A Framework for the VLSI Implementation of Systolic Tree Based Data Structure

by

Mohammed Abdul Aziz Khalid

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER ENGINEERING

September, 1994

Order Number 1359369

A framework for the VLSI implementation of systolic tree based data structures

Khalid, Mohammed Abdul Aziz, M.S.

King Fahd University of Petroleum and Minerals (Saudi Arabia), 1994

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

A FRAMEWORK FOR THE VLSI IMPLEMENTATION OF SYSTOLIC TREE BASED DATA STRUCTURES

BY

Mohammed Abdul Aziz Khalid

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE
In

COMPUTER ENGINEERING

SEPTEMBER 1994

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS

DHAHRAN, SAUDI ARABIA

This thesis, written by

Mohammed Abdul Aziz Khalid

under the direction of his Thesis Advisor, and approved by his Thesis committee, has been presented to and accepted by the Dean, College of Graduate Studies, in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER ENGINEERING

Thesis Committee:

Sadiq Sait . M

Chairman (Dr. Sadiq M. Sait)

Co-Chairman (Dr. Habib Youssef)

Member (Dr. M. S. T. Benten)

Member (Dr. Mostafa Abd-El-Barr)

Department Chairman

Dr. Ala H. Rabeh
Dean, College of Graduate Studies

Date: August 20, 1994

Dedicated to

My Grandmother, Parents,

Brothers and Sisters

Acknowledgment

Praise to the Almighty Allah Who gave me the courage and patience to carry out this work. I am happy to have had a chance to glorify His name in the sincerest way through this small accomplishment and ask Him to accept my efforts. May He guide us and the whole humanity to the right path *(Aameen)*.

Acknowledgement is due to King Fahd University of Petroleum and Minerals, Dhahran, for providing opportunity to carry out this research work.

My deep appreciation goes to my major thesis advisor Dr. Sadiq M. Sait for his constant help, guidance and the countless hours of attention he devoted throughout the course of this work. He was always kind, understanding and sympathetic towards me. Working with him was indeed a wonderful and learning experience which I thoroughly enjoyed. I express my thanks to my co-advisor Dr. Habib Youssef and committee members Dr. M. S. T. Benten and Dr. Mostafa Abd-El-Barr for their interest, cooperation, advice and constructive criticism.

I am also indebted to the department chairman, Dr. Samir Abdul-Jauwad

and other faculty members for their support.

Thanks are due to my father, younger brother Abed for their understanding and support, special thanks to my grandmother *Ammajan*, and my mother for their relentless attention throughout my academic career, thanks to my sisters and younger brother Rashed who deserve a lot of encouragement and hardwork.

Lastly, I wish to thank fellow graduate students and friends (in alphabetical order) Ahmed Mohiuddin, Asaf, Feroze, Ghouse, Haroon, Idrees, Javeed, Masood-Ul-Hassan, Mohammed, Naseer, Nisar, Sajjad, Sami, Shahid Akther, Shahid Tanvir, Waseem Akther and all my friends on the campus from whom I learned a lot and who made the long work hours pleasant.

Contents

List of Figures

Abstract

Name: Mohammed Abdul Aziz Khalid

Title: A Framework for the VLSI implementation of Systolic Tree-Based Data Structures

Major Field: Computer Engineering

Date of Degree: September, 1994

Very Large Scale Integration (VLSI) technology has provided opportunities for implementing algorithms and data structures in silicon. Many innovative designs have been proposed for the implementation of data structures in hardware. However, these designs have been presented at an abstract behavioral level. Much remains to be done in investigating the methods and strategies that employ appropriate techniques, to reduce the design turn-around time and facilitate efficient physical implementation of these designs. In this thesis, a framework for VLSI implementation of systolic tree based data structures is introduced. A layout methodology and a VLSI CAD environment that facilitate fast and efficient layout of large binary trees is described. Performance features from the layout in 2μ n-well CMOS technology are presented. In this work the layout approach used for efficient VLSI implementations of generic binary tree-based architectures is emphasized.

Master of Science Degree
King Fahd University of Petroleum and Minerals
Dhahran, Saudi Arabia
September, 1994

خلاصة الرسالة

اسم الطالـــب : محمد عبدالعزيز خالد

عنوان الرسالة : اطار عمل للتطبيق التكاملي ذو المقياس الكبير جداً للهياكل المعلوماتيه ذات قاعدة شجرية سيستولية .

التخصـــص : هندسة الحاسب الآلي .

تاريخ الشهادة : يونيو ١٩٩٤ م .

إن تقنية التكامل ذو المقياس الكبير جداً قدمت فرصاً حقيقية لتطبيق الخوارزميات والهياكل المعلوماتية بالسيليكون . لقد تم اقتراح الكثير من التصميمات المبدعة لتطبيق الهياكل المعلوماتية باستخدام أجهزة الحاسب الآلي . على أية حال ، فإن هذه التصميمات قدمت على مستوى تصرفي تجريدي . يتبقى الكثري مما يمكن عمله من ناحية التحقق والبحث في الطرق والإستراتيجيات التي توظف طرقًا تقنية مناسبة للتقليل من الوقت اللازم لدوران التصميم وكذلك لتمهد وتسهل التطبيق الفعّال لهذه التصميمات . في هذه الرسالة ، نقدم إطار عمل لتطبيق الهياكل المعلوماتية ذات القاعدة الشجرية السيستولية . هذا الإطار يمتاز بفعالية الهياكل المعلوماتية والخوارزميات المناسبة للتطبيق . يقوم بهذا عن طريق اختيار التركيب البنيوي والمخطط الإخفائي الفعّال في شبكة متسامتة من المنفذات . في هذا الرسالة أيضاً نصف طريقة للعرض وبيئة تصميم مساعد بالكمبيوتر لتكامل ذو مقياس كبير جداً مما يسهل عملية العرض الفعّال والسريع لأشجار زوجية كبيرة . خصائص التأدية من عرض تقنية CMOS مقدمة في هذه الرسالة . في هذا العمل نركز على طريقة العرض للتطبيقات المتكاملة ذات المقياس الكبير جداً للآبنية المبنية على الأشجار الثنائية الخلاّقة.

درجة الماجستير في العلوم

جامعة الملك فهد للبترول والمعادن

الظهران ، المملكة العربية السعودية

يونيو ١٩٩٤ م

Chapter 1

Introduction

1.1 Introduction

The advent of Very Large Scale Integration (VLSI) has led to the development of high performance, special purpose hardware to meet specific application requirements. This technology has given us the opportunity to implement algorithms and data structures on integrated circuits. Some of the desirable features of architectures which facilitate implementation in VLSI include modularity, regularity, and local inter-connectivity. Architectures having systolic nature, exhibit the above mentioned characteristics. Beside these, systolic architectures have strong parallelism thus yielding higher performance than conventional Von Neumann architectures. These architectures have been proposed for several applications, for example, in digital signal processing, matrix operations and in implementation of data struc-

tures [4, 30]. The architecture of a systolic array is characterized by the architecture of the processor, the size of the array and the inter-connectivity of the processors. The architecture is generally described at the behavioral level. Most systolic architectures are either 1-D or 2-D arrays. Some efficient systolic architectures are also based on tree-architectures and are commonly known as systolic trees. In systolic trees, the processors are nodes of the tree, and the edges correspond to the communications links between processors. Examples of data structures implemented as systolic trees are stack, queue, dequeue, priority queue, dictionary machine, etc. [3, 4, 17].

The objective of this work is to present a framework for the implementation of data structures in VLSI for systolic tree based architectures. As will be discussed in literature review of next chapter, not much work has been done in this area of implementation. The work is carried out in two phases: a logic synthesis phase and a layout phase. Referring to Figure 1.1, the system takes a register transfer level description (RTL). A processor (one node of systolic array) can be specified in RTL. The specification undergoes logic synthesis to give a netlist of standard gates (NAND, NOR etc.,) and flip-flops. Simulation is done both at register transfer level and gate level and compared to verify the translation process. This finishes the task of synthesis.

The netlist is given to another physical design subsystem which has placement,

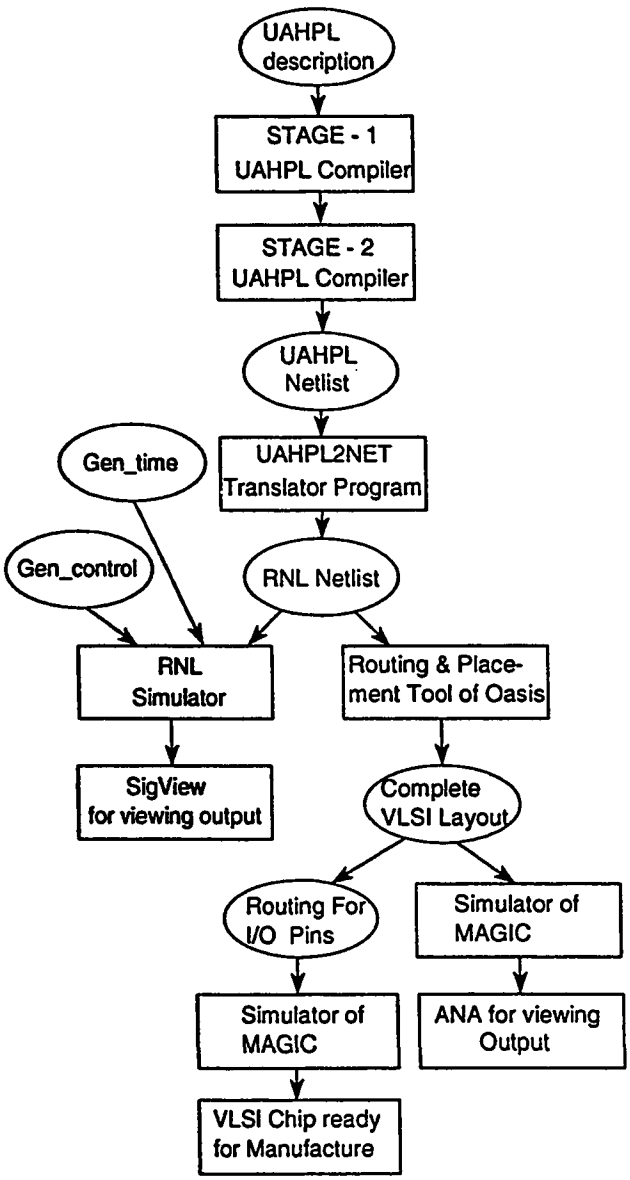


Figure 1.1: An overview for DA system for generating layout from RTL specification and its verification [25].

routing and graphical manipulation tools along with a standard cell library. The layout produced is checked for design rules and the circuit is extracted. The extracted circuit is again simulated and the results compared with the results of the functional simulation at register transfer level.

The layout system of OASIS [21] called Vanilla Place aNd Route (VPNR) performs the routing and placement of the cells and generates the standard cell layout. VPNR [21] uses a library of predesigned standard cells. It also incorporates testability by including scan-path-based testing circuitry in the layout. VPNR is used to design the layout of one node of data structure. To synthesize the data structure of variable size, appropriate number of copies of one node are made, placed, and routed. Magic's router is used for routing between the nodes. The netlist file is saved and used for interconnection.

The MOSIS pad frame can be included in the layout. Finally the routing is performed and the chip is simulated. The verification of the result enhances the chances that the chip will work correctly after fabrication. The transistor level simulation with pad frame is viewed on ana, a waveform viewer.

1.2 Systolic systems

A systolic system can be defined as a network of processors that rhythmically compute and pass data among themselves. The analogy is with the rhythmic contraction

and expansion of the heart which pulses blood through the circulatory system of the body. Each processor in a systolic network can be thought of as a heart that pumps multiple streams of data throughout the entire network [16]. As a processor pumps data items through, it performs some constant-time computation and may update some of the items. Systolic systems provide a realistic model of computations which capture the concepts of pipelining, parallelism and local interconnection structures. Many basic matrix computations can be performed by systolic systems whose underlying network is array structured [28]. Due to their regular structure and function systolic arrays are suitable for implementation in VLSI.

Unlike the closed-loop circulatory system, a systolic computer system usually has ports into which inputs flow and ports from which the results are retrieved. Thus, a systolic system can be a pipelined system in which input and output occur at every clock pulse. This makes them attractive as peripheral processors attached to the data channel of a host computer as well.

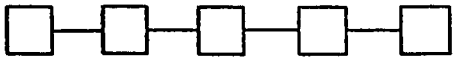
The activities of the processors in a systolic system can be assumed to be synchronous. With each pulse of a clock, a processor executes the same constant-time operation. Furthermore, each processor has equal amount of input and output lines and a constant amount of local storage. It is possible to view the processors as being asynchronous, each computing its output values when all its inputs are available, as in a data flow model. Each processing element in a systolic system is capable of performing some simple operation. Since simple, regular communications and

control structures have substantial advantages over complicated ones in design and implementation, cells in a systolic system are typically interconnected to form a systolic array or a systolic tree. Information in a systolic system flows between cells in pipeline fashion, and communication with the outside world occurs only at the "boundary" cells. Only those cells on the array boundaries have the I/O ports for the communication with the outside world.

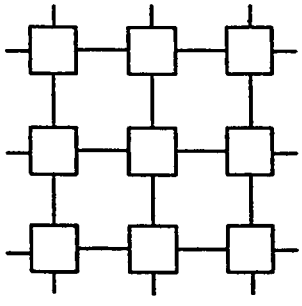
Systolic arrays can be of many different structures for different compute bound algorithms. Figure 1.2 shows various systolic array configurations. The computations which can be carried out using systolic arrays are image processing, matrix arithmetic, combinatorial, database algorithms. Due to the simplicity and strong appeal to intuition, systolic techniques have attracted a great deal of attention. However, the implementation of systolic arrays, especially systolic trees, on a VLSI chip has many practical constraints [12, 16].

1.3 Thesis outline

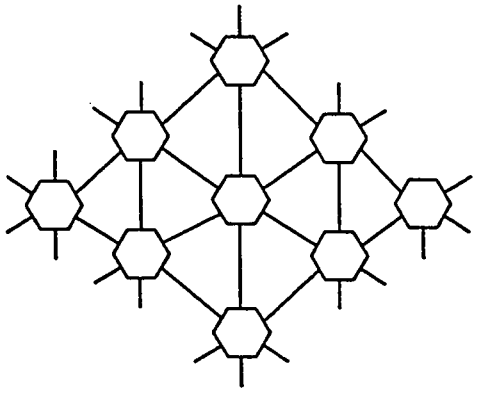
A study was carried out to devise a framework for the VLSI implementation of data structures. Two data structures, stack and queue are implemented in VLSI as examples. Chapter 2 discusses the background work done in the area of VLSI implementation of data structures and the layout of binary trees in different embedding strategies. Chapter 3 gives the design methodology at functional level. The tree



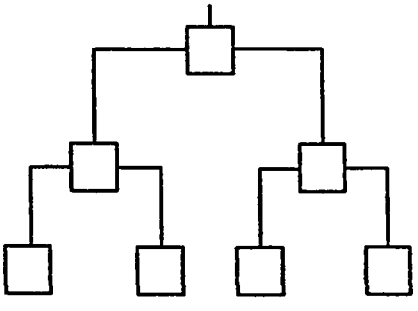
(a) One-dimensional linear array.



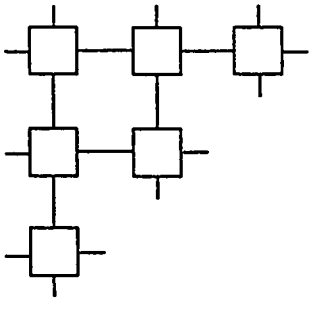
(b) Two-dimensional square array.



(c) Two-dimensional hexagonal array.



(d) Binary tree.



(e) Triangular array.

Figure 1.2: Various systolic array configurations.

embedding in 2-D grid of processors for maximum processor utilization is presented. The I/O buses and the control strategy for the node of a particular data structure is described. Chapter 4 presents the final layouts for stack and queue of variable sizes and simulation results of different size layouts. Finally, Chapter 5 summarizes the work accomplished and discusses future works which can be carried out using this framework.

Chapter 2

Literature review

2.1 Introduction

Many innovative designs have been proposed for hardware implementation of data structures, but much remains to be done in investigating the methods and strategies for VLSI implementation. In this chapter we review systolic algorithms for mapping various data structures in binary tree. Then, current literature on the problem of embedding a complete binary tree in the form of square grid of processors is presented.

2.2 High level synthesis

Implementation of data structures in VLSI results in numerous advantages. Due to this reason, a large number of researchers have been actively involved in this area. Leiserson proposed the design of a systolic priority queue [17] which uses tree topology to achieve a pipeline performance with $O(\log n)$ response time for both delete and insert operations. The software implementation of sequential machine can guarantee $O(\log n)$ performance by using a height balanced binary search tree. For sequential implementation of a priority queue, a heap is an attractive data structure because heap storage can be managed as easily as stack storage. These architectures can also be mapped on a two dimensional grid of processors. Other novel approaches appeared in the literature for the VLSI implementation of different data structures. Guibas and Lang [9] have given systolic array implementations of stack and queue. Ibarra and Culik [5] recently gave implementations of stacks and other linear systolic arrays on tree architecture.

The implementation of dictionary machine and other data structures were also proposed in [1, 22, 26]. Chang et al, proposed a simple finite state control mechanism for tree implementation of data structures [4]. Different algorithms were presented for hardware implementation of priority queue, dictionary machine, queue, stack and dequeue on a binary tree architecture. The algorithms for queue and stack are simple and straight forward but differ slightly in the implementation aspects and

can easily be implemented on a tree based architecture. It is emphasized in the implementation that all the nodes in the tree are identical in all respects, and the control mechanism of each node is independent of the size of the data structure, the input and output operations are performed through the root of the tree.

The architecture of the FIFO queue described by Kanopolous and Hellenbeck [13] consists of two parts, FIFO storage part and control section part. The length of the queue is programmable, resulting in minimum data ripple through times for applications requiring the full length of the memory. The one-to-one transformation of the design is not practical because of the complexity of the controller. This is not a binary tree architecture and we are interested in systolic binary tree architectures.

Another architecture was proposed by Kulaib [15]. This approach is almost similar to that of [13]. It has faster access time and less power dissipation. This design although feasible in VLSI, is cumbersome to implement. The logic is complex and therefore a large area is required.

2.3 Binary tree embedding

After considering the algorithm for the implementation of data structures in binary trees, the problem of embedding a complete binary tree in a square grid of processors is considered. Mapping 1-D and 2-D arrays is not as difficult as mapping a tree onto a grid or 2-D array. A scheme is required to map the nodes of a binary-tree onto a

rectangular grid so that the amount of dead space is minimum. In order to avoid performance degradation, it is preferable that strongly connected nodes (parents and children of the tree) are placed as close as possible. In order to reduce the chip area, several proposals were studied to map nodes of a binary tree into a 2-D grid of processors. This mapping is referred to in the literature as "tree embedding". Important schemes available for tree embedding are described below.

2.3.1 H-trees

H-trees represent a convenient way to place the nodes of a complete binary tree on a grid [2, 17]. The placement can be done recursively, and the final pattern looks like the letter 'H', hence the name. The nodes of the binary tree can be represented in a layout by grid points. A H-tree of order i represents a binary tree of height $2 \times i$ (see Figure 2.1 on next page). Horowitz and Zorat [11] presented a detailed algorithm for mapping a binary tree in the form of a H-tree. As seen from the Figure 2.1, this form of embedding is not area efficient. Also the length of interconnects is large resulting in significant propagation delays. This scheme is only 50% area efficient and much of the chip area is wasted. The advantages, however, is ease of placement (recursively) and routing. Bounds on edge lengths and areas for embedding of trees with leaves on the border of the grid, and a plan of binary tree layout with low maximum wire length are presented in [30]. Clearly, the above scheme is not suitable for VLSI implementation because of low area efficiency and

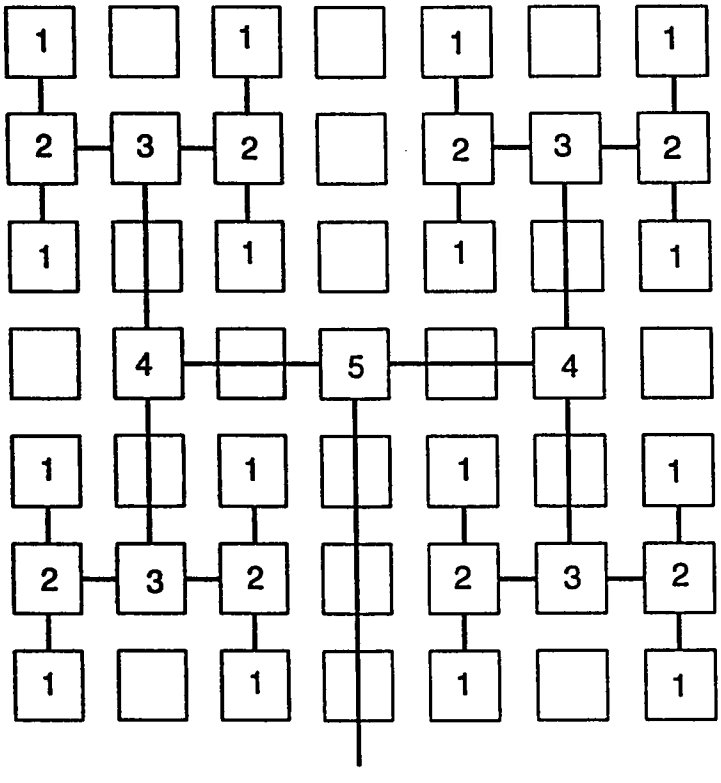


Figure 2.1: A five-level tree embedding in H-tree layout.

high interconnection lengths [11, 30].

2.3.2 Hexagonal tree embedding

Another approach to solve the problem of embedding a complete binary tree in a square or hexagonally connected VLSI array of processing elements was studied by Gordon [8, 7]. The scheme is different from other techniques. Here, instead of considering the problem as that of laying out a graph in a plane, PEs are used both as tree nodes and as connecting elements between distant nodes. This approach shows slight improvement over the H-trees [11] in terms of utilization of chip area and propagation delay. It is 71% area efficient [7, 8]. It can also be seen from Figure 2.2 that the nodes marked with *empty squares* are wasted, creating long edges and hence large propagation delays [7].

2.3.3 Tile-based tree embedding

An very efficient scheme for the layout of large binary tree architectures by embedding the complete binary tree in a 2-D array of PEs was presented by Youn and Singh [31]. This scheme, also known as "tile based design", utilizes virtually 100% of all PEs in the array, and also shows substantial improvement in propagation delay and maximum edge length over H-tree layouts and hexagonal tree layouts. In addition, layouts produced readily lend themselves to fault-tolerant designs for overcoming fabrication defects in large area and wafer scale implementations of binary

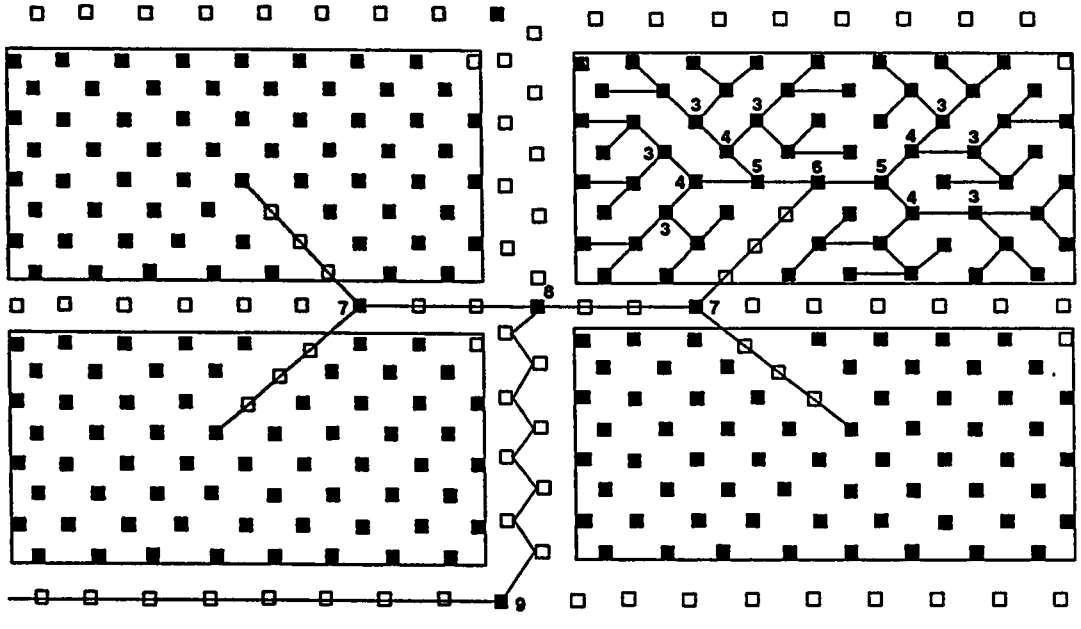


Figure 2.2: The regular embedding scheme for hexagonal array [8].

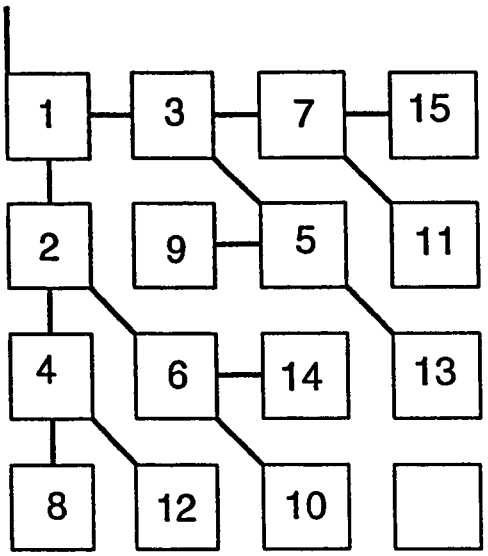


Figure 2.3: Embedding of 15 node tree in 4×4 square array of processors.

tree architectures. As an example, a 4-level (15 node) binary tree embedding is shown in Figure 2.3.

2.4 Conclusion

In this chapter various systolic topologies and algorithms for data structures and other applications were reviewed. The tree-based systolic algorithms were reviewed for stack, queue, priority queue and dictionary machine. A brief literature review of tree embedding was also presented. The embedding strategies which were studied

are H-tree embedding, Hexagonal tree embedding and tile-based tree embedding.

In the next chapter the design methodology is described for the implementation of data structures.

Chapter 3

Design Methodology

3.1 Introduction

The VLSI implementation of tree based architectures consists of the following.

1. The translation of the behavioral model of systolic tree node into an efficient minimal structural level model.

2. Synthesis of the structural model.

3. The embedding of nodes of the tree into an area efficient 2-D grid.

The data structures described by Ibarra et al., [4] at the behavioral level are queue, stack, priority queue, dictionary machine and dequeue. These behavioral models are based on the systolic tree based architectures. In this chapter operation of a queue and stack are described with examples. The behavioral models of queue

and stack are given. The block diagrams illustrating the hardware components of one node of a queue and stack are also presented.

3.2 Operation of a queue

A queue operates in a first-in first-out manner. The design of a FIFO queue is based on the tree architecture proposed by Chang et al [4]. The queue is implemented on a binary tree and has a unit response time and a unit pipeline interval. The root node contains the front of the queue, and the element to be inserted will move down the tree. All nodes of the tree are identical in all respects. Each node has two registers, a data and a buffer register, and two flags, an insert flag and a delete flag. The data register holds the data item to be stored in the node. The buffer register is used for temporary storage of data item to be passed on to the parent or child nodes in the case of delete and insert respectively. The concept underlying the queue operation can be briefly described as follows. When a node becomes active (i.e., an empty node receives an insert instruction), the data item will be stored in the data register of the node, and its insert and delete flags will be reset. When an active node receives an insert instruction, the data item is stored in the buffer register and the insert flag is complemented. Depending upon the value of the insert flag (0 or 1), the data item will be passed on to the right or left child respectively, in the following clock cycle. Successive insert instructions will make last-in data items move to the lowest

levels of the tree, i.e., to the leaf nodes. When a delete instruction is applied to an active node, the data item stored in the data register of the root node is passed on to the output bus, and the data item from its left or right child is copied into the data register of the parent node, and the delete instruction is forwarded to the left or right child depending on the value of the delete flag (0 or 1). Basically, a delete instruction will cause deletion of data item from the root node, and movement of data towards the root of the tree such that the current first-in data item is moved to the root node. Similarly, an insert instruction will cause the data item to move down eventually to the lowest level (one of the leaf nodes) of the tree. For more details, the reader is referred to [4]. The data distribution of the queue when 15 consecutive inserts are made into the empty queue is shown in Figure 3.1. Figure 3.2

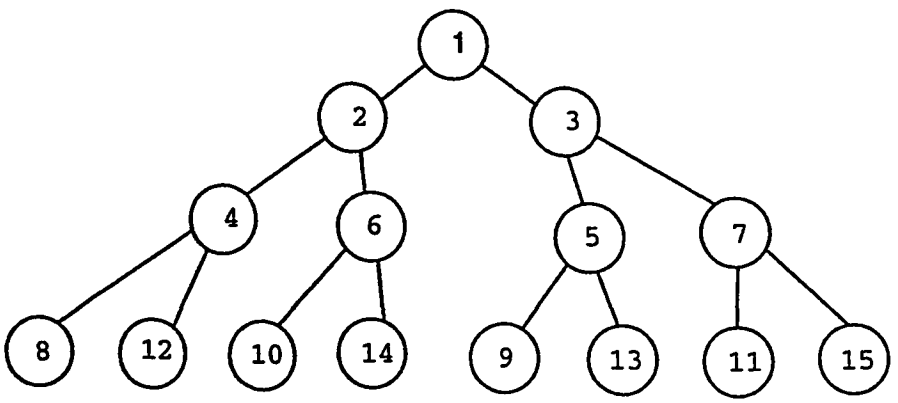


Figure 3.1: Data distribution of the queue after 15 consecutive inserts.

shows the configurations of the queue when the sequence of instructions is applied at every clock cycle. The flag values are given to the right of each node. Consider

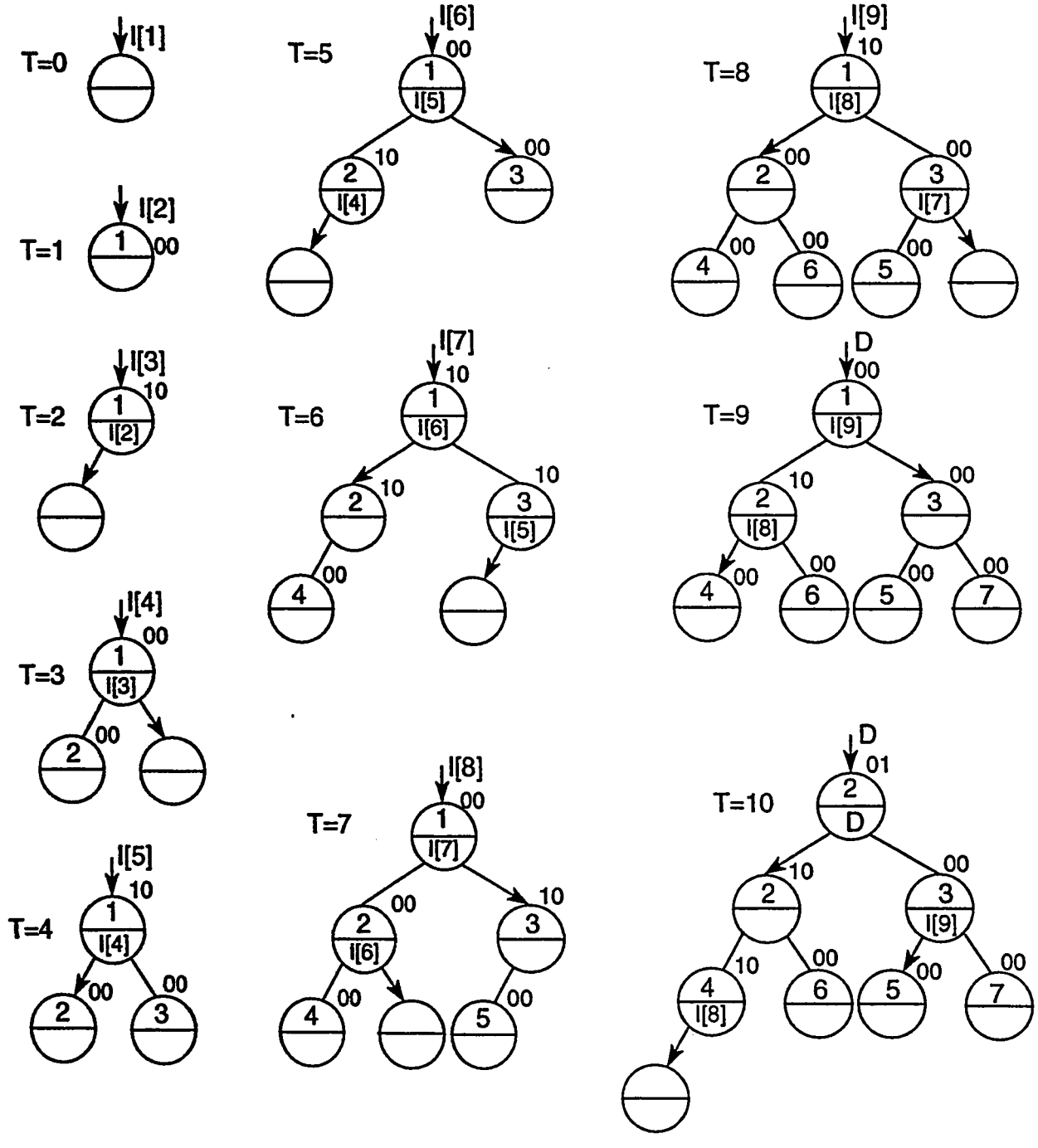


Figure 3.2: The configurations of queue under a sequence of instructions.

the effect of the delete at timestep $T = 9$. Since the delete flag of the root is 0, the data element stored in its left child is copied into the root node. Note the content of the left child (which is 2) is not immediately deleted. Instead, the delete instruction is pipelined to the left. The actual deletion of the left child takes place at $T = 10$. The arrow indicates the node to which an instruction is being directed. Hence, it can be easily seen that the queue has unit response time for both insert and delete operations. Queues have a wide range of applications in computer systems such as, the queue of tasks waiting for the line printer, for access to the disk storage and for a time sharing system, tasks queue up for the CPU. Queues also find application in sorting, searching symbol table, decision table implementations, and in simulation studies [14].

3.3 The structural model of a queue node

In this section we present the hardware implementation and discuss the hardware components of a queue node. The behavioral model of the tree node of a queue is shown in the Figure 3.3 and the block diagram is shown in Figure 3.4. The hardware components include two 8-bit registers S1, B1, and eight 1-bit flip-flops LS1, LB1, CI1, CD1, IL1, DL1, IR1, and DR1. The function of each flip-flop is as follows. LS1 and LB1 show the status (empty or full) of S1 and B1 registers. CI1 and CD1 are insert and delete flags used for the insertion and deletion of data. IL1 and IR1

Pseudo code for Insert and Delete Instructions of a queue

```
begin
    if (LS1 = 0 and Insert present)                                   1st condition
       then  S1 ← value; CI1 ← 0; CD1 ← 0; LS1 ← 1
    else
       if (LS1 =1 and Insert present)                                 2nd condition
             then   B1 ← value; CI1 ← ~CI1; LB1 ← 1
       endif
    endif
    if (LB1=LS2=LS3=0 and Delete is present)                          3rd condition
       then  LS1 ← 0
    endif
    if (LS2=LS3=0 and LB1=1 and Delete is present)                    4th condition
       S1 ← B1; CD1 ← ~CD1;
    endif
    if (~(3rd condition) & ~(4th condition and Delete present))       5th condition
       if CD1 = 0
             then   S1 ← S2 else S1 ← S3
       endif
                    CD1 ← ~CD1;
    endif
end.
```

Figure 3.3: Behavioral model of a node of a systolic-tree based queue.

specify the direction of data movement. If IL1 (IR1) is high then the value is inserted to the left (right) subtree. Similarly, if DL1 (DR1) is high then the value from the left (right) subtree is copied to the root node and the value of the root node appears on output bus TOPD1.

There are two 8-bit wide buses TOPI1 and TOPD1 for insertion and deletion of data into and from the root node respectively. Other four 1-bit input buses are INS1, DEL1, RESET and CLK. If INS1 (DEL1) is high, insert (delete) operation is performed. At the bottom of the node there are four 8-bit wide buses TOPI2, TOPD2, TOPI3 and TOPD3. The 8-bit data is inserted through TOPI2 into the left subtree, and through TOPI3 into the right subtree. The deletion of data from left and right subtrees are routed through TOPD2 and TOPD3 respectively. There are four 1-bit output lines OIL1, OIR1, ODL1 and ODR1. If OIL1 (OIR1) is high then the data is routed to the left (right) subtree. Similarly, if ODL1 (ODR1) is high then the data from the left (right) subtree is deleted.

3.4 Operation of a stack

A stack operates in last-in first-out manner. The design of a LIFO stack is based on the architecture by Chang et al [8]. It is implemented on binary tree and also has unit response time and unit pipeline interval. The root node contains the top of the stack. The element pushed will remain in the root node and the rest of the

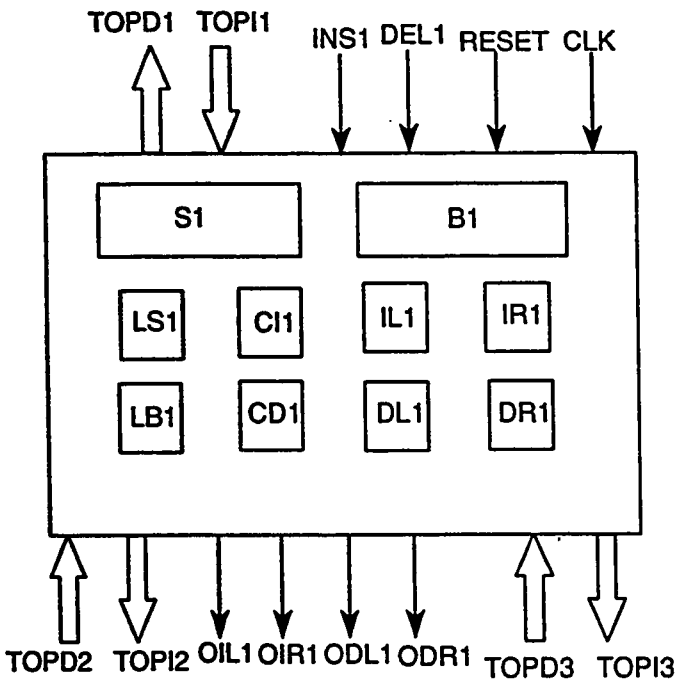


Figure 3.4: Block diagram of a queue node.

elements are pushed down the tree. All the nodes in the binary tree are identical in all respects. In the tree based implementation of the stack, each node has a binary flag, depending on this flag it is determined as to which child the next instruction is to be pipelined. The flag is complemented after each instruction. The last data which is entered into the stack is maintained in the root node of the tree. Each node except the root node gets the instruction from its parents buffer. The concept underlying the stack operation can be briefly described as follows. When a node becomes active (i.e., an empty node receives a push instruction), the value is pushed into the data register and the flag is set to 0. Since there is no instruction in the pipeline anymore, the buffer will remain empty. If the node is active, the instruction needs to be pipelined further. Suppose an element Y is entering a node containing X in the data register, then Y will be stored in the data register and X will be stored in the buffer register and the flag is complemented. If the flag is 1 or 0 the element is pipelined to the right or left child in the next cycle and this operation continues for the consecutive pushes. The contents of a stack after fifteen consecutive pushes to an empty stack is shown in the Figure 3.5. In this example the first number to be pushed is 1 and the last number pushed is 15.

When a pop instruction is applied to the active node, the data item stored into the data register is popped and the data item in the buffer register is copied to the data register of the root node. If the data items are present in the left and right children, then they are copied one after another depending on the flag. If the flag

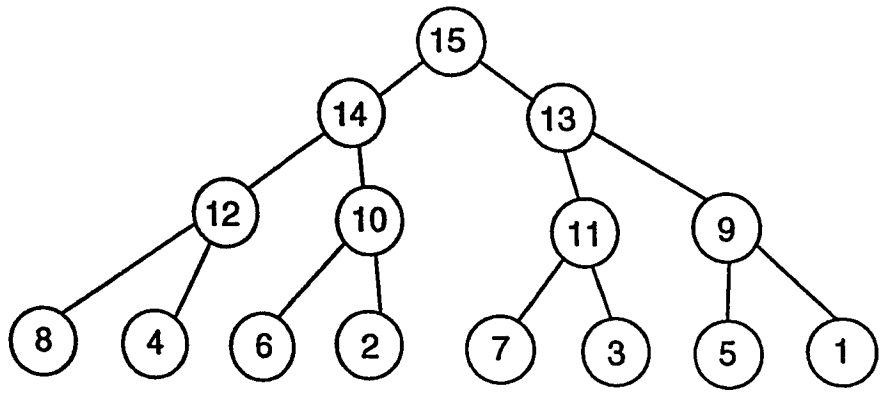


Figure 3.5: Data distribution of the stack after 15 consecutive push operations.

is 1, the data of the right child is copied and if the flag is 0 the data item of the left child is copied. Basically the pop instruction will cause the movement of the data towards the root of the tree such that the last-in data item is moved to the root node. Figure 3.6 shows the configurations of the stack when a sequence of instructions is applied at every clock cycle. The status flag is shown to the upper right corner of the node. Note that the arrow indicates the node to which an instruction is being pipelined. Hence, it is clearly seen that the stack has unit response time with respect to the push and pop operations.

3.5 The structural model of a stack node

In this section we present the hardware design and discuss the hardware components of a stack node. The behavioral model of the tree node of a stack is shown in

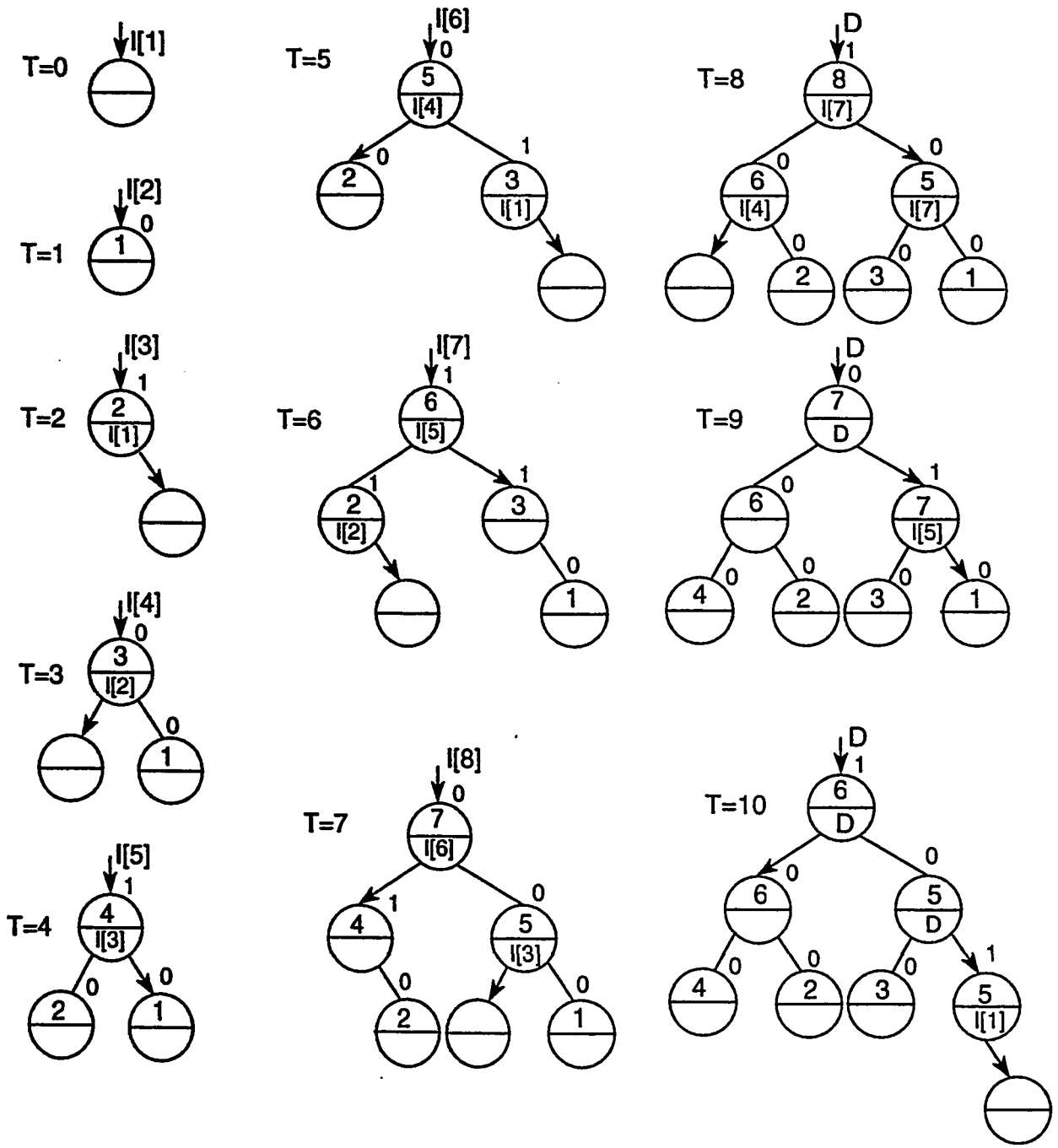


Figure 3.6: The configurations of stack under a sequence of instructions.

Figure 3.7 and the block diagram is shown in Figure 3.8. It consists of two 8-bit registers S1, B1 and seven 1-bit flip-flops LS1, LB1, C1, IL1, DL1, IR1 and DR1. LS1 shows the status of data register S1, LB1 shows the status of buffer register B1. C1 is the status flag for push and pop operations. IL1 and IR1 specify the direction of the data movement. If IL1 (IR1) is high then the value is pushed to the left (right) subtree. Similarly, if DL1 (DR1) is high then the value from the left (right) subtree is copied to the root node and the value of the root node appears on output bus TOPD1.

There are two 8-bit wide buses TOPI1 and TOPD1 for push and pop operations into and from the root node. Other four 1-bit input buses are INS1, DEL1, RESET, CLK. If INS1 is high, push operation is performed. If DEL1 is high then the pop operation is performed. At the bottom of the node there are four 8-bit wide buses TOPI2, TOPD2, TOPI3 and TOPD3. The 8-bit data is pushed through TOPI2 into the left subtree and to the right subtree through TOPI3. The data which is popped from the left and right subtrees are routed through TOPD2 and TOPD3 respectively. There are four 1-bit output lines OIL1, OIR1, ODL1, and ODR1. If OIL1 (OIR1) is high then the data is routed to the left (right) subtree. Similarly, if ODL1 (ODR1) is high then the data from the left (right) subtree is popped. The node has been modeled using the hardware description language AHPL [24]. The implementation aspect is discussed in the next chapter.

Pseudo code for Push and Pop Instructions of Stack

```
begin
    if (LS1 = 0 and Push is present)                                    1st condition
       then  S1 ← value; C1 ← 0; LS1 ← 1
    else
       if (LS1 =1 and Push is present)                                  2nd condition
             then   S1 ← value; B1 ← S1; C1 ← ~C1; LB1 ← 1
       endif
    endif
    if (LB1=LS2=LS3=0 and Pop is present)                               3rd condition
       then  LS1 ← 0
    endif
    if (LB1=1 and Pop is present)                                       4th condition
       S1 ← B1; C1 ← ~C1;
    endif
    if (~(3rd condition) & ~(4th condition and Pop present))            5th condition
       if C1 = 0
             then   S1 ← S2 else S1 ← S3
       endif
                    C1 ← ~C1;
    endif
end.
```

Figure 3.7: Behavioral model of a node of a systolic-tree based stack.

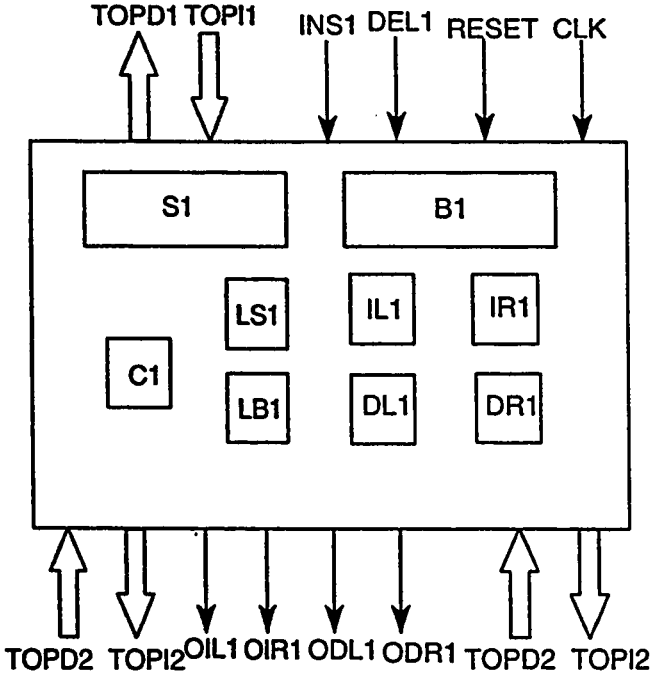


Figure 3.8: Block diagram of a stack node.

3.6 Embedding of binary-tree into tile-based schemes

A new tile (module) based design for laying out the complete binary tree in the form of two dimensional grid of processors with channel based interconnections is presented here. The design is suited to large processors with relatively narrow bus widths. The scheme is better in terms of area efficiency and propagation delay compared to previous designs. More importantly this scheme is 100 percent area efficient for practical size tree architecture, along with short propagation delay and minimum edge lengths.

Thus, using this design, tree architectures can be laid out in VLSI more efficiently than before, with significantly better performance.

3.7 Structure of basic modules

The layout scheme uses a hierarchical strategy such that any required size of tree larger than four levels is laid out by connecting the required number of basic modules of appropriate *type*. Each basic module is a 4×4 square array of processors and contains a four level tree (Figure 3.9). Thus, 15 out of the 16 processors in the basic module are used as tree nodes, the remaining unused processor in each basic module is used as a tree node at some higher level when the basic modules are connected together to build a larger tree (this strategy is also employed in the hexagonal array design in [7]). In other words, each basic module contains a four level leaf subtree

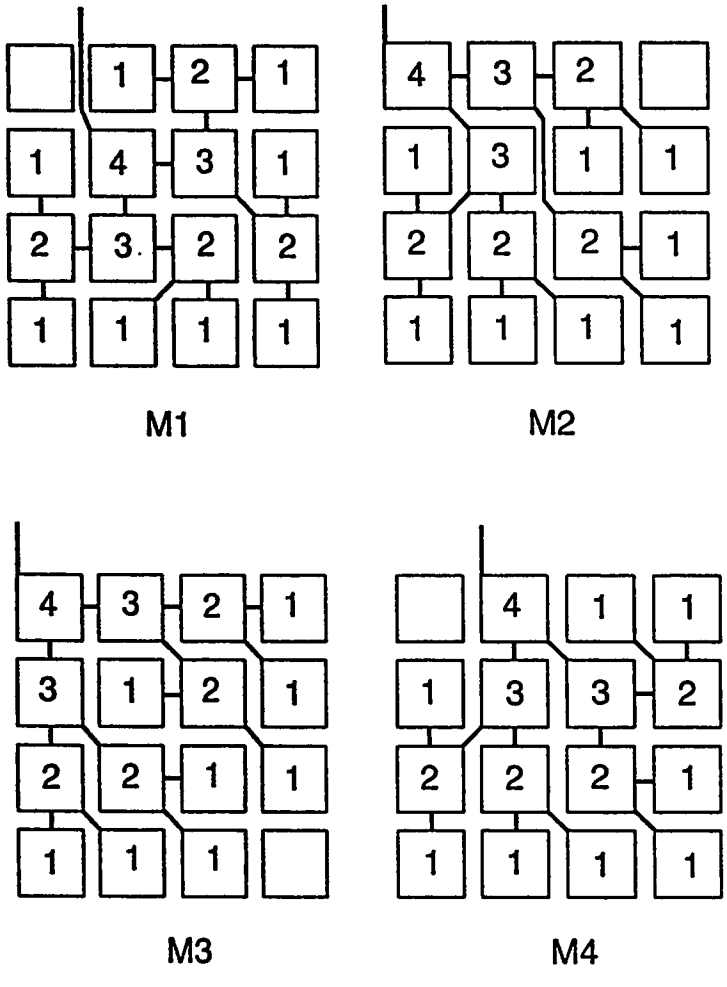


Figure 3.9: The structure of basic modules used for embedding trees with level greater than 4 [31].

of the overall tree and one additional node belonging to a higher level in the tree. Whenever a large tree is constructed using the basic modules, the root node of the tree should be located at the center of the two-dimensional array for minimum propagation delay. In order to realize this, four types of 15 processor basic modules are used. They are different from one another in the relative position of the unused node with respect to the root node for the four-level subtree in each basic module. Figure 3.9 shows the four types of basic modules. We refer to them as $M1$, $M2$, $M3$ and $M4$, respectively. The number in each box (processor) denotes the level of the node in the tree, with the leaf node being at level 1. Observe that all nodes at adjacent levels, except two nodes in the basic module $M2$, are adjacent and can be connected with short links. This allows us to construct large trees using basic modules which are not only area efficient but also have short propagation delays.

3.7.1 Five and six level trees

A five-level tree can be constructed by laying out and connecting one $M1$ basic module with one $M3$ basic module side by side as shown in the Figure 3.10 ($M3$ to the left of $M1$). Observe that the unused node in the basic module $M1$ is utilized as the root node of the five level tree. Actually, one $M1$ basic module with either an $M1$ or $M2$ basic module can also be used to construct a five level tree. However, these combinations are not preferred because they lead to a longer propagation delay. Figure 3.11 shows how a six-level tree can be constructed using two $M1$, one

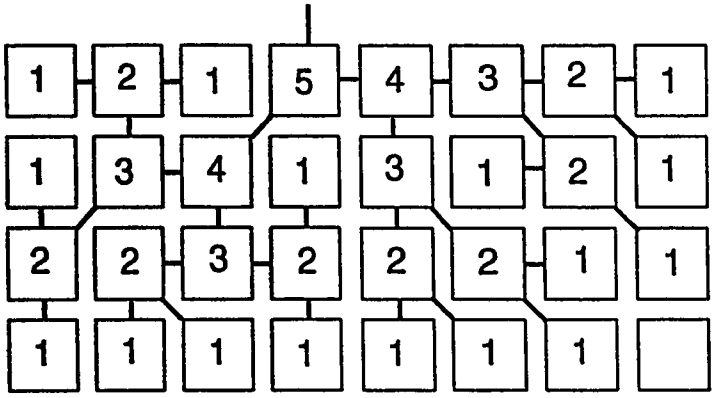


Figure 3.10: A five-level tree embedding using two basic modules [31].

$M3$, and one $M4$ basic modules. Here the number at one corner node of each basic module indicates the level of the node in the embedded tree. Also, the empty small box indicates the unused node. Other combinations of four basic modules such as four $M1$, or three $M1$ and one $M3$ can also realize the six level tree embedding while achieving minimum propagation delay.

3.7.2 Tree larger than six levels

A seven-level tree is shown in Figure 3.12. The unused processor is located in the middle of an edge and adjacent to the channel where the interconnection from the root node emerges from the array.

Trees greater than seven levels are obtained by repeatedly taking two trees of a given size and constructing a larger tree with one additional level. This process can be repeated beginning with two six level trees until a tree with the desired size is

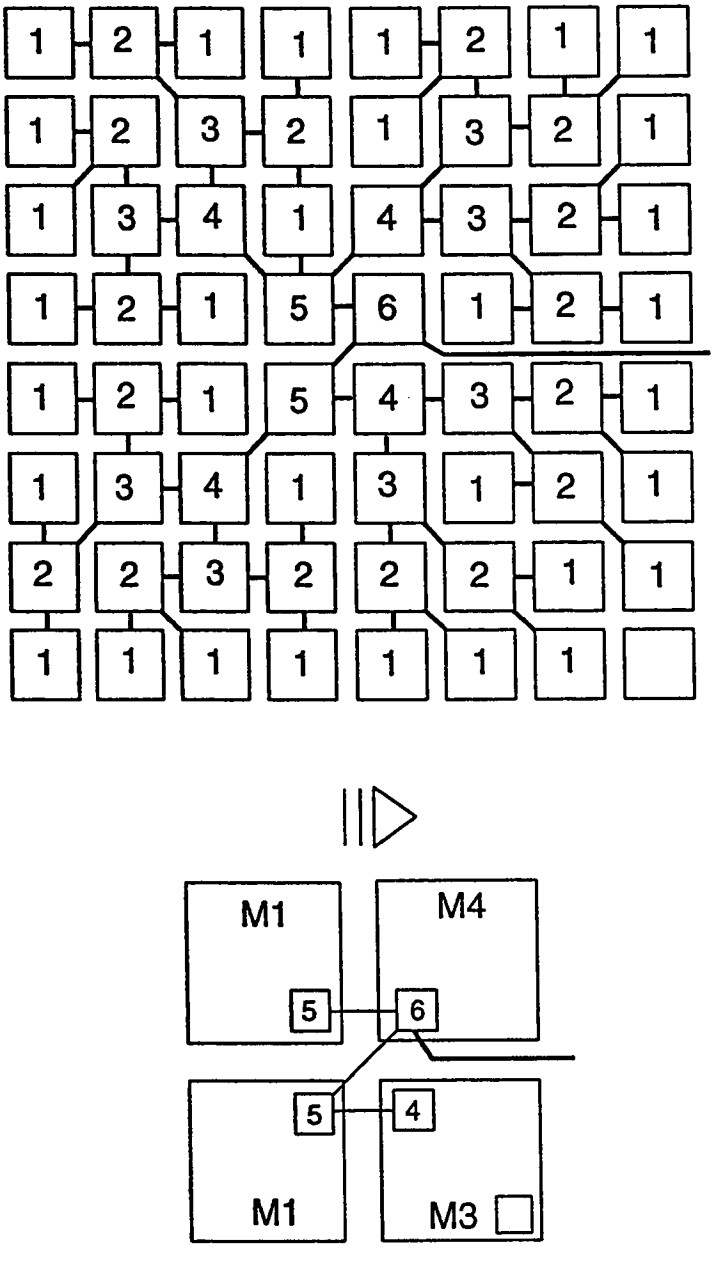


Figure 3.11: A six-level tree embedding using four basic modules [31].

obtained .

The procedure to obtain an $n + 1$ level tree from n level trees is as follows :

- obtain the mirror image of the n level tree along the edge containing the unused processor,
- combine the two layouts and locate the root of the $n+1^{st}$ level tree in the unused processor in one of the original n level tree, and
- rotate the set of modules (in their correct relative positions) about an axis parallel to the appropriate array edge so as to move the unused processor to the outside of the array.

This process can be repeated recursively until the layout of the desired size is obtained.

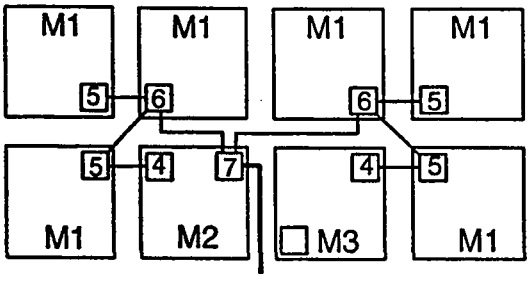


Figure 3.12: A seven-level tree embedding using four basic modules [31].

3.8 Evaluation of proposed tree layout scheme

For the evaluation of the effectiveness of embedding schemes to be used to layout large trees, the following criteria are important.

- Chip area efficiency: determined by the ratio of the active chip area utilized for actual computations to the entire chip area. When each processing element is relatively large, almost all the chip area is taken up by the processing elements. The area efficiency is then the ratio of the number of the processing elements actually utilized as tree nodes to the total number of processing elements implemented on the chip. Since the cost of a VLSI circuit increases with the chip area, this factor is a good metric of the cost effectiveness of the embedding scheme.

- Propagation delay (PD): defined as the maximum data propagation time between the root node and the leaf nodes. This determines the setup time and the data latency when the tree architecture executes a certain problem in parallel.

- Maximum edge length (MEL): defined as the maximum of the edge length between any pair of two directly connected nodes. This is important for two reasons. If the entire processor array operates with a common synchronous clock, then the maximum clock rate is determined by the propagation delay on the longest tree edge, this factor limits the execution speed of the design.

3.8.1 Area efficiency

The area efficiency of an embedding scheme has traditionally been estimated by the ratio of the number of utilized processors to the total number of processors in the host array in which the tree topology is embedded. It can be seen from Figures 3.10, 3.11 and 3.12 that *only one processor is left unutilized in our design for any tree size.* Thus, the area efficiency of our scheme is $(2^k - 1)/2^k$, $k \geq 4$, where k represents number of levels in the tree. This rapidly converges to 100 percent as the size of the embedded tree grows large. Since the number of nodes in the tree of any size is always $2^k - 1$ (always odd) and we embed in an array of 2^k processors, having one unused processor is unavoidable. Therefore, the tree embedding scheme is optimal with respect to the utilization of processors. Note that the area of interconnection buses running between the modules is not reflected in the area efficiency consideration because the interconnection area is assumed to be small in comparison to the total area of the processor.

3.8.2 Propagation delay

The worst propagation delay (PD) in a tree architecture is equal to the delay along the distance between the root and leaf nodes. This can be expressed in terms of the processor edge dimension, which is the length of one side of the processor. To obtain a uniform and consistent measure of interconnect distances independent of the layout

details, we index the processor in row i and column j as (i, j). We also assume that all data paths run horizontally and vertically (Manhattan interconnections). Then the distance between two directly connected processors located at (i_1, j_1) and (i_2, j_2) is $|i_1 - i_2| + |j_1 - j_2|$. The maximum root-to-leaf distance can be found by comparing the distances of all paths from the root node to the leaf nodes. These can be obtained by adding up the distances of each successive pair of two directly connected processors on the path from the root node to the leaf node. For example, the maximum distance of five-level tree in Figure 3.10 is 6 because the distance from the root node (1,4) to the leaf node (4,7) via (1,5), (2,5), and (3,6) is $1 + 1 + 2 + 2 = 6$.

3.8.3 Maximum edge length

The maximum edge length for the small size trees ($k \leq 6$) displays optimal maximum edge length because all the tree nodes are either adjacent or diagonal neighbors. However, for large trees, pattern interconnection of nodes at levels higher than six can result in relatively long upperlevel edges, and degrade performance. Thus, we need to develop a design which reduces the maximum edge length for trees with more than six levels, because such large designs now appear possible with WSI (wafer scale integration) technology. The propagation delay of the design is almost near optimal for any tree size. The propagation delay can be minimized by equally dividing the distance among the edges along each path from the root to a leaf node.

3.9 Buses in the layouts

Bus widths are considered negligible, as compared to a processor edge. The area efficiency of this scheme is almost twice as good as that for the H-tree scheme [31]. Thus, for bus widths upto 40 percent of the processor edge dimension, this scheme is more efficient than H-tree layouts while displaying much better propagation delay and maximum edge lengths. If the bus widths are greater than 40 percent of the processor edge, H-tree layouts will generally by more area efficient, although the tile-based design still displays better performance. The scheme is generally best on all three criteria for large processors and narrow bus widths.

3.10 Conclusion

In this chapter the operation of systolic tree-based queue and stack are described with an example each. The behavioral model of queue and stack are discussed and then internal components are explained with the help of block diagrams. Tile-based tree embedding strategy is presented. Moreover, embedding of different level binary trees is also presented.

Chapter 4

Implementation in VLSI

4.1 Introduction

In the previous chapter the detail design of the nodes of queue and stack and the interconnection of these nodes in the form of 2-D grid of processors was presented. In this chapter we give the implementation of queue and stack in VLSI. Other data structures can be implemented in same way. Only the architecture of the node (PE) needs to be changed.

4.2 RTL model for stack and queue

The hardware description languages (Verilog, CDL, DDL, AHPL, VHDL, ISPS etc.,) have been used successfully for documentation, communication and verification [10].

They have also been used as input specification languages to design automation (DA) systems which synthesize VLSI layouts [24]. The logic synthesis of the node of a systolic tree based data structure is done with the help of hardware compiler [19], for details refer to Appendix A.

4.2.1 RTL model for queue

The hardware of a single node of a systolic queue consists of many components as stated in the earlier chapter. There are two 8-bit registers S1 (data register) and B1 (buffer register), and eight 1-bit flip-flops IL1, IR1, DL1, DR1, CD1, CI1, LS1 and LB1. TOPI1 and TOPD1 are the 8-bit wide buses for insertion and deletion of data. TOPI2 and TOPD2 are the 8-bit wide buses for the left subtree. TOPI3 and TOPD3 are also the 8-bit wide buses the right subtree. INS2 and DEL2 are the insert and delete lines for left child. INS3 and DEL3 are the insert and delete lines for the right child. RESET, CLK, and START are the external inputs so they are declared as EXINPUTS.

There are many conditional statements through which the various decisions are made. The AHPL description shown in the Figure 4.1 contains the code enclosed within the keywords **BODY SEQUENCE** and **END**. The controller of the node has only one $C - step$ since the queue instructions (insert or delete) takes only one clock cycle. The operation of a queue has been simulated using the RT level functional simulator available in the AHPL DA system. The simulation results have

validated the design of the tree node and the stack at the register transfer level [24]. In the Figure 4.2 the COMSEC file is shown which is the input to the DA system. The output produced is shown in Figure 4.3.

4.2.2 RTL model for stack

The hardware of a single node of a systolic stack consists of many components as shown in the block diagram of stack node in previous chapter. All the hardware components are same as that of queue node, because the stack is also 8-bit. The operations here are push and pop. The stack is also mapped on the same architecture as that of a queue.

The AHPL description shown in Figure 4.4 is enclosed within the keywords **BODY SEQUENCE** and **END.** The controller of the node has only one $C-step$ since the stack instructions (push or pop) takes only one clock cycle. The input description file for AHPL DA system for stack is shown in Figure 4.5 and the output is shown in Figure 4.6

After the AHPL code is compiled, the simulation is performed at RT level with the help of functional level simulator [6]. Next logic synthesis is carried out [19]. UAHPL compiler generates a logic netlist of hardware circuit in terms of logic gates, flip-flops etc.,. The output of the stage 2 compiler of UAHPL is stored in the form of linked list consisting of GATE#, specifying the gate number for the element, followed by the GATE TYPE, specifying the type of the element, for example,

```
MODULE      : QNODE1.
MEMORY      : IL1; IR1; DL1; DR1; S1{8}; B1{8};CD1;CI1;LS1; LB1.
EXBUSES     : DEL2; INS3; DEL3;TOPI2{8};TOPI3{8}.
EXBUSES     : LSL1; LSR1; TOPD2{8}; TOPD3{8}; TOPD1{8}; OLS1; INS2; DEL2.
BUSES       : CC11; CC21; CC31; T11; T21; T31; T41; FALSE1.
EXINPUTS    : TOPI1{8}; INS1; DEL1; RESET; CLK; START.
BODY SEQUENCE :CLK.
1 CI1*(T11)   ⇐ 1$0; CD1*(T11) ⇐ 1$0;
  CI1*(T21)   ⇐ ~(CI1); LS1*(T11) ⇐1$1;
  LB1*(T21)   ⇐ 1$1; IL1*(INS1) ⇐(CI1 ! 1$0)*(LS1, ~LS1);
  IR1*(INS1)  ⇐ (~CI1 ! 1$0)* (LS1, ~LS1); LS1*(CC11) ⇐ 1$0;
  CD1*(CC21) ⇐ ~(CD1); DR1*(T31) ⇐ 1$1;
  DL1*(T41)   ⇐ 1$1; CD1*(CC31) ⇐ (~CD1); B1*(T21) ⇐TOPI1;
  S1*(FALSE1)⇐(TOPI1!B1!TOPD2!TOPD3) *(T11,CC21,T41,T31);
  TOPD1       = S1*DEL1;    ⇒ (~START)/(1).
ENDSEQUENCE
CONTROLRESET (1);
  TOPI2 = B1*IL1; TOPI3 = B1*IR1;
  FALSE1= (T11+CC21+T41+T31);
  INS2= IL1; DEL2= DL1; OLS1= LS1; INS3= IR1;
  DEL3= DR1; T11 = INS1 & ~LS1;
  T21 = INS1 & LS1; T31 = CC31 & (~CD1); T41 = CC31 & CD1;
  CC11 = ~(LSR1 + LSL1 + LB1) & DEL1;
  CC21 = ~(LSR1 + LSL1 + ~LB1) & DEL1;
  CC31 = ~(CC11) & ~(CC21) & DEL1.
END.
```

Figure 4.1: AHPL description of systolic tree-based queue.

```
OPTION 6.
VIEW 5 VON.
OUTPUTS INS1;DEL1;TOPI1;TOPD1;S1;B1;TOPI2;TOPI3;
  DEL2;DEL3;INS2;INS3;LS1;LB1.
EXLINES RESET=1,0;START=0;
  LSL2=0;LSR2=0;LSL3=0;LSR3=0;
    INS1=1,1,1,1,1,0,0,0,0,0,0;
    TOPI1 = '11,'22,'33,'44,'55,'66,'77,'88,'99,'AA.
```

Figure 4.2: COMSEC file for AHPL description.

```
KING FAHD UNIV OF PETROLEUM & MINERALS, COLLEGE OF COMPUTER SCIENCE
 UNIVERSAL AHPL SIMULATOR OUTPUT DATE: WED 22 SEPT, 1993 TIME: 18:35:47


 $$$$$$ UNIVERSAL AHPL FUNCTIONAL LEVEL SIMULATOR OUTPUT:

            INS1
            | DEL1
            | | TOPI1
            | | |  TOPD1
            | | |  |  S1
            | | |  |  |  B1
            | | |  |  |  |  TOPI2
            | | |  |  |  |  |  TOPI3
            | | |  |  |  |  |  |  DEL2
            | | |  |  |  |  |  |  | DEL3
            | | |  |  |  |  |  |  | | INS2
            | | |  |  |  |  |  |  | | | INS3
            | | |  |  |  |  |  |  | | | | LS1
            | | |  |  |  |  |  |  | | | | | LB1
 CLOCK #    | | |  |  |  |  |  |  | | | | | |
       0    0 0 00 00 00 00 00 00 0 0 0 0 0 0
       1    1 0 11 00 00 00 00 00 0 0 0 0 0 0
       2    1 0 22 00 11 00 00 00 0 0 0 0 1 0
       3    1 0 33 00 11 22 22 00 0 0 1 0 1 1
       4    1 0 44 00 11 33 00 33 0 0 0 1 1 1
       5    1 0 55 00 11 44 44 00 0 0 1 0 1 1

 ::::: PROGRAM REACHED THE CLOCKLIMIT. AHPL SIMULATION STOPS. :::::
```

Figure 4.3: Output of simulation at RT level for one node systolic tree-based queue.

```
MODULE      : SNODE1.
MEMORY      : IL1; IR1; DL1; DR1; S1{8}; B1{8};C1;LS1; LB1.
EXBUSES     : DEL2; INS3; DEL3;TOPI2{8};TOPI3{8}.
EXBUSES     : LSL1; LSR1; TOPD2{8}; TOPD3{8}; TOPD1{8}; OLS1; INS2; DEL2.
BUSES       : CC11; CC21; CC31; T11; T21; T31; T41; FALSE1.
EXINPUTS    : TOPI1{8}; INS1; DEL1; RESET; CLK; START.
BODY SEQUENCE :CLK.
1 C1*(T11)    ⇐ 1$0;
  C1*(T21)    ⇐ ~(CI1); LS1*(T11) ⇐1$1;
  LB1*(T21)   ⇐ 1$1; IL1*(INS1) ⇐(CI1 ! 1$0)*(LS1, ~LS1);
  IR1*(INS1)  ⇐ (~CI1 ! 1$0)* (LS1, ~LS1); LS1*(CC11) ⇐ 1$0;
  C1*(CC21)   ⇐ ~(C1); DR1*(T31) ⇐ 1$1;
  DL1*(T41)   ⇐ 1$1; C1*(CC31) ⇐ (~C1); B1*(T21) ⇐TOPI1;
  S1*(FALSE1)⇐(TOPI1!B1!TOPD2!TOPD3) *(INS1,CC21,T41,T31);
  TOPD1       = S1*DEL1;    ⇒ (~START)/(1).
ENDSEQUENCE
CONTROLRESET (1);
  TOPI2 = B1*IL1; TOPI3 = B1*IR1;
  FALSE1= (T11+CC21+T41+T31);
  INS2= IL1; DEL2= DL1; OLS1= LS1; INS3= IR1;
  DEL3= DR1; T11 = INS1 & ~LS1;
  T21 = INS1 & LS1; T31 = CC31 & C1; T41 = CC31 & (~C1);
  CC11 = ~LB1 & DEL1;
  CC21 = ~(LSR1 + LSL1 + ~LB1) & DEL1;
  CC31 = ~(CC11) & ~(CC21) & DEL1.
END.
```

Figure 4.4: AHPL description of systolic tree-based stack.

```
OPTION 6.
VIEW 7 VON.
OUTPUTS INS1;DEL1;TOPI1;TOPD1;S1;B1;S2;B2;S3;B3.


EXLINES RESET=1,0;
    LSL2=0;LSR2=0;LSL3=0;LSR3=0;
    START=0;
    INS1=1,1,1,0,0,0,0;
    DEL1=0,0,0,0,1,1,1,0;
    TOPI1 = '11,'22,'33,'00,'00,'00,'00,'00.
```

Figure 4.5: COMSEC file for AHPL description.

```
KING FAHD UNIV OF PETROLEUM & MINERALS, COLLEGE OF COMPUTER SCIENCE
 UNIVERSAL AHPL SIMULATOR OUTPUT DATE: SUN 01  MAY, 1994 TIME: 10:44:09


 $$$$$$ UNIVERSAL AHPL FUNCTIONAL LEVEL SIMULATOR OUTPUT:

            INS1
            | DEL1
            | | TOPI1
            | | |  TOPD1
            | | |  |  S1
            | | |  |  |  B1
            | | |  |  |  |  S3
            | | |  |  |  |  |   S2
            | | |  |  |  |  |   |
            | | |  |  |  |  |   |
 CLOCK #    | | |  |  |  |  |   |
       0    0 0 00 00 00 00 00 00
       1    1 0 11 00 00 00 00 00
       2    1 0 22 00 11 00 00 00
       3    1 0 33 00 22 11 00 00
       4    0 0 00 00 33 22 11 00
       5    0 1 00 33 33 22 11 22
       6    0 1 00 22 22 22 11 22
       7    0 1 00 11 11 22 11 22

 ::::: PROGRAM REACHED THE CLOCKLIMIT. AHPL SIMULATION STOPS. :::::
```

Figure 4.6: Output for one node systolic tree-based stack.

AND, OR, D flip-flop etc. The two fields in the record, ILINK and OLINK, are pointers to the IOLIST.

The AHPL netlist obtained from the stage 2 compiler is translated to RNL compatible logic level netlist using a translator written in "C" programming language. The netlist generated by the translator is in RNL format. It comprises combinational logic gates and three types of D flip-flop (set, reset and control flip-flops) with enable, asynchronous set and reset inputs. RNL compatible netlist for both queue and stack is shown in the Figures 4.7 to 4.10. The netlist generated is simulated by RNL [6], a timing and logic level simulator. The simulation results of RNL are observed using SigView, a signal viewing tool. The process was illustrated in the flow chart in chapter 1.

4.3 Layouts of systolic-tree based queue and stack

The OASIS layout design environment [21] is used for generating the layouts. OASIS is an abbreviation of Open Architecture Silicon Implementation Software. A number of design cells are integrated into the OASIS system. OASIS is a cell based system for IC design. The tools integrated into the OASIS system have been developed to automatically translate high level description of integrated circuits into testable physical layouts, using predesigned standard cells. One of the features of the OASIS system is the modularity of the software. New improved algorithms can be easily

```
(include "defl.net")
(include "drll.net")
(include "dsll.net")
(include "array.sub.net")
(include "standard.def")
(include "decal.def.W")
(include "others.def.W")
( node n100 n101 n102 n103 n104 n105 n106 n107 n108 n109)
( node n110 n111 n112 n113 n114 n115 n116 n117 n118 n119)
( node n120 n121 n122 n123 n124 n125 n126 n127 n128 n129)
( node n130 n131 n132 n133 n134 n135 n136 n137 n138 n139)
( node n140 n141 n142 n143 n144 n145 n146 n147 n148
n149)
( node n150 n151 n152 n153 n154 n155 n156 n157 n158 n159)
( node n160 n161 n162 n163 n164 n165 n166 n167 n168 n169)
( node n170 n171 n172 n173 n174 n175 n176 n177 n178 n179)
( node n180 n181 n182 n183 n184 n185 n186 n187 n188 n189)
( node n190 n191 n194 n201 n213 n314 n331 n335 n336 n338)
( node n339 n340 n342 n343 n344 n345 n347 n348 n349
n350)
( node n351 n373 n378 n383 n388 n393 n398 n403 n408 n453)
( node n455 n460 n465 n473 n478 n481 n482 n487 n491
n493)
( node n498 n502 n503 n504 n506 n508 n509 n510 n511 n513)
( node n515 n517 n519 n521 n523 n525 n563 n564 n565 n566)
( node n567 n568 n569 n570 n571 n573 n574 n575 n576 n578)
( node n579 n580 n581 n583 n584 n585 n586 n588 n589 n590)
( node n591 n593 n594 n595 n596 n598 n599 n600 n601 n603)
( node n612 n613 n614 n615 n616 n617 n618 n619 n620 n632)
( node n633 n634 n635 n636 n637 n638 n639 n649 n650 n651)
( node n652 n653 n654 n655 n656)
( node clk2 scantest Reset scanin )
( pin clk2 TOP 0 100 "pintype=clock" )
( pin scantest TOP 0 100 "pintype=pi" )
( pin scanin TOP 0 100 "pintype=clock" )
( defl n491 n190 n493 n100 clk2 scantest scanin Reset )
( defl n498 n190 n493 n101 clk2 scantest scanin Reset )
( defl n473 n190 n508 n102 clk2 scantest scanin Reset )
( defl n473 n190 n506 n103 clk2 scantest scanin Reset )
( defl n373 n190 n567 n104 clk2 scantest scanin Reset )
( defl n378 n190 n567 n105 clk2 scantest scanin Reset )
( defl n383 n190 n567 n106 clk2 scantest scanin Reset )
( defl n388 n190 n567 n107 clk2 scantest scanin Reset )
( defl n393 n190 n567 n108 clk2 scantest scanin Reset )
( defl n398 n190 n567 n109 clk2 scantest scanin Reset )
( defl n403 n190 n567 n110 clk2 scantest scanin Reset )
( defl n408 n190 n567 n111 clk2 scantest scanin Reset )
( defl n511 n190 n482 n112 clk2 scantest scanin Reset )
( defl n513 n190 n482 n113 clk2 scantest scanin Reset )
( defl n515 n190 n482 n114 clk2 scantest scanin Reset )
( defl n517 n190 n482 n115 clk2 scantest scanin Reset )
( defl n519 n190 n482 n116 clk2 scantest scanin Reset )
( defl n521 n190 n482 n117 clk2 scantest scanin Reset )
( defl n523 n190 n482 n118 clk2 scantest scanin Reset )
( defl n525 n190 n482 n119 clk2 scantest scanin Reset )
( defl n453 n190 n455 n120 clk2 scantest scanin Reset )
( defl n481 n190 n460 n121 clk2 scantest scanin Reset )
( defl n473 n190 n465 n122 clk2 scantest scanin Reset )
( defl n473 n190 n482 n123 clk2 scantest scanin Reset )

(( pin n124 BOTTOM 0 100 "pintype=pi")
( pin n125 BOTTOM 0 100 "pintype=pi")
( pin n126 BOTTOM 0 100 "pintype=pi")
( pin n127 BOTTOM 0 100 "pintype=pi")
( pin n128 BOTTOM 0 100 "pintype=pi")
( pin n129 BOTTOM 0 100 "pintype=pi")
( pin n130 BOTTOM 0 100 "pintype=pi")
( pin n131 BOTTOM 0 100 "pintype=pi")
( pin n132 BOTTOM 0 100 "pintype=pi")
( pin n133 BOTTOM 0 100 "pintype=pi")
( pin n134 BOTTOM 0 100 "pintype=pi")
( pin n135 BOTTOM 0 100 "pintype=pi")
( pin n136 BOTTOM 0 100 "pintype=pi")
( pin n137 BOTTOM 0 100 "pintype=pi")
( pin n138 BOTTOM 0 100 "pintype=pi")
( pin n139 BOTTOM 0 100 "pintype=pi")
( pin n140 BOTTOM 0 100 "pintype=pi")
( pin n141 BOTTOM 0 100 "pintype=pi")
( pin n142 BOTTOM 0 100 "pintype=po")
( connect n142 n612 )
( pin n143 BOTTOM 0 100 "pintype=po")
( connect n143 n613 )
( pin n144 BOTTOM 0 100 "pintype=po")
( connect n144 n614 )
( pin n145 BOTTOM 0 100 "pintype=po")
( connect n145 n615 )
( pin n146 BOTTOM 0 100 "pintype=po")
( connect n146 n616 )
( pin n147 BOTTOM 0 100 "pintype=po")
( connect n147 n617 )
( pin n148 BOTTOM 0 100 "pintype=po")
( connect n148 n618 )
( pin n149 BOTTOM 0 100 "pintype=po")
( connect n149 n619 )
( pin n150 BOTTOM 0 100 "pintype=po")
( connect n150 n122 )
( pin n151 BOTTOM 0 100 "pintype=po")
( connect n151 n101 )
( pin n152 BOTTOM 0 100 "pintype=po")
( connect n152 n103 )
( pin n153 BOTTOM 0 100 "pintype=po")
( connect n153 n100 )
( pin n154 BOTTOM 0 100 "pintype=po")
( connect n154 n102 )
( pin n155 BOTTOM 0 100 "pintype=po")
( connect n155 n649 )
( pin n156 BOTTOM 0 100 "pintype=po")
( connect n156 n650 )
( pin n157 BOTTOM 0 100 "pintype=po")
( connect n157 n651 )
( pin n158 BOTTOM 0 100 "pintype=po")
( connect n158 n652 )
( pin n159 BOTTOM 0 100 "pintype=po")
( connect n159 n653 )
( pin n160 BOTTOM 0 100 "pintype=po")
( connect n160 n654 )
( pin n161 BOTTOM 0 100 "pintype=po")
( connect n161 n655 )
( pin n162 BOTTOM 0 100 "pintype=po")
( connect n162 n656 )
```

Figure 4.7: First part of netlist description for one node queue.

```
( pin n162 BOTTOM 0 100 "pintype=po")
( connect n162 n656 )
( pin n163 BOTTOM 0 100 "pintype=po")
( connect n163 n632 )
( pin n164 BOTTOM 0 100 "pintype=po")
( connect n164 n633 )
( pin n165 BOTTOM 0 100 "pintype=po")
( connect n165 n634 )
( pin n166 BOTTOM 0 100 "pintype=po")
( connect n166 n635 )
( pin n167 BOTTOM 0 100 "pintype=po")
( connect n167 n636 )
( pin n168 BOTTOM 0 100 "pintype=po")
( connect n168 n637 )
( pin n169 BOTTOM 0 100 "pintype=po")
( connect n169 n638 )
( pin n170 BOTTOM 0 100 "pintype=po")
( connect n170 n639 )
( pin n171 BOTTOM 0 100 "pintype=po")
( connect n171 n343 )
( pin n172 BOTTOM 0 100 "pintype=po")
( connect n172 n348 )
( pin n173 BOTTOM 0 100 "pintype=po")
( connect n173 n351 )
( pin n174 BOTTOM 0 100 "pintype=po")
( connect n174 n335 )
( pin n175 BOTTOM 0 100 "pintype=po")
( connect n175 n336 )
( pin n176 BOTTOM 0 100 "pintype=po")
( connect n176 n338 )
( pin n177 BOTTOM 0 100 "pintype=po")
( connect n177 n339 )
( pin n178 BOTTOM 0 100 "pintype=po")
( connect n178 n331 )
( pin n179 TOP 0 100 "pintype=pi")
( pin n180 TOP 0 100 "pintype=pi")
( pin n181 TOP 0 100 "pintype=pi")
( pin n182 TOP 0 100 "pintype=pi")
( pin n183 TOP 0 100 "pintype=pi")
( pin n184 TOP 0 100 "pintype=pi")
( pin n185 TOP 0 100 "pintype=pi")
( pin n186 TOP 0 100 "pintype=pi")
( pin n187 TOP 0 100 "pintype=pi")
( pin n188 TOP 0 100 "pintype=pi")
( pin n189 LEFT 0 100 "pintype=reset")
( pin n190 LEFT 0 100 "pintype=clock")
( pin n191 TOP 0 100 "pintype=pi")
( connect n189 Reset )
( i1 n121 n194)
( i1 n122 n201)
( i1 n120 n213)
( i1 n191 n314)
( o4 n172 n174 n177 n176 n331)
( a2 n201 n187 n335)
( a2 n122 n187 n336)
( a2 n213 n173 n338)
( a2 n120 n173 n339)
( o3 n124 n125 n123 n340)
( i1 n340 n342)
( a2 n188 n342 n343)
( o3 n124 n125 n345 n344)
( i1 n123 n345)
( i1 n344 n347)
( a2 n188 n347 n348)
( i1 n171 n349)
( i1 n172 n350)
( a3 n350 n349 n188 n351)
( o4 n563 n564 n565 n566 n373)
( o4 n568 n569 n570 n571 n378)
( o4 n573 n574 n575 n576 n383)
( o4 n578 n579 n580 n581 n388)
( o4 n583 n584 n585 n586 n393)
( o4 n588 n589 n590 n591 n398)
( o4 n593 n594 n595 n596 n403)
( o4 n598 n599 n600 n601 n408)
( o2 n503 n509 n453)
( o3 n478 n504 n510 n455)
( o2 n478 n482 n460)
( o2 n478 n502 n465)
( dalf n620 n190 n189 n473 clk2 scantest scanin )
( a2 n174 n473 n478)
( a2 n194 n473 n481)
( a2 n175 n473 n482)
( a2 n122 n473 n487)
( a2 n194 n487 n491)
( a2 n187 n473 n493)
( a2 n121 n487 n498)
( a2 n171 n473 n502)
( a2 n213 n473 n503)
( a2 n172 n473 n504)
( a2 n176 n473 n506)
( a2 n177 n473 n508)
( a2 n213 n473 n509)
( a2 n173 n473 n510)
( a2 n179 n473 n511)
( a2 n180 n473 n513)
( a2 n181 n473 n515)
( a2 n182 n473 n517)
( a2 n183 n473 n519)
( a2 n184 n473 n521)
( a2 n185 n473 n523)
( a2 n186 n473 n525)
( a2 n179 n478 n563)
( a2 n112 n504 n564)
( a2 n126 n508 n565)
( a2 n134 n506 n566)
( a2 n178 n473 n567)
( a2 n180 n478 n568)
( a2 n113 n504 n569)
( a2 n127 n508 n570)
( a2 n135 n506 n571)
( a2 n181 n478 n573)
( a2 n114 n504 n574)
( a2 n128 n508 n575)
( a2 n136 n506 n576)
( a2 n182 n478 n578)
( a2 n115 n504 n579)
( a2 n129 n508 n580)
( a2 n137 n506 n581)
( a2 n183 n478 n583)
( a2 n116 n504 n584)
( a2 n130 n508 n585)
( a2 n138 n506 n586)
( a2 n184 n478 n588)
( a2 n117 n504 n589)
( a2 n131 n508 n590)
( a2 n139 n506 n591)
( a2 n185 n478 n593)
( a2 n118 n504 n594)
( a2 n132 n508 n595)
( a2 n140 n506 n596)
( a2 n186 n478 n598)
( a2 n119 n504 n599)
( a2 n133 n508 n600)
( a2 n141 n506 n601)
( a2 n188 n473 n603)
( a2 n104 n603 n612)
( a2 n105 n603 n613)
( a2 n106 n603 n614)
( a2 n107 n603 n615)
( a2 n108 n603 n616)
( a2 n109 n603 n617)
( a2 n110 n603 n618)
( a2 n111 n603 n619)
( a2 n473 n314 n620)
( a2 n112 n100 n632)
( a2 n113 n100 n633)
( a2 n114 n100 n634)
( a2 n115 n100 n635)
( a2 n116 n100 n636)
( a2 n117 n100 n637)
( a2 n118 n100 n638)
( a2 n119 n100 n639)
( a2 n112 n101 n649)
( a2 n113 n101 n650)
( a2 n114 n101 n651)
( a2 n115 n101 n652)
( a2 n116 n101 n653)
( a2 n117 n101 n654)
( a2 n118 n101 n655)
( a2 n119 n101 n656)
```

Figure 4.8: Second part of netlist description for one node queue.

```
(include "delf.net")
(include "drfl.net")
(include "dslf.net")
(include "array.sub.net")
(include "standard.def")
(include "decaf.def.W")
(include "others.def.W")
( node n100 n101 n102 n103 n104 n105 n106 n107 n108 n109)
( node n110 n111 n112 n113 n114 n115 n116 n117 n118 n119)
( node n120 n121 n122 n123 n124 n125 n126 n127 n128 n129)
( node n130 n131 n132 n133 n134 n135 n136 n137 n138 n139)
( node n140 n141 n142 n143 n144 n145 n146 n147 n148 n149)
( node n150 n151 n152 n153 n154 n155 n156 n157 n158 n159)
( node n160 n161 n162 n163 n164 n165 n166 n167 n168 n169)
( node n170 n171 n172 n173 n174 n175 n176 n177 n178 n179)
( node n180 n181 n182 n183 n184 n185 n186 n187 n188 n189)
( node n190 n192 n196 n304 n321 n325 n326 n327 n329 n330)
( node n332 n333 n334 n335 n337 n338 n339 n340 n341 n363)
( node n368 n373 n378 n383 n388 n393 n398 n443 n445 n450)
( node n458 n463 n464 n465 n470 n474 n476 n481 n485 n486)
( node n487 n489 n491 n492 n493 n494 n496 n498 n500 n502)
( node n504 n506 n508 n546 n547 n548 n549 n550 n551 n552)
( node n553 n554 n556 n557 n558 n559 n561 n562 n563 n564)
( node n566 n567 n568 n569 n571 n572 n573 n574 n576 n577)
( node n578 n579 n581 n582 n583 n584 n586 n587 n588 n589)
( node n590 n591 n592 n593 n594 n606 n607 n608 n609 n610)
( node n611 n612 n613 n623 n624 n625 n626 n627 n628 n629)
( node n630)
( node clk2 scantest Reset scanin )
( pin clk2 TOP 0 100 "pintype=clock" )
( pin scantest TOP 0 100 "pintype=pi" )
( pin scanin TOP 0 100 "pintype=clock" )
( pin clk2 BOTTOM 0 100 "pintype=clock" )
( pin scantest BOTTOM 0 100 "pintype=pi" )
( pin scanin BOTTOM 0 100 "pintype=clock" )
( delf n474 n189 n476 n100 clk2 scantest scanin Reset )
( delf n481 n189 n476 n101 clk2 scantest scanin Reset )
( delf n458 n189 n491 n102 clk2 scantest scanin Reset )
( delf n458 n189 n489 n103 clk2 scantest scanin Reset )
( delf n363 n189 n550 n104 clk2 scantest scanin Reset )
( delf n368 n189 n550 n105 clk2 scantest scanin Reset )
( delf n373 n189 n550 n106 clk2 scantest scanin Reset )
( delf n378 n189 n550 n107 clk2 scantest scanin Reset )
( delf n383 n189 n550 n108 clk2 scantest scanin Reset )
( delf n388 n189 n550 n109 clk2 scantest scanin Reset )
( delf n393 n189 n550 n110 clk2 scantest scanin Reset )
( delf n398 n189 n550 n111 clk2 scantest scanin Reset )
( delf n494 n189 n465 n112 clk2 scantest scanin Reset )
( delf n496 n189 n465 n113 clk2 scantest scanin Reset )
( delf n498 n189 n465 n114 clk2 scantest scanin Reset )
( delf n500 n189 n465 n115 clk2 scantest scanin Reset )
( delf n502 n189 n465 n116 clk2 scantest scanin Reset )
( delf n504 n189 n465 n117 clk2 scantest scanin Reset )
( delf n506 n189 n465 n118 clk2 scantest scanin Reset )
( delf n508 n189 n465 n119 clk2 scantest scanin Reset )
( delf n443 n189 n445 n120 clk2 scantest scanin Reset )
( delf n458 n189 n450 n121 clk2 scantest scanin Reset )
( delf n458 n189 n465 n122 clk2 scantest scanin Reset )
( pin n123 BOTTOM 0 100 "pintype=pi")
( pin n124 BOTTOM 0 100 "pintype=pi")
( pin n125 BOTTOM 0 100 "pintype=pi")
( pin n126 BOTTOM 0 100 "pintype=pi")
( pin n127 BOTTOM 0 100 "pintype=pi")
( pin n128 BOTTOM 0 100 "pintype=pi")
( pin n129 BOTTOM 0 100 "pintype=pi")
( pin n130 BOTTOM 0 100 "pintype=pi")
( pin n131 BOTTOM 0 100 "pintype=pi")
( pin n132 BOTTOM 0 100 "pintype=pi")
( pin n133 BOTTOM 0 100 "pintype=pi")
( pin n134 BOTTOM 0 100 "pintype=pi")
( pin n135 BOTTOM 0 100 "pintype=pi")
( pin n136 BOTTOM 0 100 "pintype=pi")
( pin n137 BOTTOM 0 100 "pintype=pi")
( pin n138 BOTTOM 0 100 "pintype=pi")
( pin n139 BOTTOM 0 100 "pintype=pi")
( pin n140 BOTTOM 0 100 "pintype=pi")
( pin n141 BOTTOM 0 100 "pintype=po")
( pin n141 TOP 0 100 "pintype=po")
( connect n141 n586 )
( pin n142 TOP 0 100 "pintype=po")
( pin n142 BOTTOM 0 100 "pintype=po")
( connect n142 n587 )
( pin n143 TOP 0 100 "pintype=po")
( pin n143 BOTTOM 0 100 "pintype=po")
( connect n143 n588 )
( pin n144 TOP 0 100 "pintype=po")
( pin n144 BOTTOM 0 100 "pintype=po")
( connect n144 n589 )
( pin n145 TOP 0 100 "pintype=po")
( pin n145 BOTTOM 0 100 "pintype=po")
( connect n145 n590 )
( pin n146 TOP 0 100 "pintype=po")
( pin n146 BOTTOM 0 100 "pintype=po")
( connect n146 n591 )
( pin n147 TOP 0 100 "pintype=po")
( pin n147 BOTTOM 0 100 "pintype=po")
( connect n147 n592 )
( pin n148 TOP 0 100 "pintype=po")
( pin n148 BOTTOM 0 100 "pintype=po")
( connect n148 n593 )
( pin n149 TOP 0 100 "pintype=po")
( pin n149 BOTTOM 0 100 "pintype=po")
( connect n149 n121 )
( pin n150 TOP 0 100 "pintype=po")
( pin n150 BOTTOM 0 100 "pintype=po")
( connect n150 n100 )
( pin n151 TOP 0 100 "pintype=po")
( pin n151 BOTTOM 0 100 "pintype=po")
( connect n151 n102 )
( pin n152 TOP 0 100 "pintype=po")
( pin n152 BOTTOM 0 100 "pintype=po")
( connect n152 n101 )
( pin n153 TOP 0 100 "pintype=po")
( pin n153 BOTTOM 0 100 "pintype=po")
( connect n153 n103 )
( pin n154 BOTTOM 0 100 "pintype=po")
( connect n154 n606 )
( pin n155 BOTTOM 0 100 "pintype=po")
( connect n155 n607 )
( pin n156 BOTTOM 0 100 "pintype=po")
( connect n156 n608 )
( pin n157 BOTTOM 0 100 "pintype=po")
( connect n157 n609 )
( pin n158 BOTTOM 0 100 "pintype=po")
```

Figure 4.9: First part of netlist description for one node stack.

```
( pin n159 BOTTOM 0 100 "pintype=po") ( connect n159 n611 )
( pin n160 BOTTOM 0 100 "pintype=po") ( connect n160 n612 )
( pin n161 BOTTOM 0 100 "pintype=po") ( connect n161 n613 )
( pin n162 BOTTOM 0 100 "pintype=po") ( connect n162 n623 )
( pin n163 BOTTOM 0 100 "pintype=po") ( connect n163 n624 )
( pin n164 BOTTOM 0 100 "pintype=po") ( connect n164 n625 )
( pin n165 BOTTOM 0 100 "pintype=po") ( connect n165 n626 )
( pin n166 BOTTOM 0 100 "pintype=po") ( connect n166 n627 )
( pin n167 BOTTOM 0 100 "pintype=po") ( connect n167 n628 )
( pin n168 BOTTOM 0 100 "pintype=po") ( connect n168 n629 )
( pin n169 BOTTOM 0 100 "pintype=po") ( connect n169 n630 )
( pin n170 BOTTOM 0 100 "pintype=po") ( connect n170 n333 )
( pin n171 BOTTOM 0 100 "pintype=po") ( connect n171 n338 )
( pin n172 BOTTOM 0 100 "pintype=po") ( connect n172 n341 )
( pin n173 BOTTOM 0 100 "pintype=po") ( connect n173 n325 )
( pin n174 BOTTOM 0 100 "pintype=po") ( connect n174 n326 )
( pin n175 BOTTOM 0 100 "pintype=po") ( connect n175 n327 )
( pin n176 BOTTOM 0 100 "pintype=po") ( connect n176 n329 )
( pin n177 BOTTOM 0 100 "pintype=po") ( connect n177 n321 )
( pin n178 TOP 0 100 "pintype=pi")
( pin n179 TOP 0 100 "pintype=pi")
( pin n180 TOP 0 100 "pintype=pi")
( pin n181 TOP 0 100 "pintype=pi")
( pin n182 TOP 0 100 "pintype=pi")
( pin n183 TOP 0 100 "pintype=pi")
( pin n184 TOP 0 100 "pintype=pi")
( pin n185 TOP 0 100 "pintype=pi")
( pin n186 TOP 0 100 "pintype=pi")
( pin n187 TOP 0 100 "pintype=pi")
( pin n188 TOP 0 100 "pintype=reset")
( pin n189 TOP 0 100 "pintype=clock")
( pin n190 TOP 0 100 "pintype=pi")
( pin n123 TOP 0 100 "pintype=pi")
( pin n124 TOP 0 100 "pintype=pi")
( pin n125 TOP 0 100 "pintype=pi")
( pin n126 TOP 0 100 "pintype=pi")
( pin n127 TOP 0 100 "pintype=pi")
( pin n128 TOP 0 100 "pintype=pi")
( pin n129 TOP 0 100 "pintype=pi")
( pin n130 TOP 0 100 "pintype=pi")
( pin n131 TOP 0 100 "pintype=pi")
( pin n132 TOP 0 100 "pintype=pi")
( pin n133 TOP 0 100 "pintype=pi")
( pin n134 TOP 0 100 "pintype=pi")
( pin n135 TOP 0 100 "pintype=pi")
( pin n136 TOP 0 100 "pintype=pi")
( pin n137 TOP 0 100 "pintype=pi")
( pin n138 TOP 0 100 "pintype=pi")
( pin n139 TOP 0 100 "pintype=pi")
( pin n140 TOP 0 100 "pintype=pi")
( pin n178 BOTTOM 0 100 "pintype=pi")
( pin n179 BOTTOM 0 100 "pintype=pi")
( pin n180 BOTTOM 0 100 "pintype=pi")
( pin n181 BOTTOM 0 100 "pintype=pi")
( pin n182 BOTTOM 0 100 "pintype=pi")
( pin n183 BOTTOM 0 100 "pintype=pi")
( pin n184 BOTTOM 0 100 "pintype=pi")
( pin n185 BOTTOM 0 100 "pintype=pi")
( pin n186 BOTTOM 0 100 "pintype=pi")
( pin n187 BOTTOM 0 100 "pintype=pi")
( pin n188 BOTTOM 0 100 "pintype=reset")
( pin n189 BOTTOM 0 100 "pintype=clock")
( pin n190 BOTTOM 0 100 "pintype=pi") ( connect n188 Reset)
( i1 n120 n192) ( i1 n121 n198)
( i1 n190 n304) ( o4 n171 n186 n176 n175 n321)
( a2 n198 n186 n325) ( a2 n121 n186 n326)
( a2 n120 n172 n327) ( a2 n192 n172 n329)
( o3 n123 n124 n122 n330) ( i1 n330 n332)
( a2 n187 n332 n333) ( o3 n123 n124 n335 n334)
( i1 n122 n335) ( i1 n334 n337) ( a2 n187 n337 n338)
( i1 n170 n339) ( i1 n171 n340) ( a3 n340 n339 n187 n341)
( o4 n546 n547 n548 n549 n363) ( o4 n551 n552 n553 n554 n368)
( o4 n556 n557 n558 n559 n373) ( o4 n561 n562 n563 n564 n378)
( o4 n566 n567 n568 n569 n383) ( o4 n571 n572 n573 n574 n388)
( o4 n576 n577 n578 n579 n393) ( o4 n581 n582 n583 n584 n398)
( o3 n464 n486 n492 n443) ( o4 n463 n465 n487 n493 n445)
( o2 n463 n485 n450) ( dsff n594 n189 n188 n458 clk2 scantest
scanin ) ( a2 n173 n458 n463) ( a2 n192 n458 n464) ( a2 n174
n458 n465) ( a2 n121 n458 n470) ( a2 n120 n470 n474)
( a2 n186 n458 n476) ( a2 n192 n470 n481) ( a2 n170 n458 n485)
( a2 n192 n458 n486) ( a2 n171 n458 n487) ( a2 n175 n458 n489)
( a2 n176 n458 n491) ( a2 n192 n458 n492) ( a2 n172 n458 n493)
( a2 n104 n458 n494) ( a2 n105 n458 n496) ( a2 n106 n458 n498)
( a2 n107 n458 n500) ( a2 n108 n458 n502) ( a2 n109 n458 n504)
( a2 n110 n458 n506) ( a2 n111 n458 n508) ( a2 n178 n476 n546)
( a2 n112 n487 n547) ( a2 n125 n491 n548) ( a2 n133 n489 n549)
( a2 n177 n458 n550) ( a2 n179 n476 n551) ( a2 n113 n487 n552)
( a2 n126 n491 n553) ( a2 n134 n489 n554) ( a2 n180 n476 n556)
( a2 n114 n487 n557) ( a2 n127 n491 n558) ( a2 n135 n489 n559)
( a2 n181 n476 n561) ( a2 n115 n487 n562) ( a2 n128 n491 n563)
( a2 n136 n489 n564) ( a2 n182 n476 n566) ( a2 n116 n487 n567)
( a2 n129 n491 n568) ( a2 n137 n489 n569) ( a2 n183 n476 n571)
( a2 n117 n487 n572) ( a2 n130 n491 n573) ( a2 n138 n489 n574)
( a2 n184 n476 n576) ( a2 n118 n487 n577) ( a2 n131 n491 n578)
( a2 n139 n489 n579) ( a2 n185 n476 n581) ( a2 n119 n487 n582)
( a2 n132 n491 n583) ( a2 n140 n489 n584) ( a2 n104 n458 n586)
( a2 n105 n458 n587) ( a2 n106 n458 n588) ( a2 n107 n458 n589)
( a2 n108 n458 n590) ( a2 n109 n458 n591) ( a2 n110 n458 n592)
( a2 n111 n458 n593) ( a2 n458 n304 n594) ( a2 n112 n100 n606)
( a2 n113 n100 n607) ( a2 n114 n100 n608) ( a2 n115 n100 n609)
( a2 n116 n100 n610) ( a2 n117 n100 n611) ( a2 n118 n100 n612)
( a2 n119 n100 n613) ( a2 n112 n101 n623) ( a2 n113 n101 n624)
( a2 n114 n101 n625) ( a2 n115 n101 n626) ( a2 n116 n101 n627)
( a2 n117 n101 n628) ( a2 n118 n101 n629)( a2 n119 n101 n630)
```

Figure 4.10: Second part of netlist description for one node stack.

substituted in the place of old ones. The entire system is controlled with a single data flow supervisor program to ensure that data consistency is maintained at all stages of the design. The data flow supervisor is template driven, the templates used in OASIS can easily be expanded to support additional software tools.

The layouts produced with the OASIS system use standard cells of uniform height. The cells are placed in an array of horizontal rows and all interconnection of nets are made by channel routing in the space between the adjacent rows. All connections of the power nets, Vdd and GND, are made by abutting the cell horizontally. Signal nets connecting cells belonging to non-adjacent rows cross the intermediate by utilizing feed-through pins (vertical strips of metal running from top to bottom of the cell) built into some cells, or by inserting feed-through cells.

A set of scalable CMOS cells compatible with the 2μ SCMOS technology are used. Input and output ports are made in the second layer of the metal. The Vdd and GND power nets from each row of cells are connected together using power rails running vertically through the entire height of the layout.

4.3.1 Placement

The placement and routing in OASIS is done using a layout subsystem of OASIS called **VPNR** (Vanilla Place aNd Route). The tools comprising VPNR create physical layouts automatically from netlist description of logic circuits, using a library of pre-designed standard cells of uniform height. The goal of VPNR is to place

cells and perform global and detail routing of interconnections in a plane so as to minimize the layout area.

VPNR is usually invoked at the end of the entire design process after the design has been simulated and verified to implement the desired functionality. Occasionally, the layout may be generated in the early stages of the design to obtain an estimate of the area taken up by the circuit. VPNR employs placement and global routing algorithms based upon the quadrisection paradigm [27]. The combined placement and global routing program receives a netlist of standard cells, partitions it into four quadrants (top-left, top-right, bottom-left and bottom-right), and processes each quadrant in the same manner until each quadrant contains one cell row in the vertical direction. Partitioning is accompanied and directed by approximate global routing.

Once the layout of one node is made, the placement of the entire tree (15-node) is done interactively using the Magic's layout editor. For three node data structure the nodes are placed in the inverted "L" shape with the root node is at the corner, and for 15 node and 63 node data structure the placement is as explained in the previous chapter.

4.3.2 Routing

Within the Cell Routing

Within the cell routing is carried out by VPNR which is a subsystem of OASIS. After the position of the cells in rows are determined, the interconnections comprising the scan chain can be chosen. The algorithm applied here is a simple snake-like path threading through the cells consisting solely of a single feed-through.

The next stage is detail global routing. The algorithm constructs a minimum spanning tree for each net, finds the exact crossing locations for nets that need to cross the cell rows, insert feed-through cells if necessary and assigns sub-nets in the channels. The nets are processed sequentially and the routing of each net takes into account all nets routed previously. After all nets are routed, the global router prepares the data for detailed channel routing. Global routing derived at the preceding stage does not include the global nets disregarded at the placement stage. These nets are assumed to follow a fix routing scheme. They cross the channel vertically in the vicinity of the vertical power rails and extend into channels horizontally to the left and right of vertical rails. Usually the horizontal extension of a given net occur in every second channel. The routing of the global nets involves inserting the vertical power signals together with the adjacent feed-through cells for the global nets, and adjusting the data for channel routing to accommodate the vertical inserts.

Each channel definition contains a list of groups of pins that will be connected. The channel router determines where to put the wires so the resulting layout occupies least amount of area. VPNR provides a choice of two channel routers: a greedy router [23] and left edge based router with channel compaction [18].

Routing between the nodes

After the placement of the modules is performed, routing is done with Magic's interactive router. A netlist file is generated with extension *.net*, and can be used for future interconnection. The VLSI layout of single node of a queue (stack) are shown in the Figure 4.11 (Figure 4.18) has dual I/O terminals, both at the top and at the bottom of the layout. Magic's router decides which of the pins, whether at the top or from bottom are to be chosen for interconnection.

4.4 Layout of systolic tree-based queue

After the generation of layout of a single node, the corresponding circuit is extracted using Magic's hierarchical circuit extractor. The extracted circuit is simulated at the transistor (switch) level using *irsim*, a switch-level simulator of Magic layout editor [20]. The simulation results are observed on another viewing tool called *ana*. Different size layouts can be made by recursively placing and routing of modules as discussed in the earlier chapter. The transistor level simulation results are compared

with RTL/gate-level simulation results. The layout of one node systolic queue and the simulation results are shown in the Figures 4.11 and 4.12 respectively. Layout and simulation result of three node queue is shown in the Figures 4.13 and 4.14. Layout of a fifteen node queue is shown in Figure 4.15, simulation result of seven node queue is shown in the Figure 4.16 and lastly, the layout of sixty three node queue is shown in the Figure 4.17.

Since the architecture is systolic, to verify the correctness of design, it is sufficient to demonstrate the simulation for three nodes of the queue. The layout of the first three nodes, that is, root and two children, is extracted, and the extracted circuit is simulated to verify the functional correctness. The entire queue is also simulated for functional correctness.

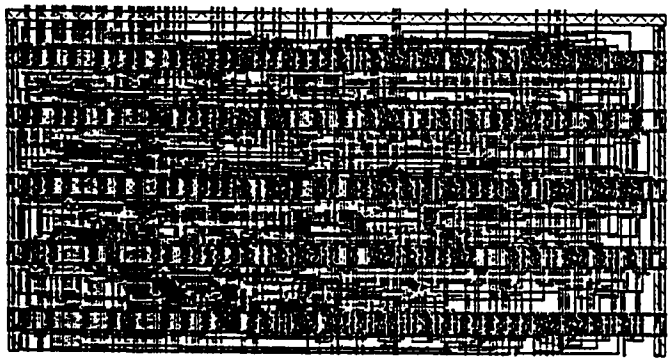


Figure 4.11: Layout of one node.

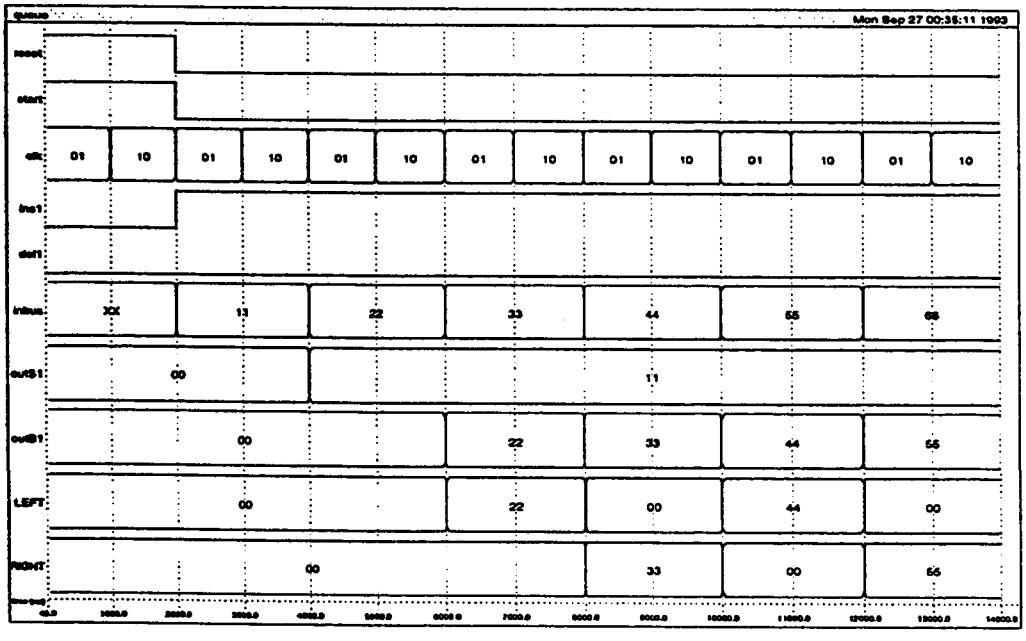


Figure 4.12: Transistor-level simulation of one node of queue.

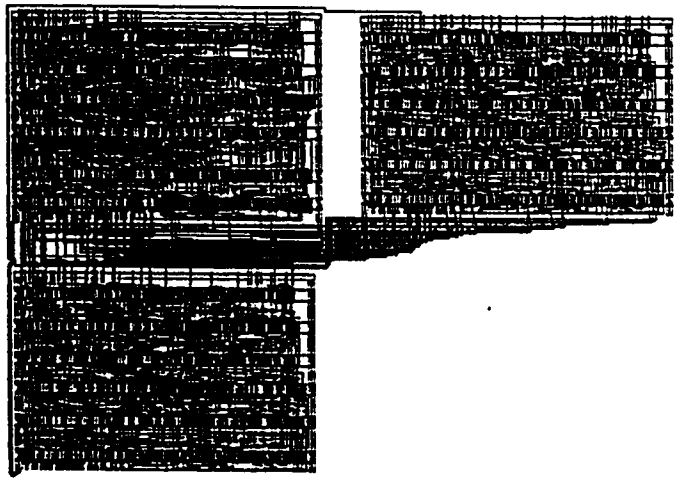


Figure 4.13: Layout of three node queue.

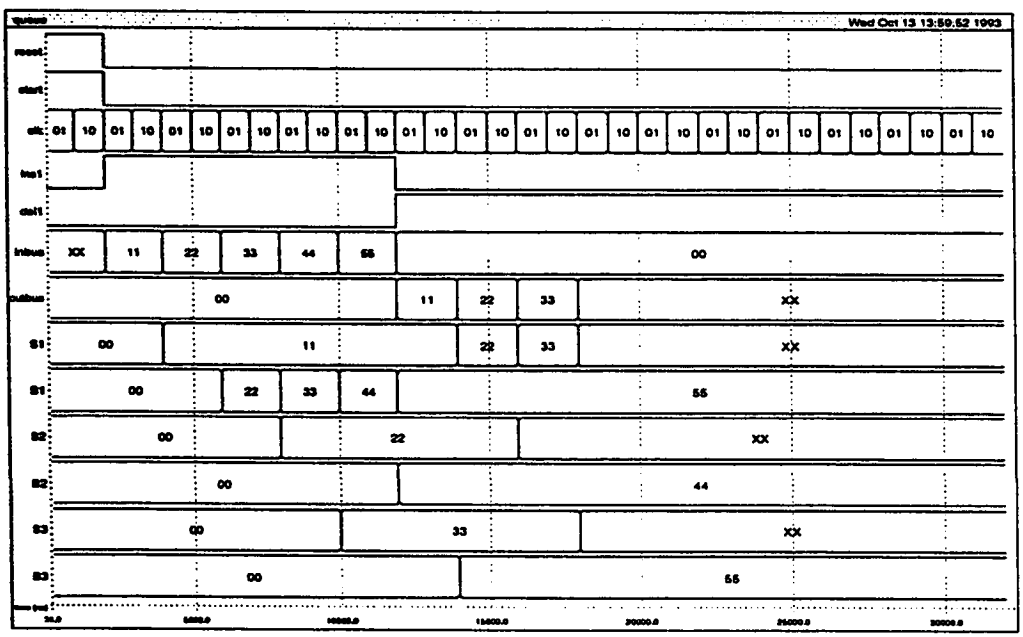


Figure 4.14: Transistor-level simulation of three node queue.

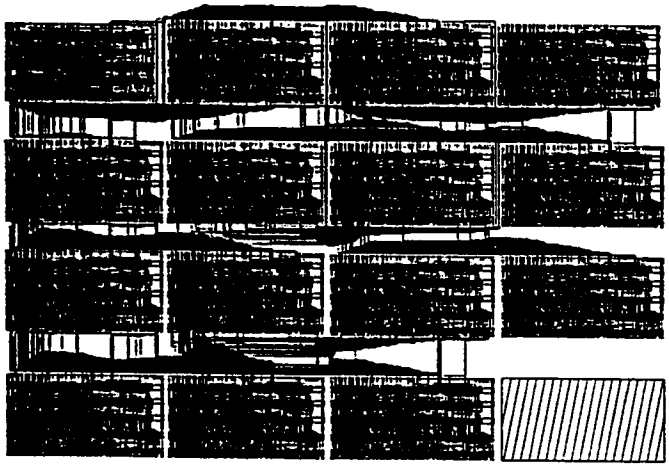


Figure 4.15: Layout of fifteen node queue.

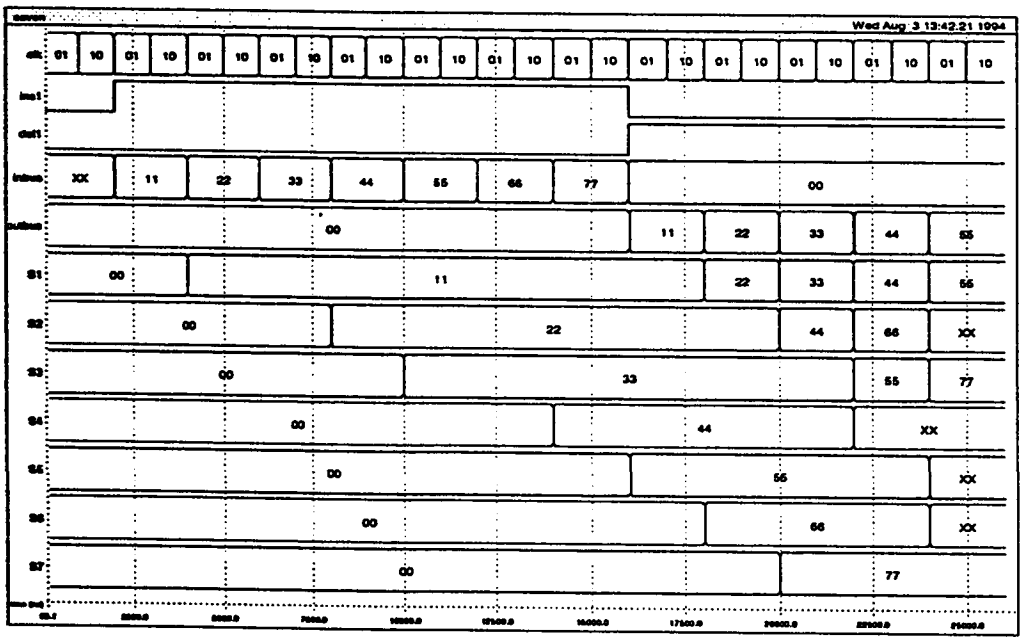


Figure 4.16: Transistor-level simulation of seven node queue.

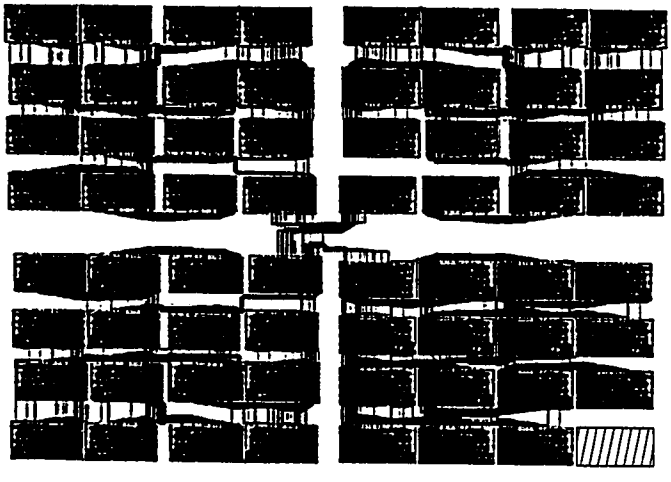


Figure 4.17: Layout of sixty three node queue.

4.5 Layout of systolic tree-based stack

After the layout is made for one node stack using OASIS, the nodes are placed in the form of 2D-grid as discussed in [31]. Routing is performed between the nodes and the desired size stack is constructed. The modules $M1$, $M2$, $M3$ and $M4$ have different interconnections and the empty node is located at different corners. The routing is performed using Magic's interactive router, once the netlist is prepared it can be used for interconnections of different placed cells.

The circuit is extracted from the layout using Magic's hierarchical circuit extractor. The layout is simulated at the transistor/switch level using *irsim*, a switch level simulator of Magic's layout editor for CMOS circuits [20]. The simulation shows the functional correctness of the layout.

The layout of one node stack and the simulation results are shown in Figures 4.18 and 4.19 respectively. Layout and simulation results of three node stack is shown in Figures 4.20 and 4.21. Layout of fifteen nodes and simulation result of the seven node circuit is shown in the Figures 4.22 and 4.23.

4.6 Conclusion

The implementation process from RTL to layout generation is described. After the laying of nodes, routing is performed interactively using Magic's router, and then the layout is simulated to verify for functional correctness.

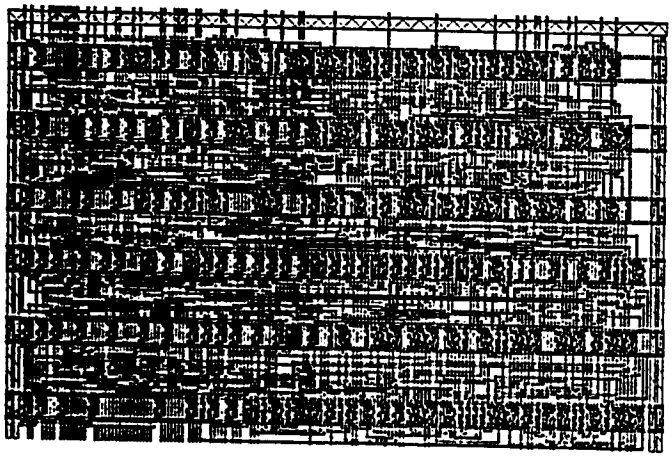


Figure 4.18: Layout of one node stack.

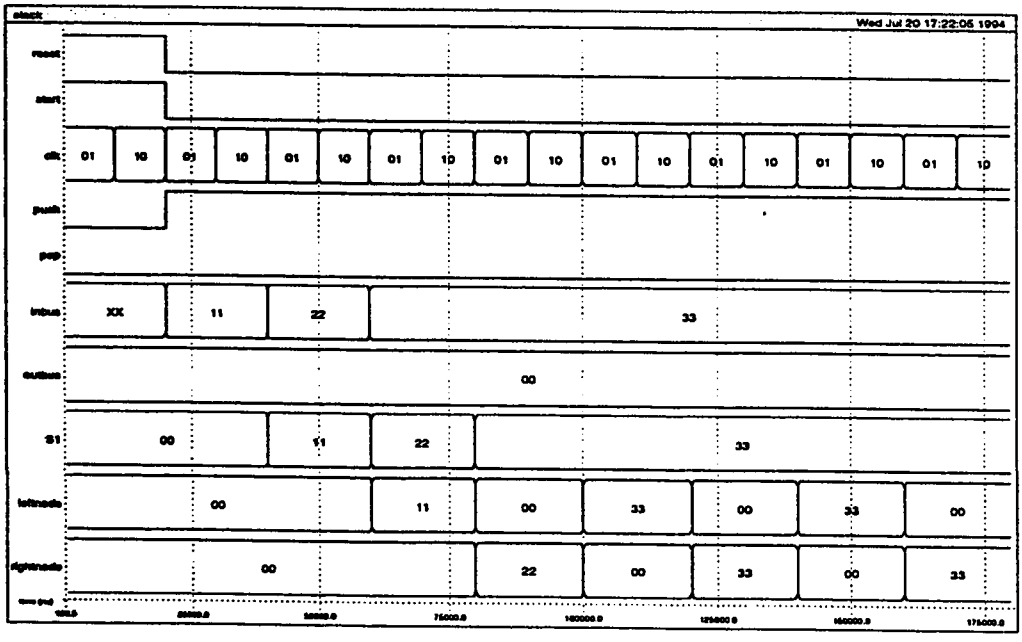


Figure 4.19: Transistor-level simulation of one node stack.

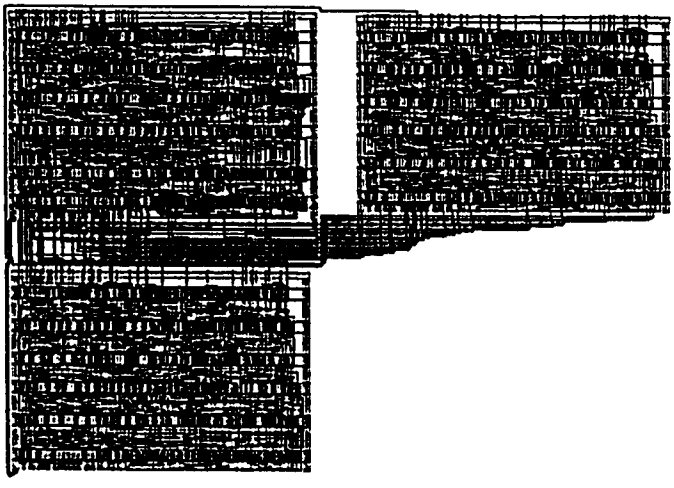


Figure 4.20: Layout of three node stack.

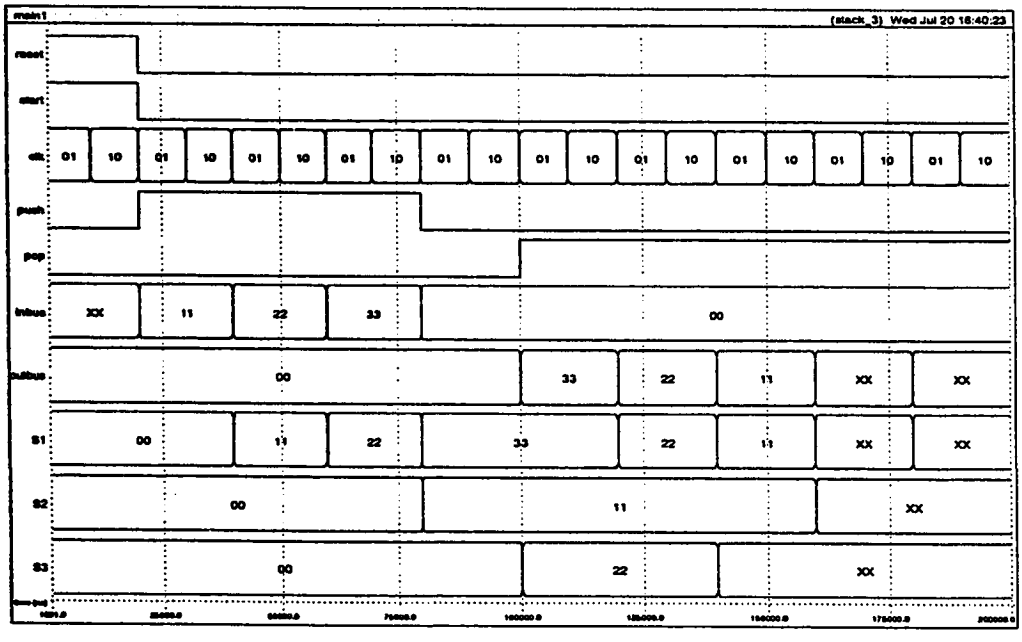


Figure 4.21: Transistor-level simulation of three node stack.

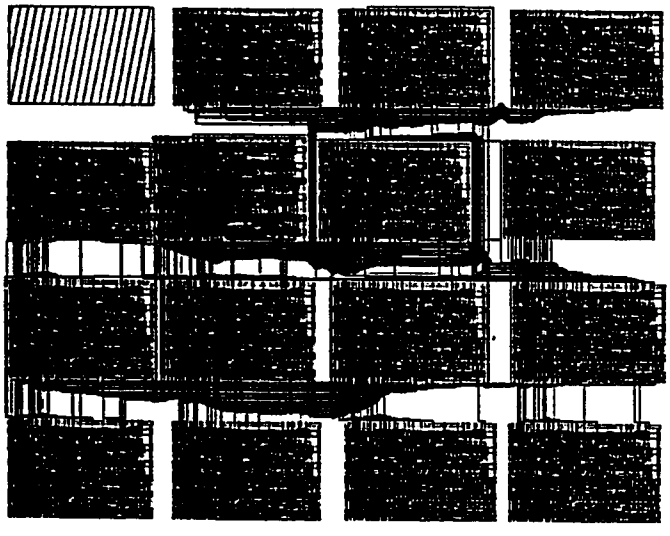


Figure 4.22: Layout of fifteen node stack.

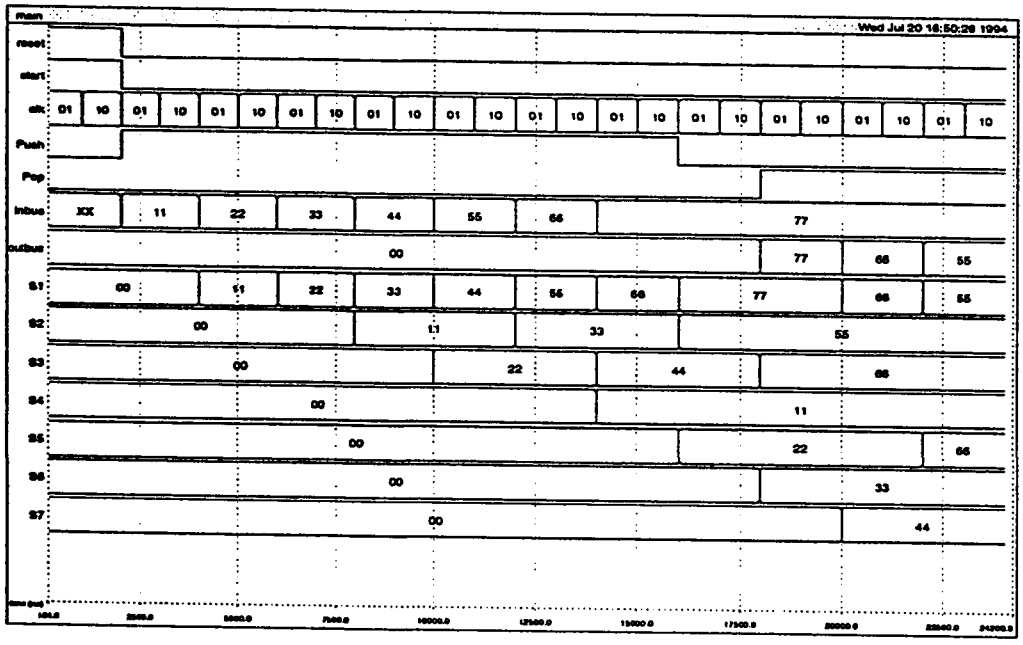


Figure 4.23: Transistor-level simulation of seven node stack.

Chapter 5

Discussion and Conclusions

A framework for the VLSI implementation of data structures have been presented in this thesis. This work provides insight into the issues involved in physical VLSI implementation of architectures based on binary trees. It also confirms that such architectures have very efficient VLSI implementations.

The layout methodology relies on the following VLSI design tools:

1. AHPL DA system for RT level modeling, functional simulation and netlist generation,

2. RNL for logic level simulator,

3. VPNR subsystem of OASIS for placement and routing of standard cells, and

4. Magic for layout, routing, circuit extraction from layout, and simulation of extracted circuit (using Magic's *irsim*). Design rules for layout, and the tech-

nology files for simulation used were obtained from MOSIS. A MOSIS pad frame is used to complete the layout for fabrication.

The logic/transistor level simulation results obtained using RNL/*irsim* were observed using viewing tools SigView/ana respectively. The simulation results were in conformity with the functional level simulation results. This verification greatly enhances the chance that the chip will work correctly.

The total area of the 15-node layout is $33.9mm^2$, and of this, 30% is used by the router (including dead space). Larger bus width data structures can be constructed easily by only modifying the size of buses and registers in the RT-level model. The bus-width is limited by the I/O pins on the IC package. The entire process can be easily automated. For design of other data structures such as priority queue, dictionary machine, and dequeue etc., only the RT level model has to be modified. The overhead in terms of number of flags and buffer registers per node is high, but the ease of implementation (modularity, local interconnectivity, etc.,) justify designing ICs using such architectures.

The work presented can be continued on practical VLSI realization of such architectures proposed for a variety of applications. Examples include, systolic tree architectures for other data structures [4], systolic tree implementation of data compression algorithms [29], etc. Different embedding strategies were reviewed in this thesis, namely H-tree embedding, hexagonal tree embedding and tile-based tree embedding. The tile based approach is most area efficient in terms of PE utiliza-

tion [31]. As an example, the complete layout of 63 node binary tree architecture implementation of a queue was presented.

Appendix A

Tools Used

UAHPL Based Design Automation System

UAHPL is an abbreviation of Universal A Hardware Programming Language. It provides several advantages to digital design system designers. UAHPL allows the design engineer to describe his circuit at an adequate level of abstraction namely the register transfer level. UAHPL provides the capability to express large iterative combinational logic networks (CLUs). It allows modeling concurrency, parallel processing, pipelining and a multimode system, thereby, allowing a well structured and modular description of the digital systems. Despite the numerous advantages of its modular approach, UAHPL lacks in certain constructs and features, which are necessary for efficient realization and testing of digital systems. UAHPL has been developed after considering the various design and test activities in different environments. UAHPL allows a better choice of register types, clocking options and bus types. It accommodates pass transistors, wired-OR gates, temporary registers, and other elements. UAHPL has good software support with a multi-stage compiler that produces the logical circuits network and simulation facilities to check the functional behavior of the circuit. The UAHPL language is described below.

UAHPL Syntax

UAHPL is a modified version of AHPL, a register transfer language. The language is based on APL, thereby, providing simple constructs for catenation, bit selection, conditional control and branching, which are quite similar to hardware primitives. Modules, Combinational Logic Units (CLUs) and Functional Registers (FNREGs) are the possible segments in a UAHPL description of a digital system. The procedural description of the circuit is given in the module. The module consists of three parts: a declaration part, a procedural part describing the control sequence, and a non-procedural part delimited by keywords ENDSEQUENCE and END. The declaration section includes various buses, registers, inputs and outputs etc. The control sequence has action and branch control. Action includes register transfer,

bus connection, the branch specifies the action to be taken based on the branch conditions. CLUs describe the iterative combinational circuits such as adders, incrementers, decoders etc., FNREGs describe circuits with memory but no sequential control, for example shift registers, counters etc.

The basic constructs and the features of the language are illustrated by the listing of UAHPL description of the nodes of queue and stack. The UAHPL description of these nodes consists of two 8-bit registers and many one-bit flags. Now we will consider the declaration section of node.

Declaration Section

This section consists of all declarations which will be used in the program. It consists of MEMORY, INPUTS, OUTPUTS, BUSES, EXBUSES, EXINPUTS, EXOUTPUTS, CLUs and so on. In MEMORY declaration all the memory elements are declared for example the registers are declared S18 means S1 is the name of register of 8-bits, and all data flip-flops are also declared here. INPUTS and OUTPUTS are used to specify the inputs and output buses or lines for providing the input to and output from the digital system. BUSES are used to connect the modules inside and outside the digital system with different components, EXBUSES are the external buses running between the modules and used to inter connect different modules, EXINPUTS are the external inputs like clock, reset and so on. CLUs are the combinational logic unit such as DEC for decoder, ADD for adder, and INC for incrementer etc. The control sequence begins with the key words BODY SEQUENCE and continuous up to the ENDSEQUENCE and consists of a number of control steps (but one in case of queue and stack).

Procedural Section

The control begins as the start signal is generated until the start becomes one it will be in the first step. After all the operations are performed in the procedural section in this case there are all conditional transfers. The value is written in a output file with an extension output.

The CONTROLRESET statement enclosed in the keywords ENDSEQUENCE and END is active always. The effect of this statement is that the control goes to step 1 whenever the reset key is activated. CLUs are not used in the UAHPL descriptions of queue and stack.

Generation of Netlist from the UAHPL Description

The UAHPL-based system is a multi-stage system with the first two stages generating the logical circuit design of the digital system described in UAHPL. The later stages are technology dependent application software, test, netlist generation and simulation facilities. Several activities are supported by the system at different stages as depicted in [24].

The UAHPL description of a circuit, discussed in the previous section, is input to the first stage compiler which performs syntax and semantic analysis and generates a tabular representation of the circuit. These tables are used by second stage compiler (also called the network synthesizer) to generate the complete circuit in terms of hardware primitives such as logic gates, flip-flops etc. The output of the second stage is stored in the form of a linked list consisting of GATELIST and a IOLIST. Each record contains the GATE, specifying the gate number for the element followed by the TYPE, specifying the type of the element, for example, AND, OR, D flip-flop etc. The two fields in the record, ILINK and OLINK, are pointers to the IOLIST. SYMLINK is another pointer to the IOLIST, in the case of D flip-flop, the register to which the gate belongs. The other field in the GATELIST are for optimization purposes. The first field of the IOLIST is GATPTR, is pointer specified by the ILINK or OLINK. The next two fields are IOLNK1 and IOLNK2, specifying the two elements in the input or the output list of the gate. The last field is NEXPTR which is another pointer to the same list giving the link for the next set of elements in the input and output list. The netlist obtained from the stage 2 compiler is used for later stages in the chip layout with the help of other VLSI design tools.

OASIS

The Open Architecture Silicon Implementation Software, OASIS, is a cell-based silicon compiler system that supports rapid implementation of digital Application Specific ICs (ASICs). By taking a high level language or a netlist description and producing a layout of testable IC. The tools integrated into the OASIS system have been developed to automatically translate high level descriptions of integrated circuits into testable physical layouts, using predesigned standard cells. There are five basic subsystem of the OASIS.

1. Logic synthesis and Optimization,
2. Simulation and Verification,
3. Test Pattern Generation,
4. Placement and Routing,

5. Standard Cell Compilation

The complex set of tools is integrated into a coherent design environment with the aid of sophisticated design management and control software. The OASIS system successfully integrates existing public domain tools while adding major new components. Several application specific ICs were designed with OASIS. OASIS system dramatically increases designers productivity. Large and complex ICs can be designed in few months only. The use of a high-level language combines with logic synthesis and automatic placement and routing relived the designer from the chores of dealing with low-level details,thus enable him/her to concentrate on more creative aspects of design.

RNL

RNL is a timing logic simulator for digital MOS circuits. It is an event driven simulator that uses a simple RC (Resistance Capacitance) model of the circuit to estimate the effects of charge sharing. The user interface is a simple LISP interface. This allows both interactive simulation and the programming of complex simulation.

To use RNL, the "filename.sim" file for the circuit to be simulated is required. This "filename.sim" file is converted to a binary file using *presim*. The RNL uses that binary file for simulation. It is designed to handle the ratioed logic, bidirectionally, and charge sharing/storage. They can be used to determine the functionality and approximate timing timing behavior of circuits commonly found in the digital designs. Basic to the operation of the simulators is the notion of an event. specifies, a node in the network, a new logic state, and a time at which the node's value is changed to the new logic state.

RNL maintains a list of events, sorted by time, that tells what processing remains to be done. Whenever the input is changed, an event is added to the list; when the list is empty the network has "settled" and RNL waits for further input. When started with an initial list, RNL sequentially processes the next event on the list, stopping when the list is empty or when a node is needed to be traced or when the specified amount of simulated time has elapsed.

Since nodes are only added to the event list when their values change, portions of circuit unaffected by the current set of changes to the inputs are not re-evaluated. The algorithm is event driven sometimes called selective trace.

Appendix B

AHPL2NET Program Listing

```
#include "stdio.h"
#include "stdlib.h"
#include "string.h"

void andgate(), nandgate(), orgate(), norgate(),
     ioclu(),xorgate(), dfcs(), memck(),buses(),
     exin(),exout();
FILE *cpin,
     *gat,
     *out,
     *io,
     *exinp,
     *exbp,
     *exoutp;
int gtype,
    x,
    y,
    ilink,
    i1,
    i2,
    p1,
    p2,
    p3,
    nout;
```

```
char place[7],
     pinlab[10];
main()
{
  int i;
  char str[80],str1[80],str2[80];
     if ((out=fopen("net.out","w"))==NULL) {
        printf("Can't open file\n");
 exit(1);
}
     if ((gat=fopen("gate.c","r"))==NULL) {
         printf("Can't open file\n");
  exit(1);
}
      while(!feof(gat)){
                 fprintf(out,"( node");
         for(i=1; i<=10;i++) {
                fgets(str,80,gat);
                        if (feof(gat)) break;
                                sscanf(str,"%d",&nout);
                                fprintf(out," n%d",nout);
                }
                 fprintf(out,")\n");
        }
      fclose(gat);

       /*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/

      fprintf(out,"( node clk2 scantest Reset scanin )\n");
      if ((cpin=fopen("iopins.c","r"))==NULL) {
           printf("Can't open file\n");
           exit(1);
          }
    while(!feof(cpin)) {
     fgets(str1,80,cpin);
     if (feof(cpin)) break;
      sscanf(str1,"%s%s%2d%3d",
pinlab,place,&x,&y);
     if (strcmp(pinlab,"clk2")==0)
      fprintf(out,"( pin %s %s %d %d
\"pintype=%s\" )\n",pinlab,place,x,y,pinlab);
```

```
        if (strcmp(pinlab,"scantest")==0)
         fprintf(out,"( pin %s %s %d %d
\"pintype=%s\" )\n",pinlab,place,x,y,pinlab);
        if (strcmp(pinlab,"scanin")==0)
         fprintf(out,"( pin %s %s %d %d
\"pintype=%s\" )\n",pinlab,place,x,y,pinlab);
          }
         fclose(cpin);

          /*%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%*/

    /*     if ((exoutp=fopen("exo.c","r"))==NULL) {
           printf("Can't open file\n");
            exit(1);
         }
         if ((exinp=fopen("exi.c","r"))==NULL) {
           printf("Can't open file\n");
            exit(1);
         }*/
         if ((gat=fopen("gate.c","r"))==NULL) {
           printf("Can't open file\n");
            exit(1);
         }
      while(!feof(gat)){
        fgets(str2,80,gat);
        if(feof(gat)) break;
        sscanf(str2,"%d%10d%10d",&nout,&gtype,&p1);
      switch (gtype){
          case  4001:
 case  4029:
 case  4031: andgate();
break;
          case 4002:
 case  4032: nandgate();
break;
          case 4003:
 case  4033: orgate();
break;
 case 4004:
 case  4034: xorgate();
break;
```

```
 case 4005: norgate();
break;
 case 4006: dfcs();
break;
 case 4009: memck();
break;
 case 4025:
 case 4026: ioclu();
break;
        case   4014: buses();
break;
 case 4018: exin();
break;
 case 4020:
        case 4013: exout();
break;
default:
               fprintf(out,"No option selected\n");
}
/*This is end of switch*/
      } /*end of while loop*/
      fclose(gat);
      fclose(exinp);
      fclose(exoutp);
      fclose(out);
} /*end of main*/

void andgate(void)
{
  int count,j,inout[10];
  char str[80];

   count=0;

   if((io=fopen("io.c","r"))==NULL) {
     printf("Can't open file\n");
     exit(1);
    }
   while(!feof(io)) {
               fgets(str,80,io);
               if (feof(io)) break;
```

```
                sscanf(str,"%d%10d%10d%10d\n",
&p2,&i1,&i2,&p3);
                if (p1==p2){
                        if (i1!=0){
                              count=count+1;
                                inout[count]=i1;
    }
                        if (i2!=0){
                                count=count+1;
                                      inout[count]=i2;
                   }
                if (p3!=0) p1=p3;
                if (p3==0) break;
              }
       }
    if (p3==0){
                if (count==1) fprintf(out,"( connect");
                else
                              fprintf(out,"( a%d",count);
                for(j=1;j<=count;j++)
                              fprintf(out," n%d",inout[j]);
                fprintf(out," n%d)\n",nout);
             }
     fclose(io);
}

void nandgate(void)
{
 int count,j,inout[10];
  char str[80];

   count=0;
   if((io=fopen("io.c","r"))==NULL) {
     printf("Can't open file\n");
     exit(1);
    }
   while(!feof(io)) {
                fgets(str,80,io);
                if(feof(io)) break;
                sscanf(str,"%d%10d%10d%10d\n",
&p2,&i1,&i2,&p3);
```

```
                if (p1==p2){
                        if (i1!=0){
                              count=count+1;
                            inout[count]=i1;
    }
                        if (i2!=0){
                               count=count+1;
                               inout[count]=i2;
                      }
                if (p3!=0) p1=p3;
                else break;
              }
        }
     if (p3==0){
                if (count==1) fprintf(out,"( i%d",count);
                else fprintf(out,"( ai%d",count);
                for(j=1;j<=count;j++)
                             fprintf(out," n%d",inout[j]);
                fprintf(out," n%d)\n",nout);
             }
      fclose(io);
}

void orgate()
{
 int count,j,inout[10];
  char str[80];

   count=0;
   if((io=fopen("io.c","r"))==NULL) {
     printf("Can't open file\n");
     exit(1);
    }
   while(!feof(io)) {
                fgets(str,80,io);
                if (feof(io)) break;
                sscanf(str,"%d%10d%10d%10d\n",
&p2,&i1,&i2,&p3);
                if (p1==p2){
                        if (i1!=0){
                       count=count+1;
```

```
                        inout[count]=i1;
    }
                         if (i2!=0){
                               count=count+1;
                               inout[count]=i2;
                        }
                    if (p3!=0) p1=p3;
                    if (p3==0) break;
                }
        }
     if (p3==0){
                if (count==1) fprintf(out,"( connect");
                    else
                       fprintf(out,"( o%d",count);
                    for(j=1;j<=count;j++)
                       fprintf(out," n%d",inout[j]);
                    fprintf(out," n%d)\n",nout);
                 }
      fclose(io);
}

void norgate()
{
 int count,j,inout[10];
  char str[80];

   count=0;
   if((io=fopen("io.c","r"))==NULL) {
     printf("Can't open file\n");
     exit(1);
    }
   while(!feof(io)) {
                    fgets(str,80,io);
                    if (feof(io)) break;
                    sscanf(str,"%d%10d%10d%10d\n",
&p2,&i1,&i2,&p3);
                    if (p1==p2){
                             if (i1!=0){
                            count=count+1;
                                inout[count]=i1;
    }
```

```
                    if (i2!=0){
                          count=count+1;
                          inout[count]=i2;
                      }
                  if (p3!=0) p1=p3;
                  if (p3==0) break;
               }
        }
     if (p3==0){
                  if (count==1) fprintf(out,"( i%d",count);
                  else
                    fprintf(out,"( oi%d",count);
                  for(j=1;j<=count;j++)
                  fprintf(out," n%d",inout[j]);
                  fprintf(out," n%d)\n",nout);
                }
      fclose(io);
}

void xorgate()
{
 int count,j,inout[10];
  char str[80];

   count=0;
   if((io=fopen("io.c","r"))==NULL) {
     printf("Can't open file\n");
     exit(1);
    }
   while(!feof(io)) {
                  fgets(str,80,io);
                  if (feof(io)) break;
                  sscanf(str,"%d%10d%10d%10d\n",
&p2,&i1,&i2,&p3);
                  if (p1==p2){
                          if (i1!=0){
                         count=count+1;
                                inout[count]=i1;
    }
                           if (i2!=0){
                                 count=count+1;
```

```
                                inout[count]=i2;
                        }
                    if (p3!=0) p1=p3;
                    else break;
                }
        }
    if (p3==0){
                    fprintf(out,"( exor");
                    for(j=1;j<=count;j++)
                    fprintf(out," n%d",inout[j]);
                    fprintf(out," n%d)\n",nout);
                }
     fclose(io);
}

void dfcs()
{
 int count,j,inout[10];
  char str[80];

   count=0;
   if((io=fopen("io.c","r"))==NULL) {
     printf("Can't open file\n");
     exit(1);
    }
   while(!feof(io)) {
                    fgets(str,80,io);
                    if (feof(io)) break;
                    sscanf(str,"%d%10d%10d%10d\n",
&p2,&i1,&i2,&p3);
                    if (p1==p2){
                            if (i1!=0){
                          count=count+1;
                                  inout[count]=i1;
    }
                            if (i2!=0){
                                  count=count+1;
                                  inout[count]=i2;
                        }
                    if (p3!=0) p1=p3;
                    if (p3==0) break;
```

```
            }
        }
    if (p3==0){
if(inout[4]!=-1){
                fprintf(out,"( dsff");
                for(j=1;j<=2;j++)
                fprintf(out," n%d",inout[j]);
                fprintf(out," n%d n%d clk2 scantest scanin
)\n",inout[4],nout);
        }
                else    {
                  fprintf(out,"( drff");
                  for(j=1;j<=2;j++)
                  fprintf(out," n%d",inout[j]);
                  fprintf(out," n%d n%d clk2 scantest scanin
)\n",inout[5],nout);
                        }
             }
     fclose(io);
}

void memck()
{
 int count,j,inout[10];
  char str[80];

   count=0;
   if((io=fopen("io.c","r"))==NULL) {
     printf("Can't open file\n");
     exit(1);
    }
   while(!feof(io)) {
             fgets(str,80,io);
             if (feof(io)) break;
             sscanf(str,"%d%10d%10d%10d\n",
&p2,&i1,&i2,&p3);
             if (p1==p2){
                        if (i1!=0){
                  count=count+1;
                          inout[count]=i1;
    }
```

```
                        if (i2!=0){
                           count=count+1;
                           inout[count]=i2;
                     }
                  if (p3!=0) p1=p3;
                  if (p3==0) break;
               }
         }
      if (p3==0){
                fprintf(out,"( deff");
                for(j=1;j<=3;j++)
                fprintf(out," n%d",inout[j]);
                fprintf(out," n%d clk2 scantest
scanin Reset )\n",nout);
               }
 fclose(io);
}

void ioclu()
{
 int j,count,inout[10];
  char str[80];

     count=0;
   if((io=fopen("io.c","r"))==NULL) {
     printf("Can't open file\n");
     exit(1);
    }
   while(!feof(io)) {
                fgets(str,80,io);
                if (feof(io)) break;
                sscanf(str,"%d%10d%10d%10d\n",
&p2,&i1,&i2,&p3);
                if (p1==p2){
                        if (i1!=0){
                     count=count+1;
                           inout[count]=i1;
    }
                        if (i2!=0){
                             count=count+1;
                             inout[count]=i2;
```

```
                }
              if (p3!=0) p1=p3;
              if (p3==0) break;
            }
       }
    if (p3==0){
                        fprintf(out,"( connect");
                        /*for(j=1;j<=3;j++)*/
                         fprintf(out," n%d",inout[1]);
                        fprintf(out," n%d )\n",nout);
            }
 fclose(io);
}

void exin()
{
 int inout,ix,iy;
 char str1[80],iplace[6],exilab[10];
     if((exinp=fopen("exi.c","r"))==NULL){
     printf("Can't open file");
     exit(1);
   }

   while(!feof(exinp)){
     fgets(str1,80,exinp);
     if (feof(exinp)) break;
     sscanf(str1,"%s%s%2d%4d%3d\n",
exilab,iplace,&ix,&iy,&inout);
     if (inout==nout)
     fprintf(out,"( pin n%d %s %d %d
\"pintype=%s\")\n",inout,iplace,ix,iy,exilab);
       }
   fclose(exinp);
}
void exout()
{
 int inp1,ox,oy,onout;
 char str1[80],exolab[10],oplace[7];
     if((exoutp=fopen("exo.c","r"))==NULL){
      printf("Can't open file");
      exit(1);
```

```
   }
   while(!feof(exoutp)){
        fgets(str1,80,exoutp);
        if (feof(exoutp)) break;
        sscanf(str1,"%s%s%2d%4d%4d%3d\n",
exolab,oplace,&ox,&oy,&onout,&inp1);
        if (nout==onout){
          fprintf(out,"( pin n%d %s %d %d
\"pintype=%s\")\n",onout,place,ox,oy,exolab);
           fprintf(out,"( connect n%d n%d
)\n",onout,inp1);
                  }
         }
   fclose(exoutp);
}
void buses()
{
 int inp1,inp2,bnout,bx,by;
 char str1[80],blab[10],bplace[7];
     if((exbp=fopen("exb.c","r"))==NULL){
      printf("Can't open file");
      exit(1);
   }

   while(!feof(exbp)){
        fgets(str1,80,exbp);
        if (feof(exbp)) break;
        sscanf(str1,"%s%s%2d%4d%4d%4d%3d\n",
blab,bplace,&bx,&by,&bnout,&inp1,&inp2);
        if (nout==bnout){
          fprintf(out,"( pin n%d %s %d %d
\"pintype=%s\")\n",nout,bplace,bx,by,blab);
          if (inp2==0) fprintf(out,"(
connect n%d n%d )\n",nout,inp1);
               else
          fprintf(out,"( o2 n%d n%d n%d
)\n",inp1,inp2,nout);
                  }
         }
   fclose(exbp);
}
```

Appendix C

AHPL Program Listing for three node Queue and Stack

Program Lisiting of Queue

```
      MODULE   : QNODE1.
      MEMORY   : IL1; IR1; DL1;DR1;S1{8}; B1{8};
                 CD1; CI1;LS1; LB1.
      EXBUSES  : LSL1;LSR1.
      INPUTS   : TOPD2{8};TOPD3{8}.
      OUTPUTS  : TOPD1{8};OLS1;INS2;DEL2;INS3;DEL3;TOPI2{8};
                 TOPI3{8}.
      BUSES    : CC11;CC21;CC31;T11;T21;T31;T41;FALSE1.
      EXINPUTS : TOPI1{8};INS1;DEL1;RESET;CLK.

    BODY SEQUENCE :CLK.

 1    CI1 * (T11)      <= \0\;
      CD1 * (T11)      <= \0\;
      CI1 * (T21)      <= ~(CI1);
      LS1 * (T11)      <= \1\;
      LB1 * (T21)      <= \1\;
      IL1 * (INS1)     <= (CI1 ! \0\) * (LS1, ~LS1);
      IR1 * (INS1)     <= (~CI1 ! \0\) * (LS1, ~LS1);

      LS1 * (CC11)     <= \0\;
      CD1 * (CC21)     <= ~(CD1);
      DR1 * (T31)      <= \1\;
      DL1 * (T41)      <= \1\;
      CD1 * (CC31)     <= ~(CD1);
      B1  * (T21)      <=  TOPI1;
```

```
   S1*(FALSE1)<= (TOPI1!B1!TOPD2!TOPD3)*(T11,CC21,T41,T31);
   TOPD1        =  S1*DEL1 ;
   => (1).
ENDSEQUENCE
CONTROLRESET (1);
TOPI2 = B1 * IL1;
TOPI3 = B1 * IR1;
FALSE1= (T11+CC21+T41+T31);
INS2=IL1;
DEL2=DL1;
OLS1=LS1;
INS3=IR1;
DEL3=DR1;
T11 = INS1 & ~LS1;
T21 = INS1 & LS1;
T31 = CC31 & ~(CD1);
T41 = CC31 & CD1;
CC11 = ~(LSR1 + LSL1 + LB1) & DEL1;
CC21 = ~(LSR1 + LSL1 + ~LB1) & DEL1;
CC31 = ~(CC11) & ~(CC21) & DEL1.
END.

    MODULE   : QNODE2.
    MEMORY   : IL2; IR2; DL2;DR2;S2{8}; B2{8};
               CD2; CI2; LS2; LB2.
    OUTPUTS  : TOPD2{8};OLS2;INS4;DEL4;INS6;DEL6;TOPI4{8};
               TOPI6{8}.
    BUSES    : CC12;CC22;CC32;T12;T22;T32;T42;FALSE2.
    INPUTS   : TOPI2{8};INS2;DEL2.
    EXBUSES  : TOPD4{8};TOPD6{8};LSL2;LSR2;LSL1.
    EXINPUTS : RESET;CLK.

  BODY SEQUENCE :CLK.

1   CI2 * (T12)      <= \0\;
    CD2 * (T12)      <= \0\;
    CI2 * (T22)      <= ~(CI2);
    LS2 * (T12)      <= \1\;
    LB2 * (T22)      <= \1\;
    IL2 * (INS2)     <= (CI2 ! \0\) * (LS2, ~LS2);
    IR2 * (INS2)     <= (~CI2 ! \0\) * (LS2, ~LS2);
```

```
   LS2 * (CC12)      <= \0\;
   CD2 * (CC22)      <= ~(CD2);
   DR2 * (T32)       <= \1\;
   DL2 * (T42)       <= \1\;
   CD2 * (CC32)      <= ~(CD2);
   B2  * (T22)       <=  TOPI2;
   S2*(FALSE2)<= (TOPI2!B2!TOPD4!TOPD6)*(T12,CC22,T42,T32);
   TOPD2              =  S2 ;
   => (1).
ENDSEQUENCE
CONTROLRESET (1);
TOPI4 = B2 * IL2;
TOPI6 = B2 * IR2;
FALSE2= (T12+CC22+T32+T42);
LSL1=LS2;
INS4=IL2;
DEL4=DL2;
OLS2=LS2;
INS6=IR2;
DEL6=DR2;
T12 = INS2 & ~LS2;
T22 = INS2 & LS2;
T32 = CC32 & ~(CD2);
T42 = CC32 & CD2;
CC12 = ~(LSR2 + LSL2 + LB2) & DEL2;
CC22 = ~(LSR2 + LSL2 + ~LB2) & DEL2;
CC32 = ~(CC12) & ~(CC22) & DEL2.
END.

    MODULE    : QNODE3.
    MEMORY    : IL3; IR3; DL3;DR3;S3{8}; B3{8};
                CD3; CI3; LS3; LB3.
    OUTPUTS   : TOPD3{8};OLS3;INS5;DEL5;INS7;DEL7;TOPI5{8};
                TOPI7{8}.
    BUSES     : CC13;CC23;CC33;T13;T23;T33;T43;FALSE3.
    INPUTS    : TOPI3{8};INS3;DEL3.
    EXBUSES   : TOPD5{8};TOPD7{8};LSL3;LSR3;LSR1.
    EXINPUTS  : RESET;CLK.

  BODY SEQUENCE :CLK.
```

```
1   CI3 * (T13)       <= \0\;
    CD3 * (T13)       <= \0\;
    CI3 * (T23)       <= ~(CI3);
    LS3 * (T13)       <= \1\;
    LB3 * (T23)       <= \1\;
    IL3 * (INS3)      <= (CI3 ! \0\) * (LS3, ~LS3);
    IR3 * (INS3)      <= (~CI3 ! \0\) * (LS3, ~LS3);

    LS3 * (CC13)      <= \0\;
    CD3 * (CC23)      <= ~(CD3);
    DR3 * (T33)       <= \1\;
    DL3 * (T43)       <= \1\;
    CD3 * (CC33)      <= ~(CD3);
    B3  * (T23)       <=  TOPI3;
    S3*(FALSE3)<= (TOPI3!B3!TOPD5!TOPD7)*(T13,CC23,T43,T33);
    TOPD3              =  S3 ;
    => (1).
 ENDSEQUENCE
 CONTROLRESET (1);

 TOPI5 = B3 * IL3;
 TOPI7 = B3 * IR3;
 FALSE3= (T13+CC23+T33+T43);
 LSR1=LS3;
 INS5=IL3;
 DEL5=DL3;
 OLS3=LS3;
 INS7=IR3;
 DEL7=DR3;
 T13 = INS3 & ~LS3;
 T23 = INS3 & LS3;
 T33 = CC33 & CD3;
 T43 = CC33 & ~CD3;
 CC13 = ~(LSR3 + LSL3 + LB3) & DEL3;
 CC23 = ~(LSR3 + LSL3 + ~LB3) & DEL3;
 CC33 = ~(CC13) & ~(CC23) & DEL3.
END.
```

Program Listing of Stack

```
      MODULE   : SNODE1.
      MEMORY   : IL1; IR1; DL1;DR1;S1{8}; B1{8};
                 C1;LS1; LB1.
      EXBUSES  : LSL1;LSR1.
      EXBUSES  : TOPD2{8};TOPD3{8}.
      EXBUSES  : TOPD1{8};OLS1;INS2;DEL2;INS3;DEL3;TOPI2{8};
                 TOPI3{8}.
      BUSES    : CC11;CC21;CC31;T11;T21;T31;T41;FALSE1.
      EXINPUTS : TOPI1{8};INS1;DEL1;RESET;CLK;START.

    BODY SEQUENCE :CLK.

 1    C1 * (T11)      <= \0\;
      C1 * (T21)      <= ~(C1);
      LS1 * (T11)      <= \1\;
      LB1 * (T21)      <= \1\;
      IL1 * (INS1)     <= (C1 ! \0\) * (LS1, ~LS1);
      IR1 * (INS1)     <= (~C1 ! \0\) * (LS1, ~LS1);

      LS1 * (CC11)     <= \0\;
      C1 * (CC21)     <= ~(C1);
      DR1 * (T31)      <= \1\;
      DL1 * (T41)      <= \1\;
      C1 * (CC31)     <= ~(C1);
      B1  * (T21)      <=  S1;
      S1*(FALSE1)<= (TOPI1!B1!TOPD2!TOPD3)*(INS1,CC21,T41,T31);
      TOPD1         =  S1*DEL1 ;
      =>(~START)/(1).
   ENDSEQUENCE
   CONTROLRESET (1);
   TOPI2 = B1 * IL1;
   TOPI3 = B1 * IR1;
   FALSE1= (INS1+CC21+T41+T31);
   INS2=IL1;
   DEL2=DL1;
   OLS1=LS1;
   INS3=IR1;
   DEL3=DR1;
   T11 = INS1 & ~LS1;
```

```
 T21 = INS1 & LS1;
 T31 = CC31 & C1;
 T41 = CC31 & ~C1;
 CC11 = ~(LSR1 + LSL1 + LB1) & DEL1;
 CC21 = ~(LSR1 +LSL1+ ~LB1) &DEL1;
 CC31 = ~(CC11) & ~(CC21) & DEL1.
END.

     MODULE    : SNODE2.
     MEMORY    : IL2; IR2; DL2;DR2;S2{8}; B2{8};
                 C2; LS2; LB2.
     EXBUSES   : TOPD2{8};OLS2;INS4;DEL4;INS6;DEL6;TOPI4{8};
                 TOPI6{8}.
     BUSES     : CC12;CC22;CC32;T12;T22;T32;T42;FALSE2.
     EXBUSES   : TOPI2{8};INS2;DEL2.
     EXBUSES   : TOPD4{8};TOPD6{8};LSL2;LSR2;LSL1.
     EXINPUTS  : RESET;CLK;START.

   BODY SEQUENCE :CLK.

1    C2 * (T12)       <= \0\;
     C2 * (T22)       <= ~(C2);
     LS2 * (T12)       <= \1\;
     LB2 * (T22)       <= \1\;
     IL2 * (INS2)      <= (C2 ! \0\) * (LS2, ~LS2);
     IR2 * (INS2)      <= (~C2 ! \0\) * (LS2, ~LS2);

     LS2 * (CC12)      <= \0\;
     C2 * (CC22)      <= ~(C2);
     DR2 * (T32)       <= \1\;
     DL2 * (T42)       <= \1\;
     C2 * (CC32)      <= ~(C2);
     B2  * (T22)       <=  S2;
     S2*(FALSE2)<= (TOPI2!B2!TOPD4!TOPD6)*(INS2,CC22,T42,T32);
     TOPD2              =  S2 ;
     =>(~START)/(1).
 ENDSEQUENCE
 CONTROLRESET (1);
 TOPI4 = B2 * IL2;
 TOPI6 = B2 * IR2;
 FALSE2= (INS2+CC22+T32+T42);
```

```
 LSL1=LS2;
 INS4=IL2;
 DEL4=DL2;
 OLS2=LS2;
 INS6=IR2;
 DEL6=DR2;
 T12 = INS2 & ~LS2;
 T22 = INS2 & LS2;
 T32 = CC32 & C2;
 T42 = CC32 & ~(C2);
 CC12 = ~(LSR2 + LSL2 + LB2) & DEL2;
 CC22 = ~(LSR2+LSL2+~LB2) & DEL2;
 CC32 = ~(CC12) & ~(CC22) & DEL2.
END.
     MODULE   : SNODE3.
     MEMORY   : IL3; IR3; DL3;DR3;S3{8}; B3{8};
                C3; LS3; LB3.
     EXBUSES  : TOPD3{8};OLS3;INS5;DEL5;INS7;DEL7;TOPI5{8};
                TOPI7{8}.
     BUSES    : CC13;CC23;CC33;T13;T23;T33;T43;FALSE3.
     EXBUSES  : TOPI3{8};INS3;DEL3.
     EXBUSES  : TOPD5{8};TOPD7{8};LSL3;LSR3;LSR1.
     EXINPUTS : RESET;CLK;START.

   BODY SEQUENCE :CLK.

1   C3 * (T13)      <= \0\;
    C3 * (T23)      <= ~(C3);
    LS3 * (T13)      <= \1\;
    LB3 * (T23)      <= \1\;
    IL3 * (INS3)     <= (C3 ! \0\) * (LS3, ~LS3);
    IR3 * (INS3)     <= (~C3 ! \0\) * (LS3, ~LS3);

    LS3 * (CC13)     <= \0\;
    C3 * (CC23)     <= ~(C3);
    DR3 * (T33)      <= \1\;
    DL3 * (T43)      <= \1\;
    C3 * (CC33)     <= ~(C3);
    B3  * (T23)      <=  S3;
    S3*(FALSE3)<= (TOPI3!B3!TOPD5!TOPD7)*(INS3,CC23,T43,T33);
    TOPD3             =  S3 ;
```

```
   =>(~START)/(1).
 ENDSEQUENCE
 CONTROLRESET (1);
 TOPI5 = B3 * IL3;
 TOPI7 = B3 * IR3;
 FALSE3= (INS3+CC23+T33+T43);
 LSR1=LS3;
 INS5=IL3;
 DEL5=DL3;
 OLS3=LS3;
 INS7=IR3;
 DEL7=DR3;
 T13 = INS3 & ~LS3;
 T23 = INS3 & LS3;
 T33 = CC33 & C3;
 T43 = CC33 & ~C3;
 CC13 = ~(LSR3 + LSL3 + LB3) & DEL3;
 CC23 = ~(LSR3+LSL3+~LB3)&DEL3;
 CC33 = ~(CC13) & ~(CC23) & DEL3.
END.
```

Bibliography

[1] M. Atallah and S. Kosaraju. A generalized dictionary for VLSI. *IEEE Transactions on Computers*, C-34:151–155, 1985.

[2] J. Bentley and H. T. Kung. A tree machine for searching problems. *Proc. Int. Conf. Parallel Processing*, pages 257–266, Aug 1979.

[3] Sally A. Browning. Computations on tree of processors. *Caltech conference on VLSI*, pages 453–478, Jan 1979.

[4] Jik H. Chang et al. Systolic tree implementation of data structures. *IEEE Transactions on Computers*, 37(6):727–735, 1988.

[5] K. Culik and O. Ibarra. Systolic tree implementation of data structures. *Comput. Mathematics*, C-20:187–204, 1986.

[6] Northwest Laboratories for Integrated Systems. VLSI design tools manual release 3.1. *University of Washington*, 1987.

[7] D. Gordon. Efficient embedding of binary trees in VLSI. *IEEE Transactions on Computers*, C-36:1009–1018, 1987.

[8] D. Gordon et al. Embedding tree structures in VLSI hexagonal arrays. *IEEE Transactions on Computers*, C-33:104–107, 1984.

[9] L. Guibas and F. LANG. Systolic stacks, queues, and counters. *Proc. Conf. Advance Res. VLSI, MIT Cambridge*, 1982.

[10] Frederick J. Hill. AHPL: Then and Now. *IEEE Design and Test of Computers*, pages 73–75, June 1992.

[11] E. Horowitz and A. Zorat. The binary tree as an interconnection network: Application to multiprocessor systems and VLSI. *IEEE Transactions on Computers*, C-30:247–253, April 1981.

[12] K. Hwang and F. A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill Book Company, 1984.

[13] N. Kanopolous and J. J. Hellenbeck. A first-in, first-out memory for signal processing applications. *IEEE Transactions on circuits and systems*, CAS-33:556–558, May 1986.

[14] R. L. Kruse. *Data Structures.* Prentice-Hall India Private Limited, 1987.

[15] M. A. Kulaib et al. Design of a programmable length FIFO memory and its controller. *International Journal of Electronics*, 65:923–932, 1988.

[16] H. T. Kung. Why Systolic Architectures. *IEEE Computer*, pages 37–46, Jan 1982.

[17] C.E. Leiserson. Systolic priority queues. *Tech. Rep. CMU-CS-79-115,Dept. of Computer Science, Carnegie Mellon University*, 1979.

[18] M. J. Lorenzetti et al. Channel routing for compaction. *Proceedings of the International Workshop on Placement and Routing*, May 1988.

[19] M. Masud and Sadiq M. Sait. Universal AHPL - a language for VLSI design automation. *IEEE Circuits and Devices Magazine*, September 1986.

[20] Robert N. Mayo et al. Decwrl/livermore MAGIC release. *Technical report, DECWRL, Digital Western Research Laboratory*, September 1990.

[21] Open System Silicon Implementation Software. Reference manual. *MCNC*, 1992.

[22] T. Ottman et al. Dictionary machine for VLSI. *IEEE Transactions on Computers*, C-32:892–897, 1982.

[23] J. E. Rose. Greedy algorithms for writing in VLSI. *Master's thesis, Department of Computer Science*, 1985.

[24] Sadiq M. Sait. Integrated UAHPL-DA system with VLSI design tool to support VLSI DA courses. *IEEE Transaction on Education*, September 1992.

[25] Sadiq M. Sait and Mohammed Shahid K. Tanvir. VLSI layout generation of a programmable CRC chip. *IEEE Transactions on Consumer Electronics*, 33:911–916, November 1993.

[26] A. Somani and V. Agarwal. An efficient unsorted VLSI dictionary machine. *IEEE Transactions on Computers*, C-34:841–852, 1985.

[27] P. R. Suaris and G. Kedem. A new approach to standard cell layout. *International Conference on Computer Aided Design*, pages 474–477, November 1987.

[28] Kung H. T and Charles E. Leiserson. *Systolic Arrays for VLSI.* Mead and Conway, 1978.

[29] C. D. Thompson and B. W. Wei. Systolic implementations of a move-to-front text compressor. *ACM 0-89791-323-X/89/0006/0283*, 1989.

[30] D. J. Ullman. Computational aspects of VLSI. *Computer Science Press*, 1987.

[31] Hee Yong Youn and Adit D. Singh. On implementing large binary tree architectures in VLSI and WSI. *IEEE Transactions on Computers*, pages 526–537, 1989.