## NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

## AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, tests publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.



Canada

# A Study of System Design Techniques in VLSI

Ramachar N. Prasad

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

June 1988

ISBN 0-315-44841-5

# ABSTRACT

A Study of System Design Techniques in VLSI

Ramachar N. Prasad

Some classes of architectures and system design approaches are well suited for VLSI due to its physical properties and high level of integration. In this thesis two design examples, each using a different design approach favoured by VLSI, are presented. The first design considered is a synchronous design of a stack. The stack design is discussed using the systolic design approach with emphasis on many issues of practical importance. The second design considered is in the domain of self timed systems. Using the newly developed request serialization algorithm, a distributed synthesis procedure for self timed general arbiters is discussed. System properties such as deadlock and fairness are proven for the self timed design approach. Also, a technique to extend the request serialization algorithm to support expandability of a general arbiter is discussed. Finally, a synchronous deadlock free algorithm suitable for general arbiter synthesis is discussed and is compared with the request serialization algorithm for its performance.

CONTENTS

# LIST OF FIGURES

## LIST OF TABLES

# CHAPTER 1

## INTRODUCTION

VLSI is a medium which allows a large number of switching devices to be integrated on to a single silicon wafer. This sheer size coupled with a variety of the physical properties of the silicon medium pose many challenges to VLSI system designer. The principal concerns of a VLSI designer are complexity management[30,34] and architectural ramifications due to properties of the VLSI medium. According to Moore, VLSI complexity is a combined effect of the technology growth rate and the development of system design techniques. Statistically stated[10], while for the past one decade the integration level has doubled every 1-2 years, the investment on design techniques has doubled every three years, thus leading to increased system design cost and complexity. The architectural ramifications in VLSI are due to the physical properties such as communication delay, switching speed and device scaling. Thus, with the advancement in technology for a given problem, a VLSI designer is posed with the challenge to arrive at a design solution suitable for VLSI without excessive cost and performance penalties. To obtain an optimal solution it is essential to understand the properties of the VLSI medium as well as the different problem solving methods applicable for VLSI. In the

following sections a brief discussion of the complexity management techniques, properties of the VLSI medium and design techniques suitable for VLSI are discussed.

## 1.1 Complexity management

Complexity is a part of any large system, hardware or software. In many aspects complexity management in VLSI is similar to that of software, thus many of the software complexity management results are applicable to VLSI also. However, considerations such as placement and routing distinguish VLSI systems from software systems. The two complexity management schemes used in VLSI are partitioning and hierarchical abstraction. In practice a particular design may use one or both of the approaches during the design process.

Partitioning is the technique of subdividing a large problem into smaller ones of manageable sizes. While, there is no unique way of partitioning a given system into sub-systems, often functional decomposition or flow based decomposition is used. Functional decomposition is a divide and conquer technique, wherein the system is decomposed using functionality of the modules. An advantage of such an approach is the possible generality of the modules. In this thesis functional decomposition is used during the design process in chapter 2.

The flow based technique is an alternative method of decomposition where a circuit is decomposed using the flow of data or control. In this method, decomposition is achieved by considering a system as transforming inputs to the desired form of outputs. This conceptualization leads to a natural decomposition of the system into input circuit, output circuit, and transformation circuit. This approach will not lead to economical hardware in general, but is useful in applications which emphasize communication between blocks such as circuit implementation of petri net[29]. While using the above approach, specific emphasis should be given to issues such as synchronization, buffering of intermediate results, and flow control.

An alternative approach to complexity management is the hierarchical abstraction approach. The concept of abstraction, by hiding all the internal details, allows one to focus on a few essential pieces of information at a design phase. For example, logic gate abstraction hides all the different transistor implementations of the gate. Register transfer level abstraction is used to represent the data operations while hiding all the timing details. Instruction set processor description is used to hide all details except the set of primitive operations. Using the concept of abstraction, system complexity is made manageable by separating the system issues into different layers of a hierarchy. This approach is called hierarchical

abstraction.

## 1.2 VLSI properties

VLSI is not just a medium which allows a high degree of integration, but also has some significant physical properties which strongly influence the system design approaches. These properties are often called the VLSI constraints. In order to derive maximum benefit from VLSI, it is essential to balance advantages and limitations of the VLSI medium. The limitations imposed by the VLSI medium can be grouped under the following categories:

(i) Regularity

(ii) Planarity

(iii) Scaling

Reduction in fabrication and design cost can be obtained by keeping the number of different types of components in the system to a minimum. This imposes a constraint of regularity on the architecture and design. Regularity in a system can be achieved by using structures like library cells, PLA, ROMs at the circuit level or by using repeated structures while building systems. Systems displaying the latter property are called homogeneous systems.

Planarity is a constraint dealing with pin limitation,

I/O band width and silicon utilization problems. The planarity constraints can be classified under the following categories[36]:

(i) Perimeter problem

(ii) Graph smashing problem

The perimeter problem refers to issues related to input-output. The number of wires leading off a chip is proportional to perimeter of the chip. Also, the amount of data input-output at any time instant is proportional to the number of input-output outlets. Consequently, a large difference in processing rate and data availability rate could lead to under utilization of silicon as discussed in [8].

Graph smashing problem is an abstraction of the layout problem in VLSI. Consider a circuit abstracted as a graph with nodes representing gates(modules) and edges representing their interconnecting wires. A planar embedding of such a graph could occupy a wide silicon area due to minimum spacing requirement between wires and restrictions on the number of wires crossing each other. This makes wires the dominant component in a layout, thus reducing the chip area used by active components. This problem becomes more acute if we consider complex systems or when graph vertices represent processing elements and each edge represent a collection of interconnection wires. As a result of the above, VLSI favours architectures which

exhibit regular and local communication.

Device scaling is performed in VLSI with the primary objective of increasing component density. But, scaling down of feature size not only increases component density but also affects physical characteristics and parameters that describe switching devices, wires. The most important effect of scaling on a VLSI design is its effect on timing aspect of the design. Due to scaling, the delay along a line increases quadratically and the switching speed reduces in linear proportion to the scaling factor. One way of representing the comprehensive effect of scaling is by considering length of the wire that can be driven in a unit time(one transit time). Consider the following example[33] for a 5 micron technology, transit time of 0.25 nsec and a scaling factor of 10. The following table gives the length of the wire, before and after scaling, that can be driven in one transit time.

| Technology | Polysilicon | Diffusion | Metal |
|---|---|---|---|
| 5 micron | 0.3 | 0.5 | 17 mm |
| 0.5 micron | 0.01 | 0.02 | 0.5 mm |

VLSI favours architectures which do not exhibit global transmission of signals or lengthy interconnection as they introduce time delay and require large drivers.

## 1.3 System design approach for VLSI

An ideal design approach to VLSI system is to apply complexity management techniques for a desirable design approach. In general, system architectures which exhibit locality and regularity in communication and a high degree of concurrency in operation are preferred for VLSI implementation. Two design techniques which exhibit some or most of these characteristics that are used in this thesis are discussed below.

Systolic networks[16] are highly suitable for VLSI design due to their property of homogeneity, uniformity, locality and the high performance due to concurrency in system operation. Details of systolic network design methods can be found in Kung[17]. A systolic network can be implemented using synchronous or asynchronous design technique. In this thesis we examine synchronous implementation of systolic network. Asynchronous implementation of systolic networks is used in fault tolerant applications as discussed in [17]. However, the absence of a general procedure to transform any given algorithm to systolic algorithm or to design systolic network for all problem classes makes them special purpose architectures.

In general, asynchronous design technique has wider

applicability in VLSI. With properties such as immunity to communication delay, the asynchronous design technique finds application even when strict locality and regularity in communication cannot be maintained in a system. Also, the absence of clock in an asynchronous systems provides a degree of freedom during placement of modules and thus layout design. Asynchronous systems provide high performance by exploiting fine grain concurrency and by operating at average speed. In this thesis we consider self timed synthesis of asynchronous systems.

Self timed systems, a type of asynchronous systems, can be defined recursively as either a self timed element or a legal interconnection of self timed systems. This organization principle is a natural definition of modular design and thus assists in complexity management also. Self timed systems are designed using the concept of equipotential region and self timed signalling[21]. Equipotential regions are areas of a chip where the communication delay can be neglected. But, communication delay for signals crossing equipotential regions are not negligible and thus self timed signalling is used during such communication. The two types of self timed signalling that can be used during the design of a self timed systems are four cycle and two cycle signalling. The basic properties of these signalling methods can be characterized as non-communication interference and non-computation

interference. Though either of the self timed signalling protocols can be used in a design, four cycle signalling is best suited for short distance communication, due to its return to zero property while two cycle signalling can be used also for long distance communication. Details of the self timed design methodology can be found in Mead[21] and Chapiro[3].

## 1.4 Thesis scope

In this thesis the application of synchronous systolic network and self timed system design approaches are examined through examples of stack and general arbiter respectively. While the systolic stack design provides a complete hierarchical exposition to the VLSI design process, the arbiter design addresses salient levels in the design hierarchy. Finally, to analyze the effect of asynchronous and synchronous design approaches on design complexity and performance, a case study of synchronous implementation of general arbiter is discussed.

The thesis is structured as follows. Hierarchical design of one slow systolic stack is presented in chapter two. Chapter three deals with verification and analysis of one slow systolic stack. A self timed design technique for general arbiters is presented in chapter four and applications of such an arbiter is also examined in that

chapter. A case study of synchronous and asynchronous design is discussed in chapter five. Conclusion and topics for further research are considered in chapter six. Material pertaining to logic simulation, circuit simulation and circuit diagrams are presented in the three appendices.

# Chapter 2

# SYSTOLIC STACK

## 2.1 Introduction

The stack is a fundamental data structure used in computer hardware and software. In this chapter we discuss synchronous, systolic implementation of stack which is suitable for VLSI. The advantage of using a systolic implementation for stack becomes evident when various different implementations of stack are examined. Some of the known implemetations of stack are:

(i) Array(RAM) with a top of stack pointer

(ii) Universal shift register

(iii) Ripple structure

Array implementation, mostly used in software, is not suitable for VLSI for the following reasons:

(i) Size of the top of stack pointer register varies with size of the stack

(ii) Both the data and control signals need broadcasting, thus resulting in driver stages whose capacity depends on size of the stack.

An architecture suitable for VLSI implementation should eliminate broadcasting of signals while maintaining constant time operation. The universal shift register and ripple architecture exhibit some of these desired properties.

While the universal shift register implementation eliminates global broadcasting of data signals, it still requires broadcast of control signals such as push and pop. The ripple implementation eliminates all global broadcasting of signals at the cost of operational speed. The operational speed of ripple implementation is linearly proportional to number of elements in the stack.

The systolic architecture for stack combines features of ripple and universal shift register implementation to provide an implementation with characteristics such as no global broadcasting, constant time operations along with added advantage of elegant handling of boundedness. Unlike many other systolic networks, dataflow in a systolic stack cannot be specified a priori, thus categorizing them as control flow network. One of the problem facing designers of such control flow networks is the non-existence of an established design procedure. The available design techniques such as fixed-point approach of Chen[5] or Moldavan's transformation technique[22] are applicable only to the design of dataflow networks. Thus, the design technique of control flow networks has remained ad hoc to a large extent. In this chapter, the design of a fast systolic stack is discussed. This design has been successfully implemented using the CMOS-1B process of Canadian Microelectronic Corporation. Before presenting the design, we discuss the systolic stack algorithm used in the

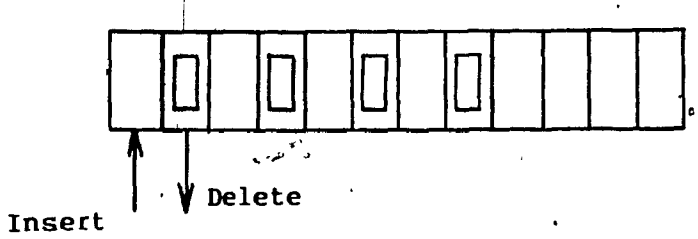
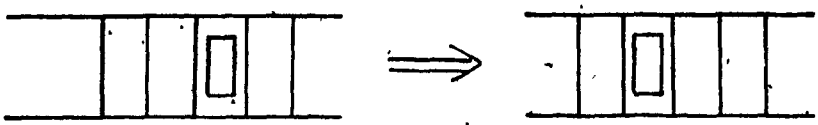



Figure 2.1 Guibas stack organization

(a) Rule 1



(b) Rule 2

Figure 2.2 Guibas rewrite rules

design.

## 2.2 Four slow stack

Guibas[14] proposed the first systolic algorithm for stack implementation and the algorithm used in this thesis is a time improved version of Guibas algorithm. Before presenting the improved algorithm, Guibas algorithm is presented first. The Guibas stack is a linear array of cells with each cell of the stack being in either occupied or empty state. The external operators such as push, pop are applied to cell 0 and cell 1 respectively as shown in figure 2.1. All other cells of the stack respond to internal commands, which are generated as a result of rewrite rules of the stack. The two rewrite rules of the Guibas stack are indicated in figure 2.2. The working of Guibas stack is illustrated in figure 2.3. A study of Guibas stack indicates that a minimum of three idle cycles, called skip cycles, are required between any two operators to ensure proper operation of the stack. The skip cycles are used by the stack to redistribute its entries so that the stack is ready for next push or pop operation. Thus, as a result of the three skip cycles(but, for the stack overflow or underflow condition) it is always ensured that cell 0 is empty(ready for a push operation) and cell 1 is occupied(ready for a pop operation). As Guibas stack requires three skip cycles between any two operators, it is

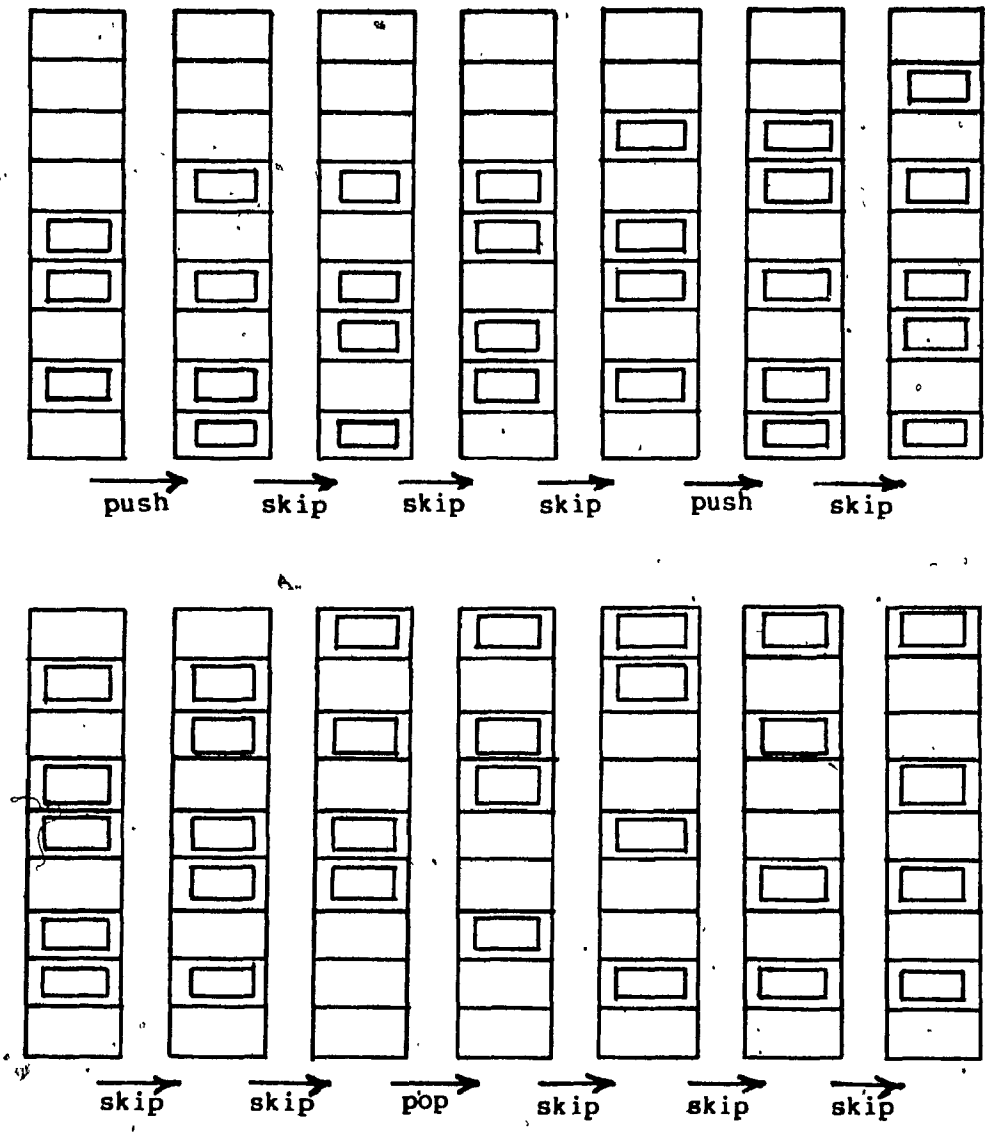



Figure 2.3 Guibas stack operation

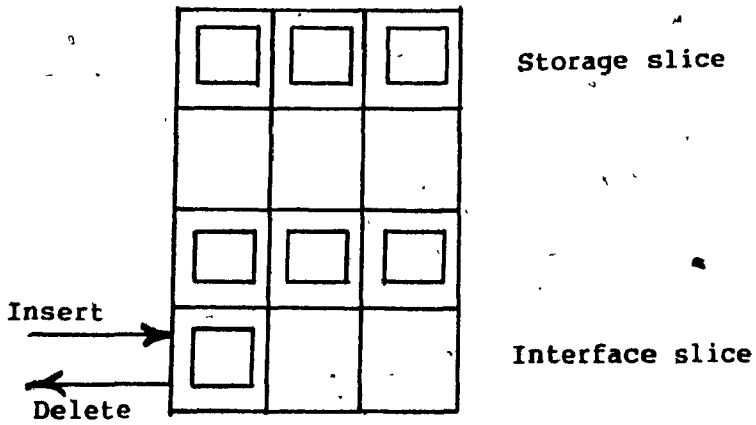



Figure 2.4 One slow stack organization

also called a four slow stack.

## 2.3 One slow stack

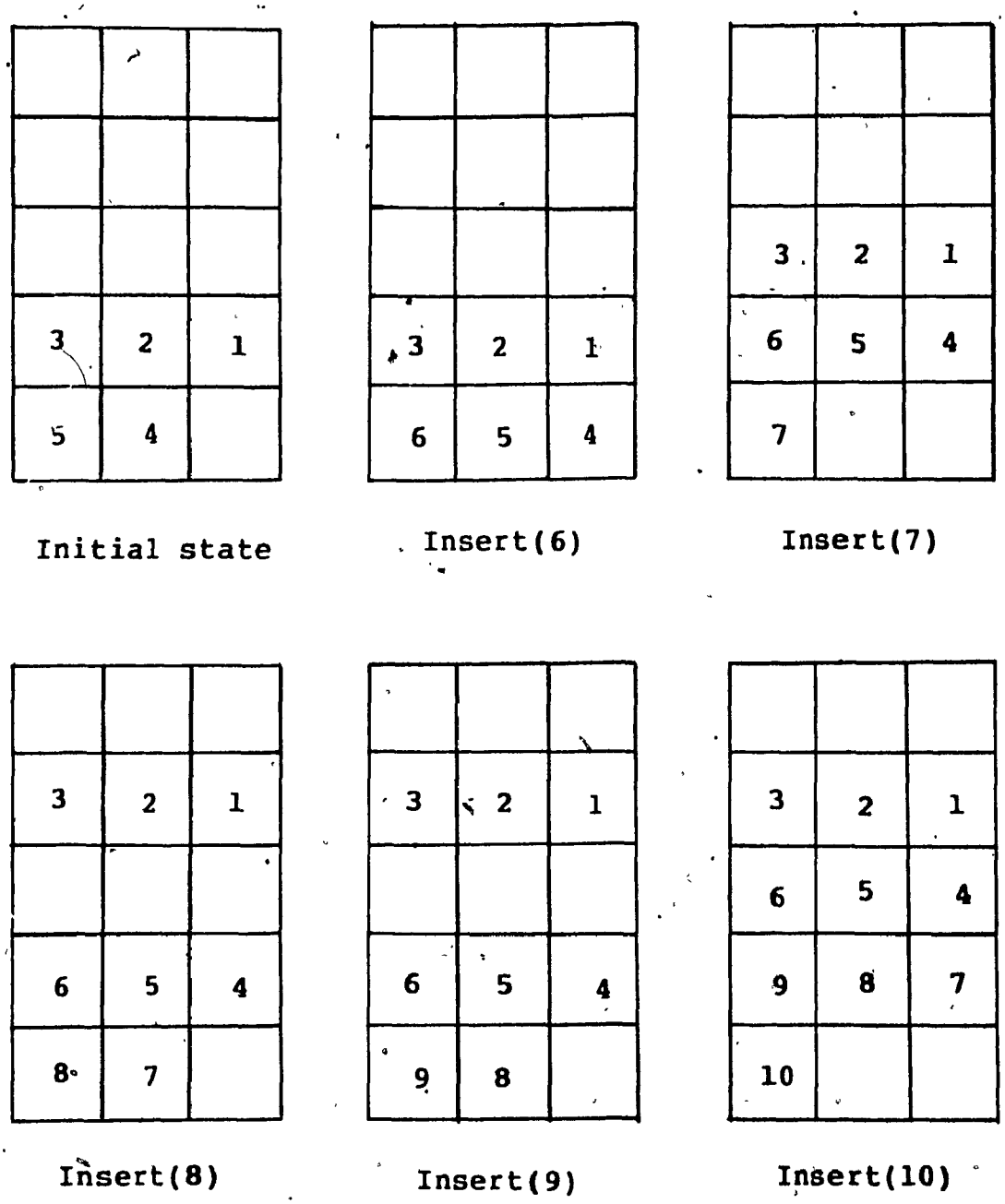
Guibas also presented a method for improving operational speed of the stack by considering state information of non-neighbouring cells. However it is difficult to extend this technique to obtain a one slow stack - a stack which allows a push or pop operation every cycle. In this chapter, a one slow algorithm using a local compaction technique is presented. The local compaction of stack entries is based on the concept of buffering. In computers the concept of buffering is used to match difference in speed between two communicating sections of a system. Under this interpretation, the speed difference in the stack occurs between the interface cell of the stack and the host(external environment). While host can issue an operator(push or pop) every cycle, the interface cells require three idle cycles between commands to be able to accept a new command and data. This mismatch in speed of a four slow stack can be eliminated by establishing a buffer, of capacity three, at the host interface.

The organization of the one slow stack is shown in figure 2.4. This stack is a linear array of slices, where each slice comprises of three stack elements. Each slice communicates with its left or right neighbour slice only,

thus providing a regular and simple interconnection structure. The slices in the stack are labelled from 0 to (n-1) and based on their functionality slice 0 is referred to as the interface slice and the slices from 1 to (n-1) are referred to as storage slices. The host interfaces to the stack through the interface slice. The external operators such as push, pop are seen only by the interface slice. The data transfer between the interface slice and the host is one element at a time, while between slices data is transferred fully in parallel, three elements at a time. Thus, a packet of three data elements are created and destroyed in the interface slice, giving a distinct functionality to the interface slice.

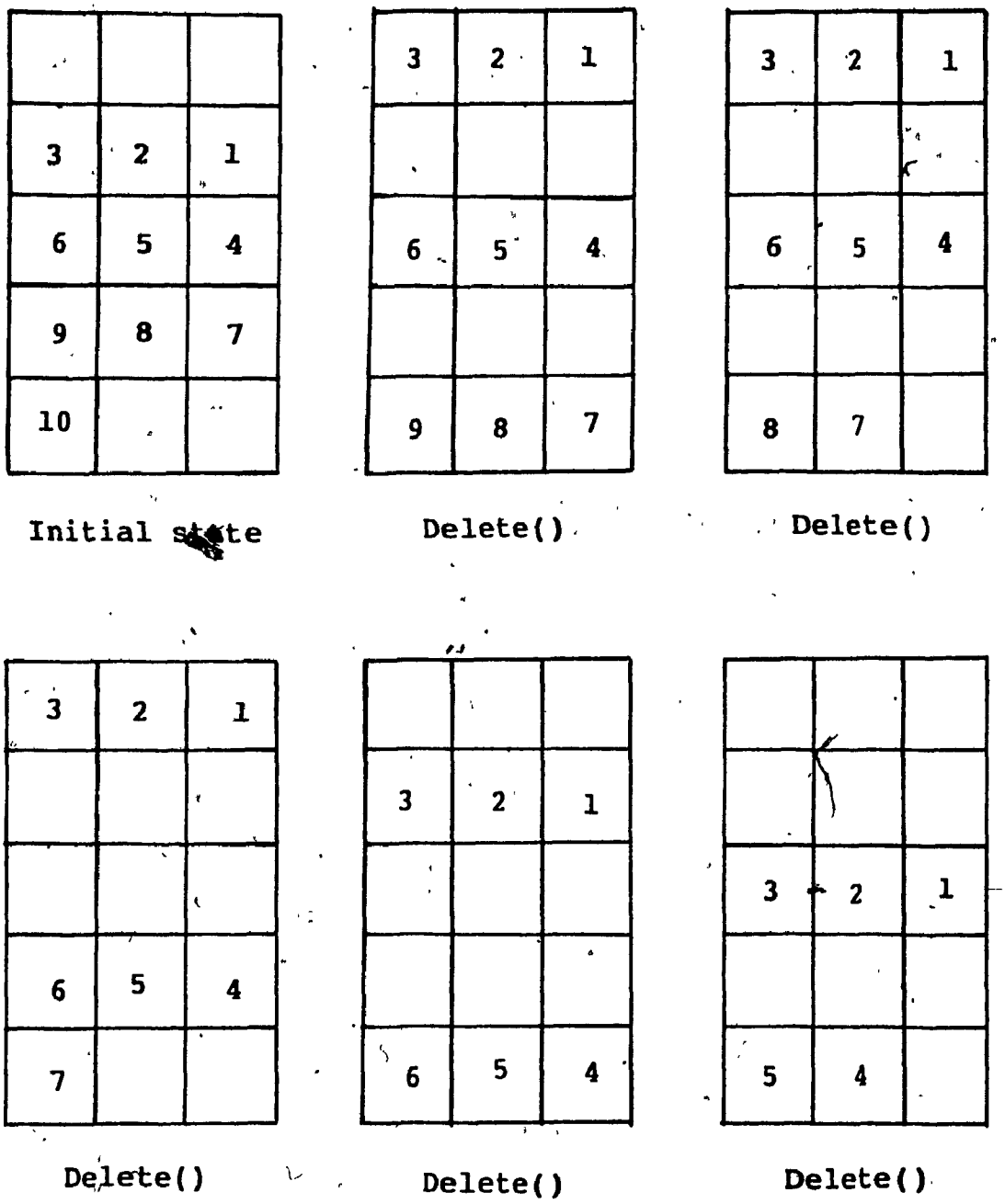
There are many differences and similarities between the Guibas stack and the one slow stack. The similarities being that all the storage slices, like the cells of Guibas stack, execute the two rewrite rules of Guibas. Consequently the redistribution of data in storage slices follow the same pattern as in the Guibas stack. Similar to Guibas stack the one slow stack also assumes a permanently occupied slice at the end of the stack to prevent stack entries from moving out of bounds.

One slow stack differs from the Guibas stack in two aspects, namely state definition of slice and data transfer characteristics. A storage slice is in occupied state when




(i) Insert operations

Figure 2.5 One slow stack operation




(ii) Delete Operations

Figure 2.5 One slow Stack operation(Cont.)

all the three cells in the slice contain data. A storage slice is in an empty state if none of the cells in the slice contain data. However, it should be noted that a storage slice can either be in an empty or occupied state only. Unlike the storage slice, the interface slice of the one slow stack can be in one of the four states: empty, one-empty, two-empty, occupied. These states of the interface slice, are defined based on the occupancy of the cells in the interface slice. Thus data in the interface slice varies smoothly as compared to storage slice. Unlike Guibas stack the interface slice of the one slow stack supports two types of data transfer - a parallel(three element at a time) data transfer with slice 1 and a serial data transfer (one element at a time) with the host.

Operation of the one slow stack is indicated in figure 2.5. When a push operation is performed, with slice 0 not in occupied state, contents of slice 0 are shifted one place to the right. Similarly a pop operation shifts contents of slice 0 one place to the left. However, when slice 0 is in occupied state a push operation will move, in parallel, contents of slice 0 to slice 1 and the new element is pushed into slice 0. An analogous situation to the above for a pop operation occurs when slice 0 is in two empty state. In such a state, a pop operation results in popping of elements from slice 0 and moving contents of slice 1 to slice 0, in parallel. This distinctive operation of the interface slice

is made possible by associating a separate set of rewrite rules to slice 0. It should also be noted that unlike Guibas stack, under one condition, both slice 0 and slice 1 will perform simultaneous parallel transfer of data to its immediate right neighbour as indicated in figure 2.5.

## 2.4 Hierarchical stack design

The design hierarchy of the systolic stack considered in this chapter is divided into following levels of abstraction.

(i) Behavioral level

(ii) Architectural level

(iii) Logic design level

(iv) Circuit design level

(v) Layout level

### 2.4.1 Behavioral level

The behavioral level of the hierarchy is a black box specification layer dealing with traces of interface signals. Behavioral layer, by defining only the external observable behaviour of a system leaves the flexibility of architectural and implementation variations to lower layers in the abstraction. The behavioral specification used in this section comprises of two parts namely, the syntatic and semantics. Syntatic part of the specification describes

names of the access operators and type associated with input and output data. The semantic specification, expressed as a set of assertions, gives all possible traces of the access operators and the value of data that can be derived. The behavioral specification comprising of the above two parts, adopted from[19], for the one slow stack is given below:

NAME : 1-Slow Synchronous n-stack

Syntax : Push : integer(input data)

Pop

top: integer(data read-out)

skip

Semantics :

1.1 legal ($push^m$)$\leftrightarrow 0 \leq m \leq n$

1.2 $0 < m \leq n \rightarrow push^m.pop = push^{m-1}$

1.3 $0 < m \leq n \rightarrow push^m.top = push^m$

1.4 $0 < m \leq n \rightarrow value(push^m.top) = a_m$

1.5 $0 \leq m \leq n \rightarrow push^m.skip = push^m$

where $push^m = push(a_1)\ push(a_2).....push(a_m)$

2.4.2 Architectural level

The architectural level of abstraction describes the architecture of the one slow systolic stack using the Register Transfer Language(RTL). RTL is used to describe the flow of data and control in the stack. An RTL description for the one slow stack discussed in section 2.3 is given below in a program form.

```
Procedure cycle
:slice 2 to slice(n-1) operation
For each i in {2,3,....,n-1} par do
  if f_i∧f_{i-1}∧f̄_{i+1} then
       R_{i+1} = R_i
       f_{i+1} = true: f_i = false:
  if f_i∧f̄_{i-1}∧f̄_{i-2} then
       R_{i-1} = R_i
       f_{i-1} = true: f_i = false:
parend
:slice 1 and slice 0 operation
if push(a) then case state0 of
  1: R_0 = a.R_0: state_0 = 2 /* a.R_0 is 'a' added to R_0 */
  2: R_0 = a.R_0: state_0 = 3: f_0 = True
  3: R_1 = R_0: R_0 = a: f_1 = true: f_0 = False: state_0 = 1
if pop then case state_0 of
  1: R_0 = R_1:f_1 = False: f_0 = True: state_0 = 3
  2: R_0 = L(R_0): state_0 = 1 /* L(R_0) is R_0 shifted left */
  3: R_0 = L(R_0): state_0 = 2: f_0 = False
if skip then null action
endcycle
```

In the above program $f_i$ indicates state of slice i and $R_i$ indicates data present in slice i. The above representation is adopted from Li[19]. However, alternative representation like linked module abstraction of Stefik[37] can also be used to represent an architecture.

### 2.4.3 Logic design level

The logic design level of abstraction decomposes the one slow systolic stack into modules and each of these modules are described to the detail of its underlying boolean function. The one-slow systolic stack is partitioned into modules using the functional decomposition technique. The functional modules thus obtained are listed below:

(1) Data cell for each storage slice

(2) Rewrite control logic for storage slice

(3) Data cell for interface slice

(4) Rewrite control logic for interface slice

(5) Flag bit logic for storage slice

(6) Flag bit logic for interface slice

(7) Stack full module

(8) Stack empty module

(9) Interface module

During logic design a flag bit is used for each storage slice to indicate the empty or occupied state of the slice and three flag bits are used to indicate state of the interface slice. Function of the modules (1) to (6) are self explanatory. Additional functionality like module (7) and (8) is introduced at the logic design level to maintain boundedness of the stack. The interface module is used for the purposes of electrical interface and synchronization of

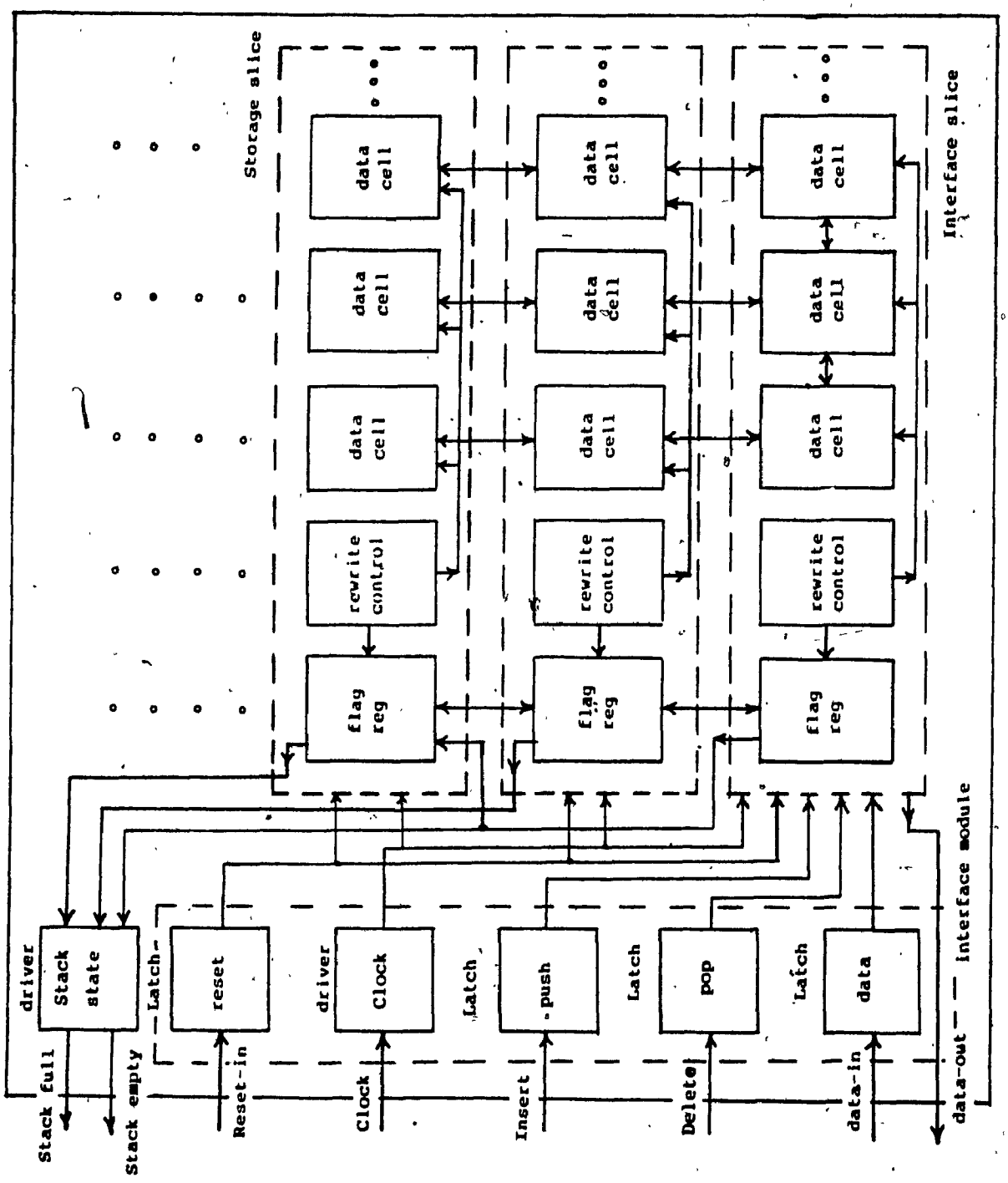



Figure 2.6 One slow stack block diagram

input signals. Figure 2.6 gives the block diagram of the one slow systolic stack indicating the functional modules. Logic design for each of the functional module is described below. While logic design can be derived from the architectural specification of section 2.4.2, for improved clarity a detailed description of the design is given below. It should be noted that skip is not an explict external operator at the logic design level. However, the skip operator is derived from the absence of external operators like push and pop.

### 2.4.3.1 Storage slice design

Each storage slice comprises of three data cells, a flag bit indicating the state of the slice and control logic implementing the rewrite rules of the storage slice.

#### Control Logic

The rewrite rules used in the storage slices are the same as Gurbas rewrite rules(section 2.2). Based on semantics of the rewrite rules, rewrite rule 1 is implemented as a push signal and rewrite rule 2 is implemented as a pop signal in the control logic. With the positive logic convention, the two control signals, for slice i, can be represented as

$$push_i = f_{i-1}.f_i.\bar{f}_{i+1}$$

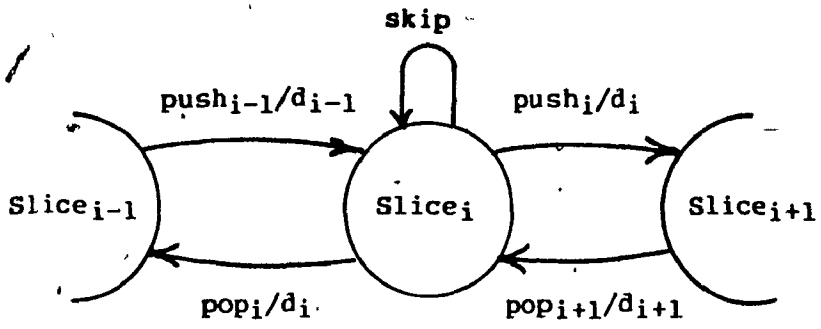



Figure 2.7 Storage slice state diagram - Datapath

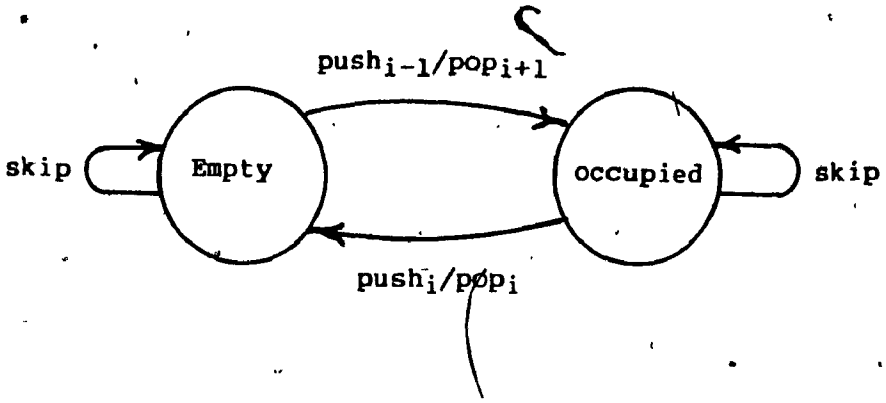



Figure 2.8 Storage slice state diagram - Flag bit

$$pop_i = \overline{f}_{i-2} \cdot \overline{f}_{i-1} \cdot f_i$$

Data cells

The state diagram representing data flow under the influence of the different control signals is shown in figure 2.7. During the design of storage slice it should be kept in mind that a storage slice transfers, in parallel, all the three data elements during a push or pop operation. From the state diagram, the logic for datapath of slice i can be described as

If $push_{i-1}$ then $D_i' = D_{i-1}$

If $pop_{i+1}$ then $D_i' = D_{i+1}$

If neither then $D_i' = D_i$

Hence,

$$D_i' = push_{i-1} \cdot D_{i-1} + pop_{i+1} \cdot D_{i+1} + \overline{push}_{i-1} \cdot \overline{pop}_{i+1} \cdot D_i$$

Where, $D_i'$ is the value assigned to $D_i$ after a clock pulse. The feedback path in the datapath necessitates use of an edge triggered or master-slave type of data register. In this version of the design a master-slave flip-flop is used due to its implementation simplicity.

Flag logic

State diagram for the flag bit of storage slice i is shown in figure 2.8. Using the state diagram, the logic for the flag bit can be described as

If $push_{i-1}$ then $f_i'$ = '1' (occupied)

If $pop_{i+1}$ then $f_i'$ = '1'

If $push_i$ or $pop_i$ then $f_i'$ = '0' (empty)

If none of the above then $f_i' = f_i$

Hence,

$$f_i' = push_{i-1} + pop_{i+1} + f_i.\overline{push_i}.\overline{pop_i}$$

To enable stack initialization a reset type master-slave flip-flop is used for the flag register. Block diagram of a storage slice comprising of the above three modules is shown in figure 2.9.

2.4.3.2 Interface slice design

Unlike Guibas stack, the design of interface slice in the one-slow stack differs from that of the storage slices due to the use of separate set of rewrite rules and state convention. The host commands like insert and delete are assumed to be synchronous signals during interface slice design. The interface slice comprises of control logic implementing the rewrite rules, data cells and logic for storing state of the interface slice(flag logic).

Control logic

The rewrite rules of the interface slice is implemented as the set of following signals:

(1) Shift left

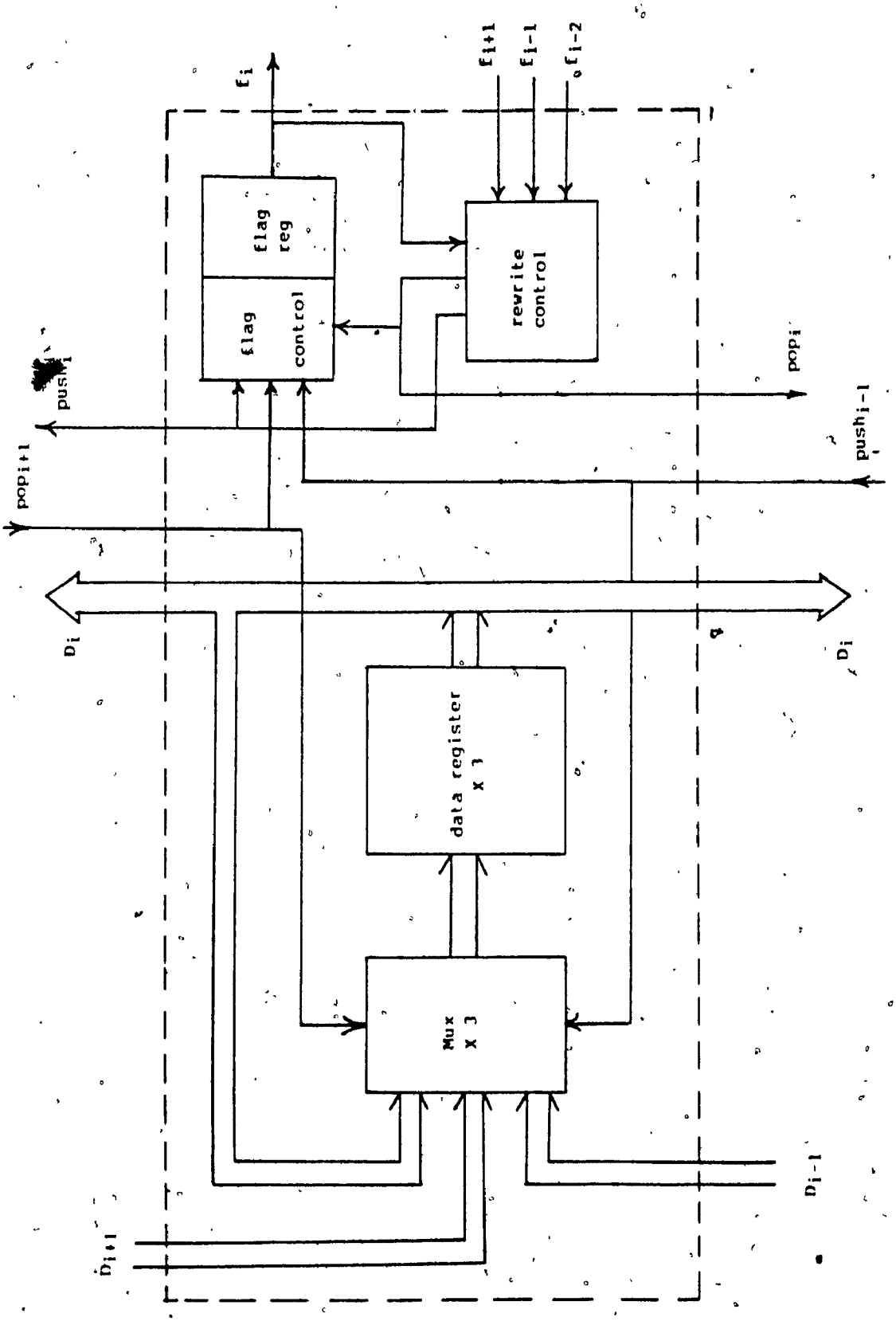



Figure 2.9 Storage slice block diagram

(2) Shift right

(3) $push_0$

(4) $pop_0$

Logic for the above set of signals is described below:

If insert and not stack-full then shift right(SR)

If delete and not stack-empty then shift left(SL)

If in two empty state and delete then $pop_0$

If in occupied state and insert and not stack full-then $push_0$

Hence,

$SR = push.\overline{sf}$

$SL = pop.\overline{se}$

$push_0 = push.f_0.\overline{sf}$

$pop_0 = pop.e_2$

Where, push and pop are the host commands - insert and delete respectively. To preserve integrity of the data present in the stack, boundedness of the stack is taken into consideration during design of the control signals for the interface slice.

Data cell

As the data occupancy of the interface slice varies smoothly as compared to the storage slices, the datapath logic is also different from that of the storage slices. The logic for the datapath can be described as

If SR then $D_{0k}' = D_{0k-1}$

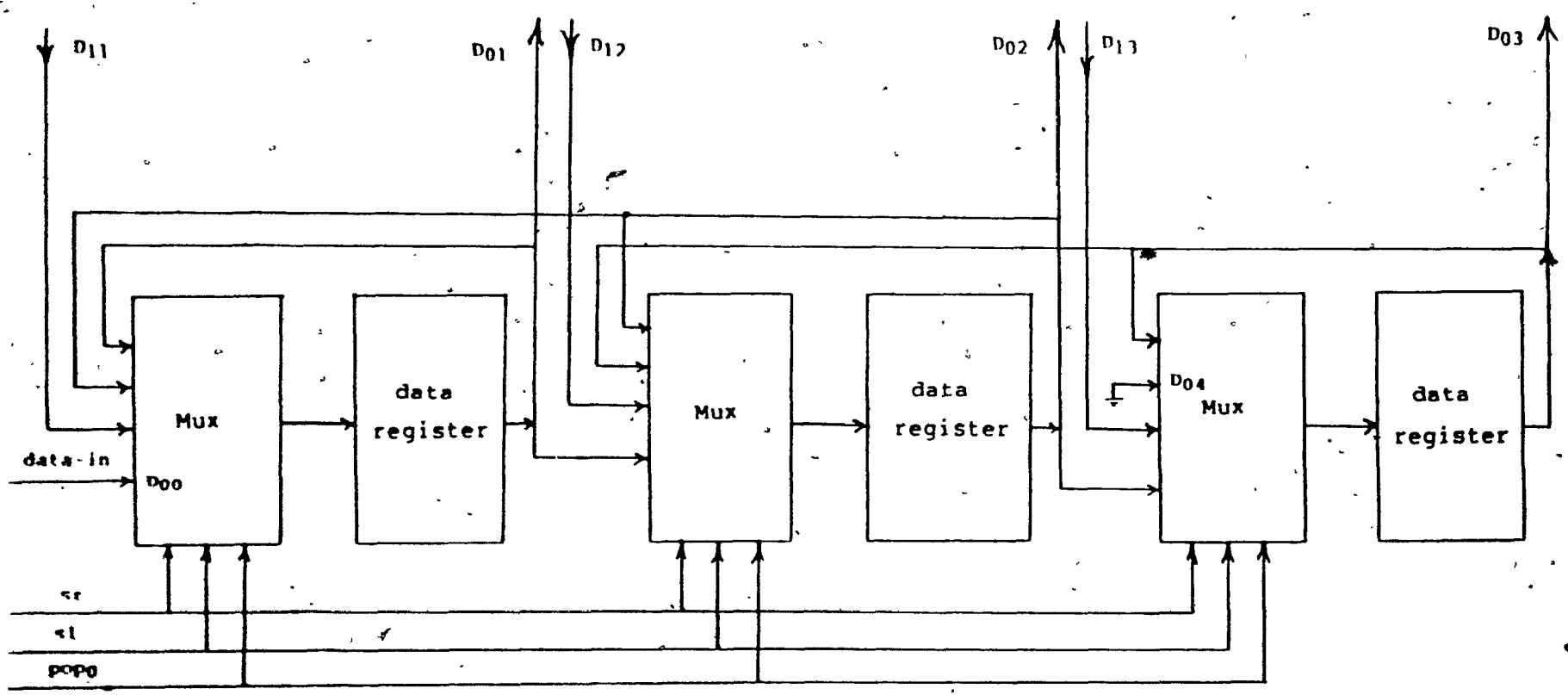



Figure 2.10 Interface slice datapath

If SL then $D_{0k}' = D_{0k+1}$

If $pop_0$ then $D_{0k}' = D_{1k}$

If none of the above then $D_{0k}' = D_{0k}$

Hence,

$$D_{0k}' = SR.D_{0k-1} + SL.D_{0k+1} + pop_0.D_{1k} + \overline{SR}.\overline{SL}.\overline{pop_0}.D_{0k}$$

Where K can take a value from 1 to 3 and K = 0 indicates the host data input and k = 4 indicates a dummy data. Figure 2.10 gives the datapath block diagram of the interface slice.

Flag logic

Unlike storage slices, the interface slice uses three flag bits, one for each datacell, to represent the state. The state diagram for the interface slice is shown in figure 2.11. Logic for the flag bits has been designed using the state machine design techniques and the functions thus obtained are:

$$e_2' = push.\overline{e_2}.\overline{e_1}.\overline{sf} + pop.e_1 + \overline{push}.\overline{pop}.e_2$$

$$e_1' = push.e_2.\overline{sf} + pop.f_1 + \overline{push}.\overline{pop}.e_1$$

$$f_0' = push(e_1.\overline{sf}+sf) + pop.se.e_2 + \overline{push}.\overline{pop}.f_0$$

Where, $e_2$ indicates two empty state, $e_1$ indicates one empty state and $f_0$ indicates occupied state of the interface slice. Figure 2.12 gives block diagram of the interface slice.

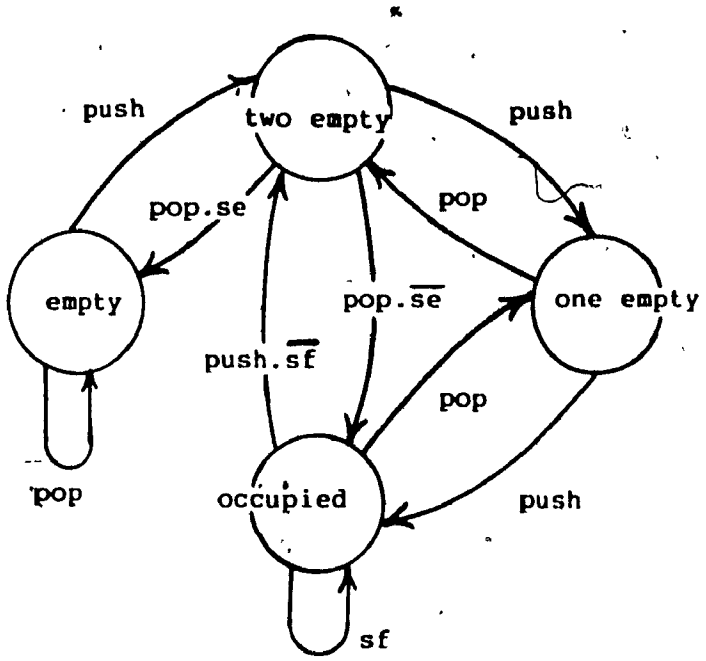



Figure 2.11 Interface slice state diagram

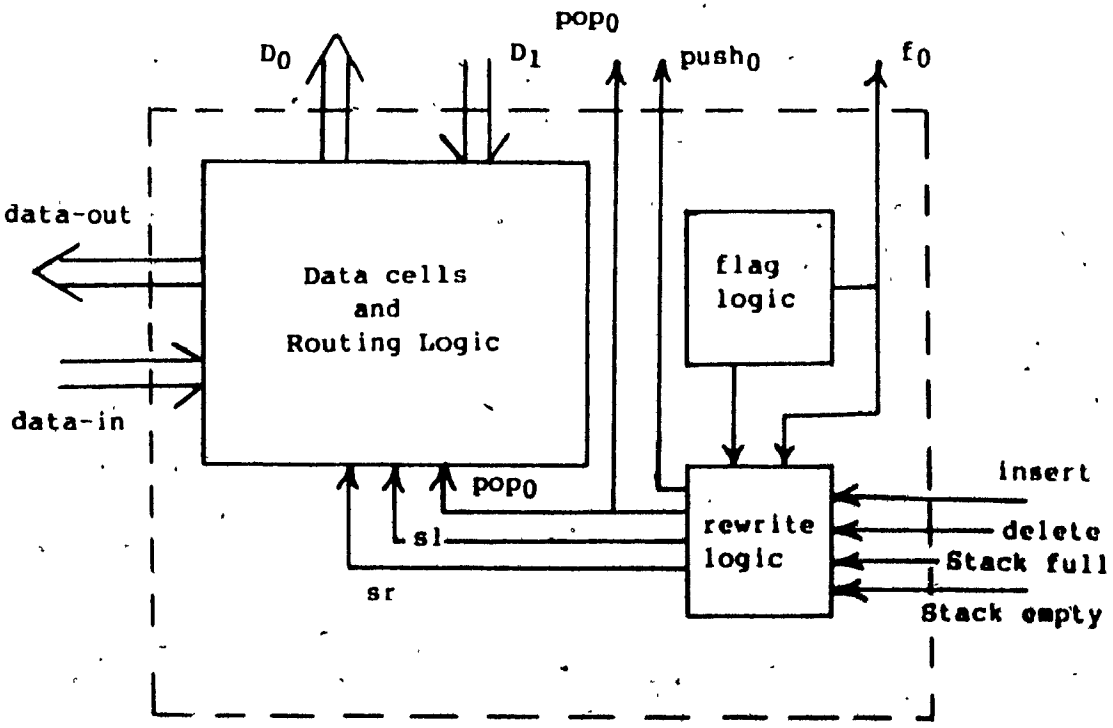



Figure 2.12 Interface slice block diagram

2.4.3.3 Stack state

Two interface signals(output signals), namely stack empty and stack full are generated to indicate state of the stack. The presence of these signals are intentionally leftout in the architectural and behavioral layer to keep the specification simple. Based on the operation of the one slow stack we can conclude that :

(1) Application of Guibas rewrite rule 1 ensures that slice 1 and slice 2 are both occupied, simultaneously, only when all the storage slices in the stack are in the occupied state(stack is full).

(2) Guibas rewrite rule 2 ensures that slice 1 is empty only when all the other storage slices in the stack are also in the empty state.

Stack empty and stack full logic use advance state detection techniques to indicate state of the stack. Logic for the stack full and stack empty signals is given below

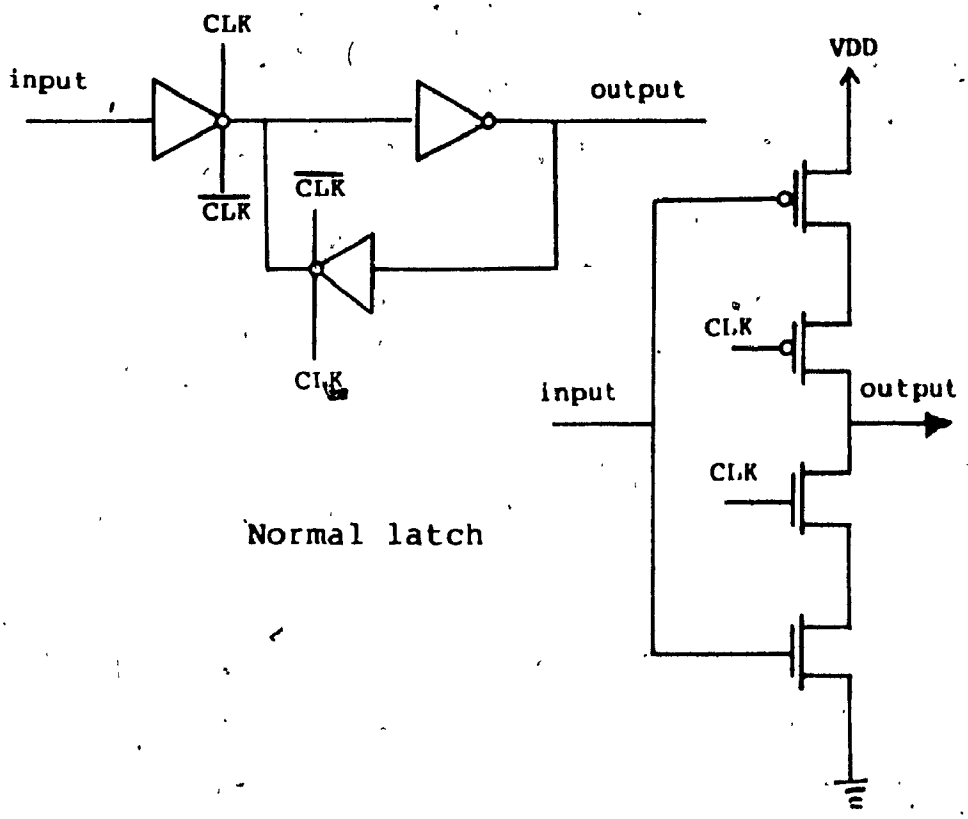
$$se' = pop.f_1.e_2 + \overline{e}_2.se.\overline{push}$$

$$sf' = push.f_1.f_2(e_1+f_0.\overline{pop})$$

The second component in the above functions indicate the ability of the stack to remain in the same state after reaching the stack full or stack empty state.

2.4.4 Circuit design level

The aim of circuit design layer is to select the type



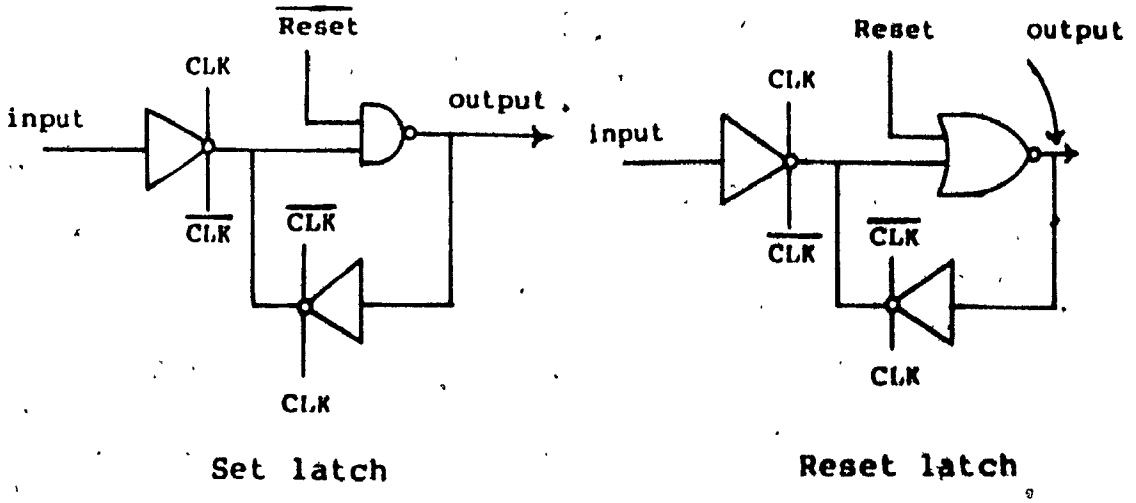
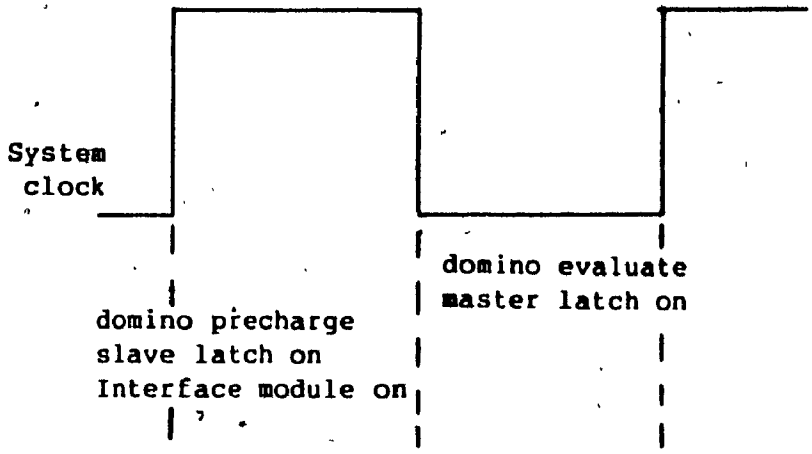




Figure 2.13 Latches for master-slave flip-flop

of CMOS logic structure, select the clocking scheme and define driving capabilities of the different stages.

The selection criteria for logic structure is based on considerations such as simplicity of the logic structure, area, speed and special applications like tristate gates. CMOS provides a number logic structure[38], and in this design both the static and dynamic logic structures are used. Complementary logic is the static logic used in the design while both the domino and clocked logic used in the implementation are dynamic logic structures. Though complementary logic is simple to design, it occupies a large area and hence its use is kept to a minimum. The clocked logic structure is primarily used to implement tristate logic as clocked logic does not provide any area, time advantage over complementary logic. In this design tristate gates are used in the design of master-slave flip-flops and latches as shown in figure 2.13. Domino logic on the other hand provides advantages like smaller area, higher operational speed and glitch free operation. Thus, domino logic is used extensively in the one slow stack design. The two fundamental problems of the domino logic - only un-inverted stages and charge sharing problem are solved as follows. Whenever inverted output is required from the domino block a static inverter stage is used. Any possibility of charge sharing in a domino block is avoided by ensuring that none of the inputs to the domino block



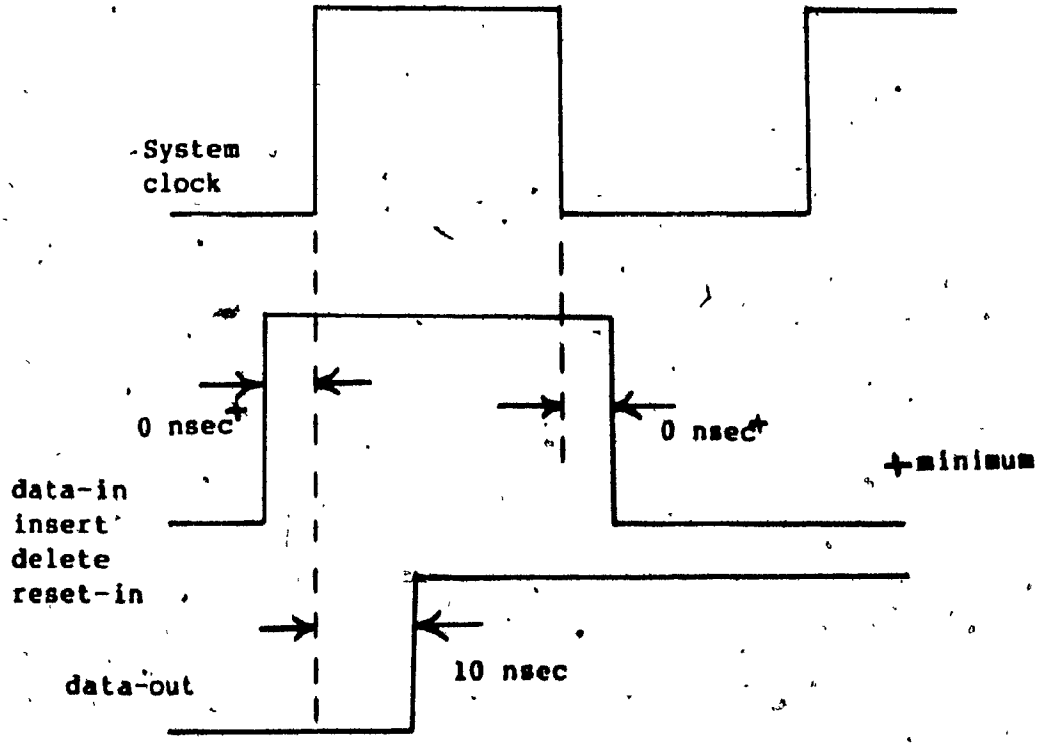




Figure 2.14 One slow stack timing

change during its evaluation period.

The second aspect of circuit design is the selection of clocking scheme. There are many clocking schemes, like single phase, two phase, four phase etc., possible in a VLSI design. The one slow stack implementation makes use of single phase clocking as dynamic structures like domino logic can be designed efficiently using single phase clocking and the single metal layer restriction of the CMOS-1B process will not be severe during clock distribution.

Final aspect of circuit design is to ensure sufficient driving capability of the various modules in the system. Transistor geometry required in the complementary logic is derived based on the series-parallel connection rule[38] and expected load. Transistor sizes of the domino circuit is derived based on precharge and evaluation time requirement. That is, the p-transistor in a domino circuit is designed to provide the required precharge time and the n-transistors in a domino block are designed for the required evaluation time.

Appendix I gives the detailed circuit diagram for some of the modules of the stack. For brevity remaining circuit diagrams are left out. The timing diagram for the one slow stack implementation is shown in figure 2.14.

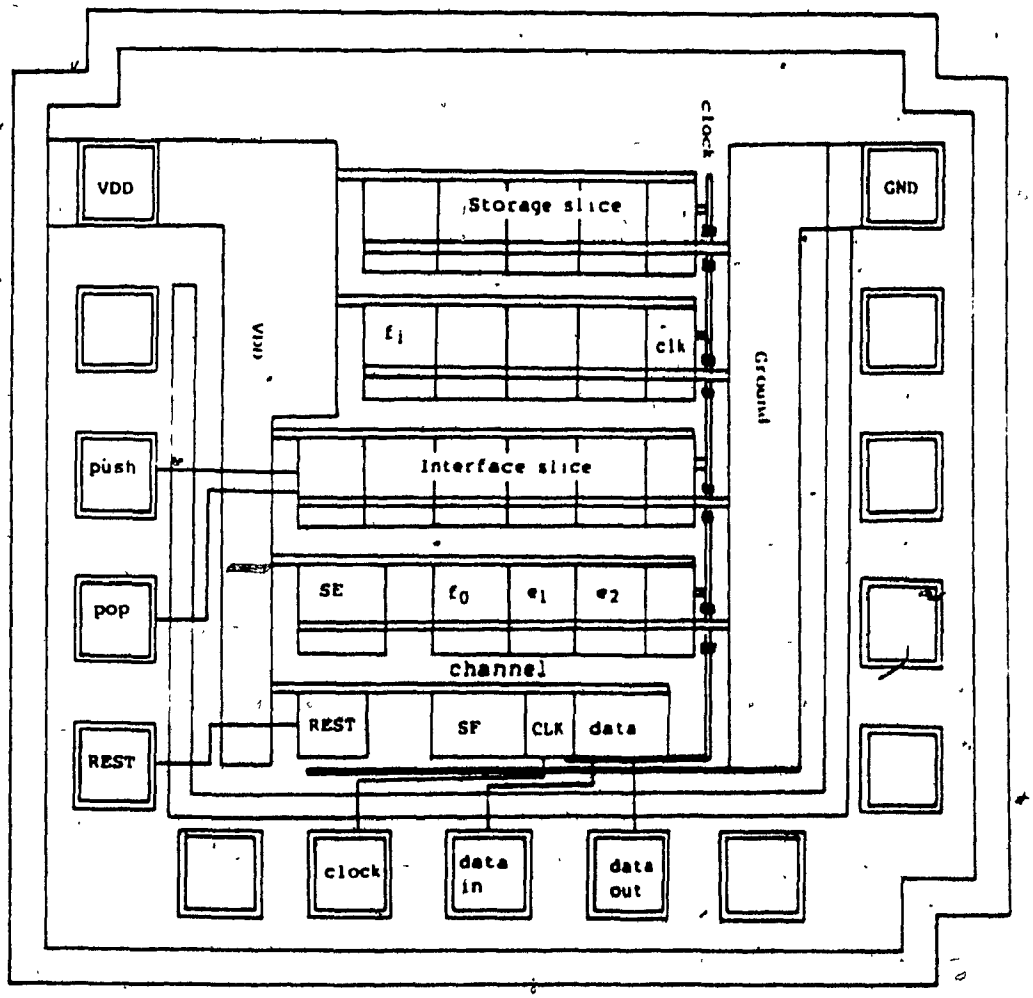
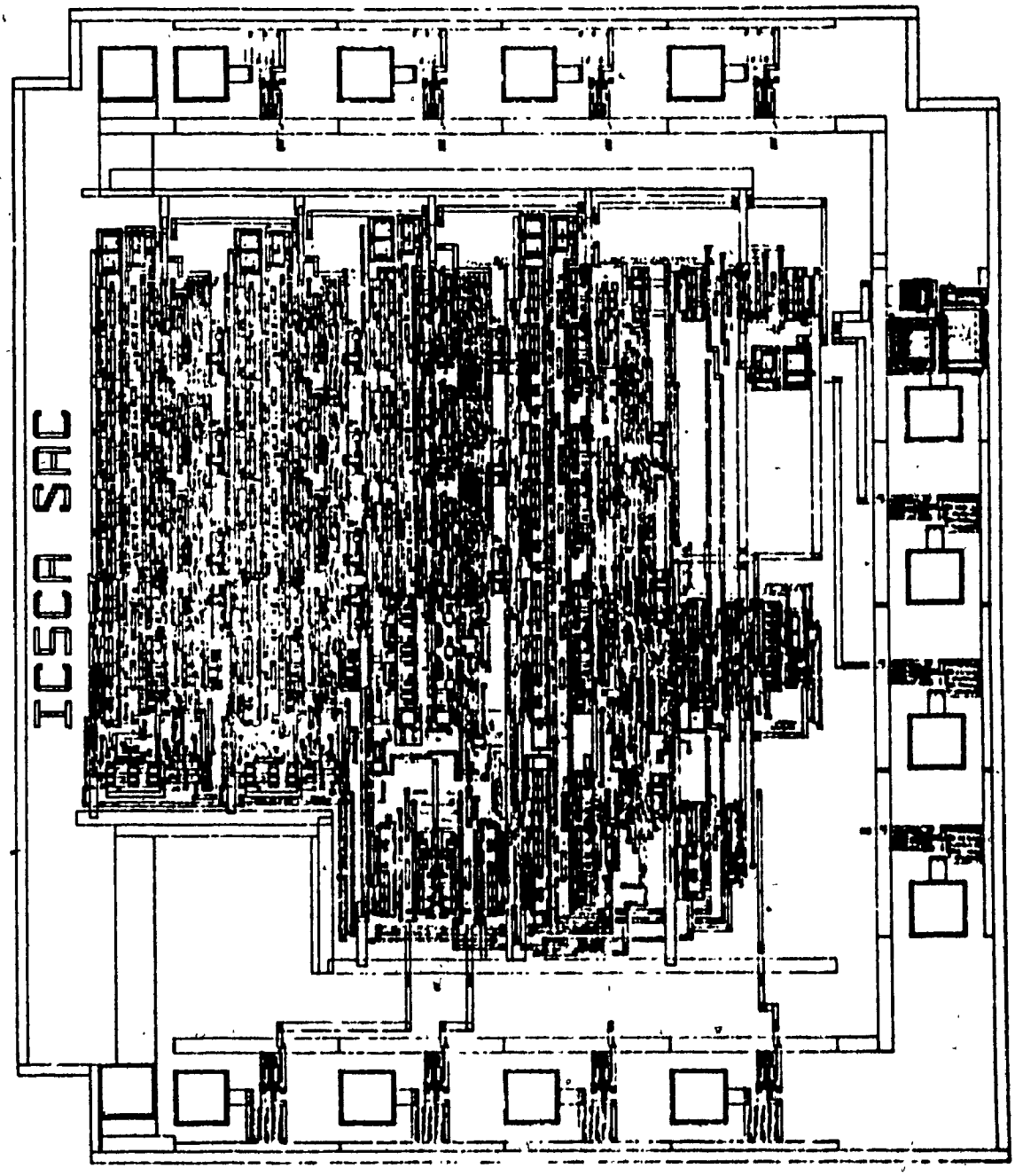



Figure 2.15 One slow stack floor plan

### 2.4.5 Layout design level

Layout design in VLSI comprises of definition of the masks to form switching devices, placement of devices and interconnection of the devices in the required pattern. Advantage of a modular design technique becomes evident at layout level as it reduces layout complexity by allowing repeated usage of modules. Often circuit design, placement and routing are done using automated tools and standard cell library. However, the one slow systolic stack is a full custom design with manual placement and routing. Figure 2.15 gives the floor plan of one slow systolic stack and the layout generated using the graphic editor KIC is shown in figure 2.16.

The layout design for a systolic system is comparatively less complex as the logical locality implied in the algorithm can be used to reduce its complexity. Also, the logical locality reduces routing area overhead and thus the communication delay in the layout. In general, two principal tasks carried out during a layout design are placement and routing. The goal of placement, during layout, is to arrive at a suitable positioning of modules in the overall floor plan of the chip so that interconnection wire length and area can be minimized. As the underlying CMOS process(CMOS-1B) provides only two layers of




2.16 One slow stack layout

interconnection wires, efficient signal routing is not a trivial task, especially considering the existence of sensitive signals like clock and power. The clock distribution and power routing used in the one slow design is discussed below.

Power routing takes top priority over signal routing due to the following reasons:

(1) Power has to reach all the active devices on the chip.

(2) Power routing should be done so as to minimize crossunders as the contact cuts in them will limit current handling capability of power rails.

(3) Power routing structure should be so as to minimize voltage drop along power rail, as reduction in supply voltage affects noise immunity and hence satisfactory operation of the system.

An ideal power routing scheme often requires more than one low resistance interconnection layer(like metal). Also, there is no known near optimal power routing scheme using a limited number of interconnection layers. Considering the above limitations an 'interdigited'[13] routing technique, as shown in figure 2.17, is used in the one slow stack design to minimize voltage drop along power rails.

Clock signals, in the same way as power buses, cannot be routed using polysilicon, as it introduces large delay and skew. Fisher[9] has suggested clock distribution scheme

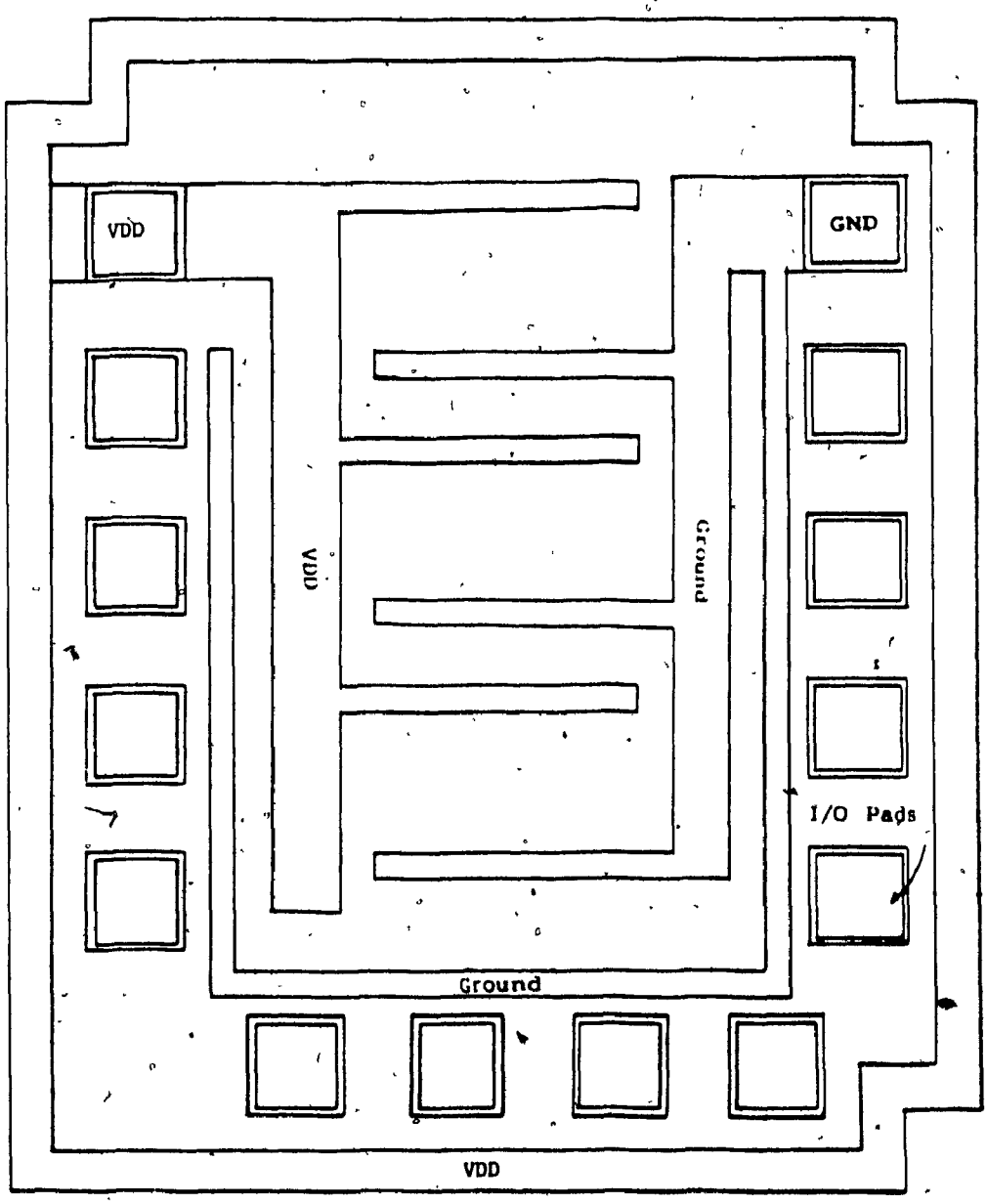



Figure 2.17 One slow stack power distribution

with minimal and bounded skew property for one dimensional arrays. The clock distribution scheme used in one slow stack is shown in figure 2.18. This scheme is similar to Fishers approach but for the buffer stages. The buffer stages are used in the clock distribution network to minimize delay in the network. Also, this approach allows use of smaller clock driver stages. The capacity of the clock drivers used in the design can be derived based on the delay folding scheme of [31]. A notable advantage of the above clock distribution scheme is its easy expandability. The clock distribution scheme used in the one slow layout has one crossunder for every slice at the slice clock driver input as shown in figure 2.15.

The data signals of the stack are routed using polysilicon in small subnets of length less than 500 micrometer. The rewrite control signals, which reach all the data cells in a slice, are grouped into a single channel and are routed between slices as shown in figure 2.15.

Expandability, though not an explicitly stated objective, is an important property of systolic network. The one slow design presented in this chapter achieves, in an elegant manner, the above objective. Expandability at the logic design layer is achieved by using modular design technique, while at the layout level suitable placement of modules ensures expandability. The one slow stack of figure

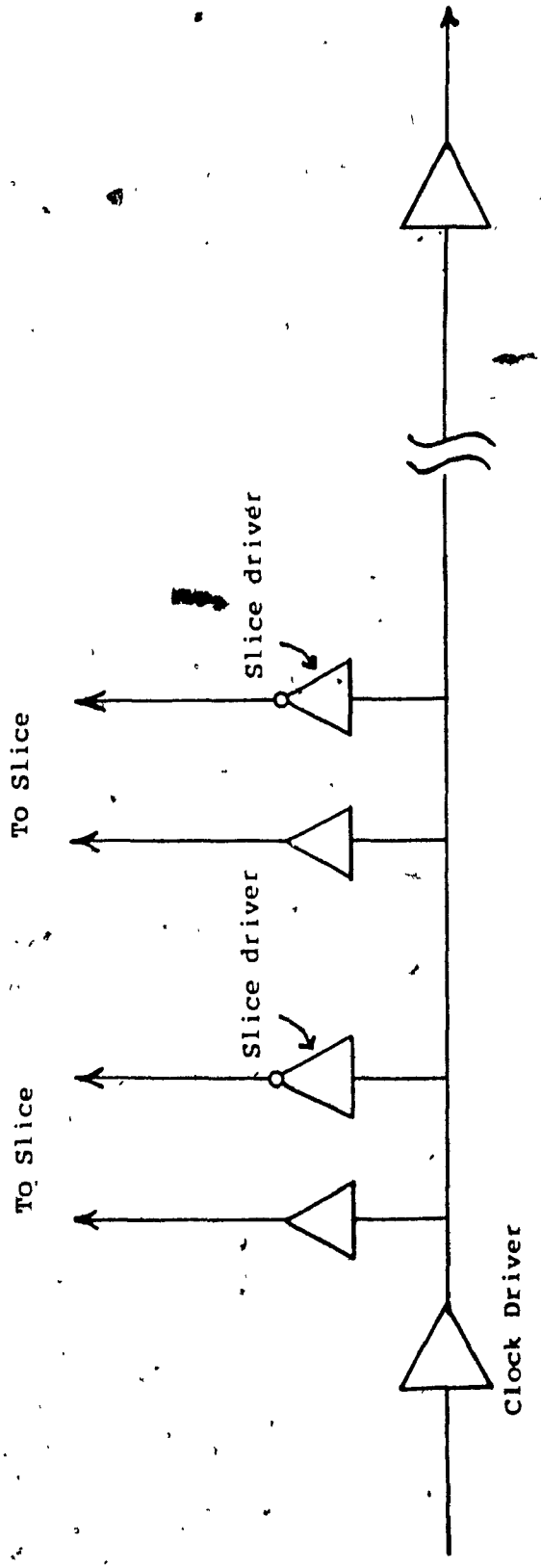



Figure 2.18 One slow stack clock distribution

2.15 is expandable both by the number of bits in a word and number of slices. The number of bits in a word can be increased by appending the required number of data blocks(three data cells) to each slice. The number of slices in a stack can be increased by stacking the desired number of slices on the present structure.

# CHAPTER 3

## SYSTOLIC STACK ANALYSIS

The enormous cost, significant design effort, and poor adaptability for modifications mean that a designer must rigorously verify and if possible prove the various aspects of a design before fabricating it as a chip. A complete verification of one slow systolic design would involve proving the correctness of systolic stack algorithm, proving the correctness of architectural specification against abstract specification, verifying the logic design for correct functional operation and verifying the circuit design for proper voltage levels, glitch free operation etc. Such an exhaustive proof and verification of the stack is beyond the scope of this thesis. Also, the correctness of systolic algorithm is dicussed in Guibas[14] and the proof for the architectural specification is discussed in Li[19]. In this chapter we discuss verification of the logic design, circuit design and analysis of the design for various system parameters. The two design verifications are carried out using logic and circuit simulators respectively.

### 3.1 Logic simulation

A one slow systolic stack comprising of an interface slice and two storage slices is simulated using the logic

simulator RNL. RNL is a switch level simulator and uses a capacitor-switch model of a transistor for simulation. The logic simulator results do not show a pronounced effect of the transistor geometry variation nor will it give an accurate estimate of switching speed. Thus, logic simulation using RNL is not suitable for determining circuit timing parameters. The logic simulation is primarily aimed at verifying the transistorized implementation of the different modules(system) against their functional specification.

An extensive simulation of the nine element stack involves testing the stacks functionality with $3^9$ strings, where each string is of length 9 and represents one possible combination of push, pop and skip signal. Such an extensive simulation is time consuming and often difficult to execute. However, taking into consideration the fact that the systolic stack algorithm and architecture are proven, the one slow stack is simulated for a few but significant string of operators at the logic simulation level. Source code for RNL simulation is given in Appendix I. For brevity, the simulation results are not enclosed. As the source code indicates, apart from proper data transfer within the stack, the one slow stack is tested for its ability to preserve its integrity in a stack full or stack empty state. The hierarchical structure of the logic modules and iterative structure of the simulation software allows easy simulation

of stack of any desired size.

## 3.2 Circuit simulation

The aim of circuit simulation is to analyse the dynamic behaviour of a transitor circuit. The circuit simulator used in this design is SPICE 2G.6 and the simulation is performed using the level 2 transistor model. There are two type of simulation performed on a circuit, namely:

(1) DC analysis

(2) Transient analysis

DC analysis is performed to determine the noise margin and threshold voltage of a circuit. The transient analysis by considering various capacitors of the circuit provides information on rise time , fall time and propogation delay of a circuit. In the one slow systolic design the DC analysis is performed only for the modules designed using complementary logic. Appendix II provides sample results of the DC analysis and table 3.1 gives a summary of the DC analysis results.

On the contrary, transient analysis is carried out for all the different datapaths in a functional module. Sample results of the transient analysis is given in Appendix III and it corresponds to the simulation of the longest path of that module. Detailed circuit analysis is also carried-out for critical path in the system and significance of such an

| Circuit | Voh | Vih | Vol | Vil | NMh | NMl |
|---|---|---|---|---|---|---|
| Rewrite control for slice 0 | 4.7V | 3V | 0.23V | 1.8V | 1.7V | 1.6V |
| Rewrite control for storage slice | 4.8V | 2.4V | 0.4V | 1.4V | 2.4V | 1V |
| clock drivers | | | | | | |
| Driver 1 | 4.8V | 3V | 0.23V | 1.8V | 1.8V | 1.58V |
| Driver 2 | 4.7V | 3V | 0.25V | 2V | 1.7V | 1.75V |

Table 3.1 DC analysis results

Where,
Vol = Output voltage low
Voh = Output voltage high
Vil = Input voltage low
Vih = Input voltage high
NMh = Noise margin high
NMl = Noise margin low

analysis is discussed in section 3.3.1.

## 3.3 Performance Analysis

While simulation is to verify proper functional operation of the circuit, additional analysis is required to define external operational and electrical characteristics of the system. In this chapter performance parameters such as maximum frequency of operation, minimum frequency of operation, power consumption are analysed for the one slow design. The layout is also analysed for its quality by examining the issues such as expandability of the layout and component density of the layout.

### 3.3.1 Maximum frequency

The maximum frequency of operation is calculated based on the propogation delay along the critical datapath in the system. Critical path in a circuit is the longest combinatorial path in the circuit. As one slow design uses single phase clock and domino logic, critical path of the system can be divided into two sections - critical path for high period and critical path for low period of clock. During critical path analysis a master-slave flip-flop is treated as two distinct latches, one operating in the high period and the other in the low period. Critical path for a particular clock period is defined as the longest path, in

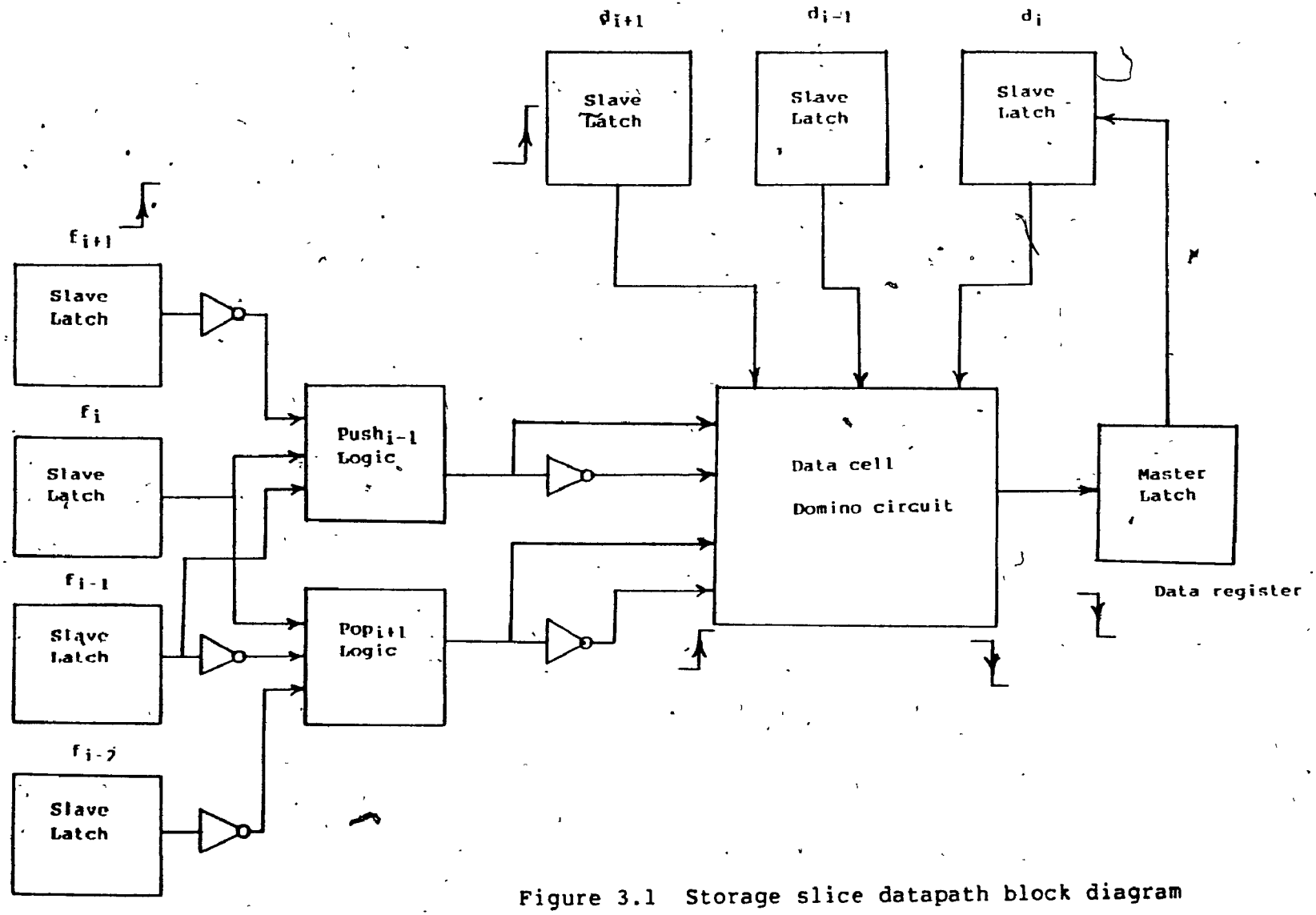



Figure 3.1 Storage slice datapath block diagram

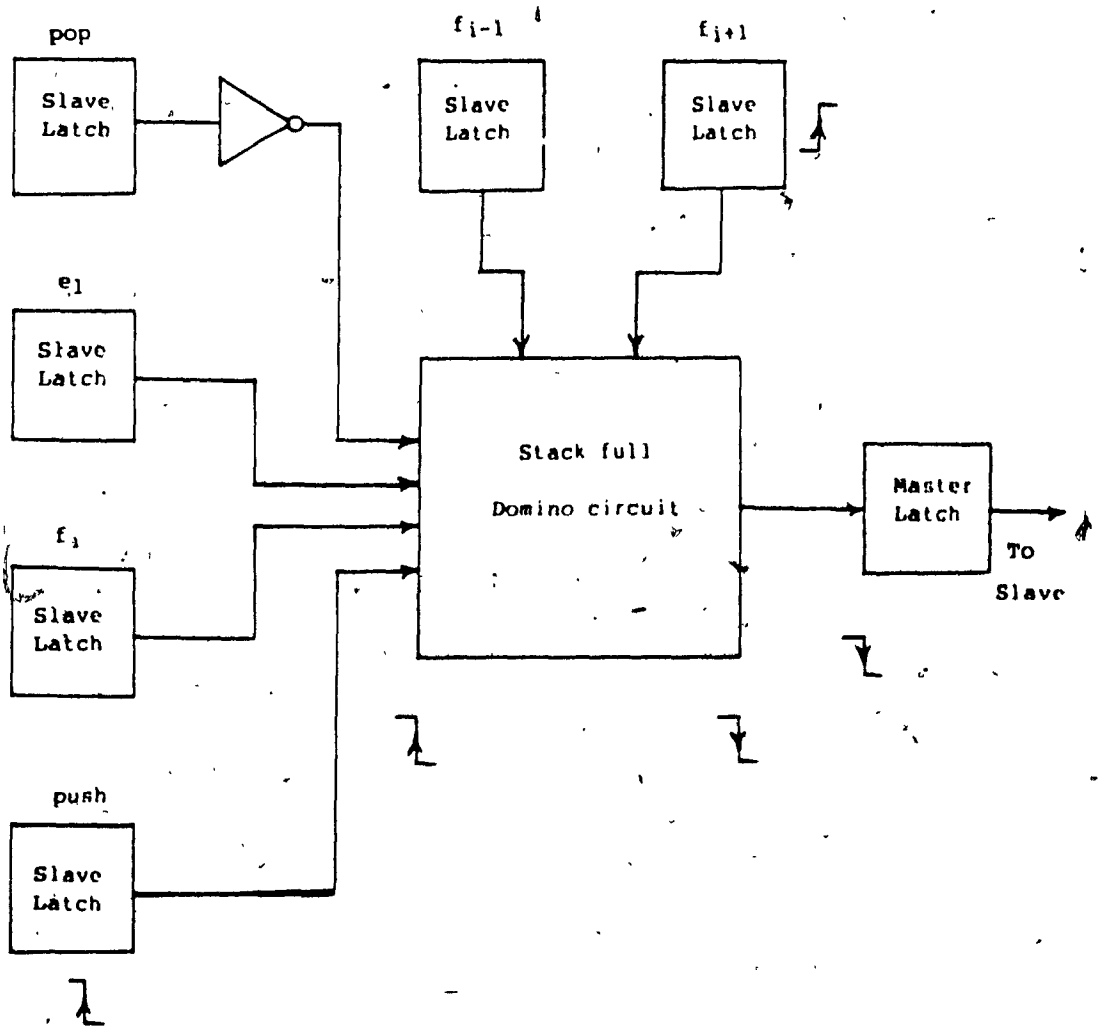



Figure 3.2 Datapath block diagram for stack full module

the system, between two successive edges of the same type. The datapath analysis indicates that storage slice has the critical path for the high period while the stack-full module has the critical path for the low period. Figure 3.1 and Figure 3.2 give the block diagram for a storage slice(data cell) and the stack-full module indicating the components in the various datapaths. Detailed circuit for the critical paths are given in figure 3.3 and figure 3.4. The minimum clock high period is given by

High period = Max($datapath_1$,.....,$datapath_n$,precharge period)

Table 3.2 gives the high period for the various modules in the system. The minuimum high period for the system is given by the maximum value of the high periods.

The minimum clock low period is given by

Low period = Evaluation time + latch delay

Table 3.2 gives the evaluation time and low period for the various modules in the system. The minimum low period is given by the maximum value of the low periods. Thus, from the table

High period = 21.5 nsec

Low period = 24 nsec

Hence,

Maximum frequency = 22 M hz

Duty cycle = 47 %

It may be noted that maximum frequency is independent of the stack size, which is an advantage resulting from the

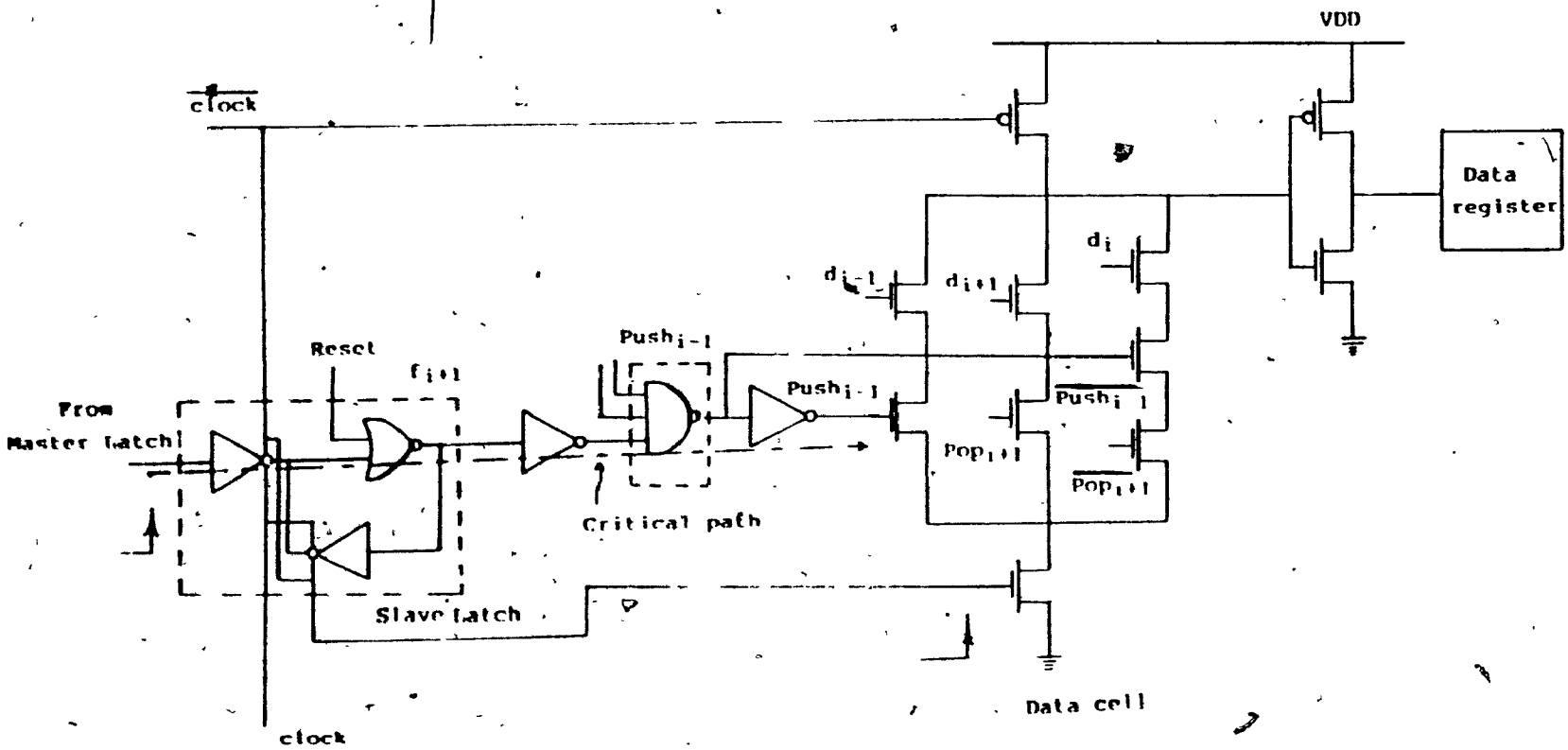



Figure 3.3 Critical path circuit for high period

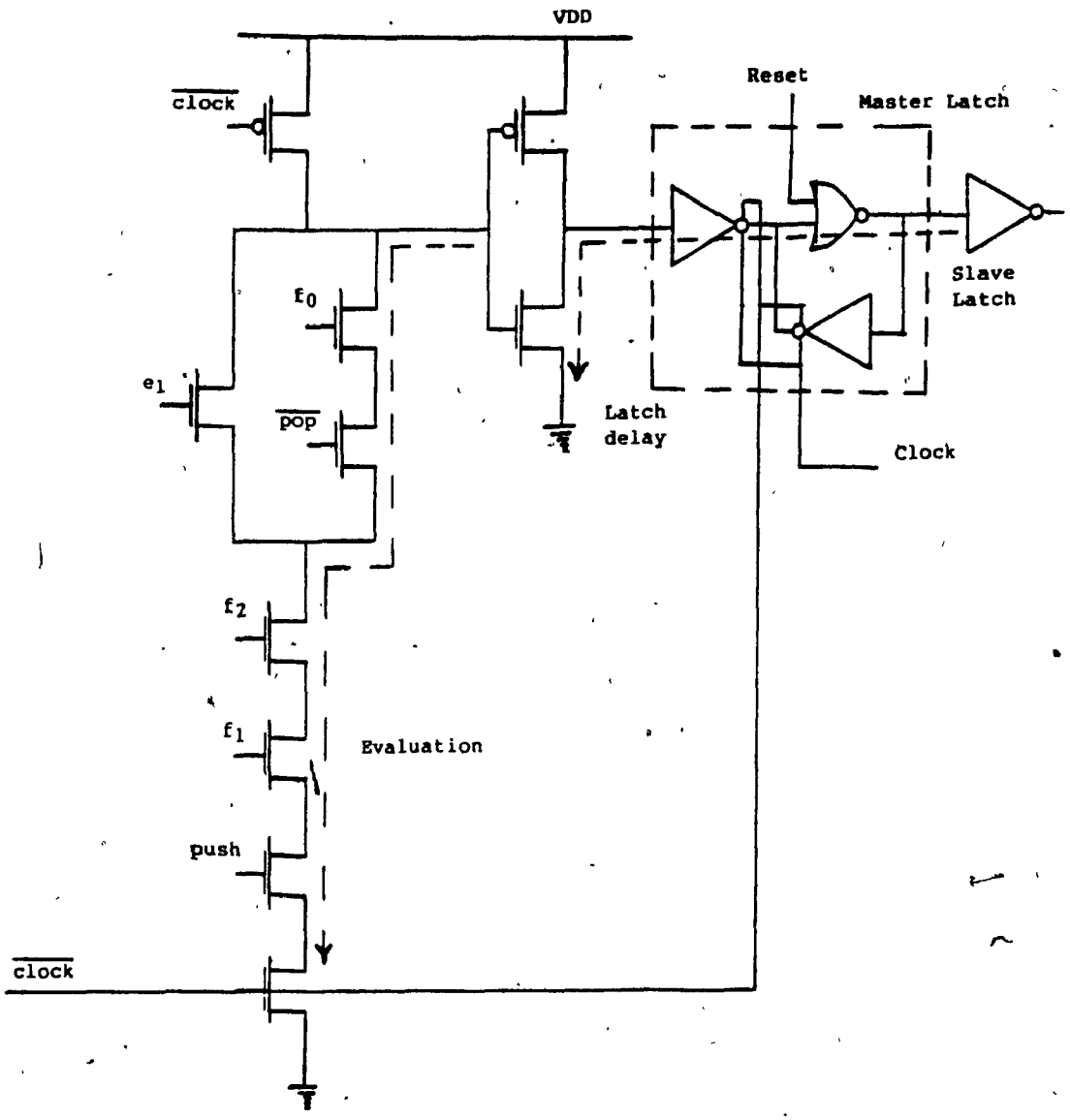



Figure 3.4 Critical path circuit for low period

| Module name | High Period in nsec | Low Period in nsec | Evaluation time in nsec |
|---|---|---|---|
| Data cell for storage slice | 21.5 | 17 | 9 |
| Data cell for interface slice | 18 | 18 | 11.5 |
| Flag bit -e2 | 10.5 | 21.5 | 10.5 |
| Flag bit-e1 | 10.5 | 20.5 | 10 |
| Flag bit-F0 | 14 | 23.5 | 14.3 |
| Flag bit for storage slices | 20 | 19 | 8 |
| Stack full | 10.5 | 24 | 14 |
| Stack empty | 23 | 11 | -- |

Table 3.2 Transient analysis results

systolic structure.

### 3.3.2 Minimum frequency

Minimum frequency of operation should be estimated for systems designed using dynamic circuits. This analysis is essential to prevent any malfunctioning of the system due to loss of charge on the charge storing capacitors of the dynamic circuit. The one slow design uses two types of dynamic circuits namely, clocked inverter and domino circuit. Though clocked inverter are used in the latch design, the feedback path in the latch makes them static. Thus the minimum frequency of operation is determined by the domino circuit only. The domino circuit for minimum frequency calculation can be modelled as shown in figure 3.5.

Where C is given by

$C = C_g + C_p$

$C_g$ = gate capacitance of the domino inverter

$C_p$ = total parasitic capacitance

$C_p = C_d + C_j$

$C_d$ = total drain capacitance

$C_j$ = total junction capacitance

For the interface slice data cell

$C_p = (C_d + C_j)$p-trans + $(C_d + C_j)$n-trans

Where,

$C_d = C_{jsw} * \text{perimeter}$

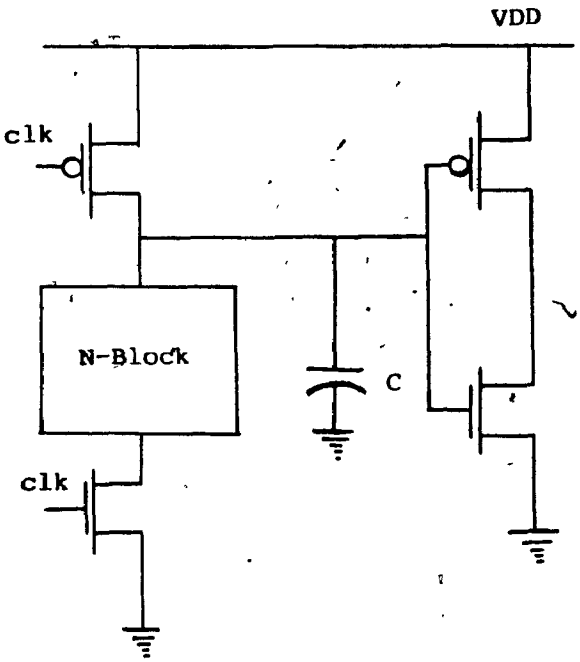



Figure 3.5 Model of a domino circuit

$C_j = C_{jo} * \text{Area}$

Substituting the numerical values for the above parameters we get

$C = 0.41$ pf

The value of $C_d$ and $C_j$ used in the above calculations are the values corresponding to the worst case model of the domino circuit + a domino circuit having maximum number of branches in the n-transistor block. The circuit considered for the minimum frequency analysis is shown in figure 3.6. In this circuit during the evaluation period only one n-transistor is considered to be off in each of the path to ground. The dotted lines in the figure 3.6 indicates the various discharge paths for the capacitor C. The capacitor discharge time constant is determined based on the following assumptions

(1) The ON resistance of a transistor is many orders of magnitude less than the OFF resistance. The estimated value of the transistor OFF resistance is $10^{10}$.

(2) The resistivity of $Sio_2$ is of the order of $10^{16}$ and more. Thus resulting in gate current of the order of pico amphere or less and a gate-source resistance of $10^{12}$ and more.

Neglecting the transistor on-resistance an equivalent resistor capacitor representation of the domino circuit can be derived as shown in figure 3.7. The effective resistance of the discharge path calculated for the parallel resistor combination is $2*10^9$. Assuming an exponential discharge

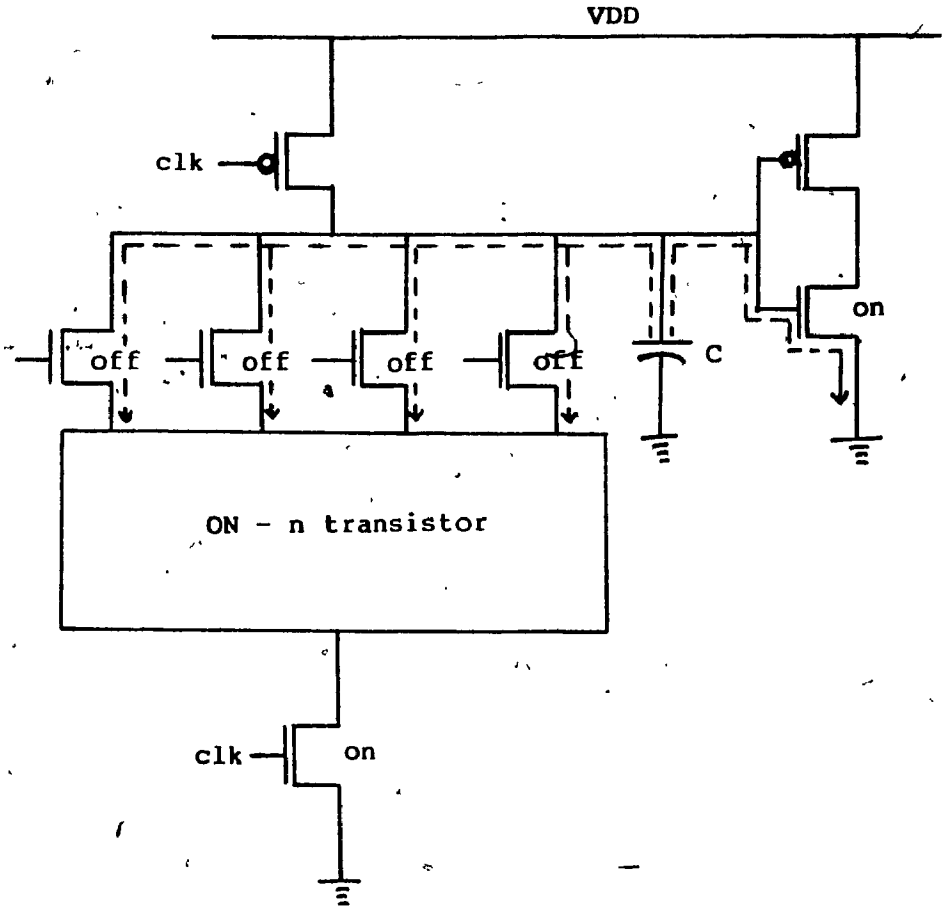



Figure 3.6 Capacitor discharge paths

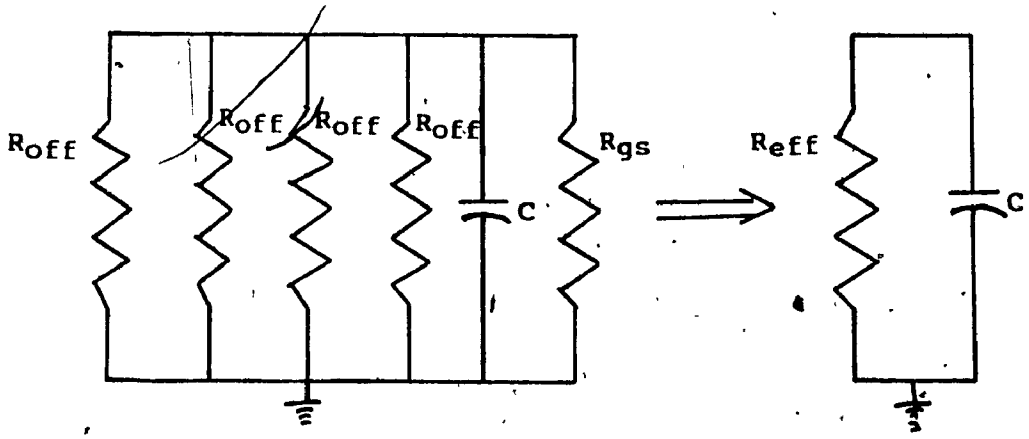



Figure 3.7 Equivalent resistance circuit

characteristics for the capacitor, the discharge time constant can be obtained using the formula

$$V = V_0.e^{-t/rc}$$

where,

V = final voltage

= $V_{ih}$ = 3 v

$V_0$ = Initial voltage

= Vdd = 5v

C = load capacitance

t = discharge path resistance

R = discharge path resistance

= $R_{eff}$

Thus we have,

$t = -RC \ln V_{ih}/Vdd$

Substituting the values we get

t = 0.41 msec.

Total period = High period + Low period

= 25 nsec + 0.41 msec

= 0.41 msec

Minimum frequency = 1/total period

= 2.4 K hz.

### 3.3.3 Power consumption

The circuit elements in CMOS are made up of both n-type and p-type transistors. These circuits do not provide direct path to ground and hence their power consumption is

quite small. The total power consumed by a CMOS circuit can be grouped under two categories - Static power and Dynamic power. Static power in CMOS systems is due to leakage current, that is current due to substrate conduction. This current is quite small (less than a nano ampere), thus resulting in negligible static power. Dynamic power is the power consumed by a circuit due to switching - capacitor charging and discharging. The average dynamic power consumed by a CMOS circuit is given by

$$P = C_l.Vdd^2.F$$

Where, $C_l$ is total load capacitance and F is the switching frequency. The above equation for dynamic power is modified to handle circuits with n-transistor chains, as shown in figure 3.8. The equation is derived based on the following concept. When the pulldown transistor in the n-transistor chain is off, remaining transistors in the chain can be considered as pass transistors. Using this concept, the voltage at each of the capacitor nodes can be determined. The total dynamic power consumed by such a circuit is given by

$$P = F[C_0.V_1^2 + C_1.V_2^2 + .....+ C_n.V_n^2] + C_g.Vdd^2.F$$

$$= F[C_0.Vdd^2 + C_1(Vdd-V_{th})^2 + .... + C_n(Vdd-nV_{th})^2] + C_g.Vdd^2.F$$

$$= F[\sum_{i=0}^{n} C_i\ (Vdd-iV_{th})^2] + C_g.Vdd^2.F$$

Where, $C_g$ is the total gate capacitance and $C_i$ is the total capacitance at node i. Also, $C_i$ is given by .

$$C_i = C_{ds} + C_j$$

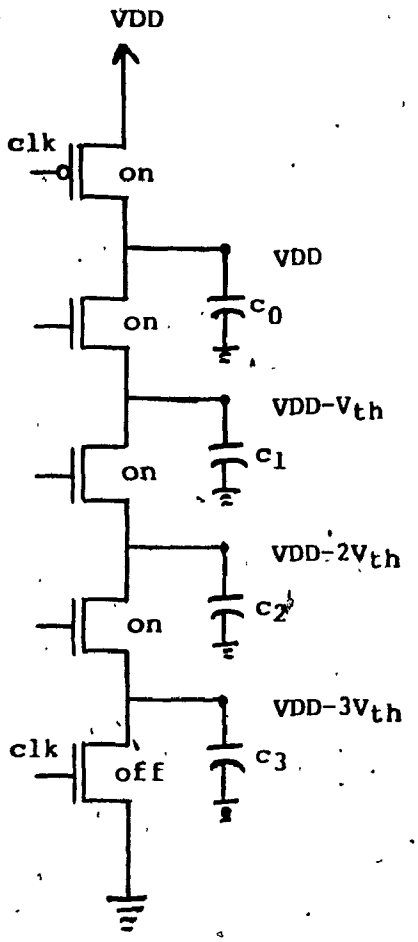



Figure 3.8 n-transistor chain.

| Module name | Dynamic power in mW |
|---|---|
| Data cell for storage slice | 0.6 |
| Data cell for slice 0 | 0.65 |
| Flag bit - e1 | 0.68 |
| Flag bit - e2 | 0.69 |
| Flag bit - f 0 | 0.7 |
| Flag bit for storage slice | 0.7 |
| Rewrite logic slice 0 | 0.37 |
| Rewrite logic storage slice | 0.24 |
| Stack full | 0.73 |
| Stack empty | 0.79 |
| Interface module | 0.84 |

Table 3.3 Power requirements

$C_{ds}$ is the drain/source capacitance and $C_j$ is the junction capacitance. The dynamic power analysis for a system is carried out by decomposing the system into circuits operating at different frequencies. The total capacitance for each frequency is calculated and power for each frequency component is estimated using the above equation. Table 3.3 gives the dynamic power consumed by each of the functional modules when the system is operated at its maximum frequency. The total power consumed by a 9 word by 1 bit stack is 12 mW. However the actual power consumed by such a system will be much less if we take into account the input data change rate.

### 3.3.4 Layout expandability

In a synchronous system layout expandability or maximum layout area depends on the routing delay, clock skew and power routing problem.

Routing delay in the one slow systolic stack can be due to the polysilicon or metal used in the routing. The poly routing in one slow stack is done in small subnets of 500 um and less, which corresponds to a delay of 0.08 nsec. For a system to operate without severe performance degradation it should possess the twire much less than tgate characteristics. Considering tgate as 5 nsec in the CMOS-1B process, it is seen that the polysilicon used in signal

routing does not introduce any system degradation. As the polysilicon routing length does not depend on the stack size, the one slow stack can be expanded to any desired size without any performance degradation. However, the length of the rewrite control signals which are routed in metal increases with the number of bits in a stack element. Using the information that the length of rewrite control signal is 700 μm per bit and the RC model for delay it is found that without degradation in performance, each element of the stack can be expanded to a maximum of 27 bits.

Using circuit analysis of the clock distribution network it is determined that there is a clock skew of 1 nsec between clock and its complement in a slice. However, as the one slow stack does not allow simultaneous data transfer between two slices it is immune to interslice clock skew problems. Thus clock skew does not impose any restriction on the one slow stack size or expandability.

Finally the effect of power routing on stack size can be determined by examining the voltage drop along power rails. Considering the power requirement per data cell(0.6 mW), the length of a data cell(700 μm) and allowable power line voltage drop without severly affecting noise immunity of the system, we determine that one slow stack can be expanded to a maximum of 28 bits per element.

In conclusion, the one slow layout discussed in chapter 2 can be expanded to a maximum of 27 bits per element. However, if the number of slices is too large the single power inlet pad used in the design could become a power limiting factor due to current density limitations of the routing metal layer.

### 3.3.5 Packaging density

Packaging density provides an estimate of the area occupied by the active components in a design, thus giving information on silicon utilization and apropriateness of the design for VLSI. The one slow design occupies an area of 1700 μm X 1800 μm and has approximately 625 transistors, giving an approximate area of 70 $\mu m^2$ per transistor. Also, after a detailed calculation of the area occupied by various components of the circuit it is found that power routing occupies 6.6% of the silicon area, interconnection wires occupies 18% of the silicon area and the remaining 75.4% is occupied by the active components in the system. Thus giving a very high silicon utilization which is in accordance to the systolic network characteristics.

Finally, to complete the analysis two known systolic algorithms for stack are compared for their area and time performance. Table 3.4 summarises the comparison results. The area, time and fanout requirements of the various known

| Architecture | Area | Time |
|---|---|---|
| Four slow | N(m+1) | 4 unit |
| One slow | (N/3)(3m+1) | 1 unit |

Table 3.4 Systolic stack comparison

| Architecture | Area | Time | Fanout |
|---|---|---|---|
| Array | N+logN | Time for increment and access time | Proportional to size |
| Shift register | N(m+1)+ | 1 unit | Proportional to size |
| Ripple | N(m+1) | Proportional to number of elements | Bounded |
| Four slow | N(m+1) | 4 unit | Bounded |
| One slow | N/3(3m+1) | 1 unit | Bounded |

Table 3.5 Stack architecture comparison

+ N = number of words
  m = number of bits per word

stack implementations are also compiled and table 3.5 summarises the results. In terms of the the actual gate complexity of the implementations, neglecting the host interface design, the four and one slow design have the similar transistor counts at the functional module level. However, at the system level, the percentage area improvement of one slow design over four slow design is the maximum, 33%, when each stack element is 1-bit wide. This area improvement decreases with the increase in the word length of the stack elements.

# CHAPTER 4

## SELF TIMED ARBITER

### 4.1 Introduction

Arbiters are used in digital system to resolve conflict which arise whenever more than one autonomous unit request access to a shared resource simultaneously or with in a short period of time from one another. In general, requests for the shared resource may be issued at any time and any number of requests may be outstanding at a given time. Functionally, arbiters ensure integrity of the shared object by allowing only mutually exclusive accesss to the shared object. The order in which access to the shared resource is permitted by the arbiter is based on one of the following allocation schemes - fixed priority, rotating priority, dynamic priority scheme. Thus an arbiter receives a set of n requests $r_1, r_2, \ldots r_n$ and returns acknowledge $a_1, a_2, \ldots a_n$ (acknowledge $a_i$ $1 \leq i \leq n$, corresponding to request $r_i$) back to the requesting source, such that the mutual exclusion requirement among the acknowledges is not violated. Stated formally, for the arbiter of fig 4.1, we have

$$a_1 = f_1(r_1, r_2, \ldots, r_n)$$

$$a_2 = f_2(r_1, r_2, \ldots, r_n)$$

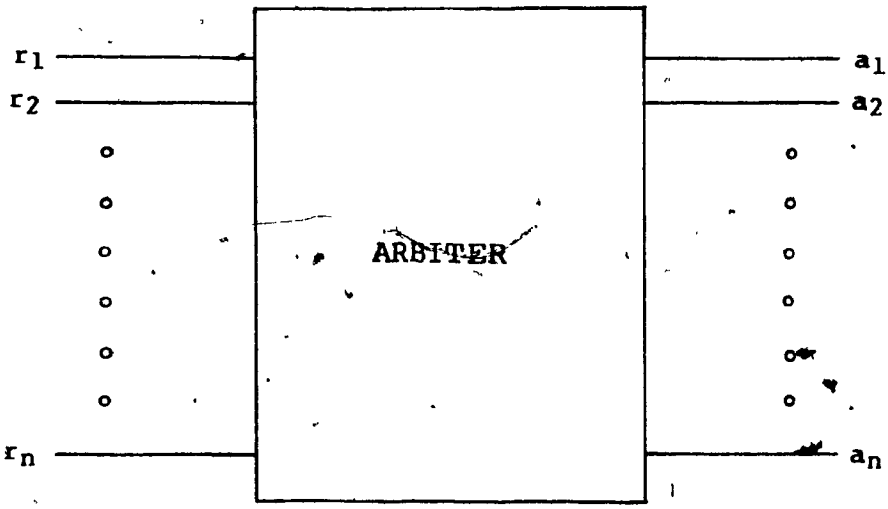



Figure 4.1 Arbiter block diagram

$$a_n = f_n(r_1, r_2, \ldots, r_n)$$

where $a_i \wedge a_j = 0$ for all i,j such that $r_i$ and $r_j$ are in conflict.

## 4.2 Arbiter classification

The arbitration structure in a system can be modelled as a conflict graph. A conflict graph is an undirected graph G = (V,E) representing the conflict(mutual exclusion requirement) between two requesting sources. The vertices $V = \{1, 2, \ldots, n\}$ of the conflict graph represent the request sources $r_1, r_2, \ldots, r_n$ and the edge (i,j) represent conflict between $r_i$ and $r_j$. Hence, an edge $(i,j) \in E$ implies that the acknowledge $a_i$ and $a_j$ are mutually exclusive.

Arbiters can be classified based on their implementation technique, arbitration scheme, or structure of the underlying conflict graph as presented in this chapter. Based on the conflict structure, there are two type of arbiters namely 1-out of-N and general arbiters. The conflict graph of a 1-out of-N arbiter is a complete graph. Thus a 1-out of-N arbiter acknowledges only one of the many requesting sources at any given time. Stated formally, in a 1-out of-N arbiter $a_i \wedge a_j = 0$ for all i,j. In contrast, the conflict graph for a general arbiter is not a complete graph. Hence, for a given set of active requests,

the general arbiter acknowledges a maximum subset of requests which are mutually exclusive.

The 1-out of-N type of arbiters find applications in computer system at the PMS(Processor-memory-switch)[35] level and the general arbiters find application at the VLSI level. The general arbitration structure can be found in the class of self-timed systems such as petri net, data flow systems, and path expression based systems. Some of these applications are discussed in section 4.6.

## 4.3 Arbiter design

Seitz[32] and Plummer[28] have proposed design technique for 1-out of-N arbiters. But, extendability of these techniques to the general arbiter design is not known. In this chapter we present a self timed design technique for the synthesis of general arbiters. Though only general arbiter design is discussed here, the design approach is equally applicable to synthesis of 1-out of-N arbiters, as they are a restricted class of general arbiters. The design presented in this chapter uses a distributed architecture and is suitable for VLSI as they exhibit characteristics such as modularity, homogeneity and locality in communication.

### 4.3.1 Design hierarchy

The design hierarchy discussed in chapter 2 is applicable to asynchronous design also. In particular to the arbiter design, the issues corresponding to the behavioral specification are beyond the scope of this thesis. Interested readers are referred to Black[2] for a detailed discussion of trace specification for asynchronous arbiters. In this thesis we discuss, in detail issues pertaining to the architectural design of self timed general arbiters.

#### 4.3.1.1 Architecture design

The distributed architecture of a general arbiter can be described as follows. For each edge(i,j) of the conflict graph a 1-out of-2 arbiter A(i,j) is associated. The local arbiter A(i,j) is used to resolve conflict between the requests $r_i$ and $r_j$ and all the local arbiters resolve conflict independent of one another. The acknowledge $a_i$, corresponding to the request $r_i$ is generated by combining all the local acknowledges corresponding to the edges incident on the vertex i. A fundamental problem in the above approach is the possibility of deadlock due to the autonomous and distributed decision made by the local arbiters. In an asynchronous environment, where signal transition can occur at any instant of time and a signal

takes arbitrary but finite time to travel from one point to another. In such an environment a solution to eliminate deadlock caused by the distributed decision requires careful consideration of issues such as synchronization. Solutions using priority scheme for each edge, though sufficient in an synchronous environment as shown in chapter 5, do not lead to acceptable solution during asynchronous implementation. Thus, in this section we present a deadlock free solution for asynchronous synthesis of general arbiter using the concept of edge request serialization. This solution is a generalization of the solution to the dining philosophers problem[15].

The concept of edge request serialization can be described as follows. In a conflict graph G, each vertex i (request source) will send the request to one of the incident edges at a time. After receiving an acknowledge from the local arbiter of that edge, the request is forwarded to the next incident edge. This process is repeated until all the edges incident on vertex i are accounted for. In the edge request serialization technique, any conflict at a local arbiter due to simultaneous requests can be resolved arbitrarily. The global acknowledge $a_i$, corresponding to request $r_i$ $1 \leq i \leq n$, is generated only after receiving the local acknowledge from the local arbiter corresponding to the last edge in the serialization order. In this chapter, a unique label is used to represent the

serialization order of the edges for a given vertex. These labels are placed on the edges next to a vertex and an edge with the highest label represents the last edge in the serialization order for that vertex. However it needs to be noted that an arbitrary ordering of the edge request can lead to deadlock. Before presenting an algorithm for deadlock free serialization of edge requests, we first establish the criterion for such a request serialization.

Consider a conflict graph in which the edges incident at every vertex have been serialized. In a cycle of such a graph, every vertex is incident at two edges with two distinct serialization numbers. Between these two edges, the one with a higher serialization number is called a H-edge, and the other(with lower serialization number) is called a L-edge. A cycle in such a conflict graph is called a serialization cycle if, while traversing it, all the vertices in it are entered along the H-edges(L-edges) and left along the L-edges(H-edges). Any conflict graph which comprises of such a serialization cycle is prone to deadlock problem. For example, consider a serialization cycle consisting of vertices 1,2,...,k, and let the requests $r_1, r_2, \ldots, r_k$ corresponding to these vertices be all raised simultaneously. Each vertex in the cycle access the arbiter on L-edge before the H-edge. If all the accesses take same amount of time, then each vertex in the cycle will wait for acknowledge from the local arbiter on the H-edge, thus

resulting in cyclic dependency and deadlock. Thus, a deadlock free operation can be ensured by avoiding any such serialization cycles in the conflict graph.

Theorem 4.1

Absence of a serialization cycle in the conflict graph guarantees deadlock free operation.

Proof: Assume the conflict graph has deadlock. Then there is a set of vertices{ {1,2,...,k} which are in cyclic dependency. Consequently, each vertex has received acknowledge from the local arbiter corresponding to one of its edges and is waiting for acknowledge from the local arbiter corresponding to the other edge. Label these edges as L-edge and H-edge respectively. Such a labelling of all the vertices in the set will result in a serialization cycle and hence the theorem.

Thus any algorithm used for edge request serialization of a given conflict graph should ensure that no serialization cycle is formed. One such algorithm is given below. This algorithm also attempts to improve the performance by allowing as many global acknowledges as possible to be generated at a given time. In the algorithm below, the concept of colouring is used to indicate all the vertices that can be serialized during a particular

iteration of the algorithm. Conceptually, working of the edge request serialization algorithm can be described as follows. During the serialization labelling of vertex i, the algorithm ensures that for each edge(i,j) incident on i, the serialization number assigned at the i-end is always not greater than that assigned at the j-end. Notice that the serialization number at the j-end is always equal to the current degree of j. If such a relation cannot be guaranteed for all the edges incident on i, then i is not considered as a candidate for labelling during the current iteration.

```
Algorithm edge request serialization;
   for i = 1 to n do
   set vertex i not visited
   select a colour
   while a vertex is yet to be coloured do begin
     select a not visited vertex i with the least degree
       if deg(i) ≠ 0
         then begin
           testcolour(i,colour)
            if colour
              then begin
                assign the current colour to vertex i
                count = 1
                for each adjacent vertex j do begin
                  mark i-end of edge(i,j) with count
                  mark j-end of edge(i,j) with deg(j)
                  count := count + 1
                  mark j as visited
                  delete edge(i,j)
                end
                delete vertex i
              end
              else begin
                reset all visited marks to not visited
                select next colour
              end
         end
         else assign the current colour to vertex i
   end{while}
end{algorithm}
```

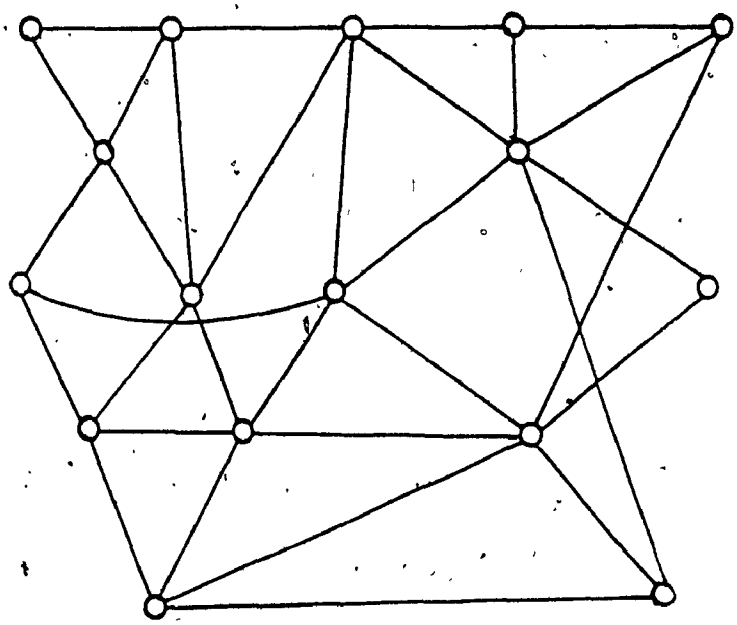


Figure 4.2 General conflict graph

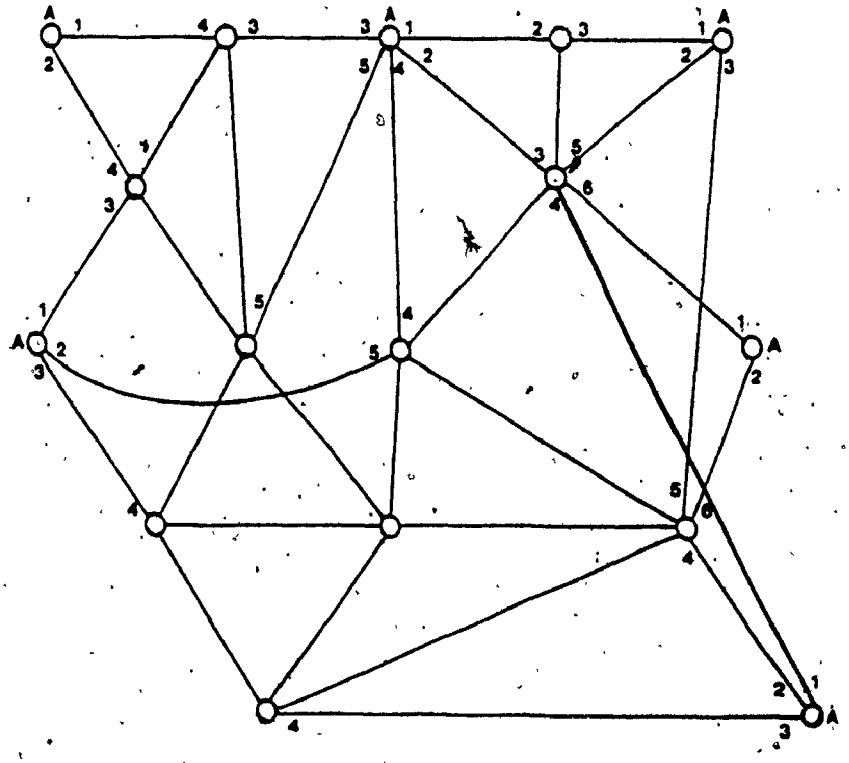


Figure 4.3 Edge request serialization for colour A

```
Procedure testcolour(i,colour)
  arrange in the array A the neighbours of i in the
ascending order of their degrees
  colour = true
  j = 1
  while j ≤ deg(i) and colour do
    if deg(A[j]) > j
       then j =j + 1
       else colour = false
end {procedure}
```

Théorem 4.2

Algorithm edge request serialization does not generate any serialization cycle.

Proof: The proof is by contradiction. Let $(i_1, i_2, \ldots, i_k, i_1)$ be a serialization cycle. Without loss of generality assume that this cycle is generated when vertex $i_k$ is being coloured and edge$(i_k, i_1)$ is being labeled by the algorithm. This serialization cycle implies that with respect to vertex $i_k$, the edge $(i_{k-1}, i_k)$ is a H-edge and edge$(i_k, i_1)$ is an L-edge. However, when vertex $i_k$ is coloured, the edge$(i_{k-1}, i_k)$ is assigned an L label and the edge$(i_k, i_1)$ is later assigned a H label leading to a contradiction. Similar contradiction arises if the cycle is formed when edge$(i_{k-1}, i_k)$ is being labelled.

The working of the edge serialization algorithm for the conflict graph of figure 4.2. is shown in figure 4.3.

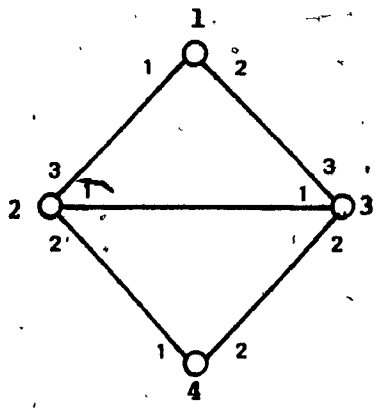



Figure 4.4 Example conflict graph

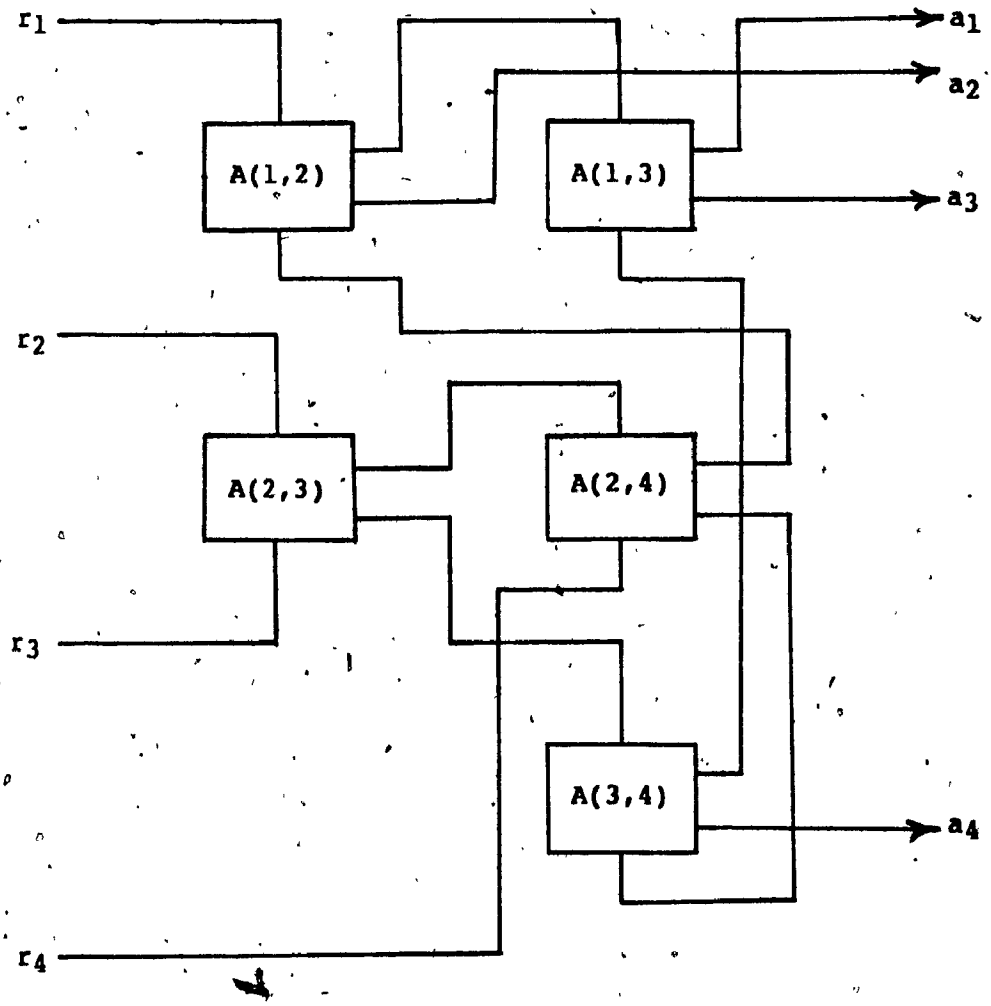


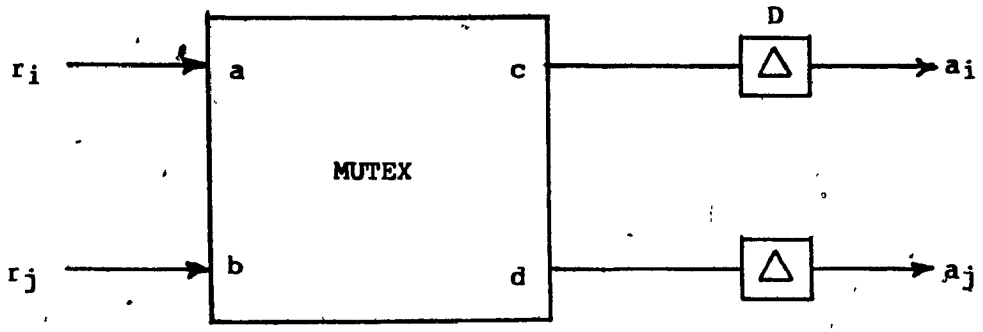

Figure 4.5 Self timed general arbiter block diagram

Figure 4.3 illustrates the operation of the algorithm for the initial colour A.

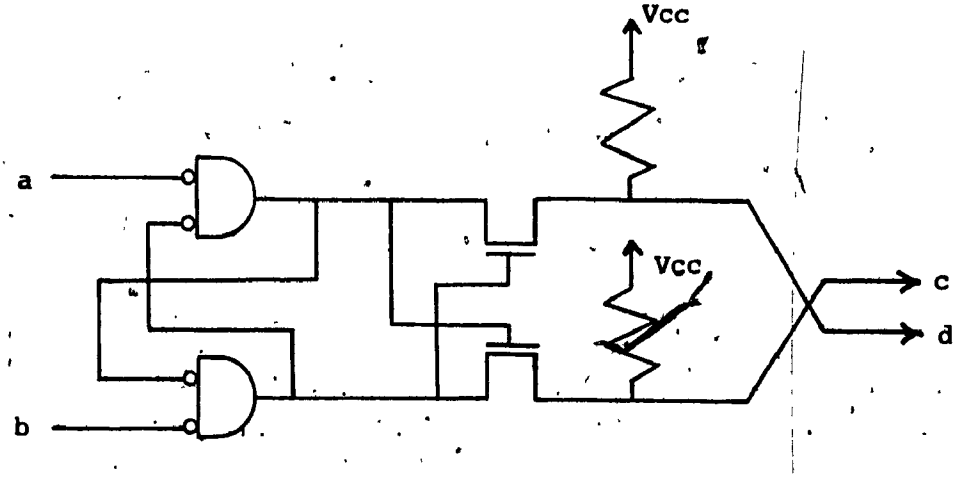
4.3.1.2 Logic and Circuit design

Design of the distributed asynchronous arbiter for the conflict graph of figure 4.4 is shown in figure 4.5. The above design is a self timed design using four cycle protocol. As shown, the distributed design is obtained by interconnecting the 1-out of-2 arbiters in the desired serialization order. There are many[2,28,32] proven designs available for 1-out of-2 asynchronous arbiter. The design used in this thesis is due to Black[2] and it comprises of Seitz's mutual exclusion element and delay elements as shown in figure 4.6. A study of the above arbiter's operation indicate that a request $r_i$ and its corresponding acknowledge $a_i$ follow a four cycle signalling protocol.

As a result of using a proven 1-out of-2 arbiter module and four cycle communication protocol in the design of general arbiters, the design concerns at the logic level of the hierarchy are not considered in this thesis. However, in general, design issues at this level vary with the design under consideration. For example, consider the issue of self timed communication between two modules. If a single set of protocol signals are used when a large number of signals need to be communicated between modules, the




ARBITER A(i,j)




nMOS Mutex

Figure 4.6 Local arbiter

variation in delay along the different signal lines will create data synchronization problems. There is more than one solution to achieve self timed data transfer between modules in such a situation. These solutions differ depending on the inter-relationship between the signal lines. When the signal lines are fully independent of each other and are unary signals, self timed communication can be achieved by using built in protocol technique and this approach requires 2N signal lines. However, if the signal lines form a data bus then by making use of the mutual exclusion property between the '1' and '0' level on a dataline it is possible to design a self timed communication mechanism using 3N signal lines. Alternatively, if it is known a priori that all the signal lines are mutually exclusive, then self timed communication between the two modules can be achieved using just N+1 signal lines. Similarly, at the logic design level a self timed system designer should give special attention to one-to-many communication. On the contrary, if a locally synchronous and globally asynchronous design technique is adopted, then the designer would be faced with the problem of ensuring metastable free operations[27] in the local modules at the logic design level. In general, at the logic design level, whenever individual modules are not locally synchronous, issues like hazards, race and state assignment need to be specifically addressed as discussed in Molnar[23] and Chu[7].

Design concerns at the circuit and layout level are mostly similar to those encountered during the design of synchronous systems, with the exception of clock related problems. As many of these issues are discussed, in detail, in chapter 2, they are not considered here again. However, asynchronous circuit designer should exercise caution while using dynamic logic structures as the absence of clock and unpredictable delay in operations could lead to loss of information in such structures.

## 4.4 Arbiter fairness

The fairness structure of an arbiter governs the waiting time of the requesting sources. In general an unfair arbiter is undesirable as it can paralyze sections of a system by forcing some of the requesting sources to wait indefinitely. The issue of fairness in asynchronous arbiters has received much attention recently and a detailed exposition of the different notions of fairness can be found in Black[2]. In this section we discuss two notions of fairness, namely strict fairness and weak fairness which are relevant to general arbiters. A 1-out of-2 arbiter is said to be strictly fair if it alternates between the two requesting sources while servicing. That is, if both the requests $r_i$ and $r_j$ are raised simultaneously at the arbiter A(i,j) and the arbiter acknowledges $r_i$, then it will not acknowledge $r_i$ again(for its next request) before

acknowledging $r_j$. However, if the arbiter receives only one request at a time, the requests will be serviced in the order of their arrival. If strict alternate servicing of requests cannot always be maintained, then such an arbiter is said to be weakly fair. Thus a weakly fair arbiter A(i,j) may service one of the request sources many times before servicing the other even though both the requests are active simultaneously. However, a weakly fair arbiter will eventually service both the active requests.

These notions of fairness can be extended to general arbiters as follows. Suppose all the requests are active simultaneously and let acknowledge $a_i$, corresponding to request $r_i$ be granted. If the global acknowledge $a_i$ for the next request of $r_i$ is not granted until all the pending requests which must be acknowledged exclusively with respect to $r_i$ are globally acknowledged, then such a general arbiter is called a strictly fair general arbiter. Thus, a strictly fair general arbiter ensures a strict ordering between $r_i$ and all the requests conflicting with it. However if such a strict ordering, between $r_i$ and all the requests conflicting with it, cannot be enforced then such a general arbiter is called a weakly fair general arbiter. Thus in a weakly fair general arbiter even though conflicting requests are raised simultaneously, one or more of these requests may have to wait for the acknowledge and during this waiting period some of its conflicting request sources may be serviced more than

once.

The local arbiter of figure 4.6 is strictly fair by construction. The delay element present on each link of the acknowledge signal introduces a delay D, equal to the arbiter resolution time, between the time an acknowledge becomes false and the time its corresponding request becomes true again, thus ensuring strict fairness.

The general arbiter designed using strictly fair local arbiters and the edge request serialization algorithm is not strictly fair. These arbiters are weakly fair. The absence of strict fairness in such general arbiters can be attributed to the concept of edge request serialization. As a consequence of request serialization even though request $r_i$ is active it is not seen by all the local arbiters which are present on the edges incident on vertex i. Thus at any given time there may be one or more local arbiters, corresponding to request $r_i$, to which the request has not yet reached. As a result of the above senario, the self timed general arbiter of this chapter cannot enforce an ordering among the conflicting request sources and hence, not strictly fair. However, as the request will eventually reach local arbiters on all the incident edges, the global acknowledge will eventually be generated and thus the general arbiters of this chapter are weakly fair.

## 4.5 Arbiter expandability

Expandability is an important property of a modular design. In this section we show how a self timed general arbiter designed using the edge request serialization algorithm can be expanded with minimal effort. A general arbiter can be expanded by adding additional conflict into the existing structure or by adding additional request sources and their corresponding conflict into the structure. The above two cases correspond to adding of edges and adding of vertices respectively. Solution to the above two types of expansion are discussed below.

Consider a conflict graph G with its edges labelled, as shown in figure 4.4, using the edge request serialization algorithm of section 4.3.1.1. The two types of extension to the graph G are shown in figure 4.7(a). One approach for obtaining a deadlock free implementation of the extended conflict graph $G_1$ or $G_2$ is to relable the graph from the scratch using the edge request serialization algorithm. But, such an approach is undesirable as it leads to redesign of the arbiter from the scratch. In this section we present an alternate solution which not only retains the existing serialization but also provides a technique for serializing the newly added edges of the graph. This extension to the procedure of section 4.3.1.1 not only results in a simpler hardware implementation but also retains the deadlock free

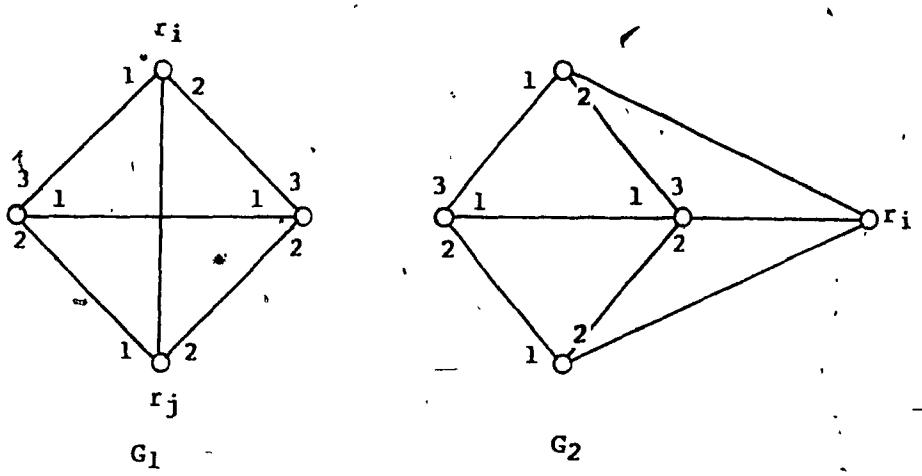



Figure 4.7(a) Unlabelled extended conflict graph

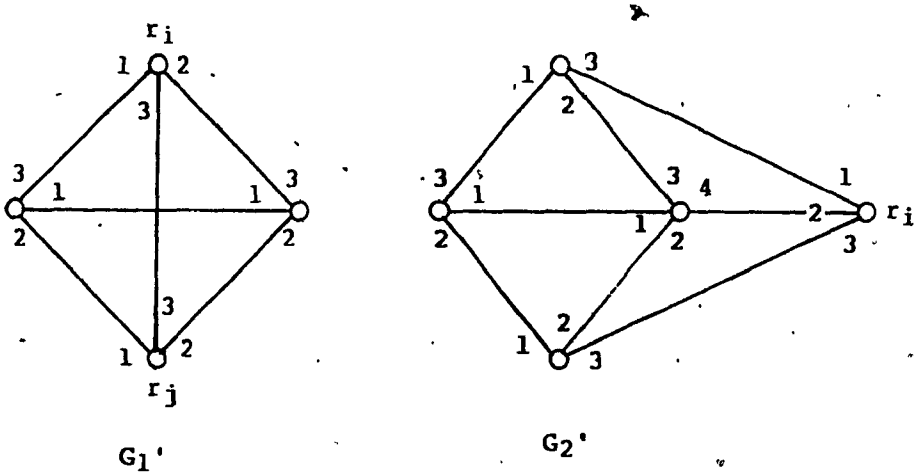



Figure 4.7(b) Labelled extended conflict graph

property of the original graph, as will be shown below.

Before discussing the proof for the deadlock free property of the serialization procedure, we first introduce the notations that are used in the proof.

4.5.1 Notations

{i,j} : Indicates i-end of the edge(i,j)

{j,i} : Indicates j-end of the edge(i,j)

$A_{L-H}$ : Vertex A is entered with a L-edge and departed with H-edge

$A_{H-L}$ : Vertex A is entered with a H-edge and departed with L-edge

$A_X$ : The serialization number for the entry and departing edge for vertex A is immaterial

Thus, a serialization cycle comprising of vertices (1,2,3,4,5,1) implies ($1_{H-L}$,$2_{H-L}$,$3_{H-L}$,$4_{H-L}$,$5_{H-L}$,$1_{H-L}$).

The serialization procedure for adding edges to a given conflict graph can be described as follows. Consider the graph $G_1$ which is obtained after adding the edge(i,j) to the conflict graph G. The newly added edge is assigned a label(serialization number) as follows.

Mark i-end of edge(i,j) with deg(i)

Mark j-end of edge(i,j) with deg(j)

Application of the above procedure to graph $G_1$ is shown in

figure 4.7(b). It can be shown that the above labelling procedure will not introduce deadlock into the graph $G_1'$. As the graph G is deadlock free we prove deadlock free property of $G_1'$ by showing that the edge(i,j) cannot be a part of any serialization cycle. This proof can be described as follows.

Let us assume that the graph $G_1'$ has deadlock and hence there is a serialization cycle including the edge(i,j). Without loss of generality let us assume that vertices (l,....,i,j,....k,l) form the serialization cycle. Using the notation of section 4.5.1, such a serialization cycle implies $(l_{H-L},...,i_{H-L},j_{H-L},....,k_{H-L},l_{H-L})$. However, as the labelling procedure assigns deg(i) to {i,j} and deg(j) to {j,i} a traversal of the cycle (l,....,i,j,....k,l) can only result in $(l_X,....,i_{L-H},j_{H-L},....k_X,l_X)$, thus a contradiction at vertex i. Hence the above edge serialization method will not introduce a deadlock into the conflict graph.

When more than one edge needs to be added to the conflict graph, it is done so by adding one edge at a time. After addition of each edge the above serialization procedure is applied to the newly added edge. This procedure is repeated until all the edges are added. The absence of deadlock in such a case directly follows from the proof described above.

Consider the conflict graph $G_2$ which is obtained after extending the graph G by adding request source $r_i$ and its associated incident edges. The serialization procedure for the newly added edges is described by the algorithm given below:

```
;Algorithm for serializing requests at vertex i
Algorithm serialize
  count = 1
  For each vertex j adjacent to i do
    mark i-end of the edge(i,j) with count
    mark j-end of the edge(i,j) with deg(j)
    count = count + 1
  end {for}
end {end of algorithm}
```

Application of the above algorithm to graph $G_2$ is shown in figure 4.7(b). As the original conflict graph G is deadlock free, the extended conflict graph G2' is shown to be deadlock free by proving that the newly added vertex i cannot be a part of any serialization cycle. This proof is discussed below.

Let us assume that the graph $G_2'$ has deadlock, which implies that $G_2'$ has a serialization cycle involving vertex i. Without loss of generality let us assume that vertices (1,.....,i,j,k,.....m,1) form the serialization cycle. Using the notation of section 4.5.1, such a serialization

cycle implies $(l_{H-L},\ldots,i_{H-L},j_{H-L},k_{H-L},\ldots m_{H-L},l_{H-L})$. However, as the serialization algorithm for the newly added vertex i assigns deg(i) to {i,j} and deg(k) to {k,j}, a traversal of the cycle (l,....,i,j,k,....m,l) can only result in $(l_X,\ldots,i_{-H},j_X,k_{H-L},\ldots,m_X,l_X)$, thus a contradiction at vertex i. Hence, the above extension to the edge request serialization algorithm do not introduce deadlock into the conflict graph.

When a conflict graph is to be extended by more than one request source, it is done so by extending the conflict graph one request source at a time. After each extension, the newly added vertex and its incident edges are serialized using the above algorithm. This procedure is repeated until all the request sources are added to the conflict graph.

It should be noted that the hardware modifications required to the existing design is simple and minimal for both type of extensions discussed above.

## 4.6 Applications

In this section two applications which directly make use of the general arbiter synthesis procedure are discussed.

### 4.6.1 Asynchronous system controller synthesis

As a general approach, loosely coupled systems are preferred in VLSI. In a loosely coupled system with autonomously operating components synchronization mechanism is required to control access to shared resource such as communication channel. In such applications an asynchronous system controller, also called synchronizer, is used to ensure proper operation of the system. Such controllers can be designed using the path expression synthesis procedure of Anantharaman[1] or Li[20]. Anantharaman's approach is a self timed design approach and is based on the regular expression synthesis procedure of Foster[11]. As discussed below and in [1] an important and integral part of path expression synthesis is the synthesis of its underlying conflict graph. Functionally, conflict graph of a path expression ensures that two mutually conflicting events of the system are not enabled simultaneously.

As an example, consider the synthesis of a controller for the reader-writer problem with two reader module and a writer module. Specification of the problem does not allow a reader and a writer to be active simultaneously. However, more than one reader can be active simultaneously. The above problem can be represented using path expression as shown below and the conflict graph structure for the same is shown in figure 4.8.

;Reader-writer controller

Path $R_1$ + W;

Path $R_2$ + W;

;End of definition

The conflict graph of reader-writer problem is a general graph. Thus, synthesis of the controller for reader-writer problem requires synthesis of its underlying conflict graph. The conflict graph synthesis procedure used by Anantharaman is a centralized design approach and suffers from the following disadvantages:

(i) Issues such as fairness and deadlock in the arbiter design is resolved by adjusting threshold voltage and active resistance of the circuit. Such an approach is highly susceptible to fabrication defects and designer has very little control over proper operation of the circuit.

(ii) The design procedure is not modular, thus expandability requires extensive redesign of the circuit.

The above disadvantages can be eliminated using the distributed design approach of this chapter. Thus, using general arbiter design approach of this chapter not only reliable, but also modular implementation of controller circuit can be obtained.

### 4.6.2 Petri net synthesis

Petri nets can be used to model computers, computations

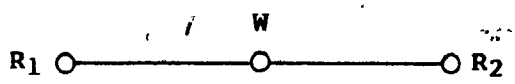



Figure 4.8 Conflict graph for reader-writer problem

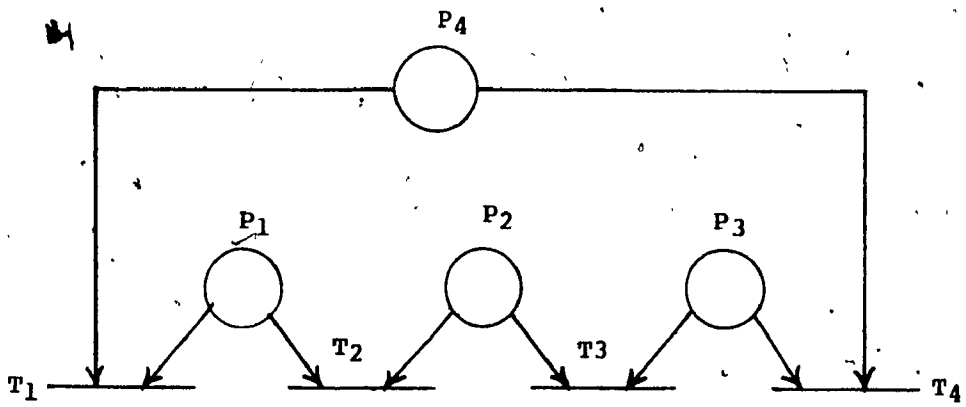



Figure 4.9 Example petri net

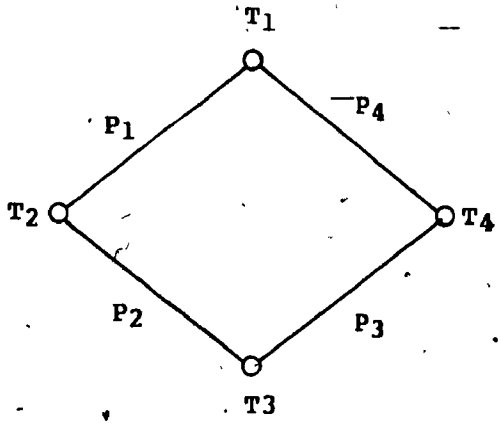



Figure 4.10 Conflict graph for petri net

and to study their performance. Patil and Denis[26] have shown how petri net can be used in the modelling and synthesis of digital system. Patil's approach to petri net synthesis is one of functional emulation of petri net in hardware. Such an approach can also be used in applications such as [4,6,24]. However it should be noted that this approach to system designing is not very efficient in general.

Patil's work on petri net synthesis[25] has addressed many issues pertaining to asynchronous implementation of petri nets. One issue that needs additional consideration is the arbitration problem which occurs during the synthesis of petri nets which have shared places. This can be described with an example as follows. Consider a petri net with four shared places connected as shown in figure 4.9. According to the firing rules of the petri net a token in shared place can be consumed by only one of the conflicting transitions. Thus a shared place represents an arbitration point in a petri net. With respect to figure 4.9 there are four arbitration points corresponding to the four shared places in the net. The arbitration structure of such a petri net can be represented using a arbitration graph, as shown in figure 4.10. Each vertex in the arbitration graph corresponds to a transition and the edges corresponds to shared places. The arbitration graph of figure 4.10 is not a complete graph. Synthesis of such a petri net requires

synthesis of underlying conflict graph. Thus the general arbiter synthesis procedure of this chapter can also used during the synthesis of petri nets. In addition, the distributed architecture of the general arbiter is suitable for petri net synthesis as petri nets by nature are modular, distributed and asynchronous.

# CHAPTER 5

# SYNCHRONOUS AND ASYNCHRONOUS DESIGN - A CASE STUDY

## 5.1 Introduction

This chapter presents a case study of synchronous and asynchronous design of general arbiters. Synchronous and asynchronous design techniques are two rivalling design techniques using two different methods to communicate information between parts of the system. Use of a particular design technique for a given problem could depend on a variety of considerations like operating environment, performance, ease of design and suitability for VLSI, etc.. Unfortunately, there is no universal rule of thumb or criterion which could be used as a guideline for selecting a particular design approach. In this chapter we present a synchronous design of general arbiters and compare it with the self timed implementation of chapter 4. The self timed general arbiter design presented in chapter 4 is based on edge request serialization algorithm. As it is possible to operate an asynchronous design in synchronous environment, the design approach of chapter 5 can also be used for synchronous design of general arbiters. But, often better algorithms can be designed by making use of the properties of the synchronous systems such as simplicity in

synchronization and a priori knowledge of delays and signal transitions. In this chapter a high performance algorithm for synchronous, distributed implementation of general arbiters is presented. During the design of synchronous distributed general arbiters the following conditions are assumed to be valid.

(1) Requests sent to the arbiter are synchronized with clock and the acknowledges from the arbiter are received synchronous to the clock.

(2) Global transmission of request signal is allowed. That is, the request signal will reach all parts of the arbiter without any time penalty(delay).

(3) There is a minimum one clock delay between R↑ and the corresponding A↑. Similarly there is a minimum of one clock delay between A↓ and the next R↑.

(4) Request and acknowledge are level signals.

## 5.2 Synchronous general arbiter design

The synchronous distributed arbiter is constructed as follows. For each edge(i,j) of the given conflict graph a 1-out of-2 local arbiter A(i,j) is associated. The arbiter A(i,j) arbitrates between the conflicting requests $r_i$ and $r_j$. If both $r_i$ and $r_j$ are raised simultaneously, then only one of the requests is locally acknowledged. The global acknowledge $a_i$, corresponding to the request $r_i$, is generated by performing a logical-AND of all the

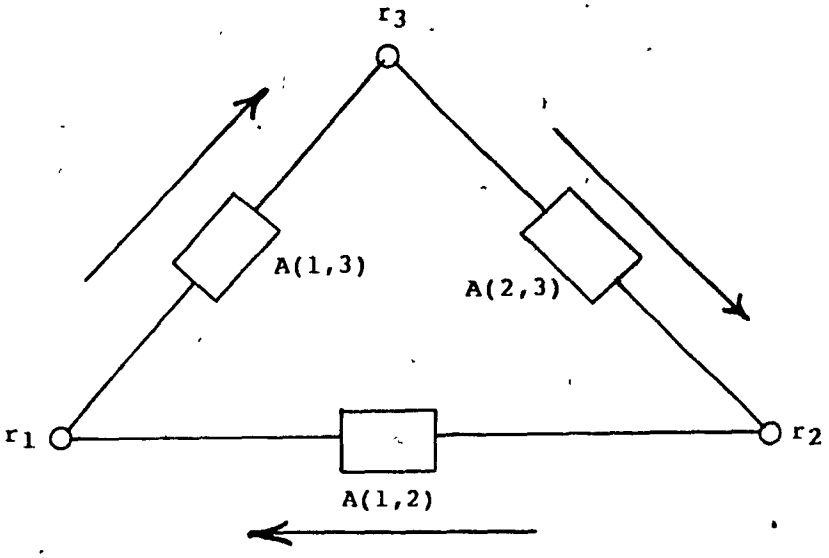



Figure 5.1 Arbitration graph with deadlock

acknowledges received from the local arbiters corresponding to the edges incident at the vertex i. An advantage of such a distributed architecture is its performance. Irrespective of the number of requests $r_j$, $1 \leq j \leq n$, conflicting with a particular request $r_i$, the distributed approach guarantees a best case resolution time equal to the local arbiter resolution time. However, the above approach can lead to deadlock due to the distributed decisions made at the local arbiters. For example, in an arbitration graph, shown in figure 5.1, with three requests, any two of them conflicting, consider the request $r_1$,$r_2$, and $r_3$ are raised simultaneously when none of the acknowledges $a_1$, $a_2$, and $a_3$ are active. Now, if A(1,2) acknowledges $r_1$, A(2,3) acknowledges $r_2$ and A(1,3) acknowledges $r_3$, the net result is a deadlock due to cyclic dependency.

The solution presented in this chapter guarantees deadlock free operation by assigning priorities to the requests. Consequently, if local arbiter A(i,j) maintains a priority $r_i > r_j$, then the local acknowledge is always sent to $r_i$ whenever both $r_i$ and $r_j$ are active simultaneously. The criteria for assigning priorities to requests is established by the following theorem.

Theorem 5.1

If priorities assigned to the request are transitive consistent (that is, $r_i > r_j \wedge r_j > r_k \Rightarrow r_i > r_k$, for any

i,j,k) the global arbitration as formed is deadlock free.

Proof : A deadlock could arise only if dependency cycle exists, which corresponds to a sequence of acknowledges in the form of $a_1, a_2, a_3, \ldots, a_k, a_1$. Such a sequence could arise only if $r_1 > r_2 > \ldots > r_n > r_1$ which is a contradiction.

In a conflict graph if $r_i$ and $r_j$ are two requests in conflict and $r_i$ has a higher priority over $r_j$, then the edge connecting i and j can be treated as a directed edge <i,j>. Thus a deadlock free solution to general arbiter is obtained by assigning priorities to the requests such that the resulting directed conflict graph is acyclic.

The arbitration operation in such a directed acyclic conflict graph G can be described as follows. Suppose that all the requests are raised simultaneously and none of the acknowledges are active. Clearly, all those requests(vertices) which do not have any incoming edge in G can be first acknowledged. Construct G' by deleting all the vertices corresponding to the serviced requests in G. Now all those requests which do not have any incoming edge in G' can be acknowledged. Extending this argument, it can be seen that order in which requests are serviced is according to the length of the longest directed path in G. The waiting time for a request source in a prioritized conflict

graph can be measured in terms of latency. The latency of a directed conflict graph is equal to the number of vertices in the longest path. It is intuitive that longer the latency, more serialization is required among requests, assuming all of them to be active. Thus, to maximize the rate at which a request can be acknowledged, the priority assignment to requests is established with the objective of minimizing the latency. Unfortunately, finding such a priority assignment is an NP-complete problem, as shown in [18].

Even though finding the minimum latency for a given conflict graph is NP-complete, a near-minimum latency can be determined using the following heuristic. This heuristic attempts to maximize the number of vertices which are allowed to generate acknowledge simultaneously (vertices of the same colour) during each iteration.

Heuristic to assign priorities to requests;

```
i = 0: G'= G
repeat
   i = i + 1
   assign priorities to all non adjacent vertices with the
    smallest degree in G'
   While there exists a set S of vertices in G' with the
    next smallest degree such that no vertex in S is
    adjacent in G' to a vertex with priority i do
```

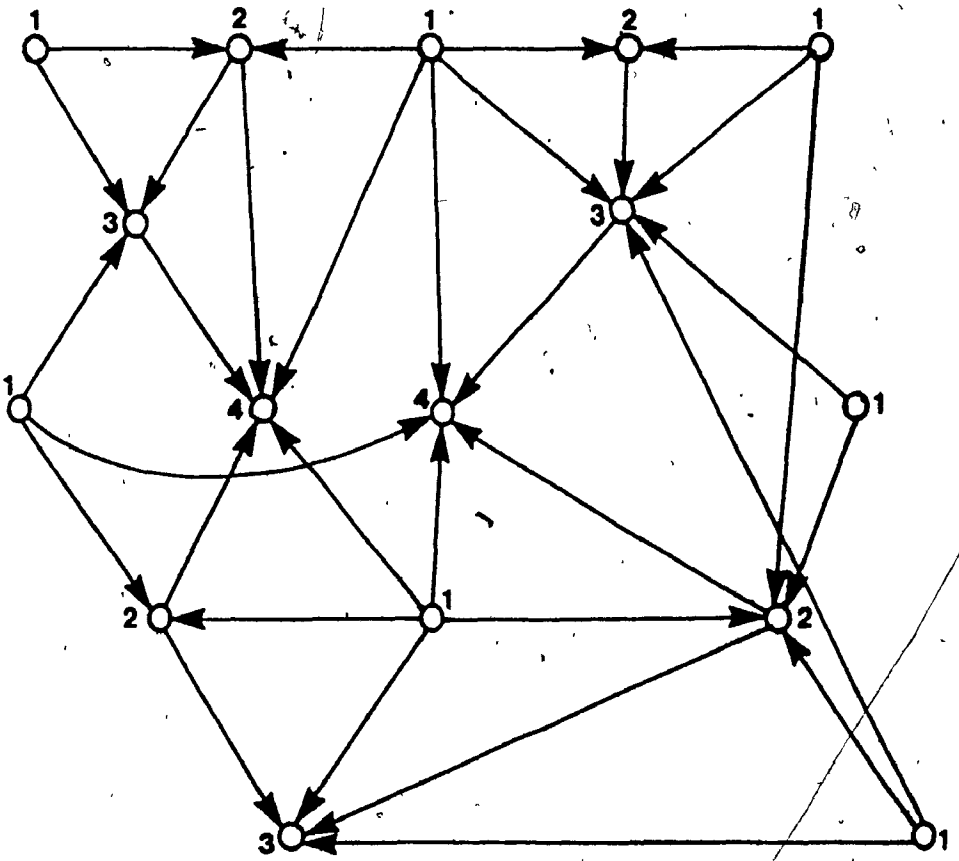



Figure 5.2 Directed, acyclic conflict graph

assign priority i to all vertices in S

delete all the verticies with priority i and the edges incident at them from G' to obtain the new G'

until G' is empty

Using the above heuristic, the latency of the conflict graph of figure 4.2 can be determined as 4, as shown in figure 5.2.

The synchronous general arbiter design for the conflict graph of figure 5.3 is shown in figure 5.4. The local arbiters used in the design is shown in figure 5.5.

The distributed architecture discussed above cannot be used in an asynchronous environment, as variations in delays can lead to deadlock. For example, consider the arbiter of figure 5.6 with priority order $r_1 > r_2 > r_3$. Let the delays from the request sources to the local arbiters be different. Let delay 1 > delay s, and delay(1-s) > arbiter resolution time. With the above operating conditions, if all the requests are raised simultaneously, and none of the acknowledges are active, then the local arbiters will acknowledge as follows. A(1,3) will acknowledge $r_3$, A(2,3) will acknowledge $r_2$, and A(1,2) will acknowledge $r_1$, thus resulting in cyclic dependency and deadlock. It should be noted that in the above case even though the requests are raised simultaneously at their respective sources the delay

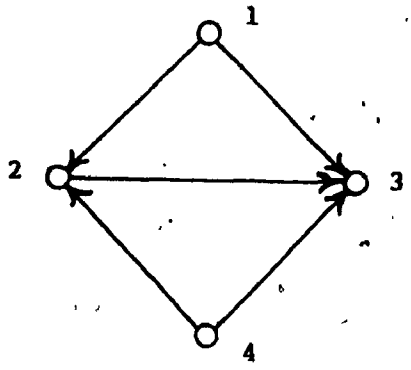



Figure 5.3 Conflict graph for synchronous arbiter

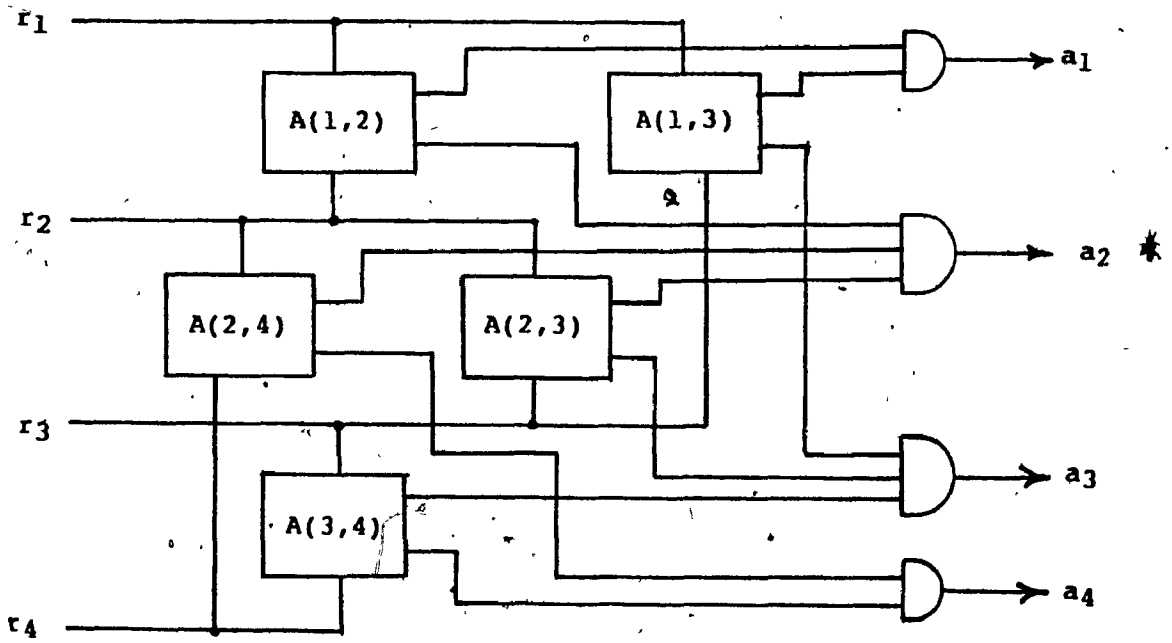



Figure 5.4 Synchronous general arbiter

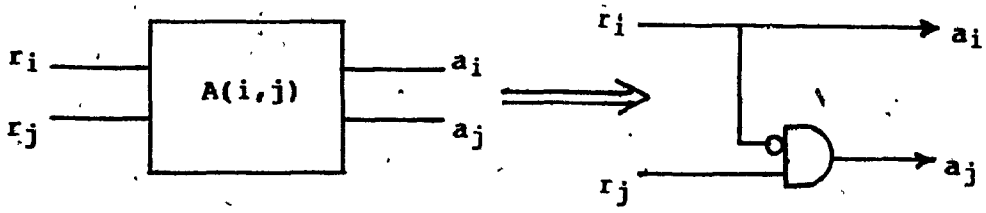



Figure 5.5 Synchronous local arbiter

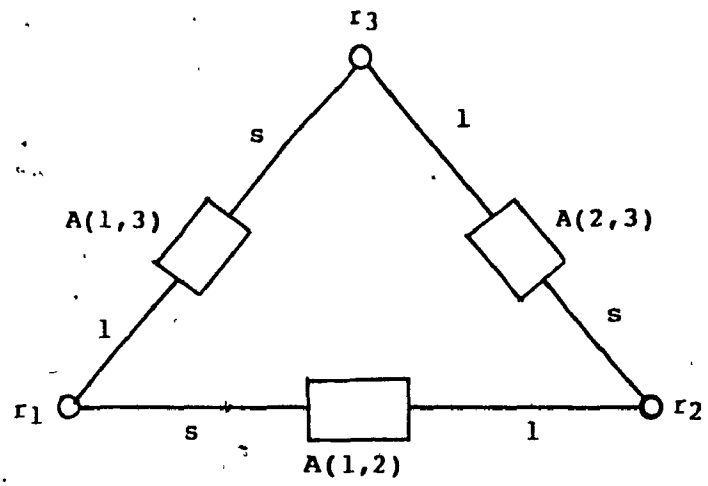



Figure 5.6 Conflict graph with unequal delays

in the communication medium leads to different affival times at the local arbiters and this leads to deadlock.

## 5.3 Arbiter fairness

In this section we discuss one of the important properties of an arbitration structures, the fairness. The local arbiters used in the synchronous design are prioritized arbiters by construction. But, under the synchronous operating conditions as listed in section 5.1, they behave as strictly fair arbiters. Such a strict fairness of the local arbiters is due to relative timing of request and acknowledge signals, as discussed in section 5.1, and the property that once an acknowledge is granted it is withdrawn only after the corresponding request is withdrawn. The synchronous distributed architecture constructed using such local arbiters is strictly fair. The concept of strict fairness used in the synchronous general arbiter is the same as discussed in section 4.4. The strict fairness of the general arbiters is due to global transmission of requests to all the local arbiters corresponding to the incident edges and the strictly fair property of local arbiter.

## 5.4 Comparison of synchronous and asynchronous design

As the synchronization requirements for the synchronous

and asynchronous implementation of general arbiters vastly differ, the communication time delay model based comparison technique of Franklin[12] cannot be used. Thus we use traditional asymptotic model to compare the two implementations. Considering the local arbiter delay as unit delay, the algorithm for synchronous distributed implementation has a best case performance of O(1) and a worst case performance of O(L), where L is the latency of the conflict graph. In contrast, the complexity of request serialization algorithm is O(N) in the best case, where N is the degree of a vertex, and it is difficult to establish the worst case complexity due to asynchronism and weak fairness of the arbiter. Thus, synchronous implementation has a better time complexity than asynchronous implementation. Also, the complexity of local arbiter used in synchronous design is less than those of asynchronous design. In conclusion, as compared to asynchronous implementation, the synchronous implementation of general arbiter has a better area, time performance.

# CHAPTER 6

## CONCLUSIONS

The need for a VLSI design approach with balanced emphasis on abstract and practical issues is often advocated in research publications. In this thesis a case study, through systolic stack, using such a design approach is presented. This study exhibits the advantage of using known conceptual concepts such as buffering in a systolic network design environment to achieve performance improvement. Like other engineering designs, a VLSI design is complete only when all the practical issues are analysed. This often neglected, but important goal is emphasized in this thesis by examining a variety of issues like maximum and minimum frequency of operation, power consumption, layout design issues like power and clock distribution etc.. In addition, the one slow systolic stack of this thesis is a modular design as shown by expandability of the stack and has a better area, time performance than Guibas stack.

Another contribution of this thesis is in the domain of asynchronous system design. Through an algorithmic approach, a synthesis procedure for self timed arbiters is presented in chapter 3. As shown, in this thesis, such an algorithmic approach provides advantages such as elegant handling of deadlock and fairness. The edge request

serialization algorithm discussed in this thesis is a general algorithm and can be used for the synthesis of 1-out of-N arbiters or general arbiters with equal ease. The request serialization approach to synchronization in an asynchronous system, discussed in chapter 4, may have a wider applicability and generality than the specific application presented in this thesis.

Finally, we have provided an additional dimension to the debate on comparison methods for asynchronous and synchronous design techniques through the case study of general arbiter. This case study not only emphasizes the role of synchronization mechanisms in system performance but also projects the limitations of the published comparison techniques.

## 6.1 Further research

While several practical and conceptual issues are addressed in this thesis, the thesis also provides scope for extensions and related further research. In this section a few problems pertinent to the material presented in this thesis are discussed.

Hierarchical abstraction and design allows flexibility in design by hiding the different architectural and implementation variations. While pursuing such a

hierarchical design approach, the designer is entrusted the responsibility to examine and experiment with different design options and select the option best suited for the given application. The approach where a designer makes all the design selection has advantages as well as disadvantages. Most importantly a designer is often limited by his knowledge and understanding of the different design choices, thus giving room for human errors and possibly inefficient designs. With the advancements in silicon compilation techniques it is no longer essential to depend solely on design engineers. Thus, one avenue for further research is to design a knowledge based silicon complier. Such a tool not only automates the design process but also eliminates inadvertent human errors. The development process of such a knowledge based silicon compiler it would also lead to associated research issues like suitable representational language for various levels in the hierarchy, representation of performance parameters required during the design seletcion process ect..

The self-timed algorithm presented in section 4.3.1.1 is an O(N) algorithm, where N is the degree of a vertex. Considering the availability of O(log N) structures for asynchronous 1-out of-N arbiters and effect of synchronization technique on asynchronous algorithm, further research is required to find efficient synchronization techniques and asynchronous algorithms for design of high

performance arbiters.

At practical level two issues, namely power distribution and logic structures, require further considerations. Power distribution affects reliable system operations by affecting noise immunity of switching devices. Thus efficient power routing techniques which minimize voltage drop along the distribution lines are necessary to implement designs which are expandable at the layout level also. Research efforts in this aspect should concentrate on developing routing techniques which make use of limited number of layers for routing.

Dynamic logic structures are not suitable for asynchronous design due to the possibility of charge loss and resulting malfunctioning of the circuit. Also, static structures like complementary logic are area, time inefficient, especially when the number of inputs are large. Thus to obtain efficient design of asynchronous systems additional developmental efforts are necessary to find area, time efficient logic structures.

## REFERENCES

(1) Anantharaman, T.S., E.M. Clarke, M.J. Foster and B. Misra - "Compiling path expressions into VLSI circuits, Technical report:CMU-CS-85-102, Carnegie-Mellon university, Pittsburg, 1984.

(2) Black, D.L. - "On the existance of delay insensitive fair arbiters : Trace theory and its limitations", Distributed Computing, Vol. 1, 1986, pp. 205-225.

(3) Chapiro, D.M. - "Globally asynchronous and locally synchronous systems", Ph.D. thesis, Stanford university, 1984.

(4) Chaudourd, C. and J.P. Elloy - "A real-time monitor and its representation by petri net", Microprocessing and Microprogramming, No. 7, 1981, pp. 241-248.

(5) Chen, M.C. - "Space time algorithm: Semantics and methodology", Ph.d. thesis, Californoa Institute of technology, 5090-TR-83, 1983.

(6) Chocron, D. and E. Cherney - "A petri net based induastrial sequencer", proceedings IEEE international conference and exhibition on induatrial control and instrumentation, 1980, pp. 18-22.

(7) Chu, T.A - "Synthesis of self timed VLSI circuits from graph specifications", VLSI memo No. 87-410, MIT, 1987.

(8) DeRuyck, D.M., L. Snyder and J.D. Unruh - "Processor displacement - an area-time trade-off method for VLSI", Proceedings of the MIT conf. on advanced research on VLSI,

1982, pp. 182-187.

(9) Fisher, A.L. and H.T. Kung - "Synchronization of large VLSI processor arrays", IEEE transcations on computers, Vol. C-34, No. 8, 1985, pp. 734-740.

(10) Folberth, O.G. - "Miniaturization of digital Si VLSI - obstacle and limits".

(11) Foster, M.J. - "Specialized silicon compiler for language recognition", Ph.D. thesis, Dept. of Computer science, Carnegie-Mellon University, Pittsburg, 1984.

(12) Franklin, M.A. and D.F. Wann - "Asynchronous and clocked staructures for VLSI based interconnection network", Ninth computer architecture symposium, 1982, pp. 50-59.

(13) Glazer, L.A., and D.W. Dobberphul - "The design and analysis of VLSI circuits", Addison-Wesley, 1985.

(14) Guibas, L.J. and F.M. Liamg - "Systolic stacks, Queues and counters", 1982 conferance on advanced research in VLSI, MIT, pp. 155-164.

(15) Hoare, C.A.R. - "Communicating sequential processes", Prentice-Hall international, 1985.

(16) Kung, H.T. - "Why Systolic architectures?", IEEE Computer, January, 1982, pp. 37-46.

(17) Kung, S.Y. - "VLSI array processors", Prentice Hall, 1988.

(18) Li, H.F., R. Jayakumar, and R.N. Prasad - "General arbiters for VLSI : Designs and complexities", Submitted for publication to IEEE transactions on computers.

(19) Li, H.F., D.K. Probst and R.N. Prasad - "Traversing the

VLSI design hierarchy for a new, fast systolic stack", IEE proceedings, Vol. 135, Pt. E, No. 1, 1988, pp. 25-40.

(20) Li, W. and P.E. Lauer - "A VLSI implementation of COSY", Tech. rep. ASM/121, Computing Laboratory, The University of Newcastle upon tyne, 1984.

(21) Mead, C and L. Conway - "Introduction to VLSI systems", Addison-Wesley, 1980.

(22) Moldovan, D.I. and J.A.B. Fortes - "Partitioning and mapping of algorithm into fixed size systolic arrays", IEEE transactions on computers, 1986, C-35, pp. 1-12.

(23) Molnar, C.E., T.P. Fang and F.U. Rosenberger - "Synthesis of delay insensitive modules", proceedings of 1985 chapel hill conf. on very large scale integration, 1985, pp. 67-86.

(24) Murata, T., N. Komoda, K. Masumoto and K. Haruna - "A petri net based controller for flexible and maintainable sequence control and its application in factory automation", IEEE transaction on industrial electronics, Vol. 33, No. 1, 1986, pp. 1-8.

(25) Patil, S.S. - "Circuit implementation of petri net", Computer structures group memo 73, Project MAC, MIT, 1972.

(26) Patil, S.S., and J.B. Denis - "The description and realization of digital systems", COMPCON 72, pp. 223-226.

(27) Pechoucek, M. - "Anamalous response times of input synchronizers", IEEE transactions on Computers, Vol. C-25, No. 2, 1976.

(28) Plummer, W.W. - "Asynchronous arbiters", IEEE

transactions on computers, C-21, No. 1, 1972, pp. 37-42.

(29) Prasad, R.N. - "VLSI implementation of petri nets", Course report for ENCS 657, Department of computer science, Concordia University, 1986.

(30) Rem, M. - "The VLSI challenge : Complexity bridling", VLSI 81, proceedings of international conf. on VLSI, Academic press, 1981, pp. 65-73.

(31) Sanjay Dhar, M.A. Franklin and D.F. Wann - "Reduction of clock delays in VLSI structures", pp. 778-783, 1984.

(32) Seitz, C.L. - "Ideas about arbiters", Lambda, First quarter, 1980, pp. 10-114.

(33) Seitz, C.L. - "Self timed VLSI systems", Caltech conf. on VLSI, 1979, pp. 345-355.

(34) Sequin, C.H. - "Managing VLSI complexity - An overview", Proceedings IEEE, Vol. 71, 1983, pp. 149-166.

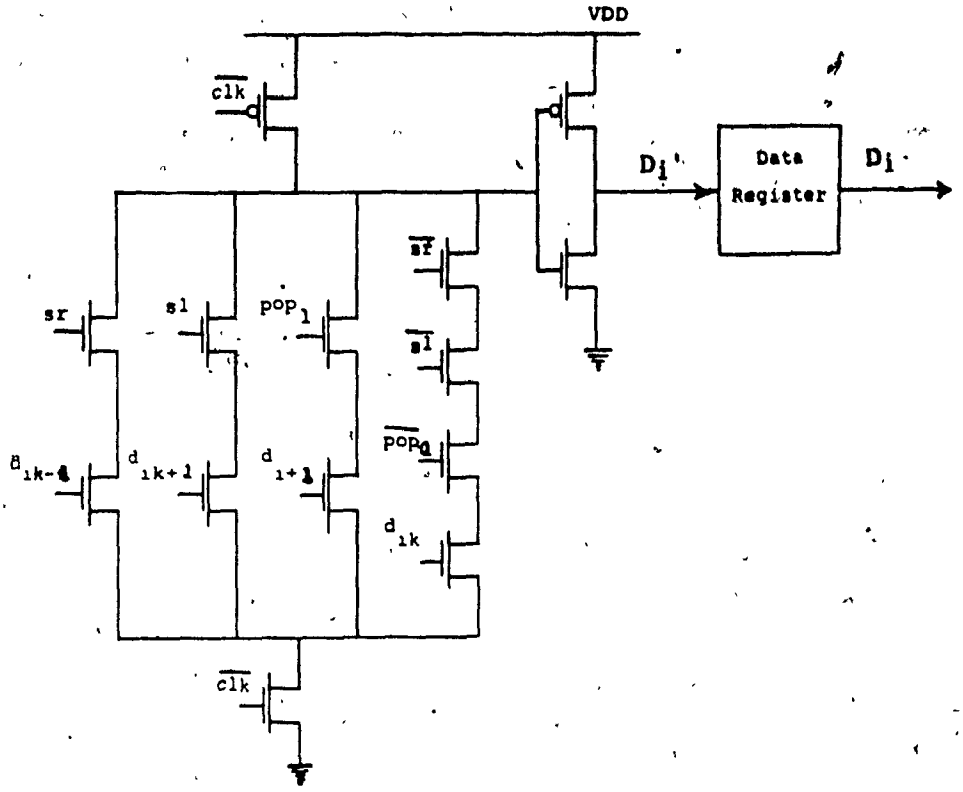
(35) Siewiorek, D.P., C.G. Bell and A. Newell - "Computer structures : Structure, Principles and examples", McGraw Hill, 1982.

(36) Snyder, L - "Supercomputers and VLSI : The effect of large scale integration on computer architecture", Advances in computers, Vol. 23, pp. 1-33.

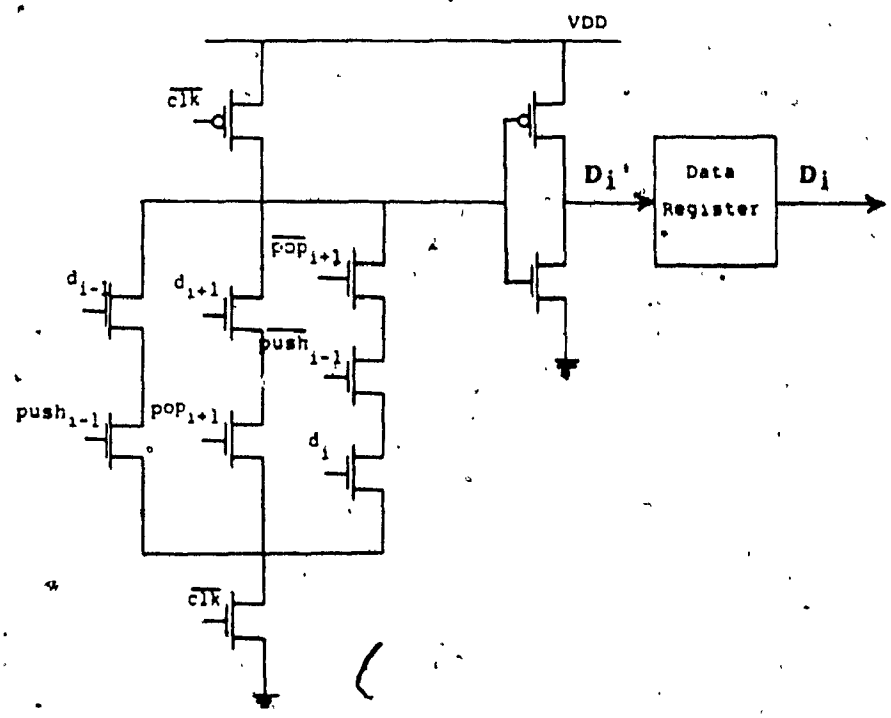
(37) Stefik, M., D.G. Bobrow and A. Bell - "The partitioning of concerns in digital system design", 1982 conference on advanced research in VLSI, MIT, pp. 43-52.

(38) Weste, N. and K. Esharghian - "Principles of CMOS VLSI design - A system perspective, Addison-Wesley, 1985.

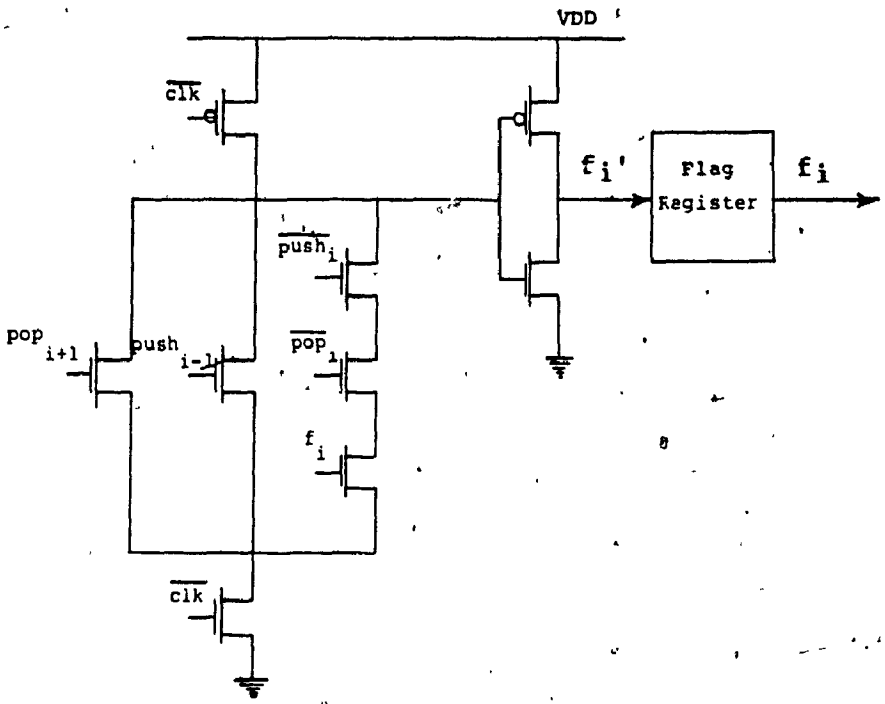
# APPENDIX I




Interface slice data cell




Storage slice data cell




Storage slice flag cell

# APPENDIX-II

```
;*****************************************************************************
;
;                              RNL COMMAND FILE
;
;*****************************************************************************
;
;
(load "uwstd.l")
(load "uwsim.l")
(read-network "proj.b")
(log-file "test")
(chflag '(c1 datain dataout e2 e1 f.0 f.1 f.2 f.3 d.0.1.1 d.0.2.1 d.0.3.1
d.1.1.1 d.1.2.1 d.1.3.1 d.2.1.1 d.2.2.1 d.2.3.1 d.3.1.1 d.3.2.1 d.3.3.1
push.0 push.1 push.2 push.3 pop.0 pop.1 pop.2 pop.3 push pop  sr sl s_empty s_full))
(def-report '("" tab tab c1 datain dataout sr sl  s_empty  s_full newline tab tab e2 e1 f.0 f.1 f.2 f.3
newline tab tab push.0 push.1 push.2 push.3
newline tab tab pop.0 pop.1 pop.2 pop.3 newline tab tab d.0.1.1 d.0.2.1
d.0.3.1 newline tab tab d.1.1.1 d.1.2.1 d.1.3.1 newline tab tab d.2.1.1 d.2.2.1 d.2.3.1 newline tab tab
d.3.1.1 d.3.2.1 d.3.3.1))
l cf.4
h cpop.4
l pop.4 d.4.1.1
(defun one()
(h '(c1))
(s '())
(l '(c1))
(s '())
)
(defun cycle(n)
(repeat i 1 n
(h '(datain))
(one)
(l '(datain))
(one)
)
)
l rest
(one)
h rest
h push
(cycle 2)
l push
h pop
(cycle 3)
exit
;
;
;*****************************************************************************
```

```
;*****************************************************************************
;
;                          RNL SOURCE CODE
;
;*****************************************************************************
;
(node f epush epop e1 sf csf cf ce1 e2 cpop pop csr sr ce1 s1 d push cpush s_empty)
(node c1 c1- rest-datain observ oe cse ce2 enlgemty cpush cepop mf dataout s_full resth)

;-----------------------------------------------
;Macro definitions
;-----------------------------------------------
(macro rstmsff (in out clk clk-)          ; Reset type of master-slave flip-flop
(local a)
(rslatch in a clk clk-)
(rslatch a out clk- clk)
)
;-----------------------------------------------
(macro rshlatch (in out clk clk-)                 ; Latch with set facility
(local a)
(clkinv a in clk clk-)
(cnand out a rest)
(clkinv a out clk- clk)
)
;-----------------------------------------------
(macro rshmsff(in out clk clk-)                  ; Set type of master-slave flip-flop
(local a)
(rshlatch in a clk clk-)
(rshlatch a out clk- clk)
)
;-----------------------------------------------
(macro rslatch (in out clk clk-)                   ; latch with reset
(local a)
(clkinv a in clk clk-)
(cnor out a resth)
(clkinv a out clk- clk)
)
;-----------------------------------------------
(macro latch (in out clk clk-)                    ; Latch without set or reset
(local a)
(clkinv a in clk clk-)
(cinvert out a)
(clkinv a out clk- clk)
)
```

```
;-------------------------------------------
(macro msff (in out clk clk-)                ; Master-slave flip-flop without set or reset
(local a)
(latch in a clk clk-)
(latch a out clk- clk)
)
;*****************************************************************************
;
;    Slice-0 logic ; the input output Slice logic
;
;-----------------------------------------------------------------------------
;
; Slice-0 flag logic
;
;-----------------------------------------------------------------------------
( macro state-0(pc1 clk clk-)
(local invc invb invc  pe pa1 pb pc  e1 b1 c1 int pc1 pc2 pc3 pc4 pb1 pb2 pb3 pa1 pe2 pe3 pe4 pe5 pe6 pb4 pb5 pa6 pa7)
(cinvert pc1 e1)
(cinvert (e2 e2)
(cinvert (f.1 f.1)
;-----------------------------------------------------------------------------
;logic for input - e2
;-----------------------------------------------------------------------------
(etrans epush pc1 invc)
(etrans csf pc2 pc1)
(etrans e2 pc3 pc2 )
(etrans ce1 pc pc3)
(etrans epop pc4 invc)
(etrans e1 pc pc4)
(etrans cepush pc5 invc)
(etrans cepop pc6 pc5)
(etrans e2 pc6 pc)
(etrans pc1 gnd pc)
(ptrans pc1 vdd invc)
(cinvert c1 invc)
;-----------------------------------------------------------------------------
;logic for input - e1
;-----------------------------------------------------------------------------
(etrans epush pb1 invb)
(etrans csf pb2 pb1)
(etrans e2 pb pb2)
(etrans epop pb3 invb)
(etrans f.1 pb pb3)
(etrans cepush pb4 invb)
(etrans cepop pb5 pb4)
(etrans e1 pb pb5)
(etrans pc1 gnd pb)
```

```
(ptrans pc1 vdd invb)
(cinvert b1 invb)
;-------------------------------------------------
;Logic for input - f.0
;-------------------------------------------------
(etrans csf pa1 inva)
(etrans e1 pa4 pa1)
(etrans sf pa4 inva)
(etrans epush pa pa4)
(etrans epop pa2 inva)
(etrans erlyemty pa3 pa2)
(etrans e2 pa pa3)
(etrans cepush pa5 inva)
(etrans cepop pa6 pa5)
(etrans f.1 pa pa6)
(ptrans pc1 gnd pa)
(ptrans pc1 vdd inva)
(cinvert a1 inva)
;-------------------------
; Flag register bits
;-------------------------
(rstmsff g1 e2 clk clk-)
(rstmsff b1 e1 clk clk-)
(rstmsff a1 f.1 clk clk-)
)                              ; end of state-0 macro
;
;-------------------------------------------------
;Logic for Stack full signal
;
(macro stkful(clk clk )
(local b a c e g h m n)
(etrans e1 c a)
(etrans f.1 b a)
(etrans epop c b)
(etrans f.(+ i 2) m c)
(etrans f.(+ i 1) h m)
(etrans epush g h)
(etrans clk- gnd g)
(ptrans clk- vdd d)
(cinvert m a)
(rslatch m s_full clk- clk)
(rslatch s_full sf clk clk-)
(cinvert csf sf)
)
```

```
;-------------------------------------------------------------
;Logic for Stack empty signal
;-------------------------------------------------------------
(macro stkemty(clk clk )
(local a c e)
(pulldown erlyemty (epop 6 2) (cf.(+ i 1) 6 2) (c2 6 2))
(pulldown erlyemty (ce2 6 2) (se 6 2) (cepush 6 2))
(ptrans epop c erlyemty 8 2)
(ptrans cf.(+ i 1) c erlyemty 8 2)
(ptrans e2 c erlyemty 8 2)
(ptrans ce2 vdd c 8 2)
(ptrans se vdd c 8 2)
(ptrans cepush vdd c 8 2)
(cinvert a erlyemty)
(rshlatch a s_empty clk- clk)
(rshlatch s_empty se clk clk-)
(cinvert cse se)
)
;-------------------------------------------------------------
;Logic for forced pop - pop.(+ 1 i)
;-------------------------------------------------------------
(macro frpop()
(pulldown cpop.i (epop 4 2) (e2 4 2 ))
(ptrans epop vdd cpop.i 4 2)
(ptrans e2 vdd cpop.i 4 2)
(cinvert pop.i cpop.i)
)
;-------------------------------------------------------------
; Logic for push()
;-------------------------------------------------------------
(macro push()()
(pulldown cpush.i (epush 6 2) (f.i 6 2) (cf.(+ i 2) 6 2))
(ptrans epush vdd cpush.i 4 2)
(ptrans f.i vdd cpush.i 4 2)
(ptrans cf.(+ i 2) vdd cpush.i 4 2)
(cinvert push.i cpush.i)
)
;-------------------------------------------------------------
;Logic for shift right
;-------------------------------------------------------------
(macro shift()
(pulldown csr (epush 4 2) (csf 4 2))
(ptrans epush vdd csr 4 2)
(ptrans csf vdd csr 4 2)
(cinvert sr csr)
```

```
;
;logic for shift left
;
(pulldown csl (*pop 4 j) (csr 4 j))
(ptrans *pop vdd c.i 4 j)
(ptrans csr vdd csl 4 j)
(cinvert sl csl)
)
;-  ---  --  ---  ---  --  -
;logic for data register of Input output slice
;-  -  -  -  -  -  -  -  -  -
(macro data0((clk fcl fcl )
(local a b c e x g h m n)
(etrans sr b.i.k.j a.i.k.j)
(etrans d.i.( k 1).j m.i.k.j b.i.k.j)
(etrans sl c.i.k.j a.i.k.j)
(etrans d.i.(+ k 1).j m.i.k.j c.i.k.j)
(etrans pop.i e.i.k.j a.i.k.j)
(etrans d.(+ 1 i).k.j m.i.k.j e.i.k.j)
(etrans csr x.i.k.j a.i.k.j)
(etrans csl g.i.k.j x.i.k.j)
(etrans cpop.i h.i.k.j g.i.k.j)
(etrans d.i.k.j m.i.k.j h.i.k.j)
(etrans clk gnd m.i.k.j)
(ptrans clk vdd a.i.k.j)
(cinvert m.i.k.j a.i.k.j)
(mff m.i.k.j d.i.k.j fcl fcl-)
)

;*****************************************************
;
;  Logic for Slice-1 to n
;
;  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
;
; Logic for slice-1 flag bit
;
;-  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -
(macro state-1(clk clk )
(local a b c e h)
(etrans f.i b a)
(etrans (push.i c h)
(etrans push.(- i 1) c a)
(etrans pop.(+ i 1) c a)
(etrans cpop.(- i 1) a h)
(etrans clk gnd c)
(ptrans clk vdd h)
```

```
(cinvert e h)
(rstmsff e f.i clk clk-)
(cinvert cf.i f.i)
)
;-------------------------------------
;Logic for data cell Di - for slice 1 to n
;-------------------------------------
(macro data1(clk fcl fcl-)
(local a b c dn e f g)
(ntrans d.(- i 1).k.j  b.i.k.j  a.i.k.j )
(etrans push.(- i 1)  f.i.k.j  b.i.k.j )
(etrans d.(+ i 1).k.j  c.i.k.j  a.i.k.j )
(etrans pop.(+ i 1)  f.i.k.j  c.i.k.j )
(ntrans d.i.k.j  dn.i.k.j  a.i.k.j )
(etrans cpop.(+ i 1)  e.i.k.j  dn.i.k.j )
(ntrans cpush.(- i 1)  f.i.k.j  e.i.k.j )
(etrans clk gnd f.i.k.j )
(ptrans clk vdd a.i.k.j )
(cinvert g.i.k.j  a.i.k.j )
(msff g.i.k.j  d.i.k.j  fcl fcl g)
)
;-------------------------------------
; Logic for Pushi and Popi - the rewrite rule logic for slice 1 to n
;-------------------------------------
(macro pushpop()
(cnand cpop.i f.i cf.(- i 1) cf.(- i 2))
(cinvert pop.i cpop.i)
(cnand cpush.i cf.(+ i 1) f.i f.(- i 1))
(cinvert push.i cpush.i)
)
;-------------------------------------
;Logic for Fi - Flag bit for slice 1 to n
;-------------------------------------
(macro state-n(clk clk-)
(local a b c e g)
(etrans pop.(+ i 1) a.i e.i)
(etrans push.(- i 1) a.i e.i)
(etrans f.i a.i b.i)
(etrans cpush.i b.i c.i)
(etrans cpop.i e.i e.i)
(etrans clk gnd e.i)
(ptrans clk vdd e.i)
(cinvert g.i e.i)
(rstmsff g.i f.i clk clk-)
(cinvert cf.i f.i)
)
```

```
;  Definition of the syster using the above macros
;
;-------------------------------------------------------------------
; Interface module definition
;
(rslatch datain d.0.0.1 cl cl-)              ; Datain latch
(rslatch push epush cl cl-)                  ; Insert(push) latch
(rslatch pop epop cl cl-)                    ; Delete(pop) latch
(rslatch d.0.1.1 dataout cl cl-)             ; Dataout latch
(cinvert resth rest)
(cinvert cepop epop)          ; A 'c' behind a signal name indicates the complement of the signal
(cinvert cepush epush)
(cinvert cl- (cl 10 2))
;
;Input-Output Slice definition
;
(repeat i 0 0                    ; i = Slice number, k = datacell number, j = number of bits
(repeat k 1 3
(repeat j 1 1
(data0 cl- cl cl)
)))
(repeat i 0 0
(state 0 cl- cl cl)                ; Flag bit for Input Output slice
(stkempty cl cl)                  ; Stack-empty module
(stkful cl cl)                    ; Stack full module
(ripop)                          ; Input Output slice rewrite rule signals
(shift)
(push0)
)
;  Slice 1 to n definition
;
(repeat i 1 3                      ; Rewrite signals for slices 1 to n
(pushpop)
)
;(repeat i 1 1
(repeat i 1 1                    ; Flag bit for Slice 1
(state-1 cl- cl)
)
;
(repeat i 1 3                     ; Datacell for slice 1 to n
(repeat k 1 3
(repeat j 1 1
(data1 cl- cl cl)
)))
(repeat i 2 3                     ; Flag bit for slice 2 to n
(state n cl- cl)
)
```

# APPENDIX–III

```
********01/31/86 ********  SPICE 2G.6    3/15/83 *********06:54:40******

CLOCK DRIVERS

****     INPUT LISTING                    TEMPERATURE =   27.000 DEG C

******************************************************************************


.SUBCKT DRIVER 1 2 3
M1 2 1 3 3 PMOD W=45U L=5U  AD=300P AS=300P PS=130U PD=130U
M2  2 1 0 0 NMOD W=20U L=5U AD=180P AS=180P PS=80U PD=80U
.ENDS DRIVER
.SUBCKT DRIVER2 1 2 3
M1 2 1 3 3 PMOD W=160U L=5U  AD=800P AS=800P PS=350U PD=350U
M2  2 1 0 0 NMOD W=60U L=5U AD=380P AS=380P PS=150U PD=150U
.ENDS DRIVER2
.SUBCKT DRIVER3 1 2 3
M1 2 1 3 3 PMOD W=110U L=5U  AD=630P AS=630P PS=250U PD=250U
M2  2 1 0 0 NMOD W=45U L=5U AD=310P AS=310P PS=130U PD=130U
.ENDS DRIVER3
.SUBCKT CLKLD 1 2 3
M1 2 1 3 3 PMOD W=30U L=5U AD=230P AS=230P PS=100U PD=100U
M2 2 1 0 0 NMOD W=10U L=5U AD=50P AS=130P PS=60U PD=30U
.ENDS CLKLD
.SUBCKT CLK6 1 5
X1 1 2 5 CLKLD
X2 1 3 5 CLKLD
X3 1 4 5 CLKLD
X4 1 6 5 CLKLD
X5 1 7 5 CLKLD
X6 1 8 5 CLKLD
.ENDS CLK6
.SUBCKT PCL 1 2 3
M1 2 1 3 3 PMOD W=15U L=5U AD=155P AS=155P PS=70P PD=70P
M2 2 1 0 0 NMOD W=10U L=5U AD=50P AS=130P PS=60U PD=30U
.ENDS PCL
.SUBCKT PCL6 1 5
X1 1 2 5 PCL
X2 1 3 5 PCL
X3 1 4 5 PCL
X4 1 6 5 PCL
X5 1 7 5 PCL
X6 1 8 5 PCL
.ENDS PCL6
.MODEL PMOD PMOS LEVEL=2 VTO=-0.9 KP=9.75U GAMMA=0.634 PHI=0.612 LAMBDA=3.0E-2
+ RD=2.0 RS=2.0
+ CBD=2.0E-14 CBS=2.0E-14 IS=1.0E-14 PB=0.7 CGSO=2.44E-10 CGDO=2.44E-10
+ CGBO=2.0E-12 RSH=75.0
+ CJ=1.54E-4 MJ=0.5 CJSW=4.37E-10 MJSW=0.5 JS=4.19E-10 TOX=8.5E-8 NSUB=1.98E+15
+ TPG=1.0 XJ=9.0E-7
+ LD=6.0E-7 UO=240.0 UCRIT=6.44E+4 UEXP=0.139 VMAX=7.33E+4  XQC=0.4

.MODEL NMOD NMOS LEVEL=2 VTO=0.9 KP=3.05E-5 GAMMA=1.592 PHI=0.695 LAMBDA=1.0E-2
+ RD=2.0 RS=2.0
+ CBD=2.0E-14 CBS=2.0E-14 IS=1.0E-14 PB=0.7 CGSO=2.84E-10 CGDO=2.84E-10
+ CGBO=2.0E-12 RSH=15.0 CJ=3.44E-4
+ MJ=0.5 CJSW=1.09E-9 MJSW=0.5 JS=1.37E-5 TOX=8.5E-8 NSUB=9.92E+15 TPG=1.0
+ XJ=1.0E-6 LD=7.0E-7
+ UO=750.0 UCRIT=1.23E+5 UEXP=0.022 VMAX=4.92E+5  XQC=0.4

VDD 5 0 DC 5
VIN 1 0 DC
X1 1 11 5 DRIVER
X2 1 3 5 DRIVER2
X7 1 4 5 DRIVER3
.OPTIONS   VNTOL=5E-3   RELTOL=5E-3 ABSTOL=5E-3
.OPTIONS NOMOD
.DC VIN 0 5 0.2
.PLOT DC V(11)
.PLOT DC V(3)
.PLOT DC V(4)
.END
```

01/31/86 ********** SPICE 2B.6 3/15/83 ********** 06:54:40

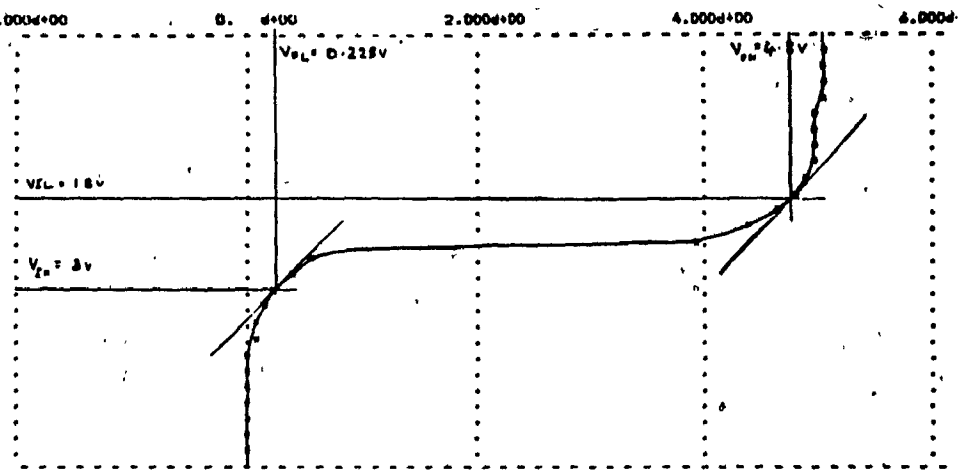
CLOCK DRIVERS

**** DC TRANSFER CURVES TEMPERATURE = 27.000 DEG C

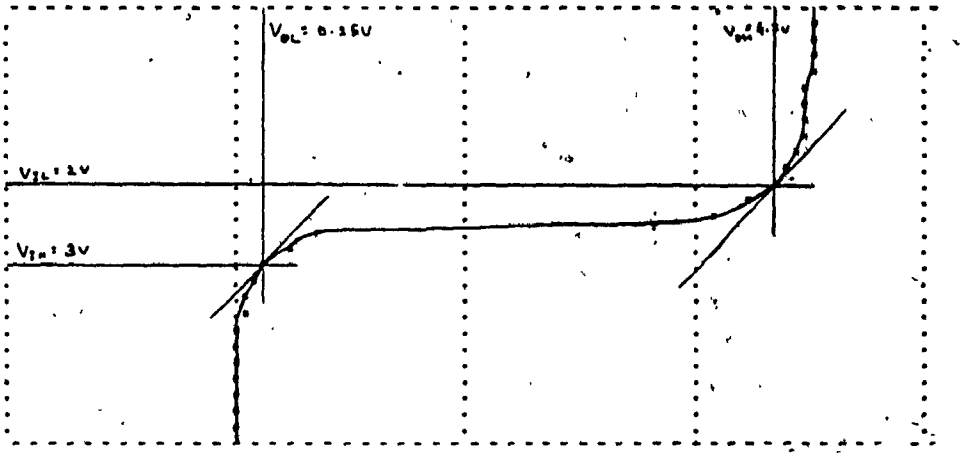
**********

X

| VIN | V(11) |
|---|---|
| 0. | 5.023d+00 |
| 2.000d-01 | 5.001d+00 |
| 4.000d-01 | 5.000d+00 |
| 6.000d-01 | 5.000d+00 |
| 8.000d-01 | 4.999d+00 |
| 1.000d+00 | 4.989d+00 |
| 1.200d+00 | 4.965d+00 |
| 1.400d+00 | 4.923d+00 |
| 1.600d+00 | 4.858d+00 |
| 1.800d+00 | 4.762d+00 |
| 2.000d+00 | 4.619d+00 |
| 2.200d+00 | 4.391d+00 |
| 2.400d+00 | 3.930d+00 |
| 2.600d+00 | 5.594d-01 |
| 2.800d+00 | 3.701d-01 |
| 3.000d+00 | 2.283d-01 |
| 3.200d+00 | 1.391d-01 |
| 3.400d+00 | 8.059d-02 |
| 3.600d+00 | 4.165d-02 |
| 3.800d+00 | 1.726d-02 |
| 4.000d+00 | 4.102d-03 |
| 4.200d+00 | -4.414d-05 |
| 4.400d+00 | -1.073d-05 |
| 4.600d+00 | -3.981d-05 |
| 4.800d+00 | -3.475d-06 |
| 5.000d+00 | -1.413d-05 |

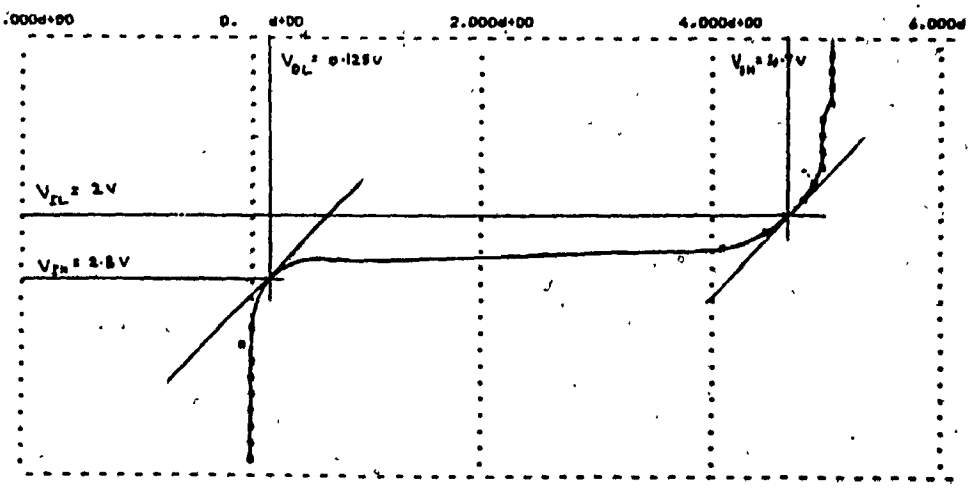



| VIN | V(3) |
|---|---|
| 0. | 5.023d+00 |
| 2.000d-01 | 5.001d+00 |
| 4.000d-01 | 5.000d+00 |
| 6.000d-01 | 5.000d+00 |
| 8.000d-01 | 4.999d+00 |
| 1.000d+00 | 4.991d+00 |
| 1.200d+00 | 4.970d+00 |
| 1.400d+00 | 4.935d+00 |
| 1.600d+00 | 4.881d+00 |
| 1.800d+00 | 4.801d+00 |
| 2.000d+00 | 4.683d+00 |
| 2.200d+00 | 4.504d+00 |
| 2.400d+00 | 4.184d+00 |
| 2.600d+00 | 7.034d-01 |
| 2.800d+00 | 4.572d-01 |
| 3.000d+00 | 2.754d-01 |
| 3.200d+00 | 1.660d-01 |
| 3.400d+00 | 9.608d-02 |
| 3.600d+00 | 4.960d-02 |
| 3.800d+00 | 2.054d-02 |
| 4.000d+00 | 4.864d-03 |
| 4.200d+00 | -6.631d-05 |
| 4.400d+00 | -1.510d-05 |
| 4.600d+00 | -5.574d-05 |
| 4.800d+00 | -5.167d-06 |
| 5.000d+00 | -1.989d-05 |



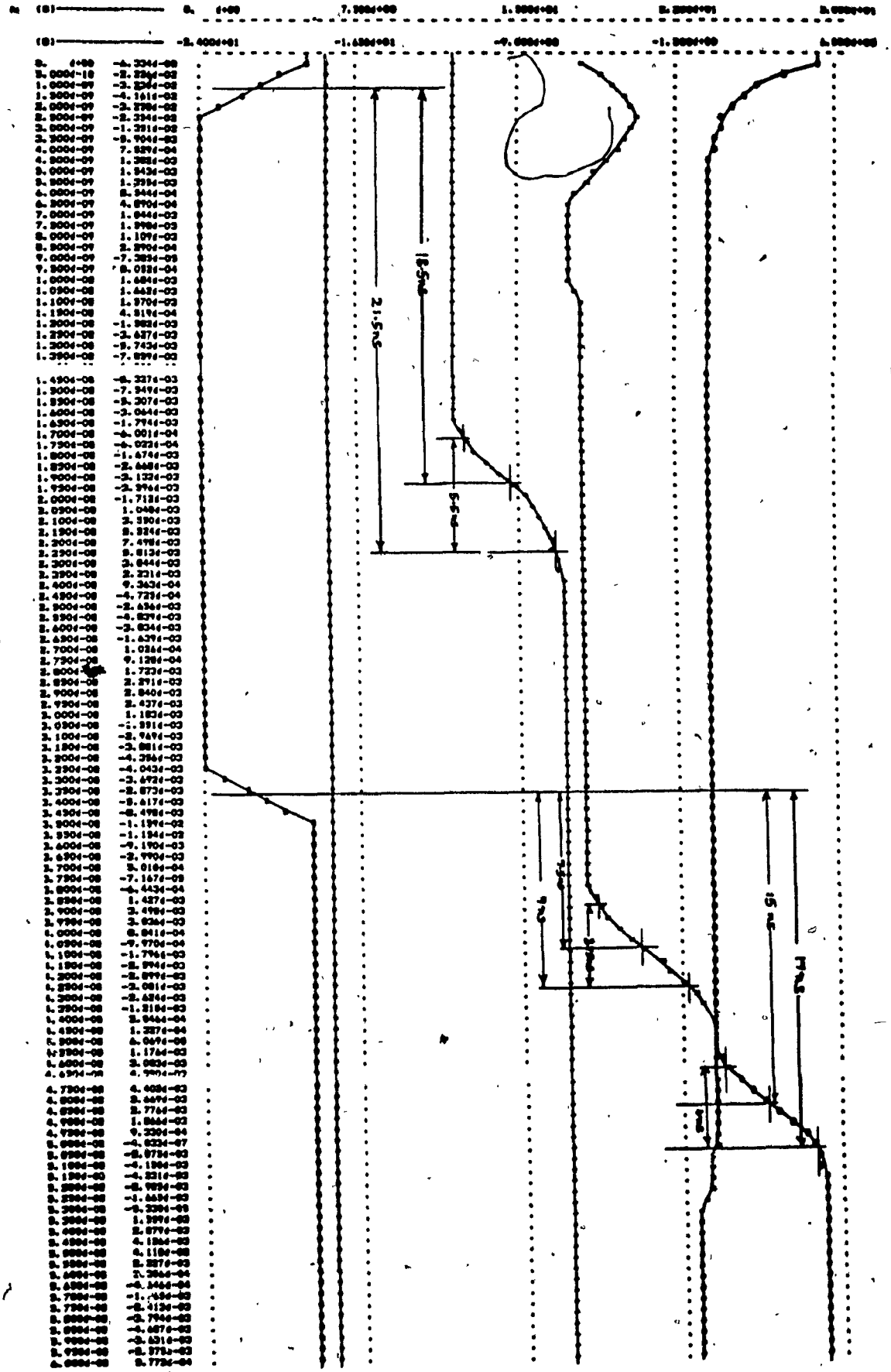

X

| VIN | V(4) |
|---|---|
| 0. | 5.023d+00 |
| 2.000d-01 | 5.001d+00 |
| 4.000d-01 | 5.000d+00 |
| 6.000d-01 | 5.000d+00 |
| 8.000d-01 | 4.999d+00 |
| 1.000d+00 | 4.990d+00 |
| 1.200d+00 | 4.968d+00 |
| 1.400d+00 | 4.929d+00 |
| 1.600d+00 | 4.870d+00 |
| 1.800d+00 | 4.782d+00 |
| 2.000d+00 | 4.652d+00 |
| 2.200d+00 | 4.450d+00 |
| 2.400d+00 | 4.068d+00 |
| 2.600d+00 | -7.068d-01 |
| 2.800d+00 | -7.085d-01 |
| 3.000d+00 | -6.769d-01 |
| 3.200d+00 | -5.483d-01 |
| 3.400d+00 | 8.428d-04 |
| 3.600d+00 | -6.300d-02 |
| 3.800d+00 | 7.842d-03 |
| 4.000d+00 | 3.004d-03 |
| 4.200d+00 | 4.541d-06 |
| 4.400d+00 | -2.641d-05 |
| 4.600d+00 | -7.101d-07 |
| 4.800d+00 | -2.744d-06 |
| 5.000d+00 | -8.901d-06 |



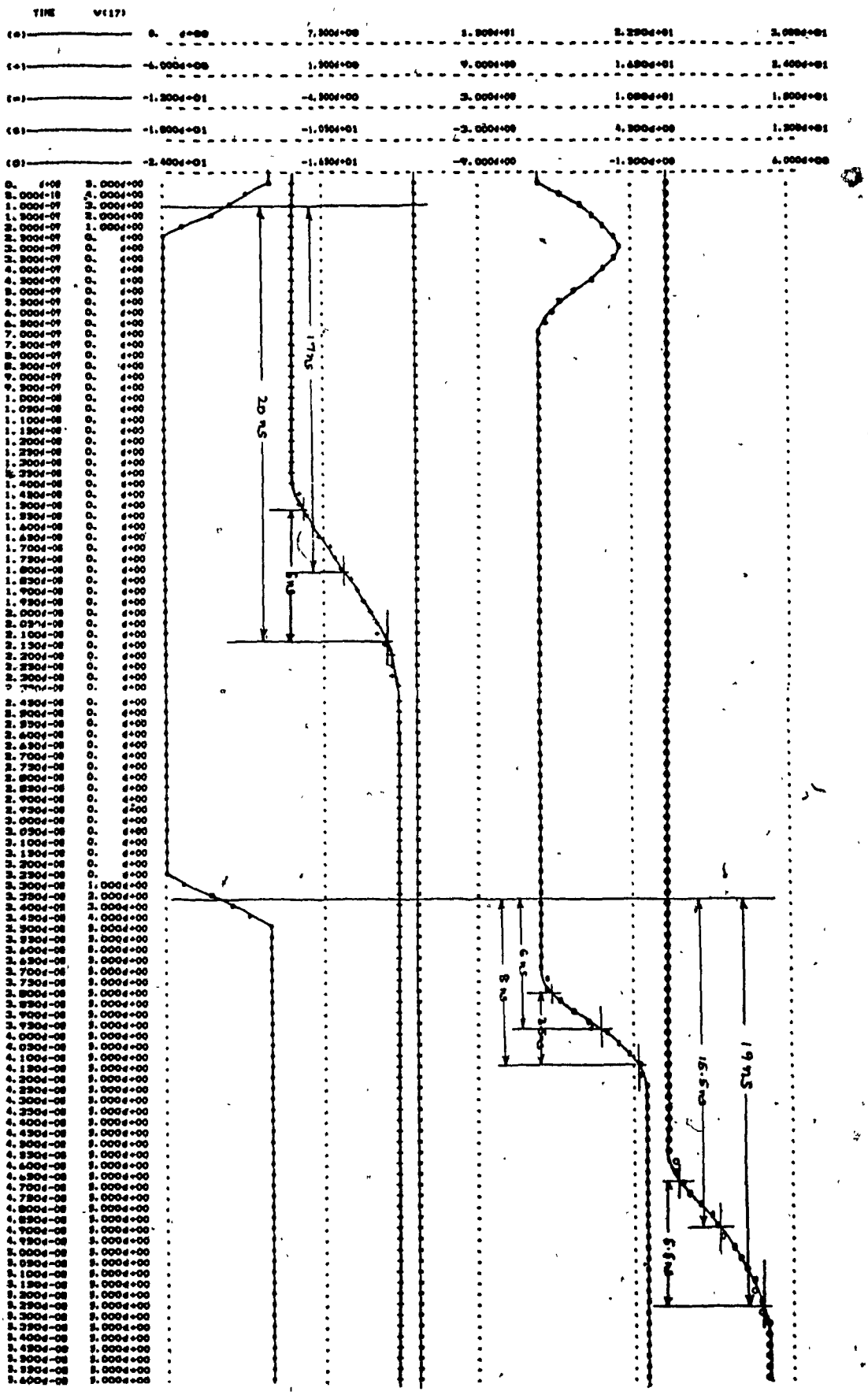
```
DATACELL FOR BLOCK 1 TO N

****      INPUT LISTING                    TEMPERATURE =   27.000 DEG C

******************************************************************************


.SUBCKT INVERTER 1 2 3
M1 2 1 3 3 PMOD W=30U L=5U  AD=225P AS=225P PS=100U PD=100U
M2  2 1 0 0 NMOD W=10U L=5U AD=130P AS=130P PS=60U PD=60U
.ENDS INVERTER
.SUBCKT LOAD4 4 5
 X4 4 6 5 LOAD
 X5 4 7 5 LOAD
 X6 4 8 5 LOAD
 X7 4 9 5 LOAD
.ENDS LOAD4
.SUBCKT CNAND3 1 2 3 4 5
 M1 4 1 5 5 PMOD W=20U L=5U AD=150P AS=150P PS=75U PD=75U
 M2 4 2 5 5 PMOD W=20U L=5U AD=150P AS=150P PS=75U PD=75U
 M3 4 3 5 5 PMOD W=20U L=5U AD=150P AS=150P PS=75U PD=75U
 M4 4 1 6 0 NMOD W=30U L=5U AD=230P AS=75P PS=65U PD=100U
 M5 6 2 7 0 NMOD W=30U L=5U AD=75P AS=75P PS=65U PD=65U
 M6 7 3 0 0 NMOD W=30U L=5U AD=75P AS=230P PS=100U PD=65U
.ENDS CNAND3
.SUBCKT LOAD 1 2 3
 M1 2 1 3 3  PMOD W=10U L=5U AS=60P AD=60P PS=35U PD=35U
 M2 2 1 0 0  NMOD W=10U L=5U AS=60P AD=60P PS=35U PD=35U
.ENDS LOAD
.SUBCKT LNAND 1 2 3
M1 2 1 3 3 PMOD W=20U L=5U AS=180P AD=180P PS=80U PD=80U
M2 2 1 0 0 NMOD W=30U L=5U AS=230P AD=230P PS=100U PD=100U
.ENDS LNAND
.SUBCKT LNAND2 1 2
X1 1 3 2 LNAND
X3 1 5 2 LNAND
.ENDS LNAND2
 .SUBCKT CLKINV 1 2 3 4 5
 M1 6 1 5 5 PMOD W=30U L=5U  AS=230P AD=75P PD=65U PS=100U
 M2 2 4 6 5 PMOD W=30U L=5U  AS=75P AD=230P PD=100U PS=65U
 M3 2 3 7 0 NMOD W=10U L=5U AD=125P AS=25P PD=60U PS=25U
 M4 7 1 0 0 NMOD W=10U L=5U AD=25P AS=125P PD=25U PS=60U
 .ENDS CLKINV
.SUBCKT LATCH 1 2 3 4 5
X1 1 6 3 4 5 CLKINV
X2 6 2 5 INVERTER
X3 2 6 4 3 5 CLKINV
.ENDS LATCH
.SUBCKT CNOR 1 2 3 4
M1 6 1 4 4 PMOD W=45U L=5U  AD=250P AS=250P PS=120U PD=120U
M2 3 2 6 4 PMOD W=45U L=5U  AD=250P AS=250P PS=120U PD=120U
M3 3 1 0 0 NMOD W=20U L=5U AD=150P AS=80P PS=30U PD=80U
M4 3 2 0 0 NMOD W=20U L=5U AD=50P AS=180P PS=80U PD=30U
.ENDS CNOR
.SUBCKT RSTLATCH 1 2 3 4 6
X1 1 7 3 4 6 CLKINV
X2 7 0 2 6   CNOR
X3 2 7 4 3 6 CLKINV
.ENDS RSTLATCH
.SUBCKT DATAI 7 8 9 10 11 12 13 14 15 99
M1 1 7 99 99 PMOD W=15U L=5U AD=155P AS=155P PS=70U PD=70U
M2 1 8 2 0 NMOD W=10U L=5U AD=50P AS=25P PS=25U PD=30U
M3 2 9 6 0 NMOD W=10U L=5U AD=25P AS=50P PS=30U PD=25U
M4 1 10 3 0 NMOD W=10U L=5U AD=50P AS=25P PS=25U PD=30U
M5 3 11 6 0 NMOD W=10U L=5U AD=25P AS=50P PS=30U PD=25U
M6 1 12 4 0 NMOD W=10U L=5U AD=50P AS=25P PS=25U PD=30U
M7 4 13 5 0 NMOD W=10U L=5U AD=25P AS=25P PS=25U PD=25U
M8 5 14 6 0 NMOD W=10U L=5U AD=25P AS=50P PS=30U PD=25U
M9 6 7 0 0 NMOD W=10U L=5U AD=50P AS=130P PS=60U PD=25U
M10 15 1 99 99 PMOD W=30U L=5U AD=230P AS=230P PS=100U PD=100U
M11 15 1 0 0   NMOD W=10U L=5U AD=130P AS=130P PS=60U PD=60U
.ENDS DATAI
```

```
.MODEL PMOD PMOS LEVEL=2 VTO=-0.9 KP=9.75U GAMMA=0.634 PHI=0.612 LAMBDA=3.0E-2
+ RD=2.0 RS=2.0
+ CBD=2.0E-14 CBS=2.0E-14 IS=1.0E-14 PB=0.7 CGSO=2.44E-10 CGDO=2.44E-10
+ CGBO=2.0E-12 RSH=75.0
+ CJ=1.94E-4 MJ=0.5 CJSW=4.37E-10 MJSW=0.5 JS=4.19E-10 TOX=8.5E-8 NSUB=1.98E+15
+ TPG=1.0 XJ=9.0E-7
+ LD=6.0E-7 UO=240.0 UCRIT=6.44E+4 UEXP=0.139 VMAX=7.33E+4   XQC=0.4

.MODEL NMOD NMOS LEVEL=2 VTO=0.9 KP=3.05E-5 GAMMA=1.592 PHI=0.695 LAMBDA=1.0E-2
+ RD=2.0 RS=2.0
+ CBD=2.0E-14 CBS=2.0E-14 IS=1.0E-14 PB=0.7 CGSO=2.84E-10 CGDO=2.84E-10
+ CGBO=2.0E-12 RSH=15.0 CJ=3.44E-4
+ MJ=0.5 CJSW=1.09E-9 MJSW=0.5 JS=1.37E-5 TOX=8.5E-8 NSUB=9.92E+15 TPG=1.0
+ XJ=1.0E-6 LD=7.0E-7
+ UO=750.0 UCRIT=1.23E+5 UEXP=0.022 VMAX=4.92E+5   XQC=0.4

VDD 99 0 DC 5
VCL1 16 0 PULSE 0 5 0 0 0 30N 60N
VCL2 17 0 PULSE 5 0 0 2.5N 2.5N 30N 60N
V11 11 0 DC 0
V13 13 0 DC 0
V15 15 0 DC 5
V10 10 0 DC 0
V24 24 0 DC 5
V28 28 0 DC 0
V19 19 0 DC 5
X01 17 2 3 4 5 6 7 8 9 99 DATAI
X1 10 2 16 17 99 LATCH
X2 11 12 16 17 99 RSTLATCH
X3 13 14 16 17 99 RSTLATCH
X4 15 18 16 17 99 RSTLATCH
X5 19 20 16 17 99 RSTLATCH
X6 14 21 99       INVERTER
X7 12 121 99      INVERTER
X8 121 20 14 7 99 CNAND3
X9 7 3 99         INVERTER
X11 18 121 41 8 99 CNAND3
X12 8 5 99        INVERTER
X10 24 4 16 17 99 LATCH
X13 28 6 16 17 99 LATCH
X14 9 30 17 16 99 LATCH
X115 30 31 16 17 99 LATCH
X20 2 99 LOAD4
X21 12 99 LNAND2
X22 14 99 LNAND2
X23 18 99 LNAND2
X24 20 99 LNAND2
X25 21 40 99 LNAND
X26 121 41 99 LNAND
X27 7 99 LOAD4
X28 3 99 LOAD4
X29 4 99 LOAD4
X30 8 99 LOAD4
X31 5 99 LOAD4
X32 6 99 LOAD4
.IC V(2)=5 V(12)=5 V(14)=5 V(18)=5 V(20)=5 V(4)=5 V(6)=5 V(30)=5 V(31)=5
.IC V(40)=5 V(41)=5 V(9)=0
.TRAN 0.5N 60N
.OPTIONS  NOMOD
.PLOT TRAN  V(2)(-10,5) V(3)(-5,15) V(5)(0,17)
.PLOT TRAN V(4)(-10,5) V(6)(-5,12) V(7)(0,20)
.PLOT TRAN V(8)(-10,5) V(9)(-5,12)
.PLOT TRAN V(3)(-6,24) V(5)(-12,18) V(9)(-18,12) V(17)(0,30) V(30)(-24,6)
 .OPTIONS   VNTOL=5E-3   RELTOL=5E-3 ABSTOL=5E-3
.END
```



```
FLAG BIT FOR 1 TO N

****     INPUT LISTING                    TEMPERATURE =   27.000 DEG C

******************************************************************************


.SUBCKT INVERTER 1 2 3
M1 2 1 3 3 PMOD W=30U L=5U  AD=230P AS=230P PS=100U PD=100U
M2  2 1 0 0 NMOD W=10U L=5U AD=130P AS=130P PS=60U PD=60U
.ENDS INVERTER
.SUBCKT CNAND3 1 2 3 4 5
M1 4 1 5 5 PMOD W=20U L=5U AD=150P AS=150P PS=75U PD=75U
M2 4 2 5 5 PMOD W=20U L=5U AD=150P AS=150P PS=75U PD=75U
M3 4 3 5 5 PMOD W=20U L=5U AD=150P AS=150P PS=75U PD=75U
M4 4 1 6 0 NMOD W=30U L=5U AD=230P AS=75P PS=65U PD=100U
M5 6 2 7 0 NMOD W=30U L=5U AD=75P AS=75P PS=65U PD=65U
M6 7 3 0 0 NMOD W=30U L=5U AD=75P AS=230P PS=100U PD=65U
.ENDS CNAND3
.SUBCKT LOAD6 4 5
 X4 4 6 5 LOAD
 X5 4 7 5 LOAD
 X6 4 8 5 LOAD
 X7 4 9 5 LOAD
 X8 4 10 5 LOAD
 X9 4 11 5 LOAD
.ENDS LOAD6
.SUBCKT LOAD 1 2 3
 M1 2 1 3 3  PMOD W=10U L=5U AS=60P AD=60P PS=35U PD=35U
 M2 2 1 0 0  NMOD W=10U L=5U AS=60P AD=60P PS=35U PD=35U
.ENDS LOAD
.SUBCKT LNAND 1 2 3
M1 2 1 3 3 PMOD W=20U L=5U AS=180P AD=180P PS=80U PD=80U
M2 2 1 0 0 NMOD W=30U L=5U AS=230P AD=230P PS=100U PD=100U
.ENDS LNAND
.SUBCKT LNAND2 1 2
X1 1 3 2 LNAND
X3 1 5 2 LNAND
.ENDS LNAND2
.SUBCKT CNOR 1 2 3 4
M1 6 1 4 4 PMOD W=45U L=5U  AD=250P AS=250P PS=120U PD=120U
M2 3 2 6 4 PMOD W=45U L=5U  AD=250P AS=250P PS=120U PD=120U
M3 3 1 0 0 NMOD W=20U L=5U AD=150P AS=80P PS=30U PD=80U
M4 3 2 0 0 NMOD W=20U L=5U AD=50P AS=180P PS=80U PD=30U
.ENDS CNOR
 .SUBCKT CLKINV 1 2 3 4 5
 M1 6 1 5 5 PMOD W=30U L=5U  AS=230P AD=75P PD=65U PS=100U
 M2 2 4 6 5 PMOD W=30U L=5U  AS=75P AD=230P PD=100U PS=65U
 M3 2 3 7 0 NMOD W=10U L=5U AD=130P AS=25P PD=60U PS=25U
 M4 7 1.0 0 NMOD W=10U L=5U AD=25P AS=130P PD=25U PS=60U
 .ENDS CLKINV
.SUBCKT LATCH 1 2 3 4 5
X1 1 6 3 4 5 CLKINV
X2 6 2 5     INVERTER
X3 2 6 4 3 5 CLKINV
.ENDS LATCH
.SUBCKT RSTLATCH 1 2 3 4 6
X1 1 7 3 4 6 CLKINV
X2 7 0 2 6   CNOR
X3 2 7 4 3 6 CLKINV
.ENDS RSTLATCH
.SUBCKT STATUSI 11 6 7 8 9 10 5 99
M1 1 11 99 99 PMOD W=15U L=5U  AD=155P AS=155P PS=70U PD=70U
M2 1 6 4 0    NMOD W=10U L=5U  AD=75P AS=75P PS=45U PD=45U
M3 1 7 4 0    NMOD W=10U L=5U  AD=75P AS=75P PS=45U PD=45U
M4 1 8 2 0    NMOD W=10U L=5U  AD=75P AS=25P PS=25U PD=40U
M5 2 9 3 0    NMOD W=10U L=5U  AD=25P AS=25P PS=25U PD=25U
M6 3 10 4 0   NMOD W=10U L=5U  AD=25P AS=75P PS=40U PD=25U
M7 4 11 0 0   NMOD W=10U L=5U  AD=25P AS=150P PS=60P PD=25U
M8 5 1 99 99 PMOD W=30U L=5U AS=225P AD=225P PS=100U PD=100U
M9 5 1 0 0   NMOD W=10U L=5U AS=130P AD=130P PS=60U PD=60U
.ENDS STATUSI
```

```
.MODEL PMOD PMOS LEVEL=2 VTO=-0.9 KP=9.75U GAMMA=0.634 PHI=0.612 LAMBDA=3.0E-2
+ RD=2.0 RS=2.0
+ CBD=2.0E-14 CBS=2.0E-14 IS=1.0E-14 PB=0.7 CGSO=2.44E-10 CGDO=2.44E-10
+ CGBO=2.0E-12 RSH=75.0
+ CJ=1.54E-4 MJ=0.5 CJSW=4.37E-10 MJSW=0.5 JS=4.19E-10 TOX=8.5E-8 NSUB=1.98E+15
+ TPG=1.0 XJ=9.0E-7
+ LD=6.0E-7 UO=240.0 UCRIT=6.44E+4 UEXP=0.139 VMAX=7.33E+4  XQC=0.4

.MODEL NMOD NMOS LEVEL=2 VTO=0.9 KP=3.05E-5 GAMMA=1.592 PHI=0.695 LAMBDA=1.0E-2
+ RD=2.0 RS=2.0
+ CBD=2.0E-14 CBS=2.0E-14 IS=1.0E-14 PB=0.7 CGSO=2.84E-10 CGDO=2.84E-10
+ CGBO=2.0E-12 RSH=15.0 CJ=3.44E-4
+ MJ=0.5 CJSW=1.09E-9 MJSW=0.5 JS=1.37E-5 TOX=8.5E-8 NSUB=9.92E+15 TPG=1.0
+ XJ=1.0E-6 LD=7.0E-7
+ UO=750.0 UCRIT=1.23E+5 UEXP=0.022 VMAX=4.92E+5  XQC=0.4
VDD 99 0 DC 5
VCL1 16 0 PULSE 0 5 0 0 0 30N 60N
VCL2 17 0 PULSE 5 0 0 2.5N 2.5N 30N 60N
V11 11 0 DC 0
V13 13 0 DC 0
V15 15 0 DC 5
V19 19 0 DC 5
X1 17 2 3 4 5 6 7 99 STATUS1
X40 22 4 14 5 99 CNAND3
X41 4 21 42 6 99 CNAND3
X42 20 42 99      INVERTER
X2 11 4 16 17 99 RSTLATCH
X3 13 14 16 17 99 RSTLATCH
X4 15 18 16 17 99 RSTLATCH
X5 19 20 16 17 99 RSTLATCH
X6 14 21 99       INVERTER
X7 18 22 99 INVERTER
X71 4 121 99      INVERTER
X8 121 20 14 71 99  CNAND3
X9 71 3 99        INVERTER
X11 18 121 21 8 99 CNAND3
X12 8 2 99        INVERTER
X14 7 77 17 16 99 RSTLATCH
X15 77 78 16 17 99 RSTLATCH
X115  4 99   LNAND2
X16 14 99   LNAND2
X17 18 99   LNAND2
X18 20 99   LNAND2
X19 21 99   LNAND2
X20 22 99   LNAND2
X21 121 99  LNAND2
X22 71 99   LOAD6
X23  3 99   LOAD6
X24  8 99   LOAD6
X25  2 99   LOAD6
X26  5 99   LOAD6
X27  6 99   LOAD6
.PLOT TRAN V(17)(0,30) V(2)(-6,24) V(3)(-12,18) V(7)(-18,12) V(77)(-24,6)
.TRAN 0.5N 60N
.IC   V(4)=0 V(14)=0 V(18)=0 V(20)=0 V(77)=0 V(7)=0 V(78)=0
.OPTIONS  NOMOD VNTOL=5E-4   RELTOL=5E-3 ABSTOL=5E-5
.PRINT TRAN V(2) V(3) V(4) V(5) V(6) V(7) V(77) V(78)    (0,5)
.END
```


TIME
W(17)
20 NS
17 NS
5 NS
8 NS
5 NS
3 NS
15 NS
19 NS
5 NS