
THE OLYMPUS SYNTHESIS SYSTEM

GIOVANNI DE MICHELI
DAVID KU
FRÉDÉRIC MAILHOT
THOMAS TRUONG
Stanford University

Olympus, a synthesis system for digital design developed at Stanford University, is a vertically integrated set of tools for multilevel synthesis, technology mapping, and simulation. The system supports the synthesis of ASICs from behavioral descriptions written in a hardware description language called HardwareC. Two internal models represent the hardware at different levels of abstraction and provide a way to pass design information among different tools. Olympus has been used to design three ASIC chips at Stanford, and it has been tested against benchmark circuits for high-level and logic synthesis.

As digital hardware systems become bigger and more complex, engineers need powerful synthesis tools to aid in the design of digital circuits. Synthesis systems have proved effective in supporting the design of ICs, in particular ASICs for a number of reasons. They allow the design of a circuit starting from a self-documented high-level specification, which can be fairly independent of the target technology or design style. Another advantage is that the design is more portable. When the description is at a high-level, we can more easily incorporate incremental changes. As a result, the design is likely to adapt more readily to design changes. Capturing and optimizing the design is a complex process, requiring integrated, computer-aided tools to do these tasks at different levels of representation. Last but not least are design turnaround times, which synthesis systems help to shorten.

Olympus is a vertically integrated synthesis system developed at Stanford University to support both the computer-aided and automatic design of general-purpose digital circuits, with specific attention to the requirements of ASIC designs. The system includes behavioral, structural, and logic synthesis tools and provides technology mapping and simulation. Since it is targeted for semicustom implementations, its output is in terms of gate netlists. Instead of supporting placement and routing tools, Olympus provides an interface to standard physical design tools. The Olympus system has the following important features:

- *HardwareC.* A synthesis-oriented hardware description language, called HardwareC,¹ is used in specifying hardware for synthesis. HardwareC supports hardware descriptions with both procedural and declarative semantics, as well as constraints on the hardware implementation.
- *Constraint-driven structural synthesis.* In this approach, the timing and resource constraints specified in the high-level description are used to produce a satisfactory implementation. The system supports the synthesis of partially bound hardware descriptions, in which certain operations are prebound to specific hardware modules, and it performs synthesis on the remaining operations. Olympus provides a flexible framework for the systematic exploration of the possible series/parallel hardware structures corresponding to area/performance trade-offs.
- *Combined high-level and logic synthesis.* Because logic synthesis techniques synthesize combined datapath and control, high-level synthesis does not have to depend on the availability of well-characterized resources in a library. Instead, each resource corresponds to a model that is synthesized and characterized by logic synthesis.

Olympus provides an environment in which detailed timing and area information at the logic level guides high-level synthesis tasks—where architectural trade-offs and scheduling are done. This methodology is particularly suitable for ASIC designs, which tend to rely on application-specific logic functions. Using logic synthesis for estimates improves the quality of the synthesized designs and avoids erroneous high-level decisions based on insufficient data or inappropriate assumptions.

- *Technology mapping.* The logic-level circuit description is mapped to a netlist that describes an interconnection of components from a user-specified library. The technology-mapping algorithm can target different design goals, from performance to area.
- *Support of both automatic and user-driven synthesis.* The synthesis flow can be fully automated, transforming a HardwareC description directly to a logic-level implementation. The system also supports user-driven synthesis. In user-driven synthesis, designers can intervene and drive the decisions, basing them on an evaluation of the possible design trade-offs.

At present, Olympus supports only a synchronous-logic implementation. The synthesis algorithms do not provide for the synthesis of pipelined structures or multiphase, synchronous logic. However, designers can use appropriate partial structures that describe circuit partitions and that correspond to pipe stages or phase stages to synthesize circuits with these features.

OLYMPUS AND OTHER SYSTEMS

How does Olympus relate to other work on synthesis tools? There are successful synthesis techniques that are based on high-level description languages such as VHDL.¹ Some fully integrated synthesis systems have been used for chip design, especially for digital signal processing. Notable examples are the Cathedral systems from the University of Leuven,² the Parsifal system developed at General Electric, and the Silcsyn system from Racal Redac. Other systems support only part of the synthesis flow, as do Architect's Workbench³ from Carnegie-Mellon, ADAM from the University of Southern California,⁴ and VSS from the University of California, Irvine—all of which specialize in high-level synthesis—and Octtools from the University of California at Berkeley that supports logic and physical design. More detailed information on these models is available elsewhere.^{5,6}

There are several differences between these systems and Olympus, however. First, Olympus is one of the few existing vertically integrated set of tools, that accepts hardware behavioral models as an input and that supports both high-level and logic synthesis. In the Yorktown Silicon Compiler,⁶ logic synthesis techniques synthesize combined datapath and control, but the system does not feed back information from the logic synthesis stage back to the high-level structural synthesis algorithms. Olympus does provide this feedback, which also distinguishes it from Architect's Workbench and ADAM. Even though ADAM uses estimates of logic-block complexity, neither it nor Architect's Workbench support logic synthesis and

technology mapping. Olympus differs from Cathedral and Parsifal by being not restricted to DSP applications. In addition, Olympus is constructed as an open system, in which tools communicate through machine-readable and human-readable interchange formats. This allows the designer to use the Olympus tools separately and to interface them easily with other design tools.

REFERENCES

1. S. Lis and D. Gajski, "VHDL Synthesis using Structured Modeling," *Proc. Design Automation Conf.*, 1989, pp. 606-609.
2. J. Rabaey et al., "Cathedral II: A Synthesis System for Multiprocessor DSP Systems," in *Silicon Compilation*, D. Gajski, ed., Addison Wesley, Reading, Mass., 1988, pp. 311-360.
3. D. Thomas et al., *Algorithmic and Register-Transfer Level Synthesis: The Systems Architect's Workbench*, Kluwer Academic Press, Boston, 1989.
4. A. Parker, J. Pizarro, and M. Mlinar, "MAHA: A Program for Data Path Synthesis," *Proc. Design Automation Conf.*, 1986, pp. 461-466.
5. *Design Systems for VLSI Circuits: Logic Synthesis and Silicon Compilation*, G. De Michell, A. Sangiovanni-Vincentelli, and P. Antognetti, eds., Martinus Nijhoff, Dordrecht, The Netherlands, 1987.
6. D. Gajski, *Silicon Compilation*, Addison Wesley, Reading, Mass., 1988.

A SYSTEM APPROACH TO SYNTHESIS

Figure 1 is a block diagram of Olympus, which comprises three major tasks:

1. *High-level synthesis* translates a behavioral description of hardware to a register-transfer level implementation.
2. *Logic-level synthesis* optimizes the area and performance of the logic representation.
3. *Technology mapping* maps the logic representation onto predefined library units.

High-level synthesis in Olympus is performed by two programs, Hercules and Hebe.² The two programs transform a behavioral description of hardware in HardwareC, through a series of translations and optimizations, to a synchronous logic implementation that satisfies the timing and resource constraints imposed on the design. Hercules performs the front-end parsing and behavioral optimizations. It generates an implementation-independent description of hardware behavior in a graph-based representation called the *Sequencing Intermediate Format*, or SIF.

A simulator, called Ariadne, simulates the SIF to provide feedback to the designer on the functional correctness of the input specification. Hebe systematically explores the design space, performs scheduling and resource binding, then maps the SIF into a logic-level implementation, which is described in the *Structural/Logic Intermediate Format*, or SLIF.

To provide accurate estimates on area and timing for Hebe to use during the scheduling phase and to ensure that the resulting implementation satisfies timing constraints, Olympus uniformly incorporates logic synthesis in guiding high-level design trade-offs. After high-level synthesis, Olympus passes the circuit representation in SLIF for both the control and data paths to logic synthesis for optimization.

A logic-synthesis-framework tool, called Mercury, supports some transformations for multiple-level logic optimization and logic-level simulation. It also provides an interface to other logic synthesis tools, such as MISII,³ and to other netlist formats.

Ceres performs technology mapping, in which the logic description is changed into a purely structural representation, which is defined in terms of cell instances of a predefined library for semicustom implementation.⁷

HARDWARE MODELING

The input to Olympus is a behavioral level description of digital circuits in a language called HardwareC.¹ HardwareC is a high-level hardware description language with a C-like syntax. The language has its own hardware semantics, and it differs from the C programming language in many respects. HardwareC supports both declarative semantics—such as the interconnection of modules—and procedural semantics—such as a set of operations ordered in time—in the modeling of hardware.

Four fundamental design abstractions correspond to block, process, procedure, and function models. At the topmost level, a design is described in terms of a block, which contains an interconnection of logic and instances of other blocks and processes. A process consists of a hierarchy of procedures and functions. It executes repeatedly and restarts itself upon completion. Since a process executes concurrently and independently with respect to the other processes in the system, designers can model coarse-grain parallelism at the functional level.

Olympus uniformly incorporates logic synthesis in guiding high-level design trade-offs.

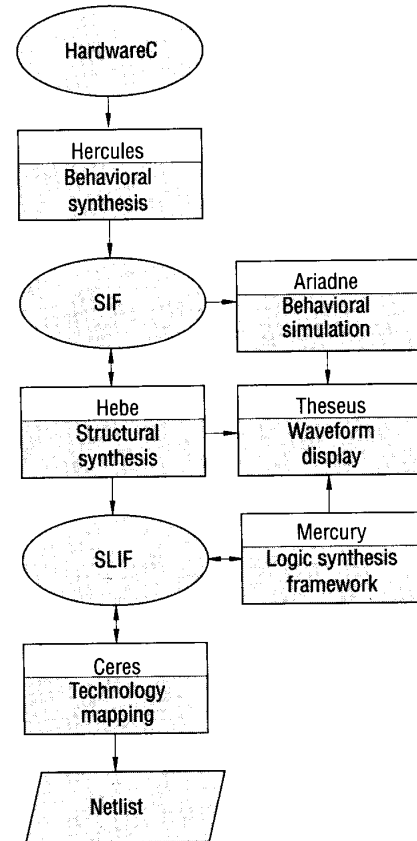


Figure 1. Block diagram of Olympus. SIF is Sequencing Intermediate Format, and SLIF is Structural/Logic Intermediate Format.

To support communication and synchronization among concurrent processes, HardwareC supports both parameter passing and message passing.

A procedure or function is an encapsulation of operations, and it may contain calls to other procedures and functions. The range of hardware implementations corresponding to a behavioral model is quite flexible, and the final implementation depends ultimately on the complexity of the target hardware. For example, we can describe a multichip system by associating each process to a chip. Alternatively, a single chip can implement multiple processes. HardwareC supports the usual iterative and branching constructs, including both fixed-iteration and data-dependent looping constructs. Data-dependent loops can be used to describe the detection of signal transitions, which is important in describing external interfaces. For example, the construct "while (data==0);" will wait until the rising transition of the signal "data."

Several features of HardwareC support hardware specification and synthesis, including interprocess communication, explicit instantiation of models, template models, parallelism, and constraint specification.

INTERPROCESS COMMUNICATION

To support communication and synchronization among concurrent processes, HardwareC supports both parameter passing and message passing. In parameter passing, we must have a shared medium such as a bus or memory that interconnects the hardware modules that implement processes. The handshaking protocols are described in the HardwareC description. In message passing, we use a synchronous send/receive mechanism to synchronize or transfer data. The corresponding hardware for communication, as well as its protocol, is automatically synthesized. Figure 2a illustrates parameter passing, while Figure 2b illustrates message passing.

EXPLICIT MODEL INSTANTIATION

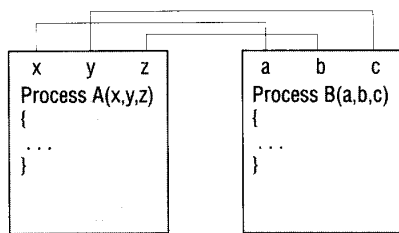
HardwareC supports hierarchical designs through the use of model calls. A call to a model can be either *unbound* or *bound*. An unbound call invokes a model without specifying the particular instance used to implement the call, while a bound call also identifies a specific instance of the model that will implement the call. Through explicit instantiation of model calls, HardwareC supports resource constraints and partial bindings of operations to resources. The designer can constrain the synthesis system to explore a subset of possible structures that corresponds to a behavioral model to satisfy a particular architectural requirement.

TEMPLATE MODELS

Templates are models that take one or more integer arguments. A single template can, for example, describe a family of adders of different sizes. Templates also support polymorphism by modeling several behaviors with a single description. Because of these qualities, templates are very useful in describing libraries of hardware operators at a high level.

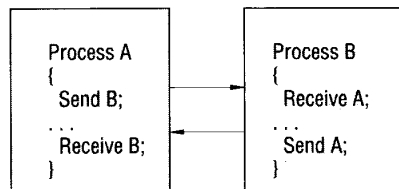
PARALLELISM

For procedural semantic models, designers can use HardwareC to adjust the degree of parallelism in a given design by grouping operations in one of three ways: sequential, data-parallel, or parallel. In a sequential grouping, operations execute sequentially. In a data-parallel grouping, all operations execute in parallel unless data dependency requires that they execute sequentially. In a parallel grouping, all operations execute in parallel unconditionally.



Parameter passing

(a)



Message passing

(b)

Figure 2. Parameter passing (a) versus message passing (b).

CONSTRAINT SPECIFICATION

Olympus supports timing constraints by tagging operations. This tagging imposes lower and upper bounds on the time separation between the tags. In specifying interfaces, designers will find the support of timing constraints useful because they can constrain the time between I/O operations. Resource constraints limit the number of operations that can be bound to resources in the final implementation.

AN EXAMPLE

Figure 3 gives an example of a HardwareC description that computes the greatest common divisor of two numbers. The model, called *gcd*, waits until the restart signal is low, samples the inputs, then computes the greatest common divisor of the inputs using Euclid's algorithm. To illustrate timing constraints, we specified a minimum timing constraint between the reading of the inputs.

ABSTRACTION LEVELS

Olympus operates on the hardware representation at two levels of abstraction, corresponding to SIF at the behavioral level and SLIF at the structural and logic level. Each representation has a corresponding machine- and human-readable interchange format to pass information among tools in a consistent way.

At the SIF level, or behavioral level, hardware is modeled as a set of tasks and dependencies among the tasks. Therefore, a natural representation is a directed acyclic graph, or DAG, in which the vertices represent the operations to be performed and the edges represent certain dependencies. These dependencies either are explicit in the hardware specification or represent dependencies from data-flow restrictions or hardware resource-sharing. The graph model is called a sequencing graph.

A vertex is enabled when all its predecessors have executed. Since a vertex may have multiple predecessors and successors, the model supports multiple threads of concurrent execution flow. Figure 4 illustrates this concurrent flow for the process *gcd*.

Vertices can be either simple or complex. Simple vertices include primitive computations in the language, such as arithmetic or logic expressions and message-passing commands. Complex vertices allow Olympus to perform groups of operations, inducing a hierarchical relationship among the graphs. These vertices include model calls, conditionals, and loops. A call vertex invokes the sequencing graph corresponding to the called model. A conditional vertex selects among a number of branches, each of which is modeled by a sequencing graph. A loop vertex iterates its body until the exit condition is satisfied; the body of the loop is also a sequencing graph.

The sequencing graph is acyclic because Olympus uses only structured control-flow constructs—no "goto." Loops are broken through the use of hierarchy. Designers can annotate additional structural information, such as resource sharing, in the SIF model.

At the SLIF level, we have a structural interconnection of logic elements. The model supports the specification of hierarchical netlists, which provides the structural information needed by the back-end tools interfacing to Olympus. In addition, SLIF supports the notion of un-mapped logic equations and synchronous delay elements. Unmapped logic equations are logic specifications that are not committed to a structure, and they are used by most optimization tools for logic synthesis.

Each representation has a corresponding machine- and human-readable interchange format that helps to pass information among tools in a consistent way.

```
process gcd (xin, yin, restart, result)
in port xin[8], yin[8], restart;
out port result[8];
[
  boolean x[8], y[8];
  tag a, b;
  constraint mintime from a to b = 3 cycles;

  /*set output to zero during computation*/
  write result = 0;

  /*wait for restart to go low*/
  while (restart)
  ;

  /*sample inputs in parallel*/
  < b:x = read(xin); a:y = read(yin); >

  /*Euclid's algorithm*/
  if ((x != 0) & (y != 0)) {
    repeat {
      while (x >= y)
        x = x - y;
      /*swap values*/
      < y = x; x = y; >
    } until (y == 0);
  } else
    x = 0;

  /*write result to output*/
  write result = x;
]
```

Figure 3. A HardwareC description of a process to find the greatest common divisor (process *gcd*) of two values.

All tools share a common user front-end, which is modeled after the Unix shell.

SYNTHESIS TOOLS

Olympus consists of an integrated set of tools that optimize at the behavioral, structural and logic levels of design abstraction. To provide designers with a consistent interface, all tools share a common user front-end, which is modeled after the Unix shell. It provides alias and history capabilities, as well as input/output redirection. We believe that complex hardware design requires a tight interaction between the human designer and the tools. Interactive programs free designers to explore design solutions, knowing that the system will constrain them to certain requirements when they need it. On the other hand, we think that eventually designers would like to be relieved from interacting with the program. For those times, we have created a batch mode in Olympus, which uses standard or specialized scripts.

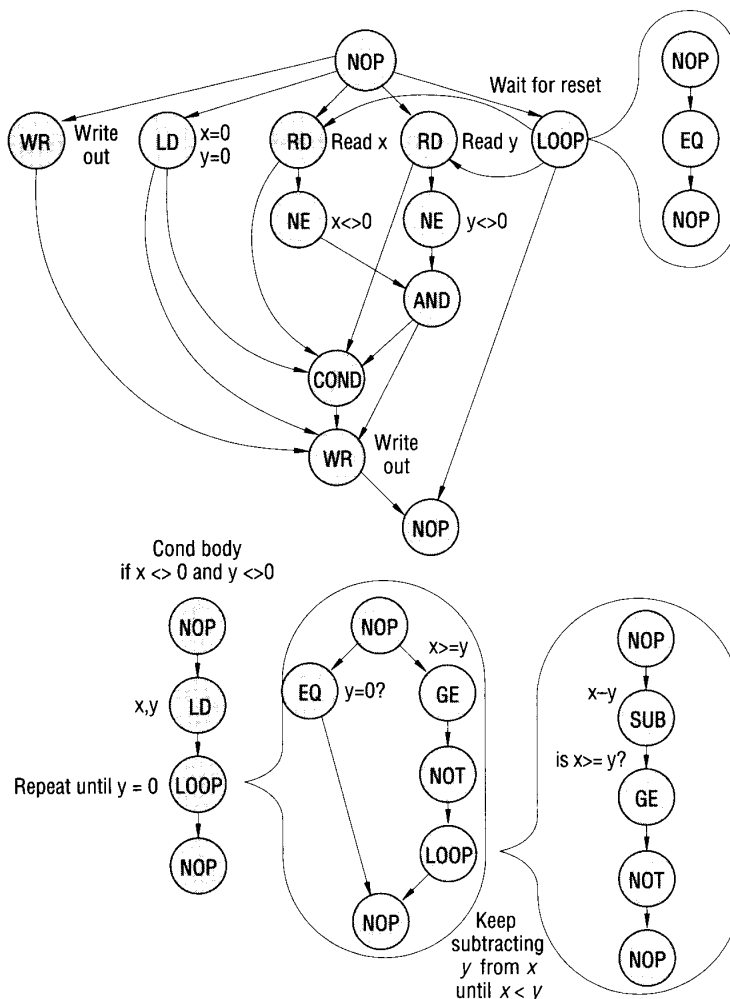


Figure 4. SIF model for process gcd.

The user interface provides three levels of interaction for the novice user, the advanced user, and the tool developer. Each level supports a set of commands and a help facility, which hides the commands of higher levels from the user. We think this feature helps users to get acquainted with the tools without being overwhelmed by the abundance of options. As users become more familiar with the tool, they can try a higher level of interaction.

HERCULES: BEHAVIORAL SYNTHESIS

The objective of behavioral synthesis is to identify the maximum parallelism that exists in the input description. The result indicates the fastest design that the system can produce, assuming that a dedicated hardware component implements each operation in the design implementation. Because of area and interconnection costs, this assumption may not be realistic in some cases. The corresponding performance is important as a limiting bound for a given behavior.

The first task in Hercules is to parse the HardwareC description (input) and translate it into an abstract syntax-tree representation. This representation provides the underlying model for semantic analysis and behavioral transformations. The transformations are either *user-driven* or *automatic*. User-driven transformations are optional and allow the designer to modify model calls and the hierarchy of the input description.

There are two types of user-driven transformations. In the first type—selective in-line expansion of model calls—a call to a model is replaced by the functionality of the called model. Once we expand these calls, we can apply optimization algorithms across the call hierarchy. The second type of user-driven transformation is selective operator-to-library mapping. Here, we map operators such as + or >= in the input description into calls to specific library template models. Although designers can synthesize an operator in a variety of different implementation styles, they are often constrained to elements of a particular library. With selective operator-to-library mapping, designers have the flexibility to select the specific implementation for the operators. If no mapping is given, then by default the operators are implemented as combinational logic expressions.

Automatic transformations optimize behavior by performing transformations similar to those found in optimizing compilers. The automatic transformations, which are carried out without human intervention, include

- *Unrolling For loops.* Fixed-iteration loops are unrolled to provide more opportunities for optimization.
- *Propagating constants and variables.* The reference to a variable is replaced by its last assigned value.
- *Resolving reference stacks.* Multiple and conditional assignments to variables are resolved by creating multiplexed values that can be referenced and assigned.²
- *Eliminating common subexpressions.* Redundant operations that produce the same results are removed.
- *Eliminating dead code.* Operations whose effects are not visible outside the model are removed.
- *Collapsing conditionals.* Conditionals with branches that have only combinational logic are collapsed to provide more opportunity to apply logic synthesis.
- *Analyzing dataflow.* Data and control dependencies among operations are identified.

The objective of behavioral synthesis is to identify the maximum parallelism that exists in the input description.

When the automatic transformations are complete, the behavior is optimized with respect to the data dependencies among operations. Hercules then performs combinational coalescing to group combinational logic operations into expression blocks. The expression blocks define the largest scope (without crossing register boundaries) in which we can apply logic synthesis. The blocks also identify the critical combinational logic delays through the data path. These expression blocks are passed directly to logic synthesis, which will try to optimize the design for area or performance. The results of this process are estimates on area and timing, which can then be used to refine the design during structural synthesis. Combinational coalescing is important, particularly for ASIC designs because such designs use logic expressions extensively in hardware specification.

HEBE: STRUCTURAL SYNTHESIS

The objective of Hebe is to explore design trade-offs to obtain a suitable structure that satisfies the user constraints on area and timing. The constraints can either be specified in the input HardwareC description, or the designer can enter them interactively. Hebe provides a flexible underlying representation of the design space that serves as the basis for both algorithmic exploration and user-driven synthesis. In particular, designers can either let Hebe explore the design trade-offs automatically, or they can manually guide the direction of synthesis by imposing constraints on the design.

Unlike microarchitectural synthesis systems, which use a predefined set of library elements as building blocks, each model in the input description is treated as a resource that can be allocated and shared among the calls to the models (either procedures or functions). Each call is implemented as an activation of a particular resource module. Hebe applies the synthesis steps bottom-up. It first synthesizes procedures that do not call other procedures using logic synthesis techniques. When this task is complete, Hebe propagates the delay and area information for the synthesized modules up the hierarchy to guide the synthesis of invoking procedures or processes.

Structural synthesis involves performing two tasks—*resource binding*, in which operations are assigned to hardware resources, and *scheduling*, in which operations are assigned to control states. An effective strategy is to perform resource binding before scheduling to provide scheduling with detailed interconnection delays, as in Caddy⁴ and BUD.⁵ We have extended this basic approach in Hebe to provide closer interaction and guidance to the designer.

Figure 5 is a block diagram of Hebe. The flow of structural synthesis in Hebe is to first bind operations to specific resources, then perform scheduling to find a schedule that satisfies the timing constraints. The process repeats for different binding alternatives. A resource pool is a set of hardware resources with an upper bound on the number of instances of each type of hardware resources that the user allows in the final implementation. A binding configuration matches operations (vertices of the sequencing graph) with specific resources in the resource pool. The design space is the entire set of binding configurations. The design space is also a complete characterization of the entire set of possible design trade-offs for a given allocation of resources.

Hebe's representation of design space offers two important advantages. First, it uniformly incorporates partial binding information. In some circuits, designers may wish to bind certain operations to resources to achieve high-level design goals. Hebe can use this information to limit

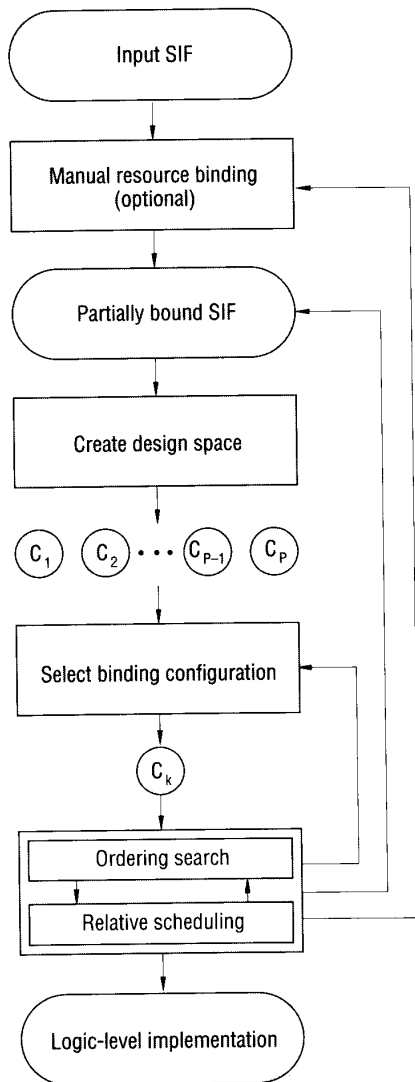


Figure 5. Block diagram of the Hebe structural synthesis system.

the design space so that the focus is on the remaining unbound operations. At the extreme, if all operations are bound, then the design space trivially reduces to a single point.

The second advantage of Hebe's representation of design space is that it supports optimum scheduling under timing constraints. Since Hebe decouples resource binding from scheduling, we can solve scheduling exactly, even under timing constraints, for each binding configuration. Heuristic scheduling is also provided.

The design space may be large because it grows exponentially with the number of shareable resources. However, ASIC designs often have a small enough number of shareable resources to make it practical to systematically explore all binding configurations. Furthermore, designers can use bounding techniques based on the evaluation of various cost criteria to prune the design space and speed the search for a suitable implementation.

Structural synthesis in Hebe involves the following tasks: select binding configuration, resolve resource conflict, and do relative scheduling.

Select binding configuration. Given a design space, Hebe supports both an exact and a heuristic search of the binding configurations. The selection is based on cost criteria that are used to evaluate the design space, including the area, interconnections, and delay of the final implementation. Deciding whether one alternative is more favorable than another depends on the relative importance of these criteria. That relative importance is, in turn, determined by the value of a weight associated with each criterion. Designers can experiment with different design goals by adjusting the values of these weights. The design goals indicate whether the emphasis of the final implementation is on area, performance, or some combination.

Resolve resource conflicts. Once a resource binding is selected by the designer or automatically by the system, Hebe orders the operations bound to the same resource component to resolve any possible resource conflicts. For example, if Hebe is synthesizing three parallel calls to a model that is bound to the same resource, it must order them serially to ensure that the calls do not simultaneously activate the resource. The goal of the ordering search is to find a set of dependencies among the operations that resolves the resource conflicts. Hebe uses a branch-and-bound approach to explore the ordering alternatives. Once it finds an ordering, the ordering is applied to the sequencing graph and scheduling is performed. If no schedule exists under timing constraints, then it tries another ordering. If no schedule exists for any valid ordering, it discards the binding configuration.

Do relative scheduling. When a sequencing graph has an ordering, it is free of resource conflicts. However, Hebe must now do scheduling to define the detailed temporal relationships among operations to satisfy the imposed timing constraints. There is a potential complication here. Some operations may have unbounded execution delays, which correspond to synchronization primitives and data-dependent loops. The unbounded delay operations invalidate the traditional scheduling formulation, in which operations are statically assigned to specific time slots. Hebe uses a novel technique called relative scheduling that uniformly supports operations with fixed and unbounded delays.⁶

An important consideration during scheduling is whether the timing constraints can be satisfied for any value of the unbounded delay operations. Hebe uses the concept of well-posed versus ill-posed timing constraints when it encounters unbounded delays.⁶ A timing constraint

The objective of Hebe is to explore design trade-offs to obtain a suitable structure that satisfies the user constraints on area and timing.

Ariadne is a graph and logic evaluator, as opposed to an event-driven or a compiled-code simulator.

is well-posed if satisfying it does not depend on any unbounded delays. Given an ordering of shareable operations to resolve resource conflicts, Hebe may need to serialize some operations to make the constraints well-posed. If it cannot impose a consistent serialization, or if it cannot satisfy the constraints, Hebe rejects the ordering as infeasible. Otherwise, it computes a minimum relative schedule using an iterative incremental scheduling algorithm. Both the time complexity of making constraints well-posed and the scheduling algorithm are polynomial. Consequently, Hebe can integrate relative scheduling effectively within the ordering search.

Once it has established a structure that satisfies both resource and timing constraints, Hebe generates a hardware implementation for the data-path operations, their interconnection, and the control circuitry to activate the resource components according to a given schedule. Hebe generates a combined logic-level description of both the data-path and the control portion in a SLIF representation. This representation allows the logic synthesis tools to optimize data-path operations and control simultaneously. The control structure is distributed, as opposed to a ROM-based control store, and it uses an interconnection of atomic control finite-state machines. With this distributed approach, Hebe can handle hierarchical control structures without penalizing the execution cycle. This approach also allows Hebe to implement a synchronization mechanism to cope with data-dependent delays. The generated structure is a hierarchical interconnection of logic circuits in SLIF.

ARIADNE: BEHAVIORAL SIMULATION

As we mentioned earlier, Olympus supports simulation at both the behavioral and logic levels. Ariadne is a behavioral-level simulator that evaluates the SIF hardware models generated by Hercules. Mercury is a logic-level simulator used to verify the SLIF models. Both simulators share the same graphic monitor, called Theseus.

Ariadne is used in conjunction with Hercules to verify the correctness of a behavioral hardware model in HardwareC. Since Hercules preserves the hardware behavior in transforming HardwareC models into SIF models and since its execution time is often negligible, the combination of the two programs provides the means for simulating the original hardware description.

Ariadne is a graph and logic evaluator, as opposed to an event-driven or a compiled-code simulator. The semantics of the SIF graph, based on the HardwareC model, assumes that the vertices in the graph are executed according to their dependencies and that their execution takes an integer number of cycles. Ariadne evaluates the SIF graph by traversing the SIF model. The SIF model already embeds the notion of partial task ordering because its edges represent the temporal and data dependencies. Ariadne visits a vertex after it has visited all the vertex's predecessors. It then simulates the operation that corresponds to the visited vertex.

Ariadne traverses the SIF graph hierarchy when it encounters a complex (conditional, loop, or call) vertex by transferring control to the linked SIF model. For conditional vertices, it evaluates conditional clauses on the fly to determine which branch to simulate. For iterative constructs, it repeats the simulation of the loop body until it evaluates the exit condition to be true.

Ariadne runs the simulation for a specified number of cycles or until the end of input test patterns, and it extracts output response vectors for display. Designers can specify a subset of the primary inputs and

outputs, as well as internal signals to monitor. They can also elect to display graphical output waveforms to facilitate the checking of simulation results in an X window using Theseus. Figure 6 shows the results of simulating the *gcd* example.

MERCURY: LOGIC-LEVEL INTERFACE

Mercury is a framework for synthesizing and simulating synchronous multiple-level logic. It supports the hierarchical description of networks that include combinational functions and registers, as described by SLIF. Mercury has three main purposes. The first is to optimize the logic representation. Optimizing transformations include constant propagation, function elimination, and local logic simplification. Another purpose of Mercury is to directly simulate synchronous circuits in SLIF. It also provides graphic output waveforms using Theseus. Mercury uses the same monitor and the same test patterns as Ariadne.

The last purpose of Mercury is to provide an interface to the input description of some commercial tools for simulation and logic synthesis. It supports the translation of SLIF to the NDL format used by the LSI Logic Design System for sea-of-gates implementations and to the ADL format used by the Actel Logic Design System for electrically programmable gate array implementations. It also provides an interactive interface to the University of California, Berkeley's MISII combinational synthesis program.³ Designers can enter commands for MISII directly from the Mercury shell. This interface capability opens an entry point to UC Berkeley's Octtools design system.

CERES: SEMICUSTOM TECHNOLOGY MAPPING

Ceres is a tool for technology mapping. It addresses the problem of conforming an arbitrary synchronous logic network to a network that uses parts from a given library. Ceres tries to achieve the optimal technology mapping by choosing a coverage of the network from a given set of library components, which minimizes the area or the cycle time. It uses algorithms, as MISII³ does, as opposed to applying rules, as most commercial tools do.

Mercury supports the hierarchical description of networks that include combinational functions and registers.

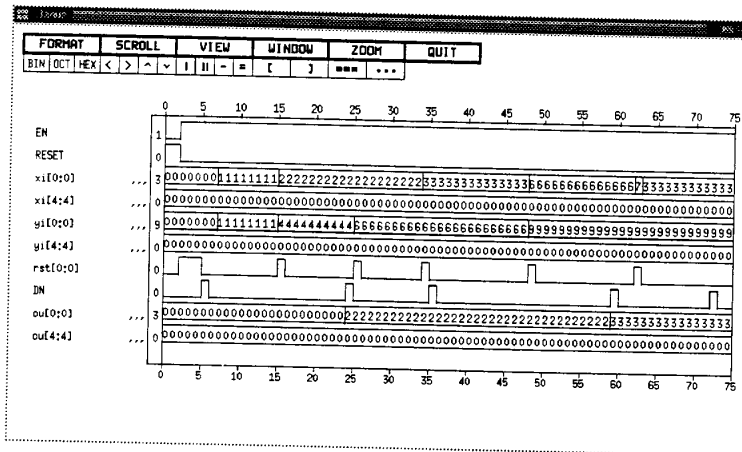


Figure 6. Simulator output waveform displayed by the Theseus graphics monitor.

Ceres addresses the problem of how to conform an arbitrary synchronous logic network to a network that uses parts from a given library.

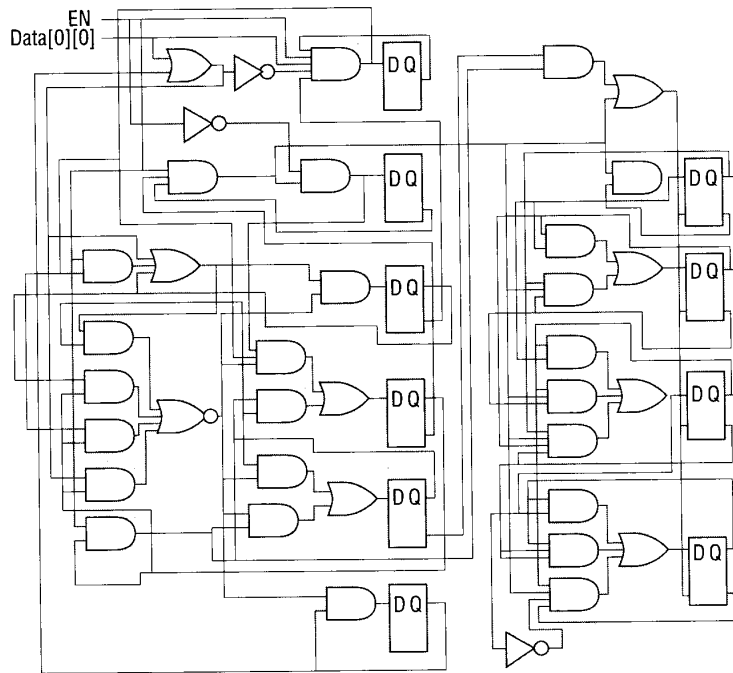


Figure 7. Partitioning example in Ceres.

Ceres differs from other technology-mapping tools because it exploits Boolean operations to match subsets of the Boolean network to library parts instead of using tree-matching techniques, as in MISII. Boolean operations allow Ceres to find matches regardless of the structural representation of the Boolean functions. In addition, it permits the use of "don't care" information, and therefore increases the number of possible matches with the library elements.⁷ Consequently, the quality of the final implementation is higher.

Since technology mapping is computationally intractable, Ceres uses a heuristic strategy, which consists of dividing the mapping problem into a set of four independent operations: partitioning, decomposition, covering, and matching.

Partitioning. Partitioning is the task of dividing the logic network into subnetworks with a single output. The output of each subnetwork is either a primary output, an input to a register, or a multiple fanout vertex in the original network. Figure 7 shows an example of such partitioning. Keutzer⁸ first proposed this method to circumvent the computationally intractable binate covering problem.

Decomposition. Decomposition consists of breaking all combinational subnetworks into an interconnection on two-input NAND/NOR/AND/OR gates, as Figure 8 shows. In this step, leaf DAGs

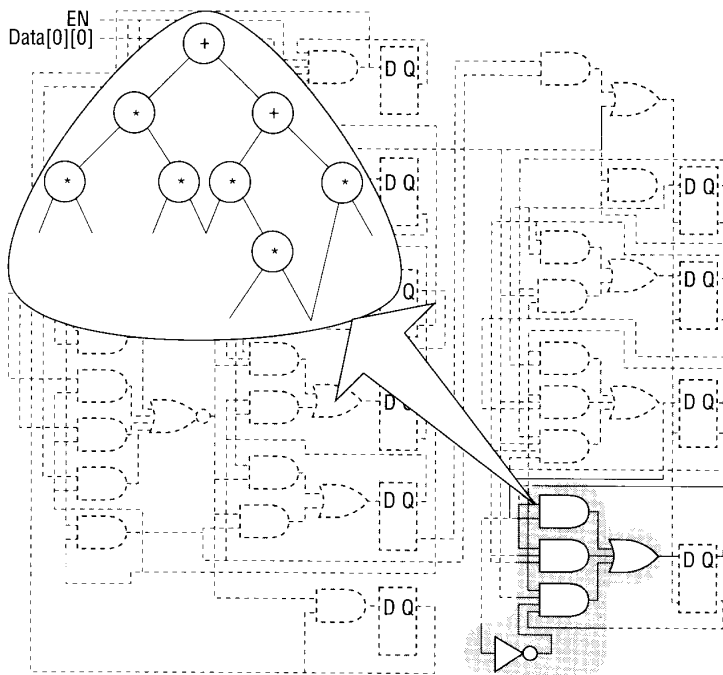


Figure 8. An example of decomposition in Ceres, a system for semicustom technology mapping.

Ceres divides the mapping problem into four independent operations, called partitioning, decomposition, covering, and matching.

represent the subnetworks with a very fine granularity of gates. In Ceres, we assume that two-input AND/OR and inverters are always available as part of the library of cells, thus ensuring that the entire network has a mapping in terms of these primitives.

Covering. Figure 9 shows an example of covering, the purpose of which is to find the optimum set of library cells to represent each subnetwork. Covering is based on a dynamic programming approach, in which the leaves of a subnetwork are processed first, then their immediate ancestors, and so on until the root vertex is reached. For each vertex being processed, Ceres considers all possible expressions for the vertex. That is, it considers the expansion of all subexpressions. Suppose, for example, we have the following decomposed subnetwork:

$$\begin{aligned} f &= bx_1 \\ b &= e + x_2 \\ x_2 &= \bar{c} + d \\ x_1 &= a + c \end{aligned}$$

where the root vertex f has six possible expressions, which are shown in Figure 10:

$$\begin{aligned} \epsilon_1 &= bx_1 \\ \epsilon_2 &= b(a + c) \\ \epsilon_3 &= (e + x_2)x_1 \\ \epsilon_4 &= (e + x_2)(a + c) \\ \epsilon_5 &= (e + \bar{c} + d)x_1 \\ \epsilon_6 &= (e + \bar{c} + d)(a + c) \end{aligned}$$

The purpose of covering is to find the optimum set of library cells to represent each subnetwork.

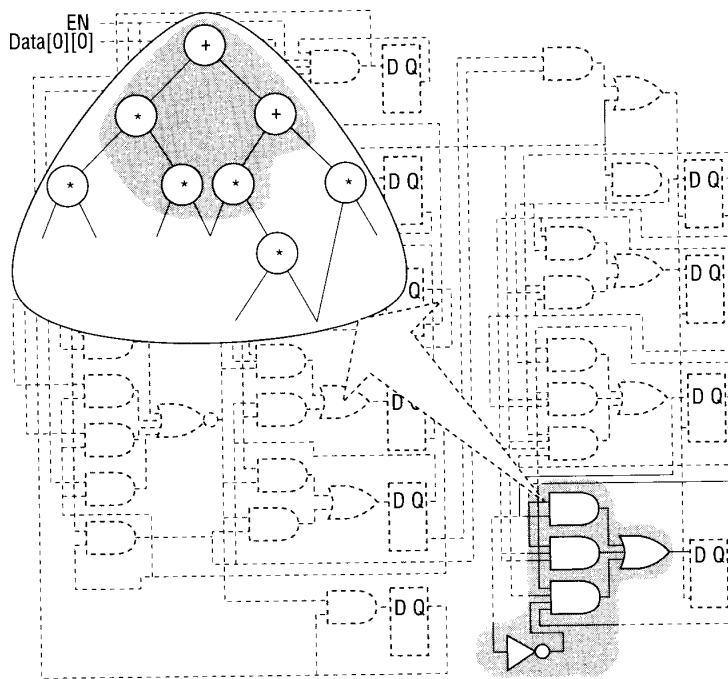


Figure 9. An example of covering in Ceres.

Ceres computes a cost for each of these expressions that has a corresponding element in the library. The cost consists of the cost of the library element plus the sum of the costs of the mapped subnetworks that correspond to the literals used in the expression. Ceres maps the vertex under consideration onto the library cell that corresponds to the expression that will lead to the best cost. The cost may represent the area or the propagation delay. Therefore, the best choice relates to the best implementation in terms of area or delay for the given block of the partition and the given decomposition.

Ceres uses matching in the covering step to check if a particular expression has a corresponding element in the library. Matching in Ceres is based on Boolean operations. It checks a logic function against other functions that represent the library cells. It does an equivalency check (modulo the "don't care" set) using Shannon decomposition recursively. It considers two functions to be equivalent when, at the end of the recursion, the two functions have the same value for each pattern not included in the "don't care" set.

Ceres considers the problem of phase assignment during matching. It considers two functions matched if there is an input phase assignment such that a function, or its complement, is equivalent to the other. By merging these two steps—matching and phase assignment—Ceres provides the possibility of a better overall implementation. The potential drawback of considering the two steps together is the computational cost.

Ceres uses techniques to reduce the search space to speed up Boolean matching. The use of symmetries, the main technique to reduce search

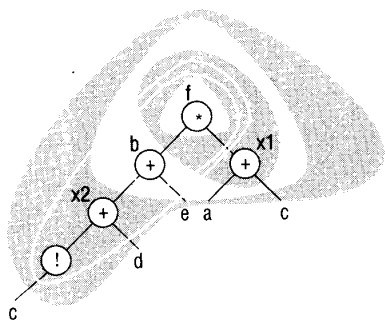


Figure 10. Subnetwork for a function, f .

space, acts as a filter to find possible candidates in the library. It also limits the number of input permutations considered during matching. Two or more variables are symmetrical in a logic function if they can be exchanged without modifying the original truth table. For example, in $f = a + bc + de$, variables b and c are symmetrical, as are variables d and e . Variables of a logic function can be grouped together because logic symmetry is an equivalence relation. Then groups are included into symmetry sets, and each symmetry set includes only groups with the same size. For example, the previous function f has two symmetry sets: $S_1 = \{(a)\}$ and $S_2 = \{(b,c),(d,e)\}$. A necessary condition for two functions to be logically equivalent, is that they have the same symmetry sets.

Ceres uses other techniques, such as unateness checking, to make Boolean matching efficient. Indeed, Boolean techniques have proved to be faster than structural mapping techniques, and they produce competitive results. We can see the advantage of Boolean techniques over other algorithmic technology-mapping approaches, when we consider libraries that contain several complex cells.⁷

DESIGN EXAMPLES

We have tested Olympus against the usual benchmark circuits for high-level and logic synthesis. In addition, others have used Olympus to design three application-specific digital circuits at Stanford University: BDCT, a bidimensional discrete cosine transform chip,⁹ DAIO, a digital/audio input/output chip,¹⁰ and MAMA, a discriminator chip for the multianode microchannel array detector used in astronomical applications.¹¹ The BDCT chip is a typical application for image processing. Its architecture is defined by a set of equations that can be solved row-by-row and column-by-column by two simpler monodimensional DCTs. It was therefore described as two concurrent processes that communicate via a shared memory array. An 8×8 BDCT architecture was described in HardwareC. Each process required approximately 500 lines of code and was synthesized and simulated at the logic level. The physical layout was then synthesized as a macro-cell using programs called Castor and Pollux.¹² The total chip image was about 9×9 sq. mm in 2μ CMOS technology.

The DAIO chip provides an interface (following the Audio Engineering Standard protocol) between a standard 16/32 microprocessor bus and audio devices, such as a compact disc or digital/audio tape player. It was modeled as a single process in HardwareC, which required about 500 lines of code. This representation was then compiled and simulated using Olympus. Finally, it was mapped onto a representation suitable for implementation in LSI Logic LMA9K sea-of-gates. The chip has about 6,000 equivalent gates. The DAIO architecture has also been implemented as two electrically programmable gate arrays.

The MAMA system consists of a photocathode for photon/electron conversion, a microchannel plate for electron multiplication, and the multianode array for event detection. The decoder chip is designed to discriminate the information generated by the multianode detector. The chip can detect two events occurring within 160 ns (for a 50-MHz clock). It is scheduled to fly in experiments in the Solar Orbiting Heliospheric Observatory in 1995 and on board the Hubble space telescope in 1996.

The MAMA decoder chip was modeled in HardwareC, which required about 2,300 lines of code. It was then synthesized and simulated, and the description was mapped onto one LSI Logic LCA10K sea-of-gate chip. The MAMA decoder chip required approximately 24,000 equivalent gates.

We have used the Olympus synthesis system to design three application-specific digital circuits.

To use Olympus efficiently, designers must think in terms of behavioral abstractions, which means they must relinquish full control of the final design implementation.


Olympus is prototype of a vertically integrated synthesis system. It is operational and has been used to design three research prototype chips from behavioral specifications in HardwareC. We conceived Olympus to serve two purposes: to provide a workbench to experiment with algorithms for computer-aided synthesis, and to provide a vehicle for fast-turnaround chip design.

As a research project, Olympus is under continuous development. We are incorporating and testing new ideas and algorithms. The modularity of the system allows us to replace tools as they become obsolete and to experiment with alternative synthesis paths.

Our experience with chip design has been fruitful because these design have prompted new ideas for improving the tools. Designers using Olympus spent most of their time in chip modeling. In comparison, the execution time of the tools was negligible.

As is true of all research projects, however, Olympus has some drawbacks. One is that to use HardwareC and Olympus efficiently, designers must think in terms of behavioral abstractions. Such an abstract approach mandates that designers relinquish full control of the final implementation. Partially bounded structures and the use of block models in HardwareC alleviate this problem somewhat. Moreover, designers often need to verify that a HardwareC model fits their needs. Although they can validate the design by simulating the model with Ariadne, if designers are not sure whether what they have modeled is what they want, simulation is not much help. We believe that tools for reasoning about behavioral hardware models would be extremely useful in reducing the modeling time, which we see as the bottleneck in speeding up the overall design process.

Future work will address experimenting with other algorithms for structural and synchronous logic synthesis, as well as exploring the relations between the two domains. We are investigating partitioning techniques at the structural level to cope with the generation of parallel and/or pipelined computing structures. We are also considering logic synthesis and technology mapping for synchronous logic circuits.

As we mentioned earlier, Olympus is fully operational. Those interested in the availability of Olympus can contact Mrs. Lilian Betters, Center for Integrated Systems, Stanford University, Stanford CA 94305, or preferably use the electronic address olympus@chronos.Stanford.Edu. 

ACKNOWLEDGMENTS

We acknowledge the contributions to Olympus by Michiel Ligthart, Roger Yip, and Jerry Yang who participated in the development of the tools. We thank Larry Augustin, Rajesh Gupta, Rindert Schutten, and Polly Siegel for their invaluable comments and criticism. We also thank Vittorio Rampa, Michiel Ligthart, and David Kasle who, as first users of Olympus, provided feedback about the design system requirements.

This research has been supported by DEC and AT&T, jointly with NSF, under a Presidential Young Investigator Award, by IBM under a Resident Study Fellowship and by Philips under a CIS Fellowship.

REFERENCES

1. D. Ku and G. De Micheli, *HardwareC: A Language for Hardware Design*, tech. rpt. CSL-TR-90-419, Computer System Lab., Stanford Univ., Aug. 1990 (Version 2.0).
2. D. Ku and G. De Micheli, "High Level Synthesis and Optimization Strategies in Hercules and Hebe," *Proc. European ASIC Conf.*, 1990, pp. 124-129.
3. R. Brayton et al., "MIS: A Multiple-Level Logic Optimization System," *IEEE Trans. Computer-Aided Design*, Vol. CAD-6, No. 6, Nov. 1987, pp. 1062-1081.

4. R. Camposano and W. Rosenstiel, "Synthesizing Circuits from Behavioral Descriptions," *IEEE Trans. Computer-Aided Design*, Vol. CAD-8, No. 1, Feb. 1989, pp. 171-180.
5. M. McFarland, "Using Bottom-Up Design Techniques in the Synthesis of Digital Hardware from Abstract Behavioral Descriptions," *Proc. Design Automation Conf.*, 1986, pp. 474-480.
6. D. Ku and G. De Micheli, *Relative Scheduling Under Timing Constraints*, tech rpt. CSL-TR-89-401, Stanford University, Stanford, Calif., Nov. 1989; also in *Proc. Design Automation Conf.*, 1990, pp. 59-64.
7. F. Mailhot and G. De Micheli, "Technology Mapping Using Boolean Matching and Don't Care Sets," *Proc. European Design Automation Conf.*, Mar. 1990, pp. 212-216.
8. K. Keutzer, "Technology Binding and Local Optimization by DAG Matching," *Proc. Design Automation Conf.*, 1987, pp. 341-347.
9. V. Rampa and G. De Micheli, "Computer-Aided Synthesis of a Discrete Cosine Transform Chip," *Proc. Int'l Symp. Circuits and Systems*, 1989, pp. 220-225.
10. M. Lighthart et al. "Design of a Digital Audio Input Output Chip," *Proc. Custom Integrated Circuit Conf.*, 1989, pp. 15.1.1-15.1.6.
11. D. Kastle, "High Resolution Decoding Techniques and Single-Chip Decoders for Multi-Anode Microchannel Arrays," *Proc. Int'l Soc. for Optical Eng.*, Vol. 1158, Aug. 1989, pp. 311-318.
12. F. Mailhot and G. De Micheli, "Automatic Layout and Optimization of Static CMOS Cells," *Proc. Int'l Conf. Computer Design*, 1988, pp. 180-185.

Giovanni De Micheli is guest editor of the theme articles on high-level synthesis. His photo and biography appear on p. 7.



Frédéric Mailhot is a PhD candidate in electrical engineering at Stanford University, where he has been involved in the design and development of Olympus for the last three years. His research interests include logic synthesis, simulation, and physical design tools. He holds a BS in physics engineering from École Polytechnique de Montréal, a DEA in microelectronics from Université de Grenoble, and an MSEE from Université de Sherbrooke. His work has been supported by a NSERC scholarship from the Canadian government and a Fonds FCAR scholarship from Quebec. He is a member of the IEEE.



David Ku is pursuing a PhD in electrical engineering from Stanford University. For the last three years, he has been involved in the development of high-level aspects of Olympus. His research interests include automata theory, sequential logic synthesis and modeling, control, and high-level synthesis. He holds a BS in computer science and a BS in electrical engineering from the University of Utah and an MS in electrical engineering from Stanford. He was awarded an AT&T fellowship in 1986 and a Stanford Center for Integrated Systems fellowship in 1989 and 1990.



Thomas Truong is a staff engineer at IBM Almaden Research Center and is pursuing a PhD in electrical engineering through an IBM Resident Study Fellowship. His research interests are in high-level synthesis and optimization. He holds a BSEECs with high honors from the University of California, Berkeley, and an MSEE from Stanford University. He is a member of the IEEE, Tau Beta Pi, and Eta Kappa Nu.

Direct questions or comments on this article to G. De Micheli, Ctr. for Integrated Systems, Stanford Univ., Stanford, CA 94305.