Single Source System to Register-Transfer Level Design Methodology Using High-Level Synthesis

Petri Solanti, Mentor Graphics Deutschland, Munich, 80634, Germany

Thomas Arndt, COSEDA Technologies GmbH, Dresden, 01099, Germany

***Abstract*—A System-on-Chip (SoC) is a combination of different technology domains on one piece of silicon. With modern semiconductor processes it is possible to manufacture chips that have large analog part, massive digital logic and multiple processors with complex memory architecture on the same die. Yet, the design methodologies of the different domains are often almost orthogonal. Digital hardware (HW) implementation is written at Register-Transfer Level (RTL), analog design is done at transistor level and the software (SW) is written in object oriented C++ or Java classes using abstract datatypes. The growing complexity and shorter design cycles increase the importance of a design entry at higher abstraction level and better collaboration between the design teams. Another new dimension for the design process is the growing demand for virtual prototypes and fully functional simulation models that can be used in automotive system simulations or digital twin models which are requiring a lot of additional work.**

The design teams are facing another challenge due to the wide variety of languages used throughout the design process. Even the digital design flow may use 4 different languages, which leads to 4 models to be updated, when the specification changes. Validation of the individual domains in the system context is difficult and slow.

A new design methodology that addresses all technology domains and reduces the number of models is needed. High-Level Synthesis (HLS) can be used to increase the abstraction level of digital hardware description, but it doesn't tackle the whole challenge. This study presents a design methodology that is based on a single modelling language and can handle analog and digital hardware as well as software design aspects.

***Keywords*— Design methodology, High-Level Synthesis, SystemC, SystemC AMS, Verification, System-Level-Design**

I. INTRODUCTION

The challenge of the System-on-Chip [1] begins at the beginning of the design process, where the paper specification merges requirements, standards, existing IP and algorithms into a single description of the product functionality, electrical characteristics and physical packaging.

Based on this paper document, each design team builds their own part of the system using their own design methodology and the full functionality may not be validated at all before the first prototype chips are processed. This validation gap is also known in digital HW design, where it is still fairly easy to solve. The analog and software dimensions increase the complexity of the verification and validation dramatically. Figure 1 illustrates the traditional design process..

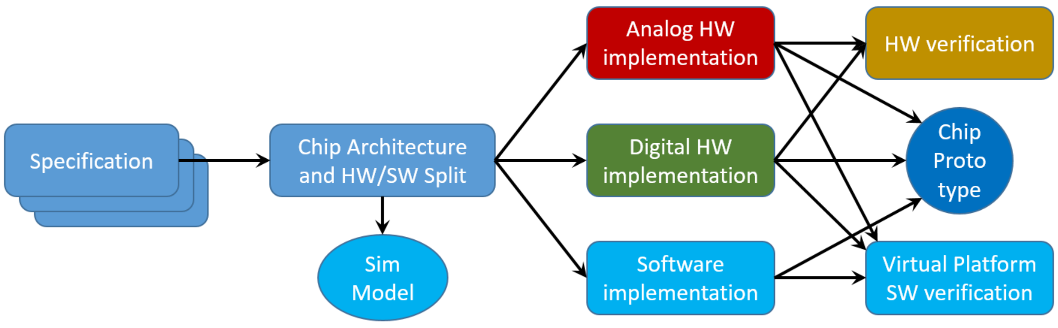


Figure 1 Traditional design process of analog-mixed-signal System-on-Chip

If any of the implementation paths runs into insurmountable problems, or the specification changes, partitioning must be changed and the whole design gets back to the architecture definition phase causing an unpredictable delay.

Having a functional simulation model - a.k.a an executable specification – can help to avoid problems in the later design phases, like re-partitioning or functional mismatches between the specification and implementation.

Executable specification can be implemented as a co-execution of multiple simulators or using one simulation environment that can handle multiple abstraction levels in one simulation. Both approaches have benefits and disadvantages. This study is focusing on the single simulator approach.

II. Languages for Multiple Abstraction Modeling

Selecting a language for an Analog-Mixed-Signal (AMS) System-on-Chip is not trivial. Different teams involved within the design process have different needs and different preferences. Analog designers need high accuracy of the floating-point data types and variable step scheduling to be able to model analog signals with realistic characteristics. Digital designers need arbitrary width datatypes, concurrent processes and discrete time scheduling. Verification engineers need higher abstraction levels, dynamically configurable processes and 100% traceability of the model. Software developers are working on a wide range of abstractions from bit-level operations to functional programming. Features like objects, abstract datatypes and polymorphism are widely used even in embedded software. Furthermore, the modelling language must be an open and compiled language that can be retargeted to different processors and is not bound to one single commercial player.

A modeling language covering all different aspects should have at least:

- Advanced typing system supporting arbitrary width types, abstract datatypes and polymorphism.
- Support for parallel processes
- Concept of time and simulation environment that can handle different scheduling schemes
- Compiler support for different target environments
- Automatic generation of gate-level netlist (HLS or RTL synthesis)

There are around 600 programming languages available according to towardsdatascience.com [2]. Most of them are niche or academic products. Among the top 10 listings of various organizations there are variations depending on the statistical method used. For this study we picked up the most popular and industry standard ones that have been used for hardware design too and the major hardware description languages.

A. *Software languages*

Java [3] is in the most programming language rankings among the top-3. It is an easy and powerful language for application development. There are also some trials to use Java for hardware modeling [4], but it has never gained popularity. Analog modeling in Java requires a simulator software that can handle continuous time signals, e.g. Ptolemy II [5]. Java is an interpreted language and it is not open source, so it will not be considered as an alternative.

Python [6] is another top-3 ranking language. It is the language for machine learning and data analytics, and it is widely used in web and graphical user interface based desktop applications. Python has a very efficient C++ interface and there are translators from Python to C++ [7] and Python to RTL generators like MyHDL [8] and PyMTL [9] available. Although the RTL generators are silicon proven, these are still at the academic level. Python has interfaces to analog simulators [10], but there is no continuous time simulation support in the language itself.

C++ [11] is the eldest member among the most popular programming languages. It was released for the first time in 1985 and standardized in 1998. Unlike many other languages, C++ is aimed to be an open language with no separate programming environment. C++ is a compiled language and there are high quality compilers available for all possible platforms, which makes C++ popular for embedded programming. The language itself doesn't have arbitrary width data types, but they are available as open source class libraries like Algorithm C (AC) [12] types or vendor specific variants. C++ is de facto source language for HLS. Yet, the language itself doesn't have a concept of time, so it is not suitable for system-level simulation without an additional scheduler.

B. *Hardware description languages*

SystemVerilog [13] is the object oriented extension of the Verilog Hardware Description Language (HDL). The extensions allow verification engineers to use more Java-like programming to create more efficient test environments for RTL verification. The abstract constructs are to some extent synthesizable with RTL synthesis tools [14]. The Universal Verification Methodology (UVM) [15] uses its Transaction-Level Modeling (TLM) features to create a dynamic configurable test environment. Analog models using Verilog-AMS [16] extension can be simulated in SystemVerilog test bench with an appropriate simulator. Although programming with SystemVerilog is similar to Java, it is not suitable for software development.

SystemC [17] is a C++ class library that provides an event-driven simulation interface enabling simulation of concurrent processes. It was originally meant to be a HDL language with C++ syntax having arbitrary width data types, clock-cycle accuracy, delta cycle timing and four-valued logic. Structural module hierarchy, parallel processes and connectivity through ports and signals are part of the language as well as the simulation kernel that is compiled with the model into a single executable. Quite soon SystemC was extended with communication transactions, abstract ports and timed events. It became the common platform for Transaction-Level Modeling and virtual platforms and formed the foundation of Electronic System-Level (ESL) design and verification. Some years later the SystemC AMS extension was released extending the capabilities of the language to the analog domain. Because SystemC is C++ based, it can be used for software development without any problems.

III. Single Source Code Design Methodology

SystemC is the only one of the considered languages that has all the required features available today without additional libraries or tools. In addition to the language features, there are open source class libraries that improve the flow, e.g. the MatchLib [18] that enables SoC level assembly and improves the verification interface to UVM. SystemC is an IEEE standard and open source software, which makes it a safe choice for the design methodology.

Due to the underlying C++ nature, SystemC supports a wide range of modeling abstraction levels from mathematical algorithms to RTL in the same model. Furthermore, the SystemC AMS extension enables abstract modelling of analog components and simulating them together with the digital blocks and software in the same executable. SystemC/AMS executable can be generated with open source libraries. This allows an easy and vendor independent exchange of executable models or compiled shared objects enabling delivery of the protected IP simulation model in almost any design phase.

Even though the SystemC language can handle the full range of abstractions needed, the validation problem between the abstraction levels remains – especially with the RTL level. This problem can be solved by using HLS to generate the RTL description from a higher abstraction level SystemC for the IP blocks and using RTL level abstraction only, when it is needed, e.g. in the SoC assembly phase.

The workflow has different paths depending on the nature of the design. Pure hardware projects need only a testbench and the design itself. Hardware/software implementation as well as AMS projects need different modeling paths with additional tool chains. Therefore the flow must be flexible and support configuration of the tool flows on a project basis. This study focuses on hardware only designs..

The hardware only workflow consists of six main steps:

A. Creating a executable specification (functional model) of the system in a high abstraction level in floating-point SystemC

B. Creating testbench for the design in SystemC, validating model functionality and application scenarios

C. Analyzing hardware implementation bottlenecks and restructuring the digital hardware model for HLS

D. Quantizing the model

E. Exploring different HW architectures using HLS and fine tuning the model

F. Generating RTL code and verifying it

A. Writing high-level functional model, a.k.a executable specification

Writing a high-level functional model of the complete SoC behavior including the firmware can be at a very abstract level in C++. When the understanding of the model architecture grows, the C++ classes can be wrapped into SystemC modules to create a model architecture that separates the digital hardware, analog and software partitions. The interfaces between the models represent the later interfaces between HW and SW and analog and digital HW. Inside the top-level modules the individual domains are developed further, each in their own favorite style: analog part in a SystemC block hierarchy with AMS models, software further with C++ class hierarchy and digital hardware in SystemC.

The signal path is modeled using double datatype to keep the functionality analysis focused on the functionality without disturbing fixed-point effects like overflow or loss of precision. The variables in the digital part are declared with specific data types individually or in groups and the type definitions should be centralized into one header file. This allows changes of the type definitions to fixed-point later in the design process.

At latest in this phase the biggest drawback of SystemC becomes obvious. It requires a lot of definitions, signal name mappings, constructors, destructors and other structures that are almost identical in every module, but still so different that they can’t be copied from one module to another one. A graphical SystemC/AMS design platform that generates automatically the header and program file templates and maintains the integrity between them, if anything has changed, makes the modeling much easier. The designer can concentrate on modeling the behavior instead of typing boilerplate and handle compile errors. The design platform must also ensure that the generated code is HLS compliant to be usable to generate RTL from the SystemC model.

The designer of the digital hardware part can use different abstractions for the different parts of the design. Large algorithms can be written as a C++ class library and wrapped into a SC_THREAD. This allows HLS to use several optimizations to generate different HW implementations. If cycle accurate modeling is required, SC_METHOD is the right SystemC structure to be used, since SC_METHOD is transferred to RTL as it is written in SystemC.

The interface between HW and SW is one critical point. In the beginning of the modeling it can be a simple array or struct carrying the required data. When both HW and SW architectures are maturing, the interface should represent the final interface including the resource type, e.g. register bank, shared memory or bus interface. This allows the software developers to write a hardware adaptation layer that is at least close to the final implementation. The hardware model can then be wrapped into a TLM2.0 wrapper and plugged into a virtual platform..

B. Creating testbench and validating functionality

The verification engineers can start the step B in parallel to the model implementation. The interfaces are fixed in the SystemC model and the requirements are available for the creation of verification components. The graphical design platform and verification libraries like the MatchLib and UVM-SystemC library [19] are a great help in this phase.

The SystemC testbench has two different purposes: validation of the executable specification against the specification and ensuring that all possible corner cases are modeled correctly. Every bug found in the simulation model can be fixed in minutes and it is fixed in the RTL too. The same testbench can be used later in RTL verification, so the effort invested in verification of the simulation model shortens the RTL verification time.

C. Analyzing hardware implementation bottlenecks

At this point the design paths diverge. Analog part is developed in an analog design environment, which can reuse the SystemC model as a test generator and reference model and digital part is further developed in SystemC. Using the digital signal path to process the data generated by the analog part the designer can analyze, if the analog performance fulfills the requirements. It also gives an early warning, if the specification can’t be reached and the digital part must do some additional processing to fix the analog problems.

When the functional model of the digital hardware is ready and validated, we can run a pre-synthesis clean-up for it. In simple design this step can be skipped. For complex designs the implementation bottleneck analysis is useful to find out code structures that may cause performance bottlenecks like memory access problems or algorithms that can’t be parallelized efficiently. Also some mathematical functions may not have fixed-point equivalents.

The bottleneck analysis is done by changing the double data types in the model to a synthesizable floating-point type, e.g. ac_ieee_float64 [11]. If the type definitions are collected into a central definition file, type switching is a straightforward operation. Just duplicate the type definitions, change the datatypes and wrap both segments in a #ifdef statement. A quick simulation confirms that the design is still working correctly.

Now the functional model of the system can be loaded into the HLS tool and synthesized to find syntax problems and potential coding style issues that prevent us from reaching the desired level of concurrency. Because some problems may require major algorithm changes, it is easier and faster to fix the problems in floating-point domain. In this phase the target technology can be any available technology. Clock frequency should be kept in a level that is not too close to the hardware speed limits and not so slow that no pipelining is required.

D. *Quantizing the digital hardware model*

After the bottleneck analysis the model can be quantized. There are lots of very comprehensive quantization methods available, but the one introduced in [20] is simple and can be applied for most of the designs.

The methodology is simple. Trace outputs are added to assignment operations in the datapath. Control variables that have a known value range do not need to be traced. The inputs and outputs of the model are quantized to the specified word lengths and casted back to floating-point. This pseudo quantization inserts the quantization effects (noise and offset) to the floating-point signal.

The interesting metrics of the trace signals are peak values, signedness and the minimum non-zero difference between two consecutive data samples. Visualization can be used to analyze the peak values and general behavior of the signals. The minimum difference requires an additional trace function. Initial fixed-point type definitions are made based on these metrics. Peak value defines the required number of integer bits and minimum non-zero value the fractional bits. Variable types can now be changed to fixed-point in the type definition file. The double and ac_ieee_float64 definitions are kept and guarded by a pre-compiler directive.

Simulation results represent now the final hardware results. The fixed-point type definitions based on the minimum and maximum values are usually very conservative. The fractional parts can be reduced until the noise level starts to increase rapidly.

E. *Exploring different implementation options and fine tuning the model*

The model is now synthesis compliant and ready for further architecture exploration with HLS. Users can analyze the performance, area and power consumption of different hardware architectures with the HLS tool and select the optimal architecture that fulfills the specification.

F. *Generating and verifying RTL*

Once the optimal architecture is found, the synthesis constraints are saved into a Tcl script. The combination of the source code, target technology and synthesis script generates always the same RTL code. The generated RTL can be verified against the SystemC by using the verification tool flow of the HLS tool or a RTL co-simulation. The hardware implementation path after this stage is the standard hardware back-end flow.

The SystemC enabled design flow is shown in Figure 2.

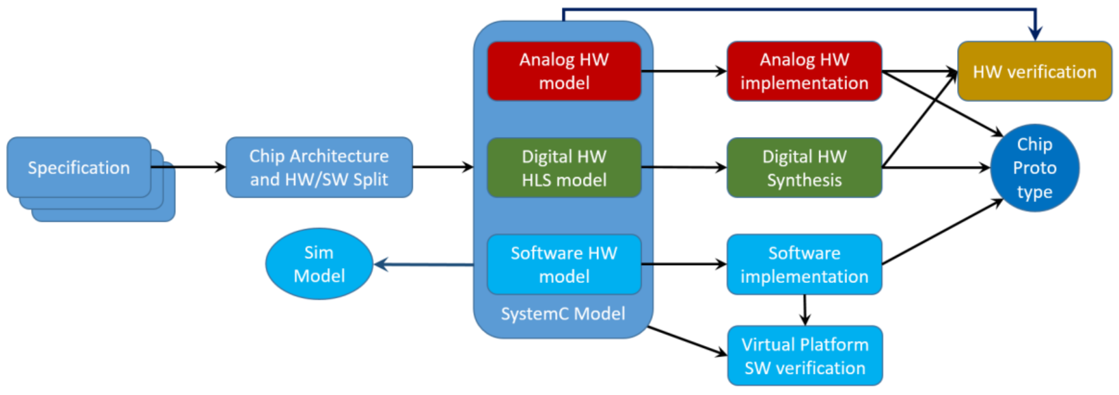


Figure 2 SystemC enabled design flow

IV. Design Example

To demonstrate the design flow, a simple filtering test design is used. The design should remove a 50Hz jammer and high frequency noise from a dual tone signal. The signal path consists of two different filters: IIR notch filter to remove the 50Hz jammer and low-pass FIR filter to remove the noise. Coefficients for both filters were calculated with a filter design tool. The filter path is running at a higher clock frequency to be able to process the data with minimum resources within one input sample time.

Both filters were implemented using a graphical SystemC design environment using a hierarchical top-down approach and connected together in the schematic. The implementation of the filters was added, when the block-level architecture was ready. To demonstrate the difference between SC_METHOD and SC_THREAD, the notch filter was implemented as SC_METHOD and FIR filter as SC_THREAD. The digital core schematic is shown in Figure 3.

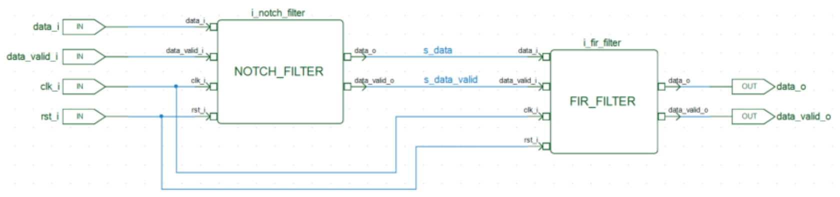


Figure 3 Digital core schematic named digital_filter_top

The testbench was assembled by using SystemC-AMS models to generate the 3 sine waves and white noise into a single input signal. For the realistic input signal we need an A/D-converter model that quantizes the signal to 16 bits and converts it to the defined output datatype. In addition to the signal generator a clock and enable signal generator block is needed. The testbench schematic is shown in Figure 4.

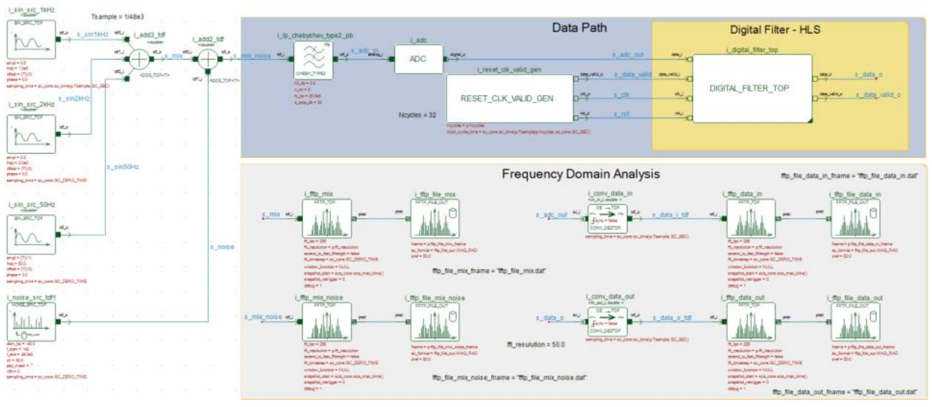


Figure 4 Testbench schematic

Trace outputs were added to the input and output signals to analyze the waveforms and FFT to analyze the frequency domain functionality. When the functionality was validated, additional traces were added into the notch filter signals to analyze the block internal signal values for the quantization.

The peak value analysis of the signals resulted in a maximum value of -1.04e+4 for two internal signals and 5.19e+3 for other internal signals. These can be grouped together to two datatypes with 15 and 14 integer bits respectively. The minimum difference between two consecutive samples was 0.0031 for the first datatype and 0.0016 for the second resulting 8 and 9 fractional bits. Other datatypes are needed for input and output signals and coefficients. Input and output data was already quantized to 16 bits with 2 integer bits. Coefficient datatype needs 2 integer bits. Because there are no meaningful differences in value, the number of fractional bits must be estimated or analyzed with simulation sweep. We are using 12 fractional bits as a starting point

Quantization of the FIR filter can be done without simulation. Input data is limited to 16 bits with 2 integer bits and maximum coefficient value is 0.14335. Principally this value would result -1 integer bits, but leaving some margin to the future changes, we use 0 integer bits. The minimum coefficient value -0.0021 can be represented accurately with 13 fractional bits. The maximum possible output of the FIR filter with the given coefficients is 2.7314, so using 4 integer bits for the accumulator gives enough safety margin. Because the output is reduced to 14 fractional bits, using 16 fractional bits in the accumulator leaves enough headroom for rounding..

The quantized model was simulated and the results compared against the floating-point version. The graphical results in Figure 5 indicate that some word lengths are too short. Because the 50Hz notch seems to be too flat, the problem is probably the coefficient datatype. Increasing the coefficient word length to 18 bits removes the problem.

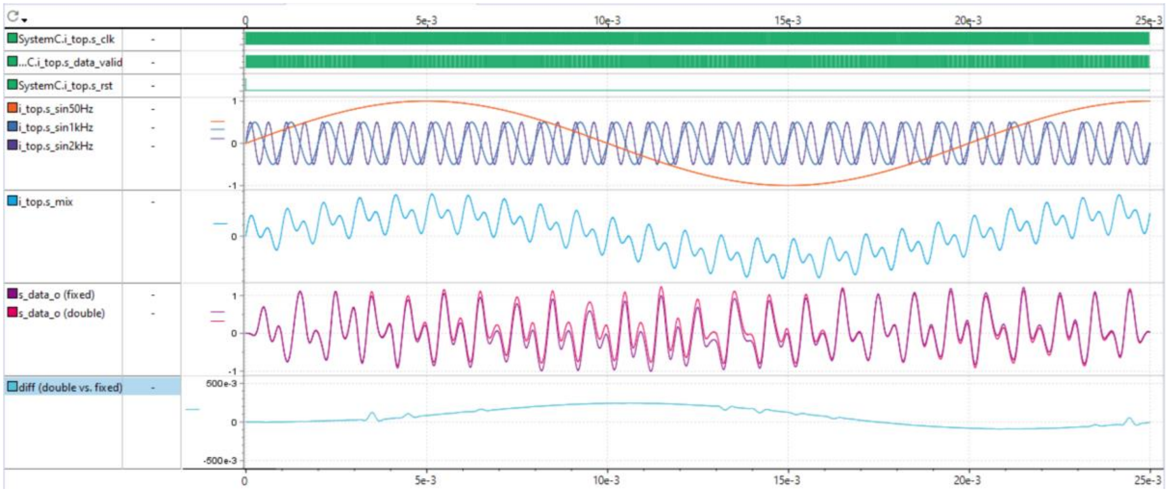
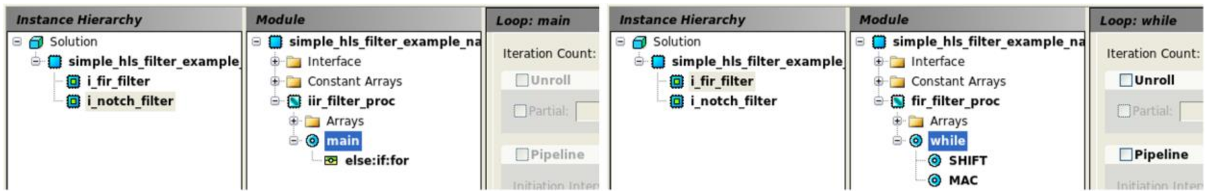


Figure 5 Simulation result comparison floating vs. fixed-point

The digital core module was synthesized with Catapult HLS for Nangate 45 nm CMOS technology with 4 different implementations. The same design can be synthesized with other HLS tools that support SystemC. The architecture view in Figure 6 shows the difference between the SC_METHOD notch filter and SC_THREAD FIR. The notch filter architecture is locked and can't be influenced with synthesis constraints.



Instance Hierarchy
- Solution
  - simple_hls_filter_example
    - i_fir_filter
    - i_notch_filter

Module
- simple_hls_filter_example_na
  - Interface
  - Constant Arrays
  - iir_filter_proc
    - Arrays
    - main
      - else:if:for

Loop: main
Iteration Count:
Unroll
Partial:
Pipeline
Initiation Inter

Instance Hierarchy
- Solution
  - simple_hls_filter_example
    - i_fir_filter
    - i_notch_filter

Module
- simple_hls_filter_example_na
  - Interface
  - Constant Arrays
  - fir_filter_proc
    - Arrays
    - while
      - SHIFT
      - MAC

Loop: while
Iteration Count:
Unroll
Partial:
Pipeline
Initiation Inter

Figure 6 Architecture settings views of notch and FIR filter

V. Conclusions and Future Work

SystemC AMS and High-Level Synthesis enable a single source code design flow from abstract Analog-Mixed-Signal system model to RTL. Graphical SystemC design platform improves productivity by taking care of the connectivity between the design modules and creating the design hierarchy files automatically. The flow enables different paths for hardware/software and analog-mixed-signal subprojects.

Several methodology options were tested during this study. Among them was a dual language flow using MATLAB model as a starting point [21] and a similar flow using Simulink as a system modeling tool. Different parts of the SystemC based design flow were tested and documented by design teams in their real projects [22].

As a future work, this methodology will be tested with hardware/software design and SoC platform integration with MatchLib.

VI. References

[1] AnySilicon, "What is a System on Chip (SoC)?", AnySilicon, July 2019, https://anysilicon.com/what-is-a-system-on-chip-soc/

[2] Towards data science, "Top 10 in-demand programming languages to learn in 2020", https://towardsdatascience.com/top-10-in-demand-programming-languages-to-learn-in-2020-4462eb7d8d3e, Feb 2020

[3] T. Kuhn, W. Rosenstiel, U. Kebschull, "Description and simulation of hardware/software systems with Java", Design Automation Conference, 1999

[4] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, "The Java® language specification", Oracle, 2020

[5] J. Liu, H. Zheng, "CT domain", Ptolemy II: Heterogeneous concurrent modeling and design in Java, vol. 3: Ptolemy II domains, pp. 19-48, Technical report No. UCB/EECS-2008-37, Apr 2008

[6] "Python 3.8.4 documentation", https://docs.python.org/3/, Python Software Foundation, 2020

[7] K. Hayen, "Nuitka user manual", http://nuitka.net/doc/user-manual.html#id9, 2020

[8] "MyHDL – From Python to silicon", http://www.myhdl.org/, 2015

[9] Batten Research Group, "PyMTL 3 (Mamba)", https://pypi.org/project/pymtl3/, Python Software Foundation, 2020

[10] "ngspice – open source spice simulator", http://ngspice.sourceforge.net/

[11] B. Stroustroup, "The C++ programming language", Pearson Education Inc., 2013

[12] Mentor Graphics, "Algorithmic C (AC) datatypes", Reference manual, 2020, https://github.com/hlslibs/ac_types
https://github.com/hlslibs/ac_types/blob/master/pdfdocs/ac_datatypes_ref.pdf

[13] "IEEE standard for SystemVerilog--Unified hardware design, specification, and verification language", IEEE Std. 1800-2017, 22 Feb. 2018, DOI: 10.1109/IEEESTD.2018.8299595

[14] S. Sutherland, D. Mills, "Synthesizing SystemVerilog", SNUG Silicon Valley, 2013

[15] "Universal verification methodology (UVM) 1.2 User's guide", Accelera, October 8, 2015

[16] "Verilog-AMS language reference manual", Accelera, May 30, 2014

[17] "IEEE Standard for standard SystemC language reference manual," in IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005) , vol., no., pp.1-638, Jan. 9 2012 doi: 10.1109/IEEESTD.2012.6134619

[18] Brucek Khailany, Evgeni Krimer, Rangharajan Venkatesan, Jason Clemons, Joel S. Emer, Matthew Fojtik, Alicia Klinefelter, Michael Pellauer, Nathaniel Pinckney, Yakun Sophia Shao, Shreesha Srinath, Christopher Torng, Sam (Likun) Xi, Yanqing Zhang, and Brian Zimmer, "A modular digital VLSI flow for high-productivity SoC design," in Proceedings of the ACM/IEEE Design Automation Conference, June 2018.

[19] Accelera, "UVM-SystemC library", SystemC Verification Working Group, July 2020

[20] 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

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

[22] S. Fontanesi, G. Formato, T. Arndt, A. Monterastelli, "Fast and furious: quick innovation from idea to real prototype", DVCon Europe, 2018