The Design And Implementation Of A Systolic Array Silicon Compiler

Xiaoling Sun

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements for the Degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

November 1987

ISBN 0-315-41644-0

ABSTRACT

The Design And Implementation Of
A Systolic Array Silicon Compiler

Xiaoling Sun

The Systolic Array Silicon Compiler (SASC) accepts systolic designs described at algorithm level and is intended to generate behavioral simulation results for validation and VLSI layouts for fabrication. The design considerations and implementation issues of SASC are discussed. SASC uses bit-serial target architecture, but can accept systolic algorithms for other architectures. It performs two-level pipelined implementation to improve system throughput and keeps all timing synchronization user transparent.

A high level description language (Systolic Array Silicon Compiler Language (SASCL)) which reflects the properties of systolic architecture is designed. This data flow language provides a simple and hierarchical way of describing systolic designs. SASCL descriptions are first translated into an intermediate form (Systolic Array Silicon Compiler Intermediate Language (SASCIL)). Each SASCIL statement can then be easily translated into the corresponding hardware. Both SASCL and SASCIL are technology independent. The generation of layout from SASCIL is technology dependent and currently can be done for the Northern TeleCom CMOS1B (5 micron) technology.

SASC uses a number of (CMOS1B) leaf cells which can be used to compose other function blocks. These leaf cells and function cells can be automatically placed and routed. SASC can easily be adapted to other technologies by redesigning the leaf cells and appropriately modifying the technology file.

Currently, SASC can generate layouts from SASCIL descriptions. The philosophies and translation rules for SASCL to SASCIL are proposed. The simulation part of SASC as well as the SASCL to SASCIL translation is left for future implementation.

ACKNOWLEDGEMENTS

I would like to express my greatest gratitude to my thesis supervisor, Dr. H. F. Li for his guidance and encouragement. He outlined the thesis project and made great contributions in its development.

I would also like to thank Dr. R. Jayakumar for his criticism and help on final completion of this thesis, and Dr. C. Lam for his advice, support and proofreading.

I am also grateful to Mr. Luc Morin for his helpful discussions, Mr. Derek Pao for his work on the general router and Mrs. Lily Lam and Mr. Paul Gill, Mrs. Pauline Dubois, Mr. Dave Hargreaves, Mr. Girish Patel and Mr. Marco Zelada for their technical expertise, cooperation and help.

Finally, I would like to extend my thanks to the Shenyang Institute of Computing Technology, Chinese Academy of Science for providing me with a study-abroad scholarship.

Table of Contents

Abstract iii

Acknowledgements v

Table Of Contents vi

List Of Figures ix

List Of Appendices xi

1 INTRODUCTION 1

1.1 Systolic Array 2

1.2 Silicon Compilation 4

1.3 About The Thesis 5

2 THE SYSTOLIC ARRAY SILICON COMPILER 9

2.1 The Design Objectives 10

2.2 The Design Considerations 10

2.3 The SASC Environment: An Overview 15

3 THE SYSTOLIC ARRAY SILICON COMPILER LANGUAGE 18

3.1 Systolic Architecture Vs Data Flow of Computation 18

3.2 Data Flow Languages and Description Languages 19

3.3 The Design Objectives And Approaches 21

3.4 The Language Features 23

3.5 The Language Description 28

3.6 The Semantic Considerations 36

3.7 Specifying An Algorithm To The Compiler 39

4 DERIVING SYSTOLIC ARRAY FROM ALGORITHM DESCRIPTION 44

4.1 The Design Objectives And Considerations 44

4.2 The Language Descriptions And Semantic Considerations 47

4.3 Deriving The Systolic Algorithm From The Algorithm Description 49

5 THE COMPONENTS OF SASC 59

5.1 The Layout Generation Hierarchy 59

5.2 The Leaf Cells 63

5.2.1 Selection of Leaf Cells 63

5.2.2 Leaf Cell Conventions 63

5.2.3 The Library Description Files 66

5.2.4 Technology Dependency 69

5.3 The Primitive Operators 69

5.3.1 Selection of Primitive Operators 69

5.3.2 Primitive Composition 71

5.3.3 Primitive Description File 76

5.3.4 Expandability Of The Libraries 77

5.4 The Logic Compiler 78

5.4.1 Assumptions 78

5.4.2 Domino Logic Structure 78

5.4.3 Logic Compilation 80

5.5 Placement, Routing and CIF File Generation 82

5.5.1 Placement and Routing 82

5.5.2 The Routing Environment 86

5.5.3 Generating Layout Geometry 87

5.6 Generation Of The Layout From The Intermediate Code 91

5.6.1 The Interpreter 91

5.6.2 Generation Of Layout From The SASCIL 96

5.7 Generating Layout For The Example Algorithm 100

6 Conclusion 117

References 120

List of Figures

Fig. 1.1.1 General Structure of Systolic Array 3

Fig. 1.2.1 Silicon Compilation Process 6

Fig. 2.2.1 Two Phase Clock Convention 12

Fig. 2.3.1 Overall Structure Of The SASC 17

Fig. 3.5.1 An Example Of Two Types Of PE's 31

Fig. 3.5.2 The Higher Level Design Hierarchy 31

Fig. 3.5.3 The Main Logic Transmission Directions 33

Fig. 3.7.1 The Systolic Array For The FIR Algorithm 41

Fig. 4.3.1 The Communication Strategies Of Target Architecture 51

Fig. 4.3.2 The Target Architecture Of The FIR Algorithm 58

Fig. 5.1.1 VLSI Design Hierarchy 60

Fig. 5.1.2 Layout Generation Hierarchy 61

Fig. 5.2.1 List of Leaf Cells 64

Fig. 5.2.2 Cell Bounding and Supply/Clock wires 67

Fig. 5.2.3 An Example Leaf Cell 68

Fig. 5.3.1 List of Primitives 72

Fig. 5.3.2 Block Diagram Of Composition Routine 74

Fig. 5.4.1 CMOS Domino Logic Basic Gate 79

Fig. 5.4.2 An Example of Variable/Product Table 81

Fig. 5.4.3 An Example of N-logic Block Layout 83

Fig. 5.4.4 Block Diagram Of Logic Compiler 84

Fig. 5.5.1 The Router Environment 88

Fig. 5.5.2 Block Diagram Of CIF-generator 90

Fig. 5.6.1 List Of Procedures For Assignment Statements 92

Fig. 5.6.2 Levels Of Data Representation 97

Fig. 5.6.3 Block Diagram Of The Interpreter 99

Fig. 5.7.1 Files p1.klc and p1.deb 108

Fig. 5.7.2 Layout Of Primitive Operator "p1" 113

Fig. 5.7.3 Layout Of Function "pe" 115

Fig. 5.7.4 Layout Of The FIR Systolic Array. 116

List of Appendices

Appendix I: The BNF Notation 133

Appendix II: The Syntax Definition Of SASCL 134

Appendix III: The System Functions Of SASCL 138

Appendix IV: The Syntax Definition Of SASCIL 143

Appendix V: The Primitive Library 147

Appendix VI: The Leaf Cell Library 171

CHAPTER 1

INTRODUCTION

When Von Neumann and his colleagues designed the first computer forty years ago, they partitioned the design into two separate parts, namely, processor and memory. This is called the Von Neumann architecture. Even though basic research and improvements in fabrication technology have motivated the development of four generations of computers, still most of today's computers are built using the Von Neumann architecture.

In the last decade, improvements in hardware technology for conventional computer architecture seem to have met the upper bound of physical constraints. Example of these are speed of light, power-cooling ratios etc, which are achievable by a single processor. Meanwhile, advances in integrated circuit technology are rapidly increasing the density and complexity of circuits which is doubled every one or two years. Now, one micron technology is practical, allowing the integration of one million transistors on a single chip. So, more and more people begin to challenge the Von Neumann architecture and computation model. The classical computer architecture suffers from two main problems: (1) Since a processor is separated from its memory by communication paths, no mater how fast the processor is, the duration of the computation is determined by the time required to move data between processor and memory. (2) The single processor architecture only supports sequential data processing. The innovation of computer architecture includes two major aspects: the Von Neumann computation model vs Non-Von Neumann computation model and general-purpose computer architecture vs special-purpose computer architecture. The development of VLSI technology opened the door for the design of innovative computer architectures and algorithms. The potential power of VLSI is the large amount of concurrency that it may support. Many new approaches to computer architecture have been investigated, designed and built [1]-[41]. Systolic architecture is one of them.

1.1 The Systolic Array

The systolic array concept was proposed by H. T. Kung at CMU in 1979 [26] as a realistic alternative to conventional Von Neumann architecture. Systolic computer is a special purpose massively parallel processor for solving compute-bound problems, for example, signal processing, matrix multiplication, FFT, etc. Conceptually, systolic architecture is a general methodology for mapping high level computations into hardware structures [35]. In a systolic system, data flow from computer memory in a rhythmic fashion, passing through many processing elements before returning to the memory. Fig. 1.1.1 shows the general structure of a systolic array. The main properties of systolic architecture are:

(1) Make multiple use of each input data flow.

Systolic systems can achieve very high throughput by having each input travel through an array of processing elements.

(2) Modular expansibility.

In a pure systolic system, only local communications and controls are involved. Global communications are restricted to system clocks, power and ground lines. So, it is theoretically infinitely expandable, though a large clock skew will slow down the global clock.

(3) Extensive concurrency.

Concurrency in a systolic system is obtained by :

(a) Pipelining the stages involved in the computation of each single result.

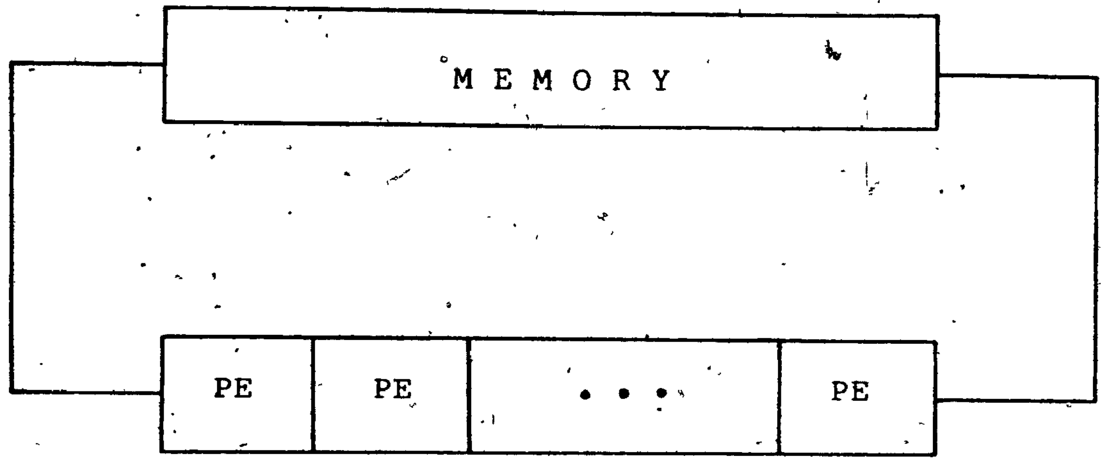


Fig.1.1.1 General Structure of Systolic Array.

(b) Multiprocessing many results in parallel.

(c) It is sometimes possible to introduce two-level of pipelining by allowing the operations inside the processing elements themselves to be pipelined.

(4) Use only a few types of simple cells.

(5) Use simple and regular data and control flow.

Many systolic computers and algorithms have been designed, built and tested [26]-[41]. Systolic algorithms use parallel pipelined processing with localized control and signal flows and exhibit extensive parallelism with minimum communication. They form a new promoting area of computer innovation.

1.2 Silicon Compilation

A common model of VLSI design partitions the design process into functional, logic block, circuit, layout and mask level. Design of VLSI systems is complicated by additional refinements needed to layout the geometry masks. Although manual design of chip masks could usually achieve high packing density, the complexity involved makes the design of custom VLSI circuits time consuming, tedious, inflexible, error prone, and very expensive. Research in the area of silicon compilation started in late 1970's [53]. It is an automatic synthesis process which takes care of design translation and verification. The correctness of a design is ensured. A silicon compiler [51]-[80] is a software system which is capable of accepting language or symbolic descriptions and producing a chip mask geometry automatically. It could also simulate a system design at the level of description and allow a rapid validation and implementation of VLSI systems. The language or symbolic description could be at different levels of design hierarchy decided by tradeoffs between application and system design requirements. Generally speaking, a silicon compiler is more concerned with functional implementation, design cost and correctness rather than optimal packing density. Fig. 1.2.1 shows a silicon compilation process.

Silicon compilers [42]-[67], [82]-[87] have many common features with software compilers.. They both take as input, a program written in a programming language and produce as output a program in another language (the code or target language). The "compiled code" is only good for a particular machine (in the case of software compilation) or a particular architecture (in the case of silicon compilation). Talking ~~about~~ language features, for example, computations could be abstracted at different levels; information hidden is one of the main decision to be made, etc. The main difference between the two compilers is that the output of a silicon compiler is not directly executable by a computer. A language, called CIF (CalTech Intermediate Form) which abstracts fabrication process into a number of conceptual layout levels that represent the physical features one observes in the final silicon wafer, is used as a standard medium to interface with fabrication technology. A silicon compiler should provide all of the control between different sections of a program since there is no operating system in the environment to handle controls as in a software compiler. There is also a need for a simulator to validate the system design.

1.3 About The Thesis

Although silicon compilation is a new area of exploitation, it grows very rapidly. A number of research interests, for example, partitioning, placement, routing, testing, design verification and simulation etc., are under this broad topic. It seems impossible to take care of all these aspects in this thesis. What we intend to do is to design a silicon compiler for a particular application area, namely systolic array, with some novelties. The thesis analyzes systolic architecture properties and exploits the natural similarity between systolic array

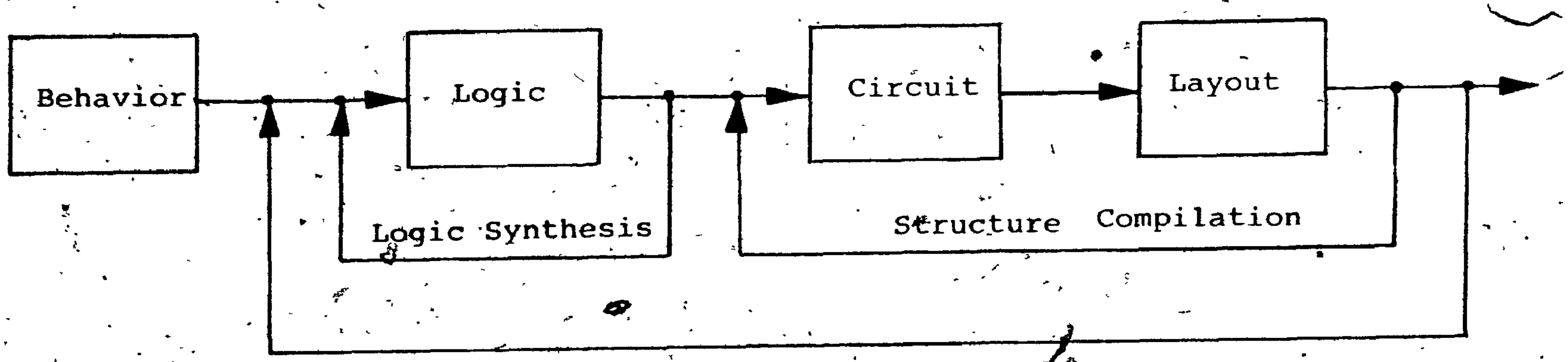


Fig.1.2.1 Silicon Compilation Process.

and data flow model of computation and sets up reasonable design objectives for both the silicon compiler and its description language. In the design process, efforts are also made on raising the level of abstraction at which a designer may think, reducing the amount of details that a designer must indulge in and adding intelligence and flexibility to the system by careful design of a high level description language as well as the compiler itself.

The thesis is organized as follows: We present in chapter 2 a discussion of previous work on silicon compilation and design the compiler of our own. In chapter 3, we pay attention to the high level description language design which is one of the main achievement of the compiler. Then, the design of the intermediate language is given in chapter 4. We discuss the layout generation process in chapter 5 and give the conclusion in chapter 6.

For this thesis work, a high level data flow language, called Systolic Array Silicon Compiler Language, and an intermediate form of data description, named Systolic Array Silicon Compiler Intermediate Language, are proposed. We present a set of philosophies and translation rules for the translating the high level description into the intermediate code. The automation of this part is beyond this thesis. We perform hand-compilation for the high level translation process in an example given.

For the translation from the intermediate code into mask geometry, we design and implement both leaf cell and primitive libraries. Leaf cell library is a set of KIC files which are symbolic descriptions of minimum logic elements supporting the target architecture. Primitive library is implemented as a set of composition routines which compose a group of logically related leaf cells together and give the initial placement of calling cells. A general router used at all levels of composition, a CIF generator adapting the intermediate data files into the standard CIF code, a logic compiler being capable of converting boolean type of data into a random logic structure implementation, and a interpreter conducting the syntax analysis of the intermediate code are designed and implemented. The incomplete part in the low level translation is the implementation of a placer as well as final chip floor plan génération. The behavioral simulator is also left for future work.

CHAPTER 2

THE SYSTOLIC ARRAY SILICON COMPILER

Silicon compilation has been a very interesting topic in computer science and technology. It covers the studies on design description, translation, simulation and verification as well as placement, routing, testing, etc. There are many existing silicon compilers. They were developed for a wide range of application areas, such as signal processing [47], data flow computers [61], finite-state machines [43] or PLA [46], etc, and support many fabrication technologies. The levels of data description cover topology, device, block, architecture and behavior. The target realizations used include standard cells [47], [57], [86],[87], gate arrays [52], [84], [85], random logic and unconstrained custom integrated circuits [58]. More recently, some silicon compilers [61] incorporate behavioral simulation, design verification and other advanced features. Although some of the existing silicon compilers can be used to design chips for systolic arrays, there are a number of problems which are crucial. The MacPitts [50], [60], [61], for instance, was designed for general signal processing applications. Therefore, systolic arrays are only a subset of the architecture supported. It is general enough, but may not be efficient enough to reflect the properties of systolic computation. For example, systolic arrays include a small set of network topologies, and the logic synchronization mechanism is simpler. On the other hand, the existing compilers may not support the fabrication technology available to us. The only way to overcome these difficulties is to design our own silicon compiler -- the Systolic Array Silicon Compiler (SASC).

2.1 The Design Objectives

The main design objectives of the SASC are as follows:

(1) The SASC is aimed at developing a powerful tool which accepts a high level language description and produces chip mask geometry automatically in order to achieve a rapid implementation of systolic arrays. System and functional level simulations will be carried out if they are required.

(2) The SASC should ensure logical correctness and design rule satisfaction of a design.

(3) It should be able to generate designs of complexity compatible with those realized by a human layout designer using manual methods.

(4) It should ensure design integrity in existing design database environments.

(5) The high level language should help users to produce correct design and be user friendly.

(6) The SASC should provide a quick, simple means to incorporate design rule changes.

2.2 The Design Considerations

Primarily, the SASC is based on the structural design methodology. In this methodology, a digital system could be regarded as a combination of register-to-register data transfer paths and finite-state machines. The SASC can be partitioned into two subsystems: data path compiler (for data path generation) and logic compiler (for FSM generation). Each of them runs from high level specification to low level mask geometry. Different methodologies are chosen to solve the problems involved in each of them. We use standard cell methodology for data path generation and give a special treatment to FSM which is to be discussed later.

The main system assumptions we make are based on the properties of systolic array, simplicity, regularity and global system clock driven as well as some implementation constraints:

(1) Algorithms are synchronous. If we go down to the structure level of a design, all logic elements are under the control of system clocks.

(2) The two-phase non-overlapping clocking scheme is used and shown in Fig. 2.2.1. Data for each function block is received at clock phase one and evaluations are to be done at clock phase two.

(3) Only bit-serial system structure is supported.

(4) The simplicity of processing element is defined as follows:

(a) A systolic system module is considered to be implemented on a single chip (this restriction will be removed in the second stage of the project).

(b) A system may consist of many processing elements and each of them should contain simple logic circuit: A boolean expression may not have more than either 9 variables with any number of products or 9 products with any number of variables (this limitation is due to the logic compiler implemented).

(5) The technology supported is the CMOS1B of Northern TeleCom. The SASC strongly supports hierarchical way of structuring a design. The system structure of the SASC is a result of overall considerations to achieve the main design objectives, as discussed below.

(1) *High level language --> intermediate code --> object code translation:* We first decided to keep the design description to the compiler at the algorithm

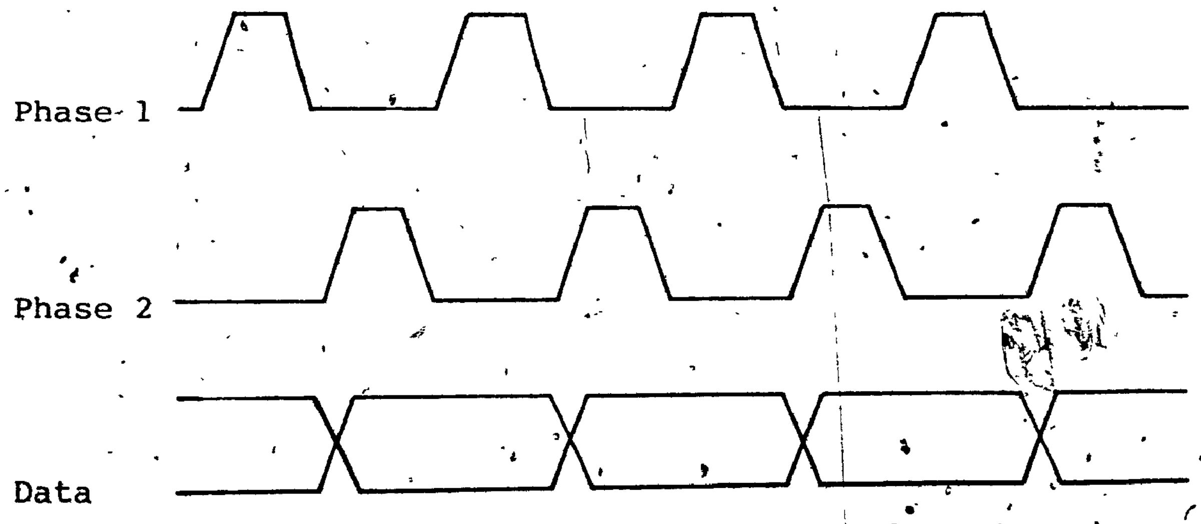


Fig.2.2.1 Two Phase Non-overlapping Clock Convention.

level. Later we discovered that it is too complicated to go through a translation directly from behavior to mask layout in single step. An intermediate language called Systolic Array Silicon Compiler Intermediate Language (SASCIL) is designed to describe the system at a suitable level lying somewhere between the above two levels. The intermediate code separates the detail of low level issues from users and make the high level language completely technology independent.

(2) *Hierarchical structure of the database system:* The center of the SASC is a database – a set of CIF codes which represents certain hardware logic elements. Primitive Cell Library implements a set of predefined primitive operators that can be parameterized. Leaf Cell Library consists of a set of minimum function elements which can be used to compose different primitives according to the user's specification. The above hierarchical structure forms lower level of system support. The higher level design description is formed by nesting the SASCL primitive functions. Appendices V and VI provide detailed information about these libraries.

(3) *Domino logic structure in implementing finite-state machines:* One of the main system assumptions we make in the compiler design is the simplicity of processing elements. CMOS domino logic [81] is a kind of random logic structure which is one alternative to CMOS complementary logic implementation. Even though speed achieved by complementary logic is usually better than that of others [43], its silicon area cost is higher if the standard library methodology is used. We use domino logic implementation since speed is not crucial for a FSM with less than ten products or variables and a good packing density could be obtained. For example, for a boolean equation with m products and n variables, at most $m * n + 4$ transistors are needed in domino logic implementation and $m * n * 2$ for complementary logic ones.

(4) *The hiding of libraries:* The SASC intends to free users from dealing with

specific instantiations of low level cells. Both primitive and leaf cell details are completely transparent to users.

(5) *Bit-serial implementation:* The advantages of bit-serial implementation over parallel ones are the simplicity, the flexibility and less human efforts needed for layout, circuit, logic and function designs of primitive operators. Automatic transformation from a non-bit-serial algorithm to the bit-serial target architecture represents an important aspect of intelligence and flexibility of the SASC. The correctness of the transformation should be ensured by the compiler, without user knowledge and interaction.

(6) *Two-level pipelined architecture:* Kung, Ruane and Yen have showed how two-level pipelined arithmetic units could be used to form high-performance systolic arrays [27]. The additional level of pipelining can greatly enhance the system throughput with only a small increase in hardware cost. This scheme is especially suitable for VLSI implementations.

The SASC user may have no idea about two-level pipelining at all and the compiler could generate the proper system structure, invoke the corresponding primitive operators, set up the necessary wire connections, and synchronize the logical events that should take place bit-serially.

(7) *Automatic floorplan generation:* The SASC is responsible for the placement and routing of the primitive/logic modules without human guidance. Special signals, e.g. power, ground and system clocks, and special components, e.g. I/O pads, buffers and frames, should not be specified by users. The SASC will generate a complete floor plan for the components and wire connections.

(8) *To accommodate design rule change:* The SASC addresses the following issues in order to accommodate design rule changes:

(a) By using intermediate language as a medium to separate the SASCL from lower level "code generation" details, the SASCL is made technology independent;

(b) By using a technology file which drives system modules, such as the router and logic compiler, technology changes can be tolerated by these utilities.

(c) By using a hierarchical structure of the databases and potential bit-serial realization, library redesign when technology changes is reduced to a minimum.

(9) *Behavioral simulations:* A simulator translates the SASCL program into behavioral description form and produces a textual file of the corresponding results.

2.3 The SASC Environment: An Overview

The highest level of the system is a high level data flow language which provides a simple and hierarchical manner of structuring the VLSI design of a systolic array.

The language compiler translates the high level description into intermediate form which deals with specific instantiations of low-level cells and has all connectivity defined explicitly.

The interpreter translates the intermediate code of a design into invoked primitives and all logic connections are transformed into physical ones. It also acts as a low level manager which calls the composition routines, logic compiler, placer and router, etc, and processes the intermediate data files and finally generates layout.

The composition routines call specific library operators and links them together according to predefined primitive definitions. A function block is a collection of logically related primitive cells, K-homogeneous (K<=5) function blocks form target system.

Logic compiler is an automated synthesis subsystem for finite-state machines. It accepts the intermediate code of boolean data type and generates the corresponding layout.

The router could be used at both the primitive and the higher functional block levels. It estimates channel width, then determines the exact coordinate positions and contact cuts for all cells, and finally connects I/O ports of various modules.

The compiler also translates a high level description into a behavioral description which is then used by the simulator.

The final floor plan of a chip could be generated automatically. Fig. 2.3.1 is the overall system structure of the SASC.

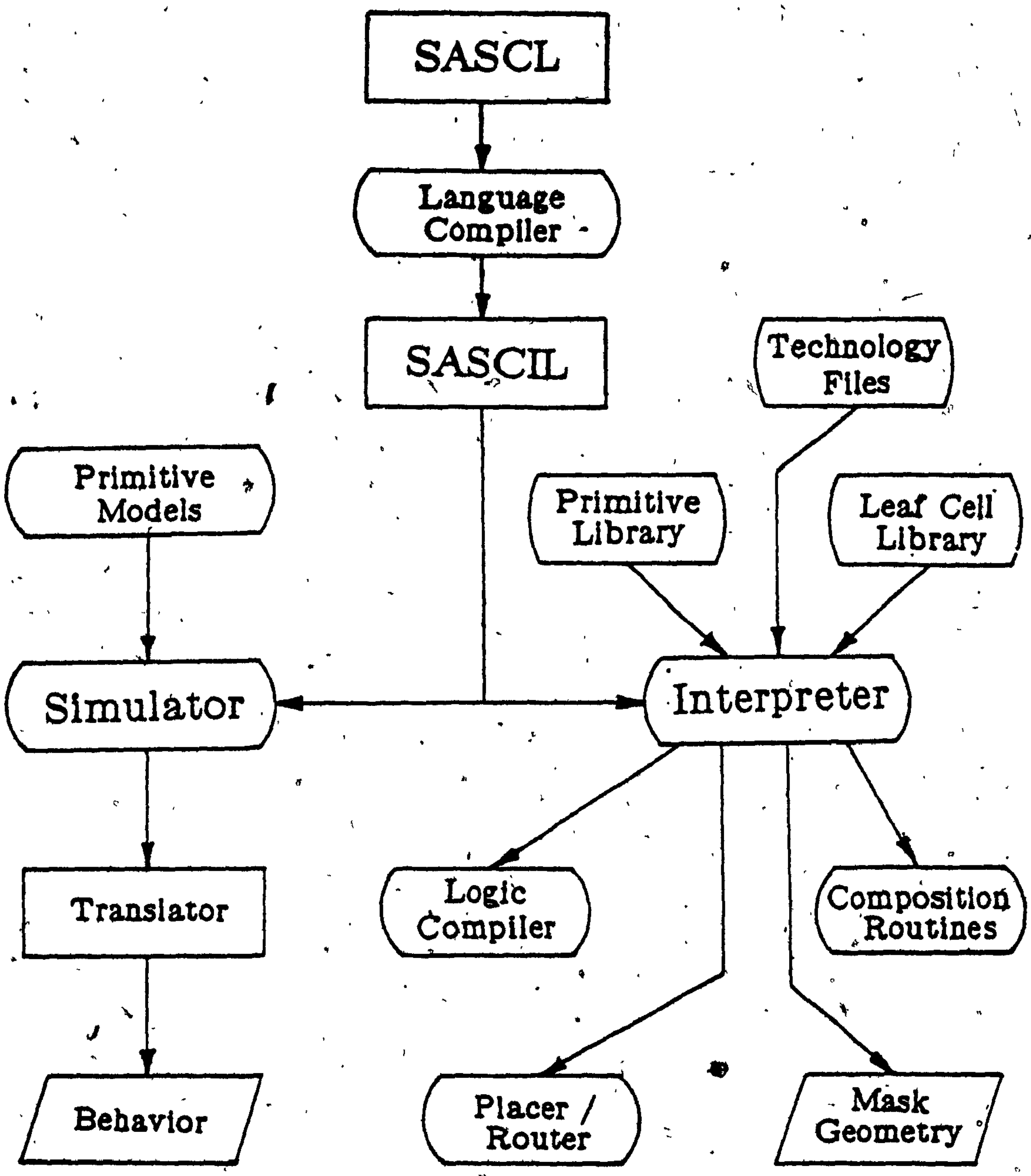


Fig. 2.3.1 The Overall Structure Of The SASC.

CHAPTER 3

THE SYSTOLIC ARRAY SILICON COMPILER LANGUAGE

In this chapter we discuss the high level description language called Systolic Array Silicon Compiler Language (SASCL). SASCL is a data flow language designed with the aim of having a high level of data abstraction in order to free the user from the description details as much as possible.

3.1 Systolic Architecture Vs Data Flow Of Computation

Data flow architectures [12], [71]-[73] are new approaches to computer architecture in pursuit of high performance computers. They are based on the data flow model [70] of computation which is fundamentally different from the sequential one-instruction-at-a-time model. It is based on two basic principles:

(1) Operations execute when the required operands are available, that is asynchronous execution.

(2) Operations have no side effects.

These principles say that, in other words, the only sequential constraints imposed by the data flow model are the data dependencies. It allows all the parallelism in a computation to be exploited. Secondly, since there is no need for a global updatable memory, no side effects are produced.

Recalling the discussions of chapter 1, it is easy to see that the computation performed by systolic arrays follow the above two principles. In particular, property (1) and (2) (see page 2) imply principle (1), and property (3) implys principle (2). The general structure of systolic array in Fig. 1.1.1 describes a typical ring type data flow architecture.

In summary we can conclude that systolic architecture is based on the data flow model of computation and forms a subclass of data flow architecture.

3.2 Data Flow Architecture and Description Languages

The data flow model is a fundamentally different approach in solving problems; it requires a new language to efficiently describe the data flow architectures. Data flow languages can be designed to reflect the model at different levels. It would seem that the natural way of expressing a directed data flow graph (DFG) would be via a graphical language [69]. High level data flow languages are desirable when applications are getting bigger and complicated and DFG's become unreadable and error-prone [68], [74], [76], [77].

Many general-purpose functional and data flow languages have been proposed for describing non-Von-Neumann computation. For example, the VAL (Value Algorithm Language) [88] is a well defined and tested high level data flow language developed at MIT. It embodies all the design principles and tries to achieve most of the design objectives of these data flow languages. It also takes a step further in supporting concurrency and machine independence. The weakness of VAL is the lack of general I/O facilities and recursion. The LAU [89] parallel system is a contribution to the development of the software concept of Single Assignment to parallel programming languages and architecture. It was designed as a tool for expressing parallel algorithms without first transforming them into equivalent sequential ones. More recently, because of the growing demand for best facilities in the investigation and implementation of novel digital system architectures and algorithms, more powerful languages, called description languages, have been developed for input description to an automatic synthesis program which can take care of design translation and verification at different design levels [56]-[63]. The FIRST [47], UNIT version-2 [57], MacPitts [61], ICEWATER [58] and XI [49] are the typical ones. Mapping a language to certain kinds of computer architecture is one of the common characteristics of these languages. Although some of these could be used for describing systolic processors, they are not very efficient and powerful enough in a silicon compilation environment which is more specialized.

The FIRST, MacPitts, and XI are designed for applications in digital signal processing systems. In particular, the FIRST is suitable for implementation of real-time signal transformation (bit-serial only) systems and the MacPitts particularly emphasizes on data path and FSM architecture. The circuit model supported by XI is synchronized, clocked elements and feadback paths associated with clocked elements. The UNIT version-2 can be used to describe an entire microcomputer system and supports both bit-serial and bit-parallel systems. The ICEWATER has the widest application area because of its symbolic description feature. The levels of data abstraction of these languages vary from algorithm level to symbolic ones. A hierarchical manner of describing a design was supported by all of them. Talking about the language features, XI is an extension of the algorithmic language C. It takes advantage of standard concepts of programming languages and environments in which they run that seem to be directly applicable to the circuit design process. It supports the full C programming language and encourages structured thinking and design. MacPitts is a good circuit description language. The data type of it, therefore, is less structured than that of XI. Both the FIRST and ICEWATER are simple, block structured languages. They deal with function block library operators and require explicit connectivity specification. Some of the language expressions used by these languages directly reflect the target architectures, for example, the extented data type of "register" in XI. The concepts of "bundle" introduced in XI and the "conceptual bus" in UNIT version-2 are good at representing a simple group of wire connections. The "COND" form in the MacPitts is convenient to express FSM.

3.3 The Design Objectives and Approaches

Any of the description languages discussed in previous section could be used to describe systolic designs. However, a common feature of all of them is their machine-dependency. MacPitts, for instance, was implemented in LISP; XI is an extension of C programming language and ICEWATER is based on an UNIX-like environment, etc.

Secondly, technology dependency of the low level compilation processes creats another problem. They all support NMOS fabrication process except the MacPitts which supports both MOS2S and NMOS technology. ICEWATER is the best one to face technology changes but pay a penalty of achieving the lowest level of design hierarchy. The technology supported by SASC at the present time is the Northern TeleCom CMOS1B, so that none of these compilers could be used under our circumstances.

Recall the design objectives of the SASC mentioned in the previous chapter. We aimed at having a high level of data abstraction in order to free users from the description details as much as possible. The levels described by FIRST, UNIT version-2 and ICEWATER are too low to match our goals. There is no doubt that MacPitts and XI are behavioral ones. One of the system assumptions made in these two languages is that users have the ability to express their circuit behaviors by an algorithm description, rather than one that describes the circuit structure. Although any control/data flow graph can be directly specified and high concurrency in data path operations are promoted by MacPitts, it is not powerful enough to reflect the characteristics of systolic computation. For example, the regularity of systolic arrays may allow some defaults (like clock, power and ground) to be used instead of specifying them explicitly; logic synchronization might be abstracted at a high level to help users avoid giving all timing details, etc. We also set up the novelties of our compiler as the automatic algorithm transformation into a bit-serial one and the two level pipelined implementation of computation without users' knowledge and interaction. In addition, behavioral simulation of the system is also incorporated in the compiler. Thus, the compiler requires a language with options for both layout generation and system simulation. It is obvious that none of the existing languages meets all our requirements. A new language is definitely required.

The Systolic Array Silicon Compiler Language (SASCL) intends to tailor the scopes of the existing description languages. It reflects the properties of systolic computation and concentrates on achieving the SASC design objectives. The design objectives of the SASCL are as follows:

(1) *To provide a simple and hierarchical manner of describing a systolic algorithm.*

SASCL meets this by using one of the most important features of functional languages: a higher order function may take another function as an argument. It makes the program remarkably short and succinct.

(2) *To provide implicit concurrency and synchronization.*

This is the most important characteristics of SASCL as a data flow language. SASCL meets this by allowing operations which could be executed independently to be defined without explicit language notations.

(3) *To allow implicit parameters, such as system clocks, to be default operands.*

System clocks, power and ground are the common default variables. Two phase system clocking scheme is used by default. SASCL hides all these and any retiming details (in case of 2-level pipelining or bit-serial conversion). A user working on the algorithm level should know how a data is "sampled" and processed, but he need not be bothered to define them.

(4) *To support Single Assignment Rule (SAR).*

A variable in a program may be assigned values by one statement only. Systolic architectures support SAR naturally due to their locality of references. Sometimes, a global variable may be used to hold certain value to be used in several computations. A modification of a variable is not allowed. This allows maximum concurrency and parallelism to be exploited since the sequentiality of a program is constrained by data dependency only.

(5) *To be user-friendly.*

SASCL meets this by:

(a) Choosing a textual form which is similar to that of well-known programming languages. The declaration part of a SASCL program is similar to that of C and PASCAL and the program body is similar to that of LISP;

(b) Making the data types, the data structures and the language expressions close to what systolic system designers normally use to describe their designs;

(c) Giving SASC as much intelligence as possible, so that user's responsibility to a design is limited to the algorithmic level description and probability of design errors is reduced by the subsequent automation.

3.4. The Language Features

The design of language features is one of the key points which may affect the merits of the target architecture and lead to clear, simple and reliable programming concepts when it is used.

(1) *Separated applications*

SASC supports two main applications : layout generation and system simulation (behavioral).

During the design process in the real world, people usually work on these two parts separately: they simulate the system first, then go on to lay out the design. This is because, due to the complexity of a VLSI design, translation from an algorithm down to a final layout may take a few months. So, people should ensure correctness of their algorithm before actually attempting the layout. However, concurrent systems are difficult to prove correct formally, and simulation is a valid means of validating them.

SASC allows a very fast validation of a design. The user having a validated design expressed in the form of a program can then generate the corresponding layout. If the generated layout is acceptable, and the simulation reveals no design error, the design process is completed. Otherwise he has to go back to modify his algorithm design.

The layout and simulation parts of SASC are actually separated. The data types used for these two purposes are completely different. A user working on systolic system design regards his variables as signal or primitive operators, while in a simulation, he would use value attributes, for example, integer, character or boolean. Actually, the data types in these two applications refer to different objects. For example, a signal in a parameter-list of the layout description refers to a physical entity, such as an I/O port, while a variable in the simulator describes the value of a signal carried by that port.

In conclusion, SASCL allows layout generation and simulation of a design to be done in two parts of a program that share some "syntactic" parts of the design. A linkage between the two could be established by a system function called "link".

(2) *Data Types*

SASCL supports two classes of data types, one for layout generation and the other for simulation.

Signal, flag and register are the data types for layout generation. Among these, flag and register are stored data types and signal is not. A variable having register data type could be used to hold values, numbers and characters. A variable having flag data type may have a boolean value true or false.

The data types supporting simulation are integer, character and logic.

(3) *Data Structures*

The data structure "bits" is used to describe signals, flags or registers which have more than one bit. The following are legal declarations (capital letters represent the reserved words of SASCL):

```
x: BITS 4 OF SIGNAL;
y: BITS n OF REGISTER;
```

The data structure "elements" could be used at both layout generation and system simulation. In the case of layout generation, for example, an "elements" may represent a set of variables:

```
rw: BITS 4 OF REGISTER;
w : ELEMENTS 3 OF rw;
```

defines a variable of register data type, four bits for each register.

(4) *Values*

SASCL does not allow any identifiers to be defined as a global updatable variable. It obeys SAR. SASCL allows an identifier bound to certain value to be accessed as a global variable only under the following case: a variable has to be globally initialized before systolic computations start. The problem for layout generation is that a global variable of register data type should be declared and the wire connections between the register and the loading I/O pads should be generated. Because I/O pads only could be called at the topmost level of design, variables of register data type should be declared as global ones.

(5) *Parallelism*

SASCL exploits maximum concurrency implicitly and explicitly. A user should know what concurrency the algorithm tolerates, but he does not take part in the exploitation of these concurrency in the implementation. Computations which do not have data dependency could be performed concurrently. For example, in the following piece of program:

```
                    .
                    .
                    .
c = (ADD ( ADD (a d) ADD (b d)))
                    .
                    .
                    .
```

the definition implies that the functions " ADD (a d) " and " ADD (b d) " could be performed concurrently.

Function "FSM" is designed for a direct description of finite state machines. Conditions in a FSM should be evaluated in parallel.

SASCL supports multi-dimensional parallelism. A number of system functions (see Appendix III) provide some means of exploiting the concurrency of algorithms. All functions for describing network topology in a layout generation program specify explicit concurrency also. For example,

```
LINEAR ( pe (xin  xout) 4);
```

will generate 4 independent tasks, each of them performing the computation defined in the function "pe". As a result, two levels of parallelism could be exploited :

(a) function LINEAR defines the top level concurrency explicitly : the four pe's work in parallel.

(b) inside each pe, computations without data dependency could be done in parallel (implicitly).

(6) *Sequentialism*

Even though SASCL is aimed at exploiting maximum concurrency of an algorithm, in some cases sequential executions are still unavoidable. Loops, for example, are the most general cases where computations should be done sequentially. SASCL supports this by using "FOR" statement. The format of it is similar to that of C programming language. An implicit sequentiality in SASCL is the data dependency of computations.

(7) *Synchronization of Computation*

As we mentioned in chapter 1, systolic array architecture is a special case of data flow computers. Sequentiality, in a data flow computer, is based on data dependency of computations. Synchronization is always required in a systolic array in order to achieve proper timing described in an algorithm. SASCL supports this kind of synchronization by two predefined functions, "synch" and "event".

"Synch" is a primitive function used to synchronize signals listed in its parameter list.

"Event" is a library function designed for describing synchronization of a set of computations where both inputs and outputs should be synchronized. The concept of "event" here is a high level timing abstraction which could be regarded as a time period. We assume that an event starts at some time and ends at some time afterwards. What an user should specify is what computations should take place inside this event, rather than actual starting and ending time. By calling this function, an user could simply put the input and output variables of the computation into the parameter list of the function event and separate the two parts by a semicolon.

(8) *Reflection on Systolic Architecture*

SASCL achieves this by the following main aspects:

(a) Some data types supported by SASCL have hardware correspondences. For example, a variable of signal data type represents a storage device, etc.

(b) A user-defined function at the highest level represents a processing element in a systolic array.

(c) Some system functions are defined to describe systolic features: for example, functions "LINEAR" and "SQUARE" for network topology and "FSM" for combinational logic, etc.

3.5 The Language Description

In this section, we give a general description of the syntax definitions of SASCL. Appendix II is a complete list of the syntax definition.

An algorithm description of a systolic array could be divided into two parts: processing element description and global algorithm description. A systolic computer usually contains a few types of processing element (PE). Each PE could be regarded as a black box with an interface and an algorithm describing the local computation. The applications of SASCL program for layout generation and simulation are separated. A SASCL program consists of four parts: program header, global declaration, function declaration and main program code, as in the following syntax.

```
<applications> ::= <layout> | <simulation>
<layout> ::= "layout" <array name> ";"
                  <global declaration>
                  <function definition>
                  <main code>
```

Global declaration is decomposed into two parts as shown below.

```
<global declaration> ::= {<const declaration>} <var declaration>
```

Constant declaration binds numbers to identifiers and variable declaration defines the variables to be used.

The syntax of the two declarations is given below:

```
<const declaration> ::= "const" <const list>
<const list> ::= <constant> {<constant>}
<constant> ::= <const name> "=" <numerics> ";"
<const name> ::= <identifier>
<var declaration> ::= "var" <var list>
<var list> ::= <variables> {<variables>}
<variables> ::= <name list> ":" <var type> ";"
<name list> ::= <var name> {"," <var name>}
<var name> ::= <identifier>
<var type> ::= <single type> | <bits type> | <elements type>
```

<single type> ::= "signal" | "register" | "flag"

<bits type> ::= "bits" <digit> {<digit>} | <const name> "of"
<single type>

<elements type> ::= "elements" <digit> {<digit>} | <const name> "of"
<var name>

Here is an example of global declaration:

```
CONST
  word = 8;
  VAR
    x : SIGNAL;
    y : BITS word OF REGISTER;
    z : ELEMENTS 4 OF y;
```

The SASCL encourages hierarchically structuring a design by using both user defined and library functions. It allows a higher order function take another function as argument, For example, consider a systolic array with two types of PE's, PE1 and PE2 which are connected alternatively. Fig. 3.5.1 shows the interconnection network. Hierarchically we may define a function called PE which consists of PE1 and PE2. The new network structure is shown in Fig. 3.5.2. Then, a library function, called LINEAR, could be called to construct the whole array.

A user defined function is defined by the following syntax:

<function definition> ::= <function head>
<ports declaration>
<function body>

<function head> ::= <function name> <parameter list>

<function name> ::= <identifier>

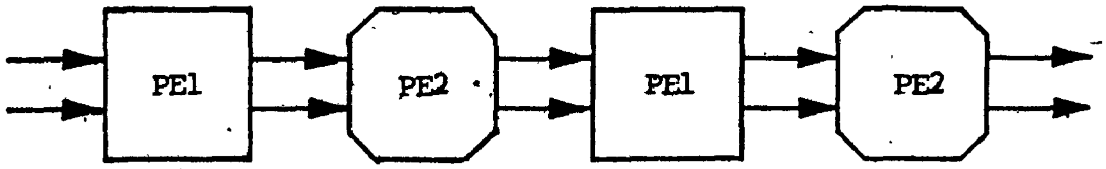


Fig. 3.5.1 An Example Of Two Types Of PE's.

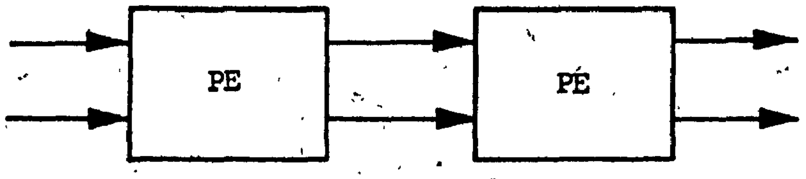


Fig. 3.5.2 The Higher Level Design Hierarchy.

```
<parameter list> ::= "(" <logic direction> {<logic direction>} ")"
<logic direction> ::= <direction> ":" {<I/O pairs> |
                        <inport name> | <outport name>}
<direction> ::= <main sides> "->" | "<-" <secondary sides>
<main sides> ::= "L" | "T" | "TL" | "TR"
<secondary sides> ::= "R" | "B" | "BL" | "BR"
<I/O pairs> ::= <I/O pair> {<I/O pair>}
<I/O pair> ::= <inport name> "," <outport name> ";"
<inport name> ::= <identifier>
<outport name> ::= <identifier>
```

I/O signals are declared at the head of a function and the logical data flow directions are defined in the parameter list. There are two types of I/O signals: single bit or multi-bit signal. An input and an output signal pair forming a generic data flow is separated from others by a semicolon, and an input signal is separated from an output signal by a comma. Logic data flow direction is proposed for the convenience of describing systolic algorithms. However, this generic information is useful as a reference for placing and routing low level hardware entities.

The logic data flow direction could be defined by four main directions: Top->Bottom (T->B), Left->Right (L->R), TopLeft->BottomRight (TL->BR) and TopRight->BottomLeft (TR->BL). The opposite directions are described by changing the direction sign " -> " into " <- ". Fig. 3.5.3 shows the four main directions. Here is an example of function definition :

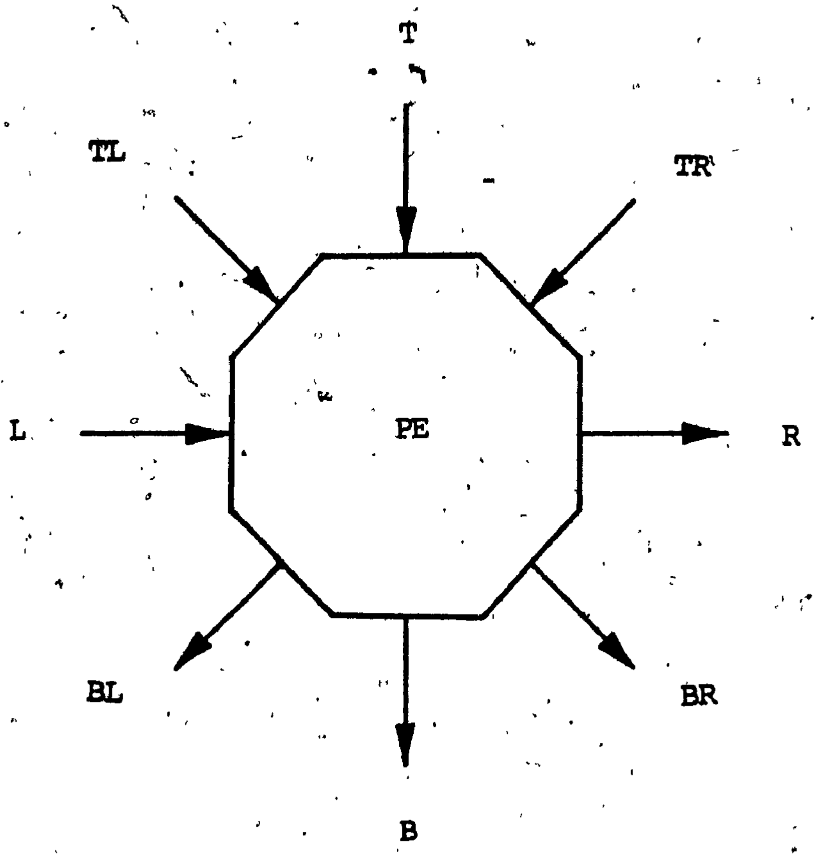


Fig. 3.5.3 The Main Logic Transmission Directions.

```
multiplier ( L->R: ai,ao; T->B:bi,bo; TL->BR:ci,co);

          ai,ao,bi,bo,ci,co: BITS 8 OF SIGNAL;

  BEGIN
    ....... ;
  END.
```

By using the above function a library function SQUARE will generate a mesh structure with diagonal connection between PEs (or a 2-D hexagonal network). A 3 by 3 systolic array for multiplication could be defined by the following function call:

```
SQUARE (multiplier (ain aout bin bout cin cout) 3 3).
```

The port type definitions are given by:

```
<ports declaration> ::= <ports name> ":" <ports type>
<ports name> ::= <port name> {<port name>}
<port name> ::= <inport name> | <outport name>
<ports type> ::= "signal" | <mult bits>
<mult bits> ::= "bits" <digit> {<digit>} | <const name> "of" "signal"
```

In the example given above, we can see that port type declaration is a special case of global declaration -- only data type "signal" and data structure "bits n of signal" are allowed.

The syntax definitions of a function body are:

```
<function body> ::= "begin" <function list> "end" ";"
<function list> ::= <local declaration> <statement list>
<local declaration> ::= {<const declaration>} <var declaration>
<statement list> ::= <statement> {<statement>}
```

```
<statement> ::= <assignment statement> | <function call> ";"
                    {<comments>}
<assignment statement> ::= <normal assign> | <state assign>
<normal assign> ::= <var name> "=" <var name> |
                        <boolean const> | <function call>
<state assign> ::= <var name> "=" <symbols> ";"
<symbols> ::= <identifier> {"," <identifier>}
```

A function body contains a set of assignment statements and function calls. Assignment statements mostly look like :

```
xout = x ;
```

State assignment is for naming states of a FSM. For example, to declare a finite state machine with four states, U,V,W and X we use:

```
VAR
 s : FLAG;
st : ELEMENTS 4 OF s;
```

In the function body, the state assignment statement should be:

```
st = U,V,W,X;
```

It says that the four states of "st" are called U,V,W and X, respectively.

A function can call either a system function or a user defined function according to the following syntax.

```
<function call> ::= <function names> "(" <argument list> ")"
<function names> ::= <function name> | <sysfunction name>
```

$$<\text{argument list}> ::= \{<\text{function call}> \mid <\text{continued symbol}> \mid <\text{const name}> \mid <\text{digit}>\}.$$

A higher order function may use another function as argument. For example, the function "pe" in the following example is the argument of the function LINEAR.

LINEAR (pe (x xout y yout) n);

A set of primitive functions and library functions is defined and it contributes to the main power of SASCL. Primitive functions are a set of functions to describe arithmetic, logic and comparison operations. Library functions reflect systolic algorithm features. The definition of these predefined functions are given in Appendix III.

3.6 The Semantic Considerations

In this section, the main semantic considerations guiding the language design are presented.

(1) *Continued symbols*

In a systolic design, continued numbers and letters are used quite often. We use double dot sign, "..", to denote this kind of continuation. For example, if we have a variable declaration as shown below:

```
VAR
x : REGISTER;
y : ELEMENTS 4 OF X;
```

an assignment in a function might be

y = 1..4;

During the compilation process, 1, 2, 3 and 4 will be assigned to y1 ,y2 ,y3 and y4 respectively.

(2) *Defaults*

By using hierarchical design methodology, a systolic array could be abstracted to have unique types of processing elements. A network contains nothing but a number of duplications of the defined PEs and a set of wires to connect them together.

To reflect this, SASCL allows a user to define I/O ports of a top-level processing element as global variables and the compiler will take care of the default names of the duplicated ones. For example, in

```
LINEAR ( pe (x xout y yout) 5 );
```

signals x and xout will represent x1,x2, ..., x5 and xout1,xout2, ..., xout5, etc. respectively. Default wires connecting the I/O ports are also implied by the above definition. I/O ports at the boundary of the array will be connected to I/O pads which will be assigned the same name.

For all the predefined functions (primitive and library functions), power, ground and system clocks are the only common default signals.

Subscript default is used in algorithm description. For example, if a variable is defined as a multi-bit register:

```
VAR
  x : BITS 4 OF REGISTER;
```

the variable name associated with a subscript is allowed to be referred directly.

For example, x[1] refers the first bit of register x.

(3) *Algorithm conversion*

According to the structure of data to be processed, we may classify systolic algorithms into four categories: bit-serial, bit-parallel, word-serial and word-parallel [41]. SASC not only supports bit-serial implementation, but also accepts other types of algorithms and transforms them into equivalent bit-serial ones. The first two types of algorithms could be directly supported by bit-serial library instances invoked by the compiler. The last two categories include algorithms requiring parallel processing operators. Two kinds of conversions are required :

word parallel ----> bit parallel

word serial ----> bit serial

SASCL provides ways to describe all these kinds of algorithms and the compiler has the ability to identify them and carry out proper conversions, transforming an algorithm into a bit-serial equivalent one with suitable retiming (data skewing). Since both transformations listed above could be done without changing the network topologies, the need of a transformation could be detected by checking the variable declaration made for the ports. The correctness of the transformation are ensured by the compiler, without user knowledge and other interaction.

(4) *Two-level pipelining conversion*

Beyond the maximum concurrency defined in a user program, one more level of pipelining may be added as a consequence of internal hardware details to enhance the system throughput. Users may not be aware of two-level pipelining. The compiler generates the proper system structure, invokes the corresponding primitive operators, sets up the necessary wire connection, and synchronizes the logical events. We assume that if a function contains arithmetic operators only, then a conversion from one to two-level pipelining structure will be conducted without user interaction.

3.7 Specifying An Algorithm To The Compiler

A number of systolic algorithm design process have been studied. Generally, the following approach is preferable to describe an algorithm in SASCL. Considering the way a systolic array works, the processing elements could be regarded as black boxes which are capable of performing certain computation. Data with fixed flow pattern are being pumped into the boxes. Information flow between PE's in a pipelined fashion, and communication with the outside world occurs only at the "boundary" PE's. We would like to separate the problem of a system design into three parts:

(1) *Processing element description:*

Assume that a systolic array consists of a few kinds of PE's, each of them can be represented by a procedure (or subroutine). The I/O ports of the PE's are the windows of the black boxes, connecting to their neighbors and represented by the parameters passed to the procedure. The body of a procedure is the algorithmic description of the computation performed. Variable declarations should be made at the beginning of a procedure. A set of primitive functions ("add", "mult" and "shift", etc,) can be used to describe the computation. The data flow directions through the PE should be defined in the parameter list.

(2) *System timing and data flow description:*

Logic synchronization is one of the most crucial parts of digital system design. The functions "delay", "synch" and "event" provide means to describe timing synchronization at different levels.

(3) *The network topology:*

Interconnections in systolic arrays are very regular. SASCL provides a set of library functions ("linear", "square", etc,) to represent network topologies. These library functions may take a procedure as a parameter like other functional programming languages.

Beyond the above three, a hierarchical manner of structuring a design is strongly encouraged. We assume that there is only one type of PE at the topmost level.

The following is a digital filtering algorithm called Finite Impulse Response (FIR) filter. Mathematically, a FIR filtering problem is defined as follows:

Given: the weights $\{w_1, w_2, \dots w_h\}$

the initial values $\{y_0, y_{-1}, \dots, y_{-k+1}\}$ and

the input data $\{x_1, x_2, \dots, x_n\}$.

Compute: the output sequence $\{y_1, y_2, \dots, y_{n+1-h}\}$ defined by

$$y_i = w_j * x_{i+j-1}.$$

The equation can be interpreted as below if h = 4:

$$y_1 = w_1 * x_1 + w_2 * x_2 + w_3 * x_3 + w_4 * x_4$$
$$y_2 = w_1 * x_2 + w_2 * x_3 + w_3 * x_4 + w_4 * x_5$$
$$y_3 = w_1 * x_3 + w_2 * x_4 + w_3 * x_5 + w_4 * x_6$$
$$y_4 = w_1 * x_4 + w_2 * x_5 + w_3 * x_6 + w_4 * x_7$$

The systolic array performing the FIR algorithm is shown in Fig. 3.7.1: The algorithm can be described as following:

(1) During system initialization (INIT(w)), the weights are loaded in wr's.

(2) For each system clock cycle, the operation of a cell is:

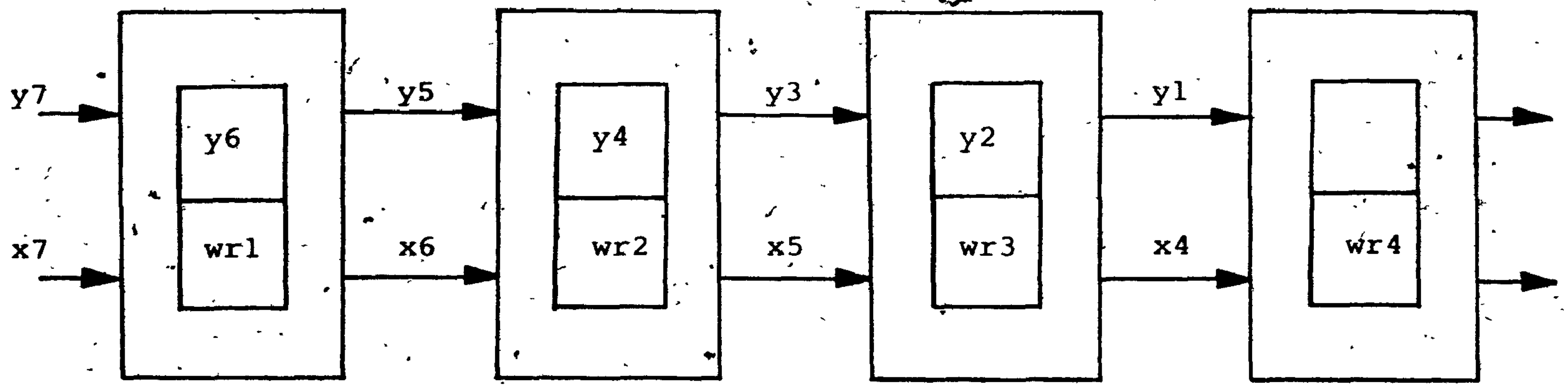


Fig. 3.7.1 The Systolic Array For The FIR Algorithm.

(a) Receive xi and yi from two input ports and set two internal registers ry and rx to xi and yi, respectively.

(b) Send previous y to the output port yo, send xi to xo and compute ry = wr * xi + yi.

The data travel by moving through one cell per clock cycle. Following is one of the possible SASCL programs for the FIR.

```
/* layout-generation program for FIR filtering algorithm */
 LAYOUT fir;

/* constant declaration */
  CONST
   word = 4;
   n = 4;

/* variable declaration */
  VAR
   xin, yin, xout, yout : BITS word OF SIGNAL;
   wr : BITS word OF REGISTER;
   w : ELEMENT n OF wr;

/* function definition -- processing element "pe" */
 pe (L->R : xi, xo; yi, yo);
/* declare I/O signals */
  xi, xo, yi, yo : BITS word OF SIGNAL;

  BEGIN
/* local variables */
   VAR
```

```
      rx, ry : BITS word OF REGISTER;

  /* synchronize I/O signals */
     EVENT ((xi yi) (xo yo));
     rx = xi;
  /* ry = wr * x + ry */
     ry = ADD (MUL (wr rx)  ry);
     xo = rx;
     yo = DELAY (ry 1);
    END;

  /* main program begin */
   BEGIN
  /* initialize w */
    INIT (w);
  /* generate the linear network */
    LINEAR (pe (xin xout yin yout) n);
  /* end of the main program */
   END.
```

CHAPTER 4

DERIVING SYSTOLIC ARRAY FROM ALGORITHM DESCRIPTION

When designing a silicon compiler, one may immediately find that directly going from behavior to layout is too complicated to be completed in a single translation step. Technology-dependency is a common feature of silicon compilers since they are always designed to support certain fabrication process. Today, advances in VLSI technology change the fabrication process significantly and frequently. In order to face this challenge, it is required to minimize the work involved in adapting to technology changes and separate it from other parts of the compiler. In SASC we achieve this by using a two stage translation. SASC, in the first stage, translates the SASCL description into an intermediate language description called the Systolic Array Silicon Compiler Intermediate Language (SASCIL), which is then translated into the final layout or behavioral description during the second stage.

4.1 The Design Objectives And Considerations

The key point of the SASCIL design is to select a proper level (between the high level language and the low level object code) where SASCIL should stay. Being at that level, SASCIL should make the two transformations of SASC (SASCL to SASCIL and SASCIL to object code) easy to deal with.

In the high level transformation process from SASCL to SASCIL, SASC performs most of the technology independent translations. For example, the parallel to serial algorithm transformation, the two-level pipelining transformation and automatic composition of timing and simulation details, etc. are carried out during this time. In the intermediate form, library functions of

SASCL become a set of primitive calls and logic connections in the SASCIL formats.

In the transformation process from SASCIL to the object code, logic representation of primitive operators and wires should be translated into physical ones. Besides the information carried by SASCIL, there are some other unknowns at this level. For example, the dimension of an user defined function block or the locations of special signals, e.g. Vdd, Gnd and Clocks, should be given by the placer or router. We classify the information to be passed between the two transformations into several categories and give a representation to each of them in SASCIL.

The design objectives of SASCIL are listed in the following:

(1) SASCIL is aimed as a simple and effective tool which isolates the technology-dependent parts of the compiler from the high level language completely in order to support technology independency of SASCL and obtain good system maintenance. SASCIL meets this by:

(a) Abstracting the level of data description at the logical one. For example, the function call in SASCL

c = add(a b);

may describe the computation of adding signals a and b, and putting the sum into the register c. The corresponding intermediate code

[p1 lib 0 Adder ... t ... a b ...]
[p2 register ... c ...t ...]

has nothing related to the technology dependent parts of SASC (for instance, physical features of the calling primitive, etc.). It only describes the logic relations of the signals to be used. The isolation of the technology dependent parts from the high level translation process supports a better system maintenance.

(b) Choosing proper data types to reduce the complexities of the two level translation as well as the amount of data to be described by the intermediate form. SASCIL divides the information for layout generation into seven categories: signal, port, temp(orary), register, boolean and lib(rary). The data types we use are intended to make a conceptually clear boundary between the two levels of compilation without increasing the complexities. The types signal, register, boolean and primitive library functions have data type or language construct correspondences in SASCL. They are easy to be identified and translated. The type port is selected in order to give a clear distinction between signals at I/O port and somewhere else. The low level translation will benefit from the use of the port data type which provides the logical location of I/O signals as references to the placer. "Temp" is another data type without direct language correspondence in SASCL. It gives convenience to the high level translation. The data type "lib" contains library primitives (except the register type) and user-defined functions. We take data type register out of type "lib" since it is a special case in the primitive library.

(c) Using regular form of language expressions. Only two kinds of statements are used: assignment statement and function call statement. It supports the simplicity and regularity of the language.

(2) SASCIL should provide a complete set of information needed for generating low level object code. SASCIL achieves this by:

(a) Providing two sets of language constructs, one for layout generation and the other for behavioral simulation.

(b) Using data types which convey the complete information required for layout generation to the low level translation process.

(3) SASCIL should be readable and has a simple and clear format to help debugging.

This is one of the important features of SASCIL. It is required since: (a) Technology dependency is one of the crucial problems faced by silicon compiler designers. When technology advances, the low level database should be updated and debugging may be required from the intermediate code to the final output. (b) SASCIL is also an interface to the simulator. The readability of SASCIL will help the debugging in both the above cases. This objective is achieved by:

(a) Having data types and primitive operator names in a readable text form. If signals are explicitly defined in SASCL programs, use the same names for the signals in the SASCIL description.

(b) Clearly distinguishing user-defined functions which are included in a pair of brackets "{" and "}".

(c) Reducing the number of unreadable representations as much as possible and let boolean type of data have an equation form, for example,

[pp boolean ... a * b + c * d].

4.2 The Language Description And Semantic Considerations

We divide the low level primitive operators and the intermediate information into two categories which are defined by the following syntax :

```
<statements> ::= {<statement1> | <statement2>}
<statement1> ::= <items> "=" <items> | <boolean type>
<items> ::= <simple type> | <register type>
<simple type> ::= <signal-temp> | <port>
<signal-temp> ::= "[" <identifier> "signal" | "temporary" "]"

    .
    .
    .

<statement2> ::= <lib type> | <event type>

    .
    .
    .
```

The first category consists of four subtypes, "signal", "temporary", "register" "port" and "boolean". All these data types except the temporary data type (which is going to be removed during the low level translation) have hardware correspondences. For example, a "signal" corresponds to a wire and a "register" may refer to one of the different types of registers in the hardware primitive library.

Logic operations, function FSM and condition statments of SASCL are going to be translated into "boolean type" in a form defined as follows:

```
<boolean type> ::= "[" <boolean head> <equation> "]"
<boolean head> ::= <identifier> "boolean" <#variable> <#product>
                        <#member> <speed factor>
<#variable> ::= <digit> {<digit>}
<#product> ::= <digit> {<digit>}
<#member> ::= <digit> {<digit>}
```

<speed factor> ::= "f" | "m" | "s"

<equation> ::= { <identifier> <logic symbol> } <identifier>

<logic symbol> ::= "*" | "+"

During the low level translation process, the interpreter translates boolean type intermediate codes into a two-dimensional product/variable table, then calls the logic compiler to generate the corresponding layout geometry.

In the second category, "library type" could be decomposed into "primitive library" and "user defined library". "Primitive library" here is predefined functions in the SASCL and "user defined library" may contain primitive calls, user defined functions and "simple type" definitions, etc.

"Event type" contains timing information which will be used for simulation purposes. At different levels of the design hierarchy, all paths an event may pass could be traced. Main attributes of an event could be found immediately following the event identifier.

The complete syntax definition of the intermediate language is given in Appendix IV.

4.3 Deriving The Systolic Array From The Algorithm Description

Deriving the systolic array form the described algorithm is a complicated process. In this section we give a detailed discussion on the high level transformation process with the FIR algorithm described in section 3.7, bringing out many of the novelties of SASC.

Conceptually the high level translation process should be carried out in the following aspects:

(1) Identify the type of the algorithm. Recall the discussion we have in

section 3.6. The algorithm transformation performed by SASC converts parallel operators and wire connections to bit-serial ones without changing the network topology. A bit-parallel algorithm is detected by checking the variable declarations at the I/O ports. Signals are transmitted either bit-serially on single wires or bit-parallelly on parallel bus. We assume that the way of data processing is same as the way of data transmission. In other words, SASCL does not support algorithms having bit-serial communication and bit-parallel operators or vice versa. Fig. 4.3.1 shows the communication strategies supported by SASC target architecture. The FIR algorithm employs bit-parallel operators as well as bit-parallel communication strategy.

(2) List primitives and signals that are explicitly defined, for example, "add", "sub", etc. This is the prime information of the target architecture.

(3) Add the defaults. The SASCL allows some implicit descriptions of systolic algorithms. The defaults which should be derived at this stage are:

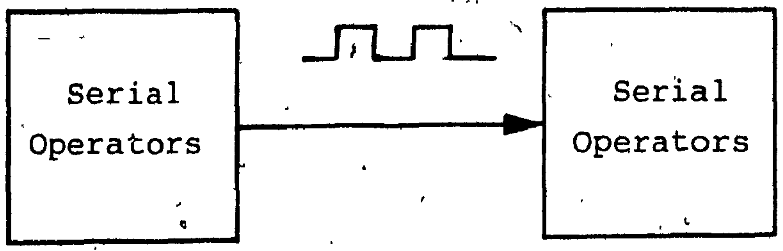
(a) Primitives at I/O ports, for example receivers, senders and corresponding signals, etc.

(b) Default interconnections of library function calls.

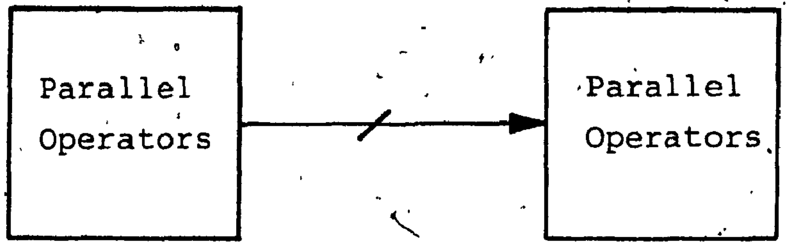
(c) Default nodes of primitive function calls, for example the statment

s = add (a b c);

may be interpreted as:



(a) Bit-serial.



(b) Bit-parallel.

Fig. 4.3.1 The Communication Strategies Of Target Architecuture.

t = add(a b); and

s = add(t c);

The "t" represents a node of temporary data type. The common default variables, power, ground and Clocks, are not added at this level since they are physically dependent on the placement and routing.

(d) Default I/O ports: a pair of I/O ports are required when a register data type variable needs to be initialized. We assume that initialization is done bit-serially.

(4) Calculate the timing required by the algorithm. Data paths traveled by signals should be traced. Proper time delays are added according to the primitives used and the interpretation of the time functions.

(5) Transform the given algorithm to a bit-serial equivalent one, if it is necessary. An automatic algorithm transformation should be conducted in order to generate the target architecture supported. The transformations are carried out according to the following rules:

(a) Replace each K-bit data wires by one-bit data wire.

(b) Replace each word-operator by a bit-operator.

(c) Replace each K-bit register by a K-bit shift register.

(d) Change the timing specified in the algorithm according to the serial architecture implementation.

(6) Conduct one to two level pipelining transformation. There is no explicit language feature for representing the one to two-level pipelining conversion since it is assumed that the conversion is automatically carried out by the compiler without user awareness. The purpose of introducing two-level pipelining is to improve system throughput for

computation-intensive problems. Algorithms with pure arithmetic computations are selected to have such conversion. The number of stages of the second level pipelining is chosen by the compiler. The correct timing should be ensured corresponding to the bit-serial implementation of the original algorithm (if an algorithm transformation is done).

Now, let's look at the FIR algorithm given in chapter 3 again and illustrate details of the translation. The key word LAYOUT denotes that it is a layout generation program. Global variables are declared under the CONSTants and VARiables. The processing element is defined by the user defined function "pe". Variables in the parameter list of "pe" represent multi-bit signals across boundary and the data flow, xi -> xo and yi -> yo, travel from the left to the right hand side. The computations that take place are multiplication and addition only. This is a parallel algorithm with pure arithmetic operation.

The primitives explicitly declared in the algorithm are:

Signals:	xi, xo, yi, yo (for function pe)
	xin, xout, yin, yout, win, wout (for function fir)
Registers:	rx,ry, rw
Primitive-functions:	add, mul
Library-functions:	init, event, linear.

The defaults involved in the algorithm description are:

I/O-ports:	wi, wo (for function pe).
Primitives:	xi, yi, wi (receivers).
	xo, yo, wo (senders).

Signals: for default primitives.

for network connections:

xin1, xin2, xin3, xin4,

yin1, yin2, yin3, yin4,

win1, win2, win3, win4,

xout1, xout2, xout3, xout4,

yout1, yout2, yout3, yout4,

wout1, wout2, wout3, wout4.

Nodes: for function call: mul(wr rx);

User-defined-functions: pe1, pe2, pe3, pe4.

The statement "yo = DELAY (ry 1)" says that signal yo is obtained by delaying ry one time period (which is the time of one word delay for the word-serial algorithm). The first bit of signal yo should appear at the output port four cycles after the first bit of the signal gets out of register ry since the algorithm transformation is done.

After the transformation, signals xin, yin, xout and yout should be one-bit wide and register wr, rx and ry are 4-bit shift registers in the FIR algorithm.

A pair of storage registers is required for a pair of assignment statements shown below:

rx = xi;

xo = rx;

Register rx could be regarded as a receiver which takes input from signal xi, and a default register (e.g. named rxs) is required as a sender and takes the output of rx as input data. The output of register rxs is connected to signal xo. The rx and rxs work at clock phase one and two, respectively. If an algorithm transformation is conducted (like in the example), either rx or rxs should be replaced by a shift register. The number of bits to be added is dependent on the word-length of the systolic array and the timing specification of the original algorithm. Since signals xi and xo are synchronized with the signals yi and yo, respectively (defined by function EVENT), the time delay required should be calculated according to the longest data path, that is, the data path for yi and yo. Two cycle delay is required (one for multiplication and one for addition). Assume that register rxs is a shift register. It should be (n+c) bits long where c is the number of cycles that the computations take minus one.

Function INIT indicates a register type of variable has to be initialized through terminals of D-flip-flops. The first parameter of the function is the name of the variable to be initialized and the second one defines the terminals by which the initialization is done. In particular, the default is terminal "d" (like in the given example); "0" and "1" specify terminals "reset" and "set", respectively. Signal win is delayed n cycles before it reaches the port wout.

Function LINEAR defines a linear array of four elements performing the computation specified in the function "pe". This translation is done in the low level translation process since it is unnecessary to generate four pe's. They can be obtained by duplicating the pe four times when it is composed in the low level translation process. The network interconnections are specified explicitly by means of assignment statements. Default is used to represent the signal names of the I/O ports of each processing element. For instance, the global signals xin and xout are a pair of signals corresponding to the signals xi and xo in the function pe. As a consequence a set of intermediate code is derived for the given FIR algorithm and it is shown below. The comments and line numbers used here are for explanation purpose only, they are not part of the intermediate form. Symbol "!" means that the corresponding terminal of that primitive is not used. It happens when (1) a signal of a primitive has more than one I/O ports located on different sides of the layout boundaries or (2) some terminals of primitive operators like "set" or "$\bar{q}$" of a D-flip-flop, are logically not used. Physically the terminal "set" should be connected to Vdd in the low level translation process.

```
/* define the fir array */
[ fir lib 1 pe 0 4 2 6 ( xin1 xin4 ) ( yin1 yin4 ) ( win1 win4 ) ]
{ /* define the network connections */
  [ xin2 signal ] = [ xout1 signal ]
  [ xin3 signal ] = [ xout2 signal ]
  [ xin4 signal ] = [ xout3 signal ]
  [ yin2 signal ] = [ yout1 signal ]
  [ yin3 signal ] = [ yout2 signal ]
  [ yin4 signal ] = [ yout3 signal ]
   [ win2 signal ] = [ wout1 signal ]
   [ win3 signal ] = [ wout2 signal ]
   [ win4 signal ] = [ wout3 signal ]
   /* define the processing element "pe" */
   [ pe lib 1 3 1 4 2 6 ( xin xout ) ( yin yout ) ( win wout ) ]
   { [ s1 signal ] = [ win port i l ]
      [ s2 signal ] = [ yin port l l ]
      [ s3 signal ] = [ xin port l l ]
      [ s4 signal ]  = [ xout port o r ]
      [ s5 signal ] = [ yout port o r ]
      [ s6 signal ] = [ wout port o r ]
      [ s6 signal ] = [ p3 register 0 1 2 n1 s6 ! ]
      [ p3 register 0 1 2 n1 s6 ! ] = [ p1 register 5 4 1 s1 n1 ! ]
      [ p1 register 5 4 1 s1 n1 ! ] = [ s1 signal ]
24    [ s5 signal ] = [ p8 register 3 4 2 n6 s5 ! ]
25    [ p8 register 3 4 2 n6 s5 ! ] = [ p7 register 3 4 2 n5 n6 ! ]
26    [ s4 signal ] = [ p9 register 3 5 2 n3 s4 ! ]
27    [ p9 register 3 5 2 n3 s4 ! ] = [ p4 register 0 1 1 s3 n3 ! ]
28    [ p2 register 3 2 1 s2 n2 ! ] = [ s2 signal ]
29    [ p4 register 0 1 1 s3 n3 ! ] = [ s3 signal ]
30    [ p5 lib 0 Adder 1 1 2 6 n5 ! n2 n4 ! ! ]
31    [ p6 lib 0 Multiplier 1 1 2 7 n4 n1 n3 ! ! ]
32    [ p10 lib 0 Clockgen30 1 1 1 3 C1 C1- IN1 ]
33    [ p11 lib 0 Clockgen30 1 1 2 3 C2 C2- IN2 ]
34    } /* end of the pe */
35  } /* end of the fir */
```

Lines 14-34 are the intermediate code corresponding to function "pe" in the FIR algorithm. The port declarations of the "pe" are represented by lines 15-20. If a signal is declared as an I/O port, an assignment statement like line 16 should be given in the intermediate form. The type "port" is assigned to the declared signal and the type "signal" (which represents wire connection) is given to a new name provided by the high level translation process. Lines 22-33 describe the computations the "pe" performs. For instance, line 30 says that p5 is a predefined library function working at clock phase two in level 1 of the design hierarchy. The two input and one output signals used are n2, n4 and n5, respectively. The intermediate code inside the inner pair of brackets ("{" and " }") define the function pe. The systolic array "fir" is described in lines 1-14, and 35 which describe nothing but the network interconnections. The target array corresponding to the above intermediate code is shown in Fig. 4.3.2.

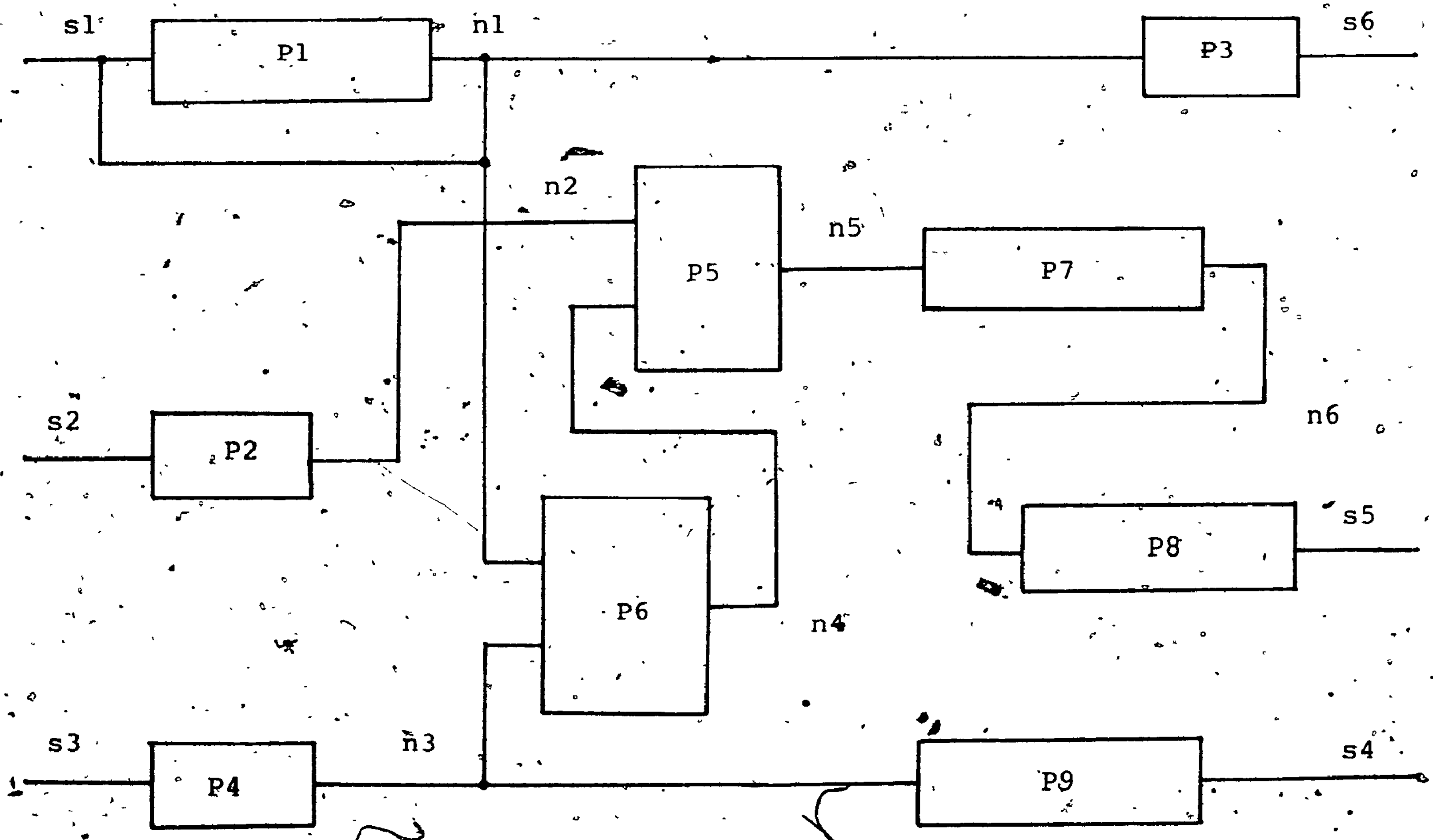


Fig. 4.3.2 The Target Architecture Of The FIR Algorithm.

CHAPTER 5

THE COMPONENTS OF SASC

We discussed in the previous chapter how a systolic array is derived from algorithm description and how the high level translation process (from the SASCL to the SASCIL) is conducted as well as the intermediate language design. In this chapter, we will show how a layout is generated from the intermediate form.

5.1 The Layout Generation Hierarchy

A VLSI design is hierarchical in nature. Special system structures are usually required. A system could be partitioned into function blocks each performing a subset of the computation. Then the function blocks could be either divided into some sub-function blocks or contain a set of computations described by the library functions provided by the description language. Fig. 5.1.1 shows the hierarchy of the design methodology supported by SASCL and indicates that at least three levels of hierarchy are included in a single design.

In contrast to the top-down design methodology supported by the high level description language, the composition of a layout is carried out bottom-up. The layout generation hierarchy is given in Fig. 5.1.2.

The lowest level composition is for primitive operators. A primitive operator can be parameterized and formed by calling a fixed set of leaf cells and linking them together according to the predefinition. The initial placement of some primitives are predefined manually and generated by a set of composition procedures. Other primitive operators may have variable parameters. For example, shift registers may have parameters such as word length and type of

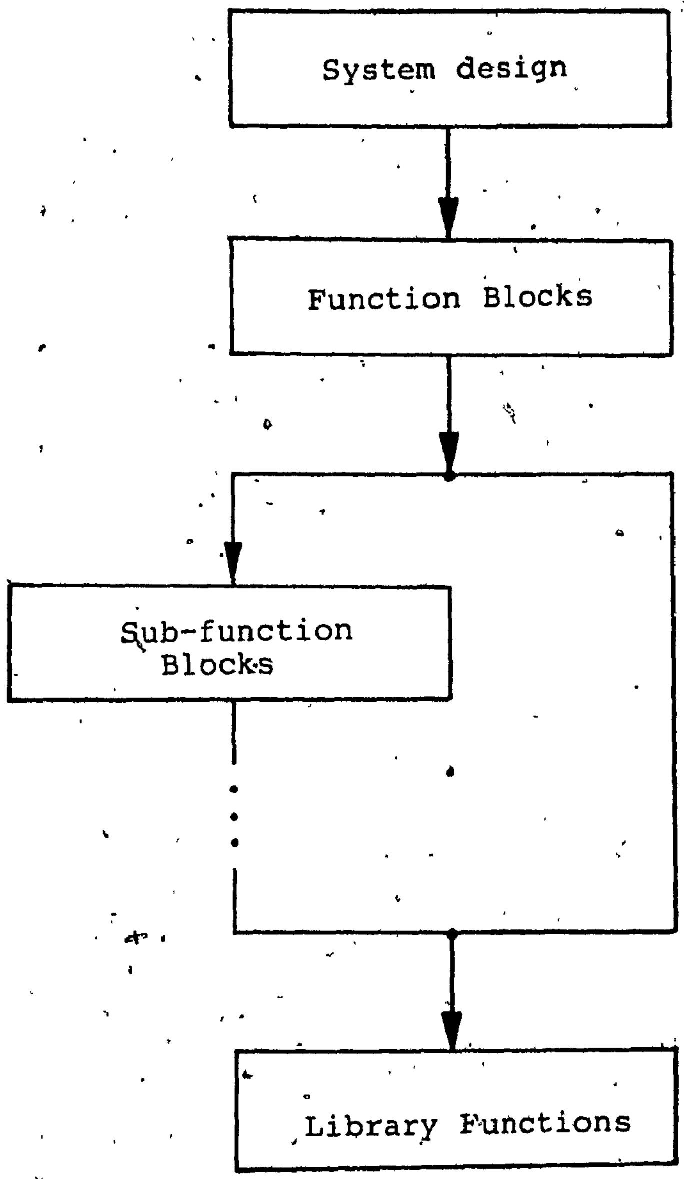


Fig.5.1.1 VLSI Design Hierarchy.

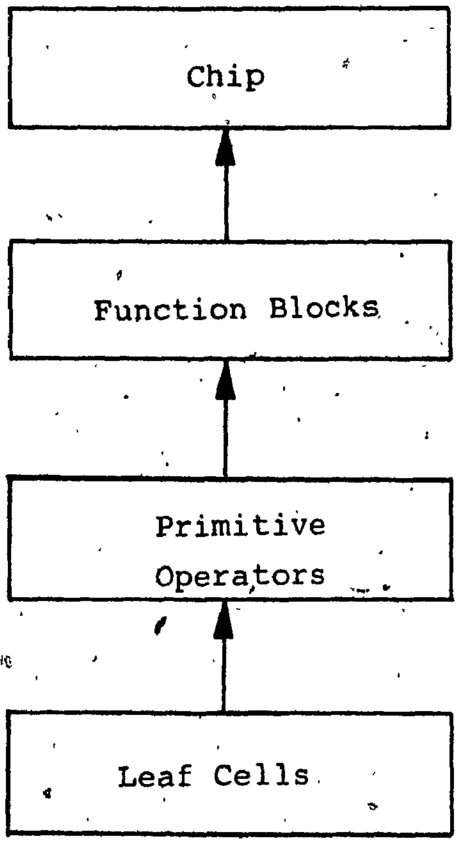


Fig.5.1.2 Layout Generation Hierarchy.

storage device used, etc. The corresponding composition routines are capable of taking into account these parameters and generating proper layout. The physical size of a primitive is not known until its composition process is completed.

The middle level composition is designed for function blocks. A function block is a collection of logically related primitive cells. Even though the main objective of SASC is to reduce engineering effort and design turn around time rather than minimizing chip area utilization, there are still some tradeoffs between the extent of automation and the penalty paid in silicon area cost. The complexities of the primitive operators varies from a transistor pair (e.g. inpad) to some powerful function blocks having hundreds of transistors inside, such as comparator, multiplier, etc. The size of primitive operators varies from a hundred by a hundred square maicrons to thousands by thousands square microns. Moreover, processing elements of systolic arrays usually have very simple functionality and a few communication ports. If a PE is specified in 5 levels of design hierarchy and the placement and routing is done for each of them separately, some gaps between large rigid building blocks are difficult to avoid. Great efforts we make in manual design of leaf cells in order to reduce the silicon area cost would be worth nothing because of the area wastage due to large gaps which may appear in each level during the design. The solution to this is to allow user to specify their designs using as many design levels as they want, but performing the middle level placement and routing only once. In other words, all primitive operators used in a PE will be placed and routed at the level next to the topmost one. The small building blocks (primitive operators) could take into account the global placement and routing requirements, tailor the shape of the PE and avoid large gaps generated between large blocks. The traffic in the routing channel will not be a problem due to the simplicity of systolic arrays and the general complexity of primitive cells. Achieving a better placement of all primitive operators, of course, is harder than that for leaf cells, but it is not a very difficult and computation intensive task under our situation of compiling systolic arrays.

The high level composition is for final floor plan generation of a chip. The placement and routing are taken care of by the SASC automatically according to the placement algorithm employed and the data generated by the lower level composition without user knowledge and interaction.

5.2 The Leaf Cells

The leaf cells are the basis of the central database to support the entire SASC system.

5.2.1 Selection of Leaf Cells

Leaf cells are the building blocks of primitive functions. As soon as primitive operators are selected, the functionalities of leaf cells could be decided. A number of standard leaf cells are defined and listed in Fig. 5.2.1. The programmable layout generation for all combinational logic modules greatly simplifies the construction of the library.

5.2.2 Leaf Cell Conventions

In order to guide the automation process and obtain a good library maintenance, leaf cell conventions were carefully defined. The decisions made are mainly based on the system assumptions meeting the design objectives, the horizontal symbolic layout rules [80] and the technology constraints, and the estimation of silicon area cost of the minimum logic element.

(1) Two-phase clocking scheme was chosen as the logic implementation

NAME	FUNCTION	
driver	inverted driver	driving
ndriver	non-inverted driver	driving
dlatch	D-latch	bit-delay
drs	D-flip-flop with R/S	bit-delay
fba	full binary adder	arithmetic
multiplexer	4-->1 switch	control
demultiplexer	1-->4 switch	control
logic compiler	logic compilation	combinational logic generator
inpad	input pad	chip generation
outpad	output pad	chip generation
vddpad	power pad	chip generation
gndpad	ground pad	chip generation
frames	frames	chip generation

Fig.5.2.1 List Of Leaf Cells.

structure. The supported CMOS fabrication technology determines the use of complementary clocks in each phase in order to drive both P and N types of transistor.

(2) Leaf cells in the library have variable lengths and standard heights. The standard heights are used in order to simplify the placement and routing algorithms. There are three possible heights of leaf cells (for the CMOS1B technology):

1H = 165 micron (11 ports on left/right sides)

1.5H = 261 micron (17 ports on left/right sides)

2H = 341 micron (22 ports on left/right sides)

(3) Each leaf cell is considered to be placed on a grid. The origin of the leaf cell is taken to be its left bottom corner. The separation of grid lines decided by the minimum port to port spacing in the given technology. The horizontal grid lines start from the origin and are located on the positions of multiples of 16 (minimum poly contact cut dimension 11 microns plus minimum poly to poly spacing 5 microns). The vertical grid lines sit on the positions of

$$l + 16 * x \ (x = 0, 1, 2 \ldots).$$

The l is a variable used to avoid the overlapping of the first vertical and the horizontal I/O ports. For the technology supported at the present time, l is equal to 8 microns (minimum poly to poly spacing 5 microns plus 3 microns contact cut extension).

(4) There are two layers available for external connections of leaf cells, namely, metal and polysilicon. In general, metal wires go horizontally and polysilicon run vertically. The input ports should appear on the top or left sides and the output ports are on the bottom or right sides. All

the I/O ports must sit on grid lines.

(5) Same signal may have more than one I/O ports. These ports should be internally connected, so that an external connection can be made to any of them.

(6) The origin of the I/O ports is defined as the left edge for vertical wires and the bottom edge for horizontal wires.

(7) The cell bounding and supply/clock wiring are defined as shown in Fig. 5.2.2. The relative positions at which the power and clock lines enter the cell are fixed. The line width of power and clock rails are 10 and 5 microns respectively through out the whole cell. No under cut could be made on them except for internal supplies of the leaf cell.

Fig. 5.2.3 shows a layout of a standard leaf cell.

5.2.3 The Library Description files

All leaf cells (except combinational logic ones) are laid out manually and represented in KIC format.. A file called "leafdescr" gives block description of the physical characteristics of each cell and is defined as the follows:

```
<leaf cell name> <height> <width> <# of I/O ports>
 <port 1> <signal name> <side> <offset from origin>
 <port 2> <signal name> <side> <offset from origin>

   .        .       .          .
   .        .       .          .
   .        .       .          .

 <port n> <signal name> <side> <offset form origin>

   .        .       .          .
   .        .       .          .
   .        .       .          .
   .        .       .          .
```

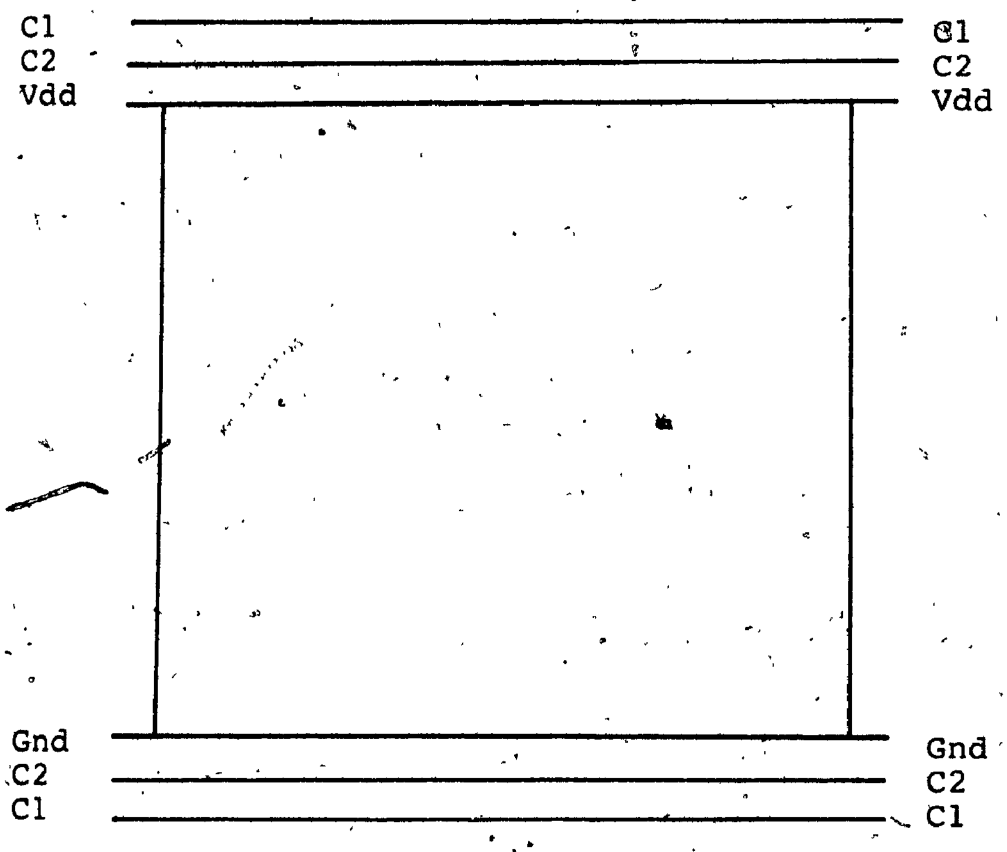


Fig.5.2.2 Cell Bounding and Supply/Clock Wires.

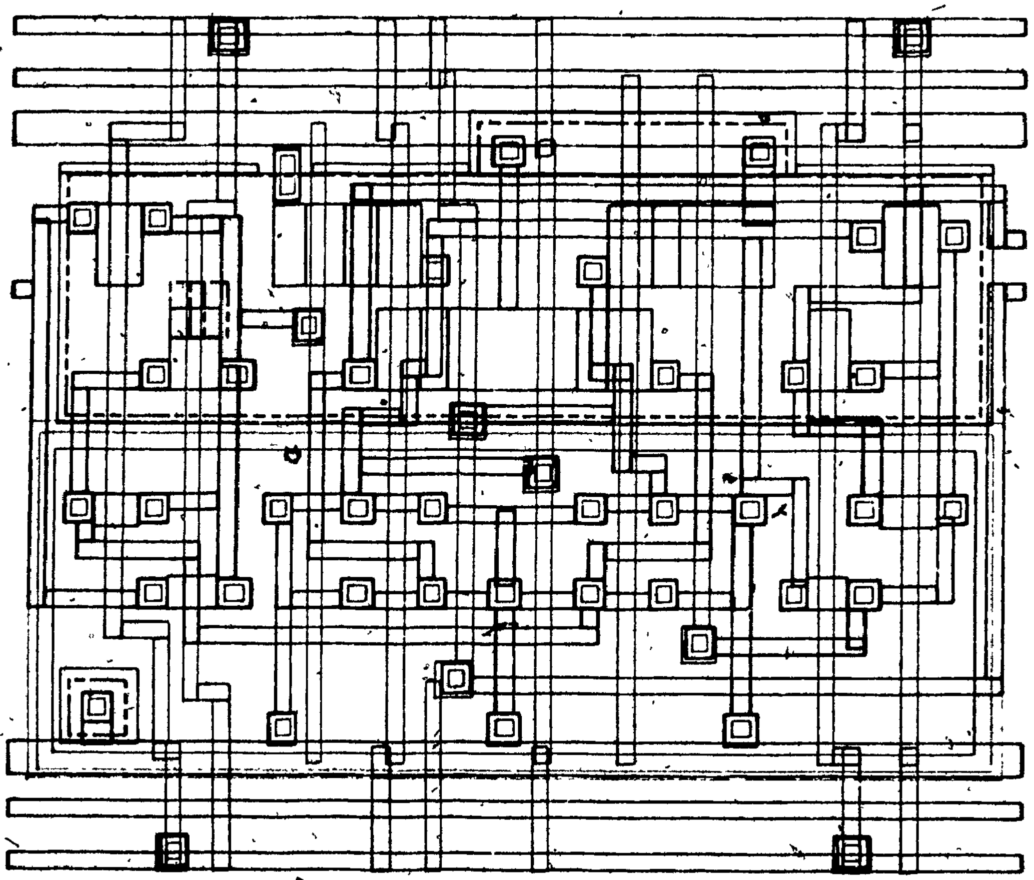


Fig. 5.2.3 An Example Leaf Cell.

The cell name and signal name should not have more than 10 characters. Unique port numbers are used to identify all the ports. Different port numbers may have same signal name but different offsets which indicate the relative physical location of each port. The numbers 0, 1, 2 and 3 are defined to represent the left, right, bottom and top side of a cell respectively.

A leaf cell index file, called "leafindex", is set up for quick retrieval of the required leaf cell from the file "leafdescr". Leafindex is searched by cell name which are stored in sorted order. A leaf cell name in leafindex is followed by the starting address of its data and the number of bytes occupied in file "leafdescr". By knowing these, the file leafdescr can be directly accessed.

5.2.4 Technology Dependency

The main problem of the standard library approach is the technology dependency. When technology changes, the cell library has to be redesigned manually. Finite State Machines (FSM) and boolean expression are complex and random in nature. VLSI designers spend a lot of time in designing them. Quite a number of leaf cells are needed if a full standard cell library strategy is chosen. In order to minimized the amount of work to be involved in the case of a technology change, we use a semi-standard library approach -- using a programmable random logic implementation for FSM and combinational logic modules and a standard library implementation for data path, which reduces the complexity of the library. As a consequence, technology dependency of the leaf cells is reduced to a minimum.

5.3 The Primitive Operators

5.3.1 Selection Of Primitive Operators

The main objective of the primitive library design is to select a minimum set of operators which could be used to construct the systolic arrays. To achieve this goal, a study of existing systolic algorithms was pursued. In many cases arithmetic operations have to be performed simultaneously, such as those expressed by the mathematical symbols "Sigma" for addition and "Pi" for multiplication. These are compute-bound computations that can benefit from the systolic approach. Existing systolic designs may be classified into following categories:

(1) Signa[illegible] Image Processing

. FIR, [illegible] filtering and 1-D convolution

. 2-D convolution and correlation

. discrete fourier transform

. interpolation

. 1-D and 2-D median filtering

. geometric warping

(2) Matrix Arithmetic

. matrix-vector multiplication

. matrix-matrix multiplication

. matrix triangularization

. QR decomposition

. solution of triangular linear systems

(3) Non-numeric Applications

. data structure: stack and queue, searching, priority queue and sorting

. graph algorithms: transitive closure, minimum spanning trees and connected components

. language recognition: string matching and regular

. expression

. dynamic programming

. encoders

. relational data-base operations.

Primitive operators selected for the SASC should be able to cover the above applications. Arithmetic and storage operators are intensively used in many of the systolic systems. Hence, adder, subtractor and multiplier are selected as the basic arithmetic operators. Two types of storage devices are needed: buffering and buffering with initial set and reset capabilities. A full programmable realization of combinational logic modules is chosen based on the simplicity of processing elements of systolic arrays. We will give detailed discussion about the design and implementation of the "logic compiler" in section 5.4. The logic compiler greatly simplifies the construction of two levels of libraries and reduces the amount of work to design leaf cells manually. Bit comparator and switch box are the basic cells for sorting, string matching and network switching problems. The use of CMOS technology brings in the necessity of adding a primitive operator, Clockgen, to generate complementary clocking signals. Drivers are definitely required if the speed of a system is crucial. In addition, a group of primitives, (for example, I/O pads and frames), are selected according to the requirements of generating final chip layout. Fig. 5.3.1 shows a list of primitives selected.

5.3.2 Primitive Composition

Composition of primitive operators is carried out by "composition routines". A composition routine is a program module capable of grouping low level objects (leaf cells) into a primitive floor plan and generating the layout of a parameterized primitive operator. The CIF (CalTech Intermediate Format), an

Arithmetic	Storage	Control	Boolean Expression	Driving	Chip Generation
Adder	Dlatch	Switchbox	Logic Compiler	Driver	Inpad
Substractor	Drs	Multiplexer		Ndriver	Outpad
Multiplier	Shiftregd	Demultiplexer		Clockgen	Vddpad
Add-Multiplier	Shiftregdrs	Comparator			Gndpad
					Frames

Fig.5.3.1 List Of Primitives.

"embedded" language, is used to enable objects to be defined and instantiated hierarchically.

In the low level translation process, a set of intermediate code describing a high level design are analyzed and a group of parameters are kept in a text file named callprl which is the input data file of the composition routines. It contains the following information:

<primitive name> <# of bits> <word length>

. . .

. . .

The name of the primitive is unique. It is a concatenation of the name of the calling primitive in the primitive library and the name of the cell in the design. For example, in Adderc2-p5, Adderc2 is the library operator used. p5 is the name of adder in the system and assigned by the high level compilation process. The second and the third parameters are digits. Furthermore, a one-to-one mapping from a primitive name to the corresponding layout geometry is conducted. Fig. 5.3.2 shows the composition routine environment. The outputs of a composition routine are:

(1). The placement file named *.pla specifies the initial cell placement. Its format format is:

<hr>

<total # of cells>

<cell name> <cell #> <x, y coordinates of the origin>

where <hr> is an important parameter which guides the routing algorithm used and we will discuss it in section 5.5. The <cell #> should correspond to the order in which the cell instance was invoked. The y coordinate of cells are quantized to integer multiples of hr.

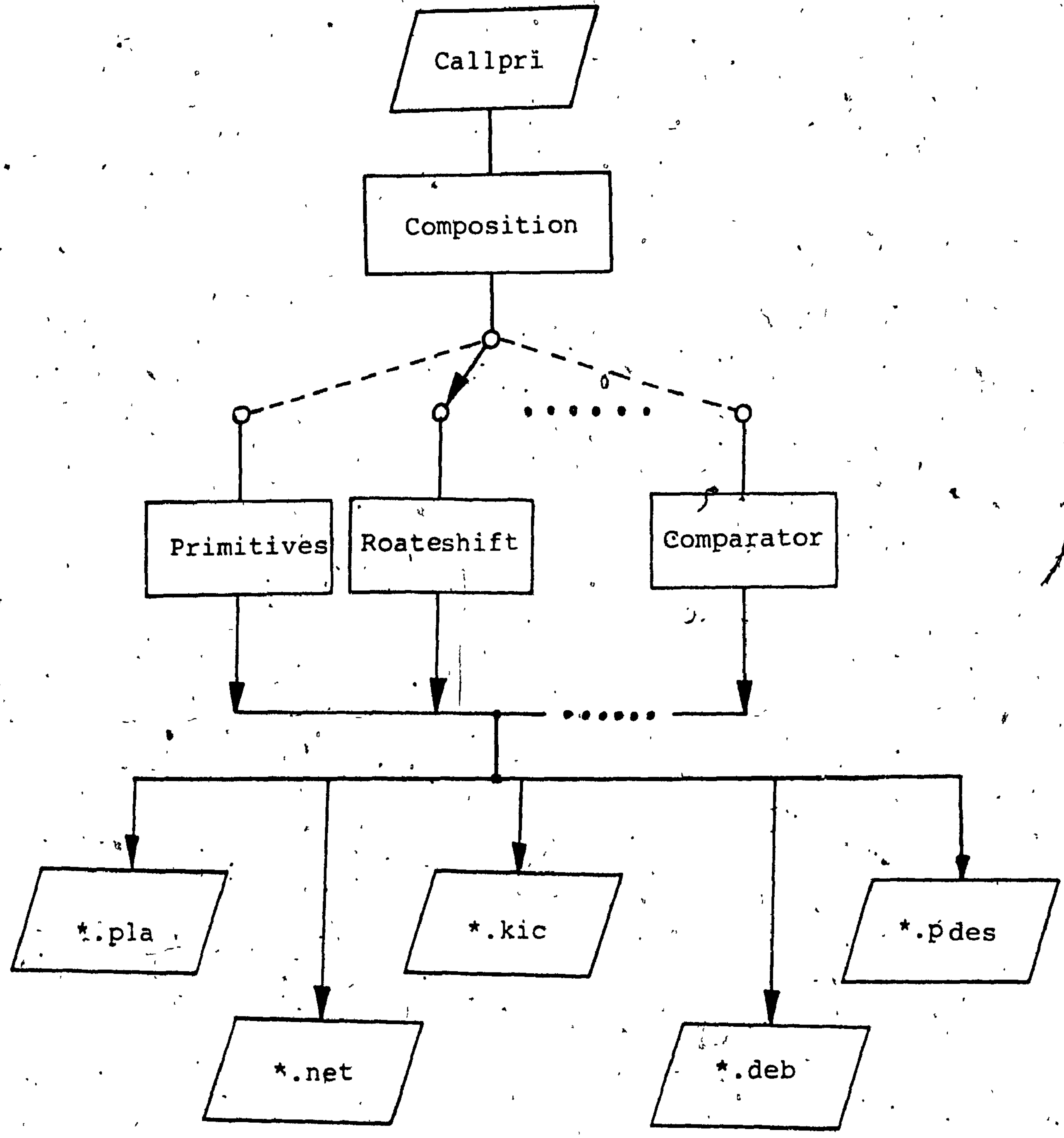


Fig.5.3.2 Block Diagram Of Composition Routine.

(2) *.net is a netlist file which gives explicit description of the internal wire connections and external ports of a primitive. The head of a *.net file is defined as below:

<# of exter-port>

<side> <# of exter-ports> <exter-port #, signal name>, ...

```
.      .      .      .
.      .      .      .
.      .      .      .
```

There are some constraints brought by the router: (a) The external ports on each side should be specified as a group and listed in a sorted order according to their physical locations starting from the origin of the block. (b) The groups of signals should appear in the following sequence: left, right, bottom and top. If there is no port on a side, the corresponding specification should be defined as

<side> 0

to declare the special case. The body of a net list is:

c <cell#, port#> t <cell#, port#> ...t <e, side, port#, order>

[Note]: c	----	connect
t	----	to
e	----	external
side	----	0 : left
		1 : right
		2 : bottom
		3 : top

(3) *.pdes is a block description file for primitive operators which has the same definition as calling leaf cells except for their names. For example, consider the primitive "Driver8" and the leaf cell "driver8". In this case,

initial placement and routing is not required. The file *.pdes functionally corresponds to a data file, named modl.dat, generated by the router for other primitives. We will introduce file modl.dat in section 5.5.

(4) A KIC file named *.kic shows the primitive floor plan generated.

(5) A text file called *.deb which gives information as the files leafdescr, leafindex and pridescr are accessed. The files *,kic and *.deb are used for debugging purposes only.

5.3.3 Primitive Description File

A primitive description file, called "pridescr", contains information about all the primitive operators. It is defined as follows:

```
<primitive name> <# of leaf cells used>
 <leaf cell 1>
 <leaf cell 2>
       .
       .
       .
 <leaf cell n>

<primitive name> <# of leaf cells used>
       .
       .
       .
```

Leaf cells under each primitive name are those which are predefined to construct the primitive operator. The sequence of leaf cell names listed in each primitive description segment implies the sequence of initial placement to be done in the corresponding composition routine.

5.3.4 Expandability Of The Libraries

The primitive and leaf cell libraries are extensible. We leave both of them open and allow new members to be added. To extend the leaf cell library, complete information about the cell being added should be given at the end of the leaf cell description file, leafdescr. Then a system routine, LEAFINDEX.GEN, is called to automatically find out the position of the new cell in file leafdescr, modify the corresponding index file and sort the file by cell name.

The primitive library actually is a set of composition routines as we mentioned in the previous section. To extend the primitive library is to design the primitive operators to be added, write corresponding composition programs, append to the file "composition" and finally add the required information into the files "pridescr" and "*.net".

Each leaf cell or primitive operator in the system has a unique name. Leaf cells and primitive operators are distinguished by the first character of their name. The first character of a primitive cell name is a capital letter and a name of leaf cell is all in lower case. Some primitive cells, for example, Driver, may have exactly the same definition with the corresponding leaf cell, driver, since the leaf cell 'driver' might be used to construct some primitive operators and the primitive 'Driver' may be called to build a function block of a target system. Some leaf cells have more than one version of layout geometry. for example, a bit-delay (dlatch) may work at clock phase 1 or 2 depending on the high level description. In each case, the corresponding layout is fixed. We append a postfix, c1 or c2, to the name of the leaf cell in order to distinguish the two. Each version of the layout has its own leaf cell description.

5.4 The Logic Compiler

The logic compiler is a layout program which deals with intermediate code of boolean type and generates corresponding layout geometry. It services both the primitive and the leaf cell libraries. A combinational logic module may be a leaf cell used in constructing a certain primitive operator or a primitive operator within a function block.

5.4.1 Assumptions

In order to simplify our problem, a few system assumptions are established. They are:

(1) The boolean equations provided by the user are optimized. This assumption could be removed later, if an optimizer becomes available.

(2) The size of the boolean equations could be:

(a) any number of products, but not more than 9 variables, or

(b) any number of variables but not more than 9 products.

The value of 9 is determined by the leaf cell conventions of section 5.2.2.

(3) Three types of speed factors can be used. They are fast, slow, and medium. For each layout of a combinational logic module, only one speed factor is supported.

5.4.2 Domino Logic Structure

Domino logic is chosen as the logic structure of implementation. It allows a single clock to precharge and evaluate a cascaded set of dynamic logic blocks. This involves incorporating a static CMOS buffer into each logic gate as shown in Fig. 5.4.1. During the precharge time, the output node of the dynamic gate is

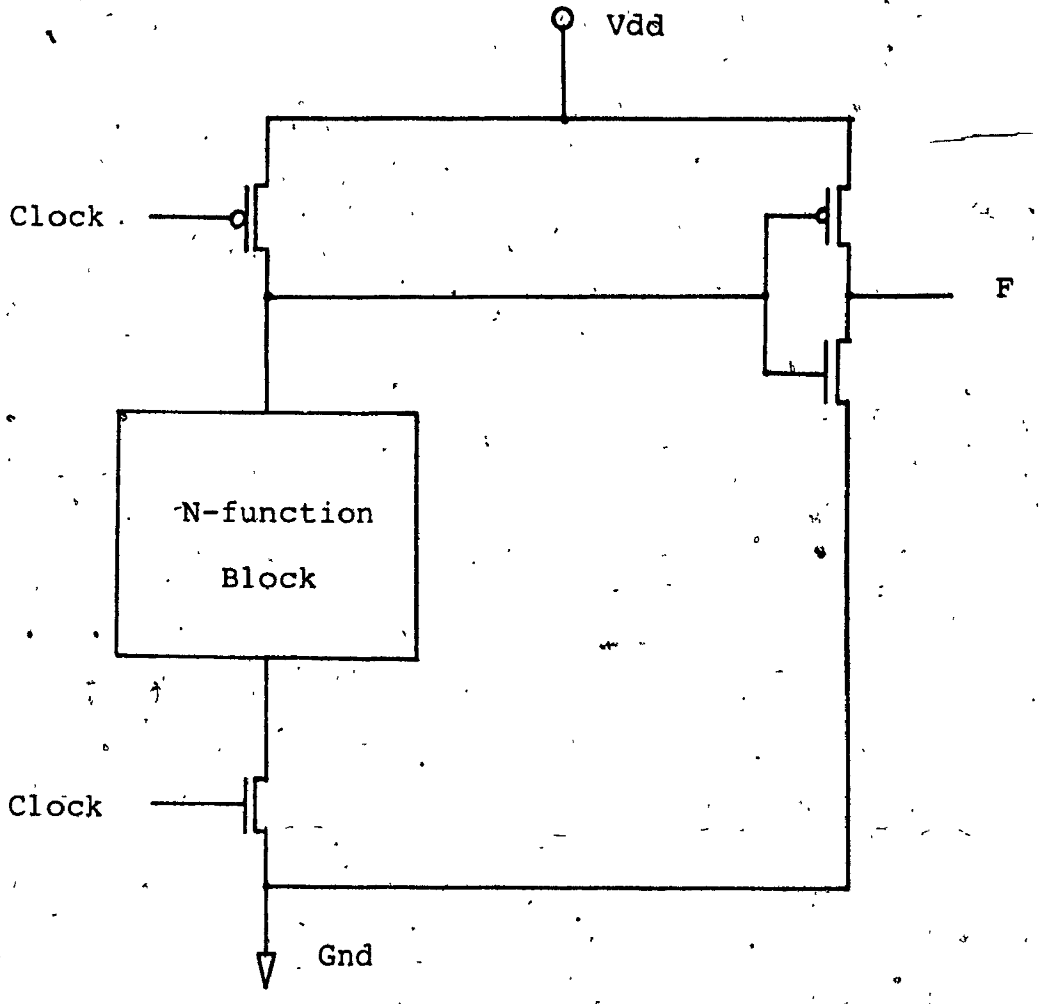


Fig.5.4.1. CMOS Domino Logic Basic Gate.

precharged high and output of the buffer is low. As the subsequent logic stages are fed from this buffer, transistors in subsequent logic blocks will be turned off during the precharge phase. When the gate is evaluated, the output will conditionally discharge, causing the output of the buffer to conditionally go to high. The limitations of the structure are (a) only non-inverting structures are possible; (b) each gate must be buffered. However, these limitations could be compensated. For example, by cascading a inverted logic block to remove restriction (a). Comparing with complementary CMOS logic implementation, domino logic has lower power dissipation. Area cost can be estimated to a first order by the number of transistors. For an M input gate, a complementary CMOS gate uses (2 * M) transistors whereas only (M + 4) transistors are required for domino implementation.

5.4.3 Logic Compilation

The logic compiler is called when boolean type of intermediate code is encountered. Let's look at an example. Assume we have a boolean type of intermediate code as shown below:

$$[\text{F boolean } 7\ 4\ 13\ 1\ a * b + c * d + e * f + g].$$

In the low level translation process, first of all lexical analysis is applied to the intermediate form. If a boolean type of data is detected, a procedure named "boolgen" is invoked. It converts the intermediate form into a two-dimensional table, called the variable/product table. The rows in this table represent the variables and columns represent the products. If a variable appears in a product, "1" should be set in the corresponding row and column. For the above example, Fig. 5.4.2 shows the corresponding variable/product table. Then, the table is mapped into a n-logic block of domino logic implementation. The vertical device wells represent the products and the horizontal poly wires are corresponding to

Pro. / Var.	P1	P2	P3	P4
A	1			
B	1			
C		1		
D		1		
E			1	
F			1	
G				1

$(F = A * \underline{B + C * D} + E * F + G)$

Fig. 5.4.2 An Example Of Variable/Product Table.

the variables. Fig. 5.4.3 shows the layout of the n-logic block for the example.

We can see that the layout in Fig. 5.4.3 is not optimal. If we let metal wires cross the cell and put contact cuts at the left/right sides, and then connect them together, some redundant contact cuts (one with a circle) could be removed. A layout optimization procedure removes redundant contact cuts and the space originally occupied by redundant contact cuts are saved. Finally, a compact layout geometry is generated. Fig. 5.4.4 presents the block diagram of the logic compiler.

5.5 Placement, Routing and CIF File Generation

This section is organized in three parts. In section 5.5.1, we discuss the basic assumptions made for the placement and routing algorithms. Then, a general channel router [90] is introduced. Next, in section 5.5.2, we present a global picture of the routing environment. In section 5.5.3, we show how a composition process is ended by generating mask geometry of a function block.

5.5.1 Placement And Routing

Routing is one of the classical problems related to design automation. Many routing algorithms have been proposed and implemented. Due to the properties of systolic array and the hierarchical layout methodology, the routing problems in the SASC environment is much simpler. The routing scheme used is basically a general channel routing method [91],[92]. The router was designed for use in all the three levels (primitive cell, function block and the final floor plan) of a chip. It can handle cells with variable heights and width. I/O ports are allowed to appear on all the four sides.

Basic assumptions:

We assume that (a) initial placement of routing cells is done before the

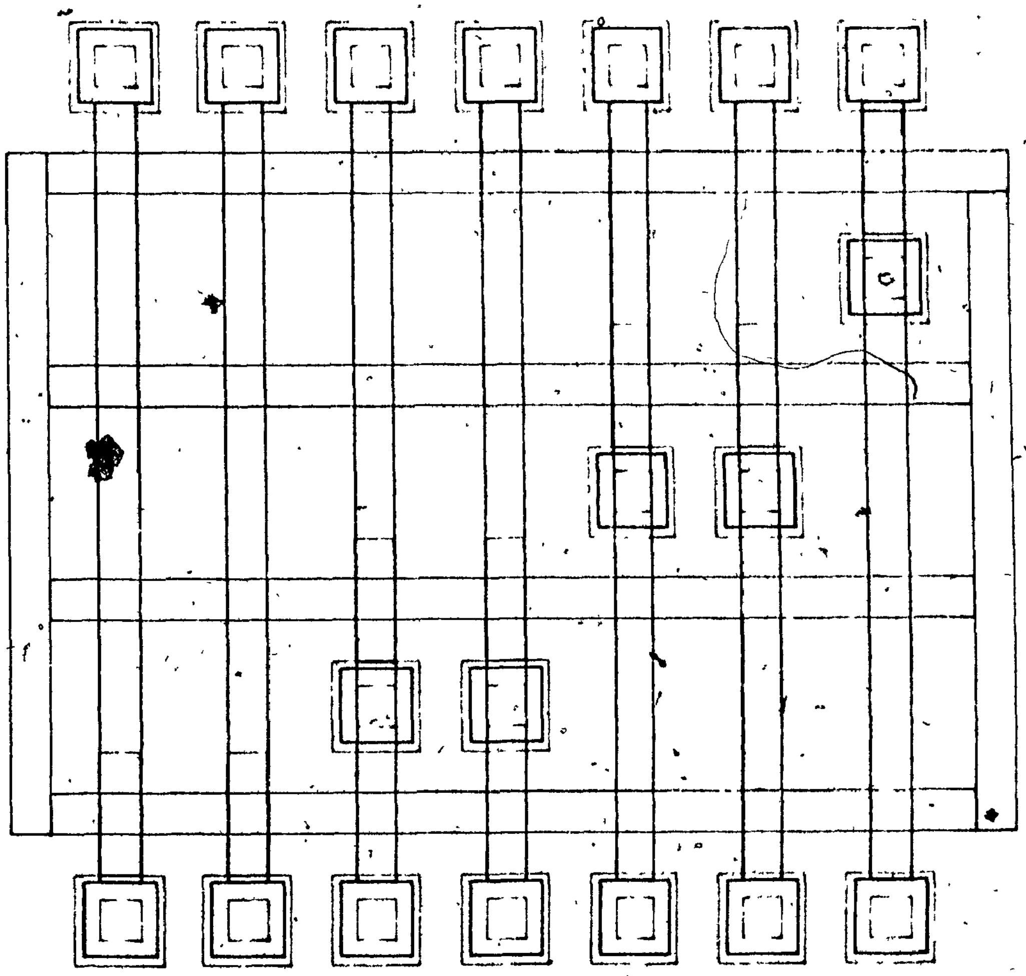


Fig. 5.4.3 An Example Of N-logic Block Layout.

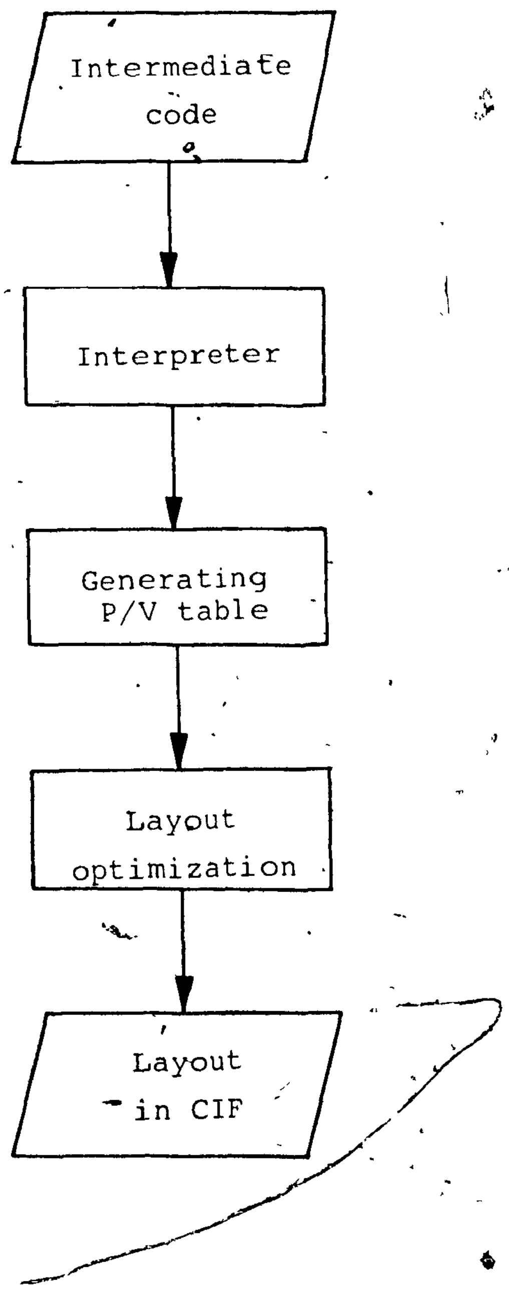


Fig.5.4.4 Block Diagram Of Logic Compiler.

router is called. The composition procedures give the initial placement of primitive operators ~~and a placer is~~ to be used for the two top level placement; (b) there are two separate layers (one metal and one poly) available for making wire interconnections. In a routing channel, wires in the two layers should run orthogonally. The leaf cell conventions we mentioned in section 5.2.2 are the basic rules in layout generation and are also followed by the router.

Routing grid:

Each functional block to be routed may be regarded as a set of black boxes placed on a grid each with the origin in the left-bottom corner. The grid is separated by equal spaces in both x and y coordinates (leaf cell convention 3). Grids are used to define tracks in which wires can run. All I/O ports must sit on the grid lines. Particularly, the left and right hand side I/O ports should be located on horizontal gird lines while the top and bottom side I/O ports should be on vertical grids.

The important integer hr:

For cells having different heights, there exists some value, hr, such that the heights of the cells are roughly equal to some integer multiple of hr. For example, 96 might be chosen ~~as the value of~~ hr for the leaf cells in the library so that

$$\begin{aligned} 1H &= 165 \text{ micron} \simeq 2 * hr \\ 1.5H &= 251 \text{ micron} \simeq 3 * hr \\ 2H &= 341 \text{ micron} \simeq 4 * hr \end{aligned}$$

By using this assumption, cells can be placed in rows of height hr. A cell may occupy 2, 3, or 4 rows according to its height. When a channel between two adjacent cells is to be routed, cells on the right or top will be relocated to make room for the wires and contact cuts. Each channel could be regarded to have infinite number of tracks to house all the wires passing through it.

The routing scheme:

The routing algorithm used could be divided into two parts: path assignment and track assignment. Let's look at an example. Assume it is desired to connect two points A and B. There may be more than one path to connect the two. The objective of path assignment is to find out a path having minimum number of turns and a minimum wire length. We assumed before that all cells are laid down in rows. According to the initial placement information given, the horizontal adjacency relationship between the cells can be derived and described by an "adjacency graph". The location of the cells are adjusted after aligning each cell. Channels are modeled by a channel graph where a node represents the intersection of a horizontal and a vertical channels. An edge represents the channel between intersections. The nodes in the channel graph are labeled and by using an algorit~~hm sim~~ilar to Lee's algorithm, the best path is selected. Finally, a path is represented by introducing dummy ports at the intersection of channels.

The algorithm used to do the track assignment is basically the left-edge algorithm. In this pass, the channel graph is broken down into individual vertical and horizontal channels. These channels are routed one after another. When assigning a wire to a track, we take into account the constraints imposed on it by all other wires of the channel.

5.5.2 The Routing Environment

The router is called in each level of composition. It accepts a placement file (*.pla), a netlist file (*.net), an index file and a description file as inputs for leaf cells or function blocks used. There are four output files produced by the router:

(1) The file mod1.dat contains the description of the block generated. It has the same format as the file "leafindex" defined in section 5.2 and is used as an input when higher level composition is carried out.

(2) The file mod2.dat gives the placement information of each cell inside the block after the wire routing is done. It has the same format as the file "*.pla" defined in section 5.3.

(3) The file wire.dat describes all the wire connections in the block, such as the origin and the relative position of each wire in the channel, the physical attributes of each wire (the name of the layer that a wire is going to run, the length, width and contact cuts), etc. The format of the file is:

```
<# of channel>
channel <channel #> <x,y coordinate>

     .     .     .
     .     .     .
     .     .     .

     * net <net #> *
<layer/contact-cut> <length> <width> <> <>

     .     .     .
     .     .     .
     .     .     .
```

(4) The file "result" is intended for debugging. It records all intermediate information generated when the router is run. Fig. 5.5.1 shows the routing environment.

5.5.3 Generating Layout Geometry

The CIF generator is a program module to produce the CIF file of a routed

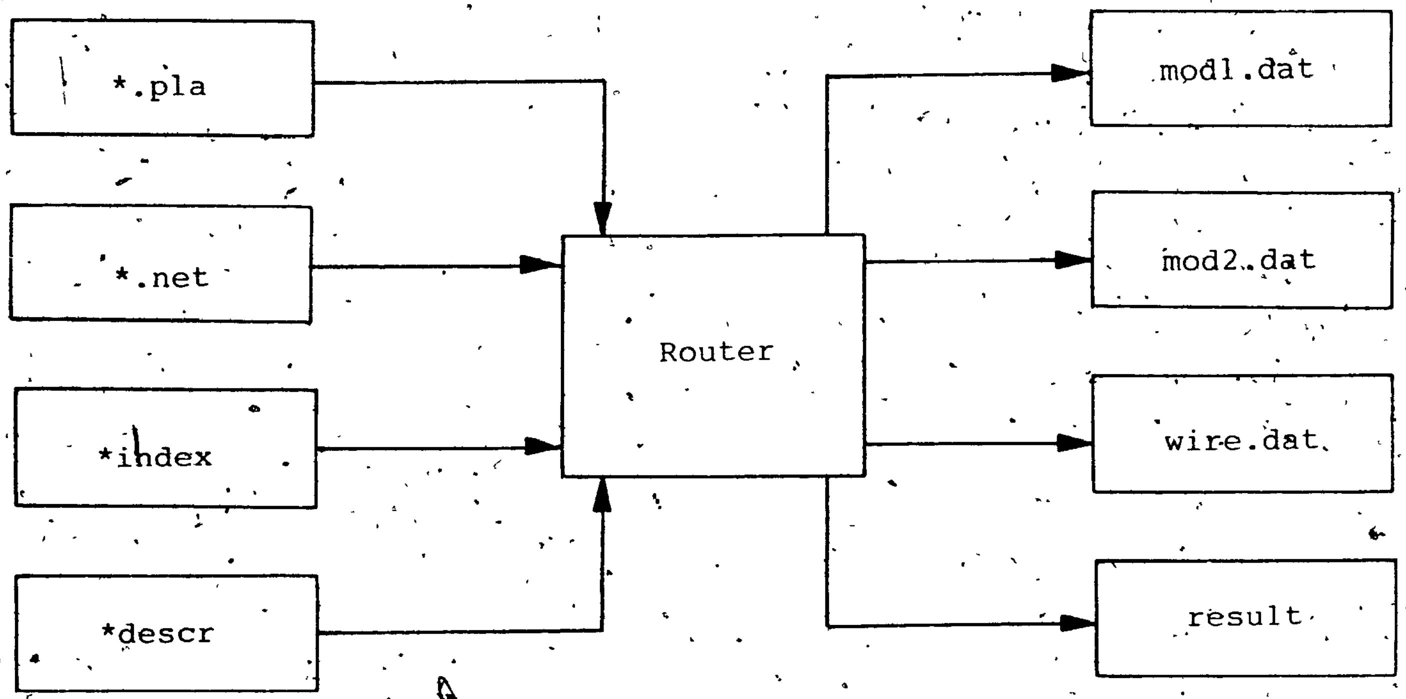


Fig.5.5.1 The Routing Environment.

primitive or function block. Data generated by the router (in file wire.dat) is local -- specifying the relative position of wires in the channel. Transformation from the local information to a global floor plan is conducted. The basic components, lower level cell layout in KIC format, and the wire connections and contact cuts are converted into the CIF format. Fig. 5.5.2 shows the block diagram of the CIF generator.

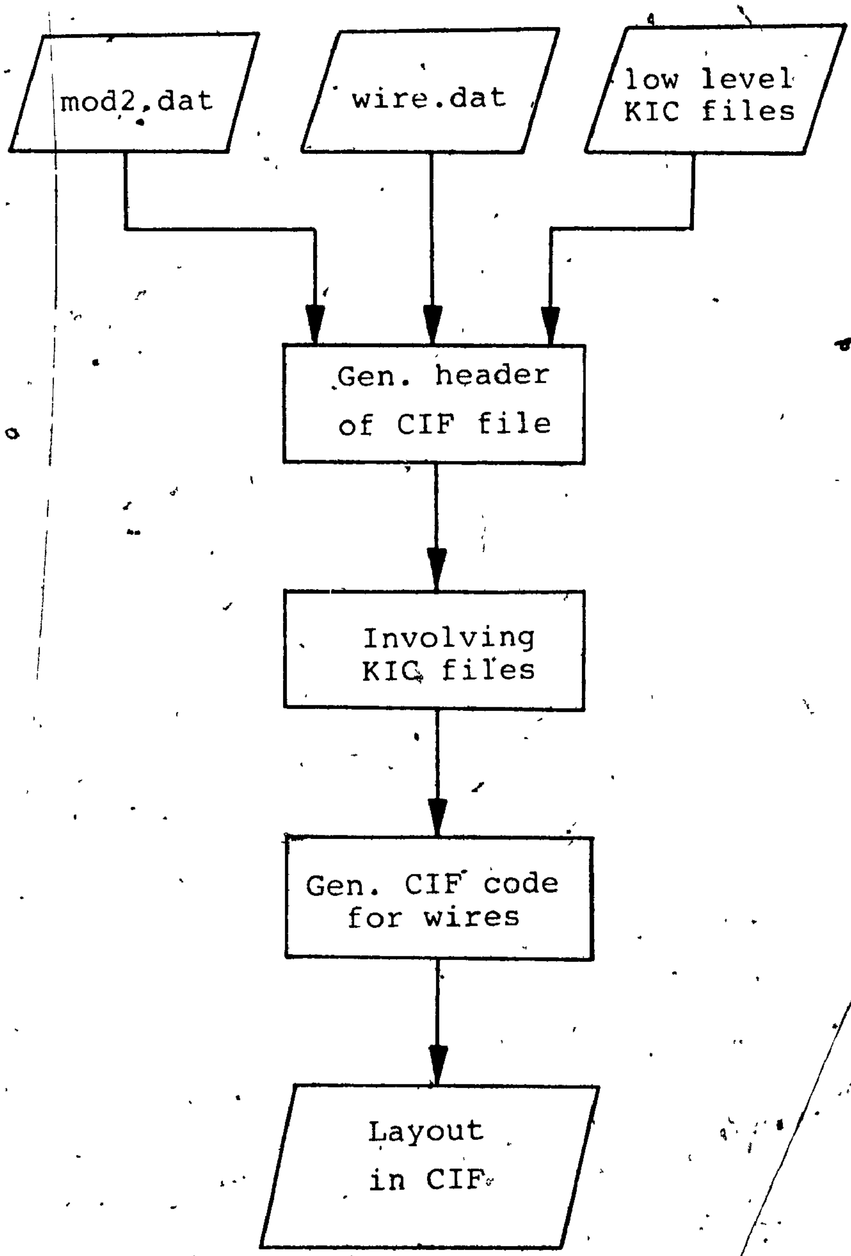


Fig.5.5.2 Block Diagram Of CIF generator.

5.6 Generation Of The Layout From The Intermediate Code

In previous sections we have introduced the main modules involved in the low level translation process and shown how the compilation proceeds in each of them. We now give a brief discussion on the design and implementation of the interpreter first. Then we will show how these modules are integrated and how a layout is generated from the intermediate form.

5.6.1 The Interpreter

The interpreter is a program which works as a lexical analyzer of SASCIL. As we mentioned in chapter 4, there are seven kinds of data types in the intermediate form. They are: signal, port, temporary, register, boolean, library and event. Also, there are two types of statements: assignment statement and library/function call. An assignment statement is a simple statement which may apply to signal, port, temporary, register and boolean data types. A function call is a set of statements which may contain both assignment statement and library/function calls. The main objective of the interpreter is to map logic description of data into corresponding physical ones. For all meaningful combinations of the data types, corresponding translation procedures are designed and implemented. The formats of a set of data files are defined. Fig. 5.6.1 is the list of procedures designed for the interpretation of the assignment statement. There are seven data files to be created during an interpretation process. They are:

(1) File callpri contains all calling primitives in the function block of the current level. It is used as an input data file accessed by the composition routines as we defined in section 5.3.2.

(2) File frecord keeps a record of all data files generated during the

Right / Procedures / Left	Signal	Port	Temp	Register	Boolean
Signal	Sigport_sigport	Sigport_sigport	Store_temp	signal_reg	Signal_bool
Port	Sigport_sigport	Sigport_sigport	Store_temp	Signal_reg	Signal_bool
Temp	Sigport	Sigport	-----	Registers	Temp_bool
Register	Reg_sigport	Reg_sigport	Store_temp	Reg_reg	Reg_bool
Boolean	-----	-----	-----	-----	-----

Fig.5.6.1 List Of Procedures For Assignment Statements.

interpretation. It is designed for the low level management and has the following format:

```
< file-name 1 >
< file-name 2 >



< file-name n >
```

(3) File connectout is one of the prime data files to be used to generate the netlist file of a function block. It consists of the internal connections of cells in the block except the common default signal Vdd, Gnd and Clocks. It is of the following form:

```
c <cell-#> <port-#> t ... t <cell-#> <port-#>
c <cell-#> <port-#> t ... t <+ / ->
c <cell-#> <port-#> t ... t <var-name> signal/port
c <var-name 1> signal/port t <var-name 2> signal/port
```

[Note]: + -- connected to Vdd
- -- connected to Gnd

(4) File priports is another prime data file designed for the generation of the netlist file of function blocks. The information contained in this file describes the statues of the calling cells except the common default ones. The format definition of this file is:

<cell #> <cell name> <port 13> <state> <port 14> <state> ...

. . . . . .

. . . . . .

. . . . . .

[Note]: <state> : ! -- not used;

<state> : + -- connected to Vdd;

<state> : - -- connected to Gnd.

(5) File databook is an intermediate data bookkeeping file and contains variable names, types, primitive/function names and their cell numbers. Consider, for example, the following assignment statement

[yout port r] = [R3 register 7 4 2 n8 n9 ! n10 !]

It implies the following:

(a) The output of register R3 is connected to an output port called "yout" which is located at the right hand side of the function block.

(b) Register R3 is a 4-bit shift register (shifts to the right and works under clock phase 2) and the storage devices used are D-flipflop with set and reset functions.

(c) The input, output and the reset signals of the primitive are named n8, n9 and n10 respectively.

When the interpreter is called, it analyzes the grammar of the statement and invokes a procedure called "signal-reg" to pursue the interpretation. The type of storage device "drs" and corresponding primitive library operator "Shiftdrsc2" is selected. The mapping between the logic description of R3 to the physical description of primitive Shiftregdrsc2 is performed. The I/O ports of R3 are the internal nodes connecting to other primitive cells or signal wires in the cell composition level. The I/O ports of Shiftregdrsc2 are predefined. The first 12 ports are reserved for Vdd, Gnd and system clocks. Ports 13 to 21 are assigned to the signal n8, n9, and n10 of R3 accordingly.

The data generated in ~~file~~ priports is:

Shiftregdrsc2_R3 5 13 n8 14 n9 15 ! 16 n9 17 n9 18 + 19 + 20 n10 21 n10

It says that the primitive Shiftregdrsc2 is to be called to generate cell R3. The cell number of R3 is 5. Port 15 (q- (representing $\overline{q}$)), 18 (set) and 19 (set) are not used. The unused ports 18 and 19 have to be connected to Vdd since the terminal "set" of a D-flip-flop is not allowed to be in uncertain status.

The data generated in file connectout describes the I/O signals to go to outside of the block, In the present example, it should be:

c 5 15 t out port r.

We interpret it as the following: connecting the port 15 of cell 5 to the port "out" which is on the right hand side of the block.

The content of file callpri declares the 4-bit shift register and to be used by the composition routines later on:

5 Shiftregdrs_R3 4.

The netlist file of primitive R3 will be generated when the corresponding composition routine is called.

In conclusion, the interpreter is capable of doing:

(a) Lexical analysis of the intermediate form.

(b) Generation of a complete set of data for the current level:

* netlist
* calling primitives

the higher level:

* cell integration
* logic description of ports

the lower level:

* physical description of ports.

5.6.2 Generation Of Layout From The SASCIL

Generation of a layout from the intermediate form is a complicated automation process. A number of program modules are involved. The central database is accessed and the intermediate data files are generated. This translation process may be viewed in different ways. From the view of data representation, it can be partitioned into three layers: logical representation, physical description of the data, and the layout geometry. Fig. 5.6.2 shows the three levels.

We give the flowchart of Fig. 5.6.3 as an alternative of the profile of the low level compilation. It shows the main functionalities of the program modules to be used. In order to give a clear explanation, we would like to partition it into three parts: the compilation of the primitive operators, function block/final chip floor plan and combinational logic.

(1) *Compiling the primitive operators:* A primitive operator is generated by going through the solid line of the flowchart. Invocation of a primitive operator, for example the primitive "Adder", is available in the SASCIL file. The interpreter first of all accesses the SASCIL file, conducts the lexical analysis of the code and produces a set of intermediate data files (file callpri, priports, etc,

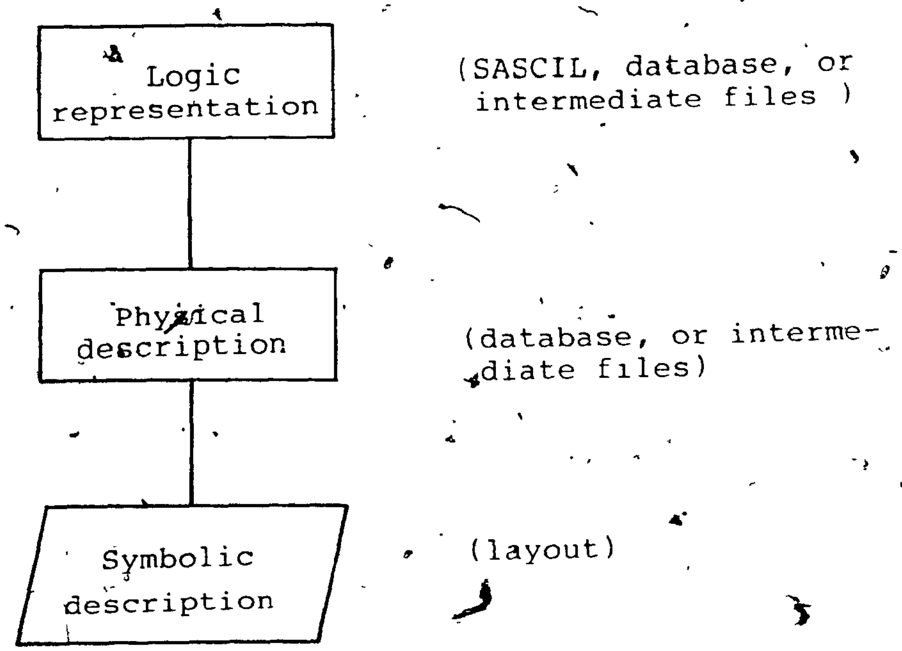

Logic representation
Physical description
Symbolic description
(SASCIL, database, or intermediate files)
(database, or intermediate files)
(layout)

Fig. 5.6.2 The Levels Of Data Representation.

introduced in previous section). Then, the primitive composition module is invoked. By accessing the central database, two leaf cells, "fba" and "drsc2" are used to compose the primitive "Adder". It reads the file "callprl" and invokes the corresponding subroutine to generate the initial placement data of the primitive. Further placement and routing in turn are carried out by the router. The descriptions of the primitive including the replacement of leaf cells, the wiring and the block descriptions are given in files mod1.dat, wire.dat and the mod2.dat, respectively. The module "cifgen" converts these files into the CIF format. We assume that the primitive Adder is going to be used as a cell in a function block. In other words, a KIC format of the primitive is required for higher level composition (a layout only at the topmost level should be presented in the CIF form). So, the program "ciftokic" is invoked, the primitive Adder in KIC format is generated. Now, we come to the end of the primitive composition.

(2) *Compiling function blocks/chips:* As we mentioned in section 5.6.1, the composition of function blocks/chips is from bottom to top. SASC goes to the lowest level of the design hierarchy where there is no user defined functions to be involved. By going to the left branch of the dashed line (see Fig. 5.6.3), and switching to the solid line several times, the translation process for all primitive operators at this level can be finished. The corresponding KIC and data description files are generated. They will be used by the higher level composition. Then, SASC goes up one level of the hierarchy, travels along the right branch of the dashed line, and switches to solid line again. The process is repeated until the topmost level layout is generated.

(3) *Compiling combinational modules.* The process is same as that we discussed in section 5.4.3 and shown by the dot-dashed line of the flowchart.

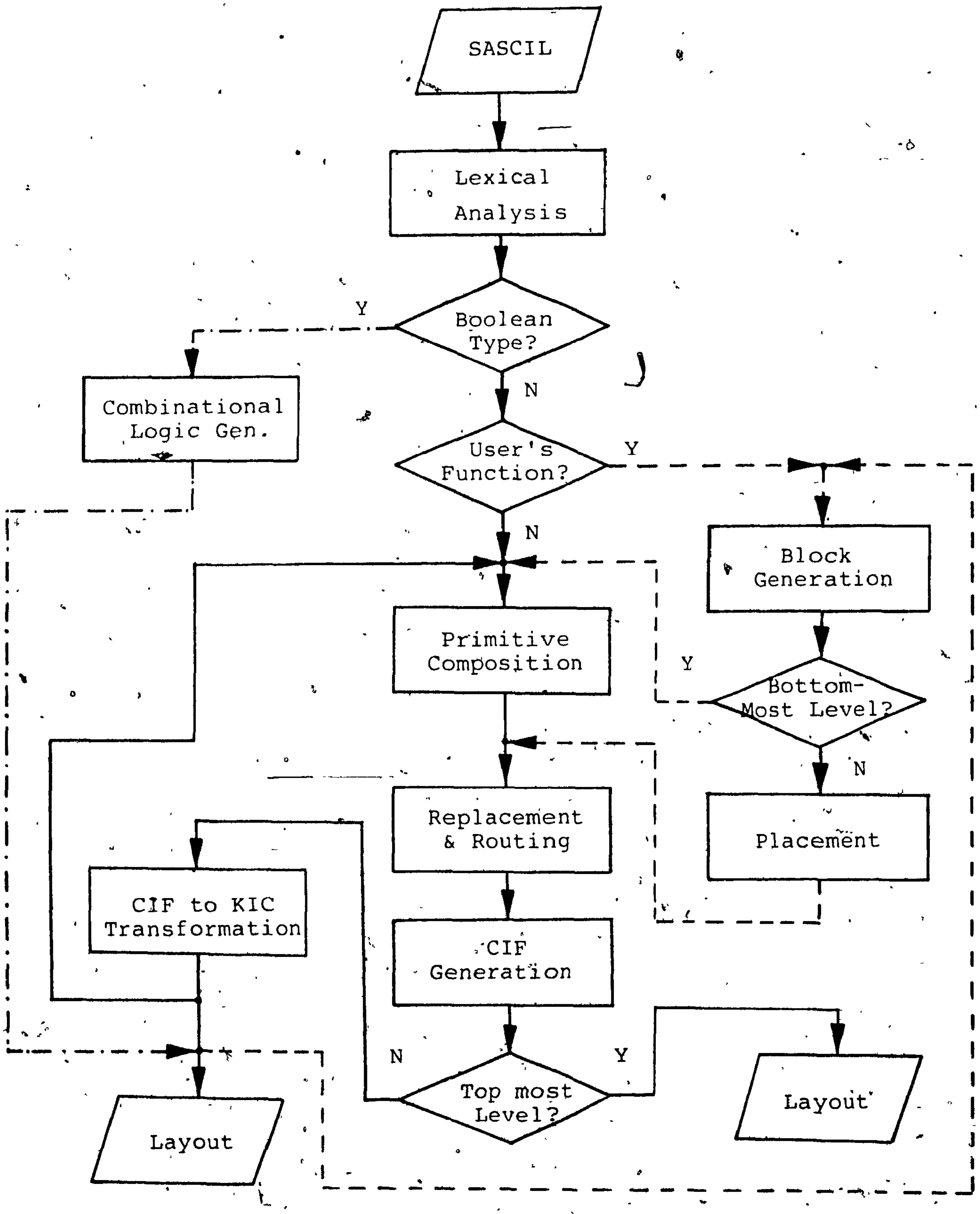


Fig.5.6.3 Block Diagram Of Low Level Compilation Process.

5.7 Generating Layout For the Example Algorithm

In this section, we demonstrate the low level translation process with an example. The intermediate form of the function pe in the FIR algorithm (given in chapters 3) is shown in the end of chapter 4.

The interpreter generates the following data files:

(1) File callpri contains eleven primitive calls:

```
1 Dlatchc2_p3 1
2 Rotateshiftdc1_p1 4
3 Shiftregdc2_p8 4
4 Shiftregdc2_p7 4
5 Shiftregdc2_p9 5
6 Dlatchc1_p4 1
7 Shiftregdc1_p2 2
8 Adderc2_p5 1
9 Multiplierc2_p6 1
10 Clockgen30c1_p10 1
11 Clockgen30c2_p11 1
```

(2) File priports:

```
1 Dlatchc2_p3 13 n1 14 s6 15 !
2 Rotateshiftdc1_p1 13 s1 14 n1 15 !
3 Shiftregdc2_p8 13 n6 14 s5 15 !
4 Shiftregdc2_p7 13 n5 14 n6 15 !
5 Shiftregdc2_p9 13 n3 14 s4 15 !
6 Dlatchc1_p4 13 s3 14 n3 15 !
8 Adderc2_p5 13 n2 14 n4 15 ! 16 n5 17 + 18 + 19 n2 20 n4 21 ! 22 + 23 +
9 Multiplierc2 13 n1 14 n3 15 ! 16 + 17 + 18 n1
                                        19 n3 20 ! 21 + 22 + 23 n4 24 !
10 Clockgen30c1_p10 13 IN1 14 C1 15 C1-
11 Clockgen30c2_p11 13 IN2 14 C2 15 C2-
```

(3) File connectout:

```
c  1 14 t s6 signal
c  2 14 t  1 13
c  2 13 t s1 signal
c  3 14 t s5 signal
c  4 14 t  3 13
c  5 14 t s4 signal
c  6 14 t  5 13
c  7 13 t s2 signal
c  7 13 t s3 signal
c 10 13 t port
c 10 14 t port
c 10 15 t port
c 11 13 t port
c 11 14 t port
c 11 15 t port
```

(4) File frecord:

```
data/intermed/callprl
data/intermed/connectout
data/intermed/prlports
data/intermed/databook
data/intermed/pe_0.pdes
```

data/intermed/frecord

(5) File databook:

```
p3 register Dlatchc2_p3 s6   1
p1 register Rotateshiftdc1_p1 n1  2
s5 signal
p8 register Shiftregdc2_p8 s5   3
s4 signal
p9 register Shiftregdc2_p9 s4   5
p2 register Shiftregdc1_p2 n2   7
p4 register Dlatchc1_p4 n3   7
p5 lib Adderc2 8 0 1 1 2 11
p6 lib Multiplier 9 0 1 1 2 12
p10 lib Clockgen30c1 10 0 1 1 1 3
p11 lib Clockgen30c2 11 0 1 1 2 3
```

(6) File pe.predescr:

```
p1  910    181   15
1  c1        0  176
2  c1        1  176
3  c2-       0  160
4  c2-       1  160
5  vdd        0  144
6  vdd        1  144
7  gnd        0   48
8  gnd        1   48
9  c2        0   32
10 c2         1   32
```

```
11  c1-      0   16

12  c1-      1   16

13  d        0   64

14  q        1  112

15  q-       1  128

p2  430    165   15

1   c1       0  160

2   c1       1  160

3   c2-      0  144

.   .        .   .

.   .        .   .

.   .        .   .

p11 320    341   15

1   c1       0  336

2   c1       1  336

.   .        .   .

.   .        .   .

.   .        .   .
```

The composition module accesses the file callpri and automatically generates the initial placement files for all the primitives except the multiplier (which is not available in the library at the present time. The "multiplier" used is designed only for the demonstration. It consists of a full binary adder and a D-flipflop). Here are the *.pla files.

p2.pla:

```
        p2 2

        96

        2

        dlatchc1 1 0 0

        dlatchc1 2 208 0

p5.pla:

        p5 2

        96

        2

        fba1 1 0 0

        drsc2 2 144 0
```

The *.net files automatically generated by the composition routines are for primitives p1, p4, p7, p8 and p9. Routing is not required for p2 and p3 since they contain one leaf cell only. File p4.net, p10.net and p11.net are given manually. They are part of the primitive library. The following shows the .net file for p5 (Adderc2).

```
        l 6 11 c1-

        l 6 9 c2

        l 6 7 gnd

        l 6 5 vdd

        l 6 3 c2-

        l 6 1 c1

        r 6 12 c1-

        r 6 10 c2

        r 6 8 gnd

        r 6 6 vdd
```

```
r 6 4 c2-
r 6 2 c1
b 6 13 a
b 6 14 b
b 6 15 cout
b 6 16 sum
b 6 17 set
b 6 18 reset
t 5 19 a
t 5 20 b
t 5 21 cout
t 5 22 set
t 5 23 reset

c 1 20 t 2 13 t e t 21 3 n
c 1 17 t 2 17 n

c 1 14 t e t 19 1 n
c 1 16 t e t 20 2 n
c 2 18 t e t 22 4 n
c 2 20 t e t 23 5 n

c 1 13 t e b 13 1 n
c 1 15 t e b 14 2 n
c 1 19 t e b 15 3 n
c 1 21 t e b 16 4 n
c 2 19 t e b 17 5 n
c 2 21 t e b 18 6 n
```

```
c 1 2 t 2 1 n
c 1 4 t 2 3 n
c 1 6 t 2 5 n
c 1 8 t 2 7 n
c 1 10 t 2 9 n
c 1 12 t 2 11 n

c 1 1 t e l 1 6 n
c 1 3 t e l 3 5 n
c 1 5 t e l 5 4 n
c 1 7 t e l 7 3 n
c 1 9 t e l 9 2 n
c 1 11 t e l 11 1 n

c 2 2 t e r 2 6 n
c 2 4 t e r 4 5 n
c 2 6 t e r 6 4 n
c 2 8 t e r 8 3 n
c 2 10 t e r 10 2 n
c 2 12 t e r 12 1 n
```

The data for p2 and p3 in file *.pdes is produced automatically when the composition routine is called. File p2.pdes is a part of the cell description file and the source file for higher level composition and routing.

```
p2  430   165  15
1   c1      0  160
2   c1      1  160
3   c2-     0  144
```

```
4   c2-     1  144
5   vdd     0  128
6   vdd     1  128
7   gnd     0   32
8   gnd     1   32
9   c2      0   16
10  c2      1   16
11  c1-     0    0
12  c1-     1    0
13  d       0   96
14  q       1   96
15  q-      1  112
```

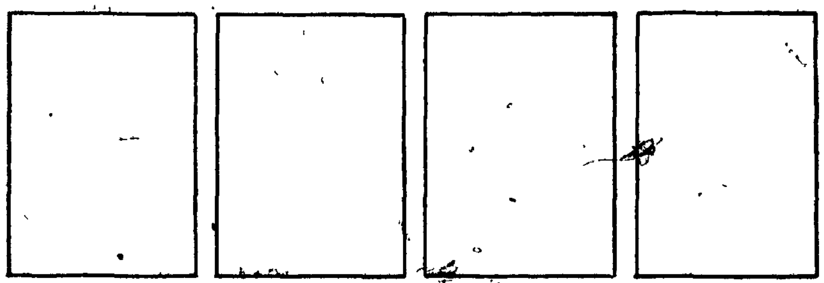
The debugging files, ~~*.lke and~~ *.deb, are shown in Fig. 5.7.1.

Now, let's see an example of replacement and routing for primitive p1. The router takes the above data files and generates the files mod1.dat (replacement), wire.dat (wiring) and mod2.dat (block description) shown below:

File mod1.dat:

```
p1  4    7
dlatchc1      16    16
dlatchc1    1136    16
dlatchc1    2256    16
dlatchc1    3376    16

channel  1     0   16  length   160  width    1
channel  2  1136   16  length   160  width    0
```



(a) pl.kic.

```
Primitive name: Shiftregdcl_pl  wordlength 4
*** Looking for primitive = Shiftregdcl  4 ***
----------------------------------------------

    dlatchcl  206  165  15

----------------------------------------------

    dlatchcl  206  165  15

----------------------------------------------

    dlatchcl  206  165  15

----------------------------------------------

    dlatchcl  206  165  15

----------------------------------------------
```

(b) pl.deb

Fig.5.7.1 Files pl.kic and pl.deb.

```
channel  3  2256   16  length   160  width    0
channel  4  3376   16  length   160  width    0
channel  5  4496   16  length   160  width    1
channel  6     0    0  length  4496  width    1
channel  7     0  192  length  4496  width    0
```

File wire.dat:

```
 **  net  1  **
p     0    0    5   96
c     0   96
m   -18   96    0  101
m     0   96   -2  101
c     0   96

 **  net  2  **
m   -18  112   -2  117

 **  net  3  **
m   -18    0   -2    5



channel  3

 **  net  1  **
m   -18    0    0    5

 **  net  2  **
```

```
m    -18    16    0    21

  ** net   3 **

m    -18    29    0    39




m      0     0 4496    5

c      0     0

c    4496    0

p      0     0    5    16

p    4496    0 4501    16
```

File mod2.dat:

```
p1  4494    181   15

 1  c1        0   176

 2  c1        1   176

 3  c2-       0   160

 4  c2-       1   160

 5  vdd       0   144

 6  vdd       1   144

 7  gnd       0    48

 8  gnd       1    48

 9  c2        0    32

10  c2        1    32
```

11	cl-	0	16
12	cl-	1	16
13	d	0	64
14	q	1	112
15	q-	1	128

The first two files associated with KIC files used in the leaf cell library are called by the module "cifgen" and the corresponding CIF file, p1.cif, is generated in the following format.

p1.cif:

```
(CIF file of symbol hierarchy rooted at p1);
DS 2 1 1;
9 dlatchcl;
L CNG;
B 19600 5600 10300 4500;
L CPG;
B 19000 5000 10300 4500;
L CPW;
B 18000 4000 10300 4500;
L CNP;
B 2300 1500 8050 3150;

.   .   .   .   .
.   .   .   .   .
.   .   .   .   .

DF;
```

```
DS 3 1 1;
9 dlatchcl;
. . . . .
. . . . .
. . . . .
DF;
. . . . .
. . . . .
. . . . .
C 1;
E
```

A KIC representation of primitive p1 is required since it is an operator in the function "pe". The file p1 is generated after the "ciftokic" transformation. Fig. 5.7.2 shows the layout of p1 (rotated shift register). By repeating the above process 11 times, all the 11 primitives are created and ready to be used.

Before the function block composition can proceed further, the following data files should be properly prepared:

(1) The data description file which is a collection of file mod2.dat given by the router and the file *.pdes produced by the composition routine. Each time mod2.dat or *.pdes file is generated, append them to the end of the existing file.

(2) The index file is generated by calling a program LEAFINDEX.GEN. The LEAFINDEX.GEN accesses the data description file, gives the physical position of each cell in the data description file and put them in sorted order in the index file.

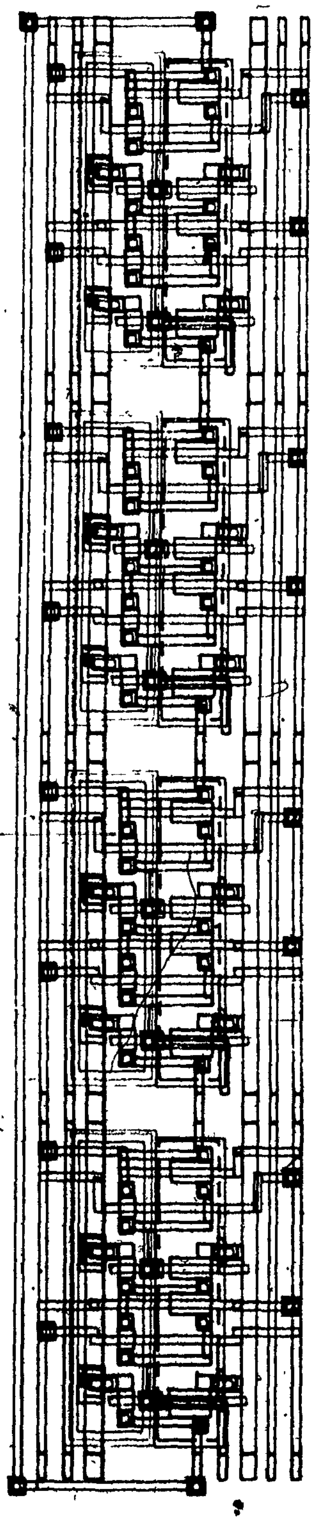


Fig. 5.7.2 Layout Of Primitive Operator p1.

(3) All primitive operators to be used must be composed and presented in the KIC format.

Now, we are ready to go to the higher level of the layout generation hierarchy -- to generate the function block "pe". At present, there are two black boxes in the function block layout generation: the design and implementation of a fully automatic placer might not be an easy task and the works involved in the block generator, therefore, are very straight forward: (a) keep track of the layout generation hierarchy; (b) access the data files produced in the lower level composition and (c) generate a netlist file for the router. In the given example, we provide the file pe.pla and pe.net manually. Then, the router and the CIF generator is called to generate the layout of function "pe" which is shown in Fig. 5.7.3. Repeating the function block generation process, the final chip floor plan of the given example can be obtained and is demonstrated in Fig. 5.7.4.

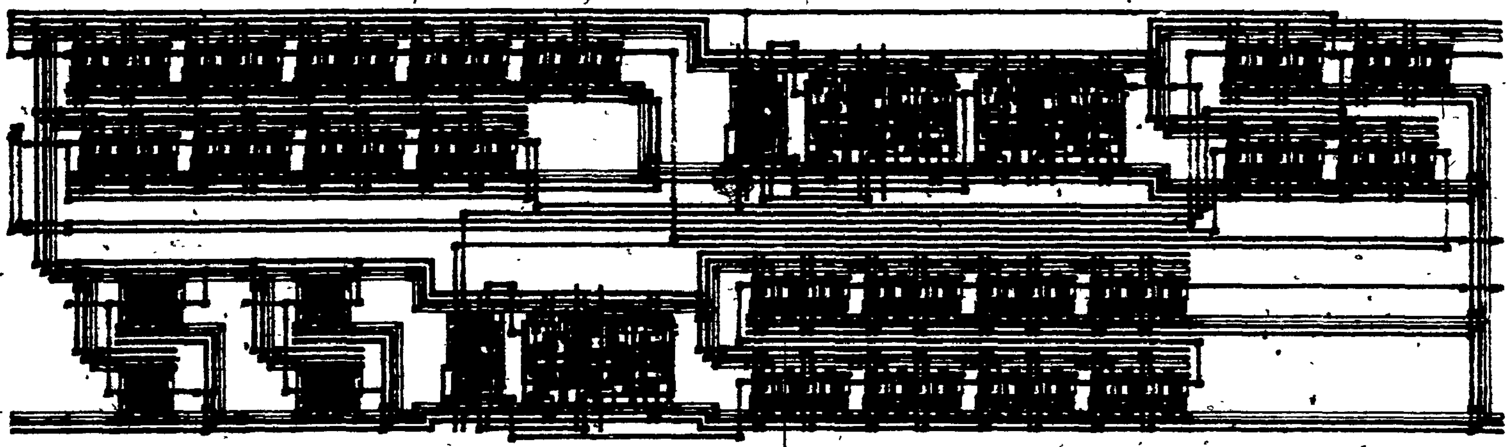
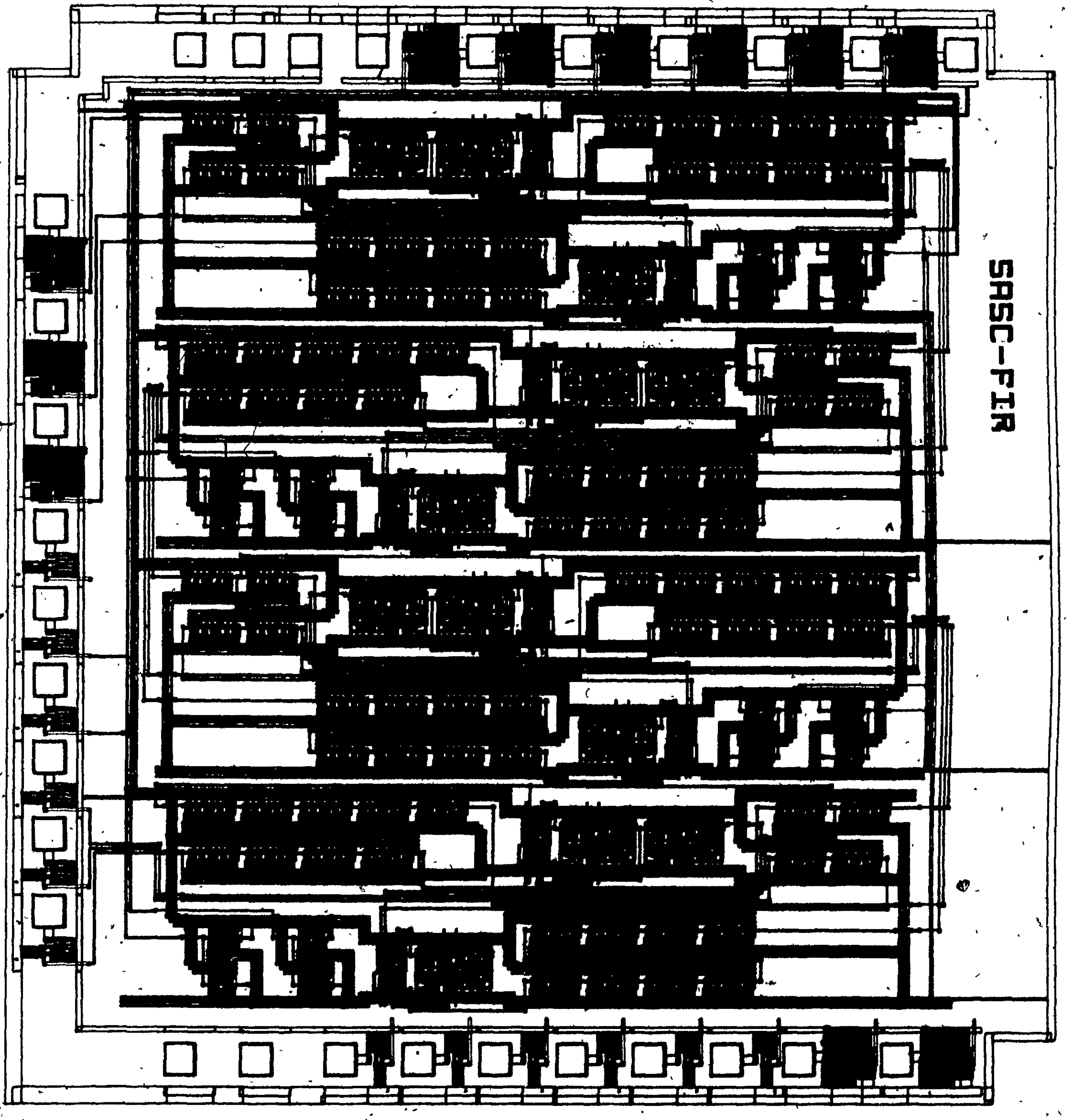


Fig. 5.7.3 Layout Of Function "pe".

Fig. 5.7.4 Layout Of The FIR Systolic Array.



CHAPTER 6

CONCLUSION

The SASC is an automatic design synthesis program specially designed to reduce engineering efforts on investigation and implementation of systolic architectures and algorithms. It accepts systolic systems described in a high level data flow language called SASCL. SASCL is specially designed for SASC and it allows systolic designs to be described at an algorithm level. It produces (1) behavioral simulation results to help validate the algorithms and (2) the corresponding VLSI layout for fabrication in Northern TeleCom CMOS1B (5 micron) process. The target architecture supported by SASC is the bit-serial systolic arrays. However, it accepts other types of algorithms (like bit-parallel, word-serial, and word-parallel) and is capable of transforming them into a bit-serial equivalent ones automatically. In order to improve the system throughput, an automatic two-level pipelined implementation (if it is possible) is carried out without user knowledge and interaction. SASC also keeps all timing synchronization user transparent. In addition, other features of SASC such as automatic chip floor plan generation, use of defaults for power, ground and system clocks, etc. add intelligence and flexibilities to the compiler.

Many of the description languages of existing silicon compilers could be used to describe systolic designs. Unfortunately, some common features of these languages (for example, machine and technology dependency, the generality of data types, data structures and language constructs) make them not ideal for this purpose. On the other hand, the novelties of our compiler, for example, automatic algorithm transformation and separate behavioral simulation and layout generation may require special language constructs. Since none of the existing languages meets all our requirements, a new language, named SASCL, is proposed.

SASCL reflects the properties of systolic architecture at a high level of data abstraction. It provides a simple and hierarchical manner of describing a systolic design. It supports implicit concurrency and implicit/explicit logical synchronization. Being a data flow language, SASCL follows the basic principles of data flow model of computation: Single Assignment Rule (SAR) and allowing a higher order function take another function as argument. User friendliness is one of the main goals of SASCL design, and it is achieved by special on selection of data types, data structures, language constructs and textual form of the program, etc. The responsibility of a user is restricted to the algorithm level and the probability of design errors is reduced.

In the compilation process, algorithms or behavioral descriptions are translated into silicon or simulation results respectively. It is accomplished in two levels of translation which involve three levels of design:

(1) Algorithmic description (SASCL)

(2) Logic description (SASCIL)

(3) Geometry description (CIF).

The high level translation process (from SASCL to SASCIL) may be viewed as an ordered sequence of interpretations. Each step refines the design by adding more details: syntax analysis of a systolic algorithm (a set of instruction in SASCL) is performed and a mapping from non-bit-serial architecture to a bit-serial equivalent is conducted if a non-bit-serial design is given. Then, a decision is made on whether two-level pipelined implementation could be applied to the design. The output of this pass is a set of intermediate code (SASCIL).

SASC is supported by a central database system which contains two levels of library cells (primitives and leaf cells), a technology file (which gives the current fabrication technology parameters) and a set of data description files (for example, the description of physical characteristics of leaf cells). Primitive operators are composed by leaf cells and can be parameterized according to the high level algorithm description. The low level translation process (from SASCIL to layout description) is an ordered sequence of interpretations with some loops because of the hierarchical way of building a VLSI design. It, first of all, translates the intermediate code into primitive invocations and logical connections of wires; Then, the composition program accesses the central database, invokes corresponding composition routines, converts logic description of the design into physical ones. Physical information such as the initial placement and netlist ~~files~~ are passed to the router which performs further placement and routing. A module, called cifgen(erator), in turn, takes the router's output and produces corresponding layout geometry (in CIF format).

For this thesis, the design of SASC, the definitions of SASCL and SASCIL, and SASCIL to layout generation part of SASC are finished. The design and implementation of the leaf cells and primitive libraries are accomplished. The leaf cells in CIF format are ready to be used and a set of composition routines are capable of generating required primitives. The philosophies and translation rules followed by the SASCL to SASCIL translation process are proposed. The low level translator are designed and implemented. A complete example of a fourth order FIR filtering algorithm is used to illustrate the two levels of compilation. The implementation of behavioral simulator is left for future work.

REFERENCES

1. Nassimi, David and Sartaj Sahni, "Bitonic Sort on A Mesh-Connected Parallel Computer," IEEE Tran. on Computer, vol. C-27, no. 1, pp. 2-7, Jan. 1979.

2. Atallah, Mikhail J. and S. Rao Kosaraju, "Graph Problems on A Mesh-Connected Processor Array," Jour. of The Association For Computing Machinery, vol. 31, no. 3, pp. 649-667, July 1984.

3. Nath, Dhurva, S. N. Maheshwari, and P. C. P. Bhatt, "Efficient VLSI Networks For Parallel Processing Based on Orthogonal Trees," IEEE Tran. on Computer, vol. 32, no. 6, pp. 569-585, June 1983.

4. Zakharov, Vasilii, "Parallelism And Array Processing," IEEE Tran. on Computers, vol. C-33, no. 1, pp. 45-78, Jan. 1984.

5. Siegel, Howard Jay, "Analysis Techniques For SIMD Machine Interconnection Networks And The Effects of Processor Address Masks," IEEE Tran. on Computer, vol. C-26, no. 2, pp. 153-161, Feb. 1977.

6. Thompson, Clark D., "Generalized Connection Networks For Parallel Processor Intercommunication," IEEE Tran. on Computer, vol. C-27, no. 12, pp. 1119-1125, Dec. 1978.

7. Kung, Sun Yuan, K. S. Arun, Ron J. Gal-Ezer, and D. V. Bhaskar Rao, "Wavefront Array Processor: Language, Architecture, And Applications," IEEE Tran. on Computer, vol. C-31, no. 11, pp. 1054-1065, Nov. 1982.

8. Horowitz, Ellis and Alessandro Zorat, "The Binary Tree As An Interconnection Network: Applications To Multiprocessor Systems And VLSI," IEEE Tran. on Computer, vol. C-30, no. 4, pp. 247-253, Apr. 1981.

9. Denyer, Peter and David Renshaw, VLSI Signal Processing: A Bit-serial Approach, Addison-Wesley, 1985.

10. Fisher, Allan L. and H. T. Kung, "Synchronizing Large VLSI Processor Arrays," IEEE Tran. on Computer, vol. C-34, no. 8, pp. 734-740, Aug. 1985.

11. Gentleman, W. Morven, David Nassimi, and Sartaj Sahni, "Data Broadcasting In SIMD Computer," IEEE Tran. on Computer, vol. C-30, no. 2, pp. 101-107, Feb. 1981.

12. Hwang, Kai and Faye A. Briggs, in Computer Architecture and Parallel Processing, McGraw-Hill Book Company, 1984.

13. Stone, Harold S., "Parallel Processing With The Perfect Shuffle," IEEE Tran. on Computer, vol. C-20, no. 2, pp. 153-161, Feb. 1971.

14. Jin, Lan and Wei-min Zheng, "Analysis And Modeling Of A Splitted-bus Distributed Multiprocessor System," Inter-

national Conference on Parallel Processing (ICPP), pp. 345-346, 1982.

15. Leiserson, Charles E. and James B. Saxe, "Optimizing Synchronours Systems," Jour. of VLSI and Computer Systems, vol. 1, pp. 41-63, 1983.

16. Nassimi, David and Sartaj Sahni, "An Optimal Routing Algorithm For Mesh-Connected Parallel Computers," Jour. Assoc. Comput. Math., vol. 27, no. 1, pp. 6-29, Jan. 1980.

17. Kung, H. T., "The Structure of Parallel Algorithms," Technical Report, MIT, Dept. of Computer Science, Aug. 1979.

18. Schwartz, J. T., "Ultracomputers," ACM Tran. on Programming Languages And Systems , vol. 2, no. 4, pp. 484-501, Oct. 1980.

19. Siegel, Howard Jay, Robert J. Mcmillen, and Philip T. Muller,Jr, "A Survey of Interconnection Methods For Reconfigurable Parallel Processing Systems," IEEE Tran. on Computer, vol. C-27, no. 1, pp. 529-541, Jan. 1979.

20. Nassimi, David and Sartaj Sahni, "Parallel Permutation And Sorting Algorithms And A New Generalized Connection Network," Journal of The Association For Computing Machinery, vol. 29, no. 3 , pp. 642-667, July, 1982.

21. Batcher, K. E., "Sorting Networks And Their Application," Spring Joint Computer Conference, pp. 307-314, 1968.

22. Patterson, David A., E. Scott Fehr, and Carlo H. Sequin, "Design Considerations For The VLSI Processor or X-Tree," Proc. of the 6th Annual Symposium on Computer Architecture, Philadelphia, PA, USA, pp. 90-101, 23-25 April 1979.

23. Hills, W. Daniel, in The Connection Machine, MIT Press, 1985.

24. Preparata, Franco P., "New Parallel-Sorting Schemes," IEEE Tran. on Computer, vol. 27, July 1978.

25. Siegel, Howard Jay, "The Organization and Language Design of Microprocessors for an SIMD/MIMD System ," Proc. of the 2nd Rocky Mountain Symposium on Microcomputers: System, Software, Architecture, Pingree Park, CO, USA, pp. 311-340, Aug. 1978.

26. Kung, H. T., "Let's Design Algorithms For VLSI Systems," In Proceedings of Conference on Very Large Scale Integration: Architecture, Design, Fabrication, California Inst. of Technology, pp. 65-90, Jan. 1979.

27. Kung, H. T., Lawrence M. ~~Ruane, and~~ David W. L. Yen, "A Two-Level Pipelined Systolic Array For Convolutions," in VLSI Systems And Computations, ed. H. T. Kung et al,

pp. 255-264, Oct. 1981.

28. Cappello, Peter R. and Denneth Steiglitz, "Digital Signal Processing Applications of Systolic Algorithms," in VLSI Systems And Computations, ed. H. T. Kung et al, pp. 245-254, Oct. 1981.

29. Fisher, Allan L., "Systolic Algorithms For Running Order Statistics In Signal And Image Processing," in VLSI Systems And Computations, ed. H. T. Kung et al, pp. 265-272, Oct. 1981.

30. Fisher, Allan L., H. T. Kung, Louis M. Monier, and Yasunori Dohi, "The Architecture of A Programmable Systolic Chip," Jour. of VLSI And Computer Systems, vol. 1, no. 2, pp. 153-169, 1984.

31. Fisher, Allan L., "Implementation Issues For Algorithmic VLSI Processor Arrays," Technical Report, MIT, Dept. of Computer Science, Oct. 1984.

32. Kung, H. T., "Some System And Implementation Issues In Systolic Algorithm Design," Digital Processings 1984, Proceeding of International Conference, Florence, Italy, 1984.

33. Kung, H. T., "Systolic Algorithms For The CMU Warp Processor," Seventh International Conference on Pattern Recognition, Montreal, vol. 1, pp. 570-576, Aug. 1984.

34. Kung, H. T., "Systolic Algorithms And Their Implementation," Proceeding of Seventeenth Hawaii International Conference on System Science 1984, Honolulu, Hi, USA, pp. 4-6, Jan. 84.

35. Kung, H. T., "Why Systolic Architectures," Computer Magazine, vol. 15, no. 1, pp. 37-46, Jan. 1982.

36. Kung, H. T. and C. E. Leiserson, "Systolic Arrays (For VLSI)," in Introduction To VLSI Systems, Section 8.3, By C. A. Mead And L. A. Conway, pp. 400-414, Addison-Wesley, 1980.

37. Yen, D. W. L. and A. V. Kulkarni, "Systolic Processing And An Implementation For Signal And Image Processing," IEEE Tran. on Computer, vol. C-31, no. 10, pp. 1000-1009, Oct. 1982.

38. Savage, Carla, "A Systolic Data Structure Chip For Connectivity Problem," in VLSI Systems And Computations, ed. H. T. Kung et al, pp. 273-284, Oct. 1981.

39. Li, H. F., R. Jayakumar, and X. Sun, "Bit-serial Systolic Sorting: general complexities and an implementation in VLSI," IEE Proceedings-E Computers and Digital Techniques, vol. 134 PartE, no. 3, pp. 125-132, May 1987.

40. Lehman, Philip L., "A Systolic (VLSI) Array For Processing Simple Relational Queries," in VLSI Systems And

Computations, ed. H. T. Kung et al, pp. 285-295, Oct. 1981.

41. Li, H. F. and R. Jayakumar, "Systolic Sorting In VLSI," Technical Report CCSD-VLSI-85-1, Dept. of Computer Science, Concordia University, Canada, 1985.

42. Smith, C. U., "Technology Transfer, VLSI and Software Engineering," Minnowbrook Conf. on Software Performance Evaluation, Blue Mountain Lake, NY, July 1984.

43. Agarwal, Prathima and Michael J. Meyer, "Automation In The Design of Finite-State Machines," VLSI Design, pp. 74-84, Sept. 1984.

44. Gajski, Daniel D., "Silicon Compilation," VLSI System Design, pp. 48-64, Nov. 1985.

45. Hill, Fredrick J., Zainalabedin Navabi, Chen H. Chiaing, Duan-Ping Chen, and Manzer Masud, "Hardware Compilation From An RTL To Storage Logic Array Target," IEEE Tran. on Computer-Aided Design of ICS and Systems, vol. CAD-3, pp. 208-217, July 1984.

46. Ayres, Ron, "Silicon Compilation - A Hierarchical Use of PLAs," CalTech Conference on VLSI, pp. 311-326, Jan. 1979.

47. Bergmann, Neil, "A Case Study of The F.I.R.S.T. Silicon Compiler," Proc. Third CalTech Conf. of VLSI, Pasedena, Calif., pp. 413-430, Mar. 1983.

✓

48. Cheng, Edmund K., "Verifying Compiled Silicon," VLSI Design, pp. 70-74, Oct. 1984.

49. Feldman, Stuart I., "Xi," Proceedings IEEE International Conference on Computer Design: VLSI In Computers (ICCD'83), pp. 652-655, Nov. 1983.

50. Fox, Jeffrey R., "Performance Prediction With The MacPitts Silicon Compiler," in Proc. Custom Integrated Circuit Conference IEEE, pp. 351-355, 1984.

51. Haynes, G. A., "SLED/SDL: A Flexible Symbolic VLSI Design System," Proceedings IEEE International Conference on Computer Design (ICCD'83), pp. 409-412, Nov. 1983.

52. Hild, Marc, "Converting PLAs To CMOS Gate Arrays," VLSI Design, pp. 58-63, March 1984.

53. Johannsen, Dave, "Bristle Blocks: A Silicon Compiler," Proc. 16th Design Automation Conf., pp. 310-313, June 1979.

54. Johnson, Stephen C., "VLSI Circuit Design Reaches The Level of Architectural Description," Electronics, pp. 156-161, May 3, 1984.

55. Krekelberg, David E., Gerald E. Sobelman, and Chu S. John, "Yet Another Silicon Compiler," 22nd Design Automation Conference, pp. 176-182, 1985.

56. Lursinsap, Chidchanok and Daniel Gajski, "Cell Compilation With Constraints," Proc. ACM/IEEE 21st Design Automation Conf., pp. 103-108, 1984.

57. Deas, A. R., Silicon Compilation: A VLSI Complexity Management Strategy, Master Thesis, University of Edinburgh, 1983.

58. Powell, Patrick A. D. and Mohamed I. Elmasry, "The ICE-WATER Language And Interpreter," 21st Design Automation Conference, pp. 98-102, 1984.

59. Rosenberg, Jonathan, David Boyer, John Dallen, Stephen Daniel, Charles Pierier, John Poulton, Durward Rogers, and Neil Weste, "A Vertically Integrated VLSI Design Environment," 20th Design Automation Conference, pp. 31-35, 1983.

60. Siskind, Jeffrey Mark, Jay Roger Southard, and Kenneth Walter Crouch, "Generating Custom High Performance VLSI Designs From Succinct Algorithmic Descriptions," 1982 Conference on Advanced Research In VLSI, MIT, pp. 28-39, 1982.

61. Southard, Jay R., "MacPitts: An Approach To Silicon Compilation," Computer, vol. 6, no. 12, pp. 74-82, Dec. 1983.

62. VLSI-Design-Staff,, "Silicon Compilers Part 2: Casting An Image," VLSI Design, pp. 65-68, Sept. 1984.

63. VLSI-Design-Staff,, "Silicon Compilers Part 1: Drawing A Blank," VLSI Design, pp. 54-58, Oct. 1984.

64. Williams, John D.,. "Sticks -- A Graphical Compiler For High Level LSI Design," National Computer Conference, pp. 289-295, 1978.

65. Hartman, Alfred C., "Software Or Silicon? The Designer's Option," Proceedings of The IEEE, vol. 74, no. 6, pp. 861-873, June 1986.

66. Fiebrich, Rolf-Dieter, Yuh-Zen Liao, George Kopp Lam, and Edward Adams, "PAI: A Symbolic Layout System," IBM Jour. Res. Develop., vol. 28, no. 5, pp. 572-580, Sept. 1984.

67. Elder, W. H, P. P. Zenewicz, and R. R. Alvarodiaz, "An Interactive System for VLSI Chip Physical Design," IBM Jour. Res. Develop., vol. 28, no. 5, pp. 524-536, Sept. 1984.

68. Ghezzi, C. and M. Jazayeri, Programming Language Concepts, Wiley, 1982.

69. David, Alan L. and Robert M. Keller, "Data Flow Program Graphs," IEEE Tran. on Computer, vol. C-30, no. 2, pp. 26-41, Feb. 1982.

70. Gostelow, Kim P. and Robert E. Thomas, "A View of Dataflow," in proc. of National Computer Conference, pp. 629-636, 1979.

71. Watson, Ian and John Gurd, "A Prototype Data Flow Computer With Token Labelling," in proc. of National Computer Conference, pp. 623-628, 1979.

72. Dennis, Jack B., "Data Flow Supercomputers," Computer, vol. 13, pp. 48-50, Nov. 1980.

73. Kung, S. Y., K. S. Arun, D. V. Bhaskar Rao, and Y. H. Hu, "A Matrix Data Flow Language/Architecture For Parallel Matrix Operations Based on Computational Wavefront Concept," in VLSI Systems And Computations, ed. H. T. Kung and et al, pp. 235-244, Oct. 1981.

74. Ackerman, William B., "Data Flow Languages," in proc. of National Computer Conference, pp. 1087-1095, 1979.

75. Gonauser, Monika and Anton M. Sauer, "Needs For High-Level Design Tools," IEEE international conf. (ICCD'83) on Computer Design: VLSI in Computers, pp. 415-418, 1983.

76. Lee, Kyu Y., Michael J. Holley, Mary L. Bailey, and Walter Bright, "A High-Level Design Language For Programmable Logic Devices," VLSI Design, pp. 50-62, June 1985.

77. Sun, X., H. F. Li, and R. Jayakumar, "Reflection on Systolic Architecture: A High Level Data Flow Language For Silicon Compilation," in Proc. Canadian Conf. on VLSI, pp. 301-306, Oct. 1987.

78. Robson, Gary, "Logic Design Using Behavioral Models," VLSI Design, pp. 36-44, Jan. 1984.

79. Kober, Rudolf and Wolfgang Wenderoth, "Problems And Practical Experience In High-Level Design," IEEE International Conf. (ICCD'83) on Computer Design: VLSI in Computers, pp. 427-430, 1983.

80. Mead, C. A. and L. A. Conway, in Introduction to VLSI systems, Addison-Wesley, 1980.

81. Weste, Neil and Kamran Eshraghian, in Principles of CMOS VLSI Design, Addison-Wesley, 1985.

82. Smith, Connie U. and Richard R. Gross, "Technology Transfer Between VLSI Design And Software Engineering: CAD Tools And Design Methodologies," Proceedings of The IEEE, vol. 74, no. 6, pp. 875-885, June 1986.

83. Lopex, Alexander D. and Hung-Fai S. Law, "A Dense Gate Matrix Layout Method For MOS VLSI," IEEE Tran. on Electron Devices, vol. Ed-27, no. 8, pp. 1671-1675, Aug. 1980.

84. Todd, L. F. et al., "A Multitechnology Gate Array Layout System," in Proc. 19th Design Automation Conf., p. 792, Las Vegas, NV, June 1982.

85. Gray, J. P., I. Buchanan, and P. S. Robertson, "Design Gate Arrays Using A Silicon Compiler," in Proc. 19th Design Automation Conf., p. 377, Las Vegas, NV, June

1982.

86. Hedges, T. S. et al., "The Siclops Silicon Compiler," in Proc. IEEE Int. Conf. on Circuits and Computers, p. 277, New York, Sept. 1982.

87. Gajski, D. G., "The Structure Of A Silicon Compiler," in Proc. IEEE Int. Conf. on Circuits and Computers, pp. 272-276, New York, Sept. 1982.

88. McGraw, James R., "The VAL Language: Description and Analysis," ACM TOPLAS, vol. 4, no. 1, pp. 44-82, Jan. 1982.

89. Plas, A., D. Courte, O. Gelly, and J. C. Syre, "LAU System Architecture: A Parallel Data Driven Processor Based on Single Assignment," Proc. 1976 Int'l Conf. Parallel Processing, Aug. 1976.

90. Pao, Derek, "A General Channel Router," Internal report, Dept. of Computer Science, Concordia Unviersity, Montreal, Canada, July, 1986.

91. Deutsch, David N., "A Dogleg Channel Router," 13th Design Automation Conference, pp. 425-430, 1976.

92. Yoshimura, Takeshi and Ernest S. Kuh, "Efficient Algorithms For Channel Routing," IEEE Tran. on Computer-Aided Design of Integrated Circuits and systems, vol. CAD-1-CAD-2, pp. 25-35, Jan. 1982.

APPENDIX I:

BNF NOTATIONS

::= means "is defined as".

{ } means "any sequence of 0 or more of the items enclosed".

[ ] means "either 0 or one concurrence of the items enclosed".

| means "or" in exclusive sense.

< > means "non-terminal symbol is enclosed".

" " means "terminal symbol is enclosed".

Items are either names of classes (e.g. "identifier") or single symbols of the language's alphabet.

APPENDIX II:

SYNTAX DEFINITION OF SASCL

```
<applications> ::= <layout> | <simulation>
<layout> ::= "layout" <array name> ";"
                 <global declaration>
                 <function definition>
                 <main code>
<array name> ::= <identifier>
<global declaration> ::= {<const declaration>} <var declaration>
<const declaration> ::= "const" <const list>
<const list> ::= <constant> {<constant>}
<constant> ::= <const name> "=" <numerics> ";"
<const name> ::= <identifier>
<var declaration> ::= "var" <var list>
<var list> ::= <variables> {<variables>}
<variables> ::= <name list> ":" <var type>
<name list> ::= <var name> {"," <var name>}
<var name> ::= <identifier>
<var type> ::= <single type> | <bits type> | <elements type>
<single types> ::= "signal" | "register" | "flag"
<bits type> ::= "bits" <digit> {<digit>} | <const name> "of"
                     <single type>
<elements type> ::= elements <digit> {<digit>} | <const name> "of"
                     <var name>
```

```
<function definition> ::= <function head>
                          <ports declaration>
                          <function body>
<function head> ::= <function name> <parameter list>
<function name> ::= <identifier>
<parameter list> ::= "(" <logic direction> {<logic direction>} ")"
<logic direction> ::= <direction> ":" {<I/O pairs> | <inport name> |
                      <outport name>}
<direction> ::= <main sides> "->" | "<-" <secondary sides>
<main sides> ::= "L" | "T" | "TL" | "TR"
<secondary sides> ::= "R" | "B" | "BL" | "BR"
<I/O pairs> ::= <I/O pair> {<I/O pair>}
<I/O pair> ::= <inport name> "," <outport name> ";"
<inport name> ::= <identifier>
<outport name> ::= <identifier>
<ports declaration> ::= <ports name> ":" <ports type>
<ports name> ::= <port name> {<port name>}
<port name> ::= <inport name> | <outport name>
<ports type> ::=  "signal" | <mult bits>
<mult bits> ::= "bits" <digit> | <const name> "of" "signal"
<function body> ::= "begin" <function list> "end" ";"
<function list> ::= <local declaration> <statement list>
<local declaration> ::= {<const declaration>} <var declaration>
<statement list> ::= <statement> {<statement>}
<statement> ::= <assignment statement> | <function call> ";"
                {<comments>}
<assignment statement> ::= <normal assign> | <state assign>
```

```
<normal assign> ::= <var name> "=" <var name> | <boolean const> |
                        <function call>
<state assign> ::= <var name> "=" <symbols>
<symbols> ::= <identifier> {"," <identifier>}
<function call> ::= <function names> "(" <argument list> ")"
<function names> ::= <function name> | <sysfunction name>
<argument list> ::= {<function call> | <continued symbol> |
                        <const name> | <digit>}
<main code> ::= "begin" <statement list> "end" "."
<sysfunction name> ::= <primitive function> | <library function>
<primitive function> ::= <arithmetic fun> | <logic fun>|
                        <comparison fun> | <other fun>
<arithmetic fun> ::= "add" | "sub" | "mul" | "div" | "mod"
<logic fun> ::= "and" | "or" | "not"
<comparison fun> ::= "max" | "min" | "equal" | "less" | "great" | "lessequal" |
"greatequal"
<other fun> ::= "delay" | "link"*
<library function> ::= <topo description> | <algo description> |
                        <simu description>
<topo description> ::= "linear" | "square" | <triangle>
<triangle> ::= "tltriangle" | "trtriangle" | "bltriangle" | "bltriangle"
<algo description> ::= "fsm" | "if" | "else" | "for"
<simu description> ::= "setnode"* | "showat"* | "active"* | "alter"*
<identifier> ::= <letter> {<letter> | <digit> | "-" }
<letter> ::= "a" | "b" | ... | "z" | "A" | "B" | ... | "Z"
<digit> ::= "0" | "1" | "2" | ... | "8" | "9"
<numerics> ::= <digit> {<digit>} | ["+" | "-"] <digit> | {<digit>}
```

[Note]: Functions with * sign are for simulation.

```
<boolean const> ::= "true" | "false"

<continued symbol> ::= <digit> {<digit>} .. <digit> {<digit>} |
                       <letter> .. <letter>

<comments> ::= "/*" {<identifier> <numerics>} "*/"
```

APPENDIX III:

SYSTEM FUNCTIONS

1. PRIMITIVE FUNCTIONS

(a) Arithmetic Functions

Arithmetic functions apply to at least two integers and produce a result. We have :

add (a b c ...) ------ a + b + c + ...

sub (a b) ------ a - b

mul (a b c ...) ------ a * b * c * ...

div (a b) ------ a / b

mod (a b) ------ a mod b

(b) Logic Functions

Logic functions allow us to combine logic values. We have :

and (a b c ...) ------ a and b and c and ...

or (a b c ...) ------ a or b or c or ...

not (a) ------ not a

(c) Comparison Functions

Comparison functions could be applied to signals, registers, and flags. We have:

max (a b c ...) ------ find the biggest.

min (a b c ...) ------ find the smallest equal (a b) ------ a = b

less (a b) ------ a < b

great (a b) ------ a > b

lessequal (a b) ------ a <= b

greatequal (a b) ------ a >= b

(d) Delay Functions

delay (x t): delay signal x by t cycles.

(e) Initialization and link Functions

init (x v): set up connections between I/O pads and variable x which is a global variable, either a register or a flag data type. The v represents terminals by which variable x to be intialized. The values of v could be "r" and "s" which corresponds to terminals "set" and "reset" of a storage register respectively. The default represents terminal "d".

link (x): linking an external program to the calling program.

2. LIBRARY FUNCTIONS

(a) For Network Topology:

linear(fname n): generate a logically linearly connected network (the physical layout may not be linear one) with n function blocks called fname.

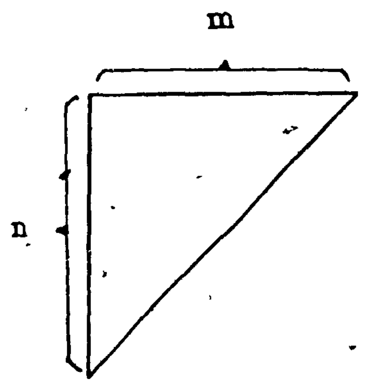
square(fname m n): generate a 2-D network with m * n function blocks, called fname. Topologies covered by this function are:

. mesh connected networks

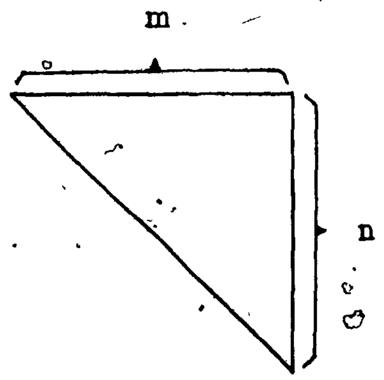
. mesh connected networks with diagonal connections between function blocks

. 2-D hexagonal array.

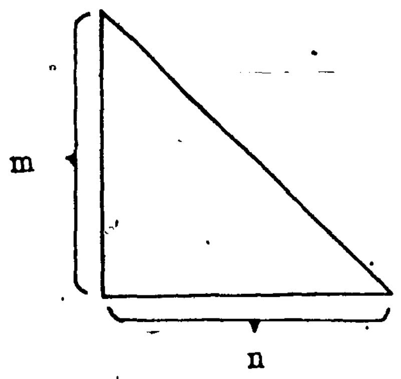
tltriangle(fname m n): generate a top-left triangle network with size of m * n.



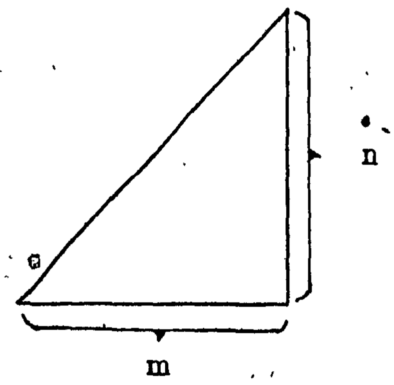
trtriangle(fname m n): generate a top-right triangle network with size of m * n.



bltriangle(fname m n): generate a bottom-left triangle network with size of m * n.



brtriangle(fname m n): generate a bottom-right triangle network with size of m * n.



(b) For Algorithm Description :

fsm(state1 (condition)(action)
state2 (condition)(action)

staten (condition)(action)): fsm is used to define finite state machine.

if((expression)(statement1)
else (statement2)) : is used to either make a decision or represent a hardware switch.

(c) For Simulation:

setnode(fname n1 n2 ...): used to set nodes inside the given function. The states of the nodes should be listed after each simulation steps.

showat(cycle1,cycle2 ...): used to set up the interrupt points of a simulation.

load(varname,value):used to set initial value of the variable.

active(fname,mode): simulate the function according to the active mode.

alter(mode1 mode2): alternation of mode1 and mode2.

APPENDIX IV:

SYNTAX DEFINITION OF SASCIL

<statements> ::= {<statement1> | <statement2>}

<statement1> ::= <items> "=" <items> | <boolean type>

<items> ::= <simple type> | <register type>

<simple type> ::= <signal-temp> | <port>

<signal-temp> ::= "[" <identifier> "signal" | "temp" "]"

<port> ::= "[" <port head> <l/o> <location> "]"

<port head> ::= <identifier> "port"

<l/o> ::= "l" | "o"

<location> ::= "l" | "r" | "t" | "b"

<register type> ::= "[" <register head> <short list> | <long list> "]"

<register head> ::= <identifier> "register"

<short list> ::= <register type1> <#bits> <phase> <d> <q> <$\bar{q}$>

<register type1> ::= "0" | "2" | "3" | "4" | "5"

<#bits> ::= <digit> {<digit>}

<phase> ::= "1" | "2"

<d> ::= <identifier>

<q> ::= <identifier> | <not used>

<$\bar{q}$> ::= <identifier> | <not used>

<not used> ::= "!"

<long list> ::= <register type2> <#bits> <phase> <d> <q> <$\bar{q}$>
        <r> <s>

<register type2> ::= "1" | "6" | "7" | "8" | "9"

```
<r> ::= <identifier> | <not used>

<s> ::= <identifier> | <not used>

<boolean type> ::= "[" <boolean head> <equation> "]"

<boolean head> ::= <identifier> "boolean" <#variable> <#product>
                   <#member> <speed factor>

<#variable> ::= <digit> {<digit>}

<#product> ::= <digit> {<digit>}

<#member> ::= <digit> {<digit>}

<speed factor> ::= "f" | "m" | "s"

<equation> ::= { <identifier> <logic symbol> } <identifier>

<logic symbol> ::= "*" | "+"

<statement2> ::= <lib type> | <event type>

<lib type> ::= <primitive lib> | <users lib>

<primitive lib> ::= "[" <identifier> "lib" "0" <lib parameters> "]"

<lib parameters> ::= <driver-clock> | <arithmetic> | <others>

<driver-clock> ::= <identifier> <drivers> | <clockgens> | <iopads>

<drivers> ::= "Driver8" | "Driver15" | "Driver30" |
              "Ndriver8" | "Ndriver15" | "Ndriver30" |
              "Inpad" | "Outpad" | "Vddpad" | "Gndpad"
               <common para> <in> <out>

<common para> ::= <level> <#bits> <phase> <#ports>

<level> ::= "0" | "1" | "2" | "3"

<#ports> ::= <digit> {<digit>}

<in> ::= <identifier>

<out> ::= <identifier>

<clockgens> ::= "Clockgen8" | "Clockgen15" | "Clockgen30"
                <common para> <in> <out> <out>
```

```
<iopad> ::= "Inpad" | "Outpad" <common para> <in1>
<in1> ::= <identifier> | <not used>
<arithmetic> ::= <add> | <multiply>
<add> ::= "Adder" | "Substractor" <common para> <sum> <cout>
               <in1> <in1> <r> <s>
<sum> ::= <identifier>
<cout> ::= <identifier> | <not used>
<multiply> ::= "Multiplier" | "Add-multiply" <common para> <product>
                  <in1> <in1> <r> <s>
<product> ::= <identifier>
<others> ::= <switch-comp> | <mult> | <demult>
<switch-comp> ::= "Switchbox4" | "Comparator" <common para> <out>
               <out> <in> <in>
<mult> ::= "Multiplexer" <common para> <out> <ctl> <ctl> <ctl>
               <in> <in> <in> <in>
<ctl> ::= <identifier>
<demult> ::= "Demultiplexer" <common para> <out> <out> <out>
               <out> <ctl> <ctl> <in> <in>
<users lib> ::= "[" <userlib name> "lib" "1" <identifier>
                  <common para> <ports> "]"
                  "{" <statements> "}"
<userlib name> ::= <identifier>
<ports> ::= { "(" <identifier> <identifier> | <identifier> ")" }
<event type> ::= "[" <event head> <event parameter> "]"
                  "{" <path list> "}"
<event head> ::= <identifier> "event"
<event parameter> ::= <function name> <level> <#path> <#cycles>
```

```
<function name> ::= <identifier>
<#path> ::= <digit> {<digit>}
<#cycles> ::= <digit> {<digit>}
<path list> ::= <path name> {<simple type>}
<identifier> ::= <letter> {<letter> | <letter> | <digit> | "-"}
<letter> ::= "a" | "b" | ... | "z" | "A" | ... | "Z"
<digit> ::= "0" | "1" | ... | "9"
```

APPENDIX V:

PRIMITIVE LIBRARY

Table of Contents

Table of Contents 148

Drlver8 (Drlver15, Drlver30) 150

Ndrlver8 (Ndrlver15, Ndrlver30) 151

Clockgen8 (Clockgen15, Clockgen30) 152

Shlftregdc1 (Shlftregdc2) 153

Shlftregdrsc1 (Shlftregdrsc2) 154

Rotateshlftdc1(Rotateshlftdc2) 155

Rotateshlftdrsc1 (Rotateshlftdrsc2) 156

Adderc1 (Adderc2) 157

Substractc1 (Substractc2) 158

Comparator 159

Swltchbox4 160

Multlplexer 161

Demultlplexer 162

Inpad 163

Outpad 164

Vddpad 165

Gndpad 166

Frame 167

Loglc 168

Multlpller 169

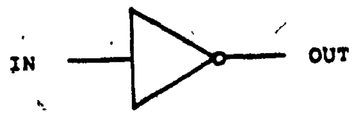
Add-multiplier 170

Name: Driver8 (Driver15, Driver30)

Function: Inverted driving for 8 (15 or 30) loads.

Leaf-Cell-Used: driver8 (driver15, driver30)

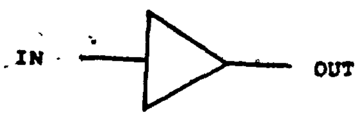
Logic-Symbol:



I/O-Ports: Input: IN

Output: OUT

Name:	Ndriver8 (Ndriver15, Ndriver30)
Function:	Non-inverted driving for 8 (15 or 30) loads.
Leaf-Cell-Used:	ndriver8 (ndriver15, ndriver30)
Logic-Symbol:	



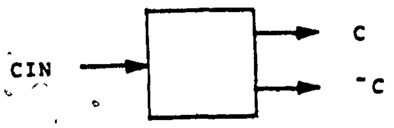
I/O-Ports:	Input: IN
	Output: OUT

Name: Clockgen8 (Clockgen15, Clockgen30)

Function: Generate complementary local clocks which can drive 8 (15 or 30) loads.

Leaf-Cell-Used: driver8 (driver15, driver30), ndriver8 (ndriver15, ndriver30)

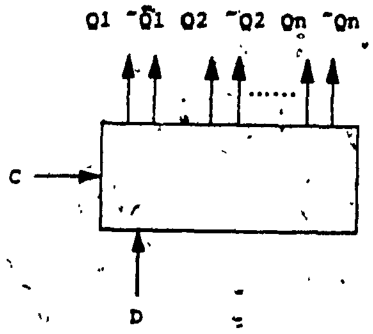
Logic-Symbol:



I/O-Ports: Input: CIN

Output: C, $\bar{C}$

Name:	Shiftregdc1 (Shiftregdc2)
Function:	Shift and store data. It is D-latch type shift register and could be triggered by system clock 1 or 2..
Number-of-Bits:	n
Leaf-Cell-Used:	dlatchc1 (dlatchc2)
Logic-Symbol:	



I/O-Ports:	Input: C1(C2), D
	Output: Q1, $\overline{Q\,1}$, Q2, $\overline{Q\,2}$, ... Qn, $\overline{Qn}$

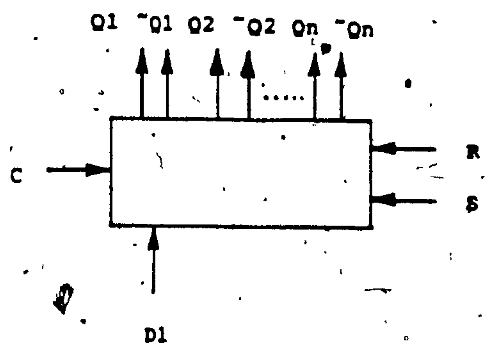
Name: Shiftregdrsc1 (Shiftregdrsc2)

Function: Shift and store data. It is D-flip-flop type shift register and could be triggered by system clock phase 1 or 2.

Number-of-Bits: n

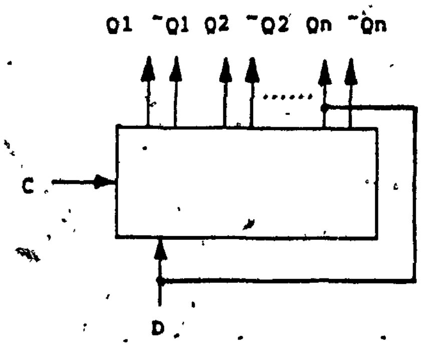
Leaf-Cell-Used: drsc1 (drsc2)

Logic-Symbol:



I/O-Ports: Input: C1 (C2), D1, R, S

Output: Q1, $\overline{Q\,1}$, Q2, $\overline{Q\,2}$, ... Qn, $\overline{Qn}$

Name:	Rotateshiftdc1 (Rotateshiftdc2)
Function:	Store data in D-latch type register. It could be triggered by system clock 1 or 2.
Number-of-Bits:	n
Leaf-Cell-Used:	dlatchc1 (dlatchc2)
Logic-Symbol:	
I/O-Ports:	Input: C1 (C2), D
	Output: Q1, $\overline{Q1}$, Q2, $\overline{Q2}$, ... Qn, $\overline{Qn}$

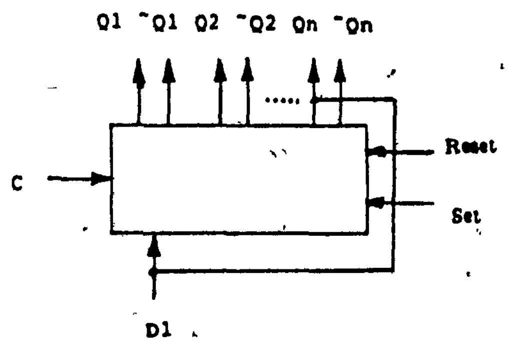
Name: Rotateshiftdrsc1 (Rotateshiftdrsc2)

Function: Store data in D-flip-flop type register. It could ~~be triggered~~ by system clock 1 or 2.

Number-of-Bits: n

Leaf-Cell-Used: drsc1 (drsc2)

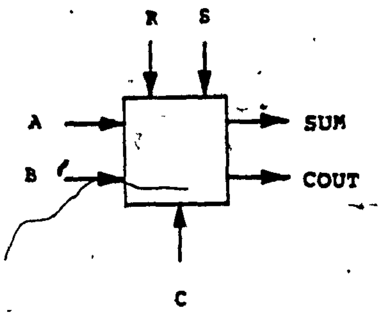
Logic-Symbol:



I/O-Ports: Input: C1 (C2), D1 Reset, Set

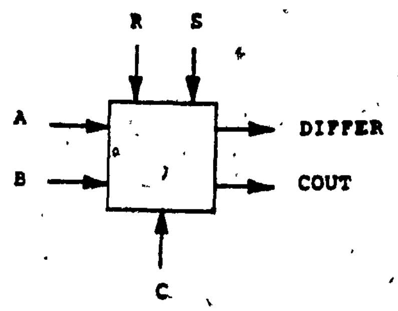
Output: Q1, $\overline{Q1}$, Q2, $\overline{Q2}$, ... Qn, $\overline{Qn}$

Name:	Adderc1 (Adderc2)
Function:	Bit addition clocked by system clock 1 or 2. Set and reset functions to the internal bit delay are provided.
Number-of-Bits:	1
Leaf-Cell-Used:	fab1, drsc1 (drsc2)
Logic-Symbol:	



I/O-Ports:	Input: A, B, R, S, C1(C2)
	Output: COUT, SUM

Name:	Substractc1 (Substractc2)
Function:	Bit substraction clocked by system clock 1 or 2. Set and Reset functions to the internal bit delay are provided.
Number-of-Bits:	1
Leaf-Cell-Used:	fba1, drsc1 (drsc2), driver8
Logic-Symbol:	



I/O-Ports:	Input: A, B, R, S, C1(C2)
	Output: DIFFER, COUT

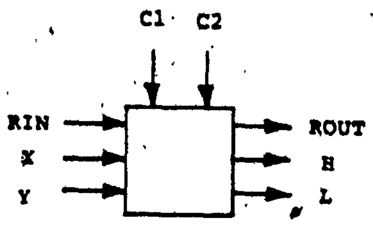
Name: Comparator

Function: One-bit compare-exchange. The Rin and Rout act as input and output reset lines respectively. Cin and Yin are the corresponding bits of the two numbers compared and the bit corresponding to the larger number is output at H and smaller number is output at L. The comparator works under two-phase system clock.

Number-of-Bits: 1

Leaf-Cell-Used: driver8, ndriver8, driver30, ndriver30, dlatchc1, dlatchc2, drsc1, Hf*, Lf*, C1*, C2*

Logic-Symbol:



I/O-Ports: Input: RIN, X, Y, C1, C2

Output: ROUT, H, L

[*]: cells are generated by Logic Compiler.

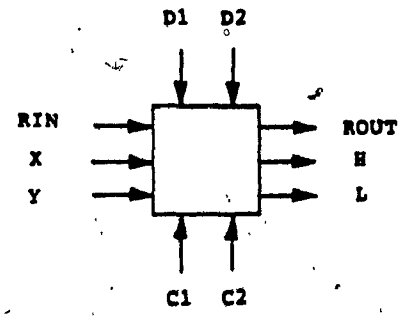
Name: Switchbox4

Function: The four-function switchbox performs data transfer under independent box control:

D1	D2	Function
0	0	straight
0	1	exchange
1	0	lower broadcast
1	1	upper broadcast

Leaf-Cell-Used: dlatchc1, dlatchc2, Hfunction*, Lfunction*

Logic-Symbol:



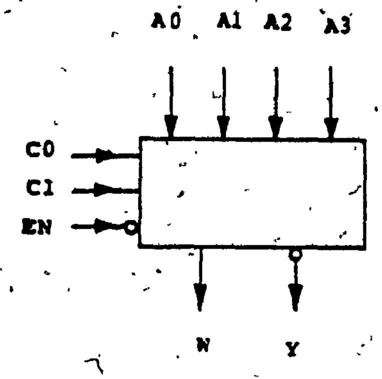
I/O-Ports: Input: RIN, X, Y, D1, D2, C1, C2

Output: ROUT, H, L

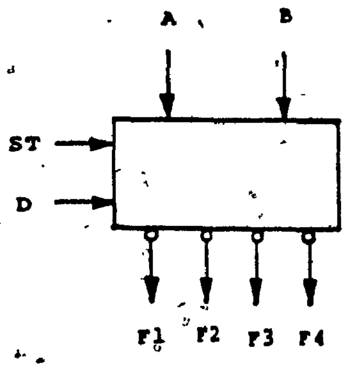
[*]: cells are generated by Logic Compiler.

Name:	Multiplexer
Function:	Two control signals (C0 and C1) are used to selectively connect one of the four inputs (A0-A3) tot the output W (Y). Signal EN enables the output.
Leaf-Cell-Used:	multiplexer4-1

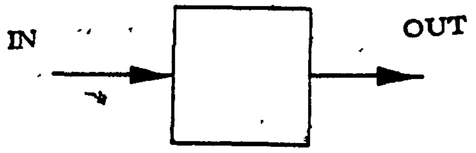
Logic-Symbol:



I/O-Ports:	Input: C0, C1, EN, A0, A1, A2, A3
	Output: W, Y

Name:	Demultiplexer
Function:	Performing the inverse function of a multiplexer: it distributes the input signal D to output lines (F1-F4) designated by the state of input A and B. Signal ST is used to strobe the output.
Leaf-Cell-Used:	demultiplexer1-4
Logic-Symbol:	
I/O-Ports:	Input: ST, D, A, B
	Output: F1, F2, F3, F4

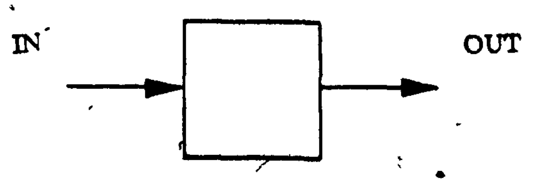
Name:	**Inpad**
Function:	**Receive signals from outside of the chip.**
Number-of-Bits:	**1**
Leaf-Cell-Used:	**inpad**
Logic-Symbol:	



I/O-Ports:	**Input: IN**
	Output: OUT

Name:	Outpad
Function:	Send internal signals to outside of the chip.
Number-of-Bits:	1
Leaf-Cell-Used:	outpad

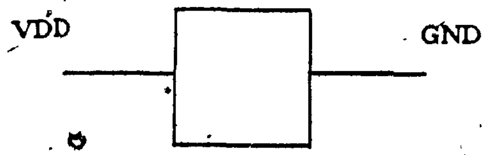
Logic-Symbol:



I/O-Ports:	Input: IN
	Output: OUT

Name:	Vddpad
Function:	Power pad.
Number-of-Bits:	1
Leaf-Cell-Used:	vddpad

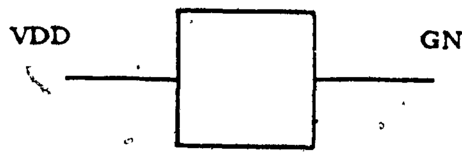
Logic-Symbol:



Ports:	VDD, GND

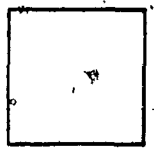
Name:	Gndpad
Function:	Ground pad
Number-of-Bits:	1
Leaf-Cell-Used:	gndpad

Logic-Symbol:



Ports: VDD, GND

Name:	Frame
Function:	Chip frame.
Leaf-Cell-Used:	frameA, frameB, frameC, frameD or free size frame*.
Logic-Symbol:	

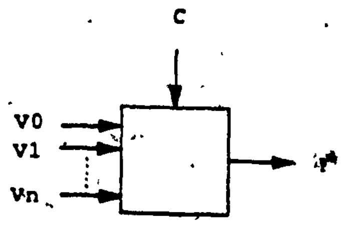


[*]: Free size frame is program generated.

Name: Logic*

Function: Generate boolean expression mask geometry according to user specification.

Logic-Symbol:

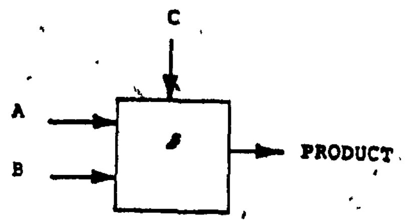


I/O-Ports: Input: C, V1, V2, ...Vn

Output: F

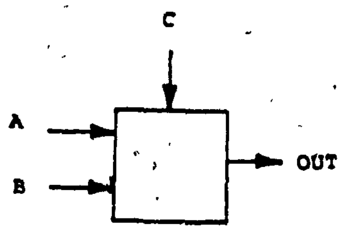
[*]: Limitations on the number of prodects and vairable in a boolean equation are: if P is infinit, V should be less than or equal to 9, and vice versa.

Name:	Multiplier
Function:	Bit multiplication:
	Product = A * B.
Number-of-Bits:	1
Leaf-Cell-Used:	
Logic-Symbol:	



I/O-Ports:	Input: A, B, Clock
	Output: Product

Name:	Add-multiplier
Function:	Bit add and multiplication.
Number-of-Bits:	1
Leaf-Cell-Used:	
Logic-Symbol:	



I/O-Ports:	Input: A, B, C
	Output: OUT

APPENDIX VI:

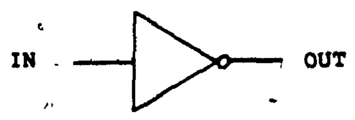
LEAF CELL LIBRARY

Table of Contents

Table of Contents 172
Driver8 (Driver15, Driver30) 173
Ndriver8 (Ndriver15, Ndriver30) 174
Dlatchc1 (Dlatchc2) 175
Drsc1 (Drsc2) 176
Fba1 177
Multiplexer4-1 178
Demultiplexer1-4 179
Inpad 180
Outpad 181
Vddpad 182
Gndpad 183
Frame 184
Logic 185

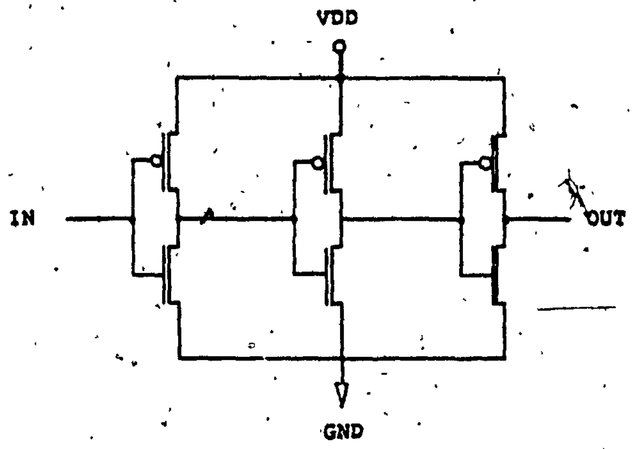
Name: Driver8 (Driver15, Driver30)

Logic-Symbol:



Function: OUT = $\overline{\text{IN}}$

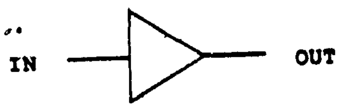
Circuit-Diagram:



Bounding-Box:	Driver8	139 x 165
	Driver15	163 x 165
	Driver30	216 x 165

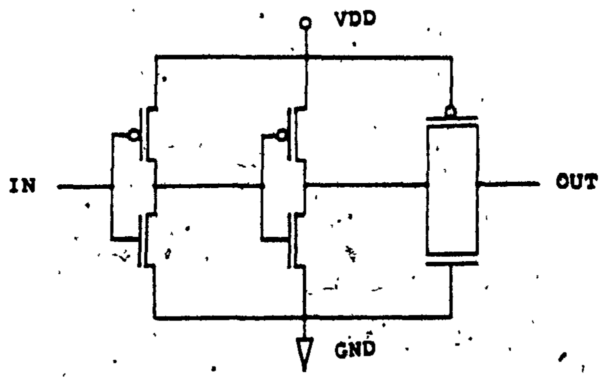
Name:	Ndriver8 (Ndriver15, ndriver30)

Logic-Symbol:



Function:	OUT = IN

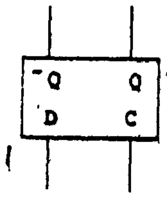
Circuit-Diagram:



Bounding-Box:	Ndriver8	119 x 165
	Ndriver15	140 x 165
	Ndriver30	210 x 165

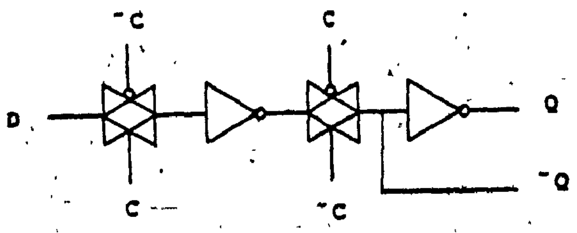
Name: Dlatch1 (Dlatch2)

Logic-Symbol:



Function: Q = D

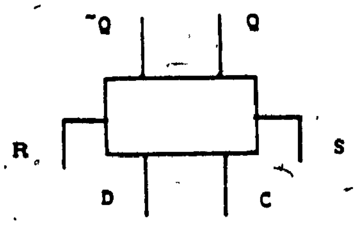
Circuit-Diagram:



Bounding-Box: 206 x 165

Name: Drsc1 (Drsc2)

Logic-Symbol:



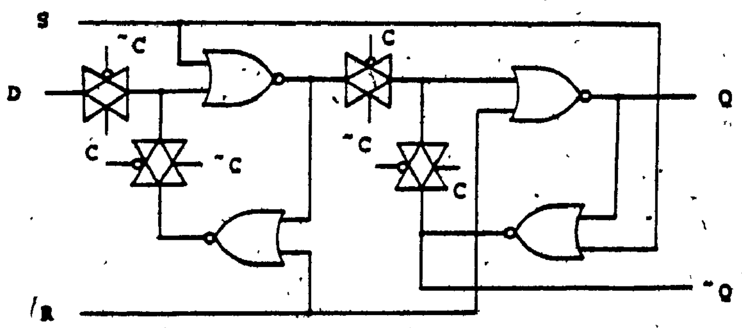
Function:

Q ≐ D

Q = S | s=1

Q = R | r=1

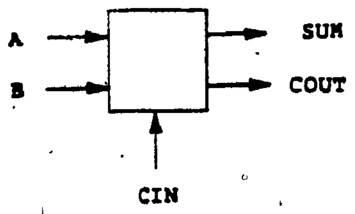
Circuit-Diagram:



Bounding-Box: 309 x 261

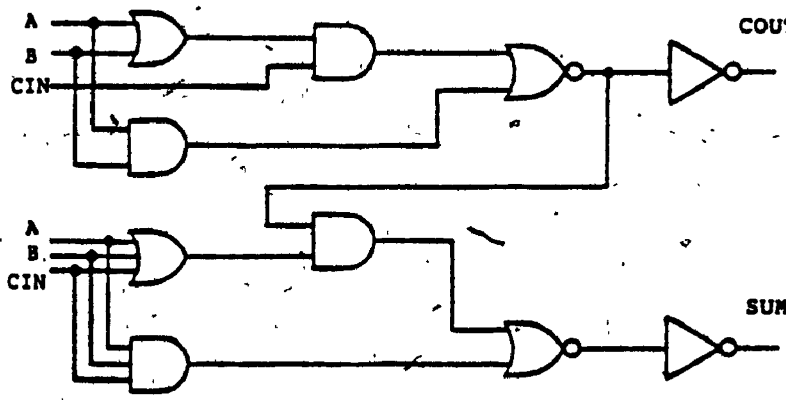
Name: Fabi

Logic-Symbol:



Function: SUM = A + B

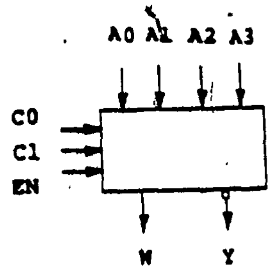
Circuit-Diagram:



Bounding-Box: 136 x 261

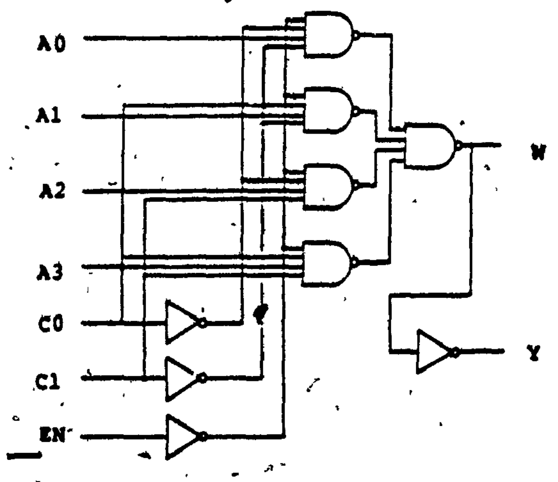
Name: Multiplexer4-1

Logic-Symbol:



Function: The 4-input multiplexer selects one of the four inputs to appear on the output. The two bits binary code C0, C1 determines which input will appear on the output. When enable is high, we will get the output from W / Y.

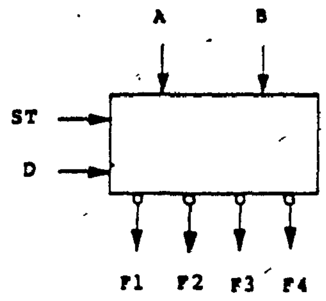
Circuit-Diagram:



Bounding-Box: 411 x 314

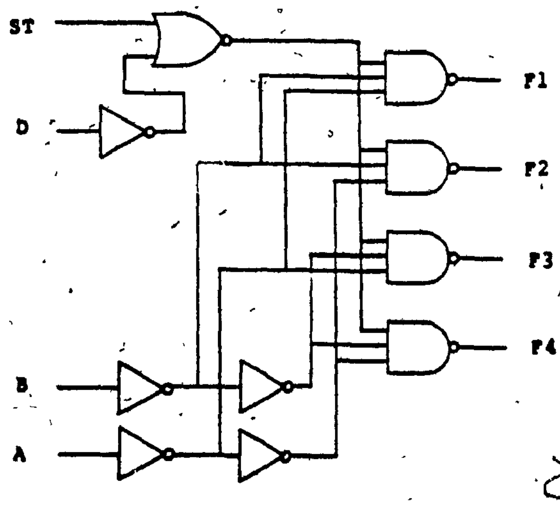
Name: Demultiplexer1-4

Logic-Symbol:



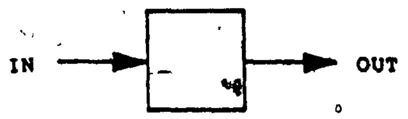
Function: Demultiplexer is the opposite of a multiplexer. In the demultiplexer, a binary input address, A or B, determines which one of the four outputs will go low.

Circuit-Diagram:



Bounding-Box: 370 x 341

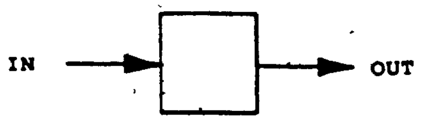
Name:	Inpad
Logic-Symbol:	



Function:	Receive signals from outside of a chip and pass them to the VLSI circuit.
Bounding-Box:	375 x 260

Name: Outpad

Logic-Symbol:

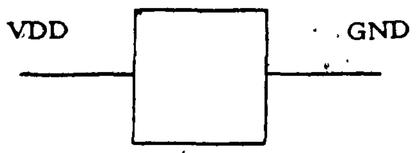


Function: Receive signals from VLSI circuit and send them to outside.

Bounding-Box: 396 x 258

Name:	Vddpad

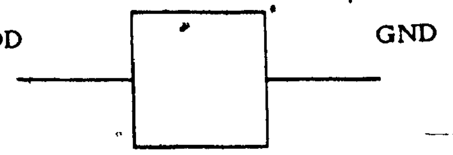
Logic-Symbol:



Function:	Pad to feed power for the chip.
Bounding-Box:	165 x 260

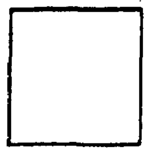
Name: Gndpad

Logic-Symbol:



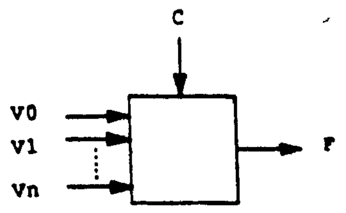
Function: The ground pad.

Bounding-Box: 165 x 260

Name:	Frame
Logic-Symbol:	
Function:	The chip frame.
Bounding-Box:	Size A: 4510 * 4510
	Size B: 4510 * 2250
	Size C: 2250 * 2250
	Size D: 1490 * 2250
	Free size: program generated

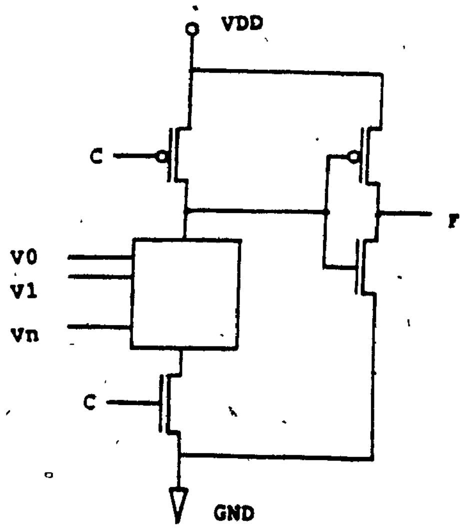
Name: Logic

Logic-Symbol:



Function: Generate boolean expression mask geometries of Hf, Lf, C1, C2, Hfunction and Lfunction which are the control parts of primitive "Comparator" and "Switchbox4".

Circuit-Diagram



Bounding-Box On-line generated.