Generic High-Level Synthesis Flow from MATLAB/Simulink Model

Petri Solanti, Siemens EDA, Munich, Germany (*petri.solanti@siemens.com*)

Shusaku Yamamoto, Siemens EDA, Tokyo, Japan (*shusaku.yamamoto@siemens.com*)

***Abstract*—Increasingly, design teams are seeking an automated path from Electronics System-Level (ESL) design environments to hardware description at the Register-Transfer Level (RTL). Automated RTL generation can be implemented as a direct ESL-to-RTL synthesis or by using High-Level Synthesis (HLS) tools. Most HLS tools use C++ or SystemC as a modeling language, which is closer to the higher-level languages used in ESL environments than VHDL or Verilog. A model translation process is still needed, the effort of manual translation is reasonable, and the translation process can be automated to some extent. Using C++ or SystemC as an intermediate language in the process provides unexpected benefits to the overall design flow. This paper introduces a generic MATLAB to RTL design flow that can be used with most common HLS tools, and it is target technology agnostic.**

Keywords—Electronics System-Level, Matlab, High-Level Synthesis, Verification

I. Introduction

When MATLAB and Simulink became the de-facto standard in algorithm development, many research teams started to look at possibilities of generating RTL directly from MATLAB code, especially for FPGA design [1][2]. Many of those solutions were targeted to certain applications [3] or technologies [4]. The approach is promising but it has a huge abstraction level chasm between the expressively powerful and dynamic MATLAB against the static and bit-level Verilog or VHDL. These generators require a modeling style that limits the freedom of algorithm developers to use the full power of the MATLAB language.

High-Level Synthesis [5] opened new possibilities to model and develop HW at a much higher abstraction level using C++ or SystemC as modelling language. The MATLAB-HLS-RTL flow introduces a new intermediate language, but it allows each language to be used in its optimal abstraction level. This reduces the verification complexity and bridges the abstraction level chasm between MATLAB and RTL. Algorithm developers can use MATLAB as they are used to, which maintains their productivity. The abstraction level difference between MATLAB and C++ is small and HLS enables architecture exploration without changing the source code. HLS-to-RTL synthesis is a reliable technology today.

Model translation from MATLAB to C++ or SystemC has been available for a long time [6]. Yet, the code generation has been targeted and optimized for simulation and embedded software, which is not necessarily optimal for supporting HLS. Editing the generated C++ code manually breaks the validation flow. HLS tools need different coding styles, which limits a fully automated flow to a specific HLS tool. Translating the dynamic MATLAB features and operator-based complex operations to HLS optimized C++ requires a certain level of synthesis. Therefore, the manual translation of MATLAB or Simulink model to HLS C++ code is still one of the best translation methods.

II. Limitations of the existing design processes

There are several problems to be solved before an abstract MATLAB algorithm model is converted to an efficient HW implementation. The main issue is the structural difference of abstract, sequential simulation code with operator-based vector and matrix operations in MATLAB and HW architecture with parallel processing and distributed memory architectures. A widely used method is to write the MATLAB code in a very low level – almost

RTL – that makes the translation easy but slows down the simulation dramatically and limits the freedom of algorithm developer. Replacing the MATLAB vector and matrix operations with C++ linear algebra library functions [7] is not possible, because their coding style is optimized for software execution and is not synthesizable with HLS tools.

Another problem is the required block-level architecture in HW. MATLAB code may have function hierarchy or system object-based class hierarchy, but it doesn't define parallelism nor HW block hierarchy. Most MATLAB designs use multiple toolbox functions or complex operations like matrix multiplication that may have to be handled as individual modules in the HW. MATLAB language doesn't have any communication channel like ac_channel [8] that is needed to define the communication between two concurrent clocked modules in HW. In Simulink the design can be partitioned graphically using subsystems communicating with each other through channels. This model hierarchy can be translated directly into HW architecture.

Reusability of the translated models and tool dependency are also significant problems. If the models are just translated and not optimized for parameterization nor for feature configurability, reusing the models from previous projects may be difficult. An earlier version of the methodology introduced in this paper [9] had a reusability problem too. Automatically generated models are typically not reusable.

III. Design Methodology

MATLAB and Simulink models can describe any algorithm, and they can be built in very different ways. MATLAB model can be anything between a single MATLAB script and hierarchical object-oriented design. Simulink models can also be constructed by using only library primitives, which is not a very efficient methodology for large designs, or by using custom MATLAB functions or System Objects and parameterizable subsystems. That makes automated design translation very difficult and requires different methods in the manual translation process too. The following five methods cover almost all possible model combinations.

Several translation projects have proven that MATLAB and Simulink models should be maintained with floating-point numbers. Using a fixed-point representation increases the number of lines of code and adds fixed-point specific functions, making it more difficult to translate and especially difficult to validate. In floating-point model a deviation of 10^{-15} between MATLAB and C++ models is normal background noise caused by order of operation differences. A meaningful difference is 10^{-3} to 10^{-6} depending on the design and fixed-point word length used.

A. *Analyzing block-level architecture*

First, the MATLAB design must be analyzed for computational resource needs and data transfers between the functions. Based on this information, the HW architecture can be sketched.

The computational resource needs of a code segment or function can be analyzed by using MATLAB's profiler, manual estimation, or simply by measuring the simulation time in relation to the total simulation time. This estimate is not accurate, but it gives an indication of which parts of the simulation model may need increased parallelism, and possibly multiple parallel instances of a certain block.

Data transfer rate is an even more important metric. In the HW the data must be stored in a memory or streamed through the signal path. If a module interface has a large data set like a high-resolution image, it needs twice as much RAM to implement a ping-pong memory interface. Optimizing the data sizes by defining the module structure in a MATLAB model saves time and effort later in the translation.

Model analysis in Simulink is easier, because the design is already modular, and Simulink has several helpful features to analyze the design. In many cases the HW block-level architecture can be extracted from the Simulink subsystem hierarchy, as demonstrated in Figure 1. In a deep hierarchy all workspace parameters that should be configurable in the HW implementation must be brought up to the top-level interface.

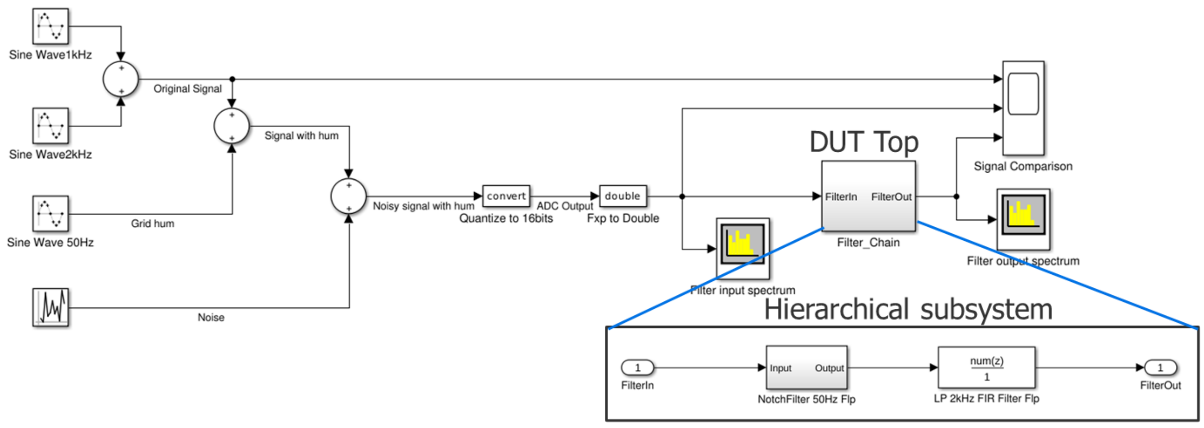


Figure 1. Extracting block hierarchy from Simulink

B. *Translating MATLAB scalar code to HLS C++*

Translation of a MATLAB scalar model is mainly a syntax conversion from MATLAB to C++. Some details must be added manually, like variable declarations that are needed in MATLAB only in special cases or declaration and initialization of static variables. Loop syntax and array indexing are typical sources of errors. In MATLAB array indexing begins with 1 and in C++ with 0. These types of coding errors can be found with linting tools.

Data type definitions should be made in a separate type definition file. This enables using a floating-point type in the conversion phase. When the functionality is validated, the C++ model can be quantized, and data types changed to fixed-point.

Arbitrary length fixed-point data types are needed in HLS. Two fixed-point data types are available as open source under Apache 2.0 license: Algorithmic C fixed-point library (ac_fixed.h) [10] and SystemC fixed-point data types [11]. SystemC data types are slower to simulate and clumsier to use than ac_fixed-types, but functionally both data types are almost identical.

```
function out=iir_filter(inData,inGain,denGain,numGain)
   persistent SReg;

   if (isempty(SReg))
       SReg = zeros(1,2);
   end

   tmpFBAccu = 0.0;
   tmpFFAccu = 0.0;
   tmpFBDiff = 0.0;
   tmpInGainOut = inData * inGain;

   for i=2:-1:1
            tmpFBGainOut = SReg(i) * denGain(i);
            tmpFBAccu = tmpFBAccu + tmpFBGainOut;
            tmpFFGainOut = SReg(i) * numGain(i+1);
            tmpFFAccu = tmpFFAccu + tmpFFGainOut;
            tmpFBDiff = tmpInGainOut - tmpFBAccu;
   end
   SReg(2) = SReg(1);
   SReg(1) = tmpFBDiff;
   outData = tmpFBDiff + tmpFFAccu;
end
```

```
void iir_filter(ac_channel<in_t> &dataIn_ch, coeff_t inGain,
               coeff_t denGain[2], coeff_t numGain[3],
               ac_channel<out_t> &dataOut_ch  )
{
   in_t inData, tmpInGainOut;
   out_rs_t outData;
   accu_t tmpFBAccu = 0.0;
   accu_t tmpFFAccu = 0.0;
   accu_t tmpFBDiff = 0.0;
   accu_t tmpFBGainOut, tmpFFGainOut;

   if (dataIn_ch.available(1)) {
       inData = dataIn_ch.read();
       tmpInGainOut = inData * inGain;
       IIR: for (int i=1; i>=0; i--) {
           tmpFBGainOut = SReg[i] * denGain[i];
           tmpFBAccu += tmpFBGainOut;
           tmpFFGainOut = SReg[i] * numGain[i+1];
           tmpFFAccu += tmpFFGainOut;
           tmpFBDiff = tmpInGainOut - tmpFBAccu;
           SReg[i] = (i==0) ? tmpFBDiff : SReg[i-1];  }
       outData = tmpFBDiff + tmpFFAccu;
       dataOut_ch.write(outData);    }
}  // End void iir_filter
```

Figure 2. MATLAB scalar code translated to C++

C. *Translating MATLAB matrix or vector model to HLS C++*

Operator-based matrix and vector operations are more complicated. In MATLAB a statement A = X * Y' is just one line MATLAB code, but the matrix multiplier implementation in C++ is a large function. Many helpful function libraries can be found in hlslibs.org [12], e.g., ac_math and ac_dsp and matrix library that has C++ function equivalents to most MATLAB operator-based matrix and vector operations.

In MATLAB the user can write multiple matrix and vector operations in a single statement. Because the C++ implementations are functions, there can be only one function call in each statement. Longer statements must be split into multiple single operation statements. As a free add-on, MATLAB indicates the size of the intermediate array variables as a side effect. Translation of a simple MATLAB matrix operation is shown in Figure 3.

```
% PP=SS*NN*NN'*SS';
SSNN = SS * NN;
SSNN_NNtick = SSNN * NN';
PP = SSNN_NNtick * SS';
```

```
private:
  vector_x_matrix_multiply_class<Tin,Tin,Taccu,Tin,MTX_ROWS,
     MTX_ROWS,MTX_COLS,MTX_COLS,false,false> Mult_10Cx10R8C;
  vector_x_matrix_multiply_class<Tin,Tin,Taccu,Tin,MTX_COLS,
     MTX_ROWS,MTX_COLS,MTX_ROWS,false,true> Mult_8Cx10R8C_T;
  vector_dot_product_class<Tin,Tin,Taccu,Tout,MTX_ROWS,false,
     true> dotProd_10x10;
 …
// Implement Matlab statement PP=SS*NN*NN'*SS';
Mult_10Cx10R8C.Product(SS, NN, SSNN);
Mult_8Cx10R8C_T.Product(SSNN, NN, SSNN_NNT);
dotProd_10x10.Product(SSNN_NNT, SS, PP);
```

Figure 3. Translation of MATLAB matrix code to C++ using matrix library from hlslibs.org

D. *Translating Simulink design hierarchy to HLS C++*

The properly assembled Simulink model has a useful design hierarchy that can be transferred into the HLS model as is. Most HLS tools do not accept any functional operations in the block hierarchy level, so the design must be "cleaned up". Hierarchy definitions can differ between the HLS tools, so the hierarchy should be kept as simple as possible to enable different hierarchy implementations.

Depending on the design hierarchy coding style of the HLS tool, the module architecture can be implemented as function or class hierarchy in C++, or an SC_MODULE hierarchy in SystemC. Functionality of the modules can be written in a C++ function that can be called from the hierarchical block. This method improves the reusability of the functional models. Figure 1. demonstrates how a block hierarchy of the DUT can be extracted from the Simulink model.

Translation of a functional model depends on the model implementation in Simulink. If the Simulink model uses MATLAB function blocks or System Objects, MATLAB's model translation can be used. Library component based schematic implementations in Simulink need a different approach.

E. *Translating Simulink leaf-level block diagram to HLS C++*

The leaf-level block diagram using primitive library elements, also called a Simulink schematic, is more difficult to translate. Figure 4. demonstrates a typical Simulink schematic implementation.

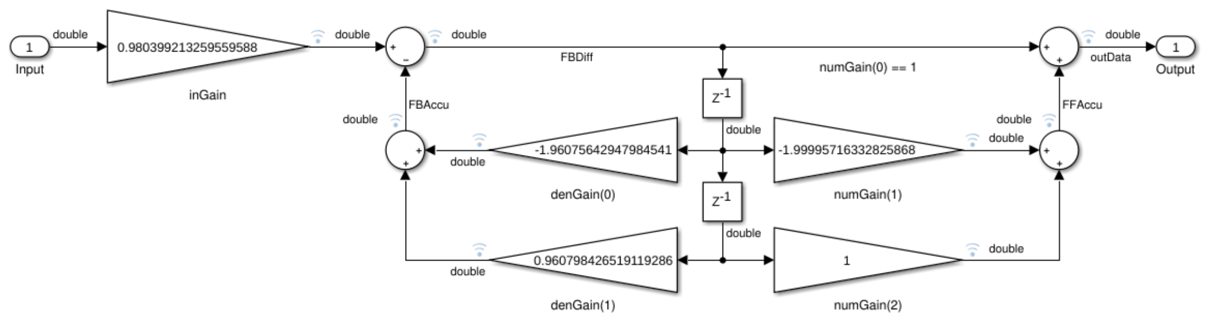
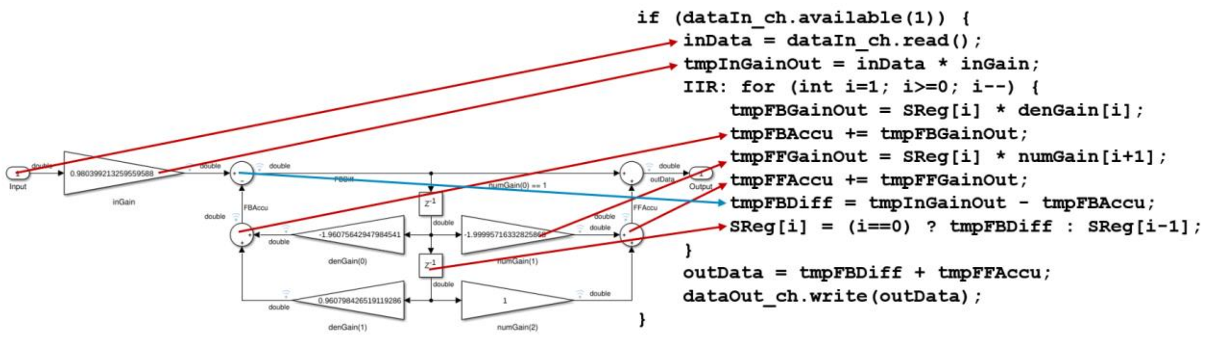


Figure 4. Simulink block diagram with primitive library elements

This example is not difficult to translate to C++. The model has five multiplication operations, four additions and two registers. It can be written as a for-loop that simplifies the structure, or it can be flattened. The translation begins from the second register, and the output of both multipliers is stored into a local variable. When the multiplications are computed, the register gets the value of the first register.

Now the multiplications connected to the first register can be calculated and summed up with the results of the previous stage. The result of the input multiplication can be added to the sum of the feedback path and assigned to the first register. The output is the sum of the first register input and the sum of the feed forward path. An example implementation with a for-loop is shown in figure 5.



```
if (dataIn_ch.available(1)) {
    inData = dataIn_ch.read();
    tmpInGainOut = inData * inGain;
    IIR: for (int i=1; i>=0; i--) {
        tmpFBGainOut = SReg[i] * denGain[i];
        tmpFBAccu += tmpFBGainOut;
        tmpFFGainOut = SReg[i] * numGain[i+1];
        tmpFFAccu += tmpFFGainOut;
        tmpFBDiff = tmpInGainOut - tmpFBAccu;
        SReg[i] = (i==0) ? tmpFBDiff : SReg[i-1];
    }
    outData = tmpFBDiff + tmpFFAccu;
    dataOut_ch.write(outData);
}
```

Figure 5. Translating Simulink schematic with primitive elements to C++

Simulink library blocks can be more complicated to translate. They can be filters, FFTs, matrix operations, memories, or other complex mathematical functions. Open-source HLS function libraries can be used to implement Simulink library blocks in the same way as toolbox functions in MATLAB.

IV. VALIDATION AND VERIFICATION FLOW

Multi-language design flows must have a seamless verification and validation process to ensure the design integrity throughout the process. Using C++ as intermediate language in the design flow increases the number of verification steps, but it provides a comprehensive toolbox that makes the verification easier.

This design flow has three different validation and verification steps:

1. Functional validation of the hand-written C++ code against the original MATLAB model with MATLAB testbench.
2. Coverage analysis of the C++ model in MATLAB's test framework
3. Verification of the synthesized RTL code against the C++ model using MATLAB stimulus

Validation of the hand-written C++ model is the most critical phase. It is implemented by using the MATLAB mex external C API. The C++ model is wrapped into a mex wrapper, compiled to a shared object, and instantiated into the MATLAB testbench, parallel to the original MATLAB reference model. Functionality of the C++ model should be within +/-10^{-15} of the MATLAB reference model. The same testbench can be used later with the fixed-point HLS model with minor changes to the scoreboard.

Coverage Analysis [13] for the C++ model provides information about the test quality. C-level code coverage tools instrument the DUT with probes that store line, decision, and expression coverage information during the simulation and stores the information into a coverage database. The instrumented C++ model can be wrapped into a mex wrapper, instantiated into MATLAB testbench, and simulated in MATLAB. The coverage information gives feedback that helps the test engineer to improve the MATLAB test setup to cover all corner cases. The same test scenarios can be reused later in UVM environment as stimulus.

C-level verification reduces the verification time dramatically, with up to 1000 times faster execution. This allows running more functional tests, resulting in better design quality. A comprehensive study by University of Oulu [14] reports that their C-level simulations ran 191 times faster than equivalent RTL simulations in a UVM environment.

Verification of the generated RTL code is a standard procedure in most HLS tools. The most common verification method is RTL co-simulation, which simulates both the generated RTL code and the HLS C++ code in parallel and compares their output. If the results are equal, the synthesis was correct, and the hardware works as specified. In addition to the RTL co-simulation, there are formal methods available. The same RTL co-simulation setup can be reused in a UVM environment.

V. QUANTIZATION OF TRANSLATED HLS MODEL

Validation of the translated model is done with floating-point or very wide fixed-point data types to keep the focus of the validation on the functionality, without introducing quantization effects. When the model functionality is completely validated against the original MATLAB or Simulink model, it can be optimized for HLS. The optimization has two major parts: structural optimization covering module and loop optimizations and fixed-point optimization a.k.a quantization.

During the quantization process, all variables in the HLS model are analyzed and declared to optimal length fixed-point or integer data types that are synthesized to RTL bit vectors of the same length as the C++ variables. Quantization process has been a subject of many studies and there are very comprehensive Signal-to-Noise-Ratio (SNR) -based methodologies [15] that analyze the effects of quantization noise in the system context, but usually the effort is far too high compared to the benefit. A simpler methodology [16] may require more simulation cycles, but the results are comparable to the SNR-based quantization results.

Value Range Analysis (VRA) -based quantization methodology [17] is simple and semi-automated methodology to find an optimal fixed-point data type for each variable in the design. VRA is simulation-based methodology that analyzes the propagation of the limited accuracy of input values through the step sizes of the internal variables and recommends word lengths that can represent the minimum step size. In this methodology, the input signals must be "pseudo quantized" having a limited number of possible values. VRA requires an instrumented data type, e.g., ac_fixed<>, that collects and stores maximum and minimum values, signedness, minimum non-zero absolute value, and minimum non-zero absolute difference between two consecutive samples during the simulation. These values are used to calculate the required number of integer and fractional bits:

```
int_bits = ceil(log2(maxval)) + signed;
frac_bits1 = -floor(log2(minval));
frac_bits2 = -floor(log2(mindiff));
```

The number of integer bits contains the sign bit, and it can be used as it is. The minimum absolute value is not necessarily the smallest value the variable must be able to represent. For example, a sine wave has a relatively large step size around zero, but a small step size around the pinnacle. To get a better estimate of the required number of fractional bits, both the minimum absolute difference and the minimum non-zero absolute value are needed and the bigger of the two fractional bit numbers should be used. A type declaration of ac_fixed using these numbers looks like

```
typedef ac_fixed<(int_bits+frac_bits2), int_bits, signed> fixed_t;
```

With SystemC fixed-point data types, the signedness defines, if sc_sfixed<> or sc_ufixed<> type must be used. Otherwise, the type declaration is the same.

VI. CONCLUSIONS

HLS-based MATLAB-to-RTL design process is a good alternative to direct synthesis. Using C++ or SystemC as an intermediate language between MATLAB and RTL moves the HW specific design from MATLAB-level to

C++ level, giving algorithm developers freedom to use the full power of MATLAB language. Model translation from MATLAB to C++ takes place in the same abstraction level making the validation and debugging of the translated model easy. A strong validation methodology that reuses the MATLAB testbench, combined with coverage analysis, ensures the functional correctness of the HLS model.

HW related modifications are made to the C++ model only. The C++ model is quantized and optimized for High-Level Synthesis without any modifications to the MATLAB model. The validation environment developed during the model translation phase provides a powerful regression environment that can be used to ensure the model correctness after every modification. HLS tools are well connected to RTL verification. Therefore, verification is mainly focused on the MATLAB-C++ level.

Translating MATLAB or Simulink model to an efficient RTL model is not fully automated yet. The manual conversion process is fast and easy when the HLS optimized library elements are available. The open-source library development enables translating more complex MATLAB models in a short time.

The design methodology described in this paper has been used in many different types of projects. It has proven to be flexible, easy to deploy, and efficient methodology for all kinds of HW designs. Manual language translation requires some knowledge about the coding style requirements of the HLS tool used, but the differences are very small.

REFERENCES

[1] P. Banerjee, M. Haldar, A. Nayak, V. Kim, V. Saxena, S. Parkes, D. Bagchi, S. Pal, N. Tripathi, D. Zaretsky, R. Anderson, J. R. Uribe, "Overview of a Compiler for Synthesizing MATLAB Programs onto FPGAs", IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 12, no. 3, March 2004

[2] T. M. Bhatt, D. McCain, "Matlab as a Development Environment for FPGA Design", Design Automation Conference, June 2005

[3] J. Kilpeläinen, "Automated flow for generating hardware description of filters", MSc thesis, Tampere University of Technology, April 2015

[4] The MathWorks Inc., "DSP Builder for Intel FPGAs", https://www.mathworks.com/products/connections/product_detail/dsp-builder.html

[5] D. Gajski, L. Ramachandran, "Introduction to High-Level Synthesis", IEEE Design and Test of Computers, Winter 1994, pp. 44-54

[6] G. Y. Paulsen, J. Feinberg, X. Cai, B. Nordmoen, H. P. Dahle, "Matlab2cpp: a Matlab-to-C++ Code Translator", 11th System of Systems Engineering Conference, June 2016

[7] C. Sanderson, R. Curtin, "Armadillo: a template-based C++ library for linear algebra.", Journal of Open Source Software, Vol. 1, pp. 26, 2016.

[8] "Algorithmic C (AC) Datatypes Reference Manual", https://github.com/hlslibs/ac_types/blob/master/pdfdocs/ac_datatypes_ref.pdf, August 2022,, pp.95-101

[9] P. Solanti, R. Klein, "Seamless MATLAB® to Register-Transfer Level Design Methodology Using High-Level Synthesis", ICDAEES 2020, Prague, September 2020

[10] "Algorithmic C (AC) Datatypes Reference Manual", https://github.com/hlslibs/ac_types/blob/master/pdfdocs/ac_datatypes_ref.pdf, August 2022,, pp.NN-NN

[11] "SystemC Language Reference Manual", IEEE Standards Association, https://standards.ieee.org/ieee/1666/4814/, 2011

[12] High-Level Synthesis library repository, https://hlslibs.org/

[13] "High-Level Verification: Verification Methodology and Flows When Using C++ and HLS", on-demand webinar, Siemens EDA, 2021, https://webinars.sw.siemens.com/en-US/high-level-verification/

[14] E. Niskanen, "High-Level Verification Flow for a High-Level Synthesis-based Digital Logic Design", Master's thesis, University of Oulu, 2022

[15] D. Menard, R. Rocher, O. Sentieys, "Analytical Fixed-Point Accuracy Evaluation in Linear Time-Invariant Systems", IEEE Transactions on Circuits and Systems, vol. 55, no. 10, November 2008

[16] R. Cmar, L. Rijnders, P. Schaumont, S. Vernalde and I. Bolsens, "A Methodology and Design Environment for DSP ASIC Fixed Point Refinement", DATE 1999 Conference

[17] P. Solanti, "Quantization of HLS designs using Value Range Analysis", white paper, May 2023, https://resources.sw.siemens.com/en-US/white-paper-quantization-of-hls-designs-using-value-range-analysis