

# An Integrated CAD System for Algorithm-Specific IC Design

C.S. Shung<sup>1</sup> R. Jain<sup>2</sup> K. Rimey E. Wang M.B. Srivastava  
E. Lettang S.K. Azim<sup>3</sup> P.N. Hilfinger J. Rabaey R.W. Brodersen

Electronics Research Laboratory  
Department of Electrical Engineering and Computer Science  
University of California, Berkeley

## Abstract

Lager is an integrated CAD system for algorithm-specific IC design. It consists of a behavioral mapper and a silicon assembler. To generate a chip from a behavioral description, the user specifies both the behavioral description and a parameterized structural description. The behavioral mapper maps the behavior onto the parameterized structure to produce microcode and parameter values. The silicon assembler then translates the filled-out structural description into a physical layout. A number of algorithm-specific IC's designed with Lager have been fabricated and tested. A robot-control chip is described here.

## 1 Introduction

Modern integrated circuits (ICs) fall into two groups: *commodity* ICs and *application-specific* ICs (ASICs). Examples of commodity ICs include memory chips, TTL MSI and LSI gates, and microprocessor chips. Commodity chips are usually produced in large volume with fully custom design. Examples of ASICs include computer peripheral chips and digital signal processing ICs. The turn-around time of ASICs is often more important than the area, emphasizing the need for computer-aided design (CAD) tools.

ASICs achieve high performance through architectural innovation. Due to their application-specific nature, each design can exploit the special conditions in the particular application to create an efficient implementation. A disadvantage of ASICs, as compared to off-the-shelf commodity ICs, is the fabrication time that they require, although this has been reported to be as short as two weeks with some gate-array design systems. Because the design time of an IC is usually much longer than the fabrication time, reducing the design time is the most effective way to reduce the overall turn-around time.

This paper focuses on ASICs that implement computational algorithms. We call these *algorithm-specific* ICs. From experience with several algorithm-specific IC designs [1,2,3], we find that diverse algorithms can often be implemented with a single, well-designed set of hardware modules. Reusing hardware modules greatly reduces design time.

<sup>1</sup>Currently at IBM Almaden Research Center.

<sup>2</sup>Currently at UCLA, Department of Electrical Engineering.

<sup>3</sup>Currently at AT&T Bell Laboratories, Allentown, Pennsylvania.

Typical application areas for algorithm-specific ICs include speech processing, image processing, robot control, computer vision, digital audio, and telecommunications. Research projects involving algorithm-specific IC designs underway in our group include inverse kinematics of robot arms, adaptive equalization for mobile radio, fingerprint recognition, frame buffer control, channel emulation for computer network protocols, and image processing based on projection-transformed data.

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 ICs. In Section 2, an overview of the Lager system is presented and comparisons are made with related work. In Section 3, the behavioral mapper is described. In Section 4, the silicon assembler is described. Two algorithm-specific IC design examples are described in Section 5.

## 2 CAD Systems for Algorithm-Specific IC Design

### 2.1 Background

Two types of architectures are used in designing algorithm-specific ICs: *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 datapath may be complicated, the control unit will be simple because there is no time-multiplexing of hardware modules. The speed of the implementation is limited by the speed of the slowest hardware module in the design. The main drawback of hardwired architectures is that they have to be redesigned for each new algorithm.

A programmable architecture consists of a carefully chosen set of hardware modules, time-multiplexed (by microcode control) according to the algorithm. The control unit for a programmable architecture is necessarily more complex than one for a hardwired architecture. The speed of the implementation depends on the total number of microcode cycles required to realize the algorithm. Hence, a programmable architecture can be used only if

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

For example, if the circuit runs at 5 MHz and the sample fre-

quency is 5 kHz, the number of cycles must be no greater than 1000. Because a single programmable architecture can be used for many applications, a programmable architecture 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.

Design representations of algorithm-specific ICs may be classified by abstraction level: *behavioral*, *structural*, *physical*. A behavioral representation specifies only the algorithm that the chip should implement, not the hardware modules that it should incorporate. It 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 interconnections. A physical representation additionally specifies placement and routing—in other words, a layout.

In this paper, generation of layout from a structural description is called *silicon assembly*. We reserve the term *silicon compilation* for the harder task of working from a behavioral description.

## 2.2 Related Work

Silicon compilation from a behavioral description has been an active research area for a number of years. A number of systems have been proposed, each with a different behavioral language. Behavioral specifications fall into three groups:

**Frequency-domain specification.** This kind of behavioral specification describes the desired frequency-domain behavior. Typical parameters include the passband ripple, stopband ripple, and stopband attenuation. A filter is synthesized from parameters such as these and its coefficients are optimized for hardware realization. The *Cathedral-I* system [7], which uses a fixed bit-serial architecture, has been designed along these lines. Frequency-domain specifications are only applicable for a limited range of applications (e.g. filters).

**Machine specification.** An example of this kind of behavioral specification is the ISPS [8] language, on which the CMU-DA [4] system is based. The ISPS language describes the instruction set of an architecture and the machine behavior in executing each instruction. The CMU-DA system synthesizes a datapath that implements the specified instruction set. Both an iterative algorithmic approach (*EMUCS*) and a knowledge-based approach (*DAA*) have been tried in the CMU-DA system.

**Algorithm specification.** In this kind of the behavioral description, an arbitrary algorithm may be specified. For example, the Cathedral-II system [5] maps algorithms expressed in the Silage language to a fixed architecture. Algorithm specification is the approach used in Lager.

Thomas et al. [4] describe the *datapath synthesis* approach to behavioral synthesis. In this approach, a datapath is first synthesized to suit the algorithm. The algorithm is then scheduled onto that datapath to produce the microcode. The datapath synthesis approach is not yet able to produce efficient (in terms of area and performance) datapath architectures for a wide range of applications. It is too difficult for a single synthesis program to weigh all the options in architecture design. In most systems, high-level decisions—bus structure, pipelined vs. parallel architecture, lumped ALU vs distributed functional modules—are pre-

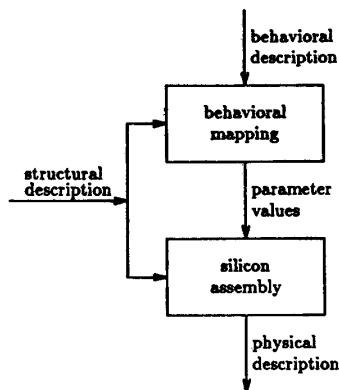


Figure 2.1: Lager system overview

determined to make the problem tractable. A rule-based implementation for the datapath synthesizer was used in Cathedral-II [5], but each rule set was found to be strongly tied to a single architecture. Ruetz et al., in their paper on real-time image processing [6], point out the difficulty of using behavioral level description when high-speed I/O is an important consideration.

A few silicon assembly systems have recently been developed in the CAD industry. LSI Logic's *silicon integrator*, VLSI Technology Inc.'s *VTItools*, Silicon Compiler Systems's *Genesil* and Seattle Silicon Technology's *Concorde* are the best known examples. The users of these systems usually specify designs using schematic entry, which can be viewed as graphical structural description. Without the capability of taking behavioral description as input, these systems cannot help users in translating algorithms to structural designs.

## 2.3 Overview of Lager

A block diagram of the Lager system is shown in Figure 2.1. Lager consists of a *behavioral mapper* and a *silicon assembler*. Structural descriptions are parameterized to facilitate the reuse of hardware modules. The silicon assembler requires both a structural description and associated parameter values (perhaps including microcode) to generate a layout. The behavioral mapper maps a behavioral description onto a user-specified structural design by generating the appropriate parameter values. Together, the behavioral mapper and silicon assembler support design with both hardwired and programmable architectures. The behavioral mapper is compatible only with programmable architectures, but the silicon assembler can be used alone for high-sample-rate applications, such as image processing, that require hardwired architectures.

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 *re-targeted* 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.

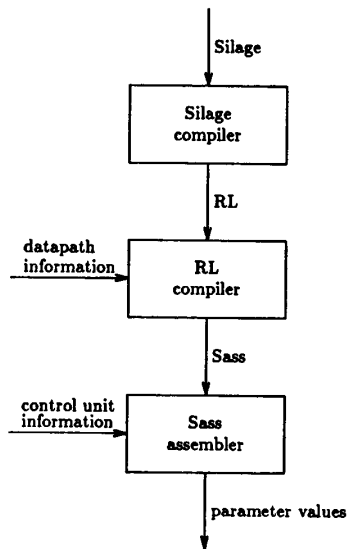


Figure 3.1: Behavioral mapper flow diagram

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 communication among tools is through an object-oriented database. Integrating the tools is easy because they follow an agreed-upon database policy. The design manager builds the database from the structural input. The layout generation tools also access an open cell library of leaf cells and parameterized modules. New modules are added to the library by providing structural descriptions; new cells are added by providing simulation models and layouts.

### 3 Behavioral Mapping

Lager can work from a behavior description. The method used *maps* the behavioral description into parameter values, completing the parameterized structural description, which in turn becomes the input to the silicon assembler.

Figure 3.1 shows the structure of the behavioral mapper. We employ three independent input languages. *Silage* is a dataflow language with very high level constructs suited to our 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. Figure 3.2 shows a simple filter described in the three

```

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

Figure 3.2a: The IIR filter described in Silage

```

fix y;

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

Figure 3.2b: The IIR filter described in RL

```

(ram y)
(cfem (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))))
  
```

Figure 3.2c: The IIR filter described in Sass

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 3.2 to 3.4.

Section 3.1 describes the Kappa family of processor architectures. Kappa architectures are small, tunable—hence appropriate as components of ASICs. RL and Sass are designed for use only with the Kappa style of architecture. Silage, however, will be the basis of our research in direct generation of structure and automatic program partitioning for parallel processors.

#### 3.1 The Kappa Family of Processor Architectures

*Kappa* [9,10] is a processor architecture that has served well as a prototype for customization. The behavioral mapper generates code for *Kappa-like* horizontal-instruction-word architectures. A horizontal instruction word is just a vector of control signals with little or no restrictive encoding. Thus horizontal-instruction-word architectures execute a machine language that resembles the horizontal microcode of a two-level general-purpose computer.

Kappa's datapath is shown in Figure 3.3. A pipeline delay of one instruction-time is associated with every register and register bank in the figure; these delays include those of the func-

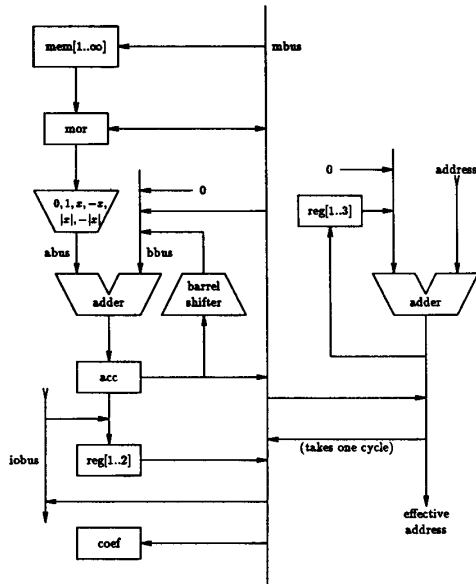


Figure 3.3: The datapath of the prototypical Kappa architecture (with some details omitted).

tional 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 instruction words are fully horizontal.

Kappa is *irregular*, meaning that its datapath topology has been chosen to suit the usage in a particular program (a robot controller) rather than to conform to the expectations of modern compilers. The latter alternative would be exemplified by an architecture in which all intermediate results are stored in a large, central, highly-multiported register bank. An irregular datapath is smaller, faster, more tunable, and hence more appropriate for inclusion in an ASIC. The problems that irregularity poses for re-targetable compilers are minimized by the use of fully horizontal instruction words.

Kappa incorporates some unusual control mechanisms. A state machine that operates in parallel with the datapath performs boolean operations and conditions certain datapath operations. For looping and more complex decision-making, the control unit (which incorporates another state machine) provides a *multi-way jump/call/return* capability.

### 3.2 Silage

Silage is a very high level language for specifying algorithms in common ASIC applications (in particular, digital signal processing and real-time control). As such, its design emphasizes expressiveness over details of implementation, by providing the programmer 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 [11,12]. The “Silage Reference

Manual” [13] specifies the language in full.

#### 3.2.1 The Silage Language

Silage is a dataflow language that operates on streams of values. As in all dataflow languages, a Silage program corresponds to a dataflow 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, and produces output streams. This model of computation is natural for our intended applications. Real-time control typically takes time-sampled input streams from sensors to produce control signals. Digital signal processing 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 these 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. For example, this code fragment adds elements 1 to 5 of array A:

```
sum[0] = 0;
(i: 1 .. 5) :: sum[i] = sum[i - 1] + A[i];
```

The result is in sum[5]. Subscripted variables are no different from simple variables. Specifically, they obey the single assignment rule. In the example, each element of sum must be defined exactly once. An iterated definition is exactly equivalent to the same definition replicated, with occurrences of the index variable replaced by values in the iteration range, as shown here:

```
sum[0] = 0;
sum[1] = sum[1 - 1] + A[1];
sum[2] = sum[2 - 1] + A[2];
sum[3] = sum[3 - 1] + A[3];
sum[4] = sum[4 - 1] + A[4];
sum[5] = sum[5 - 1] + A[5];
```

The iterator can specify both parallel and sequential computation. The only difference is in the dependence (or lack of dependence) between iterations.

#### 3.2.2 The Silage Compiler

The current Silage compiler translates Silage programs into RL to run on a single processor. The main difficulties are in converting the unique features of Silage into the conventional operations supported by RL. These include modeling dataflow 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 dataflow 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 dataflow 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 indices.

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 existing program loops resulting in code that is no less efficient than that using special addressing modes.

### 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 signal processing program. On the other hand, the small size, high speed, and plasticity required of algorithm-specific processors have led to abandonment of the architectural regularity usually expected by compiler writers.

We give only a brief overview of the RL compiler here. Rimey and Hilfinger describe, in more detail, the overall compiler [14] and the novel code-generation techniques that it uses [15].

#### 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 not usually write the machine description from scratch.

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 busses, latches, registers, and register banks;
- a list of simple register transfers (e.g.,  $abus \rightarrow acc$ ) defining the topology of the datapath;
- a list of functional register transfers (e.g.,  $abus + bbus \rightarrow acc$ ) representing capabilities of the functional units;
- macros for operations that require coordination of more than one component of the datapath.

The compiler automatically uses simple transfers to chain together functional transfers; macros need not be written to accomplish this. It selects from among 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 in the rare case that 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 straight-line segments of code, expressed in an intermediate language. Then, for each straight-line segment, the back end selects register transfers and packs them into instruction words.

In addition to routine tasks and simple optimizations, the front end performs two optimizations that take advantage of Kappa's unusual features. 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 multi-way jump/call/return.

Most of the effort in developing the RL compiler has gone into developing the algorithms used in the back end. The usual approach to generating horizontal code has been to first generate loose sequences of register transfers and then pack these tightly into a small number of instructions through *compaction* [16].

More recently, the integration of register-transfer selection and local compaction into local *scheduling* has been considered. With this comes 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. This is the source of code improvement that we have vigorously pursued. For Kappa-like architectures, scheduling with lazy data routing is both profitable and difficult. It is difficult because 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 [15].

The RL compiler is written in Lisp and compiles approximately one line of RL 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 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.

The body of a Sass program has two parts: straight-line code blocks and control flow information. A straight-line code block is a sequence of datapath instructions, uninterrupted by branches. Each datapath instruction consists of a number of microoperations, which perform arithmetic, logic, and addressing functions. Sass programs also define parameter values to complete the machine description. For example, the width of the datapath is ordinarily specified this way.

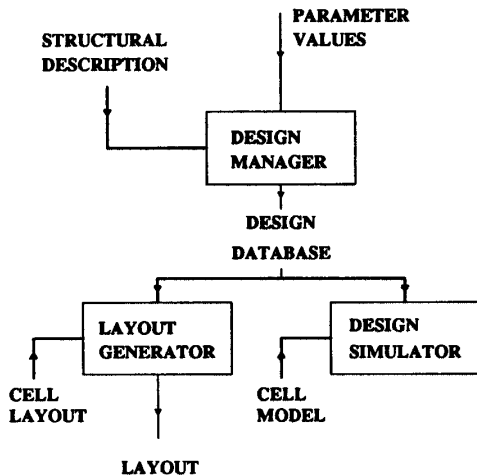


Figure 4.1: Silicon Assembler flow diagram

The machine description specifies the control signals and hardware resources used by each microoperation. The Sass assembler uses it to assemble datapath instructions into binary microcode, and to check for resource conflict errors between microoperations.

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.

## 4 The Silicon Assembler

The silicon assembler performs layout generation and simulation from a structural description. It can be used in conjunction with the behavioral mapper when the design is implemented with a programmable architecture. It can also be used directly when the design is implemented with a hardwired architecture.

The silicon assembler consists of the *design database*, the *design manager*, the *layout generator*, and the *design simulator*. The flow diagram of the silicon assembler is shown in Figure 4.1.

### 4.1 The Design Database

As the number of CAD tools in a system increases, the integration of tools becomes more and more important. Writing an interface program for each pair of tools that communicate is time consuming and error prone. A better integration scheme is to design a database and let each CAD tool read and write information in the database, thereby communicating with the other tools. The way the information is organized in the database is called the *policy* of the database. A good policy offers higher overall efficiency for the entire system. New tools designed with the database policy

in mind are easy to integrate.

The Lager silicon assembler uses an *object-oriented* database, which offers two advantages to the CAD tool developer. First, abstract data types eliminate the need for the tool developer to keep track of data types. Second, procedures communicate by *messages*. This mechanism only requires knowledge of the external behavior of an addressed procedure. Thus the implementation of one procedure can be modified without affecting others.

The current database manager is implemented using Flavors, a lisp-based, object-oriented programming system. A new version of Lager, which uses the Oct database manager [17], has recently been completed. Adapting the system to Oct was straightforward because it was possible to retain the overall design. The new version uses the same layout generation tools.

### 4.2 The Design Manager

The design manager builds the design database from the structural description of a design. The structural description consists of a hierarchy of parameterized *structural description language* (SDL) files, together with parameter values. The hierarchical organization and the use of parameters facilitate creating reusable SDL files. In practice, a design will use both library SDL files and user-defined SDL files. The latter may later be incorporated into the library.

An SDL file specifies how a cell (the *parent* cell) is constructed out of other cells (the *subcells*). A sample SDL file is shown in Figure 4.2. In the example, the parent cell, sorter, has two subcells, pr and pads. Sorter has one parameter, A. Different sorter cells are defined by supplying different values for A. The parameter values of the subcells are specified as *expressions* in terms of the parameters of the parent cell. The interconnection of cells (among subcells and between parent cell and subcell) is specified by *nets*. The SDL file also has information for layout generation, and an optional functional model for simulation.

The design manager traverses the SDL file hierarchy (cell hierarchy) in *preorder* (i.e., visiting the parent before the subcells) because the names of the subcells are not known until the parent cell is examined. When visiting the parent cell, the design manager puts new subcell, terminal, and net objects into the design database, evaluates parameter expressions, and puts parameter values into the database.

### 4.3 The Layout Generator

The layout generator integrates five tools: TimLager, DPC, Wolfe, Flint, and Padroute. TimLager, DPC, and Wolfe are used to generate building blocks such as memories, datapaths, and logic blocks. Flint and Padroute are used to put the building blocks together to form the chip. We have found these tools to be sufficient for the design of many algorithm-specific ICs.

TimLager is a tiling tool that puts together cells by abutment. It is normally used to generate array structures such as RAMs, ROMs and PLAs. A C routine is used to specify cell placement. This approach is more expressive than the *personality matrix* approach. For example, C control constructs, such as for-loops, can be exploited. DPC (Datapath Compiler [18]) generates sliced datapaths. Unlike TimLager, which abuts leaf cells, DPC performs routing in the data signal direction and abutment in the control signal direction. Wolfe is a standard-cell place-and-

```

(layout-generator Flint)
(parent-cell sorter (parameter A))
(sub-cells
  (pr pr (parameter (B (+ A 2))))
  (pads pads))

(net ctrlnet ((pads a) (pr ctrl)))
(net innet0 ((pads b) (pr in 0)))
(net innet1 ((pads c) (pr in 1)))
(net outnet ((pads d) (pr out)))

```

Figure 4.2a: Sorter SDL file

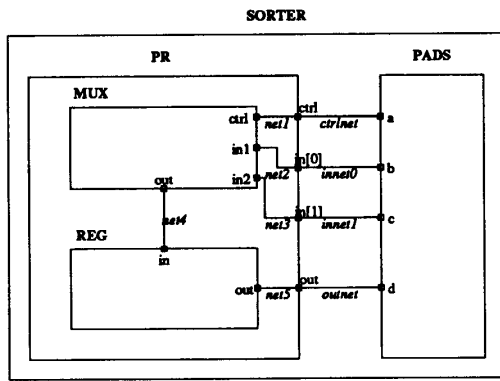


Figure 4.2b: Sorter overview

route tool. It is used to generate random logic blocks and state machines.

Flint is a general-purpose place-and-route tool. The user has the option of choosing cell placement, channel assignment, and global routing arrangements. Flint takes care of the details of routing automatically. Padroute is used to route the chip core to the ring of I/O pads. It adjusts the dimension of the pad ring, as determined by the chip core dimension and the routing area.

The layout generator traverses the cell hierarchy in *postorder* (i.e., visiting the subcells before the parent cell), because the layout of the parent cell depends on the layout of the subcells. When visiting a cell, it sends a *layout-gen* message to the cell. The external protocol of layout-gen is to determine the size and terminal locations by first sending the same message to the subcells. The procedure (*method*) that a cell uses to handle the layout-gen message is determined by the layout generation tool associated with the cell. New layout generation tools are integrated easily by defining new layout-gen methods. Existing methods need not be changed.

We use time stamps on layout files to determine whether a new layout needs to be generated. The layout of a cell is regenerated if and only if one of these three has changed since the last layout: the parameter values, the SDL file of the cell, and the layout of the subcells.

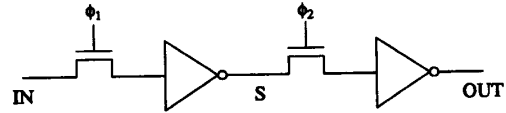


Figure 4.3a: Register-cell schematic

```

(sim-list function
  (in-term (IN width) phi_1 phi_2)
  (out-term (OUT width))
  (local S)
  (function
    (OUT
      (if (eq phi_2 '1 1) then (Xlognot S)
          elseif (eq phi_2 '0 1) then 'HZ
          else 'X))
    (S
      (if (eq phi_1 '1 1) then (Xlognot IN)
          elseif (eq phi_1 '0 1) then 'HZ
          else 'X))))

```

Figure 4.3b: Register-cell functional model

#### 4.4 The Design Simulator

The design simulator is an event-driven functional simulator. It can be used to simulate the user-specified structural description before layout generation is performed. It can also be used as a verification tool for the behavioral mapper, by comparing its simulation results with those of a behavioral simulator. The design simulator obtains the structural description, parameter values, and functional models from the design database. Using a single database helps keep design simulation consistent with layout generation.

As an example, the schematic and the functional model of a register cell are shown in Figure 4.3. In addition to the input terminal IN and the output terminal OUT, a *local terminal* S is defined. Local terminals are convenient for describing local state and for replacing common sub-expressions. The functional model contains a function definition for each output terminal and each local terminal, specifying its relationship to other terminals in the same cell. The terminal S, for example, is the logical inverse of the terminal IN when  $\phi_1$  is equal to one.

Transmission gates are handled specially by the design simulator, because their bidirectionality is difficult to model. If a gate is on, then the nets connected to the source and drain terminals are made equivalent. If the gate is off, then the nets are left independent. In effect, transmission gates have a *dynamic* functional model.

The design simulator uses parameterized functional models. This feature is useful, for example, for describing registers and busses. The width of a register or bus is a parameter. For example, the register-cell functional model in Figure 4.3 can be used to model registers of arbitrary width.

The design simulator is rather slow; it is only two to five times faster than Esim, a switch-level simulator. Although there is

room for further fine-tuning, we have decided instead to adopt the Thor functional simulator [19], which works in a similar manner but uses a compiled C functional description.

## 5 Design Examples

Application-specific ICs for a variety of applications have been designed and fabricated using the Lager system. They include a low-level trajectory controller for a two-joint robot arm, a chip set for real-time emulation of communication channels in computer networks, a frame buffer controller for use in image processing applications, and an image processing chip for converting images from the spatial domain into Radon transform space and vice versa. Still under development are chips for real-time speech recognition, 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 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 net-list description.

Other chips, such as the robot 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 Kappa architecture. They have been designed or are being designed using both the behavioral mapper and the silicon assembler.

The remainder of this section describes the design of the robot-control chip in detail to illustrate the full spectrum of capabilities of Lager. The robot-control chip [10] is the heart of a robot-control system that directs a two-joint, direct-drive robot arm along a desired trajectory in real time. The nonlinearities inherent in the dynamics of a robot arm preclude achieving high speed and precision using standard control schemes (like PID without nonlinear compensation). Further, to take into account modeling uncertainties and payload variations, an adaptive scheme is preferable, especially for a direct-drive arm. The robot controller uses a *model-reference adaptive control* (MRAC) algorithm, which takes into account the nonlinearities in the arm dynamics and adaptively determines the arm's parameters at run time.

On an IBM PC, the MRAC algorithm achieves a 7-millisecond sample period. Implemented on a TMS32010-based board, it achieves a sample period of 0.7 milliseconds, but at a significant cost in hardware and board area. For higher speed, a more complex algorithm, or a reduced amount of hardware, a custom chip is appropriate. The custom robot-control chip not only achieves a higher speed, with a sample period of less than 0.5 milliseconds, but also reduces the I/O hardware required. Customizing the chip I/O to the robot interface reduces the board area significantly.

The chip was designed using the Kappa architecture. The robot-control algorithm, unlike many digital signal processing algorithms, has many conditional branches and loops. Kappa provides hardware for efficient handling of these operations.

The chip consists of the following independent structural blocks, as shown in Figure 5.1:

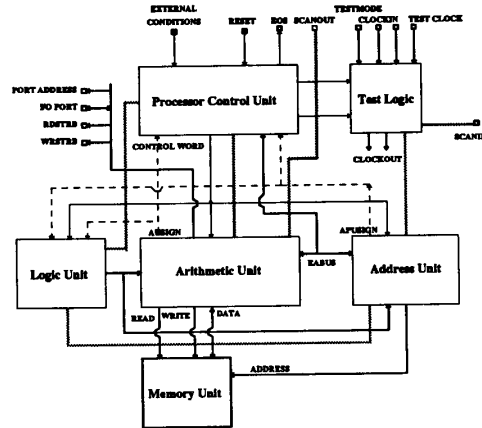


Figure 5.1: Top-level interconnection network of the robot-control chip. Dotted lines indicate data path status signals. Scan path connections are shown as cross-hatched lines.

**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 datapath, used to perform fixed-point arithmetic. It consists of a bit-slice datapath 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 datapath that performs address computations. Like the arithmetic unit, it consists of a bit-slice datapath 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 *scan-path* technique.

The layout of the chip is generated from a parameterized structural description. The chip is described hierarchically, using SDL files as shown in Figure 5.2. 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, which implements the special placement and routing strategies required for pads. The pad groups are assembled as linear tilings of pads, using the procedural tiler TimLager.

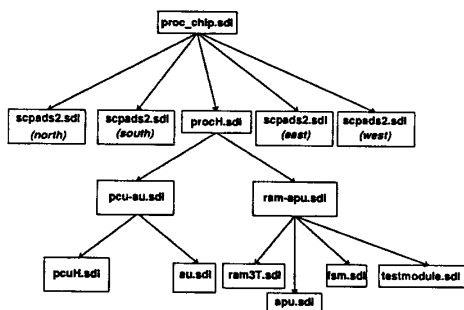


Figure 5.2: The structural hierarchy of the robot-control chip. The six cells at the bottom are the six major macrocells in the chip.

The core section consisting of the six major functional units described above is generated using the macrocell placement and routing program, Flint. The finite state machine PLAs, the control store, and the memory are generated from parameterized descriptions by TimLager. The bit-slice datapaths are generated by the datapath placement and routing program, DPC, using parameterized modules from the library.

The random logic required for decoding control words and generating local clock signals is generated from a Lisp description of the logic equations. A program called *eqn2sdl* implements the equations using combinational logic and latches from a standard-cell library. It outputs a net-list in the form of an SDL file. Then a standard-cell placement and routing program called Wolfe generates the layout for the random logic.

Flint connects the bit-slice datapath and the associated control logic to form the complete datapath. Flint is also used to connect two tiled blocks, made by TimLager, to form the memory unit. In this manner, the entire chip layout is generated hierarchically from the SDL description.

Values for all of the parameters in the SDL description need to be provided before layout can be generated. Some parameters, such as the word widths of the datapaths, are provided in the behavioral description. Other parameters, such as the contents of the control store and the personality matrix of the PLA in the finite state machine, are generated by the behavioral mapper.

The algorithm for the robot controller was written in Sass. A die photograph of the chip is shown in Figure 5.3.

Since the SDL description is parameterized, the same set of SDL files can be customized for a different application by giving different values to the parameters in order to alter, for example, the architectures of the datapaths. In fact, variants of the same SDL files are being used to implement the inverse kinematics of a six-joint robot arm and to implement an adaptive equalizer for use in digital mobile radio.

## 6 Summary

Silicon compilation systems have shown progress in the past few years, but significant breakthroughs are still required before efficient architectures will be generated from behavioral specifications.

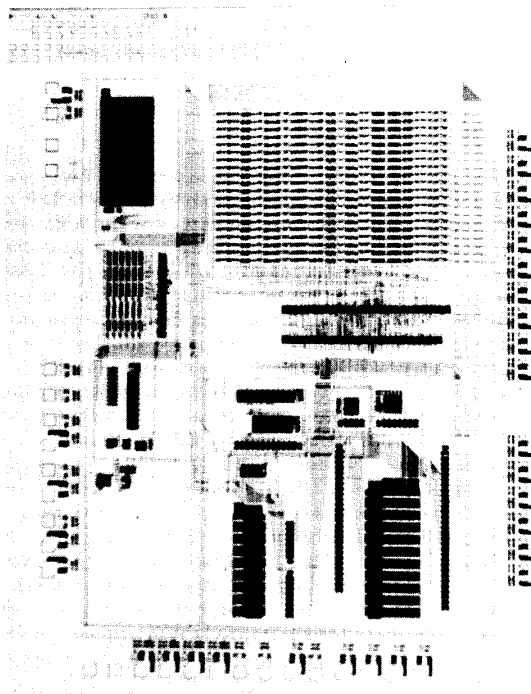


Figure 5.3: Die photograph of the robot-control chip.

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 CAD community is expending considerable effort in making silicon assembly systems *open* to new cells and CAD tools. The Lager silicon assembly system is implemented in an object-oriented environment that makes the integration of new cells and CAD tools easy.

## References

- [1] R. A. Kavalier, *The Design and Evaluation of A Speech Recognition System for Engineering Workstations*. PhD thesis, University of California at Berkeley, May 1985.
- [2] P. A. Ruetz, *Architectures and Design Techniques for Real-Time Image Processing ICs*. PhD thesis, University of California at Berkeley, May 1986.
- [3] S. P. Pope, *Automatic Generation of Signal Processing Integrated Circuits*. PhD thesis, University of California at Berkeley, Feb. 1985.
- [4] D. Thomas, C. H. III, T. Kowalski, J. Rajan, and R. Walker, "Automatic data path synthesis," *Computer*, pp. 59-70, Dec. 1983.
- [5] J. Rabaey, H. D. Man, J. Vanhoof, G. Goossens, and F. Catthoor, "Cathedral-II: a synthesis system for multi-

- processor DSP systems," in *Silicon Compilation*, Addison-Wesley, Dec. 1987.
- [6] P. A. Ruetz, R. Jain, and R. W. Brodersen, "Comparison of parallel architectures for real-time image processing ICs," in *Proceedings of ISCAS*, Dec. 1987.
- [7] R. Jain, F. Catthoor, J. Vanhoof, B. D. Loore, G. Goossens, N. Goncalvez, L. Claesen, J. V. Ginderdeuren, J. Vandewalle, and H. D. Man, "Custom design of a VLSI PCM-FDM transmultiplexer from system specification to circuit layout using a computer-aided design system," *IEEE Journal of Solid-State Circuits*, vol. 21, pp. 73-85, Feb. 1986.
- [8] M. Barbacci, "Instruction set specification (ISPS): the notation and its applications," *IEEE Trans. Computers*, vol. 30, Jan. 1981.
- [9] S. K. Azim, *Application of Silicon Compilation Techniques to a Robot Controller Design*. PhD thesis, University of California at Berkeley, May 1988.
- [10] S. K. Azim, C. Shung, and R. W. Brodersen, "Automatic generation of a custom digital signal processor for an adaptive robot arm controller," in *ICASSP*, IEEE, Apr. 1988.
- [11] P. N. Hilfinger, "A high-level language and silicon compiler for digital signal processing," in *Proceedings of the Custom Integrated Circuits Conference*, May 1985.
- [12] E. Wang, *A Compiler for Silage*. Master's thesis, University of California at Berkeley, Dec. 1988. Expected.
- [13] P. N. Hilfinger, "Silage reference manual,". Internal Report.
- [14] K. Rimey and P. N. Hilfinger, "A compiler for application-specific signal processors," in *IEEE Workshop on VLSI Signal Processing*, Nov. 1988. To appear.
- [15] K. Rimey and P. N. Hilfinger, "Lazy data routing and greedy scheduling for application-specific signal processors," in *The 21th Annual Workshop on Microprogramming*, Nov. 1988. Submitted for publication.
- [16] J. A. Fisher, D. Landskov, and B. D. Shriver, "Microcode compaction: looking backward and looking forward," in *Proceedings, National Computer Conference*, pp. 95-102, AFIPS, 1981.
- [17] 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., University of California at Berkeley, Electronics Research Lab, Nov. 1987.
- [18] M. B. Srivastava, *Automatic Generation of CMOS Data Paths in the LAGER Framework*. Master's thesis, University of California at Berkeley, May 1987.
- [19] R. Alverson, T. Blank, K. Choi, A. Salz, L. Soule, and T. Rookicki, "THOR user's manual," Tech. Rep. CSL-TR-88-348 and 349, Stanford University, Jan. 1988.