The High-Level Synthesis of Digital Systems

MICHAEL C. McFARLAND, MEMBER, IEEE, ALICE C. PARKER, SENIOR MEMBER, IEEE, AND RAUL CAMPOSANO

Invited Paper

High-level synthesis systems start with an abstract behavioral specification of a digital system and find a register-transfer level structure that realizes the given behavior. In this paper we examine the high-level synthesis task, showing how it can be decomposed into a number of distinct but not independent subtasks. Then we present the techniques that have been developed for solving those subtasks. Finally, we discuss those areas related to high-level synthesis that are still open problems.

I. Introduction

A. What is High-Level Synthesis?

The synthesis task is to take a specification of the behavior required of a system and a set of constraints and goals to be satisfied, and to find a structure that implements the behavior while satisfying the goals and constraints. By *behavior* we mean the way the system or its components interact with their environment, i.e., the mapping from inputs to outputs. *Structure* refers to the set of interconnected components that make up the system, typically described by a netlist. Ultimately, the structure must be mapped into a *physical* design, that is, a specification of how the system is actually to be built. Behavior, structure, and physical design are usually distinguished as the three *domains* in which hardware can be described.

Just as designs can be described at various levels of detail, so synthesis can take place at various levels of abstraction. The hierarchy of levels generally recognized as applicable to digital designs was first described by Bell and Newell [1], who applied it specifically to computer systems. An updated view of their hierarchy and the way it appears across the different domains is shown in Table 1. Similar schemes have appeared elsewhere in the literature [2]–[4].

At the top of their hierarchy is the so-called *PMS* level, where computer systems are described as interconnected sets of Processors, Memories, and Switches. Since this level is concerned with the overall system structure and information flow, we might also call it the *system* level. The next level is what Bell and Newell refer to as the *Instruction Set* level. At this level the focus is on the computations performed by an individual processor, the way it maps sequences of inputs to sequences of outputs. Since the term instruction set implies a computer-like architecture, while we are interested in a more general class of digital systems, we will refer to this as the *algorithmic* level.

Table 1. The Design Hierarchy

Level	DOMAINS		
	Behavior	Structure	Physical
PMS (System)	Communicating Processes	Processors Memories Switches	Cabinets Cables
Instruction Set (Algorithm)	Input-Output	Memory, Ports Processors	Board Floorplan
Register-Transfer	Register Transfers	ALUs, Regs Muxes, Bus	ICs Macro Cells
Logic	Logic Equations	Gates Flip flops	Standard Cell Layout
Circuit	Network Equations	Transistors, Connections	Transistor Layout

Below the algorithmic level is the *register-transfer* (RT) level. There the system is viewed as a set of interconnected storage elements and functional blocks. The behavior is described as a series of data transfers and transformations between storage elements. The difference between the register-transfer and the algorithmic level is the level of detail with which the internal structure is specified. At the algorithmic level, the variables in the description do not necessarily correspond to the internal registers of the design, nor do assignment statements correspond to actual register transfers; only the input/output behavior is considered fixed.

Next comes the *logic* level, where the system is described as a network of gates and flip-flops and the behavior is specified by logic equations. Below that is the *circuit* level, which views the system in terms of the individual transistors of which it is composed. Finally, one can go down one level

Manuscript received April 3, 1989; revised August 2, 1989.

M. C. McFarland is with the Department of Computer Science, Boston College, Chestnut Hill, MA 02167, USA.

A. C. Parker is with the Department of Electrical Engineering Systems, University of Southern California, Los Angeles, CA 90089-0781, USA.

R. Camposano is with the Thomas J. Watson Research Center, IBM Research Division, Yorktown Heights, NY 10598, USA.

IEEE Log Number 9034099.

0018-9219/90/0200-0301$01.00 © 1990 IEEE

further to the *device* level, where the focus is on the internal structure and behavior of the transistors.

High-level synthesis, as we use the term, means going from an algorithmic level specification of the behavior of a digital system to a register-transfer level structure that implements that behavior. The input specification gives the required mappings from sequences of inputs to sequences of outputs, where those inputs and outputs may communicate with the outside environment or with another system-level component. The specification should constrain the internal structure of the system to be designed as little as possible. From the input specification, the synthesis system produces a description of a *data path*, that is, a network of registers, functional units, multiplexers, and buses. If the control is not integrated into the data path, and it usually is not, the synthesis system must also produce the specification of the control part. In synchronous systems, the only kind we consider in this paper, control can be provided by one or more finite state machines, specified in terms of microcode, PLA profiles or random logic.

Usually there are many different structures that can be used to realize a given behavior. One of the tasks of synthesis is to find the structure that best meets the constraints, such as limitations on cycle time, area, or power, while minimizing other costs. For example, the goal might be to minimize area while achieving a certain required processing rate.

High-level synthesis as we define it must be distinguished from other types of synthesis that operate at different levels of the design hierarchy. For example, high-level synthesis is not to be confused with *register-transfer* level synthesis, where the registers and functional units in the design and the data transfer between them are already largely, or even completely, specified. Much less is it equivalent to *logic* synthesis, where the system is specified in terms of logic equations, which must be optimized and mapped into a given technology. Register-transfer and/or logic synthesis might in fact be used after high-level synthesis.

At the other end of the hierarchy, there is some promising work under way on *system level synthesis*, sometimes called architectural synthesis. One example of a system-level issue is partitioning an algorithm into multiple processes that can run in parallel or be pipelined. This work, however, is still in its preliminary stages, and we will not focus on it here.

B. Why Study High-level Synthesis?

In recent years there has been a trend toward automating synthesis at higher and higher levels of the design hierarchy. Logic synthesis is gaining acceptance in industry, and there has been considerable interest shown in register-transfer level synthesis. There are a number of reasons for this:

- **Shorter design cycle.** If more of the design process is automated, a company can complete a design faster, and thus have a better chance of hitting the market window for that design. Furthermore, since much of the cost of the chip is in design development, automating more of that process can lower the cost significantly.
- **Fewer errors.** If the synthesis process can be verified to be correct—by no means a trivial task—there is a greater assurance that the final design will correspond to the initial specification. This means fewer errors and less debugging time for new chips.
- **The ability to search the design space.** A good synthesis system can produce several designs from the same specification in a reasonable amount of time. This allows the developer to explore different tradeoffs between cost, speed, power etc., or to take an existing design and produce a functionally equivalent one that is faster or less expensive. Even if the design is ultimately produced manually, automatically synthesized designs can suggest tradeoffs to the designer.
- **Documenting the design process.** An automated system can keep track of what design decisions were made and why, and what the effect of those decisions was.
- **Availability of IC technology to more people.** As more design expertise is moved into the synthesis system, it becomes easier for one who is not an expert to produce a chip that meets a given set of specifications.

We expect this trend toward higher levels of synthesis to continue. Already there are a number of research groups working on high-level synthesis, and a great deal of progress has been made in finding good techniques for optimization and for exploring design tradeoffs. These techniques are very important because decisions made at the algorithmic level tend to have a much greater impact on the design than those at lower levels.

C. History

The roots of high-level synthesis can be traced back to the 1960s. For example, the ALERT system developed at the IBM T. J. Watson Research Center took a register-transfer level behavioral description, written in APL, and produced a logic level implementation. This early system already addressed aspects of scheduling (called sequence analysis in ALERT) and storage allocation (called automatic identification of Flip-flops) [5]. A complete IBM 1800 computer was synthesized automatically, requiring, however, more than twice as many components as used in the manual design [6]. Another example was the synthesis system of Duley and Dietmeyer, which translated register-transfer descriptions written in DDL into logic [7].

During the 1970s design automation developed explosively, but most of the effort went into automating tasks at lower levels of the design hierarchy, such as layout. High-level synthesis was still considered an academic exercise, and research was restricted mainly to universities. Nevertheless, great progress was made in the development of algorithms and techniques. In the early seventies at Carnegie-Mellon University, the Expl system was the first to explore the space of possible designs by performing series-parallel tradeoffs. It took as input a description of the register-transfer level behavior of the system to be designed, specified in the ISPL hardware description language, and built a register-transfer level structure for it. Expl used a limited set of predesigned register-transfer modules for the implementation, which simplified the synthesis process enormously.

In the late seventies, researchers at Carnegie-Mellon produced a top-down design system, the CMUDA system, which input a design specification written in ISPS (a more general and powerful successor to ISPL) and output either a CMOS standard cell or TTL implementation [8]. This sys-

tem addressed the entire range of high-level synthesis tasks listed in Section II. The CMU group experimented with a wide variety of algorithms for data path synthesis, ranging from brute-force and heuristic methods to more sophisticated polynomial-time algorithms, expert systems, and mathematical programming.

In Europe, the first effort known to the authors was MIMOLA from the University of Kiel, which started as early as 1976 [9]. MIMOLA synthesized a CPU and microcode from an input specification, iterating under user control until constraints were met. The system was ported to Honeywell in the late seventies, where it is now being used as part of a VHDL synthesis system [10]. The CADDY/DSL system, another early synthesis effort in Europe, has been under development at Karlsruhe since 1979 [11].

In the last decade, work on high-level synthesis has proliferated extensively, and is also starting to spread from the academic community to industry. High-level synthesis systems are now producing manufacturable chip designs, although the quality of these designs still seems inferior to manual ones. Research has also expanded to encompass a more diverse set of design styles and applications, including signal processing [12], pipelined processors [13], and interfaces [14], [15]. Some of the more prominent ongoing projects are those at Carnegie-Mellon University [16]–[18], at the University of Southern California [2], [19], at Carleton University [20], at the University of California at Irvine [21], at Karlsruhe University [22], at the University of Kiel [23], at AT&T Bell Laboratories [24], [25], and at the IBM T. J. Watson Research Center [26].

There is now a sizable body of knowledge on high-level synthesis, although for the most part it has not yet been systematized. In the remainder of this paper, we will describe what the problems are in high-level synthesis, and what techniques have been developed to solve them. Since this is a tutorial and not a survey, we certainly will not be able to do justice to all the work that has gone on in the field. The main objective is to help groups that want to get involved in high-level synthesis to establish a firm base from which to conduct their own explorations. To that end, Section II will describe the various tasks involved in developing a register-transfer level structure from an algorithmic level specification. Section III will describe the basic techniques that have been developed for performing those tasks. Finally, Section IV will look at those issues that have not been adequately addressed and thus provide promising areas for future research.

II. The Synthesis Task

A. Task Definition

The system to be designed is usually represented at the algorithmic level by a programming language such as Pascal [27] or Ada [14], or by a hardware description language that is similar to a programming language, such as ISPS [28], DSL [22], MIMOLA [29] or behavioral VHDL [30]. Most high-level synthesis approaches have used *procedural* languages. That is, they describe data manipulation in terms of assignments to variables that keep their values until they are overwritten. Statements are organized into larger blocks using standard control constructs for sequential execution, conditional execution, and iteration. Modularity is achieved by the use of hierarchically organized subprograms (procedures). There have also been experiments with various types of nonprocedural hardware description languages, including applicative, LISP-like languages [31], and "logic" languages such as Prolog [32]. The input specification can describe a simple algorithm such as a nonrecursive digital filter, a more complex algorithm with conditional branches, and loops like a floating-point arithmetic processor, or the instruction set of a large CPU, such as the IBM 370.

A realistic specification language should contain a mechanism to specify hierarchy, usually procedures, and a way of specifying concurrent tasks. These two decompositions should be regarded only as a specification aid, however, since the logical decomposition that is helpful for understanding the behavior is seldom the same as a good decomposition for the hardware. Therefore the synthesis system should partition the design into procedures and concurrent tasks according to hardware design criteria. This early planning and partitioning is one of the most difficult problems in high-level synthesis, one that must be considered at the system level. One problem which is particularly difficult is the decomposition of an algorithm into concurrent modules. For example, automatic synthesis of an asynchronous pipeline from a sequential processor description, including all the control necessary for pipeline stalls, the bypassing buses, etc., is beyond today's capabilities. Such an automatic pipeline synthesis would require not only the processor specification but also extensive instruction traces in order to be able to evaluate the pipeline. In practice, therefore, the specified decomposition into concurrent processes is used as given. Most current synthesis systems are also not capable of changing the hierarchy implicit in the specification. They simply synthesize each procedure separately or flatten the hierarchy.

The first step in high-level synthesis is usually the compilation of the formal language into an internal representation. Most approaches use graph-based representations that contain both the data flow and the control flow implied by the specification, although parse trees are also used [33].

To illustrate this and subsequent steps in the synthesis process, we present a simple example. It should be noted, incidentally, that the example has been kept simple for the purposes of illustration and is not representative of the large and complex input descriptions that many synthesis systems can handle. The example, shown in Fig. 1, is part of a program that computes the square-root of X using Newton's method. The number of iterations necessary in practice is very small. In the example, 4 iterations were chosen. A first degree minimax polynomial approximation for the interval $[\frac{1}{16}, 1]$ gives the initial value.

The graphical representation of the algorithm is also shown in the figure, with the data flow and control flow graphs shown separately for intelligibility. The control graph is derived directly from the explicit order given in the program and from the compiler's choice of how to parse the arithmetic expressions. The data flow graph shows the essential ordering of operations implied by the data dependencies in the specification. For example, in Fig. 1, the addition at the top of the diagram depends for its input on data produced by the multiplication. This implies that the multiplication must be done first. On the other hand, there is no data dependence between the $I + 1$ operation inside the loop and any of the operations in the chain that calculates Y, even though they are ordered in the control graph shown.

```
. . .
Y := 0.222222 + 0.888889 * X;
I := 0;
DO UNTIL I > 3 LOOP
  Y := 0.5 * ( Y + X / Y );
  I := I + 1;
ENDDO;
. . .
```

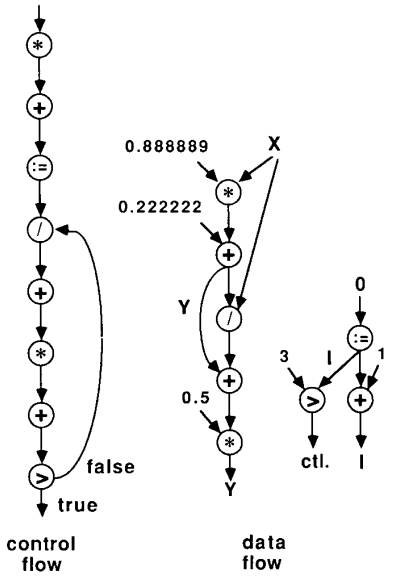


Fig. 1. High-Level specification and internal representation for sqrt.

Therefore the $I + 1$ may be done in parallel with those operations, as well as before or after them. The data flow graph can also be used to remove the dependence on the way internal variables are used in the specification, since each value produced by one operation and consumed by another is represented uniquely by an arc. This variable disambiguation, which is similar to what is done during global flow analysis in compilers [34], is important both for reordering operations and for simplifying the data paths.

It would seem that this "direct" compilation is essentially a one-to-one translation of the program into the internal representation. Nevertheless, there are several important tasks that should be performed by the compiler at this stage and that may turn "direct" compilation into a laborious task. Examples are the already mentioned variable disambiguation, taking care of the scope of variables across procedures, converting complex data structures into simple types, type checking, and converting data types where this is required. It is desirable that the input language have all of the power and comfort of a modern high-level programming language, which means that the complier must be intelligent enough to handle procedures, complex data structures, and a variety of data types and operations.

Some optimizing transformations, such as expression simplification, may be done at this stage also. In Fig. 1, for example, $0.22 + 0.88 * X$ might be rewritten as $0.22 * (1 + 4 * X)$, which is simpler to implement, since the multiplication by four can be done by a shift, "picking the right wires," thus using only one multiplication and an increment instead of a multiplication and an addition.

There is a wide variation in the types of graphs used as internal representations. In many systems, the control and data flow graphs are integrated into one structure. Examples include the Value Trace (VT) [35], [36], used by the CMUDA system, the CDFG used by the HAL system [20], and the YIF used by the Yorktown Silicon compiler [37]. The DDS of the ADAM synthesis system [38], on the other hand, maintains separate graphs for data flow and control and timing, with bindings to indicate the correspondence between elements of the two graphs. There are also differences in how much information is kept from the specification. Some control graphs, for example, do not include all the dependencies in the program, only the essential ones like conditional branches; and some data flow graphs, such as the VT, do not use the variable assignments defined by the specification in defining essential orderings of operations, while others, such as the YIF, do.

The rest of this section outlines the various steps used in turning the intermediate form into an RT-level structure, using the square root example to illustrate the different steps.

Since the specification has been written for human readability and not for direct translation into hardware, it is desirable to do some initial optimization of the internal representation. These high-level transformations [39] include such compiler-like optimizations as dead code elimination, constant propagation, common subexpression elimination, in-line expansion of procedures, and loop unrolling. Local transformations, including those that are more specific to hardware, are also used. In the example, the loop-ending criterion can be changed to $I = 0$ using a two-bit variable for I. The multiplication times 0.5 can be replaced by a right shift by one, which can be done by discarding the rightmost bit. The addition of 1 to I can be replaced by an increment operation. The internal representation after these optimizations is depicted in Fig. 2(a). Loop unrolling can also be done in this case since the number of iterations is fixed and small (Fig. 2(b)).

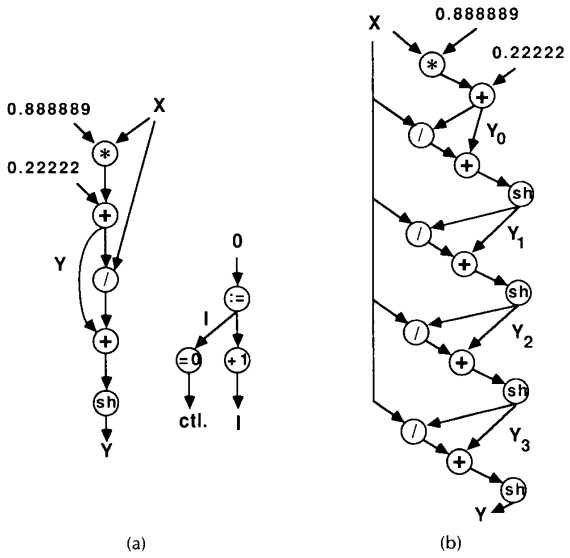


Fig. 2. Internal representation. (a) After some optimizations. (b) After loop unrolling.

The DSL/CADDY system does some high-level optimization as part of the synthesis process [22], as does the MIMOLA system [23] and the HERCULES system [33]. The System Architect's Workbench [40] contains a number of high-level transformations, although most of these must be applied interactively. The Flamel system [27] has a somewhat more limited set of transformations, but it applies them automatically, using a branch-and-bound algorithm that is very effective, at least for small hardware descriptions.

The next two steps in synthesis are the core of transforming behavior into structure: scheduling and allocation. They are closely interrelated. Scheduling involves assigning the operations to so-called control steps. A control step is the fundamental sequencing unit in synchronous systems; it corresponds to a clock cycle. Allocation involves assigning the operations and values to hardware, i.e., providing functional units, storage and communication paths, and specifying their usage. Different methods for scheduling and allocation will be examined in detail in the next section. Here we simply describe what they do and how they fit into the overall synthesis task.

The aim of *scheduling* is to assign operations to control steps so as to minimize a given objective function while meeting constraints. The objective function may include, among other things, the number of control steps, delay, power, and hardware resources. In our example, a straightforward, nonoptimized schedule uses just one functional unit and one single-port memory (Fig. 3(a)). Each operation has to be scheduled in a different control step, so the computation takes $3 + 4 * 5 = 23$ control steps. If instead of the memory, single registers for Y and I and an extra register T are provided, then the operations $I := 0$ and *shift* do not need a control step any more, and the computation takes only $2 + 4 * 4 = 18$ steps (Fig. 3(b)). To further speed up the computation, the operations in the control graph can be scheduled with maximal parallelism, packing them into control steps as tightly as possible, observing only the essential dependencies required by the data flow graph and by the loop boundaries. This form is shown for the example in Figure 3(c). Notice that two dummy nodes to delimit the loop boundaries were introduced. Since the shift operation is free, with two functional units the operations can now

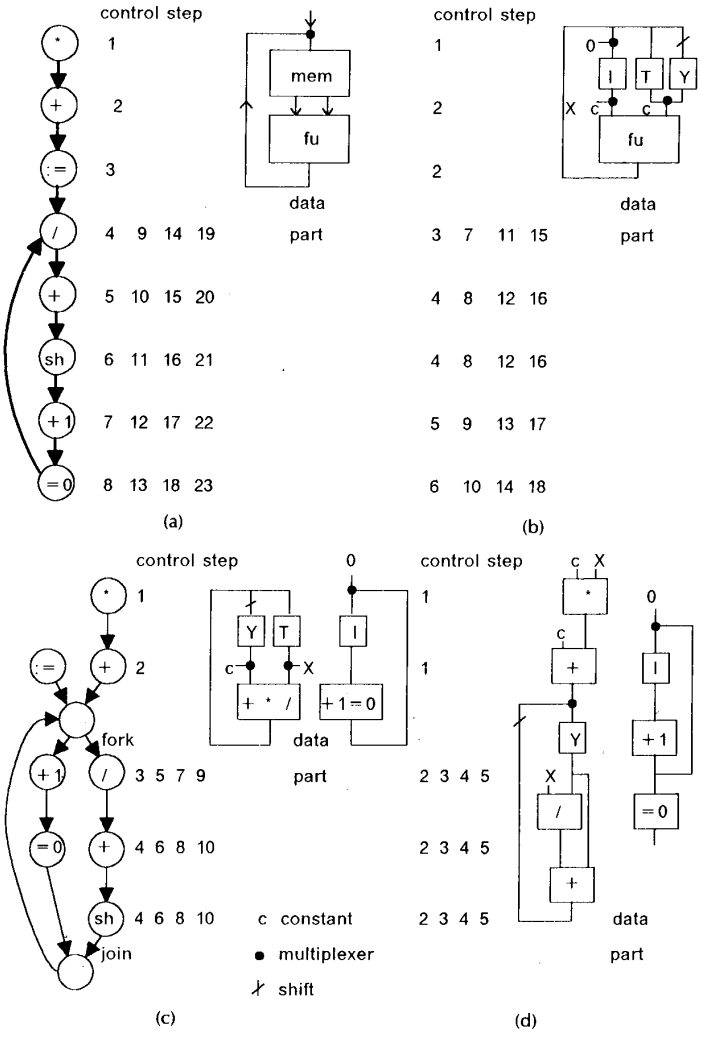


Fig. 3. Some possible schedules and data-path allocations.

be scheduled in 2 + 4 * 2 = 10 control steps. The number of control steps can be still further reduced by executing ordered operations in the same control step, (often called "chaining"). In our example, the initializations can be performed in one control step, and then each iteration in another control step, so the computation takes only 5 control steps. (Fig. 3(d)). In the extreme case, with no constraints on the hardware and the use of loop unrolling, the whole computation can be done in combinational logic taking just one control step or part of it. Notice that the tradeoff is not only between the number of control steps and the amount of hardware to be allocated, but also involves the length of each control step. As fewer control steps are used to exploit operation chaining, the length of a control step, i.e., the number of operations to be done serially in combinational logic, will increase and thus the cycle time will increase.

This discussion has assumed the nonoverlapped execution of tasks, i.e., that the operation is not pipelined. The techniques described here can be extended to cover pipeline design, and some specific techniques for pipeline design exist [13].

The clocking scheme assumed in most scheduling algorithms is a one-phase clock with master-slave registers. If two or more clock phases are necessary, scheduling is further complicated.

In *allocation*, the problem is to minimize the amount of hardware needed. The hardware consists essentially of functional units, memory elements, and communication paths. To minimize them together is usually too complex, so in many systems they are minimized separately, although in general this may lead to suboptimal results. As an example, consider the minimization of functional units. Operations can share functional units only if they are mutually exclusive, that is, they are assigned to different control steps. The problem is then to form groups of mutually exclusive operations in such a way that the number of groups is minimized. Since each group will require its own functional unit, this will minimize the number of functional units. This kind of allocation is sometimes called "folding." The grouping of operations is also affected by the capabilities of functional units. For example, if there is no functional unit that can both add and multiply, additions and multiplications must be kept in separate groups. The allocation of functional units for the schedules given in Figures 3(a), 3(b), and (3(c), is minimal in this sense. In the allocation of Fig. 3(d) this is not the case. Since the two adders are never used at the same time, a data path with only one adder would be sufficient; but this would complicate the communication path, requiring additional multiplexers.

The problems of minimizing the amount of storage and the complexity of the communication paths for a given schedule can be formulated similarly.

In storage allocation, values that are generated in one control step and used in another must be assigned to registers. Values may be assigned to the same register when their lifetimes do not overlap. Storage assignment should be done in a way that not only minimizes the number of registers, but also simplifies the communication paths. The latter criterion is illustrated by the program sequence

a1 := a1 **and** b ; c := **not** a1;

where the ";" indicates sequential execution. A value is assigned to the variable **a1**, which already has an initial value, so that **a1** has more than one source. Hence it should be stored in a register with an input multiplexer. Of course this cumbersome solution is unnecessary: duplicating **a1**, the program becomes

a2 := a1 **and** b ; c := **not** a2;

Storage allocation is not necessarily limited to single registers; multi-port memories are often much more efficient solutions [41].

Communications paths, including buses and multiplexers, must be chosen so that the functional units and registers are connected as necessary to support the data transfers required by the specification and the schedule. The simplest type of communication path allocation is based only on multiplexers. Buses, which can be seen as distributed multiplexers, offer the advantage of requiring less wiring, but they may be slower than multiplexers. Depending on the application, a combination of both may be the best solution.

Conditional branches and loops further complicate both scheduling and allocation. If pairs of conditional values or operations are mutually exclusive, that is, if they are in separate branches so that they can never exist at the same time, they may share resources. The ability of a synthesis program to detect and use this information is very important, since it can lead to a much more efficient use of resources. Many allocators, including HAL, MAHA, Sehwa, the DAA, and the YSC mark operations and values that are mutually exclusive and use this information in allocation.

A formal model of scheduling and allocation has been defined by Hafer and Parker [42]. This model provides formal rules expressing the design tradeoffs, constraints and costs (excluding interconnect hardware) for synthesizing a data path from a behavioral description.

In addition to designing the abstract structure of the data path, the system must decide how each component of the data path is to be implemented. This is sometimes called *module binding* [43]. For the binding of functional units, known components such as adders can be taken from a hardware library. Libraries facilitate the synthesis process and the size/timing estimation, but they can prevent efficient solutions that require special hardware. The synthesis of special-purpose full-custom hardware is possible, but it makes the design process more expensive by requiring extensive use of logic synthesis and possibly layout synthesis.

Some systems do preliminary module selection before scheduling and allocation. This provides more information on the costs and delays associated with various operations, which can help the scheduling and allocation algorithms produce more accurate results. The BUD system [44], for example, selects modules for its functional clusters as they are formed. Jain *et al.* have implemented a method for selecting a set of candidate modules for a design by estimating the area-time tradeoff curves for several candidate module sets and finding the one that gives the best set of design points. This is optimal only for pipelined designs [45], but it has been applied to nonpipelined designs as well [46].

Once the schedule and the data paths have been chosen, it is necessary to synthesize a controller that will drive the data paths as required by the schedule. The synthesis of the

control hardware itself can be done in different ways. If hardwired control is chosen, a control step corresponds to a state in the controlling finite state machine. Once the inputs and outputs to the FSM, that is, the interface to the data part, have been determined as part of the allocation, the FSM can be synthesized using known methods, including state encoding and optimization of the combinational logic [47], [48]. If microcoded control is chosen instead, a control step corresponds to a microprogram step and the microprogram can be optimized using encoding techniques for the microcontrol word [49], [50].

Finally, the design has to be converted into real hardware. Lower level tools such as logic synthesis and layout synthesis complete the design.

B. The Design Space

As we have seen, there are many different designs that implement a given specification. If we disregard the designs that are clearly inferior, the remaining designs represent different tradeoffs between area and performance. The plot of area versus processing time for these designs forms what is sometimes called the *design space*. Figure 4 shows an

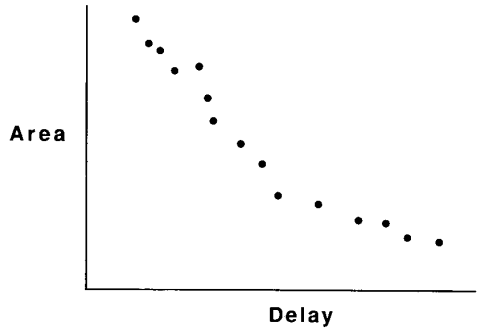


Fig. 4. The design space.

idealized view of the register-transfer level design space for a particular specification.

High-performance designs, found close to the area axis, are generally parallel implementations of the algorithm. Inexpensive designs, which are spread along the time axis, use less hardware and thus are usually slower. For pipelined designs, the cost-performance tradeoff curve is regular and well-characterized. More hardware allows more operations to be done at once, so that the throughput is higher. For nonpipelined designs, the space seems to be divided into two regions. One region consists of designs with few time steps and longer clock cycles, due to operator chaining. Designs in the other region all have clock cycles with the same length, equal to the delay of the slowest operation, but with more control steps as the designs become less expensive and require more resource sharing.

C. Problem Complexity

There are a number of problems that designers of synthesis software encounter. The major problem is the extremely large number of design possibilities that must be examined in order to select the design that meets constraints and is as near as possible to the optimal design. There are many factors that account for the differences between the various designs, and it is hard even to find a canonical set of operators that systematically search through all those designs. Furthermore, the shape of the design space is often problem-specific, so that there is no methodology that is guaranteed to work in all cases.

Finding the best solution to even a limited problem such as scheduling is difficult enough. Many synthesis subtasks, including scheduling with a limitation on the number of resources and register allocation given a fixed number of registers, are known to be NP-hard [51]. That means that the process of finding an optimal solution to these is believed to require a number of steps that is at least exponential in the problem size. Yet in high-level synthesis there are several such tasks, and they cannot really be isolated, since they are interdependent.

Another problem is the difficulty of evaluating designs that have not been implemented and manufactured. Area is not really known until the design is laid out, because it is not until the layout is done that the area due to interconnections and unused area is known. Similarly the speed of the design is hard to determine until delays are actually measured, and this cannot be done accurately until the layout is done and all the path delays are known. There is, in fact, evidence that when all of the factors that affect area and delay are taken into account, the design space becomes much less regular [52].

III. Basic Techniques

A. Scheduling

We distinguish three dimensions along which scheduling algorithms may differ: (1) the objective function and constraints that they use; (2) the interaction between scheduling and data path allocation; and (3) the type of scheduling algorithm used.

1) Objectives and constraints:

An objective is a measure that the design system seeks to maximize or minimize. A constraint is a condition that must be met. For example, if the goal is to design a system so as to minimize area while holding the cycle time below 1 microsecond, minimizing area is an objective, while the condition that cycle time is less than or equal to 1 microsecond is a constraint. Almost all systems define their objectives and constraints in terms of area (or some other measure of the amount of hardware) and time. Many systems, such as the CMUDA system [8] and the MIMOLA system [23], [29], try to minimize the number of control steps subject to area constraints. Others, such as the HAL system [53], seek to minimize area given certain time constraints. There are some systems, such as MAHA [19] and Shewa [13], that can do either. The BUD system [44], uses a combination of area and time as an objective.

Area is measured in different ways. Until recently, most systems simply counted the number of functional blocks, and perhaps registers, to get an estimate of area. Lately, several systems, including Elf [14] and Chippe [54], have added an estimate of multiplexing cost to the area measure. BUD does a rough floorplan of each design it evaluates and from that estimates total area, thus taking into account layout and wiring space as well as active area.

Time is usually taken to be the time required to process

one set of input data, or the time needed to execute one major cycle of the machine, that is, the average number of control steps per cycle times the delay per control step. For a pipeline synthesis system like Sehwa, throughput is a better measure. For interfaces and communications circuits, on the other hand, meeting global time constraints is not enough; it is necessary to meet local constraints. For example, there might be a requirement that certain data be calculated and placed on an output port within 150 ns of the time that a certain input signal is received. The Elf system, the ISYN interface synthesizer [55], and the Janus system [15] can not only measure those local constraints, but also use them to guide the scheduling.

There are many other factors that are important in evaluating designs, such as pin limitations, package selection, testability, variety of latches, library of cells, power dissipation, clock skew, etc., [56]. These should be taken into account during high-level synthesis, but at present there is no system that does so.

2) Interaction with Allocation:

As noted earlier, scheduling and operator allocation are interdependent tasks. In order to know whether two operations can be scheduled in the same control step, one must know whether they use common hardware resources—for example, the same functional unit. Moreover, finding the most efficient possible schedule for the real hardware requires knowing the delays for the different operations, and those can only be found after the details of the functional units and their interconnections are known. On the other hand, in order to make a good judgement about how many functional units should be used and how operations ought to be distributed among them, one must know what operations will be done in parallel, which comes from the schedule. Thus there is a vicious circle, since each task depends on the outcome of the other.

A number of approaches to this problem have been taken by synthesis systems. The most straightforward one is to set some limit (or no limit) on the number or total cost of functional units available and then to schedule. This is done, for example, in the CMUDA system [57], [16], in the early Design Automation Assistant [58], in the Flamel system [27], and in the *V* system [59]. This limit could be set as a default by the program or specified by the user. A somewhat more flexible version of this approach is to iterate the whole process, first choosing a resource limit, then scheduling, then changing the limit based on the results of the scheduling, rescheduling and so on until a satisfactory design has been found. This is done, for example, under user control in the MIMOLA system and under guidance of an expert system, with feedback from the data path allocator, in Chippe. The Sehwa pipeline synthesis system uses this general strategy, looking at different allocations of functional units and finding the best schedule for each. This search is guided by mathematical models that relate area and performance, and thus is much more efficient than a naive search.

Another approach is to develop the schedule and resource requirements simultaneously. For example, the force-directed scheduling in the HAL system schedules operations within a given time constraint so as to balance the load on the functional units, and thus to meet the given time constraints while using as few functional units as possible. The MAHA system uses a similar procedure. HAL also includes a feedback loop that allows the scheduling to be repeated with more precise contraints after the detailed data paths have been designed, when more is known about delays and interconnect costs. Two more recent approaches have formulated scheduling and allocation together as an optimization problem to be solved by general optimization techniques, in one case by simulated annealing [60] and in the other by integer programming [61].

The Yorktown Silicon Compiler (YSC) [37] does allocation and scheduling together, but in a different way. It begins with each operation being done on a separate functional unit and all operations being done in the same control step. Additional control steps are added for loop boundaries, and as required to avoid conflicts over register and memory usage. The hardware is then optimized so as to share resources as much as possible. If there is too much hardware or there are too many operations chained together in the same control step, more control steps are added and the data path structure is again optimized. This process is repeated until the hardware and time constraints are met.

Finally, functional unit allocation can be done first, followed by scheduling. In the BUD system, the operations to be performed by the hardware are first partitioned into clusters, using a metric that takes into account potential functional unit sharing, interconnect, and parallelism. Then functional units are assigned to each cluster and the scheduling is done. The number of clusters to be used is determined by searching through a range of possible clusterings, choosing the one that best meets the design objectives.

In the CADDY/DSL system, the data path is built first, assuming maximal parallelism. This is then optimized, locally and globally, guided by both area constraints and timing. Global optimizations include mapping different operations onto the same functional unit, unrolling small loops, sharing registers, etc. Local optimization relies on rules very much like local logic optimization. The operations are then scheduled, subject to the constraints imposed by the data path.

3) Scheduling Algorithms:

There are two basic classes of scheduling algorithms: transformational and iterative/constructive. A transformational type of algorithm begins with a default schedule, usually either maximally serial or maximally parallel, and applies transformations to it to obtain other schedules. The fundamental transformations are moving serial operations, or blocks of operations, in parallel and the inverse, moving parallel operations in series (Fig. 5). Transformational algorithms differ in how they choose what transformations to apply.

Expl [62] used exhaustive search. That is, it tried all possible combinations of serial and parallel transformations and chose the best design found. This method has the advantage that it looks through all possible designs, but of course it is computationally very expensive and not practical for sizable designs. Exhaustive search can be improved somewhat by using branch-and-bound techniques, which cut off the search along any path that can be recognized to be suboptimal. That was the approach used implicitly by Hafer and Parker [42] when they used integer-linear programming to solve a set of equations that modeled the scheduling and allocation problem. This technique was important in that it showed the power of this formal model,

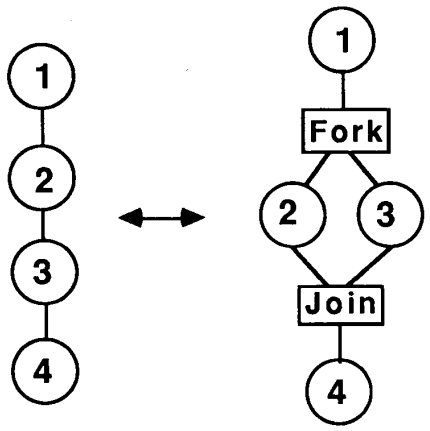


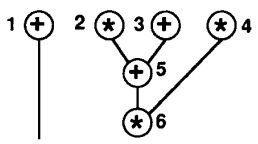
Fig. 5. Basic scheduling transformations.

but it too proved to be impractical for all but the simplest designs.

Another approach to scheduling by transformation is to use heuristics to guide the process. Transformations are chosen that promise to move the design closer to the given constraints or to optimize the objective. This is the approach used, for example, in the Yorktown Silicon Compiler, as described above, and the CAMAD design system [63]. The transformations used in the YSC can be shown to produce a fastest possible schedule, in terms of control steps, for a given specification.

The other class of algorithms, the iterative/constructive ones, build up a schedule by adding operations one at a time until all the operations have been scheduled. They differ in how the next operation to be scheduled is chosen and in how they determine where to schedule each operation.

The simplest type of scheduling, as soon as possible (ASAP) scheduling, is local both in the selection of the next operation to be scheduled and in where it is placed. ASAP scheduling assumes that the number of functional units has already been specified. Operations are first sorted topologically; that is, if operation x_j is constrained to follow operation x_i by some necessary dataflow or control relationship, then x_j will follow x_i in the topological order. Operations are taken one at a time in this order and each is put into the earliest control step possible, given its dependence on other operations and the limits on resource usage. Figure 6 shows



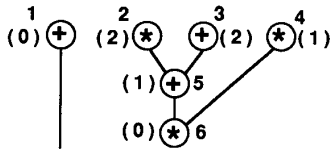
Schedule

Step	*	+
1	2	1
2	4	3
3		5
4	6	

Fig. 6. ASAP scheduling.

a dataflow graph and its ASAP schedule. This was the type of scheduling used in the early CMUDA schedulers, in the MIMOLA system and in Flamel. The problem with this algorithm is that no priority is given to operations on the critical path, so that when there are limits on resource usage, operations that are less critical can be scheduled first on limited resources and thus block critical operations. This is shown in Fig. 6, where operation 1 is scheduled ahead of operation 3, which is on the critical path, so that 3 is scheduled later than is necessary, forcing a longer than optimal schedule.

List scheduling overcomes this problem by using a more global criterion for selecting the next operation to be scheduled. For each control step to be scheduled, the operations that are available to be scheduled into that control step, that is, those whose predecessors have already been scheduled, are kept in a list, ordered by some priority function. Each operation on the list is taken in turn and is scheduled if the resources it needs are still free in that step; otherwise it is deferred to the next step. When no more operations can be scheduled, the algorithm moves to the next control step, the available operations are found and ordered, and the process is repeated. This continues until all the operations have been scheduled. Studies have shown that this form of scheduling works nearly as well as branch-and-bound scheduling in microcode optimization [64]. Figure 7 shows



Schedule

Step	*	+
1	2	3
2	4	5
3	6	1

Fig. 7. A list schedule.

a list schedule for the graph in Fig. 6. Here the priority, shown in parentheses next to each node, is the length of the path from the operation to the end of the block. Since operation 3 has a higher priority than operation 1, it is scheduled first, giving an optimal schedule for this case.

A number of schedulers use list scheduling, though they differ somewhat in the priority function they use. BUD uses the length of the path from the operation to the end of the block it is in. Elf and ISYN use the "urgency" of an operation, the length of the shortest path from that operation to the nearest local constraint. The HAL system can do list scheduling with *force* as a priority, a concept that will be explained below.

Freedom-based scheduling, an example of which is MAHA, is another type of scheduling that is global in the way it selects the next operation to be scheduled. The range of possible control step assignments for each operation is first calculated from the time constraints and the precedence relations between the operations. The operations on the critical path, those with the tightest constraints on them, are scheduled first and assigned to functional units. Then the other operations are scheduled and assigned one at a time. At each step the unscheduled operation with the least freedom, that is, the one with the smallest range of control steps into which it can go, is chosen. Thus operations that might present more difficult scheduling problems are taken care of first, before they become blocked.

The last type of scheduling algorithm we will consider is global both in the way it selects the next operation to be scheduled and in the way it decides the control step in which to put it. An example of this is the force-directed scheduling used in HAL. In force-directed scheduling, the guiding factor both for choosing which operation to schedule next and for choosing where to schedule it is the so-called *force* on

each operation. The force between an operation and a particular control step is proportional to the number of operations of the same type that could go in that control step. Thus scheduling so as to minimize force tends to balance the use of functional units, producing a schedule that minimizes the number of resources needed to meet a given time constraint.

To calculate the force for a dataflow graph, a *distribution graph* is first set up for each set of operations that could share a functional unit. The distribution graph shows, for each control step, how heavily loaded that step is, given that all possible schedules are equally likely. The distribution graph is calculated by finding the earliest and latest control step in which each operation could be done, given the time constraints and the precedence relations between the operations. This is called the mobility for that operation, and is similar to the freedom used in freedom-based scheduling. If an operation could be done in any of k control steps, then $1/k$ is added to each of those control steps in the graph. For example Fig. 8 shows a dataflow graph, the range of steps

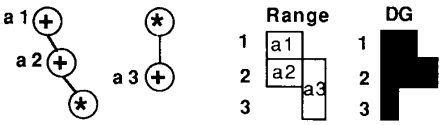


Fig. 8. A distribution graph.

for each operation, and the corresponding distribution graph for the addition operations, assuming a time constraint of three control steps. Addition $a1$ must be scheduled in step 1, so it contributes 1 to that step. Similarly addition $a2$ adds 1 to control step 2. Addition $a3$ could be scheduled in either step 2 or step 3, so it contributes $\frac{1}{2}$ to each.

For each possible assignment of an operation x to a control step i, a force is calculated as

$$F(i) = \sum_j DG(j) \times x(i, j)$$

where $DG(j)$ is the value of the distribution graph at control step j and $x(i, j)$ is the change of x's probability in control step j if x is scheduled in control step i. For example, in Fig. 8, the force involved in assigning $a3$ to step 2 is $1\frac{1}{2} \times \frac{1}{2} + \frac{1}{2} \times (-\frac{1}{2})$. The first term comes from the change involved in adding $a3$ to step two, since its probability goes from $\frac{1}{2}$ to 1, while the second term comes from removing a possible assignment to step 3, causing the probability to go from $\frac{1}{2}$ to 0. This positive force indicates that $a3$ should not go into step 2, since it is already heavily loaded. Other forces are added that reflect the effect of scheduling an operation on its predecessors and successors, since once an operation is scheduled it changes the mobilities of its neighbors.

Once all the forces are calculated, the operation-control step pair with the largest negative force (or least positive force) is scheduled. In the above example, $a3$ would first be scheduled into step 3. The distribution graph and forces are updated and the process is repeated.

Force-directed scheduling is more expensive computationally than list scheduling. Force-directed scheduling has complexity $O(cN^2)$ versus $O(cN \log N)$ for list scheduling, where c is the number of control steps and N is the number of operations. (c is equal to N in the worst case.) The advantage of force-directed scheduling is that it tends to balance the distribution graph, so that a minimum amount of hardware is used to achieve a given time constraint.

All of the scheduling algorithms mentioned above, except for Expl and Sehwa, use heuristics to guide the search for a good schedule. None of these is guaranteed to find the best possible schedule. Nevertheless many of them do give good results in practice.

B. Data Path Allocation

Data path allocation involves mapping operations onto operators, assigning values to registers, and providing interconnections between operators and registers using buses and multiplexers. The decision to use complex functional units (e.g., ALUs) instead of simple operators is also made at this time. The optimization goal is usually to minimize some objective function, such as

- total interconnect length.
- total operator, register, bus driver and multiplexer cost, or
- critical path delays.

There may also be constraints on the design which limit total area of the design, total throughput, or delay from input to output. As with scheduling, allocation programs must search the design space efficiently or reduce the size of the search space.

The techniques that perform data path allocation can be classified into two types: iterative/constructive, and global. Iterative/constructive techniques assign elements one at a time, while global techniques find simultaneous solutions to a number of assignments at a time. Exhaustive search is an extreme case of a global solution technique. Iterative/Constructive techniques generally look at less of the search space than global techniques, and therefore are more efficient, but are less likely to find optimal solutions.

1) Iterative/Constructive Techniques:

Iterative/constructive techniques select an operation, value or interconnection to be assigned, make the assignment, and then iterate. The rules which determine the next operation, value or interconnect to be selected can vary from global rules, which examine many or all items before selecting one, to local selection rules, which select the items in a fixed order, usually as they occur in the dataflow graph from inputs to outputs. Global selection involves selecting a candidate for assignment on the basis of some metric, for example taking the candidate that would add the minimum additional cost to the design.

The data path allocator used in the early CMUDA system was iterative, and used local selection [57]. The DAA uses a local criterion to select which element to assign next, but chooses where to assign it on the basis of rules that encode expert knowledge about the data path design of microprocessors. EMUCS [65] uses a global selection criterion, based on minimizing both the number of functional units and registers and the multiplexing needed, to choose the next element to assign and where to assign it. The Elf system also seeks to minimize interconnect, but uses a local selection criterion. The REAL program [66] separates out register allocation and performs it after scheduling, but prior to operator and interconnect allocation. REAL is constructive,

and selects the earliest value to assign at each step, sharing registers among values whenever possible. REAL performs a global analysis only once to sort the values.

An example of greedy allocation is shown in Fig. 9. The graph is processed from the earliest time step to the latest.

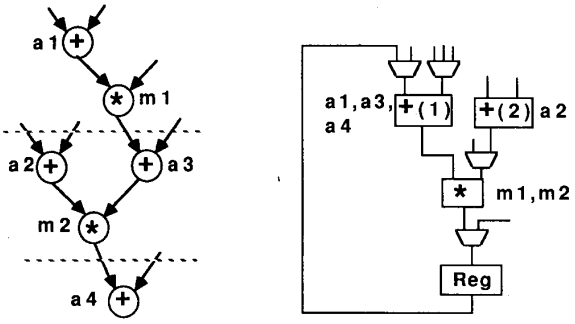


Fig. 9. Greedy data-path allocation.

Operators, registers, and interconnect are allocated for each time step in sequence. Thus, the selection rule is local, and the allocation constructive. Assignments are made so as to minimize interconnect. In the case shown in the figure, *a*2 is assigned to adder2, since the increase in multiplexing cost required by that allocation is zero. *a*4 is assigned to adder1 because there is already a connection between adder1 and the register storing one of the inputs to *a*4. Other variations are possible, each with different multiplexing costs. For example, if we assigned *a*2 to adder1 and *a*4 to adder1 without checking for interconnection costs, then the final multiplexing would be more expensive. Or we could assign *a*3 before *a*2, and achieve a different multiplexer configuration, but one which happens to have the same cost as our original design. Finally, a more global selection rule could be applied. For example, we could select the next item for allocation so as to minimize the cost increase. In this case, if we already allocated *a*3 to adder2, then the next step would be to allocate *a*4 to the same adder, since they occur in different time steps, and the incremental cost of doing that assignment is less than assigning *a*2 to adder1.

2) Global Allocation:

Global allocation techniques include graph theoretic formulations, branch and bound algorithms, and mathematical programming techniques. One popular graph theoretic formulation, used for example in Facet [16], involves creating graphs in which the elements to be assigned to hardware, whether they are operations, values, or interconnections, are represented by nodes, and there is an arc between two nodes if and only if the corresponding elements can share the same hardware. The problem then becomes one of finding sets of nodes in the graph, all of whose members are connected to one another, since all of the elements in such a set can share the same hardware without conflict. This is the so-called clique finding problem. If the objective is to minimize the number of hardware units, then we would want to find the minimal number of cliques that cover the graph, or, to put it another way, to find the maximal cliques in the graph. Unfortunately, finding the maximal cliques in a graph is an NP-hard problem,

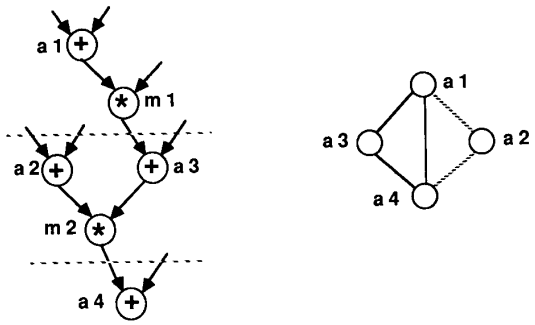


Fig. 10. Example of a clique.

so in practice, heuristics are often employed. Figure 10 shows the graph of operations from the example shown in Fig. 9. One clique is highlighted, showing that the three operations can share the same adder, just as in the greedy example.

The branch-and-bound technique uses search to find a good or optimal solution to the allocation problem. Potentially a branch-and-bound algorithm can look through all possible configurations for the data path by trying all possible allocations of operations to processors, values to registers, and connections to buses and multiplexers. This makes it very powerful, since it can reach every possible solution. By the same token, however, it can be very expensive, since the number of possible configurations is exponential or worse in the number of elements to be allocated. Branch-and-bound is more efficient than a naive search, since it keeps track of the best solution so far and cuts off the search of any portion of the solution space as soon as it recognizes that that cannot improve on the current best solution; but it is still potentially exponential. If a full search is too expensive, heuristics can be used to limit the number of possible solutions tried; but then the outcome is no longer guaranteed to be optimal. The Splicer program [67] and the MIMOLA system use a branch-and-bound search to do allocation. They include heuristics that can be used to limit the search, thus trading off processing time for solution quality.

Formulation of allocation as a mathematical programming problem involves creating a variable for each possible assignment of an operation, variable, or interconnection to a hardware element. The variable is one if the assignment is made and zero if it is not. Constraints must be formulated that guarantee that each operation must be assigned to one and only one hardware element, and so on. The objective then is to find a valid solution that minimizes some cost function. Finding an optimal solution requires exhaustive search, which is very expensive. This is done by Hafer and Parker on a small example [42], and recent research by Hafer indicates that heuristics can be used to reduce the search space, so that larger examples can be considered.

C. Alternative Approaches

The main problem in high-level synthesis, as noted earlier, is the size and irregularity of the search space. One way around this is to use expert knowledge to inform and guide the search and thus to help find good solutions.

The Design Automation Assistant [58] was the first rule-based expert system that performed data path synthesis.

The prototype DAA just encoded in its rules a standard scheme [68]. The system was improved over a period of time, however, by having experts critique the designs it produced. The knowledge gained from these critiques was entered into the system in the form of new rules. As a result, the designs produced by the new system were much closer to what an expert would produce [69].

One problem with the expert system approach is that existing expert systems are very slow. It takes orders of magnitude more processing time to achieve results comparable to those obtained by some of the nonoptimal algorithms mentioned above.

If the area of application is restricted to a particular domain, more domain-specific knowledge can be used. Each domain has its own paradigms and methods. For example, the digital signal process domain has been explored by several groups. The CATHEDRAL system [12] and the FACE system [70] are examples of very successful efforts in that area. Other programs have been able to get very good results by focusing on microprocessor design, for example the SUGAR system [71], [72]. The synthesis of data paths for synchronous pipelines is a design domain that has now been given a foundation in theory [13] and an implementation in the program Sehwa.

IV. Future Directions

There are a number of areas where high-level synthesis must continue to develop if it is to become a useful tool in designing VLSI systems. Some of these are specific design problems that still need to be solved. Others are more general issues that involve integrating synthesis into the larger context of a design system.

A. Open Design Problems

1) Design style selection:

Design Style Selection refers to the selection of a basic architecture for a design and a module set for implementing it, based on measurements made on the initial specification and perhpas on further input from the user. This should be done early in the synthesis process, before scheduling and allocation. Thomas and Siewiorek [73] showed that by measuring certain parameters in the specification, it was possible to predict the structure that human designers would choose for the design. Little has been done, however, to follow up on his work. The System Architect's Workbench [18] does have the ability to use different allocators depending on the user's choice of design style. Jain and Parker recently described a technique for design style selection between pipelined and nonpipelined implementation, which is partially automated in the ADAM system [46]. With the need to limit the search space in a synthesis system, and with the success of special-purpose synthesis systems that assume a certain design style tailored to a specific domain, it does seem that it would be beneficial to make an early decision about design style and use that to choose a specific design methodology.

2) Interface design:

One area where the use of high-level synthesis seems most attractive is in the design of interface and communications circuits. There are many such circuits needed, they are relatively low volume, and often cutting design time is more important than saving area or achieving higher performance. Yet most synthesis systems cannot handle that type of circuit because at best they can only meet global timing constraints. That is, they can design hardware that does a specified set of computations within a given amount of time or a given number of clock cycles. Interface circuits, on the other hand, must satisfy local constraints on the timing of input/output events. For example, a bus interface specification might contain the requirement that after a ready signal is received, the interface must place data on the data lines and set an acknowledge flag. The specification might also require that there be a delay of at least 75 ns between the time the data is output and the time the acknowledge flag is set and a delay of no more than 100 μs between the detection of the ready signal and the setting of the acknowledge flag. The input languages to most synthesis systems cannot even express such constraints, and the synthesis algorithms themselves are not set up to handle them.

There has, however, been some recent work on synthesis with local timing constraints. The HAL and ELF systems can handle some forms of local timing constraints. The ability to enter local timing constraints, check them for consistency, and use them in automatic synthesis has been explored in the CADDY synthesis system [74]. The Janus system [15] uses a graphical interface to allow the user to specify detailed timing constaints on the inputs and outputs of a system. These specifications are translated directly into interface logic. At a higher level of specification, the ISYN system [55], allows local timing constraints to be added to an ISPS description. It uses these constraints to set priorities in a modified list scheduler, thus ensuring that input/output events are scheduled so as to meet the constraints. CONSPEC [75] uses local timing constraints in the generation of a state machine controller.

This is a very promising area for further research. More work needs to be done, especially in integrating the interfaces with the rest of the system being designed [76].

3) System-level synthesis:

There are a number of important decisions and tradeoffs that current systems do not explore. These include trading off complexity between the control and the data paths, partitioning the control, breaking a system into interacting asynchronous processes, breaking a memory array into separate modules, collecting a set of registers into a register file, and changing the bitwidth of the data path in order to increase or decrease parallelism. Many of these are system-level issues. That is, they relate to the number and configuration of system-level components such as memories, processors, and controllers, rather than to how those individual pieces are designed. To date, very little has been done to provide design aids at this level, although Walker and Thomas have explored some system-level transformations [40], and Lagnese and Thomas have looked at system-level partitioning [77].

4) High level transformations:

As noted earlier, high level transformations on the input specification or its internal representation are an important step in the synthesis process. In particular, transformations that alter the control flow, such as the in-line expansion of procedures, moving operations out of conditionals, changing conditionals into data selects, and so on, can have a large

impact on the speed of the final design. Transformations have been classified and studied, [39], [40], and, as mentioned previously, many systems now apply some optimizations before scheduling and allocation. But when to apply them and in what order is still an open problem. There is no system that takes advantage of the full range of known transformations. Trickey found an effective way of ordering and searching through a limited number of transformations using a branch and bound algorithm [27]; but it is not clear that that method would generalize to a richer and less orderly set of transformations.

B. General Issues

1) Human Factors:

Human factors refers to the place of the designer in the design process. Certainly the human designer has an important role to play in the synthesis process, a role which goes far beyond writing the initial specification and constraints. There are certain optimizations that a human can recognize that no algorithm can find; and there are conditions, constraints, and goals that emerge only as the design process proceeds. Thus the user should have as much or as little control over the synthesis process as he or she wants. How best to achieve that is not yet known. It is simple enough to allow the user to intervene and make some of the individual decisions that the synthesis algorithms normally make, such as applying a particular optimizing transformation, scheduling an operation in a specific time-step, or allocating a register to hold certain values. This is not, however, the level at which the human designer interacts best with the system. Our experience is that with a large specification, there is too much detail for the human to be able to see these individual possibilities and understand their impact on the overall design. Humans seem to be better at seeing patterns and grasping the overall shape of things and in formulating high-level strategies than at systematically working through a series of detailed steps. New ways must be found to present the user with summary information on the specification and partial design, to allow the user to focus on different areas of the design while at the same time seeing how the parts are interrelated, to help the user guide the synthesis process without making each individual decision, and to allow the user to explore the implications of various design decisions. The synthesis system should also be able to explain the decisions it makes to the user so that the user can evaluate and, if necessary, change them.

Some synthesis systems, such as EMUCS, and DAA, and HAL, allow the user to bind certain decisions and then invoke the system to complete the design. User interaction can be used to guide the search in Sehwa. MIMOLA supports user interaction, particularly in restricting resources. The System Architect's Workbench supports user transformations on a graphical representation of behavior [18]. The CORAL system [78], which is also part of the System Architect's Workbench, displays in a graphical form the connections between the behavioral specification, the synthesized data path, and the control specification. These efforts, however, are only the beginning of the research needed on this aspect of synthesis.

There is still more work to be done on system specification as well. A full specification has many aspects, requiring different modes of specification. Certainly the procedural specification, which gives the sequences of actions to be performed, is important. Some aspects of the behavior, however, are best stated in declarative form: "When Y happens, X should be done," as in the PHRAN-SPAN system [79]. Other requirements on the design are best expressed as constraints, for example on the timing of certain events, on the size of the resulting hardware or of various parts of it, or on the power and other electrical characteristics. Somehow all of these aspects must be integrated into a consistent and coherent specification, and the synthesis system must be able to digest and use all of the information contained therein. There also remains the question of how a designer can most easily develop a specification. Should it be done graphically? Should there be an interactive, "intelligent" editor, whether textual or graphical? If a language is used, should it be a known programming language, an existing hardware description language, a natural language, or something entirely new? Is an applicative language better than a procedural one? What data types and structures should be supported? And so on. There are strongly held prejudices on these questions, but little solid basis for choosing one way or the other.

2) Design Verification:

Design verification involves showing that the synthesized design has the behavior required by the specification. This checking is normally done for manual designs by simulating both the initial specification and the final design on the same sets of inputs, and making sure that the outputs correspond. Especially now that there are languages that allow both behavioral and structural descriptions of hardware, such as VHDL, MIMOLA, and Verilog, it should be simple enough to do the same for a synthesis system, simulating both the initial behavioral specification and the synthesized structure. Furthermore, since the synthesis system actually derives the final design from the specification in a well-defined manner, it can establish links between the two, which can be used to compare the two descriptions and verify their correspondence. The CORAL system does some of this, for example. Little work has been reported on verifying the output of synthesis systems, however. More of this needs to be done in order to check the soundness of the synthesis algorithms and their implementations. When an early version of the CMUDA system was used to design a PDP-8 minicomputer and a computer was actually built according to that design and was tested, it exposed a number of flaws in the synthesis programs [80].

The MIMOLA system is one synthesis system that does do verification by simulating both the behavioral specification and the synthesized output. In addition, a retargetable microcode generator has been used to verify the data paths that have been synthesized by the system, and possibly modified manually [81]. It does this by attempting to generate microcode to execute the specified behavior on the data paths. If the code generator fails, it is an indication that the data path does not support the required behavior. A somewhat different approach is taken by the RLEXT system [82]. It allows the user to modify the data paths and schedule as desired, then automatically fixes them so that they are once again correct with respect to the specification.

There is also a growing body of work on the formal ver-

ification of hardware designs [83], [84]. This means constructing a formal proof that the behavior of the final design is consistent with the behavior implied by the specification. As with simulation, the fact that the synthesis system can maintain links between the specification and the design can help make construction of the proofs easier.

Part of the attractiveness of a synthesis system, however, is that it offers possibilities for verification other than just checking the output design against the specifications. Since the synthesis system always follows the same procedures, it is claimed, once those procedures themselves are verified, the designs produced are guaranteed always to be correct. There is no longer any need to verify them individually. The hope, then, is to be able to prove that the output of the synthesis system always has the same behavior as the input specification. This can be done in theory by viewing the action of the synthesis system as a series of transformations on a hardware description.

The initial description is the specification of the system to be designed. The final description, which is produced by applying a series of transformations, is the synthesized structure. If it can be shown that each transformation preserves the behavior of the description to which it is applied, then it follows that the synthesized structure will always have the same behavior as the initial specification. This is a promising application for formal verification techniques because the proofs only have to be done once, rather than for every design. This would justify the enormous investment of time and effort required to carry out any formal proof of a nontrivial system.

McFarland and Parker [85] have used formal methods to verify that a number of the optimizing transformations used at the beginning of the synthesis process are correctness-preserving. Beyond that, however, very little has been done. There are some formidable problems to be overcome before much more progress can be made on verification. For one, what does it mean to say that two descriptions have the "same behavior"? How is that behavior represented in the first place, and how is it modeled in a formal system? These problems become especially difficult when the descriptions being compared are at different levels of abstraction and in different domains, e.g., structural versus behavioral. And what about timing? It is difficult enough to show that behavior is preserved in terms of the values that are produced and in terms of the sequences of actions that are executed. But how do you show that all the necessary timing constraints would be observed without overly restricting the system's freedom to optimize the design? Secondly, how are complex algorithms such as clique partitioning modeled as transformations? It will probably be necessary to define a set of invariant properties for such an algorithm and then to prove both that the algorithm preserves the invariants and that the invariants are sufficient to guarantee that behavior is preserved. The third problem has to do with the level at which each synthesis step is verified. Assuming that the abstract algorithm or transformation can be verified, this does not guarantee that the code itself is correct. That is beyond the capacity of current techniques in code verification.

As the above indicates, no synthesis system will ever be verified to the point where testing and simulation can be dispensed with. This does not mean, however, that formal verification is not worth pursuing. Formal verification is a means of doing a precise, disciplined analysis of what a system is doing. The whole process of building formal models of a system and submitting them to rigorous logical analysis, if done intelligently and not mechanically, gives great insight into how a system works and what its flaws and weaknesses are. If formal verification cannot prove systems perfect, it can at least make them better.

3) Integrating Levels of Design:

High-level synthesis systems have almost all used a top-down approach to design. This approach begins with a very abstract view of behavior and structure, and makes the high-level decisions about optimization, scheduling, and allocation purely in terms of that view. Costs are measured in terms of abstract operators, and perhaps registers, and speed in terms of abstract control steps. Factors such as the particular properties of hardware modules, layout, wiring, and parasitics are not considered at all at this stage. The physical design is not done until the very end of the process, when all the key architectural decisions have been made. This division into levels has been seen as necessary in order to break the synthesis problem into a number of smaller subtasks and thus make it tractable. It is also artificial, however, and something is lost when synthesis is done that way. Human designers think in terms of a floor plan even when they are working at the overall structure of the system [86]; and there is evidence [52] that consideration of layout, wiring, and so on can cause one to choose a very different register-transfer level structure for a system. It might be desirable, for example, to have two ALU's instead of one in a system in order to simplify the wiring, even though there is no gain in parallelism.

It is necessary, therefore, for synthesis systems to be able to work across the various levels and domains of design representation. This means a number of things. For one, it means performing or predicting the effects of physical design, including floorplanning, along with the synthesis of the register-transfer level structure. The BUD program does this to some extent by partitioning the behavioral specification in a way that suggests a geometry for the design and then using that partitioning to derive the logical structure. Thus physical layout and structure evolve together. At this point, however, the process is rather crude, and needs much more work.

The second place where integration is needed is in the evaluation of designs. In order to make realistic design tradeoffs at the algorithmic and register transfer levels, it is necessary to be able to anticipate what the lower level tools will do. For example, logic optimization can change significantly the number of gates needed for a design [56]. What happens at the level of physical design is also important. Area does not just depend on the number of memory and functional units, but also on the multiplexing and wiring, and the wasted area due to layout constraints. Similarly, speed is not simply proportional to the number of control steps or clock cycles needed to execute the required functions, but also to the length of a clock cycle, which depends on the delays through all the paths in the system. Wiring can have a substantial impact here as well. An adequate evaluator must be able to anticipate the impact of these factors on a design. BUD estimates the area and performance of a register-transfer level design, including the effects of layout and wiring, by using its approximate floor

plan for the design. The PLEST program [87] performs area estimation for a structure based on a stochastic model of the number of wiring tracks needed. The models used in these sytems still have their limitations, however, More research is needed in this area.

Finally, integration across design levels means maintaining a single representation that contains all levels and domains of design information, as the ADAM Design Data Structure [2] does. With this kind of structure, it should be possible to allow different parts of a design to exist at different levels during the design process. It is sometimes important to be able to pick out a critical part of a structure and design it in great detail before working on the rest of the structure. That is because in such cases the critical part has the largest impact on the cost of the design, and no design decisions should be made on other parts that might block optimization of the critical part. It should also be possible to have multi-level specifications. For example, one might be able to specify the part of a system that does computation at the algorithmic level., while the input/output section requires specification at the logic level due to the critical timing constraints on it. No current synthesis system supports this adequately.

4) Evaluation of Synthesis Programs:

The synthesis field has now become large enough that there are a number of competing algorithms and programs. It is important to be able to compare the results of these programs on a set of common examples in order to learn more about the relative advantages of the different approaches. Beginning with the 1988 High-Level Synthesis Workshop, work has gone on to collect a set of benchmark hardware descriptions [88]. To date, however, the use of these benchmarks has been spotty and inconsistent.

There are a number of cautions that must be observed if meaningful comparisons are to be made. Scheduling is very sensitive to operator delays, so schedules should be compared using the same module library. It is a problem when programs cannot use externally defined modules. Programs that use different styles of data path design, such as buses versus multiplexers, should be compared using a common metric such as total area. Final layouts can be compared only if done in the same layout style. With regard to run times, only very large differences should be considered significant. Otherwise machine, language, or operating system differences could lead to misinterpretation of the results.

There are other factors that are important besides the results of a program when run on one or two examples and the final run times. The ability of a program to generate good designs on a variety of examples is important. The ability to search a large portion of the design space should be evaluated, as should the program's ability to deal with more than one objective function or constraint type.

Finally, fabrication of working designs produced by high-level synthesis systems is needed to prove the viability of the individual programs and of the whole approach.

C. Results

High-level synthesis systems are now being tied to logic synthesis and layout systems so that they can produce complete, manufacturable designs. We are therefore just beginning to see actual results from these systems.

The most advanced systems in this regard are the special-purpose systems for synthesizing signal processors. The Cathedral system has produced a number of chips implementing complex signal processing algorithms [89]. The FACE system has also been used to produce actual designs [70].

With regard to more general-purpose synthesis systems, a complete 801 microprocessor [90] with over 100 instructions, a streamlined architecture and a 4-stage pipeline has been synthesized with the Yorktown Silicon Compiler in less than 4 CPU hours on an IBM 3090 [91]. The synthesized processor had the same performance as an RT-level hand design (using the same logic synthesis tool), but used 26% more transistors over all (45% more combinational logic and 11% more latches). The circuits were synthesized down to the transistor level using cell generators.

A complete Motorola 68000 was synthesized using CADDY/DSL [22] in a few CPU hours on a Siemens 7561. The behavioral description was developed at a fairly low level in 1/2 person-years, resulting in a circuit quite similar to the original 68000. The synthesis result was a netlist of standard cells.

The System Architect's Workbench has recently added the LASSIE program [92] to translate the register-transfer level designs produced by the Workbench into layouts. These layouts have been used to test the effectiveness of some new design techniques [77]. Work is underway to connect the Workbench to commercial layout systems in order to produce real chip designs.

D. Conclusion

The problem of translating a high-level, behavioral description of a digital system into a register-transfer level structure has been divided into a number of subtasks. The specification is first compiled into an internal representation, usually a data flow/control flow graph. The graph is transformed so as to make the resulting design more efficient. Operations are then scheduled into time steps and hardware elements are allocated for the processors and registers that are needed. These are connected together along with the necessary multiplexing and busing, giving the basic structure for the data paths. Next, specific modules are selected to implement the abstract hardware blocks, if this has not already been done. Finally a controller is designed to generate the control signals needed to invoke the data operations in the scheduled control steps. This completed register-transfer level design is then ready to be handed on to a logic synthesis system or physical design system.

The individual tasks, particularly the key tasks of scheduling and allocating, are well understood, and there are a variety of effective techniques that have been applied to them. However, when synthesis is seen in its real context, opening up such issues as specification, designer intervention, input/output, the need to handle complex timing constraints, and the relation of synthesis to the overall design and fabrication process, there are still many unanswered questions. These are the areas where much research and development are needed in order to make high-level synthesis practical.

References

[1] C. G. Bell and A. Newell, *Computer Structures: Readings and Examples*. New York, NY: McGraw-Hill, 1971.

[2] D. W. Knapp and A. C. Parker, "A Unified Representation for Design Information," in *7th Int'l Conf. on CHDL*. Amsterdam, The Netherlands: North-Holland, 1985, pp. 337–353.

[3] D. D. Gajski and R. H. Kuhn, "New VLSI Tools," *Computer*, vol. 16, no. 12, pp. 11–14, Dec., 1983.

[4] R. A. Walker and D. E. Thomas, "A Model of Design Representation and Synthesis," in *Proc. of the 22nd Design Automation Conf.* New York, NY: ACM/IEEE, 1985, pp. 453–459.

[5] T. D. Friedman and S. C. Yang, "Methods used in an Automatic Logic Design Generator (ALERT)," *IEEE Trans. Comput.* vol. C-18, 1969, pp. 593–614.

[6] ——, "Quality of Designs from an Automatic Logic Generator (ALERT), in *Proc. of the 7th Design Automation Conf.* New York, NY: ACM/IEEE, 1970, pp. 71–89.

[7] J. R. Duley and D. L. Dietmeyer, "Translation of a DDL digital system specification to Boolean equations," *IEEE Trans. Comput.*, vol. C18, 1969, pp. 305–313.

[8] S. W. Director, A. C. Parker, D. P. Siewiorek, and D. E. Thomas, "A Design Methodology and Computer Aids for Digital VLSI Systems," *IEEE Trans. Circuits Syst.*, vol. CAS-28, no. 7, July, 1981, pp. 634–45.

[9] G. Zimmerman, "Eine Methode zum Entwurf von Digitalrechnern mit der Programmiersprache MIMOLA," *Informatik-Fachberichte*, vol. 5. Berlin: Springer-Verlag, 1976.

[10] S. J. Krolikoski, "The V-Synth System," in *Proc. COMPCON Spring 88*. New York, NY: IEEE, Feb. 1988, pp. 328–331.

[11] R. Camposano and W. Rosenstiel, "Algorithmische Synthese deterministischer (Petri-) Netze aus Ablaufbeschreibungen digitaler Systeme," 22/80, Fakultaet fuer Informatik, University of Karlsruhe, West Germany, 1980.

[12] H. DeMan, J. Rabaey, P. Six, and L. Claesen, "Cathedrall II: A Silicon Compiler for Digital Signal Processing," *IEEE Design and Test*, vol. 3, no. 6, Dec. 1986, pp. 13–25.

[13] N. Park and A. C. Parker, "Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications," *IEEE Trans. on CAD*, vol. 7, no. 3, pp. 356–370, Mar. 1988.

[14] E. F. Girczcy, "Automatic Generation of Microsequenced Data Paths to Realize ADA Circuit Descriptions," PhD Thesis, Carleton University, Ottawa, Canada, July 1984.

[15] G. Borriello and R. H. Katz, "Synthesis and Optimization of Interface Transducer Logic," in *Proc. of ICCAD-87*. New York, NY: IEEE, Nov. 9, 1987, pp. 274–277.

[16] C. Tseng and D. P. Siewiorek, "Automated Synthesis of Data Paths in Digital Systems," *IEEE Trans. on CAD*, vol. CAD-5, no. 3, pp. 379–395, July, 1986.

[17] D. E. Thomas, C. Y. Hitchcock III, T. J. Kowalski, J. V. Rajan, and R. Walker, "Automatic Data Path Synthesis," *Computer*, vol. 16, no. 12, pp. 59–70, Dec. 1983.

[18] D. E. Thomas, E. M. Dirkes, R. A. Walker, J. V. Rajan, J. A. Nestor, and R. L. Blackburn, "The System Architect's Workbench," in *Proc. of the 25th Design Automation Conf.* New York, NY: ACM/IEEE, June 1988, pp. 337–343.

[19] A. C. Parker, J. Pizarro, and M. Mlinar, "MAHA: A Program for Datapath Synthesis," in *Proc. of the 23rd Design Automation Conf.* New York, NY: ACM/IEEE, June 1986, pp. 461–466.

[20] P. G. Paulin, J. P. Knight, and E. F. Girczyc, "HAL: A Multi-Paradigm Approach to Automatic Data Path Synthesis," in *Proc. of the 23rd Design Automation Conf.* New York, NY: ACM/IEEE, June 1986, pp. 263–270.

[21] B. M. Pangrle and D. D. Gajski, "Design Tools for Intelligent Silicon Compilation," *IEEE Trans. on CAD*, vol. CAD-6, no. 6, pp. 1098–1112, Nov. 1987.

[22] R. Camposano and W. Rosenstiel, "Synthesizing Circuits From Behavioral Descriptions," *IEEE Trans. on CAD*, vol. 8, no. 2, pp. 171–180, Feb. 1989.

[23] P. Marwedel, "A new synthesis algorithm for the MIMOLA software system," in *Proc. of the 23rd Design Automation Conf.* New York, NY: ACM/IEEE, 1986, pp. 271–77.

[24] M. C. McFarland and T. J. Kowalski, "Assisting DAA: The Use of Global Analysis in an Expert System," in *Proc. of ICCD 86*. New York, NY: IEEE, Oct. 1986, pp. 482–485.

[25] C. Tseng, R. Wei, S. G. Rothweiler, M. M. Tong, and A. K. Bose, "Bridge: A Versatile Behavioral Synthesis System," in *Proc. of the 25th Design Automation Conf.* New York, NY: ACM/IEEE, June 1988, pp. 415–420.

[26] R. Camposano "Structural Synthesis in the Yorktown Silicon Compiler," in *VLSI '87, VLSI Design of Digital Systems*. Amsterdam, The Netherlands: North-Holland, 1988, pp. 61–72.

[27] H. Trickey, "Flamel: A High-Level Hardware Compiler," *IEEE Trans. on CAD*, vol. CAD-6, no. 2, pp. 259–269, Mar. 1987.

[28] M. R. Barbacci, "Instruction Set Processor Specifications (ISPS): The Notation and its Applications," *IEEE Trans. on Computers*, vol. C-30, no. 1, pp. 24–40, Jan. 1981.

[29] G. Zimmermann, "MDS—The Mimola Design Method," *J. Digital Systems*, vol. 4, no. 3, pp. 337–369, 1980.

[30] R. Camposano and R. M. Tabet, "Design Representation for the Synthesis of Behavioral VHDL Models," in *Proc. of the 9th Int'l Conf. on CHDL*. New York, NY: Elsevier/North Holland, June 1989.

[31] S. D. Johnson, *Synthesis of Digital Designs from Recursion Equations* (PhD Thesis, Indiana University, 1984). Published, Cambridge, MA: MIT Press, 1984.

[32] T. Aoyagi, M. Fujita, and H. Tanaka, "Temporal Programming Language Tokio," in *Logic Programming Conf. '85*. Berlin: Springer-Verlag, 1985, pp. 128–137.

[33] G. De Micheli and D. C. Ku, "HERCULES: A System for High-Level Synthesis," in *Proc. of the 25th Design Automation Conf.* New York, NY: ACM/IEEE, June 1988, pp. 483–488.

[34] M. S. Hecht, *Flow Analysis of Computer Programs*. New York, NY: North-Holland, 1977.

[35] E. A. Snow, D. P. Siewiorek, and D. E. Thomas, "A Technology-Relative Computer-Aided Design System: Abstract Representations, Transformations, and Design Tradeoffs," in *Proc. of the 15th Design Automation Conf.* New York, NY: ACM/IEEE, 1978, pp. 220–226.

[36] M. C. McFarland, "The VT: A Database for Automated Digital Design," DRC-01-4-80, Design Research Center, Carnegie-Mellon University, Dec. 1978.

[37] R. K. Brayton, R. Camposano, G. DeMicheli, R. H. J. M. Otten, and J. vanEijindhoven, "The Yorktown Silicon Compiler," in *Silicon Compilation*, D. D. Gajski, Ed. Reading, MA: Addison-Wesley, 1988, pp. 204–311.

[38] D. Knapp, J. Granacki, and A. C. Parker, "An Expert Synthesis System," in *Proc. of ICCAD-84*. New York, NY: IEEE, Sept. 1984, pp. 419–24.

[39] E. A. Snow, Automation of Module Set Indpendent Register-Transfer Level Design, PhD Thesis, Carnegie-Mellon University, Pittsburgh, PA, Apr. 1978.

[40] R. A. Walker and D. E. Thomas, "Design Representation and Transformation in The System Architect's Workbench," in *Proc. of ICCAD-87*. New York, NY: IEEE, 1987, pp. 166–169.

[41] M. Balakrishnan, A. K. Majumdar, D. K. Banjeri, and J. G. Linders, "Allocation of Multi-port Memories in Data Path Synthesis," in *Proc. of ICCAD-87*. New York, NY: IEEE, 1987, pp. 266–269.

[42] L. J. Hafer and A. C. Parker, "A Formal Method for the Specification, Analysis, and Design of Register-Transfer Level Digital Logic," *IEEE Trans. on CAD*, vol. 2, no. 1, pp. 4–18, Jan. 1983.

[43] G. W. Leive and D. E. Thomas, "A Technology Relative Logic Synthesis and Module Selection System," in *Proc. of the 18th Design Automation Conf.* New York, NY: ACM/IEEE, 1981, pp. 479–485.

[44] M. C. McFarland, "Using Bottom-Up Design Techniques in the Synthesis of Digital Hardware from Abstract Behavioral Descriptions," in *Proc. of the 23rd Design Automation Conf.* New York, NY: ACM/IEEE, June 1986, pp. 474–480.

[45] R. Jain, A. C. Parker, and N. Park, "Module Selection for Pipelined Synthesis," in *Proc. of the 25th Design Automation Conf.* New York, NY: ACM/IEEE, June 1988, pp. 542–547.

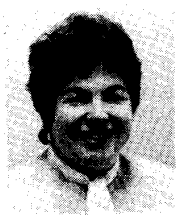
[46] R. Jain, K. Kucukcakar, M. Mlinar, and A. Parker, "Experience

with the ADAM Synthesis System," in *Proc. of the 26th Design Automation Conf.* New York, NY: ACM/IEEE, 1989, pp. 55–61.
[47] G. De Micheli, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Optimal State Assignment for Finite State Machines," *IEEE Trans. on CAD*, vol. CAD-4, no. 3, pp. 269–285, July 1985.
[48] S. Devadas, H. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli, "MUSTANG: State Assignment of Finite State Machines Targeting Multi-Level Logic Implementations," *IEEE Trans. on CAD*, vol. 7, no. 12, pp. 1290–1300, Dec. 1988.
[49] A. W. Nagle, R. Cloutier, and A. C. Parker, "Synthesis of hardware for the control of digital systems," *IEEE Trans. on CAD*, vol. CAD-1, no. 4, pp. 201–12, Oct. 1982.
[50] L. Nowak, "Graph Based Retargetable Microcode Compilation in the MIMOLA Design System," in *Proc. of the 20th Annual Workshop on Microprogramming*. New York, NY: ACM, 1987, pp. 126–132.
[51] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY: W. H. Freeman and Co., 1979.
[52] M. C. McFarland, "Reevaluating the Design Space for Register-Transfer Hardware Synthesis," in *Proc. of ICCAD-87*. New York, NY: IEEE, Nov. 1987, pp. 262–265.
[53] P. G. Paulin and J. P. Knight, "Force-Directed Scheduling in Automatic Data Path Synthesis," in *Proc. of the 24th Design Automation Conf.* New York, NY: ACM/IEEE, June 1987, pp. 195–202.
[54] F. D. Brewer and D. D. Gajski, "Knowledge Based Control in Micro-Architecture Design," in *Proc. of the 24th Design Automation Conf.* New York, NY: ACM/IEEE, June 1987, pp. 203–209.
[55] J. A. Nestor, "Specification & Synthesis of Digital Systems with Interfaces," CMUCAD-87-10, Department of Electrical and Computer Engineering, Carnegie-Mellon University, Pittsburgh, PA, Apr. 1987.
[56] R. Camposano and L. H. Trevillyan, "The Integration of Logic Synthesis and High-Level Synthesis," in *Proc. of the Int'l Symp. of Circuits and Systems*. New York, NY: IEEE, 1989, pp. 744–747.
[57] L. J. Hafer and A. C. Parker, "Register-Transfer Level Digital Design Automation: The Allocation Process," in *Proc. of the 15th Design Automation Conf.* New York, NY: ACM/IEEE, June 1978, pp. 213–219.
[58] T. J. Kowalski, *An Artificial Intelligence Approach to VLSI Design*. Boston, MA: Kluwer Academic Publishers, 1985.
[59] V. Berstis, "The V Compiler: Automating Hardware Design," *IEEE Design and Test*, vol. 6, no. 2, pp. 8–17, Apr. 1989.
[60] S. Devadas and A. R. Newton, "Algorithms for Hardware Allocation in Data Path Synthesis," *IEEE Trans. on CAD*, vol. 8, no. 7, pp. 768–781, July 1989.
[61] M. Balakrishnan and P. Marwedel, "Integrated Scheduling and Binding: A Synthesis Approach for Design Space Exploration," in *Proc. of 26th Design Automation Conf.* New York, NY: ACM/IEEE, pp. 68–74, June 1989.
[62] M. R. Barbacci, Automated Exploration of the Design Space for Register Transfer (RT) Systems, PhD Thesis, Carnegie-Mellon University, Pittsburgh, PA, 1973.
[63] Z. Peng, "Synthesis of VLSI Systems with the CAMAD Design Aid," in *Proc. of the 23th Design Automation Conf.* New York, NY: ACM/IEEE, June 1986, pp. 278–284.
[64] S. Davidson, D. Landskov, B. D. Shriver, and P. W. Mallett, "Some experiments in local microcode compaction for horizontal machines," *IEEE Trans. on Computers*, vol. C-30, no. 7, pp. 460–477, July 1981.
[65] C. Y. Hitchcock and D. E. Thomas, "A Method of Automatic Data Path Synthesis," in *Proc. of the 20th Design Automation Conf.* New York, NY: ACM/IEEE, pp. 484–489, June 1983.
[66] F. J. Kurdahi and A. C. Parker, "REAL: A Program for REgister ALlocation," in *Proc. of the 24th Design Automation Conf.* New York, NY: ACM/IEEE, June 1987, pp. 210–215.
[67] B. M. Pangrle, "Splicer: A Heuristic Approach to Connectivity Binding," in *Proc. of the 25th Design Automation Conf.* New York, NY: ACM/IEEE, June 1988, pp. 536–541.
[68] T. J. Kowalski and D. E. Thomas, "The VLSI Design Automation Assistant: A Prototype System," in *Proc. of the 20th Design Automation Conf.* New York, NY: ACM/IEEE, 1983, pp. 479–483.
[69] ——, "The VLSI Design Automation Assistant: An IBM System/370 Design," *Design and Test of Computers*, vol. 1, no. 1, pp. 60–69, Feb. 1984.
[70] A. E. Casavant, M. A. D'Abreu, M. Dragomirecky, D. A. Duff, J. R. Jasica, M. J. Hartman, K. S. Hwang, and W. D. Smith, "A Synthesis Environment for Designing DSP Systems," *IEEE Design and Test*, Apr. 1989, pp. 35–45.
[71] J. V. Rajan and D. E. Thomas, "Synthesis by Delayed Binding of Decisions," in *Proc. of the 22nd Design Automation Conf.* New York, NY: ACM/IEEE, June 1985, pp. 367–73.
[72] J. V. Rajan, Automatic Synthesis of Microprocessors, PhD Thesis, Carnegie Mellon University, Pittsburgh, PA, Dec. 1988.
[73] D. E. Thomas and D. P. Siewiorek, "Measuring Designer Performance to Verify Design Automation Systems," *IEEE Trans. Comput.*, vol. C-30, no. 1, pp. 48–61, Jan. 1981.
[74] R. Camposano and A. Kunzmann, "Considering Timing Constraints in Synthesis from a Behavioral Description," in *Proc. of ICCD 86*. New York, NY: Oct. 1986, pp. 6–9.
[75] S. Hyati and A. C. Parker, "Automatic Production of Controller Specifications From Control and Timing Behavioral Descriptions," in *Proc. of the 26th Design Automation Conf.* New York, NY: ACM/IEEE, 1989, pp. 75–80.
[76] G. Borriello, "Combining Event and Data-Flow Graphs in Behavioral Synthesis," in *Proc. of ICCAD-88*. New York, NY: IEEE, 1988, pp. 56–59.
[77] E. D. Lagnese and D. E. Thomas, "Architectural Partitioning for System Level Design," in *Proc. of the 26th Design Automation Conf.* New York, NY: ACM/IEEE, June 1989, pp. 62–67.
[78] R. L. Blackburn, D. E. Thomas, and P. M. Koenig, "CORAL II: Linking Behavior and Structure in an IC Design System," in *Proc. of the 25th Design Automation Conf.* New York, NY: ACM/IEEE, June 1988, pp. 529–535.
[79] J. J. Granacki and A. C. Parker, "PHRAN-SPAN: A Natural Language Interface for System Specifications," in *Proc. of the 24th Design Automation Conf.* New York, NY: ACM/IEEE, 1987, pp. 416–422.
[80] R. Di Russo, A Design Implementation Using the CMU-DA System, Master's Thesis, Department of Electrical Engineering, Carnegie-Mellon University, Pittsburgh, PA, Oct. 20, 1981.
[81] L. Nowak and P. Marwedel, "Verification of Hardware Descriptions by Retargetable Code Generation," in *Proc. of the 26th Design Automation Conf.* New York, NY: ACM/IEEE, 1989, pp. 441–447.
[82] D. W. Knapp, "An Interactive Tool for Register-Level Structure Optimization," in *Proc. of the 26th Design Automation Conf.* New York, NY: ACM/IEEE, June 1989, pp. 598–601.
[83] M. C. Browne, E. M. Clarke, D. L. Dill, and B. Mishra, "Automatic verification of Sequential Circuits Using Temporal Logic," *IEEE Trans. Comput.*, vol. C-35, no. 12, Dec. 1986, pp. 1035–1044.
[84] A. Camilleri, M. Gordon, and T. Melham, "Hardware Verification Using High-Order Logic," in *From HDL Descriptions to Guaranteed Correct Circuit Designs*, D. Borrione, Ed. Amsterdam, The Netherlands: North-Holland, 1987.
[85] M. C. McFarland and A. C. Parker, "An Abstract Model of Behavior for Hardware Descriptions," *IEEE Trans. Comput.*, vol. C-32, no. 7, July, 1983, pp. 621–36.
[86] M. C. McFarland, "Computer-Aided Partitioning of Behavioral Hardware Descriptions," in *Proc. of the 20th Design Automation Conf.* New York, NY: ACM/IEEE, June 1983, pp. 472–478.
[87] F. J. Kurdahi and A. C. Parker, "PLEST: A Program for Area Estimation of VLSI Integrated Circuits," in *Proc. of the 23th Design Automation Conf.* New York, NY: ACM/IEEE, June 1986, pp. 467–473.
[88] G. Borriello and E. Detjens, "High-Level Synthesis: Current Status and Future Directions," in *Proc. of the 25th Design Automation Conf.* New York, NY: ACM/IEEE, June 1988, pp. 477–482.
[89] S. Note, J. Van Meerbergen, F. Catthoor, and H. De Man,

"Automated synthesis of a high speed Cordic algorithm with the Cathedral-III compilation system," in *ISCAS 88*. New York, NY: IEEE, June, 1988, pp. 581–584.
[90] G. Radin, "The 801 Minicomputer," IBM *J. Res. Dev.*, vol. 27, no. 12, pp. 237–246, Dec. 1983.
[91] R. Camposano, "Design Process Model in the Yorktown Silicon Compiler," in *Proc. of the 25th Design Automation Conf.* New York, NY: ACM/IEEE, June 1988, pp. 489–494.
[92] M. T. Trick and S. W. Director, "LASSIE: Structure to Layout for Behavioral Synthesis Tools," in *Proc. of the 26th Design Automation Conf.* New York, NY: ACM/IEEE, 1989, pp. 104–109.

Michael C. McFarland (Member, IEEE) received the A.B. degree in physics and mathematics from Cornell University, Ithaca, NY, in 1969, and the M.S. and Ph.D. degrees in electrial engineering from Carnegie-Mellon University, Pittsburgh, PA, in 1978 and 1981, respectively. He also received the M.Div. degree in 1984, and the Th.M. degree in social ethics, both from the Weston School of Theology, Cambridge, MA.

He is currently Assistant Professor of Computer Science at Boston College, Chestnut Hill, MA. He also consults for AT&T Bell Laboratories, where he was on assignment in 1985–86. His areas of interest are computer-aided design of digital systems, formal specification and verification of hardware, system level design, and ethics in engineering and computer science.



Alice C. Parker (Senior Member, IEEE) received the M.S.E.E. degree from Stanford University, Stanford, CA, in 1971, and the Ph.D. degree in electrical engineering from North Carolina State University, Raleigh, in 1975.

She is currently an Associate Professor in the Department of Electrical Engineering Systems, University of Southern California, Los Angeles. Formerly, she was a member of the faculty of Carnegie-Mellon University, Pittsburgh, PA. Her areas of interest are automated synthesis of digital systems, CAD frameworks, CAD databases, and applications of AI techniques to design automation.

Raul Camposano received the diploma in electrical engineering from the University of Chile, Santiago, in 1978, and the Ph.D. degree in computer science from the University of Karlsruhe, West Germany, in 1981.

In 1982 and 1983, respectively, he was with the University of Siegen, West Germany, and the Universidad del Norte, Antofagasta, Chile. From 1984 to 1986, he was a researcher in the Computer Science Research Laboratory of the University of Karlsruhe. Since 1986, he has been at the IBM Thomas J. Watson Research Center, Yorktown Heights, NY. His research interests include computer-aided design and design automation for digital systems.