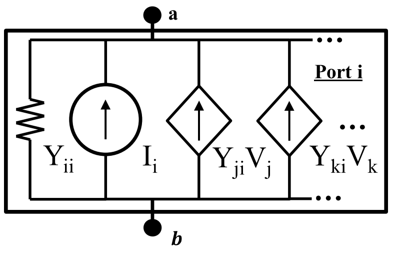

Figure 4.11: Norton Port

```
NortonPort label (conductance, transconductance_a, transconductance_b, ...) {a,b, Pa, Na, Pb, Nb, ...}
```

The *NortonPort* component represents a port of a multiport system with internal sources inside. Across the two terminals *P* and *N*, a functional current source, a series of parallel transconductances (VCCS) from other ports, and a conductance are placed. In effect, this component is like a combination of a *FunctionalCurrentSource* component, a collection of *VoltageControlledCurrentSource* components, and a *Resistor* component.

Indices *a* and *b* are the respective positive and negative terminals where the port is connected. Indices *Pa* and *Na*, and so on, are the respective positive and negative terminals of the other ports that share transconductance with the port. *Conductance* is the conductance seen across the port itself. *Transconductance_a* and so on are the transconductances from other ports seen by this port. Note this component can have a variable number of parameters and terminal connections. The limitation is there must be an even number of terminal connections and the number of transconductances must be half of the number of terminal connections minus 2 (each transconductance must have 2 terminals associated with another port a, b, etc.).

4.5 Misc. Components

Components that don't fit well in other sections.

4.5.1 Ideal Switch with Series Resistance and Inductance

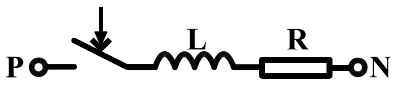

Figure 4.12: Ideal Switch with Series Resistance and Inductance

```
SeriesRLIdealSwitch label (time_step, inductance, resistance) {P,N}
```

Models an ideal switch with series inductance and resistance. None of the parameters can be zero. Indices *P* and *N* are the respective positive and negative terminals. The *time_step* parameter is the simulation time step length (in seconds) for discretization of the components state equations. The component state equation is discretized with explicit Runge Kutta 4th order method. If switch is opened, the inductor current is instantaneously set to zero.

This component is useful for modeling basic switch boxes and fault scenarios, but should **NOT** be used for creating switching power electronic systems due to inefficiency in doing so.

4.5.2 Mutual Inductance with Three Windings

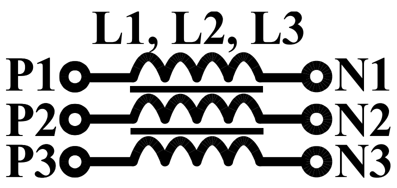

Figure 4.13: Mutual Inductance with 3 Coupled Windings

```
MutualInductance3 label (time_step, L1, L2, L3, M12, M23, M31) {P1,N1,P2,N2,P3,N3}
```

Models three inductors mutually coupled with following state equation:

$$\begin{bmatrix} V_1 \\ V_2 \\ V_3 \end{bmatrix} = \begin{bmatrix} L1 & M12 & M31 \\ M12 & L2 & M23 \\ M31 & M23 & L3 \end{bmatrix} \begin{bmatrix} \dot{I}_1 \\ \dot{I}_2 \\ \dot{I}_3 \end{bmatrix} \tag{4.1}$$

This component can be used for 3-leg/phase transformers, filters, baluns, coupled lines, etc. None of the parameters can be zero and the matrix formed by the parameters must be invertible (nonsingular). The indices correspond to positive (P) and negative (N) terminals of each inductor (1,2,3). The *time_step* parameter is the simulation time step length (in seconds) for discretization of the components state equations. The component state equations are discretized with a first order explicit method.

Chapter 5

Generated Solver Functions

This chapter discusses the C++ solver functions generated by the solver codegen tools.

5.1 Solver Function Definition

The solver generated by the codegen CLI tool specifically for a given netlist-defined system network is defined as a C++ function template, similar to the general example seen below.

```
template <int instance, typename real>
void system_solver
(
        real x_out[50],
        real& y_internal_solution1,
        real* y_internal_solution2,
        ...,
        real y_internal_solutionM[5],
        real u_component_input1,
        real u_component_input2[8],
        bool u_component_input3[12],
        ...
        int  u_component_inputN
)
{
        // solver code goes here
}
```

These definitions are stored entirely into a single C++ header (.h, .hpp) file.

Every call to a solver function in a C++ environment or application solves the solver's associated system network for a single simulation time step. Every state and variable that persists between time steps are stored as static variables within the function.

As function templates, the solvers take as template parameters two arguments when the function is called: *instance* and *real*. The *instance* parameter indicates what instance of the solver function is being called. As a solver function might be used to solve several identical systems in parallel during testing, and static variables persist between every call to same function, the instance parameter allows multiple specializations of same solver function to exist in same application without sharing of static variables. Every call to same function but with different instance number will treat each call as a call to different function, despite being from same template; in effect, the instance value changes the signature of the function. Calls to the solver function with same instance value will call same instance of the function. The instance number can be any integer value. In general, for running only one instance of the solver function, one can merely use 0 for the instance value.

The other template parameter, *real*, specifies the data type used for numerical, real-valued, decimal values within the solver function. Traditionally, *real* is set to either *float* (single precision) or *double* (double precision) floating point data types, especially for offline or CPU-based real-time simulation. For low-level embedded CPU or FPGA based simulation, *real* can be set to fixed point types, such as *IQ* for Texas Instruments platforms or *ap_fixed* for Xilinx HLS FPGA tool suite, among many others. Generally, *real* should be set to a data type that can store real numerical values and can be assigned decimal literal values.

Each generated solver function (template) has a set of arguments which correspond to the I/O signal ports and solution outputs for the system network (defined from a netlist) being solved. These arguments will be tailored specifically for the system network that to the solver corresponds. Every solver function will always have the *x_out[]* argument which is an output port containing the system nodal solutions for a single time step that the function is called. Usually, this argument will contain node voltage and branch current (of ideal voltage source) solutions. The *x_out[]* is always a fixed sized array of type *real*, with the size dependent on the number of system solutions. Index 0 of *x_out[]* corresponds to voltage of node 1, index 1 to node 2, and so forth, with last indices corresponding to currents of any ideal voltage sources that might exist in the system.

A solver function may contain output arguments for the internal solutions of component quantities in the system network, solved for a single time step. These arguments are always pass-by-reference types, either references (&), pointers (*), or be fixed sized arrays ([]), dependent on component definition in the system network. The existence of these internal solution arguments in the function definition are dependent on the components contained in the system network and may not be present if contained components do not have these outputs. The data type of these output arguments are usually *real*, but can be of different types, dependent on the components. Labels of the arguments is dependent on the components.

A solver function may also contain input arguments to control behavior or values of components in the system network. The function will read these input arguments once every time step when the function is called. These arguments are usually pass-by-value types or can be constant (read only) fixed-sized arrays ([]). The data type of the arguments can be either C++ integral types (*bool, char, int*, etc.) or be *real*. The existence of these input arguments and their type, referencing, and label are dependent on the components contained in the system network.

A generic example on how these solver functions are called is shown below.

```
    // local variables to store inputs and outputs of the solver
double x_out[10]; //output system solutions
double y_current_inductor1 = 0.0; //output internal solution for a component
bool u_converter_gates[6] = {0,0,0,0,0,0}; //input gate signals for a power electronic converter

    //...some things happen here in the middle...

    // calling the solver for one time step, using instance 0 and double data type for real
system_solver<0,double> (x_out, y_current_inductor1, u_converter_gates);
```

5.2 How to Use in Offline C++ Simulation Testbench

For the generated solver function (templates) to be used for simulation, a C++ testbench can be developed to utilize the solver function. A typical C++ testbench using a solver function is like the example code presented below.

```
#include <iostream> //include console I/O
```

```
#include <string>   //include C++ strings
#include <fstream>  //include file I/O
#include <cmath>    //include math functions like trig funcs
#include <vector>   //include C++ vector container

#include "system_solver.hpp" //include the solver function template definition

// also include other headers here for various libraries or types needed for the testbench

        //forward declarations for helper functions for testbench, functions defined elsewhere by the user
void updateConverterControl(bool gates[6], double x[10]);
void logSolutions(std::vector<double>& log, double time, double x[10]);
void dumpSolutionsToFile(const std::string& filename, std::vector<double>& log, long num_solutions, long num_steps);

int main()
{
        const static double DT = 50.0e-9; //time step (s)
        const static double TFINAL = 100.0e-3; //complete simulation time (s)

        double x_out[10] = {0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0}; //output system solutions
        double y_current_inductor1 = 0.0; //output internal solution for a component
        bool u_converter_gates[6] = {0,0,0,0,0,0}; //input gate signals for a power electronic converter

        std::vector<double> log; //log vector to store system solutions

        double time = 0.0;

                //start simulation loop

        logSolutions(log, time, x_out); //all zeros logged for time t=0s

        std::cout << "Running_Simulation..." << std::endl;

        while(time < TFINAL)
        {
                         //update simulation time

                time += DT;

                        //update the converter control with some function defined elsewhere

                updateConverterControl(u_converter_gates, x_out);

                        //update solver for this time step

                system_solver<0,double> (x_out, y_current_inductor1, u_converter_gates);

                        //log results somewhere

                logSolutions(log, time, x_out);

        }

        std::cout << "done.\n" << std::endl;

                //log simulation results to file

        dumpSolutionsToFile("sim_results.csv", log, 10, long(TFINAL/DT));

        return 0;
}
```

This particular testbench is a simple C++ source file containing the main function that is the body of the testbench application. The testbench starts by including common C++ STL features for I/O, strings, vectors, and math that could be used in a testbench. Headers for other libraries or definitions can be included using the *#include* preprocessor directive. Then, the system network function template definition, generated by the CLI tool elsewhere, is included into the testbench via the *system_solver.hpp* header. Next, forward declarations for some helper functions that a user may write for their testbench is provided so the functions can be called in the testbench; these functions are assumed to be defined elsewhere, either in the main source file or in other source files. Finally, the main function body of testbench is defined.

In the testbench's main function, true constants (*const static*) are defined for the simulation time step and simulation time length. Then, scoped variables are created to store the output solutions and input signals for the solver function. Next, a vector is defined to store solutions logged from the solver each time step. Finally, the simulation is performed in the main body via a loop where each iteration is for each simulation time step. The body of this loop defines the operations to be performed for each time step.

In the simulation loop, each time step starts with the simulation time being updated by adding the time step length to it. Then, any control or signal operations are performed, either with direct statements or via functions and objects. In this case, converter control is being updated. Then, a specialization of the solver function template is called, being instance 0 and using double data type for real values. The function takes as arguments the array to store the system solutions for this time step, a variable to store a component's current (passed by reference), and an array of boolean values to act as gate signal inputs for a simulated power converter. A single call to the solver function updates it and its states for a single time step. At end of the simulation loop iteration, the simulation time and solutions are logged into a vector called log. After the iteration, if simulation time is less than final time, then the simulation continues, otherwise, the loop is ended to finish the simulation. Once the simulation has finished, the logged solutions can then be dumped to file somewhere or be plotted. In this case, the results are dumped to a CSV file via an user defined function labeled *dumpSolutionsToFile*.

This example testbench is a very simple and general one and ones for more complicated system networks, with sophisticated external control algorithms, under complex testing will likely contain numerous files and source definitions, with inclusion of many libraries to provide. Moreover, the testbench would likely use more structured design for such testing. Design of sophisticated testbenches are outside scope of this document and leans more into traditional software engineering and development. However, some 3rd party libraries or tools are listed here which can be useful to create offline testbenches and support them:

- Datalogger [`https://github.com/MatthewMilton/DataLogger`] - useful datalogger type to log data into memory and dump to file
- Kst2 [`https://kst-plot.kde.org/`] - External plotter application to plot data files
- Gnuplot [`http://www.gnuplot.info/`] - External plotter application to plot either from files or I/O streams/pipes
- Eigen3 [`http://eigen.tuxfamily.org/index.php?title=Main_Page`] - Linear Algebra library (useful for vectorized control design or other work)
- Boost C++ [`https://www.boost.org/`] - general purpose C++ libraries for numerous things not found in C++ STL.
- Mathworks Matlab/Simulink [`https://www.mathworks.com/products/matlab.html`] - commerical linear algebra and simulations platform which supports integration of C++ code (including the generated solvers).

- GNU Octave [https://www.gnu.org/software/octave/] - open-source linear algebra and simulations platform (similar to Matlab) which supports integration of C++ code (including the generated solvers).

Chapter 6

FPGA Synthesis of Generated Solvers with Xilinx Vivado HLS

6.1 Summary

The LB-LMC based solvers generated with the codegen tools can be synthesized to execute on Field Programmable Gate Array (FPGA) devices as Intellectual Property (IP) execution cores. This conversion of the solvers into FPGA IP cores involves taking the C++ code for the original solver and then converting the code automatically into a Register Transfer Logic (RTL) behavioral design described in a Hardware Description Language (HDL); this process is called High-Level Synthesis (HLS). From the HDL code of the solver, the IP core of the solver is defined. Then, this IP core can then be Low-Level Synthesized (LLS) into a digital logic hardware design, described with a logic netlist, which can be implemented on the logic fabric of a FPGA.

The HLS process to allow LB-LMC solvers to run on Xilinx branded FPGAs can be performed with the Xilinx Vivado HLS (VHLS) application. This application provides tools to simulate C++ designs in testbenches, HLS the C++ code into HDL code, validate the HDL code for correctness to the C++ code, export the HDL code into an IP core, and implement the core to report on FPGA resource usage and timing of the design. Other applications can be used to HLS LB-LMC solver cores for Xilinx and other FPGA platforms (Intel/Altera, Lattice, etc.), but these are not tested and therefore unsupported.

This chapter will go over the procedure and provide information and examples on HLS solver cores with Xilinx Vivado HLS for Xilinx FPGAs.

For more detailed information on how to HLS C++ code into IP cores with Xilinx Vivado HLS suite, it is recommended to read the user guide for this suite here (Xilinx UG871 v2019.2 PDF):
https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug871-vivado-high-level-synthesis-tutorial.pdf

6.2 Solver HLS Procedure Overview

The general process for creation and FPGA synthesis of a LB-LMC solver is as follows:

1. Generate and test solver offline with *double* precision floating point *real* type
2. Wrap inline solver function within top-level function
3. Create Vivado HLS project and copy solver and testbench sources to it

4. Set solver internal data type *real* to fixed-point (Xilinx *ap_fixed*) and test again
5. Set HLS directives for latency, port interfaces, and inlining
6. HLS-ize solver top-level function into HDL code
7. (OPTIONAL) Run C(++)/RTL co-simulation of solver HDL code to validate it
8. Export solver HDL code as IP core and validate for resource/timing results (LLS)
9. Include solver IP core into FPGA application

```
//header file (toplevel_core.hpp)

#ifndef GUARD_HPP
#define GUARD_HPP

#include "solver_func.hpp"

void toplevel_core
(
port_real x_out[N],
PORT_TYPE * output1,
...
PORT_TYPE * outputA,
PORT_TYPE input1,
...
PORT_TYPE inputB
);

#endif //GUARD_HPP

//source file (toplevel_core.cpp)

#include "toplevel_core.hpp"

void toplevel_core
(
port_real x_out[N],
PORT_TYPE * output1,
...
PORT_TYPE * outputA,
PORT_TYPE input1,
...
PORT_TYPE inputB
)
{
//HLS directives can go here

TYPE input1_inner, ..., inputB_inner;
//convert PORT_TYPE inputX to TYPE inputX_inner here

real x_out_inner[N];
TYPE output1_inner, ..., outputA_inner;

solver_func(x_out_inner, &output1_inner, ..., &outputA_inner,
input1_inner, ..., inputB_inner);

//convert real/TYPE outputs to port_real/PORT_TYPE and forward them out here
}
```

Figure 6.1: Solver FPGA Core Top-Level Function Sources

6.3 Top-Level Function Definition

Once we have generated a LB-LMC solver function, we can wrap this function inside of a top-level function that will serve as the definition of the FPGA IP core that will be synthesized. This top-level function will provide the port interface of the core, handling the type/word-size conversions between the ports and the solver function parameters, and provide HLS directives to set how the top-level function will be HLS. While the solver function is generated as a header defined function, the top-level function will contain both a header (.h, .hpp) file for the function declaration and includes, and a source file (.cpp) for the definition. The need for a source file (.cpp) is needed as VHLS requires the top-level function for an IP core to be defined in a source file, rather than only a header file; the header file is needed if the top-level function is to be used in a testbench.

An example pseudo declaration and definition of a solver top-level function is shown in Figure. 6.1. Here, the top-level function calls the solver function, passing its parameters as arguments to the solver function. The input/output parameters of the top-level function, the ports of the resultant IP core, can have potentially different types (*port_real*, *PORT_TYPE*) than that of the solver function (*real*, *TYPE*). Therefore, the top-level function should provide internal variables and type conversions to handle differences between types of the top-level function parameters and the called solver function arguments. If the top-level function parameters are same type as the solver function arguments, than no conversion or internal variables are necessary.

Note that the top-level function cannot be a template to be HLSed as Vivado HLS does not support HLS of top-level function templates. The top-level function must be specific and concrete to support HLS. All function templates called by the top-level, including the solver function template, must be specialized with template arguments when called to be HLS by this suite.

6.4 Using Fixed Point Data Types

While floating point representation is mostly used for real numbers in C++ due to its large dynamic range and precision, floating point is actually very computationally expensive. Many floating point arithmetic operators can each take multiple clock cycles, especially division, to complete on both CPU and FPGAs, and can take massive amount of resources to reduce this latency. Unlike CPUs which have heavily optimized and fast fixed logic for floating point, FPGAs devices have performance overhead for their programmable logic and simpler computational units that need to linked together to create similar floating point logic. Also, FPGAs typically operate at much lower clock speeds which can handicap their speed to compute on floating point values compared to CPUs. Though, the main benefit to using floating point on FPGA over CPU is that multiple floating point operations can be done in parallel with unfixed amount of independent math units, unlike CPUs which have a small, fixed number of floating point math units to work with sequentially.

An alternative to floating point is to use fixed point representation. Fixed point stores real numbers as integer words where one half of the word stores the integral part while the other half stores the fractional part. The decimal point between integral and fractional parts is held fixed between the two, with fixed amount of bits assigned to each part, hence the name. Due to the fixed and integral nature, fixed point math operators are much more computationally cheaper than floating point. On FPGAs, these fixed point operations can be done in less than a single clock cycle, even division, while still maintaining a low usage of resources. This benefit of fixed-point representation makes it ideal to achieve nanosecond range simulation time steps with the LB-LMC solvers on FPGAs.

Though C++ does not have a fixed-point data type per se, it can be done by using integer words, with a binary decimal point being assumed in each word. However, Xilinx VHLS provides an optimized fixed-point type that supports arbitrary precision/range and synthesizes efficiently for FPGA execution. This type is called *ap_fixed* and is a templated numerical type that can be parameterized to support various word sizes and precision, along with rounding methods. The *ap_fixed* type supports most common math operators in C++. The common instancing of this type for the LB-LMC solvers looks like this:

```
#include <ap_fixed.h> //include ap_fixed template definition
ap_fixed<64, 24, AP_RND> fixed_point_variable;
//64b word size, 24b integral size, use traditonal rounding
```

In the above case, a fixed point variable is instanced with 64-bits word size, 24-bits for integral part, 40-bits for fractional part, and rounding is set to traditional rounding method; other rounding approaches such as truncation (AP_TRN) are supported which can impact math numerical precision and resource usage/performance. So the *ap_fixed* types can be used, we need to include the *ap_fixed.h* header which is only available in Xilinx VHLS. To use this fixed point type in LB-LMC solver functions, we can *typedef* the *real* data type of the solver function to be this type, like so:

```
typedef ap_fixed<WORDSIZE, INTSIZE, AP_RND> real;
```

where *WORDSIZE* and *INTSIZE* are placeholders for size in bits for particular solver function.

6.5 HLS Directives

As the C++ code of the LB-LMC solver top-level function only defines the abstract behavior of the solver and not how it is implemented into a logic hardware IP core, we must define special HLS directives. These HLS directives instruct the HLS tools on how to synthesize the C++ code into a hardware-oriented RTL behavioral design in a HDL, such as VHDL and Verilog, which can be made into an implementable IP core. With these directives, we can specify the execution latency of the core, the interface for data ports, the usage of dataflow or pipelined designs, resource allocation, and much more. Under Vivado HLS (VHLS), we can define these HLS directives directly in the C++ source code as *#pragma* commands. The *#pragma* commands of these HLS directives are simply placed within a function body or region of code at the top. Below is discussed the most common HLS directives used for LB-LMC solver functions.

6.5.1 Latency

Latency is the number of execution clock cycles for a FPGA core to update all of its outputs when it is triggered to update. By default, VHLS will automatically choose a latency for a C or C++ function based on its analysis of the code and general HLS strategies balancing latency and resource usage. Usually, the VHLS will pick a latency that will not match the desired one, so the latency needs to be manually set. For the LB-LMC solvers, we typically wish to run with a latency of zero so that the solver can update every clock cycle to achieve very low time steps (nanosecond range). To set the latency, we can use the titular *latency* HLS directive in VHLS, given like so:

```
#pragma HLS latency min=0 max=0
```

Here, the minimum and maximum latency is set to equal 0 so that latency will be exactly zero. A range of desired latencies can be specified with *min* and *max* to give the HLS tools some room to decide, so long as the latency values given are positive integers. Setting both min and max latency values to same value indicates that an exact latency is requested. Note that if VHLS cannot find a synthesis solution to meet the requested latency, VHLS will ignore the latency directive and give a warning on why it ignored the directive. The latency directive can be ignored if the code contains rolled loops, loops with variable number of iterations, pipelined elements, or multi-cycle operations whose latency does not fit within the top-leveled latency.

6.5.2 Array Partitioning

By default, VHLS usually synthesizes fixed-sized arrays into FPGA memory resources which use a sequential memory interface (data, address, enable). However, the use of a memory interface can add latency to a solver function to access each element in the array in sequence. Instead, we may wish to treat the array as a set of individual elements that can be accessed in parallel as regular signals. This situation is especially true for the arrays on the input/output parameters of the solver function, such as *x_out[]*. To direct VHLS to partition a C++ array into individual elements, we can use the array partitioning directive, like here:

```
#pragma HLS array_partition variable=x_out dim=0
```

Here, the array *x_out* is fully partitioned (dim=0) so each element within the array is treated as an individual object. This directive supports other features but they are usually not needed for the solvers. Normally, the array partitioning directive is only needed for the input/output arrays of the top-level solver function as VHLS is usually smart enough to automatically partition internal fixed-sized arrays, depending on the array's access pattern. This directive is not used for non-arrays and must follow after the array is declared/defined in the code.

6.5.3 Inlining

Inlining is where the code of a called function is copied into the region where the function is being called, effectively dissolving the called function into the code that called it. For VHLS, it normally synthesizes the bodies of each called function in the top-level individually before combining the synthesized results into a single unit. However, with inlining, the C++ code of the functions can be merged into one monolithic unit which for the VHLS can synthesize and optimize around better. Inlining can allow for redundant resources to be merged into one resource or outright removed, which can potentially reduce latency and resource usage. To direct for inlining during HLS, we can use the *inline* directive like so:

```
#pragma HLS inline
```

When this directive is put inside the body of a function definition, the function will then be inlined into calling regions. For a top-level function to recursively inline all functions it calls into itself, we can add the *recursive* option to the directive:

```
#pragma HLS inline recursive
```

Note that when code is inlined, the different functional units of code being synthesized is no longer distinct, being merged all together, and as such can make telling what parts of the code is using what amount of latency and resources difficult. To disable inlining of particular functions for debugging or analysis purposes, you can use *off* option for the *inline* HLS directive to tell VHLS to not inline the function or region that the directive is in. The use of the directive in this way is as follows:

```
#pragma HLS inline off
```

6.5.4 Loop Unrolling

Vivado HLS will by default synthesize loops to execute each iteration sequentially, where the execution latency is increased by one cycle per iteration, as well as adding one cycle to enter and leave the loop each. However, to achieve zero latency, we need to synthesize the loops so each iteration can be executed in parallel. This setup can be acheived by unrolling the loop using the *unroll* directive within the body of the loop:

```
#pragma HLS unroll
```

Consider that unrolling of a loop in VHLS can only work if the loop has a fixed, finite number of iterations known during synthesis, and each iteration is independent from one another in complex ways (a new iteration cannot depend on results from past iteration). However, iterations in unrolled loops can depend on one another if pattern can be parallelized (such as summation into adder trees).

```
#include "toplevel_core.hpp"

void toplevel_core
(
port_real x_out[3],
bool input1[3],
int  input2
)
{
#pragma HLS array_partition variable=x_out dim=0
#pragma HLS array_partition variable=input dim=0
#pragma HLS latency min=0 max=0
#pragma HLS inline recursive

real x_out_inner[3];

solver_func(x_out_inner, input1, input2);

for(int i = 0; i < 3; i++)
{
#pragma HLS unroll
x_out[i] = convertToPortType(x_out_inner[i]); //func defined elsewhere
}
}
```

Figure 6.2: HLS Directive Example for Solver Top-Level Function

6.5.5 Example

In Fig. 6.2 is an example of using VHLS directives in a pseudo top-level LB-LMC solver function, based on Fig. 6.1. This example shows how the *array_partition* HLS directive is used on the array parameters of the top-level function. A *latency* directive is used to set latency to equal zero.

The *inline* directive with *recursive* option will inline the function calls *solver_func()* and *convertToPortType()* into the body of the top-level during synthesis. Finally, we use a *unroll* directive to unroll the loop that is converting the type of the internal output elements *x_out_inner[]* into that of *x_out[]* and assigning them to *x_out[]*.

6.6 Simulating and Synthesizing in Xilinx Vivado HLS

Once the LB-LMC solver function and its top-level function have been defined, we can begin simulating and synthesizing them in Xilinx VHLS with fixed-point usage.

See Section 5.2 for how to create C++ testbenches around solver functions or their top-level function. As of the time this document was released, Vivado HLS (2019+) only support C++03 standard, with limited support for C++11 features. Make sure your testbenches are developed in C++03 (or limited C++11) and that any 3rd libraries used by the testbenches will compile under VHLS with this C++ version.

A later version of this document will present details on how to synthesize top-level functions into FPGA cores under Xilinx VHLS. The general procedure for synthesis and FPGA execution is the following:

1. *Install full version of Vivado HLx suite 2019.2 or higher (either Design or System Edition)*
2. *Setup Vivado licenses (floating or node-locked) for HLx suite and target FPGAs*
3. Create Vivado HLS (VHLS) project
4. Setup FPGA solution target(s) in VHLS project
5. Copy LB-LMC solver top-level C++ files to VHLS project
6. Create/copy testbench C++ files in/to VHLS project
7. Test/simulate solver in testbench with target internal data type (ap_fixed, float, etc.)
8. Setup HLS directives for desired latency, timing, and resource allocation
9. Synthesize (HLS) solver top-level to HDL (VHDL, Verilog, SystemC)
10. (Optional) Test/simulate HDL-based solver under C++/RTL co-simulation to validate correctness to C++ version
11. Export and validate solver top-level HDL as IP core to determine out-of-context timing and resource usage
12. Copy solver IP core to a Vivado (HDL) project
13. Use solver IP core in desired application under Vivado HDL project
14. Build and load application bitstream onto target FPGA
15. Run the FPGA application

Chapter 7

Troubleshooting

This chapter goes over common potential issues you may encounter and how to fix them.

7.1 Netlist Syntax Errors

In general, the solver codegen CLI tool will report what type of syntax error has occurred. Follow Chapter 3 for netlist format and syntax.

Some common errors include the following, along with their fixes.

```
failed to open given file
```

Make sure the netlist filename given to codegen tool is correct or that the file exists.

```
unsupported syntax at line #
```

Double check that syntax is correct in the given line.

```
line starts with unsupported sequence, character, or command at line #
```

Double check that syntax is correct in the given line. Lines can only start with %, #, and component names; or, can be empty.

```
redefined model name at line #
```

Make sure only one model name, defined with #name, is given in the file.

```
model name error at line #
```

Make sure model name is formatted as a valid C++ label.

```
redefined constant at line #
```

Make sure a constant with a given label is not defined more than once.

```
redefined component with same label at line #
```

Make sure a component with same label is not defined more than once in the netlist.

```
model name not defined
```

Make sure the model name, defined with #name, is given in the netlist.

```
producer doesn't exist for given component type
```

This error occurs when a component defined in a netlist is not supported by the tools. Make sure a listed component is supported by the tools; see Chapter 4 for supported components.

```
couldn't parse a node index as a number
couldn't parse a parameter as a number
```

These errors occur in component listings where a parameter or node index could not be seen as a number. Make sure the parameter or index syntax is correctly expressed as a number, as well as make sure a parameter or node index constant defined earlier in the netlist exists.

```
couldn't find start of node indices
couldn't find end of node index/indices
couldn't find start of parameters
couldn't find end of parameter(s)
```

These errors occur when the parameters or indices part of a component listing is missing a bracket; () for parameters, {} for indices. Make sure the parameter and indices lists are delimited by the appropriate brackets. Also make sure extra brackets are not accidentally included.

7.2 Singularity and Solver Stability Issues

7.2.1 Singularity

During the generation of a solver function for a given system network netlist, you could potentially encounter the following error on singularity:

```
cannot invert conductance matrix as it is singular
```

This error occurs when the set of equations generated from the system network definition is singular. In other words, the system has no unique set of solutions, only infinite number of sets. This error occurs when the system network equations have circular dependencies between solutions, where one solution depends on another one, leading to infinite number of solutions possible rather than the unique one desired. The main culprit for this error is a part of the system network is floating without a path to the system common (node 0). To fix the error, make sure the system network has at least one connection to node 0 (common).

Due to nonlinear and switching components being discretized with explicit integration, and to how the component model can be defined, it is possible a component can naturally decouple parts of the system network in regards to the set of equations for the system, causing parts of the system to lose a path to common even if network definition is correct. In other words, parts of the system can naturally become independent in the set of equations that can end up being singular from lack of the common path. To fix this, merely add a high resistance path to common in a part of the network that is decoupled from rest of system by explicitly integrated components. Components that are explicitly integrated include all nonlinear and switching components (BridgeConverter, DualActiveBridgeConverter, ModularMultilevelConverter), and some extra components (MutualInductance3, SeriesRLIdealSwitch). Linear components are implicitly integrated and so never decouples parts of a system. Read on the LB-LMC method and other Resistive Companion modeling approaches for more details on this subject.

7.2.2 Solver Stability

It is possible that a solver for a system network can become numerically unstable, despite the model definition being correct. The two main culprits for solver instability is numerical range/-precision and state latency for a given time step.

Since any computer (CPU or FPGA) are finite machines, they cannot represent numbers with infinite precision or range. As such, numbers are generally expressed with finite precision and range where there is a maximum and minimum value that can be stored by given representation. The generated solvers, using LB-LMC method, often have to deal with very large or small numbers during the solving a system network's set of equations. This situation is especially true for dealing with small or large value parameters such as resistances, capacitances, and inductances which may be inverted. Generally, double precision floating point provides more than sufficient range and precision for the generated solvers. However, in using fixed point, care must be taken to tune the integral and fractional part of the representation to ensure all possible values occurring during simulation can be represented properly. In cases where the chosen numerical representation is inadequate to deal with the needed range and precision for a solver, one may have to modify their model parameters. Otherwise, numerical errors from limited range/precision can accumulate during simulation, causing eventual instability or at least unacceptable persistent error.

Another culprit for instability in a solver is state latency (or element's time constant) for a given time step. The LB-LMC method is a mixed integration approach where some component states are integrated explicitly, others implicitly. Generally, implicit integration for discretization is inherently more numerically stable than explicit integration, though usually at trade off of computational cost without term arrangement. Due to mixed integration, the solver stability can be less stable than a purely implicitly integrated approach, especially if a system network contains many components that are explicitly integrated (nonlinear, switching components). Stability of the solver should be the same for implicitly integrated methods if system network contains only linear components (which are implicitly integrated with classic trapezoidal method). However, despite using mixed integration, the LB-LMC based solvers are generally stable and comparable to other methods for a wide range of typical energy system models, especially at the small time steps achievable by the method in real-time. Despite this, there are fringe scenarios where the solvers can become unstable. Typically, these scenarios involve using too small of latency elements (capacitance, inductance, etc.) in nonlinear (explicitly integrated) components for a time step chosen, causing time constants to be smaller than time step which can cause accumulating errors. Other, less common scenarios can involve a transient quantity in a system to raise faster than what an explicitly integrated component model can handle. A common fix for instability issues is to add damping, parasitic, or filtering linear circuit elements that might be present in physical systems. In this way, the system model becomes more implicitly integrated overall, bringing numerical stability up, as well as potentially becoming more realistic or detailed physically.

Chapter 8

Building CLI Tool From Source

This chapter presents how to build the circuit solver codegen tool from C++ source files.

8.1 Prerequisites

To build the CLI tool of the codegen tools, you will need the following:

- ORTiS Circuit Solver Codegen Tools source
  [https://github.com/OpenRealTimeSimulation/SolverCodegen]
- Eigen 3 Linear Algebra C++ Library
  [https://eigen.tuxfamily.org/index.php?title=Main_Page]
- C++ compiler suite supporting C++14 or higher
  (GCC, MinGW-w64, Intel, MSVC, etc.)
- (OPTIONAL) C++ IDE or make/build system
  (Code::Blocks, Visual Studio, Eclipse, etc.) or (make, cmake, etc.)

Presently, the tools are internally built using MinGW-w64 (GCC 10.3.0) provided through MSYS2 [https://www.msys2.org/], though other suites should work as well. The IDE Code::Blocks (20.03) [https://www.codeblocks.org/] is used to internally manage building the sources.

8.2 Downloading the Tools Source

The sources of the codegen tools are available on Github. These sources can be downloaded either manually or using GIT version control system (available on most major OS platforms).

To download the sources manually, go to the URL of the tools on github (see Prerequisites section) using your web browser of choice, click on the Code▼ button, and then click on download ZIP. Save the downloaded ZIP file of the sources to a directory of your choice and unzip/unpack the contents there.

To download the sources through GIT cloning, merely run the following in your console/shell of choice within the directory which to you wish to clone the sources:

```
git clone https://github.com/OpenRealTimeSimulation/SolverCodegen.git
```

This command will generate a directory /SolverCodegen within the current directory containing the tool sources.

To download Eigen 3, go to their project home page (see Prerequisites section), and follow their instructions for downloading latest version of the library. This library is provided primarily as a header-only library, so no pre-compilation or binaries are needed to use it. The codegen tools use the default Eigen 3 library without any custom modifications.

To start building using instructions in following sections, create a working directory on your computer where you wish to build the codegen tool sources. In this directory, download the tool sources into one subdirectory, the Eigen 3 library in another. For example, your directory organization can be this:

```
\working_directory
        \SolverCodegen //location of codegen tool sources
                \include   //location of header files of the codegen tools
                \src       //location of source files of the codegen tools
                \exe       //location of source files of the CLI tool itself
                \obj       //location created during build for object binaries
                \bin       //location created during build for main executable
        \eigen3        //location of Eigen 3 library
                \Eigen     //actual location of Eigen header files
```

8.3 Building the Sources Manually using GCC-based Suites

This section describes how to build the CLI tool from source manually using a GCC toolchain (native GCC or MinGW-w64). The given instructions assume you already have a GCC toolchain installed and are running under Windows; instructions are nearly identical for building under Linux as well, expect for the slash used for separating directory names (/ for Linux, \ for Windows) and how the GCC executables are called with extension (.exe).

This approach to building the sources is currently tedious due to complexity of the tool sources. Later releases of the source code is planned to provide a makefile, build script, or other means to easily build the sources automatically outside of using an IDE. In the meantime, it is suggested you write a shell script (bash, batch, etc.) to perform this approach; or use an C++ IDE to manage building.

To start building, you will need to compile all of the .CPP sources files (translation units) of the CLI tool into object binaries (.o). The general approach to compile C++ code with GCC suite is using the following command:

```
g++.exe <options> -c <source_file_name> -o <binary_file_name>
```

where you replace $< options >$ with compiler options such as for optimization (-O), binary library directories (-l), inclusion search directories (-I), and preprocessor macros (-D) for instance. The $< source\_file\_name >$ is replaced with relative/full filename of the source file (.cpp), while $< binary\_file\_name >$ is replaced with relative/full filename of the resultant binary object file (.o).

Following the directory example in previous section after downloading the sources, change current directory (cd) into the directory of the tool source code (\SolverCodegen) and call the following command (without line breaks) for *each and every* CPP file in the project within the \src and \exe directories (including their sub-directories):

```
g++.exe -Wall -O2 -D"VERSION <date>" -I..\eigen3 -Itest -Iinclude
-Isrc -Isrc\codegen -Isrc\codegen\components -Isrc\codegen\netlist
-Isrc\codegen\netlist\components -c <source_file_name>
-o obj\<binary_file_name>
```

Replace $<date>$ with current date/time of the build, $<source\_file\_name>$ with name of .CPP file including its relative directory from current directory, and $<binary\_file\_name>$ with same name as the .CPP file but with .o extension and without the directory attached. Calling of this command will compile the given source file and store its binary object into the \obj folder. An example for compiling just the main source file (\exe\codegen_cli_main.cpp) into an object file is the following:

```
g++.exe -Wall -O2 -D"VERSION YYYY.MM.DD" -I..\eigen3 -Itest -Iinclude
-Isrc -Isrc\codegen -Isrc\codegen\components -Isrc\codegen\netlist
-Isrc\codegen\netlist\components -c exe\codegen_cli_main.cpp
-o obj\codegen_cli_main.o
```

During compilation, there may be some warnings, but most can be safely ignored.

After all of the source files have been compiled into object files, you need to link them together into the CLI tool executable. The general way to do this linkage with GCC toolchain is using the following command:

```
g++.exe -o <executable_file_name>
<object_file_name1> <object_file_name2> ...
<object_file_nameN> <options>
```

where $<executable\_file\_name>$ is replaced with name of the executable (.exe extension in Windows) and each $<object\_file\_nameI>$ is replaced with the names of the built object (.o) files (ellipses ... are removed from the command call). For linking the codegen CLI tool binaries together into its executable, this command will be very long, so it is again recommended to make the call from a shell script. A partial example of linking the codegen CLI tool together into a standalone executable without shared/dynamic dependencies is shown here:

```
g++.exe -o bin\codegen.exe
obj\codegen_cli_main.o obj\BridgeConverter3LegIdealSwitches.o
obj\SolverEngineGenerator.o //...and the rest of the .o files
-s -static-libstdc++ -static-libgcc -static
```

After the main executable (bin/codegen.exe) has been built, you can test it out by calling it from a command console/shell. See earlier chapters for how to use the now built CLI tool for generating circuit solver source code.

8.4 Building The Sources Using Code::Blocks IDE

This section describes how to use the open-source Code::Blocks IDE [https://www.codeblocks.org/] to build the CLI tool sources into an executable. These instructions assume you already have Code::Blocks 20.03 installed and configured to use an installed GCC-based compiler suite.

The text of this section is currently pending.

Appendix

LB-LMC Method Summary

The Latency-Based Linear Multistep Compound (LB-LMC) solver method used in the solver codegen tools is a highly-parallelizable algorithm for solving nonlinear electrical/PE systems, similar to the Electro-Magnetic Transient Program (EMTP) and other Resistive Companion (RC) methods (such as SPICE). Under the method, each component of a system is defined as a discretized state space (SS) model that is embodied as a collection of voltage/current sources representing the memory terms of the model, with accompanying conductances embodying the memoryless model terms. From the models, a set of equations Gx=b is produced to be solved each simulation time step for x that consists of branch currents and node voltages of the system, with G and b respectively consisting of the memoryless and memory terms of the system component models.

Unlike EMTP or other RC methods, all nonlinear component models in a system is discretized with explicit integration (Euler Forward, Runge-Kutta), allowing nonlinear SS models to consist of only memory terms based on only past time step solutions; linear components are discretized implicitly (Trapezoidal). As such, the G term of Gx=b can be held constant throughout simulation and the set of equations for nonlinear systems can be solved without iterative process (Newton-Raphson), allowing the nonlinear system to be quickly solved using linear approaches (LU factorization or term arrangement) without loss of nonlinearity of the system. Moreover, the LB-LMC method is optimized to solve models of all system components entirely in parallel, allowing for computational speedups to achieve small time steps in real-time, especially with FPGA execution of the method. In journal papers, the method has been demonstrated to real-time execute multi-converter PE systems on FPGAs with 35-50ns time steps on modern FPGAs.