An Integrated CAD System for Algorithm-Specific IC Design

C. Bernard Shung, Rajeev Jain, *Member, IEEE*, Ken Rimey, Edward Wang, Mani B. Srivastava, Brian C. Richards, Erik Lettang, *Member, IEEE*, S. Khalid Azim, *Member, IEEE*, Lars Thon, *Student Member, IEEE*, Paul N. Hilfinger, Jan M. Rabaey, *Member, IEEE*, and Robert W. Brodersen, *Fellow, IEEE*

***Abstract*—LAGER is an integrated computer-aided design (CAD) system for algorithm-specific integrated circuit (IC) design, targeted at applications such as speech processing, image processing, telecommunications, and robot control. LAGER provides user interfaces at behavioral, structural, and physical levels and allows easy integration of new CAD tools. LAGER consists of a *behavioral mapper* and a *silicon assembler*. The behavioral mapper *maps* the behavior onto a parameterized structure to produce microcode and parameter values. The silicon assembler then translates the filled-out structural description into a physical layout and with the aid of simulation tools, the user can fine tune the data path by iterating this process. The silicon assembler can also be used without the behavioral mapper for high sample rate applications. A number of algorithm-specific IC's designed with LAGER have been fabricated and tested, and as examples, a robot arm controller chip and a real-time image segmentation chip will be described.**

I. Introduction

MODERN integrated circuits (IC's) fall into two groups: *commodity* IC's and *application-specific* IC's (ASIC's). The turn-around time of ASIC's is often more important than the area, emphasizing the need for computer-aided design (CAD) tools. Due to their application-specific nature, each design can exploit the special conditions in the particular application to create an efficient implementation. This paper focuses on ASIC's that implement real-time computational algorithms called *algorithm-specific* IC's. Typical application areas for real-time algorithm-specific IC's include speech processing, image processing, robot control, computer vision, digital audio, and telecommunications. From experience with a number of such IC designs, we find that diverse algorithms can often be implemented with a single, well-designed set of hardware modules and the re-use of these hardware modules greatly reduces design time.

Two types of architectures are used in designing algorithm-specific IC's: *hardwired* architectures and *programmable* architectures. In a hardwired architecture, a dedicated hardware module is allocated for each operation in the data-flow graph of the algorithm; an adder for an *add* operation, for example. In this way, the abstract data flow in the algorithm is realized by the physical interconnection of the hardware modules. Although the data path may be complicated, the control unit is simple because there is no time multiplexing of hardware modules. The speed of the circuit is limited by the speed of the slowest hardware module in the design, but the input of new data (sample rate) can be equal to this value. The main drawback of hardwired architectures is that they have to be redesigned for each new algorithm and become less efficient if complex decision making is required.

A programmable architecture consists of a carefully chosen set of hardware modules that are time multiplexed under microcode control according to the algorithm being implemented. The control unit for a programmable architecture is necessarily more complex than that for a hardwired architecture. The speed of the circuit depends on the total number of instruction cycles required to realize the algorithm. Hence, a programmable architecture can be used in a real-time application only if

$$\text{total number of instruction cycles} \leq \frac{\text{sample period}}{\text{circuit cycle time}}.$$

For example, if the circuit runs at 5 MHz and the sample frequency is 5 kHz, the number of cycles must be no greater than 1000. Because a single programmable architecture can be used for many applications, it is a good choice when the sample rate permits. On the other hand, high sample rate applications require hardwired architectures because only dedicated hardware modules can provide the required speed.

There are three levels of design descriptions for algorithm-specific IC's, namely, *behavioral*, *structural*, and *physical*. A behavioral representation specifies the algorithm that the chip implements, which may take the form of a program or a signal-flow graph. A structural representation specifies the chip architecture in terms of hardware modules and their logical connections. A physical representation specifies the chip layout.

This paper describes LAGER, an integrated CAD system for automatic generation—from high-level (behavioral or structural) user specifications—of chip layouts for hardwired or programmable architectural implementations of algorithm-specific IC's. In Section II we discuss design strategies of LAGER and compare them with previous work. In Sections III and IV we describe the two parts of LAGER: the *behavioral mapper* and the *silicon assembler*, respectively. We will show several design examples in Section V.

Manuscript received April 19, 1989; revised January 18, 1990. This work was supported by DARPA under Grant N00039-87-C-0182. This paper was recommended by Editor M. R. Lightner.

C. B. Shung is with the Department of Electronics Engineering, National Chiao Tung University, Hsinchu, Taiwan, Republic of China.

R. Jain is with the Department of Electrical Engineering, University of California at Los Angeles, Los Angeles, CA 90024.

K. Rimey is with the Department of Computer Science, Helsinki University of Technology, Helsinki, Finland.

E. Wang, M. B. Srivastava, B. C. Richards, L. Thon, P. N. Hilfinger, J. M. Rabaey, and R. W. Brodersen are with the Department of Electrical Engineering and Computer Science, Electronics Research Laboratory, University of California at Berkeley, Berkeley, CA 94720.

E. Lettang is with the Hewlett Packard Company, San Diego, CA.

S. K. Azim is with AT&T Bell Laboratories, Allentown, PA.

IEEE Log Number 9042087.

II. Design Strategies for Algorithm-Specific IC's

In Section II-2.1, we first review the development of some existing CAD tools for algorithm-specific IC's, and motivate the need for a new design environment. In Section II-2.2, we will describe the new problems that are solved, and the basic design methodology adopted in LAGER.

2.1. Background

2.1.1. Behavioral Description: Many behavioral descriptions have been proposed for design capture at the behavioral level. A *frequency-domain* specification is used [1], which allows the users to specify filter parameters such as the passband ripple, stopband ripple, and stopband attenuation, etc. This approach offers the highest level description; the user only specifies *what* he needs rather than *how* to do it. However, the application of this approach is limited to digital filter circuits.

Most algorithm-specific IC applications can be described by *signal-flow diagrams*. Hence, an *applicative* programming language [2], [3] is a natural choice for algorithm specification. It is also inherently parallel, so the same applicative language program can serve both a uniprocessor or a multiprocessor realization. On the other hand, *procedural* programming languages such as Pascal [4] or ISPS [5] have also been adopted, which provide a mechanism for describing the control flow of the algorithm. For high-level procedural languages, existing compilers can be exploited for algorithm simulation. Although inefficient for direct encoding of algorithms, ISPS is very useful for machine description for its low-level mechanisms such as bit operations and timing contol. In LAGER, both an applicative (Silage [2]) and a "C-like" procedural language (RL) are provided. These languages will be described in Section III.

2.1.2. Behavioral Synthesis: Behavioral synthesis [6], [7]–[11] attempts to generate a structural description from a behavioral description directly. Due to the vast amount of design alternatives inherent in the behavioral synthesis systems, some search pruning is mandatory. In most systems, high-level decisions—bus structure [7], pipelined versus nonpipelined architectures [8], lumped ALU versus distributed functional modules [11], [12]—are predetermined to make the problem tractable. These *a priori* design decisions, however, are usually too restrictive to deliver efficient designs. Also, because the end product of most behavioral synthesis research is a block diagram instead of a layout, its performance evaluation often lacks practical considerations. For example, it has been found [13] that the data-path partitioning has a significant impact on the final layout area. In algorithm-specific IC designs, there are many additional issues that are more important than the efficiency of the data-path logic. For instance, it was pointed out [14] that I/O compatibility between the raster-scan format of the camera and an image processing circuit is essential to achieve real-time image processing. It is an open question as how to incorporate these issues in the behavioral synthesis framework.

For these reasons, LAGER supports user specification of the data paths and then generates the rest of the design from a behavioral description. Support for behavioral synthesis with appropriate optimization criteria for our application focus is planned for the future [13].

2.1.3. Silicon Compilation into A Fixed Architecture: There have been a number of *silicon compilers* [1], [12], [15]–[20] that can translate the high-level description to layout for a fixed architecture. In [1], [15], [16], bit-serial architectures were used, in which the throughput rate of the chip was limited by the data wordlength in addition to the clock speed. The user interface in FIRST [15] is at the structural level so that the user has to translate the algorithm to a bit-serial architecture manually. Moreover, a hardwired floorplan is used which can result in a substantially larger chip area than that of a manual design. The INPACT compiler [16] has a higher level interface that allows the algorithm to be specified in a programming language, and performs some optimization on the floorplan. Cathedral-I [1] provides a high-level interface and performs several optimizations at the algorithmic, architectural, and floorplan level; however, the user interface and optimizations are primarily applicable to digital filter algorithms.

A second class of compilers uses microprogrammed bit-parallel architectures. While the achievable throughput rate is usually similar to that of bit-serial architectures, they provide greater flexibility in the algorithmic operations and the I/O interfaces. Examples of these compilers are given in [12], [17]–[20]. In [17], a register–transfer language is provided for describing the algorithm and a fixed floorplan strategy is used. In [12], an interface to a high-level applicative language [2] is provided and symbolic layout techniques are employed for optimizing the chip area.

The drawbacks with these compilers are that the target architecture uses a predefined data path and architectural modification by the user is very difficult. Furthermore, the layout generation techniques are fixed and cannot be influenced by the user. The fixed target architecture and predefined floorplanning limit the application of the above compilers to low throughput rate applications, typically with data rates below 1 MHz. In addition, the algorithmic operations that can be handled are limited by the predefined hardware modules that the compiler can handle.

2.2. Overview of LAGER

2.2.1. Motivation: To expand the kind of algorithms that can be handled, as well as to attack applications with high data rates (10 MHz and above), LAGER allows the user to modify or even completely specify the target architecture. In contrast to the "fixed-architecture silicon compiler" approach, a structural interface is established in LAGER. The user can use or modify predefined architectures in the library, or design a new target architecture through the structural interface.

To provide this flexibility, the LAGER environment is built from two distinct subsystems: 1) a *behavioral mapper* and 2) a *silicon assembler*. In contrast to the "behavioral synthesis" approach which attempts to synthesize the optimum architecture from a behavioral description, the LAGER behavioral mapper maps the behavioral description into a user-defined architecture. This approach allows the user to fine tune the target architecture by iteration to achieve desired performance with acceptable chip area.

The separation of the behavioral mapper and the silicon assembler has two more advantages. First, for high-speed applications in which hardwired architectures are often more appropriate, the silicon assembler can be used independently to generate a layout from a user-defined hardwired architecture. Second, the LAGER silicon assembler can be a backend of the emerging behavioral synthesis programs (as in [13]) to provide a means for automatic layout generation as well as feedback for performance and chip area evaluation.

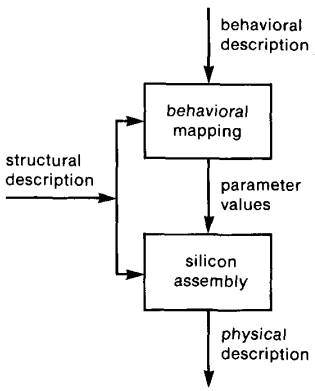


Fig. 1. Block diagram of LAGER.

2.2.2. Behavioral Mapper and Silicon Assembler: LAGER consists of a *behavioral mapper* and a *silicon assembler*, a block diagram of which is shown in Fig. 1. The structural description in LAGER is parameterized to facilitate the reuse of hardware modules or architectures. The silicon assembler requires both a structural description and associated parameter values to generate the layout. Examples of parameters are: the wordlength of the data path and the contents of the microcode PLA. The behavioral mapper maps a behavioral description onto a user-specified structural design by generating the appropriate parameter values.

The task of the behavioral mapper is similar to that of a high-level language compiler that generates microcode. This microcode is essentially one of the parameter values of the programmable architecture. The behavioral mapper can be *retargeted* to different structural descriptions. This is essential to the design cycle in which the architecture is tailored to the application: first, the algorithm is mapped to an existing architecture. Then, if the result is unsatisfactory, the architecture is modified and the algorithm is mapped to the new architecture. This process is iterated until a satisfactory architecture is obtained. The usual cause of dissatisfaction with an architecture is that frequently used instructions are not directly implemented. This situation is easily recognized from a histogram of instruction usage.

The LAGER silicon assembler integrates a number of layout generation and simulation tools under a user interface called the *design manager*. All communications among tools is through a common data base with procedural interfaces for data storage, retrieval, and modification. Integrating the tools is easy because they follow an agreed upon data base policy. The design manager builds the data base from the structural description. There are extensive cell libraries (of leaf cells and parameterized modules) which can accommodate various layout styles in the same chip design. New modules are added to the library by providing structural descriptions; new cells are added by providing simulation models and layouts.

III. Behavioral Mapping

LAGER can work from a behavioral (architecture independent) description of the algorithm. This description is mapped into parameter values, which completes the parameterized structural description, and in turn becomes the input to the silicon assembler.

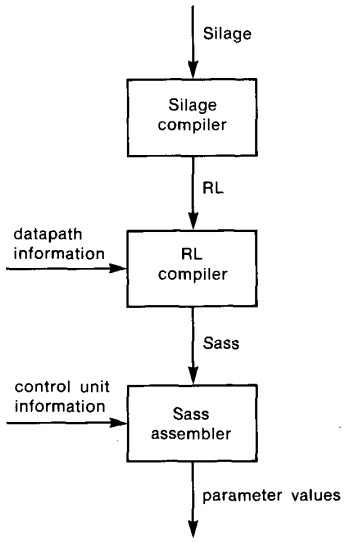


Fig. 2. Block diagram of the behavioral mapper. The user can use any of the three languages: Silage, RL, and Sass to encode the algorithm. Translators are also provided. Some require data path or control unit information for retargetable translation.

Fig. 2 shows a block diagram of the behavioral mapper. We employ three independent input languages. *Silage* is a data-flow language with very high-level constructs suited to the intended ASIC applications. *RL* is a variant of C, a procedural language with comparatively primitive features. *Sass* is an assembly language. The system supports all three languages equally; the user has the freedom to choose the input style most suited to his needs. Fig. 3 shows a simple infinite impulse response (IIR) filter, defined by the recurrence

$$x_i = \frac{3}{4} x_{i-1} + \frac{1}{4} in_i$$

described in the three languages. Each language has an accompanying translator that maps a program into the next lower level. The languages and their translators are described in Sections III-3.2–III-3.4.

Section III-3.1 describes the *Kappa* model of processor architectures. RL and Sass can be used for any architectures that fit the Kappa model. Silage is designed to be free of architectural bias, but the current Silage to RL translator is tailored to the RL compiler, and therefore, to the Kappa model.

3.1. The Kappa Architecture Model

Kappa [21] is a processor architecture model that has served well as a prototype for customization. A more detailed description of Kappa is given in Section V-5.1. The Kappa model contains a control unit and a number of data paths. The control unit generates micro-instructions to control the data paths, and the data paths feed the *status* signals (e.g., sign bits) back to the control unit, which are used as input for state transition in the control unit. The control unit can provide all control functions independent of the data paths. Some unique control mechanisms are also incorporated, as it has been found [21] that an inefficient control unit is the speed bottleneck in some applications. For example, for looping and more complex decision making,

```
func main(in: fix<8>): fix<8> =
begin
    return = (3/4) * return@1 + (1/4) * in;
end;
```

(a)

```
fix y;

main() {
    y = (3/4) * y + (1/4) * (fix) in();
    out(y);
}
```

(b)

```
(ram y)
(cfsm (0 0 nil (goto 0)))
(dp_word_size 8)
(rom (0 ((addr y) (mor=mem) (r*=rbus 0) (rbus=ioport) (ioport=extport 0))
        ((acc=abus) (abus=mor))
        ((abus=mor) (nosat) (acc=sum) (bbus=acc>* 1))
        ((acc=bbus) (bbus=acc>* 1))
        ((acc=bbus) (bbus=mbus) (mbus=r* 0) (r*=rbus 0) (rbus=acc))
        ((acc=bbus) (mor=mbus) (mbus=r* 0) (bbus=acc>* 2))
        ((bbus=acc>* 0) (abus=mor) (acc=sum))
        ((mbus=acc) (ioport=mbus) (extport=ioport 0) (addr y) (mem=mbus))))
```

(c)

Fig. 3. (a) The IIR filter described in Silage. (b) The IIR filter described in RL. (c) The IIR filter described in Sass.

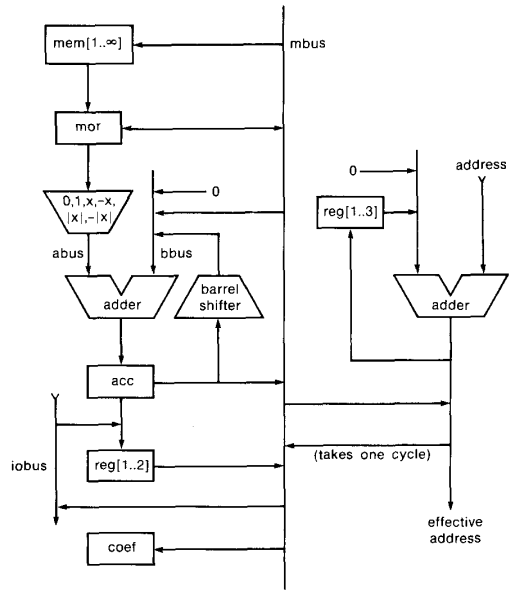


Fig. 4. A data-path example that fits the Kappa model.

the Kappa control unit provides a *multiway jump/call/return capability*.

An example of a processor data path that fits the Kappa model is shown in Fig. 4. A pipeline delay of one instruction cycle is associated with every register (mor, acc and coef) and register bank in the figure; these delays include those of the functional units. If the individual functional units are understood, the diagram completely defines how they work together—every apparent, meaningful combination of actions is possible when the instruction word is fully horizontal. A horizontal instruction word is just a vector of control signals with little or no restrictive encoding.

Kappa is *irregular*, meaning that its data path topology can be chosen to suit the usage in a particular program (e.g., a robot arm controller) rather than to conform to the expectations of modern compilers. The alternative would be exemplified by an architecture in which all intermediate results are stored in a large, multiported register bank. An irregular data path is smaller, faster, more tunable, and hence, more appropriate for inclusion in an ASIC. The problems that irregularity poses for retargetable compilers are minimized by the use of a fully horizontal instruction word.

The behavioral mapper generates horizontal microcode for the user-defined data path from a behavioral description. The three languages used in the behavioral mapping part of LAGER, Silage, RL, and Sass are described in the following sections.

3.2. Silage

Silage is a high-level language optimized for specifying "signal processing like" algorithms. As such, its design emphasizes expressiveness over details of implementation, by providing the programmer with convenient data types and operations and hiding from him quirks of the actual hardware. We will give an overview of the language and its compiler in this section. More detailed descriptions can be found elsewhere [2], [22]. The "Silage Reference Manual" [23] specifies the language in full.

3.2.1. The Silage Language: Silage is a data-flow language that operates on streams of values. As in all data-flow languages, a Silage program corresponds to a data-flow graph—a directed graph with operations performed by vertices and values carried on edges. A subtraction node, for example, combines two streams of numbers to produce a difference stream. A program graph accepts input streams, combines and alters them as they flow through the graph, to produce the output streams. This model of computation is natural for our intended applications. For example, real-time control typically takes time-sampled input streams from sensors to produce control signals. Digital signal processing (DSP) operates on streams of digitized signal samples, and a form of dataflow graph is already commonly used to express DSP algorithms. In addition to the usual arithmetic functions, Silage provides some common DSP operations: delaying a stream (Z^{-1}), taking a substream (decimation), and taking the union of streams (interpolation).

The textual Silage program is a set of equations that defines output variables in terms of input variables. Each equation defines a single variable to be an expression on other variables. Each variable is defined exactly once, following the single assignment rule of most dataflow languages. Thus the right-hand side of each definition corresponds to an expression tree; connecting the trees by linking variable definitions to uses produces the program graph.

Variables can be subscripted in the usual way. The *iterated definition* provides a general way to operate on subscripted variables. The iterator can specify both parallel and sequential computation. The only difference is in the dependence (or lack of dependence) between iterations. The example in Fig. 3(a) computes an output stream x_i (called return in the code), which equals $3/4$ times the previous sample (return @1, using the delay operator @) plus $1/4$ times the current value of *in*.

Structural preferences can be expressed by the *pragma* directives [2]. For example, in the case when the number of parallel processing elements is less than the number of independent expressions that can be computed simultaneously, we can use

pragma Processor (K, Expr);

to indicate that the expression *Expr* should be computed on processor element *K*.

3.2.2. The Silage Compiler: The current Silage compiler translates Silage programs into RL to run on a single microcoded processor. The main difficulties are in converting the unique features of Silage into the conventional operations supported by RL. These include modeling data-flow semantics in a procedural language and building the data structures for streams in terms of primitive data types. Central to a good implementation is the conversion of repetition in the data-flow graph into efficient program loops. We call this *loop folding*.

Repetition can come from iterated definitions, delay queue operations, and interpolated data streams. The loop folder processes the input program in data-flow graph form, converting repetitive connected subgraphs into loops. The algorithm first finds candidate loops by graph traversal, then creates loop bodies that are graph fragments parameterized on the loop indexes.

Loop folding typically consumes half of the total compilation time, but because it operates on the iterator-expanded graph, its running time can be exponential on the size of the input program. It is, however, both simpler and more powerful than the alternative approach that infers iteration dependency on the original iterated definitions by analyzing variable index expressions. The basic loop folder is general enough to handle all three sources of repetition. In particular, the *copy* operation needed to implement delay queues without special hardware support can often be merged into other program loops.

3.3. RL

The task of the RL compiler is to generate a control program for a processor of specified structure. The compiler must meet a pair of conflicting goals. On the one hand, it must be easy to retarget. The user should be able to evaluate a proposed change in processor structure by retargeting the compiler and then recompiling the program. On the other hand, the compiler must generate good code for diverse, irregular architectures.

We only give a brief overview of the RL compiler here. Rimey and Hilfinger describe, in more detail, the overall compiler [24] and the original code-generation techniques that it uses [25].

3.3.1. The RL Language: The inputs to the RL compiler are a source program, written in the RL language, and a machine description. Both are provided by the user, although he will usually use or modify previously specified machine descriptions.

The RL language is an approximate subset of C. It incorporates two major extensions: fixed-point types and register classes. Fixed-point types provide the programmer with convenient notation for fixed-point constants and arithmetic. Register classes, which generalize C register declarations, enable the programmer to suggest storage locations for critical variables.

3.3.2. The Machine Description: A machine description consists of

- declarations of buses, latches, registers, and register banks;
- a list of simple register-transfers (e.g., *abus* → *acc*) defining the topology of the data path;
- a list of functional register-transfers (e.g., *abus* + *bbus* → *acc*) representing capabilities of the functional units.

The compiler automatically uses simple transfers to chain together functional transfers. It selects from instances of a functional unit when there is more than one. It allows the user to declare register-transfers to be incompatible, as is necessary when the incompatibility is not apparent from a conflict in bus usage.

3.3.3. The RL Compiler: The compiler consists of two parts. The front end translates the program into successive straightline segments of code, expressed in an intermediate language. Then, for each straightline segment, the backend selects register-transfers and packs them into instruction words.

In addition to routine tasks and simple optimizations, the front end performs two optimizations that are particularly important for Kappa model architectures. First, when no parallel multiplier is provided, it reduces multiplications by constants into minimal sequences of shifts, adds, and subtracts. Second, it coalesces branches to utilize Kappa's multiway jump/call/return.

Most of the effort in developing the RL compiler has gone into the algorithms in the backend. The usual approach to generating horizontal code is to first generate loose sequences of register-transfers and then pack these tightly into a small number of instructions through *compaction* [26]. Our approach is to integrate register-transfer selection and local compaction into local *scheduling*. This creates the opportunity to perform a *lazy* routing of intermediate results between functional units, choosing appropriate sequences of simple register-transfers late in the scheduling process when more of the schedule is known. For Kappa model architectures, scheduling with lazy data routing is profitable, but also difficult. The compiler must take care that all feasible routes for a live result are not by chance closed off. A network flow algorithm that performs this test efficiently has been developed [25].

The RL compiler is written in Lisp and compiles approximately one line per second. It has been used to compile programs several hundred lines in length. Making modifications to the machine description has proven to be easy; for evaluating their impact on performance, the compiler has proven to be more reliable than intuition.

3.4. Sass

Sass is an assembly-level language. A Sass program consists of symbolic microcode and definitions of the other parameters for a Kappa processor. The body of a Sass program has two parts: straightline code blocks and control flow information. A straightline code block is a sequence of micro-instructions, uninterrupted by branches. Each micro-instruction consists of a number of micro-operations, which perform arithmetic, logic, and addressing functions. Sass programs also define additional parameter values of the processor. For example, the width of the data path is ordinarily specified this way.

In Fig. 3(c), we show a Sass program for a IIR filter. It consist of a number of Lisp *s*-expressions. The expression (ram *y*) describes that one local variable, *y*, is stored in RAM. The expression (*cfsm* · · ·) declares the control flow among the straightline code blocks. In this case there is only one code block. The expression (*dp_word_size* 8) indicates the width of the data path is 8. Finally the expression (*rom* · · ·) defines all the straightline code blocks. In Fig. 3(c) the code block consists of 8 micro-instructions, which have 6, 2, 4, 2, 5, 4, 3, 5 micro-operations, respectively.

The Sass assembler passes some of the parameter values to the output unmodified; it uses the rest to generate the control unit, which is its main task. Since the Kappa architecture can be customized, a machine description is also needed as input

(Fig. 2). The machine description specifies the control signals and hardware resources used by each micro-operation. The Sass assembler uses it to assemble data-path instructions into binary microcode, and to check for resource conflict errors between micro-operations.

The Sass assembler's main task is to output a specification (in the form of parameter values) for a Kappa control unit. The control unit includes a read-only control store containing the assembled microcode blocks and a state machine that controls transitions between the blocks. The specification also contains parameter values such as the width of the program counter and the depth of the stack. Some components are not always necessary. For example, setting the depth to zero discards the stack in the resultant silicon implementation.

The sample rate of the application that can be achieved using the behavioral mapper depends on the resulting number of instruction cycles, assuming a fixed circuit speed. The user has the freedom to modify the data path and iterate the process for speed optimization. After this is done, the parameter values and control unit specification generated by the behavioral mapper are sent to the silicon assembler for layout generation.

IV. The Silicon Assembler

The task of the silicon assembler is to generate the chip layout starting from a parameterized description of the chip architecture. Separating the silicon assembler from the behavior mapper allows the user to change the architecture directly and, furthermore, allows the silicon assembly tools to be used in conjunction with other high-level architecture synthesis aids.

A major goal in developing the silicon assembler was to handle arbitrary architectures and to create an environment for quick iteration on different architectural alternatives. A second goal was to allow the re-use of parameterized cells so that a minimum effort is required to design new cells. Thirdly, based on circuit design expertise, it was found that it was essential to allow different layout styles to be combined to achieve higher design quality.

To achieve these goals a design management tool [27] was developed. The task of this tool is to automate the layout generation procedure for any architectural description and serve as the user-interface to the silicon assembler. This also shortens the learning curve for the designer to use the silicon assembler since only one tool is seen by the user. The design manager tool in LAGER called *DMoct* is described in Section IV-4.1. Sections IV-4.2–IV-4.3 describe the layout generation tools currently interfaced to DMoct.

4.1. The Design Manager: DMoct

Fig. 5 illustrates the design methodology followed by DMoct. The input to DMoct is a parameterized hierarchial description of the chip architecture. For a given set of parameter values, DMoct generates the chip layout. DMoct uses the OCT database system [28] to store all design data, which has the advantages that it provides a procedural interface for CAD tools and the data representation can be defined by the application. Three OCT representations have been defined for the silicon assembler: a) *structure-master* view; b) *structure-instance* view; c) *physical* view. Details of these views are given in the following.

The structural description at a given level of the hierarchy can be described textually using structural description language (SDL), which is illustrated in Fig. 6, or graphically using a schematic entry by VEM [28]. In addition to nets, subcells and terminals declared in conventional netlist and hardware descriptions, unique features of SDL are: 1) for each cell the tools required to generate the layout are declared; 2) all nets can be parameterized; and 3) each cell in the hierarchy can inherit parameter values from its parent. Thus all cell and net parameter values can be declared as arbitrary functions of a unique set of parameters for the chip (the root cell), typically the algorithm parameters. This allows the user to quickly generate different versions of the chip for different algorithm parameter values without having to change the architectural description.

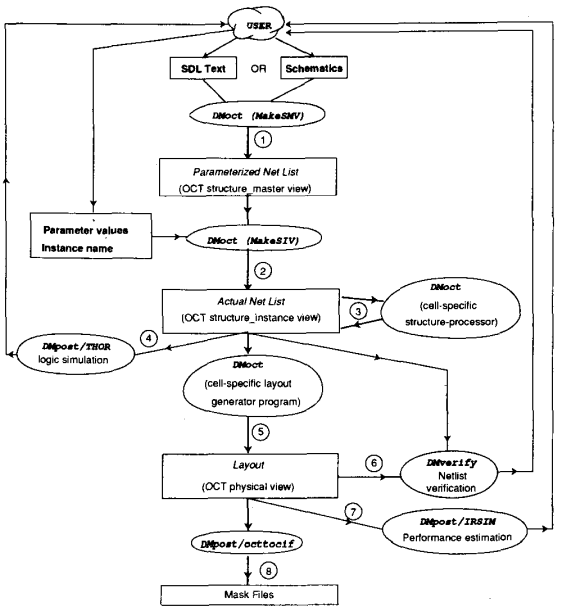


Fig. 5. Design methodology of the LAGER silicon assembler. Items in *ellipse* are tools. Items in *box* are files. Numbers in *circle* correspond to the 8 design steps.

The operations of DMoct and its associated utilities are described below.

Step 1: Master creation: DMoct parses the SDL files and stores the architecture as a structure-master view in OCT, which can also be created graphically with a schematic entry. This view is an OCT representation of the information shown in Fig. 6.

Step 2: Instance generation: DMoct traverses the design hierarchy in a top–down depth-first manner and generates the actual *instances*. At each level it evaluates the child cell parameters from the parent cell parameter values. This parameter evaluation mechanism allows a more powerful parameterization compared to pop-up forms used in some commercial CAD systems. The instances are stored in the OCT database as a *structure-instance* view. The structure-instance view is used as the common input to all the layout generation tools.

Step 3: Structure processing: Prior to layout generation, DMoct allows the structure-instance view to be modified by a special class of programs called *structure-processors*. An example of a structure-processor is a bit-slice data path processor (Section IV-4.2) that modifies the cell and net information in the structure-instance view to allow a more optimal routing by the layout generation tools.

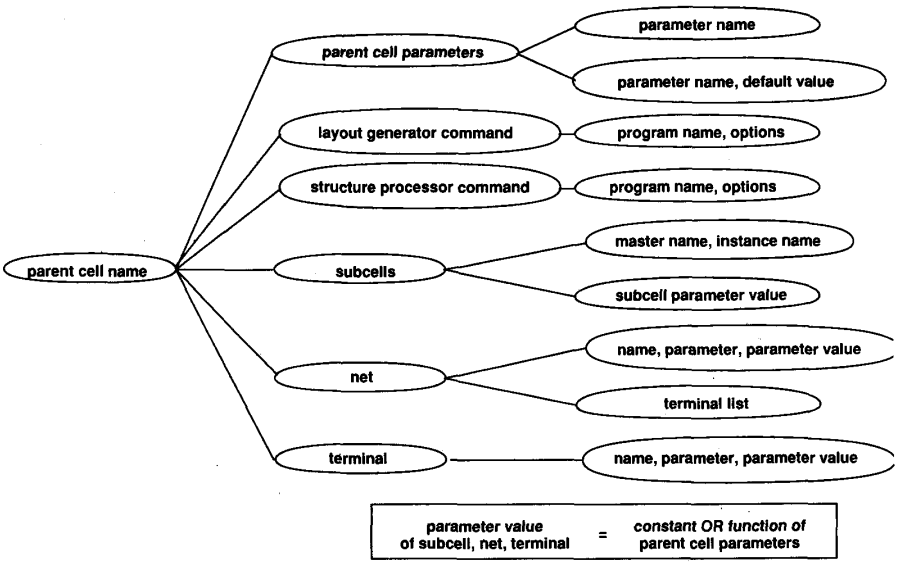


Fig. 6. Syntax diagram of the structural descriptional language (SDL).

Step 4: Functional simulation: Assuming that parameterized simulation models exist for the leaf cells in the hierarchy (which are usually the library cells). A utility called *DMpost* is then used to generate a simulation input for the entire design for the simulator THOR [29]. Based on simulation results, the designer can modify the SDL files or parameter values and rerun the simulation. When simulation results are satisfactory, layout generation can proceed.

Step 5: Layout generation: In this step, DMoct traverses the cell hierarchy in the structure-instance view in a top–down depth-first manner. When DMoct reaches a leaf node, it recurses back up the hierarchy and executes the layout generator program specified for each cell. By policy each layout generator reads its input information about the cell from the structure-instance view and stores the information about the layout as a physical view. If the physical view exists for a cell and is more recent than the corresponding structure-instance view, the layout is not generated. This avoids regeneration of cells that are not affected by design iteration.

Step 6: Verification: A verification tool *DMverify* is provided to check for isomorphism between the schematic represented in the structure-instance view and the hierarchial schematic extracted from the physical view. In contrast to conventional netlist verification tools, this program compares the cell hierarchy and connectivity (not transistor or gate level). The inherent assumption here is that at the lowest level, the cells were designed by hand and have been fully debugged using circuit simulation and test chips. The uniform policy imposed on all tools by the structure-instance and physical views takes the guesswork out of the netlist verification so that time consuming signature analysis and logic extraction procedures can be avoided.

Step 7: Performance estimation: The chip performance depends on the performance of the cell library. Currently, a MOSIS rev-6 scalable CMOS library is integrated, allowing clock rates in the region of 20–25 MHz in a 2-μm process. The library provides various data and clock buffering cells so that performance is relatively independent of the place and route tools if appropriate buffer cells are used. The choice of buffer cells is not yet automated. The chip performance can be estimated based on SPICE characterization of the library cells and an estimation of the critical path. Alternatively, the simulator IRISM [30] can be used in the linear mode to obtain performance estimations.

Step 8: Mask file generation: When the final layout is acceptable, mask files can be generated from the OCT physical view.

It is important to note that to perform all the above steps the user only interacts with DMoct and its associated utilities. The designer can rapidly iterate on the design and explore different architectural alternatives by simply changing the schematics and parameter values and rerunning DMoct. The design times for silicon assembly with DMoct are application dependent; some data are given in Section V-5.3.

The LAGER silicon assembler is cell based. The cell library contains parameterized modules and leaf cells. New modules can be added to the library by providing structural descriptions; new leaf cells can be added to the library by providing simulation models and layouts. Re-using parameterized modules and leaf cells has proven to be an important factor in improving the productivity.

A new layout generator, structure processor or simulator can be integrated in the silicon assembler by following the policy defined by DMoct. This involves writing procedures to read, write or modify the three OCT views. For information the reader is referred to [28], [31]. Integrating a new tool or cell does not require any modification of the user interface provided by DMoct or of the interface between DMoct and the OCT database. This capability is an important feature in creating an open framework for tool integration.

A prototype of the design manager was previously developed using the Flavors package in FranzLisp to implement the data base [32]. This was motivated by the facilities of parameter declaration and evaluation, and the ease of prototyping offered by Lisp, and the advantage of object-oriented paradigm for tool integration offered by Flavors. However, this version was too

slow to permit efficient design iteration and was less portable than the C-based OCT data base. Also, unlike OCT, which maintains the structure-master, structure-instance, and physical views as disk files, the Flavors-based implementation keeps all design information in the process memory, which limits the design size.

4.2. Layout Generation Tools

While the CAD framework supported by DMoct allows any tool to be integrated so long as it fits the methodology described above, currently, a set of four layout generators are interfaced to the framework which support various design styles required in algorithm-specific IC's. These are described in the following.

4.2.1. Tiler for Macrocell Layout Generation: TimLager: TimLager is a general purpose macrocell layout generator that is used for bit-sliced modules such as adders, registers, multiplexers as well as array-based modules such as RAMS's, PLA's, and ROM's. It assembles the layout for the macrocell from hand designed leafcells by *abutment*. This requires the leafcells to be pitch-matched and avoids the use of routing in macrocell generation.

For each macrocell, TimLager requires a set of leafcell layouts, and a tiling procedure that describes how the marocell is constructed from the leafcells as a function of the macrocell parameters. The procedure is written using the C language. To allow rapid creation of tiling procedures for new macrocells TimLager provides two tiling functions **Addright**( ) and **Addup**( ) (Fig. 7). The full capability of C can be exploited for parameterizing the layout. Hierarchical tiling can be performed using subroutines. This allows greater flexibility than the personality matrix approach traditionally used in tiling.

Each tiling function has a set of 20 optional arguments that allows geometric transformation on the cell to be placed. The arguments also allow several bookkeeping operations to be automated, such as the naming of terminals. Thus the cell terminals as defined in the library can be changed arbitrarily to names more pertinent to a given design. The optional arguments also provide a stretching mechanism that can be used to pitch-match one macrocell to another as well as a mechanism for adding metal feedthrough lines in between cells. These options can be exploited for optimization of a higher level cell (e.g., a data path) constructed from several macrocells generated by TimLager.

The tiling procedure for each macrocell is compiled and stored in the library along with associated leafcell layouts. At runtime TimLager dynamically links the required procedure and executes it. The parameter values are read from the structure-instance view and a physical view is generated for the macrocell layout.

4.2.2. Random Logic Macrocells Using Standard Cells: Stdcell: Stdcell provides an interface to a standard cell place and route tool, *Wolfe* [28], to generate standard cell modules for a given logic schematic. Wolfe in turn uses the *TimberWolfSC* [33] standard cell placement program and the YACR [34] channel router. The standard cell modules can also be generated from a high-level logic description (see Section IV-4.3.3). Stdcell directly reads the structure-instance view to obtain the logic schematic and generates a physical view for the layout.

4.2.3. Macrocell Place and Route Tool: Flint: Higher level cells can be constructed with Flint [35] from the macrocells generated by TimLager or Stdcell (or any other macrocell generator that produces OCT physical views). Flint supports three floorplan definition alternatives: 1) interactive-graphical (Fig. 8), 2) interactive-textual using a floorplan description language (FDL), or 3) automatic. An automatically generated floorplan can be further refined using the interactive modes. A floorplan definition contains the relative placement of the macrocells, the definition of the channel areas, and the specification of the global routes for signal, power, ground, and clock nets. In order to make the global routing process tractable in an interactive mode, Flint clusters groups of nets in so called *cables* (Fig. 8). A cable is a set of nets which have identical sources and destinations.

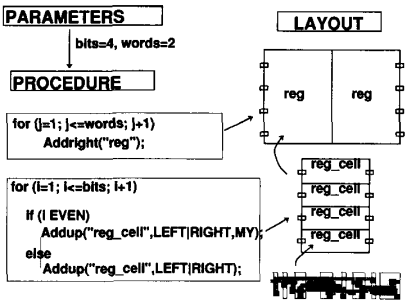


Fig. 7. Tiling procedure used in TimLager. In this example, leafcell *regcell* is abutted vertically by **addup**( ) calls to form *reg*, which is abutted horizontally by **addright**( ) calls to form the module. The module has 2 parameters: *bits* and *words*.

The automatic mode is based on min-cut and slicing. This technique has the advantage that a realizable channel structure is automatically obtained. The placement is then further refined using a modification of Stockmeyer's algorithm [36], an efficient tree-traversal method which determines the optimal orientation of each macrocell. The global routing is based on Dijkstra's [37] shortest path algorithm. The use of the cable concept mentioned above increases the efficiency of this phase of the floorplanning process. Finally, the global routes of power and clock nets are determined using an approximation technique for the Steiner tree problem [38]. Flint does not build the power and ground network as a single tree, but rather as a "forest of trees." In fact, it is not necessary to have a connected power network at every level of the chip hierarchy. All the above routines are extremely efficient and complete in less than a minute CPU time for complex examples (up to 20 macrocells and 1000 nets) on a SUN 3/60.

Given the floorplan, Flint carries out all other steps automatically including channel routing, power/ground/clock routing, absolute placement, and creation of terminals on the top level cell (interface for the next higher level in the hierarchy). Detailed routing in Flint is performed by a completely gridless channel router. Although the router is a variant of the classical left-right router, it has some features which distinguish it from other routers. First, it routes signal, power, and ground nets together using a priority scheme, which gives special priority to more sensitive nets such as power and ground. Secondly, it automatically sizes the power and ground nets, using terminal current information if it is available, or else it is based on a saturating weighted sum of the connecting terminal widths. Finally, it supports three layer routing, which is essential when high density routing is necessary as in the case of bit-sliced data paths.

4.2.4. Pad frame generation and routing: Padroute: The creation of a pad frame and routing of signals from the core of

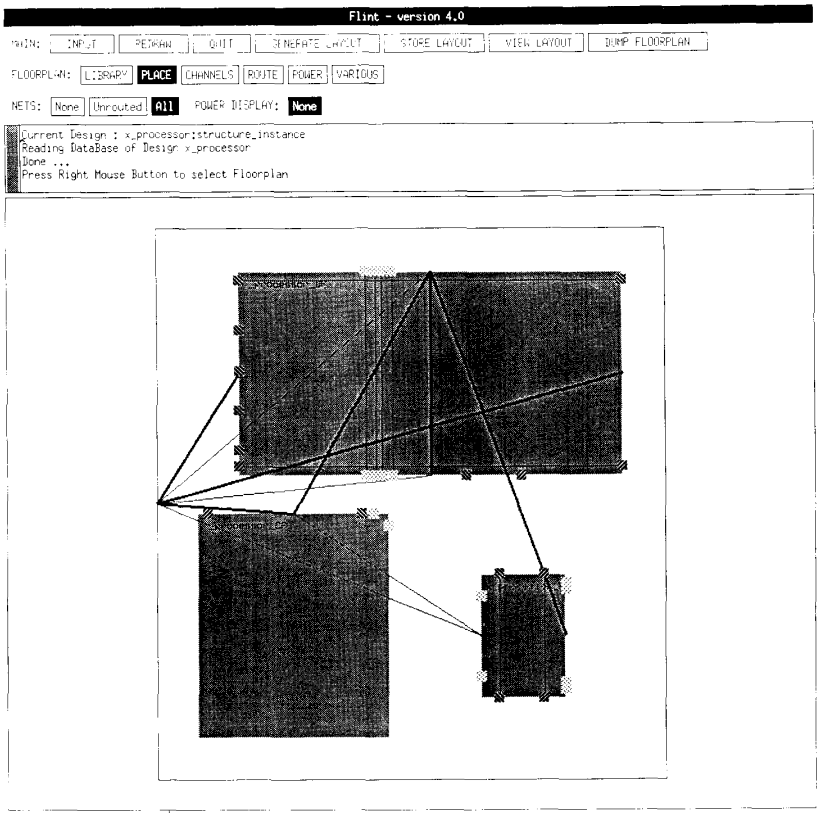


Fig. 8. Interactive-graphical floorplan definition in Flint.

the chip to the pad frame is done by a specialized tool called *Padroute* [39]. If the terminals on one side of the core connect only to pads on the same side of the pad frame, a simple channel router can be used. However, in practice, this is often not the case and hence, a special ring router is necessary.

The ring routing algorithm is a modified channel routing algorithm, which allows Padroute to route a channel that does not have left and right ends. Padroute creates radial and circumferential constraint graphs (Fig. 9). The radial constraint graphs serves the same purpose as a vertical constraint graph in a regular channel router. It shows the relative positions of the tracks the nets must occupy. A track in Padroute runs the entire circumference of the ring-shaped routing region. The circumferential constraint graph represents nets that may be placed in the same track. Padroute continues by checking for cycles in the radial constraint graph. If a cycle is detected, a dog-leg is added to one of the nets involved in the cycle.

Once the radial constraint graph is cycle free, nets are assigned to tracks. The first pass simply assigns one net per track. The second pass tries to combine nets onto single tracks as much as possible to reduce the space occupied by the routing. After track assignment, Padroute verifies that the routing will fit into the initial pad frame. If the routing cannot fit, the pad frame is enlarged.

4.3. Structure Processor Tools

This section describes structure processor tools that are used to preprocess the user-specified structural description before actual layout generation is performed. It should be remembered these are tools called by DMoct and are, therefore, hidden from the user.

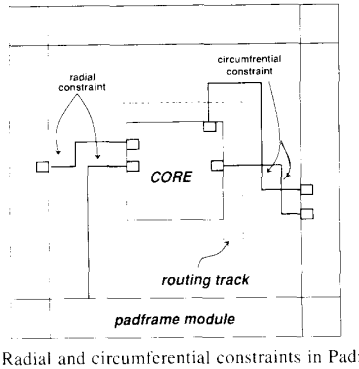


Fig. 9. Radial and circumferential constraints in Padroute.

4.3.1. Data-Path Processor: dpp: The bit-slice data-path generation tool, *dpp* [40], is perhaps the most important utility for algorithm-specific IC's. Much of the processing power of such circuits comes from the ability to dedicate the data-path architecture to the exact needs of the algorithm. With LAGER the designer can very quickly reconfigure a data path and iterate on several designs while evaluating their area and performance. Given the schematic of a bit-slice data path, dpp does the placement, channel definition, and global routing of the data path

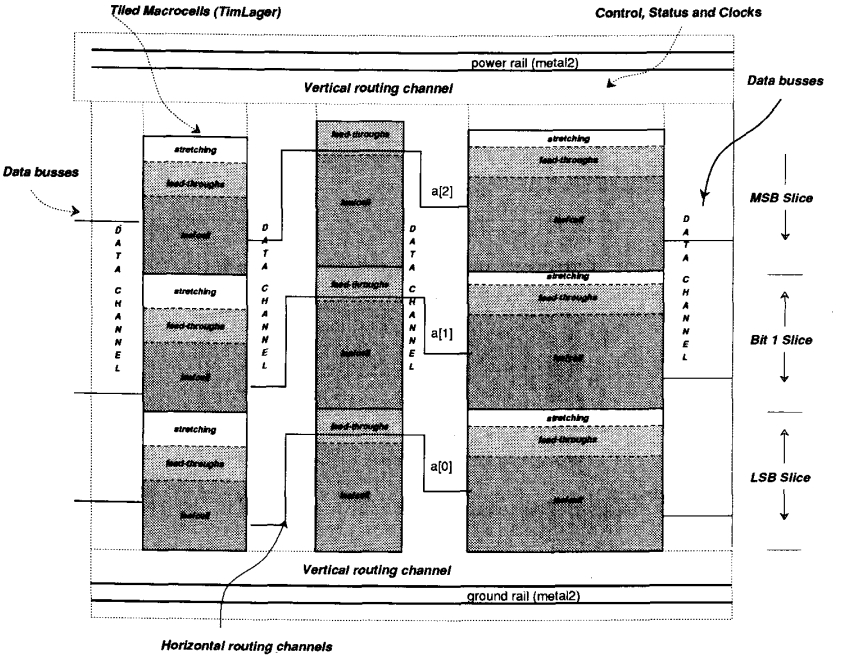


Fig. 10. Bit-slice data path by dpp. Each cell in the block consists of a *leafcell* (dark), a *feedthrough* (grey) and an optional *stretching* (white). Horizontal channels are used to route data signals between blocks. Global and local channels are used to route control, status, and clock signals. Each block is generated by TimLager. Routing of blocks is done by Flint.

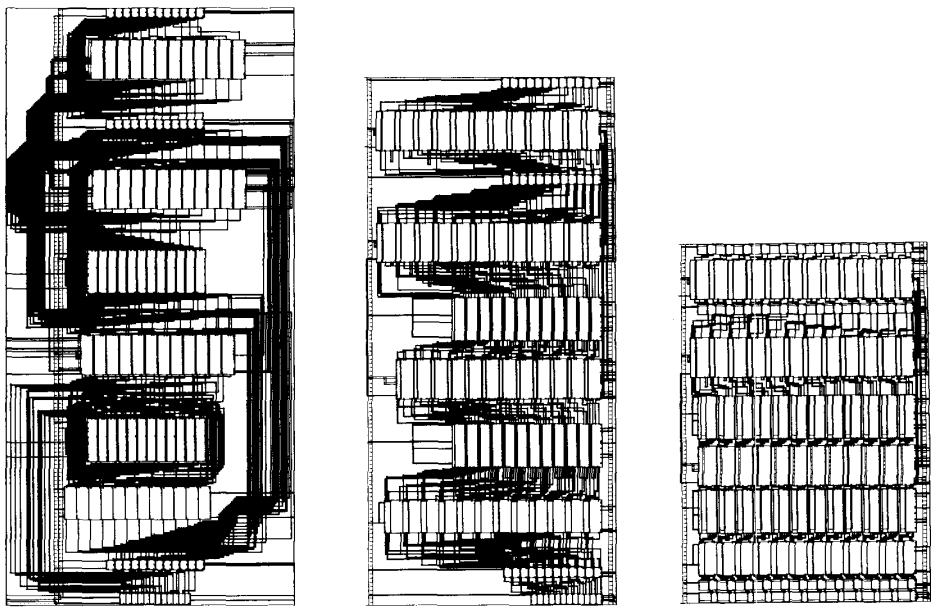
and produces a floorplan (a FDL file) for Flint to route the individual channels and generate the actual layout. It also back-annotates the structure-instance view of individual macrocells in the data path with geometric constraints and feedthrough information for TimLager.

Bit-sliced data paths (Fig. 10) are viewed by dpp to consist of macrocells that are tiled in the vertical direction and placed linearly along the horizontal direction with the bottom edges of the macrocells being co-linear. Horizontal routing channels separate adjacent macrocells. *Local* vertical channels are placed along the top edge of each macrocell in order to equalize the heights of each macrocell. Finally, *global* vertical channels spanning the entire width of the data path are placed at the top and the bottom of the data path.

The control, status, clock, and supply nets run vertically. Within a macrocell they are routed implicitly by the abutment of leafcell terminals during the tiling process. The global routing of these nets between macrocells and to the outside is done by dpp using the global vertical channels. The data buses flow in the horizontal direction and are routed explicitly using the horizontal routing channels between the adjacent macrocells. Data buses connecting nonadjacent macrocells are routed *through* the intervening macrocells. This is done by back-annotation of the structure-instance view of the macrocells with information for TimLager to generate enough feedthroughs for data buses going across the macrocell. Feedthroughs already provided by the leafcell designer are used first before extra feedthroughs are generated.

The process of guiding the lower level layout generation according to the requirements of the upper level layout generator is a key feature of dpp. It makes the macrocells appear *porous* to Flint and saves the area wasted by the macrocell place-and-route approach in routing around the *opaque* macrocells. As shown by the example in Fig. 11(a) and (b), this results in a 24% reduction in area. A problem with our approach is that there may be a mismatch between the heights of adjacent macrocells resulting in a *staircase* effect or congestion in the horizontal routing channels. Back-annotation is again used here to force TimLager to stretch the heights of the leafcells used in each bit position in all the macrocells to a uniform value. This equalizes the macrocell heights resulting in much better routing channels. A further 32% reduction in area is obtained in the same example (Fig. 4.7(c)). It can be shown that the percent area penalty due to the staircase and the macrocell opacity effect increases with the number of bits. Consequently, the percent area reduction obtained by making the macrocells porous and stretchable also increases as the number of bits increases. For example, a 24-b version of the same data path shows a 63% reduction in area.

The other crucial step in dpp is placement. The goal is to find a suitable ordering of the macrocells so as to minimize the area. With our approach of through-the-macrocell routing of global buses and equalization of the macrocell heights, the problem can be quite accurately modeled as minimization of the height of the tallest macrocell taking the extra feedthroughs required into account. Dpp directly calls TimLager in an estimation mode to obtain information about the physical characteristics of the macrocells. This information is then used by the placement procedure which is based on the Kernighan and Lin's min-cut placement algorithm [41], but tries to minimize the height of the tallest macrocell taking the extra feedthroughs required into account, instead of the number of nets crossing a partition.



(a) (b) (c)

Fig. 11. (a) Data-path layout with Flint; area = $4.2 \times 10^6\lambda^2$. (b) Data-path layout with Flint after processing with dpp to add feedthroughs; area = $3 \times 10^6\lambda^2$. (c) Data-path layout with Flint after processing with dpp to add feedthroughs and stretching of cells; area = $2.18 \times 10^6\lambda^2$.

4.3.2. Logic Synthesis: Bds2stdcell: The logic synthesis tools BDSYN [42] and mis-II [43] are used by Bds2stdcell to translate a combinational logic description for a module in the BDS language to a logic schematic using a standard cell library. The standard cell layout can then be generated using Stdcell. Bds2stdcell reads the logic description and external terminal information from the structure-instance view of the logic module. After executing BDSYN and mis-II it back-annotates the structure-instance view with the actual schematic (nets and subcells). Thus the designer can define the logic module as part of the input architecture description and connect it to other modules in the chip without having to provide the detailed schematic.

4.3.3. PLA Optimization: Plagen: The *plagen* structure-processor reads the BDS logic description for a PLA from the structure-instance view and back-annotates it with the input-plane and output-plane bit patterns for the PLA. After being minimized by *espresso* [44], the PLA layout can then be generated by a PLA generator using TimLager.

In summary, we see that structure-processor tools can be used to back-annotate the structure-instance view with information for the layout generation tools. This back-annotation may consist of defining parameter values required by the macrocell generation (as in plagen and dpp) or defining the schematic itself (as in bds2stdcell). This allows the designer to exploit high-level tools on individual cells in the architecture while retaining a common input structural description.

V. Design Examples

Application-specific IC's for a variety of applications have been designing using the LAGER system. They include a low-level trajectory controller for a two-joint robot arm [45], a chip set for real-time emulation of communication channels in computer networks [46], a chip set for continuous speech recognition using hidden Markov models [47], [48], a real-time image segmentation chip [49], and an image processing chip for Radon transformation [50], etc. Still under development are chips for digital mobile radio, machine vision, and robotics.

Some of the above chips use only the silicon assembler portion of the LAGER system. These chips use hardwired architectures because programmable architectures are not suitable due to either higher computation requirements, as in image processing applications, or specialized I/O requirements, as in the chips for the network channel emulator. Use of a hardwired architecture precludes the use of the behavioral mapper. However, the design of these chips is facilitated by the silicon assembler, which enables fast, automatic generation of layout from a netlist description.

Other chips, such as the robot arm controller, the adaptive equalizer for digital mobile radio, and the inverse kinematics processor for a six-joint robot arm, use algorithms that are better suited to the programmable architectures such as the Kappa model. They have been designed or are being designed using both the behavioral mapper and the silicon assembler.

Section V-5.1 describes a robot arm controller chip, which was designed using both the behavioral mapper and the silicon assembler. Section V-5.2 describes a chip for real-time image segmentation, which was designed with the silicon assembler. In Section V-5.3 we present the design results of several other algorithm-specific IC's in hardwired architectures.

5.1. Robot Arm Controller Chip

The robot arm controller chip [45] is the heart of a robot-control system that directs a two-joint, direct-drive robot arm along a desired trajectory in real time. It uses a *model-reference adaptive control* (MRAC) algorithm, which takes into account

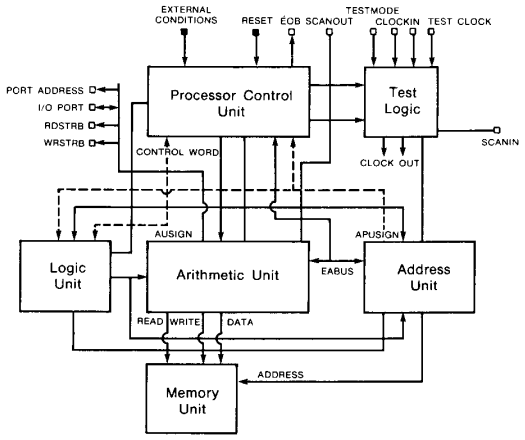


Fig. 12. Processor architecture of a Kappa example used in the robot arm controller chip. The solid lines from the processor control unit (PCU) are the control signals. The dotted lines from the data paths are the status signals.

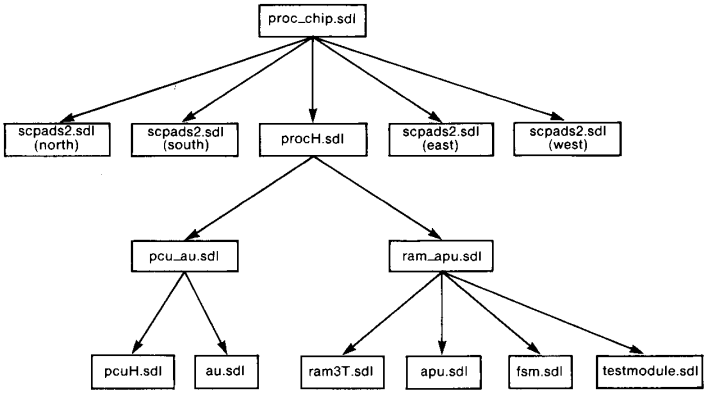


Fig. 13. The structural hierarchy of the robot arm controller chip. The six blocks at the bottom correspond to the six components in Fig. 12.

the nonlinearities in the arm dynamics and adaptively determines the parameters of the arm at runtime.

On an IBM PC, the MRAC algorithm achieves a 7-ms sample period. Implemented on a TMS32010-based board, it achieves a sample period of 0.7 ms, but at a significant cost in hardware and board area. For a higher speed, a more complex algorithm, or a reduced amount of hardware, a custom chip is appropriate. The custom robot arm controller chip not only achieves a higher speed with a sample period of 0.04 ms, but also reduces the required I/O hardware by customizing the chip I/O.

The chip was designed using the Kappa model architecture. The robot-control algorithm, unlike many DSP algorithms, has many conditional branches and loops. The Kappa control unit provides hardware for efficient handling of these operations. The chip consists of the following functional units, as shown in Fig. 12.

Processor Control Unit: This controls program execution and provides support for branching, looping, and subroutines. It consists of a finite state machine and a control store.

Arithmetic Unit: This is the main data path, used to perform fixed-point arithmetic. It consists of a bit-slice data path and a block of random logic for decoding control signals.

Logic Unit: This is a finite state machine implementing Boolean operations.

Address Processing Unit: This is an auxiliary data path that performs address computations. Like the arithmetic unit, it consists of a bit-slice data path and a block of random logic for decoding control signals.

Memory Unit: This is a random-access, read-write memory, closely tied to the arithmetic unit.

Testing Module: This is an interface for an external tester. It supports testing and debugging of the chip using the *scanpath* technique.

The layout of the chip is generated from a parameterized structural description. The chip is described hierarchically, using SDL files as shown in Fig. 13. The components are generated using the tools best suited to their layout style; an appropriate layout generation tool is associated with each SDL file.

At the top-most level of the hierarchy, the chip consists of four pad groups (one for each side) and a core section. The five units are connected by Padroute. The pad groups are assembled as linear tilings of pads using TimLager. The core section consisting of the six functional units described above is generated using Flint. The finite state machine PLA's, the control store,

and the memory are generated from parameterized descriptions by TimLager. The bit-slice data paths are generated by the data-path placement and routing program, dpc (which is the predecessor of dpp). The random logic required for decoding control words and generating local clock signals is generated by Stdcell.

Values for all of the parameters in the SDL description need to be provided before layout can be generated. Some parameters, such as the widths of the data paths, are provided in the behavioral descriptions. Other parameters, such as the contents of the control store and the finite state machine, are generated by the behavioral mapper. The algorithm was coded in all three input languages in the behavioral mapper. The hand-coded Sass version was finally used in the chip, for its slightly better performance than the compiled code. A die photograph of the robot arm controller chip is shown in Fig. 14. The dimension of the chip is 8.4 mm^2 by 7.15 mm^2 in the MOSIS 2-μm SCMOS technology. The chip is tested to operate at 15 MHz [21].

5.2. *Real-Time Image Segmentation Chip*

Low-level image segmentation reduces the image data rate (usually from gray-level to a few classes) and thereby enables more sophisticated image processing in the following stages. Image segmentation using *supervised pattern recognition* [51] involves: 1) feature extraction and 2) classification. Feature extraction extracts local (such as windowed MIN, MAX) and global (such as the histogram) *features* that are of discriminatory power. Classification attempts to associate, for each pixel, one of the few predefined classes, based on the extracted features.

Feature selection is highly application specific, and special purpose VLSI chips were extensively used in feature extraction [52]–[54]. Classification is usually done by evaluating a number of *decision functions* (one for each class), comparing their results and selecting the one with the maximum value. In Fig. 15, $X_1, \cdots, X_K$ are extracted features, $DFE_1, \cdots, DFE_C$ are the C decision function evaluators (DFE's) and *COMP* is the comparator. The most popular decision functions are low-order (first, second, or third) polynomials. For not too small number of features, polynomial classifiers are not feasible for hardware implementation because of the vast number of multipliers and the wiring cost incurred by the *crossterms* in second- or third-order polynomials.

A classical without crossterm is proposed [49] in which the decision functions are of the form

$$DFE_C(X_1, \cdots, X_K) = \sum_{i=1}^{K} g_{iC}(X_i).$$

This classifier can easily be implemented by look-up tables (for $g_{iC}(\,)$) and adders. The impact of the no-crossterm classifier is that the principle axes of the *decision region* for each class are in parallel with the feature axes. However, simulation indicates little classification inaccuracy results from this restriction.

A modular decision function evaluator (mDFE) and a modular comparator (mCOMP) architecture are developed which enable a two-chip set to be used in cascade to realize any number of features and classes. The mDFE handles four features and has a built-in partial sum chain for collecting contributions from various mDFE's. Because the features are extracted simultaneously, a variable delay is inserted to align the timing of various mDFE's along the partial sum chain. The mDFE chip architecture is shown in Fig. 16. An mCOMP in turn handles four DFE results and generates as output both the class label and the maximum DFE results, which may be used by the next mCOMP for comparison when the number of classes is large. An example of a classifier with 12 features and 7 classes is shown in Fig. 17.

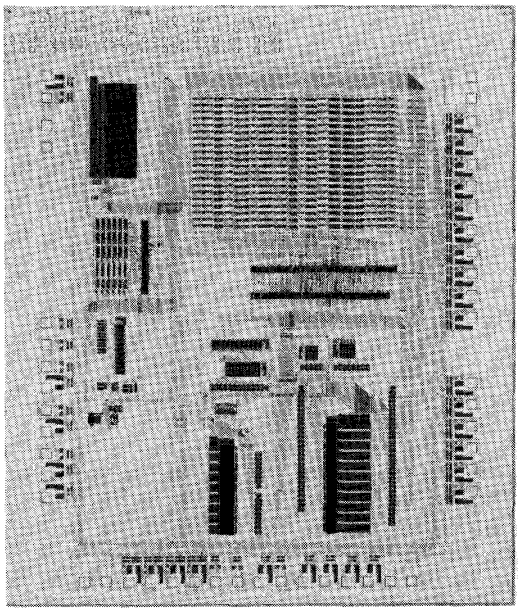


Fig. 14. The die photo of the robot arm controller chip.

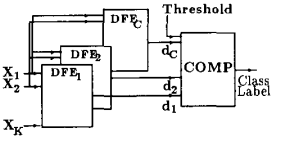


Fig. 15. Decision analyzer overview.

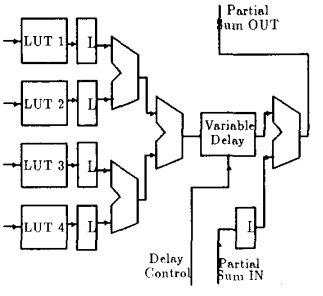


Fig. 16. mDFE chip architecture.

In Figure 18 the die photo of the mDFE chip is shown. The dimension of the chip is 9.5 mm^2 × 7.5 mm^2 in the MOSIS 1.6-μm SCMOS technology. The four look-up tables are implemented by static RAM's, and are generated using TimLager. Adders, latches, and a variable delay are all incorporated in a bit-slice data path, generated by dpp. The Global control unit is implemented with Stdcell and the placement and routing of the whole chip is done with Flint. The mDFE chip has been tested to operate at 20 MHz.

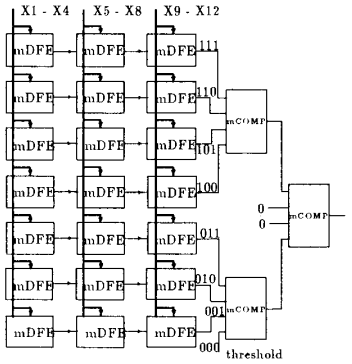


Fig. 17. A decision analyzer with 12 features and 7 classes.

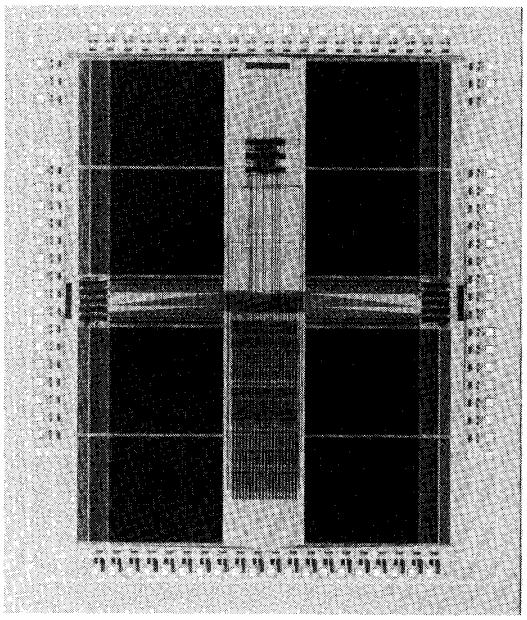


Fig. 18. The die photo of the mDFE chip.

5.3. Other Design Examples

The design results of several other chips are tabulated in Table I. These chips are designed with hardwired architectures; hence, only the LAGER silicon assembler is used. They include:

- (FIR) a predistortion FIR filter chip [55];
- (RT) an image processing chip for Radon and inverse Radon transformation [50];
- (SDSP) a fully-asynchronous DSP chip using self-timed circuits [56];
- (WP) a word processor chip for large vocabularly continuous speech recognition system [47].

The SDL entry time is the accumulated *engineering* time spent in entering the design, including modifications based on DMoct results, but not including the initial architecture definition and simulation time. The SDL files of FIR is program generated [55]. Therefore, its SDL entry time is very small.

TABLE I
DESIGN TIME AND PREFORMANCE OF SEVERAL OTHER CHIPS

Chip	SDL Entry	DMoct	Area (mm^2)	Transistors	Speed
FIR	2 min	45 min	2.7 × 6.8	11.5 K	25 MHz
RT	10 h	10 h	7.2 × 7.2	120 K	17 MHz
SDSP	20 h	16 h	8.5 × 6.5	20 K	73-330 ns
WP	10-15 h	200 h	10 × 11	25 K	20 MHz

The DMoct time is the accumulated *runtime* spent on silicon assembly, including iterations due to SDL modifications (except FIR, which indicates the silicon assembly time of one iteration), but not including the time spent in debugging the layout using IRSIM. The SDL entry and DMoct times are based on the runtime of a SUN 3/60 workstation.

All chips except RT are fabricated with the MOSIS 2-μm SCMOS technology. RT uses the MOSIS 1.6-μm SCMOS technology. The cycle time of SDSP is variable because it is input dependent.

VI. CONCLUSION

Silicon compilation systems have shown progress in the past few years, but significant breakthroughs are still required before efficient architectures and layouts can be generated from behavioral specifications. The LAGER system accomplishes this by taking both a structural input and a behavioral input. The basic design cycle involves the tuning of *both* of these inputs. The LAGER silicon assembly system is implemented using an object-oriented data base that makes the integration of new cells and CAD tools easy.

LAGER has been applied to a number of algorithm-specific IC designs. It initially was used mainly in research projects at the University of California at Berkeley, and recently has been applied at other academic and industrial institutions. Two design examples, a robot arm controller chip and a real-time image segmentation chip, are shown in this paper.

ACKNOWLEDGMENT

The authors are grateful to the following students of the University of California at Berkeley and Los Angeles who have contributed to the development of LAGER and provided valuable feedback: A. Stolzle, G. Jacobs, W. Baringer, J. Sun, S. Lee, M. Thaler, L. Svensson, P. Yang, P. Duncan, P. Tjahjadi, and W. Jao. They thank Prof. W. Reese and his group at MSU for their help in debugging and evaluating LAGER. They also want to thank the reviewers for the many helpful and constructive comments.

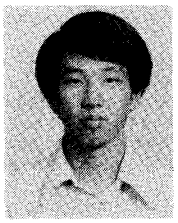
REFERENCES

[1] R. Jain, F. Catthoor, J. Vanhoof, B. D. Loore, G. Goossens, N. Goncalvez, L. Claesen, J. V. Ginderdeuren, J. Vandewalle, and H. De Man, "Custom design of a VLSI PCM-FDM transmultiplexer from system specification to circuit layout using a computer-aided design system," *IEEE J. Solid-State Circuits*, vol. SC-21, pp. 73-85, Feb. 1986.

[2] P. N. Hilfinger, "A high-level language and silicon compiler for digital signal processing," in *Proc. Custom IC Conf.*, May 1985.

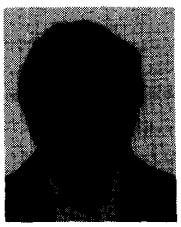
[3] P. LeGuernic, A. Benveniste, P. Bournai, and T. Gautier, "SIGNAL: A data flow oriented language for signal processing," in *VLSI Signal Processing*. New York: IEEE 1984, pp. 282–293.
[4] H. Trickey, "Flamel: A high-level hardware compiler," *IEEE Trans. Computer-Aided Design*, vol. CAD-6, pp. 259–269, Mar. 1987.
[5] M. Barbacci, "Instruction set specification (ISPS): The notation and its applications," *IEEE Trans. Comput.*, vol. 30, Jan. 1981.
[6] D. Thomas, C. Hitchcock III, T. Kowalski, J. Rajan, and R. Walker, "Automatic data path synthesis," *Comput.*, pp. 59–70, Dec. 1983.
[7] C. Tseng and D. P. Siewiorek, "Automatic synthesis of data paths in digital systems," *IEEE Trans. Computer-Aided Design*, vol. CAD-5, pp. 379–395, July 1986.
[8] N. Park and A. C. Parker, "Sehwa: A software package for synthesis of pipelines from behavioral specifications," *IEEE Trans. on Computer-Aided Design*, vol. CAD-7, pp. 356–370, Mar. 1988.
[9] P. G. Paulin and J. P. Knight, "Force-directed scheduling for the behavioral synthesis of ASIC's," *IEEE Trans. Computer-Aided Design*, vol. 8, pp. 661–679, June 1989.
[10] S. Devadas and A. R. Newton, "Algorithms for hardware allocation in data path synthesis," *IEEE Trans. Computer-Aided Design*, vol. 8, pp. 768–781, July 1989.
[11] B. Haroun and M. Elmasry, "Architectural synthesis for dsp silicon compilers," *IEEE Trans. Computer-Aided Designs*, vol. 8, pp. 431–447, Apr. 1989.
[12] J. Rabaey, H. De Man, J. Vanhoof, G. Goossens, and F. Catthoor, "Cathedral-II: A synthesis system for multiprocessor DSP systems," in *Silicon Compilation*. Reading, MA: Addison-Wesley, 1988, pp. 311–360.
[13] C. Chu, M. Potkonjak, M. Thaler, and J. Rabaey, "HYPER: An interactive synthesis environment for high performance real time applications," in *Proc. ICCD 89*, Oct. 1989, pp. 432–435.
[14] P. A. Ruetz, R. Jain, and R. W. Brodersen, "Comparison of parallel architectures for real-time image processing ICs," in *Proc. ISCAS*, Dec. 1987.
[15] N. Bergmann, "A case study of the F.I.R.S.T silicon compiler," in *Proc. Third Caltech Conf. on VLSI*, 1982.
[16] J. R. Jassica, S. Noujaim, R. Hartley, and M. J. Hartman, "A bit serial silicon compiler," in *Proc. ICCD 85*, Oct. 1985.
[17] J. Rabaey, S. Pope, and R. Brodersen, "An integrated automatic layout generation system for dsp circuits," *IEEE Trans. Computer-Aided Design*, vol. CAD-4, pp. 285–296, July 1985.
[18] J. Schuck, M. Glesner, and M. Lacken, "First results and design experience with silicon compiler ALGIC," in *VLSI Signal Processing, II*. New York: IEEE, Nov. 1986.
[19] J. R. Southard, "Macpitts: An approach to silicon compilation," *IEEE Comput. Mag.*, vol. 16, pp. 74–82, Dec. 1983.
[20] G. Zimmermann, "The Mimola design system: A computer aided digital processor design method," in *Proc. 16th Design Automation Conf.*, June 1979, pp. 53–58.
[21] S. K. Azim, "Application of silicon compilation techniques to a robot controller design," Ph.D. dissertation, Univ. of California at Berkeley, May 1988.
[22] E. Wang, "A Compiler for Silage," Master's thesis, Univ. of California at Berkeley, Dec. 1988.
[23] P. N. Hilfinger, "Silage reference manual," Internal rep.
[24] K. Rimey and P. N. Hilfinger, "A compiler for application-specific signal processors," in *VLSI Signal Processing, III*. New York: IEEE, Nov. 1988, pp. 341–351.
[25] ——, "Lazy data routing and greedy scheduling for application-specific signal procesors," in *Proc. 21th Ann. Workshop on Microprogramming*, Nov. 1988, pp. 111–115.
[26] J. A. Fisher, D. Landskov, and B. D. Shriver, "Microcode compaction: Looking backward and looking forward," in *Proc. Nat. Computer Conf.*, 1981, pp. 95–102.
[27] P. Ruetz, R. Jain, C. Shung, J. Rabaey, G. Jacobs, and R. Brodersen, "Automatic layout generation of real-time digital image processing circuits," in *Proc. CICC*, May 1986.
[28] W. Baker, J. Burns, S. Chow, D. Harrison, M. Igusa, C. Kring, T. Laidig, B. Lin, P. Moore, J. Reed, R. Rudell, C. Sechen, R. Segal, R. Spickelmier, A. Wang, A. R. Newton, and A. Sangiovanni-Vincentelli, "OCT tools distribution 2.0," Tech. Rep., Univ. of California at Berkeley, Electron. Res. Lab, Nov. 1987.
[29] R. Alverson, T. Blank, K. Choi, A. Salz, L. Soule, and T. Rokicki, "THOR user's manual," Tech. Rep. CSL-TR-88-348 and 349, Stanford Univ., Jan. 1988.
[30] A. Salz and M. Horowitz, "IRISM: An incremental MOS switch-level simulator," in *Proc. 26th ACM/IEEE Design Automation Conf.*, June 1989, pp. 173–178.
[31] "LagerIV distribution 1.0: silicon assembly system manual," Tech. Rep. Electron. Res. Lab., Univ. California Berkeley, June 1988.
[32] C. Shung, "An integrated CAD system for algorithm-specific IC design," Ph.D. dissertation, Univ. of California at Berkeley, June 1988.
[33] C. M. Sechen, "Placement and global routing of integrated circuits using simulated annealing," Ph.D. dissertation, Univ. of California at Berkeley, Dec. 1987.
[34] J. Reed, "YACR: Yet another channel router," Master's thesis, Univ. of California at Berkeley, Feb. 1985.
[35] S. Lee, "Automatic floorplanning techniques for macrocell-based layouts," Master's thesis, Univ. of California at Berkeley, 1989.
[36] L. Stockmeyer, "Optimal orientation of cells in slicing floorplan designs," *Inform. Contr.*, vol. 57, pp. 91–101, 1983.
[37] E. W. Dijkstra, "A note on two problems in connection with graphs," *Numer. Math*, vol. 1, pp. 269–271, 1959.
[38] L. Kou, G. Markowsky, and L. Berman, "A fast algorithm for steiner trees," *Acta Inform.*, vol. 15, pp. 141–145, 1981.
[39] E. R. Lettang, "Padroute: A tool for routing the bounding pads of integrated circuits," Master's thesis, Univ. California at Berkeley, May 1989.
[40] M. B. Srivastava, "Automatic generation of CMOS data paths in the LAGER framework," Master's thesis, Univ. California at Berkeley, May 1987.
[41] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell Syst. Tech. J.*, vol. 49, pp. 291–308, Feb. 1970.
[42] R. Rudell and R. Segal, "BDSYN user's manual," Tech. Rep. Berkeley CAD Tool Documentation, Univ. California at Berkeley, Jan. 1988.
[43] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, "MIS: A multiple-level logic optimization system," *IEEE Trans. Computer-Aided Design*, vol. 6, pp. 1062–1081, Nov. 1987.
[44] R. L. Rudell and A. Sangiovanni-Vincentelli, "Multiple-valued minimization for PLA optimization," *IEEE Trans. Computer-Aided Design*, vol. CAD-6, pp. 727–750, Sept. 1987.
[45] S. K. Azim, C. Shung, and R. W. Brodersen, "Automatic generation of a custom digital signal processor for an adaptive robot arm controller," in *Proc. ICASSP*, Apr. 1988.
[46] J. S. Sun, "Design and implementation of integrated circuits for a real-time flexible emulator applying silicon assembly tools," Master's thesis, Univ. California at Berkeley, Mar. 1988.
[47] A. Stolze, S. Narayanaswamy, K. Kornegay, J. Rabaey, and R. Brodersen, "A VLSI word-processor subsystem for a real-time large vocabulary continuous speech recognition system," in *Proc. CICC*, May 1989.
[48] D. Chen, R. Yu, J. Rabaey, and R. Brodersen, "A VLSI implementation for the grammar processor subsystem for a real-time large vocabulary continuous speech recognition system," in *Proc. CICC*, May 1990.
[49] C. B. Shung, W. E. Blanz, and D. Petković, "Real-time decision analysis—algorithms, architecture and implementation," Res. Rep. RJ 6526 (63327), IBM, Nov. 1988.
[50] W. Baringer, B. Richards, R. Brodersen, J. Sanz, and D. Petkovic, "A VLSI implementation of PPPE for real-time image processing in radon space—work in progress," in *Proc. 1987 Workshop on Computer Architecture for Pattern Analysis and Machine Intelligence*, Oct. 1987, pp. 88–93.
[51] J. Sklansky and G. Wassel, *Pattern Classifiers and Trainable Machines*. New York: Springer-Verlag, 1981.
[52] L. Palmier, M. P. Gayrard, and B. Zavidovique, "VLSI architecture of the "curve" function in image processing," in *Proc. 1985 IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, Los Alamitos, CA, Nov. 1985, pp. 284–287.
[53] M. Hatamian, "A real-time two-dimensional moment generating algorithm and its single chip implementation," *IEEE Trans. Acoust., Speech, Signal Processing.*, vol. ASSP-34, pp. 546–552, 1986.

[54] P. A. Ruetz and P. Ang, "A 20 MHz chip-set for image processing," in *Tech. Dig. 1988 Int. Solid-State Circuits Conf.*, San Francisco, CA, Feb. 1988.
[55] R. Jain, P. Yang, B. Chung, and C. Chien, "A CAD system for automatic layout generation of high-performance FIR filters," in *Proc. CICC*, May 1990.
[56] G. Jacobs and R. Brodersen, "A fully-asynchronous digital signal processor using self-timed circuits," in *Tech. Dig. 1989 Int. Solid-State Circuits Conf.*, (San Francisco, CA), Feb. 1989.



C. Bernard Shung received the B.S. degree from the National Taiwan University in 1981, and the M.S. and Ph.D. degrees from the University of California, Berkeley in 1985 and 1988, respectively, all in electrical engineering.

In 1988 he joined the IBM Research Division, Almaden Research Center in San Jose, CA, where he was involved in designing VLSI chips for computer vision for process inspection and error-correction coding for magnetic recording. Since 1990 he has been an Associate Professor with the Department of Electronics Engineering, National Chiao Tung University in Hsinchu, Taiwan, Republic of China. His research interests include computer-aided design for VLSI circuits, and VLSI system and architecture design for signal/image processing and communications.



Rajeev Jain (S'83–M'84) received the B. Tech. degree from the IIT Delhi and the Ph.D. degree from the Katholieke Universiteit Leuven.

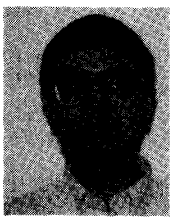
He has worked at Siemens AG, Munich on DSP design for an oversampled PCM CODEC. He was a group leader at IMEC, Leuven, for the ESPIRIT sponsored Cathedral silicon compiler project. During 1985–1988 he was on the research staff at ERL, University of California Berkeley where he developed the LagerIV silicon assembly system. Since 1988 he has been on the faculty at University of California at Los Angeles where he has initiated research in computer-aided design of high-performance DSP circuits.

Ken Rimey received the B.S. degree in physics from the Stevens Institute of Technology, Hoboken, NJ, in 1981, and the M.A. degree in physics and the Ph.D. degree in computer science from the University of California, Berkeley, in 1983 and 1989, respectively.

He is currently an Acting Assistant Professor in the Department of Computer Science at the Helsinki University of Technology, Finland. His research interests include compiler design, Lisp systems, and computer algebra.

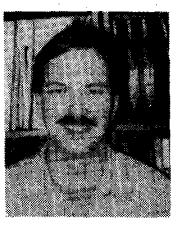
Edward Wang is currently a graduate student with the Department of Electrical Engineering of California at Berkeley.

Mr. Wang is a member of the ACM.



Mani B. Srivastava received the B.Tech. degree (summa cum laude) in electrical engineering from the Indian Institute of Technology at Kanpur, India, in 1985 and the M.S. degree in electrical engineering and computer sciences from the University of California at Berkeley, California in 1987. He is currently working towards the Ph.D. degree at the University of California at Berkeley.

His research interests include custom VLSI and board-level hardware, architecture and computer-aided design tools for real-time systems in digital signal processing and robot control.



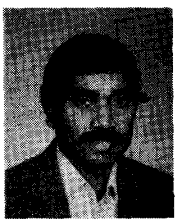
Brian C. Richards received the B.S. degree in electrical engineering from the California Institute of Technology in 1983, and the M.S. degree in electrical engineering and computer science from the University of California, Berkeley in 1986.

In 1986, he joined the technical staff at the University of California, Berkeley, where he is currently maintaining and continuing the development of several VLSI and system design CAD tools.

Erik Lettang (S'86–M'88) received the B.S. degree and M.S. degree in electrical engineering and computer science from the University of California at Berkeley in 1986 and 1989, resepctively.

From 1986 to 1989 he worked as a research assistant at University of California at Berkeley. He is currently employed at Hewlett-Packard's San Diego Division.

Mr. Lettang is a member of Eta Kappa Nu, and Tau Beta Pi.



S. Khalid Azim (M'88) received the B.Sc. degree in electrical engineering from the Bangladesh University of Engineering and Technology, Dhaka, in 1976, the M.S.E.E. degree from the University of Houston, Texas, in 1980, and the Ph.D. degree in electrical engineering and computer science from the University of California, Berkeley, in 1988.

He was an engineer with IBM World Trade Corp., Bangladesh, from 1977 to 1978 and worked in the microprocessor group at National Semiconductor Corporation, Santa Clara, CA from 1980 to 1983. Since 1988 he has been a Member of the Technical Staff at AT&T Bell

Laboratories in Allentown, PA. His technical interests are in the area of design of VLSI subsystems for digital signal processing and communications, CAD, and exploration of automated design techniques for custom chips.

Dr. Azim is a member of Sigma Xi.

Lars Thon (S'87) received the B.Sc. degree in electrical engineering from the Norwegian Institute of Technology in 1984 and the M.Sc. from the University of California at Berkeley in 1987. He is currently working towards the Ph.D. at University of California at Berkeley.

His research interests include high-level integrated circuit design systems, and architecture and implementation of application specific processors for numerical algorithms, with an emphasis on robotic applications.

Paul N. Hilfinger received the A.B. degrees in mathematics from Princeton University in 1973 and the Ph.D. degree from Carnegie Mellon University in 1981.

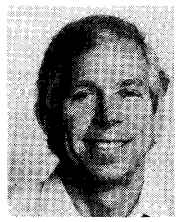
He worked on the Ada language design in 1980, served as a member of the Ada Board until 1987, and is presently a member of the ISO working group on Ada. In 1982, he joined the Department of Electrical Engineering and Computer Service, University of California at Berkeley, where he is currently an Associate Professor. His research interests include languages and software support for scientific computation, parallel computation, program semantics, compiler technology, and software engineering.

Dr. Hilfinger is a member of the ACM.

Jan M. Rabaey received the E.E. and Ph.D. degrees in applied sciences from the Katholieke Universiteit Leuven, Belgium, respectively, in 1978 and 1983.

From 1983 to 1985, he was with the University of California, Berkeley as a Visiting Research Engineer, where he developed an automated synthesis system for multiprocessor DSP architectures. From 1985 to 1987, he was Head of the Architectural and Algorithmic Strategies' Group in the VSDM (VLSI System Design Methodologies) section of the IMEC Laboratory, Leuven, Belgium. In 1987, he joined the faculty of the University of California, Berkeley, where he is currently an Assoicate Professor. His main interests are in the study of signal processing architectures and in computer-aided analysis, and synthesis and design of digital signal processing systems. He has authored or co-authored more than 50 publications.

Dr. Rabaey received the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN Best Paper Award in 1986. In 1989, he received the Presidential Young Investigators Award.



Robert W. Brodersen (M'76-SM'81-F'82) received the B.S. degrees in electrical engineering and mathematics from the California State Polytechnic University in 1966, and the Eng., M.S., and Ph.D. degrees from the Massachusetts Institute of Technology in 1968 and 1972, respectively.

From 1972 to 1976, he was with the Central Research Laboratory, Texas Instruments, Inc., Dallas, TX. In 1976, he joined the faculty of the University of California at Berkeley, where he is currently a Professor. His research interests include the use of MOS technology for signal processing applications.

Dr. Brodersen received the W. G. Baker Award for the Outstanding Paper in IEEE Journals and Transactions. In 1986, he received the IEEE Circuits and Systems Society's Best Paper Award for the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN. In 1987, he received the Circuits and Systems Society Technical Achievement Award. He is a member of the National Academy of Engineering.