THE FAULT-TOLERANT SINGLE-CHIP VECTOR PROCESSOR: ARCHITECTURE AND PERFORMANCE ANALYSIS USING LIVERMORE LOOP BENCHMARKS

BY

SUNDARARAJAN GANESH

Bachelor of Engineering

Annamalai University

Chidambaram, India

1988

Submitted to the faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirement for
the Degree of
MASTER OF SCIENCE
May, 1992

Dinosaur
1978
[illegible]

THE FAULT-TOLERANT SINGLE-CHIP VECTOR
PROCESSOR: ARCHITECTURE AND
PERFORMANCE ANALYSIS USING
LIVERMORE LOOP BENCHMARKS

Thesis Approved:

Jong J. Lee
Thesis Adviser

Richard L. Cummins

Chas G. Hutchens

Thomas C. Collins
Dean of the Graduate College

ACKNOWLEDGEMENTS

I wish to express my sincere gratitude to all the people who have assisted me in this work and during my stay at Oklahoma State University. I sincerely thank Dr. J. J. Lee for entrusting to me this topic which is one of his research areas, and for serving as my graduate student advisor. I am also extremely grateful to Dr. Lee for providing me with financial support during my graduate study.

My sincere thanks also goes to Dr. Richard L. Cummins and Dr. Chris G. Hutchens for serving in my graduate committee. I am thankful to Dr. Cummins for giving technical help in thesis-writing. I owe my gratitude to Dr. Hutchens for offering good suggestions in improving the quality of my thesis. His pep-talks were always a morale-booster, and I thank him for the talks. I also thank Leslie Fife and Philis for their patient help in improving my thesis quality.

I am thankful for the unforgettable friendship and company of (breakdown) Paneer, Krishnan, Shankar, Gopal (Mr. El Paso), Balachandar, Shekar, Vijay (Mr. B), Ravi (Blue), Venkat and Saravanan (rat) in Stillwater. I am extremely grateful to my childhood friends Sivakumar, Jagdish, Sriram, Sunil and Babu, who haven't forgotten me even after my long period of absence from India.

I am greatly indebted to my parents and my brother Jayesh, who make me feel that I have the best family in the world. My special thanks goes to my uncle and aunt for their constant support and encouragement. I also thank Manu, Priya, Shubha, Shruthi and Latha for their ever-refreshing letters which kept me going in Stillwater. I also thank Ramesh and Uma for extending their hospitality in U. S., and Rajendran for his telephone calls.

TABLE OF CONTENTS

Chapter		Page
I.	INTRODUCTION	1
II.	SINGLE-CHIP VECTOR PROCESSOR	9
	Interconnection Network	9
	Pipeline Net	14
	Basic Structure of FTVP	19
	Structure of the Vector Register	21
	Structure of an Arithmetic Unit	24
	Chaining Capability	26
	Basic Instructions	31
III.	FAULT-TOLERANCE IN THE FTVP	37
	Types of Fault and Fault Vectors	38
	Translation Procedure	41
	Example	47
	Fault Free Condition	47
	Pipeline Fault Condition	51
	Switch Fault Condition	51
	Pipeline and Switch Fault	54
	Link and Switch Fault	54
IV.	EVALUATION	57
	Speedup and Throughput	57
	Proposed Architecture	65
V.	CONCLUSIONS AND FUTURE WORK	70
REFERENCES		71
APPENDIX A - LIVERMORE LOOPS		74
APPENDIX B - PROGRAM GRAPH OF THE LIVERMORE LOOPS		77
APPENDIX C - WAFER SCALE INTEGRATION		82

LIST OF TABLES

Table — Page

I. The Trend in Supercomputers and High-End Main Frame Systems 2

II. A Taxonomy of Fault Tolerance Techniques in Commercial Computing Systems 7

III. Basic Instructions for the FTVP 32

IV. Evaluation Time of the Livermore Loops 60

V. Livermore Loop Parameters 66

VI. Comparison of the Packages of Figure of Merits 85

LIST OF FIGURES

Figure		Page
1.	Architecture of a Register-Register Type Vector Processor	4
2.	8x8 Cube Switching Network	10
3.	8x8 Conventional Crossbar Network	10
4.	Program Graph	17
5.	Pipeline Net Implementation	17
6.	Crossbar Network Implementation	18
7.	Hardware Model of the FTVP	20
8.	Structure of a Vector Register	22
9.	Structure of an Arithmetic Unit	24
10.	FTVP Implementation of the Example Loop	27
11.	Chaining Operation for Livermore Loop 9	29
12.	Instructions for the Example Loop	35
13.	WSI Model for the Proposed Study	39
14.	1-to-1 Mapping Procedure	46
15.	Architecture of an 8 Pipelined Structure With Switch Indexes	48
16.	Routing for Livermore Loop 1 with a Fault Free Condition	50
17.	Routing for Livermore Loop 1 in case of Faulty Pipelines	52
18.	Routing for Livermore Loop 1 in case of Faulty Switches	53
19.	Routing for Livermore Loop 1 in case of Pipeline and Switch Faults	55
20.	Routing for Livermore Loop 1 in case of Link and Switch Fault	56
21.	Loop Length vs. Throughput	62

Figure Page

22. Speedup vs. Loop Length 63

23. Speedup vs. Loop Length for Livermore Loops with
No Recurrence Relationship 64

24. Speedup vs.. Loop Length for Livermore Loops with
Recurrence Relationship 65

25. Program Graph for Livermore Loop 1 66

26. Hardware of the Recommended Structure 67

CHAPTER I

INTRODUCTION

The development of processors with pipelined arithmetic units has offered an economical way of increasing the speed of vector supercomputers since the 1960s when the first generation vector supercomputers arrived [1]. The hardware, called the *pipeline*, is divided into a series of substages called the *pipeline stages*. Each substage of the pipeline will execute a portion of the overall task (process) performed by the pipeline. The input for the overall task performed by the pipeline is streamed into the first pipeline stage, and the output of the overall task emerges from the last stage of the pipeline. Each intermediate stage of the pipeline will receive its input datum from the previous pipeline stage, compute and send the results directly to the subsequent pipeline stage. While the results are being sent to the subsequent pipeline stage, a new input datum from the previous stage may be received. As soon as a pipeline stage receives a new input datum, it starts computing the output of the portion of task assigned to it, independent of the other stages, resulting in an overlapped execution similar to the assembly line in an industry [1]. This overlapped execution makes it possible for the input of the overall task to be continuously streamed to the first stage of the pipeline without waiting for the output to emerge from the last stage of the pipeline. Thus, the throughput of a processor that has pipelines is increased. The architecture of processors that use the pipelining technique has evolved from that of a single pipelined architecture similar to the TI ASC [2] to a multipipelined architecture similar to the NEC SX-2 [3 and 4]. Table I [5] shows the architecture configuration of some of the supercomputers

TABLE I

THE TREND IN SUPERCOMPUTERS AND HIGH-END MAIN FRAME SYSTEMS

System Model	Architecture configuration	Max no of processors	Processor type	Max memory capacity	Peak performance
Cray X-MP/4	MP with SM and direct interconnect	4 processors	Custom ECL	16MW in CM 128MW in SSD	840 Mflops
Cray 2	MP with SM and direct connect	4 processors 1 IOP	Custom ECL	256MW	2 Gflops
Cray 3	MP with SM	16 processors	GaAs/ECL	2GW	16 Gflops
Cyber 205	UP with scalar processor and 4 vector pipelines	1 processor	Custom CMOS	4MW	400 Gflops
ETA-10	MP with SM	8 processors 18 IOPs	Custom	256mW	10 Gflops
Fujitsu VP-200	UP with multiple functional pipes	1 processor	Custom ECL	32MW	533 Mflops
NEC SX-2	UP with 16 functional pipes	1 processor	Custom	32MW	1 3 Gflops
Hitachi S-810	UP with multiple pipelines	1 processor	Custom	32MW	840 Mflops
HEP-1	MP with SM and switch network	16 processors	Custom	256MW	160 Mflops
IBM 3090/400/VF	MP with SM and direct connect	4 processors	Custom TCM	2 GB CM 16 TB EM	480 Mflops
Univac 1194/ISP x 2	MP with SM and direct connect	4 Processors, 4 IOPs, 2 ISPs	Custom	16MW	67Mflops
CDC Cyberplus	MC with DM and ring connect	64 processors	Custom	512 KW per processor	65 Mflops and 620MIPS per processor
Connection machine	SIMD with DM hypercube embedded in a global mesh	64 K processing elements	VLSI/CMOS gate array	32 MBytes	>1000 Mflops 250 Mflops
BBN Butterfly	MP with SM via butterfly switch network	256 processors	M68020 custom coprocessor	128 MW	256 Mips
Loral MPP	SIMD 128x128 mesh with DM	16K processing elements	CMOS/SOS 8 processing elements per chip	128 MB	470 Mflops
IBM GF 11	SIMD with reconfigurable Benes network	576 processing elements	Custom floating point processor	2 MB per processor 1 1 GB total	20 Mflops per processor 11 Gflops
IBM RP3	MP with SM/DM and fast network	512 processors	32-bit RISC	128 MW	800 Mflops 1300 Mips
Cedar	Hierarchical MP with SM	256 processors	Alliant/FX clusters	256 MW	3.2 Gflops

MP - Multiprocessor, SM - Shared memory, CM - Central memory, SSD - Solid state device, IOP - I/O processor, MM - Memory - Memory, UP - Uniprocessor, DM - Distributed memory, TCM - Thermal Conduction Module, MC - Multicomputer, EM - Extended Memory, ISP - Integrated Scientific Processors

and mainframe systems available today. The CRAY-1 [1, 3 and 6] has twelve pipelines, with each pipeline executing a different function. The Cyber 205 [1] and the Fujitsu VP 200 [1 and 6] are also multipipelined. The NEC SX-2 processor has four sets of pipelines. Each pipeline set consists of an adder unit, a multiply/divide unit, a logical unit and a shift unit [4]. Each of these machines has a separate scalar processing unit and a separate vector processing unit. The Titan design [7 and 8] combines a vector floating-point unit and a scalar floating-point unit into a unified structure. The designers of the CDC/NASF [1] have proposed a fault-tolerant architecture which has an extra pipeline to be used if a fault occurs in another pipeline.

The vector processor in the computer systems shown in Table I employs the pipelining technique to increase the speed of the processor, which can be further increased by dynamically linking the pipelines present in the processor; this is termed as the *pipeline chain*. Pipeline chaining is a linking process that occurs when the result obtained from one pipeline unit is directly fed to another [1]. The Cray computer calls the dynamic link *chaining* [1] while the Cyber 205 [1] terms the dynamic link *short-stopping*. In recent RISC processors, such as the Intel i860, a primitive form of pipeline chaining is achieved. The i860 can link a multiplier pipeline and an adder pipeline in a pipeline chain in its dual-mode [9].

The architecture of a vector processor in computer systems like the Cray and the NEC SX-2 that are available commercially is classified as a *register-register* architecture. A typical register-register architecture of a vector processor is defined as the pipelines in the processor connected to the vector registers by an interconnection network. Such an architecture is shown in Figure 1. The interconnection network may be a multistage switching network, as in the case of the HEP-1 (Table I), or a crossbar network [10 and 11], as in the case of the Burrough Scientific Processor (BSP) [1].

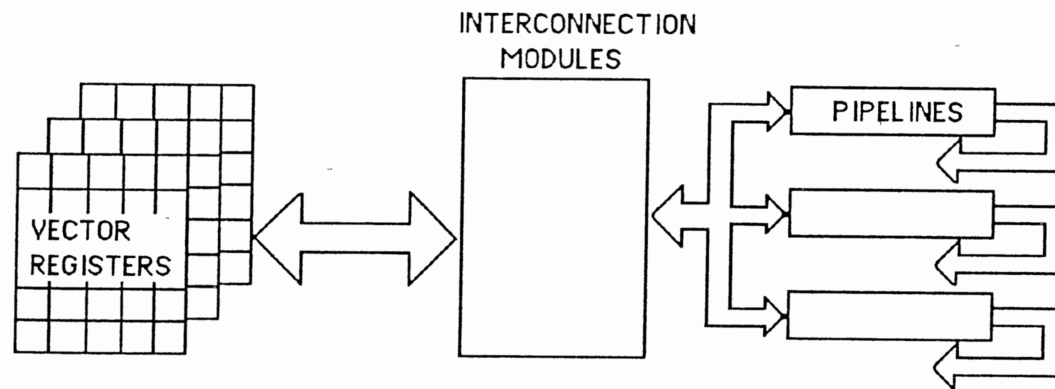


Figure 1. Architecture of a Register-Register Type Vector Processor

As seen in Table I, most of the supercomputer systems were fabricated with the prevalent transistor and ECL technologies. These machines tend to be large and costly. Advances in the area of Very Large Scale Integrated (VLSI) circuit fabrication technology today have made the fabrication of such large systems as single-chip processors possible [12]; single-chip fabrication of these systems will lead to a reduction in cost and area occupied by these systems. Wafer Scale Integration (WSI) is a single-chip technique to fabricate such complex systems [12, 13 and 14].

In 1966, TI fabricated the first Large Scale Integrated (LSI) circuit by fabricating much smaller-sized components on an intact substrate, and then wiring the components that are functional, directly with each other [12]. This process, WSI, can be regarded as a special form of packaging in which the extra wiring normally used to interconnect the packages is fabricated on the surface of a wafer substrate containing the components and mounted inside a single package [12]. Internal wiring significantly reduces problems like the wiring capacitances present in conventional circuits or ceramic carriers [12]; wiring capacitances present will reduce the speed of the fabricated system. Further, in WSI, interconnection densities are increased since the wire dimensions used are smaller when compared to conventional circuits or ceramic carriers [12]. Further, closeness of the components fabricated using WSI leads to shorter interconnection wiring, which in turn enhances the speed of the fabricated circuit, and decreases the power requirement for the I/O drivers of the fabricated circuit [12].

But the major design challenge in WSI is the presence of faulty modules in the fabricated circuit [12]. With the increased application of computer systems in important activities like telecommunication, banking, that control everyday life, it is absolutely essential to make the computer systems reliable and cheap [15]. The existing technique of replacing circuit boards would be impractical if faulty modules are present in the computer systems fabricated using WSI, because we cannot assure

a design which will be one-hundred percent fault-free. Therefore, we have to design the system fabricated using WSI in such a way that we will be able to achieve a functional design even though there are faulty modules in the system; a system designed in this manner is termed as a *fault-tolerant system*. Since we cannot assure one-hundred percent functional cells or modules in the computer systems fabricated using WSI, we need a technique that will identify the faulty modules in the fabricated system. After this identification, the technique should be able to construct a fault-free system using the fault-free modules or cells in the wafer [14].

Faults in a system fabricated using WSI can be classified into two broad types: static faults and dynamic faults [17]. The static faults are permanent ones, such as broken bonds or cracks in the wafer which lead to the loss of a submodule or part of a circuit in the wafer [17]. The dynamic faults are temporary ones; for example, a shift in the threshold voltage [17]. In this report we will deal only with the static faults.

The usual method used to compensate for a faulty module in a system fabricated using WSI is to add a spare module to the system during the fabrication process itself [17]. After fabrication, the faulty modules present in the system are detected, and a suitable reconfiguration algorithm developed prior to the fabrication, is applied to substitute the spare module in place of the faulty module [17]. The designers of the WSI memory chip have used this technique, termed the *redundancy technique*, to achieve fault-tolerance [13]. Designers of commercial systems like the VAX 8600 [18] and the IBM 3090 [19] have also achieved fault-tolerance by using the techniques shown in Table II [15], which also shows the techniques applied for achieving fault-tolerance in some of the other computer systems available commercially.

The first objective of this thesis is to develop the architecture of a register-register type single-chip vector processor. As our second objective, the vector

TABLE II

A TAXONOMY OF FAULT TOLERANCE TECHNIQUES
IN COMMERCIAL COMPUTING SYSTEMS

Structure	Detection	Recovery	Sources of failure tolerated	Techniques
Uniprocessor				
VAX 8600	Hardware	Software	Hardware	Hardware error detection
IBM 3090	Hardware	Hardware / Software	Hardware	Hardware error detection, retry, workaround
Multicomputer				
Tandem	Hardware / Software	Software	Hardware, design, environment	Checkpointing, "I'm alive" messages
VAX 3000	Hardware	Hardware	Hardware, environment	Duplication and matching
Multiprocessor				
Teradata	Hardware	Software	Hardware, environment	Duplication
Sequoia	Hardware	Software	Hardware, environment	Duplication and matching

processor designed must have the capability to dynamically link the pipelines in the processor. WSI is the technique to be used in the proposed fabrication of the single-chip vector processor. Therefore, due to the presence of faulty modules in the fabricated single-chip processor, fault-tolerance has to be achieved in the intended processor design, and it is our third objective.

To achieve our first two objectives, we propose a three-level vector processor structure which is referred to as *Fault-Tolerant Vector Processor* (FTVP). To meet our third objective, a simple translation procedure is provided to achieve fault-tolerance within the FTVP. The study of FTVP originates from a two-level pipeline structure, called the *pipeline net*, proposed in [20]. The proposed pipeline net, which will be discussed in Chapter II, is capable of dynamically linking pipelines with the help of interconnection networks in the structure.

We have used the crossbar network as an interconnection network in the FTVP to provide full interconnection capability for fast data access. The dynamic pipeline-linking capability, *chaining*, of the FTVP is demonstrated by using the Livermore loop benchmark programs [21] listed in Appendix A. Speedup and throughput analysis for the FTVP are done using the Livermore loops. The analysis follows the steps taken by [20]. Based on the analysis presented here, an FTVP hardware architecture is recommended for fabrication using the WSI.

CHAPTER II

SINGLE-CHIP VECTOR PROCESSOR

Interconnection Network

As seen from Figure 1, the interconnection network routes the vector data from the vector registers to the pipelines in the vector processor and vice-versa; therefore, it is an important aspect of the processor. The two major network techniques that exist today are the multistage switching network technique and the crossbar network technique. A three-stage cube switching network [1] and a two-sided normal crossbar network are shown in Figure 2 and Figure 3, respectively.

The crossbar network provides full interconnection capability between any input-output terminal pair. It must be noted here that we will be dealing with only input-output combinations where no two input-output terminal pairs have the same output; that is, each input terminal is associated with an unique output terminal. Due to the full interconnection capability, the crossbar network can connect all possible multiple input-output combination without any conflict, that is, "blocking" [1]. Hence, the crossbar network is termed a *non-blocking network*; it also provides a fast data transfer rate due to its non-blocking nature [11]. But, the major concern in the development of the crossbar network is the cost of an NxN crossbar network, which is proportional to $O(N^2)$, where N is the number of inputs or outputs in the network and N^2 is the number of switches in the network; this cost growth rate which is proportional to $O(N^2)$ is prohibitively high for large N. Therefore, the $O(N^2)$ cost growth rate has proved to be the major obstacle in the crossbar network fabrication using LSI or VLSI technology [11].

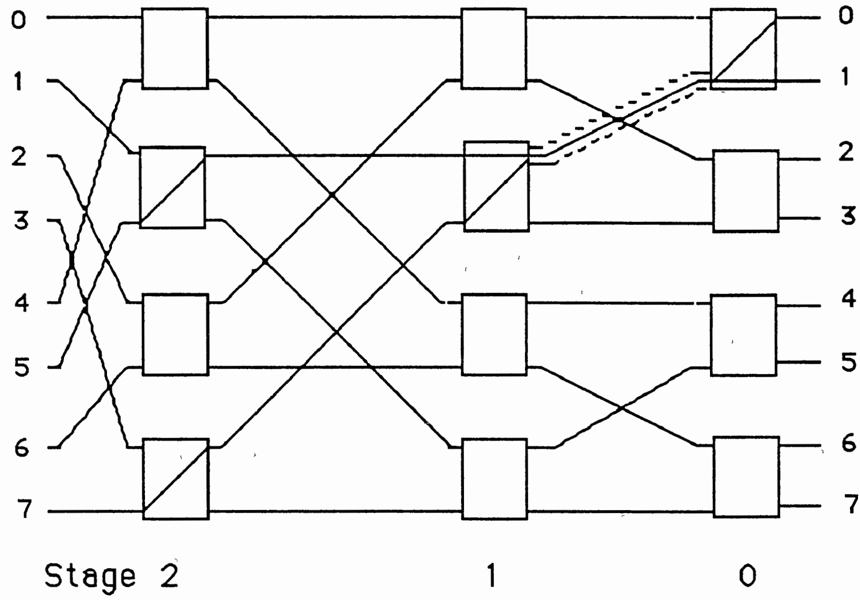


Figure 2. 8x8 Cube Switching Network

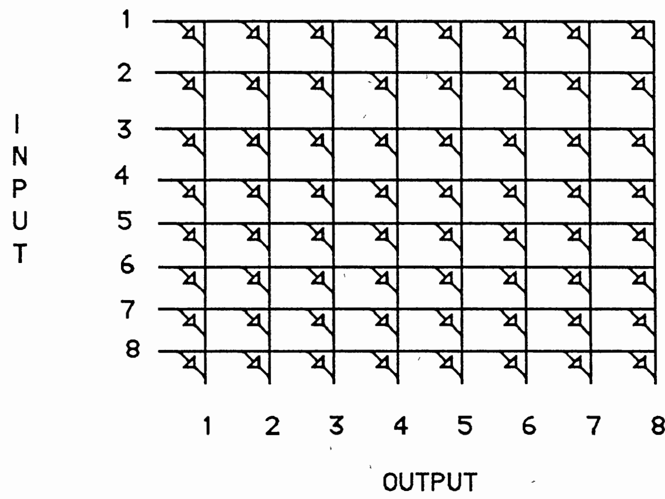


Figure 3. 8x8 Conventional Crossbar Network

On the other hand, an NxN multistage switching network has a cost growth rate which is proportional to $O(N\log_2 N)$ [1], where N is the number of inputs or outputs and $N\log_2 N$ is the number of switches in the network. The $O(N\log_2 N)$ cost growth rate is less than the $O(N^2)$ cost growth rate. For example, let us consider a case with N = 32. The cost growth rate for the crossbar network is proportional to O(1024), as the number of switches in the network is 1024; in case of a multistage network the cost growth rate is proportional to O(160), as the number of switches in the network is 160. However, the apparent advantage in cost for a multistage switching network is offset by an increase in the transfer delay between any input-output terminal pair due to the complexity of each multistage switch, and due to data blocks (explained in the next paragraph) in the network [11]. For example, in case of a Banyan tree multistage switching network the transfer delay through the network is proportional to $O(N^a(\log_2 N)^2)$, where $0 < a < 1$ [11]. Whereas, the transfer delay through a crossbar network is proportional to O(N) [11]. Let us consider the case when N = 32. The transfer delay through the crossbar network is proportional to O(32) and for the Banyan tree is proportional to O(830) ($= O(32(\log_2 32)^2)$), if $a = 1$. We see an enormous increase in the transfer delay in case of the multistage switching networks.

Most multistage switching networks are blocking networks. A network is defined as *blocking* when there are conflicts in the use of network communication links for the simultaneous connection of more than one input-output terminal pair. Once again it must be noted that we will be dealing only with multiple input-output combinations where no two input-output terminal pair have the same output. For example, as shown in Figure 3, in the case of a multistage cube network there is a conflict in the network communication link between stage *1* and stage *0* (enclosed between hatched lines) for the input-output terminal pairs (5-0) and (7-1); therefore, data to be sent through one input-output connection has to be held and

sent after the other connection is accomplished, thereby increasing the overall data transfer time.

Multistage switching networks may be used to effect a compromise between the data transfer rate and network cost. A trade-off has to be accomplished by sacrificing speed for the network cost while selecting a multistage network. If a multistage switching network is present in the system with multiple data flow, it will slow the system due to its blocking nature. Therefore, in applications where speed is not of prime importance, the multistage switching networks can be used to achieve a low cost design by sacrificing speed [11]. When any multistage switching network is fabricated using WSI, the network is vulnerable to failures during the fabrication process due to the complexity involved in the multistage switches and interconnection links [11]; these failures lead to the presence of faulty switches and links in the network. Therefore, to improve the ability of multistage networks to tolerate the presence of faulty modules, designers have proposed to add one or more extra switch stages to the existing networks [22, 23 and 24]; however, fault-tolerance in these designs is restricted to the faulty module in a certain location rather than any general location. The extra stage cube proposed in [22] is an example of such a design. In the case of an eight-input extra stage cube network, the network cannot tolerate multiple faulty switches in stages n, 0 and stages n-1, . . . 1, simultaneously (where stage *n* is the extra stage provided) [22]. This does not map to the real situation where faulty modules are distributed at random locations. Further, adding an extra stage to the existing multistage network increases the complexity of the network, making it vulnerable to failures during fabrication. Since a switch in the multistage network is used to connect more than one input-output terminal pair, in case of fault in the switch, the input-output connections done through that switch cannot be accomplished. But, in case of a crossbar network

since a switch connects only one input-output pair, a switch fault affects only that input-output connection.

The multistage switching networks also result in an irregular hardware structure due to the complex switches and criss-crossing of interconnection links; if system designers wish to modify the existing network structure, that structure must be altered in its entirety [11]. The crossbar network, on the other hand, has a regular matrix-like structure similar to the memory structure; this matrix-like structure gives system designers more flexibility in adding switches and links without redesigning the whole structure. In addition, the crossbar network has a simple control mechanism due to the nature of switch access and switch function when compared to any multistage switching network; multistage switches are usually multifunctional, whereas crossbar switches perform only one function [11]. The crossbar network is arranged in the form of a matrix, and the arrangement is referred to as *a switching matrix* [11]. Only two control signals, represented as the row and column signals, need to be generated by the control unit to access any particular crossbar switch in the switch matrix [10]. This simple access leads to the design of a simple control unit, which reduces the area occupied by the control unit and reduces the control overhead [11]. This reduction in control unit area implies that more area is available for the expansion of the existing crossbar network.

Further, the VLSI implementation of a multistage network does not necessarily result in lesser area when compared to a crossbar network as stated earlier in this section. The Banyan multistage switching network, which is a representation of the other multistage switching networks like the Cube, the Omega, has a cost growth rate which is proportional to $O(N^2)$ and not the predicted $O(N\log_2 N)$ [11]. Further, crossbar networks of size 32x32 and 32x64 have already been fabricated using WSI [11]. The areas occupied by these networks are 3.4x3.4 mm^2 and 3.4x7.8 mm^2, respectively.

Therefore, based on the above discussion, we can state that for the WSI single-chip fabrication of a vector processor, a crossbar network is more suitable than a multistage switching network. Since the vector processor requires a network that provides full interconnection capability and has fast data-transfer rate, the crossbar network is ideal.

Pipeline Net

As stated in Chapter I, the study of the FTVP to be introduced in the next section originates from the pipeline net discussed in this section. Therefore, we will discuss about the pipeline net as an introduction to the FTVP. Using the pipelines and the crossbar network, a pipeline net is proposed in [20]. The pipeline net is constructed from interconnecting multiple functional pipelines through two buffered crossbar networks [20]. The pipeline net is a two-level structure, and is made of multiple functional pipelines (FP), two buffered crossbar networks (BCN) and a set of vector registers (R) [20]. Multiplexers are used to connect the Rs to the FPs, or the FPs to the FPs [20]. All FPs are identical and multifunctional [20], and each FP can execute addition, subtraction, division, multiplication or a logic function during a particular cycle [20]. The Rs hold the operand and results. The BCNs provide a dynamic connecting path among the FPs and Rs. A collection of fetch/store pipelines are used to transfer data between the main memory and Rs, similar to the memory access pipelines present in the Cray X-MP and the Fujitsu VP-200 [20].

The pipeline net is used for the computation of Vector Compound Functions (VCF) [20]. The VCFs are a collection of linked scalar operations to be executed repeatedly many times in a looping structure [20]. This looping structure is referred to as the *forpipe loop*. The VCFs are converted into the forpipe loops, and are evaluated by the pipeline net [20]. The syntax of a forpipe loop is

forpipe i := 1 to n do <body>

All VCFs are represented in this syntax [20]. For example, consider the following FORTRAN loop:

```
    DO 1 I = 1 to 400
1       X[I] = Q + Y[I] * (R * Z[I+10] + T * P[I+11])
```

This is represented in the syntax of forpipe loop as

```
forpipe i:= 1 to 400 do
begin
    x[i] := q + y[i] * (r * z[i+10] + t * p[i+11])
end,
```

where i is the loop index, and the compound statement within begin-end forms the loop body.

The VCFs are evaluated in two steps. In the first step, configuration of the pipeline net is done using the SET instructions. The actual execution is done in the second step by the START instruction, enabling the operations of a particular cycle. The SET instruction is used to select the function of a pipeline, or the connection pattern in a crossbar network. The syntax of a SET instruction is

SET unit, value

The unit is either a functional pipeline or a crossbar network [20]. If the unit refers to a functional pipeline, then the value denotes the arithmetic or logic operation performed by the pipeline. If the unit refers to a crossbar network, then the value denotes the connection pattern in the crossbar network [20].

A START instruction is issued to enable the pipeline net operation. The syntax of a START instruction is

START m, k

This implies "start to execute for m clock periods with an operand entering the pipeline net every k clock period" [20].

The FORTRAN loop presented previously is used as an example for the pipeline net implementation. The program graph of the above loop is shown in

Figure 4. Let the add and multiply functions require two and four clock periods, respectively. The program graph is mapped to the pipeline net as shown in Figure 5. Following are the sequence of setup instructions needed to set up the pipeline net:

*SET FP1, *;* : Set FP1 to multiplication
*SET FP2, *;*
SET FP3, +;
*SET FP4, *;*
SET FP5, +;
SET BCN1, α; : Set BCN1 to connection pattern α
SET BCN2, ß;

Figure 6 shows the crossbar network implementation obtained after the execution of SET instructions. After the pipeline net is configured, the VCFs are evaluated by passing the operand from the Rs through the pipeline net by issuing a START instruction. The final result is stored back in an register (R).

Even though the pipeline net is capable of vector processing, it does not favor WSI fabrication due to the irregularity in its structure. Multiplexers, present in the pipeline net, require additional control signals apart from the control signals required for the crossbar switches. This leads to the design of an additional control unit for the multiplexers. Functional pipelines that can execute all the basic arithmetic and logic functions are difficult to design, and are vulnerable to failures during fabrication due to their complexity. The methods for introduction of the delays in the crossbar network and the procedures to convert the program graphs to pipeline nets are complicated. It hasn't been clearly determined whether the software or hardware is going to execute them. Further, the register-register transfer is not considered in the pipeline net design. Finally, to start the execution phase of any pipeline net operation, an exact prediction of the number of clock periods required for the execution of the VCFs is needed, as shown in the START instruction; this will be difficult.

The proposed FTVP overcomes all the above difficulties in various manners. For instance, the FTVP proposed in this study has a regular structure favoring WSI

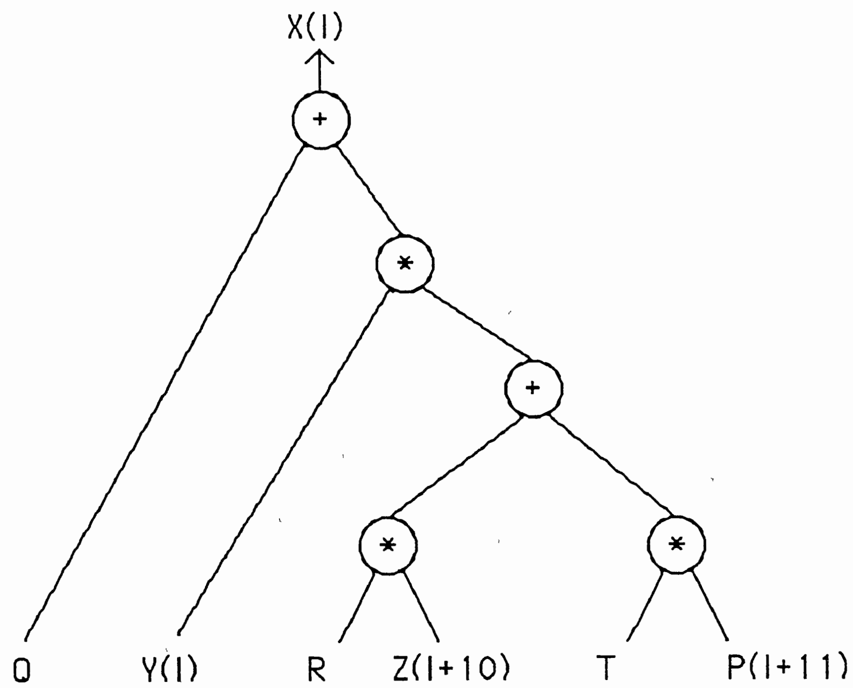


Figure 4. Program Graph

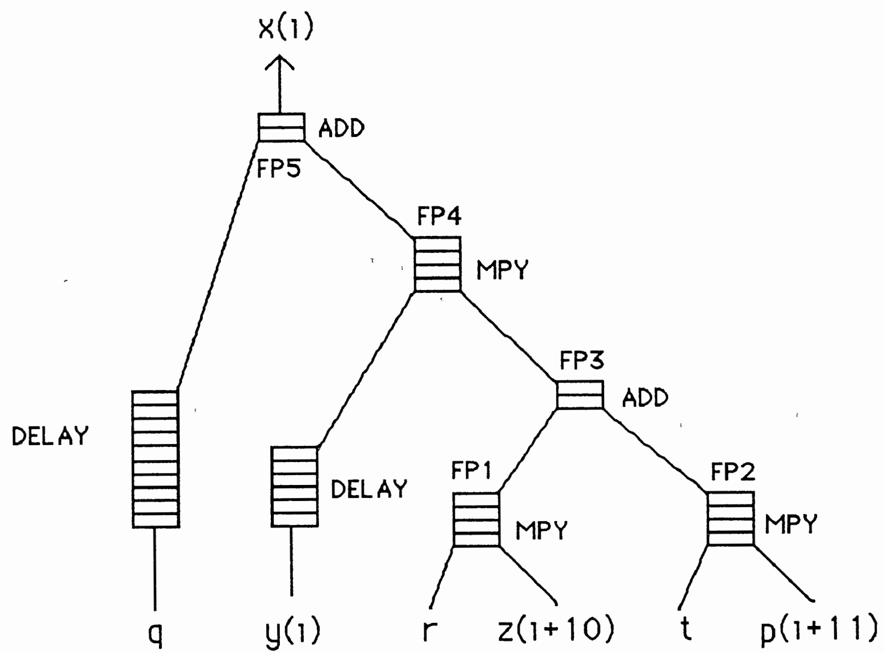


Figure 5. Pipeline Net Implementation

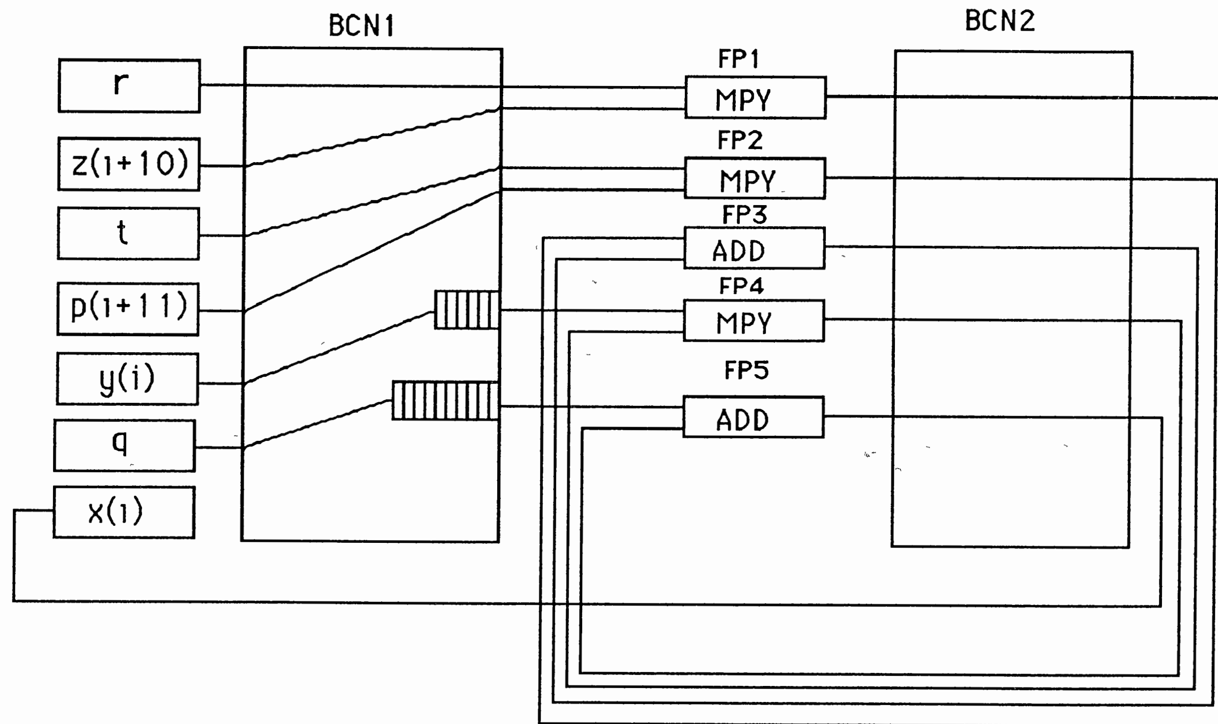


Figure 6. Crossbar Network Implementation

fabrication. No multiplexers are present, and the switch control is simple. A pipeline which can execute either multiplication or addition is considered, which leads to a simpler pipeline design than that required for the pipeline net. Buffers in the interconnection networks are eliminated, and data buffering (to be explained in the section on the arithmetic units of the FTVP) is done by the hardware. Hardware data buffering eliminates the need for an accurate prediction of the number of clock periods required for the execution of any FTVP operation. Finally, the FTVP is designed to execute all types of data transfer involving in the vector processor.

Basic Structure of FTVP

A structure of the FTVP suitable for WSI is shown in Figure 7. A three-level crossbar network is used to interconnect the arithmetic units and vector registers. The first-level network is Crossbar Network 1 (CBN1), the second-level is CBN2, and the third-level is CBN4. CBN1 is logically separated into two parts, CBN1 and CBN3, to simplify understanding; but, physically CBN3 is a part of CBN1. Both vector and scalar processing are supported by this architecture as explained in the subsequent sections. The vector registers are connected to the arithmetic units by CBN1. Feedback connections from the arithmetic units to the arithmetic units are done by CBN2 and CBN3. CBN2 and CBN4 connect the arithmetic units to the vector registers. The register-register connection is done by CBN4 through CBN1.

Various connection patterns in the crossbar networks are accomplished by crossbar switch settings. The control signals needed to accomplish the various connection patterns are simple; this is due to the fact that the nature of switch access in the crossbar network is simple as explained in the section on interconnection networks. Each vector register of the FTVP is a simple register

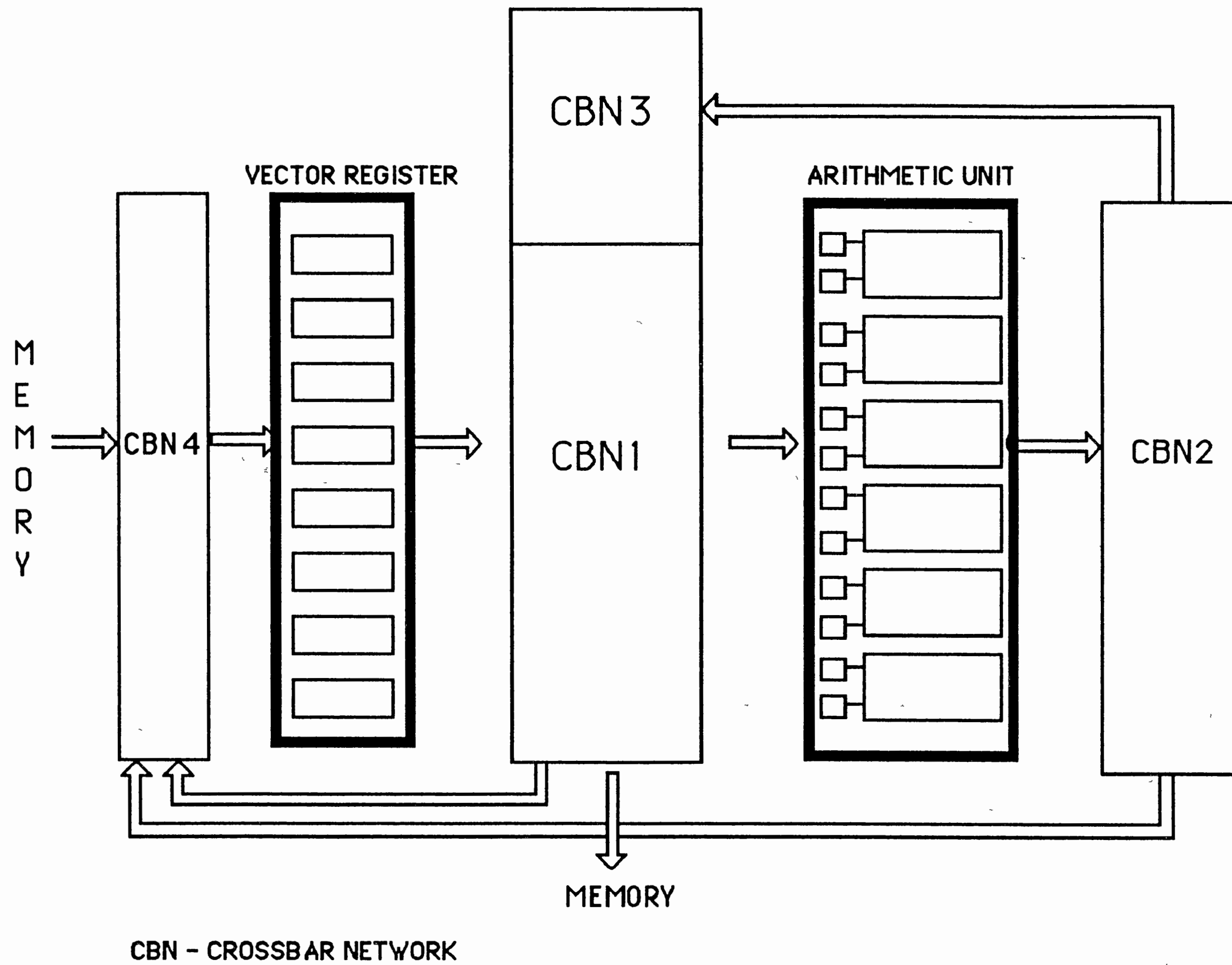


Figure 7. Hardware Model of the FTVP

which can store only one vector datum. Each arithmetic unit of the FTVP consists of two buffers and a pipeline. The pipeline in the arithmetic unit can execute either addition or multiplication. The two buffers present in the arithmetic unit provide data buffering (to be explained in the section on the arithmetic units of the FTVP).

Since the FTVP is proposed as a single-chip processor, the presence of faulty modules in the FTVP due to WSI fabrication are of major concern. The types of faulty modules that may be present in the fabricated FTVP, and the method for achieving fault-tolerance are discussed in Chapter III. Simple instructions are proposed to allow the FTVP to build a long pipeline chain. This will be the major application of the FTVP. The hardware resources are "exposed" to the software and are controlled by simple instructions as in the case of a RISC processor [24 and 37]. We will restrict our discussion to the setting up of one pipeline chain at a time in the FTVP. Multiple pipeline chains are not allowed to be set up in the FTVP. If more than one pipeline chain needs to be set up, the pipeline chains are set up sequentially. Having given a general introduction of the FTVP, we will discuss the structure of a vector register and the arithmetic unit present in the FTVP next. The crossbar network used to interconnect the arithmetic units and vector registers has been discussed earlier in this chapter, and therefore, will not be discussed further.

Structure of the Vector Register

Figure 8 shows the structure of a vector register in the FTVP, each of which consists of an array of scalar registers and a set of control logic. Each vector register can hold only one vector datum and is associated with four control logic codes: count, skip, SO/DEST flag and busy flag. The control logic code is set explicitly by a special instruction which will be discussed in the section on basic instructions. The count gives the number of scalar elements of a vector stored in the vector register.

Scalar data is stored in a vector register with count = 1. The skip gives the skip distance to the next element of a vector stored in the vector register; the SO/DEST flag indicates whether the particular vector register is the source register or destination register for a pipeline chain operation; the busy flag indicates the availability of the vector register.

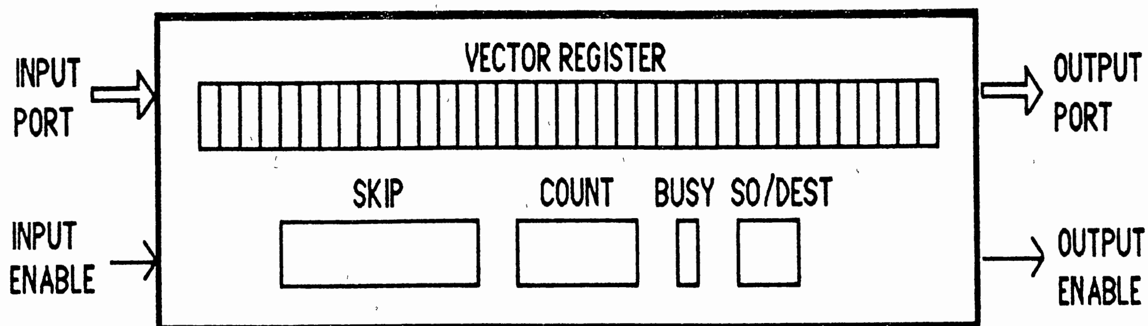


Figure 8. Structure of a Vector Register

Each vector register has one input port and one output port. Data is written into the vector register through its input port, and read from the vector register through its output port. Each port of the register has an enable signal accompanying the data in the port. Data in the ports of any vector register are valid only when the enable signal of the respective ports is true. Each vector register allows only one access at a time, as each register has only one control logic code. Either a read or write operation can be performed by the vector register at any particular period of time, since the registers allow only one access at a time. Simultaneous read and write operations are not allowed by the vector registers of the FTVP.

To access the data stored in a vector register, the busy flag of the register needs to be checked. If the busy flag of the register is set, then it implies that the register is busy with some read or write operation, and it cannot be accessed. If the busy flag of the vector register is not set, then the access mode of the register may be set by initializing the count, skip and SO/DEST codes. A trigger signal from the control unit is then sent to access the vector register. This trigger signal automatically sets the busy flag of the register, thereby preventing it from another access until the task assigned to it is completed.

If a vector register is specified as a *source register*, then, upon receipt of the trigger signal, the data stored in the register are sent through the output port, one element per clock cycle, and the control logic is updated. During the read operation, the output port enable signal of the vector register is true until all the elements of the stored data are sent, thus indicating the validity of the emerging data. The count logic of the register is decremented for every element of the vector sent through the output port. If count = 0, then all the elements of the vector would have been sent, and the enable signal of the output port is automatically reset.

If a vector register is specified as a *destination register*, then when the enable signal of the input port becomes true, data in the input port is written into the register, one element per clock cycle, updating the control logic until the register receives all the elements of the assigned vector. When the enable signal of the input port becomes low, it implies that all the elements of the assigned vector have been written into the register. After data are sent or received by a register, the busy flag of the register is automatically reset. If all the busy flags of the vector registers present in the FTVP are kept in a centralized control unit, the set of flags will be equivalent to the scoreboard register in a RISC processor [25].

A vector register of the FTVP can be read or written by the external memory or by an arithmetic unit or by another vector register. All source registers for a

particular pipeline chain send the stored data synchronously to the arithmetic units with a common trigger signal from the control unit. This is termed the *execution phase* and is started by a special instruction discussed in the section on the basic instructions of the FTVP.

Structure of an Arithmetic Unit

Figure 9 shows the structure of an arithmetic unit in the FTVP. Each arithmetic unit consists of two buffers and a pipeline which is either an adder or a multiplier. Each arithmetic unit consists of two input ports and one output port. Data in each port of the arithmetic unit has a signal, termed as *the enable signal,* to indicate the availability of valid data in the port.

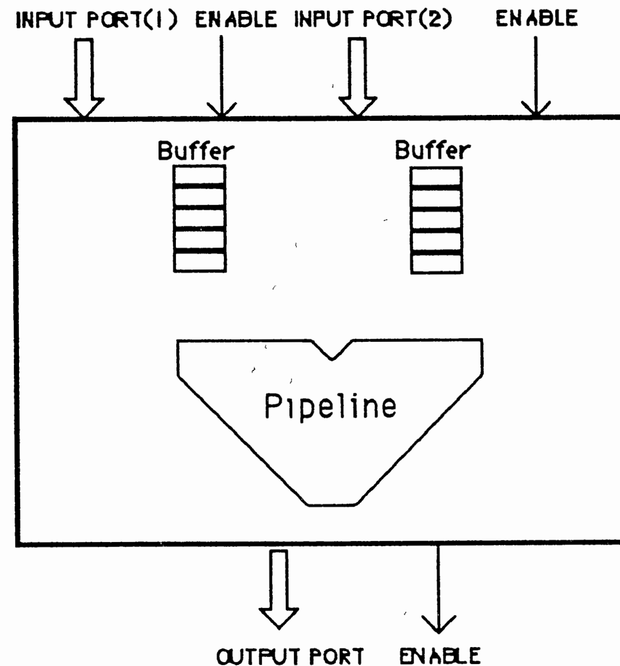


Figure 9. Structure of an Arithmetic Unit

The enable signal to the input ports of the arithmetic unit will originate either from the output port of a vector register or from the output port of another arithmetic unit. When the results begin to emerge out of the last stage of the pipeline in the arithmetic unit, the enable signal of the output port automatically becomes true. The enable signal remains true until all the results have emerged out of the pipeline.

Since an arithmetic unit may be involved in a long pipeline chain, the two input vectors to that arithmetic unit may arrive at different times, traversing different paths in the FTVP. These path delays have to be equalized in order to synchronize the arrival of both input vectors to the pipeline. This is referred to as *data buffering* and is done by the two buffers present in the arithmetic unit. For example, consider an arithmetic unit which requires two vectors IN(1) and IN(2) for evaluation. Suppose, IN(1) has arrived at the input port(1) of the arithmetic unit before IN(2). Since IN(1) has arrived before IN(2), IN(1) is held in a buffer of the arithmetic unit until IN(2) arrives. The arrival of IN(2) is indicated by its enable signal. When the enable signal of IN(2) becomes true, it implies that IN(2) has arrived; therefore, IN(1) held in the buffer and IN(2) which has arrived at the input port(2), are sent to the pipeline in the arithmetic unit for processing. In this way, we can ensure that both the input vectors are fed to the pipeline in the arithmetic unit at the same time. This hardware buffering frees the compiler from the need to provide data buffering, as in the case of the pipeline net design [20].

In a situation where one input vector to an arithmetic unit is shorter than the other input vector, the last element of the shortest input vector to the arithmetic unit is held in a buffer and used as the input to the pipeline until all the elements of the other input vector are sent to the pipeline. For example, if IN(1) and IN(2) are the two vectors to an arithmetic unit, and the number of elements of IN(1) is 10 and IN(2) is 20. After the first 9 elements of both IN(1) and IN(2) have been sent to

the pipeline, the 10th element of IN(1) (which is the last element) is held in the buffer of the arithmetic unit and used as the input to the pipeline, until all the 10 remaining elements of IN(2) have been sent to the pipeline. In this way, the FTVP can handle scalar-vector operations because a scalar data is a vector with one element.

Chaining Capability

Pipeline chaining in the FTVP is a linking process that occurs when the results obtained from one arithmetic unit are fed directly to another arithmetic unit. FTVP has the capability of providing long pipeline chains, and accomplishes chaining by various switch settings in the interconnection networks. This differs from the dual-mode of the i860 where two pipelines are linked through the special dual-mode instructions [9]. Consider the FORTRAN loop example whose program graph was given in Figure 4. The program graph is translated to the FTVP implementation as shown in Figure 10. Since all the source registers for a particular pipeline chain send the data stored in them simultaneously upon receipt of a trigger signal, Y and Q arrive at the input ports of AU3 and AU6, respectively, before the other input vectors to AU3 and AU6 arrive. Therefore, Y and Q are held in the buffers of AU3 and AU6, respectively, until the other input vectors to AU3 and AU6 arrive. The delay involved due to different path lengths is thus equalized automatically by this hardware buffering.

A series of FORTRAN kernels have been developed by the US National laboratories to evaluate the performance of vector processors and supercomputers [21]. These FORTRAN kernels are called the *Livermore loops*. The Livermore loops have been extracted from various vector processing applications. Fourteen Livermore loops have been commonly used over a decade as the *primary means* to

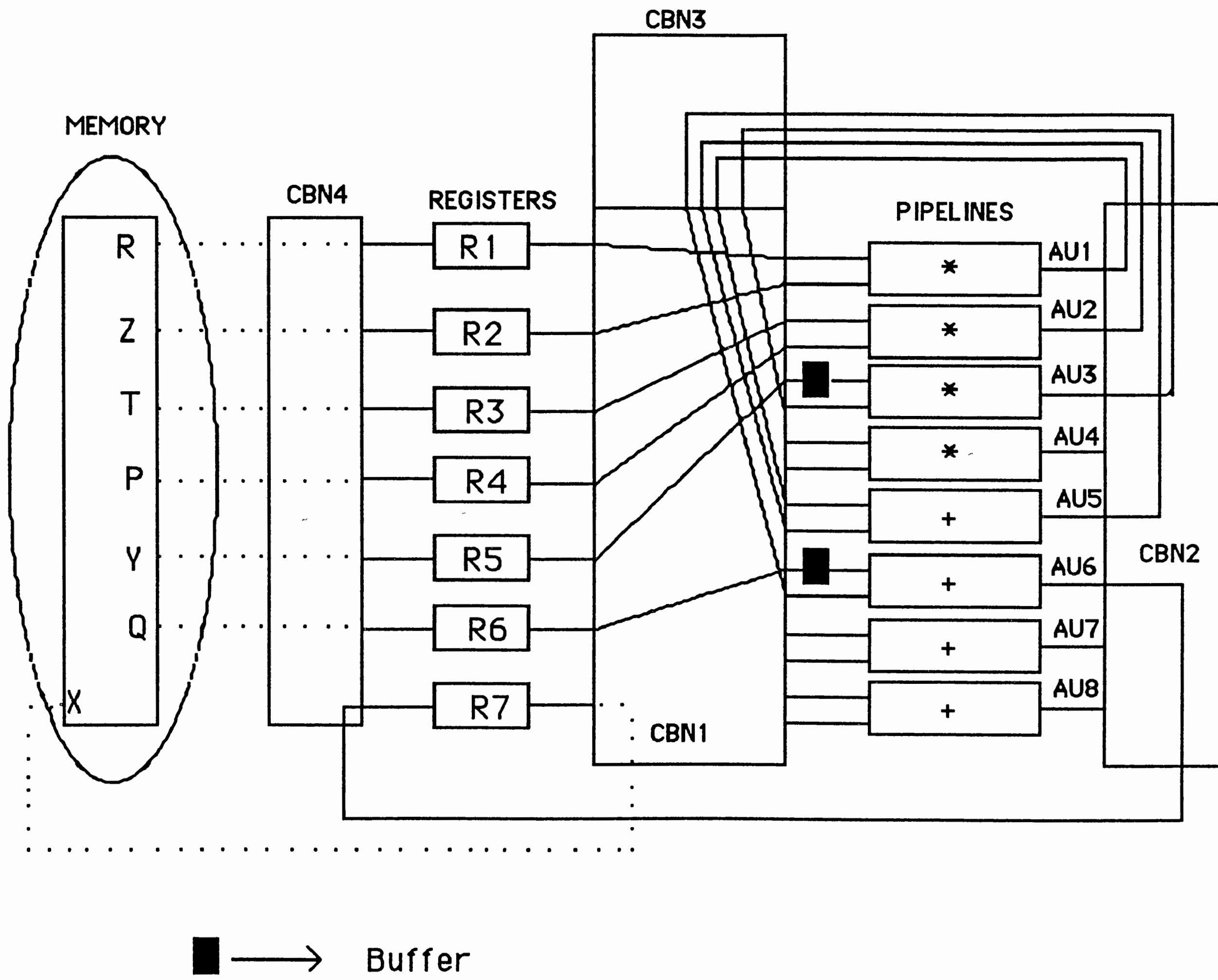


Figure 10. FTVP Implementation of the Example Loop

obtain a simple *snapshot* of the complex architectural performance of a vector processor [21]. The Livermore loops provide typical benchmark programs for vector processing applications. Even though the operation and performance measurement of a vector processor cannot be expressed as a simple set of numbers, the Livermore loops have served as the primary benchmark programs for nearly a decade [21]. Since we have not actually fabricated the FTVP shown in Figure 7, the Livermore loops present a theoretical way of evaluating the FTVP performance. The Livermore loops are listed in Appendix A.

Figure 11 shows the pipeline chaining performed for Livermore loop 9. The result obtained from one arithmetic unit is directly sent to the next arithmetic unit through CBN2 and CBN3. Only the final result is sent back to a destination vector register. Similar kinds of chaining operations can also be done for other Livermore loops. Livermore loops 8, 13 and 14 are not evaluated because of the indirect array addressing in loop 13, and the unknown index calculations in loop 8 (SIG) and loop 14 (GRD, DEX). Livermore loops 4, 5, 6, 9 and 11 can be evaluated by the FTVP only if simultaneous read and write operations are allowed by the vector registers, since they have recurrence relationship. This is due to the fact that Livermore loops 4, 5, 6, 9 and 11 have the same source and destination registers. Even though the FTVP will not be able to evaluate the vector loops with recurrence relationships right now, we proceed assuming that the FTVP will be able to evaluate when the registers allow simultaneous read and write operations. But Livermore loops 2 and 3 can still be evaluated by the FTVP even though there is a recurrence relationship, because, scalar data stored in the register Q for the loops 2 and 3 will be sent towards the arithmetic units before the results of the respective pipeline chains are written back into the register Q. For this reason, no simultaneous read and write operations is required for these loop evaluations.

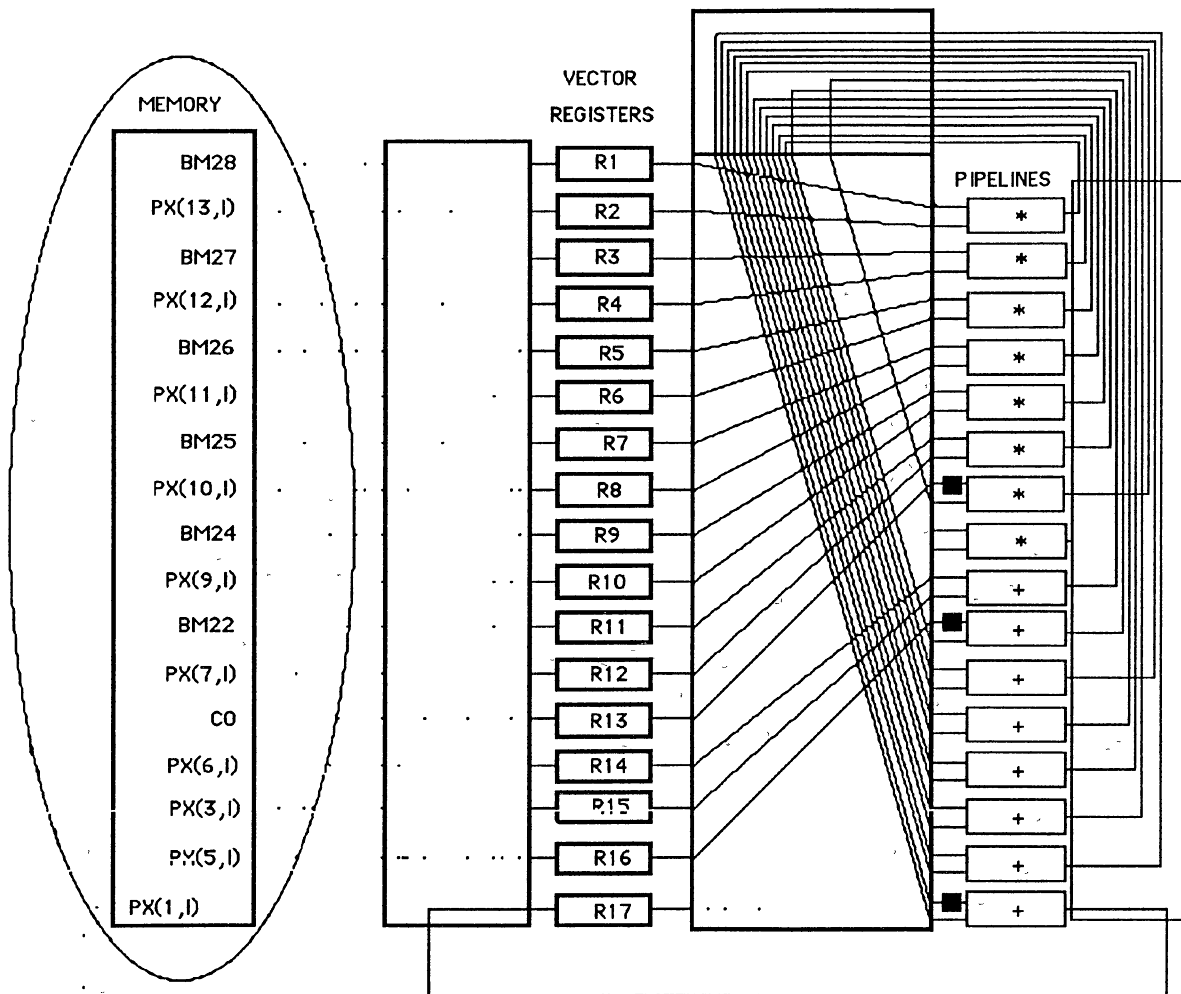


Figure 11. Chaining Operation for Livermore Loop 9

The evaluation of Livermore loops 4, 5, 6, 9 and 11 by the FTVP will be slow as they are essentially scalar operations. For example, consider the Livermore loop 11 shown below:

```
         DO 11 I = 2, 1000
   11    X(K) = X(K-1) + Y(K)
```

We see that there is a recurrence relationship between X(K) and X(K-1). A new iteration for the calculation of X(K) cannot be started until the previous iteration is completed. If a pipeline with three stages is assumed, then a new iteration can be initiated only when the pipeline has sent the results of previous iteration; therefore, no overlapping is done. Due to this non-overlapping execution a new iteration can only be initiated every third clock period. Only one stage of the pipeline will be evaluating at a particular clock period, while the other two stages are idle due to this non-overlapping evaluation. This type of problem has been studied by Kogge who proposed a *double cycling method* to reduce the pipeline idle time in this kind of situation [26]. The double cycling method serves to reduce the vector loop that has a recurrence relationship to a vector loop that has a latency of one [26]. *Latency* is defined as the number of clock periods that elapse between two successive iterations. If the pipelining is achieved, then a new iteration may be done every clock period due to overlapping. But in Livermore loop 11, since no overlapping is done, the latency is high (three, if a pipeline with three stages is assumed). To reduce the latency in the above loop, consider the following modification.

$$X(K) = X(K-1) + Y(K) \quad \ldots\ldots\ldots (1)$$
$$X(K-1) = X(K-2) + Y(K-1) \quad \ldots\ldots\ldots (2)$$
$$X(K-2) = X(K-3) + Y(K-2) \quad \ldots\ldots\ldots (3).$$

Substituting (2) and (3) in (1) and substituting it back in the original equation, we have

```
         DO 11 K = 2, 1000
         B(K) = Y(K-2) + Y(K-1) + Y(K)
   11    X(K)  = X(K-3) + B(K)
```

This implies that the calculation of X(K) is dependent on X(K-3) and not X(K-1), which implies that the next iteration in the calculation of X(K) is dependent on the results obtained three iterations before and not on the results from the previous iteration. Therefore, overlapped evaluation can be done, and latency can be reduced. The calculation of B(K) can be overlapped with the calculation of X(K), and new iteration can be initiated every clock period.

The Kogge's double cycling method is a two-step evaluation method. For example, in the modified Livermore loop 11 shown above, calculation of B(K) is the first step, and calculation of X(K) is the second step. The two steps can be combined into a single step and evaluated, as shown here:

```
        DO 11 K = 2, 1000
11      X(K)  = Y(K-2) + Y(K-1) + Y(K) + X(K-3)
```

The Kogge double cycling method is used to recast Livermore loop 5 also, as shown in Appendix B. The Kogge double cycling method is effective only when the number of stages in all the pipelines of the FTVP are same, so that the method can be applied uniformly to all the pipelines no matter what function they execute [26]. In addition, to apply Kogge's double cycling method in the FTVP, the vector registers of the FTVP must allow simultaneous read and write operations.

Basic Instructions

The basic instructions required for the FTVP are divided into four groups: the register read/write and register control instructions, the arithmetic instructions, the network instructions and the execution phase instructions, all of which are shown in Table III. Each instruction shown in Table III is assumed to be executed in one clock period. The initial group consists of a memory-register write operation and a resister-memory read operation which are executed by the LOAD and STORE instructions, respectively. The MOVE instruction executes the

TABLE III

BASIC INSTRUCTIONS FOR THE FTVP

Instructions	Type	Comments
Group - 1	**Register read/write and register control instructions**	
LOAD reg., variable	Memory to Register Write	Load the variable in the memory to the specified register.
STORE reg., variable	Register to Memory Write	Store the data in the register in the memory.
MOVE source, destination	Register - register transfer instruction	Register - register transfer instruction.
REG(reg., len, skip, SO/DEST)	Register control	Loads the control parameters into the control logic of the specified register. reg - Specified register len - Length of the data stream skip - Skip distance SO/DEST - Source or destination
Group - 2	**Arithmetic Instructions**	
add s1, s2, d	Pipelined add instruction	s1, s2 denote the two sources (registers "r" or temporary variables "t"); d denotes the destination (registers "r" or temporary variables "t").
mul s1, s2, d	Pipelined multiply instruction	
Group - 3	**Network instruction**	
set(k, i, j)		Set the crossbar switch specified by k, i, j, where "k" is the network index (CBN1, CBN2, CBN3, CBN4), "i" and "j" are the row and column indexes of the crossbar switch.
reset(k, i, j)		Resets the crossbar switch specified by k, i, j.
Group - 4	**Execution phase instruction**	
start (list of registers)	Starts the execution phase	Starts the execution phase and triggers the vector registers
wait (list of registers)		Holds the FTVP idle until all the specified registers are ready.

register-register data transfer. The REG instruction sets the control logic code of the specified vector register. All register read or write instructions must be preceded by a REG control instruction to set the access mode of the register. The second group are the *add and mul* arithmetic instructions. Each arithmetic instruction has two source variables and one destination variable. The source and destination variables of an arithmetic instruction can be either a register or a temporary variable (denoted by t). Temporary variables are provided in the arithmetic instructions to facilitate setting up of a pipeline chain. The third group of instructions, set and reset, are used to set and reset a crossbar switch, respectively. The final group of instructions, start and wait, start and hold the execution phase in the FTVP, respectively. The wait instruction holds the FTVP from executing any new instruction until all the registers specified in the list receive their data. During the start of the execution phase, all vector registers that send their data towards the arithmetic units are triggered by a common trigger signal from the control unit, and this achieved by the start instruction.

Since we have restricted ourselves to the execution of one pipeline chain at a time in the FTVP, the typical pattern of instructions that occur for every pipeline chain is shown below:

Sequence	Pattern
1	Set the control logic of the registers for the memory-register access.
2	LOAD instruction.
3	Wait for the data to be loaded.
4	Set the control logic of registers for the pipeline chain.
5	Generate the arithmetic instructions.
6	Virtual to physical translation. Generate the switch settings.
7	Start the execution phase.
8	Wait for the results.
9	Set the register control of the result register for the register-memory access.
10	Store the result in the memory.

The control logic for the registers that need data from the memory are initially set by the REG instructions for memory-register access. Memory access is then done by the LOAD instruction. A wait instruction is issued to hold the FTVP from executing any new instruction until all the registers in the list receive data from the memory; this is due to the fact that data loaded from the memory to the vector registers will vary in length, and the FTVP needs to wait until every specified register receives every element of the assigned vector from the memory. After the memory access, the mode of all source registers involved in a pipeline chain is reset for the pipeline chain operation. This is due to the fact that during the LOAD operation these registers received data from the memory, and therefore, were destination registers. For a pipeline chain, these registers send their data towards the arithmetic units, and therefore, are source registers. The arithmetic instructions are then generated. The compiler then generates the virtual addresses of the pipelines based on the arithmetic instructions. A translation procedure generates the physical pipeline addresses from the virtual addresses. The crossbar switch settings are then computed from the physical addresses. A translation procedure, and the reasons for the translation procedure are discussed in Chapter III. Once a pipeline chain is set, the start instruction is issued to start the execution phase. A wait instruction is issued to hold the FTVP from executing any new instruction until the destination registers of the pipeline chain specified in the list receives every element of the results. After the results are received, the mode of the destination registers are set for the register-memory access. The final results are sent back to the memory after completion of the execution phase by the STORE instruction.

Figure 12 shows the instructions generated for the example FORTRAN loop we have been considering till now. Scalar data loaded in the registers r1, r3 and r6 have count = 1. Once the execution phase is started, data stored in the source

Sequence	Instructions	Comments
1	REG(r1, 1, 1, DEST) REG(r2, 400, 1, DEST) REG(r3, 1, 1, DEST) REG(r4, 400, 1, DEST) REG(r5, 400, 1, DEST) REG(r6, 1, 1, DEST)	The register control logics are set for memory-to-register read operation.
2	LOAD r1, R LOAD r2, Z LOAD r3, T LOAD r4, P LOAD r5, Y LOAD r6, Q	The variables are read from the memory to the registers.
3	wait(r1, r2, r3, r4, r5, r6)	Wait till all the registers have received their data.
4	REG(r1, 1, 1, SO) REG(r2, 400, 1, SO) REG(r3, 1, 1, SO) REG(r4, 400, 1, SO) REG(r5, 400, 1, SO) REG(r6, 1, 1, SO) REG(r7, 400, 1, DEST)	The source registers for the pipeline chain operations are once again set. Register r7 is the final destination register.

5	mul r1, r2, t1
	mul r3, r4, t2
	add t1, t2, t3
	mul r5, t3, t4
	add r6, t4, r7
		Arithmetic instructions.

6	Virtual to physical address translation and generation of switch settings	Translation procedure.
7	start(r1, r2, r3, r4, r5, r6)	Start of the execution phase
8	wait(r7)	Wait till all results are written into register r7.
9	REG(r7, 400, 1, SO)	Register-to-memory write.
10	STORE r7, X	Final results are stored in the memory.

Figure 12. Instructions for the Example Loop

registers are sent to the arithmetic units, one element per cycle. The final results stored in register r7 are sent back to the memory.

The compiler is responsible for setting up one pipeline chain at a time. Since the FTVP is a RISC type processor, the compiler is assumed to know the exact content of the number of hardware resources in the FTVP. If the instructions for a particular vector loop are more than the hardware resources in the FTVP, the compiler divides the bigger loop into smaller vector loops, and evaluates the smaller vector loops one by one. The rules for division of a big vector loop into smaller vector loops are presented in Chapter IV. Further, the compiler is responsible for the allocation of all source and destination registers for a particular pipeline chain. A pipeline chain is removed by the reset instructions if the need arises. Otherwise the settings of the previous pipeline chain may be maintained for future use. The method by which the compiler achieves these is beyond the scope of this thesis.

We choose the software approach to build a pipeline chain in the FTVP over the hardware approach to minimize the control unit hardware required; this minimization leads to reduction in the chip-space occupied by the FTVP. But this software approach will introduce a large control overhead which hopefully will be offset by the speed enhancement achieved by single-chip implementation.

CHAPTER III

FAULT-TOLERANCE IN THE FTVP

As stated in Chapter I, the first objective of this thesis is the proposed fabrication of the FTVP introduced in Chapter II as a single-chip processor. The technique for the proposed single-chip fabrication is WSI. A short discussion on WSI is included in Appendix C. As discussed in Chapter I, a major drawback to the FTVP fabrication using WSI is the presence of faulty modules in the FTVP. We must find a way to detect the faulty modules present in the fabricated FTVP, and provide a technique that would make the FTVP fault-tolerant. However, detection of faulty modules in the FTVP is not the area of study of this thesis. Various techniques for fault-detection exist and are dealt extensively in [27, 28 and 29]. One technique for achieving fault-tolerance is by providing redundancy in the form of extra modules during fabrication to compensate for the faulty modules in the system. After fabrication, suitable routing algorithms developed prior to fabrication are applied to avoid the faulty modules, and utilize the good ones to obtain a functional design. In our proposed FTVP, a simple translation procedure achieves fault-tolerance. No redundancy in the form of extra FTVP modules will be provided during the proposed fabrication. Our translation procedure avoids the bad modules in the FTVP, and uses the good ones to set up a pipeline chain. The translation procedure converts the virtual addresses generated by the compiler to the physical addresses in the FTVP. Based on the physical addresses, the compiler computes the crossbar switch settings to build a pipeline chain.

Types of Fault and Fault Vectors

The proposed WSI architecture of the FTVP is shown in Figure 13. The general types of faulty modules that may be present in the FTVP are classified as a faulty arithmetic unit, a faulty register, a faulty crossbar network link, and a faulty crossbar network switch. A faulty arithmetic unit will include both the buffer and pipeline faults, and will be henceforth referred to as a pipeline fault. If a pipeline in the fabricated FTVP is faulty, a new fault-free pipeline is selected. A new fault-free register is selected in the case of a faulty register. Links in the crossbar networks are classified as horizontal and vertical. Links are represented by the vertical and horizontal lines in Figure 13. The links interconnecting CBN2 and CBN3 are considered as the horizontal links of CBN2. The links interconnecting CBN2 and CBN4 are considered as the vertical links of CBN4. In the case of a faulty horizontal link in CBN1 or CBN4, the register connected to that particular link cannot be accessed. Therefore, a new register must be selected. Since a new register is selected, we consider a horizontal link fault in CBN1 or CBN4 as a register fault. In the case of a faulty horizontal link in CBN2 or CBN3, a new fault-free horizontal link is selected. A vertical link fault in CBN1, CBN2 or CBN3 is considered as a pipeline fault and a new pipeline is selected. This is due to the fact that a faulty vertical link in CBN1, CBN2 or CBN3 denies access to the pipeline connected to that particular vertical link. Therefore, a new pipeline must be selected. A new fault-free vertical link is selected in the case of a faulty vertical link in CBN4. A switch fault present in any of the four networks is considered a vertical link fault for simplicity. Therefore, the steps taken for avoiding a faulty vertical link discussed previously, are implemented in case of a switch fault.

The compiler, while assigning the pipelines and networks through the translation procedure, does not know the exact physical location of the faulty

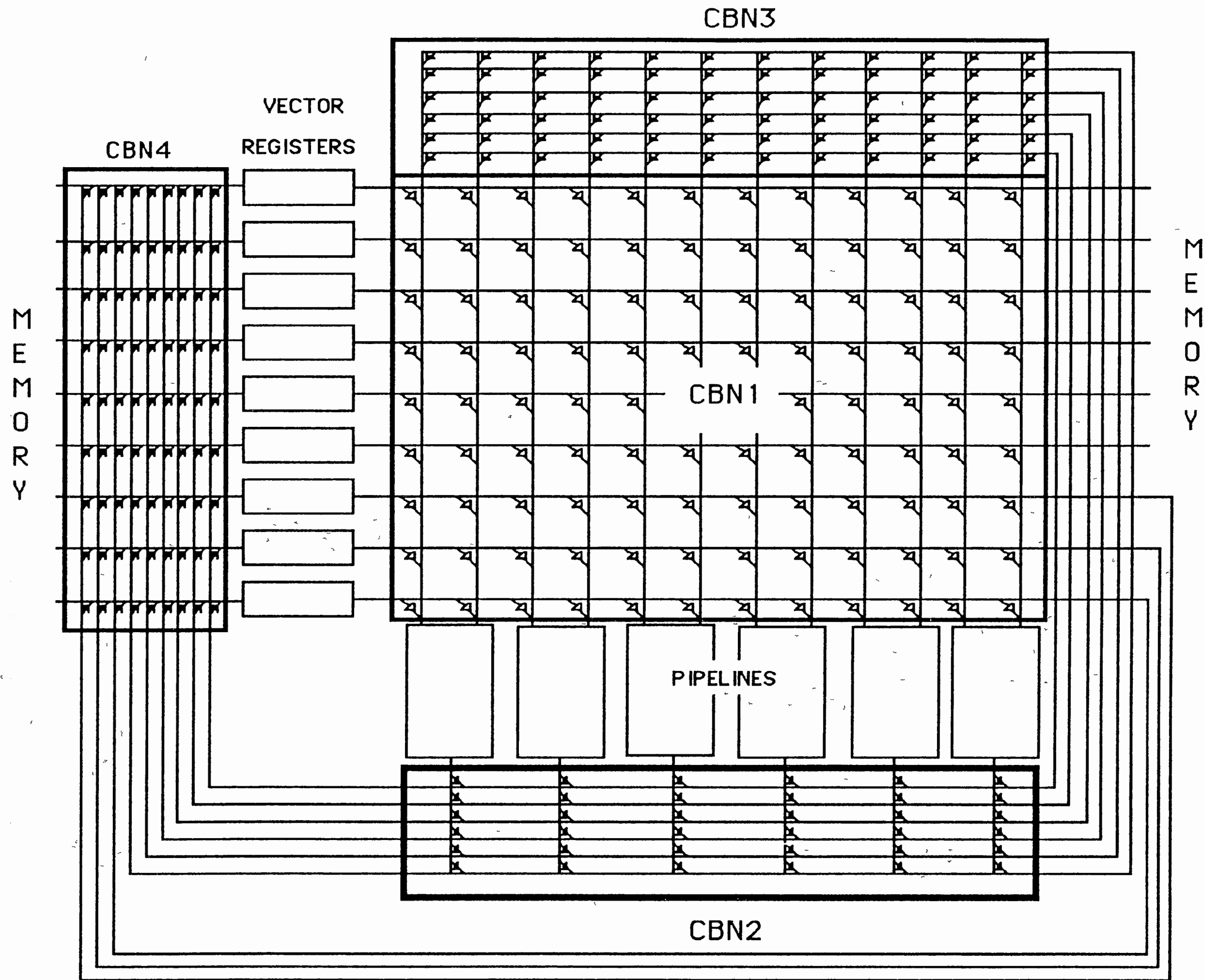


Figure 13. WSI Model for the Proposed Study

modules in the fabricated FTVP. But since the proposed FTVP is considered to be a RISC-type processor, the compiler is assumed to know that there are faulty modules in the fabricated FTVP. Therefore, the compiler generates the virtual addresses which have to be mapped to the exact physical addresses of the FTVP, taking into account the faulty modules in the FTVP. This mapping is accomplished by a translation procedure. The translation procedure uses the fault vectors presented below while determining the physical address in the FTVP.

The fault-free modules in the FTVP are represented at the software level by the fault vectors. It is important to note here that the fault vectors will contain the physical addresses of the fault-free modules in the FTVP. The fault-free registers in the FTVP are represented by the register fault vector. Similarly, the fault-free pipelines and the fault-free links are represented by the pipeline fault vector and the link fault vector. Since a switch fault is considered as a vertical link fault, no separate fault vector is provided for the fault-free switches.

The pipeline fault vector contains the physical address of the fault-free pipelines in the FTVP. A pipeline fault in the FTVP may occur as a result of the pipeline fault itself, or in the case of CBN1, CBN2 or CBN3, a vertical link fault or a switch fault. Therefore, the physical address of the pipeline which satisfies any of these situations is not present in the pipeline fault vector. Since the FTVP will have two sets of pipelines, namely adders and multipliers, each set of pipelines will have a pipeline fault vector to represent that particular set.

The register fault vector contains the physical address of the fault-free registers in the FTVP. A register fault in the FTVP may occur as a result of the register fault itself, or a horizontal link fault in CBN1 or CBN4. Therefore, the physical address of the register which satisfies any of these situations is not present in the register fault vector.

The link fault vector contains the physical address of the fault-free horizontal links in CBN1, CBN2 or CBN3. In the case of CBN4, the link fault vector contains the physical address of the fault-free vertical links.

Translation Procedure

Since the proposed FTVP is considered to be a RISC type processor, the compiler is assumed to have the knowledge of the number of fault-free hardware resources available for processing. Further, the compiler is assumed to be aware that there are faulty modules in the fabricated FTVP. But the compiler will not be aware of the exact location of the faulty modules in the FTVP. Therefore, the compiler generates a virtual address which is mapped to the physical address in the FTVP avoiding the faulty modules. A translation procedure maps the virtual addresses generated by the compiler to the physical addresses of the FTVP, avoiding the faulty modules. The compiler then computes the switch settings needed to set up a pipeline chain, using the fault-free pipelines and links. The steps involved in the translation procedure for each pipeline chain are:

Step 1: Determine the type of network.

Step 2: Assign the virtual pipeline and row addresses.

Step 3: Obtain the physical pipeline and row addresses by a 1-to-1 mapping procedure.

Step 4: Calculate the switch settings.

Based on the type of instruction generated by the compiler, the switching networks for that particular instruction are assigned in step 1 of the translation procedure. The virtual pipeline address (p') and the horizontal link virtual address corresponding to CBN2 (i'') for an instruction are assigned in step 2 of the translation procedure. The virtual addresses generated in step 2 are 1-to-1 mapped

to the physical addresses of the FTVP using the fault vectors in step 3. The switch settings are then computed from the pipeline physical addresses to set up a pipeline chain in step 4. Once a pipeline chain is set, a start instruction is issued to start the execution phase. The group I instructions shown in Table III are not considered as instructions required for a pipeline chain, as no pipelines are involved while these instructions are executed; only arithmetic instructions use the pipelines in the FTVP.

Now we will present the rules for assigning the switching networks for the arithmetic instructions as per step 1 of the translation procedure. As seen in Table III, each arithmetic instruction generated by the compiler has three variables: two source variables and one destination variable. If the source variable in an arithmetic instruction is a register, it implies that the arithmetic instruction requires input datum from a register. If the source variable in an arithmetic instruction is a temporary variable, it implies that the arithmetic instruction requires input datum from another arithmetic unit. If the destination variable in an arithmetic instruction is a register, it implies that the results of the arithmetic operation have to be stored in a register. If the destination variable in an arithmetic instruction is a temporary variable, it implies that the results of the arithmetic operation are to be sent to another arithmetic unit. Therefore, networks are assigned to each variable of the arithmetic instruction so as to achieve the above conditions, and the rules for assigning the networks are given below:

Rule 1: If the source variable in an arithmetic instruction is a register, the network assigned for that source variable is the CBN1. This is due to the fact that the arithmetic operation performed corresponding to the instruction, requires data from a register, and the data has to be fetched through CBN1.

Rule 2: If the source variable in an arithmetic instruction is a temporary variable (denoted by t), the network assigned for that source variable is the CBN3. This is

due to the fact that the arithmetic operation performed corresponding to the instruction, requires data from the output port of another arithmetic unit, and the data has to be fetched through CBN3.

Rule 3: If the destination variable in an arithmetic instruction is a temporary variable, the network assigned for that destination variable is the CBN2. This is due to the fact that the results of the arithmetic operation performed by the instruction, have to be sent to another arithmetic unit, and it has to be done through CBN2.

Rule 4: If the destination variable in an arithmetic instruction is a register, the networks assigned for that destination variable are the CBN2 and CBN4. This is due to the fact that the results of the arithmetic operation performed corresponding to the instruction, have to be stored in a register, and this has to done through CBN2 and CBN4.

For example, consider the instruction *add r1, t1, r2*. The switching networks assigned for this instruction are

Variable	Switching Network	Rule
r1 (source)	*CBN1*	*1*
t1 (source)	*CBN3*	*2*
r2 (destination)	*CBN2* *CBN4*	*4*

After step 1 is executed, the virtual addresses for an instruction are generated by the compiler in step 2. The virtual pipeline address for a particular instruction ranges from 0 to (number of fault-free pipelines - 1). For example, if there are four fault-free pipelines in an FTVP, then the virtual pipeline address ranges from 0 to 3. The virtual horizontal link address for a particular instruction ranges from 0 to (number of fault-free horizontal links of CBN2 - 1). The virtual to physical mapping procedure is executed then to obtain the physical addresses in step 3. The switch settings required for building a pipeline chain are then computed from the physical addresses in step 4. This final step is discussed before step 3, however, for ease in understanding the entire process.

In step 4 of the translation procedure, the crossbar switches are set in the interconnection networks by the compiler through the *set* instructions. Each network assignment in step 1 has a corresponding *set* instruction. As seen from Table III, a *set* instruction requires 3 parameters: *"k"* - the type of network, *"i"* - the row number of a crossbar switch in that network, and *"j"* - the column number of the switch in that network. The following are the rules for determining the three parameters:

Rule 5: k for a switch setting is determined based on the rules 1, 2, 3 and 4.

Rule 6: In the case of CBN1 or CBN4, the index i equals to the physical address of the assigned register in the arithmetic instruction. For CBN2, a virtual index i'' is generated by the compiler, which is mapped to i by a 1-to-1 mapping procedure to be discussed later. The physical index i thus generated is also the physical index i of CBN3 for the same variable.

Rule 7a: For CBN1, CBN2 or CBN3, j is computed from the pipeline physical address (p) obtained from the mapping procedure in step 3 of the translation procedure. If j corresponds to the first source variable in the arithmetic instruction, then $j = 2p$ (where p is the physical address of the pipeline). If j corresponds to the second source variable in the arithmetic instruction, then $j = 2p+1$ (where p is the physical address of the pipeline). If j corresponds to the destination variable in the arithmetic instruction, then $j = p$ (where p is the physical address of the pipeline). Consider the instruction *add r1, t1, r2*. Let the pipeline physical address (p) assigned to this instruction be *2*. The j indexes corresponding to the three variables in the instruction are

Variable	j Index
r1 (first source)	2 * 2 = 4
t1 (second source)	2 * 2 + 1 = 5
r2 (destination)	2

Rule 7b: For CBN4, the physical index *j* is equal to the physical index *i* of the preceding CBN2 switch setting (for the arithmetic instructions, every CBN4 assignment will be preceded by a CBN2 assignment, according to rule 4).

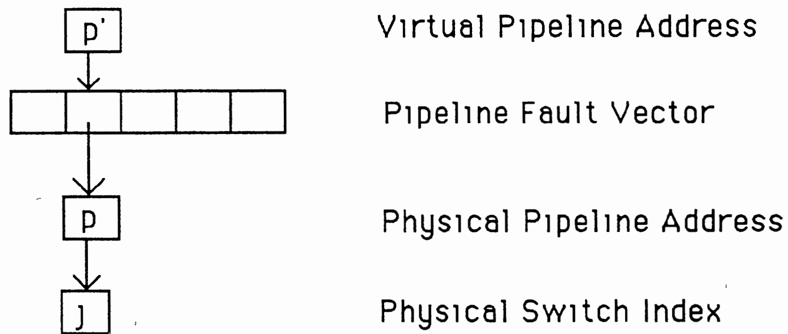
Rule 8: For the MOVE instruction only one network is assigned and the network is CBN4. This is due to the fact that the register-register data transfer performed corresponding to the instruction has to be accomplished by CBN4 in an FTVP.

The virtual to physical 1-to-1 mapping procedure executed in step 3 of the translation procedure is shown in Figure 14. The mapping procedure involves two steps. In the first step, the pipeline physical address (*p*) is obtained from the pipeline virtual address (*p'*). In the second step, the virtual index *i"* generated for CBN2 is mapped to the physical address (*i*) by the mapping procedure. The two steps involved in the mapping procedure are explained below.

Step 1: The virtual pipeline address (*p'*) generated by the compiler in step 2 of the translation procedure forms the column number of the pipeline fault vector. The address stored in that column of the pipeline fault vector is the physical address of the pipeline. The physical column index *j* required for the switch settings in CBN1, CBN2 or CBN3 are computed based on rule 7a.

Step 2: The virtual index link address for CBN2 (*i"*) generated by the compiler in step 2 of the translation procedure forms the column number of the link fault vector of CBN2. The address stored in that column of the link fault vector (*i'*) forms the column number of the link fault vector of CBN3. The address stored in that column of the link fault vector of CBN3 is the physical index *i* for CBN2 and CBN3 by rule 6. But, if a CBN2 assignment is followed by a CBN4 assignment, then the index *i'* obtained forms the column number of the link fault vector of CBN4. The address stored in that column of the link fault vector is the physical index *i* for CBN2 and the physical index *j* for CBN4 according to rule 7b.

Step 1: For CBN1, CBN2 and CBN3



Step 2: For CBN2, CBN3 and CBN4

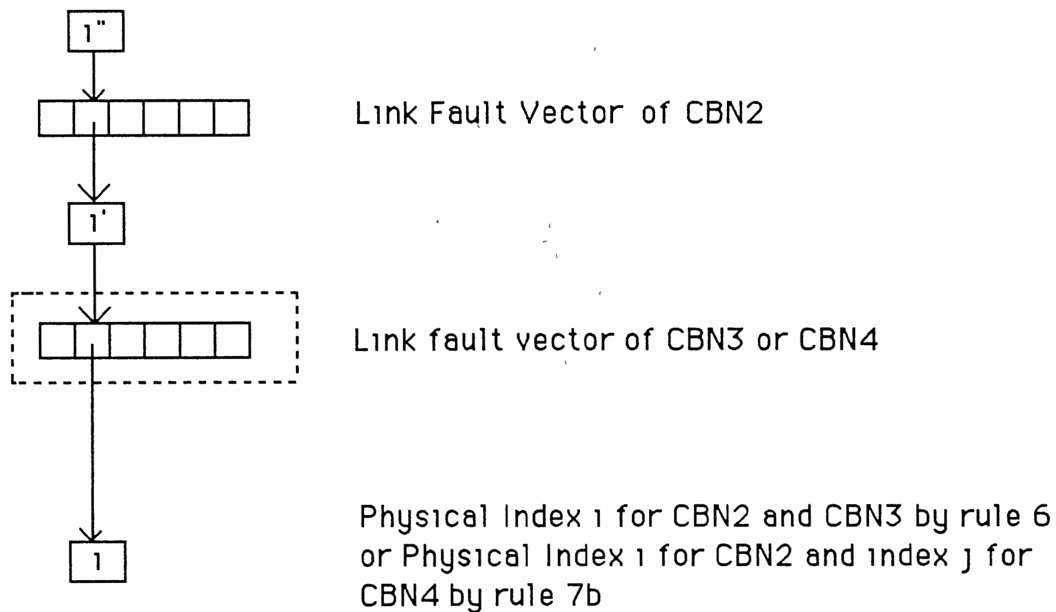


Figure 14. 1-to-1 Mapping Procedure

Example

The translation procedure discussed in the previous section is demonstrated using the example FORTRAN loop for various fault conditions. An eight pipeline structure shown in Figure 15 is considered with no register fault. Let the index *k* for CBN1, CBN2, CBN3 and CBN4 be 0, 1, 2, 3 and 4, respectively. The indexes *i* and *j* are shown in Figure 15. Let the physical address for *r1 -> r7* be *0 -> 7*, respectively. Pipeline fault vector for the multipliers is denoted as *pipeline fault vector(1)* and for the adders as *pipeline fault vector(2)*. The arithmetic instructions for the loop (from Figure 12), and the networks assigned by the rules 1, 2, 3 and 4 are

Arithmetic instruction				Network assigned		
mul	r1	r2	t1	CBN1	CBN1	CBN2
mul	r3	r4	t2	CBN1	CBN1	CBN2
add	t1	t2	t3	CBN3	CBN3	CBN2
mul	r5	t3	t4	CBN1	CBN3	CBN2
add	r6	t4	r7	CBN1	CBN3	CBN2
						CBN4

The virtual addresses generated are

Operation	Virtual pipeline	Virtual index i" for CBN2
mul	0	0
mul	1	1
add	4	2
mul	2	3
add	5	4

The link fault vectors with no link faults are shown below.

Column	0	1	2	3	4	5	6	7	
CBN1	0	1	2	3	4	5	6	7	Horizontal link physical address
CBN2	0	1	2	3	4	5	6	7	Horizontal link physical address
CBN3	0	1	2	3	4	5	6	7	Horizontal link physical address
CBN4	0	1	2	3	4	5	6	7	Vertical link physical address

Fault Free Condition

Physical addresses of the pipelines stored in the pipeline fault vectors with fault free condition are

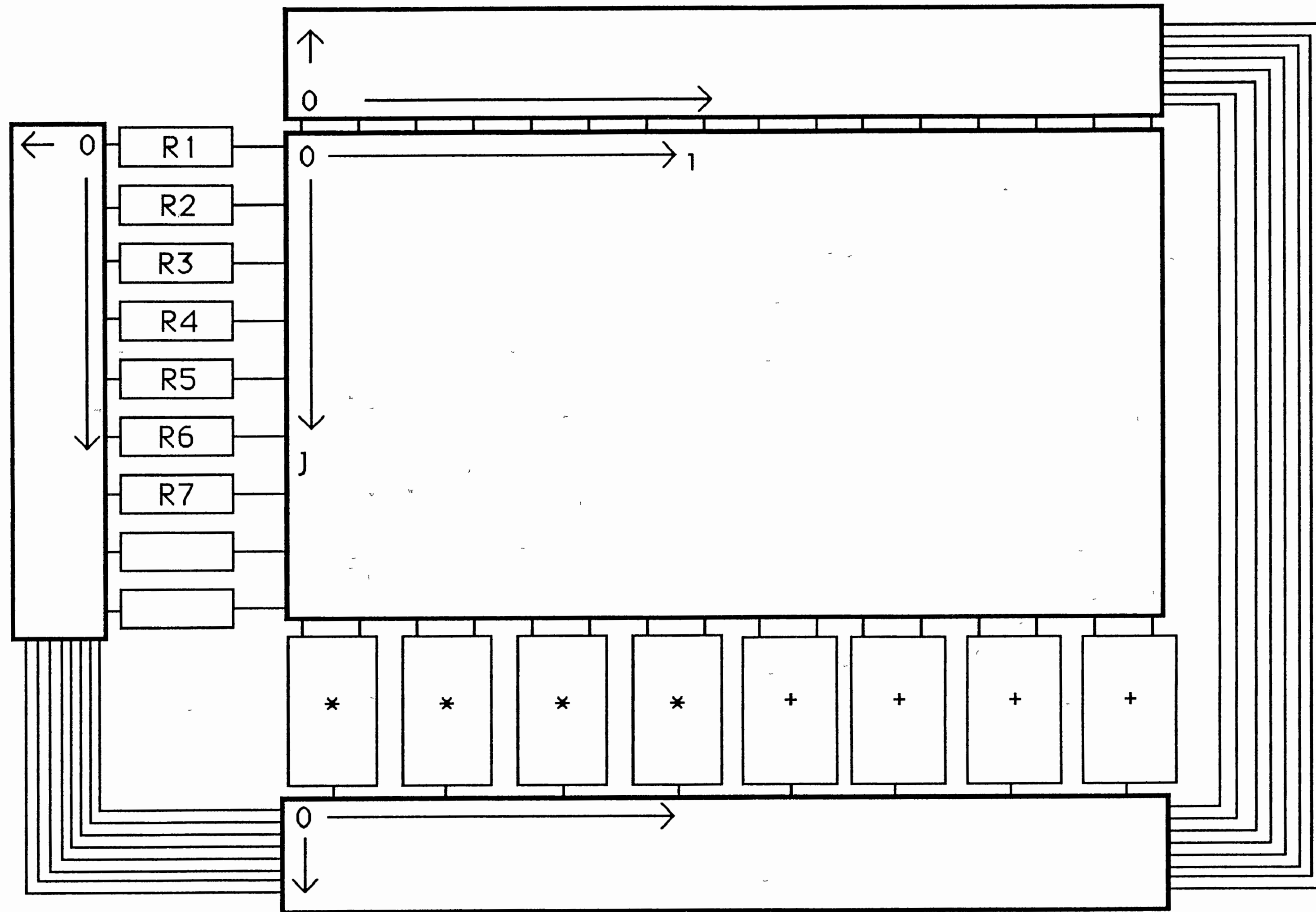


Figure 15. Architecture of an 8 Pipelined Structure with Switch Indexes

Column	0	1	2	3	4	5	6	7
Pipeline fault vector(1)	0	1	2	3				
Pipeline fault vector(2)					4	5	6	7

We will now consider two instructions *mul r1, r2, t1* and *add r6, t4, r7* for obtaining the switch settings. First consider the *mul r1, r2, t1* instruction. The virtual pipeline address assigned to this instruction is *0* which forms the column number of the pipeline fault vector(1) (because the arithmetic instruction is multiplication). The address, *0*, stored in that column is the pipeline physical address for the instruction. The virtual index *i"* generated for CBN2 is *0*, and this forms the column number of the link fault vector of CBN2; the address stored in that column is *0*, and this forms the column number for the link fault vector of CBN3. The address stored in that column is the physical index *i* for CBN2. The switch setting generated for the variables are

Variable	Switch setting	Rules
r1	(0, 0, 0)	5, 6, 7a
r2	(0, 1, 1)	5, 6, 7a
t1	(1, 0, 0)	5, 6, 7a

The switch settings generated for the rest of instructions are

Operation	Physical pipeline	Physical switch setting		
mul	1	(0, 2, 2)	(0, 3, 3)	(1, 1, 1)
add	4	(2, 0, 8)	(2, 1, 9)	(1, 2, 4)
mul	2	(0, 4, 4)	(2, 2, 5)	(1, 3, 2)

However, the instruction *add r6, t4, r7* is a different case. Since the CBN2 assignment is followed by a CBN4 assignment, rule 7b is applied. The virtual index *i"* generated for this instruction is *4*. This forms the column number of the link fault vector of CBN2. The address stored in that column forms the column number of the link fault vector of CBN4. The address stored in that column of CBN4, 4, is the physical index *i* of CBN2 and physical index *j* of CBN4. The switch settings are

Variable	Switch setting	Rules
r6	(0, 5, 10)	5, 6, 7a
t4	(2, 3, 11)	5, 6, 7a
r7	(1, 4, 5)	5, 6, 7a
	(3, 6, 4)	5, 6, 7b

Figure 16 shows the FTVP implementation.

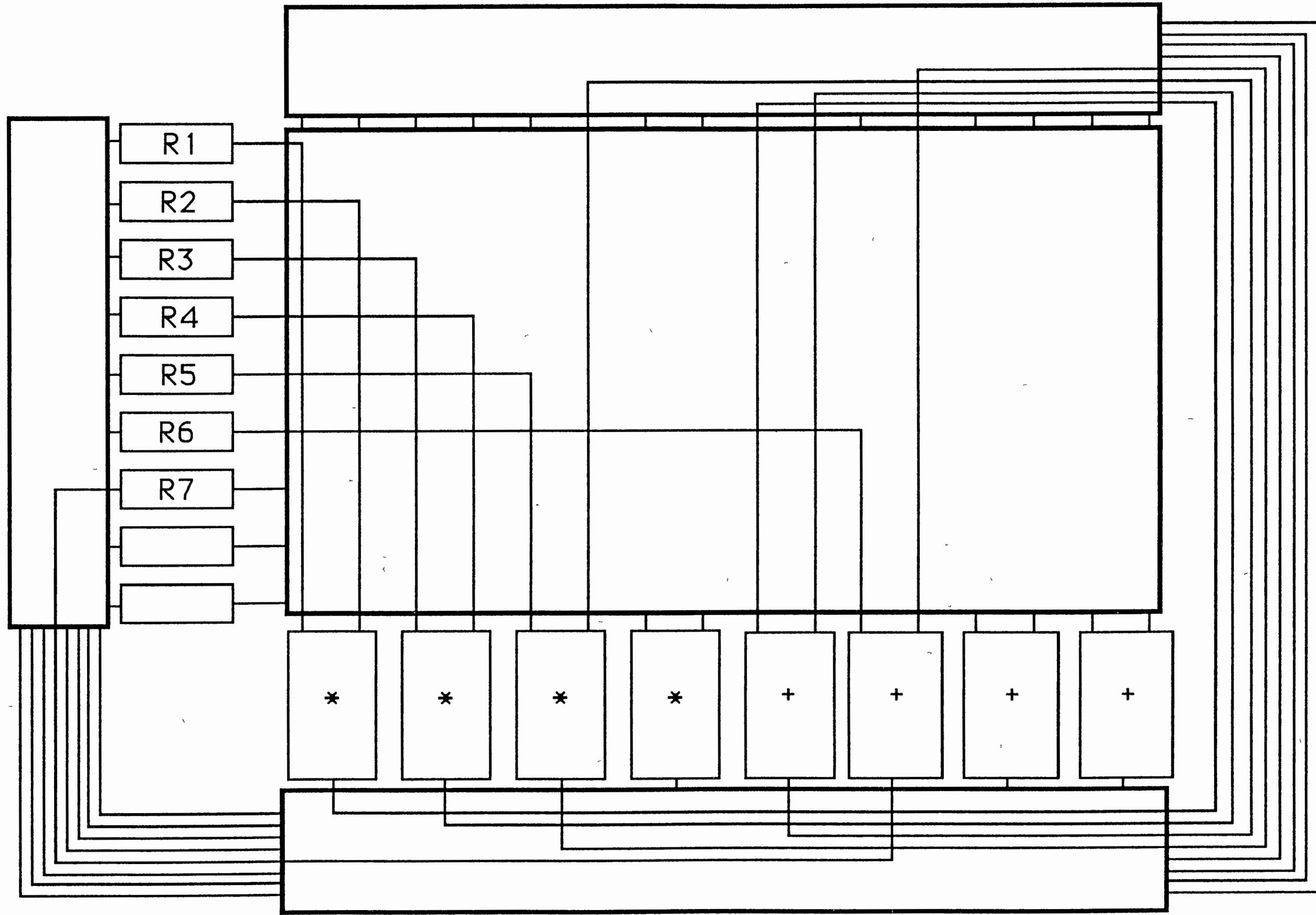


Figure 16. Routing for Livermore Loop 1 with a Fault Free Condition

Pipeline Fault Condition

Let the pipelines with physical addresses *0* and *4* be faulty. The pipeline fault vectors corresponding to this situation are

Column	0	1	2	3	4	5	6	7
Pipeline fault vector(1)	1	2	3					
Pipeline fault vector(2)					5	6	7	

The switch settings are

Virtual pipeline	Physical pipeline	Physical switch setting		
0	1	(0, 0, 2)	(0, 1, 3)	(1, 0, 1)
1	2	(0, 2, 4)	(0, 3, 5)	(1, 1, 2)
4	5	(2, 0, 10)	(2, 1, 11)	(1, 2, 5)
2	3	(0, 4, 6)	(2, 2, 7)	(1, 3, 3)
5	6	(0, 5, 12)	(2, 3, 13)	(1, 4, 6)
				(3, 6, 4)

Figure 17 shows the FTVP implementation.

Switch Fault Condition

Figure 18 shows the location of switch faults in the FTVP. Since switch faults are considered as vertical link faults, the corresponding pipelines cannot be accessed. Therefore, the pipeline fault vectors in such situation are

Column	0	1	2	3	4	5	6	7
Pipeline fault vector(1)	1	2	3					
Pipeline fault vector(2)					4	6	7	

Virtual pipeline	Physical pipeline	Physical switch setting		
0	1	(0, 0, 2)	(0, 1, 3)	(1, 0, 1)
1	2	(0, 2, 4)	(0, 3, 5)	(1, 1, 2)
4	4	(2, 0, 8)	(2, 1, 9)	(1, 2, 4)
2	3	(0, 4, 6)	(2, 2, 7)	(1, 3, 3)
5	6	(0, 5, 12)	(2, 3, 13)	(1, 4, 6)
				(3, 6, 4)

Figure 18 shows the FTVP implementation.

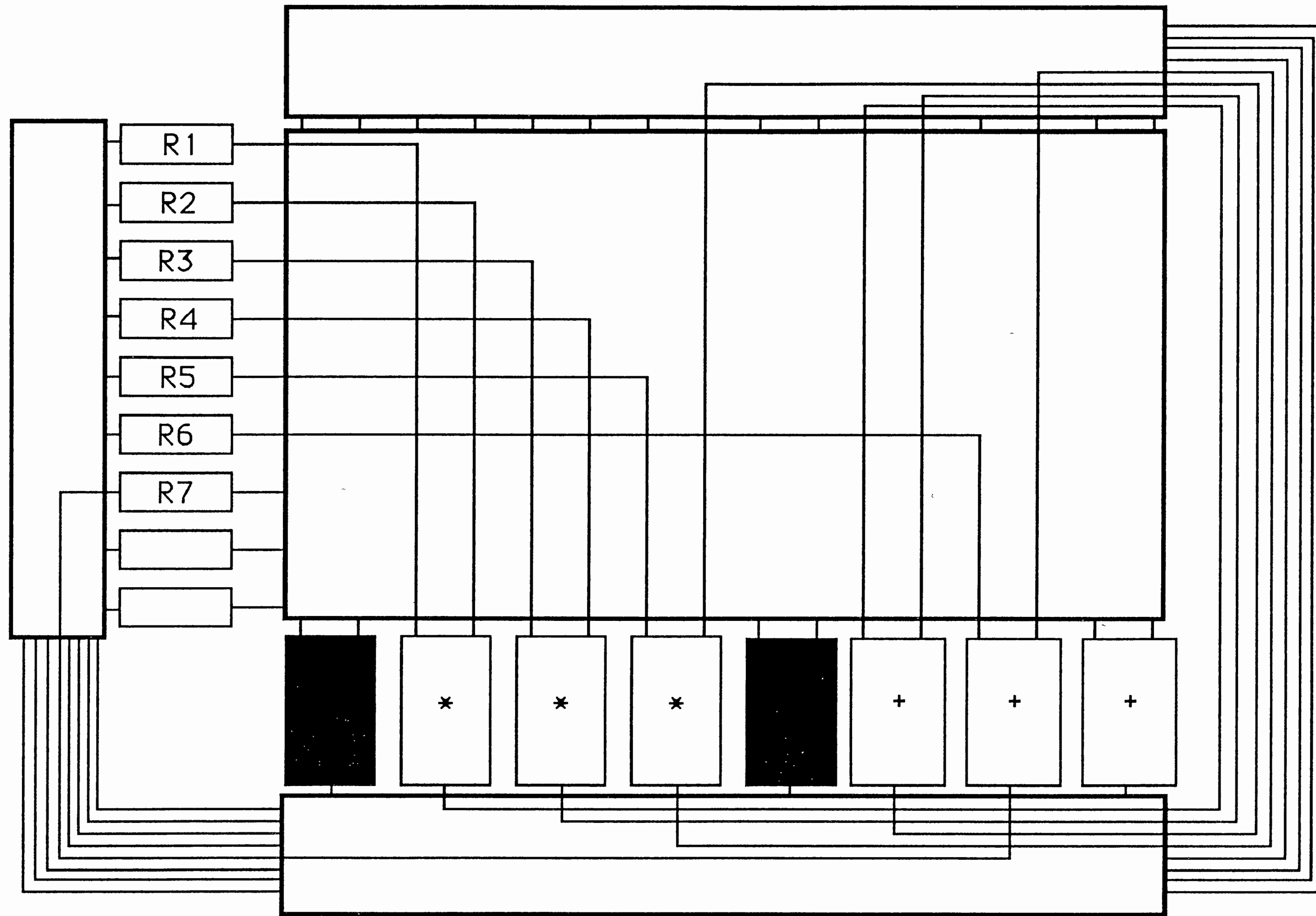


Figure 17. Routing for Livermore Loop 1 in case of a Faulty Pipelines

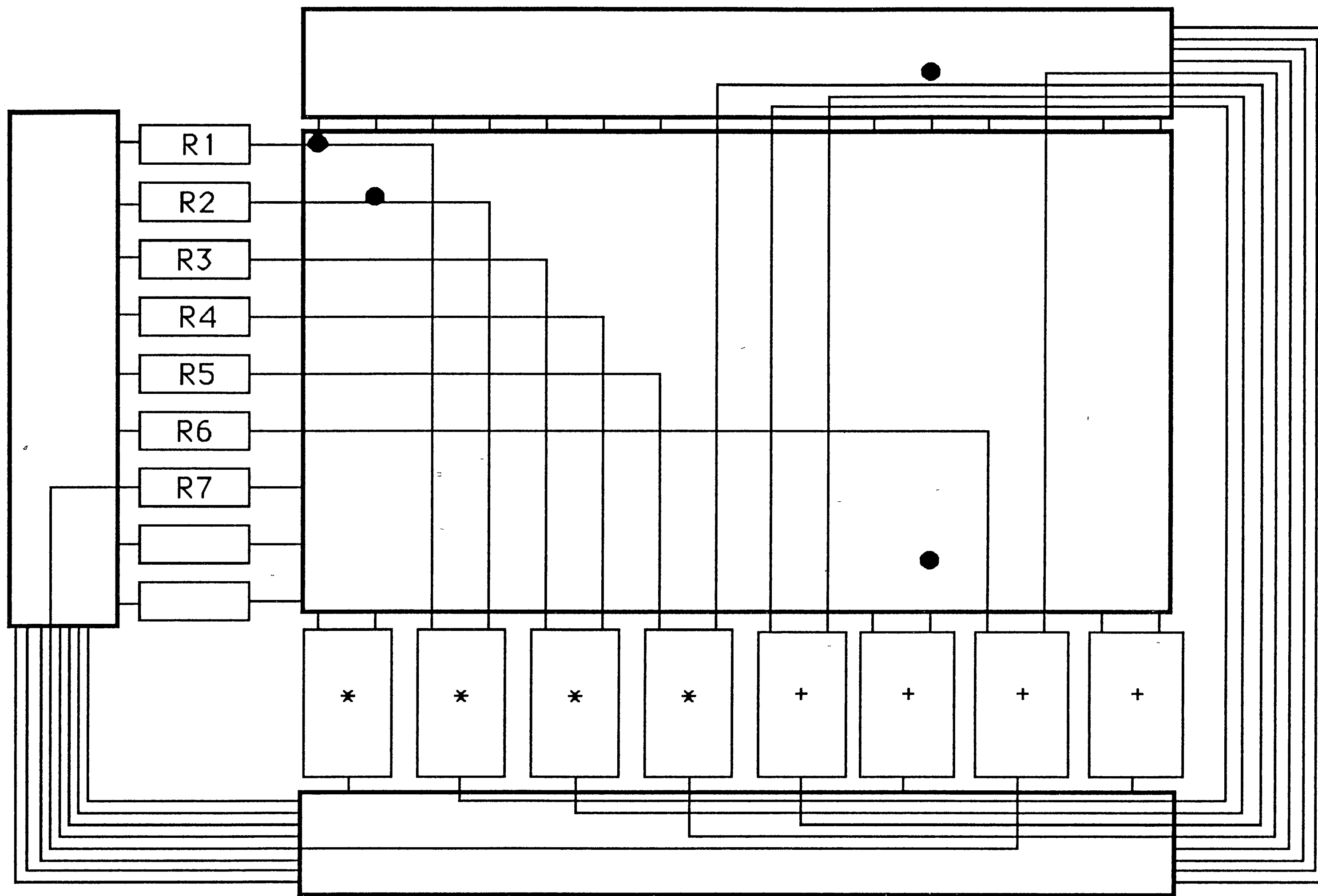


Figure 18. Routing for Livermore Loop 1 in case of Faulty Switches

Pipeline and Switch Fault

Figure 19 shows the location of faulty pipelines and switches in the FTVP.

Column	0	1	2	3	4	5	6	7
Pipeline fault vector(1)	1	2	3					
Pipeline fault vector(2)					5	6		

Virtual pipeline	Physical pipeline	Physical switch setting		
0	1	(0, 0, 2)	(0, 1, 3)	(1, 0, 1)
1	2	(0, 2, 4)	(0, 3, 5)	(1, 1, 2)
4	5	(2, 0, 10)	(2, 1, 11)	(1, 2, 5)
2	3	(0, 4, 6)	(2, 2, 7)	(1, 3, 3)
5	6	(0, 5, 12)	(2, 3, 13)	(1, 4, 6)
				(3, 6, 4)

Figure 19 shows the FTVP implementation.

Link and Switch Fault

Figure 20 shows location of faulty links and switches. The pipeline fault vectors and link fault vectors are

Column	0	1	2	3	4	5	6	7
Pipeline fault vector(1)	1	2	3					
Pipeline fault vector(2)					4	5	6	7

Column	0	1	2	3	4	5	6	7	
CBN1	0	1	2	3	4	5	6	7	Horizontal link physical address
CBN2	1	2	3	4	5	6	7		Horizontal link physical address
CBN3	0	2	3	4	5	6	7		Horizontal link physical address
CBN4	0	1	2	3	5	6	7		Vertical link physical address

The switch settings are

Virtual pipeline	Physical pipeline	Physical switch setting		
0	1	(0, 0, 2)	(0, 1, 3)	(1, 2, 1)
1	2	(0, 2, 4)	(0, 3, 5)	(1, 3, 2)
4	4	(2, 2, 8)	(2, 3, 9)	(1, 4, 4)
2	3	(0, 4, 6)	(2, 4, 7)	(1, 5, 3)
5	5	(0, 5, 10)	(2, 5, 11)	(1, 6, 5)
				(3, 6, 6)

Figure 20 shows the FTVP implementation

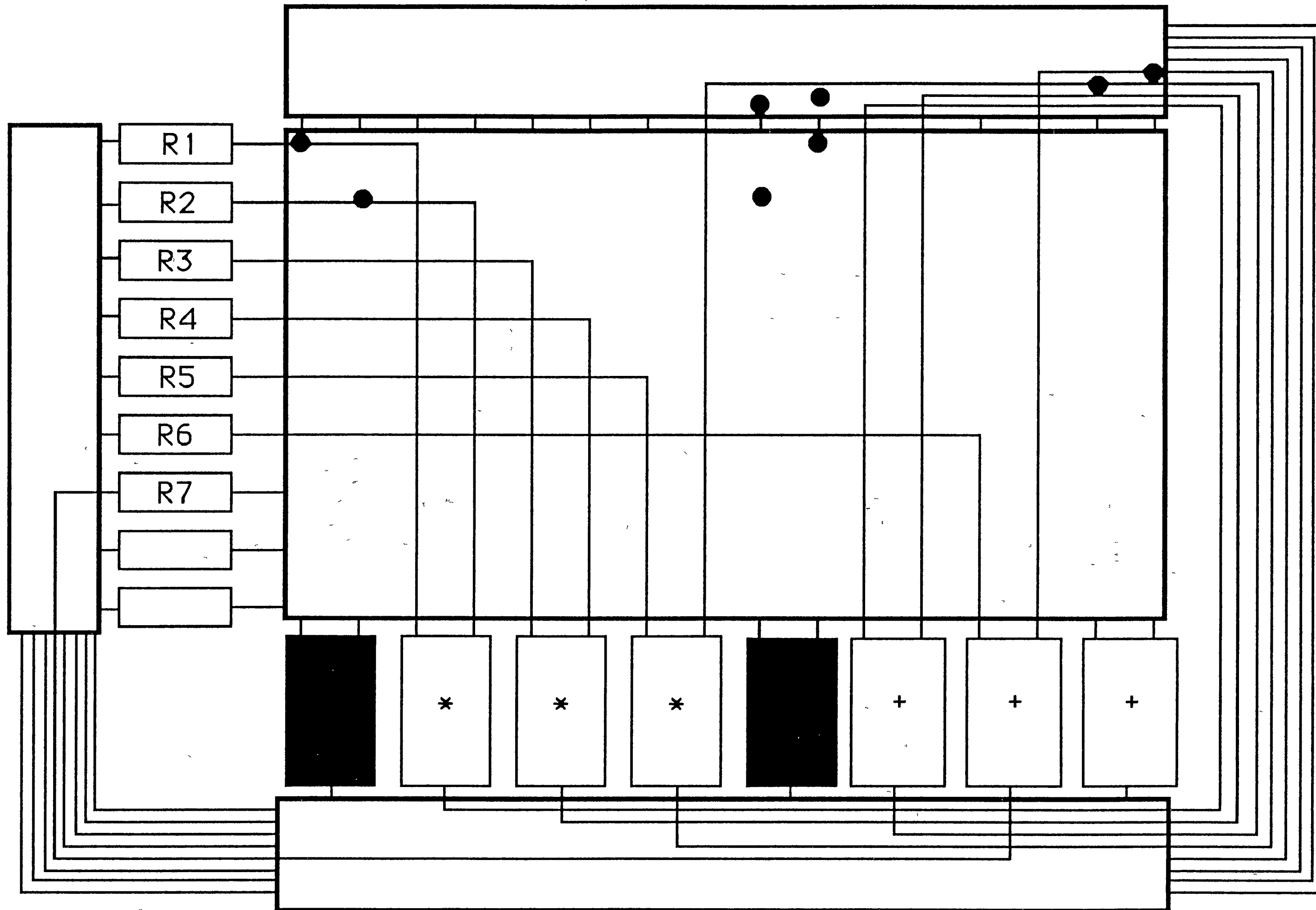


Figure 19. Routing for Livermore Loop 1 in case of Pipeline and Switch Faults

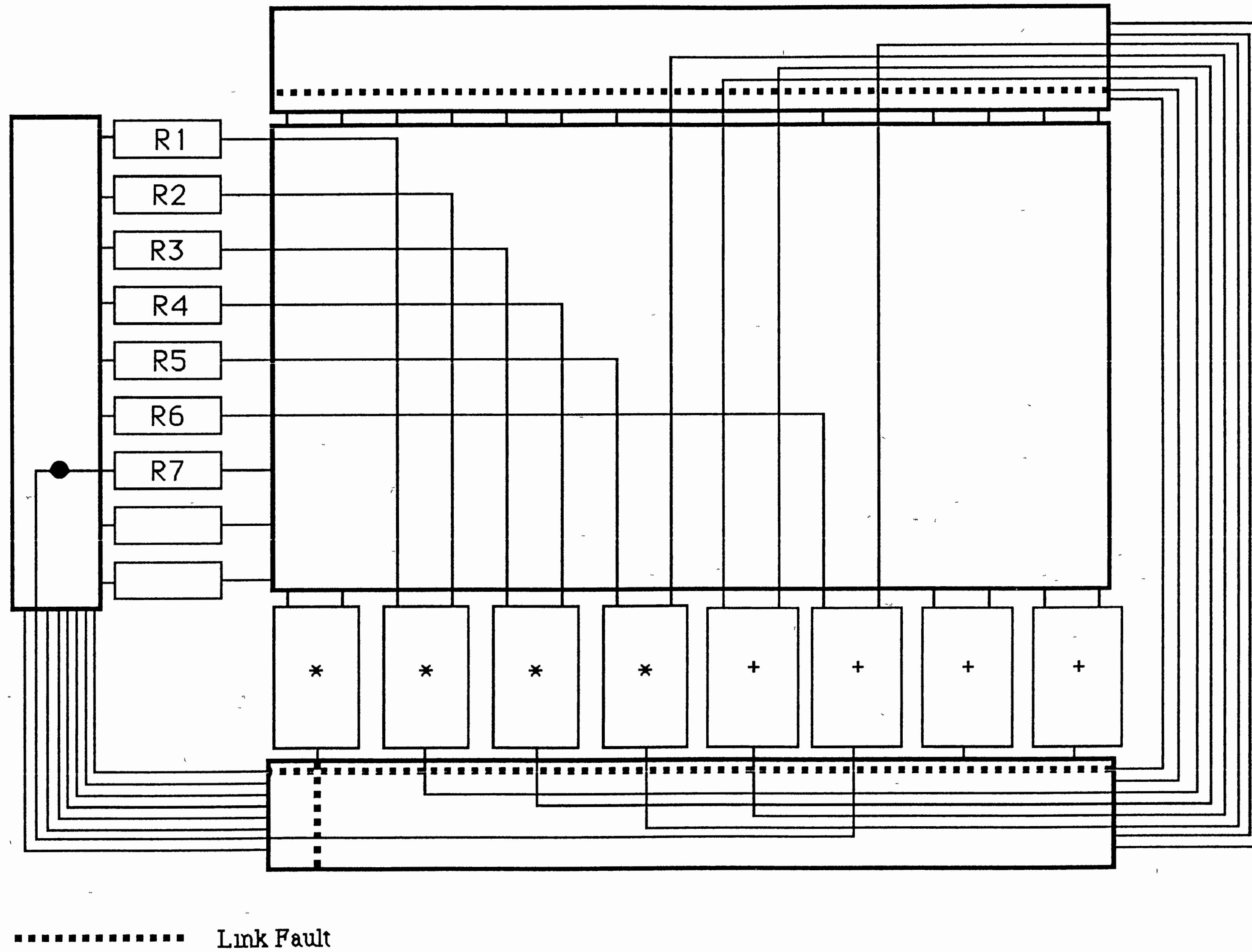


Figure 20. Routing for Livermore Loop 1 in case of Link and Switch Faults

CHAPTER IV

EVALUATION

Speedup and Throughput

Evaluation of the FTVP is accomplished by using the Livermore loops listed in Appendix A. The evaluation follows the steps taken in [20]. The time taken by a pipeline chain in evaluating an assigned vector loop is computed first. Since vectors from all the source registers of a pipeline chain emerge simultaneously, the total time taken for the evaluation of a vector loop will be equal sum of the time required to set up a pipeline chain and the time required by a vector datum to traverse through the longest path from the input to the output of this pipeline chain. The longest path is termed the *critical path*. Let S be the time taken to build a pipeline chain, C be the number of pipelines in the critical path, α be the number of stages in an interconnection network (CBN1, CBN2, CBN3 or CBN4), β be the number of stages in a pipeline of the FTVP, σ be the number of clock periods elapsed before the next element of the result vector emerges out of a pipeline chain, and N be the assigned vector loop length.

We call the parameter S the *setup time* for a pipeline chain. The setup time depends upon the time taken for decoding the arithmetic instructions corresponding to the pipeline chain, the translation procedure involved and the execution of *set* instructions to build that pipeline chain. For the calculation of setup time we need an accurate prediction of the time taken for decoding the arithmetic instructions, the execution of the translation procedure and the execution of *set* instructions. Even though we have assumed in Chapter II that each arithmetic instruction will be

decoded in one clock cycle, we do not have the exact time taken for execution of the translation procedure. We therefore consider the setup time as a function of the time taken to execute the *set* instructions only; even though this will not be an accurate setup time, it serves our purpose for the evaluation of FTVP's performance. Each arithmetic instruction requires three *set* instructions corresponding to the network assignment in step 1 of the translation procedure; the exception is when the destination variable of an arithmetic instruction is a register. We need an extra *set* instruction for the CBN4 assignment. In Chapter II we assumed that each *set* instruction is executed in one clock cycle; therefore, the setup time will be a function of the number of set instructions generated for a pipeline chain. Consider the example FORTRAN loop in which there are 5 arithmetic instructions. The number of *set* instructions for this loop are $5*3 + 1 = 16$. Since each *set* instruction is executed in one clock cycle, the setup time for this example is 16 clock cycles. It is possible to set more than one switch in a clock cycle, but we will stick to the original assumption of setting only one switch in a clock cycle.

The parameter C is the number of pipelines in the critical path. In Figure 4, $C = 4$. The latency between each element of the result vector emerging out of the pipeline chain is σ. Latency is the number of clock cycles elapsed between each result. If there is a recurrence relationship in a vector loop similar to the loops discussed in Chapter II, σ will be equal to the number of stages in a pipeline of the FTVP, because the pipeline needs to be drained before the next iteration in the loop can be started; otherwise, σ will be equal to one, as explained in Chapter II. To reduce the latency in the case of a recurrence relationship, we use the Kogge's double cycling method discussed in Chapter II [26]. But to apply the Kogge's method, the number of stages in all the pipelines of the FTVP need to be equal (for explanation refer Chapter II). To meet this requirement, we assume a pipeline similar to the Intel *i860's* pipeline [9]. The *i860* has a three-staged multiplier and a

three-staged adder. Further, to apply Kogge's double cycling method, the vector registers of the FTVP must allow simultaneous read and write operations.

We represent the time taken to evaluate the given vector loop by a pipeline chain as $T(n)$. From Figure 7 we see that a vector datum in the critical path of a pipeline chain crosses CBN1 *once* at the start of the execution phase, CBN2 and CBN3 $(C-1)$ times within the pipeline chain, and CBN2 and CBN4 *once* at the end of the execution phase. Also, the vector datum passes through a pipeline which is β staged, C times. Therefore, the time taken for the first element of the result to be written into the destination register of the pipeline chain is $(C)(\beta + 2\alpha) + \alpha$ clock periods. After the first element of the result is written into the register, the remaining elements of the result are written into the register every σ clock period. Therefore, the time taken for the N results of a pipeline chain to be written into the register is $(C)(\beta + 2\alpha) + \alpha + \sigma(N - 1)$, and the total time taken for evaluation is

$$T(n) = S + (C)(\beta + 2\alpha) + \alpha + \sigma(N - 1)$$

For the FTVP, the number of stages (α) in any interconnection network is 1. Each pipeline of the i860 has three stages; that is, $\beta = 3$. In our initial discussion in Chapter II we stated that each element of a vector is written into the vector register every clock period. Therefore, $\sigma = 1$ and

$$\begin{aligned} T(n) &= S + 1 + (C)(3 + 2) + 1(N - 1) \\ &= S + 5C + N \end{aligned}$$

In the example FORTRAN loop, $N = 400$, $S = 16$ (derived before) and $C = 4$. So, the number of clock cycles taken for evaluation of the example loop is 436 cycles. The total evaluation time for each of the Livermore loops based on the program graphs in Appendix B are shown in Table IV. We assume that the same adder can perform the required subtraction for Livermore loops, by two's complement addition through a special instruction *sub*.

TABLE IV

EVALUATION TIME OF THE LIVERMORE LOOPS

Loop Number	Number of arithmetic instructions	C	σ	S	T(n)
1	5	4	1	16	436
2	10	5	3	31	654
3	2	2	3	7	3015
4[1]	3	2	3	11	532
5[1,2]	10	10	1	33	416
6[1]	8	6	3	28	1055
7	16	7	1	49	209
9[1]	15	5	1	46	171
10[1,3]	9	9	1	46	1091
11[1,2]	3	2	1	10	1019
12	1	1	1	4	208

In a vector loop that has recurrence relationship, we cannot achieve less evaluation time by dividing the vector loop into smaller loops and isolating the recurrent portion of the loop. This can be demonstrated by an example. Consider the case of Livermore loop 2 which is divided into two loops as

<u>Step 1</u>:

```
      DO 2 K = 1, 996, 5
2       T(K) = Z(K)*X(K)+Z(K+1)*X(K+1)+Z(K+2)*X(K+2)+
        Z(K+3)*X(K+3)+Z(K+4)*X(K+4)
```

<u>Step 2</u>:

```
      DO 3 K = 1, 996, 5
3       Q = Q+T(K)
```

The σ for step 1 is "*1*" as there is no recurrence relationship. The σ for step 2 is "*3*" because there is a recurrence relationship. Each element of the input vector, *T(k)*, to the adder has to be delayed by the number of clock cycles equal to the number of stages in a pipeline of the FTVP for step 2. The total execution time for step 1 is *268* cycles, and for step 2 is *607* cycles. The total execution time for Livermore loop 2 is *875* cycles, as opposed to *654* cycles shown in Table IV, because of the delay caused by step 2. Therefore, it is more advantageous to build a long pipeline chain

1 Can be executed only if the registers allow simultaneous read and write
2 Modified by Kogge's double Cycling Method
3 Has to be divided into small vector loops

to evaluate a large vector loop than few short pipeline chains to evaluate small vector loops. But a long pipeline chain may result in the use of large buffers. Therefore, a trade-off has to be done in selecting the maximum allowable buffer size against the evaluation time.

The next step is to evaluate the FTVP's *throughput*, which is defined as the ratio of the total number of arithmetic operations performed by the FTVP to the total time taken for performing these operations. Let M be the total number of pipelines present in an FTVP; this implies that the total number of arithmetic operations performed by the FTVP is MN. Therefore, throughput of the FTVP is

$$H_m = \frac{MN}{S + 5C + N}$$

Two parameters have been proposed in [30] to measure the throughput performance of a vector processor. H_α is the maximum throughput obtained when N approaches infinity, and $N_{1/2}$ is the minimum vector length needed to obtain half the maximum throughput. For the FTVP, $H_\alpha = M$ and $N_{1/2} = S + 5C$; this implies that, in order to achieve the highest throughput (H_α), all the pipelines in the FTVP have to be utilized. The initial time to set up a pipeline chain has to be minimized in order to achieve at least half the maximum throughput for short vectors. By limiting the number of *set* instructions and executing more than one *set* instruction at a time, the initial setup time can be minimized. Figure 21 shows the results of the throughput analysis of the FTVP for various vector loop lengths with a constant number of pipelines. While calculating the setup time for Figure 21, we assumed M arithmetic instructions for M pipelines so that the highest throughput is achieved. The number of pipelines in the critical path (C) is 4 (except for $M = 2$ where $C = 2$). The results of Figure 21 demonstrates that for large vector loop lengths, the throughput is constant.

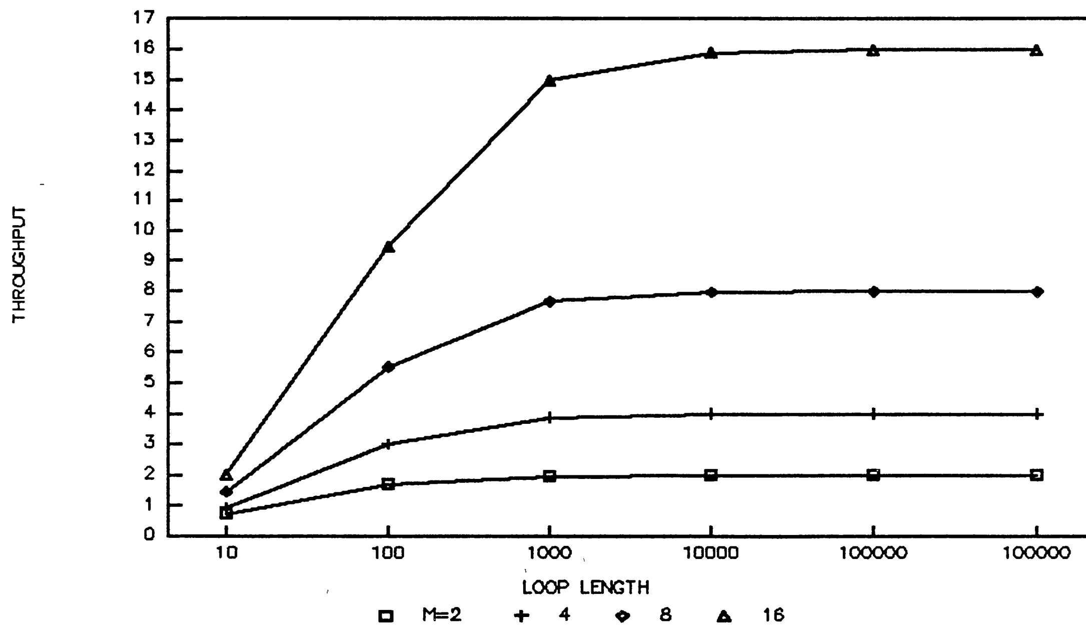


Figure 21. Loop Length vs. Throughput

The next step is to evaluate the relative speedup of the time taken for evaluation of a vector loop using M pipelines in the FTVP over the time taken for evaluation of that vector loop using a single pipeline in the FTVP. A pipeline in the FTVP can evaluate in two modes; the first one is the vector mode when there is no input-output recurrence relationship in the assigned vector loop. The second one is the scalar mode when there is a recurrence relationship in the assigned vector loop. In the FTVP, each vector element emerging from a pipeline in the scalar mode will have a latency which is equal to the number of stages in the pipeline, as the pipeline has to be drained before the next input element can be sent to the pipeline. Therefore, the time taken for evaluation by a pipeline in the scalar mode is $T_S = \beta N = 3N$. The time taken for evaluation by a pipeline in the vector mode is $T_V = S1 + 3\alpha + \beta + N - 1$, as the datum passes through CBN1, CBN2, CBN4 and a pipeline *once*, before the results are written into a register. $S1$, the setup time for a single pipeline, for which we need four set instructions (*2* for CBN1, *1* for CBN2,

and *1* for CBN4), is *4*. Moreover, for the FTVP $\alpha = 1$ and $\beta = 3$, resulting in $T_V = N + 9$. If we need M_1 number of pipelines in the scalar mode, and M_2 number of pipelines in the vector mode, the time needed to evaluate the vector loop by a single pipeline is $T(1) = T_S M_1 + T_V M_2 = 3M_1 N + M_2(N + 9)$.

Speedup (*Sp*) of the FTVP is the ratio of the time taken for evaluation of a vector loop by a single pipeline, to the total time taken for the evaluation of a vector loop in a pipeline chain. Therefore,

$$Sp = \frac{T(1)}{T(n)} = \frac{3M_1 N + M_2(N + 9)}{S + 5C + N}$$

Figure 22 shows the speedup analysis of the FTVP for various vector loop lengths with a constant R (where $R = M_1/M$ and $M = M_1 + M_2$). The figure shows that the speedup is constant for large vector loop lengths. We assumed that the number of pipelines in the critical path (C) is 4, and the total number of pipelines (M) in the FTVP is 16 for the generation of Figure 23.

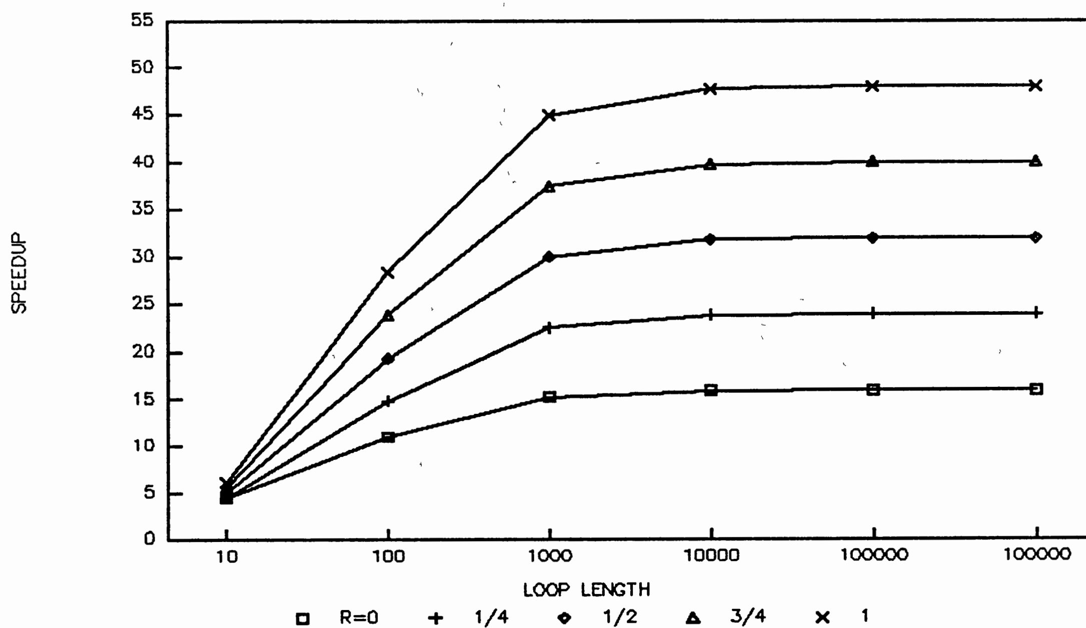


Figure 22. Speedup vs. Loop Length

Figure 23 shows the speedup analysis for the Livermore loops that have no input-output recurrence relationship, and Figure 24 shows the speedup analysis for the Livermore loops that have input-output recurrence relationship. In calculating the speedup, we do these following steps. Consider the evaluation of Livermore loop 1 in an FTVP with one adder pipeline and one multiplier pipeline; that is, $M = 2$. $M = M_2$ as there is no recurrence relationship in this loop. Evaluation of the loop is divided into three steps, as there are only two pipelines in the FTVP. The three steps are

Step 1: 1 multiplication. $T_1(N) = 4+400+5 = 409$

Step 2: 1 multiplication and 1 addition. $T_2(N) = 7+400+10 = 417$

Step 3: 1 multiplication and 1 addition. $T_3(N) = 417$

$T(n) = T_1(N) + T_2(N) + T_3(N) = 1243$

Therefore, $Speedup\ (Sp) = 2*(400 + 9)/1243 = 1.93$

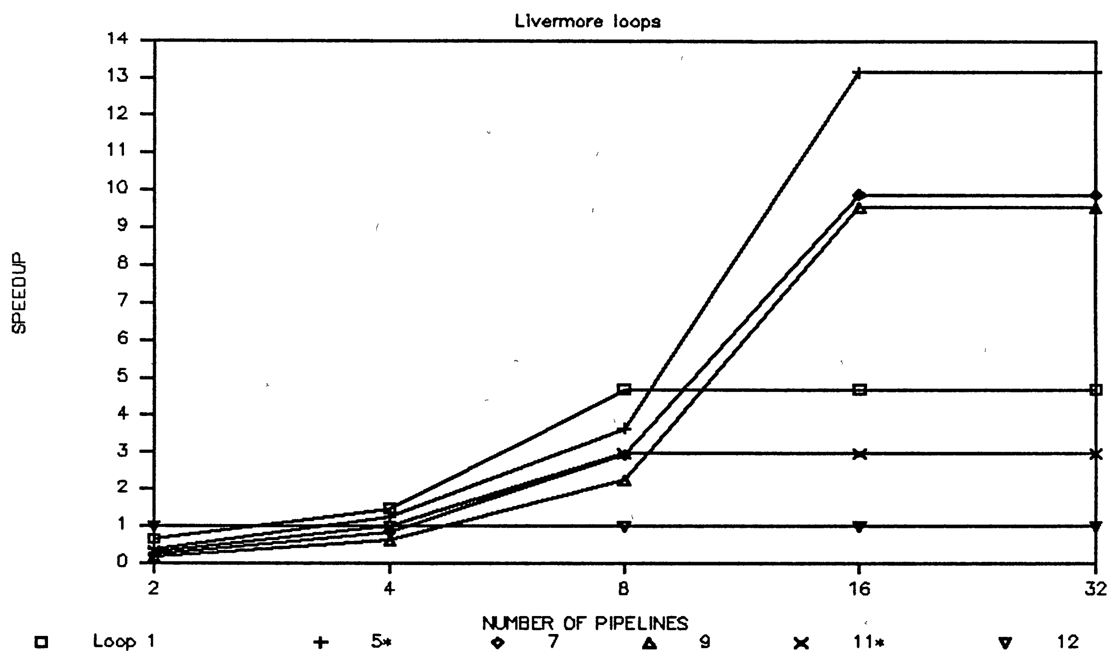


Figure 23. Speedup vs. Loop Length for Livermore Loops with No Recurrence Relationship

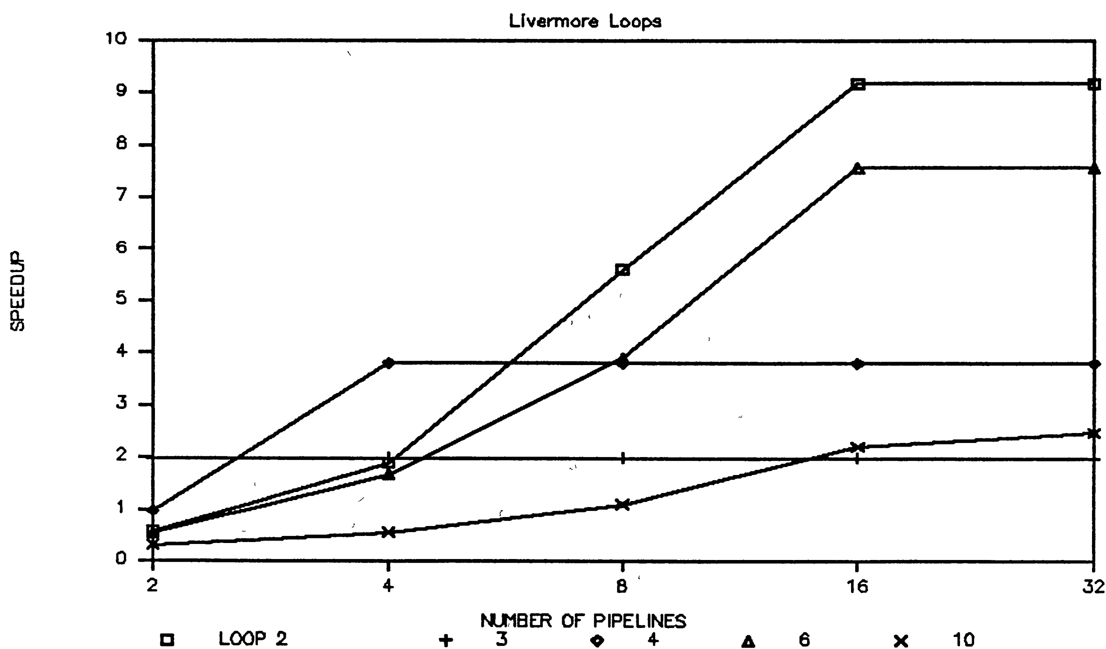


Figure 24. Speedup vs. Loop Length for Livermore Loops with Recurrence Relationship

Proposed Architecture

As a final step in the evaluation we propose an architecture of the FTVP based on the Livermore loop analysis. This architecture is proposed to keep the number of interconnection links and crossbar switches in the FTVP to a minimum. The links in the FTVP are laid out using good conduction lines on the semiconductor wafer. These links occupy much of the chip space. Therefore, the number of links must be kept to a minimum. Further, reducing the number of links reduces the number of crossbar switches in the interconnection networks, which further conserves the chip space. For example, since most of the vector operations send only their results to the registers, the number of links connecting CBN2 and CBN4 can be reduced; this leads to a reduction in the number of crossbar switches in CBN2 and CBN4.

Figure 25 shows the program graph for Livermore loop 1. An FTVP hardware with 5 pipelines, 4 pipeline-pipeline links and 1 pipeline-register link is sufficient for evaluation of this loop. Similar kind of results are obtained for the other Livermore loops as shown in Table V.

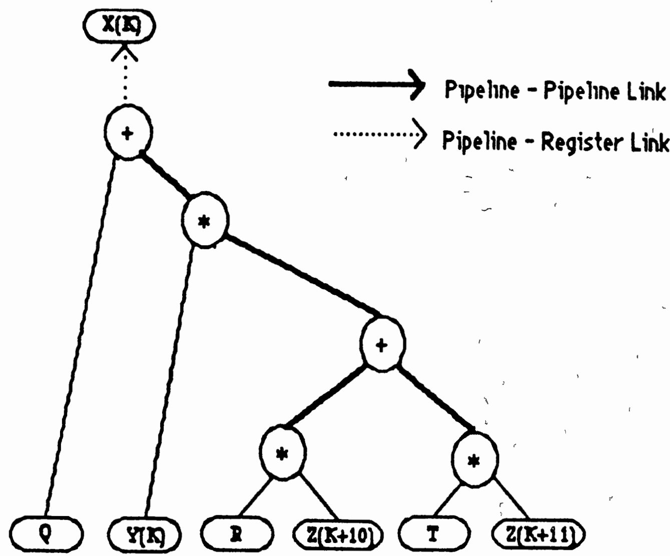


Figure 25. Program Graph for Livermore Loop 1

TABLE V

LIVERMORE LOOP PARAMETERS

Loop number	Number of Pipelines Mul	Add	Pipe-Pipe links	Pipe-Register link	Register-Register
1	3	2	4	1	0
2	5	5	9	1	0
3	1	1	1	1	0
4[4]	1	2	1	2	0
5[4,5]	5	5	9	3	0
6[4]	3	5	6	4	0
7	8	8	15	1	0
9[4]	7	8	14	1	0
10[4]		9	8	9	10
11[4,5]		3	2	1	1
12		1	0	1	0

4 Can be executed only if the registers allow simultaneous read and write
5 Modified by Kogge's double Cycling Method

As seen from Table V, an FTVP hardware with 16 pipelines (8 adders and 8 multipliers), 15 pipeline-pipeline links, 3 pipeline-register links and 4 register-register links is sufficient for evaluation of most of the Livermore loops. These requirements are incorporated into the proposed architecture shown in Figure 26.

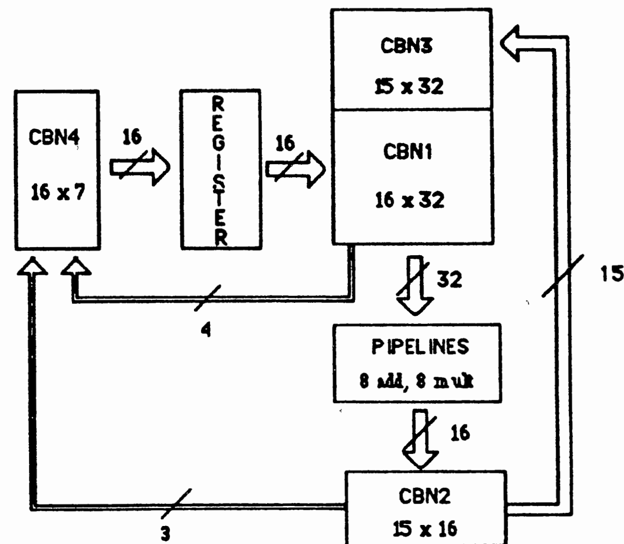


Figure 26. Hardware of the Recommended Structure

From Table V we can see that Livermore loop 10 is an exception that has to be handled by the proposed hardware in Figure 26. In such cases the original vector loop has to divided into smaller vector loops and evaluated one by one. Certain criteria for handling these exceptional situations have been established.

Criterion 1 for the division of a big vector loop is to divide according to the number of destination registers and register-register transfers required in the vector loop. Criterion 2 is to divide the vector loop according to the number of pipelines available in the FTVP. The register-register transfer instructions that can be moved outside the loop by rearranging the original vector loop and executed independently, are done so; this is criterion 3. We can see that by rearranging the instructions of

Livermore loop 10, some of the register-register transfer instructions can be moved outside the loop and performed independently. The rearranged loop is

```
NO. 10      DO 10 I = 1, 100
            AR = CX(5,I)
            BR = AR-PX(5,I)
            CR = BR-PX(6,I)
            PX(5,I) = AR
            PX(6,I) = BR
            AR = CR-PX(7,I)
            BR = AR-PX(8,I)
            PX(7,I) = CR
            PX(8,I) = AR
            CR = BR-PX(9,I)
            AR = CR-PX(10,I)
            PX(9,I) = BR
            PX(10,I) = CR
            BR = AR-PX(11,I)
            CR = BR-PX(12,I)
            PX(14,I) = CR-PX(13,I)
            PX(11,I) = AR
            PX(12,I) = BR
            PX(13,I) = CR
     10     CONTINUE
```

The instructions AR = CX(5,I), PX(11,I) = AR, PX(12,I) = BR, and PX(13,I) = CR can be moved outside the loop. The register-register transfer involved in AR = CX(5,I) should be done before commencement of the pipeline chain operation, and the other three instructions should be done after the pipeline chain operation. The following is the vector loop after the four instructions were moved outside the loop.

```
NO. 10      DO 10 I = 1, 100
            BR = AR-PX(5,I)
            CR = BR-PX(6,I)
            PX(5,I) = AR
            PX(6,I) = BR
            AR = CR-PX(7,I)
            BR = AR-PX(8,I)
            PX(7,I) = CR
            PX(8,I) = AR
            CR = BR-PX(9,I)
            AR = CR-PX(10,I)
            PX(9,I) = BR
            PX(10,I) = CR
            BR = AR-PX(11,I)
            CR = BR-PX(12,I)
            PX(14,I) = CR-PX(13,I)
     10     CONTINUE
```

Since the architecture proposed in Figure 26 has only 4 register-register links, applying criterion 1 to the above instructions, Livermore loop 10 is divided into two steps as shown below.

Step 1:

```
NO. 10      DO 10 I = 1, 100
            BR = AR-PX(5,I)
            CR = BR-PX(6,I)
            PX(5,I) = AR
            PX(6,I) = BR
            AR = CR-PX(7,I)
            BR = AR-PX(8,I)
            PX(7,I) = CR
            PX(8,I) = AR
            CR = BR-PX(9,I)
       10   CONTINUE
```

Step 2:

```
NO. 10      DO 10 I = 1, 100
            AR = CR-PX(10,I)
            PX(9,I) = BR
            PX(10,I) = CR
            BR = AR-PX(11,I)
            CR = BR-PX(12,I)
            PX(14,I) = CR-PX(13,I)
       10   CONTINUE
```

Therefore, Livermore loop 10 can be evaluated in two steps by the proposed architecture in Figure 26. In obtaining the two steps for the evaluation of Livermore 10, we had assumed that there are no faulty modules in the architecture proposed in Figure 26. If faulty modules are present, then the number of hardware resources available for processing will be less; in such a case the original loop has to be further subdivided. It is the duty of the compiler to identify these kinds of exceptional situations, and divide the assigned vector loop applying the three criterion.

CHAPTER V

CONCLUSIONS AND FUTURE WORK

A vector processor that can be fabricated as a single-chip processor using the WSI fabrication technique was designed in this thesis. The chaining capability of the vector processor was demonstrated using the Livermore loops. A basic instruction set and the translation procedure for the vector processor was developed, a speedup analysis was done, and the fault-tolerant capability of the vector processor was demonstrated. Based on the Livermore loop analysis, a hardware structure was recommended for fabrication. A method of handling large vectors by this proposed hardware was also discussed.

As seen from the Livermore loop analysis, an FTVP with 16 pipelines is sufficient for most practical problems. If more pipelines are required, the problem to be executed can be broken down into many small vector loops and executed one by one, or the number of pipelines in the FTVP can be increased.

Further study needs to be done on providing multiple pipeline chains in a single FTVP. An *intelligent compiler* needs to be developed to implement the proposed translation procedure. The vector registers of the FTVP need to be expanded to store more than one vector datum, to allow irregular vector accesses for complex vector applications, and to allow simultaneous read and write operations. In conclusion, the FTVP provides an efficient dynamic chaining and fault-tolerance capability. The fault-tolerant capability, along with the WSI fabrication, paves the way for an efficient, single-chip processor. The FTVP can be used for applications like wave equations, heat transfer and signal processing.

REFERENCES

[1] Kai Hwang and Faye A. Briggs, *"Computer Architecture and Parallel Processing"*, McGraw-Hill series, 1984.

[2] Douglas I. Thesis, *"Vector Supercomputers"*, Computer, Volume 4, Number 7, pp 52 - 61, 1974.

[3] Olaf Lubek, *"Supercomputers Performance: The Theory, Practices and Results"*, Advances in Computers, pp 308 -360.

[4] Rod A. Fatoohi, *"Vector Performance Analysis of NEC SX-2"*, Computer Architecture News, Volume 18, Number 3, pp 389-400, Sep 1990.

[5] Kai Hwang, *"Advanced Parallel Processing with Supercomputer Architectures"*, Proceedings IEEE, Volume 75, Number 10, October 1987.

[6] Olaf Lubek, James Moore and Raul Mendez, *"A Benchmark Comparison of Three Supercomputers: Fujitsu VP-200, Hitachi S810/20, Cray X-MP/2"*, Computer, pp 10 - 23, December 1985.

[7] Tom Diede, Carl F. Hagen Maier, Glen S. Miranker, Jonathan J. Rubinstein and William S. Worley, Jr. , *"The Titan Graphics Supercomputer Architecture"*, Computer, pp 13 - 25, September 1988.

[8] Norman P. Jouppi, Jonathan Bertoni and David W. Wall, *"A Unified Vector/Scalar Floating-Point Architecture"*, Computer Architecture News, Volume 17, Third International Conference on ABPLOS, April 3 - 6, pp 134 - 143, 1989.

[9] Les Kohn and Neal Margulis, *"Introducing i860 64 Bit Microprocessor"*, IEEE Micro, Volume 10, pp 15 - 29, August 1989.

[10] Christos J. Georgiou, *"Fault tolerant Crosspoint Switching Networks"*, Proceedings of 14th International Symposium on Fault-Tolerant Computing, pp 240 - 245, 1984.

[11] W. Chen, Prof. J. Mavor, Prof. P. B. Denyer and D. Renshaw, *"Superchip Architecture for Implementing Large Integrated Systems"*, IEE Proceedings, Volume 135, Number 3, pp 137 - 150, May 1988.

[12] Jack F. McDonald, Hans J. Greub, Randy H. Steinvorth, Brain J. Donlan and Albert S. Bergendahl, *"Wafer Scale Interconnection for GaAs Packaging - Applications To RISC Architecture"*, Computer, pp 21-35, April 1987.

[13] Gilman Chesely, *"WSI Architecture"*, Computer, V-17, pp 94-5, November 1984.

[14] Martin Gold, *"Wafer Scale Integration is Still a Challenge to Design, Fabricate, Test*, Electronic Design, v32, pp 87, May 3, 1984.

[15] Daniel P. Siewierek, *"Fault Tolerance in Commercial Computers"*, Computer, pp 26 - 37, Volume 23, Number 7, July 1990.

[16] Adit D. Singh and Singaravel Murugesan, *"Fault-Tolerant Systems"*, Computer, Volume 23, Number 7, pp 15 - 17, July 1990.

[17] Rajiv Gupta, Alessandro Zorat and I. V. Ramakrishnan, *"Reconfigurable Multipipelines for Vector Supercomputers"*, IEEE Transactions on Computer, Volume 38, Number 9, pp 1297-1307, September 1989.

[18] William F. Bruckert and Ronald E.Josephson, *"Designing Reliability into VAX 8600 Pipeline"*, Digital Technical Journal, pp 71 - 77, Number 1, August 1985.

[19] Werner Buchholz, *"The IBM System/370 Vector Architecture"*, IBM Systems Journal, Volume 25, Number 1, pp 51 - 62, 1986.

[20] Kai Hwang and Zhiwei Xu, *"Multipipeline Networking for Compound Vector Processing"*, IEEE Transactions on Computer, Volume 37, Number 1, January 1988.

[21] John P. Riganati and Paul B. Schneck, *"Supercomputing"*, Computer, Volume 17, Number 10, pp 91 - 113, Oct. 84.

[22] George B. Adams and Howard J. Siegely, *"Extra Stage Cube: A Fault Tolerant Interconnection Network for Super Systems"*, IEEE Transactions on Computer, Volume c-31, Number 5, pp 443-454, May 1982.

[23] Nirependra N. Biswas, S. Srinivas and T. Dharanendra, *"A Centrally Controlled Shuffle Network for Reconfigurable and Fault Tolerant Architecture"*, Computer Architecture News, Volume 15, Number 1, pp 81-87, March 1987.

[24] George B. Adams III, Dharma P. Agrawal and Howard J. Seigel, *"Fault Tolerant Multistage Interconnection Network"*, Computer, pp 14 -27, June 1987.

[25] David A. Patterson, *"Reduced Instruction Set Computers"*, Communications of ACM, Volume 28, Number 1, pp 8 - 21, January 1985.

[26] Peter M. Kogge, *"The Architecture of Pipelined Computers"*, New York: McGraw Hill, 1981.

[27] Dharma P. Agrawal, *"Testing and Fault Tolerance of Multistage Interconnection Networks"*, Computer, pp 41 -53, April 1982.

[28] Tse - yun Feng and Chuan - Lin Wu, *"Fault Diagnosis for a Class of Multistage Interconnection Network"*, IEEE Transactions on Computer, Volume c-30, Number 10, pp 743-758, October 1981.

[29] Israel Koren, *"Defect and Fault Tolerance In VLSI Systems"*, Volume 1, Plenum Press, 1989.

[30] R. W. Hockney and C. R. Jesshope, "*Parallel Computer*", Bristol: Adam Hilger, 1981.

[31] A. J. Blogett and D. R. Barbour, "*Thermal Conduction Module: A High-Performance Multilayer Ceramic Package*", IBM Journal of Research and Development, Volume 26, Number 1, pp 30-36, January 1982.

[32] Alessandro De Gloria, "*VISA: A Variable Instruction Set Architecture*", Computer Architecture News, Volume 18, Number 2, pp 76-84, June 1990.

[33] Robert P. Colwell, Robert P. Nix, John J. O' Donnell, David B. Papworth and Paul K. Rodman, "*A VLIW Architecture for a Trace Scheduling Compiler*", Computer Architecture News, Volume 15, Second International Conference on ASPLOS, pp 180-192, 1987.

[34] Tse - yun Feng, "*A Survey of Interconnection Network*", Computer, pp 12-27, December 1981.

[35] John P. Hayes, "*Computer Architecture and Organization*", McGraw-Hill series, 1988.

[36] Norman P. Jouppi, "*Superscalar vs. Superpipelined Machines*", Computer Architecture News, Volume 16, Number 3, pp 71-80, June 1988.

[37] David A. Patterson and Carlo H. Sequim, "*A VLSI RISC*", Computer, pp 8 - 20, September 1982.

[38] Dileep Bhandarkar and Richard Brunner, "*Vax Vector Architecture*", Computer Architecture News, Volume 18, Number 2, pp 204 - 215, June 1990.

[39] Richard O. Carlson and Constantine A. Neugebauer, "*Future Trends in Wafer Scale Integration*", Proceedings of the IEEE, Volume 74, Number 12, pp 1741-1751, December 1986.

[40] Koichi Yamashita, Akinori Kanasugi, Shinpei Hijiya, Gensuke Goto, Nobutake Matsumura and Takehide Shirato, "*A Wafer-Scale 170 000-Gate FFT Processor with Built-In Test Circuits*", IEEE Journal of Solid-State Circuits, Volume 23, Number 2, pp 336-341, April 1988.

APPENDIX A

LIVERMORE LOOPS

No. 1
```
      DO   1     K = 1, 400
1     X(K) = Q+Y(K)*(R*Z(K+10)+T*Z(K+11))
```

No. 2
```
      DO 2 K = 1, 996, 5
2     Q = Q+Z(K)*X(K)+Z(K+1)*X(K+1)+Z(K+2)*X(K+2)+
            Z(K+3)*X(K+3)+Z(K+4)*X(K+4)
```

No. 3
```
      DO 3 K = 1, 1000
3     Q = Q+Z(K)*X(K)
```

No. 4
```
      DO 4 J = 30, 870, 5
      X(L-1) = X(L-1)-X(LW)*Y(J)
4     LW = LW+1
```

No. 5
```
      DO 5 I = 2, 998, 3
      X(I) = Z(I)*(Y(I)-X(I-1))
      X(I+1) = Z(I+1)*(Y(I+1)-X(I))
5     X(I+2) = Z(I+2)*(Y(I+2)-X(I+1))
```

No. 6
```
      DO 6 J = 3, 999, 3
      I = 1000-J+3
      X(I) = X(I)-Z(I)*X(I+1)
      X(I-1) = X(I-1)-Z(I-1)*X(I)
6     X(I-2) = X(I-2)-Z(I-2)*X(I-1)
```

No. 7
```
      DO 7 M = 1, 120
      X(M) = U(M)+R*(Z(M)+R*Y(M))+T*(U(M+3)+R*(U(M+2)
            +R*U(M+1))+T*(U(M+6)+R*(U(M+5)+R*U(M+4))))
```

No. 8
```
      DO 8 KX = 2, 3
      DO 8 KY = 2, 21
      DU1 = U1(KX,KY+1,NL1)-U1(KX,KY-1,NL1)
      DU2 = U2(KX,KY+1,NL1)-U2(KX,KY-1,NL1)
      DU3 = U3(KX,KY+1,NL1)-U3(KX,KY-1,NL1)
      U1(KX,KY,NL2) = U1(KY,NL1)+A11*DU1+A12*DU2
            +A13*DU3+SIG*(U1(KX+1,KY,NL1)-2*U1(KX,KY,
            NL1)+U1(KX-1,KY,NL1))
      U2(KX,KY,NL2) = U2(KY,NL1)+A21*DU1+A22*DU2
            +A23*DU3+SIG*(U1(KX+1,KY,NL1)-2*U1(KX,KY,
            NL1)+U1(KX-1,KY,NL1))
      U3(KX,KY,NL2) = U3(KY,NL1)+A31*DU1+A32*DU2
            +A33*DU3+SIG*(U1(KX+1,KY,NL1)-2*U1(KX,KY,
```

```
              NL1)+U1(KX-1,KY,NL1))
      8    CONTINUE

No. 9      DO 9 I = 1, 100
           PX(1, I) = BM28*PX(13,I)+BM27*PX(12,I)
              +BM26*PX(11,I)+BM25*PX(10,I)+BM24*PX(9,I)+
              BM22*PX(7,I)+CO*(PX(5,I)+PX(6,I)
              +PX(3,I))
      9    CONTINUE

NO. 10     DO 10 I = 1, 100
           AR = CX(5,I)
           BR = AR-PX(5,I)
           PX(5,I) = AR
           CR = BR-PX(6,I)
           PX(6,I) = BR
           AR = CR-PX(7,I)
           PX(7,I) = CR
           BR = AR-PX(8,I)
           PX(8,I) = AR
           CR = BR-PX(9,I)
           PX(9,I) = BR
           AR = CR-PX(10,I)
           PX(10,I) = CR
           BR = AR-PX(11,I)
           PX(11,I) = AR
           CR = BR-PX(12,I)
           PX(12,I) = BR
           PX(14,I) = CR-PX(13,I)
           PX(13,I) = CR
     10    CONTINUE

No. 11     X(1) = Y(1)
           DO 11 K = 2, 1000
     11    X(K) = X(K-1)+Y(K)

No. 12     DO 12 K = 1, 199
     12    X(K) = Y(K+1)-Y(K)

NO. 13     DO 13 IP = 1, 128
           I1 = P(1,IP)
           J1 = P(2,IP)
           P(3,IP) = P(3,IP)+B(I1,J1)
           P(4,IP) = P(4,IP)+C(I1,J1)
           P(1,IP) = P(1,IP)+P(3,J1)
           P(2,IP) = P(2,IP)+P(4,J1)
           I2 = P(1,IP)
           J2 = P(2,IP)
           P(1,IP) = P(1,IP)+Y(I2+32)
           P(2,IP) = P(2,IP)+Z(J2+32)
           I2 = I2+E(I2+32)
           J2 = I2+F(J2+32)
           H(I2,J2) = H(I2,J2)+1.0
```

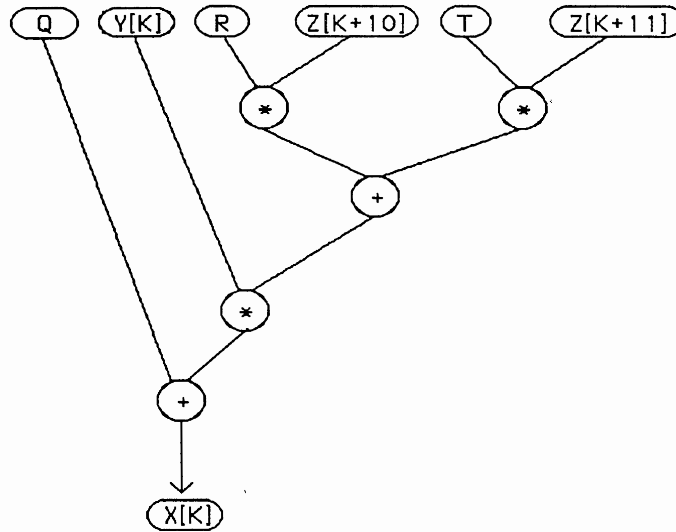
```
      13     CONTINUE

No. 14       DO 14 K = 1, 150
             IX=GRD(K)
             XI=IX
             VX(K) = VX(K)+EX(IX)+(XX(K)-XI)*DEX(IX)
             XX(K) = XX(K)+VX(K)+FLX
             IR = XX(K)
             RI = IR
             RXI = XX(K)-RI
             IR=IR-(IR/64)*64
             XX(K) = RI+RXI
             RH(IR) = RH(IR)+1.0-RXI
             RH(IR+1) = RH(IR+1)+RXI
      14     CONTINUE
```

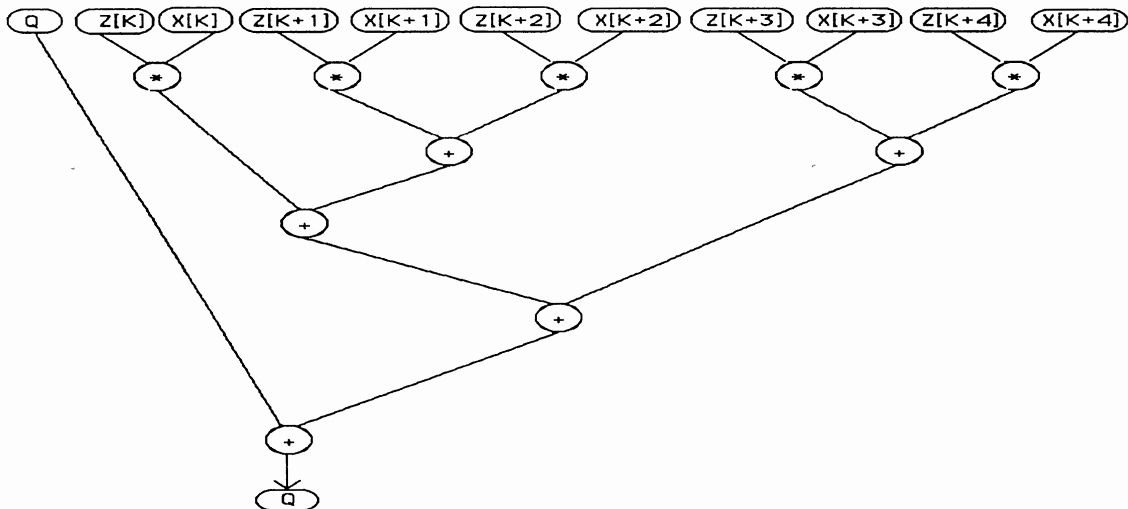
APPENDIX B

PROGRAM GRAPH OF THE LIVERMORE LOOPS

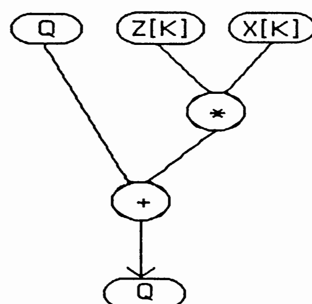
Loop 1



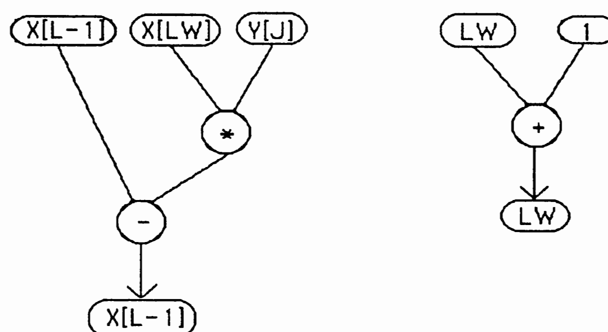
Loop 2



Loop 3



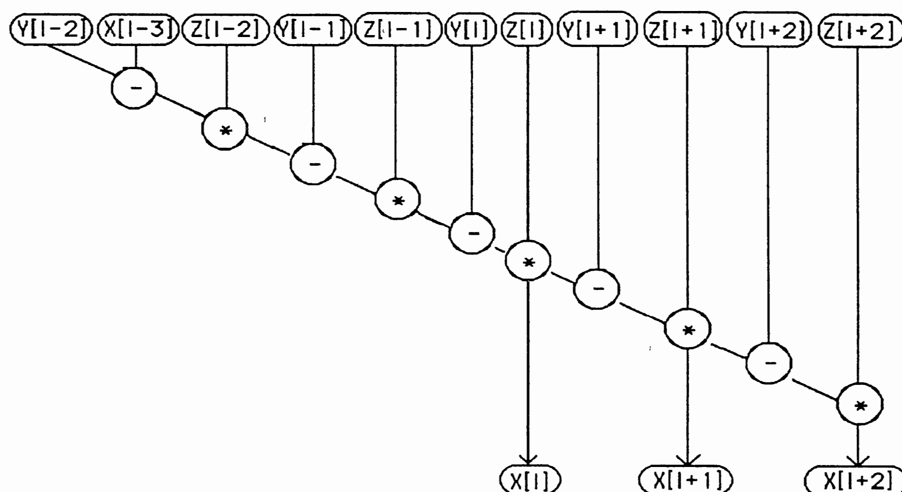
Loop 4



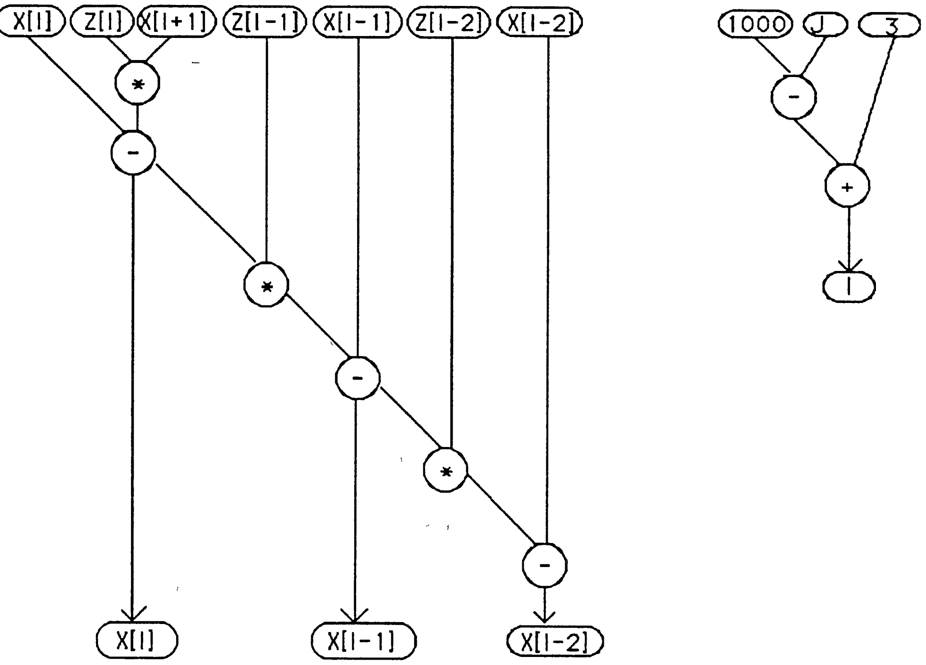
Loop 5
Modified by Kogge's double cycling method

DO 5 I = 2, 998, 3

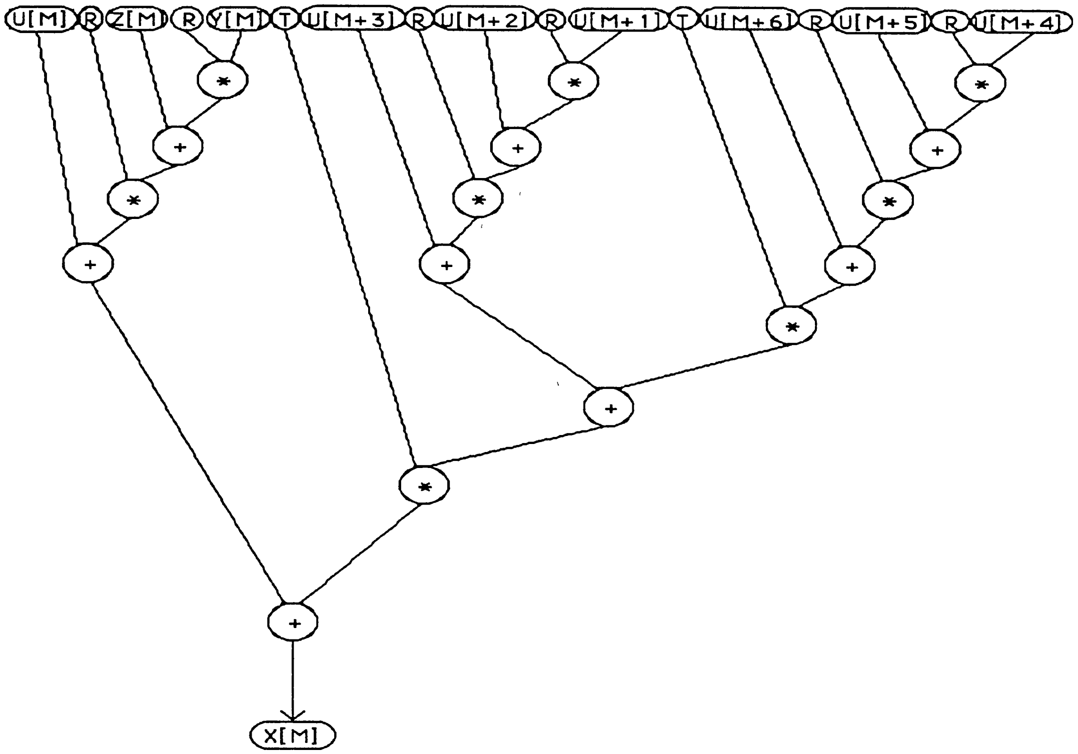
*5 X[I] = Z[I] * {Y[I] - Z[I-1] * (Y[I-1] - Z[I-2] * [Y[I-2] - X[I-3]])}*
*X[I+1] = Z[I+1] * (Y[I+1] - X[I])*
*X[I+2] = Z[I+2] * (Y[I+2] - X[I+1])*



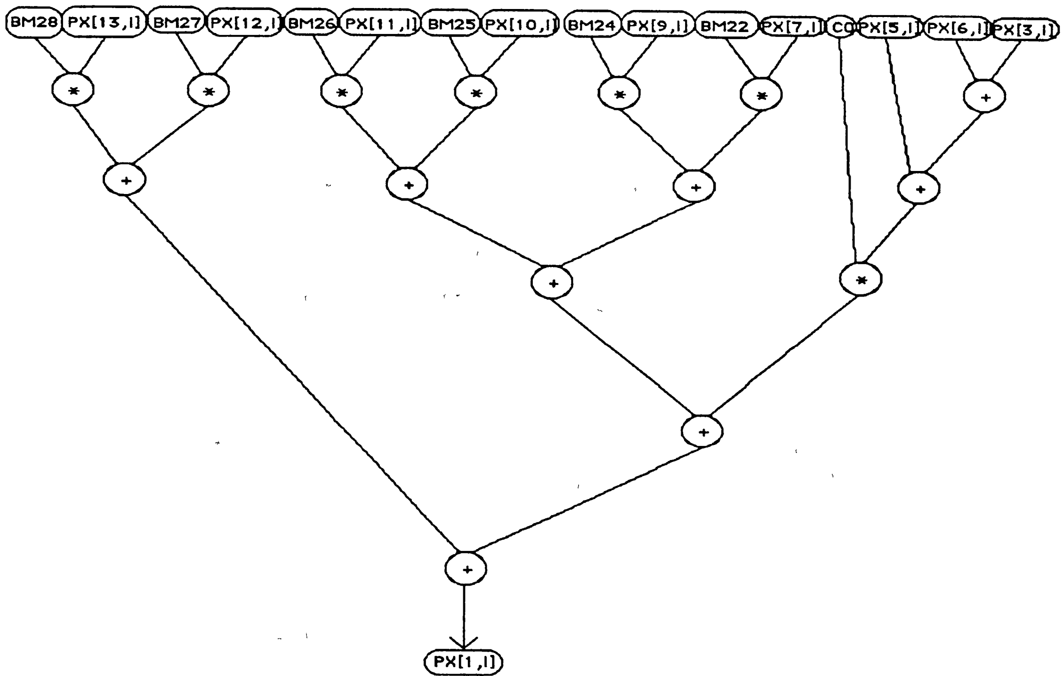
Loop 6



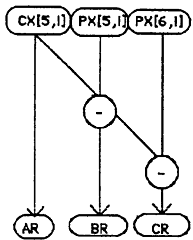
Loop 7

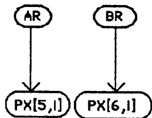


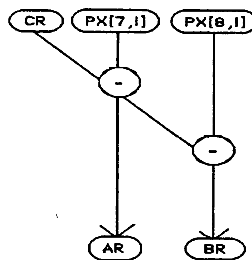
Loop 9

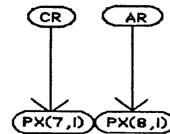


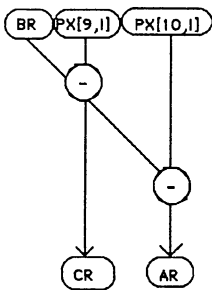
Loop 10

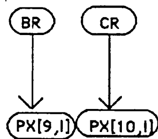


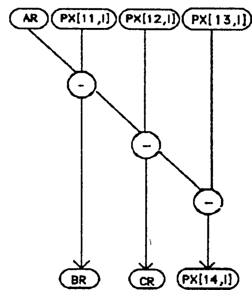


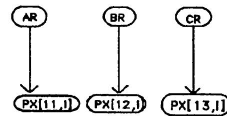




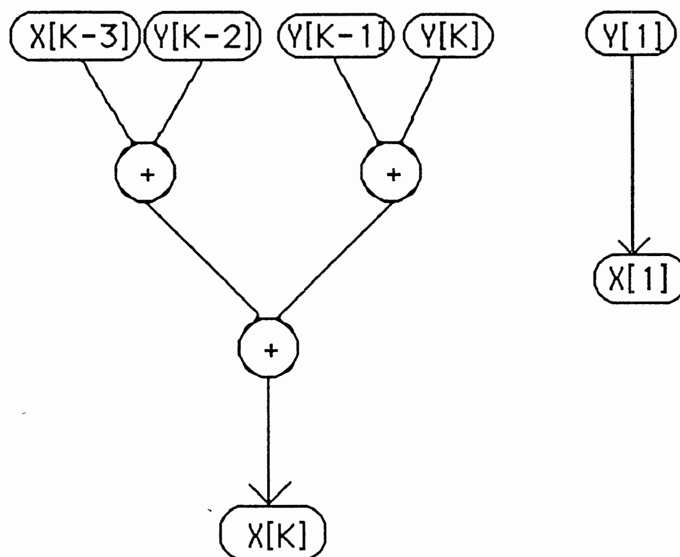




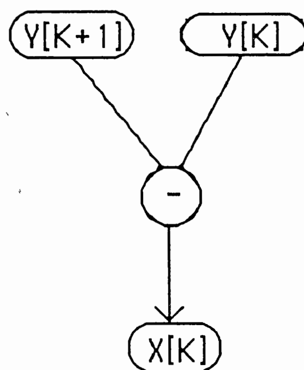




Loop 11
Modified by Kogge's double cycling method
X[1]= Y[1]
DO 11 K = 2, 1000
11   X[K] = X[K-3] + Y[K] + Y[K-1] + Y[K-2)]



Loop 12



APPENDIX C

WAFER SCALE INTEGRATION

Wafer Scale Integration (WSI) can be regarded as a special form of packaging in which extra wiring, normally used to interconnect the package containing working components, is fabricated on the surface of a wafer substrate containing the components and mounted inside a single package [12]. This internal wiring eliminates many of the problems encountered in the conventional printed circuit boards or ceramic carriers [12]. Internal wiring leads to reduced wiring length, which, in turn, increases the system speed and decreases the power requirement of the I/O drivers. The reduced wiring length achievable in WSI will not necessarily translate into reduced propagation delay, unless the other wire dimensions are scaled properly [12]. For example, suppose a metal wire of rectangular cross section with length *l*, width *w* and thickness *t* is located at a distance *d* from a ground plane of the same metal [12]. Then the *RC* charging delay of the distributed system is approximately given by [12]

$$T_{RC} = \frac{p \epsilon l^2}{td}$$

A typical integrated circuit line is made of Aluminium ($p = 2.63 \times 10^{-6}\ \Omega$ cm) on SiO_2 ($\epsilon_r = 3.9$) [12]. If $l = 20$cm, $t = d = 0.5\ \mu$m, then $T_{RC} = 160$ns [12]. If the dielectric constant is reduced to unity, then the delay would be only 600ps. This will be two orders of magnitude faster than the metal line of a conventional IC processing technique. $l = 20$cm line represents a worst-case length for WSI wiring on wafers three to four inches in diameter. This is small when compared to the chip-level wiring length of 25m for the IBM 3081 processor unit fabricated using the

conventional LSI technology [31]. Fifty percent of the CPU time of the IBM 3081 processor is dominated by the wiring delay [31]. Further, it is possible to obtain high propagation speeds in WSI using simple extensions of the existing technology. High propagation speeds can be achieved by fabricating thick film LC transmission lines rather than thin film lines with RC charging behavior [12]. The thick film lines that favorably effect the propagation times can also help improve the discretionary wiring yield, depending on the type of fabrication employed [12].

The statical uncertainty in the wiring delays resulting from wiring of the random collection of working components has been one drawback of traditional WSI. That is, since the locations of the working cells are not fixed, the wiring delay of a given path may vary from wafer to wafer. However, since these delays can be much shorter than those found in most other packaging arrangements, this could be less of a problem than might be expected [12].

There are few subjects in solid-state electronics that bring forth many negative comments as WSI [39]. These are partly due to the dominance of the prevailing VLSI technology which is expected to dominate the field well into the future, and lessen the need for WSI [12]. But many designers overlook the fact that VLSI actually makes poor use of the enormously large silicon area available. The average VLSI chip area grows very slowly even as higher levels of integration are achieved. The name WSI implies a quantum jump in more components integrated on a monolith piece of silicon than the state-of-art VLSI. The WSI silicon piece is much larger than the one used in the state-of-art VLSI, and is normally of wafer size [39].

The attractiveness of WSI lies in its promise of reduced cost, high performance, higher level of integration, greatly increased reliability and application potential. Traditionally, the increased component density of a VLSI chip is achieved primarily by a downscaling of the feature sizes; only in a secondary manner

is this increased component density obtained by the use of larger chip dimensions. The increase in component density of a VLSI chip due to the shrinkage of the minimum feature size has been of several orders of magnitude, while the increase of maximum feasible chip area has been modest [39]. Since the practical limit of scaling has not been reached, VLSI will continue to dominate WSI. But WSI tries to increase the component density still further by bringing an increased chip area, which has contributed very little to the VLSI performance. Further, redundancy and fault-tolerance in WSI adds reliability to the fabrication increasing the growing number of advantages it has over VLSI.

To avoid multilevel metallization in WSI, it is necessary that the circuit design avoid cross-wafer data communication as much as possible. Therefore, instead of cross-wafer data communication, the cells fabricated should communicate with the neighboring cells. This is possible with the pipelined and bus-oriented architectures; therefore, WSI is more suitable for pipelined and bus-oriented architectures [39]. The FTVP proposed in this study is a pipelined architecture favoring WSI fabrication. The important consequence of this pipelined structure fabrication is the avoidance of multilevel metallization. This, in turn, makes it practical to apply the state-of-art VLSI fabrication technology to WSI, thereby giving it a significant density advantage over the equivalent VLSI implementation. To remain competitive over VLSI, any WSI process must satisfy the following requirements [39]:

* Make use of the densest VLSI fabrication process available.
* Avoid the introduction of additional process as much as possible to keep the complexity to a minimum.
* Avoid cross-wafer communication by using pipeline architectures.
* Provide multiple external power and ground contacts on the wafer at regular intervals.

But it must be noted that any advances made in WSI will be reflected on VLSI and vice-versa. The major competitors for WSI are the VLSI technology itself, because of the rapid decrease in scaling technology which helps in achieving higher chip density and the multichip VLSI technology. Table VI shows the figure of merit of various technologies [39].

TABLE VI

COMPARISON OF THE PACKAGES OF FIGURE OF MERITS

Packaging Approach	Power xDelay	Size or Weight	Cost	Overall Figure of Merit
Printed wiring board	1.00	1.00	1.00	1.00
Thick-film multilayer on ceramic	1.08	0.42	1.02	0.46
Ceramic multilayer hybrid	0.34	0.2	0.65	0.044
Thin-film multilayer hybrid, populated on one side	0.19	0.14	0.6	0.016
Wafer Scale Integration	0.10	0.09	0.46	0.0041
Thin-film multilayer hybrid, populated on both sides	0.08	0.07	0.44	0.0025

As seen from Table VI, the multichip hybrid technology has an advantage over WSI because it is risk-free and offers high performance due to the advancement in VLSI technology. But the multichip hybrid technology requires a large number of metallurgical bonds, and has difficulty in heat removal [39].

VITA

SUNDARARAJAN GANESH

Candidate for degree of

Master of Science

Thesis: THE FAULT-TOLERANT SINGLE-CHIP VECTOR PROCESSOR: ARCHITECTURE AND PERFORMANCE ANALYSIS USING LIVERMORE LOOP BENCHMARKS

Major Field: Electrical Engineering

Biographical:

Personal Data: Born in Ganapathi Agraharam, Tamilnadu, India, September 24, 1967, the Son of V. Sundararajan and R. Malathy.

Education: Graduated from ST. Joseph of Cluny higher secondary school, Neyveli, India in June 1984; received Bachelor of Engineering degree in Electronics and Instrumentation from Annamalai University, Chidambaram, India in July 1988; completed requirements for Master of Science degree at Oklahoma State University in May 1992.

Professional experience: Research assistant to Dr. J. J. Lee, Department of Electrical Engineering, Oklahoma State University, January, 1990 to December 1991; Technical assistant to Mr. Bill Barnes, Department of Biochemistry, Oklahoma State University, September, 1989 to April 1990; member of Tau Beta Phi and Eta Kappa Nu.