J. Webster (ed.), *Wiley Encyclopedia of Electrical and Electronics Engineering*
Copyright © 1999 John Wiley & Sons, Inc.

HIGH LEVEL SYNTHESIS

High level synthesis (also known as behavioral synthesis) has been a popular research and development area since the 1970s. Several thousand papers on high level synthesis have been published and more than a hundred prototype and production-quality high level synthesis systems have been developed. There are several survey papers and books on high level synthesis (1,2,3,4,5). Recently, system-level synthesis and hardware–software codesign have also received widespread attention (6). Therefore, a comprehensive survey of the field is beyond the scope of a single article. Our goal here is to explain the key concepts and some of the most interesting directions as well as to provide a good starting point for readers who want to study high level synthesis.

This article is organized in the following way. We first discuss computational and hardware models, define the main tasks within high level synthesis, and list the key design objectives. After that we explain some of the techniques recently proposed in several research areas, which have received much attention in high level synthesis in the last few years.

Computational Model

A number of computational models well suited for high level synthesis are surveyed in Ref. 7. In general, the *DSP* systems have multiple inputs, multiple outputs, and finite number of states. They accept streams of samples on each of the inputs and produce streams of samples on each of the output ports. We represent an algorithm for a system by a hierarchical directed control-data flow graph (*CDFG*). In a CDFG the nodes represent data operators or subgraphs, data edges represent the flow of data between nodes, and control edges represent sequencing and timing constraints between nodes.

Often, designers restrict themselves to operators that are synchronous in that they consume at every input, and produce at every output, a fixed number of samples on every execution. This restriction has two important ramifications. First, the operators, and hence the system, are determinate in that a given set of input samples always results in the same outputs independent of the execution times. Second, the system is well behaved in that the data sample rate at any given data edge in the CDFG is independent of the inputs, and the ratio between any two data sample rates will be a statically known rational number. Mathematically, such a synchronous CDFG is equivalent to a continuous function over streams of data samples. Because CDFGs are, of course, causal, they are equivalent to a function that expresses the ith set of output samples in terms of the ith and earlier sets of input samples.

The system state is represented in a CDFG by special delay operator nodes that are initialized to a user-specified value. A delay operator node (often referred to as just delay or state) delays by one sample the stream of data on its sole input port. A CDFG corresponds to an algorithm for computing the output samples and the new samples to be stored at the delay nodes (i.e., the new state) given the input samples and the old (current) samples at the delay nodes (i.e., the old state). Intuitively, one can think of delay operators as representing registers holding states and the other operators as combinational logic.

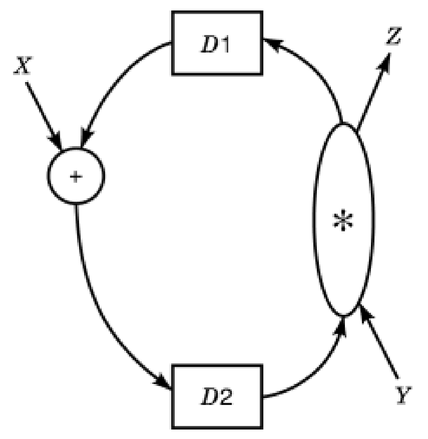


Fig. 1. An example CDFG constructed for the following computation

$$z(t+1) = y_*^{(t)} D2(t)$$
$$D2(t+1) = x(t) + D1(t)$$
$$D1(t) = z(t)$$

where x and y are inputs, z is the output, and $D1$ and $D2$ are computation states.

A system is completely represented by a CDFG and the initial values for all the delays in the CDFG. We further restrict ourselves to single-rate systems where the data rate is identical on all the inputs. The term CDFG often refers to such a single-rate synchronous CDFG.

Figure 1 shows an example CDFG with two inputs X and Y, and one output Z. The two delay nodes U and V are represented by boxes with the letter D. Timing relationships associated with a CDFG are also timing constraints specified by the user. These constraints arise from requirements of the interface to the external world and from performance requirements.

Consider a CDFG with P inputs, Q outputs, and R delay nodes (state nodes). Let $X[n] = (X_1[n]\ X_2[n]\ \ldots\ X_P[n])^T$ be the vector of nth samples at the P input nodes of the CDFG, $Y[n] = (Y_1[n]\ Y_2[n]\ \ldots\ Y_Q[n])^T$ be the vector of nth samples at the Q output nodes of the CDFG, $S[n] = (S_1[n]\ S_2[n]\ \ldots\ S_R[n])^T$ be the vector of nth samples at the R delay nodes of the CDFG.

Given initial values $S[0]$ of the samples in the delay nodes, the CDFG repeatedly computes output samples $Y[n]$ and new samples $S[n]$ for delay nodes from input samples $X[n]$ and old samples $S[n-1]$ at the delay nodes for $n = 1, 2, 3, \ldots$. Various timing parameters are associated with a CDFG as shown in Fig. 2. Because we have restricted ourselves to single-rate synchronous CDFGs, the data rates are identical at all nodes so that the intersample time interval is identical and constant for all nodes. The maximum rate at which such a CDFG can process samples is called its *throughout*, and the inverse of this rate, called *sample period*, is the minimum required time between successive samples at a node in the CDFG. The sample period, denoted by T_S, is an important timing parameter that is usually constrained not to exceed a maximum value.

A second set of timing parameters associated with a CDFG is the set of pairwise latencies. The latency $T_L(i, j)$, from the ith input node to the jth output node, is the delay between the arrival of a sample at the ith input and the production of the corresponding sample at the jth output. The sample correspondence is defined by the initial CDFG given by the user to specify the system—the nth sample at an output corresponds to the nth sample at an input. However, adding or deleting pipeline stages during algorithm transformation changes this correspondence. For example, if one were to add one level of pipelining to the initial CDFG, then

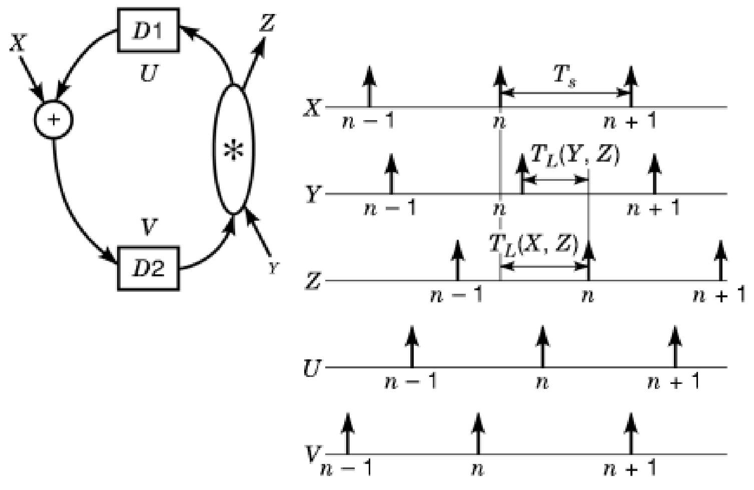


Fig. 2. Timing parameters associated with a CDFG.

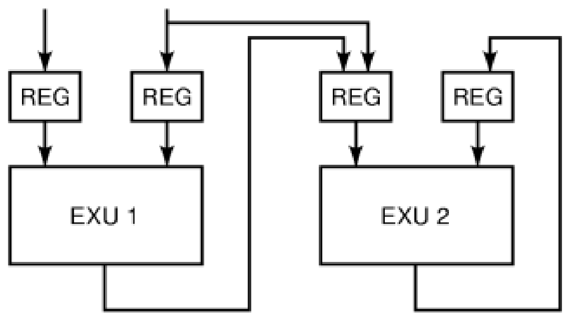


Fig. 3. A datapath with two functional units and four dedicated register files.

the latencies will be defined in terms of the $(n+1)$-th input samples and the nth output samples. The pairwise latencies $T_L(i, j)$ associated with a CDFG are also important timing parameters because they are constrained not to exceed specified maximum values in latency-critical systems.

Hardware Models

There are a number of hardware models used in high level synthesis. Most often the emphasis is on datapath, although recently control logic hardware cost modeling and memory have received considerable attention.

One popular hardware model is shown in Fig. 3. To stress the importance of interconnect minimization early in the design process, this model clusters all registers in register files, connected only to the inputs of the corresponding execution units. Other popular hardware models include a variation where any register file could be connected to any functional unit. Until recently, a large number of high level synthesis systems were assuming hardware models where registers are not grouped. However, recently, it become apparent that for that model, interconnect and control logic overhead is too high for realistic designs.

High Level Synthesis Tasks

The main high level synthesis tasks follow.

(1) Transformations The structure of computation is reorganized in such a way that input/output behavior is not altered. The goal is to make the computation more amenable for good implementation.
(2) Scheduling Operations are assigned to control steps in which the operations are to be executed so as to satisfy design constraints.
(3) Resource Allocation Variables and operations are assigned to registers and functional units, respectively, and the interconnection of the various resources in terms of buses and multiplexors is determined.
(4) Template Matching It is the process of mapping high-level algorithmic descriptions to specialized hardware libraries or instruction sets. The difficulty for template matching lies in the fact that the number of template matches can be extremely large and the possibility of enumerating all matches is prohibited.
(5) Hardware Mapping It refers to the process of selecting, for each operation in the CDFG, the type of functional unit that will perform it. For example, ripple carry adder, carry lookahead adder, or carry select adder can be selected for addition.
(6) Clock Selection It refers to the process of choosing a suitable clock period for the controller and data-path circuit. The choice of the clock period is known to have a significant effect on area, performance, and power consumption.
(7) Partitioning The behavior of an architecture is divided in multiple chips, or into multiple parts within the bounds of a single chip while minimizing the global interconnect between parts. Partitioning in high level synthesis is different from partitioning in physical design because, due to the possibility of resource sharing of global interconnect, the number of connections between parts does not directly correspond to the amount of global interconnect required.

Design Metrics

Applications and implementation technology have had a significant impact on the relative popularity of design objectives. Even though area and speed were initially dominant metrics, recently focus has moved to metrics such as low power, testability, suitability for debugging, and protection of intellectual rights of designers.

Transformations for High Level Synthesis

Transformations alter the structure of a computation in such a way that the user-specified input/output relationship is maintained (8). Examples of transformations include pipelining, retiming, unfolding, folding, associativity, distributivity, and strength reduction. A simple application of associativity to reduce the critical path length is shown in Fig. 4.

Many transformations have been known for a long time and have been used in the compiler domain. For example, transformations such as constant and copy propagation and common subexpression elimination techniques are often used. Comprehensive reviews of use of transformations in parallelizing compilers state-of-the-art general-purpose computing environments, and *VLSI* DSP design are given in Refs. 9, 10, and 11 respectively.

In high level synthesis, transformations have received widespread attention (2,12,13,14,15) because of strong experimental evidence that they are most effective at the highest levels of abstractions, such as high level synthesis, although they have been widely used at all levels of abstraction in the synthesis process.

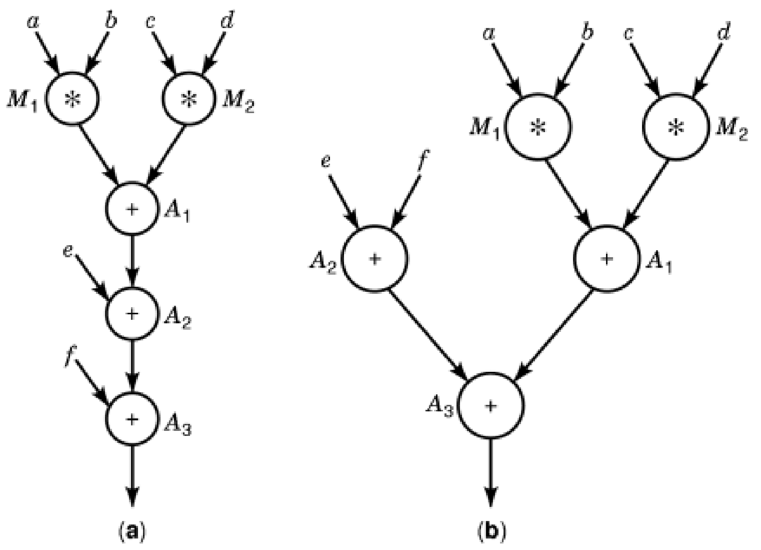


Fig. 4. Applying associativity to reduce the length of the critical path: CDFG (a) before and (b) after.

Transformations have been recognized as the high level synthesis step with the highest impact on design metrics. It has been demonstrated that exceptionally high improvements in all design metrics including area (16), throughput and latency (17,18), power (19), transient and permanent fault-tolerance overhead (14,20,21), memory (22), and testability (15) are achievable. Most of these approaches, however, consider only individual or small sets of transformations, which limits the improvement mainly for the following reasons:

- Transformations are notorious for their ability to alter unpredictably the numerical properties of a design and therefore its required word-length.
- The implementation and maintenance of a large number of transformations in compiler and high level synthesis environments is a formidable software task.
- The accurate prediction of effects of transformations has been rarely addressed, and it is widely considered infeasible.
- Their full potential can be explored only with proper orders of transformations. This is the objection that is most often raised and quoted with respect to the application of transformations. However, deriving such orders is mainly an art practiced by experienced developers of compiler and *CAD* tools, which often result in disappointing outcomes.

The previous approaches for transformation ordering can be classified in seven groups: local (peephole) optimization, static scripts, exhaustive search-based “generate-and-test” methods, algebraic approaches, probabilistic search techniques, bottleneck removal methods, and enabling-effect-based techniques.

Probably the most widely used technique for ordering transformations is local (peephole) optimization (23), where a compiler considers only a small section of code at a time in order to apply one by one iteratively and locally all available transformations. The advantages of the approach are that it is fast and simple to implement. However, performance is rarely high and is usually inferior to other approaches.

Another popular technique is a static approach to transformations ordering where their order is given a priori, most often in the form of a script (24). Script development is based on the experience of the compiler/synthesis software developer. This method has at least three drawbacks: it is a time-consuming process which involves a lot of experimentation on random examples in an ad-hoc manner; any knowledge about the relationship among transformations is only implicitly used; and the quality of the solution is often relatively low for programs/design which have different characteristics than the ones used for the development of the script.

The most powerful approach to transformation ordering is enumeration-based “generate and test” (25). All possible combinations of transformations are considered for a particular compilation, and the best one is selected using branch-and-bound or dynamic programming algorithms. The drawback is the large run time, often exponential in the number of transformations.

Another interesting approach is to use a mathematical theory behind the ordering of some transformations. However, this method is limited to only several linear loop transformations (26). Simulated annealing, genetic programming, and other probabilistic techniques in many situations provide a good trade-off between the run time and the quality of solution when little or no information about the topology of the solution space is available. Recently, several probabilistic search techniques have been proposed for ordering of transformations in both compiler and high level synthesis literature. For example, backward-propagation-based neural network techniques were used for developing probabilistic approach to the application of transformations in compilers for parallel computers (27), and approaches that combine both simulated annealing-based probabilistic and local heuristic optimization mechanism were used to demonstrate significant reductions in area and power (19).

In high level synthesis, several bottleneck identification and elimination approaches for ordering of transformations have been proposed (28). This line of work has been mainly addressing the throughput and latency optimization problems, where the bottlenecks can be easily identified and well quantified. Finally, the idea of enabling and disabling transformations has been recently explored in a number of compilation (29) and high level synthesis papers (13,30). Using this idea, several very powerful transformations scripts have been developed, such as one for maximally and arbitrarily fast implementation of linear computations (13), and joint optimization of latency and throughput for linear computations (18). Also, the enabling mechanism has been used as a basis for several approaches for ordering of transformations for optimization of general computations (31). The key advantage of this class of approaches is related to the intrinsic importance and the power of the enabling/disabling relationship between a pair of transformations.

Hong et al. (32) proposed an approach for the development of fully automatic, fast, and effective ordering of transformations for a variety of design metrics, such as throughput, area, and power. They introduced a new potential-driven statistical approach based on two new synthesis ideas. The first idea is to identify the characteristics of all transformations and the relationships between them based on their *potential* to reorganize a computation such that the complexity of the corresponding implementation is reduced. The second one is based on the observation that transformations may disable each other not only because they prevent the application of the other transformation but also because both transformations target the same *potential* of the computation. These two observations drastically reduce the search space to find efficient and effective scripts of transformations.

We explain the key ideas of their approach for ordering transformations using two small, but meaningful examples. We first consider the example in Fig. 5. Figure 5(a) shows the CDFG of a computation that consists of three additions and three multiplications. We assume that each operation takes one clock cycle and that the available time is two cycles. Note that because the length of the critical path is also two cycles, all operations are on the critical path. Obviously, regardless of the scheduling algorithm used, the final implementation requires at least three multipliers and three adders. This design has relatively low (50%) resource utilization for execution units.

An easy and effective way to improve this design is to apply transformations. Figure 5(b) shows the same CDFG after the application of distributivity on the isolated component which computes output x. Again all operations are on the critical path. Therefore, the only feasible schedule is the one shown in Fig. 5(b). It is easy to see that only two multipliers and two adders are required.

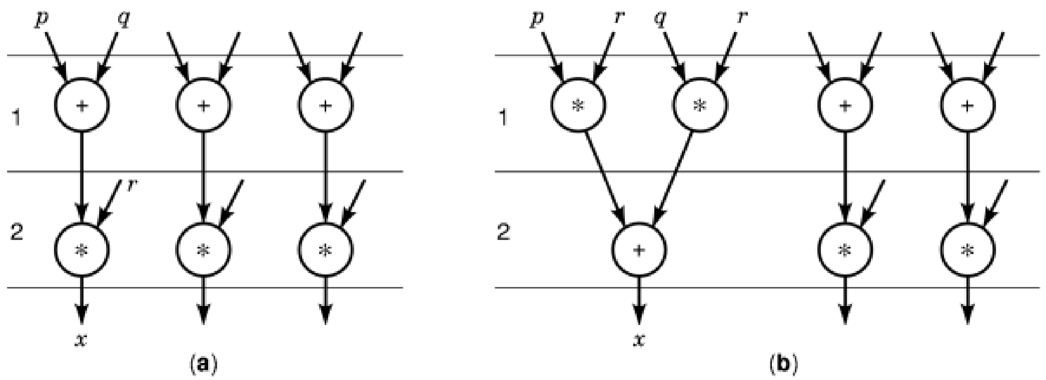


Fig. 5. An explanatory example to show the importance of considering the *potential* of computation and transformations for ordering transformations: (a) Initial CDFG and (b) transformed CDFG.

It is interesting and important to analyze why the CDFG in Fig. 5(b) is more amenable for the implementation with high resource utilization. The transformed design has one more operation than the initial design. So, one may expect that the transformed CDFG is inferior. However, this design has more evenly distributed operations of the same type along the time axis.

How can one generalize these observations so that they can be incorporated in a synthesis program? The essence of the approach can be summarized in the following way. The initial CDFG has a high *potential* to be improved with respect to the area of the final implementation because it has data precedence constraints that force the low resource utilization of execution units. This can be observed by comparing the current solution (or bounds on the quality of the current solution) with the solution derived with the assumption that all units can be used in all control steps. In this example, the bounds of the initial solution are three multipliers and three adders, whereas the bounds for fully utilized units are two multipliers and two adders. To achieve improvement, one needs such transformation that will more evenly distribute operations of the same type along the time axis (i.e., alter data dependencies in such a way that some of multiplications can be moved in the second control step, and some of additions in the first control step). It is easy to deduce that distributivity is such a transformation. As shown, distributivity is indeed effective.

Another important observation is that, when all execution units are well utilized, no further improvement can be achieved using transformations which alter only data and timing dependencies. This is because all *potential* along the dimension has already been realized (as indicated by absolute bounds on the number of execution units assuming 100% utilization). Thus, for further improvement, transformations that address some other *potential* of the computation are required. The substitution of constant multiplications with shifts and additions is such a transformation.

The second motivational example demonstrates the importance of simultaneous consideration of the effects of more than one transformation. It also illustrates the conceptual complexity of transformations ordering and provides initial hints about how this task can be simplified while preserving the power of transformations.

The design goal is area optimization under throughput constraint. Figure 6(a) shows the initial CDFG, which has five multiplications and two additions. The available time is three clock cycles. In the same fashion as in the first example, all operations are on the critical path. Therefore, the only feasible schedule is shown in Fig. 6(a). It is easy to see that three multipliers and two adders are required. The resource utilization is low for an ASIC design, for multipliers 55% and for adders 33%.

It is often required to apply one or more enabling transformations in order to eventually apply a transformation that will improve a design. Following this direction, we observe that our goal is to move some additions from the second control step to the first one and to move some multiplications from the first control step

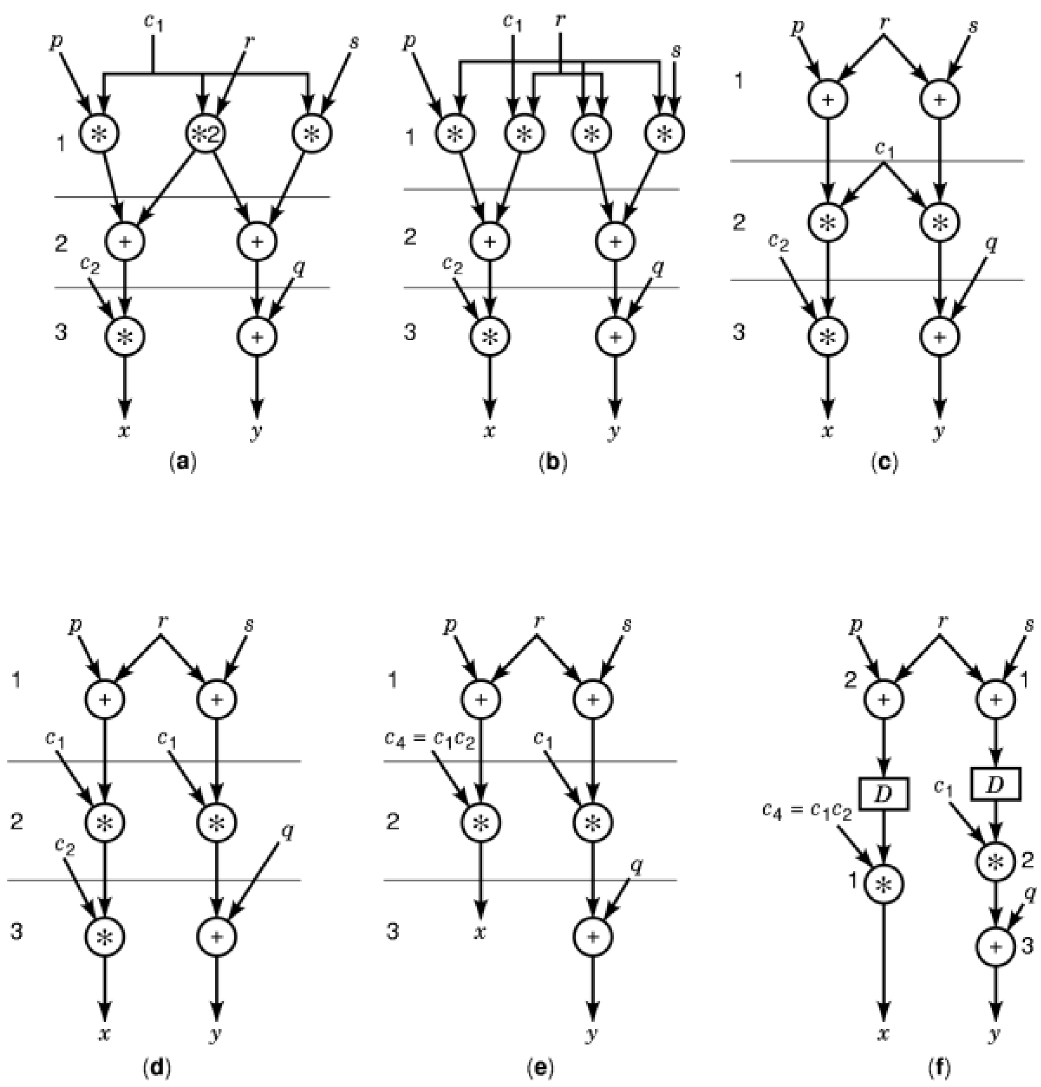


Fig. 6. An explanatory example to show the importance of considering enabling and disabling effects for ordering transformations. All data denoted by c_i are constants: (a) three multipliers, two adders; (b) four multipliers, two adders; (c) two multipliers, two adders; (d) two multipliers, two adders; (e) two multipliers, two adders; and (f) one multiplier, one adder.

to the second one. It is easy to verify that distributivity can accomplish this task. However, the application of distributivity is disabled because the result of multiplication $*2$ is used in two places. We first replicate multiplication $*2$ as shown in Fig. 6(b). If we stop the transformation process at this stage, it is easy to see that the corresponding implementation requires four multipliers and two adders, even more hardware than in the initial design. However, we can now apply distributivity twice, as shown in Fig. 6(c). We need only two multipliers and two adders. More importantly, we can continue the optimization process, by first replicating constant c_1 and then applying constant propagation, as shown in Figs. 6(d) and 6(e). Finally we can introduce a pipeline stage (if the computation is in a loop, retiming is performed) and obtain the functionality equivalent computation shown in 6(f). The numbers next to the nodes–operations indicate the clock cycle in which a particular operation is scheduled. We need only one multiplier and one adder.

This small example illustrates not only high effectiveness, but also an exceptional richness of degrees of freedom during optimization using sequences of transformations. The authors have shown, however, that, once the obtained insights are coupled with statistical methods, there is a conceptually simple, effective, and efficient solution for ordering of transformations. On a large set of diverse real-life examples, improvements in throughput, area, and power by large factors have been obtained. Both qualitative and quantitative statistical analysis supported the effectiveness, high robustness, and consistency of their approach to ordering transformations.

High Level Synthesis Techniques for Design Debugging

Functionality and timing debugging dominates both development time and cost of modern design process. The key technological and application trends indicate that the cost and time expenses of debugging follow sharply ascending trajectories. The technological trends, inducing additional constraints on debugging, are mainly related to increasingly reduced design observability and controllability. For example, the UltraSPARC design team reported that debugging efforts (mainly architecture and functional verification) took two times longer than the design activities (33). Similarly, the Hitachi design team reported that specification of the GMICRO500 microprocessor took less than 36 man-months, functional design took 80 man-months, standard cell design took only 3 man-months, physical design took less than 24 man-months, and finally, debugging efforts took more than 270 man-months (34). The difficulty of verifying designs is likely to worsen in the future. The Intel development strategy team foresees that a major design concern for year-2006 microprocessor designs will be the need to test exhaustively all possible computational and compatibility combinations (35). The same team also states that the circuitry in their future designs devoted to debugging purposes is estimated to increase sharply to 6% from the current 3% of the total die area. Because the product is expected to encompass 350 million transistors, the computation power of two state-of-the-art, 10 million transistor processors is expected to support debugging activities solely.

Two factors, most directly related to the expected increased difficulty of debugging future designs, are rapid growth in the number of transistors per pin in each new generation of designs and the increasing levels of hardware sharing caused by increasing clock speeds (see Fig. 7). Higher levels of hardware sharing are associated with more complex information flows in application-specific systems. For example, the analysis of physical data of state-of-the-art microprocessors (according to The Microprocessor Report) indicates that in less than 2 years (from late 1994 to middle of 1996) the number of transistors per pin increased by more than a factor of 2, from slightly more than 7000 to 14,100 transistors per pin. Whereas in 1994 eight microprocessors had a total of 16.1 million transistors and 2296 pins, in 1996 nine microprocessors had a total of 53.4 million transistors and 3781 pins. There is wider perspective on the trend of huge discrepancy in growth of number of transistors and pins per chip. Figure 7(a) points to the fact that the packaging technology has showed significant improvements over the past two decades. However, the rapid decrease of the silicon feature size has resulted in an even-higher ascending amount of available resources (transistors/gates) on chip. In Fig. 7(b), collected transistor per pin ratios for the most popular general purpose processors over the past three decades clearly point to the resulting exponential semiconductor over packaging technology trend.

Similarly, the size of an average embedded or DSP application has been approximately doubling each year, the time to market has been getting shorter for each new product generation, and there has been a strong market need for user customization of application-specific systems. These three application factors have resulted in shorter available debugging time of increasingly more complex designs. Finally, design and CAD trends that additionally emphasize the importance of debugging include design reuse, introduction of system

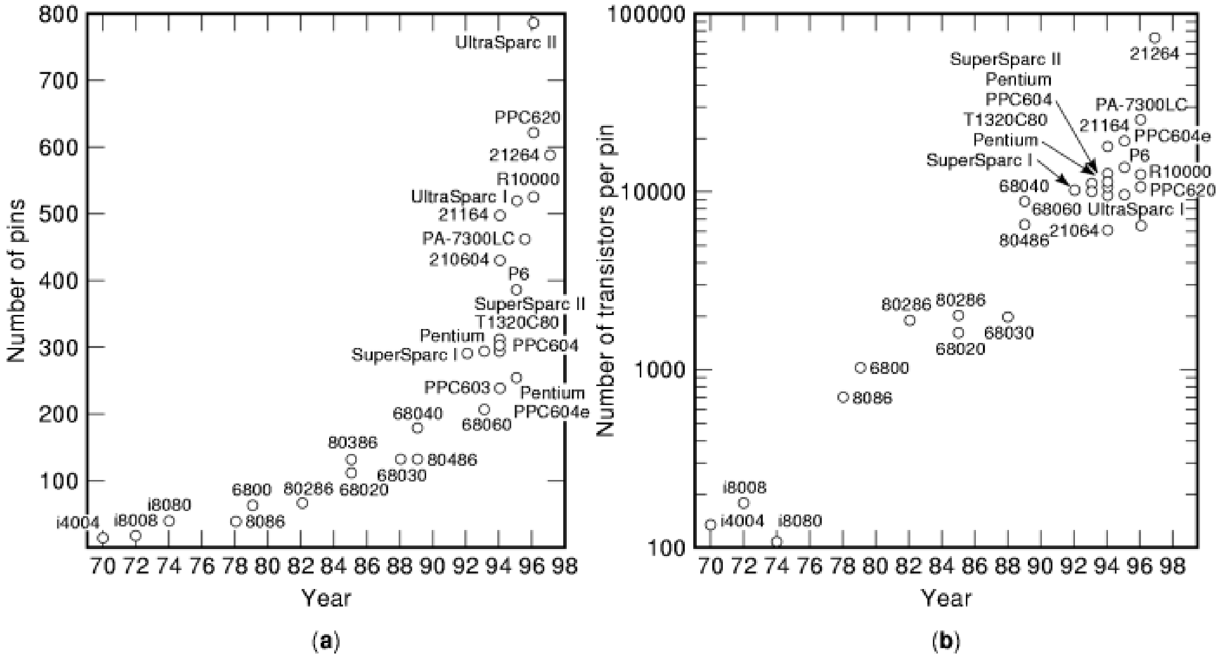


Fig. 7. Trends in the physical design of general-purpose microprocessors.

software layer, and increased importance of collaborative design. These factors result in increasingly intricate functional errors, often due to interaction of parts of designs written by several designers.

The Debugging Process. After the functional design of a hardware component is specified, the functional debugging is used to partially or fully verify and validate the design scheme. In the search for an error, a number of steps are taken, and they can be partitioned into the following four debugging phases (see Fig. 8).

Test Input Generation In the first phase of debugging, the goal is to generate and execute the input, data likely to make functional errors visible. In order to create the most promising set of input vectors designer's intuition can be used as well as such sophisticated methods as specialized databases or expert systems (36).

Error Detection In this phase, the designer discovers that the design does not function correctly for a particular input. The approach to detect an error in the functionality or timing depends on the system for design functional execution. If design simulation is used, then a set of conditional breakpoints selected by the designer may be used to check for functional or timing errors. In the case of executing the design functionality using an emulation system, the error discovery can be obtained either by output-matching software run by the supervising workstation or design-internal error events defined at design time. Upon an event that undoubtedly specifies functional error, the chip signals the supervising station that terminates the emulation.

Error Diagnosis In the third phase, the designer identifies the statement in the specification that causes incorrect behavior. This step in the debugging process may be exceptionally time consuming because of the characteristics of the current widely used functional execution approaches: design simulation and emulation. Modern design simulation is six to ten orders of magnitude slower than emulation and thus is used primarily for short, focused test sequences. Emulation has the required speed but imposes

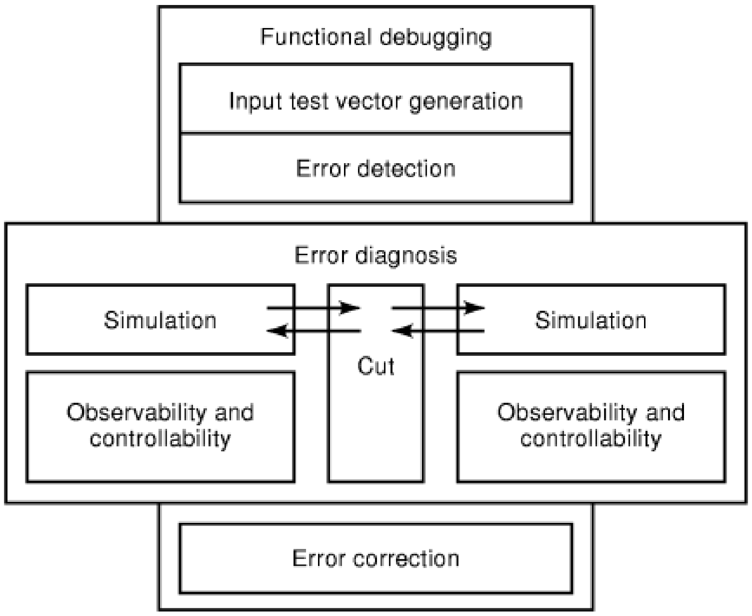

Functional debugging
Input test vector generation
Error detection
Error diagnosis
Simulation
Cut
Simulation
Observability and controllability
Observability and controllability
Error correction

Fig. 8. The four phases of functional debugging: test vector generation, error detection, error diagnosis, and error correction. Emerging debugging techniques rely on cut-based combining simulation and emulation for fast error diagnosis.

strict limitations on signal observability and controllability. To facilitate the advantages of both execution domains, Kirovski and Potkonjak (37,38) developed a technique that integrates emulation and simulation. Their technique uses a set of tools, transparent to both the design and debugging process, to enable the user to run long test sequences in emulation and, upon error detection, to roll-back to an arbitrary instance in execution time and switch over to simulation-based debugging for full design visibility and controllability.

Error Correction In the final phase, the faulty section or statement responsible for the observed fault is replaced by the corrected section. The design is recompiled if simulated, or its specification is updated and the design is, again, emulated or fabricated.

Existing Debugging Techniques. The existing, widely employed approaches for functional debugging are either design simulation, or chip emulation or fabrication.

Design simulation is typically run on a workstation environment and results in an arbitrary accurate overview of the simulated architecture (39,40). Unfortunately, simulation is an extremely computation-intensive process and results in two to ten orders slower functional execution in comparison to the fabricated chip. Most important, the actual simulation speed heavily depends on the simulation accuracy. State-of-the-art *VHDL* or Verilog *RTL*-level simulators are equipped with debuggers capable of performing error trace and timing analysis (Interra's Picasso) (41) and simulation backtracking (Synopsys' Cyclone) (42). For programmable processor simulation, instruction-set simulators providing full system visibility and controllability and of varying degrees of accuracy are used (40).

Emulated or fabricated designs provide real-life execution speed and yet extremely limited observability and controllability of the design internal structure (see Table 1). For example, the Hewlett-Packard functional verification team for the new HP PA8000 processor reports almost six orders of magnitude difference in speed

Table 1. Advantages and Disadvantages of Design Simulation, Emulation, and Coordinated Cut-Based Simulation and In-Circuit Emulation (38)

		Simulation		Emulation		Coordinated Cut-Based Simulation and Emulation
Execution speed	−	Exponentially decreases with design complexity	+	Order of magnitude slower than fabrication	+	Same as emulation
Controllability and observability	+	Provided	−	Limited and expensive (probing or providing addressability)	+	Same as simulation
Execution roll-back	+	Some simulators have implemented (42)	−	Not provided	+	Efficient solution

between the RTL-level simulated (0.5 Hz on a typical workstation) and FPGA based emulated (300 kHz) functional execution of their PA8000-based 200 MHz workstation (43).

An in-circuit emulator circuitry is made up of capture logic, which monitors the contents of the program address register, the internal data bus, and the control lines of the processor; trace circuitry comprising a FIFO buffer, which puts data from the capture logic to the output pins of the chip; and a content addressable memory and a software programmable logic array with emulation counters that together function as a finite-state machine, which performs the desired predetermined testing of the system. The debugging circuitry in the emulator, usually implemented using a *JTAG* boundary scan methodology (44), or dedicated high-bandwidth debug ports, enables controllability and observability of particular internal states of the emulated processor (45). The lack of full controllability and visibility of variables can be overcome by injecting debug instructions over the debug port directly into processor's pipeline. The injected instructions read or write data into user-specified registers. This approach modifies the pipeline and thus introduces novel design challenges.

The in-circuit emulation has evolved into logic (functional) porting of the processor model into arrays of rapid prototyping modules [arrays of gates, *FPGA*s (46,47), specialized processors such as Hydra (48)]. Such emulation engines provide both high execution speed and relatively high observability and controllability of all registers. The system that captures the signal values of the emulated design provides variable observability and controllability either by addressing the user-customized *SRAM* memory cells (Arkos emulator) (42,47) or by probing nets into the FPGA testbed [MP4 (46); *HDL-ICE* (47)]. The former case raises expenses, and the latter reduces visibility performance [only 256 probes are available in MP4 and 1152 in the HDL-ICE (47) for data inspection or update; moreover, probe reprogramming (i.e., repositioning) takes a considerable amount of time]. This methodology has been used during development of Intel's Pentium processor (49) and Sun's UltraSPARC-I and -II (47). While this approach is significantly faster than typical software simulation, it is still one to two orders of magnitude slower than the real operation speed (49). A method for optimal cutover from one simulation/emulation method to the other while system debugging has been patented (50).

In the CAD domain, Powley and De Groat developed a VHDL model for an embedded controller, which supported debugging at the application software (51). In order to speed up the design validation process, Naganuma et al. (52) combined structured analysis approaches (34) with algorithmic debugging techniques to speed up the design validation process. Potkonjak et al. (53) proposed a technique for design-for-debugging. Their technique focuses only on the error diagnosis phase. It assumes that the designer specifies the debugging variables at design time and only provides controllability and observability of the specified variables. The technique is applicable only to hard-wired ASIC designs and assumes a single functional fault tolerance. Hosseini et al. proposed a code generation methodology and analysis for improved functional verification of microprocessors (54). Finally, Kirovski and Potkonjak developed an approach that coordinates in-circuit emulation and simulation using a cut-based computation state transferring mechanism. They have shown that the effective combination of simulation and emulation can be achieved both for *ASIC* (37) and programmable

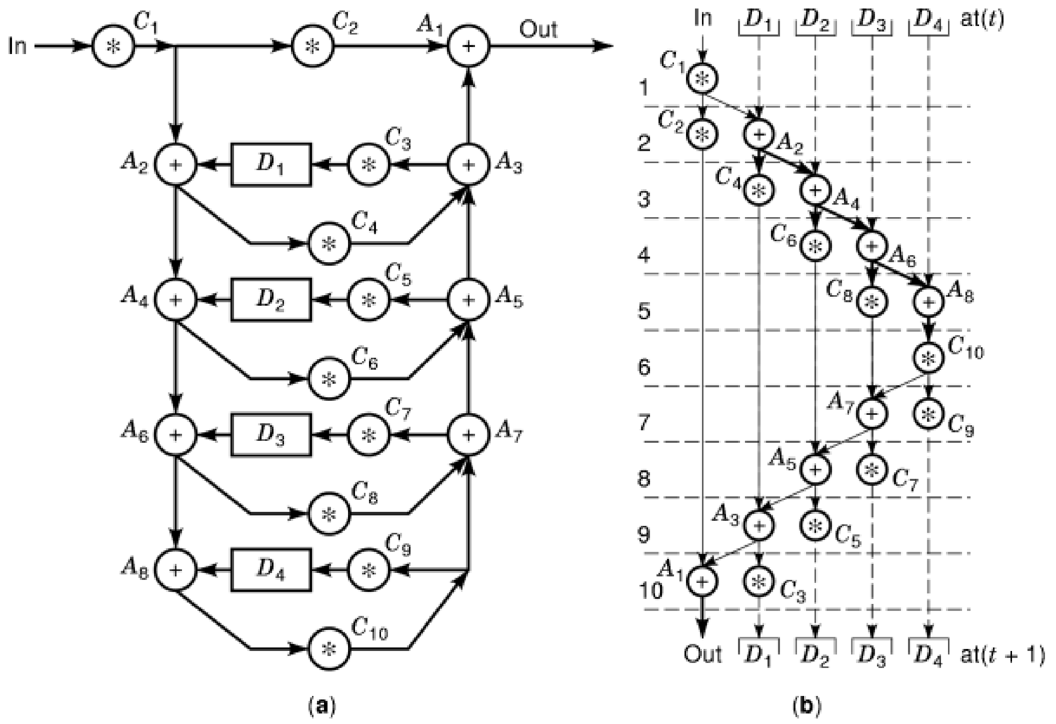


Fig. 9. Optimal cut example for fourth-order CF IIR filter: (a) CDFG and (b) scheduled CDFG.

system-on-silicon (38) design verification. It has been proven experimentally that this approach enables both fast functional execution and complete controllability and observability of the design under debugging with minimal hardware overhead.

Emerging Debugging Techniques. In the remainder of this section, the way, how their approach is conducted, is explained as are the enabling mechanisms for switching the functional execution between the two functional execution domains. First, the notion of a complete cut of a computation is introduced. Then, using an explanatory example, the design-for-debugging trade-offs are elucidated.

A complete cut is a set of variables generated within one computation iteration, which bisects all possible paths in the CDFG. Such a set of variables fully determines the state of the machine between two iterations (i.e., the set of primary inputs and the optimal cut of a particular iteration fully determine the set of primary outputs of the same computation iteration). Thus, a complete cut corresponds to a subset of variables in the computation that bisect all paths between the states that delimit successive iterations of the computation. Clearly, if one has complete controllability and observability over the values for all variables in the complete cut for a specific breakpoint, the computation can be continued functionwise correctly from that breakpoint. In a sense, the cut contains the complete information about the complete history of the computation process and its primary inputs until a given point in time (breakpoint).

An Explanatory Example. The diagnosis approach and accompanying optimization issues are illustrated using fourth-order continued fraction infinite impulse response (*CF IIR*) filter (55). Figure 9(a) shows the CDFG for this popular filter structure. In Fig. 9(b) the computation control flow graph is presented with respect to the particular control step.

Recall that the goal of the design-for-debugging step is to allocate minimal hardware resources that enable computation observability and controllability. One possible solution would be to select variables D_1, D_2, D_3, and

D_4 [dotted lines in Fig. 9(b)] as a complete cut. In this case, because all variables of the cut are concurrently alive, they have to be stored in four different registers. In order to provide full controllability and observability of the design, the designer must have the ability to read/write into those registers from the designated I/O pins. Because four registers are in the cut, we must allocate four sets of register-to-I/O connections in order to enable variable observability and controllability.

If the cut is defined among the output variables of adders A_2, A_4, A_6, and A_8 [bold lines in Fig. 9(b)], only one register is required to hold the values of all those variables because they are not alive simultaneously during the computation. In this case, only one instance of the selection hardware is dedicated to the register that holds the cut. Cut dispensing is performed in four consecutive control steps (cycles 2, 3, 4, and 5). Note that in both cases the variable included in the cut but not mentioned is the actual output of the chip. The primary output of the computation is always used as a part of the complete cut because its dispensing is inevitable.

Considering Testability in High Level Synthesis

Recently, the importance and advantages of addressing testability early in the design process has been established. The topic has attracted attention in both the testing and high level synthesis research literature. A summary of the developments in this emerging field can be found in Ref. 56. Even though several scheduling, assignment, and allocation algorithms for testability enhancement have been proposed, only limited attention has been paid to exploring the relationship between other high level synthesis tasks, like transformations and partitioning, and testability.

Clearly, it is important to evaluate the potential of high level synthesis techniques for testability improvements during the high level design process. Here, a set of techniques are presented. They use a variety of transformations to reduce the area overhead required by design-for-testability (*DFT*) techniques to make the final implementation highly testable. During testability optimization, area minimization and timing (throughput) constraints are simultaneously targeted.

In the last few years, numeral high level synthesis approaches that address testability at the behavioral level have been reported (56). Whereas several systems target built-in-self-test (*BIST*) and hierarchical testability as the test strategy, a majority of high level synthesis techniques explore the relationship between hardware sharing and sequential automatic test pattern generation (*ATPG*) methods. The presence of loops in a sequential circuit is a major source of problems for sequential ATPG. Partial scan is an effective technique to break loops in the circuit by scanning a subset of flip-flops (*FF*s) (57,58). Empirical evidence shows that breaking all nontrivial sequential cycles (with at least two FFs) is an effective heuristic for making a circuit highly testable (57,58). An approach that minimizes the number of scan registers required to break all nontrivial loops in the data path is described in more detail later. This number is used as a measure of testability.

It has been demonstrated that a design can be synthesized from a high level specification in such a way that a relatively small increase in the area of a data path can often be sufficient to reduce significantly the cardinality of the minimum feedback vertex set (*MFVS*) and, hence, the number of scan-registers needed to break all loops (59). This is achieved by alternating classical scheduling and assignment algorithms in such a way that the testability overhead is taken into account (60).

The mandatory tasks during high level synthesis are allocation, scheduling, and assignment (1,61), all of which have been shown to have a significant impact on the testability of the synthesized designs. Existing high level synthesis for testability techniques can be broadly classified according to the testing methodology targeted: BIST, gate-level sequential ATPG, or hierarchical test pattern generation.

Several research groups (57,62) have developed high level synthesis systems that target sequential ATPG testability. These systems synthesize data paths without loops, by using proper scheduling and assignment, and scan registers to break loops (63,64). A data path synthesized from a behavioral specification may contain several types of loops (e.g., CDFG, assignment, and sequential false loops) (64). A CDFG loop is formed in

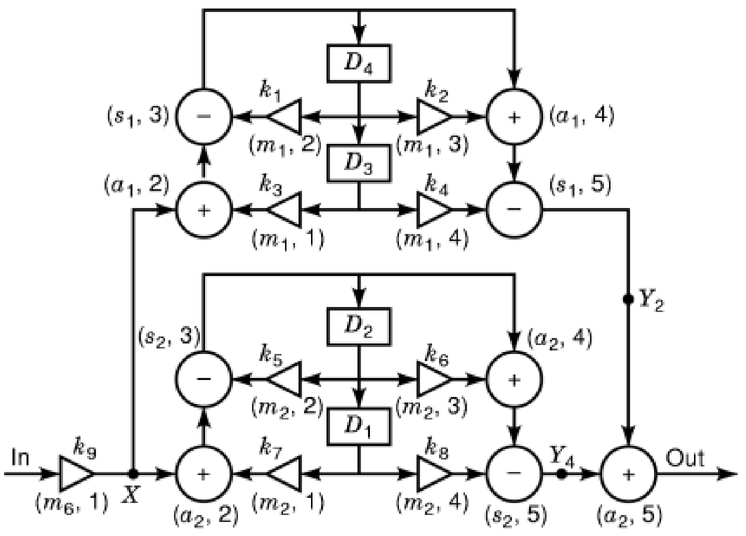


Fig. 10. Motivational example for demonstrating use of transformation for testability: fourth-order parallel IIR filter.

the data path when there exists a cycle consisting of data-dependency edges in the CDFG. The other types of loops are introduced in the data path during high level synthesis, specifically as a result of hardware sharing. For instance, when operations along a CDFG path from operation u to operation v are assigned to n separate modules, with u and v assigned to the same module, an assignment loop of length n is created in the data path. A comprehensive analysis of the formation of loops, in circuits synthesized by high level synthesis techniques, is presented in Ref. 64.

Recently, Bhatia and Jha (65) reported high level synthesis techniques that target hierarchical test pattern generation. Other works addressing testability during high level design are related to the use of testability analysis to guide test statement and test hardware insertion (66,67).

Transformations are changes in the structure of a computation so that a particular objective is achieved, whereas the functional and timing dependencies between the inputs and the outputs are preserved. Recently, a new transformation technique that increases the complexity of the behavioral description while reducing the structural complexity of the resulting data path was developed (60). Application of the new transformation technique to reduce the partial scan overhead for generating easily testable data paths was demonstrated (59).

An Explanatory Example of Testability in High Level Synthesis. The use of transformations for testability optimization is illustrated using the CDFG of the four-order parallel IIR filter, shown in Fig. 10. It is assumed that each operation takes one control cycle. The available time is six control cycles. The critical path is also six cycles long. To meet the available time of six clock cycles, the minimal resource allocation requires three multipliers (three multiplications have to be scheduled in the first control step), two adders (two additions must be scheduled simultaneously in the second control step), and two subtracters (two subtractions must be scheduled in third control step simultaneously).

The result of scheduling and assignment, using an existing behavioral test synthesis system, BETS (64), is shown in Fig. 10. For instance, the first operation, a multiplication by k_9, is assigned to multiplier m_3, and scheduled in control cycle 1, shown by the tuple (m_3, 1). To generate the easily testable design shown in Fig. 11 (64), BETS performs allocation, scheduling, and assignment, simultaneously considering testability overhead and area of the implementation. The resulting minimum feedback set contains four scan registers shown shaded in Fig. 11 .

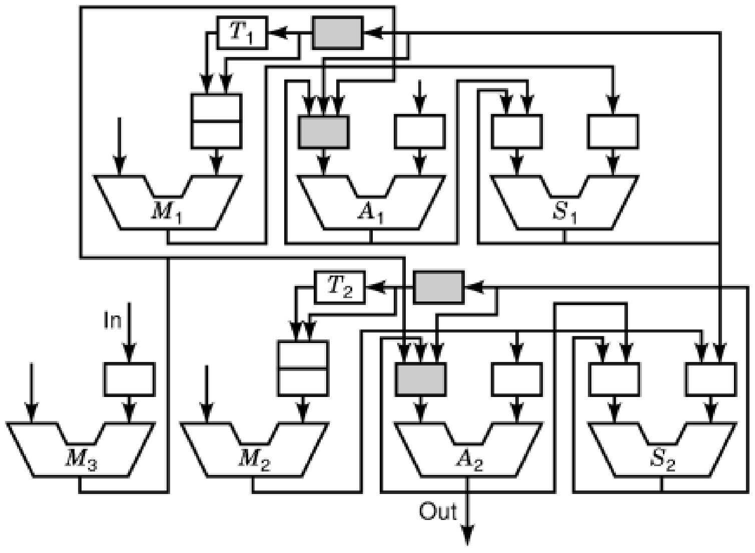
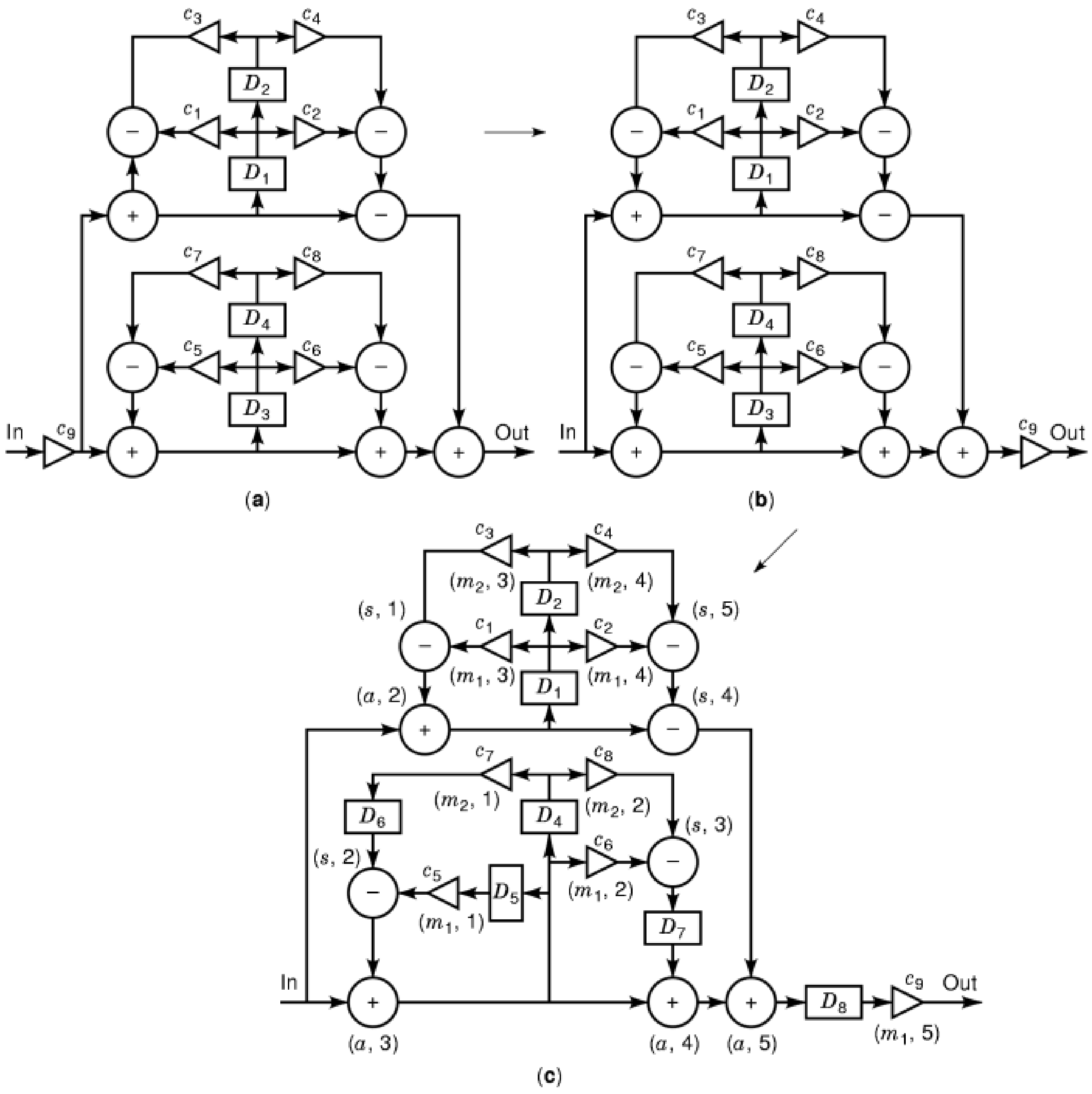


Fig. 11. The data path of the initial fourth-order parallel IIR filter. When the shaded registers are scanned, all loops in the data path are broken.

It can be shown that it is impossible to reduce the number of scan registers using any other scheduling and assignment. Consider the lower biquad in Fig. 10. In order to break the loops in the corresponding part of the data path, at least two scan registers are required. If the register file of the transfer unit on which delay D_1 is assigned is selected, then all the CDFG loops are broken. However, in this case, another scan register is required to break assignment loops, which are formed when the two + operations have to be assigned to the same adder A_2. If the input variable to the transfer unit is not selected for scanning, then it is obvious that at least two scan registers are needed just for breaking the CDFG loops. Similarly, it can be shown that a minimum of two scan registers are needed for the upper biquad in Fig. 10. Note that the variables corresponding to delays cannot share the same scan register because they are simultaneously alive in the first control step.

Consider now how transformations can be used to reduce simultaneously the area and testability cost of the design. A sequence of transformations shown in Fig. 12 is applied, so that eventually the CDFG shown in Fig. 12(c) is obtained. First, algebraic transformations are applied to the CDFG in Fig. 10 to obtain the functionally equivalent CDFG shown in Fig. 12(a). It can be shown that there is a correspondence between the coefficients c_i and k_i used in the two structures. The next transformation applied is the scaling operation (13), where two feedforward cuts are multiplied by b and $1/b$, to obtain the CDFG shown in Fig. 12(b). Finally, retiming is used to relocate the delay D_3 to three new positions D_5, D_6, and D_7. In addition, the CDFG is pipelined by introducing a delay D_8, which is retimed back across the multiplication by c_9. The final CDFG, shown in Fig. 12(c), is significantly more suitable for BETS to produce a testable implementation.

The schedule and assignment obtained by BETS is shown in Fig. 12(c). Note that, although not targeted, the throughput is also improved, because the critical path is reduced to five control steps, and the schedule uses five control steps. The resulting data path is shown in Fig. 13. The hardware requirements are reduced to only two multipliers, one adder, and one subtracter. Simulation using Hyper (68) indicates that all the computational structures of the fourth-order parallel IIR filter, shown in Figs. 10 and 12, have identical numerical properties and therefore identical word-length requirements.



c_1 c_2 c_3 c_4 c_5 c_6 c_7 c_8 c_9
D_1 D_2 D_3 D_4 D_5 D_6 D_7 D_8
In
Out
(a)
(b)
(c)
$(m_2, 3)$ $(m_2, 4)$ $(s, 1)$ $(s, 5)$ $(m_1, 3)$ $(m_1, 4)$ $(a, 2)$ $(s, 4)$ $(m_2, 1)$ $(m_2, 2)$ $(s, 3)$ $(s, 2)$ $(m_1, 2)$ $(m_1, 1)$ $(a, 3)$ $(a, 4)$ $(a, 5)$ $(m_1, 5)$

Fig. 12. The sequence of transformations for the simultaneous optimization of testability and area. First (a) associativity and the inverse element law are applied; next (b) scaling transformation; and (c) finally retiming. The data path corresponding to the final CDFG is shown in Fig. 13.

High Level Synthesis for Low Power

Low power consumption has been established as an important design metric for VLSI design mainly because the remarkable success and growth of the class of personal computing devices and wireless communication systems, which demands high-speed computation and complex functionality with low power consumption.

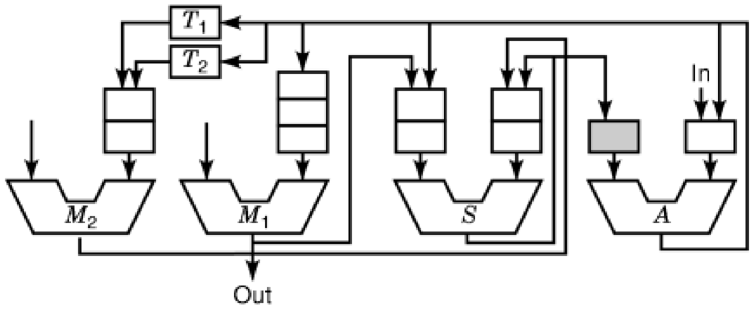


Fig. 13. The data path of the transformed fourth-order parallel IIR. Only one scan register (shaded) is required to break all loops in the data path, compared with four scan register in the initial design.

Recent work (19,69,70,71) has shown that the most savings in power consumption are often obtained at the higher levels of design hierarchy, in particular at the high level synthesis.

In CMOS technology, there are three sources of power dissipation: switching, short-circuit, and leakage currents. The switching component is the only one that cannot be made negligible if proper design techniques are followed (69). Average switching power is given by $P = \alpha C_L V^2{}_{\text{dd}} f$, where f is the system clock frequency, V_{dd} is the supply voltage, C_L is the load capacitance, and α is the switching activity (the probability of a $0 \to 1$ transition during a clock cycle) (69). The product term αC_L is called the effective switched capacitance. The equation for power consumption implies that optimizing for power entails an attempt to reduce one or more of the three degrees of freedom: supply voltage, switching activity, and physical capacitance. The supply voltage, V_{dd} offers the most effective means to minimize power consumption because of its quadratic contribution. An unfortunate side-effect of decreasing V_{dd} is that the delay of the circuit increases, which can be shown to be $k[V_{\text{dd}}/(V_{\text{dd}}) - V_{\text{T}})^2]$, where V_{T} is the device threshold voltage and k is a technology-dependent constant (69). Reducing switching activity and physical capacitance can also be used to reduce power consumption, although the effect is not as drastic as that of supply voltage reduction.

High Level Power Estimation. The availability of efficient and accurate high level power estimation tools is key for increasing the effectiveness of high level synthesis. High level power estimation addresses the problem of estimating the power consumed by a design from a high level description. Because the gate level structure of the design components is unavailable for power estimation techniques at the high level, the estimation techniques usually rely on some abstract notions of switching activity and physical capacitance.

The approaches for high level power estimation can be categorized into four groups. First, information–theoretic models depend on information–theoretic measures of activity such as entropy, which characterizes the randomness of a sequence of inputs, to obtain fast power estimates (72,73). It is intuitive that if the switching activity for a sequence of inputs is high, the sequence is likely to be random, resulting in high entropy. Average entropy plays a key role in estimating physical capacitance and switching activity. Second, complexity-based models rely on the assumption that the power consumption of a design can be related to its complexity, which can be determined by such parameters as the number and type of operations and the number of states and transitions in a controller description. Third, profiling-based models capture relevant data statistics such as the number of operations of a given type, bus and memory accesses, and I/O operations by profiling based either on stochastic analysis of the high level description and input streams (19) or on direct simulation of the design under a typical input stream (74). Last, power macro-modeling methods use regression-based or sampling-based switched capacitance models for circuit modules (75,76).

High Level Power Optimization. High level power optimization techniques have been proposed for all the various high level synthesis steps. Some techniques deal with only one high level synthesis step whereas others consider several synthesis steps simultaneously.

Several works (77,78) present scheduling algorithms, where operations are scheduled to reduce the switching activity of the functional units by minimizing the switching activity of their input operands. The scheduling algorithm proposed in Ref. 79 attempts to schedule operations so that the algorithm maximizes the possibility of shutting down resources that are not performing useful computations in a given control step to avoid unnecessary power dissipation. A number of research groups have addressed the scheduling problem with multiple (in their software implementation restricted to two or three) different voltages (80,81,82,83).

In the resource allocation procedure, functional units, registers, and interconnections are assigned to variables and operations. The power dissipated by such resources usually depends on the input switching activity induced by the data being stored or processed. Assuming that the probability density functions of the inputs at the functional units. Chang and Pedram (84) formulated the allocation problem for functional units as a max-cost multicommodity network flow problem, which can be solved optimally. Musoll and Cortadella (78) also proposed a resource-binding algorithm to reduce the switching activity of functional units. Chang and Pedram (85) proposed an effective graph-based register allocation algorithm based on an accurate computation of the probability density functions at the inputs of various resources, given the probability distributions for the system primary inputs. Raghunathan and Jha (86) demonstrated how hardware sharing affects the capacitance and switching activity, and hence the power dissipation of resources and proposed an Integer Linear Programming (*ILP*) formulation for the allocation problem, simultaneously considering registers and functional units. Mehra et al. (87) proposed an allocation algorithm to reduce power consumption in interconnects by exploiting locality in a computation. Locality in a computation relates to the degree to which a computation has natural clusters of operations (87).

Several authors noted that the various high level synthesis steps must be considered simultaneously to explore the possible trade-offs resulting from their interaction. Raghunathan and Jha (71) proposed an iterative improvement algorithm that considers scheduling, clock and module selection, and resource allocation simultaneously. Mehra and Rabaey also proposed the scheduling and resource allocation techniques for exploiting regularity in a computation to reduce power consumption in interconnects. Regularity in a computation refers to the repeated occurrence of the computational patterns such as a multiplication followed by an addition (88).

Transformations alter the structure of a computation in such a way that the user-specified input/output relationship is maintained (8). The problem of automatically finding computational structures that result in the lowest power consumption for a specified throughput using transformations has been presented by Chandrakasan et al. (19,89). They have used two key approaches: reducing the supply voltage by utilizing speed-up transformations and reducing the capacitance being switched using a variety of transformations. Most of the transformations used are basically the same as the high level transformations targeting area and speed, but the different cost functions are used to evaluate the results obtained through such transformations.

In Refs. 77 and 90, high level transformation techniques such as loop interchange, operand reordering, and loop folding are used to reduce the switching activity of functional units. Some high level transformations to reduce off-chip memory references have been proposed by Ref. 22.

It is well known that operations whose corresponding hardware implementations require less energy per computation than others exist. For example, multiplications usually require more energy than additions. Therefore, strength reduction transformations are used to substitute constant multiplications with shifts and additions/subtractions. Chandrakasan et al. (19) demonstrated the effectiveness of the transformation by showing a significant reduction in power through applying the transformation on several designs. Potkonjak et al. (91) extended the transformation by considering multiple constant multiplications simultaneously.

One of the most effective ways to reduce the total switched capacitance consists of reducing the number of operations in the CDFG. Various transformations can be used to reduce the number of operations. Figure 14 illustrates how the number of operations is reduced by applying distributivity transformation. Potkonjak and Rabaey (13) addressed the minimization of the number of multiplications and additions in linear computations in their maximally fast form so that the throughput is preserved. Potkonjak et al. (91) presented a set of techniques for minimization of the number of shifts and additions in linear computations. Sheliga and Sha (92)

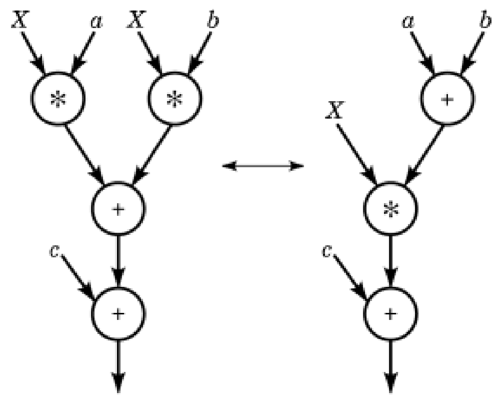


Fig. 14. An example to show how the number of operations is reduced by distributivity transformation.

presented an approach for minimization of the number of multiplications and additions in linear computations. Srivastava and Potkonjak (30) developed an approach for the minimization of the number of operations in linear computations using unfolding and the application of the maximally fast procedure.

Hong et al. (93) introduced an approach for power optimization using a set of compilation techniques. They proposed a novel divide-and-conquer compilation technique to minimize the number of operations for general computations. Their technique not only optimizes a significantly wider set of computations than the previously published techniques but also outperforms (or performs at least as well as other techniques) on all their design examples. A variant of the technique by Ref. (30) is used in the "conquer" phase of their approach. Their approach is different from Ref. (30) in two respects. First, the technique of Ref. (30) can handle only very restricted computations which are linear, whereas their approach can optimize *arbitrary* computations. Second, their approach outperforms or performs at least as well as the technique of Ref. (30) for linear computations.

We illustrate the key ideas of their approach for minimizing the number of operations by considering the computation of Fig. 15(a). Each node represents a subpart of the computation. They make the following assumptions only specifically for clarifying the presentation of this example and simplifying the example. Assume that each subpart is linear and can be represented by state-space equations. Also assume that every subpart is dense, which means that every output and state in a subpart is a linear combination of all inputs and states in the subpart with no 0, 1, or -1 coefficients. The number inside a node is the number of delays or states in the subpart. Assume that when there is an arc from a subpart X to a subpart Y, every output and state of Y depends on all inputs and states of X.

The number of operations per input sample is initially 2081, where the number of operations of the computation is calculated as that of the maximally fast tree representation (13) for the computation. Using the technique of Ref. 30, which unfolds the entire computation, the number can be reduced to 725 with an unfolding factor of 12. Their approach optimizes each subpart separately. This separate optimization is enabled by isolating the subparts using pipeline delays. Figure 15(b) shows the computation after the isolation step. Because every subpart is linear, unfolding is performed to optimize the number of operations for each subpart. Unfolding results in simultaneous processing of consecutive iterations of a computation. The subparts P_1, P_2, P_3, P_4, and P_5 cost 120.75, 53.91, 114.86, 129.75, and 103.0 operations per input sample with unfolding factors 3, 10, 6, 7, and 2, respectively. The total number of operations per input sample for the entire computation is 522.27. Subpart merging is now applied to further reduce the number of operations. Their heuristic considers merging of adjacent subparts. Initially, the possible merging candidate pairs are P_1P_2, P_1P_3, P_2P_5, P_3P_4, and P_4P_5, which produce the gains of -51.48, -112.06, -52.38, 122.87, and -114.92, respectively. P_3 and P_4 are merged with an unfolding factor of 22. In the next iteration, there are now four subparts and four candidate

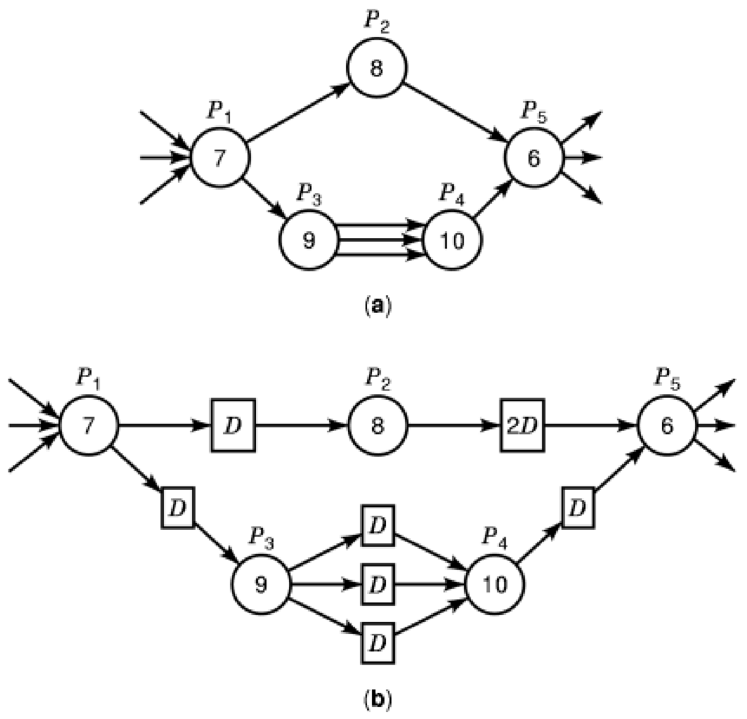


Fig. 15. An explanatory example to show the minimization of the number of operations by the divide-and-conquer method: (a) the initial computation and (b) the computation after the isolation by pipelining.

pairs for merging all of which yield negative gains. So, the heuristic stops at this point. The total number of operations per input sample has further decreased to 399.4. Their approach has reduced the number of operations by a factor of 1.82 from the previous technique of Ref. 30, while it has achieved the reduction by a factor of 5.2 from the initial number of operations.

For experimental results, they used many real-life designs, which include all the benchmark examples used in Ref. (30) as well as the following typical portable DSP, video, communication, and control applications: DAC—four-stage NEC digital-to-analog converter (*DAC*) for audio signals; modem—two-stage NEC modem; GE controller—five-state GE linear controller; APCM receiver—Motorola's adaptive pulse code modulation receiver; Audio Filter 1—analog-to-digital converter (ADC) followed by 14-order cascade IIR filter; Audio Filter 2—ADC followed by 18-order parallel filter; Video Filter 1—two ADCs followed by 10-order two-dimensional IIR filter; Video Filter 2—two ADCs followed by 12-order 2D IIR filter; and VSTOL—VSTOL robust observer structure aircraft speed controller. DAC, modem, and GE controller are linear computations, and the rest are nonlinear computations. The benchmark examples from Ref. (30) are all linear, which include ellip, iir5, wdf5, iir6, iir10, iir12, steam, dist, and chemical. Their method has reduced the number of operations by an average factor of 1.77 (average 43.5%) for the examples that previous techniques are either ineffective or inapplicable.

Conclusion

In this survey, we have provided a nonexhaustive review of existing methodologies and tools for high level synthesis, mainly concentrating on research works targeting new set of design metrics such as power, testability, and debuggability. We expect this field to remain quite active in the foreseeable future.

BIBLIOGRAPHY

1. M. C. McFarland, A. C. Parker, R. Composano, The high level synthesis of digital systems, *Proc. IEEE*, **78**: 301–317, 1990.
2. R. A. Walker, R. Camposano, *A Survey of High-level Synthesis Systems*, Norwell, MA: Kluwer, 1991.
3. D. D. Gajski *et al.*, *High-level Synthesis: Introduction to Chip and System Design*, Norwell, MA: Kluwer, 1992.
4. G. De Micheli, *Synthesis and Optimization of Digital Circuits*, New York: McGraw-Hill, 1994.
5. Y.-L. Lin, Recent developments in high-level synthesis, *ACM Trans. Des. Autom. Electron. Syst.*, **2** (1): 2–21, 1997.
6. W. H. Wolf, R. O'Donnell, Hardware-software co-design of embedded systems (and prolog), *Proc. IEEE*, 82: 965–989, 1994.
7. E. A. Lee, T. M. Parks, Dataflow process networks, *Proc. IEEE*, **83**: 773–801, 1995.
8. M. C. McFarland, Formal analysis of correctness of behavioral transformations, *Formal Methods Syst. Des.*, **2** (3): 231–257, 1993.
9. U. Banerjee *et al.*, Automatic program parallelization, *Proc. IEEE*, **81**: 211–243, 1993.
10. D. F. Bacon, S. L. Graham, O. J. Sharp, Compiler transformations for high performance computing, *ACM Comput. Surv.*, **26** (4): 345–420, 1994.
11. K. K. Parhi, High-level algorithm and architecture transformations for DSP synthesis, *J. VLSI Signal Process.*, **9** (1–2): 121–143, 1995.
12. D. C. Ku, G. De Micheli, *High Level Synthesis of ASICs under Timing and Synchronization Constraints*, Norwell, MA: Kluwer, 1992.
13. M. Potkonjak, J. Rabaey, Maximally fast and arbitrarily fast implementation of linear computations, *IEEE/ACM Int. Conf. Comput.-Aided Des.*, 1992, pp. 304–308.
14. L. Guerra, M. Potkonjak, J. Rabaey, High level synthesis for reconfigurable datapath structures, *IEEE/ACM Int. Conf. Comput.-Aided Des.*, 1993, pp. 26–29.
15. M. Potkonjak, S. Dey, R. K. Roy, Considering testability at behavioral level: Use of transformations for partial scan cost minimization under timing and area constraints, *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, **14**: 531–546, 1995.
16. M. Potkonjak, M. B. Srivastava, Behavioral synthesis of high performance, low cost, and low power application specific processors for linear computations, *Int. Conf. Appl. Specific Array Processors*, 1994, pp. 45–56.
17. K. K. Parhi, D. G. Messerschmitt, Static rate-optimal scheduling of iterative dat-flow programs via optimum unfolding, *IEEE Trans. Comput.*, **40**: 178–195, 1991.
18. M. B. Srivastava, M. Potkonjak, Transforming linear systems for joint latency and throughput optimization, *Eur. Des. Autom. Conf.*, 1994, pp. 267–271.
19. A. P. Chandrakasan *et al.*, Optimizing power using transformations, *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, **14**: 12–31, 1995.
20. R. Karri, A. Orailoglu, High-level synthesis of fault-secure microarchitectures, *Des. Autom. Conf.*, 1993, pp. 429–433.
21. A. Orailoglu, R. Karri, Coactive scheduling and checkpoint determination during high level synthesis of self-recovering microarchitectures, *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, **2**: 304–311, 1994.
22. F. Catthoor *et al.*, Global communication and memory optimizing transformations for low power signal processing systems, *VLSI Signal Process.*, **7**: 178–187, 1994.
23. A. S. Tanenbaum, H. van Straven, J. W. Stevenson, Using peephole optimization on intermediate code, *ACM Trans. Programm. Lang. Syst.*, **4** (1): 21–36, 1982.
24. J. D. Ullman, *Database and Knowledge-Based Systems*, Rockville, MD: Computer Science Press, 1989, vol. 2.
25. H. Massalin, Superoptimizer: A look at the smallest program, *Int. Conf. Arch. Support Programm.Lang. Oper. Syst.*, 1987, pp. 122–126.
26. M. E. Wolf, M. S. Lam, A loop transformation theory and an algorithm to maximize parallelism, *IEEE Trans. Parallel Distrib. Syst.*, 2: 452–471, 1991.
27. G. C. Fox, J. G. Koller, Code generation by a generalized neural network, *J. Parallel Distrib. Comput.*, **7** (2): 388–410, 1989.
28. Z. Iqbal *et al.*, Critical path minimization using retiming and algebraic speedup, *Des. Autom. Conf.*, 1993, pp. 573–577.
29. D. Whitfield, M. L. Soffa, An approach to ordering optimizing transformations, *ACM Symp. Prin. Pract. Parallel Program.*, 1990, pp. 137–147.

30. M. Srivastava, M. Potkonjak, Power optimization in programmable processors and ASIC implementations of linear systems: Transformation-based approach, *Des. Autom. Conf.*, 1996, pp. 343–348.
31. S.-H. Huang, J. M. Rabaey, Minimizing the throughput of high performance DSP applications using behavioural transfomrations, *EDAC-ETC-EUROASIC*, 1994, pp. 25–30.
32. I. Hong, D. Kirovski, M. Potonjak, Potential-driven statistical ordering of transformations, *Des. Autom. Conf.*, 1997, pp. 347–352.
33. L. Yang *et al.*, System design methodology of UltraSPARC-I, *Des. Autom. Conf.*, 1995, pp. 7–12.
34. S. Narayan *et al.*, System specification and synthesis with the SpecCharts language, *IEEE/ACM Int. Conf. Comput.-Aided Des.*, 1991, pp. 266–269.
35. A. Yu, The future of microprocessors, *IEEE Micro*, **16** (6): 46–53, 1996.
36. K. Saab *et al.*, LIMSoft: Automated tool for sensitivity analysis and test vector generation, *IEEE Proc. Circuits, Devices Syst.*, **143**: 386–392, 1996.
37. D. Kirovski, M. Potkonjak, Quantitative approach to functional debugging, *IEEE/ACM Int. Conf. Comput.-Aided Des.*, 1997.
38. D. Kirovski, M. Potkonjak, L. M. Guerra, *Functional debugging of systems-on-silicon*, unpublished manuscript, Computer Science Department, UCLA, 1998.
39. M. Rosenblum *et al.*, Complete computer system simulation: The SimOS approach, *IEEE Parallel Distrib. Technol.: Syst. Appl.*, **3** (4): 34–43, 1995.
40. V. Zivojnovic, H. Meyr, Compiled HW/SW co-simulation, *Des. Autom. Conf.*, 1996, pp. 690–695.
41. [Online], Available http://www.interrainc.com/picasso.html
42. [Online], Available http://www.synopsys.com/products/simulation/cyclone-cs.html
43. S. T. Mangelsdorf *et al.*, Functional verification of the HPA PA 8000 processor, *Hewlett-Packard J.*, August, 1997; [Online], Available http://www.hp.com:80/hpj/97aug/au97a3.pdf
44. C. Maunder, JTAG, the Joint Test Action Group, *IEEE Colloq. New Ideas Test.*, 1986, pp. 6/1–4.
45. P. C. Ching, Y. C. Cheng, M. H. Ko, An in-circuit emulator for TMS320C25, *IEEE Trans. Educ.*, **37**: 51–56, 1994.
46. [Online], Available http://www.aptox.com:80/Products/mp4.html
47. [Online], Available http://www.quickturn.com/prod/hdlice/hdlice.htm
48. K. Olukotun *et al.*, Hydra project, [Online], Available http://ogun.stanford.edu
49. A. Saini, Design of the Intel Pentium (tm) Processor, *Int. Conf. Comput.-Aided Des.*, 1993, pp. 258–261.
50. J. D. Myers, J. L. Rivero, *System for optimal electronic debugging and verification employing scheduled cutover of alternative logic simulations*, US Patent No. 5,566,097, 1993.
51. G. S. Powley, J. E. DeGroat, Experience in testing and debugging the i960 MX VHDL model, *VHDL Int. Users Forum*, 1994, pp. 130–135.
52. J. Naganuma *et al.*, High-level design validation using algorithmic debugging, *EDAC*, 1994, pp. 474–480.
53. M. Potkonjak, S. Dey, K. Wakabayashi, Design-for-debugging of application specific designs, *IEEEACM Int. Conf. Comput.-Aided Des.*, 1995, pp. 295–301.
54. A. Hosseini, D. Mavroidis, P. Konas, Code generation and analysis for the functional verification of microprocessors, *Des. Autom. Conf.*, 1996, pp. 305–310.
55. R. E. Crochiere, A. V. Oppenheim, Analysis of linear networks, *Proc. IEEE*, **63**: 581–595, 1975.
56. L. J. Avra, E. J. McCluskey, High level synthesis of testable designs: An overview of university systems, *Int. Test Conf.*, 1994.
57. K. T. Cheng, V. D. Agrawal, A partial scan method for sequential circuits with feedback, *IEEE Trans. Comput.*, **39**: 544–548, 1990.
58. D. H. Lee, S. M. Reddy, On determining scan flip-flops in partial-scan designs, *IEEE Int. Conf. Comput.-Aided Des.*, 1990, pp. 322–325.
59. S. Dey, M. Potkonjak, R. K. Roy, Synthesizing designs with low-cardinality minimum feedback vertex set for partial scan application, *VLSI Test Symp.*, 1994, pp. 2–7.
60. S. Dey, M. Potkonjak, Transforming behavioral specifications to facilitate synthesis of testable designs, *Int. Test Conf.*, 1994.
61. C. Papachristou *et al.*, SYBNTEST: A method for high-level SYNthesis with self TESTability. *IEEE/ACM Int. Conf. Comput.-Aided Des.*, 1991, pp. 458–462.

62. A. Majumdar, K. Saluja, R. Jain, Incorporating testability considerations in high-level synthesis, *Int. Symp. Fault-Tolerant Comput.*, 1992.
63. T. C. Lee, N. K. Jha, W. H. Wolf, Behavioral synthesis of highly testable data paths under non-scan and partial scan environment, *Des. Autom. Conf.*, 1993, pp. 292–297.
64. S. Dey, M. Potkonjak, R. Roy, Exploiting hardware-sharing in high level synthesis for partial scan optimization, *IEEE/ACM Int. Conf. Comput.-Aided Des.*, 1993, pp. 20–25.
65. S. Bhatia, N. K. Jha, Genesis: A behavioral synthesis system for hierarchical testability, *Eur. Des. Test Conf.*, 1994, pp. 272–276.
66. P. Vishakantaiah, J. A. Abraham, M. Abadir, Automatic test knowledge extraction from VHDL (ATKET), *Des. Autom. Conf.*, 1992, pp. 273–278.
67. C.-H. Chen, D. G. Saab, BETA: Behavioral testability analysis, *IEEE/ACM Int. Conf. Comput.-Aided Des.*, 1991, pp. 202–205.
68. J. Rabaey *et al.*, Fast prototyping of datapath-intensive architectures, *IEEE Des. Test Comput.*, **8** (2): 40–51, 1991.
69. A. P. Chandrakasan, S. Sheng, R. W. Broderson, Low-power CMOS digital design, *IEEE J. Solid-State Circuits*, **27**: 473–484, 1992.
70. R. Mehra, J. Rabaey, Behavioral level power estimation and exploration, *Int. Workshop Low Power Des.*, 1994, pp. 197–202.
71. A. Raghunathan, N. K. Jha, An iterative improvement algorithm for low power data path synthesis, *Int. Conf. Comput.-Aided Des.*, 1995, pp. 597–602.
72. D. Marculescu, R. Marculescu, M. Pedram, Information theoretic measures for power analysis, *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, **15**: 599–610, 1996.
73. M. Nemani, F. N. Najm, Towards a high-level power estimation capability, *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, **15**: 588–598, 1996.
74. N. Kumar *et al.*, Profile-driven behavioral synthesis for low power VLSI systems, *IEEE Des. Test Comput.*, **12**: 70–84, 1995.
75. S. Gupta, F. N. Najm, Power macromodeling for high level power estimation, *Des. Autom. Conf.*, 1997, pp. 365–370.
76. P. E. Landman, J. M. Rabaey, Architectural power analysis: The dual bit type method, *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, **3**: 173–187, 1995.
77. E. Musoll, J. Cortadella, High-level synthesis techniques for reducing the activity of functional units, *Int. Symp. Low Power Des.*, 1995, pp. 99–104.
78. E. Musoll, J. Cortadella, Scheduling and resource binding for low power, *Int. Symp. Syst. Synth.*, 1995, pp. 104–109.
79. J. Monteiro *et al.*, Scheduling techniques to enable power management, *Des. Autom. Conf.*, 1996, pp. 349–352.
80. J.-M. Chang, M. Pedram, Energy minimization using multiple supply voltages, *Int. Symp. Low Power Electron. Des.*, 1996, pp. 157–162.
81. M. C. Johnson, K. Roy, Datapath scheduling with multiple supply voltages and level converters, *ACM Trans. Des. Autom. Electron. Syst.*, **2**: 1997.
82. Y.-R. Lin, C.-T. Hwang, A. C.-H. Wu, Scheduling techniques for variable voltage low power designs, *ACM Trans. Des. Autom. Electron. Syst.*, **2** (2): 81–97, 1997.
83. S. Raje, M. Sarrafzadeh, Variable voltage scheduling, *Int. Symp. Low Power Des.*, 1995, pp. 9–14.
84. J.-M. Chang, M. Pedram, Module assignment for low power, *EURO-DAC Eur. Des. Autom. Conf.*, 1996, pp. 376–381.
85. J.-M. Chang, M. Pedram, Register allocation and binding for low power, *Des. Autom. Conf.*, 1996, pp. 29–35.
86. A. Raghunathan, N. K. Jha, Behavioral synthesis for low power, *Int. Conf. Comput. Des.*, 1994, pp. 318–322.
87. R. Mehra, L. Guerra, J. Rabaey, Low-power architectural synthesis and the impact of exploiting locality, *J. VLSI Signal Process.*, **13** (2–3): 239–258, 1996.
88. R. Mehra, J. Rabaey, Exploiting regularity for low-power design, *Int. Conf. Comput.-Aided Des.*, 1996, pp. 166–172.
89. A. P. Chandrakasan *et al.*, Hyper-lp: A system for power minimization using architectural transformation, *IEEE/ACM Int. Conf. Comput.-Aided Des.*, 1992, pp. 300–303.
90. D. Kim, K. Choi, Power-conscious high level synthesis using loop folding, *Des. Autom. Conf.*, 1997, pp. 441–445.
91. M. Potkonjak, M. Srivastava, A. P. Chandrakasan, Multiple constant multiplications: Efficient and versatile framework and algorithms for exploring common subexpression elimination, *IEEE Trans Comput.-Aided Des. Integr. Circuits Syst.*, **15**: 151–165, 1996.

92. M. Sheliga, E. H.-M. Sha, Global node reduction of linear systems using ratio analysis, *Int. Symp. High-Level Synth.*, 1994, pp. 140–145.
93. I. Hong, M. Potonjak, R. Karri, Power optimization using divide-and-conquer techniques for minimization of the number of operations, *IEEE/ACM Int. Conf. Comput.-Aided Des.*, 1997, pp. 108–113.

INKI HONG
DARKO KIROVSKI
MIODRAG POTKONJAK
University of California at Los Angeles