High-level synthesis in an industrial environment

by R. A. Bergamaschi
R. A. O'Connor
L. Stok
M. Z. Moricz
S. Prakash
A. Kuehlmann
D. S. Rao

The use of modern hardware-description languages in the chip design process has allowed designs to be modeled at higher abstraction levels. More powerful modeling styles, such as register-transfer and behavioral level specifications, have spurred the development of high-level synthesis techniques in both industry and academia. However, despite the many research efforts, the technology is not yet in widespread use in industry. This paper presents the IBM High-Level Synthesis System (HIS), which is the first such system to be used in production in IBM. HIS synthesizes gate-level networks from VHDL models at various levels of abstraction. The main algorithms, modeling capabilities, and methodology considerations in the HIS system are presented. Results show that HIS is capable of producing implementations comparable to or better than those of the existing methodology, while shortening the design time significantly. The HIS system is currently in production use and evaluation in several IBM sites for processors and peripheral chip designs, as well as being an external commercial product.

1. Introduction

Computer-aided design (CAD) tools are in use today in almost all aspects of digital system design. Among the most common are tools for physical design, simulation, and synthesis. While tools for physical design and simulation have been in use in industry for a long time, synthesis tools have not, except for a few proprietary systems.

Synthesis tools have usually followed simulation and modeling tools in terms of abstraction level. The first hardware description languages used in industry were basically transistor-level and gate-level netlists. These were followed by languages capable of modeling designs at the logic level, register-transfer (RT) level, and behavioral level. As expected, the evolution in simulation tools followed a similar path. Circuit-level simulation was followed by logic simulation, and more recently by functional simulation. As the hardware description languages evolved, simulators were developed for them.

Clearly, this evolution was not isolated from other design aspects, in particular, silicon technology. As fabrication technology matured, it became feasible to design and characterize larger and larger blocks. The development of gate arrays and standard cells contributed significantly to this evolution.

0018-8646/95/$3.00 © 1995 IBM

Increasing levels of integration created a design problem. The complexity of the logic in a chip quickly made design intractable at the transistor level or even at the gate level. Design teams had to become larger to maintain productivity, with the result that each designer was able to understand only a very small part of the overall system. This caused a productivity bottleneck and created performance and verification problems.

A paradigm shift was needed; larger integration scales could not be modeled by low levels of abstraction. Higher levels of abstraction were needed to increase designers' productivity and to describe the design concisely, allowing fewer designers to understand the whole system and the implications of design changes in its performance and functionality.

More powerful languages and simulators allowed more complex designs to be modeled, which required more powerful synthesis systems. Synthesis tools evolved according to the existing modeling capabilities. Tools for minimizing two-level Boolean expressions were among the first to be developed. These were followed by tools for synthesis of combinational networks and PLAs. Current tools can synthesize complex RT-level networks automatically from language descriptions.

Despite many years of research in academia and industry on all aspects of synthesis, only in recent years has the technology reached the marketplace. The first company to recognize the importance of and make use of logic synthesis in large-scale designs was IBM, in the early '80s. Systems such as ALERT [1], MINI [2], Espresso [3], YLE [4], and LSS [5] demonstrated the capabilities of logic synthesis and paved the way for the tools available on the market today. As early as 1984, the LSS system was used to design 90 percent of the chips in a large-processor mainframe. Systems for logic and RT-level synthesis of application-specific ICs (ASICs) are currently available from CAD vendor companies as well as from internal industry groups.

One of the reasons for the delay in logic synthesis tools reaching the marketplace was the lack of an integrated methodology involving modeling, simulation, synthesis, and verification, which allowed the designer to speed up the design process, maintain a certain level of control over the final implementation, and verify that the final circuit correctly implemented the simulated functionality. Modern logic synthesis systems achieve many of these goals by combining efficient algorithms, control mechanisms for the designer to influence the synthesis outcome, and verification mechanisms (using formal methods and/or simulation techniques) to check the correctness of the synthesis process.

The trend in modeling, simulation, and synthesis is to push the abstraction level even further toward the behavioral and system level. Behavioral modeling is becoming more acceptable to designers as they are faced with increasingly complex circuits. The acceptance of high-level modeling and functional simulation has spurred the development of high-level synthesis in industrial environments.

Despite the many research efforts, high-level synthesis systems have so far failed to make a successful transition to the marketplace, except for systems for special applications, such as digital-signal processing (e.g., [6]). The main reason has been the lack of an integrated high-level design methodology, added to the fact that designs produced by most high-level synthesis systems are not yet competitive.

Many high-level synthesis systems have been developed in universities. Most notably, research at CMU [7], Stanford [8], USC [9], and Irvine [10], among others, has helped to create an algorithmic basis for others to build upon. Among the efforts in industrial environments, work at IMEC [6], BNR [11], IBM [12], AT&T [13], and GM [14] has contributed significantly to bringing high-level synthesis closer to production use. The work being done at GM [14, 15] corroborates the point that a high-level synthesis tool must be smoothly integrated with the rest of the design methodology.

This paper presents the HIS system, which is the first high-level design system to be used in IBM and among the first in industry. HIS was developed at IBM Research and EDA–IBM Microelectronics; it is currently in production use and evaluation at several sites. The main algorithms, modeling capabilities, and methodology considerations in the HIS system are presented.

The main goal throughout this project was to make high-level synthesis sufficiently practical and efficient for a production environment. Early experiences in using HIS pointed out various bottlenecks ranging from methodology to algorithmic problems. These experiences led to the development of a design methodology and algorithms for high-level synthesis which are proving to be efficient for production use.

This paper is structured as follows. Section 2 presents an overview of the high-level design methodology and discusses the specification and modeling aspects in the HIS system. Section 3 explains the main synthesis algorithms in HIS, including the data model, data-flow analysis, scheduling, allocation, and optimizations. Section 4 gives details on the current production use of HIS as well as future applications. Section 5 presents the conclusions.

2. A methodology for high-level synthesis

• *Overview*

The high-level synthesis process encompasses various design aspects ranging from modeling and simulation to logic synthesis, layout, and test. As a result, in order

for high-level synthesis to produce good results and be adopted by designers, it must be tightly integrated with other aspects in the methodology.

In the previous production methodology, designs were first specified using hardware description languages, at the RT or gate level; then submitted to logic synthesis (e.g., BooleDozer™) for logic minimization and technology mapping, and finally to physical design tools for layout. HIS is being introduced on top of this existing methodology. The input description allows behavioral specifications in addition to all lower levels. The output of high-level synthesis is a gate-level description which can be processed by the existing tools. The goal was to provide designers with higher-level modeling and synthesis capabilities, integrated smoothly with the rest of the design process, to deliver results of comparable or better quality, and to shorten the whole design cycle.

Figure 1 shows an overview of the high-level synthesis methodology. The design process starts with a language specification written by the user, which is the input to simulation. The goal of simulation depends on the level of the description. If the description is fully behavioral, the goal is to check functionality, not necessarily with a defined cycle-by-cycle behavior. If the description is at RT or gate level, simulation is used for checking functionality as well as cycle behavior.

HIS accepts descriptions written in VHDL or Verilog HDL. The examples and the discussion in this paper use VHDL, but all of the techniques described are equally applicable to Verilog.

The specification is then input to high-level synthesis, which maps it to a technology-independent gate-level netlist. At this point the designer can, within the high-level synthesis tool, estimate the delay and area of the design using a target technology. If the estimations represent an unacceptable design point, the designer can backtrack and explore the design space either by changing the input description or by running high-level synthesis with different constraints. Once a satisfactory design point is found (that is, the estimations are close to the requirements), the design flow can proceed, and the netlist can be submitted to logic synthesis for optimization and technology mapping.

Although the high-level synthesis process works on technology-independent representations, it can query information from a technology library in order to make better design decisions. For example, to determine whether two operators should be shared with multiplexed inputs, it is important to compute the area savings resulting from one less operator and possibly two extra multiplexors. The area for various hardware primitives can be obtained from the technology library.

The capability of having area and delay estimations inside HIS is important for reducing total design time. In

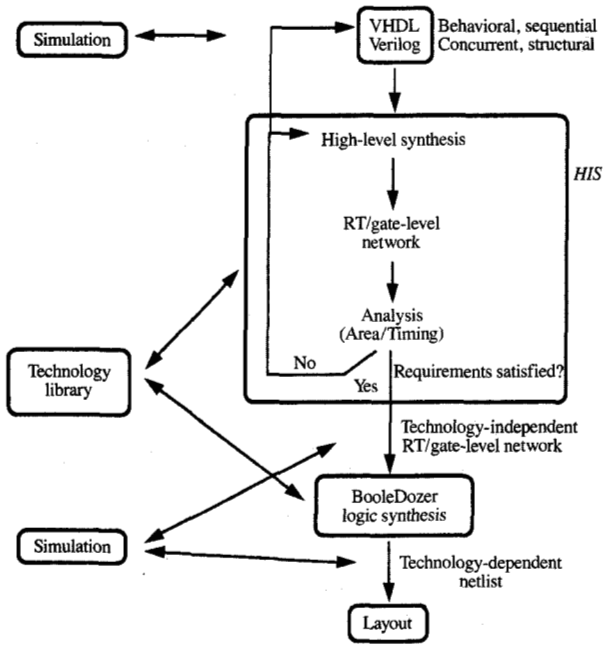


Figure 1

Overview of high-level synthesis methodology.

traditional methodologies (based only on logic synthesis), the design loop is closed after logic synthesis (or even after layout and back-annotation). At that point, the designer checks whether the requirements are being satisfied. If not, the alternatives are either to rerun logic synthesis with different constraints or to change the input description and rerun synthesis. The drawback with this methodology is that potential design problems are discovered only after a significant amount of time has been invested in logic synthesis, which can be very time-consuming for large designs.

Using the HIS system, the designer can obtain area and delay estimations much earlier in the design process, and can explore the design space by either rerunning HIS with different constraints or modifying the VHDL input. Refining the design at the high level speeds up the design cycle considerably, as the execution time for HIS is in general one to two orders of magnitude faster than logic synthesis techniques.

- *Specification and modeling*

The specification and modeling of a design play a central role in the efficient use of high-level synthesis. On one hand, designers want to have some control over

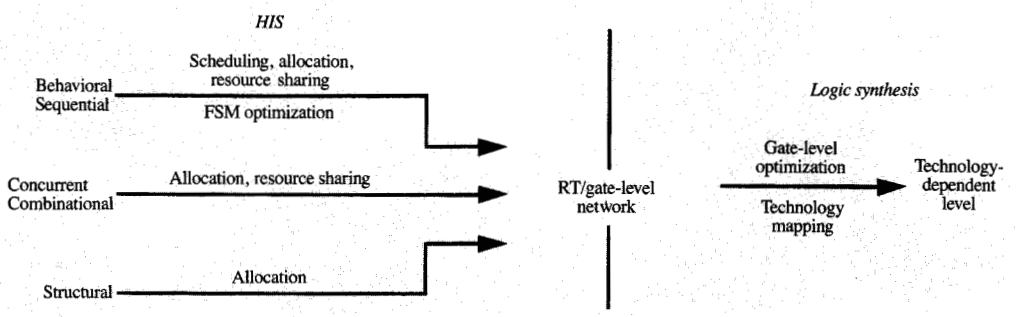


Figure 2

Abstraction levels and synthesis algorithms in HIS.

the implementation to guarantee a certain level of performance, but at the same time they want to use more sequential and behavioral modeling in order to benefit from lower complexity and faster simulation. These somewhat opposing requirements must be reconciled by the high-level synthesis system.

HIS is able to satisfy these requirements by supporting VHDL descriptions in behavioral, sequential, concurrent, and structural levels, and by providing the user with a rich set of controls over various implementation characteristics.

An important feature of VHDL is the ability to describe designs at different levels of abstraction. VHDL constructs such as processes, blocks, concurrent statements, and component instantiations can be used for modeling designs at the behavioral, sequential, concurrent, and structural levels. The designer can combine different abstraction levels in the same design. Algorithmic parts are more easily described at the behavioral or sequential level, while random logic and gate-level networks can be directly described at the RT or concurrent level.

The types of design styles supported by HIS range from fully behavioral descriptions to finite-state machines to combinational logic. HIS automatically detects the level of the description and applies the appropriate synthesis algorithms. **Figure 2** shows the basic synthesis algorithms applicable to each abstraction level.

In HIS, the user can influence synthesis and control several implementation details by using special-purpose VHDL attributes. Attributes are the mechanism in VHDL for attaching extra information to any VHDL element. For example, attributes can be attached to a VHDL design unit, function, or component, or even a simple statement. These special-purpose attributes allow the user to specify, among other things,

- The number and type of functional units that each VHDL process can use.
- Directives for resource sharing.
- Directives for input/output (ports) mapping and timing.
- The technology cell to which a certain VHDL component should be mapped.
- The technology cell to which a given VHDL expression should be mapped.
- Whether or not scan latches should be used and a scan chain generated.
- Timing constraints to be passed to logic synthesis.
- Any other information that should be passed to back-end layout tools.

Special-purpose synthesis attributes allow the user to exert control over the implementation while taking advantage of the high-level synthesis capabilities.

3. Synthesis algorithms in HIS

The synthesis task performed by HIS consists of generating a network representation which implements the functionality specified in the input description. This network is fully defined in terms of registers, logic gates, arithmetic operators, multiplexors, buses, etc.

HIS accepts VHDL descriptions at various abstraction levels. According to the level of the description, certain algorithms are applied. Behavioral or sequential descriptions require scheduling and allocation. Concurrent or structural descriptions require only allocation. The exact definition of these tasks is given in the following sections. **Figure 3** shows the main synthesis steps in HIS.

- *Data model*

The data model used in HIS consists of five graphs which represent the design at various stages during synthesis:

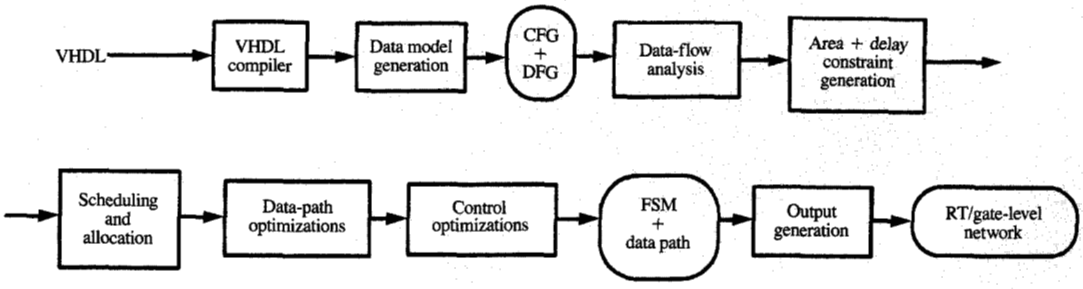


Figure 3

HIS main synthesis steps.

- Control-flow graph (CFG).
- Data-flow graph (DFG).
- Control-automaton graph.
- Data-path graph.
- Network graph.

These graphs are connected by links which keep the correspondence between nodes in the graphs, lines in the VHDL description, and the final network, as shown in **Figure 4**.

The input description goes through a compilation step which performs syntactical and semantical checks, followed by a translation step which generates the control and data-flow graphs. High-level synthesis creates the control-automaton graph and the data-path graph, which represent the finite-state machine and the data path of the implementation, respectively. The last synthesis step is the generation of the network graph, which represents the final implementation and is also the input to logic synthesis.

The control and data-flow graphs used in HIS are similar to those presented in [16] for sequential VHDL, plus extensions for handling concurrent and structural VHDL as well as other languages such as Verilog HDL.

The control-flow graph is a directed graph which specifies the sequence of operations in the input description. It is defined as $CFG = (O, P)$, where O is the set of nodes (representing the operations), and P is the set of edges. An edge $p = (o_1, o_2, c)$ implies that operation o_2 is executed after operation o_1, if condition c is true. Conditional operations, such as IF, CASE, etc., are modeled as conditional edges. The conditions are represented as binary-decision diagrams (BDDs). Two nonconditional operations in series are connected by an edge with condition c = true.

The data-flow graph is a directed graph which specifies the data dependencies among operations and operands in the input description. It is defined as $DFG = (O \cup V, D)$, where O is the set of operation nodes, V is the set of operand nodes (variables, signals, inputs, outputs, constants), and D is the set of edges. Data-flow edges $d = (o_1, v_1)$ or $d = (v_1, o_1)$ imply that operation o_1 assigns a value to operand v_1, or that operand v_1 is an input to operation o_1, respectively.

Figure 5 shows a VHDL example which is used throughout the paper; **Figure 6** gives the corresponding CFG and DFG.

- *Data-flow analysis*

Data-flow analysis is a technique commonly used in programming language compilers for determining the *lifetimes* of values [17]. High-level synthesis systems use data-flow analysis for two purposes: 1) to *unfold* variables in the data-flow graph [7, 18]; and 2) to derive the lifetimes of registers during resource sharing [19]. Most systems (e.g., [7]) apply data-flow analysis on basic blocks only. HIS applies data-flow analysis *globally*, through the whole graph, taking into account conditional operations and loops.

Lifetime analysis computes the *definition-use* chain for all variable assignments in the CFG and DFG. The *definition-use* chain of a given assignment to a variable tells exactly when the value (being assigned) is created and the last time it is used, thus defining its *lifetime*.

HIS uses data-flow analysis extensively in all synthesis steps. Data-flow analysis is performed on the CFG and DFG. The lifetimes of values are computed and attached to each node in the CFG, for future use during scheduling and allocation. The data structure for storing this information is a vector of bits (called a *lifetime vector*), where each bit position can assume the value 0 or 1. Each

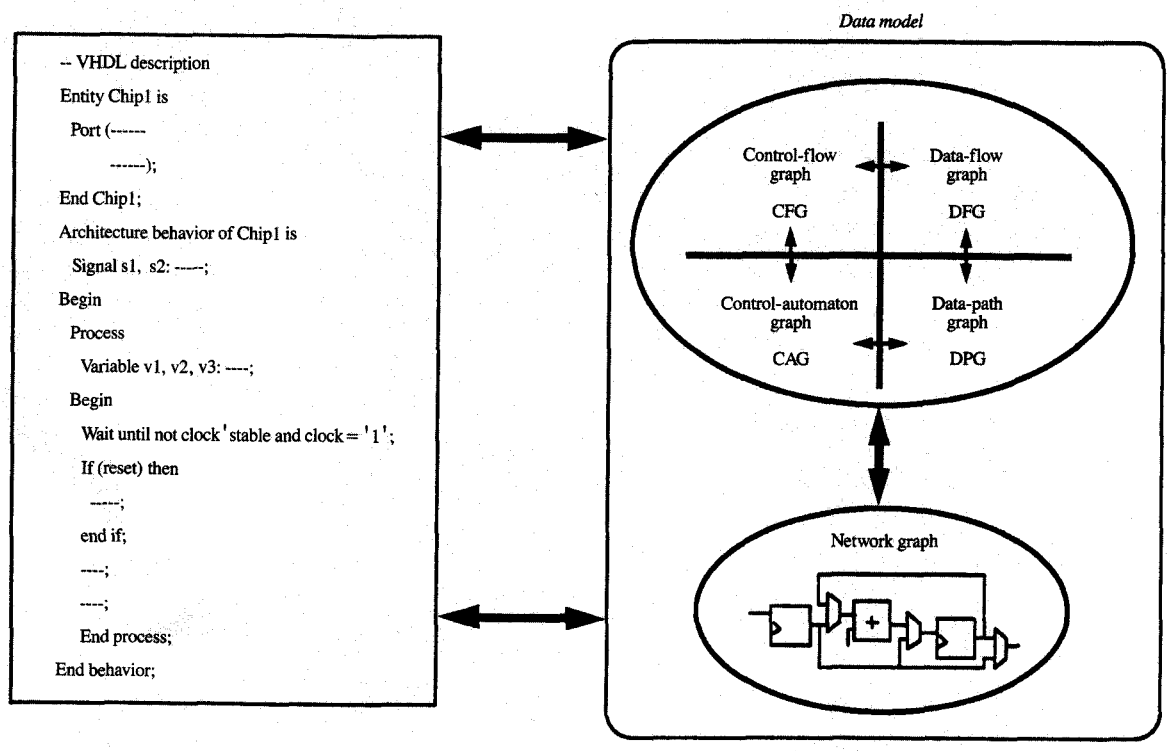


Figure 4

HIS data model.

assignment to a variable (assignment edge) is given a position number in this vector. Therefore, if n is the total number of assignment edges to variables in the DFG, the lifetime vectors will contain n bits.

The information in the lifetime vectors can be summarized as follows:

> Given a control-flow node o_i with a lifetime vector *lifevec*(o_i), and given an assignment edge in the DFG, d_j, with position p_j, then
>
> 1. If *lifevec*(o_i)[p_j] = 1, the value being assigned through edge d_j is *alive* at node o_i.
> 2. If *lifevec*(o_i)[p_j] = 0, the value being assigned through edge d_j is *not alive* at node o_i.

Hence, by following the value of a given position (for a given assignment edge) in the lifetime vectors of successive operations in the CFG, one can determine exactly which operation created the value and which operation is the last one to use it.

In order to illustrate the use of global data-flow analysis, consider Figure 6. The lifetime vectors are displayed beside each node in the CFG. For example, to determine the values of variable B which can be used as an input to operation/node 6, in Figure 6, one must follow these steps:

1. Get the lifetime vector from node 6, in the CFG: *lifevec*(6) = [01101011].
2. Get the positions of all assignment edges to B from the DFG: *positions*($d1$, $d2$, $d4$, $d6$) = 1, 2, 4, 6.
3. Check entries in the lifetime vector in positions 1, 2, 4, 6: *lifevec*(6)[1] = 0; *lifevec*(6)[2] = 1; *lifevec*(6)[4] = 0; *lifevec*(6)[6] = 0.

Only edge d_2 has a 1 entry in the lifetime vector; hence, the only value of B valid (or alive) at operation 6 is the value being produced by operation 4 (the predecessor of edge d_2). Similarly, there are four values of B alive at operation 19, produced by operations 2, 4, 7, and 9. This type of analysis is used during synthesis in order to

```
ENTITY Example IS
   PORT (
      CLOCK  : IN BIT;
      MODE1  : IN BIT_VECTOR(0 TO 1);
      MODE2  : IN BOOLEAN;
      IN1, IN2 : IN BIT_VECTOR(0 TO 1);
      OUT1, OUT2, OUT3, OUT4: OUT BIT_VECTOR(0 TO 1));
END Example;

ARCHITECTURE Behavior OF Example IS
BEGIN
P1 : PROCESS
   VARIABLE A, B : BIT_VECTOR(0 TO 1);

   BEGIN
      WAIT UNTIL (not CLOCK'stable) and (CLOCK='1');          — [1]

      B := b"00";                                             — [2]

      CASE  (mode1) IS                                        — [3]
      WHEN b"00" ⇒ B := IN1 + IN2;                            — [4]
                   WAIT UNTIL (not CLOCK'stable) and (CLOCK='1'); — [5]
                   A := A + B;                                — [6]
      WHEN b"01" ⇒ B := "11";                                 — [7]
      WHEN b"10" ⇒ A := IN2;                                  — [8]
      WHEN b"11" ⇒ B := b"01";                                — [9]
                   A := b"00";                                — [10]
      END CASE;                                               — [11]

      OUT1 ⇐ A;                                               — [12]

      IF (mode2) Then                                         — [13]
         A := IN1;                                            — [14]
      ELSE
         OUT2 ⇐ B;                                            — [15]
         WAIT UNTIL (not CLOCK'stable) and (CLOCK='1');       — [16]
         OUT3 ⇐ A + IN1;                                      — [17]
      END IF;                                                 — [18]

      OUT4 ⇐ A + B;                                           — [19]
   END process;                                               — [20]
END  behavior;
```

Figure 5

VHDL example.

determine, among other things, necessary interconnections and which values must be stored in registers.

- *Scheduling*

The scheduling task is responsible for mapping operations in the input description to control steps in the finite-state machine [20]. It creates the control-automaton graph in the HIS data model.

In general, a description may have to be scheduled in multiple control steps in order to

- *Implement fully behavioral constructs.*
  If the description contains behavioral constructs such as unbounded loops (for/while), scheduling is required in order to control the loop iteration.
- *Satisfy user constraints.*
  In HIS the user may specify the number and type of available functional units (FUs). If a description contains operations which exceed the user constraint on the number of FUs, it will be partitioned into control steps, so that in each control step only the maximum allowed

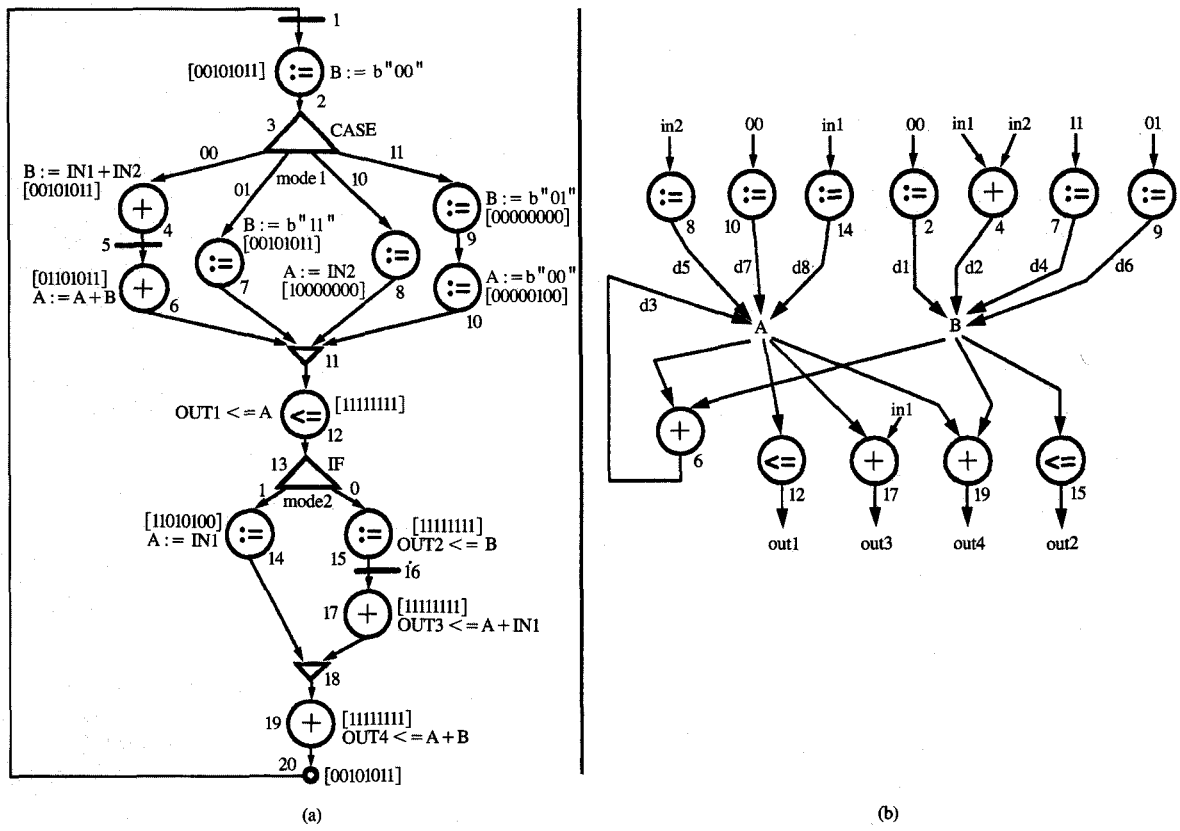


Figure 6

(a) Control and (b) data-flow graphs for VHDL example in Figure 5.

number of FUs are used. In the allocation step, the FUs can be merged so that the overall number of FUs does not exceed the constraints.

- *Implement explicit scheduling operations.* Each language contains specific ways of specifying an explicit schedule. In VHDL, the statement *Wait Until Not Clock'Stable and Clock* = '1'; can be used to force operations in different control steps. For conciseness, this statement is referred to simply as *Wait Until Clock* in the remainder of this paper. In Verilog HDL, the *Event* control (@) statement is used to specify clock transitions. If such operations are present in the description, scheduling is required in order to implement the correct semantics.

The scheduling algorithms in HIS are able to handle the cases above in a general way, including multiple user constraints and multiple *Wait* statements. If the description contains unbounded loops or user constraints are given, a path-based scheduling algorithm is applied. Alternatively, if the description has no unbounded loops and no constraints are given, a faster, register-transfer-level scheduling algorithm is applied. If the description contains no user constraints, no *Wait* statements, and no unbounded loops, it is implemented in a single control step (by either algorithm). In this case, no finite-state machine is required and the implementation consists only of combinational logic plus data registers. The decision as to which scheduling algorithm to use is made automatically, depending on the constructs used in the description and on the user-defined constraints. The use of both algorithms makes HIS capable of synthesizing large control- and/or data-dominated designs. Currently, there is no provision in HIS for the automatic synthesis of pipelines.

Path-based scheduling

Many algorithms in HIS are based on the concept of *path-based* synthesis. Path-based synthesis comprises

techniques for scheduling and allocation based on the execution *paths* in the input specification. A path represents a sequence of operations in the control-flow graph which is executed under certain input conditions. Different execution paths result from the presence of conditional operations and loops.

HIS uses a path-based scheduler called the *As-Fast-As-Possible* (AFAP) algorithm, first presented in [21, 22] and further extended in [23]. One important difference between the AFAP algorithm and most other scheduling algorithms is the treatment of conditional operations and loops. The AFAP algorithm handles conditional operations and loops in a transparent way because it works directly on the different execution paths caused by conditionals and loops. Most other scheduling algorithms (e.g., force-directed scheduling [11], Maha [9], Bud [24], Elf [25]) are based on the parallelization of data-flow operations, without considering different control paths. The result is that data-flow schedulers are able to optimize the number of control steps on only the critical path, while the AFAP algorithm can optimize the number of control steps on all execution paths.

The designer can specify constraints on the final implementation which are used to guide the AFAP scheduler. They typically define bounds in the implementation, such as the maximum number of functional units available or the maximum clock cycle. Constraints are applied to each execution path as *intervals*. A *constraint interval* delimits a sequence of operations which cannot be executed in the same control step. Therefore, in order for the constraint to be satisfied, the sequence of operations must be scheduled in two or more control steps.

The AFAP algorithm schedules each path in the control-flow graph independently, taking into account the order of operations in the input description and the set of constraints applicable to each path. The minimum number of control steps is obtained for all paths (not just the critical path) using exact clique-covering techniques.

To illustrate the AFAP algorithm, consider the CFG in Figure 6. The first step is the derivation of all execution paths in the CFG. Any sequence of operations starting at a given *first operation* and ending at a *path-breaking* operation defines a path. The *first operation* in a path can be 1) the entry operation in the CFG; 2) the first operation in a loop; or 3) the operation representing a *state-transition* operation (such as the *Wait Until Clock* in VHDL). A *path-breaking* operation can be 1) an operation with no successors; 2) an operation succeeded only by feedback edges; or 3) an operation representing a state transition. This CFG contains ten paths, as shown in **Figure 7**.

The second step is the mapping of all user constraints onto the paths. As an example, assume that a constraint of one adder is used, which implies that the final

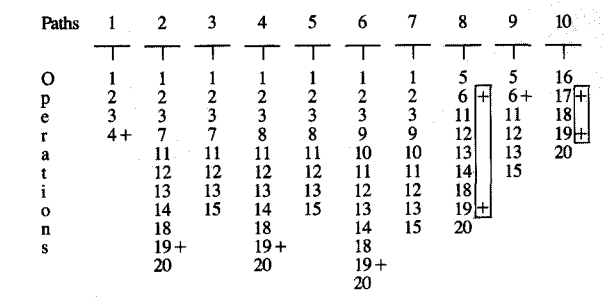


Figure 7

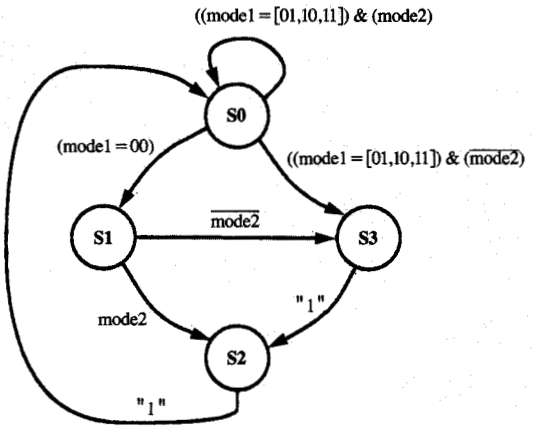
Execution paths in CFG in Figure 6.

implementation should not contain more than one adder. This results in the creation of two constraint intervals, attached to paths 8 and 10 in Figure 7. These constraints imply that the operations contained in each interval should not be executed in the same control step. For example, in order to satisfy the constraint on path 10, the scheduler must place operations 17 and 19 (the addition operations) in different control steps. In this way, the adders used in different states can be shared, and only one is needed overall.

Given paths and constraints, the scheduler must *cut* the constrained paths into as many control steps as necessary. If multiple constraints are present on the same path, clique-covering techniques are used to find the minimum number of *cuts* which satisfy all constraints. This results in the minimum number of control steps per path. In Figure 7, all paths except paths 8 and 10 are unconstrained and can therefore be scheduled in one control step. Paths 8 and 10 have one constraint each; hence, they need to be scheduled into two control steps each. In addition, the scheduler overlaps the control steps for all paths in order to minimize the total number of states in the final finite-state machine.

In Figure 7, the constraints are *cut* between nodes 14 and 18, and between nodes 17 and 18, so that both are satisfied and the control steps starting at node 18 can be overlapped on both paths. Once the control steps are defined, the scheduler generates the required finite-state machine. The final schedule for the description in Figure 6 under the constraint of one adder is given in **Figure 8(a)**. The conditions controlling the execution of each operation in a state are given in **Figure 8(b)**.

Note that each execution path is scheduled in the minimum number of control steps satisfying the constraints. This may require some operations to be



(a)

State	Operation node	Condition
S0	1,2,3	"1"
	4	mode1 = 00
	7	mode1 = 01
	8	mode1 = 10
	9,10	mode1 = 11
	11,12,13	mode1 = [01,10,11]
	14,18,19,20	(mode1 = [01,10,11]) & (mode2)
	15	(mode1 = [01,10,11]) & (mode2)
S1	5,6,11,12,13, 14	"1"
	15	mode2
		mode2
S2	18,19,20	"1"
S3	16,17	"1"

(b)

Figure 8

(a) Final FSM for description in Figure 6 under the constraint of one adder; (b) table of operation nodes and conditions in each state.

scheduled in multiple states, which is usually not allowed in data-flow-based schedulers (e.g., FDS [11], Maha [9], Bud [24], Elf [25]).

The output of the scheduler is the control-automaton graph, representing the required finite-state machine, plus the equations describing the state transitions and the conditions controlling the execution of each operation in a state.

The complexity of path-based scheduling is proportional to the number of paths, which may grow exponentially with the number of conditional operations. Paths represent the different functions being performed by the design, which tends to be a bounded number for most data-path-intensive applications but can grow very large for control-intensive designs. Examples with over twenty thousand paths have been successfully synthesized using this algorithm. In the case of an exponential explosion in the number of paths, special techniques can be applied in order to trade off optimality for execution time. One of these techniques is based on partitioning of the control-flow graph, which inserts *path-breaking* operations in order to reduce the number of paths. As a result, the scheduling may not find the optimum number of cycles for all paths, but it can then handle designs with millions of paths.

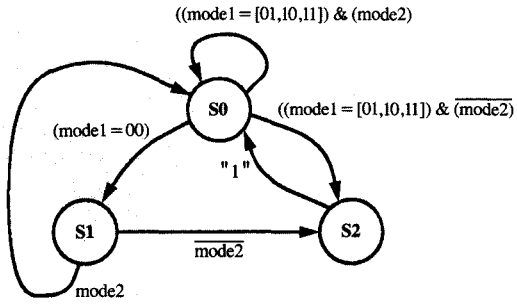
Register-transfer-level scheduling

In the absence of unbounded loops and user constraints, HIS uses a fast scheduling algorithm called *RTL scheduling*. The name indicates that scheduling is based only on the explicit scheduling constructs present in the description, such as the *Wait Until Clock* statements in VHDL.

The scheduling task in this case is simpler than in the general case because the state transition points can be extracted solely from the input description. However, the RTL scheduling algorithm must still derive the states and the state transition equations required in the finite-state machine. These steps can be very complex in the general case, when multiple explicit scheduling constructs are present in the same description. In fact, many VHDL synthesis systems from vendor companies as well as academic ones are not able to process multiple *Wait Until Clock* statements in a general way. HIS can handle descriptions containing any number of *Wait Until Clock* statements, even if placed within conditional statements, etc.

The RTL scheduling algorithm is performed in the following steps:

1. Traverse the CFG and identify the state transition points based on the explicit scheduling constructs. The operations following state transition points represent the first operations in the states.
2. Perform a breadth-first traversal of the CFG. As a node is visited, place it in the current state. The current state represents the state beginning at the "last visited" state transition point.
3. As a node is visited, compute and store its *activating condition*. This condition is given by a set {*current state*, *path condition*}, where the path condition is given



(a)

State	Operation node	Condition
S0	1,2,3	"1"
	4	mode1 = 00
	7	mode1 = 01
	8	mode1 = 10
	9, 10	mode1 = 11
	11, 12, 13	mode1 = [01,10,11]
	14, 18, 19, 20	(mode1 = [01,10,11]) & (mode2)
	15	(mode1 = [01,10,11]) & (mode2)
S1	5, 6, 11, 12, 13, 14, 18, 19, 20	"1"
		mode2
	15	mode 2
S2	16, 17, 18, 19, 20	"1"

(b)

Figure 9

(a) FSM produced by the RTL scheduling algorithm for the description in Figure 6; (b) table of operation nodes and conditions in each state.

by the conditions along the paths from the previous state transition point to the node. If a node must be scheduled in multiple states, the *activating condition* will be given by the *OR* of all sets {*current state*, *path condition*} applicable to the node.

4. Compute the conditions along the paths from the starting point to each transition point. These conditions represent the state transition equations.

Applying the RTL scheduling algorithm to the description in Figure 6 produces the finite-state machine shown in **Figure 9(a)**. The conditions controlling the execution of each operation in a state are given in **Figure 9(b)**. Note that some operations are scheduled in two states, which is necessary in order to implement correctly the semantics of the description.

The complexity of the RTL scheduling algorithm is proportional to the number of operations in the CFG, which in most cases is proportional to the number of lines in the input description. If the description contains a large number of conditional operations (therefore a large number of paths), this algorithm is considerably faster than the AFAP path-based algorithm.

- *Allocation*

The allocation task consists of the generation of the data-path elements required in the implementation. It creates the data-path graph in the HIS data model.

The data path is formed by functional units, storage elements, and interconnection elements. In HIS, allocation is performed in two steps: generation of an initial data path, and data-path optimization (or resource sharing).

The main problems in the generation of the initial data path lie in the derivation of the necessary storage elements, interconnections, and control signals, for which the techniques of data-flow analysis and path analysis [26] play a central role.

The initial data path is generated in the following steps:

1. *Functional unit allocation*
   Functional units are created in a one-to-one manner. For each operation in the data-flow graph, there is a corresponding element created in the data-path graph. An operation scheduled in multiple states is mapped onto the same functional unit. Functional units created in this step exhibit the necessary concurrency to allow resource sharing as defined by user constraints.
2. *Register allocation*
   Registers must be created to store values that are generated in one control step and used in another. These values are derived from the scheduled control-flow graph and the lifetime vectors produced by data-flow analysis. All assignments alive at the first operation in each state correspond to values produced in a previous state and therefore need to be stored. Multiple assignments to the same variable, if they must be stored, are stored in the same register (instead of *unfolding* the registers). Registers may also be created to implement correctly the semantics of the language

description. In VHDL, for example, all signals assigned in processes containing *Wait Until Clock* statements are stored.

3. *Allocation of multiplexors and interconnections*
   Interconnections are created to connect functional units, registers, and ports, as specified in the input description. Multiplexors are used to converge the outputs of multiple sources to one destination. Path analysis and data-flow analysis are again used to determine the correct sources for any destination.
4. *Generation of control signals*
   Control signals are required to *activate* the data-path elements. They include *load-enable* signals for registers, *select* signals for multiplexors, and *select* signals for multifunction functional units. Control signals are represented as BDDs, which are computed on-the-fly during scheduling and allocation. The ordering of the input variables in the BDDs is derived from the ordering of the conditional operations in the CFG [26]. BDDs are used for representing the control logic only; for example, in the statement *if A < B then s <= val1;*, the comparator is actually implemented in the data path, and its output is a control variable which becomes part of the BDDs. The BDDs are implemented as a two-level network if its size is manageable. If the two-level representation becomes too large, the system implements the BDDs as a multilevel network (whose unoptimized size is proportional to the number of BDD nodes).

As an example of initial allocation, consider again the description in Figure 6, scheduled as shown in Figure 9. The initial data path is given in **Figure 10**.

Four adders are created in the initial data path, based on the four *add* operations in the data-flow graph.

Variables A and B must be stored in registers. Note that the value assigned to B by operation 4 ($Add1$) in state $S0$ must be stored because it is used by operation 6 in state $S1$. Certain assignments to variables may or may not have to be stored, depending on the outcome of the conditional operations (that is, depending on which path is executed). For example, the assignment to B by operation 2 only has to be stored if path 5 (see Figure 7) is executed. Under all other paths, this assignment is overwritten by another assignment and therefore is not used.

Interconnections are also derived on the basis of the paths being executed. For example, the inputs to adder $Add4$ implementing operation 19 come from various sources. There are two possible values representing variable A—one coming from register A itself (if the FSM is in state $S2$ and path 10 is executed), and the other coming from the input $IN1$ (through the assignment in operation 14). Both sources converge to multiplexor $Mux3$, whose output is connected to one input of $Add4$. The other input to operation 19 comes from variable B, which can have four different values, depending on which path is executed. One of these values comes from the output of register B (if the FSM is in states $S1$ or $S2$), and the others come from the assignments at operations 2, 7, and 9 (in state $S0$). These four values reach the second input of $Add4$ via multiplexor $Mux4$. The select signals for all multiplexors and load-enable signals for registers A and B are given in Figure 10.

• *Data-path and control optimizations*

Optimizations based on resource sharing are performed on the data path. Registers, functional units, and multiplexors are merged using global algorithms based on coloring [27, 28] and clique-covering techniques [29].

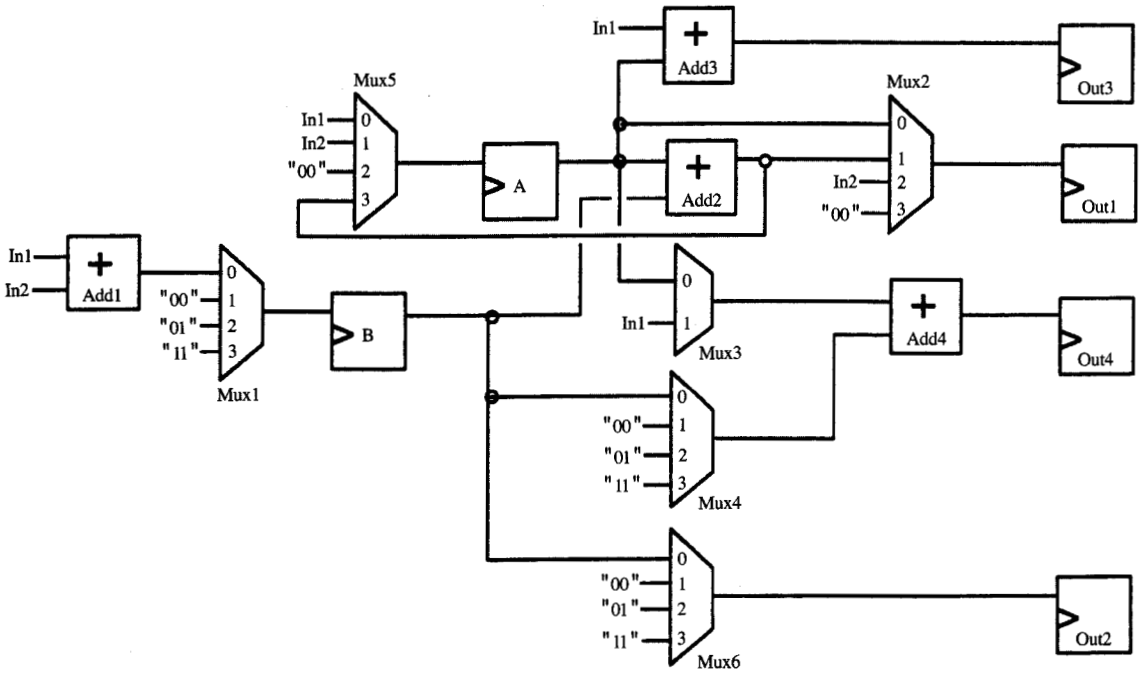
The first step in resource sharing is the determination of which hardware blocks can be shared. This is derived by building the *conflict graph* for the blocks. The conflict graph is defined by $CG = \{H, E\}$, where H is the set of data-path elements and E is the set of conflict edges. An edge $e = \{h_i, h_j\}$ implies that elements h_i and h_j are in conflict and therefore cannot be shared. In general, two registers can be shared if their lifetimes do not overlap. Two functional units can be shared if they are always used under mutually exclusive conditions.

Two merging algorithms have been implemented in HIS. The first one minimizes the final number of functional units and registers by coloring the conflict graph in the minimum number of colors. All elements with the same color can be combined into the same data-path element. The second algorithm merges data-path elements based on a cost function, taking into account the area of the blocks being merged, the final area after merging, and the required multiplexors.

The novelty in the resource sharing in HIS lies in the generation of the conflict graph. It is important that the conflict graph represent exactly the maximum sharing possibilities. A conflict graph with unnecessary edges may restrict the sharing possibilities, resulting in suboptimal results.

Given a scheduled CFG and the lifetime vectors from data-flow analysis, register lifetimes and conflict edges are determined in the following way:

> The lifetimes of assignment edges computed during data-flow analysis are *projected* onto the states of the FSM. A register, storing a variable, is *alive* in a state if any assignment to that variable is alive at the first operation in the state. Therefore, in order to determine all registers alive in a state, one must check all assignments alive at the first operation in the state. Two registers alive in the same state have overlapping lifetimes and cannot be shared; therefore, a conflict edge should be created between them.



Control signals (unoptimized, unencoded)

MUX1: SEL_0 = S0.(mode1 = 00)
SEL_1 = S0.(mode1 = 10).$\overline{\text{mode2}}$
SEL_2 = S0.(mode1 = 11).$\overline{\text{mode2}}$
SEL_3 = S0.(mode1 = 01).$\overline{\text{mode2}}$

MUX2: SEL_0 = S0.(mode1 = 01)
SEL_1 = S1
SEL_2 = S0.(mode1 = 10)
SEL_3 = S0.(mode1 = 11)

MUX3: SEL_0 = S2
SEL_1 = S0.(mode1 = [01,10,11]).mode2 | S1.mode2

MUX4: SEL_0 = S1.mode2 | S2
SEL_1 = S0.(mode1 = 10).mode2
SEL_2 = S0.(mode1 = 11).mode2
SEL_3 = S0.(mode1 = 01).mode2

MUX5: SEL_0 = S0.(mode1 = [01,10,11]).mode2 | S1.mode2
SEL_1 = S0.(mode1 = 10).$\overline{\text{mode2}}$
SEL_2 = S0.(mode1 = 11).$\overline{\text{mode2}}$
SEL_3 = S1.$\overline{\text{mode2}}$

MUX6: SEL_0 = S1.$\overline{\text{mode2}}$
SEL_1 = S0.(mode1 = 10).$\overline{\text{mode2}}$
SEL_2 = S0.(mode1 = 11).$\overline{\text{mode2}}$
SEL_3 = S0.(mode1 = 01).$\overline{\text{mode2}}$

REG A: Load_A = S0.(mode1 = 01).mode2 | S0.(mode1 = [10,11]) | S1
REG B: Load_B = S0.(mode1 = 00) | S0.$\overline{\text{mode2}}$

Figure 10

Initial data path for description in Figure 6, scheduled according to Figure 9.

Consider, for example, the FSM in Figure 9. In order to determine which registers have overlapping lifetimes on state $S1$, one must first retrieve the lifetime vector from the first operation in state $S1$. The lifetime vector of operation 6 (see Figure 6) shows that there are live assignments at positions {2, 3, 5, 7, 8}, which correspond to assignments to variables A and B. This implies that both variables are stored and that their lifetimes overlap on state $S1$. Hence, a conflict edge is created between registers A and B. By repeating this procedure for all states, the complete conflict graph is generated. This algorithm, because it uses global data-flow analysis, is able to take into account conditional operations and loops, and finds the exact conflict graph for register sharing.

The REAL [19] algorithm also computes the lifetimes of registers, but in a more limited way because it only looks at single *if-then-else* blocks. As a result, sharing may be more restrictive.

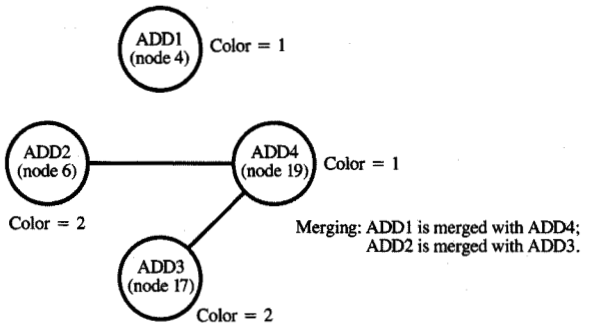


Figure 11

Conflict graph for addition operations in description in Figure 6, scheduled according to Figure 9.

The conflict graph for functional units is generated on the basis of the mutual exclusiveness of the operations implemented in the FUs. Mutual exclusiveness of operations is determined as follows:

> Two operations are *NOT* mutually exclusive if the conditions controlling their execution are not orthogonal. Let $C(op_i)$ and $C(op_j)$ be the conditions controlling operations op_i and op_j. Operations op_i and op_j are not mutually exclusive if $C(op_i) \wedge C(op_j) \neq 0$. Given any two operations scheduled in the same state, if the *AND* of their conditions is not null, they are not mutually exclusive, and a conflict edge is created between the corresponding functional units. This procedure is repeated for all pairs of operations in all states. The conditions controlling the execution of operations are computed during scheduling and allocation (see Figure 9).

This algorithm is able to derive general conflict information on all functional units present in the design, including those scheduled in different states as well as in different branches of conditional operations.

Consider again Figure 9. The conflict graph for the adders in this design is shown in **Figure 11**. Operations 4 and 19 are both scheduled in state *S*0; however, they are indeed mutually exclusive, because the conditions controlling both operations are orthogonal. Note that although these operations are not in different branches of the same *if-then-else*, HIS was still able to determine that they are never executed at the same time, and therefore can be shared.

Other resource-sharing algorithms, including those from vendors and universities (e.g., [11, 29, 30]) are more limited because they consider only operations in different states and operations in different branches of a conditional operation. Those algorithms may not be able to determine that operations 4 and 19 can be shared.

The final data path for the FSM in Figure 9, after resource sharing, is given in **Figure 12**.

The techniques used in HIS for control optimization are based on two-level minimization, behavioral don't-cares, state minimization, and state encoding. The control signals can be based on binary as well as multivalued variables. Behavioral don't-cares are extracted during synthesis and used in the minimization of the control signals. Details concerning the control optimization algorithms in HIS can be found in [31].

4. Production use of HIS

The HIS system is currently in production use by several groups at different IBM sites. The previous synthesis methodology used by these groups consisted of automatic logic synthesis starting from gate-level descriptions. Among the key characteristics in HIS which were required by these groups to switch to a high-level design methodology were the following:

- The ability to synthesize designs as efficiently or better than in the previous methodology.
- The ability to synthesize different levels of abstraction, including sequential VHDL, and automatic inference of registers.
- The close integration of HIS with the BooleDozer logic synthesis system, used for gate-level optimization and technology mapping.
- The means to control the synthesis process if necessary. A significant level of control was possible by making HIS accept the same synthesis directives accepted by the previous methodology, which were already familiar to the designers. In addition, special-purpose attributes for controlling the high-level synthesis steps and the interaction with BooleDozer logic synthesis proved very useful.

The use of higher-level descriptions significantly increased design productivity by speeding up the VHDL coding and simulation turnaround. The use of high-level synthesis and logic synthesis combined gave the designers the ability to process higher-level descriptions and obtain results comparable to or better than in the previous methodology.

The savings in total design time, considering specification, simulation, synthesis, and layout, depend upon the previous methodology used. For example, in [14], significant savings were achieved in going from a

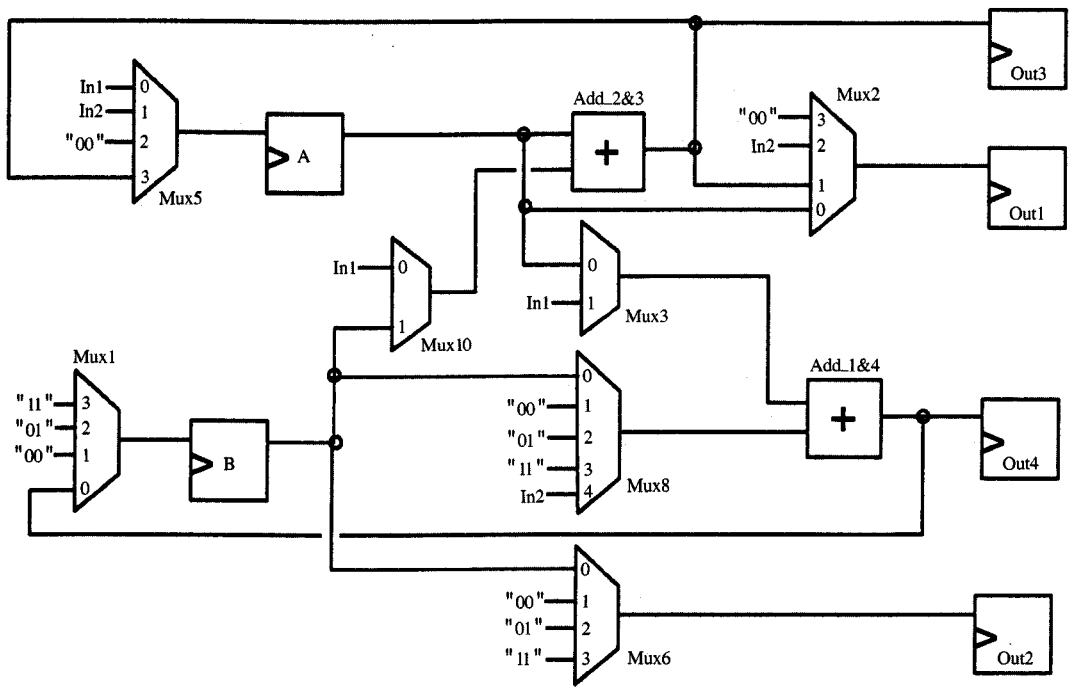


Figure 12

Final data path for description in Figure 6, scheduled according to Figure 9, after sharing of adders.

schematics-based methodology to a high-level synthesis methodology. This is clearly to be expected, since neither the previous nor the new methodology included logic synthesis (which is the most time-consuming step).

In IBM, the previous methodology was already a language-based and logic-synthesis-based methodology, although dominated by concurrent and structural styles. In this case, savings in synthesis time (considering high-level and logic synthesis) were not expected in a single synthesis pass, since most of the synthesis time was dominated by logic synthesis. However, considering that most designs need several synthesis runs, it was always faster to refine the design at a high level (running high-level synthesis only), and then proceed to logic synthesis. Most of the design time savings came from four sources: 1) the ability to use sequential descriptions and have all the control and data-path logic and registers automatically generated; 2) designer control over implementation directly in the description; 3) the ability to refine the design at a high level (before logic synthesis); and 4) faster simulations due to sequential descriptions.

Table 1 Characteristics of four designs produced by HIS.

Design	*VHDL lines*	*Entities*	*Gates* (thousands)	*Cycle time* (ns)
Chip1	13500	9	70	30
Chip2	8100	9	40	30
Chip3	5900	1	20	15
Chip4	6300	8	5	30

In addition to its common use as a *pushbutton* synthesis system, HIS is being used in different ways, as highlighted below.

- *Automatic synthesis from VHDL*
  HIS is being used as an automatic synthesis tool, integrated with the BooleDozer logic synthesis system. Among the chips fabricated to date are ASIC peripheral chips and controllers in CMOS technology. **Table 1** shows the numbers for four designs, in terms of VHDL

lines, number of VHDL entities in which the design was partitioned, number of final gates after logic synthesis, and clock frequency. The execution time for HIS is usually one to two orders of magnitude faster than the execution time for the other design stages (e.g., logic synthesis, layout, simulation).

- *Fast synthesis for emulation*
  HIS and BooleDozer are also being used as a fast synthesis path for mapping VHDL into a gate-level netlist for emulation purposes. Since the purpose is to produce a functional netlist, without being overly concerned for the final area or delay, fast algorithms can be used both in HIS and in BooleDozer, resulting in fast turnaround time. This methodology is being used in the design of a CMOS microprocessor. Under this methodology, HIS and BooleDozer have been used to synthesize over 175 thousand lines of VHDL, producing netlists with approximately 1.75 million gates.
- *Fast synthesis for verification*
  Similarly to the previous case, HIS and BooleDozer are being used as a fast synthesis path for formal verification purposes. This methodology combines custom CMOS design with VHDL design, synthesis, simulation, and verification. In the design of a high-performance microprocessor, a number of chip partitions are custom-designed at the transistor level. These partitions are also coded in VHDL and simulated. HIS and BooleDozer are used to synthesize the VHDL descriptions to a gate-level netlist. Then a formal verification tool, VERITY [32], is used to verify that the gate-level netlist synthesized from the VHDL is functionally equivalent to the transistor-level netlist for the same partition. In this way, the functional simulation can be confined to the VHDL level, and need not be repeated at the transistor level. Under this methodology, HIS and BooleDozer have been used to synthesize approximately 40 thousand lines of VHDL representing over 400 custom partitions.
- *Fast mapping for cycle simulation*
  The ability of HIS to provide a fast synthesis path from VHDL to a gate-level netlist makes possible its integration with other tools which typically require inputs strictly at the gate level. One such application is cycle simulation. Most VHDL simulators are *event-driven* simulators, which is a direct consequence of the semantics of the language. However, event-driven simulators are, in general, slower than gate-level cycle simulators. As a result, most VHDL simulators are still slower than existing gate-level cycle simulators. For this reason, it would be efficient if one could simulate VHDL using a cycle simulator. One possible way to achieve this goal is to use HIS as a VHDL front end to a cycle simulator. HIS can be used to generate a gate-level netlist from VHDL, which can then be submitted to the cycle simulator.
- *Early estimation and analysis tool*
  The HIS system is being integrated with timing analysis and floorplanning tools with the goal of providing an estimation and analysis tool to be used early in the design process. Under this tool, the design is initially synthesized by HIS to produce a technology-independent network consisting of random logic and data-path elements. Given a technology library, HIS then estimates the area and delay of the elements of the network. Different algorithms are used for estimating the random logic and data-path elements. Timing analysis is then performed using these estimations in order to provide timing information on the whole design. The floorplanning tool is used for placing the blocks and deriving more accurate wire capacitance information, which can be used to obtain a more precise timing analysis and can be fed forward to logic synthesis. This *forward* design methodology allows layout considerations to be taken into account much earlier in the design process, reducing the expensive steps of post-logic-synthesis layout, back-annotation, and resynthesis.

5. Conclusions

This paper has presented the main algorithms, modeling capabilities, and methodology considerations in the HIS system.

HIS is able to synthesize VHDL descriptions in different abstraction levels, ranging from behavioral to structural. HIS contains general algorithms for scheduling and allocation which can handle behavioral, sequential, concurrent, and structural constructs, and user-defined constraints. Different algorithms, specially tuned to the abstraction level of the description, are used in order to synthesize designs efficiently for execution time and memory usage.

The scheduling and allocation algorithms in HIS are based on global techniques such as global data-flow analysis and path analysis. The scheduler is able to find the exact minimum number of control steps required by a design under user constraints. The resource-sharing algorithms are more general than previous algorithms in determining the set of elements that can be shared. Global data-flow analysis, path analysis, and BDDs are used to derive the exact maximum set of sharable data-path elements.

Among the main characteristics required for production use of HIS were 1) efficient high-level synthesis algorithms, 2) a rich set of directives for controlling the synthesis process, and 3) integration with the BooleDozer logic synthesis system.

The capabilities of the system have proved to be powerful enough for it to be used in production as part of the synthesis methodology, and in domains for which it was not originally intended, such as a front end for

verification and cycle simulation. Several CMOS chips, ranging from small to large, have been produced using HIS.

Acknowledgments

The authors thank Kurt Carpenter, Pete Osler, and Coby Sella for being patient early users of the system and for many helpful suggestions for improving its practicality.

References

1. T. D. Friedman and S. C. Yang, "Methods Used in an Automatic Logic Design Generator (ALERT)," *IEEE Trans. Computers* **C-18,** 593–614 (1969).
2. S. J. Hong, R. G. Cain, and D. L. Ostapko, "MINI: A Heuristic Approach for Logic Minimization," *IBM J. Res. Develop.* **18,** 443–458 (September 1974).
3. R. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1985.
4. R. Brayton and C. McMullen, "Synthesis and Optimization of Multistage Logic," *Proceedings of the International Conference on Computer Design*, IEEE, October 1984, pp. 23–28.
5. John A. Darringer, Daniel Brand, John V. Gerbi, William H. Joyner, Jr., and Louise Trevillyan, "LSS: A System for Production Logic Synthesis," *IBM J. Res. Develop.* **28,** 537–545 (September 1984).
6. J. Vanhoof, K. Van Rompaey, I. Bolsens, G. Goossens, and H. De Man, *High-Level Synthesis for Real-Time Digital Signal Processing*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1993.
7. D. E. Thomas, E. D. Lagnese, R. A. Walker, J. A. Nestor, J. V. Rajan, and R. L. Blackburn, *Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1990.
8. D. C. Ku and G. De Micheli, "Relative Scheduling Under Timing Constraints: Algorithms for High-Level Synthesis of Digital Circuits," *IEEE Trans. Computer-Aided Design* **CAD-11,** 696–718 (June 1991).
9. A. C. Parker, J. T. Pizarro, and M. Mlinar, "MAHA: A Program for Datapath Synthesis," *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, June 1986, pp. 461–466.
10. D. Gajski, N. Dutt, A. Wu, and S. Lin, *High-Level Synthesis*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1992.
11. P. G. Paulin and J. P. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's," *IEEE Trans. Computer-Aided Design* **CAD-8,** 661–679 (June 1989).
12. R. Camposano, R. A. Bergamaschi, C. Haynes, M. Payer, and S. M. Wu, "The IBM High-Level Synthesis System," *High-Level VLSI Synthesis*, R. Camposano and W. Wolf, Eds., Kluwer Academic Publishers, Dordrecht, The Netherlands, 1991, pp. 79–104.
13. C.-J. Tseng, R.-S. Wei, S. G. Rothweiler, M. M. Tong, and A. K. Bose, "Bridge: A Versatile Behavioral Synthesis System," *Proceedings of the 25th ACM/IEEE Design Automation Conference*, Anaheim, CA, July 1988, pp. 415–420.
14. R. W. Hunter, T. Fuhrman, and D. E. Thomas, "Working Chips from High-Level Synthesis: A Case Study from Industry," *Proceedings of the IEEE Custom Integrated Circuits Conference*, San Diego, May 1994, pp. 144–147.
15. T. Fuhrman, "Industrial Extensions to University High Level Synthesis Tools: Making It Work in the Real World," *Proceedings of the 28th ACM/IEEE Design Automation Conference*, San Francisco, June 1991, pp. 520–525.
16. R. Camposano and R. M. Tabet, "Design Representation for the Synthesis of Behavioral VHDL Models," *Proceedings of the 9th International Symposium on Computer Hardware Description Languages and Their Applications*, Elsevier Science Publishers B.V., Washington, DC, June 1989, pp. 49–58.
17. A. Aho, R. Sethi, and J. Ullman, *Compilers, Principles, Techniques and Tools*, Addison-Wesley Publishing Co., Reading, MA, 1986.
18. R. Camposano and W. Rosenstiel, "Synthesizing Circuits from Behavioral Descriptions," *IEEE Trans. Computer-Aided Design* **CAD-8,** 171–180 (February 1989).
19. F. J. Kurdahi and A. C. Parker, "REAL: A Program for Register Allocation," *Proceedings of the 24th ACM/IEEE Design Automation Conference*, June 1987, pp. 210–215.
20. M. C. McFarland, A. C. Parker, and R. Camposano, "The High-Level Synthesis of Digital Systems," *Proc. IEEE* **78,** No. 2, 301–318 (February 1990).
21. R. Camposano and R. A. Bergamaschi, "Synthesis Using Path-Based Scheduling: Algorithms and Exercises," *Proceedings of the 27th ACM/IEEE Design Automation Conference*, June 1990, pp. 450–455.
22. R. Camposano, "Path-Based Scheduling for Synthesis," *IEEE Trans. Computer-Aided Design* **CAD-10,** 85–93 (January 1991).
23. R. A. Bergamaschi, R. Camposano, and M. Payer, "Scheduling Under Resource Constraints and Module Assignment," *INTEGRATION, the VLSI Journal* **12,** 1–19 (December 1991).
24. M. C. McFarland and T. J. Kowalski, "Incorporating Bottom-Up Design into Hardware Synthesis," *IEEE Trans. Computer-Aided Design* **CAD-9,** 938–950 (September 1990).
25. E. F. Girczyc, R. J. A. Buhr, and J. P. Knight, "Applicability of a Subset of ADA as an Algorithmic Hardware Description Language for Graph-Based Hardware Compilation," *IEEE Trans. Computer-Aided Design* **CAD-4,** 134–142 (April 1985).
26. R. A. Bergamaschi, R. Camposano, and M. Payer, "Allocation Algorithms Based on Path Analysis," *INTEGRATION, the VLSI Journal* **13,** 283–299 (September 1992).
27. G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register Allocation Via Coloring," *Computer Science Technical Report RC-8395 (#36543)*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, April 1980.
28. T. K. Philips, "New Algorithms to Color Graphs and Find Maximum Cliques," *Computer Science Technical Report RC-16326 (#72348)*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1990.
29. C. J. Tseng and D. P. Siewiorek, "Automated Synthesis of Data Paths in Digital Systems," *IEEE Trans. Computer-Aided Design* **CAD-5,** 379–395 (July 1986).
30. B. Gregory, D. MacMillen, and D. Fogg, "ISIS: A System for Performance Driven Resource Sharing," *Proceedings of the 29th ACM/IEEE Design Automation Conference*, June 1992, pp. 285–290.
31. R. A. Bergamaschi, D. Lobo, and A. Kuehlmann, "Control Optimization in High-Level Synthesis Using Behavioral Don't Cares," *Proceedings of the 29th ACM/IEEE Design Automation Conference*, Anaheim, CA, June 1992, pp. 657–661.
32. A. Kuehlmann, A. Srinivasan, and D. P. LaPotin, "Verity—A Formal Verification Program for Custom CMOS Circuits," *IBM J. Res. Develop.* **39,** 149–165 (1995, this issue).

Received May 27, 1994; accepted for publication September 26, 1994

Reinaldo A. Bergamaschi *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (rab@watson.ibm.com).* In 1982, Dr. Bergamaschi graduated in electronics engineering (with honors) from the Aeronautics Institute of Technology, Sao Jose dos Campos, Brazil, and in 1984 he received the M.E.E. degree (with distinction) from the Philips International Institute, Eindhoven, The Netherlands. In 1989 he received the Ph.D. degree in electronics and computer science from the University of Southampton, England, and joined the IBM Thomas J. Watson Research Center, where he currently leads the high-level synthesis project. His main interests are in the areas of computer-aided design, high-level synthesis, logic synthesis, and computer design.

Richard A. O'Connor *IBM Microelectronics Division, EDA Laboratory, 522 South Road, Poughkeepsie, New York 12601 (oconnorr@vnet.ibm.com).* Mr. O'Connor received his undergraduate degree in electrical engineering from Rensselaer Polytechnic Institute in 1990. Currently, he is a member of the synthesis development team at the IBM Corporation in Poughkeepsie, New York, and on a part-time basis is pursuing his master's degree in computer science. Prior to working for IBM, he spent time working for Hewlett-Packard in Andover, Massachusetts, and Asea Brown-Boveri in Windsor, Connecticut.

Leon Stok *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (stokl@watson.ibm.com).* Dr. Stok studied electrical engineering at the Eindhoven University of Technology, The Netherlands, from which he graduated with honors in 1986. In 1991 he received the Ph.D. degree from the Eindhoven University. He is currently working in the System, Technology and Science Department on BooleDozer, the IBM synthesis tool. Prior to this assignment, Dr. Stok worked in the "Unternehmensbereich Kommunikations-und Datentechnik" of Siemens AG in Munich in 1985, and in the Mathematical Sciences Department of the IBM Thomas J. Watson Research Center during the second half of 1989 and the first half of 1990. Dr. Stok has published several papers on various aspects of high-level and architectural synthesis and on automatic placement and routing for schematic diagrams. In 1993, he received an IBM Research Division Award for his contributions to the BooleDozer Synthesis system. His research interests include high-level and logic synthesis, layout synthesis, and system synthesis and verification. He is a member of the Institute of Electrical and Electronics Engineers.

Michael Z. Moricz *IBM Microelectronics Division, EDA Laboratory, 522 South Road, Poughkeepsie, New York 12601 (moricz@vnet.ibm.com).* Dr. Moricz received his master's and Ph.D. degrees in computer engineering from Syracuse University, Syracuse, New York, in 1989 and 1992, respectively. Currently he is a staff engineer at IBM in Poughkeepsie, New York. His research interests include logic synthesis, behavioral synthesis, and parallel processing. He has been a member of the IEEE Circuits and Systems Society since 1988.

Shiv Prakash *IBM Microelectronics Division, EDA Laboratory, 522 South Road, Poughkeepsie, New York 12601 (sprakash@vnet.ibm.com).* Dr. Prakash received his bachelor's degree (B.Tech.) in electrical engineering from the Indian Institute of Technology, Kanpur, India, in 1982, his master's degree in computing science from Simon Fraser University in Canada in 1987, and his Ph.D. degree in electrical engineering from the University of Southern California, Los Angeles, in 1993. Currently, he is a staff member in the IBM Microelectronics Division in Poughkeepsie, New York. His areas of interest include high-level synthesis, system-level synthesis, and multiprocessor systems. He is a member of the IEEE Computer Society and the IEEE Circuits and Systems Society.

Andreas Kuehlmann *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (kuehl@watson.ibm.com).* Dr. Kuehlmann received the Dipl.-Ing. degree and the Dr. sc. techn. degree in electrical engineering from the Technical University Ilmenau, Germany, in 1986 and 1990, respectively. From 1990 to 1991 he worked at the Fraunhofer Institute of Microelectronic Circuits and Systems, Duisburg, Germany, where he was engaged in the development of a design system for embedded microcontrollers. In 1991 Dr. Kuehlmann joined the IBM Thomas J. Watson Research Center in Yorktown Heights, New York. After working on various problems in high-level and logic synthesis, he concentrated primarily on verification techniques for large custom CMOS designs. Dr. Kuehlmann's primary research interests are in the design and verification of digital VLSI circuits, particularly in system design, high-level and logic synthesis, design verification, and layout generation.

D. Sreenivasa Rao *IBM Microelectronics Division, EDA Laboratory, 522 South Road, Poughkeepsie, New York 12601 (dsr@vnet.ibm.com).* Dr. Rao received his Ph.D. in computer engineering from the University of California in 1993; he has been with IBM's EDA Laboratory in Poughkeepsie, New York, since then. His area of expertise is high-level synthesis, but his research interests extend to other areas of CAD such as system design, routing and layout, and combinatorial optimization.