Perspectives

High-Level Synthesis: Status, Trends, and Future Directions

Andres Takach
Mentor Graphics

Editor's notes:
The author provides a status of the significant current industrial relevance of high-level synthesis and how it is advantageous, in particular, in a complex design matrix where verification, power, performance, area optimization as well as design reuse play a key role.
—*Jörg Henkel, Karlsruhe Institute of Technology*

The availability of increasing computational power in system-on-chips (SoCs) continues to drive the development of new and ever more complex applications. Hardware implementations of digital signal processing algorithms enable faster, lower power and less costly wireless, wired, or optical fiber communication. They also enable the higher data compression, processing, and analysis of image and video. The Internet of Things (IoT) is expected to further drive the development of sensors, data processing on those sensors, and their interconnectivity within themselves and with data processing centers.

The need to deliver performance within a given power budget drives the industry to design application-specific hardware. Power considerations are not only important for mobile applications, but also in applications where power density becomes a limiting factor. The challenges and the cost of supplying the power and the cooling for data centers are well known. The use of application-specific hardware accelerators are one of the most promising ways to keep power consumption in check while delivering increasing levels of functionality and performance.

The design and verification complexity and the associated cost are ever increasing. While traditional register-transfer level (RTL) design and verification has evolved over time, design and verification costs continue to increase at a fast pace. Design reuse has been a way to reduce cost by enabling reuse in what is known as intellectual property (IP). However, IP blocks written in RTL need to be manually optimized if an implementation for a new technology node is required, or manually modified if specifications and standards evolve. The key limiting factor of RTL is that it embeds significant detail required to implement the functionality for a specific target technology node and performance goals.

The key to enable more efficient productivity for both design and verification is to raise the level of abstraction. Even within RTL there can be differences in the level of abstraction in which functionality is specified. Going from structurally instantiated multipliers and adders to an expression such as $a^*b + d^*e$ in a hardware description language (HDL) is to a small degree raising the level of abstraction. The expression is not only more compact and easier to read, but it is also more amenable for data path optimizations that could, for instance, create a carry-save-adder (CSA) Wallace compression tree that optimizes the

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.
Digital Object Identifier 10.1109/MDAT.2016.2544850
Date of publication: 01 April 2016; date of current version: 28 April 2016.

2168-2356/16 © 2016 IEEE
Copublished by the IEEE CEDA, IEEE CASS, IEEE SSCS, and TTTC

expression as a whole to speed up the computation. An additional optimization is the retiming of registers to effectively pipeline the computation of the expression. The equivalent structural implementation of that pipeline is far more verbose, and less flexible for design reuse.

The example above could be considered "behavioral" synthesis of RTL (as behavioral versus structural). The data path and retiming capabilities have in fact been incorporated in RTL synthesis as it matured over time. Academic research in high-level synthesis (HLS) has used the term "behavioral synthesis" and "high-level synthesis" interchangeably. From the onset, the specification of the behavior to be synthesized was a program-like specification of behavior which was not "timed" (no clock or reset appeared in the specification). Early in HLS, the central research topic was the scheduling of the operations into control steps, and the mapping of operations into hardware resources and variables into registers. The hardware that is then built consist of a data path and a controller which cycles through the control steps to orchestrate the movement of data from/to registers, functional units using multiplexers, or busses.

It should be evident that by leaving out clock information in the specification, HLS is able to optimize far more than RTL synthesis can, even with retiming of registers. The absence of cycle timing details in the specification enables a higher level of abstraction in a number of fronts.

1) Mapping of local and interface variables or array of variables to memories, registers or interface components is determined during synthesis rather than specified at the source.
2) Execution of operations inside loops. The user can direct synthesis to exploit the parallelism between loop iterations that is required to meet the performance for a given application.
3) Mapping of operations to functional resources. Selection between combinational and pipelined units is done during synthesis.

Loop transformations are very important in HLS. Consecutive loops may be merged to overlap the execution of their loop bodies. Loops may be partially or fully unrolled to create longer sequences of code from which parallelism can be analyzed and exploited to reduce the cycle latency of its execution. A loop may be pipelined, by overlapping the execution of iterations of the loop. Figure 1 shows the effect of loop pipelining in shortening the latency. If the multiplier is combinational, then two of them are required. If it is pipelined, then one multiplier is sufficient.

The specification language

Leading commercial HLS tools support C++/SystemC. HLS users and tool providers worked together within the Accellera Systems Initiative and have developed a synthesis standard around C++/SystemC that has been recently approved [1]. The Accellera synthesis standard defines all the C++ and SystemC constructs that are expected to be supported by HLS tools and serves as a basis for synthesis of C++ and SystemC, while still enabling tool providers to further innovate by adding support for items outside that subset. Pointers are a very important feature in C++. The standard defines support for pointers, and functionality around them, such as arrays and virtual functions that is pragmatic for hardware design. The key requirement is that the

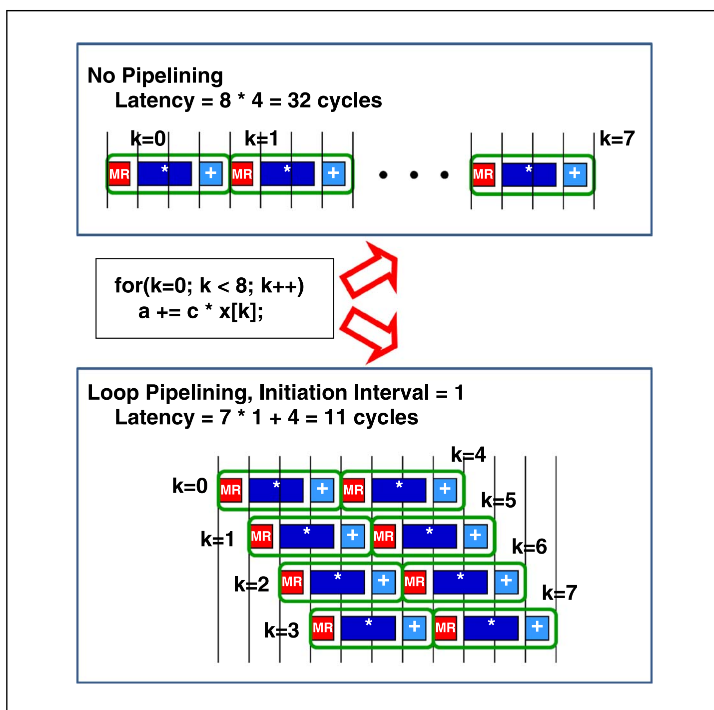

Figure 1. Execution profile of the body of the loop with and without pipelining assuming the array access is mapped to a memory read (MR).

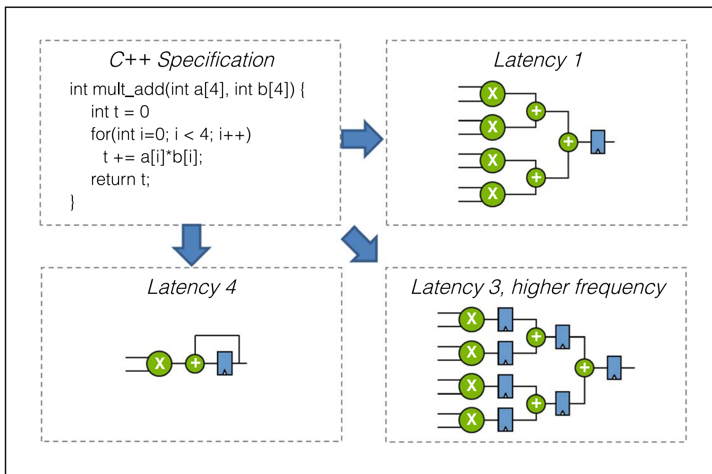

Figure 2. Different hardware implementations generated from the same C++ source.

object pointed to needs to be statically determinable during the synthesis process.

SystemC provides C++ classes that add hardware modeling constructs such as modules and ports for specifying structure and threads (processes), signals, and events for specifying concurrency.

The main advantage of C++ is that it is a general language that is already in wide use for modeling algorithms/behavior at the high level for both hardware and software. SystemC is being used for modeling and virtual prototyping of the hardware to enable performance analysis and early software development.

SystemC also provides bit-accurate integer and fixed-point and bit/logic vector datatypes. They are orthogonal to the core hardware constructs listed above in the same way as both *std_logic_arith* and *numeric_std* are orthogonal to the core modeling features of VHDL. In fact, faster and more consistent datatypes for bit-accurate integer and fixed-point datatypes are provided by the publicly available algorithmic C datatypes [2]. The package also provides datatypes that are not available in SystemC; specifically datatypes for bit-width parameterized floating-point and complex numbers. Any of these datatypes can be used with SystemC.

Using the core hardware modeling features of SystemC allows users to refine the design by adding more implementation detail. However, the more detail that is in the specification, the less abstract it is and the less room to refine/optimize the design using HLS.

The design specification can be as simple as a function in C++ that computes outputs based on inputs. Figure 2 shows a simple example of an algorithm that multiples four pairs of numbers and adds them. HLS creates RTL based on choices the designer provides on how the function arguments map to hardware interfaces and other architectural constraints. One of the implementations streams the input arrays and it takes a latency of four cycles to process the algorithm. The other two implementations read all inputs every cycle, but are pipelined differently.

Hardware hierarchy can be inferred from the C++ call structure of functions. Hierarchy and the streaming of data can also be modeled using Kahn process networks (KPNs). In C++, the KPN is modeled by adding a thin layer over the function calls using channels (FIFOs) to communicate between the functions. KPN models communicating processes with writes that are nonblocking since the FIFOs are assumed to be unbounded, while reads block until there is data available in the input FIFOs. Many subsystems are implemented in HLS today with this modeling style. One of the advantages of the KPN model is that the generated RTL is equivalent to the C++ independently of the latencies for the various processes. The latencies, determined by scheduling in HLS, only have an impact on the minimum first-in–first-out (FIFO) sized in the generated RTL that are required to avoid deadlocks. This is because reads will block and wait until data become available. Modeling styles that do not block on reads require either more careful synchronization or constrains on the length of the schedule.

At the next level of refinement, modules, concurrency, and interfaces are explicitly defined using SystemC. Cycle-accurate and pin-accurate details of the protocols are encapsulated so that a read or a write to an interface appears as a function call in the process thread that is otherwise untimed. Encapsulating protocols helps maintain the abstraction level of the design by isolating the lower abstraction behavior into a set of library interfaces.

Given that SystemC is built on C++, the choice is not about languages, but rather about abstraction, i.e., whether some modeling SystemC features are used and the extent in which they are used.

Complex subsystems are being built with plain C++ or C++ using KPN. SystemC provides the

hardware modeling constructs that are useful to model the integration of such subsystems and to model explicit complex cycle-accurate protocols. As HLS evolves in capabilities to address more complex architectural refinement, less explicit refinement will be necessary in the input code/specification that HLS consumes.

Coding for synthesis

Just like in RTL specifications, the way the source specification is written has an impact on the quality of the hardware generated by synthesis. For example, an image filtering algorithm may average the value of neighboring pixels around a center pixel. The operation is repeated by selecting the neighbor pixel as the center pixel. A subset of the array accesses are common and can be buffered using a shift register to minimize memory accesses in the hardware that is synthesized as shown in Figure 3. In this example, the feedback multiplexers are used to duplicate the pixel value when the window reaches the edge of the image row. Both performance and power are improved with such a rewrite.

Many algorithms are initially implemented using floating-point numbers because they have a wide dynamic range and require less concern about numerical overflow and underflow. For synthesis, floating-point numbers should be replaced by a fixed point, an integer, or a floating point with bit widths that have been reduced to get efficient hardware while still meeting the numerical requirements of the algorithm. Using bit-accurate datatypes enables the verification of the selected bitwidths using C++ which is much faster than verifying that numerical refinement in a manually created RTL specification. HLS may still find opportunities to further reduce bitwidths based on range properties of the operations in the algorithm.

Understanding the implications of conditional behavior to the complexity of hardware is very useful to write C++ specifications that produce the best results. Conditional behavior could imply a chaining of conditions and result in inefficient hardware.

A good reference for how to write C++ code for synthesis is the *HLS Blue Book* [3]. It is likely that additional capabilities and tools will emerge over time to help perform numerical refinement and to better optimize generically written code.

Verification

It is estimated that the number of verification engineers has increased at 3.5 × faster rate than the number of designers in the 2007–2014 period [4]. Verification of an SoC has many facets. Virtual prototyping is often used to have an early model of the hardware so that performance analysis and software development can be started. As the hardware is developed, many functional verification cycles take place while the design is debugged. Verification of the hardware relies on the use of a combination of simulation, formal, FPGA prototyping, and emulation techniques. Beyond the base functionality, other aspects such as performance, clock, and power domains and power management need to be verified. Recently, security/safety have also become a part of the verification of the SoC [4].

One of the main drivers to move hardware design to a higher level of abstraction is to greatly improve verification productivity. A higher level of abstraction of the design specification makes it possible to simulate it at speeds that are orders of magnitude faster than the simulation of the implementation in RTL. Simulation of the C++/SystemC specification does not currently eliminate the need to verify the RTL, but it cuts down the verification-debug cycles that are prevalent in manually generated RTL. HLS users have reported improvements in verification time of 2× or more with fewer computer resources to achieve it.

Additional improvements in verification are being deployed with more in the horizon as the size of the HLS market increases.

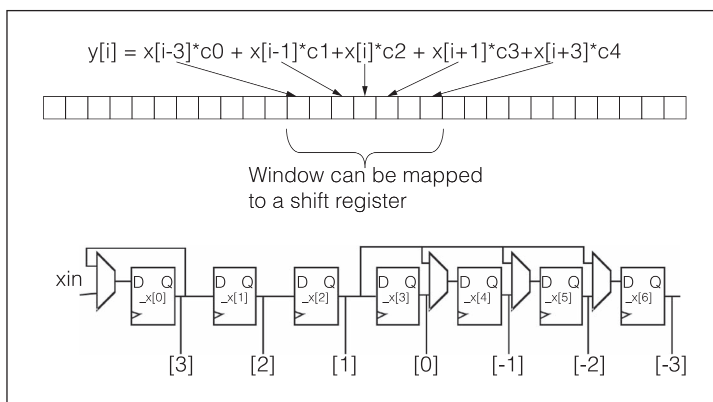

Figure 3. Seven tap sliding window and the shifter implementation that buffers previous array reads. One element is read instead of five elements every cycle.

They can be classified into two categories:

- functional verification of the source HLS specification;
- verification of the generated RTL.

Verifying the C++ specification

Adapting existing methodologies. Verification methodologies and flows that are currently targeted to RTL can also be applied to the C++/SystemC specification. For example, formal property checking of the C++ specification is available and being now deployed by HLS users.

In RTL verification, there is a methodology to develop and measure test vectors effectiveness using coverage tools. Designers would like to use the same rigor to develop test vectors for the C++/SystemC specification. The same stimulus could also be used for the RTL design, although additional vectors may be required to provide the same level of coverage [5].

FPGA prototyping. An HLS specification can be effectively targeted to different technologies. This enables the acceleration of the input specification using an FPGA technology as compared to C++. For some applications, it may be possible to accelerate the algorithm to run at the intended target speed. Such capabilities are particularly useful for developing and validating complex design decisions for evolving standards such as the 5G wireless communication standard.

Verification of the generated RTL

The HLS generated RTL is currently verified using standard verification techniques primarily using simulation, emulation, and FPGA prototyping. The verification infrastructure is typically created for the whole SoC. The stimulus may be created from a subset of vectors used in virtual prototyping or other high level models, but also include vectors derived from directed tests and constrained random stimulus generation techniques.

In this section, the discussion is focused on the areas that have had the greatest interaction with how HLS works.

RTL coverage. The RTL that is generated by HLS needs to go through the same verification flows as RTL that is manually generated. RTL coverage is used as one of the metrics to gauge the quality of the stimulus used for functional verification. RTL code coverage adds to the line/branch coverage that is used in programming languages with additional requirements to exercise logical expressions. Each term in the expression is set in turn to both true and false while the remaining terms are set to let that term make a difference to the output of the expression.

Logic redundancy introduces points that cannot be covered with any test vector and thus artificially lower RTL coverage. Verification engineers often spend a fair amount of effort to understand points that are not covered and either rewrite the RTL or waive such redundancies as part of their flow. It is a more tedious process in machine generated RTL. Redundancies may be present in the source specification. Having better coverage tools for C++ and SystemC would be useful to identify them. Redundancies could also be the result of range properties of inputs that either HLS is not aware of or it has not leveraged during optimizations. Redundancies may also be inadvertently introduced during the synthesis process. Low coverage in HLS generated RTL due to redundancies can be addressed by enhancements in HLS and also by using formal tools that can identify redundancies so that they can be waived during coverage measurement.

RTL coverage is targeted to cover lines/branches and to exercise condition expressions. It is not intended to target arithmetic. For example, for the expression $x <= 0$, it is sufficient to have two values of x to get true/false values for the expression. Synthesis optimization may rewrite the expression in terms of finer logic in ways that require many more vectors to achieve full coverage. HLS optimizations need to therefore take into account such effects in order not to artificially lower coverage.

Emulation. For manually generated RTL designs, simulation-based verification is commonly used before emulation since it provides faster turnaround for exposing and fixing bugs that only require short simulation traces. The use of HLS enables a faster transition from simulation to emulation since the C++ source specification is far more extensively tested than manually generated RTL. Simulation

can then be focused on some areas of concern such as integration verification.

Generated RTL against C++. The base functional verification of the RTL for the subsystems that are generated using HLS can be verified in their own context to add confidence that when the system is assembled only integration issues would be found. This section covers two avenues for verification that are of particular interest in HLS design flows: simulation and formal verification.

Simulation. Using an HLS design methodology it is possible to create the SystemC verification infrastructure to test the generated RTL against the C++ using the stimulus generated by the C++ testbench. The test can be configured to inject stalls to exercise features that may not be present in the C++/SystemC model. This infrastructure can be automatically generated and can aid in performance analysis and extraction of switching activity for power analysis provided the stimulus is representative of actual use scenarios. Power analysis is key for understanding the power/performance/area tradeoffs of the design.

Formal verification. Formal equivalence checking relies on comparison of two specifications to see if they can be proved that they are equal. It is extensively used to verify the gate level netlist generated by RTL synthesis against the RTL design. After the gate and RTL state (registers) are mapped, formal equivalence checking proves that the next state that the gate and RTL logic compute are equivalent. Typically, the synthesis tool needs to convey information on how it arrived at the gate implementation. For example, if a Wallace tree is used to implement the expression $a^*b + c^*d$, that needs to be conveyed to a formal tool so that it can build a structurally similar representation for that expression and prove the equivalence in an efficient manner. This example illustrates that the more refinement takes place in synthesis, the wider the abstraction gap and the more synthesis information needs to be conveyed to the formal tool.

Formal techniques for HLS [6] operate on similar principles of matching up the state and proving that the next state for both the generated RTL and the C++/SystemC is equivalent. Formal verification of the generated RTL against the C++ source is an area of active development and promises to add a very important tool to current verification methodologies to effectively tackle the ever increasing verification challenges. Formal verification of blocks and subsystems could be complemented with simulation to verify their integration and with emulation to do comprehensive verification.

Verification methodologies around HLS will evolve over time on ways to partition the verification problem to get the most confidence on functional correctness with a combination of techniques and tools.

Power analysis and optimization

Most power reduction occurs by the decisions made at higher levels of the design. HLS is uniquely positioned to enable designers to reduce power consumption. Because of design complexity, it is not always intuitive which architecture will be the best in terms of power. Having power analysis accessible within HLS enables the comparison of power consumption for different architectures choices. For example, *Catapult*, a commercial HLS tool, has such a capability [7].

Clock gating is used extensively in RTL design to reduce power consumed by registers and by clock trees. In essence, an analysis is performed to capture the conditions under which each register is loaded and the logic circuit rewritten to use that as an enable input to the register. The new enable condition should be more narrow (exercised less often) than the original enable condition (if it had one). This step is called enable extraction/strengthening. RTL synthesis tools can then implement the register enable by gating the clock (the enable input is ANDed with the clock). When many register bits share the same condition, the clock subtree that feeds them can be gated further reducing switching activity.

There are sequential properties of the logical circuit that may be used to further strengthen register enable conditions. If all registers in the input cone of logic that feeds the data input of a register are disabled, then the data input will be stable and the register could be safely disabled under those conditions. This is called stability-based enable strengthening.

Another sequential property used to strengthen enable conditions is whether the output of a

register is used (observed) under some conditions. This is observability-based enable strengthening. The additional benefit is that unnecessary switching at the output register can be reduced which can also reduce the power consumption of operations in the fan-out of the register.

All forms of enabling strengthening can be performed in an HLS design flow. Power savings in the range of 7%–51% were reported using *Catapult* [7] in a set of real designs with clock gating using the combination of techniques for enable extraction and strengthening described before. There are tradeoffs involved in some forms of strengthening. Using switching activity provides the information required for power optimization to get the best results. One of the ways such switching activity can be collected is the verification infrastructure that is generated by HLS to feed the same input stimulus from the C++ testbench to the generated RTL.

Early switching activity analysis or back annotation of such activity from the RTL, or any other intermediary model generated by HLS, could be used to estimate the effect of optimization decisions made during synthesis. As the design scope of HLS increases to larger subsets of the SoC, additional capabilities will likely be added to HLS to help designers incorporate power management techniques.

Design reuse

The higher the level of abstraction, the easier it is to reuse a specification. A design can be reused to target a different performance and/or technology (ASIC technology node or FPGA device). In many cases, the C++ specification can remain unchanged. In some cases, the source may need modifications if a different memory architecture is required to meet the performance goals for the technology.

Another common way to reuse a C++ design is to modify the source to include additional functionality. For example, in video encoding/decoding, there are many profiles and it is common to support a subset of those profiles and to progressively add profiles in later revisions of the SoC.

The C++ language is ideal for creating IP that is parameterized using C++ template parameters. Such parameters can be used to select the algorithm, the memory architecture, etc. For example, parameters could be used to select the radix of an FFT algorithm, how intermediate results are stored, etc. Parameterization does increase the effort to validate results for all parameter combinations. However, such IP has a lot of value because it can be used to generate RTL using HLS for a wide range of needs. Currently, most HLS IP is produced by users for their internal company use with some provided as part of the HLS tool as reference designs to facilitate its adoption. Commercialization of IP for HLS would likely require encryption of the source to protect the investment to develop it and yet provide sufficient visibility so that the user can change constraints on synthesis to get the desired results.

Status, trends, and future

Over the past decade, HLS has been used for thousands of ASIC tapeouts and FPGA designs. It has been used primarily to design differentiating IP in many cases for algorithms that are part of evolving standards. For example, for wireless communication it was used for 3G and 4G and now it is actively being used for 5G. It has been heavily used for implementing hardware for video decoding standards. A publicly available use of HLS is for the design of the VP9 video standard [8]. Video decoders for the H.264 standard and its successor the HEVC standard [9] have also been designed using HLS. The ability to add functionality to an existing design has enabled more aggressive schedules for incorporating additional capabilities, for example, adding a 10-b profile for video decoding to a design that handles 8-b profiles.

The ability to deliver complex functionality within tight market windows has made HLS an important tool for many companies to be competitive. As companies have gained experience in wider deployment of HLS-based design methodologies, they see the value of rewriting existing RTL IP into C++ IP that is more abstract so they improve the power, performance, and area for future designs that use that IP. In addition, having more of the system represented in C++ provides additional productivity gains in terms of being able to verify more of the system at a high level.

As HLS is more widely adopted, verification will be tailored to get the most confidence with a mixture of techniques as outlined before. The biggest

changes that are taking place or are likely to take place are as follows:

- a shortening of the simulation-based verification and an earlier transition to emulation;
- the use of formal for property checking at the C++ specification;
- the use of formal equivalence checking for verifying the generated RTL against the C++ specification;
- adapting of other verification methodologies to the C++ specification.

HLS flows could also generate additional models that are more abstract than the RTL, but incorporate the required refinements to aid in functional and performance verification, power analysis, and verification of power domains and power management.

In addition to formal property checking, more lightweight checking for likely source of errors is to evolve. Source linting is a lightweight checker. Compilers for C++ do provide warnings for software development, but additional checks that are relevant for hardware could be added and proven valuable. The challenge with linting is that it tends to be very syntax driven and in fact can get in the way of the most natural and clean way to express behavior. The other challenge with linting, as seen in RTL linting, is that rules that check how to best code for synthesis remain, even after synthesis tools have evolved to better handle such cases.

Formal could also be used to prove properties of the design that would enable synthesis to better optimize a design; i.e., prove that the FIFO sizes are sufficient to avoid deadlocks or prove certain numerical properties of the design.

Currently RTL generated by HLS goes through the same flows as manually generated RTL. One area where a distinction should be made is in RTL linting. Most of the linting rules are intended as a safeguard for catching potential errors in manually generated code so that specification errors and verification/debug cycles can be reduced. Because the specification is created and verified in C++ and the RTL is machine generated, the value of linting is significantly reduced. In some cases, it has been proved to be a hindrance, as different HLS user companies have contradictory linting rules.

HLS tools that target ASIC design have the capability to do incremental synthesis in order to facilitate ECO flows. While ECOs are commonplace in manually created RTL, they remain fairly rare in RTL generated by HLS. The main reason for ECOs is that RTL synthesis and physical place and route are being worked on even before verification of the RTL has completed. In order to reduce the impact on work that has been completed, incremental patches are applied (ECO) with a methodology that minimizes the disturbance to the design. Because the C++ specification is verified with many vectors, specification bugs are found and fixed early on. This greatly reduces the need for an ECO in HLS flows.

Given how uncommon ECOs are in HLS-based flows, it is still too early to know how ECO tools and flows may evolve over time. Making the design modular is currently the best way to reduce the scope of the change and therefore minimize the effort involved in the event an ECO is required.

Teaching and research

As HLS becomes adopted more broadly, there will be a point where RTL will become like assembly language in software development: still required, but done by a very small percentage of developers. Some universities have already started to offer courses in hardware design using HLS. Such courses enable students to tackle more interesting design challenges. For example, accelerating algorithms can be done not only for traditional electrical engineering and computer science fields like communication, image, and video processing/analysis, but also with other scientific interdisciplinary fields where computation is key for analysis and modeling.

While research in hardware/software codesign began more than 20 years ago, it is only now that there is a solid HLS foundation to build on to evaluate partition and mapping algorithms for complex designs. The use of HLS to build hardware accelerators for embedded applications and for building more energy-efficient implementations are areas of active research [10], [11].

The transition to synthesis from higher levels of abstraction is taking place in industry. Designing at abstraction levels higher than RTL offers significant advantages in verification, power/performance/area optimization, and design reuse. This article provides perspectives on the ways in

which HLS is helping and shaping verification, power optimization, and design reuse. This transition is also changing the way hardware design is taught and the research projects that are enabled by the availability of mature HLS tools. ■

References

[1] Accellera Systems Initiative, [Online]. Available: www.accellera.org

[2] *Algorithmic C (AC) Datatypes*. [Online]. Available: https://www.mentor.com/hlslp/downloads/acdatatypes

[3] M. Fingeroff, *High-Level Synthesis Blue Book*. Bloomington, IN, USA: Xlibris Corp., 2010.

[4] W. Rhines, "Design verification challenges: Past, present, and future," presented at the Design Verif. Conf. Exhibit., San Jose, CA, USA, Feb. 29–Mar. 3, 2016, keynote.

[5] A. Takach, "Design and verification using high-level synthesis," presented at the Asia South Pacific Design Autom. Conf., Macau, Jan. 25–28, 2016.

[6] A. Mathur, M. Fujita, E. Clarke, and P. Urard, "Functional equivalence verification tools in high-level synthesis flows," *IEEE Design Test Comput.*, vol. 26, no. 4, pp. 88–95, Jul./Aug. 2009.

[7] Mentor Graphics Corporation, "Catapult synthesis." [Online]. Available: http://www.mentor.com/esl/catapult/

[8] WebM, "VP9 video hardware RTLs." [Online]. Available: http://www.webmproject.org/hardware/vp9/

[9] G. J. Sullivan, J.-R. Ohm, W.-J. Han, and T. Wiegand, "Overview of the high efficiency video coding (HEVC) standard," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 22, no. 12, pp. 1649–1668, Dec. 2012.

[10] J. Cong, "Compilation for customized computing—From single-chips to data centers," in *Asia South Pacific Design Autom. Conf.* Macau, Jan. 25–28, 2016, keynote.

[11] L. Carloni, "High-level synthesis of accelerators in embedded scalable platforms," in *Asia South Pacific Design Autom. Conf.* Macau, Jan. 25–28, 2016.

Andres Takach is a Senior Architect and R&D manager at Mentor Graphics working on high-level synthesis and power optimization and integration of HLS with Formal Verification. He chairs Accellera's SystemC Synthesis Working Group. He rejoined Mentor in 2015 after his HLS group was spun out into Calypto in 2011. Prior to Calypto, he was a Chief Scientist at Mentor where he worked on HLS since 1997. From 1993 to 1997, he was Professor at Illinois Institute of Technology where he conducted research on HLS and Hardware/Software Codesign. Takach has a PhD from Princeton University and an MSEE and a BSEE from the University of Wisconsin-Madison.

■ Direct questions and comments about this article to Andres Takach, Mentor Graphics, Wilsonville, Oregon 97070-7777, USA; andres_takach@mentor.com.