Introduction to High-Level Synthesis

Daniel D. Gajski
Loganath Ramachandran
University of California, Irvine

The basic problem of high-level synthesis is the mapping of a behavioral description of a digital system into a RTL design system consisting of a data path and a control unit. The authors introduce the FSMD model, which forms the basis for synthesis. They discuss the main considerations in a high-level synthesis environment: the input description language, the internal representation, and the main synthesis tasks—allocation, scheduling, and binding. They conclude with some problems that must be solved to make high-level synthesis a widely accepted methodology.

Today's VLSI technology allows companies to build large, complex systems containing millions of transistors on a single chip. To exploit this technology, designers need sophisticated CAD tools that enable them to manage millions of transistors efficiently.

Until recently, most ASIC (application-specific IC) and system houses used a capture-and-simulate design methodology. Following this methodology, the design department starts with a specific set of product requirements, usually supplied by the marketing department. Since these requirements contain no information about implementation of the product, a team of chief architects produces a rough block diagram of the chip architecture, which serves as a preliminary, incomplete specification. In some cases, the architects further refine this initial block diagram before giving it to a team of logic and layout designers.

The logic and layout designers convert each functional block into a logic or circuit schematic, which is captured by schematic-capture tools, and simulated to verify its functionality, timing, and fault coverage. Designers can also use the captured schematic to drive physical design tools for placement and routing of gates in gate-array technologies, or to map gates into standard or custom cells before placement and routing in custom technologies.

Only in the last few years has logic synthesis become recognized as an integral part of the design process, leading to an evolution in methodology from capture-and-simulate to describe-and-synthesize. The new methodology's advantage is that it allows us to describe a design in a purely behavioral form, devoid of implementation details, and then to synthesize the design structure with CAD tools.

Designers can apply the describe-and-synthesize methodology on several levels of abstraction. On the gate level, they can synthesize functional and control unit logic by means of combinational logic synthesis. They also can synthesize controllers from finite-state machine diagrams by means of sequential synthesis. On the register-transfer level (RTL), they can describe the behavior of ASICs with programs, algorithms, flowcharts, dataflow graphs, instruction sets, or generalized FSMs in which each state performs arbitrarily complex computations. Then they can synthesize these ASICs by means of high-level (or behavioral) synthesis techniques.

High-level synthesis is a sequence of tasks that transforms a behavioral representation into an RTL design. The design consists of functional units such as ALUs and multipliers, storage units such as memories and register files, and interconnection units such as multiplexers and buses.

Figure 1 shows a generic high-level synthesis system. The compiler converts the behavioral description into an internal representation. The RTL library contains the physical and simulation models of components to be used during synthesis. The netlister generates the final RTL structure, consisting of a netlist of RTL components and a simulation model of each component. To verify the synthesized design's correctness, the designer can simulate the input description and the

Originally published in *IEEE Design and Test of Computers,* Winter 1994 , pp. 44–54. Copyright © 1994 IEEE. All rights reserved.

generated netlist by means of the simulation environment, an adjunct to the high-level synthesis system.

The main advantages of high-level synthesis are productivity gains and better design space exploration. It achieves productivity gains by moving the design process to higher abstraction levels, where designers can specify, model, verify, synthesize, simulate, and debug designs in less time. The automation provided by high-level synthesis ensures a more systematic and efficient search of the large design spaces created by the shift to higher abstraction levels.

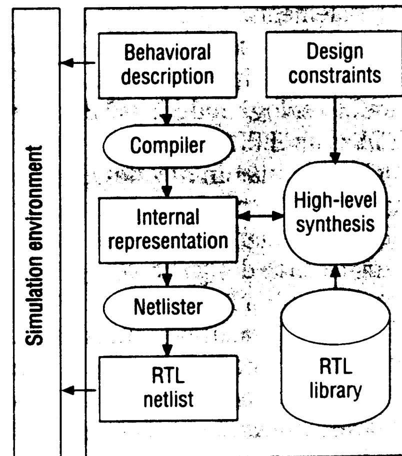

Figure 1. A generic high-level synthesis system.

Glossary

Allocation: determination of the type and quantity of resources to implement a design for given performance and area constraints

Binding (resource sharing): assignment of operations, memory accesses, and interconnections from the behavioral design descriptions hardware units for optimal area and performance

Chaining: the scheduling of two data-dependent operations in the same control-step

Describe-and-synthesize: design paradigm in which a designer specifies the intent of a design and uses automatic synthesis tools to implement the design at the next-lower abstraction level

FSM: finite-state machine design model for representing controllers that assigns Boolean constraints to output signals in every clock cycle

FSMD: finite-state machine with data path model for representation of control-dominated and data-dominated designs that augments the FSM model with variables and expressions that specify conditions and actions in each state

Multicycling: the scheduling of one operation in multiple dock cycles, enabling the use of slower functional units with a faster clock

Resource constrained-scheduling: assignment of operations into control steps, given the set of resources

Scheduling: the partitioning of design behavior into control steps such that all operations in a control step execute in one clock cycle

Syntactic variance: description style differences that generate differences in design quality from semantically equivalent behaviors

Time-constrained scheduling: assignment of operations into control steps, given a fixed execution time

VHDL: a hardware description language (IEEE Std 1076-1987) used by designers to describe design behavior and stature at various levels of abstraction

Originally published in *IEEE Design and Test of Computers,* Winter 1994 , pp. 44–54. Copyright © 1994 IEEE. All rights reserved.

Input description

The input description specifies the intent of the design. It is usually an algorithmic description of design behavior that does not contain structural information such as component types and their interconnections. Nor does it provide information about circuit structure, such as the number of pipeline stages or clock phases.

Designers usually write input descriptions in a special language called a hardware description language (HDL). Many HDLs support specification of both design behavior and design structure. Although the synthesis task becomes easier as the amount of structural detail increases, designers prefer specifying behavior—much easier and less time-consuming than specifying structure.

To specify design behavior, typical HDLs provide a set of variables and a set of operations for computing the variables' current values. Variable assignment statements assign values to the variables. Most languages provide condition constructs such as if and case statements, to allow conditional execution of the assignment statements. Repeated iterations through a sequence of statements can be specified with loop statements.

Examples of languages used in high-level synthesis include VHDL,[1] HardwareC,[2] Verilog,[3] and Silage.[4] Because VHDL is an IEEE standard and is becoming popular among designers, it has extensive commercial support. Widely used for hardware description, it can describe design behavior at several abstraction levels. The language supports sequential and concurrent assignments, conditional constructs, loops, procedures, and functions. However, it does not explicitly support hardware pipelining, interrupts, and hierarchical behavior.

Figure 2 shows a VHDL behavioral description. It consists of port and variable declarations followed by a process that encapsulates the design behavior. The process has five local variables: C, D, E, F, and Timer. The process waits until an external input X becomes 1, and then it executes a series of assignment statements if the value of Timer is not equal to 0. Because VHDL's semantics are primarily designed for simulation, high-level synthesis with VHDL is difficult.

HardwareC is based on the widely popular C language, augmented to support hardware features such as timing and synchronization, but it suffers from disadvantages similar to VHDL's. Silage is an applicative language, designed specifically to model dataflow applications. It provides explicit constructs for modeling computations on data streams, which are frequent in signal-processing applications.

Designers can write the behavioral description in many different styles. Although the styles are semantically equivalent, they differ syntactically in their use of certain language constructs. Most high-level synthesis tools are very sensitive to description style. Two designs synthesized from two semantically equivalent but syntactically different descriptions may differ significantly in quality. The quality of the synthesized design depends on the type and order of constructs used in the description. This is known as the problem of syntactic variance.

High-level synthesis systems have used two approaches to solve the syntactic variance problem. In the first approach, a set of modeling guidelines restricts the input description style, to achieve a unique input format. In the second approach, the modeler is free to use any convenient construct, but the synthesis tool transforms all descriptions into a unique internal form used during synthesis.

```
  entity sqrt_part is
      port (A,B: in integer);
          X: in bit);
          Y: out integer);
  end sqrt_part;

    architecture arch of sqrt_part is
    begin

      P0: process
        variable C, D, E, F: integer,
        variable Timer: integer;

    begin
        wait until X = 1;
        if (Timer != 0) then
          D :=shr  (A, 1);
          C := shr (B. 3);
          E :=B – C;
          F :=D + E;
          Y := max (F, B);

    else
        Timer :=Timer- l;
    end if,
  end process P0;
end architecture A;
```

Figure 2. Sample VHDL input description.

Originally published in *IEEE Design and Test of Computers,* Winter 1994 , pp. 44–54. Copyright © 1994 IEEE. All rights reserved.

Internal representation

The high-level synthesis system compiles the behavioral description into an internal representation. All synthesis tasks work from this representation. There are several types of internal representation. The most convenient type is the one that matches the problem most closely. For example, for a digital filter, which repeatedly performs a series of operations on an infinite input data stream, we want to represent the data, the arithmetic operations, and the read and write dependencies that define the order of execution. A dataflow graph (DFG) is the best way to do this.

A DFG consists of a set of nodes, each node representing one operation in the original description. Two nodes o_i and o_j are connected by an arc if there is a data dependence between them (that is, the result of operation oj is an input to operation o_i). In other words, the dependency arc connecting o_j and o_j indicates that operation o_j cannot execute before o_j executes. Since the DFG representation is based on data dependency alone, it is the most parallel representation of the description. Figure 3 shows the DFG for the expression $Y = \max((A \text{ shr } 1) + (B - (B \text{ shr } 3)), B)$.

The DFG is not sufficient to represent reactive or embedded systems, in which the control sequence is based on external conditions. In these cases, we must represent the control flow in addition to data dependencies. We do this by augmenting the DFG with control nodes. An example of an augmented representation style is the control-dataflow graph (CDFG), which allows representation of control constructs such as branches and loops. The CDFG contains special nodes to represent if conditions, case constructs and loops, and computational sequences.

Figure 4 shows the CDFG representation for the VHDL description in Figure 2. In this example, the expression $Y = \max((A \text{ shr } 1) + (B - (B\text{shr}3)), B)$ computes the value of Y only if an external input X is asserted and Timer is not equal to 0. The representation contains a wait node that checks for assertion of the external condition X. The if-begin node checks that Timer is not equal to 0, and the if-end node represents the end of the branch statement. The actual computation of the expression is embedded in a basic (dataflow) block.

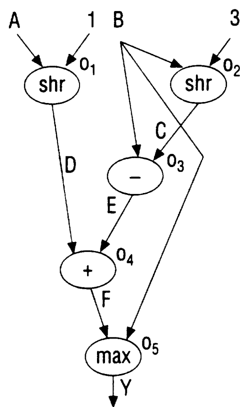

Figure 3. DFG representation.

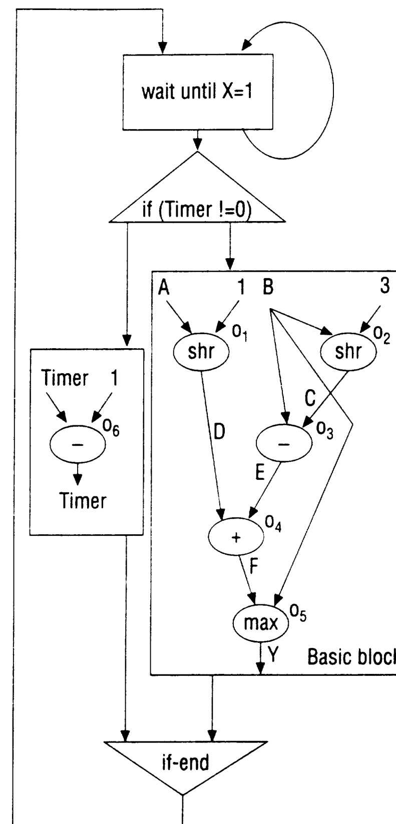

Figure 4. CDFG representation.

Originally published in *IEEE Design and Test of Computers,* Winter 1994 , pp. 44–54. Copyright © 1994 IEEE. All rights reserved.

The CDFG representation maintains the control structure specified by the designer. The CDFG represents each block of assignment statements in the original behavioral description as a separate basic block. Data dependency information is represented only within the basic block; data dependencies across basic blocks are not explicitly represented. A synthesis system that works directly from the CDFG representation must maintain the basic-block structure. For example, two operations in two sequential basic blocks never execute together, although they may not have any real dependencies. This is one of the major disadvantages of using a CDFG representation directly for synthesis.

We can make synthesis more efficient by removing user-defined control constructs and introducing an execution order based on data dependencies. This would result in the least restricted and most efficient internal representation. The value trace[5] and ADD[6] design representations incorporate these ideas. They also solve the syntactic variance problem by transforming the input description to a unique representation.

High-level synthesis model

Logic synthesis is based on the formalism of Boolean algebra, whereas sequential synthesis is based on the FSM model. For high-level synthesis, we extend the FSM model by adding variable assignments.

The FSM model consists of a set of states, a set of transitions between states, and a set of actions associated with these states or transitions. More formally, an FSM is a quintuple

$$<S, I, O, f: S \times I \rightarrow S, h: S \times I, \rightarrow O>$$

Here S is a set of states, I is a set of input values, O is a set of output values, and f and h are next-state and output functions that map a cross product of S and I into S and O, respectively. Functions f and h can be specified with Boolean equations, state tables, or state (bubble) diagrams.

The FSM model works well for up to several hundred states. Beyond that, the model becomes incomprehensible to human designers. Even low-complexity components such as I/0 interfaces and bus controllers can have several thousand states if we count all storage elements. To adapt the FSM model for more complex designs, we introduce a set of integer and floating-point variables stored in registers, register files, and memories. Each variable replaces thousands of different states. For example, a 16-bit integer variable represents 2^{16} or 65,536 different states; thus, the introduction of a 16-bit variable reduces the number of states in the FSM model by 65,536. The use of variables leads to the concept of an FSM with a data path (FSMD).

We formally define an FSMD as follows: A set of storage variables VAR, a set of expressions $EXP = \{f(x, y, z, \ldots) \mid x, y, z, \ldots, \in VAR\}$, and a set of storage assignments $A = \{X \Leftarrow e \mid X \in VAR, e \in EXP\}$. We further define a set of status signals as the logical relation between two expressions from the set EXP, $STAT = \{Rel(a, b) \mid a, b \in EXP\}$. Given these definitions, we define an FSMD as the quintuple

$$<S, I \times STAT, O \times A, f, h>$$

Here S is a set of states; we have extended the set of input values to include status expressions and the output set to include storage assignments. We define f and h as mappings of $S \times (I \times STAT) \rightarrow S$ and $S \times (I \times STAT) \rightarrow (O \times A)$ respectively. Thus, the FSMD's next state and outputs depend not only on the present state and the external signals but also on internal status signals that indicate whether a relation between two data path quantities is true or false.

The FSMD computes new values for variables stored in storage units in the data path. In addition, the FSMD assigns values of external signals. This is shown on the excerpt of the state diagram in Figure 5a, where the expression L_1 includes external and status variables, and L_2 includes variable assignments.

An FSMD is a universal model that represents all hardware designs. It can represent both control-dominated and data-dominated designs. A control-dominated design consists of a large control unit, possibly with a small data path. An example is a serial-to-parallel converter, which could have a single shift register as the data path. A data-dominated design, such as an FIR (finite impulse response) filter, has minimal control but a large data path to perform the filtering operations.

The FSMD model is usually implemented with a control unit and a data path; each state in the model corresponds to a clock cycle in the implementation. The control unit implements the FSM model, using a state register and two combinational logic circuits that compute the value of the state register for the next clock cycle (next-state function f) and the values of the output and control signals (output function h). Figure 5b shows these two combinational blocks as next-state logic and control logic. The data path implementation consists of a set of storage units (registers, counters, register files, memories), a set of functional units (ALUs, multipliers, shifters, comparators), and a set of interconnection units (wires, buses, multiplexers).

Originally published in *IEEE Design and Test of Computers,* Winter 1994 , pp. 44–54. Copyright © 1994 IEEE. All rights reserved.

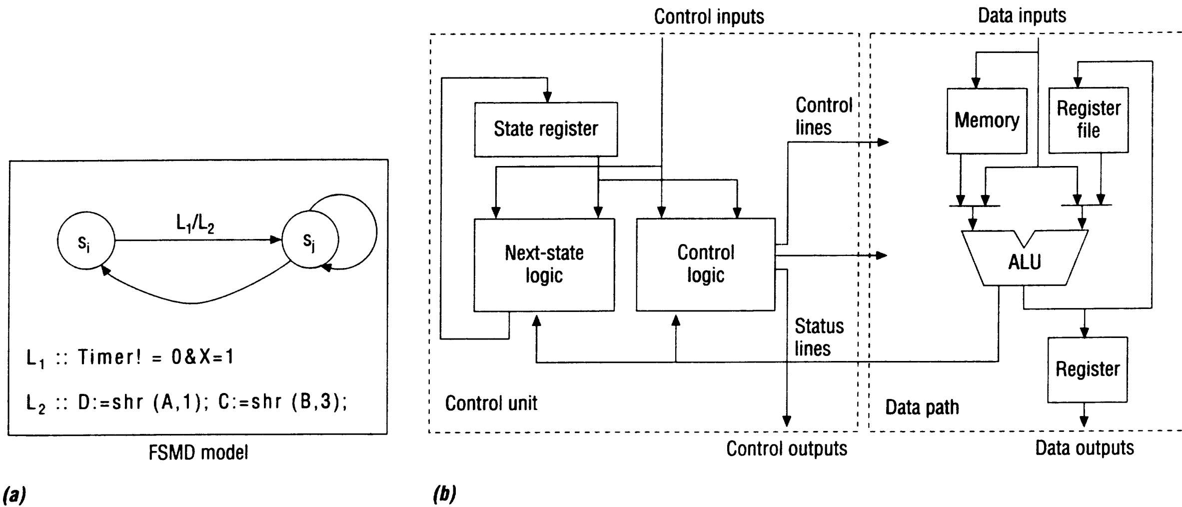

Figure 5. FSMD: model (a) and implementation (b).

Pipelining the FSMD implementation increases performance. Figure 6 symbolically illustrates the following three pipelining styles. Each changes the design's area and performance characteristics.

- *Component pipelining.* To increase the utilization of functional units within the data path, designers usually pipeline these units. Figure 6a shows an example in which the ALU is pipelined into two stages.
- *Control pipelining.* The FSMD model performs three tasks in each state: It computes control signals, it computes a new value for one or more variables stored in the storage units, and it computes the next state. Since these three tasks are repeated in each state, they can be pipelined into three stages. To achieve this pipelining effect, the control and the status lines must be latched as shown in Figure 6b. The FSM must be restructured to accommodate pipeline delay during branching operations.
- *Data path pipelining.* In signal-processing applications, the FSMD model performs the same sequence of operations on each element of an input data stream. Since these operations execute repeatedly in the data path, we can pipeline them as shown in Figure 6c. In this pipelining style, the control unit for each stage is usually very simple or nonexistent.

Synthesis tasks

High-level synthesis maps a behavioral description into the FSMD model so that the data path executes variable assignments and the control unit implements the control constructs. Since the FSMD model determines the amount of computation in each state, we must first define the number and type of resources (storage units, functional units, and interconnection units) to be used in the data path. Allocation is the task of defining necessary resources for a given design constraint.

The next task in mapping a behavioral description into an FSMD model is to partition the behavioral description into states (or control steps) so that the allocated resources can compute all the variable assignments in each state. This partitioning of behavior into time intervals is called scheduling.

Although scheduling assigns each operation to a particular state, it does not assign it to a particular component. To obtain the proper implementation, we assign each variable to a storage unit, each operation to a functional unit, and each transfer from I/0 ports to units and among units to an interconnection unit. This task is called binding (or resource sharing).

Binding defines the structure of the data path but not the structure of the control unit. The final task, control synthesis, consists of reducing and encoding states and deriving the logic network for next-state and control signals in the control unit. Control synthesis employs well-known logic synthesis and FSM synthesis techniques outside the scope of this article (see De Micheli[7]).

Allocation. The allocation task determines the type and quantity of resources used in the chip architecture. It also determines the clocking scheme, memory hierarchy, and pipelining style.

Originally published in *IEEE Design and Test of Computers,* Winter 1994 , pp. 44–54. Copyright © 1994 IEEE. All rights reserved.

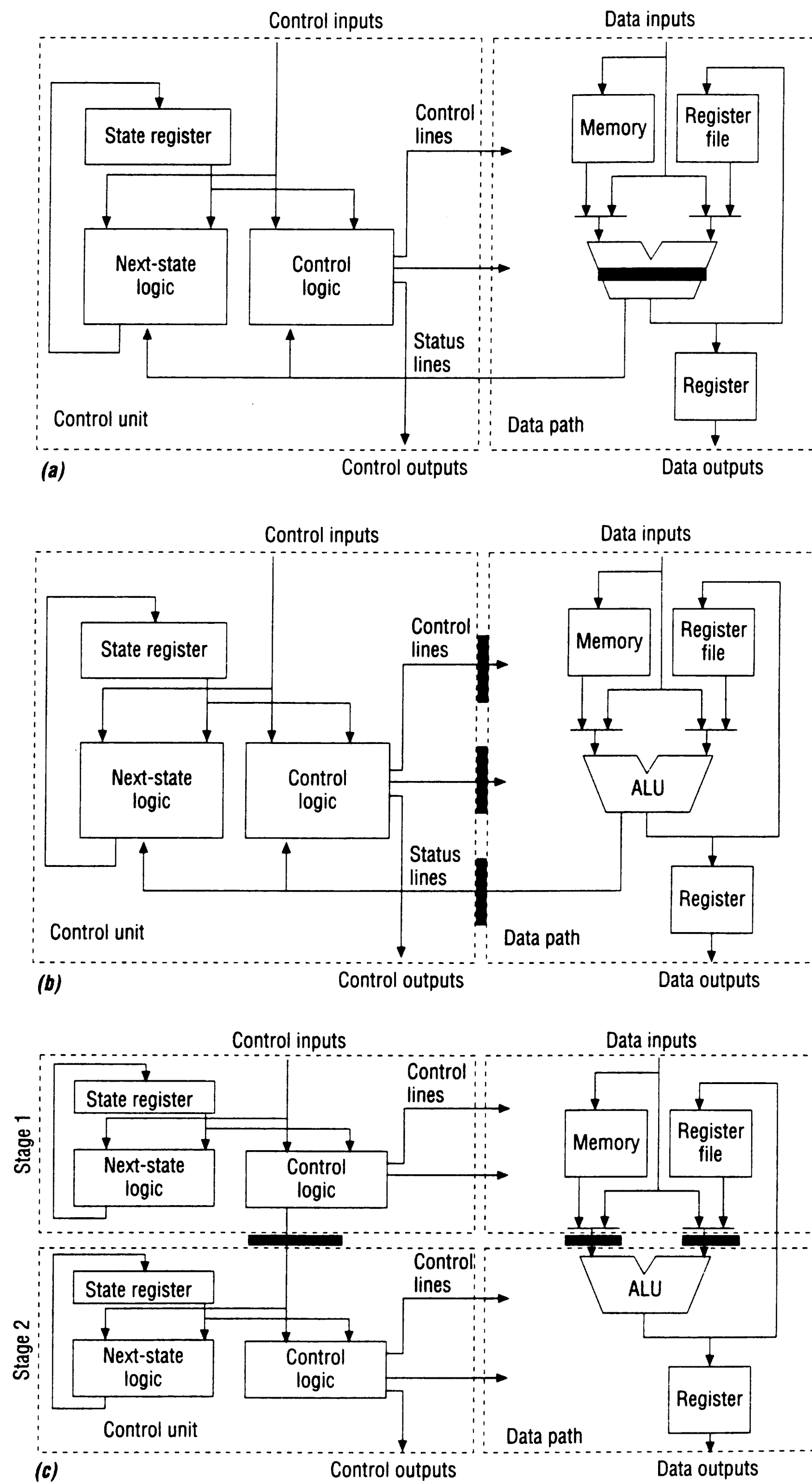

Figure 6. Pipelined FSMD model: component pipelining (a), control pipelining (b), and data path pipelining (c).

.

 Copyright © 1994 IEEE. All rights reserved.

The goal of allocation is to make appropriate trade-offs between the design's cost and performance. If the original description contains inherent parallelism, allocating more hardware resources increases area and cost, but it also creates more opportunities for parallel operations or storage accesses, resulting in better performance. On the other hand, allocating fewer resources decreases area and cost, but it also forces operations to execute sequentially, resulting in poorer performance.

To perform the required trade-offs, allocation must determine the exact area and performance values. A simple approximation of cost and performance consists of the number of functional units and the number of control steps, respectively. We can arrive at a more accurate estimate by means of the physical models stored in the RTL library. Table 1 shows a library containing three components: two ALU implementations (ALU-F and ALU-S), which perform the shift, add, and subtract operations, and MAX, which performs the max operation. ALU-F is fast, capable of executing an operation in 20 ns; ALU-S takes 70 ns to complete the operation but is smaller and cheaper.

Table 1. An RTL Library

Components (operations)	Delay (ns)	Area (μm^2
ALU-F (+/-/shr)	**20**	**600**
ALU-S (+/-/shr)	**70**	**400**
MAX (max)	**80**	**800**

Using this library, we can obtain area and performance trade-offs for our earlier example. Table 2 shows five allocations of functional units for the DFG in Figure 3. A simple estimate of the area for each allocation choice consists of the sum of the areas of the individual library components. We can also estimate the performance for each allocation, as shown in Figure 7. The figure shows that allocation A consumes the smallest area but results in the worst performance. Allocation E produces the best performance but most expensive design. Our selection of one of the five allocations for further synthesis would depend on the criticality of the application.

Allocation	No. of ALU-F	No. of ALU-S	No. of MAX	Area (μm^2)
A	0	1	1	1,200
B	1	0	1	1,400
C	0	2	1	1,600
D	1	1	1	1,800
E	2	0	1	2,000

Table 2. Possible allocations for DfG in Figure 3.

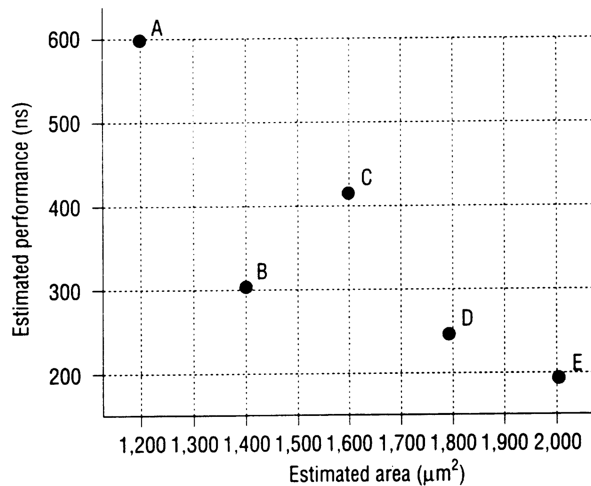

Figure 7. Area-performance trade-off curve for possible allocations listed in Table 2.

Originally published in *IEEE Design and Test of Computers,* Winter 1994 , pp. 44–54. Copyright © 1994 IEEE. All rights reserved.

We can compute similar curves to determine the trade-offs between performance and number of storage units, number of ports on each storage unit, and number of interconnection units. By selecting appropriate points on these curves, we determine the optimal number of resources of each type.

Instead of searching automatically through the large design space, most of today's high-level synthesis systems allow the user to allocate the type and mix of hardware resources. To help the designer make the right choice, the allocation tool must provide metrics that accurately reflect area, performance, clock slack, and resource usage.

In the future, allocation algorithms will require improvements for exploring more complex but more realistic architectural styles. To broaden the scope of high-level synthesis for different application domains, allocation algorithms supporting multiple pipelining styles, memory hierarchies, and interconnection topologies will be necessary.

Scheduling. The next step schedules operations and memory accesses into clock cycles. Scheduling algorithms are of two types, based on the optimization goal and the specified constraints. If the user has completely specified all the available resources and the clock cycle length during allocation, the scheduling algorithm's goal is to produce a design with the best possible performance, or the fewest clock cycles. In other words, scheduling must maximize usage of the allocated resources. We call this approach resource-constrained scheduling. If a list of resources is not available prior to scheduling, but a desired overall performance is specified, the scheduling algorithm's goal is to produce a design with the lowest possible cost, or the fewest functional units. This is the time-constrained scheduling approach.

Resource-constrained scheduling usually constructs the schedule one state at a time. It schedules operations so as not to exceed resource constraints or violate data dependencies. It ensures that at the instant for which it schedules an operation o_i into control step s_j a resource capable of executing o_i is available and all the predecessors of node o_i have been scheduled.

In the example described earlier, the allocation task determined that two fast ALUs (ALU-F) and one MAX unit are necessary to achieve an estimated performance of 200 ns. Let us attempt to schedule our example with this allocation (allocation E) and a clock cycle of 50 ns. The algorithm can schedule both shift operations (o_1 and o_2) in the first control step, since they are not dependent on any prior operations and two ALUs are available. When the algorithm schedules nodes in the second control step, it determines that both the addition operation o_3 and the subtraction operation o_4 can be scheduled, since two fast ALUs are available. The MAX node cannot be scheduled until its predecessor (operation o_4) computes the result, and hence it is scheduled in the next state. Figure 8 shows the final schedule.

In Figure 8, operations o_3 and o_4 are scheduled in the same state although they are dependent nodes. This process of scheduling two dependent nodes into the same state is called chaining; it is possible only if sufficient components are available and component delays are shorter than the clock cycle. In our example, the allocated ALUs have a delay of 20 ns and two ALUs are available, so we can chain the add and subtract operations operating under a clock period of 50 ns.

The opposite effect occurs if the unit delay is longer than the clock cycle. The scheduler must allow several states for the operations to complete. The MAX component has a delay of 80 ns, while the clock period is only 50 ns. Thus, operation o_5 requires two clock cycles to complete. Scheduling in which operations take more than one clock cycle to complete is called multicycling.

In time-constrained scheduling, the maximum number of control steps available for operations is fixed. Based on this performance constraint and the dependency constraints, we can compute the earliest control step e_i and the latest control step l_i into which a node o_i can be scheduled. Using the e_i and l_i bounds for all nodes, we can estimate the maximum number of functional units or the cost of the design. Time-constrained scheduling algorithms select a node o_i, evaluate the cost of scheduling it in each control step between e_j and l_i, and select the state s_j that results in the least cost. The important goal is to minimize the number of functional units in any time step.

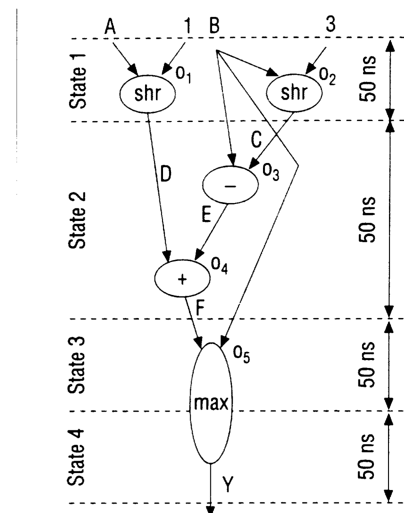

Figure 8. Example schedule based on allocation E.

Originally published in *IEEE Design and Test of Computers,* Winter 1994 , pp. 44–54. Copyright © 1994 IEEE. All rights reserved.

These basic scheduling algorithms become very complex in real-world situations. Actual libraries can have multiple implementations of the same component, each with its own area and delay characteristics. Scheduling must ensure that the design uses faster functional units for operations on the critical path and slower units for operations outside the critical path.

A great deal of research effort has focused on both resource-constrained and time-constrained scheduling. Several books on high-level synthesis provide details of the important work in scheduling.[5, 7–12] We hope to see this work extended to incorporate more realistic architectures, realistic libraries, data path and control pipelining, and memory hierarchy.

Binding. The binding task assigns the operations and memory accesses within each clock cycle to available hardware units. A resource such as a functional, storage, or interconnection unit can be shared by different operations, data accesses, or data transfers if they are mutually exclusive. For example, two operations assigned to two different control steps are mutually exclusive since they will never execute simultaneously; hence they can be bound to the same hardware unit.

Binding consists of three subtasks based on the unit type:

- *Storage binding* assigns variables to storage units. Storage units can be of many types, including registers, register files, and memory units. Two variables that are not alive simultaneously in a given state can be assigned to the same register. Two variables that are not accessed simultaneously in a given state can be assigned to the same port of a register file or memory.
- *Functional-unit binding* assigns each operation in a control step to a functional unit. A functional unit or a pipeline stage can execute only one operation per clock cycle.
- *Interconnection binding* assigns an interconnection unit such as a multiplexer or bus for each data transfer among ports, functional units, and storage units.

Although listed separately here, the three subtasks are intertwined and must be carried out concurrently for optimal results.

We illustrate the binding process with the example scheduled in Figure 8. To bind the variables onto registers, we must partition all the variables in the description into compatible sets. A set of variables is compatible if all the variables in the set are not alive at the same time. To determine the compatible sets, we must determine the lifetimes of the variables, as shown in Figure 9a. From the lifetimes, we create a compatibility graph, in which each node represents a variable, and an edge connects two variables with mutually exclusive lifetimes. In our example, variables *C* and *D* are written in state 1 and read in state 2, variable *E* is written and read in state 2, and variable *F* is written in state 2 and read in states 3 and 4. Figure 9b shows the compatibility graph for these variables.

Next, the compatibility graph must be partitioned into cliques. A clique is a fully connected subgraph—in other words, a subgraph containing several nodes, each node connected to all its neighbors. A clique indicates a set of mutually exclusive nodes that can be bound to the same resource.

For our example, a clique partition of the compatibility graph results in two possible solutions, shown in Figure 9c. Both solutions require two registers for storing the four variables. The first solution uses register Rl to store variables *C* and *F* and register R2 to store variable *D*. Variable *E* does not require storage because it is not alive over a state boundary and is implemented with a wire connection.

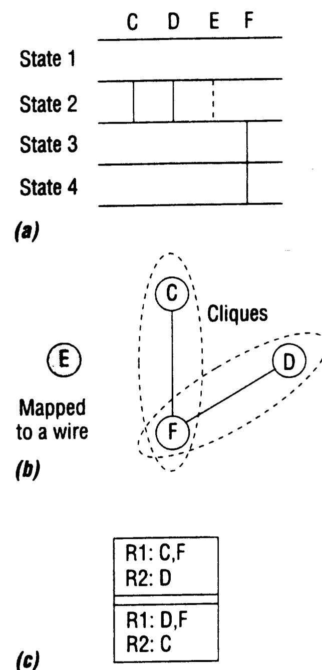

Figure 9. Storage binding: variable lifetimes (a), compatibility graph (b), and possible solutions (c).

Originally published in *IEEE Design and Test of Computers,* Winter 1994 , pp. 44–54. Copyright © 1994 IEEE. All rights reserved.

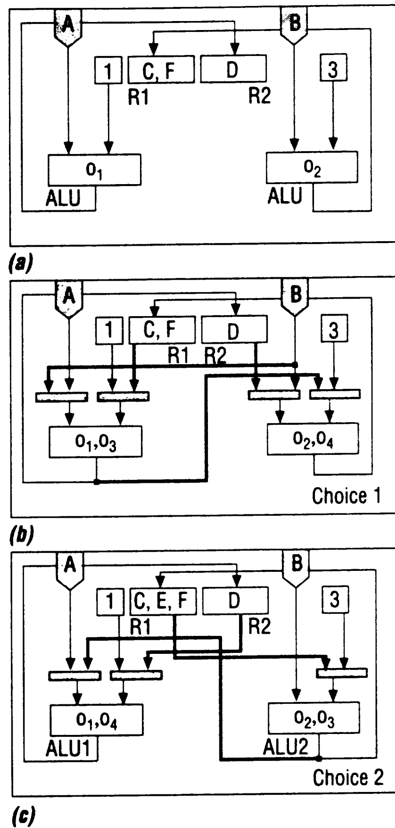

Figure 10. Functional-unit binding: partial design after binding operations in state 1 (a); state 2, choice 1 (b); and state 2, choice 2 (c).

The next task in the binding process is functional-unit binding, which assigns the operations in each state to the allocated functional units. We can construct a solution to the functional-unit binding problem one state at a time. The first state provides two binding choices, since both operations o_1 and o_2 can be bound to either of the available ALUs. Assume that o_1 is bound to the first and o_2 is bound to the second. Figure 10a shows the partial design after binding of the operations in the first state.

The binding process continues for operations in the second state. Since state 2 has two operations, o_3 and o_4, and both ALUs are capable of executing both operations, we have two different choices: bind o_3 to ALU1 and o_4 to ALU2 or vice versa. Figure 10b shows the resulting partial design for the first choice. New connections are shown in bold lines. The design requires four additional two-input multiplexers or eight new tristate drivers to complete the partial binding of states 1 and 2. Figure 10c shows the result of the second choice: Three additional multiplexers or six tristate drivers are sufficient because the connection from B to the second ALU is reused. Thus, the second choice leads to a more optimal design.

However, this binding algorithm uses a "greedy" approach and could lead to suboptimal solutions. The high-level synthesis books detail other binding approaches.[5, 7–12]

Binding can reduce the design's wiring area by storing variables in regular structures such as register files or n-port memories instead of in distributed registers. The algorithm achieves this by mapping all the scalar variables on a scheduled flow graph into a minimal set of register files based on the variables' access patterns.

If array variables are present in the description, simple binding algorithms proceed by mapping each array variable into a separate memory module of the same dimensions as the array variable. This approach is unacceptable because it leads to inefficient designs. A more efficient approach is to cluster many array variables into a single memory module. This leads to a solution with fewer memory modules and fewer ports; the reduction in ports reduces the size of the memory modules. However, the clustering of array variables requires an address translation for array access, thereby degrading performance. Researchers must develop algorithms capable of binding variables into a hierarchical memory organization containing caches, register files, and memories.

Design methodology. In a typical design environment, high-level synthesis can be part of a top-down or a bottom-up design methodology. Following a top-down methodology, the designer synthesizes the behavioral description into an RTL netlist, using a generic library of parameterized components. Since the components are parameterized, they may not exist during the high-level synthesis phase. However, an estimate of the area and delay and a list of the control and data ports are sufficient to complete high-level synthesis. Each component in the synthesized RTL netlist is designed at a later time by means of logic synthesis and technology mapping.

The top-down methodology is suitable for gate arrays and standard-cell-based designs, which must be completely flattened to the gate-level structure during layout. After the flattening, the layout process may place the components of the flattened netlist in different parts of the design, making it difficult to predict accurately the design's area and delay.

In a bottom-up methodology, high-level synthesis uses a library of pre-designed components. Thus, the exact shape, size, and timing of components are known during high-level synthesis. High-level synthesis uses this information to predict various design parameters such as the clock period and floor plan. This methodol-

Originally published in *IEEE Design and Test of Computers,* Winter 1994 , pp. 44–54. Copyright © 1994 IEEE. All rights reserved.

ogy is useful for custom designs that combine predesigned components to build the system.

A mixture of the top down and bottom-up methodologies would be ideal in all design environments. The top-down methodology allows rapid exploration techniques during the early design-planning phases. The bottom-up methodology allows optimization during the final layout phases.

Future directions

With no limitations in sight on the growth of ASICs, designers need new design methods to cope with ASIC complexities. High-level synthesis deals with these complexities by allowing design at higher abstraction levels. Several tools available from universities and research organizations demonstrate the potential of high-level synthesis, especially for specialized applications such as signal and image processing.

Despite this potential, designers use high-level synthesis sporadically, owing mainly to insufficient designer training, the lack of a clearly defined methodology, and insufficient offerings from the design automation industry. Now, however, several vendors are developing a new generation of synthesis tools embodying the concepts discussed here. These tools should help make high-level synthesis an integral part of the design process.

Although high-level synthesis promises to reduce time to market and generate high-quality designs, it is not a panacea. For example, its use in microprocessor design may be limited because in specifying the instruction set and the architecture, the designer has already completed scheduling and binding. Also, in the highly competitive microprocessor industry, the use of automated CAD tools may not be efficient. Instead, manual optimization is necessary to extract every bit of performance from the design. Furthermore, microprocessors are heavily pipelined with instruction look-ahead, and many synthesis tools do not adequately handle such advanced architectural features.

Wide acceptance of high-level synthesis will depend on how well the remaining open problems are solved. Among the most important are the following:

- *Syntactic variance.* The results of high-level synthesis must not depend on the description style. Eliminating syntactic variance would ensure that designers who are not proficient in HDL modeling and who don't understand high-level synthesis algorithms obtain satisfactory designs from high-level synthesis tools.
- *Interactivity. A* lack of interactivity in high-level synthesis tools makes it difficult for designers to control the design process and produce the designs they want. The design community does not accept push-button solutions.
- *Libraries.* Synthesis tools must be capable of using a wide variety of user-defined libraries and cores. Without library transparency, synthesis algorithms must be tuned to a specific library, making tool maintenance difficult.
- *Interfaces to lower levels.* Designers must consider the layout implications of any architectural choices made by high-level synthesis. Thus, tools that enable interaction with logic and layout tools and that provide accurate estimation techniques would greatly improve the quality of synthesized designs.

The future tutorials in this series will discuss the principles of high-level synthesis in more detail and evaluate proposed solutions of its basic problems.

References

1. IEEE Standard VHDL Language Reference Manual, IEEE, New York, 1988.
2. D. Ku and G. De Micheli, "HardwareC—A Language for Hardware Design," Tech. Report CSL-TR-90419, Stanford Univ., Stanford, Calif., 1990.
3. D. Thomas and P. Moorby, *The Verilog Hardware Description Language*, Kluwer, Boston, 1991.
4. P. Hilfinger, "A High-Level Language and Silicon Compiler for Digital Signal Processing," *Proc. Custom Integrated Circuits Conf,* 1985, pp. 213-216.
5. D.E. Thomas, et al., *Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench*, Kluwer, Boston, 1990.
6. V. Chaiyakul, D.D. Gajski, and L. Ramachandran, "High-Level Transformations for Minimizing Syntactic Variances," *Proc. 30th Design Automation Conf.,* IEEE Computer Society Press, Los Alamitos, Calif., 1993, pp. 413-418.
7. G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, New York, 1994.
8. D.D. Gajski, et al., *High-Level Synthesis*, Kluwer, Boston, 1992.
9. J. Vanhoof, et al., *High-Level Synthesis for Real-Time Digital Signal Processing*, Kluwer, Boston, 1993.
10. R. Camposano and W. Wolf, *High-Level VLSI Synthesis*, Klwer, Boston, 1991 .
11. R. Walker and R. Camposano, *A Survey of High-Level Synthesis Systems*, Kluwer, Boston, 1991.
12. P. Michel, U. Lauther, and P. Duzy (eds.), *The Synthesis Approach to Digital System Design*, Kluwer. Boston, 1992.

Daniel D. Gajski is a professor in the Department of Computer Science at the University of California, Irvine. Previously, he was a professor in the Department of Computer Science at the University of Illinois in

Originally published in *IEEE Design and Test of Computers,* Winter 1994 , pp. 44–54. Copyright © 1994 IEEE. All rights reserved.

Urbana-Champaign. Still earlier, he worked in industry. His current interests are supercomputer architectures and programming tools. ASIC and system design methodology, high-level synthesis, and hardware-software codesign. Gajski served as technical chair of the High-Level Synthesis Workshop in 1992 and general chair of the High-Level Synthesis Symposium in 1994. He received the Dipl. Ing. and MS in electrical engineering from the University of Zagreb and the PhD in computer and information sciences from the University of Pennsylvania. He is a senior member of the Computer Society and a fellow of the IEEE.

Loganath Ramachandran is a design engineer in the Software R&D Department of LSI Logic Corporation, Milpitas, California. Previously, he worked as a software engineer in the Design Automation Division of Texas Instruments (India). His interests include system-level specification and synthesis, design methodology, high-level synthesis, and hardware-software codesign. He received his BTech in electrical engineering from the Indian Institute of Technology, Madras, and his MS and PhD in computer science from the University of California, Irvine, where he was a chancellor's fellow. He is a member of the IEEE.

Send correspondence about this article to Daniel D. Gajski, UC Irvine, Dept. of Computer Science, Irvine, CA 92717-3425; or gaiski@uci.edu.

Originally published in *IEEE Design and Test of Computers,* Winter 1994 , pp. 44–54. Copyright © 1994 IEEE. All rights reserved.