An Introduction to High-Level Synthesis

Philippe Coussy
Université de Bretagne-Sud, Lab-STICC

Michael Meredith
Forte Design Systems

Daniel D. Gajski
University of California, Irvine

Andres Takach
Mentor Graphics

Editor's note:
High-level synthesis raises the design abstraction level and allows rapid generation of optimized RTL hardware for performance, area, and power requirements. This article gives an overview of state-of-the-art HLS techniques and tools.

—Tim Cheng, Editor in Chief

The growing capabilities of silicon technology and the increasing complexity of applications in recent decades have forced design methodologies and tools to move to higher abstraction levels. Raising the abstraction levels and accelerating automation of both the synthesis and the verification processes have for this reason always been key factors in the evolution of the design process, which in turn has allowed designers to explore the design space efficiently and rapidly.

In the software domain, for example, machine code (binary sequence) was once the only language that could be used to program a computer. In the 1950s, the concept of assembly language (and assembler) was introduced. Finally, high-level languages (HLLs) and associated compilation techniques were developed to improve software productivity. HLLs, which are platform independent, follow the rules of human language with a grammar, a syntax, and a semantics. They thus provide flexibility and portability by hiding details of the computer architecture. Assembly language is today used only in limited scenarios, primarily to optimize the critical parts of a program when there is an absolute need for speed and code compactness, or both. However, with the growing complexity of both modern system architectures and software applications, using HLLs and compilers clearly generates better overall results. No one today would even think of programming a complex software application solely by using an assembly language.

In the hardware domain, specification languages and design methodologies have evolved similarly.[1,2] For this reason, until the late 1960s, ICs were designed, optimized, and laid out by hand. Simulation at the gate level appeared in the early 1970s, and cycle-based simulation became available by 1979. Techniques introduced during the 1980s included place-and-route, schematic circuit capture, formal verification, and static timing analysis. Hardware description languages (HDLs), such as Verilog (1986) and VHDL (1987), have enabled wide adoption of simulation tools. These HDLs have also served as inputs to logic synthesis tools leading to the definition of their synthesizable subsets. During the 1990s, the first generation of commercial high-level synthesis (HLS) tools was available commercially.[3,4] Around the same time, research interest on hardware-software codesign—including estimation, exploration, partitioning, interfacing, communication, synthesis, and cosimulation—gained momentum.[5] The concept of IP core and platform-based design started to emerge.[6-8] In the 2000s, there has been a shift to an electronic system-level (ESL) paradigm that facilitates exploration, synthesis, and verification of complex SoCs.[9] This includes the introduction of languages with system-level abstractions, such as SystemC (http://www.systemc.org), SpecC (http://www.cecs.uci.edu/~specc), or SystemVerilog (IEEE 1800-2005; http://standards.ieee.org), and the introduction of transaction-level modeling (TLM). The ESL paradigm shift caused by the rise of system complexities, a multitude of components in a product (hundreds of processors in a car, for instance),

0740-7475/09/$26.00 © 2009 IEEE Copublished by the IEEE CS and the IEEE CASS

a multitude of versions of a chip (for better product differentiation), and an interdependency of component suppliers forced the market to focus on hardware and software productivity, dependability, interoperability, and reusability. In this context, processor customization and HLS have become necessary paths to efficient ESL design.[10] The new HLS flows, in addition to reducing the time for creating the hardware, also help reduce the time to verify it as well as facilitate other flows such as power analysis.

Raising the hardware design's abstraction level is essential to evaluating system-level exploration for architectural decisions such as hardware and software design, synthesis and verification, memory organization, and power management. HLS also enables reuse of the same high-level specification, targeted to accommodate a wide range of design constraints and ASIC or FPGA technologies.

Typically, a designer begins the specification of an application that is to be implemented as a custom processor, dedicated coprocessor or any other custom hardware unit such as interrupt controller, bridge, arbiter, interface unit, or a special function unit with a high-level description capture of the desired functionality, using an HLL. This first step thus involves writing a functional specification (an untimed description) in which a function consumes all its input data simultaneously, performs all computations without any delay, and provides all its output data simultaneously. At this abstraction level, variables (structure and array) and data types (typically floating point and integer) are related neither to the hardware design domain (bits, bit vectors) nor to the embedded software. Realistic hardware implementation thus requires conversion of floating-point and integer data types into bit-accurate data types of specific length (not a standard byte or word size, as in software) with acceptable computation accuracy, while generating an optimized hardware architecture starting from this bit-accurate specification.

HLS tools transform an untimed (or partially timed) high-level specification into a fully timed implementation.[10-13] They automatically or semiautomatically generate a custom architecture to efficiently implement the specification. In addition to the memory banks and the communication interfaces, the generated architecture is described at the RTL and contains a data path (registers, multiplexers, functional units, and buses) and a controller, as required by the given specification and the design constraints.

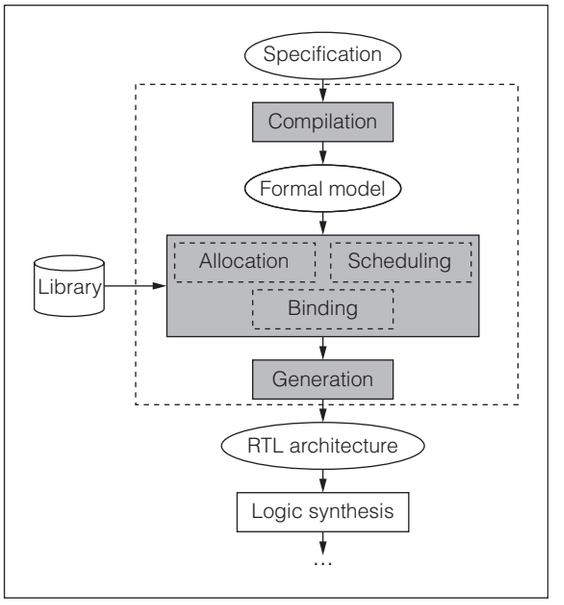

Figure 1. High-level synthesis (HLS) design steps.

Key concepts

Starting from the high-level description of an application, an RTL component library, and specific design constraints, an HLS tool executes the following tasks (see Figure 1):

1. compiles the specification,
2. allocates hardware resources (functional units, storage components, buses, and so on),
3. schedules the operations to clock cycles,
4. binds the operations to functional units,
5. binds variables to storage elements,
6. binds transfers to buses, and
7. generates the RTL architecture.

Tasks 2 through 6 are interdependent, and for a designer to achieve the optimal solution, they would ideally be optimized in conjunction. To handle real-world designs, however, the tasks are commonly executed in sequence to manage the computational complexity of synthesis. The particular order of some of the synthesis tasks, as well as a measure of how well the interdependencies are estimated and accounted for, significantly impacts the generated design's quality. More details are available elsewhere.[10-13]

Compilation and modeling

HLS always begins with the compilation of the functional specification. This first step transforms the input description into a formal representation.

This first step traditionally includes several code optimizations such as dead-code elimination, false data dependency elimination, and constant folding and loop transformations.

The formal model produced by the compilation classically exhibits the data and control dependencies between the operations. Data dependencies can be easily represented with a data flow graph (DFG) in which every node represents an operation and the arcs between the nodes represent the input, output, and temporary variables.[12] A pure DFG models data dependencies only. In some cases, it is possible to get this model by removing the control dependencies of the initial specification from the model at compile time. To do so, loops are completely unrolled by converting to noniterative code blocks, and conditional assignments are resolved by creating multiplexed values. The resulting DFG explicitly exhibits all the intrinsic parallelism of the specification. However, the required transformations can lead to a large formal representation that requires considerable memory to be stored during synthesis. Moreover, this representation does not support loops with unbounded iteration count and nonstatic control statements such as `goto`. This limits the use of pure DFG representations to a few applications. The DFG model has been extended by adding control dependencies: the control and data flow graph (CDFG).[12-15] A CDFG is a directed graph in which the edges represent the control flow. The nodes in a CDFG are commonly referred to as *basic blocks* and are defined as a straight-line sequence of statements that contain no branches or internal entrance or exit points. Edges can be conditional to represent `if` and `switch` constructs. A CDFG exhibits data dependencies inside basic blocks and captures the control flow between those basic blocks.

CDFGs are more expressive than DFGs because they can represent loops with unbounded iterations. However, the parallelism is explicit only within basic blocks, and additional analysis or transformations are required to expose parallelism that might exist between basic blocks. Such transformations include for example loop unrolling, loop pipelining, loop merging, and loop tiling. These techniques, by revealing the parallelism between loops and between loop iterations, are used to optimize the latency or the throughput and the size and number of memory accesses. These transformations can be realized automatically,[14] or they can be user-driven.[10] In addition to control dependencies, data dependencies between basic blocks can be added to the CDFG model as shown in the hierarchical task graph representation used in the SPARK tool.[14,16]

Allocation

Allocation defines the type and the number of hardware resources (for instance, functional units, storage, or connectivity components) needed to satisfy the design constraints. Depending on the HLS tool, some components may be added during scheduling and binding tasks. For example, the connectivity components (such as buses or point-to-point connections among components) can be added before or after binding and scheduling tasks. The components are selected from the RTL component library. It's important to select at least one component for each operation in the specification model. The library must also include component characteristics (such as area, delay, and power) and its metrics to be used by other synthesis tasks.

Scheduling

All operations required in the specification model must be scheduled into cycles. In other words, for each operation such as $a = b$ op c, variables b and c must be read from their sources (either storage components or functional-unit components) and brought to the input of a functional unit that can execute operation op, and the result a must be brought to its destinations (storage or functional units). Depending on the functional component to which the operation is mapped, the operation can be scheduled within one clock cycle or scheduled over several cycles. Operations can be chained (the output of an operation directly feeds an input of another operation). Operations can be scheduled to execute in parallel provided there are no data dependencies between them and there are sufficient resources available at the same time.

Binding

Each variable that carries values across cycles must be bound to a storage unit. In addition, several variables with nonoverlapping or mutually exclusive lifetimes can be bound to the same storage units. Every operation in the specification model must be bound to one of the functional units capable of executing the operation. If there are several

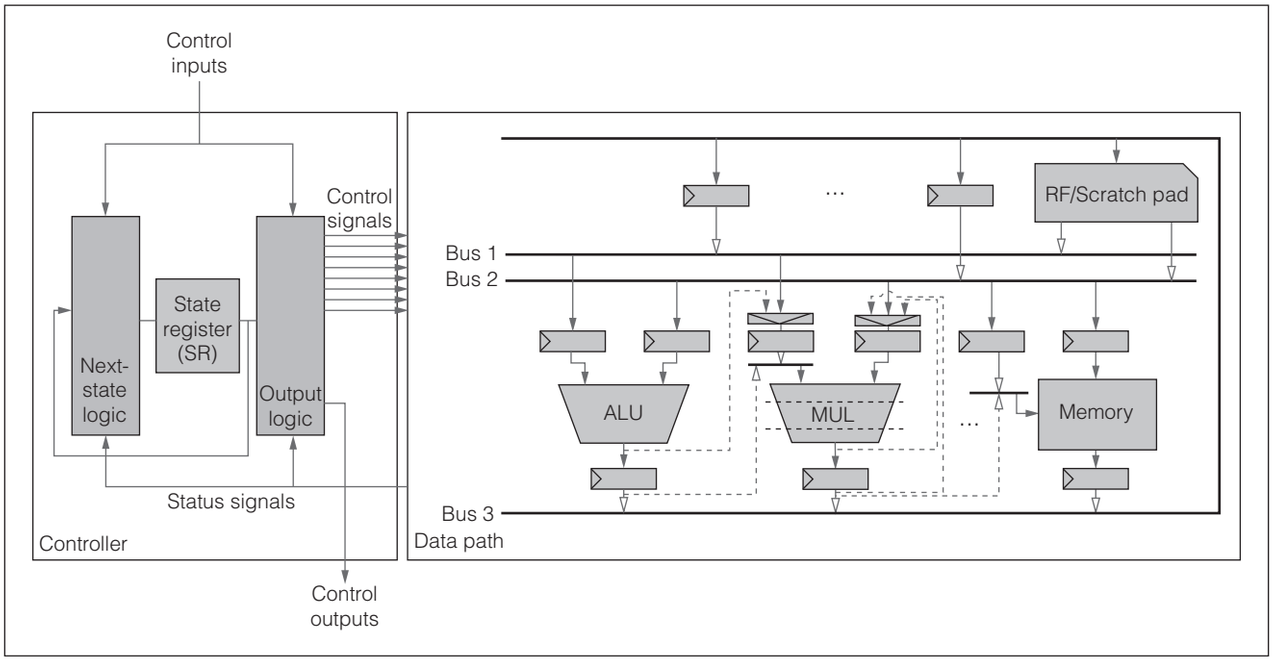

Figure 2. Typical architecture.

units with such capability, the binding algorithm must optimize this selection. Storage and functional-unit binding also depend on connectivity binding, which requires that each transfer from component to component be bound to a connection unit such as a bus or a multiplexer (see, for example, http://www-labsticc.univ-ubs.fr/www-gaut/). Ideally, high-level synthesis estimates the connectivity delay and area as early as possible so that later HLS steps can better optimize the design. An alternative approach is to specify the complete architecture during allocation so that initial floorplanning results can be used during binding and scheduling (see http://www.cecs.uci.edu/~nisc).

Generation

Once decisions have been made in the preceding tasks of allocation, scheduling, and binding, the goal of the RTL architecture generation step is to apply all the design decisions made and generate an RTL model of the synthesized design.

Architecture. The RTL architecture is implemented by a set of register-transfer components. It usually includes a controller and a data path (see Figure 2). A data path consists of a set of storage elements (such as registers, register files, and memories), a set of functional units (such as ALUs, multipliers, shifters, and other custom functions), and interconnect elements (such as tristate drivers, multiplexers, and buses). All these register-transfer components can be allocated in different quantities and types and connected arbitrarily through buses. Each component can take one or more clock cycles to execute, can be pipelined, and can have input or output registers. In addition, the entire data path and controller can be pipelined in several stages.

Primary input and output ports of the design interface with the external world to transfer both data and control (used for interface protocol handshaking and synchronization). Data inputs and outputs are connected to the data path, and control inputs and outputs are connected to the controller. There are also control signals from the controller to the data path and status signals from the data path to the controller. However, some architectures may not have all the connectivity just described, and in general some of the controller functions may be implemented as part of the data path—for example, a counter plus other logic in the data path that generates control signals.

The controller is a finite state machine that orchestrates the flow of data in the data path by setting the values of control signals (also called control

word) such as the select inputs of functional units, registers, and multiplexers. The inputs to the controller may come from primary inputs (control inputs) or from the data path components such as comparators and so on (status signals). The controller consists of a state register (SR), next-state logic, and output logic. The SR stores the present state of the processor, which is equal to the present state of the finite-state machine (FSM) model describing the controller's operation. The next-state logic computes the next state to be loaded into the SR, whereas the output logic generates the control signals and the control outputs.

The controller of a simple dedicated coprocessor is classically implemented with hardwired logic gates. On the other hand, a controller can be programmable with a read-write or read-only program memory for a specific custom processor. In this case, the program memory can store instructions or just control words, which are longer but require no decoding. In such a circumstance, SR is called a program counter, the next-state logic is an address generator, and the output logic is RAM or ROM.

Output model. According to the decisions made in the binding tasks, the description of the architecture can be written on RTL with different levels of detail (that is, without binding or with partial or complete binding). For example, $a = b + c$ executing in state (n) can be written as Figure 3 indicates:

```
Without any binding:
  state (n): a = b + c;
  go to state (n + 1);

With storage binding:
  state (n): RF(1) = RF(3) + RF(4);
  go to state (n + 1);

With functional-unit binding:
  state (n): a = ALU1 (+, b, c);
  go to state (n + 1);

With storage and functional-unit binding:
  state (n): RF(1)= ALU1 (+, RF(3), RF(4));
  go to state (n + 1);

With storage, functional-unit, and connectivity binding:
  state (n): Bus1 = RF(3); Bus2 = RF(4);
  Bus3 = ALU1 (+, Bus1, Bus2);
  RF(1) = Bus3;
  go to state (n + 1);
```

Figure 3. RTL description written with different binding details.

When the RTL description includes only partial binding of resources, the logic synthesis step that follows HLS must perform the binding task and the associated optimization. Leaving components unbound in the generated RTL provides the RTL and physical synthesis the flexibility to optimize the bindings on the basis of updated timing estimates that take into account wire loads due to physical (floorplanning and place-and-route) considerations.

Several design flows

Allocation, scheduling, and binding can be performed simultaneously or in specific sequence depending on the strategy and algorithms used. However, they are all interrelated. If they are performed together, the synthesis process becomes too complex to be applied to realistic examples. The order in which they are realized depends on the design constraints and the tool's objectives. For example, allocation will be performed first when scheduling tries to minimize the latency or to maximize the throughput under a resource constraint. Allocation will be determined during scheduling when scheduling tries to minimize the area under timing constraints.[17] Resource-constrained approaches are used when a designer wants to define the data path architecture,[18,19] or wants to accelerate an application by using an FPGA device with a limited amount of available resources.[20] Time-constrained approaches are used when the objective is to reduce a circuit's area while meeting an application's throughput requirements, as in multimedia or telecommunication applications.[21]

In practice, the resource-constrained problem can be solved by using a time-constrained approach or tool (and vice versa). In this case, the designer relaxes the timing constraints until the provided circuit area is acceptable. Latency, throughput, resource count, and area are now well-known constraints and objectives. However, recent work has considered features such as clock period, memory bandwidth, memory mapping, power consumption, and so forth that make the synthesis problem even more difficult to solve.[10,22-24]

Another example of how synthesis tasks can be ordered concerns the allocation and the binding steps. The types and numbers of resources determined in the allocation task are taken as input for the binding task. In practical HLS tools, however, resources are often allocated only partially. Additional resources

are allocated during the binding step according to the design constraints and objectives. These additional resources can be of any type: functional units, multiplexers, or registers. Hence, functional units can be first allocated to schedule and bind the operations. Both registers and multiplexers can then be allocated (created) during the variable-to-register binding step.

The synthesis tasks can be performed manually or automatically. Obviously, many strategies are possible, as exemplified by available EDA tools: these tools might perform each of the aforementioned tasks only partially in automatic fashion and leave the rest to the designer.[10]

Industrial tools

Here, we take a brief look first at commercially available HLS tools for specifying the input description, and then we more carefully examine two state-of-the-art industrial HLS tools.

Input languages and tools

The input specification must capture the intended design functionality at a high abstraction level. Rather than coding low-level implementation details, the designer uses the automation provided by the HLS tool to guide the design decisions, which heavily depend on performance goals and the target technology. For instance, if there is parallelism in an HLS specification, it is extracted using dataflow analysis in accordance with the target technology's capabilities and performance goals (on a slow technology, more parallelism is required to achieve the performance goal).

The latest generation of HLS tools, in most cases, uses either ANSI C, C++, or languages such as SystemC that are based on C or C++ that add hardware-specific constructs such as timing, hardware hierarchy, interface ports, signals, explicit specification of parallelism, and others. Some HLS tools that support C or C++ or derivatives are Mentor's Catapult C (C, C++), Forte's Cynthesizer (SystemC), NEC's CyberWorkbench (C with hardware extensions), Synfora's PICO (C), and Cadence's C-to-Silicon (SystemC). Other languages used for high-level modeling are MathWork's Matlab and Simulink. Tools that support Matlab or Simulink are Xilinx' AccelDSP (Matlab) and Synopsys' SimplifyDSP (Simulink); both use an IP approach to generate the hardware implementation. An alternative approach for generating an implementation is to use a configurable processor approach, which is the approach of both CoWare's Processor Designer and Tensilica's Xtensa.

Other languages, not based on C or C++ but which are tailored to specific domains, have also been proposed. Esterel is a synchronous language for the development of reactive systems. Esterel Studio can generate either software or hardware. Bluespec's BSV is language targeted for specifying concurrency with rule-based atomic transactions.

The input specification to HLS tools must be written with some hardware implementation in mind to get the best results. For example, video line buffering must be coded as part of the algorithm to generate high-throughput designs.[25] Ideally, such code restructuring still preserves much of the abstraction level. It is beyond the scope of this article to provide an overview of the specific writing styles required in the specification for various tools. As tools' capabilities evolve further, fewer modifications will be required to convert an algorithm written for software into an algorithm that is suitable as input to HLS. An analogous evolution has occurred with RTL synthesis—for example, with the optimization of arithmetic expressions.

Catapult C synthesis

Catapult takes an algorithm written in ANSI C++ and a set of user directives as input and generates an RTL that is optimized for the specified target technology. The input can be compiled by any standard compiler compliant with C++; pragmas and directives do not change the functional behavior of the input specification.

Synthesis input. The input specification is sequential and does not include any notion of time or explicit parallelism: it does not hardcode the interface or the design's architecture. Keeping the input abstract is essential because any hard-coding of interface and architectural details significantly limits the range of designs that HLS could generate. Required directives specify the target technology (component library) and the clock period. Optional directives control interface synthesis, array-to-memory mappings, amount of parallelism to uncover by loop unrolling, loop pipelining, hardware hierarchy and block communication, scheduling (latency or cycle) constraints, allocation directives to constrain the number or type of hardware resources, and so on.

Native C++ integer types as well as C++ bit-accurate integer and fixed-point data types are supported

for synthesis. The generated RTL faithfully reflects the bit-accurate behavior specified in the source. Publicly available integer and fixed-point data types provided by Mentor Graphics' (http://www.mentor.com/esl) Algorithmic C data types library (ANSI C++ header files) as well as the synthesizable subset of the SystemC integer and fixed-point data types are supported for synthesis. The support of C++ language constructs meets and exceeds the requirements stated in the most current draft of the Synthesis Subset OSCI standard (http://www.systemc.org).

Generating hardware from ANSI C or C++. One of the advantages of keeping the source untimed is that a very wide range of interfaces and architectures can be generated without changing the input source specification. Another advantage of an untimed source is that it avoids errors resulting from manual coding of architectural details. The interface and the architecture of the generated hardware are all under the control of the designer via synthesis directives. Catapult's GUI provides an interactive environment with both the control and the analysis tools to enable efficient exploration of the design space.

Interface synthesis makes it possible to map the transfer of data that is implied by passing of C++ function arguments to various hardware interfaces such as wires, registers, memories, buses, or more complex user-defined interfaces. All the necessary signals and timing constraints are generated during the synthesis process so that the generated RTL conforms and is optimized to the desired interfaces.

For example, an array in the C source might result in a hardware interface that streams the data or transfers the data through a memory, a register bank, and so forth. Selection of a streaming interface implies that the environment provides data in sequential index order whereas selection of a memory interface implies that the environment provides data by writing the array into memory. The granularity of the transfer size—for example, number of array elements provided as a stream transfer or as a memory word—is also specifiable as a user directive.

Hierarchy (block-level concurrency) can be specified by user directives. For example, a C function can be synthesized as a separate hardware block instead of being inlined. Hierarchy can also be specified in a style that corresponds to the Kahn process network computation model (still in sequential ANSI C++). The blocks are connected with the appropriate communication channels, and the required handshaking interfaces are generated to guarantee the correct execution of the specified behavior. The blocks can be synthesized to be driven by different clocks. The clock-domain-crossing logic is generated by Catapult. Communication is optimized according to user directives to enable maximal block-level concurrency of execution of the blocks using FIFO buffers (for streamed data) and ping-pong memories to enable block-level pipelining and thus improve the throughput of the overall design.

All the HLS steps consider accurate component area and timing numbers for the target ASIC or FPGA technology for the designer's RTL synthesis tool of choice. Accurate timing and area numbers for components are essential to generate RTL that meets timing and is optimized for area. During synthesis, Catapult queries the component library so that it can allocate various combinational or pipelining components with different performance and area trade-offs. The queried component library is precharacterized for the target technology and the target RTL synthesis tool. Component libraries can also be built by the designer to incorporate specific characterization for memories, buses, I/O interfaces, or other units of functionality such as pipelined components.

Verification and power estimation flows. The synthesis process generates the required verification infrastructure in SystemC so that the input stimuli from the original C++ testbench can be applied to the generated RTL to verify its functionality against the (golden) source C++ specification using simulation. The synthesis process also generates the required verification infrastructure (wrappers and scripts) to enable the use of sequential equivalence checking between the source C++ specification and the generated RTL. Automatic generation of the verification infrastructure is essential because the interface of the generated hardware heavily depends on interface synthesis.

Power estimation flows with third-party tools let the designer gather switching activity for the design and obtain RTL and gate-level power estimates. By exploring various architectures, the designer can rapidly converge to a low-power design that meets the required performance and area goals.

Catapult has been successfully used in more than 200 ASIC tape-outs and several hundred FPGA designs. Typical applications include computation-intensive

algorithms in communications and video and image processing.

Cynthesizer SystemC synthesis

Cynthesizer takes a SystemC module containing hierarchy, multiple processes, interface protocols, and algorithms and produces RTL Verilog optimized to a specific target technology and clock speed. The target technology is specified by a user-provided .lib file or, for an FPGA implementation, by the user's identifying the targeted Xilinx or Altera part.

Synthesis input. The input to the HLS flow used with Cynthesizer is a pin- and protocol-accurate SystemC model. Because SystemC is a C++ class library, no language translation is required to reuse algorithms written in C++. The synthesizable subset is quite broad, including classes and structures, operator overloading, and C++ template specialization. Constructs that are not supported for synthesis include dynamic allocation (malloc, free, new, and delete), pointer arithmetic, and virtual functions.

The designer puts untimed high-level C++ into a hardware context using SystemC to represent the hardware elements such as ports, clock edges, structural hierarchy, bit-accurate data types, and concurrent processes. As Figure 4 shows, a synthesizable `SC_MODULE` can contain multiple `SC_CTHREAD` instances and multiple `SC_METHOD` instances along with submodule instances and signals for internal connections. I/O ports are signal-level SystemC *sc_in* and *sc_out* ports. `SC_MODULE` instances are C++ classes, so they can also contain C++ member functions and data members (variables), which represent the module behavior and local storage respectively.

Clocked thread processes implemented as SystemC `SC_CTHREAD` instances are used for the majority of the module functionality. They contain an infinite loop that implements the bulk of the functionality along with reset code that initializes I/O ports and variables. The SystemC clocked thread construct provides the needed reset semantics. Within a thread, the designer can combine untimed computation code with cycle-accurate protocol code. The designer determines the protocol by writing SystemC code containing port I/O statements and `wait()` statements. Cynthesizer uses a hybrid scheduling approach in which the protocol code is scheduled in a cycle-accurate way, honoring the clock edges specified by the designer as SystemC `wait()` statements.

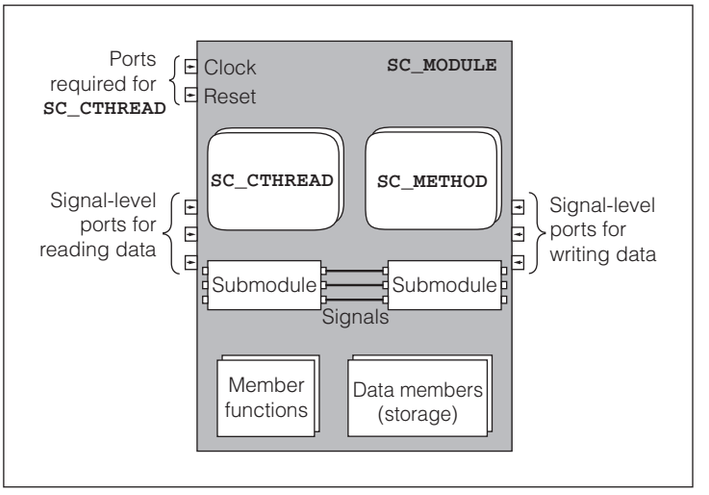

Figure 4. SystemC input for synthesis.

The computation code is written without any `wait()` statements and scheduled by the tool to satisfy latency, pipelining, and other constraints given by the designer.

Triggered methods implemented as SystemC `SC_METHOD` instances can also be used to implement behaviors that are triggered by activity on signals in a sensitivity list, similar to a Verilog `always` block. This allows a mix of high- and low-level coding styles to be used, if needed.

Complex subsystems are built and verified by combining modules using structural hierarchy, just as in Verilog or VHDL. The high-level models used as the input to synthesis can be simulated directly to validate both the algorithms and the way the algorithm code interacts with the interface protocol code. Multiple modules are simulated together to validate that they interoperate correctly to implement the functionality of the hierarchical subsystem.

Targeting a specific process technology. To ensure that the synthesized RTL meets timing requirements at a given clock rate using a specific foundry and process technology, the HLS tool requires accurate estimates of the timing characteristics of each operation. Cynthesizer uses an internal data path optimization engine to create a library of gate-level adders, multipliers, and so on. This takes a few hours for a specific process technology and clock speed and can be performed by the designer given any library file. Cynthesizer uses the timing and area characteristics of these components to make trade-offs and optimize the RTL. Designers have the

option of using the gates for implementation or of using RTL representations of the data path components for logic synthesis.

Synthesis output. Cynthesizer produces RTL Verilog for use with logic synthesis tools provided by EDA vendors for ASIC and FPGA technology.

The RTL consists of an FSM and a set of explicitly instantiated data path components such as multipliers, adders, and multiplexers. More-complex custom data path components that implement arithmetic expressions used in the design are automatically created, and the designer can specify sections of C++ code to be implemented as data path components. The multiplexers directing the dataflow through the data path components and registers are controlled by a conventional FSM consisting of a binary-encoded or one-hot state register and next-state logic implemented in Verilog `always` blocks.

Strengths of the SystemC flow. SystemC is a good fit for HLS because it supports a high level of abstraction and can directly describe hardware. It combines the high-level and object-oriented features of C++ with hardware constructs that let a designer directly represent structural hierarchy, signals, ports, clock edges, and so on. This combination of characteristics provides a very efficient design and verification flow in which behavioral models of multiple modules can be concurrently simulated to verify their combined algorithm and interface behavior. Most functional errors can be found and eliminated at this high-speed behavioral level, which eliminates the need for time-consuming RTL simulation to validate interfaces and system-level operation, and substantially reduces the overall number of slow RTL simulations required. Once the behavior is functionally correct, the models that were simulated are used directly for synthesis, eliminating opportunities for mistakes or misunderstanding.

One indicator of how far HLS has come since its early days in the late 1970s is the preponderance of different HLS tools that are available, both academically and commercially. However, many features must yet be added before these tools become as widely adopted as layout and logic synthesis tools. Moreover, many specific embedded-system applications need particular attention. As HLS moves from block-level to subsystem to full-system design, the interaction of hardware and software becomes both a challenge and an opportunity for further automation. Additional research is vital, because we are still a long way from HLS that automatically searches the design space without the designer's guidance and delivers optimal results for different design constraints and technologies. ■

References

1. A. Sangiovanni-Vincentelli, "The Tides of EDA," *IEEE Design & Test,* vol. 20, no. 6, 2003, pp. 59-75.
2. D. MacMillen et al., "An Industrial View of Electronic Design Automation," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems,* vol. 19, no. 12, 2000, pp. 1428-1448.
3. D.W. Knapp, *Behavioral Synthesis: Digital System Design Using the Synopsys Behavioral Compiler,* Prentice Hall, 1996.
4. J.P. Elliot, *Understanding Behavioral Synthesis: A Practical Guide to High-Level Design,* Kluwer Academic Publishers, 1999.
5. W. Wolf, "A Decade of Hardware/Software Co-design," *Computer,* vol. 36, no. 4, 2003, pp. 38-43.
6. H. Chang et al., *Surviving the SoC Revolution: A Guide to Platform-Based Design,* Kluwer Academic Publishers, 1999.
7. M. Keating and P. Bricaud, *Reuse Methodology Manual for System-on-a-Chip Designs,* Kluwer Academic Publishers, 1998.
8. D. Gajski et al., *SpecC: Specification Language and Methodology,* Kluwer Academic Publishers, 2000.
9. B. Bailey, G. Martin, and A. Piziali, *ESL Design and Verification: A Prescription for Electronic System Level Methodology,* Morgan Kaufman Publishers, 2007.
10. P. Coussy and A. Morawiec, eds., *High-Level Synthesis: From Algorithm to Digital Circuit,* Springer, 2008.
11. D. Ku and G. De Micheli, *High Level Synthesis of ASICs under Timing and Synchronization Constraints,* Kluwer Academic Publishers, 1992.
12. D. Gajski et al., *High-Level Synthesis: Introduction to Chip and System Design,* Kluwer Academic Publishers, 1992.
13. R.A. Walker and R. Camposano, eds., *A Survey of High-Level Synthesis Systems,* Springer, 1991
14. S. Gupta et al., *SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits,* Kluwer Academic Publishers, 2004.
15. A. Orailoglu and D.D. Gajski, "Flow Graph Representation," *Proc. 23rd Design Automation Conf.* (DAC 86), IEEE Press, pp. 503-509.

16. M. Girkar and C.D. Polychronopoulos, ''Automatic Extraction of Functional Parallelism from Ordinary Programs,'' *IEEE Trans. Parallel and Distributed Systems,* vol. 3, no. 2, 1992, pp.166-178.
17. P. Paulin and J.P. Knight, ''Algorithms for High-Level Synthesis,'' *IEEE Design and Test,* vol. 6, no. 6, 1989, pp. 18-31.
18. M. Reshadi and D. Gajski, ''A Cycle-Accurate Compilation Algorithm for Custom Pipelined Datapaths,'' *Proc. Int'l Symp. Hardware/Software Codesign and System Synthesis* (CODES+ISSS 05), ACM Press, 2005, pp. 21-26.
19. I. Auge and F. Petrot, ''User Guided High Level Synthesis,'' *High-Level Synthesis: From Algorithm to Digital Circuit,* P. Coussy and A. Morawiec, eds., Springer, 2008, pp. 171-196.
20. D. Chen, J. Cong, and P. Pan, *FPGA Design Automation: A Survey,* Now Publishers, 2006.
21. W. Geurts et al., *Accelerator Data-Path Synthesis for High-Throughput Signal Processing Applications,* Kluwer Academic Publishers, 1996.
22. L. Zhong and N.K. Jha, ''Interconnect-Aware Low-Power High-Level Synthesis,'' *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems,* vol. 24, no. 3, 2005, pp. 336-351.
23. M. Kudlur, K. Fan, and S. Mahlke, ''Streamroller: Automatic Synthesis of Prescribed Throughput Accelerator Pipelines,'' *Proc. Int'l Conf. Hardware/Software Codesign and System Synthesis* (CODES+ISSS 06), ACM Press, pp. 270-275.
24. M.C. Molina et al., ''Area Optimization of Multi-cycle Operators in High-Level Synthesis,'' *Proc. Design, Automation and Test in Europe Conf.* (DATE 07), IEEE CS Press, 2007, pp. 1-6.
25. G. Stitt, F. Vahid, and W. Najjar, ''A Code Refinement Methodology for Performance-Improved Synthesis from C,'' *Proc. IEEE/ACM Int'l Conf. Computer-Aided Design* (ICCAD 06), ACM Press, 2006, pp. 716-723.

Philippe Coussy is an associate professor in the Lab-STICC at the Université de Bretagne-Sud, France, where he leads the high-level synthesis (HLS) research group. His research interests include system-level design and methodologies, HLS, CAD for SoCs, embedded systems, and low-power design for FPGAs. He has a PhD in electrical and computer engineering from the Université de Bretagne-Sud. He is a member of the IEEE and the ACM.

Daniel D. Gajski holds the Henry Samueli Endowed Chair in Computer System Design at the University of California, Irvine, where he is also the director of the Center for Embedded Computer Systems. His research interests include embedded systems and information technology, design methodologies and e-design environments, specification languages and CAD software, and the science of design. He has a PhD in computer and information sciences from the University of Pennsylvania. He is a life member and Fellow of the IEEE.

Michael Meredith is the vice president of technical marketing at Forte Design Systems and serves as president of the Open SystemC Initiative. His research interests include development of printed-circuit board layouts, schematic capture, timing-diagram entry, verification, and high-level synthesis tools.

Andres Takach is chief scientist at Mentor Graphics. His research interests include high-level synthesis, low-power design, and hardware-software codesign. He has a PhD from Princeton University in Electrical and Computer Engineering. He is chair of the OSCI Synthesis Working Group.

■ Direct questions and comments about this article to Philippe Coussy, Lab-STICC, Centre de recherche, rue de saint maude, BP 92116, Lorient 56321, France; philippe.coussy@univ-ubs.fr.