# AN INTEGRATED CAD SYSTEM FOR ALGORITHM-SPECIFIC IC DESIGN

by

Chuen-Shen Shung

Memorandum No. UCB/ERL M88/44

14 June 1988

COVER PAGE

# AN INTEGRATED CAD SYSTEM FOR ALGORITHM-SPECIFIC IC DESIGN

by

Chuen-Shen Shung

Memorandum No. UCB/ERL M88/44

14 June 1988

**ELECTRONICS RESEARCH LABORATORY**

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

# AN INTEGRATED CAD SYSTEM FOR ALGORITHM-SPECIFIC IC DESIGN

by

Chuen-Shen Shung

Memorandum No. UCB/ERL M88/44

14 June 1988

**ELECTRONICS RESEARCH LABORATORY**

College of Engineering
University of California, Berkeley
94720

# An Integrated CAD System for Algorithm-Specific IC Design

Ph.D. Chuen-Shen Shung EECS Department

## ABSTRACT

LagerIII, an integrated CAD system for algorithm-specific IC design is described. In particular, applications such as speech processing, image processing, telecommunication, and robot control are targeted. Designing such circuits usually requires the collaboration of algorithm developers, architecture designers, and circuit designers. LagerIII provides the user interface at behavioral, structural, and physical levels to facilitate this collaboration. It also provides an interface for integrating new CAD tools.

Because direct synthesis from a behavioral description has yet to produce efficient results in a wide range of applications, our approach requires the user to specify a behavior and a parameterizable datapath. The silicon compilation subsystem translates the behavioral description into datapath instructions and parameter values, which, together with the datapath specification, make up the structural description. The silicon assembly subsystem in turn translates the structural description into a physical layout. With the aid of simulation tools, the user can fine-tune the datapath by iterating this process.

The silicon compiler provides two kinds of behavioral languages: an applicative language called Silage, and an "extended subset" of C called RL. Two external programs, the Silage translator that translates Silage into RL and the RL compiler that translates the program into datapath instructions, have been linked into the system to allow silicon compilation.

The silicon assembler can be used independently of the silicon compiler for high-sample-rate applications such as image processing. It consists of a structural interface, a database manager, module generation tools and a functional simulator. The structural interface processes the parameterized structural description and enters it into the database, for use by the simulator and by the module generation tools that generate the layout. The database is based on *Flavors*, an object-oriented programming system, to facilitate the integration of new CAD tools. The silicon assembler uses an open cell library consisting of parameterizable modules and leaf-cell layouts and functional models.

LagerIII has been applied to a number of algorithm-specific IC designs. Two examples, a frame buffer controller chip and a pitch tracker chip, are described in this thesis.

Robert W. Brodersen

Committee Chairman

# TABLE OF CONTENTS

## ACKNOWLEDGEMENTS

I would like to thank my research adviser, Prof. Robert W. Brodersen, for his encouragement, guidance and support throughout the course of this thesis work.

The discussion with Dr. R. Jain in the early phase of the LAGER project was very fruitful. K. Rimey, E. Wang, M. Srivastava, E. Lettang, Dr. R. Jain, Prof. J. Rabaey and Prof. R. Brodersen contributed a great deal to the software development of the LAGER system. K. Azim used the LAGER system in its early stage of development and provided valuable feedback.

I also want to thank P. Schrupp, B. Richards, C. Lee and W. Baringer for sharing with me their knowledge on image frame buffers and hardware design.

The research project is funded by DARPA under grant N00039-87-C-0182. The Ministry of Education of the Republic of China supported me for my first two years of graduate study.

I am indebted to C. Lee who patiently read most of the thesis manuscript and corrected my numerous writing errors.

Finally, I want to thank my wife, Debbie, and my family for their constant care, support and love.

# CHAPTER 1

# Algorithm-specific ICs

## 1.1. Commodity, Application-specific and Algorithm-specific ICs

Most of the integrated circuits (ICs) designed today can be categorized 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. The commodity chips are usually produced in large volume with fully custom design. Because they are standard products, *price* is the only figure of merits. The companies that produce commodity chips have to constantly improve the price-performance of their commodity IC products, usually by more advanced circuit design, and manufacturing processing technology, in order to be competitive.

The ASICs, as the name suggests, are small-volume products and hence often cannot afford the fully custom design approach. 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 (cost), which also brings about the need for computer-aided design (CAD) tools. The ASICs achieve higher performance through architecture innovations. Due to their application-specific nature, each design can exploit the special conditions in the particular application to create an efficient architecture.

In the past several years we have seen a steady growth in the ASIC market, and it is predicted that by 1990 the commodity ICs and ASICs will each occupy 50% of the IC market. It is due to the following reasons that many believe the trend of the increasing importance of the ASICs will remain after 1990. First, in commodity ICs while scale down the technology has been fruitful over the past 10 years, significant investment is required. Second, many new systems and new architectures have been proposed which were implemented with ASICs as the key components. Even though the cost of an ASIC is higher than that of a commodity IC with similar complexity and technology, the overall system had lower overall cost due to a more efficient system architecture.

Another reason of the increasing ASIC designs is the maturity of CAD tools. It is not until recently that high-quality module generators, silicon compilers and integrated CAD environments were available. The main products of the CAD industry have changed from layout editor, switch or circuit simulator (which were used by commodity IC designers) to higher level front-end tools like the schematic entry tools and functional simulators, and automatic layout generators. The latter played an important role in the design process of ASICs. In the future we expect to see additional CAD tools developed for ASIC design.

Finally, there is the issue of *fast prototyping* of design ideas. The system designer often wants to quickly evaluate the price/performance of hardware implementation of an application. It is often the case that the designer has the choice of a board with commodity chips or an ASIC to implement his idea. What has made the ASIC a more attractive choice is because there are much more tools available at the chip level than at the board level. The advantage of ASIC design is that there are a number of cell design styles (gate-array, standard-cell, semi-custom, etc.) with different turn-around times and performances. Once the design idea is verified, the performance can be improved with a different cell design style.

One main disadvantage of ASICs is the extra fabrication time as compared to off-the-shelf commodity ICs. However, the fabrication time has been reported to be as short as two weeks with some gate array design systems. Moreover, from our experience the design time is still much longer than the fabrication time. Therefore, to reduce the design time has a larger impact on the overall turn-around time.

The focus of this thesis research is on a subset of ASICs in which some computational algorithms are implemented. I will call them the *algorithm-specific* ICs. From the experiences of several algorithm-specific IC designs[1,2,3] , we found that even though the algorithms in various applications are very different, they however can all be implemented with a limited set of hardware modules (e.g. ram, data path). By reusing hardware modules, many of the design difficulties of most ASICs are alleviated. Specifically, this can be done by (1) selecting hardware modules to be used in a particular algorithm and (2) creating new hardware modules if necessary in such a way that they are reuseable.

On the other hand, by restricting the design domain to algorithm-specific ICs, the CAD system design becomes simpler. First, it is possible to develop some special-purpose module generators for commonly-used hardware modules. Second, one of the tasks of the CAD system is to translate higher level design representations to layout, which is almost untractable unless we restrict the design domain.

Typical application areas of algorithm-specific ICs include speech processing, image processing, robot control, computer vision, digital audio and telecommunications. For example, inverse kinematics, adaptive equalizers for digital cellular telephones, fingerprint recognition, frame buffer controller, channel emulator for computer network protocols, image processing based on projection-transformed data, milling machine overseer and robot controller are a number of active research projects which involve algorithm-specific IC designs, which are underway in our research group.

## 1.2. Dedicated and Programmable Architectures

To achieve higher performance in algorithm-specific ICs, efficient architectures have larger impact than efficient layout just as in the case with ASICs. In general, the most efficient architecture is one that is *dedicated* to the particular algorithm. A dedicated architecture is often obtained by studying the data flow operations in the algorithm and allocating a dedicated hardware module for each operation, for example, an adder for an *add* operation. This way the logical data flow in the algorithm is implemented by the physical interconnection of the hardware modules. The control unit of the hardware modules is simple since in general there is no time-multiplexing of hardware modules, even though the data path may be very complicated. The speed of the implementation depends on the slowest hardware module in the design.

The drawback of dedicated architectures is that they have to be redesigned for new algorithms. Therefore, another type of architectures is often used in low sampling rate applications which consists of a number of most commonly used hardware modules, which are time-multiplexed (by microcode control) according to the algorithm. We call this type of architectures the *programmable* architectures. The control unit in a programmable architecture is complicated in order to control the time-multiplexing of hardware modules. The speed of the implementation depends on the total number of microcode cycles to realize the algorithm. A programmable architecture is applicable only if

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

For example, if the circuit runs at 5 MHz, and the sample frequency is 5 kHz, then the maximum number of cycles of an algorithm is 1000.

It is interesting to compare the differences and similarities of an algorithm-specific IC in programmable architecture, and an off-the-shelf microprocessor. Both of them offer some programmability. However, a microprocessor is *software programmable* in the sense that software programs can be compiled and stored in its program memory for execution. On the other hand, an algorithm-specific IC in programmable architecture is *hardware programmable*. Not only can different algorithms be compiled and stored, but the architecture itself can be tailored for the algorithms. Examples of typical parameters of a programmable architecture are the word length of the data path, and the depth of the stack, etc. Contrast to software programmable chips that are programmed after the chips are fabricated, the hardware programmable chips are programmed before the chips are fabricated, i.e. at the design phase of the chip. However, the fact that the programmable architecture is subject to customization of the particular algorithm makes the compiler design more challenging.

In summary, for low sampling rate applications, the programmable architecture is a good choice because not only the hardware modules but the architecture are reused to reduce design difficulties. On the other hand, dedicated architectures are necessary for high sampling rate applications because only dedicated hardware modules can provide the speed required. In the next chapter, a description of the CAD system which supports both types of architectures will be given.

# CHAPTER 2

# The LagerIII CAD System

## 2.1. The Challenge

The design representation of algorithm-specific ICs can be divided into three categories (by abstraction level): *algorithm, architecture* and *layout.* At the algorithm level, the design can be represented by an application program, or graphically by a signal flow graph. Because this representation describes only the *behavior* that the chip should implement without specifying what hardware modules to use, it is also called a *behavioral* level representation. At the architecture level, the design can be represented by interconnections of hardware modules. It can also be called a *structural* level representation. At the layout level, the design is represented by physical IC mask layout. It is the most detailed design representation a designer has to deal with. The hardware modules specified at the structure level are further refined such that their physical implementation with leaf cells is specified. The layout level representation can also be called the *physical* level.

A good CAD system should allow the user to specify the designs of algorithm-specific ICs at high levels (behavioral and structural) and generate the physical layout automatically. The productivity of the designer increases when s/he can use high level representations to describe the designs, an experience we learned from high-level programming languages.

The design of algorithm-specific ICs requires collaboration of people with expertise in various areas. Signal processing algorithm developers, VLSI architects and circuit designers is one typical combination. A CAD system should be easy to use for people of all levels of expertise such that (1) they can try out ideas easily and (2) they can work together.

The CAD system should provide a means through which the architectures designed by the architects can be used by the algorithm developers. An algorithm developer can simply design and simulate the algorithm in a high-level language, and *choose* an architecture that is designed to realize the algorithm. The architecture has to be *parametrizable* such that it can be tailored according to the algorithm. For example, the round-off error characteristics of an algorithm may dictate the word length of the data path. The CAD system should also provide a means through which the hardware cells designed by the circuit designers can be used by the architects. All the architect needs to do is to determine how and what cells to use in the architecture. Basic cells (e.g. adder) and functional blocks (e.g. control unit) can be described in a parametrized form to encourage re-use and sharing of the leaf cells and functional blocks, and to facilitate the design of new architectures.

A set of layout generation and simulation tools is needed for quickly evaluating the area and performance of the architecture. In order to reduce the effort of integrating new design tools in the future, the CAD system must have a *policy* to deal with how the tools are integrated.

As the design gets increasingly complicated, design representation and documentation become more important. Because the CAD system can carry out the design from either the algorithm or architecture level, the input descriptions the user specified at those levels are the best design documentations. The design can be reproduced by giving the same high-level description and the choices of architectures and circuits.

In summary, the CAD system should be designed based on four considerations: (1) By providing high-level interfaces such that it can be used by users of different expertise. (2) It facilitates the sharing and re-using of leaf cells and functional blocks through parameterization. (3) New design tools can be integrated easily. (4) It should help fast prototyping the user's designs through automatic layout generation. Therefore, in the CAD system described in this thesis four user interfaces are provided: (1) a *behavioral* interface for entering algorithms, (2) a *structure* interface for entering architectures, (3) a *cell* interface for entering leaf cells and (4) a *tool* interface for entering new design tools.

Whether or not to automate the leaf cell design is also an issue to the CAD system design. The automatic approach is less vulnerable to the design-rule changes, and can produce shorter circuit delays if timing optimization is performed. It is however less efficient in terms of area and run-time. The automatic approach often introduces constraints to the leaf cells and the design tools. In our CAD system, the leaf cells are manually designed.

## 2.2. The Philosophy

In this section I will discuss the approaches and their rationales that are taken in LagerIII. I will define the *silicon compilation* process as the process of translating behavioral level information to structural level information, and *silicon assembly* process as the process of translating structural level information to physical layout.

The *data path synthesis* approach[4] is often adopted in the silicon compilation process. It tries to mimic while automate the human designer's design methodology. It investigates the algorithm and finds out the minimal data path necessary to implement the algorithm. Then it proceeds to schedule the algorithm onto the data path and thereby produces the microcode. Some constraints on the number of hardware modules of each type may be imposed in some systems, which make the scheduling a bit more difficult.

However, the data path synthesis approach is not yet able to produce very efficient (in terms of area and performance) data path architectures for a wide range of applications. The main reason is that there are simply too many possibilities in architecture design to be captured in one software program. In most systems, some high-level decisions such as the bus structure, pipelined or parallel, lumped ALU or distributed functional modules, are pre-determined, which reduce the architecture design domain, and make the problem tractable. Researchers have used rule-based implementations[5] for the data path synthesizer but it was found that a set of rules was only applicable to a limited number of applications.

In some high performance applications special i/o considerations have significant impact on the overall performance of the architecture[6] ,which are not addressed in most behavioral level descriptions.

The data path synthesis approach suffers from the fact that it can only generate data paths that are similar to what have been designed. It is not yet able to *invent* new data paths for exotic applications.

In LagerIII, we use a *data path mapping* approach where the data path is manually designed first and the silicon compilation system *maps* the behavioral level description onto the pre-defined data path. This approach results in more efficient data paths. This approach is made feasible by providing a dedicated data path module generator that allows easy generation and modification of data paths. For algorithms that have relatively similar sampling rate and contain the same set of primitive functions (and therefore can exploit the same data path implementation), the data path mapping approach eliminates the data path synthesis step. On the other hand, for algorithms that need very different data path designs, this data path mapping approach allows human designers to invent new data paths.

The implementation of data path mapping calls for a compiler that *compiles* the high-level behavioral description into microcode. The challenge of this compiler is that it has to be able to generate corresponding microcode for each manually designed data path. This is done through the input of a *code generation table* which essentially describes how each primitive functions in the behavioral description is implemented by a sequence of microcode of each data path. Each data path has one corresponding code generation table. The code generation table in this approach is effectively the same as the rules in a rule-based data path synthesizer; both describe the data path in an abstract way and allow the behavioral description to translate into structural level information.

The data path mapping approach is applied in the following way. First, each pre-designed data path is picked in turn, and the corresponding microcode will be generated by the compiler with the code generation table of the chosen data path. If none is satisfactory, new data path has to be designed and its code generation table written, and send to the silicon compilation system to generate the microcode. Usually, it is easy to find out how to modify an old data path to increase the performance for a particular algorithm.

Even though the above discussion concentrates on the data path design, the same argument applies to the control unit design as well. In LagerIII, instead of trying to synthesize the control unit structure, we map the control flow operation (e.g. branch, if-then-else) in the algorithm into pre-defined control unit structures. For example, the contents of FSM and ROM in the control unit design[7] can be customized with the behavioral description in the silicon compilation system in LagerIII.

In summary, the silicon compilation system in LagerIII maps the behavioral level descriptions to pre-defined structural designs. This approach leaves room for the user to enter new manually designed data path and control unit architectures. This approach is the most feasible one under today's situation where the direct synthesis approach can not produce efficient architectures. It is a necessary first step toward direct synthesis to get more experiences in architectural design through manually design practice.

In the silicon assembly process, there is another issue: whether to let the CAD system or the user choose the hardware modules and cells in the design. In general, there is more than one cell in the cell library with the same functionality, each differing from another in, say, driving capability and layout area. The CAD system can choose the cells based on timing optimization and/or area optimization criteria. However, the drawback of this approach is that the CAD system needs to be re-compiled whenever a new cell or hardware module is integrated, which increases the maintenance cost of the cell library. New versions of the old cells and hardware modules create the same problem.

In LagerIII, we let the user specify all hardware modules and cells in the whole design hierarchy. The cell specification includes the cell name and the directory path name. Different versions of the same cell are stored in different directories.

## 2.3. System Overview

The relationship between the silicon compilation subsystem and the silicon assembly subsystem is shown in Figure 2.1. To facilitate the reuse of the hardware modules, the structural level description is parametrized. Therefore, the silicon assembly subsystem requires both the structural description and the parameter values in order to generate the layout. Typical parameters include the word length of the data path, the content of the ROM, etc. The silicon compilation subsystem maps the behavioral description of the algorithm onto the pre-defined structural design by generating the appropriate set of parameter values. The architecture designer can use the silicon assembly subsystem to generate layout, and the algorithm developer can use both subsystems to generate layout from an algorithm given the structural design.

Figure 2.2 shows the block diagram of the silicon compilation subsystem. The behavioral level description language used in LagerIII is *Silage*[8] . Silage is an *applicative* language designed for describing the signal flow diagrams easily. A Translator translates the Silage program into a *procedural* language similar to C, in which control flow operations are put in. A *Compiler* compiles the procedural language into the symbolic microcode based on a code generation table that describes a pre-defined data path. The symbolic microcode has very general control flow primitives, which can be implemented in a lot of control units. A *Control Generator* generates the parameter values for a particular control unit. Note that the structural level information, in the form of a code generation table and control unit information, contributes to and affects the silicon compilation subsystem.

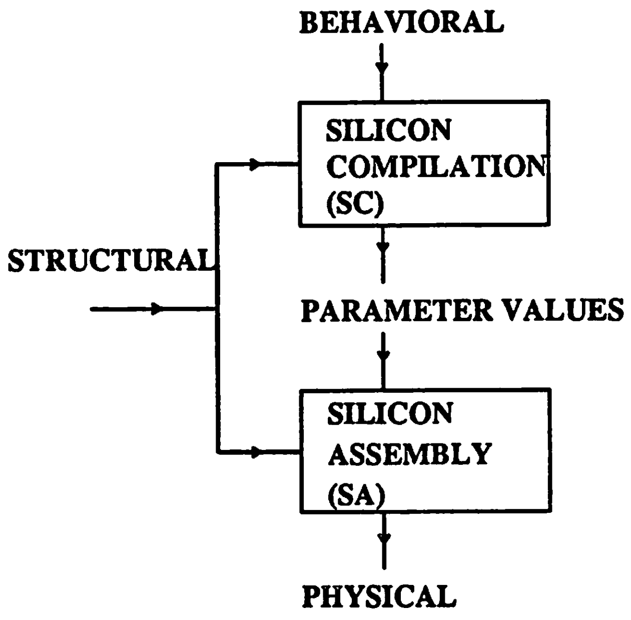

**Figure 2.1 LagerIII system overview**

The silicon compilation subsystem will be described in more details in Chapter 5.

Figure 2.3 shows the block diagram of the silicon assembly subsystem. It consists of three software components (the *Design Manager*, the *Layout Generator* and the *Design Simulator*) and a cell library. The Design Manager translates the parametrized structural description and the parameter values to an internal database. The Layout Generator integrates a set of module generators. The Layout Generator accesses the design information from the internal database, and the leaf cell information from the cell library. The Design Simulator is an event-driven, functional/switch level simulator that allows the user to simulate the structural description. The Design Simulator accesses the design information from the internal database, and leaf cell models from the cell library. It is very important to let the Layout Generator and the Design

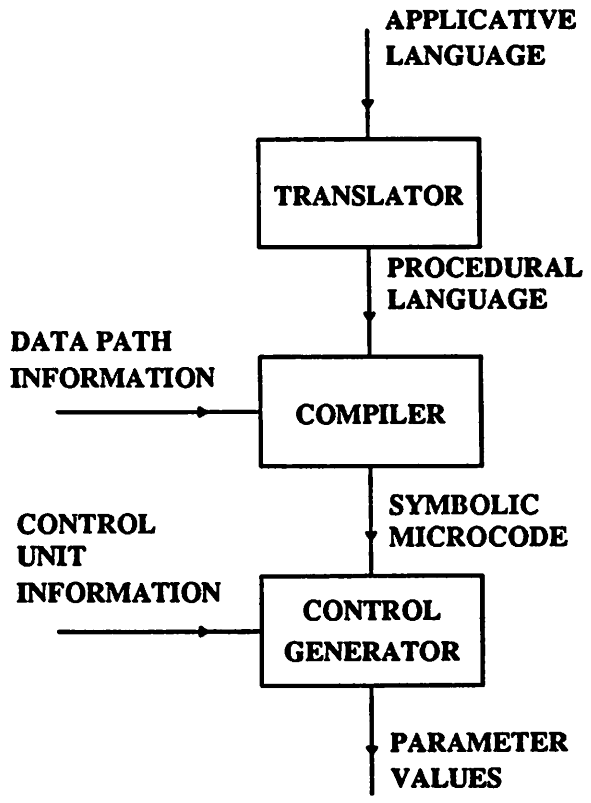

**Figure 2.2 The silicon compilation subsystem overview**

Simulator take the same design information input, which eliminates the need of verifying that the two inputs are the same.

The silicon assembly subsystem will be described in more details in Chapter 3.

In previous chapter, it is said that there are two types of architectures, dedicated and programmable, for implementing algorithm-specific ICs. Both types of architectures are supported in LagerIII, but in different ways. The silicon compilation subsystem is designed to be used for programmable architecture and the silicon assembly subsystem is designed to be used for both programmable and dedicated architectures. This is because (1) The silicon compilation subsystem maps the algorithms onto reuseable pre-defined architectures, but the dedicated architectures is very hard to reuse. (2) Once the dedicated architecture is described by the structural description, the parameter values are often easy to obtain. (3) The programmable architecture, because of its nature of simple data path and complicated control, needs the silicon

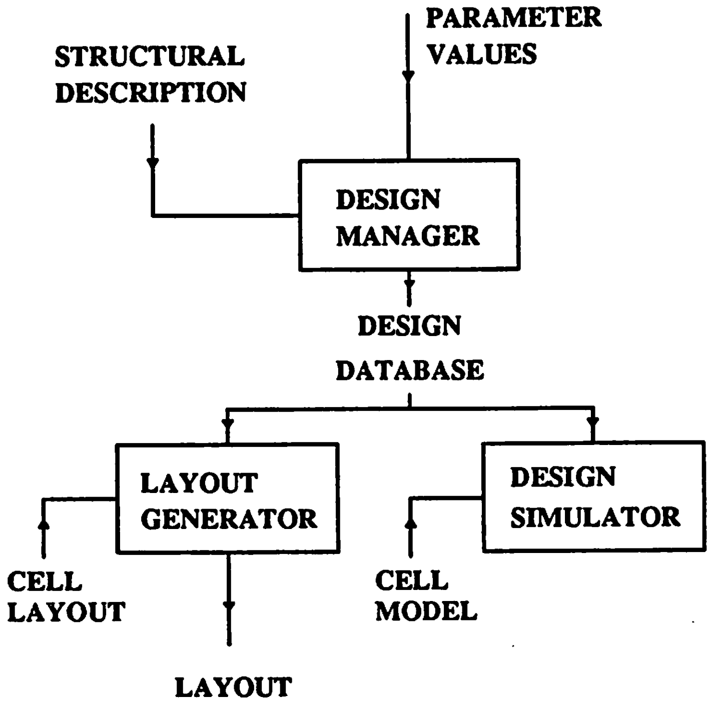

Figure 2.3 The silicon assembly subsystem overview

compilation subsystem to help generate the control unit. The dedicated architecture has simple control and hence the silicon compilation subsystem is not critical.

## 2.4. Related Work

The discussion in this section is organized by a taxonomy of *silicon compilation languages* (SCL), which are the input languages to silicon compilation systems.[9] Silicon compilation languages can be divided into two major categories: *structural* and *functional*. A structural SCL specifies how a circuit is constructed. A functional SCL specifies the input/output mapping of a circuit. A functional SCL can be subdivided into two groups: *architectural* and *behavioral*. A functional SCL is called architectural if it has predictable structural semantics; otherwise it is a behavioral SCL.

There are quite a few general purpose silicon assembly systems developed recently 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 best known examples. The users of these systems usually specify the designs using a schematic entry, which can be viewed as a graphical structural description. Without the capability of taking behavioral level description as input, these systems cannot help the users in translating an algorithm to a structural design.

The CAD tools in these systems are often developed independently. Therefore, as the number of CAD tools increases, the tool integration becomes a problem. This issue has recently attracted much attention and research and development efforts in the CAD industry. Moreover, the fact that the cell library is not parameterizable makes it difficult to maintain the cell library and to share cell modules among designs.

A few special purpose silicon assembly systems have been reported that work on *architectural* level descriptions. These systems usually have restricted target architectures and application domains. The GE silicon compiler[10] , for example, is based on a *bit-serial* architecture. The MacPitts silicon compiler[11] is based on an architecture that contains a bit-sliced data path and a PLA control unit. Furthermore, the input languages to these systems do not support control flow operations very well. The MacPitts uses an embedded Lisp language as input, and the only control flow construct is the Lisp **cond** function.

The silicon compilation started with a behavioral level description has been a hot research area for a number of years. A number of systems were proposed, each with different behavioral description language. The behavioral specifications are divided into three groups in the following discussion:

(1) *frequency-domain specification*. This behavioral specification describes the frequency-domain behavior desired in the chip. Typical parameters include the passband ripple, stopband ripple, stopband attenuation, etc. Based on these specification, a filter is synthesized and its coefficients are optimized for hardware realization. The *Cathedral-I* system was reported[12] in which a fixed bit-serial architecture has to be used to make the architecture mapping and optimization tractable.

(2) *machine specification*. One important example of this kind of behavioral specification is the ISPS[13] 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 the data path that implements the instruction set. Both a iterative algorithmic approach (*EMUCS*) and a knowledge-based approach (*DAA*) have been tried in the CMU-DA system, both neither has yet been able to come close to the human designed data paths. Furthermore, the scope of CMU-DA is limited in generating the architecture of the data path, without actually generating the layout of the data path.

(3) *algorithm specification*. In this kind of the behavioral description, a particular algorithm is specified. It is *lower* than the *frequency-domain specification* because the implementation in terms of algorithm has been fixed. However, a frequency-domain specification usually is only applicable for a limited range of applications (e.g. filters). An algorithm specification is different from a *machine specification* because the architecture can be tailored by the particular algorithm, which is, however, a very difficult task. For example, the LagerI system[14] used an assembly level *design file* as input and the Cathedral-II system[5] used the Silage language. For simplicity, they all assume a fixed architecture onto which the behavioral description will be mapped.

In summary, even though a lot of silicon assembly and compilation systems have been reported, none was able to cover the entire design spectrum from behavioral description to layout. Silicon assembly systems are more mature than silicon compilation systems. However, considerable effort is being devoted to make the silicon assembly system *open* to new CAD tools and cells. Silicon compilation systems have shown promising progress in the past few years, but significant breakthroughs are still required before the most efficient architecture can be generated given an algorithm specification. The LagerIII system attempts to do this by providing a structural input and a behavioral input. Because it is developed to support the design of algorithm-specific ICs, the LagerIII uses an algorithm specification (Silage) as behavioral description. A unique feature that separate LagerIII silicon compilation subsystem from LagerI and Cathedral-II is the structural input, with which the user can modify the target architecture. Finally, the LagerIII silicon assembly system is implemented in an object-oriented environment which makes it easy for new CAD tools and cells to be integrated.

# CHAPTER 3

# The Silicon Assembly Subsystem

The LagerIII silicon assembly subsystem is composed of four parts: an internal design database (section 3.1), a structural interface called the Design Manager (section 3.2), a Layout Generator that integrates a set of module generation tools (section 3.3) and an event-driven, functional level simulator called the Design Simulator (section 3.4).

## 3.1. Internal Design Database

The importance of a consistent internal database in a silicon assembly system will be discussed first in section 3.1.1. Second, in section 3.1.2 the object-oriented programming paradigm will be described, which is shown to be a suitable tool for implementing the database. The organization and actual implementation of the design database using Flavors and Lisp will be described in section 3.1.3. The Lisp interpreter provides a nice interactive environment in which the user can send queries to access the database. In section 3.1.4, a *sorter* example is used to show the internal organization of the database by using the Lisp interpreter. Finally, in section 3.1.5 some comparisons are made between the design database with the OCT database.

### 3.1.1. Integration Policy

The *integration policy* defines how the design information is stored in the internal design database. In traditional CAD program implementations, each program has its private data structures and the communication is through reading and writing files. The integration policy is the *common data structure* that is shared by all CAD programs, and hence eliminates the file transfer overhead in program communications. A well-thought integration policy also makes the integration of new CAD programs easier because the interface problem is confined between the new CAD program and the internal design database.

Each CAD program has its own optimal data structures. However, since the integration policy is shared by all CAD programs, it is difficult to satisfy them all. If the common data structure were to be constructed to be the union of all private optimal data structures, then the overall database will be inefficient due to its enormous size. The integration policy has to compromise and be optimized for the overall performance. Therefore, it often depends on the CAD programs involved.

After the integration policy is defined, the CAD programs can be written as a sequence of *queries* to the design database to access, process and finally store the information. The productivity of the CAD tool designer increases when they can implement the CAD programs with high-level queries. The implementation detail of the database is encapsulated and can be modified without affecting the CAD programs.

In LagerIII, an *object-oriented* programming system (*Flavors*[15,16]) is used to implement the design database. The object-oriented programming system provides a way of implementing highly modular systems and *generic operations*, which allows the high-level queries to be easily implemented.

### 3.1.2. Object-oriented System

A *flavor* is the fundamental entity in the Flavors system, which designates a class of objects that have common characteristics. A flavor consists of *local state* and a set of operations (called *methods*) that can be performed on it. An *instance* is created by *instantiating* a flavor. The elements of local state are called *instance variables*. The values of the instance variables are different from instance to instance, though their number and names are the same for all instance of a flavor.

An instance is asked to perform an operation by specifying the generic name of the operation and by specifying arguments to that operation (a value may be returned). This is also called *sending a message*. Associated with each instance is a means by which a piece of code (*method*) can be found from the name of an operation. When a message is sent to an instance, the instance finds the appropriate method and runs it, giving it the supplied arguments.

This paradigm permits the implementation of *generic operations*. A set of messages is defined, which specifies what the external behavior must be if an instance is to implement the message. The message does not define how the operation is implemented in the instance. This feature allows the implementation detail to be hidden in the instance. A good analogy of the messages is the computer network protocols, where the hardware, software and firmware implementations of the protocols is hidden from other computers in the network.

The terminologies used in Flavors are slightly different from other object-oriented systems. For example, in Smalltalk[17], a flavor is called a *class* and an instance is called an *object*. Flavors is supported in a number of Lisp dialects. In the following, only the Franz Lisp Flavors is described. Several functions are included here for completeness and ease of discussion, and a detailed description is available[15].

A flavor is defined by the special form

```
(defflavor flavor-name
    ( (var1 [init-var1] ) ( var2 [init-var2] ) ... )
    (flav1 flav2 ...)
    ( option1 option2 ...) )
```

where *flavor-name* is a symbol which is the name of the flavor. *var1*, *var2*, ... are the names of instance variables and *init-var1*, *init-var2*, ... are their initial values. An initialization is not required but is useful for assigning default values to the instance variables. *option1*, *option2*, ... are options to the defflavor form. *flav1*, *flav2*, ... are the names of the *component flavors* which the *flavor-name* flavor *inherits*. The instance variables and methods of the component flavors are inherited by the *flavor-name* flavor. The inheritance mechanism is one of the major differences between Flavors and other object-oriented programming systems. In Smalltalk, for instance, a class can only inherit from one parent class (called *superclass*). Therefore, the only possible ways to modify an existing class are by adding additional instance variables, by adding additional methods, or by *shadowing* (replacing) existing methods. Because of the strict hierarchy among classes, this scheme cannot handle *orthogonal attributes* in a modular form. When there are several features that need to be combined in a pick-and-choose fashion, the single-superclass scheme becomes hard to use.

In Flavors, any flavor can inherit more than one component flavors, whose *order* is important in determining which method to inherit if there are more than one defined in the component flavors. The following flavor organization conventions are recommended. A *base* flavor is a flavor that defines a whole family of related flavors, all of which have that base flavor as a component. A *mix-in* flavor is a flavor that defines one particular feature of an object. A mix-in flavor cannot be instantiated, because it is not a complete description. A usable flavor can be constructed by choosing the mix-ins for the desired characteristics and combining them, along with the appropriate base flavor.

An instance can be instantiated by the special form

(**make-instance** *'flavor-name* [*init-option value*] . . . )

A method is defined by the special form

(**defmethod** (*flavor-name message-name* ) *argument-list*
   *form1 form2* . . . )

where *flavor-name* is the name of the flavor in which the message is defined. The name of the message is specified by the symbol *message-name*. The message-name has to be a Lisp *keyword* (which starts with a ":"). *argument-list* is a list of auxiliary variables used by the method. *form1*, *form2*, . . . are the method body.

A message is sent to an instance by the special form

(**send** *instance message-name argument-list* )

where *instance* is the receiving instance to which the message *message-name* and the *argument-list* are sent. A message can be handled by the instance only if the appropriate method has been defined for the instance by a **defmethod**, otherwise it results in an error.

### 3.1.3. Design Database Implementation

In an IC design system, we find that the most important objects are **terminals**, **nets** and **cells**, which are defined to be the base flavors in the design database. Their *instance variables* are defined in Table 3.1 to 3.3. Some instance variables used by the Design Simulator will be discussed later. A cell **x** is said to be the *sub-cell* of a cell X if X contains x. X is called the *parent-cell* of x. The notions of the parent-cell and the sub-cell are useful in a hierarchical design system, which allows us to focus the discussion on one level.

The *generic-parameter-list* stores a list of parameter names by which the cell is parametrized. The *instance-parameter-list* stores a list of parameter name and value pairs, which is obtained by combining the parameter names with the input parameter values through a parameter passing mechanism (see section 3.2). For example, suppose a cell is parametrized by two parameters, **a** and **b**, then the generic-parameter-list of

| instance variable | brief description |
|---|---|
| generic-name | flavor name |
| instance-name | instance name |
| layout-generator | type of layout generator used |
| sub-cell-list | sub-cells described in sdl |
| instance-sub-cell-list | list of instantiated sub-cells |
| generic-parameter-list | list of parameter declarations |
| instance-parameter-list | list of parameter definitions |
| generic-terminal-list | list of formal terminals |
| instance-terminal-list | list of instantiated terminals |
| VGC-terminal-list | list of Vdd, GND and clock terminals |
| net-list | net-list described in sdl |
| instance-net-list | list of instantiated nets |
| generic-equivalent-list | list of lists of equiv terminals |
| generic-feed-thru-list | list of lists of feed-thru terminals |
| xbot | minimum x coordinate |
| ybot | minimum y coordinate |
| xtop | maximum x coordinate |
| ytop | maximum y coordinate |
| cell-number | cell enumeration |
| sim-list | simulation model in a list |
| sim-level | simulation level |
| geometric-constraint-list | list of geometric constraints |

Table 3.1 Instance variables of the cell flavor

a cell instance is (a b) while the instance-parameter-list is ((a 2) (b 5)).

The three most important instance variables of the cell flavor are (1) *instance-sub-cell-list* which contains a list of sub-cell instances, (2) *instance-net-list* which contains a list of net instances and (3) *instance-terminal-list* which contains a list of terminal instances. This information can be used to traverse the whole design hierarchy. The instance variable instance-terminal-list contains only the terminals that are defined in the structural descriptions. The special terminals like **Vdd**, **GND** and clock terminals are not defined in the structural descriptions. Instead, they are created as a result of the layout generation. The special terminals are stored in *VGC-terminal-list*.

| instance variable | brief description |
|---|---|
| side | side which terminal is on |
| name | terminal name |
| index | bit number in a bus |
| coord | terminal coordinate |
| layer | layer which terminal is on |
| net-number | number of net which terminal is on |
| net-name | name of net which terminal is on |
| inward-net-name | name of net in lower level which terminal is on |
| cell-name | name of cell which terminal is in |
| function | function definition for simulation |

Table 3.2 Instance variables of the **terminal** flavor

| instance variable | brief description |
|---|---|
| name | net name |
| number | net number |
| connect-list | list of (cell terminal)'s on the net |
| fanin-list | list of fan-in nets for simulation |
| fanout-list | list of fan-out nets for simulation |
| state | Forced,HZ,Weak for simulation |
| value | 1,0,X value for simulation |
| driven-cell-list | list of driven cells for simulation |
| driving-cell-list | list of driving cells for simulation |

Table 3.3 Instance variables of the **net** flavor

From the terminal instance t, the net instance that t is on can be readily obtained by the *net-name* instance variable of t. From the net instance n, in order to find out whether or not a particular terminal instance is on n, the *connect-list* instance variable of n can be searched. The information can sometimes be retrieved in more than one approach for efficiency. For example, if we want to find out all the terminals in the sub-cells that are connected to a particular terminal t in the parent-cell p, we can either start from the instance-net-list of p to search all terminals of the sub-cells that are in the same *connect-list* that t is in, or we can start with the instance-sub-cell-list of p to get all the terminals in each sub-cell, which have the same *net-number* as t.

The integration policy is essentially how the structural information is organized using the *cell*, *terminal* and *net* flavors. Each CAD tool designer needs only to understand the relationship among them to be able to access the structural information efficiently. In the next section, a small example is given to illustrate the design database.

### 3.1.4. An Example

A typical design may consist of tens or hundreds of instances (of cell, terminal or net flavors). Therefore, it is crucial to get the correct instance efficiently. Due to the hierarchical organization of these instances it is possible to reach every instance in the hierarchy by traversing up and down the hierarchy and by exploiting the integration policy. However, this is not efficient because it involves considerable list searching.

Another approach is to give every instance a unique name. Since a name can be used as a pointer to the instance in Lisp, we can get to any instance by its unique name. The naming convention is to concatenate the instance names of the parent-cell of an instance with the instance name of the cell, net or terminal instance. To increase readability, special characters are inserted between every two instance names. The character "-" is used to concatenate a cell; the character "." is used to concatenate a terminal; the character "@" is used to concatenate a net; the characters "[" and "]" are used to delimit an index number. For examples,

sorter-pr-mux

refers to the *mux* cell of the *pr* cell of the *sorter* cell. *Sorter* is the root cell in the structural hierarchy.

sorter@ctrlnet

refers to the *ctrlnet* net of the root cell.

sorter-pr.in[0]

refers to the *in* terminal (bit 0 in the bus) of the *pr* cell of the root cell. Note that the terminals and the nets can only appear at the end; they cannot be the ancestors of any instances.

Now let us consider the example depicted in Figure 3.1, where the cell names are shown with boldface, net names with Italic and terminal names with Roman fonts. The root of the design hierarchy is

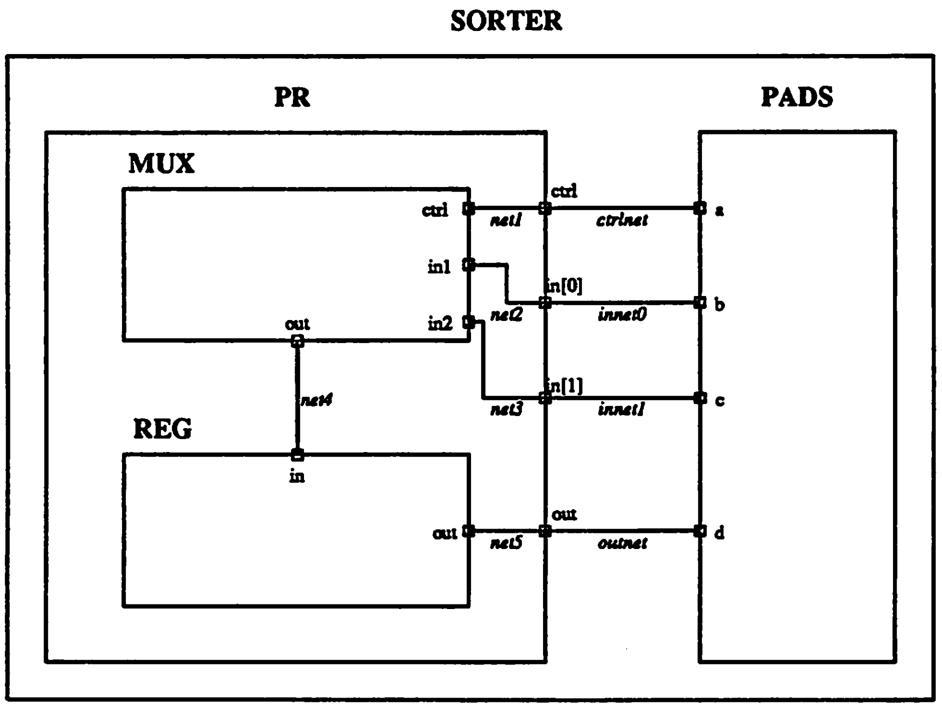

**Figure 3.1 Sorter example**

*sorter*, which contains two sub-cells, *sorter-pr* and *sorter-pads*. *Sorter-pr* has two sub-cells, *sorter-pr-mux* and *sorter-pr-reg*. After the design database is created with the Design Manager by translating the structural description, the following Lisp queries can be used to access the information. The "=>" is the Franz Lisp prompt sign. The queries are shown in bold-face. The returned values (returned by evaluating the queries) are found next to the queries. An instance in Flavors is identified by its flavor name and a unique ID (e.g. #<pr 1152884>).

```
=> (send sorter :instance-sub-cell-list)

(#<pads 1153436> #<pr 1152884>)

=> (send sorter :instance-net-list)

(#<net 1153900> #<net 1154140> #<net 1154192> #<net 1154244>)

=> (send sorter :instance-terminal-list)

nil
```

The values of the instance variables can be accessed externally by sending messages to the instance. In this case, the message name is the same as the instance variable name with the character ":" as the prefix . This effectively shields the instance variables from the external world. If the instance variables are changed later, the CAD programs remain unaffected if there are new methods defined for the messages. For example, if the instance variables *xtop*, *xbot*, *ytop* and *ybot* are changed to *xdim*, *xbot*, *ydim* and *ybot*, then the messages :xtop and :ytop can still be handled by the following methods

```
(defmethod (cell :xtop) () (+ xbot xdim))

(defmethod (cell :ytop) () (+ ybot ydim))
```

and the CAD programs are unaffected. Within the method body, an instance variable can be accessed simply by providing the name. Sometimes the notion of *self* is handy, which allows an instance to refer to itself. For instance, the above methods can also be written as

```
(defmethod (cell :xtop) () (+ (send-self :xbot) (send-self :xdim)))
```

```
(defmethod (cell :ytop) () (+ (send-self :ybot) (send-self :ydim)))
```

Note that the *instance-net-list* of *sorter* contains the nets between *sorter-pr* and *sorter-pads*. The *instance-terminal-list* contains the terminals on the perimeter which, for the case of *sorter*, is null. The *instance-sub-cell-list* shows that there are two instances in the list, and if we *evaluate* the cells *sorter-pr* and *sorter-pads* as

```
=> sorter-pr
#<pr 1152884>
=> sorter-pads
#<pads 1153436>
```

we can verify that they are indeed the sub-cells of *sorter* by comparing the id's. For readability, queries can be designed such that they produce names (as defined by the naming convention) instead of instances. However, creating new queries in the form of *methods* will dictate the recompilation of the database software. An alternative is for the user to create his own Lisp functions on top of the primitive set of queries. For example, the **printlcell** function defined as

```
(defun printlcell (cell)
  (dolist (a (send cell :instance-sub-cell-list))
    (format t "~a~%" (send a :instance-name))))
```

can be used to return the names of the sub-cells giving the parent-cell. For example,

```
=> (printlcell sorter)
sorter-pads
sorter-pr
nil
```

Now suppose we start from the terminal instance *sorter-pr.in[0]* and would like to find out what terminals it connects to in its parent-cell and in it sub-cells. We can get the *external* connection information by the *net-name* with

```
=> (send sorter-pr.in\[0\] :net-name)

sorter@innet0
```

Since "[" and "]" are special characters in Franz Lisp, a back-slash escape character is used in front of them. We can find out the terminal instances on the *sorter@innet0* net by accessing its *connect-list* instance variable as

```
=> (send sorter@innet0 :connect-list)

((#<pads 1153436> #<terminal 1154764>) &)
```

or by using the user-defined **printIconn** function to get the names of the cell instances and terminal instances, as

```
=> (printIconn sorter@innet0)

(sorter-pads b)

(sorter-pr in[0])

nil
```

which shows that the *in[0]* terminal on *sorter-pr* is connected to the *b* terminal on *sorter-pads*. However, if we insert the **(send sorter-pr.in[0] :net-name)** query into the **(send sorter@innet0 :connect-list)** query directly, we will get an error.

```
=> (send (send sorter-pr.in\[0\] :net-name) :connect-list)

Error: funcall: Bad function sorter@innet0

Form: (send (send |sorter-pr.in[0]| :net-name) :connect-list)

c{1}
```

This is because **(send sorter-pr.in\[0\] :net-name)** returns the *name* instead of the flavor instance that is required for the query. The fix is to insert an additional *evaluation* step as

```
=> (send (eval (send sorter-pr.in[0] :net-name)) :connect-list)

((#<pads 1153436> #<terminal 1154764>) &)
```

The function **symbol-value** can also be used in place of **eval**.

The *internal* net that **sorter-pr.in[0]** connects to can be found by the *inward-net-name* instance variable of **sorter-pr.in[0]**. Using the **printIconn** function, it is shown that the **in1** terminal of the **sorter-pr-mux** cell is connected with **sorter-pr.in[0]**.

```
=> (printIconn (symbol-value (send sorter-pr.in[0] :inward-net-name)))

(sorter-pr in[0])

(sorter-pr-mux in1)

nil
```

Another Lisp query that often comes handy is the **describe** function, which takes an instance as argument and returns the values of all the instance variables. For a list of generic Lisp query functions, the reader is referred to the Franz Lisp manual.[15]

All the queries above are used to *access* the information in the design database. The queries to *store* information can be formed by concatenating the string ":set-" to the instance variable names. For example, the following query store *0* to the *xbot* instance variable of *sorter*. Note that this query has an argument.

```
=> (send sorter :xbot)

nil

=> (send sorter :set-xbot 0)

0

=> (send sorter :xbot)

0
```

This example shows that design information can be accessed and stored by issuing Lisp queries in the form of message passing. This mechanism allows easy access to the design database while keeping the implementation detail of the design database independent of the CAD programs.

### 3.1.5. Comparison with the OCT Database

The design database was developed in parallel with the OCT database project[18] . In this section, some comparisons between them are presented.

First, the instantiation scheme of the two systems are different. In LagerIII, an instance is instantiated with all of its parameter expressions evaluated. Instances of the same flavor description can be different due to different parametrizations. In OCT, the *masters* (the counterpart of flavors) are not parametrized. All instances of the same master are the same except the id's. Therefore, a 3x3 PLA is a different master from a 4x4 PLA in OCT, while in LagerIII they are two instances of the same PLA flavor with different parameter values. It is a trade-off between run-time and storage space. The LagerIII scheme consumes more run-time for evaluating the parameter expressions during instantiation; however, it allows more economic cell library and design database.

The integration policy in LagerIII corresponds to the *symbolic* policy in the OCT database. In LagerIII, the physical layout is stored in Magic[19] format and hence no *physical* policy is developed. The structural description language can be thought of as the textual form of a schematic policy. For the symbolic policy, there are noticeable differences between LagerIII and OCT. For example, In LagerIII a net name can be readily obtained from the terminal instance with the *:net-name* message. In OCT, a *generator* has to be used to search the net name given a terminal instance. In OCT, the *actual* terminals (terminals on the sub-cells) need not to be instantiated explicitly, whereas they do in LagerIII because of the possible variations from instance to instance. Nevertheless, a link has been implemented to generate an OCT database with symbolic viewtype from the design database of LagerIII, which makes it possible to exploit the CAD tools attached to OCT.

Other differences are listed below. In LagerIII, the *layout generator* information is stored as an instance variable of the cell flavor in order to perform the layout generation automatically. In OCT, a shell script is usually needed to *manage* the layout generation process. OCT treats Supply and Ground terminals the same way as other signal terminals. Therefore Supply and Ground are also considered as formal terminals, which means that the user has to specify the clustering of the Supply and Ground nets. In LagerIII, the Supply and Ground terminals are defined after the layout generation and hence they are not defined in the structural descriptions. OCT has the advantage that the database is non-volatile. The design database in LagerIII is only in the main memory without having a back-up image in the secondary memory, which makes it impossible to recover if the program crashes. OCT also has a powerful *property* set which provides an easy way to introduce new attributes. OCT uses full path names and LagerIII uses a special path search mechanism to direct the searching of remote files.

## 3.2. The Design Manager

The Design Manager is responsible for translating the structural descriptions to the design database, which was described in section 3.1. In this section, the syntax and the semantics of the structural description language are described first, which is followed by the description of the implementation of the Design Manager.

### 3.2.1. Structural Description Language

A structural description language (*sdl*) file is used to describe the *structure* (cell hierarchy, interconnection, etc.) of a cell. *Sdl* files are inputs to the Design Manager, the structural interface program of LagerIII. The *cell-name.sdl* file specifies the relationship between the cell *cell-name* and its sub-cells, which are described by other *sdl* files. To implement an architecture, a set of *sdl* files is required. The Design Manager provides a path mechanism that allows using user-specified *sdl* files along with library *sdl* files. *Sdl* files can be parameterized to facilitate the re-use of library *sdl* files.

A *sdl* file includes 7 sections, the order of which is not critical. Each section is recognized by a keyword. **Layout-generator, parent-cell, sub-cells, net, geometric-constraint-list, sim-list** and **lisp-function** are the keywords for the 7 sections. The sdl file is written in a lisp-like format; each section is composed of one or more *lists*. A *list* has zero or more elements, each of which can be a *lists* itself, enclosed by a pair of parentheses. The first element of the list in each section is the corresponding keyword of the section.

1. Layout-generator section

The *layout-generator* section is a list with two elements. The first element is the keyword **layout-generator** and the second element is the name of the layout generator. At this moment, six layout generators (**TimLager, dpc, Flint, Padroute, stdcell** and **mosaico**) can be used.

Example:

**(layout-generator TimLager)**

2. Parent-cell section

The *parent-cell* section has one list of 3 elements. The second element is the name of the parent cell, which has to be the same as *cell-name* (the name of the *sdl* file). The third element is a list whose first element is the keyword **parameters**, and the rest of the list is a number of parameter declarations that parameterize the parent cell.

Example:

**(parent-cell rom (parameters row column))**

3. Sub-cells section

The *sub-cells* section is one list of the keyword **sub-cells**, followed by *n* sub-cell definitions where *n* is the number of sub-cells in the parent cell. Each sub-cell definition is a list of 4 elements: its *generic* name, its *instance* name, a list of parameter definitions and an optional *flag-expression*. The *generic* name specifies the name of the *sdl* file where the sub-cell is defined. The *instance* name is used to refer to the sub-cell in the *sdl* file. If more than one instance of the same generic sub-cell are used, then their *instance* names can be used to distinguish them.

The first element in the parameter definition list is the keyword *parameters*, with each following element of the form *(parameter expression)*. The result of evaluating the *expression* defines the value of the *parameter*. The *expression* is a Lisp expression constructed by primitive Lisp functions and/or user-defined Lisp functions (see *lisp-function* section), which returns an integer, a literal symbol or an array of strings (in this case the *parameter* is a truth table). The *flag-expression* is used to decide whether to include the specific instance in the sub-cell list of the parent-cell. If *flag-expression* is evaluated into a non-zero value (default), then the instance is included; otherwise it is removed. If the instance is removed then all associated nets and terminals are removed as well.

For example, an instance *dec1* of the generic *rom-decoder* cell being a sub-cell of the parent cell *rom* can be represented by

```
        (sub-cells
            (rom-decoder dec1 (parameters (row row) (column (/ column 2))))
            ... (other sub-cells) )
```

From the example, we see that the *rom-decoder* has two parameters *row* and *column*, (which are defined in *rom-decoder*.sdl). In the instance of *dec1*, the value of row is the same as the value of *row* in the parent cell (*rom*), and the value of column is the value of *column* in the parent cell divided by 2. The combination of parameter declaration (in the parent-cell section) and parameter definition (in the sub-cell section) provide a mechanism for parametrizing the design with very few parameters. Note that since there is no *flag-expression* field in the definition the sub-cell *dec1* is included.

4. Net section

The *net* section consists of one or more lists, each of which contains 3 or 4 elements. The first element in the list is the keyword net. The second element is the net name. The third element is an optional *bus-expression* which, if present, evaluates into the *width* of the net (or, the *number* of nets); otherwise *one* net is included. If the *width* is zero, the net is removed. The last element of the net definition is a *connectivity list* that is used to specify the connectivity. Each element in the *connectivity list* is a *terminal definition*. Each *terminal definition* defines a terminal on the net.

A *terminal definition* can be used to refer to one or more terminals, depending on the *width* of the net. The *terminal definition* is a list of 2, 3 or 4 elements. The first element is the *instance* name of the cell where the terminal is on, or the keyword **parent** if the terminal is on the parent cell. The second element is the name of the terminal. The third element is an optional *starting index* and the fourth element is an optional *increment*. The *starting index* and *increment* are useful when the *terminal definition* is to define the connection of a bus. The *increment* element can be present only if the *width* of the net is greater than 1. The *increment* is 1 by default, and the default value of *starting index* is 0 if the *width* of the net is greater than 1. If the *width* of the net is 1, then the terminal is not indexed. Note that the *increment* and the *starting index* can, in general, be Lisp expressions.

For examples,

```
(net net1 ((parent x) (dec1 out) (dec1 in 3)))
```

shows that the *out* terminal and the *in[3]* terminal of the sub-cell *dec1* connect to the *x* terminal of the parent cell on the net *net1*; The *width* of *net1* is 1 since there is no *bus-expression*.

```
(net net2 row ((dec1 in1 (- row 1) -1) (other out)))
```

shows the *in1* bus of the sub-cell *dec1* connects to the *out* bus of the sub-cell *other* in bit-reversed order. The *width* of *net2* is equal to the parameter value of *row*. *Net2* can be viewed as a shorthand of the following set of net definitions:

```
(net net2[0] ((dec1 in1 (- row 1)) (other out 0)))

(net net2[1] ((dec1 in1 (- row 2)) (other out 1)))

...

(net net[row-1] ((dec1 in 0) (other out (- row 1))))
```

A special syntax is created for describing the connection of buses to a single terminal. The third element in this case is the keyword **mergeNet**. The fourth element in the net definition is again the list of *terminal definitions* except that the first *terminal definition* defines only 1 terminal (i.e. no *increment* field) and the rest have a mandatory third and fourth elements for *starting index* and *ending index* respectively.

For example,

```
(net net3 mergeNet ((dec1 cin) (other cin 0 6 2)))
```

connects the *cin* terminal of the sub-cell *dec1* and the *cin[0]*, *cin[2]*, *cin[4]* and *cin[6]* terminals of the sub-cell *other* on *net3*.

The *generic* terminals (the terminals on the boundary) of the parent cell are implicitly defined in the net section through the *parent* keyword. For completeness, *feed-through* terminals and *equivalent* terminals can be defined by the net constructs also. If more than one *generic* terminal with no *instance* terminal (the terminals of the sub-cell instances) appear in the *connectivity list*, the *generic* terminals in the list constitute a feed-through. If more than one *generic* terminal with at least one *instance* terminal appear in the *connectivity list*, the *generic* terminals in the list are said to be *equivalent*.

## 5. Geometric-constraint-list section

Strictly speaking, geometric constraints, which are properties in the layout aspect of the parent cell, should not be considered as part of the *structures* of the parent cell. They are put together in the same sdl file for convenience. The geometric constraint definition is a list whose first element is the keyword **geometric-constraint-list**. Each following element deals with one aspect of geometric constraints. At this moment only the *side* information of the *generic* terminals can be specified. Others like the *aspect ratio* and the *placement* information are under consideration.

The *terminal* aspect of the geometric constraints is specified by a list whose first element is the keyword **terminal**. Each following element is in turn a list of 2 elements: the first being the name of the terminal; the second being a list whose first element is the keyword side, followed by the *side specification* (top, bottom, right or left, or a Lisp expression that returns one of the four side names), and a real number between 0 and 1 which specifies the *location* of the terminals.

For example,

```
(geometric-constraint-list
    (terminal
     (in (side top 0.65))
     (out (side right 0.2))
    ))
```

At this moment there is no construct to specify the side information of a bus except to list them one at a time.

6. Sim-list section

*Sim-list* is used to describe the simulation model of the parent cell, which is used by the Design Simulator. *Sim-list* is a list of at most 5 elements. The first element is the keyword **sim-list**. The second element is a list of the keyword **in-term** and all the input terminals/buses. A bus is specified by a list of *name* and *expression* that evaluates to the *width* of the bus. The third element in the *sim-list* is a list of the keyword **out-term** followed by all the output terminals/buses. The fourth element is a list of the keyword **local** and all the local states. If there are no input terminals/buses or local states, then the corresponding element can be absent. The fifth and last element is a list of the keyword **function** followed by a number of *function definitions*.

Each output terminal/bus and local state should have a *function definition*. Each function definition is a list of the name of the output terminal/bus or the local state, and a Lisp expression which evaluates to the simulation result of the corresponding output or local state. In the Lisp expression, one can use the names of the input terminals/buses and local states and output terminals/buses to refer to the values *carried* by those terminals/buses/local states, and the parameter name to refer to the parameter values.

For example, let us create a *sim-list* of an inverter,

```
(sim-list
        (in-term in)
        (out-term out)
        (function
          (out (Xlognot in))
        ))
```

where *Xlognot* is a built-in function in the Design Manager which implements 1's complement. Note that the Design Simulator works on strings instead of integers.

7. Lisp-function section

Often a Lisp expression is used more than once in a sdl file, in which case a *lisp-function* definition can be used to avoid entering the Lisp expression over and over again. The lisp-function section consists of one or more lists, each of which starts with the keyword **lisp-function** and is followed by a Lisp function definition (e.g. defun).

For example,

```
(lisp-function (defun add2 (n)
          (add n 2)))
```

### 3.2.2. An Example

The set of sdl files for the *sorter* example in previous section is shown in Figure 3.2 to 3.6.

The way the Design Manager is invoked is illustrated in figure 3.7. The user manual of the Design Manager is listed in Appendix A. Note that only the *root* sdl file name is required because the rest can be inferred recursively from the *generic* names of the sub-cells.

```
(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 3.2 sorter.sdl**

```
(layout-generator Flint)

(parent-cell pr (parameter b))

(sub-cells
        (mux mux)
        (reg reg))

(net net1 ((parent ctrl) (mux ctrl)))
(net net2 ((parent in 0) (mux in1)))
(net net3 ((parent in 1) (mux in2)))
(net net4 ((mux out) (reg in)))
(net net5 ((reg out) (parent out)))
```

**Figure 3.3 pr.sdl**

```
(layout-generator leafcell)

(parent-cell pads)

(net a ((parent a)))
(net b ((parent b)))
(net c ((parent c)))
(net d ((parent d)))
```

**Figure 3.4 pads.sdl**

```
(layout-generator leafcell)

(parent-cell mux)

(net ctrl ((parent ctrl)))
(net in1 ((parent in1)))
(net in2 ((parent in2)))
(net out ((parent out)))
```

**Figure 3.5 mux.sdl**

```
(layout-generator leafcell)

(parent-cell reg)

(net in ((parent in)))
(net out ((parent out)))
```

**Figure 3.6 reg.sdl**

```
yosemite 8>> DM_new -I
=> (DM)
Please enter root type (generic name) : sorter
Please enter root name (instance name) : sorter

Please enter parameter file name (if none enter N) : N
Please enter parameter value of a in cell sorter (root) : 4
Parameter values incorporated in file sorter.par.
start creating structures

Now if you want to continue with layout generation
Type (LG) to the coming prompt
Or type (DSIM) to continue with event-driven simulation
nil
=>
```

**Figure 3.7 Invoking the Design Manager**

The -I option specifies the interactive mode. The parameter values can be entered one by one interactively or through a file. In this example, there is only one parameter value to be entered. In *sorter.sdl*, the parameter *b* of the sub-cell *pr* is defined to be (a + 2). Therefore, when the user enters 4 as the value of the parameter *a* of *sorter*, the parameter value of *b* in *sorter-pr* will be 6. In general, the user only needs to enter the parameter values of the root cell, and the parameter values of all parameters in the design hierarchy will be evaluated accordingly. After the Design Manager finishes and new Lisp prompt

(=>) appears, the design database is created. The user can access the design database with the queries discussed in previous section, or proceed to perform layout generation or simulation. A log file with the name *root*-dm.log is created which contains verbose information about the status of the Design Manager operation. All the error and warning messages detected in the sdl files are also reported in the log file.

The sdl files can be either in the working directory or in some remote library directories. A user uses the *.lager* file to specify *where* the Design Manager looks to find the sdl files. The Design Manager first attempts to find the *.lager* file in the working directory. If it fails to find one there, then the home directory is searched for a .lager file. The .lager file can be used by any tools in LagerIII to specify library paths.

The .lager file may consist of any number of lists, each of which takes the form

(*keyword* element [elements ...])

where the *keyword* specifies an attribute of a tool that makes use of the list. Normally each *element* in the list is a directory path (absolute or relative). Note that the order of the elements determines the priority. For example, the Design Manager has only one attribute *dm.sdlfile* in the .lager file. The user can use the .lager file to specify the directory paths that the Design Manager needs to search other than the working directory. Comments in the .lager file can be entered in the same way as in the sdl file. An .lager file example is shown in Figure 3.8.

## 3.2.3. The Implementation of the Design Manager

The Design Manager is basically implemented by one generic operation: the *create-structures* message. The create-structure method is implemented in the *cell* flavor because it is a generic operation common to all cell instances. The main program of the Design Manager can be constructed by the 3-line pseudo code:

```
(open-parse-sdl-file  root)

(parameter-value-binding  root)
```

```
(dm.sdlfile
        ~lager/LagerIII/processor/sdl
        ~lager/LagerIII/lib/stdcell
        ~lager/LagerIII/lib/dpc/leafcells)
(TimLager.o ~lager/LagerIII/lib/TimLager/scpads/scpads1.25)
(TimLager.leafcells ~lager/LagerIII/lib/TimLager/scpads/scpads1.25/leafcells)
(DPC.cd ~lager/LagerIII/lib/dpc/leafcells)
(DPC.mag ~lager/LagerIII/lib/dpc/leafcells)
(bin ~lager/LagerIII/bin)
(octbin ~cad/bin /usr5/octtools/bin)
(stdcell.leafcell ~lager/LagerIII/lib/stdcell)
(Padroute.hdl ~bilbo/moslib/frames)
```

**Figure 3.8 A .lager file example**

```
(send root :create-structures)
```

The *open-parse-sdl-file* function expects one argument, which is a sdl file name without the *.sdl* suffix. In the case of the Design Manager main program, the root sdl file is specified by the user. The sdl file will be searched first in the working directory. If not found, then the library paths specified by the *dm.sdlfile* list in the .lager file will be searched in order. The *open-parse-sdl-file* function creates an instance of the *root* and fills in the instance variables with values obtained from parsing the sdl file. Since the sdl file is of Lisp syntax, the parsing is reduced to (1) reading the lists in the sdl file and (2) recognizing each list by the first symbol in the list (e.g. parent-cell, net, etc.). These lists are stored in the instance variables of the instance of *root* for further processing by the create-structures method.

The *parameter-value-binding* function binds the parameter values defined in the user-defined parameter value file to the parameters of the *root*. The parameter value file contains a number of lists of the form

(*parameter-name* parameter-value)

where the parameter-value can be a number, a symbol-name or a truth table. If any parameter is defined in the parameter value file but not in the parameter list of the *root* sdl file, then a message

**Warning: unused parameter:** ***parameter-name*****.**

is reported to the user. On the other hand, if any parameter is defined in the *root* sdl file but not in the parameter value file, the Design Manager will ask the user to enter its value, by

**Please enter parameter value for** ***parameter-name*****:**

As a result of the *parameter-value-binding* function, the value of the *instance-parameter-list* instance variable is created, which consists of a number of (*parameter-name* parameter-value) pairs. Each *parameter-name* is made to be an instance variable too, which makes it possible to directly access the parameter values. Otherwise, the *instance-parameter-list* instance variable would have to be searched given a *parameter-name* for its parameter value.

The create-structures method is defined as the following:

```
(defmethod (cell :create-structures) ()
        (send-self :instantiate-sub-cells)
        (send-self :instantiate-nets)
        (send-self :instantiate-terminal-lists)
        (dolist (a (send-self :instance-sub-cell-list))
                (send a :create-structures)))
```

The create-structures is a *recursive* method. It first creates all the sub-cell instances, net instances and terminal instances, and then send the create-structure message to its sub-cell instances. It is a top-down implementation since the structures of the parent-cell are always created before those of the sub-cells are. The three methods, which are also defined for the *cell* flavor, *instantiate-sub-cells*, *instance-nets* and *instance-terminal-lists*, serve to create the *instance-sub-cell-list*, *instance-net-list* and *instance-terminal-list* instance variables. In doing so, the three methods make use of the instance variables deposited by the *open-parse-sdl-file* function.

The *instantiate-sub-cells* method works on the *sub-cells* section in the sdl file. It then calls the *open-parse-sdl-file* function to access and parse the sdl files of the sub-cells. It calls a *evaluate-expr* function to evaluate the parameter passing expressions in the sub-cell definitions, giving all the parameter values of the parent-cell. The parameters of the sub-cells are made to be instance variables of the sub-cell instances too.

The *instantiate-nets* method works on the *net* section in the sdl file. Each net declaration can generate zero or more net instances depending upon the value of the parameter in the net. The instantiate-nets method creates a *connect-list* instance variable for each of the net instance, which consists of a number of (*cell-instance terminal-instance*) pairs. The *cell-instance* in the connect-list instance variable can be either the parent cell instance or any of the sub-cell instances. The terminal instances of the sub-cells are instantiated by the instantiate-nets method of the parent-cell except for the *root* cell, where the terminal instances are instantiated by the *root* itself. Sometimes not all the terminals in the sub-cells are used for interconnection, in which case only the used terminals are instantiated.

The instantiate-terminal-lists method generates the *generic-terminal-list* instance variable by collecting all the terminals of the parent cell defined in the *net* section. For verification, the *generic-terminal-list* instance variable is compared with the *instance-terminal-list* instance variable created by the parent cell of the parent cell. The *instance-terminal-list* should be a subset of the *generic-terminal-list*, which indicates some generic terminals are not used for interconnection. On the other hand, if some terminal instances in the *instance-terminal-list* are not defined in the *generic-terminal-list* will cause a warning message. The instantiate-terminal-lists method also creates the *generic-equivalent-list* and the *generic-feed-thru-list* instance variables. Terminals in the generic-equivalent-list are generic terminals that are connected inside the parent cell along with some terminals in the sub-cells. Terminals in the generic-feed-thru-list are generic terminals that are connected inside the parent cell without connecting to any sub-cell terminals.

## 3.3. The Layout Generator

The Layout Generator integrates a number of layout generation tools and automatically generates the layout. In this section, the layout generation tools are described first, followed by the descriptions of tool integration and automatic layout generation. The user manual of the Layout Generator is listed in Appendix A.

### 3.3.1. The Layout Generation Tools

Six layout generation tools are currently available in the LagerIII silicon assembly subsystem: **TimLager, DPC, Wolfe, Flint, Padroute** and **Mosaico.** Brief descriptions of them are included here, and detailed information can be found in[20, 18] .

Ideally, all layout generation tools should access the internal database directly for design information. However, some existing tools were written in C and were designed to use files as input and output. The Layout Generator thus provides an interface routine for each such tool. The interface routine explodes the internal database to generate the corresponding input file, and store the information from the output files generated by the tools into the internal database. There are tools that are attached to the OCT database, and the Layout Generator provides a link to generate the OCT database from the internal design database.

**TimLager** is a tiling tool that puts together cells by abutment. The cells can be leaf cells or TimLager-tiled cells. **TimLager** is normally used to generate RAM, ROM, PLA, The abutment is performed according to a C routine that specifies how the cells are placed. The primitive functions used in the C routine are *addup()* and *addright().* Addup() initiates a new row of tiled cells and addright() adds a new cell to the right of the current row. The advantage of the C-routine approach over the *personality matrix* approach is that a C routine is more expressive than a matrix of symbols. For example, C control constructs such as for-loop can be exploited. Furthermore, the C routine can be designed to read in parameter values to generate different cells with the same C routine.

TimLager requires a *pdl* file which indicates the parameter values of the cell. The format of the pdl file is similar to the parameter file required by the Design Manager. *TimLager* generates a layout file in either Kic or Magic format, and a *hdl* file which describes the dimension of the cell and the terminal locations. The C routines and leaf cells should be stored in the cell library and be accessed using the path mechanism in the .lager file.

DPC (Data Path Compiler) is used to generate bit-sliced data paths. The user specifies the interconnections of *blocks* in the form of a sdl file. The blocks are built a priori out of leaf cells of the same functionality. For example, the adder block consists of a number of interleaved adder.even and adder.odd cells. The actual size of the block depends on the number of bits in the data path as defined by the user as a parameter value.

The DPC cell library consists of 30-40 leaf cells which can be classified into three types: *arithmetic-logic* cells such as adder and xor cells, *storage* cells such as scanlatch and scanreg2Port cells and *buffer-mux* cells. Each cell in general have an even and an odd instances. Some have a msb instance as well. All leaf cells are of the same height and with the same *well* orientation.

Unlike TimLager which abuts leaf cells, DPC performs routing in the data signal direction (within each bit-slice) and abutment in the control signal direction (between bit-slices). The routing strategy is described in[21] . DPC accesses the internal database directly by being a tightly-coupled subroutine of the Layout Generator. DPC produces a Magic layout file.

Wolfe is a standard-cell place and route tool. It is used to generate random logic blocks or state machines. Standard-cells are placed in rows, and Wolfe tries to find the optimum placement of the standard-cells to minimize the routing area between the rows. The user can choose the number of rows and the number of trials per iteration in the placement state, otherwise default values are used. The larger the number of trials per iteration, the denser the layout and the longer the run-time.

The **Wolfe** cell library contains roughly 50 standard-cells, which contains essentially logic gates and storage elements such as flip-flops. The library cells are in both Magic and OCT physical view formats. **Wolfe** takes as input an OCT symbolic view, which is translated from the internal database by the Layout Generator. **Wolfe** generates as output an OCT symbolic view in which the physical placement of the leaf cells and the geometric implementations of nets are embedded. A Magic layout file is created from the OCT physical view.

**TimLager**, **DPC** and **Wolfe** are layout generation tools that generate cell modules from leaf cells directly. **Flint**, **Padroute** and **Mosaico** which will be described below, are used to put together the cell modules and the leaf cells by placement and routing.

**Flint** is a general purpose place and route tool. The user has the option of choosing the cell placement, channel assignment and global routing arrangements, and **Flint** takes case of detail routing automatically. The user interaction is done via a graphical interface implemented on engineering workstations. User-specified cell placement, channel assignment and global routing assignments can be stored for later use. The cell placement is restricted to *slicing* structures.

**Flint** can take two kinds of input specifications: an OCT format and a *hdl* format. The OCT format contains the symbolic view definition of the parent cell and the physical view definitions of the sub-cells. The symbolic view definition is generated from the internal database in a similar way as in the **Wolfe** situation. The physical view definitions of the sub-cells contain only the boundary geometries because **Flint** does not utilize the *protection frame* information to perform any over-the-cell routing. The hdl format essentially contains the pdl file of the parent cell and hdl files of the sub-cells. These files are stored in a file system that has similar structure as the design hierarchy. **Flint** generateds a Magic or a Kic layout file, and a OCT physical view or a hdl file depending on the input format used.

**Padroute** is used to route the chip core to a ring of I/O pads. Pads are divided into four pad groups which are constructed by **TimLager** and one pad group is placed on each side of the chip. **Padroute** adjusts the dimension of the pad ring which is determined by the chip core dimension and the routing area.

It also connects the pad groups to form the pad ring by extending the pad groups and putting in corner pads.

**Padroute** accepts the OCT or the hdl formats as **Flint** does except a few minor differences. The nature of the **Padroute** problem dictates that there be five sub-cells, namely the chip core and the four pad groups. There is one special parameter called *fplan* required for the sub-cells, which is used to specify where to put the sub-cells. The five *fplan* parameter values are: *middle*, *top*, *bottom*, *left* and *right*. In the hdl format, all five hdl files can be in the working directory or be accessed through the path mechanism of the .lager file.

**Mosaico** is another general purpose place and route tool. It only takes the OCT format as input. **Mosaico** tries to obtain the optimum placement automatically by a generic algorithm called *simulated annealing*. The trade-off between **Flint** and **Mosaico** is that **Flint** takes some user's time to expedite the layout generation while **Mosaico** takes much more cpu time but generates the layout automatically. **Flint** encourages cell hierarchy in order to reduce the number of cells the user needs to interact with at any given level. **Mosaico** desires that all cells (including pads) be *flat* so as to fully exploit the simulated annealing algorithm. **Mosaico** requires the user to specify the partitioning of the Supply and Ground nets when there is more than one Supply and Ground pads. **Flint** lets the user specify the Supply and Ground routing at the global routing stage and hence the Supply and Ground nets are not specified in the structural description.

### 3.3.2. Implementation of the Layout Generator

The automatic layout generation can be thought of as a recursive function call. In order to perform the layout generation of the parent cell, the layout of the sub-cells has to be generated first. In the Layout Generator, the automatic layout generation is implemented using a *layout-gen* message. The external protocol of the layout-gen message is pretty clear: it accesses the size and terminal location information of the sub-cells and returns the size and terminal location information of parent cell as a result of the layout generation. However, depending upon the layout generation tool involved with each cell, the *method* of han-

```
(defmethod (cell :dmtopdl) ( )
(prog (port cable cable-temp pin j)

;;; open pdl file

        (setq port (outfile (concat (send-self :instance-name) ".pdl")))

;;; print cell instance name and generic name

        (format port "(module (name ~a) (type ~a) ~%" (send-self
                :instance-name) (send-self :generic-name))

;;; generate parameter values
;;; the parameter value can be a integer, a symbol or a list (for 2-d
;;; array), the list value is printed differently from the other two

        (dolist (a (send-self :instance-parameter-list))
                (cond [(listp (cadr a))                 ; list
                        (patom "(" port) (patom (car a) port) (terpri port)
                        (cond [(listp (caadr a)) (dolist (text (cadr a)) (print-list text port))]
                        [t (print-list (cadr a) port)])
                        (patom ")" port) (terpri port)]
                [t                                      ; integer or symbol
                        (patom "(" port) (patom (car a) port) (patom " " port)
                        (patom (cadr a) port) (patom ")" port) (terpri port)]))
        (format port ") ~%")

;;; generate terminal information
;;; including (1) the name (2) the net number (3) the connection

        (dolist (a (send-self :instance-terminal-list))
                (setq cable-temp (send (symbol-value (send a :net-name)) :connect-list))
                (dolist (pin cable-temp)
                        (cond [(eq self (car pin))]
                        [t (setq cable (cons (send (car pin) :instance-name) cable))]))
                (setq cable (cons 'cable cable))

;;; actually printing

        (format port "(term (name ~a) (net ~d)  ~a)~%"
                ;name
                (substring (send a :name) (+ 2 (string-length (send-self :instance-name))))
                ;net
                (send a :net-number)
                ;cable
                cable))

;;; close pdl file and quit

        (close port)))
```

**Figure 3.9 dmtopdl method**

dling the layout-gen message is different. In the **Flavors** system, this is taken care of by the use of mix-in flavors.

The implementation of the Layout Generator involves defining seven mix-in flavors: *TimLager-mixin*, *dpc-mixin*, *stdcell-mixin*, *Flint-mixin*, *Padroute-mixin*, *mosaico-mixin* and *leafcell-mixin*. Each mix-in flavor has a method for handling the layout-gen message. The layout-gen method for the *leafcell-mixin*, for example, parses the Magic layout file of the leaf cell and returns the size and terminal location information of the leaf cell. Each cell flavor in the design hierarchy inherits the *cell* base flavor (defined in 3.1) and one layout generation mix-in flavor according to the layout generation tool used.

The use of mix-in flavors contributes to the software modularity. In the method of any layout generation mix-in flavor, the layout generation of the sub-cells can be enforced by

```
(dolist (a (send-self :instance-sub-cell-list))
        (send a :layout-gen))
```

without worrying about what kind of layout generators are used for the sub-cells.

New layout generation tools can be integrated very easily by defining new mix-in flavors and new layout-gen methods. The implementation of the existing layout generation mix-in flavors and methods will not be affected. Since the layout generation tools have several common input/output formats such as hdl, pdl and OCT formats, some generic methods such as *dmtopdl*, *dmtohdl*, *hdltodm*, *dmtoPhysical*, etc. have been created. These generic methods are defined in the *cell* base flavor in order to be shared by all cell instances, no matter which layout generation mix-in flavors they inherits. For example, the *dmtopdl* method is shown in Figure 3.9.

The layout-gen method of TimLager can be represented by the following pseudo code:

```
(defmethod (TimLager-mixin :layout-gen) ()
        (send-self :dmtopdl)
*** exec TimLager ***
(send-self :hdltodm))
```

where **TimLager** takes a pdl file as input and generates a hdl file and a layout file. Note that cells generated by **TimLager** normally do not have sub-cells.

The Layout Generator main program is simply

```
(send  lgroot  :layout-gen)
```

which will send layout-gen message to the sub-cells of the *lgroot* cell, which will in turn send layout-gen messages to their sub-cells, and so on. The *lgroot* may or may not be the same as the *root* of the entire design hierarchy, which makes possible to optionally perform layout generation on some part of the design.

A *design flow* is implemented which is similar to the *make* program in Unix. The time stamp of the layout file will be checked to determine whether new layout generation needs to be performed. If any of the three in the following are updated, including (1) the parameter file of the whole design, (2) the sdl file corresponding to cell A or (3) any layout files of the sub-cells of cell A, then the layout of the cell A is re-generated; otherwise the layout generated will be bypassed if it has already been generated. This mechanism is particularly useful in the debugging phase in which modifications of the design need to be done quickly.

## 3.4. The Design Simulator

The Design Simulator is an event-driven functional simulator that is used to simulate the functional correctness of the sdl files. In this section, the functional modeling of the leaf cells and the algorithm and implementation of the simulator are described. The Design Simulator requires an event file as input and produces an output value file, which are described at the end of this section.

### 3.4.1. Simulator Overview

The circuit under simulation is a network consisting of a number of *cells* which are interconnected through *nets*. The task of the simulation is to calculate the *values* of certain nets given the values of some input nets. The value of a net can be 1, 0, X or a vector. Logic 1 stands for high voltage. Logic 0 stands for low voltage. Logic X stands for unknown. To speed up the simulation, the values of a bus (*N* nets) can be stored in a hypothetic net. The value of the hypothetic net is a vector which is the concatenation of the values of the *N* nets. Because the values of some nets in the bus may be X, the value of the hypothetic net can not be represented by an integer. Also, if we assign the value of the hypothetic net to be X when some net values in the bus are X, then we lose the information of the values of other nets in the bus.

The values are stored as Lisp symbols. For example, to store a logic 1 in the net *sorter@innet0* and a logic X in *sorter@innet1*, we do

```
(send sorter@innet0 :set-value '|1|)
(send sorter@innet1 :set-value 'X)
```

Note that the vertical bars are required to turn an integer (or any identifier starting with a digit) to a symbol.

The *state* of a net describes the dynamic status of the net. A net can be in one of three possibles states, namely *forced*, *weak* or *high-imp*. The state of a net is *forced* if it driven by some user input. Therefore, the value of the net cannot be changed unless the user input is removed. The state of a net is *high-imp* if the net is not driven by any cells, and hence the net should retain its previous value. The state of a net is *weak* if the net is driven by some cells, and the value of the net is determined by the cells that drives the net. Note that the state of a net is not a static value because the cells can be turned on or turned off (by clocks, for example), which renders the net to be *weak* or *high-imp*.

The basic framework of the Design Simulator is given below. Each output terminal has a *function* definition which describes its dependency on input terminals of the same cell. The value of a net can be calculated by evaluating the functions of all the output terminals connected to the net. There is no notion of *strength* in the Design Simulator. That is, all the cells are assumed to have the same driving capability. Thus, if the functions of two output terminals of two cells evaluate to different values, then the final value of the net is logic X regardless of the physical dimension of the cells. When the calculated value of the net is different from its old value, an *event* is said to occur which triggers further nets to be simulated.

The *transmission* gate is handled in a special way by the Design Simulator. It is special because of its bidirectional nature which makes its modeling very difficult. In the Design Simulator it is assigned a special instance variable called *IsON* which indicates whether the value of the net connected to the gate terminal of the transmission gate is a logic 1 or a logic 0. If the net has a logic 1 value, then the transmission gate is ON and we consider the two nets connected to the source and drain terminals to be equivalent. The simulation of the two nets is done separately with the effect of the transmission gate removed, and their values are compared. If the values are not the same we assign a logic X to both of them. If the transmission gate if OFF then the two nets are independent, and can hold different values.

### 3.4.2. Functional Model

A functional model of a cell is a description of the cell's behavior in terms of its input and output relationship. In addition to the input and output terminals, it is convenient to introduce the notion of *local* terminals in the functional model to simplify the description. For example, if some output terminal functions have a *common sub-expression*, a local terminal can be defined to have the common sub-expression as its function, and hence simplify the description by replacing the common sub-expression in the functions of the output terminal by the local terminal A local net is defined for each local terminal to store the simulation value. In summary, a functional model of a cell contains an input terminal list, an output terminal list, a local terminal list, and a function description for each output terminal and local terminal. The function description of a terminal is essentially a Lisp expression which describes the relationship between the terminal with other terminals in the cell.

The local terminal can also be used in the functional models of the cells that have internal states. For example, a local terminal can be defined for the internal node S of the *register* cell depicted in Figure 3.10. During $\phi_1$, the value of S will be replaced by the complement of IN. During $\phi_2$, the value of OUT will be replaced by the complement of S. However, the local terminal is not the only way to describe internal states. In memory cells (e.g. RAM, ROM), it will be very inefficient if a local terminals has to be defined for each memory location. since the initialization of the huge number of internal states is a problem.

One unique feature of the functional models used in the Design Simulator is that the model can be parametrized. For example, the memory contents of a memory cell is initialized and stored as a parameter value of the cell by the Design Manager. In addition, the parameter values can be changed as a result of simulation! This is useful for RAM cell whose memory content may be different after the simulation. The parameter values of a cell can be accessed in the functional model of the cell by specifying the parameter name. In the register cell example, the value of S depends on the value of IN. The Design Simulator first searches IN in the terminal list. If it fails to find IN, then the parameter list will be searched. This mechanism is powerful and general. However, the side effect is that the name of the parameter cannot be the same as the terminals if it is used in the model.

The parametrizable functional models are useful when there are *buses* in the cell. As mentioned in section 3.4.1, buses are treated as a hypothetic net whose value is a vector that is the result of concatenating all the values of the nets in the bus. The local net is always a hypothetic net whose bus width is the same as

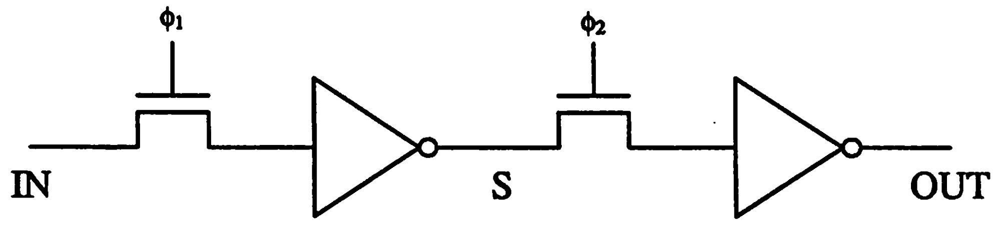

**Figure 3.10 Register cell schematics**

the buses it connects to. For example, if one register instance has a 5-bit IN bus with IN[0]=1, IN[1]=0, IN[2]=X, IN[3]=X and IN[4]=0, then when $\phi_1$ goes high, the value of the local net S will be |1XX10|.

A couple of examples are now presented. In Figure 3.11, the sim-list of the register cell is shown. IN and OUT are defined to be buses of width *width*, which is a parameter of the register cell, by specifying (IN width) and (OUT width) respectively. *Xlogout* is a built-in function of the Design Simulator that is similar to Franz Lisp bit-wise inversion function, *lognot*, except that it can handle X value as well.

Figure 3.12 shows the sim-list of an adder cell, which adds INA, INB and CARRYIN to produce OUT and CARRYOUT. In order to speed up the computation, the *binary2decimal* built-in function is used to convert symbol values of INA and INB to decimal values, and the *decimal2binary* function is used to convert the decimal result to a symbol to be stored in OUT. (*Nallone* N) returns the value $2^N - 1$.

The functional model of a cell is contained in the sdl file of the cell for convenience. First, all the descriptions of a cell are stored in the same file. Second, the user can check for discrepancy between the terminal lists defined in the model (sim-list) and that in the net list.

```
(sim-list function
 (in-term (IN width) φ1 φ2)
 (out-term (OUT width))
 (local S)
 (function
  (OUT
   (if (eq φ2 '|1|) then (Xlognot S)
   elseif (eq φ2 '|0|) then 'HZ
   else 'X))

  (S
   (if (eq φ1 '|1|) then (Xlognot IN)
   elseif (eq φ1 '|0|) then 'HZ
   else 'X))
  ))
```

**Figure 3.11 Functional model of register cell**

```
(sim-list function
 (in-term (INA N) (INB N) CARRYIN)
 (out-term (OUT N) CARRYOUT)
 (function
  (OUT
   (prog (x)

   (cond ((or (eq INA 'X) (eq INB 'X) (eq CIN 'X)) (return 'X)))

   (setq x (add (binary2decimal INA) (binary2decimal INB)
     (binary2decimal CARRYIN)))

   (return (decimal2binary x N))
  ))

  (CARRYOUT
   (prog (x)

   (cond ((or (eq INA 'X) (eq INB 'X) (eq CIN 'X)) (return 'X)))

   (setq x (add (binary2decimal INA) (binary2decimal INB)
     (binary2decimal CARRYIN)))

   (if (> x (Nallone N)) then (return '|1|) else (return '|0|))
  ))
))
```

**Figure 3.12 Functional model for adder cell**

### 3.4.3. Implementation of the Design Simulator

The Design Simulator requires some data structures in addition to those are created by the Design Manager.

A cell is said to be *driving* (*driven* by) a net if the net is connected to an output (input) terminal of the cell. All the driving (driven) cells of a net are stored in an *driving-cell-list* (*driven-cell-list*) instance variable of the net. A net **a** is said to be the *fanin* (*fanout*) net of net **b** if the simulation of **b** (**a**) requires the value of net **a** (**b**). More specifically, net **b** (**a**) is connected to an output terminal whose function definition *depends* on an input terminal of the same cell, which is connected to net **a** (**b**). All the fanin (fanout) nets of a net are stored in an *fanin-list* (*fanout-list*) instance variable of the net.

A *create-sim-structures* message is used to create the additional data structures required by the Design Simulator, by processing the sim-lists in the sdl files. It is a recursive method like the create-structures method, except that the recursion stops when a functional model is found. If the functional models are specified only for the leaf cells, then the create-sim-structures message will be sent to the entire design hierarchy. However, functional models can also be specified for high-level cell modules. to increase the simulation speed. In this case, the create-sim-structures message does not have to be sent to the sub-cells of the cell modules.

After the create-sim-structures message is processed, a *flatten* message is sent to the *root* to flatten the design hierarchy such that the design is represented by the interconnection of leaf cells or cell modules that have functional models defined. Flattening the design hierarchy not only simplifies the implementation but speeds up the simulation, because the net values no longer need to be passed through the intermediate levels in the design hierarchy. However, flattening changes the interconnection information in the design database. In addition, the parameter values of a cell may be modified by simulation as mention above. Therefore, the Layout Generator can not be invoked directly after the simulation. Once the design is simulated and proven to be correct, the user needs to rerun the Design Manager before running the Layout Generator.

For instance, in the sorter example in Figure 3.1, if there are simulation models defined for **mux**, **reg** and **pads** cells, then after the flattening, the root sorter cell will see three sub-cells: **sorter-pr-mux**, **sorter-pr-reg** and **sorter-pads**. The **pr** hierarchy, which is there for layout generation reason, is removed. If a simulation model is created for the **pr** cell then the models of **mux** and **reg** cells are not used and the root sorter cell has **sorter-pads** and **sorter-pr** sub-cells.

The create-sim-structures method for the transmission gate cell is different from other cells. Therefore, two mix-in flavors are created: a *switch-mixin* flavor for the cells that can be modeled by switches, and a *function-mixin* flavor for the cells that can be described by functional models. The transmission gate cell is a *switch* cell and its create-sim-structure method simply determines to which net the gate is connected and to which nets the source and drain are connected to. For the *function* cell, the create-sim-structure method involves the creation of the values of the fanin-list, fanout-list, driven-cell-list and driving-cell-list instance variables for all the nets that are connected to the cell. In addition to the base cell flavor and a layout generation mix-in flavor, every cell instance also inherits either the function-mixin flavor or the switch-mixin flavor. Therefore, the create-sim-structure can be used as a generic message for all cell instances.

The event-driven simulation algorithm is described as follows. In order to exercise the simulator, the user has to provide an event file (described in 3.4.4). An event is defined to a triplet of an *event net* name, a *value* and a *state*. We call that a *value* is applied to the *event net* with the *state*. The simulator collects input events until a *run* command is issued, and then the input events are put into a *net-event-list* list. First, all the nets that are affected by these event nets (i.e. that are fanout nets of some event nets) are found and put into the *affected-net-list* list. Next, we simulate the affected nets and if there are any affected net that have new values as a result of the simulation, those nets are put into the net-event-list and iterate until no new values is found. Then it proceeds to the next set of input net events until all the input net events are processed. The top-level simulation algorithm is summarized in Figure 3.13.

The presence of the switch cell makes the simulation a little bit more complicated. Before simulating a net in the affected-net-list, we first have to find out if there are any other nets in the affected-net-list that are *equivalent* to the net. The nets are *equivalent* if they are connected by switch cells which are ON. The equivalent nets have to be simulated collectively. The simulation of a net can be performed by simply evaluating all the Lisp function of the terminals connected to the net in its driving-cell-list instance variable. If there is any conflict in value, then the value will be assigned to logic X and a warning is reported. Furthermore, if there is any conflict in values returned by simulation of equivalent nets, then the value of all the equivalent nets will be logic X and warnings are reported.

Since most of the nets are *affected* by either $\phi_1$ or $\phi_2$ clocking signals, we can use static variables to store the fanout-lists of the $\phi_1$ and $\phi_2$ nets at initialization phase. This approach can be called the *static*

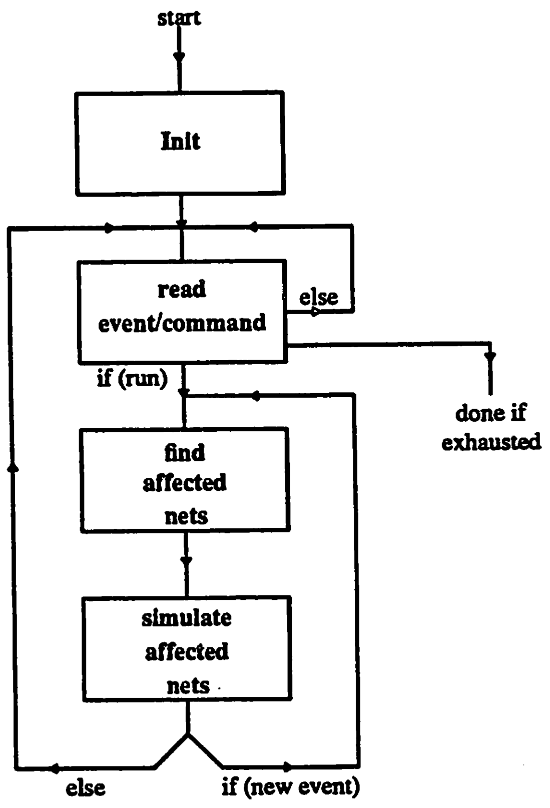

**Figure 3.13 Design Simulator event-driven algorithm**

*scheduling* for clock signals as opposed to the *dynamic scheduling* which involves calculating the affected-net-list at run time.

Sometimes there are cells that provides values to the output nets without any input nets (e.g. a static pull-down). This type of cells is called *self-generated.* Due to the nature of the event-driven algorithm, the output nets of the self-generated cells are not *affected* by any nets and hence will not be invoked for simulation. Therefore, these *self-generated* nets have to be recognized and simulated at the initialization phase and their values will not change during the simulation. The *self-generated* nets can be recognized by the fact that their fanin-list is null.

Properly handling the bus is an important issue in order to speed up the simulation. For each bus (which is a collection of nets), a hypothetic local net is created to store the ensemble value of the bus in a vector. For the input (output) bus, we treat the hypothetic local net as the fan-out (fan-in) net of all the nets in the bus by defining proper *packing* (*unpacking*) functions. This approach makes the hypothetic local nets act equally as the real nets. The simulation within the cell deals with the local nets instead of the input and output buses. Therefore, for a bus of size N, we need to perform just one simulation instead of N simulations.

### 3.4.4. Input/Output of Design Simulator

The Design Simulator is a batch mode simulator which takes an event file as input and generates a file of output values. The input event file consists of a number of commands, each of which is a list. The name of the command is specified by the first element in the list. The rest in the list specifies the *nets* or the *buses* that the command operates on. A *bus* is specified by the common name of the nets in the *bus* (the names of the nets are different only in the index part). A *net* can be specified either by a *net-name* or by a list of a *cell-name* and a *terminal-name*. Note that the *net-name*s and the *cell-name*s are full names (described in 3.1.4). Therefore, the *event* file has to be updated once the *instance-name* of the design is changed.

The commands in the input event file are used to specify input events and control the simulation. For examples, the command (h net1 net2 ...) sets net1, net2, ... to the value 1. The command (w net1 net2 ...) enters net1, net2, ... into the *watch* list whose values will be printed by the print command defined by (p). The (r) (run) command demands the simulation to be performed based on the events specified before the command. The user can define clock signals in the input event file and ask the simulation to be done in *major* cycles (a complete sequence of all phases of clock signals) instead of *minor* cycles. For a complete set of commands that are available for the Design Simulator, the reader should consult the user manual of the Design Simulator, which is listed in Appendix A.

The output of the Design Simulator is a file that consists of sections of output values of nets specified by the (w) command. Each section corresponds to a (p) command in the input event file.

## 3.4.5. Remarks on Simulator Performance

In general, the simulation of VLSI circuits can be performed on several levels which, from lower to higher abstraction levels, include process, circuit, timing, switch, logic, function and behavioral levels. The higher the abstraction level, the less accurate the result and the faster the simulation. The LagerIII Design Simulator can be categorized to a functional level simulator.

A *switch* level simulator uses a simple switch model of the MOS transistor. Even with these simplifications, the simulations of the entire chip using timing or switch-level simulators sometimes are found unfeasible. This is particularly true for the case of microprogrammed processors. To simulate the processing of one data input, hundreds or thousands of cycles of microprograms have to simulated. *Logic* simulators deal larger primitives such as AOI gates, than the switch level simulators. If a logic simulator allows functional models to be used to describe large functional blocks, it is referred to as the *functional* simulator. In general, the functional simulators are one order of magnitude faster than the switch-level simulator because larger primitives are used.

The Design Simulator in LagerIII is slow because it is only two to five times faster than esim, which is a switch-level simulator. The speed of the Design Simulator may be improved by fine tuning the simulation algorithm and the coding. However, it is found that the Lisp and the Flavors cause inherent performance degradation to the Design Simulator. The message sending is implemented by Lisp functions, and there is some overhead in finding the Lisp function from the message name.[15] However, the Lisp and Flavors are excellent development tools which allows the Design Simulator to be prototyped in a very short amount of time, which may even be shorter than interfacing existing functional simulator to the Design Database. For efficient simulation, dedicated hardware such as the Lisp machine should be used to alleviate performance problem.

# CHAPTER 4

# Frame Buffer Controller Chip

In this chapter, a frame buffer controller chip is illustrated which serves as an example of the use of the silicon assembly subsystem.

## 4.1. Frame Buffers

A *frame buffer* is a device to store a *frame* of image data in image processing systems. When the image processing algorithm requires random accessing of the image data, a frame buffer is necessary to hold the incoming image data (usually from a camera) which is in a *raster-scan* format. The frame buffer normally can work on one of two modes: *flash in* mode when the frame buffer keeps receiving image data from the camera, or *flash end* mode when one image frame is selected and stored in the frame buffer to be processed. If the image processing circuitry needs to access the frame buffer at the same time when the camera is sending image data to the frame buffer, a *double-buffered* scheme has to be used. A double-buffered frame buffer use two buffers for reading and writing, and their roles switch every other frame.

Frame buffers are commercially available [22], which can typically store a frame up to the size of 1024 x 1024 x 8 (256 gray levels) x 3 (RGB colors) bits in a board. Besides being expensive, they also occupy a lot of space, which leads to increased cost of the entire image processing systems. With the advancement of the memory technology, we are now in a position to realize more compact frame buffers. One of the key components in doing this is the *frame buffer controller*. A frame buffer controller interfaces the frame buffer to various other devices in the image processing system, and therefore has to be flexible to deal with different characteristics (e.g. interlaced or non-interlaced, number of lines in the image, synchronization, etc.) of various components. The TI VSC frame buffer controller chip [23] that was designed around TI's 256K VRAMs attempts to do this through user programmability, but is found to be not flexible enough for some configurations and requires costly supporting hardware.

In this chapter, a frame buffer controller for *single-buffered* frame buffers is presented. It was designed for an image processing system that consists of a GE TN2250 camera which generate analog image signals at 10 Mhz rate, a RS170 monitor, a 512 x 512 x 8 frame buffer implemented in two 1-Mbit DRAMs, a multibus host processor interface and a custom image processing board, and yet its architecture can easily be re-configured for other image processing system organizations. Furthermore, by modularizing the architecture and customizing the design for the particular image processing system, we can take advantage of the most advanced memory technology, and integrate peripheral glue-logic chips to reduce the board area.

## 4.2. Image Signal

A frame of image signal consists of N *lines* and each line consists of M *pixels* (picture elements). A typical timing specification [24] of image data is shown in Figure 4.1a and 4.1b. The timing of the lines are also referred to as the *vertical* timing. The timing of the pixels are also referred to as the *horizontal* timing. There is a horizontal blanking period which corresponds to the elapsed time to sweep from the rightmost pixel back to the leftmost to start the next horizontal line. Likewise, there is a vertical period which corresponds to the elapsed time to sweep from the bottom line back to the top to start the next frame. The blanking period consists of three parts: *front porch*, *sync* and *back porch*. The horizontal and vertical sync signals are used for the image processing devices to synchronize with one another. The timing is usually different for different devices. For example, the horizontal front porch is equivalent to 16 pixel periods for the GE TN2250 camera, and 11 pixel periods for the SONY XC-37 camera.

Synchronization can be done in two possible ways. A device can either receive the externally generated horizontal and vertical *drive* signals that *trigger* the horizontal and vertical sync signals, or send out the horizontal and vertical sync signals. In an image processing system, the sync signals can be generated by one of the devices or by an external signal generator.

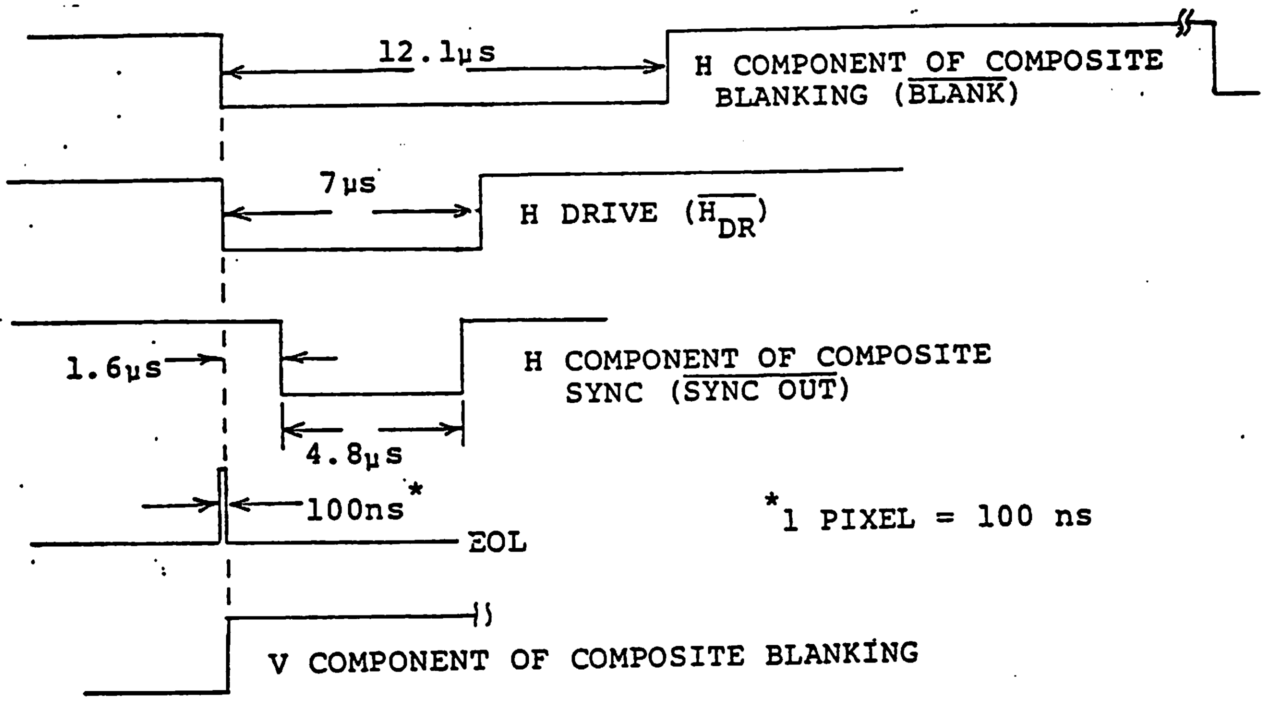
12.1μs
H COMPONENT OF COMPOSITE BLANKING (BLANK)
7μs
H DRIVE ($\overline{H_{DR}}$)
1.6μs
H COMPONENT OF COMPOSITE SYNC (SYNC OUT)
4.8μs
100ns*
EOL
*1 PIXEL = 100 ns
V COMPONENT OF COMPOSITE BLANKING

**Figure 4.1a Horizontal timing**
**front porch = 1.6 μs, sync = 4.8 μs, back porch = 5.7 μs**

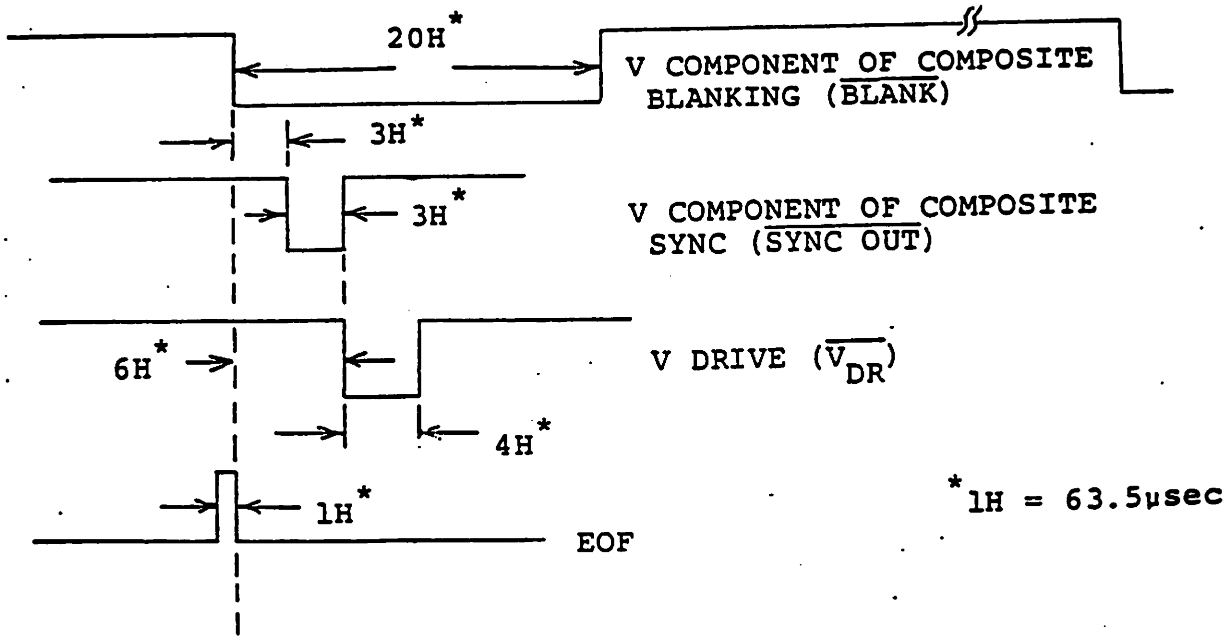
20H*
V COMPONENT OF COMPOSITE BLANKING (BLANK)
3H*
3H*
V COMPONENT OF COMPOSITE SYNC (SYNC OUT)
6H*
V DRIVE ($\overline{V_{DR}}$)
4H*
1H*
EOF
*1H = 63.5μsec

**Figure 4.1b Vertical timing**
**front porch = 3 lines, sync = 3 lines, back porch = 14 lines**

There are two standards to arrange the line signals. An *interlaced* device generates or receives image data in the order of line 0, line 2, line 4, ... line N-2, line 1, line 3, line 5, ... line N-1, where N is the total number of lines (assumed to be an even number). Even half and odd half frames each take $\frac{1}{60}$ sec to sweep. A *non-interlaced* device generates or receives image data in the order of line 0, line 1, line 2, ... line N-1.

A frame buffer controller should be able to adapt to these variations of signal timing, synchronization and line signal standard. The TI VSC chip provides the user a way to program the chip for different situations. In my design, parametrizable cell modules are used such that new chips can be generated for different image processing systems.

## 4.3. Chip Architecture

The frame buffer controller chip architecture is composed of four parts: (1) an *H-V control unit* which deals with the image data synchronization, (2) a *memory control unit* which generates timing signals to control the read and write of the frame buffer memories, (3) an *address generator* that generates the memory addresses, and (4) a *data path* that connects the data ports of various devices in the image processing system and controls the data flow among them. This hardware modularization is of critical importance to re-configuration. For example, suppose a new camera is used, then only the H-V control unit has to be adjusted for the new image data timing. Likewise, changing frame buffer memories only affects the memory control unit, and the addressing mode of the host processor only affects the address generator.

The H-V control unit is implemented by a state machine. There are 16 states, which are listed in Table 4.1 in order of actual timing.

There are five external signals that control the state transition: *horizontal sync*, *vertical sync*, *composite blank*, *eof* (end of frame) and *eol* (end of line). For example, the state (vertical front porch, horizontal front porch) transfers into the state (vertical front porch, horizontal sync) when the *horizontal sync* goes

| state bits | state description |
|---|---|
| 0000 | (vertical front porch, horizontal front porch) |
| 0001 | (vertical front porch, horizontal sync) |
| 0010 | (vertical front porch, horizontal back porch) |
| 0011 | (vertical front porch, horizontal signal) |
| 0100 | (vertical sync, horizontal front porch) |
| 0101 | (vertical sync, horizontal sync) |
| 0110 | (vertical sync, horizontal back porch) |
| 0111 | (vertical sync, horizontal signal) |
| 1000 | (vertical back porch, horizontal front porch) |
| 1001 | (vertical back porch, horizontal sync) |
| 1010 | (vertical back porch, horizontal back porch) |
| 1011 | (vertical back porch, horizontal signal) |
| 1100 | (vertical signal, horizontal front porch) |
| 1101 | (vertical signal, horizontal sync) |
| 1110 | (vertical signal, horizontal back porch) |
| 1111 | (vertical signal, horizontal signal) |

Table 4.1 The 16 states of the H-V control unit

from HIGH to LOW, and then transfer into the state (vertical front porch, horizontal back porch) when the *horizontal sync* goes from LOW back to HIGH. When the *eol* signal is HIGH, the state (X, horizontal signal) transfers to the state (Y, horizontal front porch) where X and Y may or may not be the same vertical sub-state. When the *eof* signal is HIGH, the state (vertical signal, horizontal signal) transfers into the state (vertical front porch, horizontal front porch). The state transition of the H-V control unit is described in *bdsyn* format in Appendix B.

The fact that the GE camera does not provide separate vertical and horizontal blanking signals make the design of the H-V control unit a bit more difficult. A counter is required to make sure that the transition from the state (X, horizontal back porch) to the state (X, horizontal signal) does not have to depend on external signals. For the case of the GE camera, the (X, horizontal back porch) state is 55 cycles or 5.5 μs. To implement this, we have the choice of either a loadable down counter or a resettable up counter with a constant modulus. The second architecture is selected based on macrocell availability and the fact that the LagerIII silicon assembly sub-system makes it easy to program the constant modulus through parametrization. The architecture of the H-V control unit is shown in Figure 4.2.

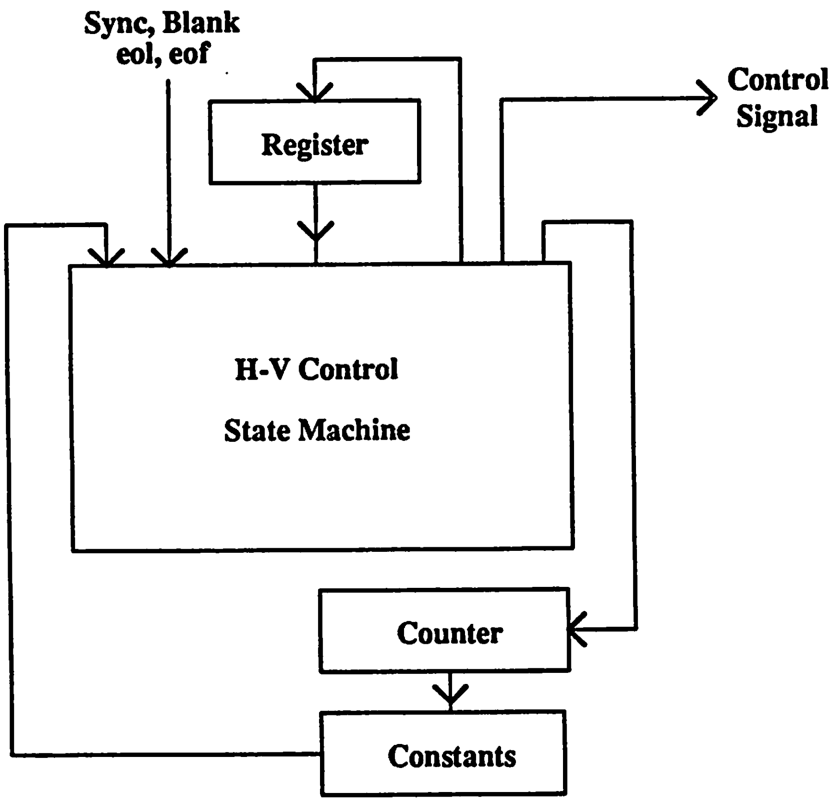

**Figure 4.2 H-V control unit architecture**

The *memory control unit* provides five modes of memory access timing: *refresh*, *read*, *write*, *column read* and *column write*. Each of the five memory access operations takes different numbers of cycles to complete. A refresh operation takes 3 cycles. A read operation and a write operation each take 3 cycles. A column read operation and a column write operation each take 1 cycle (with some initial cycles to set up the row). The memory control unit is also implemented by a state machine. The inputs to the state machine include the state of the H-V control unit, *host* bits and the *flash* bit. The host bits encodes the status of the host processor: 00 = no host operation, 01 = host read, 10 = host write and 11 = illegal. The flash signal determines whether to receive image data from the camera. The H-V control unit state is also used to determine which memory operation to perform. For example, if the H-V control unit state is (X, horizontal front porch) then *refresh* operations are performed. The state transition of the memory control unit is described in Appendix B.

There is a priority among the five memory operation modes. The *refresh* operation has the highest priority. Therefore, during the (X, horizontal front porch) H-V control unit state, the refresh operation will not be interrupted. The *column write* operation has the second priority. Therefore, during the (X, horizontal signal) H-V control unit state and if the flash bit is HIGH then the column write operations are performed without interrupt. This ensures a correct content in the frame buffer memory. The *read* and *write* operations have the third priority. The *column read* operation has the lowest priority. Since the column read operation can be interrupted by host read or write operations, some *dark spots* on the monitor may be created as a result. However, the priorities of the five operations can be rearranged easily if the system designer is willing to, for example, buffer the host read and write operations such that the monitor display will not be interrupted.

The memory control unit state machine has four output signals: *cas**, *ras**, *wr** and *oe**. These timing signals are used to control the memory operations. For Toshiba TC514256 1-Mbit DRAMs that the designed frame buffer controller is targeted to, the state machine generates new output signals every 100 ns except that the *ras** changes every 50 ns during the column read or column write modes. The architecture of the memory control unit is shown in Figure 4.3.

The *address generator* provides three types of address: host *random access* (row and column) addresses, *raster-scan* (row and column) addresses and refresh (row) addresses. In the TI VSC chip, the host addresses are provided in an indirect way. The VSC chip has X-Y registers on chip, which store the row and column addresses of a particular image pixel that the host processor needs to access. The values of the X-Y registers can be adjusted by *X-Y adjustment code* to move around the image frame. The advantage of this scheme is that fewer host address bits need to be specified. The disadvantage is that it does not provide full degree of freedom of random accessing. The address generator in my design assumes that the host processor specifies both the row and column addresses completely.

The address generator generates the raster-scan and refresh addresses internally. The raster-scan row address is generated by a *row counter*, which is incremented under the control of the H-V control unit

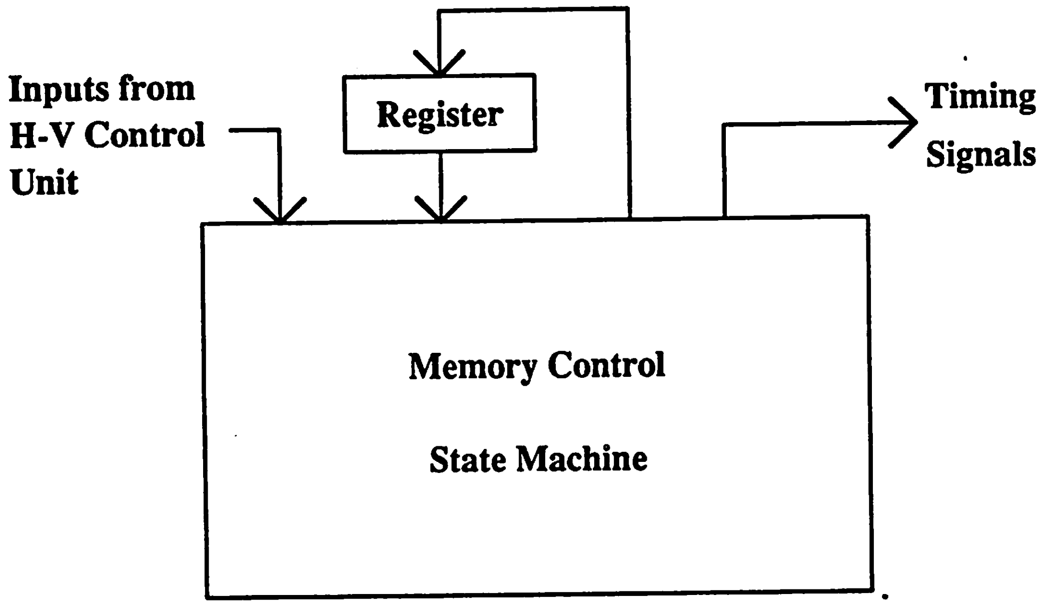

**Figure 4.3 Memory control unit architecture**

when the (X, horizontal signal) state transfer into the (Y, horizontal front porch) state. The raster-scan column address is generated by a *column counter*, which is incremented every (100ns) cycle.

Because a refresh operation takes 3 cycles, one approach is to let the refresh row address change every 3 cycles. However, it is easier to break the refresh address generator into two counters, in which the higher bits (N-2 with $2^N$ equals the number of lines) increment when the H-V control unit enters the (X, horizontal front porch) state, and the lower bits (2) increment every cycle. Because 3 is co-prime with $2^2 = 4$, and the horizontal front porch time segment is 16 ($> 3 \times 4 = 12$) cycles, this scheme guarantees that all four rows are refreshed in one horizontal front porch time segment. The Toshiba TC514256 memory chips require *512 refresh cycles/8ms*. The refresh address generator will sweep all 512 rows in

$$\frac{512}{4} \times 63.5\mu\text{sec} = 8.1ms$$

which is only slightly off the specification.

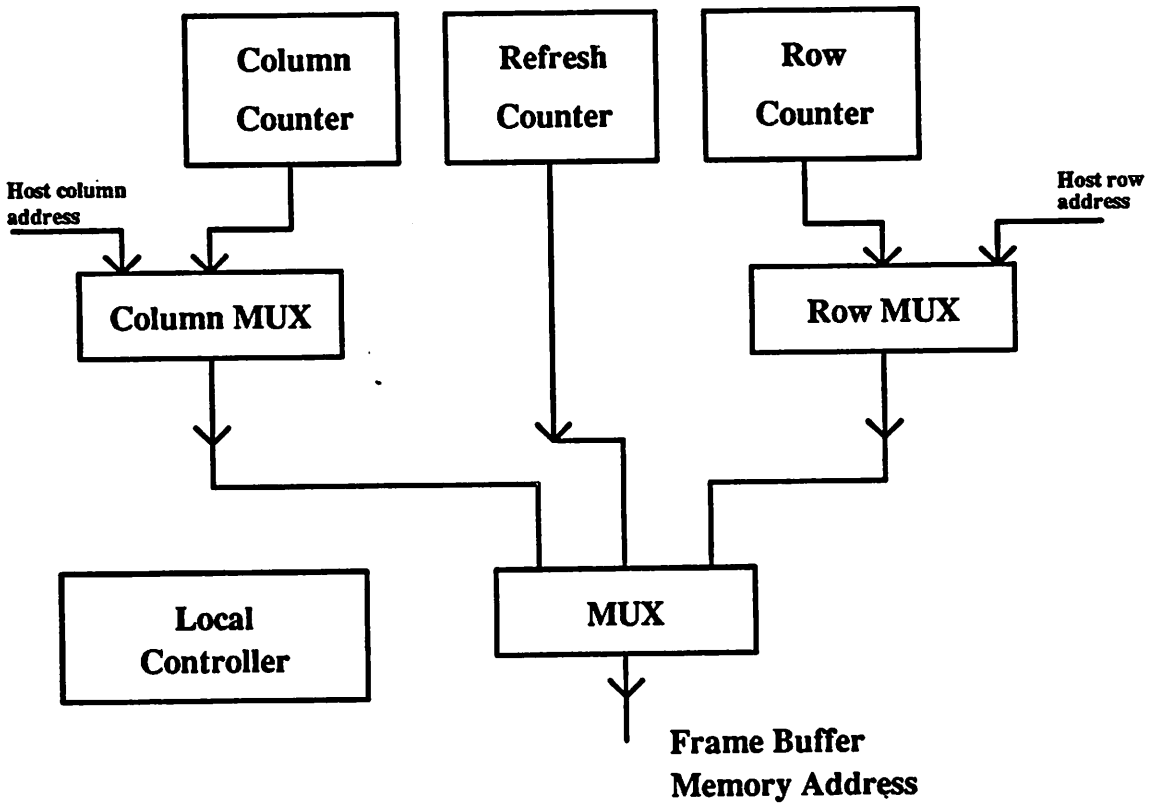

**Figure 4.4 Address generator architecture**

There are a number of multiplexers in the address generator which are used to select one of the addresses generated. Since the refresh operations have the highest priority, hence in the (X, horizontal front porch) state of the H-V control unit, the refresh address is selected. The choice of host addresses or raster-scan addresses are determined by the *host* external control signals that control the memory control unit as well. The choice of row or column address depends on the memory timing. The Toshiba TC514256 memory chip requires that the column address follows the row address. The architecture of the address generator is shown in Figure 4.4.

The frame buffer controller provides a *data path* through which various devices in the image system can communicate with each other. The *sources* of image data include the camera, the host processor and the frame buffer memory. The *destinations* of image data include the monitor, the host processor, the image processing board and the frame buffer memory. The data path has to make sure that only one of the sources is sending image data at any given time. This is done by a simple logic design. First, if the *flash* signal is HIGH, then the camera is sending data and the host and the frame buffer memory are prohibited from sending data. Otherwise if the *host* signals signify *host write* operations, then the host processor is sending out image data. If camera and host are both inactive, then the frame buffer memory is enabled. An A/D converter macrocell[25] can be integrated which takes the camera analog signal as input and generates 8-bit digital values. The architecture of the data path is shown in Figure 4.5.

## 4.4. Layout Generation

The layout of the frame buffer controller chip is generated using the LagerIII silicon assembly subsystem. A set of sdl files are created to specify the design hierarchically. Three test chips are also prepared to test the functionality of the H-V control unit (HVCtest chip), the memory control unit (MCtest chip) and the address generator (AGtest chip) respectively. Because the design is hierarchical, the test chips are

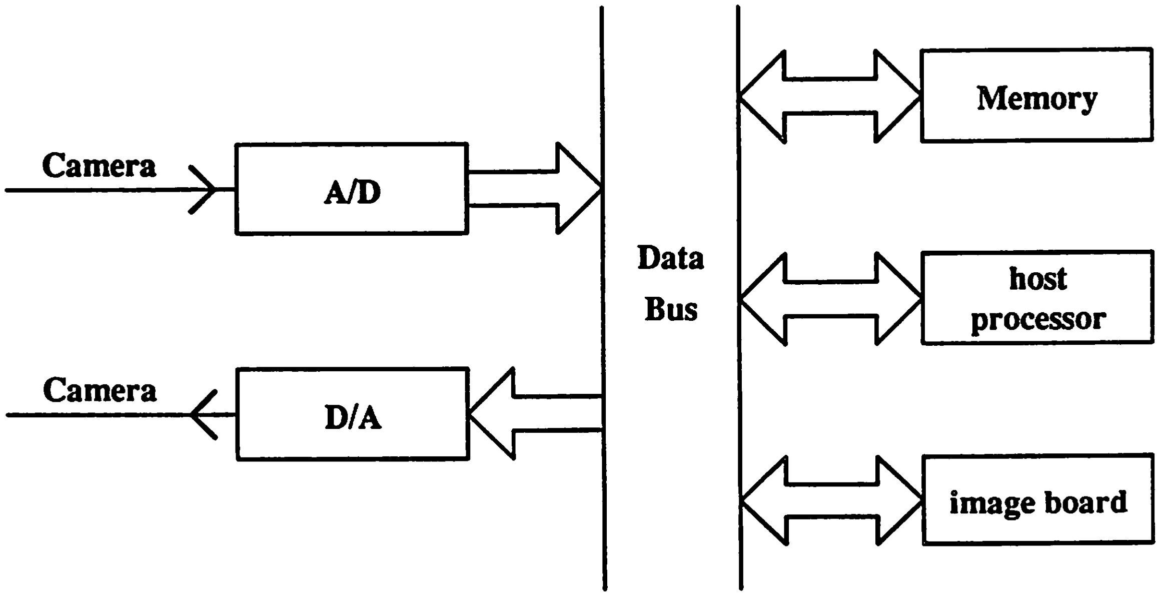

**Figure 4.5 Data path architecture**

conveniently generated.

The layout generation makes use of four layout generation tools: **Flint**, **DPC**, **TimLager** and **Wolfe**. The counter and constant cells in the H-V control unit are generated by **TimLager**. The registers in H-V control unit and memory control unit, and counters and multiplexers in the address generator are generated by **DPC**. Local control units in each of the four parts of the chip, and the combinational blocks of the H-V control unit and memory control unit are generated by **Wolfe**. The placement and routing of the four parts and the whole chip are done by **Flint**.

LagerIII only provides *equation* level input for specifying the logic blocks with limited logic minimization. This is used to specify the local control units. However, it is cumbersome to describe the state transitions using a equation level input, and hence the two combinational blocks are designed in *bdsyn* formats (Appendix B, in which *sm1.bdsyn* is for the H-V control unit and *sm2.bdsyn* is for the memory control unit) and are minimized by a multi-level logic minimizing program, mis. Alternatively, the bdsyn input can also be minimized by a two-level logic minimizer, espresso, which results in a PLA-based realization.

The data path contains an 8-bit 10MHz analog-to-digital converter which is manually designed.[25] The LagerIII silicon assembly system can incorporate macro cells designed through other means by treating them as big leaf cells. The macro cells must be designed or generated in Magic format. The two combinational blocks in the the H-V control unit and the memory control unit are incorporated the same way by first translating the OCT physical view formats generated by Wolfe into Magic formats.

With the aid of LagerIII silicon assembly system, the frame buffer controller chip were designed, generated and simulated in two months. The die photos of the frame buffer controller chip and three test chips are shown in Figure 4.6 - 4.9.

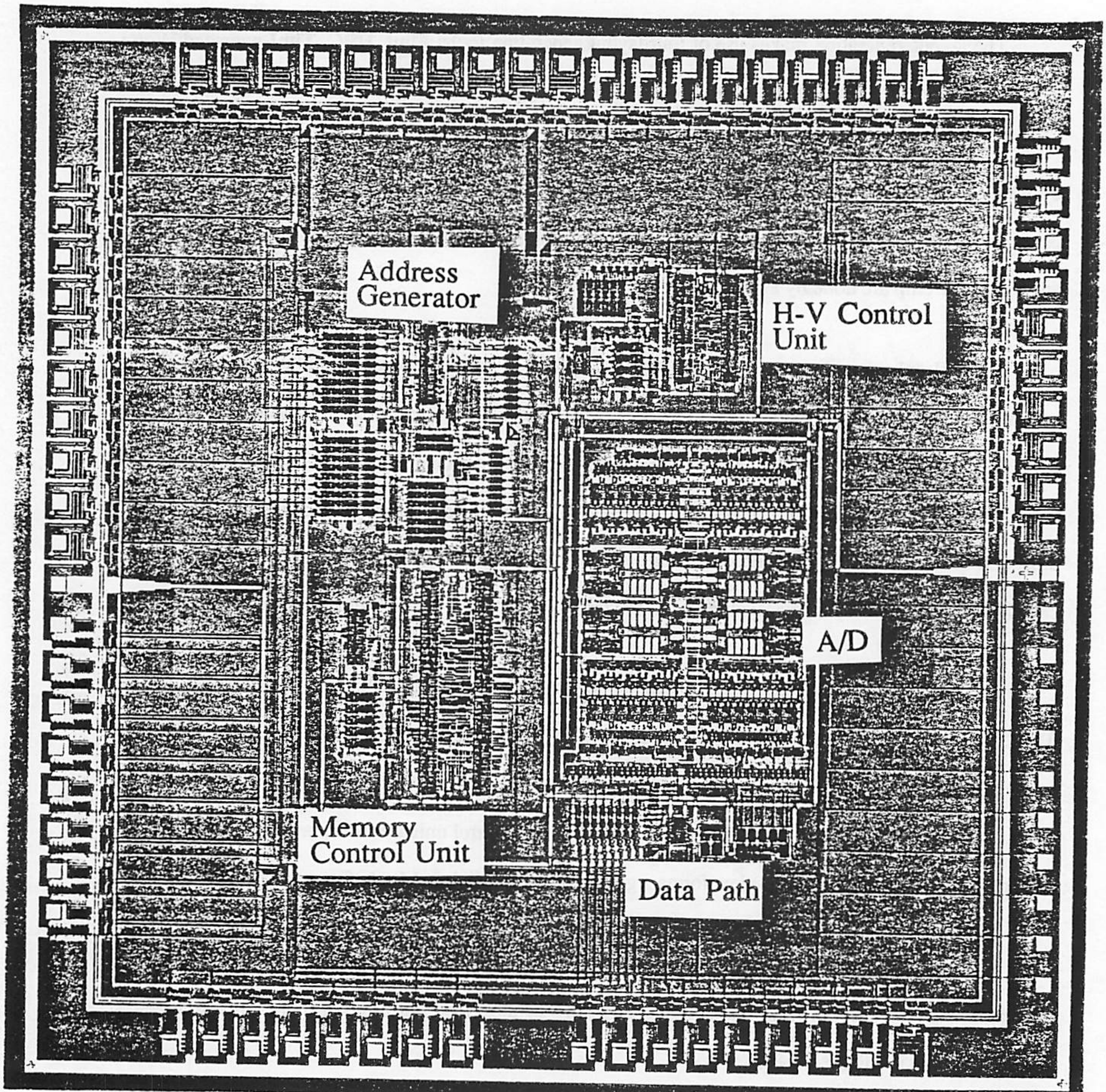

**Figure 4.6 The frame buffer controller chip die photo**

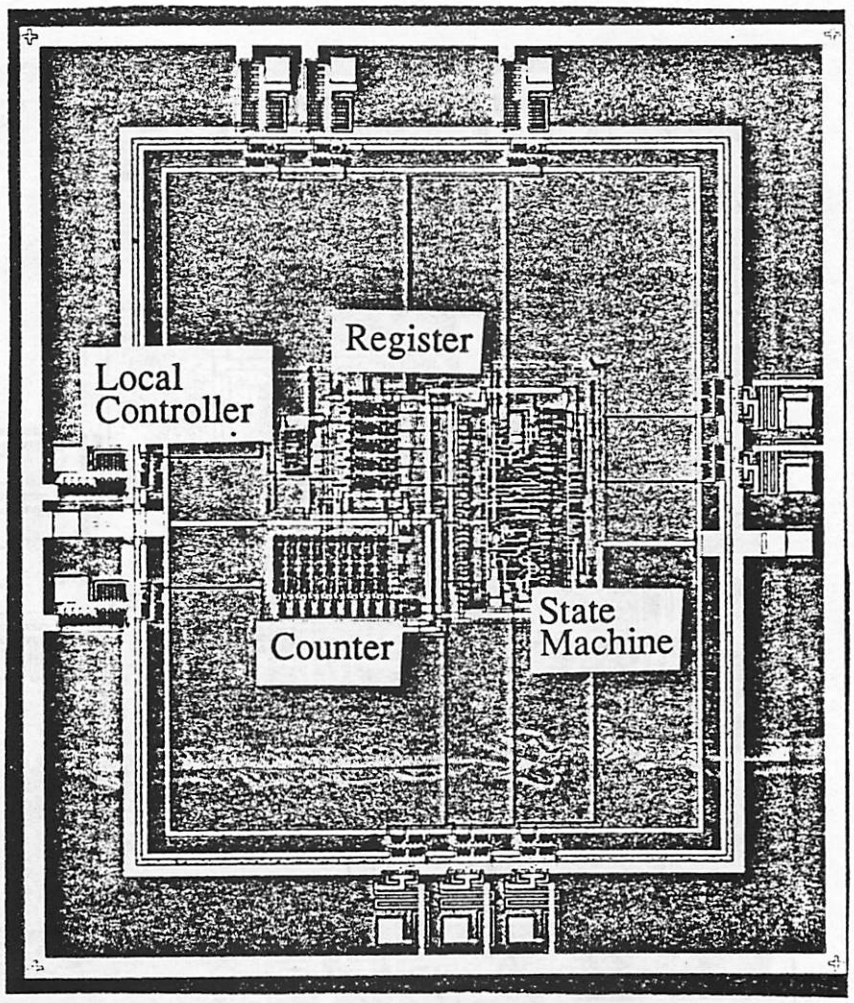

**Figure 4.7 The H-V control unit test chip die photo**

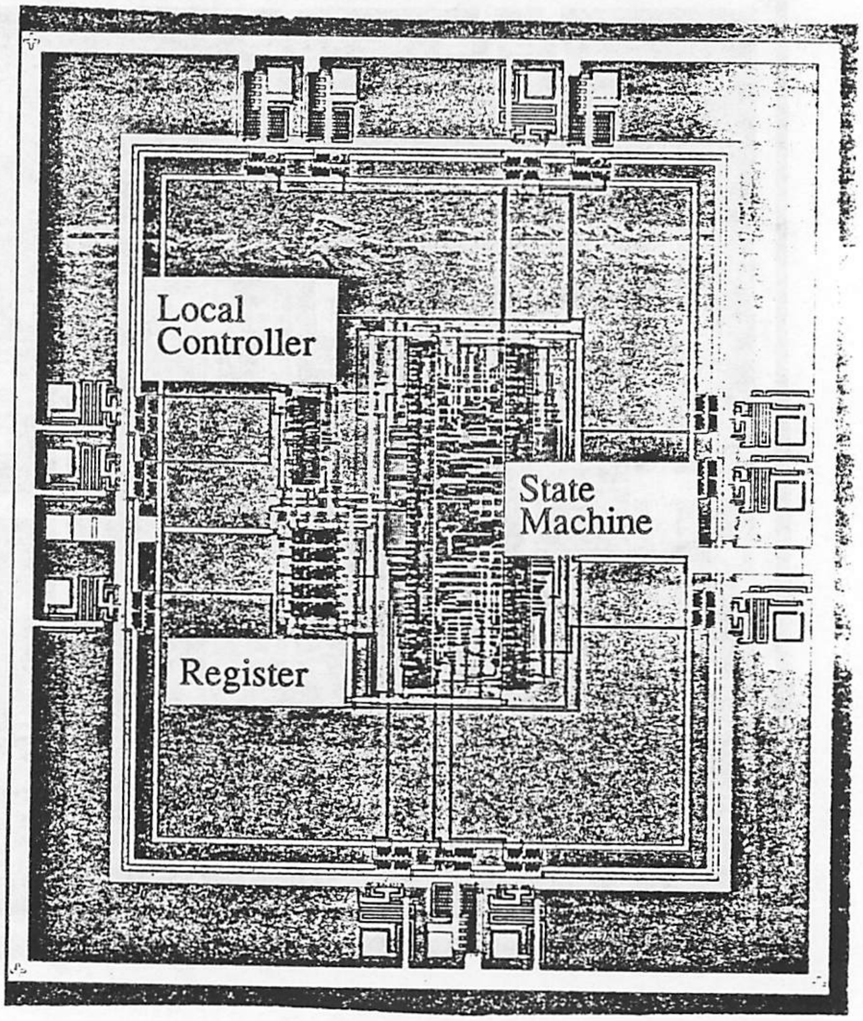

**Figure 4.8 The memory control unit test chip die photo**

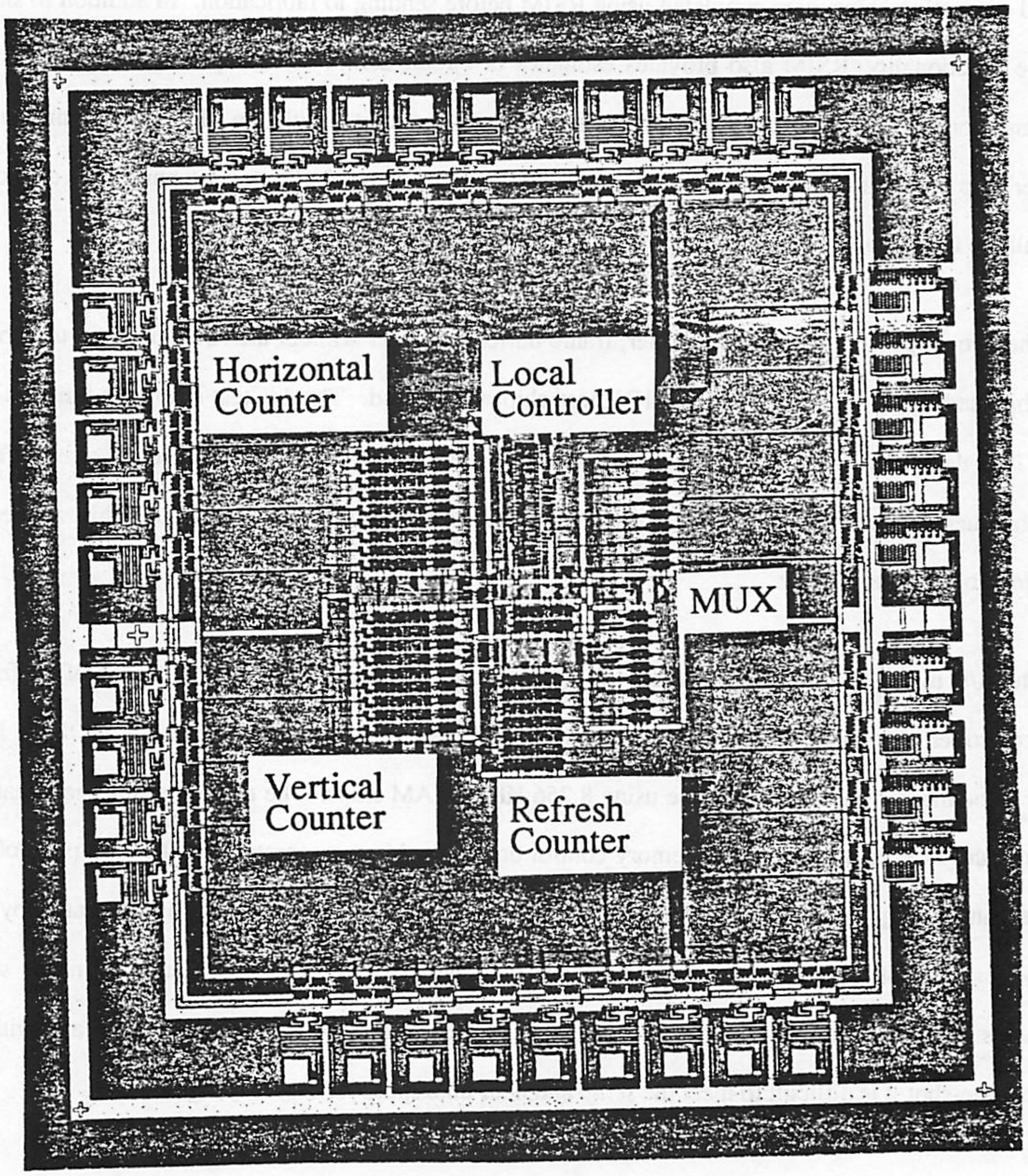

**Figure 4.9 The address generator test chip die photo**

## 4.5. Simulation and Testing Results

All the testing chips were simulated using RSIM before sending to fabrication. In addition to simulating the functionality, RSIM also provides estimates of circuit delays using a primitive model for the MOS transistor. However, since RSIM cannot handle analog circuit simulation, the entire frame buffer controller chip is not simulated with the analog-to-digital converter (A/D). For testing purpose, a chip containing all the frame buffer controller circuits except the A/D is simulated and fabricated.

The five chips (frame buffer controller, frame buffer controller without the A/D and three test chips) were fabricated by MOSIS. All but the HVCtest chip were tested. The MCtest chip was the first one tested. Two design errors caused by mistakes in entering the design input files were discovered. These were subsequently removed in the design of the frame buffer controller chip. The AGtest chip was tested to function correctly at 10MHz.

The A/D macro cell is currently under test by a separate test chip. Therefore, the testing of the frame buffer controller chip is performed with the one without the A/D. Due to inavailability of the 1 Mbit DRAM in the market, the testing is done using 8 256 Kbit DRAM chips. The testing results show that the four parts, the H-V control unit, the memory control unit, the address generator and the data path, of the chip are functioning by itself. However, the timing of the memory control signals generated by the memory control unit, and the memory address signals generated by the address generator needs some adjustments. The main problem is that all signals are derived from a 10 MHz system clock, and without using any one-shot it is difficult to meet the *setup* and *hold* time specs for arbitrary signal edges.

# CHAPTER 5

# The Silicon Compilation Subsystem

The LagerIII silicon compilation subsystem consists of three parts: a *translator* that translates the applicative *Silage* program to a procedural intermediate language called *RL*, a *compiler* that compiles the RL language into symbolic microcode (also called a *rass* program) and a *control generator* that produces the parameter file output from the rass program. A unique feature of the silicon compilation subsystem is that the behavioral description can be *retargeted* to different pre-defined structural descriptions. One structure design which is called the *KAPPA* architecture, that the behavioral description is currently mapped onto is also described. This work has been done in collaboration with Edward Wang (Silage translator)[26], Ken Rimey (RL compiler)[27] and Khalid Azim (KAPPA structure design)[28].

## 5.1. The KAPPA architecture

The KAPPA architecture is based on a simple architecture model[29] (Figure 5.1) that consists of a control unit and a data path. The control unit generates *control signals* that control the operations of the data path, and the data path provides *status signals* that affect state transitions in the control unit. The KAPPA data path contains four components: an arithmetic unit (AU), an address processing unit (APU), a logic unit (LGU) and a data memory (RAM). Typical status signals from the data path include the sign bits of the AU and APU, and state bits of the LGU.

The block diagram of the AU is shown in Figure 5.2. It does not contain a multiplier and hence multiplications are performed by a series of shift-and-add operations. There are only two data manipulation operators in the AU: the shifter and the adder. In addition, complementation or zeroing can be performed to modify the data at the two input buses (*abus* and *bbus*) of the adder. The operands of the adder can come from RAM, the local registers or the APU (immediate addressing mode). The result of the adder can

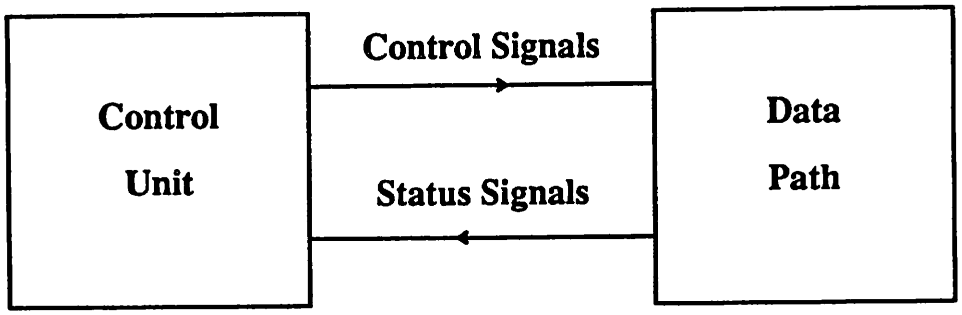

**Figure 5.1 The architecture model that KAPPA is based on**

go to RAM or the local registers. There is an I/O port, which allows data communication between the local registers and the outside world (e.g. off chip, other processor, etc.).

The block diagram of the APU is shown in Figure 5.3. The APU calculates the effective address of the RAM using the address field in the control signal generated by the control unit. The APU supports four addressing modes: index, relative, immediate and looping. The *index* addressing can be performed by using one of local registers as the index register. The *relative* addressing can be performed by storing the base address in the register and supply the offset address through the address field in the control signals. The *immediate* addressing can be performed by supplying the data directly through the address field in the control signals. The immediate data is transferred to AU by the connection of the AU mbus with the APU eabus. The magnitude of the immediate data is limited by the word length of the APU, which is usually smaller than the word length of AU. The *looping* addressing mode is usually handled by a *loop counter* in the control unit (discussed later). The APU plays an auxiliary role when there are *nested* looping operations. A looping operation can be supported in the APU by storing the *loop count* (number of iterations) in one of the registers and decrement its value when one loop is completed. When the register reaches zero, a control signal is sent to the finite state machine to change the state.

The LGU is implemented by a state machine whose combinational part is implemented by a PLA. The primary inputs of the PLA include the sign bits of the APU and AU and optionally any pertinent exter-

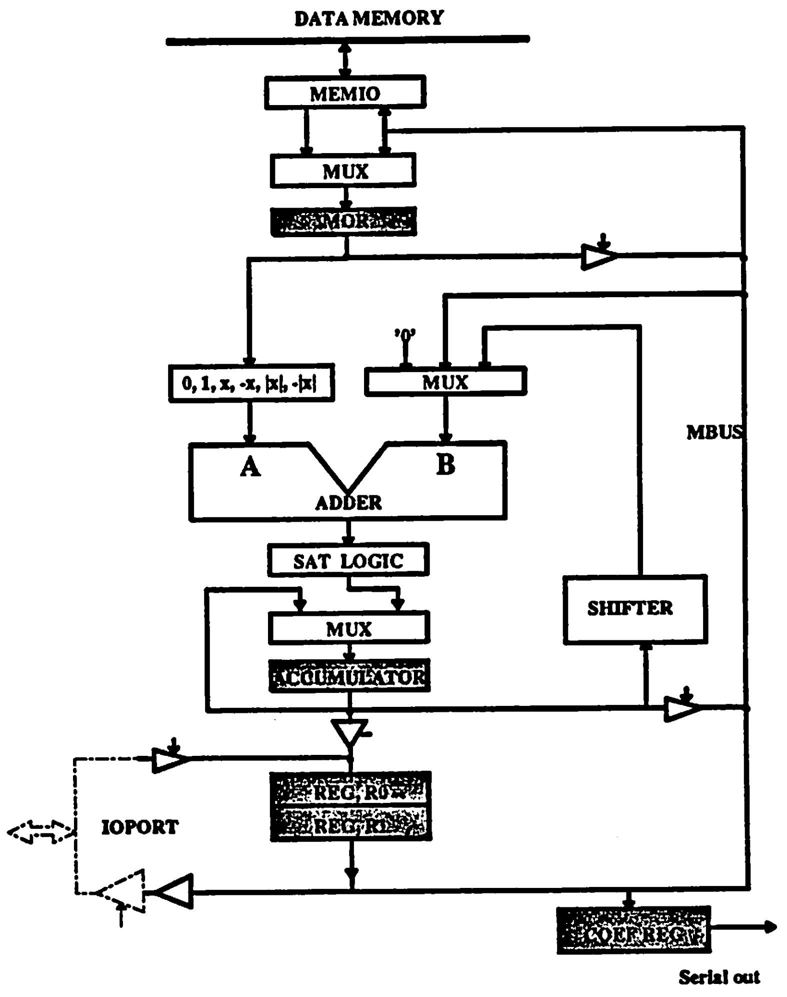

**Figure 5.2 The arithmetic unit (AU) of KAPPA**

nal signals. The primary outputs of the PLA form part of the status signals that feedback to the control unit. The logic operations in LGU are determined by the particular behavioral description. Contrary to most processor architecture in which the arithmetic and logic operations are performed together in an ALU, the AU and LGU are two separate functional units in the KAPPA architecture.

The KAPPA control unit contains six major components: a finite state machine, a program counter, a control store, a stack, a loop counter and a timer (Figure 5.4). The control store stores a number of blocks of control signals. The address of the control store contains two parts: the higher bits (block address) are

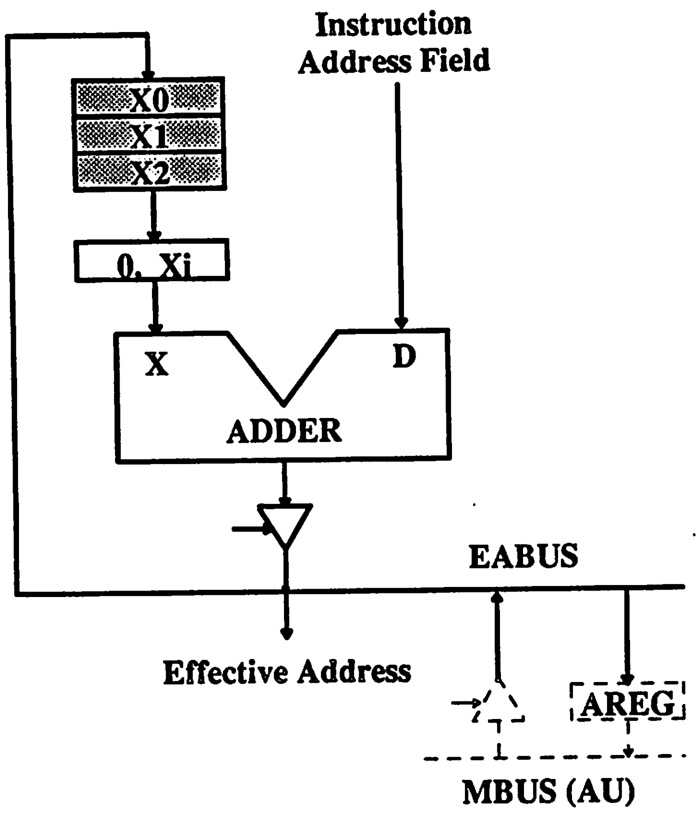

**Figure 5.3 The address processing unit (APU) of KAPPA**

generated by the finite state machine and the lower bits (line address) are generated by the program counter. If the program execution reaches the end of a block of control signals, a eob (end of block) signal signifies the finite state machine to change state and thereby generates a new block address, and reset the program counter. Otherwise, the finite state machine stays at the same state and the program counter increments the line address at each cycle to generate the next set of control signals.

The stack and the loop counter provide more control flow operations in addition to the *branch* operation supported by the finite state machine. The stack efficiently supports the *subroutine call* and *return* operations. The loop counter efficiently supports the *looping* operation. The number of iterations (*loop count*) for each looping operation can be extracted from the algorithm and loaded in the loop counter. When executing the looping operation, the loop counter increments and compares with a pre-stored loop

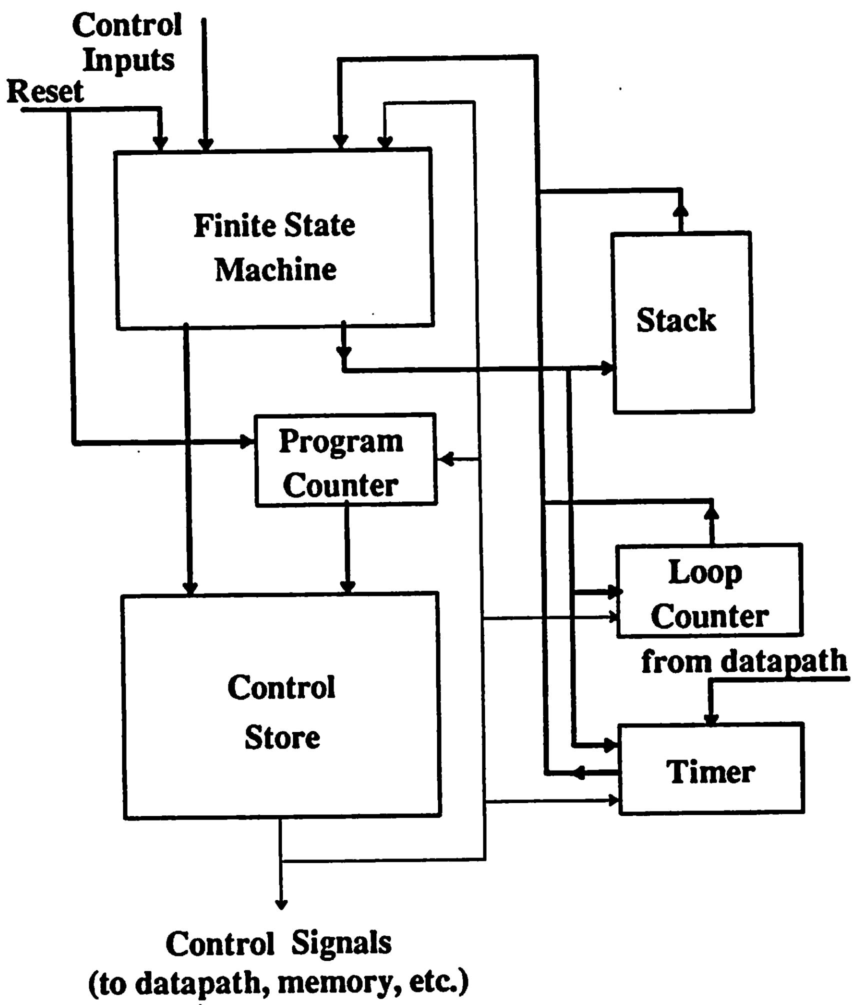

**Figure 5.4 The control unit of KAPPA**

count. A control signal is generated by the loop counter that depends on the comparison result. The control signal is sent to the finite state machine to control the state transition. If the content of the loop counter is smaller than the loop count, then the state machine stays at the current state. If the content of the loop counter is equal to the loop count, then the state machine transits to the next state. When there are several looping operations in the algorithm with different loop counts, all loop counts are stored. Each loop count results in a control signal to the state machine. When there are *nested* looping operations, only the inner most looping operations are handled by the loop counter and the rest are handled in the APU.

The *timer* is used to synchronize the program execution with the data samples. Due to the more sophisticated control flow operations with some of which are *data-dependent*, the total number of instructions executed may vary from sample to sample. The timer stores the *worst* case (largest) number of instructions and if the program finishes earlier, the finite state machine will enter a *wait* state till the timer is done.

The KAPPA architecture can be parametrized. For example, the word length of the AU is determined by the fixed-point arithmetic accuracy required by the algorithm. The word length of the APU is determined by the sizes of the RAM, which is in turn determined by the algorithm. Other parameters include the memory contents of the LGU and the finite state machine and control store in the control unit, the size of the stack and loop counter, etc. The possibility of parametrization is one of advantages of the KAPPA architecture over the commercial signal processor[30] in which the physical dimensions of functional modules cannot be tailored to match the algorithm. The parametrizable architecture also lends itself easily for the silicon compilation subsystem to map the behavioral description of the algorithm to the architecture.

## 5.2. The Relationship Between the Instruction Set and the Architecture

Most conventional architecture designs are referred to as the *instruction set architecture*[31] because the architecture is conceived by the programmer as an implementation of a particular instruction set. This approach leads to a top-down design methodology in which the instruction set is designed before the architecture is designed. It also causes the computer architects to put emphases on the design of instruction set, rather than the structural implementation of the instruction set. For example, the RISC[32] concept was proposed to advocate the use of a simpler instruction set. One major problem of this approach is, however, that the instruction set designer does not have a good understanding of how long each instruction takes to execute until the implementation of the instruction set is completed. Moreover, since the instruction set designer does not know how the structural implementation is designed, the addition of new instructions may require a major redesign of the structural implementation.

In the LagerIII silicon compilation subsystem, since the structural implementation is designed first, a bottom-up approach is employed in which the instruction set is *extracted* from the existing structural design. Each instruction extracted is called a *primitive instruction.* Each primitive instruction describes an operation that a piece of data is transferred from the *source(s)* to the *destination.* Both the source and the destination are called *resources* which can be divided into two types: *buses* and *registers*. Each primitive instruction can be executed in one instruction cycle. Two primitive instructions are said to have a *conflict* in resource if they both transfer data to the same destination. Several primitive instructions may be executed in the same cycle if no conflicts is resulted in. For example, the

(mor=mbus)

primitive instruction transfers the data in the *main bus* (mbus) to the *memory output register* (mor). The

(mbus=r* 2)

primitive instruction transfers the data in the 2nd register of the *register file* to the main bus. This example shows that an argument can be used to customize the primitive instruction, and hence the total number of primitive instructions is reduced. The sets of primitive instructions extracted from the AU and APU in the KAPPA architecture are described in Table 5.1 and 5.2, respectively. The sets of buses and registers defined in the KAPPA architecture are described in Table 5.3.

One of the advantages of extracting the primitive instructions is that the *user instructions* (or assembly level instructions) can easily be formed by grouping a number of primitive instructions. For example, the user instruction which reads the 3rd element in the array A,

r(A[3])

can be constructed by the group of

(mor=mem)    (addr A)    (offset 3)

primitive instructions. Two user instruction are said to have a conflict if the corresponding two groups of primitive instructions result in a conflict. For example, by setting the *bbus* to ONE, an *add1* user instruction is formed which simply increments the *abus* operand. By setting the *bbus* to ONE and *complementing*

| instruction | description | argument |
|---|---|---|
| mor=mem | *read* from memory to *memory output register* | |
| mem=mbus | *write* to memory from *main bus* | |
| mcondload | conditional *write* | |
| mor=mbus | | |
| r*=rbus | load *register bank* by *register bus* | *register bank* address |
| rcoef=mbus | load *multiplication coefficient register* form *main bus* | |
| mbus=mor | | |
| mbus=r* | load *main bus* by the register | *register bank* address |
| mbus=acc | load *main bus* by accumulator | |
| rbus=acc | load *register bus* by accumulator | |
| rbus=ioport | input from ioport to *register bus* | |
| ioport=extport | latch external data to ioport | external port address |
| ioport=mbus | output to ioport from *main bus* | |
| extport=ioport | strobe ioport data to external port | external port address |
| acc=0 | clear accumulator | |
| acc=sum | store adder result to accumulator | |
| acc=abus | *bbus* is zero | |
| acc=bbus | *abus* is zero | |
| abus=1 | carry-in to adder is set high | |
| abus=mor | | |
| abus=-mor | *abus* gets the 2's complement of *memory output register* | |
| abus=absmor | *abus* gets the absolute value of *memory output register* | |
| abus=-absmor | *abus* gets the 2's complement of the absolute value of *memory output register* | |
| abus=coef.mor | same as abus=mor if the LSB of the *multiplication coefficient register* is HIGH | |
| abus=coef.-mor | same as abus=-mor if the LSB of the *multiplication coefficient register* is HIGH | |
| abus=coef.absmor | same as abus=absmor if the LSB of the *multiplication coefficient register* is HIGH | |
| abus=coef.-absmor | same as abus=-absmor if the LSB of the *multiplication coefficient register* is HIGH | |
| abus=‾coef.mor | same as abus=mor if the LSB of the *multiplication coefficient register* is LOW | |

**Table 5.1 Primitive instructions of AU**

| instruction | description | argument |
|---|---|---|
| abus=˜coef.-mor | same as abus=-mor if the LSB of the *multiplication coefficient register* is LOW | |
| abus=˜coef.absmor | same as abus=absmor if the LSB of the *multiplication coefficient register* is LOW | |
| abus=˜coef.-absmor | same as abus=-absmor if the LSB of the *multiplication coefficient register* is LOW | |
| bbus=mbus | | |
| bbus=acc>* | *bbus* gets right-shifted accumulator | bits shifted |
| bbus=acc<* | *bbus* gets left-shifted accumulator | bits shifted |
| acondload | conditionally load the accumulator | |
| shrcoef | right shift the *multiplication coefficient register* | |
| nosat | turns off saturation of accumulator | |
| aip | accumulate if positive | |

**Table 5.1 (cont.) Primitive instructions of AU**

| instruction | description | argument |
|---|---|---|
| x*=eabus | load *x register* by *effective address bus* | *register bank* address |
| xcondload | conditionally load *x register* | |
| addr | address from the *control store* | variable name |
| offset | used together with *addr* primitive instruction to address an array | array index |
| xip | accumulate if positive | |
| xbus=x* | load *xbus* by *x register* | *register bank* address |
| xbus=0 | clear *xbus* | |
| eabus=sum | store adder result in *effective address bus* | |
| eabus=mbus | load *effective address register* by *main bus* of AU | |
| areg=eabus mbus=areg | load *main bus* of AU by *effective address bus*; it takes two cycles | |
| timerreg=eabus | load *timer register* in control store by *effective address bus* | |

**Table 5.2 Primitive instructions of APU**

| resource name | description |
|---|---|
| mor | memory output register |
| acc | accumulator |
| r0, r1 | AU register bank |
| rcoef | multiplication coefficient register |
| x0, x1, x2 | APU register bank |
| mem | data memory |
| areg | hypothetical APU register |
| timerinreg | timer register |
| ioport | i/o port |
| mbus | AU main bus |
| rbus | AU register bus |
| abus | AU adder inputA |
| bbus | AU adder inputB |
| eabus | effective address bus |
| xbus | APU adder inputA |
| dbus | APU adder inputB |
| extport | external port |

**Table 5.3 Resources (registers and buses) in KAPPA**

the *abus* operand, a *unary minus* user instruction is formed. These two user instructions conflict with each other because they both use the *abus* and bbus in a different way.

In summary, from the structural design of the architecture, a set of primitive instructions can be extracted. The set may not be the exhaustive list of all possible primitive instructions in the architecture, some judgements on which primitive instruction is *useful* should be made. In addition, a number of user instructions can be obtained by grouping the primitive instructions. This approach is contrary to the instruction set architecture approach because the architecture is designed before the instruction set is.

Because a fixed architecture does not work well in a wide range of applications, it is very important to be able to tailor the architecture accordingly to the particular application. In LagerIII silicon compilation subsystem, this can be done in an iterative way. First, an existing architecture and its instruction set are used to which the algorithm is mapped. If the result is unsatisfactory, then the architecture is modified and new sets of primitive instructions and user instructions are obtained. The new user instruction set is then used for the algorithm. This process iterates until a satisfactory architecture is obtained. In general, the reason that an architecture is not efficient for an algorithm is that some frequently used instructions are not directly implemented, which can easily be recognized from the histogram of the user instruction set.

For example, the AU in the KAPPA architecture does not have a multiplier and the multiplication is done by a series of shift-and-add's. There are two possible ways to place the shifter: either before the adder and hence it shifts one of the operands, or after the adder and hence it shifts the result (in the accumulator). To analyze the trade-offs of the two arrangements, the following 2nd-order finite impulse response (FIR) filter is used:

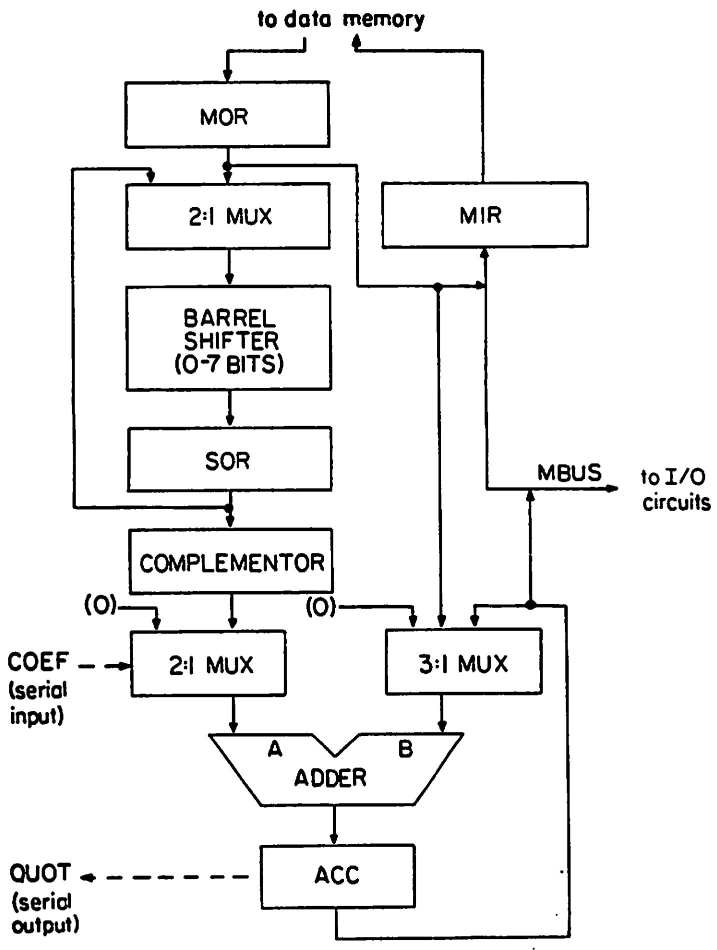

**Figure 5.5 A shifter-before-adder data path architecture (which was used in LagerI)**

$$y = x[0] + a_1 \times x[1] + a_2 \times x[2]$$

with $a_1 = 0.101$, $a_2 = 0.011$. Using the AU in KAPPA architecture, the FIR filter can be realized in the following code block:

```
r(x[0]);
r(x[1]),  acc=mor;                /* acc = x[0] */
          acc=mor, reg1=acc;
mor=r1, acc=mor+acc>2;
r(x[2]),  acc=mor+acc>1;          /* acc = x[0] + .101 * x[1] */
          acc=mor, reg1=acc;
mor=r1, acc=mor+acc>1;
          acc=mor+acc>2;          /* acc = x[0] + .101 * x[1] + .011 * x[2] */
w(y)=acc;
```

Because the shifter shifts the contents of the accumulator, the *partial sum* has to be moved to the register file (in this example, reg1) before the next multiplication is performed. This in general requires extra cycles. Moreover, during the series of shift-and-add's, this architecture may produce an intermediate result that is larger than the final result, which may create superfluous overflows and result in arithmetic errors. Special hardware has to be devoted[28] to correct the overflows in the intermediate results. On the other hand, using an AU[3] (Figure 5.5) with the shifter before the adder, the FIR filter can be realized in the following code block:

```
r(x[1]);
r(x[0]),  sor=mor>1;
r(x[2]),  sor=sor>2,     acc=mor+sor;
          sor=mor>2,     acc=acc+sor;    /* acc = x[0] + .101 * x[1] */
          sor=sor>1,     acc=acc+sor;
                         acc=acc+sor;    /* acc = x[0] + .101 * x[1] + .011 * x[2] */
w(y)=acc;
```

This realization uses 2 less cycles because the accumulator does not have to be reset. Also, the intermediate values during the shift-and-add operations are always less than the final result. However, this approach is proven to be numerically inferior. It results in $\frac{N}{2}$ inaccurate bits after N shift-and-add operations while the KAPPA architecture always produce a $\frac{1}{2}$ bit error regardless of the number of shift-and-add operations. Therefore, for the same numerical performance, this architecture requires a wider word length. The trade-off between the two architectures is the size of the arithmetic unit versus the size of the control store (or code size).

For simplicity and ease of comparison, the above code blocks only show the multiplication and summation part of the FIR filter. It assumes the x array is stored in the data memory. A complete implementation also needs to deal with the updating of the x array, i.e. moving input data to x[0], x[0] to x[1], x[1] to x[2], and so on. This requires a lot of data memory access and will limit the code efficiency. Further investigation shows that the code size difference between the two architectures is less significant in the complete implementation. Implementations of a number of other more complicated algorithms also show that the code size of the shifter-before-adder architecture is only slightly smaller the that of the shifter-after-adder architecture.

In the following sections, the software part of the LagerIII silicon compilation subsystem will be discussed. The challenges are, first of all, that KAPPA is not a conventional architecture and hence new compiler techniques have to be developed to map the high level behavioral description to the KAPPA architecture. Furthermore, the compiler has to be designed with the possibility of modifying the target architecture in mind, to allow the improvement of the architectural design through iteration. The discussion will follow the actual software flow, that is, (1) silage translator, (2) RL compiler and (3) control generator.

## 5.3. Silage Translator

The Silage Translator is the work of Edward Wang.[26]

*Silage*[8] is the highest level behavioral description language in the LagerIII silicon compilation subsystem. Silage is a *functional* (or *applicative*) language that differs from conventional *procedural* (or *imperative*) languages in that the user does not have to program the control flow of the algorithm. A functional language has two advantages: (1) side-effect-free functions. Since a functional language program consists only of function definitions and function applications, library functions (e.g. filters) can be easily combined. (2) no *over-specificity*. The lack of explicit control flow specifications in the functional language allows more freedom in exploiting the concurrency in the algorithm.

A number of features of the Silage language facilitates its use in signal processing applications. Delays are supported as a language construct. Therefore, the user does not need to explicitly allocate an array for all the past values of a variable, and update the array in every sample. *Decimation* and *interpolation* are supported as library functions. The Silage language provides the following data types: integer, boolean and *fixed-point*. The fixed-point data type is especially useful in signal processing applications, where expensive floating-point hardware is often unnecessary.

A Silage program describing a 16-tap FIR filter is shown in Figure 5.6. The *lwb()* and *upb()* functions return the lower and upper bounds of an array, respectively. The in@i notation denotes the *i*th delay of the *in* variable.

```
#define word fix<8>

coefs = [word(5/128), 7/128, 8/128, 9/128, 12/128, 16/128, 27/128, 81/128,
        -81/128, -27/128, -16/128, -12/128, -9/128, -8/128, -7/128, -5/128];

func main(in: word): word =
begin
        s[upb(coefs) + 1] = 0;
        (i: lwb(coefs) .. upb(coefs)) :: s[i] = s[i + 1] + in@i * coefs[i];
        return = s[lwb(coefs)];
end;
```

**Figure 5.6 A FIR filter in Silage**

The *Silage translator* consists of two parts: a *syntactic analysis* part and a *semantic analysis* part. The syntactic analyzer parses the Silage program and converts it into Lisp s-expressions. The semantic analyzer is organized in nine phases, which are described below. In phase 0, transformations are performed to *canonicalize* the Silage input. For example, the definition

```
coefs = [word(5/128), 7/128, 8/128, ...];
```

is transformed into

```
coefs[0] = word(5/128);
coefs[1] = 7/128;
coefs[2] = 8/128;
```

In phase 1, name references are resolved: uses of variable and function names are matched to corresponding definitions. In phase 2, *manifest* expressions (whose values can be calculated at compile time) are recognized. Expressions that must be manifest (e.g. iterator bounds) are checked. In phase 3, the type of each variable is determined. Since Silage has no type declarations other than those of function parameters, the type of a variable is determined by its definition. If a variable is manifest, its value is also calculated in phase 3. Phase 4 handles decimation and interpolation functions such that the *data rate* of every variable is determined. In phase 5, iterated definitions are expanded, and manifest expressions are replaced by their values. In this phase, a signal flow graph is generated. In phase 6 (optional), *loop folding* is performed to reduce code size by creating loops out of repeated parts of the signal flow graph. In phase 7, imperative code is generated from the signal flow graph. Finally in phase 8, a program in RL language syntax is generated.

## 5.4. RL Compiler

The RL Compiler is the work of Ken Rimey.[27]

The RL language[27] is a procedural language that is an *extended subset* of the C language. In addition, two major extensions are made. First, a fixed-point data type is added. Second, a declaration syntax is provided for specifying which register bank a variable is stored in. The 16-tap FIR filter programmed in RL is shown in Figure 5.7.

The first task of the RL compiler is to separate the control flow operations (e.g. branch, loop) from data flow operations (e.g. add, multiply) in the RL program. The data flow operations are grouped into *straight-line* code blocks, which contain no control flow operations. The entire program is composed of a number of straight-line code blocks and a number of control flow operations that control the execution flow of the straight-line code blocks. Each straight-line code block corresponds to a *state* of the control unit. At the end of each straight-line code block, a control flow operation is executed to change the state to another straight-line code block.

The main task of the RL compiler is to compile straight-line code blocks of RL code into code blocks of primitive instructions of the target architecture. To enable the compiler to do this for different target architectures (so that it is *retargetable*), an abstract description of the target architecture is required. The description contains three kinds of definitions: *define-register*, *define-move* and *define-operation*. A register bank or a bus can be defined by the *define-register* definition with the following arguments. The *capacity* (default value = 1) gives the number of registers in a register bank. The *delay* argument is used to differentiate a register and a bus. A register has delay 1 and a bus has delay 0.

Data flow operations can be divided into two classes: *transfers* that move data from one place to the other and *computations* that transform data, cause side-effect or move data in a data-dependent manner. A *define-move* definition describes a *transfer* operation that moves data from some source to some destination. The source and destination can each be either a bus or a register. The delay of the transfer is defined to be the delay of the destination. The define-move definition also describes how the transfer is performed by one or more primitive instructions (see section 5.2). A *define-operation* definition describes a *computation* operation by describing a sequence of primitive instructions that implement the computation operation. The define-move and define-operation functions constitute the *code generation tables.*

```
#pragma mult_hardware
#pragma word_length 8

#define N 16

fix c[N] = { 5/128, 7/128, 1/16, 9/128, 3/32, 1/8, 27/128, 81/128,
            -81/128, -27/128, -1/8, -3/32, -9/128, -1/16, -7/128, -5/128 };

fix a[17];

void init()
{
 register int i;

 for (i = 0; i < N; i++)
   a[i] = 0;
}

void loop()
{
 register fix total;
 register int i;

 a[N] = in();
 total = 0;
 for (i = 0; i < N; i++) {
   total += c[i] * (a[i] = a[i + 1]);
 }
 out(total);
}
```

**Figure 5.7 A FIR filter in RL**

A scheduling algorithm is implemented to compact the primitive instructions corresponding to the transfer and computation operations in the straight-line code block. The goal is to allocate registers to the computation operations and route data using the transfer operations in an optimal way. Register allocation has been extensively studied in conventional compiler design, but little has been done for pipeline registers. Furthermore, the KAPPA architecture has *volatile* registers which can hold data for only one cycle. These complexities make the design of the compiler challenging.

The RL compiler also transforms the control flow operations in the RL program into three primitive control instructions: *branch*, *subroutine call* and *return* (from subroutine). The control unit of the KAPPA architecture can perform *multiway* control instructions (e.g. multiway branch). Therefore, optimization is performed in the RL compiler to take advantage of this.

## 5.5. Control Generator

The input to the *control generator* (or the output of the RL compiler) is called a *rass* program. A rass program includes the description of an algorithm in primitive data and control instructions, and some hardware information (e.g. the word length of the data path). For a detailed description of the rass language and the Control Generator program, the reader is referred to the user manual in Appendix A. A rass file consists of six sections: (1) variable declarations, (2) constant declarations, (3) logic instruction declarations, (4) control flow (5) code blocks of primitive instructions and (6) hardware information. The variable declarations are given by the form

(**ram** *scalar1 scalar2 ... array1 array2 ...*)

which declares all the local variables used. A local variable can be either a scalar or an array with constant dimension. The constants can be declared by the form

(**const** *init-scalar ... init-array ...*)

in which *init-scalar* is a scalar with an initial value and *init-array* is an array with an initial values for every element in the array. The logic instructions are declared by the form

(**dfsm** <logic-inst> <logic-inst> ...)

In a primitive instruction code block, a logic instruction can be invoked by referring to its instruction name which is defined in the logic instruction declaration section. The control flow of the algorithm is specified by the form

(**cfsm** <state-trans> <state-trans> ...)

<state-trans> = (*state-name block-number* <cond> <control>)

<control> = (**goto** *state-name*) | (**call** *state-name state-name*) | (**return**)

which is organized as a set of state transitions. Each state transition is composed of a present state name, a corresponding code block number, an optional condition and a control instruction. The control instruction can be one of **goto**, **call** and **return**. The **call** instruction requires two arguments: a subroutine state name and a return state name. A multiway control operation can easily be described by a number of state transitions with the same present state name and different (complementary) conditions. The primitive instruction code blocks is described by the form

**(rom** <block0> <block1> ...)

<block*i*> = (**block*i*** <u-inst> <u-inst> ...)

<u-inst> = (<u-op> <u-op> ...) | (logic-inst-name <u-op> <u-op> ...) Each code blocks consists of a number of *micro instructions* (<u-inst>). Each micro-instruction consists of a number of primitive instructions and at most one logic instruction defined in the logic instruction declaration section. Last, the hardware information contains the following forms:

**(dp_word_size** *value*)

**(reset_timer** *list*)

**(max_sample_intvl** *value*)

**(stack_depth** *value*)

**(loop_test** *list*)

**Dp_word_size** specifies the word length of the data path. **Reset_timer** specifies a list of *block numbers*, each of which specifies a primitive instruction code block. In each code block that is specified by the **reset_timer** list the *timer* is reset. **Max_sample_intvl** specifies the worst case sample interval that is used by the *timer* to produce a constant sampling period. **Stack_depth** specifies the maximum *depth* of the subroutine nesting. If no subroutine is used, then **stack_depth** is zero. **Loop_test** specifies a list of *loop counts* that are used by the *loop counter* of the control unit.

For example, the rass program for the 16-tap FIR filter is shown in Figure 5.8.

In generating the parameter file from the rass file, the control generator has to know two pieces of information. The first information is the control signal names involved in each primitive instruction. The

```
(ram (v95 17) (a85 17))

(const (c107 16 (-5 -7 -8 -9 -12 -16 -27 -81 81 27 16 12 9 8 7 5)))

(dfsm (0 (_bc _xsign)))

(cfsm (0 0 nil (goto 1)) (1 1 _bc (goto 1)) (1 1 (not _bc) (goto 2))
    (2 2 nil (goto 0)))

(dp_word_size 8)

(stack_depth 0)

(reset_timer nil)

(max_sample_intvl 1)

(rom (0
    ((r*=rbus 0) (rbus=ioport) (x*=eabus 0) (addr 0) (acc=0)
     (ioport=extport 0))
    ((mbus=r* 0) (r*=rbus 0) (rbus=acc) (addr a85) (offset 16)
     (mem=mbus))
    ((mbus=r* 0) (addr v95) (mem=mbus)))
    (1 ((xbus=x* 0) (addr a85) (offset 1) (mor=mem))
     ((xbus=x* 0) (mbus=mor) (addr a85) (mem=mbus))
     ((xbus=x* 0) (addr c107) (mor=mem))
     ((xbus=x* 0) (r*=rbus 0) (rbus=ioport) (ioport=mbus) (mbus=mor)
      (addr v95) (mor=mem))
     ((xbus=x* 0) (r*=rbus 1) (rbus=ioport) (ioport=mbus) (mbus=mor)
      (addr a85) (mor=mem))
     ((mbus=mor) (x*=eabus 1) (addr 0) (xbus=x* 0) (rcoef=mbus))
     ((mor=mbus) (mbus=r* 0) (xbus=x* 1) (x*=eabus 0) (addr 1))
     ((fsm 0) (xbus=x* 0) (addr -16) (nosat) (acc=abus)
      (abus=coef.mor) (shrcoef))
     ((nosat) (acc=sum) (abus=coef.mor) (shrcoef) (bbus=acc>* 1))
     ((nosat) (acc=sum) (abus=coef.mor) (shrcoef) (bbus=acc>* 1))
     ((nosat) (acc=sum) (abus=coef.mor) (shrcoef) (bbus=acc>* 1))
     ((nosat) (acc=sum) (abus=coef.mor) (shrcoef) (bbus=acc>* 1))
     ((nosat) (acc=sum) (abus=coef.mor) (shrcoef) (bbus=acc>* 1))
     ((nosat) (acc=sum) (abus=coef.mor) (shrcoef) (bbus=acc>* 1))
     ((mor=mbus) (mbus=r* 1) (acc=sum) (abus=coef.-mor) (shrcoef)
      (bbus=acc>* 1))
     ((bbus=acc>* 0) (abus=mor) (acc=sum))
     ((xbus=x* 1) (mbus=acc) (addr v95) (offset 1) (mem=mbus)))
    (2 ((addr v95) (offset 16) (mor=mem))
     ((ioport=mbus) (mbus=mor) (extport=ioport 0))))
```

**Figure 5.8 A FIR filter in rass language**

second information is the architecture of the control unit. The control signal information is organized in a so-called *sadl* file, which is read in by the control generator at run-time. In addition, the set of registers and buses as well as the set of the primitive instructions are also defined in sadl file. Once the architecture is modified, and new primitive instructions are defined, the sadl file can be modified accordingly to make the control generator program unaffected. The sadl file for the KAPPA architecture is shown in Appendix C.

The three primitive control operations, branch, call and return, can be implemented in a number of different control unit architectures. Therefore, the control generator has to know the particular control unit architecture that the control operations are mapped into. Ideally, the control unit architecture should be described externally, like the sadl file, such that the same control generator program can be applied to different control unit architectures. The current control generator, however, hard-wires the KAPPA control unit architecture in its implementation.

The algorithm of the control generator is described in the following. First, the hardware information section is passed to the output without modification. For the variable and constant declarations in the rass program, a *hash table* is created to store the variable or constant names and values. Later, when a variable or constant name is used as a key then its address can be obtained from the hash table. For the logic instruction declarations and the control flow sections, the boolean variables involved are analyzed to determine each of which to be one of the *primary inputs*, *primary outputs* or *feedbacks*. The logic instructions will be implemented by the LGU and the control flow will be implemented by the finite state machine in the control unit. The bit order of the primary inputs, primary outputs and feedbacks has to be consistent with the structural description of the control unit. The primitive instructions in the code block section are translate to the control signal patterns by the help of the sadl file. The control generator will check whether the primitive instructions in the same cycle have any conflicts. Finally, the sizing parameters of each of the functional blocks in the architecture are determined, and an output parameter value file is generated.

## 5.6. Summary

The software flow of the LagerIII silicon compilation subsystem is summarized in Figure 5.9. It consists of three components: a Silage translator that translates a Silage program to an RL program, an RL compiler that compiles the RL program to a rass program and a control generator program that generates a parameter value file from the rass program. This subsystem maps a Silage program to a pre-defined architecture, such as KAPPA. It provides a means for the user to tailor the architecture and create new primitive instructions to match the algorithm better. In particular, a code generation table is used by the RL compiler, and a sadl file is used by the control generator, to allow customization of the subsystem.

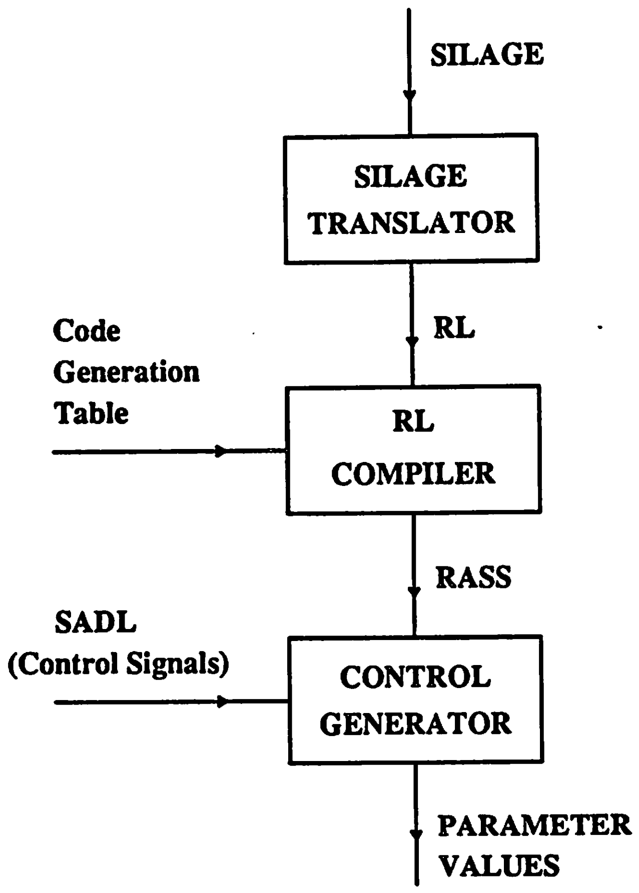

Figure 5.9 The software flow of LagerIII silicon compilation subsystem

The Cathedral-II system[5] has been developed in parallel with the LagerIII silicon compilation subsystem. Both of these systems use Silage as a behavioral description language for describing the algorithms. There are two major differences between LagerIII and Cathedral-II in their silicon compilation parts. First, in LagerIII the system is adapted to new architecture by modifying two tables: the code generation table used by the RL Compiler and the sadl file used by the Control Generator, and in Cathedral-II it is adapted by modifying a set of hardware-dependent *rules*. Because the *unification* process in a rule-based system searches the set of rules sequentially, the order of the rules is critical. Thus, the insertion and deletion of rules, as required to modify the architecture, is hard to do. Second, the LagerIII system schedules the Silage program to generate the control flow operations (using the Silage translator) before the compilation of the data path operations. On the other hand, the Cathedral-II system first compiles the Silage program into symbolic microcode which does not have absolute timing, and subsequently schedules the microcode to generate the control flow.

In addition, the LagerIII system has a unique structural interface to allow quick layout generation of new architectural designs.

# CHAPTER 6

# Pitch Tracker Chip

In this chapter, a pitch tracker chip design is used as an example of the use of the LagerIII silicon compilation subsystem.

## 6.1. Pitch Tracking Algorithm

The pitch tracking algorithm used is referred to as the second modification of the Gold pitch tracker[33] , a summary of which can also be found in[3] . In this section, a brief description of the algorithm is provided.

The algorithm is shown in Figure 6.1. The *pitch* is the fundamental period in a speech signal. The pitch is a low frequency signal and hence a low pass filter (LPF) is used to filter out the high frequency components in the speech signals. However, harmonics of the fundamental may also pass through the LPF. The goal of the algorithm is to reject the harmonics and extract the fundamental period. The peaks and valleys of the filtered speech signal are then found, and the six new signals that are combinations of present peaks or valleys and previous peaks or valleys (Figure 6.2) are calculated. The six signals are sent to six identical pitch detectors.

A *pitch detector* forms an estimate of the time interval between major peaks in their input. The minor peaks are rejected based on the following algorithm. First, after a major peak is detected, a *blanking period* of $3ms$ duration is entered during which all peaks are rejected. In other words, the pitch period cannot be shorter than $3ms$. Following the blanking interval, an exponentially-decaying threshold signal is computed. This threshold is initialized with the amplitude of the previous peak, and decay with a time constant of $5ms$. Minor peaks which fail to exceed this threshold are rejected.

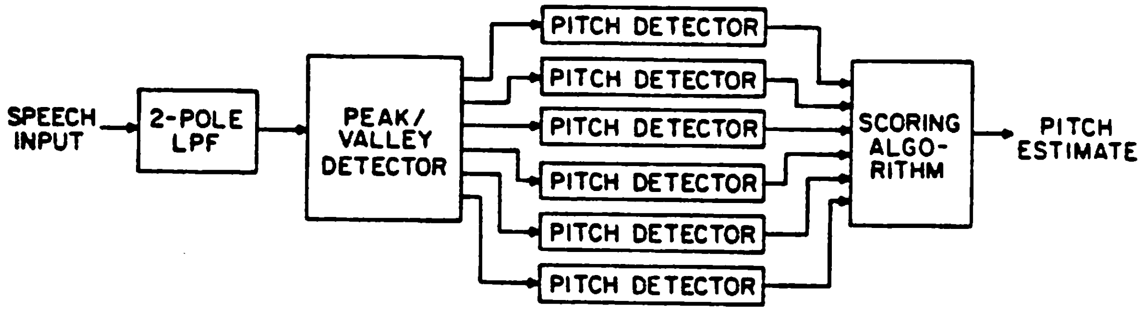

**Figure 6.1 Gold pitch tracker algorithm**

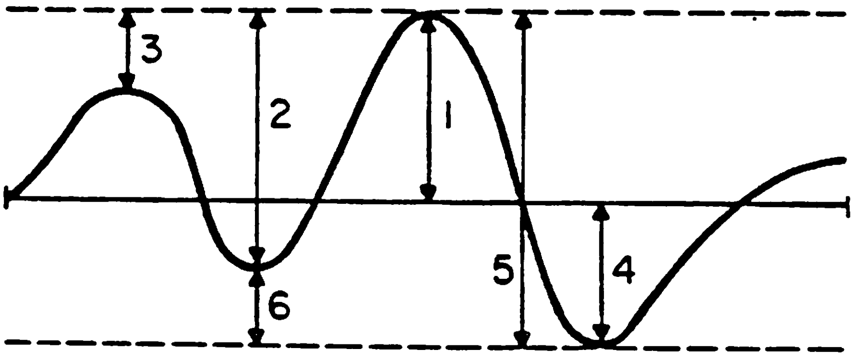

**Figure 6.2 Six signals formed after peak-valley detection**

The six pitch detectors are working in parallel and six pitch estimates are obtained. Each estimate is considered as a candidate for possibly being the actual pitch period. A *scoring* algorithm is used to select one of the six as the best estimate. Each candidate is given a score ranging from one to eighteen by performing a *window comparison* between the candidate and each of the following eighteen values: the six candidates themselves, the six previous estimates and the six sums of the current and previous estimates, where a window comparison is to compare if two values are within a pre-defined window (distance). If none of the estimate has a score greater than a fixed threshold then the input speech is considered as *unvoiced.*

## 6.2. Chip Implementation

To use the LagerIII silicon compilation subsystem, the algorithm is programmed in Silage. At the development phase of the silicon compilation subsystem, RL and rass programs are also written manually and compared with the ones produced by the Silage translator and RL compiler. In particular, the code size generated by the RL compiler is about 8% longer than that of the manually written rass program. The Silage, RL and rass programs for the pitch tracking algorithm are shown in the Appendix D.

The KAPPA architecture is used to implement the pitch tracking algorithm. The sampling rate of the speech signal is 8 KHz. The circuit clock rate of the KAPPA architecture is 5 MHz. Therefore, the maximum number of cycles in a sample is 625. For the pitch tracking algorithm, the total number of cycles is 310 for the manually written rass program and 335 for the rass program generated by the RL compiler. The KAPPA architecture provides a sufficient implementation because the sampling rate can be met.

After applying the silicon compilation subsystem, a parameter value file is generated. By using this parameter value file and the set of structural description language files that describe the KAPPA architecture as inputs to the LagerIII silicon assembly subsystem, the chip layout can be generated automatically, which is shown in Figure 6.3.

In the layout generation process, the macro cell place and route tool, **Flint**, is used extensively. It has a nice feature that once the floorplanning and global routing are in place, then the detail routing can be performed very quickly. Therefore, by using the same architecture, the algorithm developer can reuse not only the architecture but also the existing floorplans. However, one drawback is that the floorplan files use the absolute names and hence slight modifications are necessary if the root instance name is changed. This should be improved in the future.

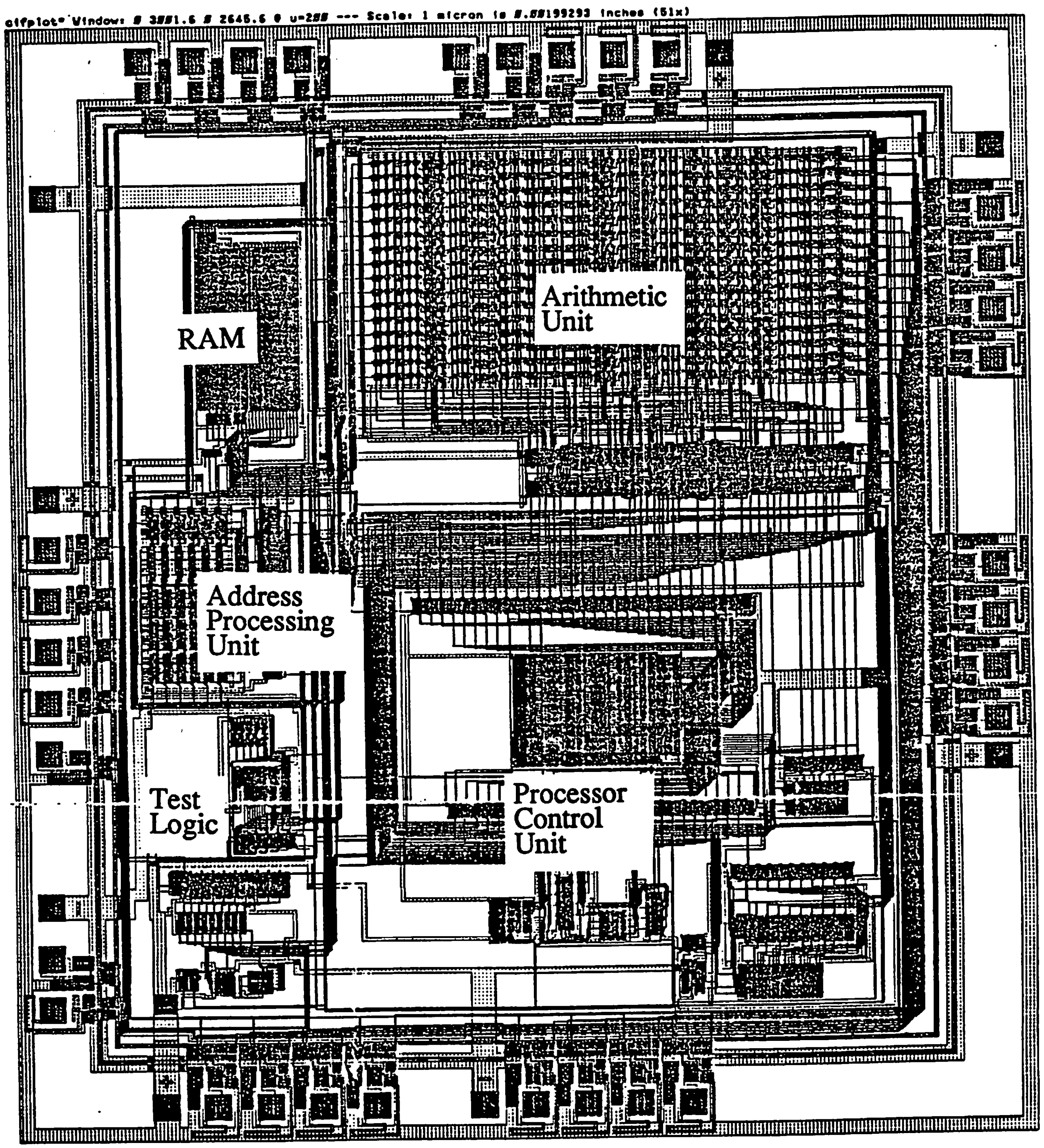
cifplot* Window: 0 3881.6 0 2645.6 @ u=200 --- Scale: 1 micron is 0.00199293 inches (51x)
RAM
Arithmetic Unit
Address Processing Unit
Test Logic
Processor Control Unit

**Figure 6.3 The pitch tracker chip CIF plot**

# CHAPTER 7

# Conclusions and Remarks

## 7.1. Major Accomplishments

One of the key accomplishments of the LagerIII system is that it provides four interfaces such that users with different expertise can work together and make use of other people's results. The collaboration has proven to be very important in the design of algorithm-specific ICs. A *behavioral* interface is provided for algorithm developers to enter new algorithms. A *structural* interface is provided for architecture designers to enter new architectures. A *CAD tool* interface is provided for CAD tool designers to enter new module generation tools. A *cell library* interface is provided for the circuit designers to enter new leaf cells.

Because the direct synthesis of a structural description from a behavioral description has not yet able to result in efficient architectures in a wide range of applications, the LagerIII silicon compilation subsystem employs a novel approach to map a behavioral description to a pre-defined parametrizable structural description by generating a set of parameter values that is tailored to the behavioral description. The capability of selecting the architecture and iteratively optimizing it is useful in gaining experience of architectural design for new algorithms. This experience is very valuable in the direct synthesis research.

The LagerIII silicon assembly subsystem contains a number of module generation tools, a simulator and a cell library. It is designed with emphasis on providing an open system such that new module generation tools and cells can be integrated easily. To do this, a database with a consistent integration policy is developed.

Even though the algorithms in various applications are different, it is found that they can all implemented by the same set of functional modules, which include memories, data paths, etc. The use of parametrizable functional modules is a unique feature of the LagerIII system. The parametrizability not only facilitates the reuse of functional modules, but also alleviate the problem of cell library maintenance.

## 7.2. Remarks on Future Improvements

Nevertheless, a few improvements can be made to make the LagerIII system better. These points are discussed in this section.

The LagerIII system employs a cell-based design methodology. All the leaf cells in the cell library are manually designed. One of reasons that this approach is taken is the relatively poor performance of the automatically generated leaf cells. However, there have been a number of new and promising approaches in the automatic generation of leaf cells, including standard cells with transistor sizing[34] , gate matrix[35] , and sea of gates. These approaches should be looked into and employed if advantageous to alleviate the problem of design rule dependence in the manually designed leaf cells.

The LagerIII silicon assembly subsystem is implemented in Lisp with the Flavors object-oriented programming system, which is very nice developing tools and helps to prototype of the system in a short amount of time. However, it suffers from speed penalty unless high performance, dedicated hardware is used. In addition, the capacity of a Lisp process in Unix environment limits the size of the design. The volatile database also contributes to the capacity problem.

A development project (which is dubbed LagerIV) is currently undertaken which re-implements the LagerIII silicon assembly subsystem in C language and replaces the Flavors based database by the OCT database. This should solve the capacity problem because OCT is a non-volatile database and hence the intermediate results can be saved in the secondary memory. A C based simulator, which is called Thor[36,37] , will be in place for the Lisp based Design Simulator. Besides speed improvements, the Thor simulator has two additional advantages. First, it is backed by a complete set of library models for off-the-shelf chips. This makes the simulation of the whole board possible. Second, an interface to a switch level simulator, **rsim**, which can be used to simulate the extracted layout, is provided. This helps to verify a circuit model against its layout.

Some believe that the automatic layout generation is *correct by construction*, and therefore no design verification. However, verification is still required at least in the development phase. Magic provides an incremental design rule checking environment which can be used to verify the design at the physical level. An interface is established between **Thor** and **rsim** to verify the cell model against the cell layout. After the cell models are verified, all further simulation can be carried out at the structural level using **Thor**. The verification between the behavioral level description and structural level description remains, however, an open research problem.

Due to the parametrization mechanism in the LagerIII system, special care has to be taken in the LagerIV project. First, a Lisp interpreter[38] is still required to evaluate the parameterization expression in the structural descriptions. Second, a pre-processor is required to generate the Thor model instances from the model templates which are parametrized.

The LagerIII silicon subsystem tries to generate the layout of all the cell modules, even with those which have the same parameter values and hence have the physical layout. This is also fixed in LagerIV by modifying the syntax and semantics of the structural description language.

In the LagerIII silicon compilation subsystem, one drawback is that it is left to the user to diagnosis the efficiency of a particular architecture and find the remedy. The subsystem provides user the number of instruction cycles that the algorithm will require if using the architecture, as the only figure of merits. Moreover, the user has to provide the code generation table as well as the sadl file at every iteration. Some architecture level design aids that facilitates the user in exploring the design space by quickly providing estimates of the architectural efficiency will be very helpful.

## 7.3. Applications of LagerIII

The LagerIII system has been applied to a number of algorithm-specific IC designs, two of which were reported in this thesis. In addition, a robot arm controller chip has been designed and generated using LagerIII.[28] It implements an adaptive control algorithm which compensates the inherent non-linearity in the robot arm dynamics. This chip was generated by both the silicon compilation and silicon assembly subsystems using the KAPPA architecture.

A set of four chips has been designed, fabricated and individually tested which is used for implementing a *channel emulator*.[39] A channel emulator emulates the architecture of a computer network and can be used to study the performance and effects of different network protocols. The four chips include a *tap switch* which emulates the physical connection of a computer to the network, a *variable register delay line* to simulate the time delays through different network paths, a *crossbar switch* to emulate the topology of the network and a *mask OR crossbar switch*. The mask OR crossbar switch is essentially similar to the crossbar switch with additional flexibility in the topology it can emulate. The four chips were generated using the LagerIII silicon assembly subsystem.

The LagerIII silicon assembly system was also applied to the development of VLSI implementation of projection-based image processing algorithms.[40] In all the applications, algorithm-specific ICs were found to produce much higher performance. By using the LagerIII system, the design difficulty and cost of algorithm-specific ICs are reduced, which further makes the algorithm-specific ICs approach more attractive.

## References

1. R. A. Kavaler, *The Design and Evaluation of A Speech Recognition System for Engineering Workstations,* University of California at Berkeley (May 1985). Ph. D. Thesis

2. P. A. Ruetz, *Architectures and Design Techniques for Real-Time Image Processing ICs,* University of California at Berkeley (May 1986). Ph. D. Thesis

3. S. P. Pope, *Automatic Generation of Signal Processing Integrated Circuits,* University of California at Berkeley (February 1985). Ph. D. Thesis

4. D. Thomas, C. Hitchcock III, T. Kowalski, J. Rajan, and R. Walker, "Automatic Data Path Synthesis," *IEEE Computer Magazine,* pp. 59-70 (December 1983).

5. J. Rabaey, H. De Man, J. Vanhoof, G. Goossens, and F. Catthoor, "Cathedeal-II: A Synthesis System for Multiprocessor DSP Systems," in *Silicon Compilation,* Addison-Wesley (December 1987).

6. P. A. Ruetz, R. Jain, and R. W. Brodersen, "Comparision of parallel architectures for real-time image processing ICs," *Proc. ISCAS,* (December 1987).

7. S. K. Azim, C-S Shung, and R. W. Brodersen, "Automatic Generation of A Custom Digital Signal Processor for An Adapter Robot Arm Controller," *IEEE ICASSP,* (April 1988).

8. P. N. Hilfinger, "Silage: A Language for Signal Processing," *Proceedings of CICC,* (May 1985).

9. A. Goldberg, S. Hirschhorn, and K. Lieberherr, "Approaches toward Silicon Compilation," *IEEE Circuits and Devices Magazine,* pp. 29-39 (May 1985).

10. J. R. Jasica, S. E. Noujaim, R. Hartley, and M. J. Hartman, "A Bit-Serial Silicon Compiler," *Proceedings of ICCAD,* pp. 91-93 (November 1985).

11. J. R. Southard, "MacPitts: An Approach to Silicon Compilation," *IEEE Computer Magazine,* pp. 74-82 (December 1983).

12. R. Jain, F. Catthoor, J. Vanhoof, B. De Loore, G. Goossens, N. Goncalvez, L. Claesen, J. Van 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. 21, No. 1,* pp. 73-85 (February 1986).

13. M. Barbacci, "Instruction Set Specification (ISPS): The Notation and its Applications," *IEEE Trans. Computers, Vol. 30, No. 1,* (January 1981).

14. J. Rabaey, S. Pope , and R. Brodersen, "An Integrated Automatic Layout Generation System for DSP Circuits," *IEEE Trans. Computer-aided Design, Vol. 4, No. 3,* (July 1985.).

15. Franz Inc., *Franz Lisp User Manual, Opus 43.1,* Franz Inc. (1987).

16. H. Cannon, *Flavors: A Non-Hierarchical Approach to Object-Oriented Programming,* MIT Artificial Intelligence Lab (). unpublished paper

17. A. Goldberg and D. Robinson, *Smalltalk-80: The Language and its Implementation,* Addison-Wesley (1984).

18. 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,* University of California at Berkeley, Electronics Research Lab (November 1987).

19. W. Scott, R. Mayo, G. Hamachi, and J. Ousterhout, editors, *1986 VLSI Tools,* University of California at Berkeley, Report No. UCB/CSD 86/272 (December 1985).

20. C-S Shung, R. Jain, M. B. Srivastava, and R. W. Brodersen, *LagerIII User's Manual,* Universilty of California at Berkeley, internal documentation (August 1987).

21. M. B. Srivastava, *Automatic Generation of CMOS Data Paths in LAGER Framework,* University of California at Berkeley (May 1987). M. S. Thesis

22. J. Perl and R. Chapman, "New Image-Processing Frame Memory," *Computer Graphics World,* pp. 73-75 (June 1986).

23. Texas Instruments, *TMS34061 User's Guide,* Texas Instruments (1986).

24. GE Company, *TN2250 512x512 CID Automation Camera Interface Specification,* GE Company (October 1986).

25. A. J. Burstein, *A 9 Bit 10 MHz A/D Macrocell*, University of California at Berkeley (November 1987). M. S. Thesis

26. E. Wang, *private communication*

27. K. Rimey, *A Compiler for Horizontal-Instruction-Word Signal Processors*, University of California at Berkeley, Qualifying Proposal (April 1988).

28. S. K. Azim, *Applications of Silicon Compilation Techniques to A Robot Controller Design*, University of California at Berkeley (in preparation). Ph. D. Thesis

29. Davio, Marc, and Deschamps, *Digital Systems with Algorithm Implementation*, John Wiley & Sons (1983).

30. G. Frantz, K. Lin, J. Reimer, and J. Bradley, "The Texas Instruments TMS320C25 Digital Signal Microcomputer," *IEEE Micro, Vol. 6, No. 6*, pp. 10-28 (December 1986).

31. Sieworek, Bell, and Newell, *Computer Structures: Principles and Examples*, McGraw Hill (1982).

32. D. A. Patterson, "Reduced Instruction Set Computers," *Communications of the ACM, Vol. 28, No. 1*, pp. 8-21 (January 1985).

33. B. Gold and L. R. Rabiner, "Parallel Processing Techniques for Estimating Pitch Periods of Speech in the Time Domain," *J. Acoustical Society of America, V. 34, No. 7*, pp. 916-921 (1962).

34. J-M Shyu, A. Sangiovanni-Vincentelli, J. Fishburn, and A. Dunlop, "Optimization-Based Transistor Sizing," *IEEE J. Solid-State Circuits*, (April 1988).

35. A. D. Lopez and H. A. Law, "Design Gate Matrix Layout Method for MOS VLSI," *IEEE Trans. on Electronic Devices, vol. ED-27*, pp. 1671-1675 (August 1980).

36. Robert Alverson, Tom Blank, Kiyoung Choi, Arturo Salz, Larry Soule, and Thomas Rokicki, *THOR User's Manual: Tutorial and Commands*, Stanford University (January 1988). Technical Report CSL-TR-88-348

37. Robert Alverson, Tom Blank, Kiyoung Choi, Arturo Salz, Larry Soule, and Thomas Rokicki, *THOR User's Manual: Library Functions*, Stanford University (January 1988). Technical Report CSL-TR-

88-349

38. W. C. Baker, *LightLisp: A Light-Weight Lisp Interpreter*, University of California at Berkeley, Electronic Research Lab (November 1987).

39. J. S. Sun, *Design and Implementation of Integrated Circuits for a Real-time Flexible Emulator Applying Silicon Assembly Tools*, University of California at Berkeley (March 1988). M. S. Thesis

40. W. B. Baringer, B. C. Richards, and R. W. Brodersen, "A VLSI Implementation of PPPE for Real-Time Image Processing in Radon Space - Work in Progress," *Workshop on Computer Architectures for Pattern Analysis and Machine Intelligence*, pp. 88-93 (October 1987).

# APPENDIX A

## (part of) LagerIII User Manuals

1. Design Manager (DM)

2. Layout Generator (LG)

3. Design Simulator (DSIM)

4. Control Generator (rassCG)

## NAME

*DM_new – Design Manager*

## SYNOPSIS

*DM [options]*

## DESCRIPTION

*DM_new* is a structural interface program which serves as the front-end of the layout generation and the simulation. *DM_new* requires the following information: *sdl* files, parameter values of all the parameters defined in the *sdl* files and a list of *options*. Since the names of the *sdl* files of the sub-cells can be inferred from the *sdl* file of the parent cell, only the name of the *sdl* file of the *root* of the design hierarchy is needed. The parameter values can be provided by the user through a file, or interactively for any parameter values not specified in the file.

*DM_new* can be run in both an *interactive* and a *batch* mode, depending on whether a command-line option -I is set or not. In the interactive mode, the Lisp prompt (=> for *Franz* Lisp 43.1) will appear. The user can then invoke the *DM* Lisp function by

```
=> (DM)
```

*(DM)* will ask a series of questions:

```
Please enter root type (generic name) : rom
Please enter root name (instance name) : ROM1
Please enter parameter file name (if none enter N) : N
Please enter parameter value of row in cell rom (root) : 12
```

The *generic* name of the *root* is the name of the *root sdl* file name without the ".sdl" suffix. The *instance* name of the *root* is a name for the particular design. The user has the option of entering the parameter values through a file, or interactively as shown in the above queries. The parameter file may contain any number of lists, each of which has two elements. The first element is the name of the parameter and the second is the value of the parameter. Because the parameters of the sub-cells can be defined by the parameters of the parent cell, only the parameter values of the *root* need to be specified. Those parameter values that the user enters interactively are collectly and incorporated into the parameter file such that the user doesn't have to type them in again next time. If there is no parameter file specified at the first time, the file *root-instance-name*.par is used.

If the *(DM)* finishes creating the structures, the user will be prompted to enter (LG) or (DSIM) depending on whether he/she wants to generate the layout or simulate the design. If *(DM)* aborts because some of the *sdl* files in the design hierarchy are not found by *(DM)*, see the PATH MECHANISM section. Otherwise, *(DM)* should report a number of *DM-ERROR*s, see the ERROR DIAGNOSIS section.

## PATH MECHANISM

In LAGER-III all the tools have a joint startup file called *.lager* that should be in the working directory or home directory. If both locations have a .lager file, the one in the working directory prevails. The user may use the .lager file to specify *where* the library files for any tools can be found.

The .lager file may consist of any number of lists, each of which takes the form

```
(keyword element [elements ...])
```

where the *keyword* specifies an attribute of a tool that makes use of the list. Normally each *element* in the list is a directory path (absolute or relative). Note that the order determines the priority. For example, *DM* has only one attribute *dm.sdlfile* in the .lager file. The user can use the .lager file to specify the directory paths that DM needs to search other than the (default) working directory. The working directory has to be represented by \. (back-slash and a dot, for some lispish reasons) or a full absolute path. Comments in the .lager file can be entered in the same way as in the sdl file.

A .lager file example:

```
(dm.sdlfile      \
    ~lager/LagerIII/processor/sdl
    ~lager/LagerIII/lib/stdcell
    ~lager/LagerIII/lib/dpc/leafcells)
(TimLager.o ~lager/LagerIII/lib/TimLager/scpads/scpads1.25)
(TimLager.leafcells ~lager/LagerIII/lib/TimLager/scpads/scpads1.25/leafcells)
(DPC.cd ~lager/LagerIII/lib/dpc/leafcells)
(DPC.mag ~lager/LagerIII/lib/dpc/leafcells)
(bin ~lager/LagerIII/bin)
(octbin ~cad/bin /usr5/octtools/bin)
(stdcell.leafcell ~lager/LagerIII/lib/stdcell)
(Padroute.hdl ~bilbo/moslib/frames)
```

## OPTIONS

**I** -- to invoke interactive mode DM

The following options are for batch mode DM

**ganme** *name* -- the root generic name (without the .sdl extension)

**iname** *name* -- the root instance name

**pfile** *filename* -- parameter file name

**c** -- scmos technology (default)

**n** -- nmos technology

**m** -- magic layout (default)

**k** -- kic layout

**hdl** -- use hdl format for Padroute and Flint (default is oct format)

**LG** -- run LG automatically following DM

**DSIM** -- run DSIM automatically following DM

**efile** *filename* -- the event file input of DSIM

**ofile** *filename* -- the output file of DSIM

## LOG FILES

There are 3 *log* files to bookkeep the program status. Their names have the *instance* name of the *root* as a prefix. The "-dm.log" file is the *log* file for *DM(1)*. The "-sim.log" file is for *DSIM(1)* and "-lg.log" is for *LG(1)*.

An *instance-name*-dm.log file contains the following information: the paths of the *sdl* files if they are found through .lager file, the number of nets generated for each net definition, all instances of parameter evaluation and warnings such as to default the starting index to zero. The information is organized in a top-down way, the parent cell being in front of the sub-cells.

## NAMING CONVENTION

Each cell, net and terminal in the entire design hierarchy has a unique name. These names are used all the *log* files. The naming conventions are:

1. The name of the *root* is the *root instance name* entered interactively by the user when (DM) is invoked.

2. The name of a sub-cell is the concatenation of the name of the parent cell, the separator "-" and the *instance-name* of the sub-cell.

3. The name of a net is the concatenation of the name of the parent cell, the separator "@" and the net name.

4. The name of a terminal is the concatenation of the name of the parent cell, the separator "." and the name of the terminal.

5. In the case of a bus, the net name and the terminal name are indexed by the bit number (starting from 0) enclosed in brackets ("[" and "]").

## ERROR DIAGNOSIS

An important task of DM is to find syntax errors and inconsistencies in the *sdl* files that the user provides. The errors will be reported in the *instance-name*-dm.log file starting with "DM-ERROR". If there is any DM-ERRORs found, then the number of DM-ERRORs will be reported after (DM) finishes. The common error messages are:

1. "*param* not evaluated to a number, net ignored"

The parameter *param* specifies the *width* of a net definition but the value of *param* is not a number.

2. "terminal *term* connects to net *neta* and *netb*."

The terminal *term* appears on two nets, *neta* and *netb*.

3. "net name *neta* repeated, 2nd one deleted"

The same net name *neta* is used to referred to two nets. The second net definition is removed.

4. "instance terminal *term* in cell *cell* not defined as a generic terminal"

The terminal *term* of the sub-cell *cell* is instantiated; however, in the *sdl* file corresponding to *cell* (*generic-name*.sdl), the terminal *term* is not defined as a generic terminal.

5. "Error: Can't find *generic-name*.sdl"

The *generic-name*.sdl is not in the working directory, nor can it be found from the paths of the *dm.sdlfile* entry in the .lager file. This is a fatal error and DM will halt.

6. "*generic-name*.sdl: too many parentheses?" "after form *form*"

Lisp read fails. Usually the error is because of unmatched left and right parentheses. DM will show you what the last successful Lisp form (*form*) read.

7. "*generic-name*.sdl: unrecognized keyword *word*" "after form *form*"

DM recognized different constructs by keywords, see *sdl(5)*. This error reports an unrecognized keyword *word* found after the last successfully read *form*.

## SEE ALSO

sdl(5), DSIM(1), LG(1)

## FILES

~lager/LagerIII/src/DM/DMinit.l

~lager/LagerIII/lisplib/{*.l,*.o}

~lager/LagerIII/bin/DM

## AUTHORS

Chuen-Shen Shung

Rajeev Jain

## NAME

*LG – Layout Generators*

## SYNOPSIS

*(LG)*

## DESCRIPTION

(LG) is one of the two back-ends of (DM). Before (LG) can be run, the user has to run (DM). (DM) creates the necessary data structures for (LG) to perform layout generation. The layout generation is performed in a *bottom-up* way. The layouts of the sub-cells are generated before the layout of the parent cell is generated. This is because that the layout generation of the parent cell usually requires some information (e.g. size, terminal locations) of the sub-cells.

(LG) serves as a centralized data manager. Instead of letting module generators communicate with each other directly, each module generator communicates with (LG) only. Before the layout generation is performed, the module generator gets the information of the sub-cells from (LG); after the layout generation is done, the module generators sends the result to (LG) for use of the parent cell. New module generators can be integrated by simply interfacing with (LG), which has no effect on the existing module generators.

The entire layout generation is pseudo automatic because some of the layout generators require human interaction. Ideally if all the module generators were written in Lisp, then they can access the data structures directly. However, most tools are in C, and were designed to use files as input and output, hence (LG) has to create the input files and to read back in the output files. These layout generator oriented issues are discussed in the LAYOUT GENERATORS section.

(LG) creates a *instance-name*-lg.log file to record the status of the layout generation. (LG) reports only one kind of LG-ERROR which indicates that some terminals of the sub-cells are not connected by the parent cell.

## LAYOUT GENERATORS

This section describes the input and output files/directories and the necessary user interactions of the layout generators. In case of any problems during the layout generation, the user can consult the input and output files/directories in addition to the *log* file to troubleshoot.

1. Flint

**Flint** assumes a *directory* as the input. The name of the *directory* is the *instance* name of the parent cell. Each sub-cell occupies a *sub-directory* which contains a *pdl* file and a *hdl* file. The time stamp of the *hdl* file has to be later than that of the *pdl* file to ensure proper operation. The parent cell also requires a *pdl* file and a *con* file that contains a list of the sub-cell names.

The output of **Flint** includes a *hdl* file which shows that size of the parent cell and the coordinates of the terminals of the parent cell, and a layout file. In the *hdl* file that **Flint** generates, the names of the terminals are the instance names of the sub-cells. (LG) changes the names into the generic names of the parent cell in order to be consistent with the higher level.

**Flint** requires the user to place the cells, to define the channels and to do the global routing manually. However, these information can be dumped into a *floorplan* file that can be used later. **Flint** does the detail routing automatically, except that it fails for the nets which contains nothing but two terminals on the same cell (feedback nets). In general, some kind of *rip-up* routing has to be done manually as well.

2. TimLager

**TimLager** assumes a *pdl* file as the input. When **TimLager** is invoked by (LG), the *net* and the *cable* fields in the *pdl* file are not used. **TimLager** generates a layout file and a *ihdl* file (used with (LG)) or a *hdl* file (used stand-alone). The only difference between the *ihdl* file and the *hdl* file is that the *hdl* file copies the *net* and the *cable* fields from the input *pdl* file and the *ihdl* file doesn't.

3. Stdcell

The standard cell module generator is called Wolfe. Wolfe takes the *contents* facet of a symbolic view (the view name is designated to be "wolfe" by (LG)) of the parent cell as the input. After the layout generation, Wolfe writes the information back into the same *contents* facet, and creates a layout file. The OCT facet is not in ascii format, hence the user has to use **vem** or **attache** to browse it.

The user will be asked of two questions:

> Enter the number of rows desired (n => decided by wolfe) : **2**
>
> Enter number of iterations of sim. anneal. desired (n => 100) : **5**

The first question should be self-explanatory. The placement step in **Wolfe** uses a technique called *simulated annealing*. The second question asks the user to specify the number of *simulated anealing* iterations. The larger the number, the slower the program runs and the better the result.

**Wolfe** doesn't extend the Vdd and GND terminals to the boundary of the parent cell. (LG) provides a kludgy way to get around the problem. After the layout is generated, (LG) will interrupt the running process and ask the user to fix the layout file. The user can resume the process by

```
c{1} ?ret
```

where c{1} is the Lisp *break* prompt.

4. Dpc

**Dpc** is written in Lisp such that it can access the data structures directly. **Dpc** generates a layout file and a *hdl* file.

5. Padroute

**Padroute** takes 5 hdl files as input, which are created automatically by (LG). **Padroute** generates a layout file. Usually **Padroute** is used as the module generator of the *root*, so no terminal is present on the boundary.

6. Mosiaco

Like **Wolfe**, **Mosaico** takes the *contents* facet of a symbolic view (the view name is designated to be "mosaico" by (LG)) as the input. In addition, **Mosaico** requires the *contents* and *interface* facets of the *physical* view of all the sub-cells. All these facets are created by (LG). However, (LG) doesn't call **Mosaico** directly. The user should ask the maintainer of the **Mosaico** program to run it. Again the process can by interrupted to wait for the layout generation to be done.

## DEBUG MODE

When the option **g** is specified in (DM) prior to the (LG) run, then the process is in the *debug* mode. In this case most of the module generators will ask the question "Do you want to generate the layout for ... cell?". If the user replies **n** (No) then the layout generation of that cell and all its sub-cells are by-passed. This is useful when some of the cells in the design hierarchy have been generated successfully and the user wants to avoid duplicating the effort.

## SEE ALSO

sdl(5), hdl(5), pdl(5), DM(1), DPC(1), Padroute(1), TimLager(1), Flint(1)

Oct Tools Distribution 1.0

**FILES**

~lager/LagerIII/src/DM_v3/*.l -- source files

~lager/LagerIII/bin/DM -- executable file.

**AUTHORS**

Chuen-Shen Shung

Mani B. Srivastava

## NAME

*DSIM – Design Simulator*

## SYNOPSIS

*(DSIM)*

## DESCRIPTION

(DSIM) is one of the two back-ends of DM. Before (DSIM) can be run, the user has to run (DM). (DM) creates the necessary data structures for (DSIM) to perform simulation. (DSIM) is an event-driven simulator and requires an *event* file as the input. The *event* file can be thought of as the input waveform specification without the absolute timing information.

The simulation models are stored in the *sdl* files using the *sim-list* constructs. If the simulation models are defined for both the parent cell and the sub-cell, then the model of the parent cell will be used. The simulation runs faster if higher level models are used. However, the user has the responsibility to verify the higher level models if he/she wants to create one out of lower level models.

(DSIM) will ask a series of questions after it is invoked:

```
=> (DSIM)

Please enter a list of global clock signals: (APU@PHI1 APU@PHI2)

Please enter input events filename : APU.edl
Output file will be APU.sa

t
=>
```

When simulation finishes, *t* and a Lisp prompt are returned. The user can run another (DSIM) with a different input *event* file. However, the name of the output file is always *instance-name*.sa, so the new simulation results will overwrite the previous simulation results.

The list of global clock signals (*clock list*) is used in conjunction with the *(R)* command in the input *event* list. Unless the *clock list* is *nil* (which can be specified by ()), it has to be a list of two elements. Each of the two elements can be either a symbol or a list. The *clock list* supports a *two-phase non-overlapping* clocking scheme. Let's represent the first element in the *clock list* by *phi1* and the second element by *phi2*. The *(R)* command is equivalent to the following sequence of commands: *phi1*=1 *phi2*=0 (r), *phi1*=0 *phi2*=0 (r), *phi1*=0 *phi2*=1 (r), *phi1*=0 *phi2*=0 (r).

(DSIM) creates a *instance-name*-sim.log file to record the status of the layout generation. (DSIM) reports only one kind of LG-ERROR which indicates that some terminals of the sub-cells are not connected by the parent cell.

## EVENT FILE FORMAT

The *event* file consists of a number of commands, each of which is a list. The name of the command is specified by the first element in the list. The rest in the list specifies the *nets* or the *buses* that the command operates on. A *bus* is specified by the common name of the nets in the *bus* (the names of the nets are different only in the index part). A *net* can be specified either by a *net-name* or by a list of a *cell-name* and a *terminal-name*. Note that the *net-names* and the *cell-names* are full names (see the NAMING CONVENTION section of *DM(1)*). Therefore, the *event* file has to be updated once the *instance-name* of the design is changed.

1. alias command

```
(a (name1a name1b) (name2a name2b) ...)
```

After this command, name1a will be replaced by name1b where it appears; name2a replaced by name2b, ... and so on.

2. break command

(b)

The simulation gets interrupted when the break command is read, the user can resume the simulation by **?ret**.

3. set-high command

(h net1 net2 ...)

The values of net1, net2, ... are set to "1".

4. set-low command

(l net1 net2 ...)

The values of net1, net2, ... are set to "0".

5. set-vector command

(V *bus string*)

Set the value of each net in the *bus* to the corresponding bit in the *string*.

6. clear command

(x net1 net2 ...)

Remove net1, net2, ... from pre-set status. The values of them will be determined by simulation.

7. watch command

(w net1 net2 ...)
(W *bus* start-index end-index)

Put net1, net2, ... and the nets from start-index to end-index in the *bus* to the *watch list*.

8. print command

(p)

Prints the values of all the nets in the *watch list*. Note that the single nets and buses are printed differently.

9. run command

(r)
(R)

Start the simulation with all the *events* since last run command as the input. *(R)* is used in conjunction with the *clock list* specification to run a *major cycle*. *(r)* runs a *minor cycle*.

## SEE ALSO

sdl(5), DM(1), esim(1)

## FILES

~lager/LagerIII/src/DM_v3/*.l -- source files

~lager/LagerIII/bin/DM -- executable file.

~lager/LagerIII/lib/processor/*.sdl
~lager/LagerIII/lib/stdcell/*.sdl
~lager/LagerIII/lib/dpc/leafcells/*.sdl

**AUTHORS**

Chuen-Shen Shung

**BUGS**

Slow. (DSIM) can't run one order of magnitude faster than esim, even though the latter works off transistors.

## NAME

*rassCG – Control Generator* from the *rass* file

## SYNOPSIS

*rassCG* [-i *sadl file*] < *rass file* > *parameter file*

## DESCRIPTION

CG is a pre-processor for DM. For a complicated design, the parameter file needed by DM is often too tedious and error-prone to be created manually. CG can be used to generate the parameter file from a behavioral description, the *rass* file.

Even though the idea of CG is general, the implementation is constrained by the target architecture, which is described by a set of *sdl* files. The names of the parameters used in the *sdl* files have to be the same as the names used by the CG program to generate the parameter file. At this moment CG support only one target architecture, *KAPPA*, (described by ~lager/LagerIII/processor/sdl/*.sdl). *KAPPA* is a programmable architecture which consists of data paths controlled by a microprogram and a control unit that stores the microprogram. Nontheless, the *KAPPA* architecture can be used in quite a wide range of applications, in which cases the user only needs to program the applications in *rass* files and use the set of CG-DM-LG programs to generate the layout automatically.

A *rass* file describes the behavior of an algorithm and some hardware information (e.g. word length of the data path). The user has the options to write the *rass* file directly (in which case it should be more optimized) or write a high-level language (*RL* or *Silage*) and have the *RL* and *Silage* compilers generate the *rass* file. The *RL* compiler is written in such a way that the user can describe the architectures for it to generate different code. CG takes the *rass* file as the input and translates it to the parameter file in which the decoded microprogram is considered to be one of the parameter values of the control unit in *KAPPA*.

By using a *sadl* file as a second input to the CG program, the user can change the data paths in the *KAPPA*. The *sadl* file describes the instruction set and the control signals of each instruction in the target architecture. If *sadl* is not specified, the one which describes *KAPPA* is used by CG (see appendix).

However, if the user wants to change the control unit in the *KAPPA* as well, then the *rass* file and the CG program have to be changed in a big way. Therefore, the name of the program is called *rassCG*. It is expected that some users will rewrite the CG program for their novel architectures.

## RASS FILE FORMAT

A *rass* consists of a number of lists. The first element of each list is used to distinguish the list. The order of these lists is not critical.

1. Hardware information

> **(dp_word_size** *value*)
> **(reset_timer** *list*)
> **(max_sample_intvl** *value*)
> **(stack_depth** *value*)
> **(loop_test** *list*)

**Dp_word_size** specifies the word length of the data path. **Reset_timer** specifies a list of *block numbers*, each of which specifies a microprogram block (see the *Microprogram blocks* list). In each microprogram that is specified by the **reset_timer** list the *timer* is reset. **Max_sample_intvl** specifies the worst case sample interval that is used by the *timer* to produce a constant sampling period. **Stack_depth** specifies the maximum *depth* of the subroutine nesting. If no subroutine is used, then **stack_depth** is zero (default). **Loop_test** specifies a list of *test vectors* that are used by the hardware *loop counter*. An internal conditional input, lctest*i*, which is used to control the state machine in the control unit of *KAPPA*, is asserted when the value of the hardware *loop counter* matches the *i*th *test vector*.

2. Local variables

> **(ram** *scalar1 scalar2 ... array1 array2 ...***)**

This list declares all the local variables names (including scalars and arrays). In the microprogram, these names can be used to refer to the local variables. A *scalar* is represented by a *symbol* starting with a character (A-Z, a-z). Both "_" (underscore) and "-" (hyphen) are allowed in the *symbol*. An *array* is represented by a list of a *symbol* and a *integer*. The *symbol* is the name of the array and the *integer* is the length of the array. The order of scalars and arrays is not critical.

3. Constants

> **(const** *init-scalar ... init-array ...***)**

The list declares a number of constants (initialized local variables whose values are not changed in the entire microprogram). An *init-scalar* is represented by a list of the a *symbol* and the an initial value of the *symbol*. An *init-scalar* is represented by a list of a *symbol* (the array name), an *integer* (the length of the array) and a list of initial values, each for one element in the array (in order).

4. Logic state machine

> **(dfsm** <logic-inst> <logic-inst> ...)
> <logic-inst> = (*inst-name* <logic-prim> <logic-prim> ...)
> <logic-prim> = (*out-name* <in-exprs>)
> <in-exprs> = <in-expr> | (**and** <in-expr> <in-expr> ...)
> <in-expr> = *in-name* | (**not** *in-name*)

This list defines the *logic instructions* that are used in the microprogram. In the microprogram, a *logic instruction* can be invoked by referring to its *inst-name*. A *logic instruction* consists of one or more *logic primitives*. Each *logic primitive* describes the logic relation between the state *out-name* and some states *in-names*. Note that only **and** and **not** can be used to describe the logic relation. The **or** logic can be produced by specifying a *logic primitive* for each or-clause in the same *inst-name*.

5. Control state machine

> **(cfsm** <state-trans> <state-trans> ...)
> <state-trans> = (*state-name block-number* <in-expr> <control>)
> <in-expr> see *Logic state machine* list
> <control> = (**goto** *state-name*) | (**call** *state-name state-name*) | (**return**)

This list is a list of *state transitions*. Each *state transition* is specified by a list of the present *state name*, the *block number* of the present *state name*, a logic relation and a *destination control*. Because several *state names* may use the same block of microprogram, a *block number* field is required. The *destination control* implements 3 functions: (1) go to next state, (2) call a subroutine state and push the return state into the stack, and (3) return from a subroutine. It can be shown that the multi-way branch and the looping can be implemented by the *go to* with proper logic relations.

6. Microprogram blocks

> **(rom** <block0> <block1> ...)
> <block*i*> = (**block***i* <u-inst> <u-inst> ...)
> <u-inst> = (<u-op> <u-op> ...) | (logic-inst-name <u-op> <u-op> ...)

The microprogram is broken into *blocks* in which there is no *state transition* present. We call these *blocks* the *straight-line code blocks*. Each *block* has a block number and a list of *micro-instructions*. Each *micro-instruction* consists of a list of *micro-operations* with one optional *logic instruction* defined in the *Logic state machine* list. The entire set of *micro-operations* and their relations (e.g. some of them can co-exist in the same *micro-instruction*, while some of them can't) are recorded in the *.sadl* file of the target architecture.

## SADL FILE

The *.sadl* file consists of three parts: (1) the set of *micro-operations*, (2) a list of rom control signals and (3) a list of all the *resources* in the target architecture. Each *micro-operation* requires a number of controls signals and occupies a number of *resources* (e.g. registers and buses). If two *micro-operations* need to occupy the same *resource* then they can't co-exist in the same *micro-instruction*. The CG program is able to find these conflicts and report them to the user.

Most of the *micro-operations* are of the form

(*dest=src* [arg])

which means the contents of the resource *dest* will be equal to the contents of the resource *src* after the execution of the *micro-operation*. The *micro-operation* occupies the resource *dest*. Some *micro-operations* require one additional argument.

The *.sadl* file of the *KAPPA* architecture is in the appendix.

## FILES

~lager/LagerIII/processor/{ctrl_gen.l,rom_gen.l,fsm_gen.l} -- source files

~lager/LagerIII/bin/rassCG -- executable file

~lager/LagerIII/processor/KAPPA.sadl

~lager/LagerIII/processor/sdl/*.sdl -- *sdl* files for *KAPPA*

## SEE ALSO

S. Khalid Azim, "Customizable Processor Design for Rapid Implementation of ASICs"

## AUTHOR

Chuen-Shen Shung

# APPENDIX B

## Frame Buffer Controller input files

```
::::::::::::::
sm1.bdsyn
::::::::::::::
MODEL pla1                 ! state transitions of H-V control unit
                 ! in bdsyn format

        ! this is specially for the GE camera
        ! which would provide vsyn, hsyn and cblank signals.

        ! OUTPUT
        count,
        nextstate<3:0> =
        ! bring the h and v signals (these are used by pla2)
        ! from the output of the register to delay one cycle.
        ! h<1:0>=nextstate<1:0>, v<1:0>=nextstate<3:2>

        ! INPUT
        presentstate<3:0>,
        vsyn,
        hsyn,
        cblank,
        eol,
        eof,
        sign;

 CONSTANT
        VAHA=0,                 ! 0000
        VAHB=1,                 ! 0001
        VAHC=2,                 ! 0010
        VAHD=3,                 ! 0011

        VBHA=4,                 ! 0100
        VBHB=5,                 ! 0101
        VBHC=6,                 ! 0110
        VBHD=7,                 ! 0111

        VCHA=8,                 ! 1000
        VCHB=9,                 ! 1001
        VCHC=10,                ! 1010
        VCHD=11,                ! 1011

        VDHA=12,                ! 1100
        VDHB=13,                ! 1101
        VDHC=14,                ! 1110
        VDHD=15;                ! 1111

ROUTINE main;

        SELECT presentstate FROM            ! a multiway switch based on the
                                ! the value of 'presentstate'

        [VAHA]: BEGIN

                count=0;

                IF (vsyn EQL 0) THEN nextstate=VBHA
                ELSE IF (hsyn EQL 0) THEN nextstate=VAHB
                ELSE nextstate=VAHA;

                END;

        [VAHB]: BEGIN

                count=0;

                IF (hsyn EQL 0) THEN nextstate=VAHB
                ELSE nextstate=VAHC;

                END;

        [VAHC]: BEGIN

                count=1;

                IF (sign EQL 1) THEN nextstate=VAHD
                ELSE nextstate=VAHC;

                END;

        [VAHD]: BEGIN

                count=0;

                IF (eol EQL 1) THEN nextstate=VAHA
                ELSE nextstate=VAHD;

                END;

        [VBHA]: BEGIN

                count=0;

                IF (vsyn EQL 1) THEN nextstate=VCHA
                ELSE IF (hsyn EQL 0) THEN nextstate=VBHB
                ELSE nextstate=VBHA;

                END;
```

```
[VBHB]: BEGIN

        count=0;

        IF (hsyn EQL 0) THEN nextstate=VBHB
        ELSE nextstate=VBHC;

        END;

[VBHC]: BEGIN

        count=1;

        IF (sign EQL 1) THEN nextstate=VBHD
        ELSE nextstate=VBHC;

        END;

[VBHD]: BEGIN

        count=0;

        IF (eol EQL 1) THEN nextstate=VBHA
        ELSE nextstate=VBHD;

        END;

[VCHA]: BEGIN

        count=0;

        IF (hsyn EQL 0) THEN nextstate=VCHB
        ELSE nextstate=VCHA;

        END;

[VCHB]: BEGIN

        count=0;

        IF (hsyn EQL 0) THEN nextstate=VCHB
        ELSE nextstate=VCHC;

        END;

[VCHC]: BEGIN

        count=1;

        IF (sign EQL 1) THEN nextstate=VCHD
        ELSE nextstate=VCHC;

        END;
        [VCHD]: BEGIN

                count=0;

                IF (cblank EQL 1) THEN nextstate=VDHD
                ELSE IF (eol EQL 1) THEN nextstate=VCHA
                ELSE nextstate=VCHD;

                END;

        [VDHA]: BEGIN

                count=0;

                IF (hsyn EQL 0) THEN nextstate=VDHB
                ELSE nextstate=VDHA;

                END;

        [VDHB]: BEGIN

                count=0;

                IF (hsyn EQL 0) THEN nextstate=VDHB
                ELSE nextstate=VDHC;

                END;

        [VDHC]: BEGIN

                count=1;

                IF (sign EQL 1) THEN nextstate=VDHD
                ELSE nextstate=VDHC;

                END;

        [OTHERWISE]: BEGIN          !!! note !!!
                                ! mis insists to have an OTHERWISE state
                                ! which serves as the default

                count=0;

                IF (eol EQL 1) AND (eof EQL 1) THEN nextstate=VAHA
                ELSE IF (eol EQL 1) THEN nextstate=VDHA
                ELSE nextstate=VDHD;

                END;

        ENDSELECT;

ENDROUTINE;
ENDMODEL;
```

```
::::::::::::::
sm2.bdsyn
::::::::::::::
MODEL pla2              ! state transitions of memory control unit
             ! in bdsyn format

        ! OUTPUT
        rasINV<0>,
        casINV<0>,
        oeINV<0>,
        wrINV<0>,
        nextstate<4:0> =

        ! INPUT
        h<1:0>,          ! horizontal regions
                         ! h=0 (HA), h=1 (HB), h=2 (HC), h=3 (HD)
        v<1:0>,          ! vertical regions
                         ! v=0 (VA), v=1 (VB), v=2 (VC), v=3 (VD)
        flash<0>,        ! flash=1 (flcont), flash=0 (flend)
        host<1:0>,       ! host=0 (no host), host=1 (host read),
                         ! host=2 (host write), host=3 (illegal)
        presentstate<4:0>;

! state assignments
CONSTANT
        IDLE=0, REF1=1, REF2=2, REF3=3,
        CWR1=8, CWR2=9,
        RE1=12, RE2=13, RE3=14, RE4=15,
        WR1=20, WR2=21, WR3=22, WR4=23,
        CRD1=28, CRD2=29, CRD3=30, CRD4=31;

ROUTINE main;

        nextstate = IDLE;

        SELECT presentstate FROM

        [IDLE]: BEGIN

                rasINV=1; casINV=1; oeINV=1; wrINV=1;

        ! priority of the five memory operations:
        ! REF (refresh)    (1st)
        ! CWR (column write)(2nd)
        ! RE  (host read)  (3rd)
        ! WR  (host write)  (4th)
        ! CRD (cloumn read) (5th)

                IF h EQL 0 THEN nextstate=REF1
                ELSE IF (h EQL 3) AND (v EQL 3) AND flash THEN
                        nextstate=CWR1
                ELSE IF (h NEQ 0) AND NOT (flash AND (h EQL 3) AND (v EQL 3))
                        AND (host EQL 1) THEN
                        nextstate=RE1
        ELSE IF (h NEQ 0) AND NOT (flash AND (h EQL 3) AND (v EQL 3))
                AND (host EQL 2) THEN
                nextstate=WR1
        ELSE IF (h EQL 3) AND (v EQL 3) AND NOT flash AND (host EQL 0)
                THEN nextstate=CRD1
        ELSE nextstate=IDLE;

        END;

[REF1]: BEGIN

        rasINV=1; casINV=0; oeINV=1; wrINV=1;

        nextstate=REF2;

        END;

[REF2]: BEGIN

        rasINV=0; casINV=0; oeINV=1; wrINV=1;

        nextstate=REF3;

        END;

[REF3]: BEGIN

        rasINV=0; casINV=0; oeINV=1; wrINV=1;

        IF (h EQL 0) THEN nextstate=REF1
        ELSE nextstate=IDLE;

        END;

[CWR1]: BEGIN

        rasINV=0; casINV=1; oeINV=1; wrINV=0;

        nextstate=CWR2;

        END;

[CWR2]: BEGIN

        rasINV=0; casINV=1; oeINV=1; wrINV=0;

        IF (h EQL 3) AND (v EQL 3) AND flash THEN
                nextstate=CWR1
        ELSE    nextstate=IDLE;

        END;

[RE1]: BEGIN
```

```
          rasINV=1; casINV=1; oeINV=1; wrINV=1;

          nextstate=RE2;

          END;

[RE2]: BEGIN

          rasINV=0; casINV=1; oeINV=0; wrINV=1;

          nextstate=RE3;

          END;

[RE3]: BEGIN

          rasINV=0; casINV=0; oeINV=0; wrINV=1;

          nextstate=RE4;

          END;

[RE4]: BEGIN

          rasINV=1; casINV=1; oeINV=0; wrINV=1;

          IF (h NEQ 0) AND NOT ((h EQL 3) AND (v EQL 3) AND flash)
                    AND (host EQL 1) THEN
                    nextstate=RE1
          ELSE      nextstate=IDLE;

          END;

[WR1]: BEGIN

          rasINV=1; casINV=1; oeINV=1; wrINV=1;

          nextstate=WR2;

          END;

[WR2]: BEGIN

          rasINV=0; casINV=1; oeINV=1; wrINV=0;

          nextstate=WR3;

          END;

[WR3]: BEGIN

          rasINV=0; casINV=0; oeINV=1; wrINV=0;

          nextstate=WR4;
```

```
        END;

[WR4]: BEGIN

        rasINV=1; casINV=1; oeINV=1; wrINV=0;

        IF (h NEQ 0) AND NOT ((h EQL 3) AND (v EQL 3) AND flash)
                AND (host EQL 2) THEN
                nextstate=WR1
        ELSE    nextstate=IDLE;

        END;

[CRD1]: BEGIN

        rasINV=1; casINV=1; oeINV=1; wrINV=1;

        nextstate=CRD2;

        END;

[CRD2]: BEGIN

        rasINV=0; casINV=1; oeINV=0; wrINV=1;

        nextstate=CRD3;

        END;

[CRD3]: BEGIN

        rasINV=0; casINV=0; oeINV=0; wrINV=1;

        IF (h EQL 3) AND (v EQL 3) AND NOT flash AND (host EQL 0) THEN
                nextstate=CRD3
        ELSE    nextstate=CRD4;

        END;

[CRD4]: BEGIN

        rasINV=1; casINV=1; oeINV=0; wrINV=1;

        IF (h NEQ 0) AND NOT ((h EQL 3) AND (v EQL 3) AND flash)
                AND (host EQL 1) THEN
                nextstate=RE1
        ELSE IF (h NEQ 0) AND NOT ((h EQL 3) AND (v EQL 3) AND flash)
                AND (host EQL 2) THEN
                nextstate=WR1
        ELSE    nextstate=IDLE;

        END;

! Note no OTHERWISE state is required
```

```
        ! This is because there are state numbers that are not
        ! output of this swtich (such as 4, 5, 6, 7, 10, 11, ...)

        ENDSELECT;

ENDROUTINE;
ENDMODEL;
```

# APPENDIX C

## The KAPPA Sadl file

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Name   : KAPPA.sadl
;;; Purpose: Description of the Lager3 Kappa instruction set architecture
;;; Architecture, Instruction Design,
;;;          and Control Signal Specifications: Syed Khalid Azim
;;; Author : Chuen-Shen Bernard Shung
;;; Changes: Lars Thon Mar 1988 (brush-up)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(declare (specials t))
(declare (macros t))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; AU (arithmetic unit)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Register load instructions

(defun mor=mem ()
 (grab mor mem)
 (high pR pLDMOR)
 (low pSELMORIN))        ; R: ram, LDMOR, LDMORINV, MORSELMEM

(defun mem=mbus ()
 (grab mem)
 (high pW))              ; W: ram, WEN

(defun mcondload ()
 (high pWC))             ; WC

(defun mor=mbus ()
 (grab mor)
 (high pSELMORIN pLDMOR))   ; MORSELMBUS

(defun r*=rbus (n)
 (caseq n
        (0 (grab r0) (high pLDR0))          ; LDR0, LDR0INV
        (1 (grab r1) (high pLDR1))          ; LDR1, LDR1INV
        (2 (grab r2) (high pLDR2))          ; LDR2, LDR2INV
        (3 (grab r3) (high pLDR3))          ; LDR3, LDR3INV
        (4 (grab r4) (high pLDR4))))        ; LDR4, LDR4INV
```

```
(defun rcoef=mbus ()
 (grab rcoef)
 (high pLDCOEF))        ; LDCOEF, LDCOEFINV

;;; Move (into a bus) instructions

(defun mbus=mor ()
 (grab mbus)
 (high pXMITMOR))      ; XMITMOR, XMITMORINV

(defun mbus=r* (n)
  (caseq n
    (0 (grab mbus) (high pOENR0) (low pXMITMOR))        ; ONER0 ONER0INV
    (1 (grab mbus) (high pOENR1) (low pXMITMOR))        ; ONER1 ONER1INV
    (2 (grab mbus) (high pOENR2) (low pXMITMOR))        ; ONER2 ONER2INV
    (3 (grab mbus) (high pOENR3) (low pXMITMOR))        ; ONER3 ONER3INV
    (4 (grab mbus) (high pOENR4) (low pXMITMOR))))      ; ONER4 ONER4INV

(defun mbus=acc ()
 (grab mbus)
 (high pXMITACC)
 (low pXMITMOR))               ; XMITACC XMITACCINV

(defun rbus=acc ()
 (grab rbus)
 (high pACC2REG))              ; ACC2REG ACC2REGINV

(defun rbus=ioport ()
 (grab rbus)
 (high pRDPORT))               ; RDPORT, RDPORTINV

(defun ioport=extport (n)  ; RDSTRB
 (grab ioport extport)
 (high pRDSTRB) (low pWRPORT)          ; ioport!=mbus
 (caseq n
        ; PORT ADDRESS = 0000,0001,0010, and so on
        (0 (low pPORTADDRESS3 pPORTADDRESS2 pPORTADDRESS1 pPORTADDRESS0))
        (1 (low pPORTADDRESS3 pPORTADDRESS2 pPORTADDRESS1) ; 0001
          (high pPORTADDRESS0))
        (2 (low pPORTADDRESS3 pPORTADDRESS2 pPORTADDRESS0) ; 0010
          (high pPORTADDRESS1))
        (3 (low pPORTADDRESS3 pPORTADDRESS3 pPORTADDRESS2) ; 0011
          (high pPORTADDRESS1 pPORTADDRESS0))
        (4 (high pPORTADDRESS2)                            ; 0100
          (low pPORTADDRESS3 pPORTADDRESS1 pPORTADDRESS0))
        (5 (high pPORTADDRESS2 pPORTADDRESS0)                  ; 0101
          (low pPORTADDRESS3 pPORTADDRESS1))
        (6 (high pPORTADDRESS2 pPORTADDRESS1)                  ; 0110
          (low pPORTADDRESS3 pPORTADDRESS0))
        (7 (high pPORTADDRESS2 pPORTADDRESS1 pPORTADDRESS0)            ; 0111
          (low pPORTADDRESS3))
        (8 (low pPORTADDRESS0 pPORTADDRESS1 pPORTADDRESS2  )           ; 1000
          (high pPORTADDRESS3 ))
        (9 (low pPORTADDRESS1 pPORTADDRESS2 )                  ; 1001
          (high pPORTADDRESS3 pPORTADDRESS0 ))
        (10 (low pPORTADDRESS0 pPORTADDRESS2 )                    ; 1010
          (high pPORTADDRESS3 pPORTADDRESS1 ))
        (11 (low pPORTADDRESS2)                          ; 1011
          (high pPORTADDRESS3 pPORTADDRESS1 pPORTADDRESS0))
        (12 (low pPORTADDRESS0 pPORTADDRESS1)                     ; 1100
          (high pPORTADDRESS3 pPORTADDRESS2))
        (13 (low pPORTADDRESS1)                          ; 1101
          (high pPORTADDRESS3 pPORTADDRESS2 pPORTADDRESS0 ))
        (14 (low pPORTADDRESS0)                          ; 1110
          (high pPORTADDRESS3 pPORTADDRESS2 pPORTADDRESS1 ))
        (15 (high pPORTADDRESS3 pPORTADDRESS2 pPORTADDRESS1 pPORTADDRESS0 ))))

(defun ioport=mbus ()
 (grab ioport)
 (high pWRPORT))                    ; WRPORT, WRPORTINV

(defun extport=ioport (n)
 (grab extport)
 (high pWRSTRB)                     ; WRSTRB
 (caseq n
        ; PORT ADDRESS = 0000,0001,0010, and so on
        (0 (low pPORTADDRESS3 pPORTADDRESS2 pPORTADDRESS1 pPORTADDRESS0))
        (1 (low pPORTADDRESS3 pPORTADDRESS2 pPORTADDRESS1) ; 0001
          (high pPORTADDRESS0))
        (2 (low pPORTADDRESS3 pPORTADDRESS2 pPORTADDRESS0) ; 0010
          (high pPORTADDRESS1))
        (3 (low pPORTADDRESS3 pPORTADDRESS3 pPORTADDRESS2) ; 0011
          (high pPORTADDRESS1 pPORTADDRESS0))
        (4 (high pPORTADDRESS2)                          ; 0100
          (low pPORTADDRESS3 pPORTADDRESS1 pPORTADDRESS0))
        (5 (high pPORTADDRESS2 pPORTADDRESS0)                     ; 0101
          (low pPORTADDRESS3 pPORTADDRESS1))
        (6 (high pPORTADDRESS2 pPORTADDRESS1)                     ; 0110
          (low pPORTADDRESS3 pPORTADDRESS0))
        (7 (high pPORTADDRESS2 pPORTADDRESS1 pPORTADDRESS0)            ; 0111
          (low pPORTADDRESS3))
        (8 (low pPORTADDRESS0 pPORTADDRESS1 pPORTADDRESS2 )            ; 1000
          (high pPORTADDRESS3 ))
        (9 (low pPORTADDRESS1 pPORTADDRESS2 )                     ; 1001
          (high pPORTADDRESS3 pPORTADDRESS0 ))
        (10 (low pPORTADDRESS0 pPORTADDRESS2 )                    ; 1010
          (high pPORTADDRESS3 pPORTADDRESS1 ))
        (11 (low pPORTADDRESS2)                          ; 1011
          (high pPORTADDRESS3 pPORTADDRESS1 pPORTADDRESS0))
        (12 (low pPORTADDRESS0 pPORTADDRESS1)                     ; 1100
          (high pPORTADDRESS3 pPORTADDRESS2))
        (13 (low pPORTADDRESS1)                          ; 1101
          (high pPORTADDRESS3 pPORTADDRESS2 pPORTADDRESS0 ))
        (14 (low pPORTADDRESS0)                          ; 1110
          (high pPORTADDRESS3 pPORTADDRESS2 pPORTADDRESS1 ))
        (15 (high pPORTADDRESS3 pPORTADDRESS2 pPORTADDRESS1 pPORTADDRESS0 ))))

(defun acc=0 ()
 (grab abus bbus acc)
 (high pNOABS pZERO_BIN)              ; ZERO_BIN, ZERO_AIN
 (low pANDCOEF pMINUS pCOEFCOMP))

(defun acc=sum ()
 (grab acc))

(defun acc=abus ()                    ; the old bbus=0
 (grab acc bbus)
 (high pZERO_BIN))                    ; ZERO_BIN

(defun acc=bbus ()                    ; the old abus=0
 (grab acc abus)
 (high pNOABS)                        ; ZERO_AIN
 (low pMINUS pANDCOEF pCOEFCOMP))

(defun abus=1 ()              ; actually abus=0 and cin=1
 (grab abus)                          ; used for incr bbus
 (high pNOABS pMINUS)
 (low pANDCOEF pCOEFCOMP))                     ; COMPLA, COMPLAINV

(defun abus=mor ()
 (grab abus)
 (high pNOABS pCOEFCOMP)
 (low pMINUS pANDCOEF))

(defun abus=-mor ()
 (grab abus)
 (high pNOABS pMINUS pCOEFCOMP)
 (low pANDCOEF))

(defun abus=absmor ()
 (grab abus)
 (high pCOEFCOMP)
 (low pNOABS pMINUS pANDCOEF))

(defun abus=-absmor ()
 (grab abus)
 (low pNOABS pANDCOEF)
 (high pMINUS pCOEFCOMP))

(defun abus=coef.mor ()
 (grab abus)
 (high pNOABS pANDCOEF)
 (low pMINUS pCOEFCOMP))

(defun abus=coef.-mor ()
 (grab abus)
 (high pNOABS pMINUS pANDCOEF)
 (low pCOEFCOMP))

(defun abus=coef.absmor ()
 (grab abus)
 (high pANDCOEF)
 (low pNOABS pMINUS pCOEFCOMP))

(defun abus=coef.-absmor ()
 (grab abus)
 (high pMINUS pANDCOEF)
 (low pNOABS pCOEFCOMP))

(defun abus=~coef.mor ()
 (grab abus)
 (high pANDCOEF pCOEFCOMP pNOABS)
 (low pMINUS))

(defun abus=~coef.-mor ()
 (grab abus)
 (high pANDCOEF pCOEFCOMP pNOABS pMINUS))

(defun abus=~coef.absmor ()
 (grab abus)
 (high pANDCOEF pCOEFCOMP)
 (low pNOABS pMINUS))

(defun abus=~coef.-absmor ()
 (grab abus)
 (high pANDCOEF pCOEFCOMP pMINUS)
 (low pNOABS))

(defun bbus=mbus ()
 (grab bbus)
 (low pZERO_BIN)
 (low pSELBBUSIN))

(defun bbus=acc>* (n)
  (grab bbus)
  (low pZERO_BIN)
  (high pSELBBUSIN)
  (caseq n
    (0 (low pS2 pS1) (high pS0))
    (1 (low pS2 pS0) (high pS1))
    (2 (low pS2) (high pS1 pS0))
    (3 (high pS2) (low pS1 pS0))
    (4 (high pS2 pS0) (low pS1))
    (5 (high pS2 pS1) (low pS0))
    (6 (high pS2 pS1 pS0))
    (t (format t "Illegal instruction: right-shift (~a) out of bound~%" n))))

(defun bbus=acc<* (n)
  (grab bbus)
  (high pSELBBUSIN)
  (low pZERO_BIN)
  (caseq n
    (1 (low pS2 pS1 pS0))
    (t (format t "Illegal instruction: left-shift (~a) out of bound~%" n))))

(defun acondload ()
 (high pSUMCOND))

(defun shrcoef ()
 (high pSHIFTCOEF))

;;; Miscellaneous "instructions"

(defun nosat () (high pNOSAT))
(defun aip   () (high pAIP))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; APU (address processing unit)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Regsiter load instructions

(defun x*=eabus (n)
  (caseq n
    (0 (grab x0) (high pLOADX0)) ; LOADX0, LOADX0INV
    (1 (grab x1) (high pLOADX1)) ; LOADX1, LOADX1INV
    (2 (grab x2) (high pLOADX2)) ; LOADX2, LOADX2INV
    (3 (grab x3) (high pLOADX3)) ; LOADX3, LOADX3INV
    (4 (grab x4) (high pLOADX4))))          ; LOADX4, LOADX4INV

(defun xcondload ()
 (high pCONDLD))

;;;Move (into a bus) instructions

(defun addr fexpr (l)
  (grab dbus)
  (ramdecodebase (car l)))

(defun offset (l)
  (ramdecodeoffset l))

(defun xip ()
 (high pXIP))

(defun xbus=x* (n)
  (caseq n
    (0 (grab xbus) (low pXBUSZERO) (high pOENX0))       ; OENX0, OENX0INV
    (1 (grab xbus) (low pXBUSZERO) (high pOENX1))       ; OENX1, OENX1INV
    (2 (grab xbus) (low pXBUSZERO) (high pOENX2))       ; OENX2, OENX2INV
    (3 (grab xbus) (low pXBUSZERO) (high pOENX3))       ; OENX3, OENX3INV
    (4 (grab xbus) (low pXBUSZERO) (high pOENX4))))     ; OENX4, OENX4INV

(defun xbus=0 ()
 (grab xbus)
 (high pXBUSZERO))                ; XBUSZERO

(defun eabus=sum ()
 (grab eabus)
 (high pOENEALATCH))              ; OENEALATCH, OENEALATCHINV
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Communication between AU and APU
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun eabus=mbus ()
   (grab eabus)
   (high pMBUS2EABUS))

(defun areg=eabus ()
   (grab areg)
   (high pEABUS2MBUS))

(defun mbus=areg ()              ; this must always  be asserted
   (grab mbus)                   ; following areg=eabus
  (low pXMITMOR))

(defun timereg=eabus ()
   (grab timerinreg)
   (high pLDTIMER))              ; LDTIMER, LDTIMERINV

;;; No operation (nop). Handles all the defaults.

(defun nop ()
  (high pXBUSZERO) (ramdecodebase 0)            ; xbus=0, dbus=0
  (high pSELMORIN)                              ; mor=mbus
  (high pXMITMOR)                               ; mbus=mor
  (high pNOABS) (low pMINUS pANDCOEF pCOEFCOMP)          ; abus=0
  (high pZERO_BIN)                              ; bbus=0
  (low pAIP pSUMCOND)                                ; acc=sum
  (low pS2 pS1) (high pS0)                      ; shifterout=acc>0
  (high pWRPORT))                               ; ioport=mbus


;;; Ken insists to use (fsm #) instead of an identifier
;;; to refer to the dfsm instruction

(defun fsm (n) n)

;;; Ken invented

(defun immed (n) (grab dbus) (ramdecodeoffset n))
(setq reg+bus 20)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Register inventory
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(setq mor 0)
(setq acc 1)
(setq r0 2)
(setq r1 3)
(setq rcoef 4)
(setq x0 5)
(setq x1 6)
```

```
(setq x2 7)
(setq mem 8)
(setq areg 9)            ; fictitious
(setq timerinreg 10)     ; write only

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Bus inventory
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(setq ioport 11)
(setq mbus 12)
(setq rbus 13)
(setq abus 14)
(setq bbus 15)
(setq eabus 16)
(setq xbus 17)
(setq dbus 18)
(setq extport 19)  ; fictitious

;
; EOB
;

(setq EOB 0)
(setq pLDTIMER 1)
;;; if stack is used then need EOB2
;;; (setq EOB2 2)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Arithmetic Unit microcode bits (26 in this version)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(setq pWC 2)
(setq pAIP 3)
(setq pNOSAT 4)
(setq pLDMOR 5)
(setq pLDR0 6)
(setq pLDR1 7)
(setq pLDCOEF 8)
(setq pXMITMOR 9)
(setq pXMITACC 10)
(setq pSELMORIN 11)
(setq pANDCOEF 12)
(setq pCOEFCOMP 13)
(setq pNOABS 14)
(setq pMINUS 15)
(setq pSELBBUSIN 16)
(setq pZERO_BIN 17)
(setq pS0 18)
(setq pS1 19)
(setq pS2 20)
(setq pOENR0 21)
(setq pOENR1 22)
(setq pACC2REG 23)
```

```
(setq pSUMCOND 24)
(setq pSHIFTCOEF 25)
(setq pRDPORT 26)
(setq pWRPORT 27)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Address Processing Unit microcode bits (12 in this version)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(setq pLOADX0 28)
(setq pLOADX1 29)
(setq pLOADX2 30)
(setq pCONDLD 31)
(setq pOENX0 32)
(setq pOENX1 33)
(setq pOENX2 34)
(setq pXBUSZERO 35)
(setq pEABUS2MBUS 36)
(setq pMBUS2EABUS 37)
(setq pOENEALATCH 38)
(setq pXIP 39)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; I/O related microcode bits (6 in this version)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(setq pRDSTRB 40)
(setq pWRSTRB 41)
(setq pPORTADDRESS0 42)
(setq pPORTADDRESS1 43)
(setq pPORTADDRESS2 44)
(setq pPORTADDRESS3 45)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; RAM control microcode bits (2 in this version)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(setq pR 46)
(setq pW 47)

(setq NR_CTR 48) ;0-48 makes 49 bits

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; "Other fields "
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(setq FSMFIELD 48)
(setq ADDRCODE 49)
(setq ADDRCODE1 50)
```

## APPENDIX D

## Pitch Tracker Inputs

1. Silage program

2. RL program

3. Rass program

```
/*
 *       Gold Pitch Tracker in Silage
 */

#define W 16                        /* data path width */

VOICED = 5;
BLANK = 12;
DECAY = 1 - 3/128;
DELTA = 4;

/*       Interpretation of the constants:
 *
 *       After locating a pitch, a blanking interval of duration BLANK
 *       if entered during which all peaks are rejected.
 *
 *       Then, an exponentially-decaying threshold signal with time
 *       constant DECAY is computed.  Minor peaks fail to exceed this
 *       threshold is rejected.
 *
 *       Window comparison: two values are said to be the same
 *       if their difference is less than or equal to DELTA
 *
 *       If all scores are less than VOICED then the speech are
 *       considered as unvoiced
 */

#define N 6

#define compare(a, b)       (abs(a - b) < DELTA -> 1 || 0)

#define is_peak(x)          (x@1 & !x)
#define is_valley(x)        (!x@1 & x)

func main(in: fix<16>): int<8> =
begin
        Y = Y@1 >= N - 1 -> int<W>(0) || Y@1 + 1;

        sig = fix<W>(in);           /* new input value */
        slp = sig > sig@1;                      /* slope */

        lp = is_peak(slp) -> sig@1 || lp@1; /* last peak */
        lv = is_valley(slp) -> sig@1 || lv@1;           /* last valley */

        /* compute 6 pitch candidates */
        signal[0] = sig;
        signal[1] = - sig;
        signal[2] = sig / 2 - lv / 2;
        signal[3] = - signal[2];
        signal[4] = lp / 2 - sig / 2;
        signal[5] = - signal[4];

        score[0] = int<W>(0);
        (X: N) :: pp, score: begin
                newppc = ppc@1 + 1;
                after_blank = newppc > BLANK;
                newthresh = after_blank -> DECAY * thresh@1 || thresh@1;

        /* new candidate if (A) after blanking period (B) greater than
         * threshold (C) peak or valley
         */
                ping = after_blank & signal[X] > newthresh &
                        (is_peak(slp) | is_valley(slp));

                lpp = ping -> pp[X]@1 || lpp@1;
                pp[X] = ping -> newppc || pp[X]@1;
                ppc = ping -> int<W>(0) || newppc;
                thresh = ping -> signal[X] || newthresh;

        /* compute score */
                score[X+1] = score[X] + compare(pp[X], pp[Y])
                                      + compare(lpp, pp[Y])
                                      + compare(pp[X] + lpp, pp[Y]);
        end;

        /* compare scores */
        bingo = score[N] > topscore@1;
        newtopscore = bingo -> score[N] || topscore@1;
        winner = bingo -> pp[Y] || winner@1;

        /* if top score > VOICED then update pitch else retain pitch */
        pitch = (Y = N-1) -> (newtopscore < VOICED -> 0 || winner)
                || pitch@1;
        topscore = (Y = N-1) -> 0 || newtopscore;
        return = int<8>(pitch);
end;
```

```
/*
 *       Gold Pitch Tracker in RL
 */

macro   N = 6;
macro   VOICED = 5,    BLANK = 12,    DECAY = (1 - 3/128),      DELTA = 4;
/* see constant interpretation is Silage program */

macro compare(a, b) = abs(a - b) < DELTA;

macro is_peak(x, old_x) = old_x and not x;
macro is_valley(x, old_x) = not old_x and x;

fix     signal[N], sig, old_sig, last_peak, last_valley;
int     score, topscore, pitch, winner, pp[N], old_pp[N];
bool    slope, old_slope;

main() {
   pitch = (topscore < VOICED) ? 0 : winner;
   topscore = 0;

   for I = 0 to N-1 do {
         fix x;

         out(pitch);

         low_pass(sig, old_sig, fix in());

         old_slope = slope;
         slope = (sig > old_sig);

         x = old_sig;
         signal[0] = x;
         signal[1] = - x;

         last_valley = is_valley(slope, old_slope) ? old_sig : last_valley;

         x = old_sig/2 - last_valley/2;
         signal[2] = x;
         signal[3] = - x;

         last_peak = is_peak(slope, old_slope) ? old_sig : last_peak;

         x = last_peak/2 - old_sig/2;
         signal[4] = x;
         signal[5] = - x;

         score = 0;

         for J = 0 to N-1 do {
            int ppc[N], ppc_J;
            fix thresh[N], thresh_J;
            bool is_extremum, after_blank;
            ppc_J = ppc[J] + 1;
            ppc[J] = ppc_J;
            after_blank = (ppc_J > BLANK);

            is_extremum = is_peak(slope, old_slope)
                    or is_valley(slope, old_slope);

            thresh_J = thresh[J];
            thresh_J = after_blank ? DECAY * thresh_J : thresh_J;
            thresh[J] = thresh_J;


        /* new candidate if (A) after blanking period (B) greater than
         * threshold (C) peak or valley (or is_extreme)
         */
            if after_blank and is_extremum and signal[J] > thresh_J then {
                old_pp[J] = pp[J];
                pp[J] = ppc[J];
                ppc[J] = 0;
                thresh[J] = signal[J];
            } else {        /* refresh */
                old_pp[J] = old_pp[J];
                pp[J] = pp[J];
            }

            tally_score(score, pp[I], pp[J], old_pp[J]);
        }

        if score > topscore then {
            topscore = score;
            winner = pp[I];
        }
    }
}

/* low pass filter */
inline low_pass(in out z, out old_z, in x)
fix     x, y, z, old_z;                       /* y is internal state. */
{
    old_z = z;
    y = -x/4 + (3/4)*y;
    z = y + (3/4)*z;
}

/* compute raw scores */
inline tally_score(in out score, in a, in b, in c)
int     score, a, b, c;
{
    score = compare(a, b) ? score + 1 : score;
    score = compare(a, c) ? score + 1 : score;
    score = compare(a, b + c) ? score + 1 : score;
}
```

```
; this rass program implements the (modified) Gold pitch tracking
; algorithm.  The LPF front-end is included, whose transfer
; function is
;
;                    -(1/4) / (1 - 3/4 * 1/z) ^ 2
;
; local variables
(ram f g (thresh 6) (ppc 6) (pp 7) (lpp 6) (signal 6)
        ls lp lv score topscore pitch winner)

; VOICED, DELTA, DECAY and BLANK constants are
; hardwired in the program, and hence are not defined
; as constants
(const (timer 350))

; define logic instructions
(dfsm
  (SET (cc (not AU1SIGN)))
  (AND_MINUS (cc (and cc AU1SIGN)))
  (APV (cc (and cc even (not slp) lsp))
    (cc (and cc (not even) slp (not lsp)))
    (even (not even)))
  (VPE (cc APU1SIGN)
    (even ONE))
  (SIP (cc (and (not slp) lsp)))
  (SIV (cc (and slp (not lsp))))
  (SSL (lsp slp)
    (slp (not AU1SIGN)))
)

; define control flow
; syntax in each state
; (1) state name (2) code block number (3) condition (4) control flow operation
(cfsm
  (INITTIMER    0       0           (goto RSTCOUNTER))
  (RSTCOUNTER           5       0               (goto LPFPV))
  (LPFPV        1       0               (goto PITCH))
  (PITCH        2       (not APU1SIGN) (goto PITCH))
  (PITCH        2       APU1SIGN        (goto IDLE))
  (IDLE 4       EOS             (goto SCORE))
  (IDLE 4       (not EOS)               (goto IDLE))
  (SCORE        3       APU1SIGN        (goto RSTCOUNTER))
  (SCORE        3       (not APU1SIGN) (goto LPFPV))
)

; some hardware information
(reset_timer INITTIMER)
(max_sample_intvl 350)
(dp_word_size 16)

; code blocks
(rom
  (block0
    ((mor=mem) (addr timer) (xbus=0) (eabus=sum))  ; r(timer)
```

```
 ((mbus=mor) (eabus=mbus) (timereg=eabus))       ; ldtimer
)

(block1
 ; LPF
 ((mor=mem) (addr f) (rbus=ioport) (r*=rbus 1) (xbus=0)
  (eabus=sum) (ioport=extport 0))          ; r(f), r1=in(port0)

 ((bbus=mbus) (mbus=r* 1) (acc=bbus))             ; acc=r1

 ((abus=-mor) (bbus=acc>* 0) (nosat) (acc=sum))            ; acc=-mor+acc

 ((mor=mem) (addr g) (bbus=acc>* 2) (acc=bbus) (xbus=0)
  (eabus=sum)) ; acc=acc>2, r(g)

 ((abus=-mor) (acc=abus) (rbus=acc) (r*=rbus 1) (nosat) (eabus=sum)
  (mbus=acc) (mem=mbus) (addr f) (xbus=0))       ; acc=-mor, r1=acc, w(f)=acc

 ((abus=mor) (bbus=acc>* 2) (acc=sum) (mor=mbus)
  (mbus=r* 1))             ; acc=mor+acc>2, mor=r1

 ((abus=-mor) (bbus=acc>* 0) (acc=sum) (addr ls) (mor=mem))
        ; acc=-mor+acc, r(ls)

 ; peak/valley detector
 ((mem=mbus) (addr g) (abus=-mor) (bbus=acc>* 0) (eabus=sum) (xbus=0)
  (acc=sum) (rbus=acc) (r*=rbus 1) (mbus=acc))    ; w(g)=acc, acc=-mor+acc, r1=acc

 (SSL (mem=mbus) (addr signal) (mbus=r* 1) (acc=0) (mor=mbus) (xbus=0)
  (eabus=sum)) ; w(signal)=r1, acc=0, mor=r1, SSL

 ((abus=-mor) (acc=abus) (mem=mbus) (addr score) (xbus=0)
  (mbus=acc) (eabus=sum)) ; acc=-mor, w(score)=acc

 ((bbus=acc>* 1) (acc=bbus) (mem=mbus) (mbus=acc) (addr signal)
  (eabus=sum) (xbus=0) (offset 1))          ; acc=acc>1, w(signal[1])=acc

 ((rbus=acc) (r*=rbus 0) (mor=mem) (addr lv) (xbus=0)
  (eabus=sum))             ; r(lv), r0=acc (-g/2)

 ((mbus=r* 0) (mor=mbus) (abus=mor) (acc=abus)); mor=r0, acc=mor

 ((abus=mor) (bbus=acc>* 1) (acc=sum))  ; acc=mor+acc>1

 ((mbus=acc) (mem=mbus) (addr signal) (offset 3) (mor=mbus)
  (eabus=sum) (xbus=0))           ; w(signal[3])=acc, mor=acc

 ((abus=-mor) (acc=abus) (mor=mem) (addr lp) (xbus=0)
  (eabus=sum))            ; acc=-mor, r(lp)

 ((mem=mbus) (mbus=acc) (addr signal) (offset 2) (xbus=0)
  (eabus=sum)) ; w(signal[2])=acc

 ((mbus=r* 0) (mor=mbus) (acc=abus) (abus=mor)); mor=r0, acc=mor
 ((abus=mor) (bbus=acc>* 1) (acc=sum))  ; acc=mor+acc>1

 ((mbus=acc) (mem=mbus) (addr signal) (offset 5) (xbus=0)
  (eabus=sum) (mor=mbus))       ; w(signal[5])=acc, mor=acc

 ((abus=-mor) (acc=abus) (mor=mem) (addr pitch) (xbus=0) (eabus=sum))
                                ; acc=-mor, r(pitch)

 ((mem=mbus) (mbus=acc) (addr signal) (offset 4) (xbus=0)
  (eabus=sum)) ; w(signal[4])=acc

 ((mbus=mor) (ioport=mbus) (extport=ioport 0) (addr 0) (xbus=0)
  (eabus=sum) (x*=eabus 1))                 ; out(port0)=mor, x1=0
) ; end block1

(block2; pitch detector
 ((mor=mem) (addr thresh) (xbus=x* 1) (eabus=sum))         ; rx1(thresh)

 ((mor=mem) (addr ppc) (xbus=x* 1) (abus=mor) (eabus=sum)
  (acc=abus))                 ; rx1(ppc), acc=mor

 ((addr 12) (areg=eabus) (abus=mor) (acc=abus) (xbus=0) (eabus=sum)
  (rbus=acc) (r*=rbus 1))             ; rc(12), acc=mor, r1=acc (thresh)

 ((mbus=areg) (abus=1) (mor=mbus)
  (bbus=acc>* 0) (acc=sum))       ; acc=1+acc, mor=areg

 ((abus=-mor) (bbus=acc>* 0) (acc=sum) (mbus=r* 1) (mem=mbus)
  (addr thresh) (xbus=x* 1) (eabus=sum) (rbus=acc) (r*=rbus 0))
                    ; acc=-mor+acc, r0=acc (newppc), wx1(thresh)=r1

 ((bbus=mbus) (mbus=r* 1) (nosat) (acc=bbus) (mor=mbus)
  SET)                          ; acc=r1, mor=r1, SET

 ((addr ppc) (xbus=x* 1) (mem=mbus) (mbus=r* 0) (abus=mor) (eabus=sum)
  (bbus=acc>* 1) (nosat) (acc=sum))        ; wx1(ppc)=r0, acc=mor+acc>1

 ((bbus=acc>* 6) (acc=bbus) (mor=mem) (addr lp) (eabus=sum)
  (xbus=0))                     ; acc=acc>6, r(lp)

 ((mor=mbus) (mbus=acc) (acc=abus) (abus=mor))
        ; mor=acc, acc=mor

 ((abus=-mor) (bbus=mbus) (mbus=r* 1) (acc=sum) (mor=mem) (eabus=sum)
  (addr signal) (rbus=acc) (r*=rbus 0) (xbus=0))
                    ; acc=-mor+r1, r(signal), r0=acc (lp)

 ((mbus=acc) (mcondload) (mem=mbus) (addr thresh) (xbus=x* 1) (eabus=sum)
  (abus=-mor) (acc=sum) (bbus=acc>* 0) APV)
                   ; acc=-mor+acc, wx1c(thresh)=acc, APV

 (AND_MINUS (abus=mor) (acc=abus) (mor=mem) (addr lv) (xbus=0)
  (eabus=sum))             ; AND_MINUS, acc=mor, r(lv)
  ((mbus=acc) (mcondload) (mem=mbus) (addr thresh) (xbus=x* 1) (acc=abus)
   (abus=mor) (eabus=sum))          ; wx1c(thresh)=acc, acc=mor

  ((mor=mem) (addr pp) (xbus=x* 1) (eabus=sum) (rbus=acc) (r*=rbus 1))
                     ; rx1(pp), r1=acc (lv)

  ((mbus=mor) (mcondload) (mem=mbus) (addr lpp) (xbus=x* 1) (eabus=sum))
                                           ; wx1c(lpp)=mor

  ((mor=mem) (addr ppc) (xbus=x* 1) (acc=0)
   (eabus=sum)) ; rx1(ppc), acc=0

  ((mbus=acc) (mcondload) (mem=mbus) (addr ppc) (xbus=x* 1)
   (eabus=sum)) ; wx1c(ppc)=acc

  ((mbus=mor) (mcondload) (mem=mbus) (addr pp) (xbus=x* 1)
   (eabus=sum)) ; wx1c(pp)=mor

  ((mbus=r* 0) (mem=mbus) (addr lp) (xbus=0) (eabus=sum))          ; w(lp)=r0

  ((mbus=r* 1) (mem=mbus) (addr lv) (xbus=0) (eabus=sum))          ; w(lv)=r1

; modify the score
  ((mor=mem) (addr pp) (xbus=0) (eabus=sum))                ; r(pp)

  ((mem=mbus) (mbus=mor) (addr pp) (offset 6) (xbus=0))
                     ; w(pp[6])=mor

  ((mor=mem) (addr pp) (offset 1) (xbus=x* 2) (eabus=sum)) ; rx2(pp[1])

  ((mor=mem) (addr pp) (xbus=x* 1) (eabus=sum) (abus=-mor)
   (acc=abus))                      ; rx1(pp), acc=-mor

  ((addr 4) (areg=eabus) (abus=mor) (bbus=acc>* 0)
   (acc=sum))                       ; rc(4), acc=mor+acc

  ((mor=mbus) (mbus=areg) (addr -8) (xbus=0)
   (eabus=sum)) ; rc(-8), mor=areg

  ((abus=mor) (bbus=acc>* 0) (acc=sum) (mor=mbus) (mbus=areg))
                                    ; acc=mor+acc, mor=areg

  ((abus=mor) (bbus=acc>* 0) (acc=sum) SET (mor=mem)
   (addr score) (xbus=0) (eabus=sum))       ; acc=mor+acc, SET, r(score)

  ((abus=mor) (acc=abus) AND_MINUS)              ; acc=mor, AND_MINUS

  ((mor=mem) (addr pp) (offset 1) (xbus=x* 2) (abus=1) (eabus=sum)
   (bbus=acc>* 0) (acc=sum))      ; rx2(pp[1]), acc=1+acc

  ((mor=mem) (addr lpp) (xbus=x* 1) (abus=-mor) (eabus=sum)
   (acc=abus) (rbus=acc) (r*=rbus 1))        ; rx1(lpp), acc=-mor, r1=acc

  ((addr 4) (areg=eabus) (xbus=0) (eabus=sum) (abus=mor) (bbus=acc>* 0)
 (acc=sum))                          ; rc(4), acc=mor+acc

((addr -8) (mbus=areg) (areg=eabus) (xbus=0) (eabus=sum) (mor=mbus))
                 ; rc(-8), mor=areg

((abus=mor) (acc=sum) (bbus=acc>* 0) (mor=mbus) (mbus=areg))
                                     ; acc=mor+acc, mor=areg

((mem=mbus) (mcondload) (addr score) (xbus=0) (eabus=sum) SET
 (mbus=r* 1) (abus=mor) (bbus=acc>* 0) (acc=sum))
                             ; wc(score)=r1, SET, acc=mor+acc

((addr score) (xbus=0) (eabus=sum) AND_MINUS (abus=1)
 (acc=abus))    ; AND_MINUS, r(score), acc=1

((mor=mem) (addr pp) (offset 1) (xbus=x* 2) (eabus=sum) (abus=mor)
 (bbus=acc>* 0) (acc=sum))       ; rx2(pp[1]), acc=mor+acc

((mor=mem) (addr pp) (xbus=x* 1) (abus=-mor) (rbus=acc) (r*=rbus 1)
 (acc=abus))                 ; rx1(lpp), acc=-mor, r1=acc

((addr 4) (areg=eabus) (abus=mor) (bbus=acc>* 0) (xbus=0) (eabus=sum)
 (acc=sum))                          ; rc(4), acc=mor+acc

((addr -8) (areg=eabus) (mbus=areg) (mor=mbus)
 (xbus=0) (eabus=sum))               ; rc(-8), mor=areg

((mbus=areg) (mor=mbus) (abus=mor) (bbus=acc>* 0) (acc=sum))
                             ; mor=areg, acc=mor+acc

((mcondload) (mem=mbus) (mbus=r* 1) (addr score) (xbus=0)
 (eabus=sum) SET (abus=mor) (acc=sum) (bbus=acc>* 0))
                             ; SET, wc(score)=r1, acc=mor+acc

((mor=mem) (addr score) (abus=1) (xbus=0) (eabus=sum)
 (acc=abus) AND_MINUS)                  ; r(score), acc=1, AND_MINUS

((abus=mor) (bbus=acc>* 0) (acc=sum) (xbus=x* 1) (addr 1)
 (eabus=sum) (x*=eabus 1))              ; acc=mor+acc, x1=x1+1

((mcondload) (mem=mbus) (mbus=acc) (addr score) (xbus=0)
 (eabus=sum)) ; wc(score)=acc

((addr -5) (xbus=x* 1) (eabus=sum))     ; ea=x1-5
) ; end block2

(block3; scoring

 ((mor=mem) (addr score) (eabus=sum) (xbus=0))          ; r(score)

 ((mor=mem) (addr topscore) (abus=mor) (eabus=sum) (xbus=0) (acc=abus))
                             ; r(topscore), acc=mor

 ((abus=-mor) (bbus=acc>* 0) (acc=sum) (rbus=acc) (r*=rbus 1)
 (mor=mem) (addr pp) (xbus=x* 2) (eabus=sum))
                  ; acc=-mor+acc, r1=acc, rx2(pp)

(SET (abus=mor) (acc=abus) (addr 1) (xbus=x* 2)
 (eabus=sum) (x*=eabus 2))              ; SET, acc=mor, x2=x2+1

((mcondload) (mem=mbus) (addr winner) (xbus=0) (eabus=sum)
 (mbus=acc)) ; wc(winner)=acc

((mcondload) (mem=mbus) (addr topscore) (xbus=0) (eabus=sum)
 (mbus=r* 1)) ; wc(topscore)=r1

((addr -6) (xbus=x* 2) (eabus=sum))                ; ea=x2-6

(VPE (mor=mem) (addr winner) (xbus=0) (eabus=sum))   ; VPE, r(winner)

((addr topscore) (xbus=0) (eabus=sum) (mor=mem)
 (abus=mor) (acc=abus))                 ; r(topscore), acc=mor

((addr pitch) (mem=mbus) (mcondload) (xbus=0) (eabus=sum)
 (mbus=acc) (abus=mor) (acc=abus))      ; wc(pitch)=acc, acc=mor

((addr 5) (xbus=0) (eabus=sum) (acc=0) (rbus=acc)
 (r*=rbus 0))              ; rc(5), r0=acc, acc=0

((mor=mbus) (mbus=areg))        ; mor=areg

((addr topscore) (mem=mbus) (mcondload) (xbus=0) (eabus=sum)
 (mbus=acc))               ; wc(topscore)=acc

((addr signal) (mor=mem) (xbus=0) (eabus=sum) (abus=mor)
 (bbus=mbus) (mbus=r* 0))               ; r(signal), acc=mor+r0

((acc=0) AND_MINUS) ; acc=0, AND_MINUS

((mcondload) (mem=mbus) (mbus=acc) (addr pitch) (abus=mor)
 (acc=abus) (xbus=0) (eabus=sum) SIP)  ; wc(pitch), acc=mor, SIP

((mcondload) (mem=mbus) (mbus=acc) (addr lp)
 (xbus=0) (eabus=sum) SIV)              ; wc(lp), SIV

((mcondload) (mem=mbus) (mbus=acc) (addr lv)
 (xbus=0) (eabus=sum))                  ; wc(lv)

((mem=mbus) (mbus=acc) (addr ls) (xbus=0) (eabus=sum)) ; w(ls)

((addr -6) (xbus=x* 2) (eabus=sum))                ; ea=x2-6
) ; end block3

(block4; IDLE
 ((nop))
)

(block5; handles x2 counter
  ((addr 0) (xbus=0) (eabus=sum) (x*=eabus 2))
 ) ; end block0

)
```