A BEHAVIORAL SIMULATOR FOR DSP ICS

by

Chuen-Shen Shung

Memorandum No. UCB/ERL 86/4

6 January 1986

A BEHAVIORAL SIMULATOR FOR DSP ICS

by

Chuen-Shen Shung

Memorandum No. UCB/ERL 86/4

6 January 1986

TITLE PAGE

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Table of Contents

Figure Contents

1. Introduction

As the complexity of integrated circuits (IC) keeps increasing, the need for simulation and design automation of IC chips before their fabrication is becoming increasingly significant. Simulations help find out design errors and can evaluate the chip performance without actually fabricating the chips. Some automatic design aids, such as *programmable logic array* (PLA) minimization programs or placement and routing packages, make life easier for desiners. Both the simualtion and automatic design tools help improve the productivity of IC chip design by greatly reducing the design turnaround time and are becoming indispensible in today's IC design methodology.

In the automatic layout generation system, or *Lager* for *La*yout *ge*nerator, which has been developed recently [1], it is possible for the signal processing specialists who know relatively little on chip design to generate IC chips automatically from behaviorial descriptions of the chips . Since our philosophy is to release these users of the need to know *how* the layouts are generated, the behaviorial descriptions must be high-level. The current Lager system uses a *register transfer level* (RTL) language called the *design file* to specify the chips' behavior. Users can simulate from the design files using a RTL simulator. The users prepare the input sample value file and let the RTL simulator emulate the chip behavior and generate the output file. If the output meets the specifications, then the users start the layout generation and fabriction process; otherwise, they can modify the designfile and simulate again.

However, users still have to know quite a few details when writing the chip design file. These details include the architecture, its arithmetic properties, and the datapath design of the resultant chips. The design file is similar to the assembly-level descriptions often used in microprogramming a processor. It is not easy to arrive optimized code even for experienced hardware designers, not to mention a novice to the Lager system.

Silage descriptions, on the other hand, requires less knowledge of the hardware than the design file descriptions. It is much easier to write Silage code from signal processing algorithms, which makes Silage more attractive to the users. Nevertheless, the Lager system has to do more to simulate and to generate layouts from the more abstract chip descriptions. This report describes such a behavioral-level simulator using Silage as the chip description language.

Silage is similar to other *high level languages* (HLLs) like C, Fortran and Pascal. And although painfully, we can use any of these HLLs to describe the chip behavior. There are two reasons we choose to develop a new language to do the job. First, we can optimize the language ourselves such that it fits our purposes. Second, Silage is a *functional* (applicative) language, which makes it more natural for describing signal processing algorithms.

The report is devided into 7 chapters. Chapter 2 talks about the Lager system in more detail. Chapter 3 describes the Silage and functional languages in general. Chapter 4 and 5 discuss and compare the two methods of implementation of the simulator, viz., the *demand-driven* approach and the *data-driven* approach. Chapter 6 deals with the arithmetic issues and gives some examples. Finally, Chapter 7 summarizes the report.

2. The Lager System

2.1. Overview

The Lager system[1][2] is an integrated set of computer-aided design (CAD) tools built on top of a cell library (CMOS and NMOS) to facilitate the signal processing IC designs. The goal is to allow signal processing specialists to be able to design IC chips . The system allows them to enter higher-level descriptions of the algorithms and automatically generates the chip layouts.

The flow diagram of current Lager system (without the Silage input) is depicted in Fig. 1. The highest level description language is the *design file*, which is specified by the users. A design file simulator called **Demon** is used to test the algorithms and to tune the coefficients. Demon has a debugging capability also, which allow users to find out any programming errors. After verifying the algorithms, the users can then run **Archer,** the architecture intepreter, which will extract architectural information from the design file and generate the *parameter description language* (pdl) file. The pdl file contains basically the parameters of each *macrocell* and the connectivity among these macrocells. A macrocell is a large block of circuitry containing perhaps several thousand transisters and is assembled by tiling small rectangular cells (leaf cells) in two dimensions. The parameters in the pdl file are used to direct the tiling (or module generation) process. There are three kinds of parameters at the moment, namely, the *name* of the macrocell (rom, ram, etc.), the *type* of the macrocell (column-decoded or non-column-decoded ram, for example) and the *width* of the macrocell (the word lendth of the datapath, the length of the control word, etc.).

Modgen is the tool that does the module generation from pdl files and produces the macroell layout file and the *hardware description language* (hdl) file. Modgen also refines the connectivity data in the pdl file by determining the real coordinates of each terminal of

each macrocell, and storing them in the hdl file. There is a placement, routing and graphical display tool, called **Flint,** to place the macrocells and to route them together using the connectivity data in the hdl file. Flint is implemented on Sun workstations. All other software tools can run on any machine that runs Unix (4.2 BSD).

To summarize, the Lager tools make possible quick turnaround time for designing chips from high level descriptions. The methodology has been used to design several signal processing chips, including a speech vocoder[3], 300 baud modem[4] and digital scrambler.

2.2. Architecture

The target architecture of the Lager system, as illustrated in Fig. 2, consists of several concurrent signal processors. Interprocessor communication is performed over bit-serial lines with parallel-serial (and serial-parallel) conversion and data buffering at the processor side. The present version allows only one of the processors to input and output sampled data via a bidirectional parallel bus at the sample rate. The chip may optionally have a host-IO unit to interface with a host processor at frame rate. The host-IO unit can connect with one or more processors through bit-serial lines.

Each processor is composed of seven macrocells (fig. 3). They are the AAU (address arithmetic unit), AUIO (arithmetic unit and i/o unit), PC (program counter), SPC (subprogram counter), ROM, RAM, and FSM (finite state machine). The AAU is used to do index addressing. There are two index registers, *ix* and *iy*. The ix-register counts the iterations of the subprogram. The iy-register is normally implemented as a counter with a user-defined modulus. The counter is incremented at the start of each sample period. This kind of indexing can be used for decimation. Another option is to implement the iy-register as a register, which can be loaded from the mbus. This allows for table lookup and memory processing.

AUIO is composed of the arithmetic unit and the i/o unit. The arithmetic unit is a pipelined datapath, as shown in fig. 4. The mor (memory output register), sor (shift register), acc (accumulator) and the mir (memory input register) are the four pipeline stages. The i/o unit consists of the connection to the i/o bus (for only one of the processors) and the serial interconnections with the other processors and the host i/o units. These serial i/o units include serial-to-parallel and/or parallel-to-serial converters and also the buffering latches for the global variables.

The FSM is the only place where some logical operations can be performed. The only type of conditional operation is the conditional write which is useful for decision-making type of applications. The input to the FSM can come from the sign-bit of accumulator, or any bit or bits of ix- and iy-registers. The output of the FSM is a set of control bits which are used to control the write-enable of the RAM. The FSM is implemented by a programmable logic array (PLA) which is also user-defined (by the designfile).

ROM, PC and SPC make up a very simple control sequencer. Looping is restricted to one level of subroutine call. Branching is prohibited. For every sample, each processor starts with the execution of a piece of microcode (the so-called main program) and may optionally execute a user-defined number of iterations of a single sub program. The main program and sub-program have fixed lengths and contain no branching. The ROM is where the main and sub microcodes are stored. The PC and SPC are used as cycle-counters for main and sub programs, respectively.

The Lager architecture, although very primitive, has proved to be sufficient for many signal processing applications. Because of its simplicity, four or five processors may fit into a single chip and thus achieves considerable power. The original scheme uses bit-parallel multiplications (Fig. 4), which takes a number of cycles equal to the wordlength. A new AUIO has been designed to use a Booth multiplication[5] scheme, which requires one half of that number. Effort is underway to enhance the control sequencer and the

conditional operations.

2.3. Lager -- from User's Point of View

The current Lager system requires the user to write the designfile, which is no harder than any assembly language programming except that it's much more difficult to optimize the code because of the pipelining. There are several constraints that the user needs to know. The total computational power is constrained by the product of the number of processors and the size of the microcode. The number of processors per chip is basically limited by the chip size and is usually less than five. The maximum number of microinstructions that one processor can perform for each sample is constrained by the ratio of clock rate to sampling rate. For example, our cell library is designed for a maximum clock rate of 5 MHz and for an application of 20 KHz sampling rate (digital audio), the maximum number of microinstructions is 400 cycles. If the chip has 5 processors, then it could compute 2,000 microinstructions per sample.

To partition the signal processing algorithms into several loosely-coupled sequences of computational tasks is by no means trivial. It is very important to balance the size of the microcode among the processors since they execute synchronously. Any processors requiring less time than others are in fact executing nops (no-operation) at the end of their program. The vocoder chip [3](Fig. 5) is a good example of program optimization and balanced design within these various constraints.

In terms of program development time it is usually best to be able to program in a high level language (HLL). The goal is that future users of Lager will be able to design in Silage, which is not only easier to write but also provide further abstraction of the architectural details. The Silage compiler will generate the design file, optimize the code to fully utilize the pipelining and (probably with some guidance by the user) should partition the code into several balanced pieces. An advantage of programming in a HLL is that

the *source* code can potentially be compiled into microcode for different architectures.

2.4. The Future Lager System

The Lager system has succeeded in converting a high level description into chip layouts. However, its application has the following four limitations:

(1) It is targeted for signal processing applications.

(2) Its architecture is fixed.

(3) The system doesn't have the intelligence to 'figure out' the best design, instead each design is a subset of some architecture.

(4) The underlying cell library is vulnerable to a change in the technology.

Since design automation is a very complicated process, it is mandatory to restrict the problem domain to achieve acceptable performance. It thus seems justifiable not to worry about the universality of the methodology for the moment. The third and fourth limitations are actually orthogonal to the Lager system in that they can be removed easily and independently of the current system. The third can be achieved by bringing in some efforts from the data path compiling area. The fourth can be done simply by rebuilding the cell library in procedural layout.

In my opinion, the second point above, the fixed architecture, is the only unnecessary restriction. Future Lager systems ought to be able to target into several different architectures since there is no fundamental reason why this can not be the case. In our plan, not only are we trying to map Silage into more than one architecture, but we want to be able to generate chips that are not microprogrammed. Higher sampling rate applications, such as required in image processing, are the other possibilities of extending the design space.

3. Silage and Functional Languages

In this chapter, I will emphasis the different nature of *Functional Languages* and conventional programming languages. The latter can also be called *Imperative Languages*. Silage will be cited as an example to illustrate those points.

3.1. Functional Languages

A summary of the differences of Imperative (Procedural) Languages and Functional (Applicative) Languages has been given [6] in the following way:

> A program in an Imperative Language is used to convey a list of commands, to be executed in some particular order, such that on completion of the commands the required behavior has been produced.
>
> A program in a Functional Language is used to define an expression which is the solution to a set of problems: this definition can then be used by a machine to produce an answer to a particular problem from the set of problems.

The two programming language types, Imperative and Functional, can be illustrated by the use of an analogy in which one is asked to specify a physical object. For example, to describe a garden shed, an imperative language programmer might say:

To build a shed

(1) Lay the foundation;

(2) Build the walls;

(3) Lay the floor;

(4) Put the roof on.

A functional langauge programmer may say:

A shed consists of

(1) Walls supported by the foundation;

(2) A floor supported by the foundation;

(3) A roof supported by the walls.

Looking closely into the two descriptions, we can see clearly that the first approach is more commanding (imperative) while the second one is more static and definitional. Another obvious difference is that the order of statements is important in an imperative approach but is not important in the functional approach. Functional languages do not have typical imperative control structures. This relieves the programmer of the burden of explicitly specifying the control flow, and reduces the errors which are introduced by incorrect control sequencing.

Signal-flow graphs (very similar to the data-flow graphs in compter science terminology) are heavily used in signal processing to represent digital networks and algorithms. Fig. 6 shows a simple first-order digital filter. The signal-flow graphs use arcs to represent the functions to be performed on the signals and vertices (and arcs actually) to represent the dependency of the signals. The data-flow graphs use vertices to represent the functions performed and arcs the dependency. Notice although different, there is a one-to-one correspondence between the signal-flow graph and the data-flow graph.

Because of the explicit and implicit concurrency of the signal-flow graph, it is necessary to over-specify the control flow in a conventional language description of a signal-flow graph. Since the programming of a functional language is basically nothing but recording the dependency in a consistent way, it avoids over-specificity. This is the most important reason why a functional language was chosen. choose Silage in the first place.

Technically speaking, a functional language is a language whose main body is nothing but function definitions and function applications. There are no global or static variables, and the local variables can only be called-by-value. There can be no assignment statements so the programmer often need some time to get used to using only equations. For example.

```
i = i + 1;
```

is not allowed, which makes a description of looping more complex In a conventional language, one would write

```
S := 0;
for i := 1 to 10 do S := S + A[i];
```

but in a functional language, one has to say

```
PS[0] = 0;
(i: 1..10) :: PS[i] = PS[i-1] + A[i];
S = PS[10];
```

In functional languages, variables are not used for storage, but instead are more like macro names for expressions. An identifier can be defined only once within a certain scope while allowing it to be refered as many times as desired.

Another difference of the two types of languages is the way they manipulate arrays and data structures. Arrays and data structures can be viewed two ways - either as single, large objects or as collections of small objects. In functional languages, arrays and data structures are almost always viewed as a single object, and some constructs that copy the entire array or substitute in new values for particular chosen elements are often provided. The programmer has to create new data structures with small changes from the original ones, rather than changing individual elements. This approach may sound more expensive but it has the advantage that the entire array or data structure can have one set of control information rather than a control block for each element of the data structure

or array.

3.2. Features in Design of Silage[7]

Being a functional language, Silage has almost all the essential properties mentioned above. Moreover, since we want Silage to be a special-purpose language for signal processing IC design, several features are designed for that purpose.

All data objects in Silage are actually infinite arrays indexed by an integer quantity that can be thought of as "time" or "sample number". Thus, when you see

```
Q = A + 1;
```

this is really a definition of a vector:

```
(n: -infinity.. +infinity) :: Q[n] = A[n] + 1;
```

We usually don't have to mention the index since all the samples are doing the same. Sometimes it is necessary to refer to a value computed during a previous sample. For this purpose, there is the notation

```
X@n
```

where n is a compile-time integer constant expression, to mean "the value of X, n samples ago."

Within each sample, a quantity may be an integer, a fixed-point number, a boolean, or an array of any of these. These *types* must be specified by the programmers. A quantity with value n and type **num <w,d>** represents a fixed point number of the form $n\,2^{-d}$, where $-2^w \leq n < 2^w$. The numeric type **num <1,0>** is *boolean* type, and the numeric type **num <w,0>** is an *integer*. Square brackets can be used along with type declaration to declare the dimension of an array. For example, **num[k]<w,d>** declares an array of k elements of type **num <w,d>** Moreover, **num <w>** is the same as **num <w,w>**, and **num** is used to specify fixed point numbers of arbitrary range and precision.

Defining functions in Silage is fairly simple. Suppose we want to write a function that adds one to each input, this is accomplished by:

```
func add1 ( in: num ): num =
begin
        return = in + 1;
end ;
```

Multiple-return functions are allowed. For example,

```
func dummy ( in: num )
        inc: num , dec: num ;
begin
        inc = in + 1;
        dec = in - 1;
end ;
```

is a function that defines two *subfunctions*. Function application is very flexible. $X.y$ refers to the subfunction y in the function definition of x. One can invoke any subfunction alone without altering the other subfunctions. The calls

```
out = add1 ( in );
```

and

```
out = dummy.inc ( in );
```

are exactly the same. You may notice these functions resemble macro definitions. In fact, they are macros in terms of implementation, so that when applying a function, it is a macro expansion rather than a function call. Looping is dealt with in the same way, so the sentence

```
(i: 1..3):: B[i] = B[i-1] + A[i];
```

is expanded to

B[1] = B[0] + A[1];

B[2] = B[1] + A[2];

B[3] = B[2] + A[3];

internally. This implementation does not allow recursive calls to the same function.

An entire program consists of a sequence of definitions of quantities and functions, including one definition of a function called *main*. The arguments and outputs of the function main comprise the inputs and outputs of the system.

Silage offers, besides arithmatic and logical operations, conditional operations for decision-making . A typical command looks like

y = **if** C_1 -> E_1 ‖ C_2 -> E_2 ‖ ... ‖ E **fi** ;

and means that y equals E_i if C_i is TRUE (C_i is boolean) or E if none of C_i is TRUE. In addition, Silage has special library functions like Interpolation and Decimation to facilitate the programming for signal processing applications.

There are several constructs in Silage which are designed especially for working with hardware generated by the Lager system. These are pragmatic directives whereby the programmers can specify some non-algorithmic information about their programs. The general format is

pragma *expression*

where *expression* is interpreted differently from elsewhere in the language. For instance,

pragma Processor(n,E1,E2,...)

means that those expressions defining E1, E2, ... have to be computed in processor n. Since the chip generated by Lager may have more than one processor, this *pragma* gives the programmer the opportunity to determine the computation allocation between the processors. The pragmas, **pragma** Stored(C) and **pragma** Implicit(C) can be used to direct Lager to assign local memory for the constant C or not. The pragmas may appear wherever a

definition may appear.

4. Demand-driven Simulation

4.1. Intermediate Format of Silage

For encapsulation purposes (see fig. 7), a Silage program will go through a *front-end* , in which the parsing and semantic analysis are done, and generate a intermediate format. The intermediate format will then go through various *back-ends* such as the compiler and the simulator. The intermediate format is represented by the so-called *decorated Abstract Syntax Tree* (d-AST). The plain (undecorated) AST emerges directly from the parser, and contains all the syntactical information of the Silage program. After the semantic processing, the semantic information of the Silage program is incorprated by augmenting the AST with various attributes, depending on the type of the node. These attributes can be thought of as hanging off the tree nodes and thus we say that the AST is *decorated*. Each node in the d-AST is a C structure defined as:

```
typedef struct NODETYPE {
        short LineNum, CharNnm;
        char *Filename;
        NodeKindType kind;
        struct NODETYPE *L; *R;
        AttrType *Attrs;
} NodeType;
```

The *LineNum* and *CharNum* indicate source position of text producing the node. The *Filename* indicates which file the source came from. *Kind* indicates the type of the node. For instance, the expression

$$E_1 + E_2$$

is represented by a tree whose root has a Kind entry of PlusNode, and whose left son is a subtree representing E_1 and whose right son is a subtree representing E_2. Later in this

report. I will use the notation

(PlusNode E_1 E_2)

to indicate the internal AST representation. There are various node types for arithmetic and logical operations, e.g. PlusNode, MinusNode, ..., LTNode, GTNode, ..., etc. There are node types for other language constructs. IdentLeaf, IntegerLeaf, and FixedLeaf are the types for leaf nodes, and each indicates an identifier, integer constant and fixed point constant, respectively. DefnNode is the type name of the equal sign because equations in Silage can be viewed as definitions.

Some semantic information, the numeric type (**num<w,d>**) for example, is stored as an attribute in the d-AST. The data structure of an attribute is defined by:

```
typedef struct ATTRTYPE {
        short NumAttrs;
        struct ATTR {
                short AttrId;
                short Type;
                int *Value;
        } Attrs[1];
} AttrType;
```

which is a variable length array of the structure *Attrs*. There are several types of attribute that display different types of information. Each node in the d-AST may have none, one or more attributes depending on the *kind* of the node. For example, the *numeric type* attribute exists for every IdentLeaf and operator node of any arithmetic or logic expressions. There is a pointer (Attrs) declared inside each node which points to the attribute data structure (AttrType). If the node doesn't have any attribute, the pointer simply stores NULL.

4.2. Traversing the Decorated AST

Consider the scenario that a symbol (of *kind* IdentLeaf) is defined once and is referenced several times in a certain scope. If we were to define an IdentLeaf of every occurence of the symbol, then it would be extremely difficult to pass information between these IdentLeaves. So in the d-AST every symbol has one unique IdentLeaf associated with it. This makes the d-AST not a tree (because of the node sharing), but rather a *Directed Acyclic Graph* (DAG). Actually it is a general directed graph when attributes are taken into account. The traversing of a DAG is more difficult than traversing a tree, but that's the trade-off that is made to simplify the manipulation of each symbol.

Because the common symbols in our d-AST are only leaf nodes, the traversing problem can be accomplished by small amendments to a tree-treversing algorithms. First we need to build a symbol table to include all the symbols. Then we traverse the d-AST as if we were traversing a tree. If we encounter an IdentLeaf in the symbol table, we check to see if the symbol was labelled. If yes, we stop; if not, we label it and visit that IdentLeaf.

However, since the d-AST is not *threaded*, in general we can only traverse it top-down but can't do it bottom-up. As we will see later, demand-driven simulation involves top-down traversing while data-driven simulation requires bottom-up traversing of the d-AST. We will have to use more data structures in data-driven simulation to overcome the traversing problem.

4.3. Demand-driven Approach

Given the d-AST representation of the Silage program, the simulation task can be viewed as evaluating (interpretating) the d-AST. There are two approaches in this evaluation, the *demand-driven* approach and the *data-driven* approach. The demand-driven approach starts at the root of the d-AST and evaluation of each node proceeds by a

demand for its left son and right son to be evaluated in the proper order. For example, when PlusNode is evaluated, it first demands that both the left and right son be evaluated, and then adds the two results from the two sons to get the result for PlusNode itself. When DefnNode is evaluated, it requests its right son to be evaluated and passes the result to the left son. The entire simulation is done by simply forwarding the *demand* of evaluation from the root down to the leaves and at the same time collecting the result of evaluation from the leaves to the root. All the input nodes are located among the leaf nodes of the d-AST, and all the output nodes should be the left sons of some DefnNodes. For each input sample, the evaluation process is done once. Output samples are generated when their parent DefnNodes are evaluated.

An attribute slot called Result is specially designed to store the temporary evaluation result for each node. *GetAttr* and *SetAttr* are two macros devised to access the attributes and can be invoked as:

```
out = GetAttr ( T, Result );
SetAttr ( T, Result, out );
```

The former gets the *Result* attribute of node *T* and assigns it to *out*. The latter sets the *Result* attribute of node *T* to be *out*.

The evaluation process is basically a big recursive routine called NodeProcess. At this beginning of the routine, a switch statement selects the proper code segment, depending on the kind of the node which is encountered. For example, if a node has a kind PlusNode is being evaluated, the simulator would be:

```
case PlusNode:
        NodeProcess ( T->L );
        result_left = GetAttr ( T->L, Result );
        type_left = GetAttr ( T->L, Type );
        NodeProcess ( T->R );
```

```
result_right = GetAttr ( T->R, Result );
type_right = GetAttr ( T->R, Type );
out = Plus ( result_left, type_left, result_right, type_right);
SetAttr ( T, Result, out );
break;
```

The Plus function will be discussed in Chapter 5. The code segment for DefnNode is:

```
case DefnNode:
        NodeProcess ( T->R );
        result_right = GetAttr ( T->R, Result );
        SetAttr ( T->L, Result, result_right );
        break;
```

For simplicity, the case when the numeric types of the right and left son are different has been omitted. For each input sample, we evaluate the d-AST once by calling Nodeprocess at the root which in turn generates calls to traverse the remainder of the d-AST.

Most signal processing algorithms involve z^{-1} delays. This is represented internally by the DelayNode . Although theoretically every symbol in Silage is an infinite array, there is no way we can directly handle such arrays. Before run-time (before calling NodeProcess), we traverse the d-AST once to determine the maximum number of delays defined the Silage program. This can be done because the delay number (located at the right son of each DelayNode) is manifest (known at compile-time). We dynamically allocate storage in the simulator (*malloc()*) for each symbol with an array whose length equals the maximum delay. Before evaluating the d-AST, we preset all the past samples of every symbol to zero; after evaluating the d-AST once, we shift all past samples one step into the past and throw the oldest sample away.

The array symbols are the most complicated to handle. A one-dimensional (1-d) array is stored internally as a two-dimensional (2-d) since each element in the array is

itself an array of present and past samples. Also, since an array is viewed as a single object, only one symbol appears in the d-AST. Thus the subtrees

(ArrayIndexNode, a, 2)

and

(ArrayIndexNode, a, 3)

which stands for a[2] and a[3] respectively are two distinct trees with a common left son. The GetAttr and SetAttr of ArrayIndexNode, however, has to manipulate the same, single object. In other words, there is a 2-d array for the array symbol *a* which is commonly refered for any element, present or past, of the array. Address calculation is required in the 2-d array. For example,

a[2] @ 3

is stored at the 20th slot in the 2-d array if the maximum delay is 8. When a symbol is an array name, the Result attribute stores the pointer pointing to the 2-d array where all the values are located. There are special attribute macros for addressing into the array:

out = GetAttr2 (T, Result, n, d);

SetAttr2 (T, Result, n, d, out);

get and set the Result attribute of a[n] @ d (T is the node of array symbol, or *a* in our example).

5. Data-driven Simulation

5.1. Data-driven Approach

We have seen the essentials of evaluating a d-AST using a demand driven approach. Another technique views the problem of evaluating the d-AST from the *supply* point-of-view and leads to quite different method of implementing the simulator. Consider the d-AST with input or constant leaf nodes at the bottom and with an output node at the top, the values at the leaf nodes, which were demanded by their parents, were in turn demanded by higher ancestors themselves. Now, if the leaf nodes are to supply their values to their parents , and if these parents could supply, after interaction (computation) their values upward, the evaluation result would eventually get to the output node. In other words, the important thing is that the result propagate to the output node rather than how this propagation is initiated.

Let's use an example to illustrate how it works. Suppose we are to evaluate

```
(DefnNode, y,
        (PlusNode, a, b)
)
```

and *a* is known, but *b* is not. The node *a* will supply its value to the PlusNode so the PlusNode has one operand. Since the node *b* does not yet have a value to give to the PlusNode, the calculation at the PlusNode can not proceed. Once node *b* finally gets its value and gives it to the PlusNode, the calculation at PlusNode is made and result is supplied upward to DefnNode. Then the DefnNode will take the value at the PlusNode and bring it to node *y*, and then every operation pending the value of *y* can proceed. The entire evaluation process is triggered by data, so we call it a *data-driven* approach.

Since the traversing is upwards, the data-driven approach requires a node to know its parent. Also as the node *a* is waiting for the value from the node *b* in the above example, care must be taken to not lose the value of *a* These aspects reflect that the data structure which was designed for demand-driven simulation is no longer sufficient.

5.2. Data Structure

It was decided not to use the previous node structure, NodeType. Rather, a set of new data structure was devised in which every node of the d-AST comes in as an element of a big array, *TL* (for treelist). A node in the d-AST, which was previously referred by a pointer to the structure *NodeType* in the demand-driven approach, is now referred by an index of TL in the data-driven approach. The TL is created by reading in the d-AST and fitting it into the TL array, one entry per node. It takes extra time and memory space to do this, but it has the advantage that the data structure of the back-end (simulator) is independent of that of the front-end (intermediate format, or the d-AST). Each node inthe TL is defined by:

```
typedef struct treelist {
        short up;        /* interpolation factor */
        short down;      /* decimation factor */
        short count;     /* how many more operands */
                         /* still needed before fire-up */
        short Isconst;   /* boolean, TRUE if const */
        short Iswait;    /* boolean, TRUE if exist a waitoperand */
        int waitoperand; /* value of waitoperand */
        NodeType *ptr;/*
        Parent *phead;/*
        Valueq *vhead;/*
        Valueq *vtail;/*
```

```
        short arlen:      /* array length. equal 1 if scalar */
        Parent *pp[1];/*
                          /* non-NULL only for array-type identleaf */
} TL[PROGRAMSIZE];
```

Since a node may have more than 1 parent (because of the common nodes), a linked-list is defined to keep the parent information:

```
typedef struct parent {
        int ParentIndex;      /* index of parent in the TL */
        short IsRight;        /* boolean, TRUE if the node is */
                              /* the right child of the parent */
        struct parent *pnext;/*
} Parent;
```

The parent information is created along with TL initially when the d-AST was read in. The parent list of an array-type identleaf is special (deviced by *pp*) in that parents with different array index need to be treated differently. Thus an array (length equals *arlen*) of parent list pointers are created for these array-type identleaves.

Each node has a *count* field to determine how much more data is still needed before it can operate and give the result. Initially, the count field is at the maximum. For example, the count field for PlusNode is 2 since it needs two data to operate. For DefnNode it is 1. For DelayNode it is 0 since it is assumed that all the past values are zero and are available initially. If right or left child of a node is available or is a constant, then the count field decrements. When the count field equals zero, the operation is carried out and the count fields of the children are reset to the maximum value. The result of the operation (e.g. Plus) will again trigger its parents' count field to be decremented. The process then repeats for the parents.

The temporary value of a node is kept in the *Valueq* (Value queue), not in the Result attribute. The advantages for this are that the length of the Valueq is variable, thus every symbol needs not reserve the maximum number of past samples. Secondly, the past values are stored at the Valueq of the DelayNode rather than at the symbol sites. Address calculation for past samples is then not needed. For a DelayNode with delay number n, a Valueq of length n is initially allocated and initialized to all zeros. In all cases, a new value enters the *tail* of the Valueq and the output value is taken from the *head*. *Vhead* and *vtail* in the TL are used for this purpose. Each element in a Valueq is defined as:

```
typedef struct valueq {
        int value;
        struct valueq *vnext;
} Valueq;
```

If a symbol has more than 1 parent, when it has got the value ready for its parents, it is often the case that some parents are ready to use the value (their count fields will be zero) while some parents are not. If we let parents wait until all parents are ready, those parents that are currently not ready may in fact wait for those parents who are ready, and then a deadlock occurs. On the other hand, if we let those parents who are ready use the value freely, when a parent is ready later it could find that the value is gone. The solution to this is to send the value to all parents, ready or not ready. The ready parents can use the value immediately; the parents which are not ready can keep the value pending by storing it in the *waitoperand* field. The *Iswait* is a boolean variable to indicate if there is a waiting operand. These complexities exist partly in order to simplify the manipulation of the Valueq.

Constants are another trouble-maker. Since constants can be viewed as everlasting sources of data, we must be careful not to treat them as other ordinary symbols. The solution is to propagate the constants right at the beginning of the evalution. If a node has

one child which is constant, then the constant value will be used as a *waitoperand* and the maximum count of the node (the number to reset to) will be only 1. If both children are constants, computation is carried out and the node is degenerated to an constant leaf. The *Isconst* in TL is a boolean variable to indicate if a node is a constant.

The *up* and *down* are used to record the sampling frequency ratio of the node. When interpolation and decimation are parts of the program, there is more than one sampling frequency. The sampling frequency ratio is defined as the ratio of the sampling frequency to the input sampling freqnency and can be represented by a rational number that is stored as two integers, *up* and *down*. *Ptr* is the pointer pointing to the NodeType structure. *Arlen* indicates if the node is an array symbol. If it is not, *arlen* equals unity and *pp* equals NULL, otherwise *pp* points to the array structure.

After the data structure is set up, the evaluation algorithm is fairly simple. In the initialization phase, TL and Parent structures are created, then the constants are handled and Valueqs for DelayNodes are allocated. An *Availq* (available queue) is defined to store all the nodes whose count fields are zeros. At the beginning, only DelayNodes are in the Availq. Each element in the Availq is fetched sequentially and the following process is executed:

> For each parent of a node *A*, decrement the count field by 1. If the count field reaches zero, take the value of the child, make the appropriate computation and store the result in the Valueq. Meanwhile put the node *A* into the Availq. If the count field is not zero yet, take the value and put it onto the waitoperand field and set the Iswait field.
>
> After every parent is finished, the node *A* leaves the Availq with its count field reset to initial maximum number. However, delayNode would be reset to 1 even though it is 0 initially.

Whenever the Availq is empty, read in a new input sample and put the input symbol onto the Availq. Whenever the output node gets anything to its Valueq, output it.

The most remarkable part of the algorithm is that it needs no bookkeeping for Valueq, since everything is done in a first-in-first-out basis.

5.3. Comparison of the Two Approaches

We have noticed that the data-driven approach requires a more complex data structure. Furthermore, it runs slower. This is because the large amount of queue manipulation and the frequent searching through the big TL array. Why do we find it interesting after all? The answer is *flexibility*.

One feature of functional languages is that the statements need not obey a certain order. However, at execution time, the proper order is very important and this is obtained through a data-flow analysis. In Silage, this is part of the semantic analysis and the d-AST is a result of this ordering. The demand-driven approach relies on this fact since the forwarding of evaluation demand is a rigid process which has to obey a predefined order. On the contrast, the data-driven approach does not require the built-in execution order. It will work as long as the d-AST reflects the Silage program syntactically. However, the current implementation did not take the advantage because we use the same d-AST to create the TL.

Notice there is no explicit need to implement the z^{-1} delay in the data-driven approach. That is, no bookkeeping is necessary when the new input samples are read in. Each new value entering at the tail of the Valueq makes all the values previously in the Valueq "one time step older" automatically. This property makes the handling of asynchronous input almost painless since the evaluation process is not "synchronous" to any input. In the same light, the handling of conditional expressions are also easier in the data-driven approach because asynchronous inputs can be treated as special cases of

conditional expressions.

The statement

Y = Interpolate (A, N);

means that Y is defined by the result of interpolating the values A[0].....A[N-1] in that order where A is an array. The statement

pp = Decimate (X, N, P);

means that *pp* is defined as the result of sampling every N^{th} element of X, starting with the one at time Pr, where *r* is the sample interval of X. The sampling frequency of Y is N times that of A, and the sampling frequency of pp is 1/N times of that of X. When dealing with these different sampling rates among symbols, the demand-driven approach will require a more complex address calculation scheme. For example, a reference like

a = X @ k ;

is internally treated as

a = A [k-(k/N)*N] @ (k/N) ;

and

a = pp @ k ;

is treated as

a = X @ (k*N-P) ;

Also the maximum number of past samples to be allocated is no longer the 'literal' maximum delays of any sample, rather it is determined also by the sampling frequency. The lower sampling frequency a symbol has, the more past delay samples it should reserve. Surprisingly, the same decimation and interpolation problems cause no extra effort in the data-driven approach. For those symbols with higher sampling rate, the Valueq will expand and shrink more rapidly. Again, no bookkeeping is necessary for Valueq.

Last but not least, data-driven approach is proven to be essential if the evaluation process is to be scheduled into small tasks when we have multi-processor enviroment, or single processor with instruction pipelining[9].

6. Arithmetic

6.1. Lager Arithmetic

Lager make use of 2's complement binary arithmetic. Every data object, independent of wordlength, has value between -1 and 1. Data are interpreted as having a binary point after the most significant bit (MSB), which is also the sign bit. This format has the advantage that the binary point position is the same for operands and result, and hence leads to consistent interpretation of data objects[10]. Signal samples as well as the coefficients and parameters of digital filters should be properly scaled before being sent to Lager chips. Some algorithms, like the lattice filter, bound the coefficient values between -1 and 1 hence no scaling is necessary. If the computation leads to values greater than 1 or less than -1, the partial result is satuated at 1 or -1.

Miltiplication is implemented by bit-parallel shift-and-add. There are two kinds of multications in Lager. Variable multiplication is done by a subroutine-like piece of code which basically implements the equation

$$xy = -xy_0 + \sum_{i=1}^{n-1} \frac{x}{2_i} y_i$$

where x is the multiplicand and y is the multiplier. Thus, if y is negative (y_0 is 1), -x is first put into the accumulator and in successive cycle that follows, y_i is being tested while x is right shifted. If y_i is one, then shifted x is added. The entire length of the multiplication routine is n+3, where n is the wordlength of the multiplier. Variable multiplication is expensive in a sense that lots of cycles are spent regardless of how many adds (the number of 1's in the multiplier) are actually required. Moreover, the Lager hardware dictates that the multiplier must be stored in a global variable before shift-and-adds can be performed and global storage is very expensive in terms of silicon area in the Lager implementation.

Other alternatives are feasible when the multiplier is a known constant. Then we can make use of the *barrel shifter* of the Lager arithmetic unit to shift over a string of zeros. Optimum usage of the shift-and-add cycles is when the multiplier is formated into the *canonical sign digit* (CSD) representation [11]. The CSD format has trinary symbols for each bit, 0, 1 and -1 (depicted by -). For example, the 8-bit constant

00111100

in binary format is equivalent to

01000-00

in CSD format. The latter has fewer "active" (nonzero) bits than the former so that the CSD representation can result in fewer microcode cycles. The results from constant or variable multiplication are different because of their different truncation properties. Although more expensive, we can choose to use variable multiplication when the multiplier is a constant. It is done by simply loading the constant to a temporary local and then following the variable multiplication routine.

The division operation in Lager is by means of *long division*. Each cycle a bit is generated so the number of cycles depends linearly on the wordlength of the quotient. The quotient is also stored in a global.

6.2. Implementation

As explained in Chapter 3, each data object has a numeric type, which is treated as an attribute. Inside the attribute a pointer is stored which points to a node having the structure

(NumTypeNode, IntegerLeaf, IntegerLeaf)

The first (left) IntegerLeaf stores the *w* and the second (right) IntegerLeaf stores the *d* of the numeric type **num <w,d>**. Each function in the arithmetic library (Plus, Minus, Mult

and Div) will take 4 parameters, namely, the values and the type pointers of the two operands (except for Mult which has three as explained below). Based on the 4 parameters, the arithmetic functions determine the resultant value of the operator using the following semantic rules as defined in the Silage:

Expression	Type of Result
L+R L-R	**num** <w,d>, assuming w=*w'*, d=*d'*
L*R L/R	**num** <w,d+*d'*-w>, assuming w=*w'* **num** <w,d+w-*d'*>, assuming w=*w'*
M+L, L+M M-L, L-M M*L, L*M M/L, L/M	**num** <w,d>
M+N, M-N M*N, M/N	**num**
Notes: M and N are type **num**. L is type **num** <w,d>. R is type **num** <*w'*, *d'*>.	

Thus, in each arithmetic function there is no need to set the *Type* attribute. Rather, the resultant value has to be properly scaled to fit the numeric type. The drawback of this approach is that it enforces some type relation between the two operands. Note if we had let the back-end simulator figure out both the value and the numeric type, it might require that new NumTypeNodes be declared and included in the d-AST.

The Mult function has one more parameter than other functions which determines whether variable multiplication or CSD constant multiplication is to be used. In every function, special effort is made to make the result "inaccurate" in the same (truncation) pattern as that would happen in the Lager architecture. The encapsulation of the arithmetic functions from other parts of the simulator makes it possible that we may simulate different architectures by simply changing the arithmetic routines.

6.3. Examples

In appendix A, a Silage program example of a low-pass filter is given and run through the two versions of simulator, *demandriven* and *datadriven*. A flattened (for the ease of printout) AST is also included in appendix A. In appendix B, the designfile program of the same low-pass filter example is shown which is run by Demon. Note the results are the same for Silage simulator and Demon.

7. Conclusion

This report describes a behavioral-level simulator which uses Silage to specify the chip behavior. Armed with this high level simulator and the Silage compiler, the users of Lager system would not need to write the designfile and use the RTL Lager simulator (Demon) any more. The simulator can be used to simulate the behavior of more than one architectures by simply replacing the arithmetic module.

It is instructive to study the different approaches of implementing the simulator. We compared the relative merits of the demand-driven approach and data-driven approach in terms of speed, run-time storage and flexibility.

A final note: Silage has been changed from the version reported here. Interested reader is encouraged to consult with Prof. Paul Hilfinger.

Reference

(1) J. Rabaey, "Lager : An Automated Layout Generating System for Digital Signal Processing Circuits", User Manual V1.3, Memorandum No. UCB/ERL M85/5, U. C. Berkeley, Feb. 1985

(2) S. Pope, J. Rabaey, R. Brodersen, "Automated Design of Signal Processing ICs Using Macrocells" in VLSI Signal Processing, pp 239-251, IEEE Press 1984

(3) S. Pope, "Automatic Generation of Signal Processing ICs", Ph.D. Thesis, Memorandum No. UCB/ERL M85/11, Feb. 1985

(4) W. L. Abbott, "Design of A 300-baud FSK Modem Using Customized Digital Signal Processors", M.S. Thesis, Memorandum No. UCB/ERL M84/93, August 1984

(5) Anton Stoelzle, "Multiplication-Unit for the LAGER System Implementing a Modified 2nd Order Booth-Algorithm", internal documentation

(6) H. Glaser, C. Hankin, D. Till, "Principles of Functional Programming", Prentice-Hall, 1984

(7) P. Hilfinger, "Silage: A Language for Signal Processing", internal documentation

(8) P. Hilfinger, "Intermediate Representation Scheme for Silage", internal documentation

(9) E. Lee, D. Messerschmitt, "A Coupled Hardware and Software Approach for Programmable Digital Signal Processors", paper in preparation

(10) A. W. Burks, H. Goldstein, J. von Neumann, "Preliminary discussion of the Logical Design of an Electronic Computing Instrument", Report to the U.S. Army Ordinance Dept., 1946

(11) G. Rietweisner, "Binary Arithmetics", Advances in Computers, vol.1, Acadamic Press, N.Y. 1960

Appendix A: Silage Program Example

```
Script started on Mon Nov 25 10:23:08 1985
oz 1% cat FIR.sl
#define word num<8>

my_coefs = { 5, 7, 8, 9, 12, 16, 27, 81, -81, -27, -16, -12, -9, -8, -7, -5};

func main(in: word): word =
begin
    return = h_transform(in,my_coefs);
end;

func h_transform(in: word; coefs: num []): word =
begin
    filter[15] = in;
    (i: 0 .. 14):: filter[i] = filter[i+1]@2;
    Sum[16] = 0;
    (i: 0 .. 15):: Sum[i] = Sum[i+1] + filter[i]*coefs[i];
    return = Sum[0];
end;
oz 3% cat infile
127
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
oz 4% demandriven FIR.il < infile
Silage Simulator Version 2.1 (Mon Jul 15 16:14:49 PDT 1985)
Identifier my_coefs
Identifier filter
Identifier in
Identifier Sum
Identifier return
Identifier return
Identifier return
-10
```

```
0
-11
0
-11
0
-14
0
-15
0
-18
0
-30
0
-85
0
oz 5% datadriven FIR.il < infile
Silage Simulator Version 2.3 (Wed Aug  7 17:15:47 PDT 1985)
-10
0
-11
0
-11
0
-14
0
-15
0
-18
0
-30
0
-85
0
oz 6% exit
script done on Mon Nov 25 10:26:33 1985
/*******************************************************
 *
 * the following is the flattened AST representation
 *
 *******************************************************/
oz 2% ~hilfingr/bin/prntTree FIR.il
#1      ProgramNode  L: #2, R: -*-,
#2      SemiNode  L: #3, R: #4,
#3      SemiNode  L: #5, R: #6,
#4      DefnNode  L: return#7, R: return#8,
#5      SemiNode  L: #9, R: #10,
#6      DefnNode  L: return#8, R: return#11,
#7      return#7 Type: #12, ValueClass: 24,
#8      return#8 Type: #12, ValueClass: 16,
#9      SemiNode  L: #13, R: #14,
#10     DefnNode  L: return#11, R: #15,
#11     return#11 Type: #12, ValueClass: 16,
#12     NumTypeNode  L: (8), R: (8),
#13     SemiNode  L: #16, R: #17,
#14     DefnNode  L: #18, R: #19,
#15     ArrayIndexNode  L: Sum#20, R: (0), Type: #12,
```

```
#16     SemiNode  L: #21, R: #22,
#17     DefnNode  L: #23, R: #24,
#18     ArrayIndexNode  L: Sum#20, R: (0), Type: #12,
#19     PlusNode  L: #25, R: #26, Type: #12,
#20     Sum#20 Bounds: #27, Type: #28, ValueClass: 16,
#21     SemiNode  L: #29, R: #30,
#22     DefnNode  L: #31, R: #32,
#23     ArrayIndexNode  L: Sum#20, R: (1), Type: #12,
#24     PlusNode  L: #33, R: #34, Type: #12,
#25     ArrayIndexNode  L: Sum#20, R: (1), Type: #12,
#26     MultNode  L: #35, R: #36, Type: #12,
#27     CommaNode  L: (0), R: #37,
#28     ArrayTypeNode  L: #38, R: -*-,
#29     SemiNode  L: #39, R: #40,
#30     DefnNode  L: #41, R: #42,
#31     ArrayIndexNode  L: Sum#20, R: (2), Type: #12,
#32     PlusNode  L: #43, R: #44, Type: #12,
#33     ArrayIndexNode  L: Sum#20, R: (2), Type: #12,
#34     MultNode  L: #45, R: #46, Type: #12,
#35     ArrayIndexNode  L: filter#47, R: (0), Type: #12,
#36     ArrayIndexNode  L: my_coefs#48, R: (0), Type: #12,
#37     CommaNode  L: (16), R: -*-,
#38     NumTypeNode  L: (8), R: (8),
#39     SemiNode  L: #49, R: #50,
#40     DefnNode  L: #51, R: #52,
#41     ArrayIndexNode  L: Sum#20, R: (3), Type: #12,
#42     PlusNode  L: #53, R: #54, Type: #12,
#43     ArrayIndexNode  L: Sum#20, R: (3), Type: #12,
#44     MultNode  L: #55, R: #56, Type: #12,
#45     ArrayIndexNode  L: filter#47, R: (1), Type: #12,
#46     ArrayIndexNode  L: my_coefs#48, R: (1), Type: #12,
#47     filter#47 Bounds: #57, Type: #28, ValueClass: 16,
#48     my_coefs#48 Bounds: #57, Type: #58, ValueClass: 144,
#49     SemiNode  L: #59, R: #60,
#50     DefnNode  L: #61, R: #62,
#51     ArrayIndexNode  L: Sum#20, R: (4), Type: #12,
#52     PlusNode  L: #63, R: #64, Type: #12,
#53     ArrayIndexNode  L: Sum#20, R: (4), Type: #12,
#54     MultNode  L: #65, R: #66, Type: #12,
#55     ArrayIndexNode  L: filter#47, R: (2), Type: #12,
#56     ArrayIndexNode  L: my_coefs#48, R: (2), Type: #12,
#57     CommaNode  L: (0), R: #67,
#58     ArrayTypeNode  L: #68, R: -*-,
#59     SemiNode  L: #69, R: #70,
#60     DefnNode  L: #71, R: #72,
#61     ArrayIndexNode  L: Sum#20, R: (5), Type: #12,
#62     PlusNode  L: #73, R: #74, Type: #12,
#63     ArrayIndexNode  L: Sum#20, R: (5), Type: #12,
#64     MultNode  L: #75, R: #76, Type: #12,
#65     ArrayIndexNode  L: filter#47, R: (3), Type: #12,
#66     ArrayIndexNode  L: my_coefs#48, R: (3), Type: #12,
#67     CommaNode  L: (15), R: -*-,
#68     NumTypeNode  L: -*-, R: -*-,
#69     SemiNode  L: #77, R: #78,
#70     DefnNode  L: #79, R: #80,
#71     ArrayIndexNode  L: Sum#20, R: (6), Type: #12,
```

```
#72     PlusNode  L: #81, R: #82, Type: #12,
#73     ArrayIndexNode  L: Sum#20, R: (6), Type: #12,
#74     MultNode  L: #83, R: #84, Type: #12,
#75     ArrayIndexNode  L: filter#47, R: (4), Type: #12,
#76     ArrayIndexNode  L: my_coefs#48, R: (4), Type: #12,
#77     SemiNode  L: #85, R: #86,
#78     DefnNode  L: #87, R: #88,
#79     ArrayIndexNode  L: Sum#20, R: (7), Type: #12,
#80     PlusNode  L: #89, R: #90, Type: #12,
#81     ArrayIndexNode  L: Sum#20, R: (7), Type: #12,
#82     MultNode  L: #91, R: #92, Type: #12,
#83     ArrayIndexNode  L: filter#47, R: (5), Type: #12,
#84     ArrayIndexNode  L: my_coefs#48, R: (5), Type: #12,
#85     SemiNode  L: #93, R: #94,
#86     DefnNode  L: #95, R: #96,
#87     ArrayIndexNode  L: Sum#20, R: (8), Type: #12,
#88     PlusNode  L: #97, R: #98, Type: #12,
#89     ArrayIndexNode  L: Sum#20, R: (8), Type: #12,
#90     MultNode  L: #99, R: #100, Type: #12,
#91     ArrayIndexNode  L: filter#47, R: (6), Type: #12,
#92     ArrayIndexNode  L: my_coefs#48, R: (6), Type: #12,
#93     SemiNode  L: #101, R: #102,
#94     DefnNode  L: #103, R: #104,
#95     ArrayIndexNode  L: Sum#20, R: (9), Type: #12,
#96     PlusNode  L: #105, R: #106, Type: #12,
#97     ArrayIndexNode  L: Sum#20, R: (9), Type: #12,
#98     MultNode  L: #107, R: #108, Type: #12,
#99     ArrayIndexNode  L: filter#47, R: (7), Type: #12,
#100    ArrayIndexNode  L: my_coefs#48, R: (7), Type: #12,
#101    SemiNode  L: #109, R: #110,
#102    DefnNode  L: #111, R: #112,
#103    ArrayIndexNode  L: Sum#20, R: (10), Type: #12,
#104    PlusNode  L: #113, R: #114, Type: #12,
#105    ArrayIndexNode  L: Sum#20, R: (10), Type: #12,
#106    MultNode  L: #115, R: #116, Type: #12,
#107    ArrayIndexNode  L: filter#47, R: (8), Type: #12,
#108    ArrayIndexNode  L: my_coefs#48, R: (8), Type: #12,
#109    SemiNode  L: #117, R: #118,
#110    DefnNode  L: #119, R: #120,
#111    ArrayIndexNode  L: Sum#20, R: (11), Type: #12,
#112    PlusNode  L: #121, R: #122, Type: #12,
#113    ArrayIndexNode  L: Sum#20, R: (11), Type: #12,
#114    MultNode  L: #123, R: #124, Type: #12,
#115    ArrayIndexNode  L: filter#47, R: (9), Type: #12,
#116    ArrayIndexNode  L: my_coefs#48, R: (9), Type: #12,
#117    SemiNode  L: #125, R: #126,
#118    DefnNode  L: #127, R: #128,
#119    ArrayIndexNode  L: Sum#20, R: (12), Type: #12,
#120    PlusNode  L: #129, R: #130, Type: #12,
#121    ArrayIndexNode  L: Sum#20, R: (12), Type: #12,
#122    MultNode  L: #131, R: #132, Type: #12,
#123    ArrayIndexNode  L: filter#47, R: (10), Type: #12,
#124    ArrayIndexNode  L: my_coefs#48, R: (10), Type: #12,
#125    SemiNode  L: #133, R: #134,
#126    DefnNode  L: #135, R: #136,
#127    ArrayIndexNode  L: Sum#20, R: (13), Type: #12,
```

```
#128    PlusNode  L: #137, R: #138, Type: #12,
#129    ArrayIndexNode  L: Sum#20, R: (13), Type: #12,
#130    MultNode  L: #139, R: #140, Type: #12,
#131    ArrayIndexNode  L: filter#47, R: (11), Type: #12,
#132    ArrayIndexNode  L: my_coefs#48, R: (11), Type: #12,
#133    CommaNode  L: #141, R: #142,
#134    DefnNode  L: #143, R: #144,
#135    ArrayIndexNode  L: Sum#20, R: (14), Type: #12,
#136    PlusNode  L: #145, R: #146, Type: #12,
#137    ArrayIndexNode  L: Sum#20, R: (14), Type: #12,
#138    MultNode  L: #147, R: #148, Type: #12,
#139    ArrayIndexNode  L: filter#47, R: (12), Type: #12,
#140    ArrayIndexNode  L: my_coefs#48, R: (12), Type: #12,
#141    CommaNode  L: #149, R: #150,
#142    DefnNode  L: #151, R: (0),
#143    ArrayIndexNode  L: Sum#20, R: (15), Type: #12,
#144    PlusNode  L: #152, R: #153, Type: #12,
#145    ArrayIndexNode  L: Sum#20, R: (15), Type: #12,
#146    MultNode  L: #154, R: #155, Type: #12,
#147    ArrayIndexNode  L: filter#47, R: (13), Type: #12,
#148    ArrayIndexNode  L: my_coefs#48, R: (13), Type: #12,
#149    CommaNode  L: #156, R: #157,
#150    DefnNode  L: #158, R: #159,
#151    ArrayIndexNode  L: Sum#20, R: (16), Type: #12,
#152    ArrayIndexNode  L: Sum#20, R: (16), Type: #12,
#153    MultNode  L: #160, R: #161, Type: #12,
#154    ArrayIndexNode  L: filter#47, R: (14), Type: #12,
#155    ArrayIndexNode  L: my_coefs#48, R: (14), Type: #12,
#156    CommaNode  L: #162, R: #163,
#157    DefnNode  L: #164, R: #165,
#158    ArrayIndexNode  L: filter#47, R: (14), Type: #12,
#159    DelayNode  L: #166, R: (2),
#160    ArrayIndexNode  L: filter#47, R: (15), Type: #12,
#161    ArrayIndexNode  L: my_coefs#48, R: (15), Type: #12,
#162    CommaNode  L: #167, R: #168,
#163    DefnNode  L: #169, R: #170,
#164    ArrayIndexNode  L: filter#47, R: (13), Type: #12,
#165    DelayNode  L: #171, R: (2),
#166    ArrayIndexNode  L: filter#47, R: (15), Type: #12,
#167    CommaNode  L: #172, R: #173,
#168    DefnNode  L: #174, R: #175,
#169    ArrayIndexNode  L: filter#47, R: (12), Type: #12,
#170    DelayNode  L: #176, R: (2),
#171    ArrayIndexNode  L: filter#47, R: (14), Type: #12,
#172    CommaNode  L: #177, R: #178,
#173    DefnNode  L: #179, R: #180,
#174    ArrayIndexNode  L: filter#47, R: (11), Type: #12,
#175    DelayNode  L: #181, R: (2),
#176    ArrayIndexNode  L: filter#47, R: (13), Type: #12,
#177    CommaNode  L: #182, R: #183,
#178    DefnNode  L: #184, R: #185,
#179    ArrayIndexNode  L: filter#47, R: (10), Type: #12,
#180    DelayNode  L: #186, R: (2),
#181    ArrayIndexNode  L: filter#47, R: (12), Type: #12,
#182    CommaNode  L: #187, R: #188,
#183    DefnNode  L: #189, R: #190,
```

```
#184    ArrayIndexNode  L: filter#47, R: (9), Type: #12,
#185    DelayNode  L: #191, R: (2),
#186    ArrayIndexNode  L: filter#47, R: (11), Type: #12,
#187    CommaNode  L: #192, R: #193,
#188    DefnNode  L: #194, R: #195,
#189    ArrayIndexNode  L: filter#47, R: (8), Type: #12,
#190    DelayNode  L: #196, R: (2),
#191    ArrayIndexNode  L: filter#47, R: (10), Type: #12,
#192    CommaNode  L: #197, R: #198,
#193    DefnNode  L: #199, R: #200,
#194    ArrayIndexNode  L: filter#47, R: (7), Type: #12,
#195    DelayNode  L: #201, R: (2),
#196    ArrayIndexNode  L: filter#47, R: (9), Type: #12,
#197    CommaNode  L: #202, R: #203,
#198    DefnNode  L: #204, R: #205,
#199    ArrayIndexNode  L: filter#47, R: (6), Type: #12,
#200    DelayNode  L: #206, R: (2),
#201    ArrayIndexNode  L: filter#47, R: (8), Type: #12,
#202    CommaNode  L: #207, R: #208,
#203    DefnNode  L: #209, R: #210,
#204    ArrayIndexNode  L: filter#47, R: (5), Type: #12,
#205    DelayNode  L: #211, R: (2),
#206    ArrayIndexNode  L: filter#47, R: (7), Type: #12,
#207    CommaNode  L: #212, R: #213,
#208    DefnNode  L: #214, R: #215,
#209    ArrayIndexNode  L: filter#47, R: (4), Type: #12,
#210    DelayNode  L: #216, R: (2),
#211    ArrayIndexNode  L: filter#47, R: (6), Type: #12,
#212    CommaNode  L: #217, R: #218,
#213    DefnNode  L: #219, R: #220,
#214    ArrayIndexNode  L: filter#47, R: (3), Type: #12,
#215    DelayNode  L: #221, R: (2),
#216    ArrayIndexNode  L: filter#47, R: (5), Type: #12,
#217    CommaNode  L: #222, R: #223,
#218    DefnNode  L: #224, R: #225,
#219    ArrayIndexNode  L: filter#47, R: (2), Type: #12,
#220    DelayNode  L: #226, R: (2),
#221    ArrayIndexNode  L: filter#47, R: (4), Type: #12,
#222    CommaNode  L: #227, R: #228,
#223    DefnNode  L: #229, R: #230,
#224    ArrayIndexNode  L: filter#47, R: (1), Type: #12,
#225    DelayNode  L: #231, R: (2),
#226    ArrayIndexNode  L: filter#47, R: (3), Type: #12,
#227    DefnNode  L: my_coefs#48, R: #232,
#228    DefnNode  L: #233, R: in#234,
#229    ArrayIndexNode  L: filter#47, R: (0), Type: #12,
#230    DelayNode  L: #235, R: (2),
#231    ArrayIndexNode  L: filter#47, R: (2), Type: #12,
#232    AggregateNode  L: #236, R: -*-,
#233    ArrayIndexNode  L: filter#47, R: (15), Type: #12,
#234    in#234 Type: #12, ValueClass: 4,
#235    ArrayIndexNode  L: filter#47, R: (1), Type: #12,
#236    CommaNode  L: (5), R: #237,
#237    CommaNode  L: (7), R: #238,
#238    CommaNode  L: (8), R: #239,
#239    CommaNode  L: (9), R: #240,
```

```
#240    CommaNode  L: (12), R: #241,
#241    CommaNode  L: (16), R: #242,
#242    CommaNode  L: (27), R: #243,
#243    CommaNode  L: (81), R: #244,
#244    CommaNode  L: #245, R: #246,
#245    NegateNode  L: (81), R: -*-, Type: #247,
#246    CommaNode  L: #248, R: #249,
#247    NumTypeNode  L: -*-, R: -*-,
#248    NegateNode  L: (27), R: -*-, Type: #247,
#249    CommaNode  L: #250, R: #251,
#250    NegateNode  L: (16), R: -*-, Type: #247,
#251    CommaNode  L: #252, R: #253,
#252    NegateNode  L: (12), R: -*-, Type: #247,
#253    CommaNode  L: #254, R: #255,
#254    NegateNode  L: (9), R: -*-, Type: #247,
#255    CommaNode  L: #256, R: #257,
#256    NegateNode  L: (8), R: -*-, Type: #247,
#257    CommaNode  L: #258, R: #259,
#258    NegateNode  L: (7), R: -*-, Type: #247,
#259    CommaNode  L: #260, R: -*-,
#260    NegateNode  L: (5), R: -*-, Type: #247,
```

Appendix B: Designfile Program Example

```
Script started on Mon Nov 25 09:49:32 1985
tuborg 1>> cat fir.df
 .global
    begin
    in<8>;
    out<8>;
    tap<8>;
    end

 .io <8>
    begin
    in : signal_in;
    out : signal_out;
    end

 .processor : one<8>
    begin
    .local
      begin
      dly[32], result;
      end

    .constant
      begin
        weight[31] = 5,0,7,0,8,0,9,0,12,0,16,0,27,0,81,0,
                -81,0,-27,0,-16,0,-12,0,-9,0,-8,0,-7,0,-5;
      end

 .main_pr  /* initialization, in- and  output handling */
    begin
    r(result), mbus = in, le;
    w(dly[31]), mbus = mor, out = mbus, acc := 0;
    w(result), le;
    end

 .sub_pr <31>  /* implements one tap of the fir */
    begin
    rx(weight);
    rx(dly[1]), mbus = mor, tap := mbus;
    r(result), sor:=mor;
                          sor:=sor>1,acc:=mor + coef.-sor,coef = tap;
                          sor:=sor>1,acc:=acc+coef.sor;
                          sor:=sor>1,acc:=acc+coef.sor;
                          sor:=sor>1,acc:=acc+coef.sor;
                          sor:=sor>1,acc:=acc+coef.sor;
    rx(dly[1]),           sor:=sor>1,acc:=acc+coef.sor;
```

```
    mbus = mor, le,      sor:=sor>1,acc:=acc+coef.sor;
    wx(dly),             acc:=acc+coef.sor;
    w(result), le;
    end
 end
tuborg 2>> cat infile
127
0
0
0
0
0
0
0
0
0
0
0
0
0
0
tuborg 3>> ~/Demonnew/Demon fir.df
        The U.C. Berkeley LAGER Layout Generation System
        Program : Demon - Version 2.3

    input readin and check ...
    read_in finished ...
    control and i/o structure set up ...
    connect data files ...

    Current Data File Selection :
    -----------------------------
      in : infile
      out : outfile

    Any changes wanted ? (y/n) [n] > n

    start emulation ...
Demon > run

    input data exhausted ...
    16 samples processed ...
    done ...
Demon > quit

    16 samples processed
    emulation halted !
    elapsed time :   0h : 0m : 14s
tuborg 4>> cat outfile
0
-10
0
-11
0
-11
```

```
0
-14
0
-15
0
-18
0
-30
0
-85
tuborg 5>> exit
tuborg 6>>
script done on Mon Nov 25 09:51:36 1985
```

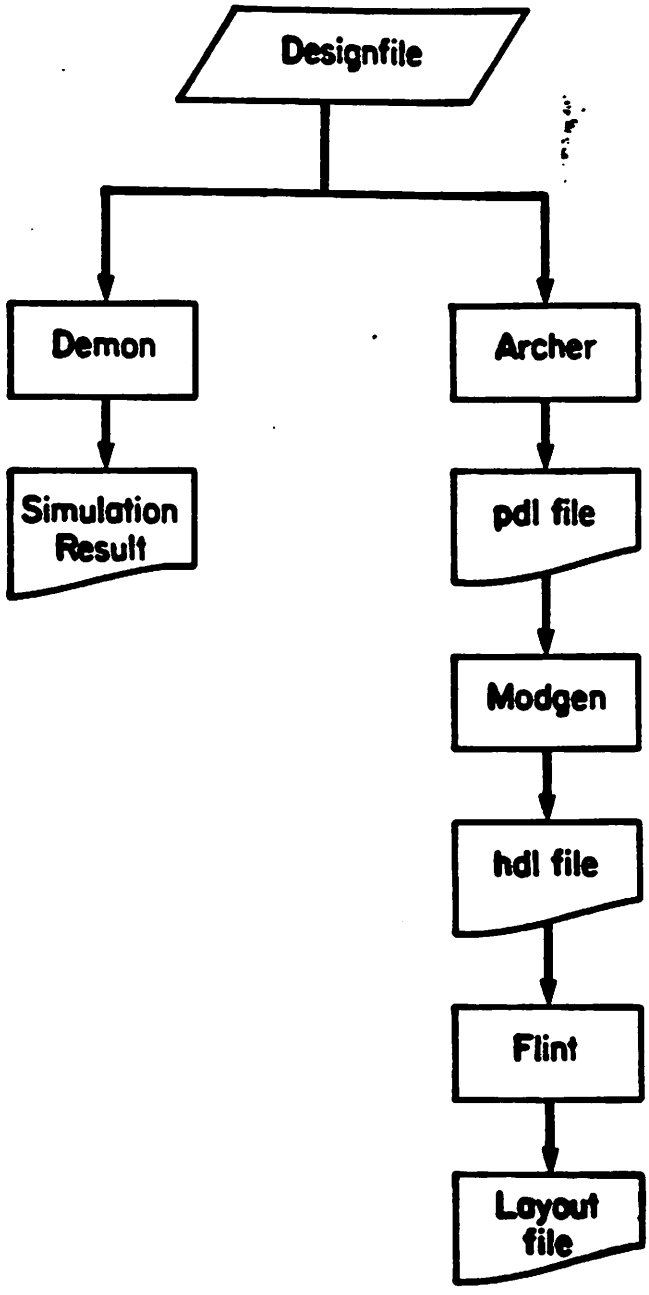

Figure 1. **Lager System Flow Diagram**

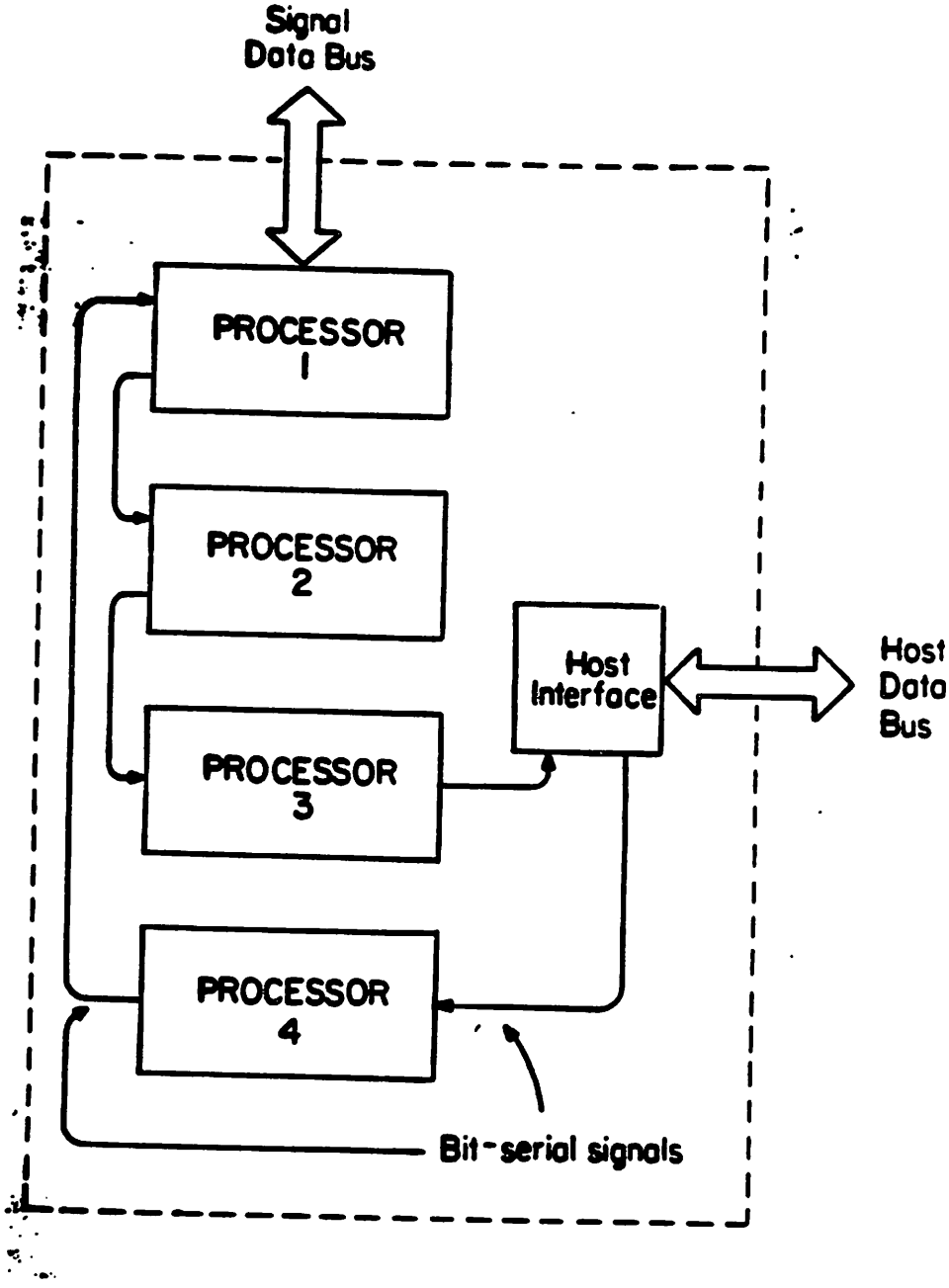

Figure 2. **Lager Multiprocessor Architecture**

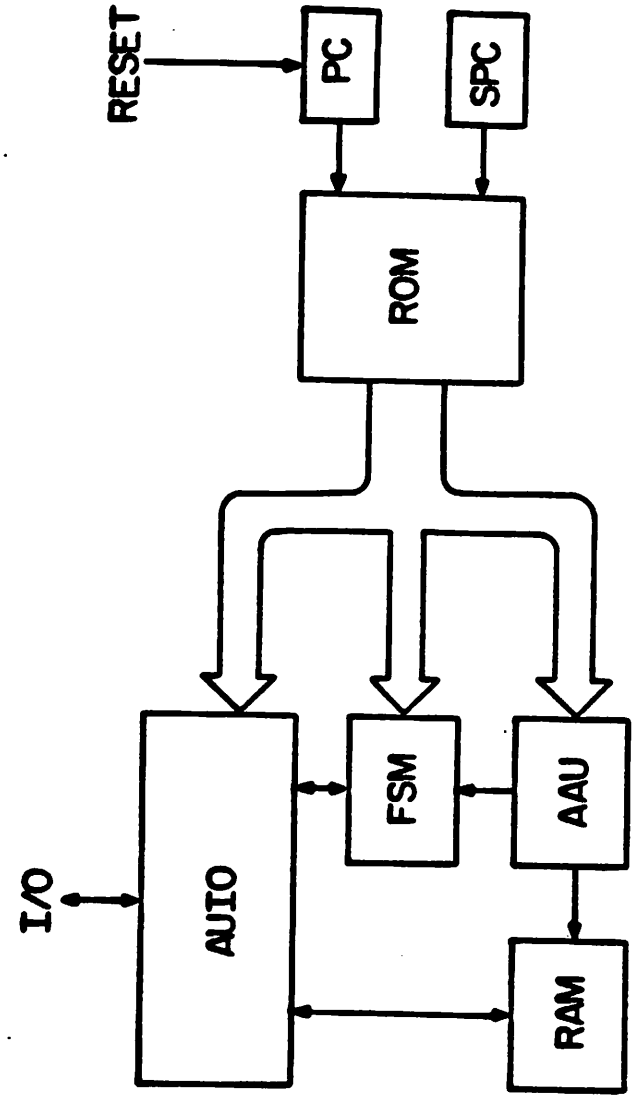

Figure 3. Lager Processor Architecture

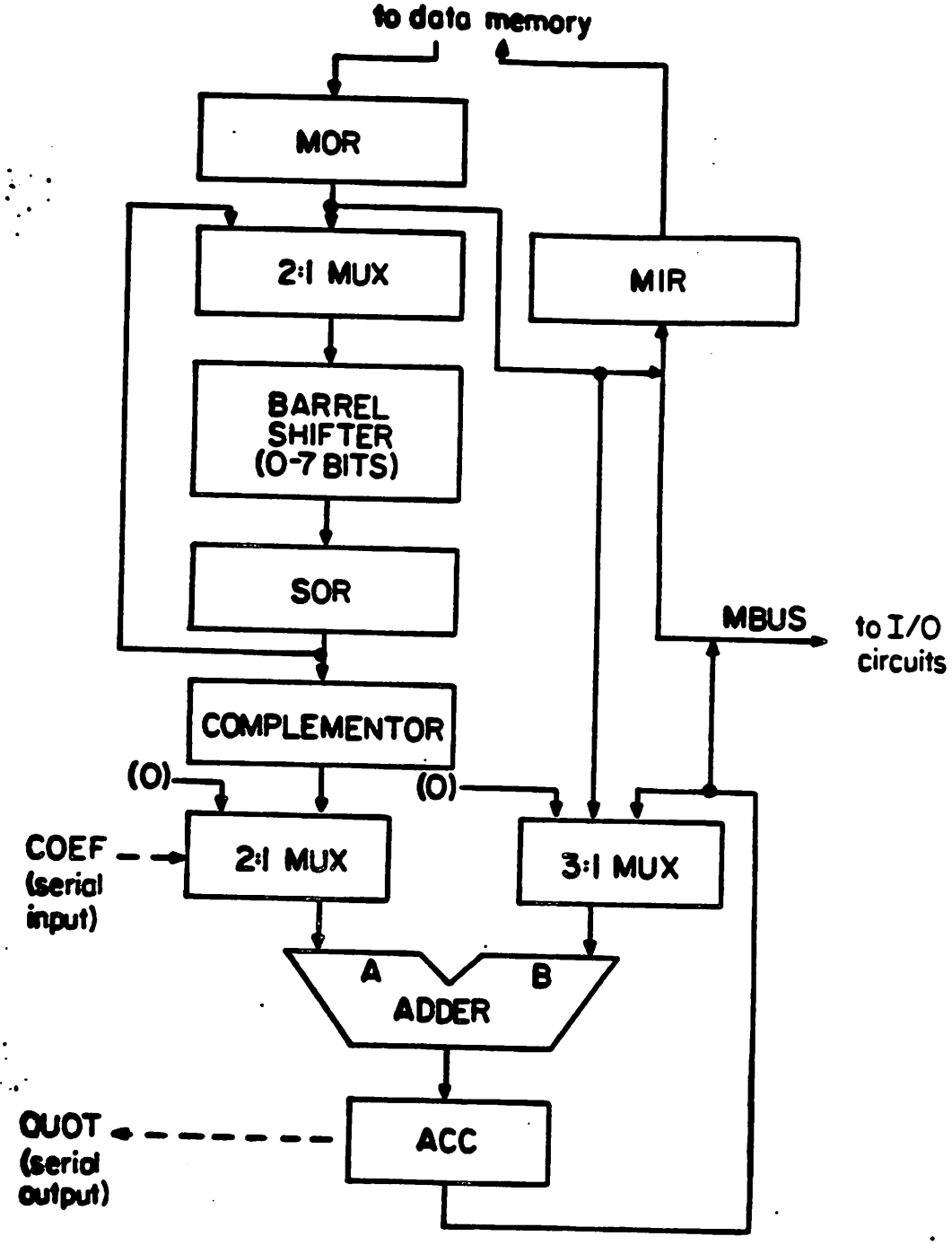

Figure 4. Lager DataPath

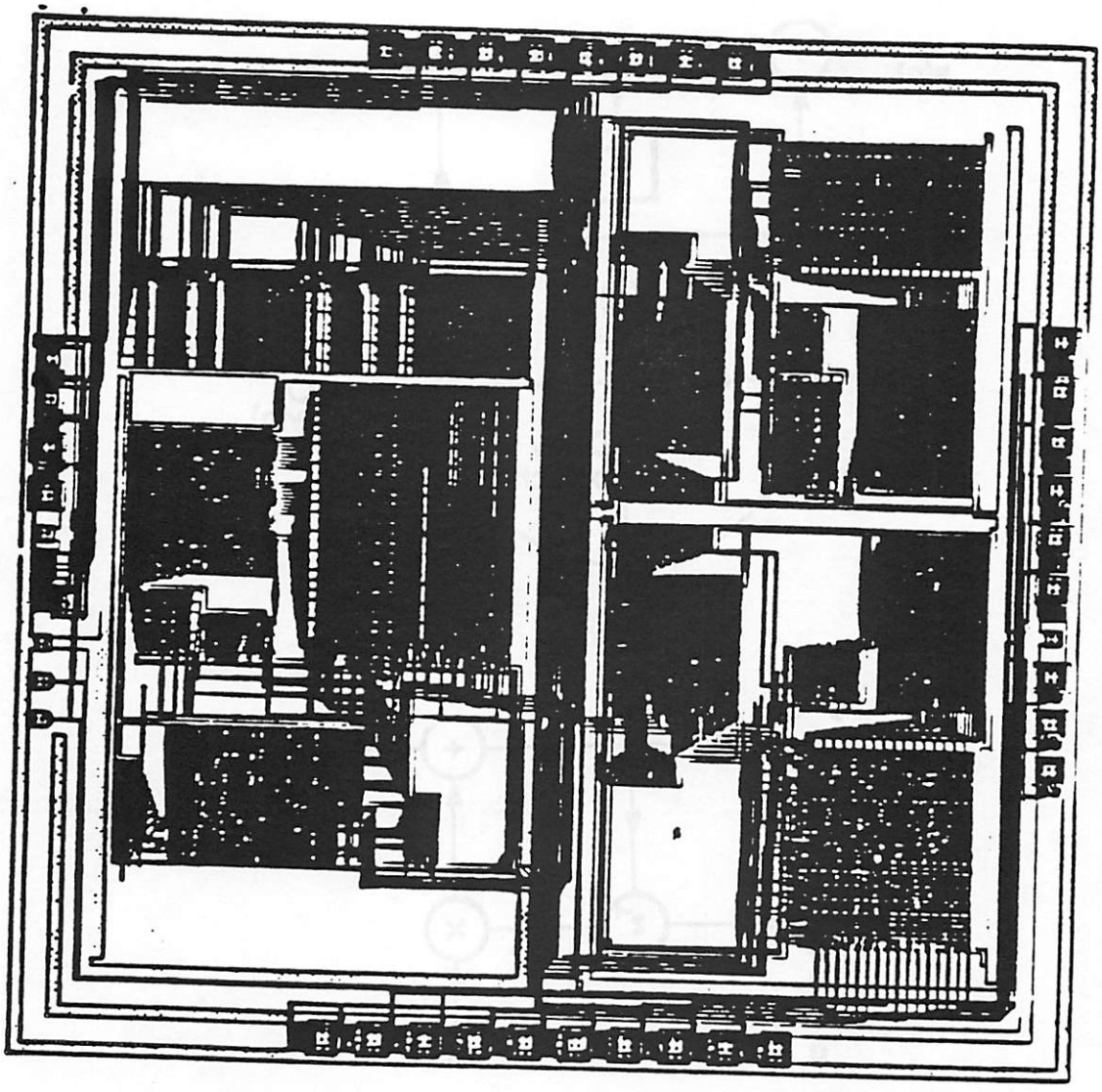

Figure 5. **Speech Vocoder[3] Layout**

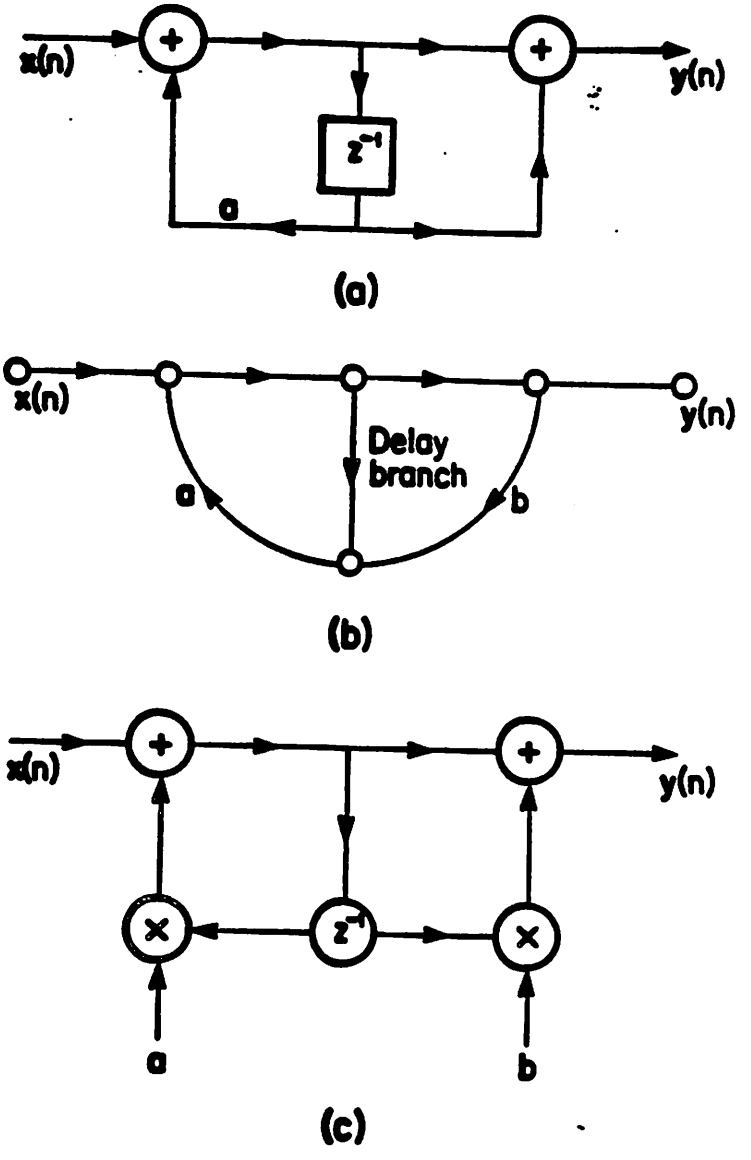

Figure.6 (a) Block Diagram of A First-order Digital Filter

(b) Signal-flow Diagram of (a)

(c) Data-flow Diagram of (a)

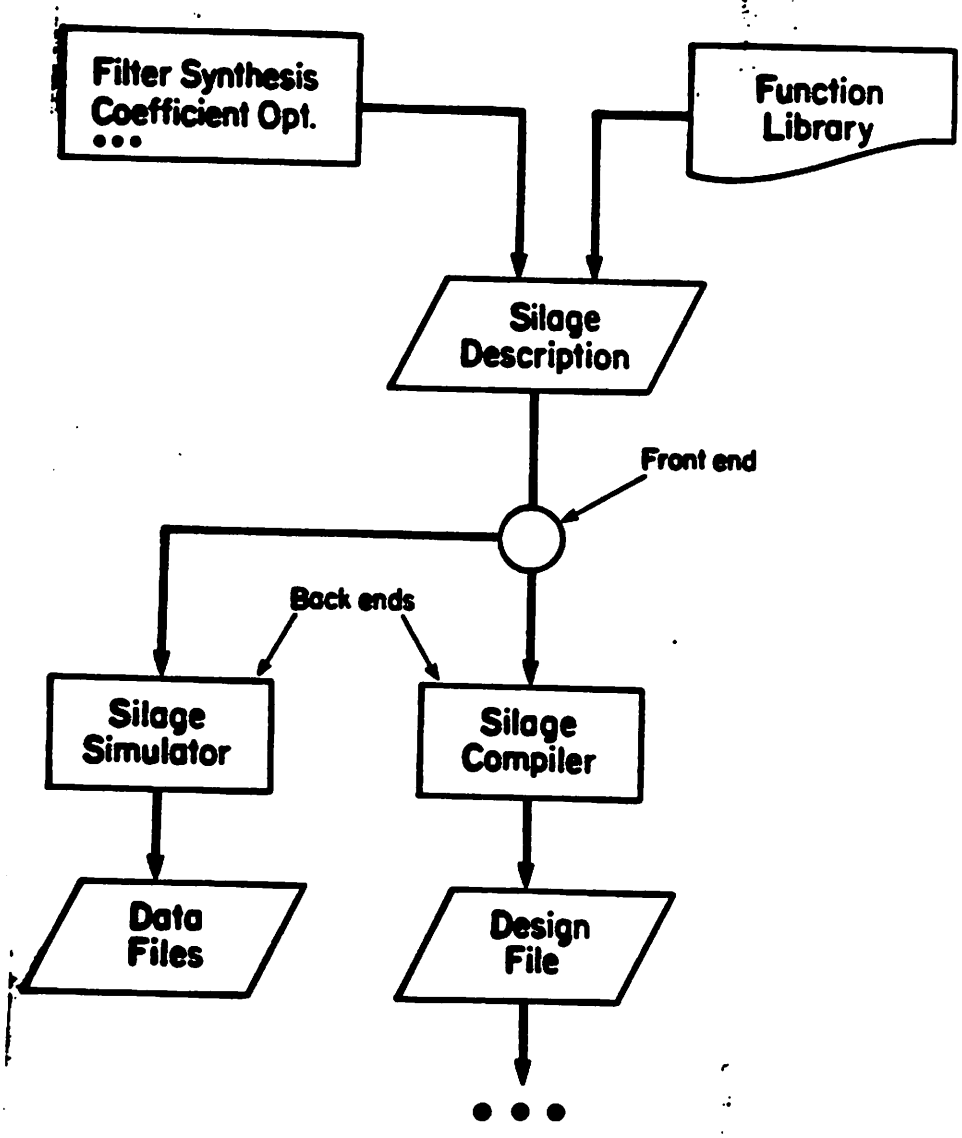

Figure 7. Silage Front-end and Back-end