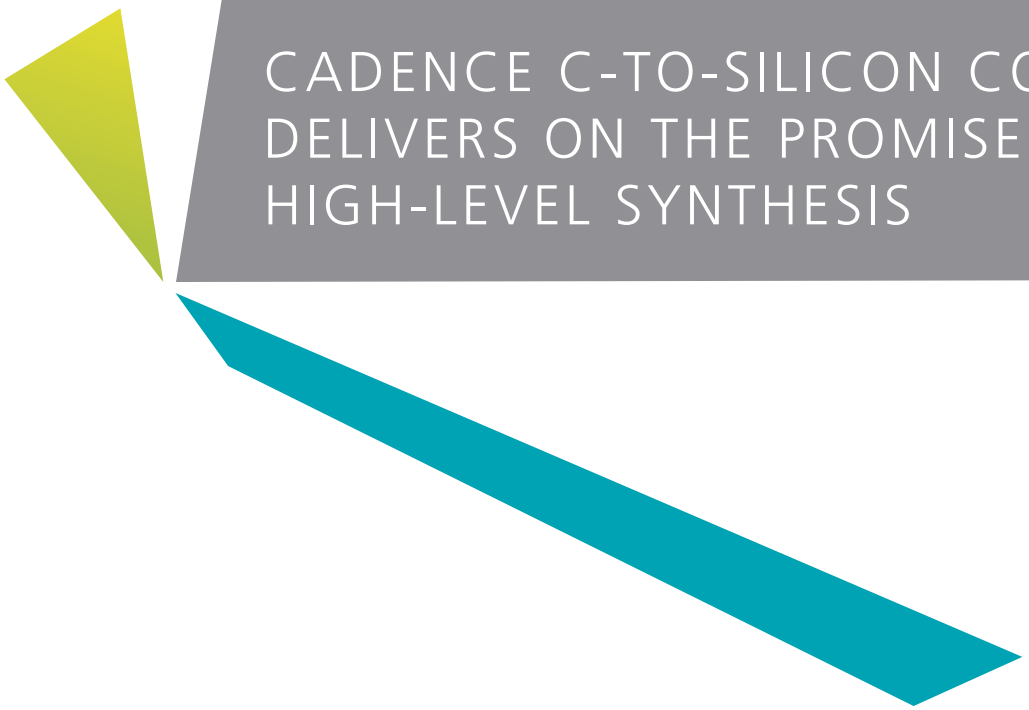
TECHNICAL PAPER

CADENCE C-TO-SILICON COMPILER DELIVERS ON THE PROMISE OF HIGH-LEVEL SYNTHESIS



cadence™

INTRODUCTION

Good RTL design is about more than just implementing an algorithm in hardware. It involves design space exploration and analysis to determine the optimum micro-architecture for the target application, while maximizing the design's reusability in other applications and portability to other manufacturing processes. All this must also be done in a way that takes into account the practical realities of implementation, such as timing closure and late-stage engineering change orders. Additionally, since the RTL is often required for software development and hardware/software (HW/SW) co-verification, dimensional analyses must be executed as early as possible in the design process.

RTL developers have always found these design tasks challenging, and high-level synthesis (HLS) has long attempted to automate the process. But the hard reality is that HLS tools have been unable to deliver the quality of results (QoR) and expected net productivity gains to justify broad adoption. Consequently, RTL developers work much the same way today as they did in the early 1990s—except for the fact that today's IC designs are 50 to 100x larger and more complex.

Now, Cadence® C-to-Silicon Compiler—the next-generation of HLS technology—eliminates historical barriers to HLS adoption by delivering the quality of results and net productivity gains that engineers need. It produces RTL with better quality than average human-generated design, while increasing engineering productivity up to 10x.

HISTORICAL BARRIERS TO HLS ADOPTION

Engineers evaluating current generation HLS tools have reported the following limitations:

1. Unsatisfactory QoR for designs with mixed datapath and control logic

   HLS tools have not been able to deliver consistent and reliable RTL QoR across a broad set of design styles—especially mixed control/datapath designs—because they lack full integration with the implementation flow. Until now, current-generation HLS tools have at best had only a loose coupling to logic synthesis via TCL scripts. This lack of integration prevents the HLS tool from obtaining the continuous, detailed area, timing and power feedback required to achieve high QoR. Analogous to the link between logic synthesis tools and physical design tools, HLS tools must deploy embedded logic synthesis in an integrated design process to achieve good results.

2. Inability to perform incremental ECO synthesis

   Regardless of whether a change is large or small, current-generation HLS tools will generally require re-synthesis of the entire algorithm(s). Consequently, even small changes in the algorithm often result in the generation of totally different RTL, so that the designer must then repeat verification and logic synthesis of the entire design with each iteration. The additional re-synthesis and re-verification effort nullifies any productivity gains and frequently causes designers to switch their "golden" source from high-level of abstraction to RTL. A better solution would be for the HLS tools to generate new RTL for only those portions of the algorithm that have been changed, and leave the rest of the design untouched.

3. Designs are difficult to retarget and reuse

   Using current-generation HLS tools, design reuse can be just as tedious and time-consuming as with manually developed RTL because the input from those tools typically fails to separate functionality from implementation constraints. The algorithm input description is usually modified with pragmas and logic synthesis directives—implementation-specific information that is generally not reusable with a different process node or other applications. Reuse and retargeting thus require extensive rework of the design to modify existing pragmas and directives, or add new ones. This drawback, combined with poor flexibility and controllability resulting from lack of incremental synthesis capability, leads users to conclude that it is ultimately easier and more productive to write RTL manually.

4. Limited verification support

   The current-generation HLS tools generally lack the support required to perform fast and reliable verification of the synthesizable RTL output. Much of the IP generated today includes both hardware and software, and in many situations (such as with drivers, firmware or diagnostics) the software performance is heavily dependent on the timing of the hardware. While Sequential Logic Equivalency Checkers (such as Calypto's) can help with static verification, they cannot dynamically check the hardware working properly with the software. Unless the user ports the synthesizable RTL output into emulation or FPGA prototyping, the verification of the IP with the rest of the system and software require unacceptably long RTL simulations. To get around this limitation, the user is often forced to manually create a SystemC model of the IP with timing. This generally works, but exposes the user to the risk of manual translation errors and creates additional design and verification burdens (along with a need to maintain two different "golden" sources), all of which diminishes the productivity gains from high-level synthesis.

WHICH INPUT LANGUAGE? C, C++, OR SYSTEMC?

In addition to the issues outlined above, deficiencies in the input language have historically limited the type and size of designs that the HLS tool can synthesize. For example, some current-generation HLS tools use the C language. Because C is sequential, untimed, non-hierarchical, and non-object-oriented, it cannot effectively represent the parallelism, hierarchy and internal interdependencies of many algorithms. Often, the only way that a C-based HLS tool can identify and extract parallelism from an algorithm is to constrain the language constructs used—such as eliminating pointers—in order to make the parallelism more visible. Such tools are generally restricted to synthesis of datapath designs because synthesis of control algorithms requires hierarchy and interdependency information that C lacks. Without hierarchy, such tools cannot easily handle large designs. In summary, as an HLS input language, C is a poor choice because of its inability to adequately capture hierarchy, hardware behavior, structure and timing (BST), all of which are critical to high-level synthesis.

C++-based HLS tools have been marginally more successful because the object-oriented nature of the C++ language at least enables capture of the design description in a modular and hierarchical manner. However, C++ still lacks the necessary semantics to represent hardware attributes such as timing, bit-accuracy, asynchronicity, and/or concurrency.

SystemC, which is a superset of C/C++, is the ideal system-level specification language for HLS. SystemC was developed as a simple, IEEE standard C++ class library that retains features of C++ while adding the ability to represent timing, bit-widths, concurrency, etc. for use in both synthesis and verification. SystemC eliminates the aforementioned limitations of C/C++, making it possible to synthesize and verify designs of any size having both datapath and control functionality. For these reasons, more recent HLS tools have been built around or retrofitted to work with SystemC. The combination of the new SystemC TLM 2.0 and future standardization of a synthesizable SystemC subset will further improve the capabilities of HLS tools to handle complex designs.

There is, however, an important caveat. It is still necessary for SystemC-based HLS tools to support many of the rich semantics of C/C++, and support features such as automatic transformation of pointer-based algorithm descriptions into hardware. Without such capabilities, any SystemC-based HLS tool will be severely limited, and will all too often require designers to extensively recode their C/C++ algorithms.

Taken together, all these barriers have restricted the use of current-generation HLS tools to niche applications and prevented them from achieving widespread adoption.

WHAT DOES C-TO-SILICON COMPILER DELIVER?

Cadence C-to-Silicon Compiler is the next-generation HLS technology that eliminates all of these adoption barriers. C-to-Silicon Compiler fully supports algorithmic models written in C, C++ and SystemC, and also serves as the front-end of an integrated algorithm-to-layout design and verification flow. It produces RTL with better quality than average human-generated RTL code, while increasing designer productivity up to 10x. C-to-Silicon Compiler achieves these results by leveraging four unique features and capabilities

- **Embedded Logic Synthesis (ELS)** analyzes and synthesizes both control logic and datapath logic in a unified development environment, delivering better QoR than with manual methods.
- A unique **Behavior-Structure-Timing (BST) database** enables incremental synthesis for faster turnaround time to verification and implementation, "what if" scenarios, and a tight link between RTL and the original source file (patent pending).
- **Constraint-Functionality Separation (CFS)** technology strictly separates algorithm functionality from implementation detail, facilitating reuse across multiple applications and process technologies (patent pending).
- **Automatic generation of cycle-accurate, fast hardware models (FHM)** capture timing and functionality equivalent to the generated RTL, and execute at 80 to 90 percent of the speed of an untimed, programmer's view model (patent pending). FHMs thus provide an alternative for accelerating verification and enabling fast and accurate hardware/software co-development

Table 1 compares the features and benefits of C-to-Silicon Compiler with those of current-generation HLS tools:

Features	Benefits	C-to-Silicon Compiler	Current Generation HLS
Embedded Logic Synthesis (ELS)	Delivers accurate timing estimation, enabling better-than-manual QoR	Yes	No
Behavior-Structure-Timing (BST) Database	Enables "true" incremental synthesis, minimizing design and verification cycle-time for design changes	Yes	No
Constraint-Functionality Separation (FCS)	Maintains independence between functionality and constraints, facilitating reuse across multiple applications and process technologies	Yes	No
Fast Hardware Model (FHM)	RTL-equivalence (at I/Os) with 80% to 90% of untimed PV-model speeds, enabling earlier, faster verification and HW/SW co-development	Yes	No

Table 1: Cadence C-to-Silicon Compiler eliminates the barriers to HLS adoption

Realizing C-to-Silicon Compiler's unique capabilities required several years of development by researchers at Cadence Berkeley Labs, as well as major product development investments by Cadence and other development partners.

HOW DOES C-TO-SILICON COMPILER DELIVER THESE FEATURES AND CAPABILITIES?

How C-to-Silicon Compiler Delivers

1. High QoR using Embedded Logic Synthesis (ELS)

   ELS is one of the most important keys to better-than-manual QoR and high productivity delivered by Cadence C-to-Silicon Compiler. In fact, C-to-Silicon Compiler is the only HLS tool available today with this capability, which uses an "under the hood" Cadence RTL Compiler logic synthesizer to guide and refine the HLS process and to ensure the generated RTL will synthesize exactly as predicted (*see figure 1*).

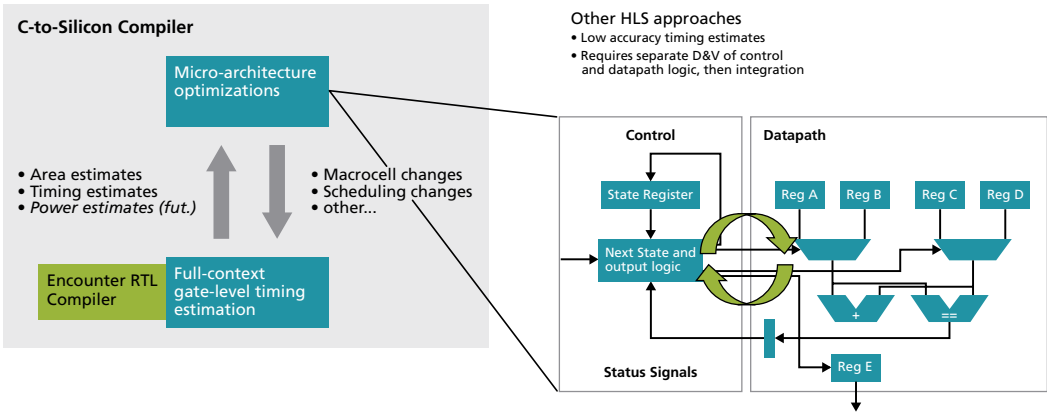


Figure 1: C-to-Silicon Compiler uses continuous feedback from Embedded Logic Synthesis to achieve optimum QoR

With its ability to obtain highly accurate timing estimates, C-to-Silicon Compiler overcomes one of the foremost challenges historically faced by HLS tools—optimal scheduling of hardware resources. With ELS, C-to-Silicon Compiler analyzes the logic in its full-context, using back-annotation of full-context, gate-level timing estimates from Cadence Encounter® RTL Compiler global synthesis. This accurate timing estimation contrasts with the estimation accuracy of the best current-generation HLS tools, which use pre-characterized technology libraries to make nominal gate delay estimates. These nominal estimates differ substantially from real timing because they do not comprehend implementation detail such as fan-in/fan-out variations and the presence or absence of buffers.

C-to-Silicon Compiler delivers full-context, gate-level timing and area estimation using ELS, which are critical to the parallel optimization of control and datapath logic. Because control logic elements are usually more spatially distributed than datapath logic, control logic is typically more sensitive to variations in accuracy of timing estimates. Consequently, with current-generation HLS tools, designers are generally forced to design and verify control and datapath logic separately, integrate them manually, and then reiterate the whole process if the result fails to meet the specification. This work requires users to repeat the verification process multiple times and forces manual intervention that can introduce more errors and consume additional time and resources. The accurate timing estimation and unique ability to jointly optimize control and datapath logic provided by C-to-Silicon Compiler thus delivers better-than-manual results with considerably less effort.

2. The Unique Behavior-Structure-Timing (BST) Database Enables Incremental Synthesis, Fast Turnaround Time to Verification and Implementation, "what if" Scenarios, and a Tight Link Between RTL and the Original Source File.

   By providing the ability to perform true incremental synthesis, C-to-Silicon Compiler eliminates the need to repeat the design/verification cycle every time a small design change is made. The BST database serves to map the relationships between the behavioral elements of the input algorithm to the structural elements implementing the algorithm. It also tracks all the various timing relationships, where different parts of the algorithm are mapped to execute across the different structural elements during different clock-cycles. The BST database thus enables the visibility and controllability to adequately track, map and analyze data transformations from SystemC source-files all the way through the HLS, logic synthesis, and physical design phases.

In practice, incremental synthesis using C-to-Silicon Compiler works as follows:

When formally releasing a design crafted with the C-to-Silicon Compiler to the implementation team, the designer saves the final BST database that stores these key relationships of the design. If during the implementation process the need arises to change some part of the RTL (perhaps a bug in the specification is discovered; or marketing wants one more feature; or floorplanning needs the timing of a few signals adjusted to accommodate a different layout), then the designer runs C-to-Silicon Compiler using the final BST that was saved as input. The C-to-Silicon GUI makes it simple for the designer to identify what parts of the C code were implemented by the particular RTL (and therefore which of those must be changed). To effect the design changes, the user modifies the input SystemC code (or perhaps just changes the constraints on how that code is implemented), and runs C-to-Silicon Compiler on the modified input design and/or constraints in ECO mode. C-to-Silicon Compiler then implements everything—wherever possible—the exact same way it did previously, producing different RTL/logic only where absolutely required. C-to-Silicon Compiler also produces a report clearly identifying all the new RTL/logic that was inserted so that the verification and implementation teams know what parts may need retesting and re-layout. Once the team is satisfied with the modified RTL, the designer will save the updated BST to use as a new baseline should other design changes arise later.

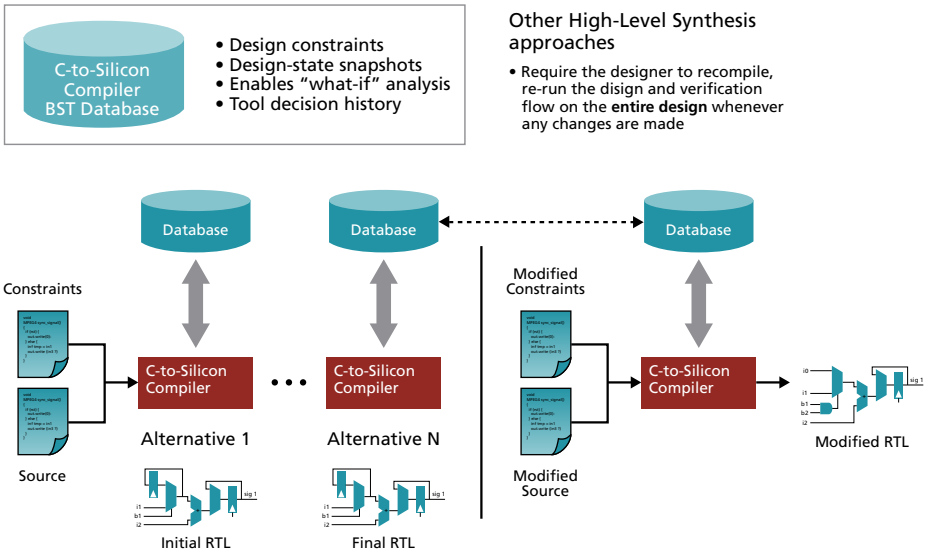


Figure 2. The C-to-Silicon Compiler BST database enables incremental synthesis

- **Fast turnaround time to design and verification**

  The combination of C-to-Silicon Compiler and Cadence Conformal® ECO Designer provides a full and fast incremental synthesis flow from C/C++/SystemC to implementation (silicon layout).

  The combination of C-to-Silicon Compiler incremental synthesis and the fast compile times provided by Cadence Incisive® Palladium® accelerator/emulator delivers a short turnaround time from C/C++/SystemC to full system verification (using emulation).

- **"What-if" analysis**

  The BST database records every decision being made by the C-to-Silicon Compiler tool, every step of the design transformation, and every user intervention. Therefore, the user can leverage this capability to "undo" the latest decision being done by the tool at any point, and provide different constraints/guidelines to influence the implementation depending on the specific application. This flexibility allows the user to perform "what-if" analysis and architecture trade-offs early in the design process with the ability to change and see the results immediately.

- **Tight link between RTL and the original source**

  The BST database enables transparent interactions between the RTL designer and the system engineer. Any critical path found in the RTL source can be linked into the equivalent path at the high-level of abstraction code. This capability encourages communication with the system designer and increases the probability of keeping the C/C++/SystemC code as the "golden" source during the final static timing analysis phase.

3. Design Reuse and Retargeting Using Constraint-Functionality Separation (CFS)

   The CFS technology in C-to-Silicon Compiler strictly separates design functionality ("the what") from the implementation/constraints ("the how"). Functional behavior is expressed as a C/C++ or SystemC model, while all information guiding/affecting implementation, structure, and timing for high-level synthesis is specified in one separate constraint-file. The CFS technology system supports a methodology in which users keep their source code as a pure algorithmic description of the function, and avoid corrupting that source code with any implementation or constraint-related information.

   The separate constraints file is a particular convenience with interrelated and interdependent parameters and constraints, where changes to some values require changes to others. With the SystemC functional model and the separate constraints file, users can quickly perform an extensive design space exploration to determine the optimum micro-architecture to implement the function for different applications, without having to maintain different copies of the algorithm source code for each case. For example, by changing just the constraints file, the same MPEG-4 video decoding algorithm can not only be retargeted at different manufacturing processes but can also be easily retargeted to wholly different target applications, such as a mobile phone, a laptop, or a big-screen TV, each with very different area and performance constraints (*see figure 3*). Additionally, if a bug is discovered in the algorithm (or any other change is required, such as upgrading to a new MPEG format), the change can be effected to the algorithm description only once, and then the same constraint files as before can be used as-is to quickly generate new chips for the different applications and technology nodes

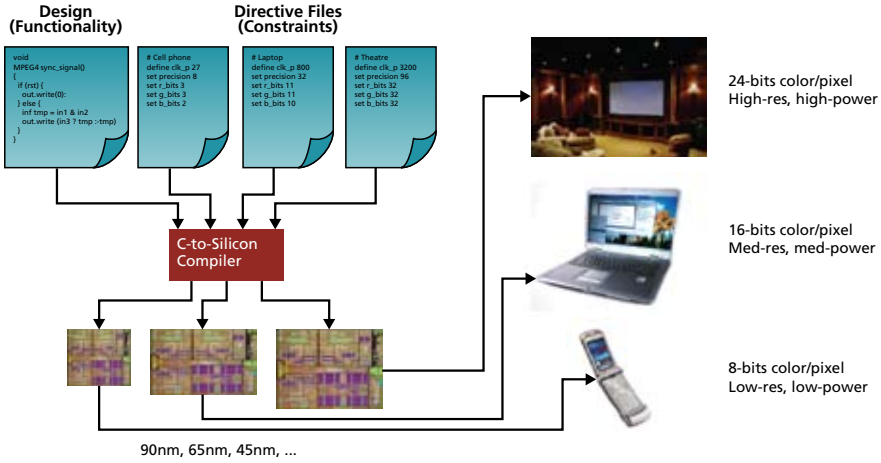


Figure 3: C-to-Silicon Compiler retargets the design using only the separate constraints file

4. Verification, HW/SW Co-Development and the Fast Hardware Model (FHM)

   C-to-Silicon Compiler automatically generates a cycle-accurate SystemC FHM of the RTL functionality (*see figure 4*). The FHM executes at 80 to 90% of untimed programmer's view (PV) model speeds, enabling early, fast and accurate verification and hardware/software co-development.

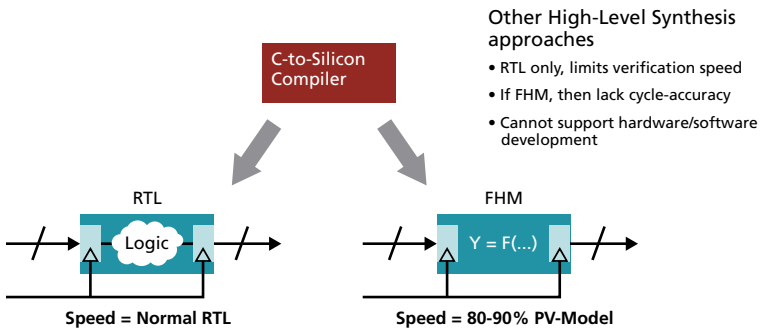


Figure 4: Fast Hardware Model simulates at near untimed programmer's view model speeds

This capability is made possible by the fact that C-to-Silicon Compiler generates both the system-level FHM and RTL from the same BST database, ensuring they are both precise images of the same design, with the same functionality and timing of their input/output behaviors. The FHM is instrumented with extensions available in the Cadence Incisive® verification environment to provide extra visibility of variables and signals, enabling analysis and debugging. In this way, using the FHM generated by C-to-Silicon Compiler enables the designer to leverage the entire Cadence Incisive SystemC-to-RTL verification infrastructure.

PUTTING IT ALL TOGETHER: C-TO-SILICON WITHIN THE COMPLETE DESIGN ENVIRONMENT

Unlike other HLS technologies, which were developed as point tools, Cadence C-to-Silicon Compiler was developed from the outset to integrate with Cadence design and verification tools and flows (*see figure 5*). Because C-to-Silicon Compiler uses precise feedback from Encounter RTL Compiler global synthesis, the generated RTL is guaranteed to synthesize gates exactly as predicted when later fed into RTL Compiler. C-to-Silicon Compiler also was built using the Incisive functional verification platform as its regression system, which was instrumental in enabling Cadence to bring this product to market quickly and with high quality. The C-to-Silicon Compiler flow using the Cadence Palladium and Cadence Xtreme® hardware emulation/acceleration product line was extensively used internally and externally as well.

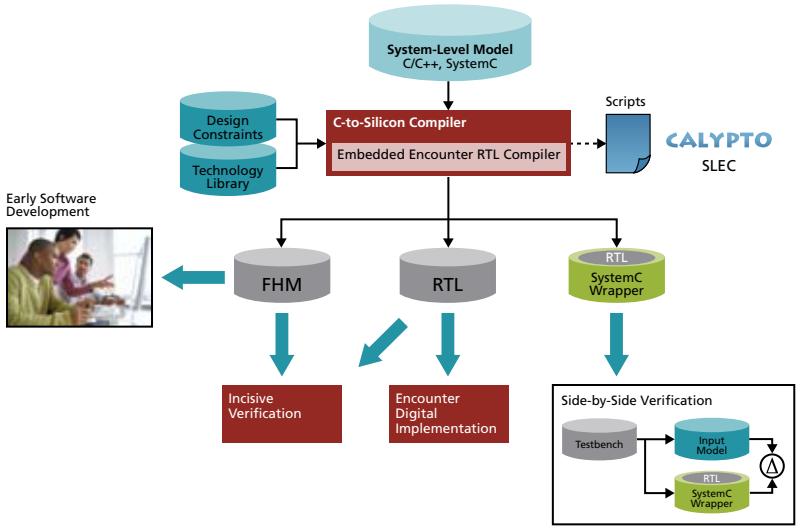


Figure 5: C-to-Silicon Compiler leverages comprehensive flows and infrastructure

C-to-Silicon Compiler fully supports both testbench-based and formal-based functional verification methodologies to ensure that the generated RTL and FHM are truly functionally equivalent to the system-level input model.

In the testbench-based method, C-to-Silicon Compiler automatically generates SystemC wrappers for the RTL, enabling rapid integration into a system-level verification environment for side-by-side simulation with the SystemC functional description. The SystemC wrapper provides the RTL with the same external interfaces as the SystemC model, so the designer can connect both of those to the same system-level testbench. Internally, the wrapper instantiates the synthesized RTL, connects to the RTL's inputs and outputs, and converts between RTL and SystemC data types. The wrapper is optimized to work with the Cadence Incisive Simulation, but can be modified for use with other simulators.

In the formal-based method, C-to-Silicon Compiler automatically generates scripts for sequential logic equivalence checking (SLEC). SLEC verifies that the FHM and RTL are functionally equivalent by mathematically analyzing if—when starting from their reset states—both the SystemC model and RTL would always produce equivalent output sequences in response to the same input sequences. SLEC can verify both cycle-by-cycle equivalence and transaction-level equivalence.

SUMMARY

By eliminating each of the previous barriers to HLS adoption, Cadence C-to-Silicon Compiler finally delivers on the promise of High Level Synthesis—which numerous HLS tools and technologies over the past two decades have failed to fulfill. With its unique Embedded Logic Synthesis, Behavior-Structure-Timing Database, Constraint-Functionality Separation, and Fast Hardware Model generation capability, Cadence C-to-Silicon Compiler delivers better-than-manual QoR, order-of-magnitude increases in designer productivity, easy IP reuse and retargeting across applications and manufacturing processes, and faster verification and earlier hardware/software co-development—all while integrating smoothly with the rest of the existing integrated circuit verification and implementation flow. In this way, C-to-Silicon Compiler does all of the things that HLS was meant to do.

'Clean C' targets standard for multicore programming by Richard Goering, www.SCDsource.com

For more information about this and other products contact:

info@cadence.com

or log on to:

www.cadence.com

cadence™

Cadence Design Systems, Inc.

CORPORATE HEADQUARTERS
2655 Seely Avenue
San Jose, CA 95134
P: +1.800.746.6223 *(within US)*
+1.408.943.1234 *(outside US)*
F: +1.408.943.5001
www.cadence.com

20553 07/08 MK/FLD/CS/PDF