INFORMATION TO USERS

This reproduction was made from a copy of a document sent to us for microfilming. While the most advanced technology has been used to photograph and reproduce this document, the quality of the reproduction is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help clarify markings or notations which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure complete continuity.

2. When an image on the film is obliterated with a round black mark, it is an indication of either blurred copy because of movement during exposure, duplicate copy, or copyrighted materials that should not have been filmed. For blurred pages, a good image of the page can be found in the adjacent frame. If copyrighted materials were deleted, a target note will appear listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed, a definite method of "sectioning" the material has been followed. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For illustrations that cannot be satisfactorily reproduced by xerographic means, photographic prints can be purchased at additional cost and inserted into your xerographic copy. These prints are available upon request from the Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases the best available copy has been filmed.
WDDM - A WAFER-SCALE DATA DRIVEN MULTIPROCESSOR
WDDM- A WAFER-SCALE DATA DRIVEN MULTIPROCESSOR

A DISSERTATION

APPROVED FOR THE SCHOOL OF ELECTRICAL ENGINEERING AND

COMPUTER SCIENCE

BY

[Signatures]

[Names]
ACKNOWLEDGMENT

I would like to express my gratitude and thanks to Dr. Ali R. Hurson, my advisor and chairman of the Committee, for his patience, guidance, and encouragement during the course of preparing this manuscript. His insightful discussions and enthusiasm for the investigation of data driven systems and other computer architecture related topics, have inspired me to continue my work in this field. I would especially like to thank Dr. Hurson for reading and correcting this dissertation many, many times. I would like to extend my gratitude to the Committee members, Professors Bredeson, Lakshmivarahan, Minor, and Carr, for their valuable suggestions and guidance.

I would like to express my appreciation to my parents, Ali and Sourour Shirazi, for their financial, mental, and all-around support to continue my educational career. And last, but not least, I would like to thank my wife, Mary, for her endless support and patience in the course of preparing this work. Her help in typing and editing this manuscript has been invaluable.
TABLE OF CONTENTS

Title page .................... i
Signature page .................. ii
Copyright ...................... iii
Acknowledgment ................ iv
Table of contents ................ v
List of tables ................... vii
List of figures ................... viii
Abstract ........................ xii

CHAPTER

I. INTRODUCTION .................... 1

1. Dissertation Objectives .......... 6
2. Dissertation Contents .......... 8

II. THE BACKGROUND WORK .......... 10

1. Data Driven Computations .......... 11
2. Background Work ................ 19
3. The Block Driven Processors .... 60
4. Summary ........................ 68

III. THE PROPOSED ARCHITECTURE .... 69

1. The Proposed Architecture ........ 73
2. Summary ........................ 99

IV. MEMORY MANAGEMENT OPERATIONS .... 101

1. The Previous Work .............. 103
2. Memory Management Operations in WDDM .. 114
3. Summary ........................ 144
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 5.01</td>
<td>Matrix multiplication in WDDM</td>
<td>174</td>
</tr>
<tr>
<td>Table 6.01</td>
<td>ALU operation list[109]</td>
<td>190</td>
</tr>
<tr>
<td>Table 6.02</td>
<td>Sequence of the operations in the proposed model</td>
<td>216</td>
</tr>
<tr>
<td>Table 6.03</td>
<td>Execution and geometry for different configurations</td>
<td>220</td>
</tr>
<tr>
<td>Table 6.04</td>
<td>The 3-bit pattern of modified Booth's algorithm</td>
<td>222</td>
</tr>
<tr>
<td>Table 6.05</td>
<td>Sequence of the operations in the proposed model</td>
<td>227</td>
</tr>
<tr>
<td>Table 6.06</td>
<td>Comparison with other multiplier units</td>
<td>235</td>
</tr>
<tr>
<td>Table 7.01</td>
<td>The distribution and the execution time of arithmetic operations</td>
<td>258</td>
</tr>
</tbody>
</table>
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 2.01:</td>
<td>An example of a data driven graph program.</td>
<td>13</td>
</tr>
<tr>
<td>Figure 2.02:</td>
<td>Block diagram of the conventional computers.</td>
<td>21</td>
</tr>
<tr>
<td>Figure 2.03:</td>
<td>Organization of the Class I data driven machines.</td>
<td>23</td>
</tr>
<tr>
<td>Figure 2.04:</td>
<td>Block diagram of the Manchester data driven machine.</td>
<td>25</td>
</tr>
<tr>
<td>Figure 2.05:</td>
<td>Block diagram of the Irvine data flow machine.</td>
<td>29</td>
</tr>
<tr>
<td>Figure 2.06:</td>
<td>A processing element in the Irvine data flow machine.</td>
<td>31</td>
</tr>
<tr>
<td>Figure 2.07(a):</td>
<td>Structure of the Data Flow Multi-processor.</td>
<td>34</td>
</tr>
<tr>
<td>Figure 2.07(b):</td>
<td>An activation processor.</td>
<td>34</td>
</tr>
<tr>
<td>Figure 2.08:</td>
<td>Organization of the Class II of the data driven machines.</td>
<td>37</td>
</tr>
<tr>
<td>Figure 2.09:</td>
<td>Central structure of the data flow processor.</td>
<td>39</td>
</tr>
<tr>
<td>Figure 2.10:</td>
<td>The arbitration network.</td>
<td>41</td>
</tr>
<tr>
<td>Figure 2.11:</td>
<td>The organization of the Control and distribution networks.</td>
<td>41</td>
</tr>
<tr>
<td>Figure 2.12:</td>
<td>Data structure handling in the MIT machine.</td>
<td>44</td>
</tr>
<tr>
<td>Figure 2.13:</td>
<td>Organization of the MIT data flow multi-processor.</td>
<td>46</td>
</tr>
<tr>
<td>Figure 2.14:</td>
<td>Block diagram of the TI data flow machine.</td>
<td>47</td>
</tr>
<tr>
<td>Figure 2.15:</td>
<td>Organization of a processing element in the TI data flow machine.</td>
<td>49</td>
</tr>
<tr>
<td>Figure 2.16:</td>
<td>Recursive definition of DDM1.</td>
<td>51</td>
</tr>
<tr>
<td>Figure 2.17:</td>
<td>Structure of a processing element in the DDM1.</td>
<td>53</td>
</tr>
<tr>
<td>Figure 2.18:</td>
<td>The overall organization of the Class III data driven machines.</td>
<td>59</td>
</tr>
<tr>
<td>Figure 2.19(a):</td>
<td>An example of a joint computation.</td>
<td>62</td>
</tr>
<tr>
<td>Figure 2.19(b):</td>
<td>Steps in an interleaved computation.</td>
<td>62</td>
</tr>
<tr>
<td>Figure 2.20:</td>
<td>Block-driven data-flow architecture.</td>
<td>65</td>
</tr>
<tr>
<td>Figure 2.21:</td>
<td>The general organization of the Machine d'Assignation Unique Dynamique (MAUD).</td>
<td>67</td>
</tr>
</tbody>
</table>
Figure 6.04: Circuit diagram of the general function block of Figure 6.3. ........................ 193
Figure 6.05: The stick diagram of the carry chain block ........................ 196
Figure 6.06: A control line driver. ............................................. 198
Figure 6.07: The transistor diagram of the ALU[109] ................ 200
Figure 6.08: The transistor diagram of a 1-of-4 decoder ................ 203
Figure 6.09: Selector transistor diagram. ........................................ 204
Figure 6.10: The block diagram and the truth table of a priority generator function block. 206
Figure 6.11: The transistor diagram of the priority generator function block .......... 207
Figure 6.12: A traditional word-oriented multiplication ........................ 210
Figure 6.13: An example of the proposed scheme. .............................. 213
Figure 6.14: The overall architecture of the proposed multiplier unit. ........ 214
Figure 6.15: Stick diagram of the basic cell type 1 ................................ 218
Figure 6.16: Stick diagram of the basic cell type 2 ................................ 219
Figure 6.17: The overall organization of the proposed multiplier ............... 224
Figure 6.18(a): Geometry area of the previous model .......................... 229
Figure 6.18(b): Geometry area of the proposed architecture .................... 229
Figure 6.19: The basic cell blocks ................................................. 231
Figure 6.20(a): Stick diagram of the basic cell type 1 .......................... 232
Figure 6.20(b): Stick diagram of the basic cell type 2 .......................... 233
Figure 6.21: A comparand cell and its associated circuitry ........................ 237
Figure 6.22: A memory cell and its associated circuitry ................................ 238
Figure 6.23: Stick diagram of a comparand cell ................................... 239
Figure 6.24: Stick diagram of a memory cell .................................... 240
Figure 6.25: I-generator block ..................................................... 241
Figure 6.26: Stick diagram of the I-generator block ................................ 243

Figure 7.01: A model for the processing module operations ........................ 251
Figure 7.02(a): A model for the proposed system operations ......................... 253
Figure 7.02(b): A model for the proposed system operations ......................... 254
Figure 7.03: The effect of parallelism in a program ................................ 263
Figure 7.04: Average waiting time in the queue .................................... 264
Figure 7.05: Maximum queue length .............................................. 265
Figure 7.06: Average block execution time ....................................... 267
Figure 7.07: Effect of parallelism among the blocks ................................ 268
Figure 7.08: The MIPS performance of the system .................................. 270
Figure 7.09: The average processing module utilization ........................... 271
Figure 7.10: The MIPS performance of the system .................................. 273
Figure 7.11: The average processing module utilization ........................... 274
Figure 7.12: The effect of data structure operations on MIPS measurement. 275
Figure 7.13: The effect of data structure operations on average processing module utilization. 276
ABSTRACT

It has been shown that the existing von-Neumann design machines fail to meet the requirements of the real-time scientific and the 5th generation applications. A computer organization, such as data driven architecture, which embeds parallelism in its definition and underlying design is needed to satisfy such requirements. In this project we will introduce a new data driven multiprocessor which: i) eliminates some of the problems of the existing data driven systems; ii) takes advantage of state-of-the-art technology; and iii) achieves parallelism at the data, instruction, and program block levels. The system consists of n processing modules, a host module, and a data structure module, connected via a double star network with the host and the data structure modules in the center of the stars. Processing modules are capable of the simultaneous execution of independent program blocks in data driven fashion. The host module is responsible for memory management tasks, while the data structure module facilitates the manipulation of the data structures.
CHAPTER I

INTRODUCTION

Advances in computer technology from vacuum-tubes to VLSI have reduced the size and cost of the computer components while increasing the speed and reliability of their operations. However, during the same time period, computer systems have evolved from uni-processor-single-user to multi-processor-multi-user organizations. Computer applications have expanded from scientific applications with a small volume of data to a vast and varied spectrum of applications manipulating massive volumes of data. Examples of such applications include: database management systems, text retrieval systems, natural language processing, artificial intelligence, fluid dynamics simulations, speech and pattern recognition, etc. These applications require a computational power of 100M-to-1G LIPS (Logical Inferences Per Second) [130,51,16,13]. This computation demand is an order of magnitude more than the computational power of the existing machines [47,16,23,13]. Such a gap between the computational power demanded by the applications and that of
the available computers, called the computation gap, is a major problem in computer science today.

The computation gap has been the result of factors such as: i) the sequential nature of the von-Neumann architecture; ii) the technological constraints imposed by the physical laws; and iii) the software/hardware complexities introduced by the existing traditional concurrent systems. In order to close the computation gap and improve the performance of computer systems, efforts have been concentrated in three areas:

1) Improvement of technology and development of new hardware algorithms bearing the constraints imposed by the current technology: In this direction, one can reduce the execution time of the basic operations and hence, improve the overall performance of a system. However, due to physical laws there is a limit for such improvements [18, 16]. For example, it is predicted that by VLSI technology and the Josephson junction's method, a gate delay of 15 pico seconds is achievable [117, 86] in the near future. Such a speed is close to the limitations dictated by physical laws.

2) Development of parallel and pipelined processors: These machines allow concurrency either by the execution of several operations through the replication of the existing von-Neumann design processors, or by staging a function into a group of partially independent phases. Examples of such architectures include: multi-functioned processors, array
processors, multi-processors, and pipelined processors. These systems can improve the performance through the concurrency in the operations. However, the traditional concurrent systems are basically extensions of the sequential von-Neumann architecture. It has been shown that these systems increase the complexity of the system both at the hardware as well as the software levels. For example, in most cases a complicated software package is necessary to reorganize user programs in order to achieve parallelism. Furthermore, it has been shown that, in practice, systems such as Cray-1 [86] and ILLIAC IV [15] achieve maximum performance only in special-purpose applications such as vector processing.

3) Introduction of new special- and general-purpose computer architectures: The deficiencies of the first two directions have given rise to the design and implementation of: i) systems which are tailored for special applications or ii) systems which bear parallelism in their basic definition. The former approach has resulted in the development of special-purpose computer systems, while the latter is more geared towards general-purpose computing environments.

A special-purpose organization is designed for the efficient execution of a specific set of applications. For example, database management systems [90,105,72] and text retrieval systems [116,17,71] are developed for the
efficient manipulation of non-numeric applications. The speech recognizers and pattern recognition devices are used in artificial intelligence applications. And, high-level language machines [107,108] are designed for closing the semantic gap between the requirements of the specific high-level languages and the capabilities of the underlying machines.

The new general-purpose computer architectures are based on decentralization of the computing power. A decentralized computer system, which is based on the parallel execution of independent tasks, can take advantage of the embedded parallelism in a program to speed up the execution. Thus, the decentralized computation approach encourages the design and development of systems which are inherently parallel. The data driven architecture [128,126, 36,3] is an example of such an organization.

In a data driven environment a program is represented as a directed graph in which nodes are operations to be performed and the arcs direct the data among the nodes. The operations in this model are functions and therefore, they do not have any undesirable global effects beyond their own nodes. In such an organization, the availability of the operands of an operation causes the immediate execution of that operation, provided that the succeeding operations are ready to accept the result. Therefore, in a data driven environment, many operations can be executed concurrently.
and asynchronously. This radical departure from the sequential von-Neumann type organizations has eliminated familiar concepts such as: the program counter, addressing schemes, an updatable storage, use of identifiers, and all of their associated by-products such as global side-effects and aliasing.

The concepts of "functionality", "concurrency", and "asynchrony" which are embedded in the definition of a data driven architecture provide grounds for a high degree of implicit parallelism. This has been to the extent that the leading candidates for the future supercomputers are considered to be data driven and multi-processor systems [63,13,16]. For example, it has been stated that a decentralized computer organization, such as data driven architecture, is needed to accomplish performance goals required by the 5th generation applications [104,130]. In fact, the development of an ultimate supercomputer, based on the data driven concepts, is an ongoing project in the United States, Japan, and Europe [50,49,46].
1. DISSERTATION OBJECTIVES

Our research goal is to combine the merits of the three directions discussed in the first section in order to achieve a higher performance. Such a goal is accomplished by the introduction of a new data driven architecture which:

i) incorporates the requirements for an "easy" implementation of the system using the recent computer technology, namely VLSI; ii) is based on multi-processing principles and achieves parallelism at the data, instruction, and program block levels; and iii) eliminates some of the problems of the existing data driven systems.

i) In order to comply with the constraints imposed by the VLSI technology, the proposed architecture is designed so that it consists of only a few building block elements which are replicated many times across the system. Additionally, in order to solve the I/O pin limitation problem [57], we have utilized one of the most recent developments in the VLSI technology, namely Wafer-Scale Integration [94].

ii) The proposed architecture takes advantage of the embedded parallelism in a program by providing hardware
facilities to: a) process arrays in parallel in a SIMD fashion; b) process independent instructions concurrently in a data driven fashion; and c) execute independent program blocks (such as loop iterations) simultaneously in a multiprocessing fashion.

iii) Some of the shortcomings of the existing data driven models are eliminated by: a) consideration of the implementation issues with regard to the technology constraints; b) distribution of the processing power among the basic memory cells; and c) closing the semantic gap between the requirements of the data driven languages and the underlying proposed architecture.
2. DISSERTATION CONTENTS

Chapter II of this manuscript provides a basic background in the data driven organization area. The concept of the data driven computation is first discussed, followed by a study of the existing data driven computers. We will introduce a new classification for the data driven architectures. Several machine organizations (along with their advantages and disadvantages) are studied in each class.

The proposed Wafer-Scale Data Driven Multiprocessor (WDDM) is presented in Chapter III. The overall machine organization and the structure of each system unit are discussed in detail.

The system memory management organization is introduced in Chapter IV. The issues discussed in this chapter include: the compiler role in memory management operations, the memory hierarchy organization, the processor allocation, and the procedure call/return mechanism.

Chapter V covers the manipulation of data structures in the proposed system. The data structure operations at both the high-level and the machine level are discussed.
The VLSI design of the proposed architecture is conclusively discussed in Chapter VI. The VLSI timing/geometry evaluations of the system components are presented according to the available technology.

Chapter VII presents the performance evaluation of the proposed architecture. The system behavior is simulated and the effects of parallelism in the programs, procedure calls, data structure operations, and different system configurations are studied.

Finally, Chapter VIII is the conclusion of the manuscript. It also enumerates further research and development projects in this area.
CHAPTER II

THE BACKGROUND WORK

The primary goal of our work is the design of a new VLSI-implementable data driven machine which overcomes the shortcomings of the previously proposed data driven architectures. Therefore, we first need to study some of the existing data driven machines and pinpoint their drawbacks. In order to organize the study, we will propose a classification of the data driven machines based on the relationship and interconnection among data values, instructions, and processing elements. During the course of this study, we will introduce a number of technical terms which are used in the subsequent chapters.

The first section of the chapter addresses the concept of data driven computation and the characteristics of data flow programs. The classification and study of the existing data driven machines are covered in the second section along with a discussion of their drawbacks. Section three discusses the block driven architecture as a special case of the data driven systems. Finally, section four is the summary of the chapter.
1. DATA DRIVEN COMPUTATIONS

Most of the existing programming languages are based on control-flow operations. In such languages, the logical sequence of instructions determines the order of the execution of the operations. Even the conventional approaches towards parallelism are based on control-flow operations [101,11]. The major disadvantage of control-flow operations is that they impose a sequential ordering on the execution of a set of instructions which could otherwise be processed in parallel. Such parallelism must be detected during the compile time or the run time. However, the detection of parallelism in a control-flow language is "complicated" and "time-consuming" at both the compile time and the execution time [119,83]. This is in part due to the use of global or shared memories in the conventional languages.

Data driven computations overcome such disadvantages by eliminating side-effects (no global storage) in a program. Furthermore, the graphical representation of a data driven program explicitly reveals the independent operations and the parallelism embedded in that program. A graph
representation of a data driven program is a directed graph in which the nodes are the actors or links and the arcs indicate the flow of data among nodes. Figure 2.1 depicts a small data driven program which computes the roots of a quadratic equation. The links which copy and distribute the data are represented by solid nodes, while the actors which execute operations are represented by hollow circles. The data values (tokens) flow from one node to another in the direction of the arcs.

In a data driven program the availability of the operands of an operation causes the immediate execution of that operation, provided that the succeeding instructions are ready to accept the result [40]. Therefore, the instructions can be executed regardless of the logical sequencing imposed by the program. Once an operation is executed, its result is simultaneously made available to the destined instructions. If the other operands of the succeeding instructions are already available, they can be concurrently executed.

Hence, a node (operation) is fired or executed as soon as the following two conditions (firing conditions) are satisfied:

1) There is a token available on all of the node's input arcs.
2) There is no token on the node's output arc.
Figure 2.1: An example of a data driven graph program.
In order to implement condition (1), a counter in each instruction is used to identify the number of unavailable operands. As an operand arrives, the counter is decremented by one, and when the counter becomes zero, the instruction is enabled for execution [119]. Condition (2) is enforced either through an acknowledge signal communication system [42] or by the utilization of result queues (token queues) [133]. In the former approach, a node becomes enabled for execution after it receives an acknowledge signal from each of its successor nodes indicating that they are ready to receive the result. In the latter approach, a node may be fired as soon as its input operands are available. However, the results are queued at the successor nodes and processed in sequence.

While a growing amount of research is being devoted to the data driven machines, there has also been some criticism of these systems. It has been indicated that data driven designs have difficulty dealing with large arrays, have excessive computational overhead, and perform poorly on some classes of non-parallel applications [62].

The problem of manipulating large arrays stems from the data flow concept which treats both the single and the array data items as a single token value. Therefore, if a process modifies only one element of a large array, the entire array must be passed to that process. This is a tremendous waste
of transmission capacity. However, many solutions have been suggested for such a problem. These solutions will be discussed in Chapter V, where data structure manipulation is covered.

A second disadvantage associated with the data driven machines is the amount of computation that has to be devoted to bookkeeping operations. These operations are the result of the tagging (labelling) scheme used in some data driven designs. In this scheme, each token is augmented with a tag which indicates the activation of the program block (such as a loop) to which the token belongs. Therefore, by using different tags for different activations of a block, it is possible to execute them concurrently. However, it has been pointed out that the parallelism gained through token tagging can compensate for the sacrifice of speed involved in tag creations [92].

The fact that the data driven systems are inefficient in handling non-parallel problems, is generally acknowledged by the data driven designers (i.e. data driven machines are intended for problems that are inherently parallel).

**Data Flow Program Characteristics:**

Data driven programs are determinate, functional, and side-effect free, as discussed in the following paragraphs:

(1) **Determinism:** We mentioned that instructions of a
data driven program are executed in a concurrent and asynchronous fashion. However, data driven programs must be determinate, implying that their results are independent of the relative order of the execution of the instructions. In other words, for the same set of input data items, the program must always produce the same output.

(2) Functionality: It is unreasonable to expect the programmers to explicitly define all the concurrent tasks in a program. A high-level language should let programmers focus on the presentation of an algorithm but limit their style sufficiently so that concurrency can be easily identified by the compiler. A data driven programming language can meet such a design criterion by using completely functional language features. Like pure Lisp, the programmers write their programs through the utilization of expressions and functions only. The important property of the functions and expressions is that once the input values for them are known, their execution can commence independently. Therefore, the rule to determine concurrency is:

"If two operations (expressions or functions) do not depend on the outcome of each other, then they can be executed simultaneously" [95].

By the restriction of allowing only expressions and
functions, a data driven compiler can easily detect concurrent operations according to the above rule. Since pure Lisp is not readable, often an algebraic-type syntax similar to PASCAL has been chosen to implement these languages. VAL [95] is an example of such data driven languages.

(3) Lack of side-effects: Most conventional languages have concepts like "variables" and "memory updating". In a data driven environment such concepts are non-existent because objects (arrays or scalar values) are simply passed from one operation to the next. Once the operation is executed, the input object is also destroyed. In a data driven language the concept of binding values to identifiers is still possible. However, identifiers cannot be used as variables. Such a concept is implemented through the single-assignment rule:

"Once an identifier is bound to a value, that binding must remain in force for the entire scope of access to that identifier" [95].

Therefore, each time a scope of access (block, function, expression) is entered, new bindings can be made which remain in force until that scope of access is completed. For example, if a constant value is to be used in several functions, it must be bound to an identifier within each
function.

The identifier/value bindings can be allowed for expressions, too. For example, in VAL [95], the LET-IN expression construct is used for this purpose:

\[
\text{LET} \\
\hspace{1em} \cdot \\
\hspace{2em} \cdot \text{ bindings} \\
\hspace{1em} \cdot \\
\text{IN MAKE expression} \\
\text{ENDLET.}
\]

In LET-IN expressions, it is possible to bind identifiers to arrays and constant values and use them in the expression. The information provided in the Let-in section allows the compiler to initialize the constant values for operations or to generate arrays for use in the expression.

It is obvious that the single-assignment rule eliminates the side-effects among functions and expressions and guarantees the possibility of concurrency among functions. For example, it is possible to input the same object (array/scalar value) to two different functions at the same time without any possibility of side-effects. This is due to the fact that the objects are bound to different identifiers within the two functions. Thus, they can be used independently and destroyed upon the completion of the function.
2. BACKGROUND WORK

Due to the nature of data driven computations, all the data driven computers can be classified as Multiple Instruction stream-Multiple Data stream (MIMD) processors. This is because multiple instructions may be fired concurrently, performing operations on different data values. In order to implement such concepts, many of the data driven machines are either distributed processors [32] or multifunctional processors [119].

Hwang and Briggs [82] have categorized data driven computers into static and dynamic machines. In a static data driven system, there must not be more than one token on any arc at any given instance. Therefore, the static firing condition consists of the presence of tokens on each of the input arcs of an operation and the absence of any token on its output arcs. These machines often use an acknowledge signal system to implement the firing rule. The MIT machine [40,42] is an example of the static category.

A dynamic data driven system uses a token labelling (tagging or coloring) scheme to allow multiple tokens on an arc simultaneously. Each label represents a different
activation of a program block. Therefore, the multiple tokens occupying a single arc at the same time must each have a different label. The dynamic firing condition is an extension of its static counterpart; i.e. there must be tokens with the same label on each of the input arcs and no token with the same label on the output arcs. The machines in this category utilize a queuing system to temporarily store the multiple tokens for each operation. The Irvine machine [11] and the Manchester machine [133] are two examples in this class.

The above classification groups the data driven machines according to the firing condition and its implementation. However, it does not provide much information about the underlying architecture of the machines. Therefore, we have proposed a classification of the data driven computers based on the relationship and communication among the data memory, the instruction memory, and the processing elements. This classification not only characterizes the architecture of the machines, but it also indicates the trend of the development in this direction from the conventional systems to the future data driven supercomputers.

Figure 2.2 depicts the inter-communication among the data, the instructions, and the processors in the von-Neumann design computers. The data and instructions are stored in the same memory unit, but occupy disjoint storage
Figure 2.2: Block diagram of the conventional computers.
space. During the instruction cycle, the memory is accessed to fetch the next instruction, fetch the operands, and save the result in the memory. We have used a similar approach as the basis for the grouping of the data driven computers. The proposed classification divides the data driven machines into three major categories. The characteristics of the classes and the architecture of several machines in each class are discussed next. Of course, this is by no means an exhaustive survey of the data driven machines. Interested readers may refer to [128, 33, 115, 131, 111, 98, 100] for additional information about the ongoing data driven projects.

1. Class I:

Figure 2.3 depicts the general organization of the data driven machines in this class. The memory organization consists of the instruction memory and the data memory. During the instruction cycle, the memory is accessed in three phases due to the following operations: i) the available operand(s) of an instruction cause the instruction to be fired; ii) the operation packets (opcode, operands, and destination addresses) are sent to the processing unit; and iii) the result packet(s) are routed to the data memory. The machines in this class include:
CLASS I

- Separate units for data, instructions, and processing elements

Figure 2.3: Organization of the Class I data driven machines.
The Manchester Machine:

This machine was developed by Watson and Gurd [133,134, 67,68] at the University of Manchester beginning in 1975. A fully operational prototype model has been constructed with a performance of 2 MIPS - over four times the speed of a VAX 11/780 at a fraction of its cost [36,68]. The system is a backend pipelined data driven machine which utilizes "token labelling" to allow the parallel execution of several activations of a program block. The machine consists of five basic units on a "ring". Figure 2.4 depicts the block diagram of the Manchester data driven computer. The units and the sequence of the machine operations are as follows:

i) The Switch Unit: interfaces the backend data driven machine to the host computer. Initial values are supplied through this switching network and the output results (carrying a special label) are collected here.

ii) The Token Unit: is a FIFO organized buffer which represents the notion of arcs in a dataflow graph.

iii) The Matching Store Unit: is a temporary storage for the tokens. It is used to detect the availability of the operands for the instructions. An instruction can be a binary or a unary operation. Therefore, the matching store is used to match the operands of the binary instructions. The unmatched tokens are temporarily stored in the matching store until their partners arrive.
Figure 2.4: Block diagram of the Manchester data driven machine.
A token contains the data value, its label, the destination address, and some control bits identifying the token type (e.g. binary or unary operand token). As a token reaches the front of the token queue, it is passed to the match unit. For the binary operations, a search through the matching store is performed to find a token with the same label and destination as the newly received token. If a match does not occur, the token is stored in this unit, awaiting its match. In the case of a match, the two tokens are the operands of the same instruction addressed at "destination". The instruction belongs to an activation of a procedure denoted by the "label". Therefore, the two matched tokens are used to form a token packet which is then sent to the instruction store. If a token is the operand of a single-operand instruction (indicated by control bits), it bypasses the matching store and is directly routed to the instruction store. The matching unit performs the search in an associative fashion, based on the label and destination fields. However, the unit has been simulated through a hardware hashing technique based on the works of Goto and Ida [65].

iv) The Instruction Store Unit: holds the instructions of a data driven program. The operands of the instructions are held in the token queue and the matching store. According to the implementation of the machine [133], each instruction can specify at most two
destination addresses. Therefore, an instruction is composed of an opcode, destination address one, and destination address two, which may be replaced by a constant literal value.

This unit receives tokens from the matching store. The instruction is then fetched and an operation packet is formed containing the operands, opcode, destination address(es), and control bits. The operation packets are subsequently sent to the processing unit for execution.

v) The Processing Unit: is a multi-functional processor implemented by ten microprogrammed microprocessors. It utilizes an arbitration network and a distribution network [133] to control the input/output to/from the processing unit.

The units on the ring communicate with each other in an asynchronous, overlapping, pipelined fashion. The communication is performed in a handshaking fashion. For example, if all processors are busy, the instruction store stops routing packages and the ring operations are suspended.

The Irvine Data Flow Machine:

The Irvine data flow (Id) machine project originated by Arvind [7,9] at the University of California at Irvine and now is continued at MIT [8,11]. The data flow programs
executed on Id are processed by the U-interpreter which uses a sophisticated token "tagging" scheme to allow multiple tokens on a single arc. By allowing multiple tokens on the input arcs of an operator, it is possible to execute several activations of the operator in parallel.

Figure 2.5 depicts the general organization of the Id. The Processing Elements (PE) are data driven computers which are connected to each other through an NXN network. Each PE is a complete computer with an instruction set, a memory, and some special elements which will be explained shortly. In order to improve the communication, a short-circuit path is provided from a processing element to itself. Therefore, local communication does not require a passage thru the full NXN network. Because of the continuous modifications and experimentations, the designers did not specify a particular network. However, a packet communication network, which permits the asynchronous transmission of data, was considered to be most suitable [11].

A data flow program is uniformly distributed over the PE's. Even though only one physical copy of a program exists at all times, many activations of program blocks may be executed concurrently through a dynamic tagging scheme. The single execution of an operator (an activity) is uniquely identified by a tag which is composed of four fields <u,c,s,i>. The context field (u) uniquely identifies the context in which a code block is invoked. For example,
Figure 2.5: Block diagram of the Irvine data flow machine.
if the procedure \((f)\) with activity name \((v)\) calls one of its subprocedures \((g)\), the context field (activity name) of the activities in \((g)\) are prefixed by \((v)\), thereby creating a unique context for the activities within \((f)\). Item \((c)\) is a code block identifying a particular procedure or loop. Each processing element can hold up to eight blocks. Therefore, \((c)\) is a 3-bit field denoting the block containing the activity. Item \((s)\) is the statement number of the activity (an offset within the block). Item \((i)\) is an initiation number which identifies the loop iteration in which the activity occurs. For activities outside a loop, the value of \((i)\) is 1.

The basic operations of the PE's are similar to the operations of the Manchester data driven machine; namely, (1) bringing tokens with identical tags together; (2) determining the instruction indicated by the tag; (3) carrying out the operation; and (4) producing new tokens with proper tags. The internal organization of a processing element is represented in Figure 2.6. The system is a 5-stage pipeline consisting of: i) the input section; ii) the waiting-matching section; iii) the instruction fetch section; iv) the service section; and v) the output section.

Each processing element holds a program segment and has storage for I-structures. An I-structure is a set of components, each having a unique selector. For example, an array is an I-structure in which the array elements are the
Figure 2.6: A processing element in the Irvine data flow machine.
components and the selectors are the indexing integers. The difference between I-structures and other data structures such as streams and arrays is that in an I-structure, it is possible for a program to read an element before it exists. This is because the storage for an I-structure is allocated in advance and the instructions are executed asynchronously. Therefore, each element of an I-structure has a presence bit which indicates whether or not that element has been created. The attempted read operations must be deferred in the I-Read section until the corresponding elements are created. Hence after each write operation, the deferred read instructions must be checked for a possible valid read operation.

The input section receives tokens from either the network or the processing element itself. The tokens may be operation tokens manipulating the I-structures (I-structure instructions) or data tokens conveying the result of an operation. If the token is an I-structure instruction, it is routed to the service section for processing. Otherwise, it must be a result token and thus is sent to the waiting-matching section. The operations of the waiting-matching section are similar to the operations of the matching store unit in the Manchester machine. The operands of single-operand instructions are directly routed to the fetch section, while others wait for their matching operands. The instruction fetch section receives data
packets from the waiting-matching section. It then decodes the tag information \((u,c,s,i)\) to fetch the instruction to which the data packet belongs. Subsequently, an operation packet consisting of the opcode, operands, and destination information is formed and passed to the service section. The service station uses the information in an operation packet to execute the instruction and set the tag fields of the result tokens. Furthermore, it is responsible for the manipulation of the I-structures; namely, queuing the read operations in the I-Read section and checking them after each write operation.

The result tokens from the service station are passed to the output section. This unit either sends the tokens to the input section of the same processing element if the destination addresses are local or routes them to the network if the destinations are global.

The Data Flow Multiprocessor:

The overall organization of the data flow multiprocessor proposed by Rumbaugh [119,120] is depicted in Figure 2.7(a). The peripheral processor provides communication to the outside world. The programs are held in the program memory while the data structures are stored in the structure memory. Each activation processor is capable of executing an activation of a program procedure.
Figure 2.7(a): Structure of the Data Flow Multiprocessor.

Figure 2.7(b): An activation processor.
independent from other processors. However, when an activation becomes idle, it is temporarily stored in the swap memory through the swap network. This case occurs when some instructions in the activation must wait for the output of some other activation(s). The scheduler is responsible for the management of the memory hierarchy which is composed of the program memory, swap memory, and the activation processors. Furthermore, it provides communication among the activations by managing the procedure calls and returns. Several activation processors share one of the structure controllers which are used for the manipulation of the data structures.

As represented in Figure 2.7(b), the organization of an activation processor is similar to that of the TI data flow computer which will be discussed later. The activity list is a queue of enabled instructions. Each queue entry holds the address of the instruction in the instruction memory as well as the address of its operands in the data memory. After receiving the entry at the front of the activity list queue, the decoder fetches the fired instruction and its operands according to the contents of the entry. It then forms an operation packet and routes it to the proper functional unit. The results of the operations are sent to the updater unit which keeps track of the unavailable operands of each instruction in the enabling count memory. The result data is stored in the data memory and the count
of the destination instruction is decremented by one. If the count becomes zero, the instruction is enabled for execution. Therefore, an entry containing the address of the instruction and its operands is formed and placed at the end of the activity list queue. The circular pipelined operations, thus, continue.

The procedure calls and returns are handled by the call unit which is controlled by the scheduler. The operations of the scheduler and the procedure call/return mechanism are discussed in detail in Chapter IV. Similarly, the data structure operations are covered in Chapter V.

2. Class II:

The overall architecture of the machines in this class is presented in Figure 2.8. As opposed to the organization of Class I, the data memory is eliminated and the operands are contained in the instructions. During the instruction cycle, the fired instructions are routed to the processing unit and the result tokens produced by the processing unit are sent to the destination instructions. Some examples of the machines in this class include:
CLASS II

-Data and instructions are integrated into one unit

Figure 2.8: Organization of the Class II of the data driven machines.
The MIT Machine:

The MIT machine was proposed by Dennis in 1973/1974 [37,38,39,40,41,42,43]. It is a single process, stored program data driven computer whose general organization is depicted in Figure 2.9. As an instruction cell becomes enabled, the cell contents are routed to the proper processing unit through the arbitration network. The results of the boolean and scalar computations are sent to their destinations via the control and the distribution networks, respectively. These results may fire other instruction cells. The machine is composed of three major sections:

i) The Memory Section: This unit is a collection of instruction cells. A cell holds an instruction, its operands, and the destination addresses. Each operand is augmented with a number of control bits which indicate the characteristics of the operand, such as its type and its availability status.

The memory is passive in the sense that the instruction cells do not have computational power. However, some degree of processing capability is built into each cell to manipulate the control bits and to detect the firing condition. When a cell is fired, it produces an operation packet containing the opcode, an array of operands, and an array of destination addresses. The operation packet is
Figure 2.9: Central structure of the data flow processor.
routed to the proper processing element through the arbitration network. The correct order of execution is preserved through the acknowledge signal communication among the instruction cells [42]; i.e. a cell is not fired until it receives a signal from its successors indicating that they are ready to accept the result of the operation.

(ii) The Networks: These units are used to direct operation and data packets to one of several possible physical units. Figure 2.10 depicts the structure of the arbitration network which delivers operation packets from the memory section to the processing section. The network consists of arbitration, switch, and buffer units. The arbitration unit uses a round-robin discipline to pass the packets arriving at its input ports to its output port one at a time. The switch unit assigns a packet at its input, to one of its output ports according to the type of the operation. The partial decoding of the opcode by the switch units directs the packets to the proper processing unit. The buffer units are used as temporary queues to provide the smooth flow of data.

The arbitration network is divided into three stages. The first two stages of the network funnel operation packets into a smaller number of more heavily used channels. The switch units in stage two separate the packets for each processing unit. The output of the switch units are merged by stage three into a single stream for each processing
Figure 2.10: The arbitration network.

Figure 2.11: The organization of the control and distribution networks.
The distribution and the control networks provide an interface between the processing elements and the memory cells. They route data and boolean result packets to the proper destination cell(s). The structure of the control and distribution networks is shown in Figure 2.11. Stage one uses the destination address to route the control or result packets towards their address in the memory section. Stage two funnels the packets for each block of the memory, and stage three distributes the packets to their addresses within the blocks.

iii) The Processing Section: This unit is a multifunctional processor consisting of five units, including: a multiplier, an adder, a distributor, an integer processor, and a control processor. Each unit consists of a control section and an ALU section. The control section receives operation packets from the arbitration network. It then decodes the opcode and initiates appropriate signals to the ALU. Once the result of an operation is available, the control section forms a result packet and routes it to one of the control or distribution networks, according to the type of the result.

The MIT data flow computer discussed here is the representation of the Basic Data Flow Machine proposed by Dennis [40]. Obviously, the system cannot be considered as a general purpose computer because it lacks many
requirements of such a definition. For example, the memory is a small single-level memory, data structure handling is not discussed, multiprocessing is not supported, etc. It should however be noted that this basic organization has later been modified to improve the functionality of the system. The major changes include:

(1) Memory Hierarchy: In the Basic Data Flow Machine, each memory cell holds an instruction, its operands, the destination addresses, and a number of control bits. Moreover, the cells have a limited processing power for the detection of the enabled instructions. Therefore, memory cells are large and expensive, which in turn force the size of the memory to be small. To overcome the problem, Dennis and Misunas [41] added a two-level memory hierarchy to the system for the management of the memory section. The instruction cells are stored in a large passive secondary storage. When a cell is fired, the block containing that cell is brought to the main memory and processed. The details of these operations are covered in Chapter IV.

(2) Data Structure Manipulation: Figure 2.12 depicts an organization proposed by Dennis and Weng [43] which allows the manipulation of data structures. The execution controller decodes the fired instructions. If they are scalar operations, they are routed to the functional units. Otherwise, they are sent to the structure controller which manipulates data structures stored in the structure memory.
Figure 2.12: Data structure handling in the MIT machine.
The manipulation of data structures is discussed in Chapter V.

(3) Multiprocessing: Dennis has modified the system to allow multiprocessing capabilities [37,39]. The modified organization is shown in Figure 2.13. The memory is organized as a collection of cell blocks, each consisting of a number of instructions. The cell blocks are partitioned into several groups, such that each group has its own arbitration network and set of processors \((P_i, 1 \leq i \leq K)\). However, the result packets are available to all instructions through the distribution network. Different processes are assigned to different groups and are executed in parallel. The groups can communicate through the distribution network. Thus, the inter-connection length is the same for both the inter- and intra-group instruction communications.

The TI Data Flow Computer:

The Distributed Data Processor (DDP) project began in 1976 at the Texas Instrument Company [31,32,83,84]. The computer and its supporting software were operational in 1978. The system consists of 4 data driven processors and a host front-end computer as shown in Figure 2.14. The units are connected via a ring bus. Provisions are made for the addition of optional processing elements. The operations
Figure 2.13: Organization of the MIT data flow multiprocessor.
Figure 2.14: Block diagram of the TI data flow machine.
within each processor are based on the data driven concepts. The different procedures of a program are partitioned and allocated to different processors.

Figure 2.15 depicts the internal organization of a processing element. The major units of the system are: the operational unit, the update controller, the memory, and the pending instruction list. Each instruction has the following format:

\[
\text{[opcode, control bits, count, } \text{2 operands, 3 destinations]}\]

The count field holds the number of operands yet to arrive.

The update controller receives result tokens either from the operational unit or from another processor through the ring (Figures 2.14, 2.15). It then assigns the result to the proper destination(s) and decrements the count of the destination instruction(s) accordingly. If the destination instructions' count becomes zero, they are enabled and thus are placed in the pending instruction queue. If the queue is full, the fired instructions are linked to the last entry of the pending queue, using the control bits as a pointer field. Therefore, the capacity of the hardware resources cannot be overflowed. The entry at the front of the pending queue is routed to the operational unit to be processed. Thus, the execution cycle continues according to the above flow of operations.
Figure 2.15: Organization of a processing element in the TI data flow machine.
Data Driven Machine # 1:

DDM1 was developed by Davis [35] at the University of Utah. The system consists of a set of asynchronous modules, organized in a tree structure, such that each internal node is of out-degree 8. Each module (Processor Store Element (PSE)) consists of a processor module (P) and a local storage module (S). Figure 2.16 depicts the recursive structure of the system. The PSE at level (n) has eight successors at level (n+1). The processor module contains an Atomic Processor (AP) which has no substructures. Similarly, the storage unit consists of an Atomic Storage Unit (ASU) with no further substructures. The PSE's are capable of data flow processing.

An application program is passed to the root PSE for execution. It is then decomposed into concurrent tasks which are assigned to the free PSE successors. The process recursively continues until either the task is not decomposable or until there are no free PSE successors. The allocated PSE's then execute different blocks of the program in parallel and in a data flow fashion. Messages (instructions or data) are passed from one PSE to another as a serial character string with a header field which provides destination address(es) and decomposition information (as set by the compiler to reduce runtime analysis of the program).
Figure 2.16: Recursive definition of DDM1.
Figure 2.17 depicts the internal organization of a PSE. The Input Queue (IQ) contains instructions (as decomposed by the predecessor PSE) or data (as produced by the predecessor PSE). The Instructions are sent to the ASU to be executed when they are fired. The data tokens are routed to their destinations. The destination instruction is checked for firing conditions. If the instruction is enabled for execution, it is sent to the AP to be processed. Otherwise, the data is stored as an operand of the instruction. Once an instruction is executed, one copy of the result is generated for each destination address. If the destination is local to the executing PSE, the result packet (value and destination) is stored in the Agenda Queue (AQ) to be sent to its destination in the next cycle. If the destination of the result is another PSE, it is placed in the Output Queue (OQ) to be sent to the predecessor PSE or the switch to be sent to a successor PSE. The AP services the switch, the AQ, and the IQ in descending order of priority. Thus, as much work as possible is done in the low level fast areas of the machine before any work is accepted from above.

DDM1 has a recursive structure consisting of asynchronous cells. This organization allows an easy extension of the system by adding more cells to the bottom of the tree. Also, regular use of a small number of module types make the architecture suitable for VLSI implementation. However, DDM1 resolves the limited I/O pin
Figure 2.17: Structure of a processing element in the DDM1.
limitation problem of the VLSI technology at the expense of the serial information flow among the PSE's. This discipline may degrade the performance of the system. Another disadvantage of the DDM is the time it requires for program decomposition during the execution. In order to facilitate the run-time decomposition, additional information is supplied by the compiler. Such information must be carried out during the execution, resulting in redundant storage. In addition, the fixed-tree organization of the system prohibits the optimal allocation of the processing power. For example, it is not possible to reallocate the unused PE's to the tasks which are to be carried out by the other branches of the tree.

Disadvantages of the Existing Data Driven Machines:

The first two classes of the data driven processors suffer from two major disadvantages:

(1) Lack of VLSI Suitability:

It is predicted that the 5th generation computers will extensively utilize the VLSI technology [104,130]. Ravishankar has shown that data driven computations have many features that make them almost ideally suited for VLSI implementation [114]. However a common problem in the data
driven machines in Classes I and II is that some of the proposed computers are not fitted for VLSI implementation. There are three factors contributing to such a problem:

i) Some of the data driven machines were designed before the widespread application of the VLSI in computer technology, and therefore, they do not comply with the design rules imposed by the VLSI. Examples of such machines are: the MIT machine [40], the TI machine [31], the Manchester machine [133], etc.

ii) In developing specialized hardware for VLSI implementation, a technical constraint is that the system should be composed of a few basic structures which are duplicated many times [56]. The memory system of the existing data driven machines, because of their regular organization, is suitable for VLSI implementation. Nevertheless, the implementation of the whole system in VLSI becomes prohibitively expensive because of the irregularity of different components. The Irvine machine [11] and the Data Flow Multiprocessor by Rumbaugh [120] use several copies of the same processing element. However, the units within each processing element are very complex and require internal communication.

iii) VLSI technology provides great on-chip capabilities, while off-chip communication is its major drawback (long delays and limited number of I/O pins) [57, 114]. In many of the data driven computers, such as the
MIT [38] and the Irvine [11] machines, the modules communicate through operation packets or data packets. Because of the high degree of information which is carried by the packets, the size of these packets is in excess of 128 bits. In order to have a fast communication system, the packets must be transferred in a bit-parallel fashion. However, due to the limited number of I/O pins imposed by the VLSI technology, this cannot be achieved.

It must be mentioned that a few of the existing data driven machines are suitable for VLSI implementation. However, they suffer from other disadvantages. For example, the latest version of the MIT machine (Data Flow Supercomputer [37,39]) uses communication paths of equal length for both the inter- and the intra-block communications. This is in contrast to the "locality of effect" principle in a data flow environment. This principle implies a much higher level of intra-block communications. The Utah machine (DDM1) is suitable for VLSI implementation, but suffers from inefficient use of the hardware resources [35].

(2) Utilization of Separate Memory and Processing Units:

Most of the proposed data driven computers utilize disjoint memory and processor units. The separation of the processing power from the memory causes two major drawbacks:

i) A memory access overhead always exists. Since the
machines in Class I utilize separate data and instruction memories, the memory overhead of the systems in this class is more critical than those in Class II. For example, in the Data Flow Multiprocessor proposed by Rumbaugh [119], the data memory must be sequentially accessed twice, if an instruction addresses two destinations. The enabled instructions are also sequentially accessed from the instruction memory according to their position in the enabled instruction queue. A similar problem exists in the TI machine [32].

ii) A fast interconnection network is required to connect the processors to the memory cells and vice-versa. Furthermore, in Class I of the data driven computers, the interconnection from the processors to the instruction cells must consist of two parts: a) a network connecting the processing unit to the data memory; and b) a network connecting the data memory to the instruction memory. The simulation results have shown that the arbitration network in the MIT machine [102] and the match unit in the Manchester machine [133] are the bottle-necks in these systems, and therefore, a faster communication is required.

It is worthwhile to note that the separation of the memory and the processor units has the advantage that it reduces the cost of the system by sharing processing resources among instruction cells. However, the cost factor has become less important because of the advances in
3. Class III:

The Class III of the data driven machines eliminates the two major problems associated with the previous two classes. In order to reduce the communication delay between the memory and the processor units, it seems promising to distribute the processing power among the memory cells. This trend of thought indicates that the future generations of the data driven computers will use processing elements as the basic building block for the whole system. Such a regular and repetitive organization is most suitable for VLSI implementation.

Figure 2.18 presents the overall structure of the machines in the third class. These machines will have a "cellular architecture", composed of a number of identical cells. Each cell holds an instruction and its operands. When the operands are available, the cell directly executes the instruction and routes the result values to the succeeding instructions which are stored in other cells. As an example of the machines in this class, the organization of the proposed architecture is fully discussed in detail in the next chapter.
CLASS III

-Data, instructions, and processors are all integrated into one unit

Figure 2.18: The overall organization of the Class III data driven machines.
3. THE BLOCK DRIVEN PROCESSORS

In order to achieve maximum throughput, the organization of the computer systems has evolved from simple uniprocessors to complex distributed processors such as: array processors, multiprocessors, pipelined processors, data driven processors, etc. At the same time, the programming environment has been gradually developed from low-level machine programming to structured programming, modular programming, stepwise refinement programming, top-down design and programming, etc. Such a trend is due in part to the efforts for closing the semantic gap between the requirements of the applications and the underlying structure of the computer systems.

Data driven languages are no exception to the evolution of the programming languages. The locality of effect in data driven programs implies that the instructions should not have unnecessary far-reaching data dependencies [1]. Thus, data driven languages are highly block-oriented. As such, the data driven programs provide an opportunity for the parallel execution of the independent blocks. Such a discipline has given rise to another class of data driven
machines called the Block Driven Processors in which the firing of an instruction is subject to the firing of the block to which it belongs. In the remainder of this section we will represent the definition of a block as defined by Chang and Fisher [29]. The block firing rule will then be introduced.

The computational steps in a block driven program can be divided into branch computations and joint computations. The branch computations involve a sequence of chain computations with the constraint that each succeeding computation requires one data item from its previous computation. A joint computation is defined as a pair of branch computations which are linked together by a computation node at their ends. A joint computation forms a "BLOCK". Figure 2.19(a) shows an example of a joint computation.

We now introduce two different disciplines to define block firing conditions for block driven programs. First, a block becomes enabled for execution if every input arc to the block contains a data token and there are no tokens present on the output arc out of that block. This scheme is very similar to the firing conditions defined for an operation in the data driven programs. Second, a block becomes enabled for execution as soon as a subset of the instructions within the block is enabled. The particular subset of the instructions consists of one or both of the
Figure 2.19(a): An example of a joint computation.

\[ x = f_2(a_2, f_1(a_1, a_0)) \]
\[ y = g_2(b_2, g_1(b_1, b_0)) \]
\[ z = h(x, y) \]

Figure 2.19(b): Steps in an interleaved computation.
instructions at the "front" of the branch computation(s). The firing of those instructions is subject to the firing rules in data driven programs. For example, the block of Figure 2.19(a) becomes enabled if \( f_1 \) or \( g_1 \) or both are enabled. The first approach has the advantage that once a block is fired, it can be processed without any interrupts or delays. This is due to the fact that all the input tokens are available before the execution of the block is started. However, the disadvantage of this method is that the execution of some fired instructions is delayed because of the time it takes to have all the input data available. The second approach solves this problem by immediately executing a block as soon as a subset of its instructions become enabled. Nevertheless, the disadvantage of it is that the block processor to which a fired block is assigned may remain idle, awaiting data tokens to arrive for non-fired instructions.

In order to execute a block (joint computation), Chang and Fisher [29] have proposed a processing pair organization in a ring format. Each element of the pair consists of a Memory unit (M) and a Processor unit (P). The instructions in the first element of the pair are passed from its M unit to its P unit for execution. The results are passed to the M unit of the second element of the pair. This element will, in turn, route its results to the M unit of the first element. Figure 2.19(b) depicts the operations of the
processing pair for the block of Figure 2.19(a). Chang and Fisher [29] have suggested the following advantages for such an organization: i) the succeeding instructions can be driven by local data with an address field of minimum bits and ii) reliability can be greatly improved by connecting the processors in this manner.

Figure 2.20 represents the general organization of the block driven data flow machine proposed by Chang and Fisher [29]. A fired block passes through a pipe of instruction buffer units to the block control master. This unit then distributes the blocks among the processing pairs and the execution of independent blocks commences. If the results are to be sent to a block in the block section, it is routed to its destination through the distribution tree. Otherwise, the buffering tree routes the result to the destination processing pair.

The buffering tree is a complete binary tree with processing pairs used as its terminal nodes. It works on a 2-phase push-pull mechanism. In the push mode, data are pushed forward from the lower level nodes to the higher level nodes: whereas, in the pull mode, data are pulled backward in the opposite direction.

The performance of the system highly depends on the modularity of application programs. Maximum throughput occurs when the local communication rates are much greater than global communication rates.
Figure 2.20: Block-driven data-flow architecture.
Machine d'Assignation Unique Dynamique (MAUD) [91] developed by Lecouffe in 1979 is another example of block driven machines. The system consists of: i) a set of execution processors which execute the blocks; ii) an updating processor which updates the waiting blocks with data tokens produced by the execution processors; iii) a builder processor which builds a block using a library of existing blocks when an execution processor calls the library routine; and iv) a number of buffer memories which facilitate the communication among the different units of the system. The general organization of MAUD is represented in Figure 2.21.
Figure 2.21: The general organization of the Machine d'Assignation Unique Dynamique (MAUD).
4. SUMMARY

In this chapter, a basic set of technical terms was introduced through an introduction of the data flow computations and a survey of some of the existing data driven machines. A classification of the data driven processors based on the relationship among the data, instructions, and processors has been proposed.

In the machines of the first class, the instructions and the data are stored in separate storage units. The machines of the second class store the data as the operands of the instructions in the same memory unit. These two classes suffer from the lack of suitability for VLSI implementation and the memory access overhead during the execution. The third class of the data driven machines use a cellular architecture to resolve these shortcomings by distributing the processing power among the memory cells.

In addition to the above classification, this chapter contains a discussion of the block driven processors as a special case of the data driven architecture. In a block driven machine, program blocks are fired when their input arguments are available. Therefore, many blocks may be executed concurrently.
CHAPTER III
THE PROPOSED ARCHITECTURE

This chapter introduces a new data driven machine based on the characteristics of the Class III of these machines which was proposed in Chapter II. In addition, the Wafer-scale Data Driven Multiprocessor (WDDM) [79] incorporates principles of the block driven computations. WDDM is designed with three objectives in mind: i) To take advantage of the most recent developments in computer technology; ii) To overcome some of the shortcomings of the existing data driven computers; and iii) To take full advantage of the inherent parallelism in a program at the data, instruction, and block levels. These objectives are further explored below:

i) VLSI technology: The VLSI technology exploits very-large-scale integration to produce high performance computer components at a modest cost with greater functionality than conventional machines [52, 54, 56, 99, 127, 129, 130]. A VLSI implementable algorithm (or system) must consist of a few simple and regular sets of operations (or
building blocks) which can be replicated in time or space. Furthermore, since VLSI offers a high logic-to-pin ratio, the number of I/O pins per chip becomes limited. Such a limitation can be a handicap in setting up fast interconnection networks among modules of a system.

In order to comply with the constraints imposed by the VLSI technology, WDDM is designed so that it consists of only a few building block elements which are duplicated many times across the system. Additionally, to solve the interconnection problem, we have utilized one of the most recent developments in the VLSI technology, namely wafer-scale integration [94,25]. This technique uses the entire wafer, instead of dicing, to condense more functionality into a single device. The cable connector delays of multi-chip systems are therefore eliminated and packaging problems are reduced. The disadvantages associated with this technology, such as isolation of faulty elements, higher power requirement, and potential timing problems, are expected to be resolved through advances of technology in near future [94]. The characteristics, advantages, disadvantages, and current research efforts in the area of wafer-scale integration are studied in depth in Chapter VI.

ii) Eliminating previous shortcomings: In Chapter II, we associated two major disadvantages to the existing data driven machines; namely, the lack of suitability for
implementation by the current technology and the utilization of separate memory and processing units. In contrast to some of the proposed data driven machines, we have been concerned with the implementation aspects of the proposed architecture. Therefore, as previously mentioned, an attempt is made to comply with the constraints imposed by the current technology. Furthermore, we have incorporated a high degree of modularity in the design of the system. This approach has numerous advantages including: a) independent design of the modules with the goal of maximizing the performance of each module; b) taking advantage of the latest technology for the implementation of each module; c) allowing overlap of operations among the modules; and d) ease of system expansion by adding additional modules.

The problems caused by separation of the memory unit from the processing unit have been eliminated by the distribution of the processing power among the storage cells. Naturally, because of the economic reasons, we would still utilize a mass storage media for storing the passive portions of a program. Thus, only the active program blocks will be in the more expensive high speed memory modules with the processing power.

iii) Taking advantage of embedded parallelism: The proposed architecture takes advantage of the embedded parallelism in a program by providing hardware facilities to: a) process arrays in parallel in a SIMD fashion; b)
process independent instructions concurrently in a data flow fashion; and c) execute independent program blocks (such as loop iterations) simultaneously in a multiprocessing fashion.
1. THE PROPOSED ARCHITECTURE

**System Overview:**

Data flow programs are determinate, functional, and block-oriented with a high degree of "locality of effect" and virtually no global variables [1, 12, 64, 95, 101]. Such characteristics imply that: First, independent blocks of a program can be executed in parallel. Second, if possible, all the activations of the same block can be executed concurrently. For example, in the absence of dependencies, all the iterations of a loop can be carried out simultaneously. Third, the degree of communication among the instructions of a block is much higher than the inter-block communication. In order to close the semantic gap between the high-level data flow languages and the underlying computer architecture, we have incorporated all such characteristics in our hardware design.

Figure 3.1 depicts the general organization of the WDDM. The system consists of m identical processing modules, each on a silicon wafer, a host module, and a data structure module. The units communicate through a double
$PM_i$ = The $i^{th}$ Processing Module.

Figure 3.1: The overall organization of the proposed system.
star interconnection network, with the host module and the data structure module as the centers of the stars. The two centers can communicate with each other via a dedicated bus. Each processing module is capable of executing an independent procedure in data flow fashion, thus giving the system multiprocessing capabilities. Such a scheme is a step towards closing the above mentioned semantic gap. For example, by placing the processing modules on silicon wafers, two advantages have become feasible:

i) The instruction level communication within a block is carried out at the on-chip level (on a wafer) speed. The delays resulting from off-chip communications are eliminated. Since each program block is assigned to a processing module, this is a step in the direction of reducing the semantic gap by allowing potentially faster speeds for intra-block communications.

ii) The pin limitation problem is solved for both the local and the global communications. For the communication on a wafer, the I/O pins are eliminated and the bandwidth can be increased significantly by increasing the number of communication bits. For the global communication among the system modules, the increased area of the wafers allows a high number of I/O pins (up to 2000 [94]) as opposed to multichip systems (up to 256 for Pin-Grid Array pakages [21]).
The Choice of the Double Star Interconnection Network

As depicted in Figure 3.1, WDDM is composed of a number of processing modules which are connected to each other through a double star interconnection network. The program blocks are assigned to the processing modules and their execution may proceed concurrently. These blocks communicate to each other via a procedure call/return mechanism which is handled by the host module. The criteria for selecting such an interconnection scheme is explained in the following segments:

1) Resource Management: The optimal utilization of the limited processing resources is one of the major goals in the design of a new system. In WDDM, passive program blocks are stored in the host module and assigned to a processing module only when they are enabled for execution. Furthermore, whenever an executing block becomes inactive, it is returned to the host module so that the processing module can be assigned to another enabled block. It is obvious that in such an organization, there is a high degree of traffic between the host and the processing modules. In addition, the type of communication is bulk transmission of the blocks, requiring a direct link between the two transmitting stations. These requirements make a star network, with the host module at the center, the natural choice.
The processing modules are also used to process data structure elements in a parallel fashion. Thus, the need for bulk transmission of data structures between the data structure module and the processing modules imposes the utilization of another star network with the data structure module as the center. In addition, the host and the data structure modules communicate via a dedicated bus in order to schedule SIMD (Single Instruction stream-Multiple Data stream) type data structure operations such as FORALL and OVAL (discussed in Chapter V).

ii) Data Flow Semantics: In a data flow and block driven environment, the only means of communication among the program blocks is a procedure call/return mechanism [1, 12, 29, 91, 95]. In WDDM, program blocks may be in the processing modules or in the host module. Moreover, as it will be discussed later, within the host module the blocks are distributed between a high speed cache memory and a mass storage main memory. Therefore, all the communication among the program blocks must be routed to the host module, where the destination block is located and the actual data transmission is performed.

iii) Ease of System Expansion: One desirable feature in any multiprocessing system is the capability of the system for future expansion, as the need arises. The star network seems to be ideal for this purpose. The processing modules can easily be added to the system without the need
for an extensive hardware or software modifications for the network operations.

**iv) Fault-tolerance:** The system's computational power is distributed over the processing modules. Therefore, the system will still be operational even if a subset of the processing modules have failed. The host module contains the operating system and provides communication to the outside world. Naturally, the continuous functioning of the host module is so critical that its failure halts the whole system.

One of the major drawbacks of a star interconnection network is that it is potentially prone to creating a bottleneck at the center station. However, as we will discuss in the next chapter, many measures can be taken to reduce such a possibility. For example, we have proposed several steps for reducing the overhead and providing opportunities for overlapped operations in the host module.

**The Processing Module Organization**

Figure 3.2 depicts the general organization of a processing module on a wafer. The instructions of a data flow program block are distributed among the n Elementary processing units (E-unit). When the input data for an instruction become available, two cases may occur. First, if the instruction is a simple operation such as fixed-point
Figure 3.2: The overall organization of a processing module.
addition, it is immediately executed by the E-unit and the result token is sent to the "sub-net". Second, for complex operations such as floating-point division, the E-unit forms an operation token and routes it to the corresponding Functional Unit (FU) via the sub-net. The sub-net receives tokens from the n E-units, the FU's, the host module (I₂ port), and the data structure module (I₁ port). It then passes them to the queue, the host module (O₂ port), or the data structure module (O₁ port) according to their destination. The queue places the tokens on the common bus, which in turn distributes them among the E-units and the FU's. The Active/Inactive Detector (AID) unit detects the status of a processing module when it becomes inactive. The architecture of each unit of a processing module is covered in details in Chapter VI, where the VLSI design of the system is proposed. The general functions of the units are discussed next:

1. The Elementary Processing Units:

The organization of an Elementary processing unit (E-unit) is shown in Figure 3.3. Each data flow instruction is assigned to an E-unit. The E-units consist of two I/O ports, a controller, and an elementary processor and have storage for the opcode, the input values (operands), and the destination addresses.
Figure 3.3: The organization of an elementary processing unit.
The controller is the central control for an E-unit. It performs 3 functions:

i) The controller matches the address(es) of the token in the input port against the E-unit's IDentifying number (ID). In case one of the addresses matches the E-unit's ID, the value token is accepted as an operand.

ii) It keeps a count of the unavailable operands. As an operand arrives, the count is decremented by one. When the count is zero, the controller fires the operation.

iii) If the instruction is a simple operation such as addition, subtraction, AND, OR, comparison, etc., the controller initiates the elementary processor for the execution of the operation. The result data token is sent to the output port from where it is submitted to the sub-net.

There are six types of tokens which are used to convey the information among different system units. They consist of the data, operation, call, data structure, and two return tokens.

**Data token:** These tokens are produced as the result of the execution of the instruction by the E-unit. A data token has the following format:

\[ \text{[value, dest}_1, \text{dest}_2, \ldots, \text{dest}_k], \]

where, \( \text{dest}_i \) \( (1 \leq i \leq k) \) denotes the \( i \text{th} \) destination address. Each data value is augmented with a tag bit indicating whether it is a scalar value or a pointer to a data
structure. In order to reduce the number of tokens which travel through the sub-net, we will **NOT** have a data token for each destination address. Instead, one data token is routed for all the destinations. Once a data token is placed on the common bus, each E-unit can check all the destination addresses in that token for a match against its own ID.

A destination address is composed of two segments. The first segment identifies the ID number of the destination E-unit, while the second segment indicates the operand number within the instruction.

**Operation token:** For complex operations which cannot be carried out by the E-units, the controller forms an operation token and assigns it to one of the FU's. These tokens are routed to the FU's via the sub-net, where the instruction is executed. An operation token is composed of:

\[ [\text{FU-Addr}, \text{opd}_1, \text{opd}_2, \ldots, \text{opd}_j, \]
\[ \text{dest}_1, \text{dest}_2, \ldots, \text{dest}_k] \]

The FU-addr is the address of the functional unit which is to carry out the operation. Therefore, if an E-unit is unable to execute an instruction, it will decode the opcode and sets the FU-addr accordingly. The \( \text{opd}_i \) denotes the \( i \)th operand value of the instruction and \( \text{dest}_m \) indicates the \( m \)th destination address (\( 1 \leq i \leq j \) and \( 1 \leq m \leq k \)). The result of the execution of the instruction by the FU is a data token which carries the result to the perspective destinations.
Since the operation token is larger than the data token, it is passed to the network in two stages. Of course, each segment will contain the destination address for the token (FU-Addr) and a tag field which shows the association of the two segments. The association tag is the ID number of the E-unit from which the segments are originated. This information is used by the destination unit (FU) to reconstruct the token.

**Call token:** The call token is produced when a CALL statement is fired. The token is routed to the host module, where the called block is enabled. A call token is similar to an operation token with the following exceptions: the FU-Addr field is replaced by the called block label, the operand fields are substituted by the passed arguments, and the destination fields are replaced by the point of return:

\[ \text{[label, list of arguments, point of return].} \]

The point of return consists of the label of the calling block and the E-unit identifying number(s) to which the result is to be routed. Similar to the operation tokens, the call tokens are also routed in two stages.

**Data structure token:** When an instruction requires the manipulation of data structures, a data structure token is formed and sent to the data structure module. These tokens have the format shown below:

\[ \text{[opcode, PM, DS}_1, \ldots, \text{DS}_k, \]
\[ \text{dest}_1, \text{dest}_2, \ldots, \text{dest}_m]. \]
The opcode field indicates the type of operation to be performed on the data structures. The PM field is the processing module number from which the operation is initiated. This information is used to return the result of the operation to the corresponding processing module. The DS_i (1≤i≤k) is the name or pointer of the i^{th} data structure involved in the operation. The dest_j (1≤j≤m) is the j^{th} destination address to which the result of the operation is to be sent. Due to the length of these tokens, they are routed in two segments.

**Return tokens:** In addition to the above tokens, there are two more tokens which are special cases of a data token. A procedure return token is generated by a RETURN statement. It contains the output of a called procedure and is composed of the result value and the point of return address(es). The result of a data structure operation may be a single value or a pointer to a data structure, distinguished by a tag bit. In either case, the result is routed to the destination address(es) by a data structure return token.

Since there are six different types of tokens travelling through the sub-net, each token is augmented by a set of 3-bit control bits for identification. The information in the control bits is used by the sub-net to route the tokens to one of the queue, the host, or the data structure module ports. In addition, once a token is placed on the common bus, the E-units will first check the control bits.
If it is a data token, the E-units will try to match the destination addresses against their own; otherwise, they will ignore the token. Similarly, the FU's will process the operation tokens and reject the input data tokens.

2. The Sub-net:

The intra-block interconnection network in WDDM requires a network which has a dynamic topology, provides decentralized control, allows asynchronous communication, and utilizes a packet switching methodology. The dynamic topology must be used because of the dynamic nature of the proposed environment. The decentralized control and asynchronous control are desirable since the E-units execute the instructions concurrently and asynchronously; i.e. as soon as their operands become available. Finally, the packet switching methodology is to be used because the post-office type networks are most suitable for data flow operations [31]. In a post-office type network, a sender delivers a message to the network for later delivery without the knowledge or permission of the receiver. This is in contrast to the telephone type networks (circuit switching) in which a continuous path from the sender to the receiver is established first and then the information is transmitted.

The interconnection networks with the above properties
are called packet switching networks \([44,53,136]\). The type of the switching elements used in these networks is shown in Figure 3.4. For \(n \geq 2\), the \(n\)-input, \(n\)-output switches are called square switches; the \(n\)-input, 1-output elements are arbitration switches; and, 1-input, \(n\)-output elements are called the distribution switches. Once a packet has arrived at an input terminal, the switch first uses the information in the packet to select one of its output terminals and then transfers the packet to that terminal (decentralized control).

If the number of input and output terminals of a packet switching network is equal, the network is a square packet switching network. If the number of its input terminals is larger than its output terminals it is called an arbitration network. Otherwise, it is named a distribution network.

The sub-net in a processing module is an arbitration network with \(n+2\) inputs and 3 outputs, where \(n\) is the number of E-units and FU's. It receives tokens from the E-units, the FU's, the host, and the data structure modules. The sub-net's output terminals are connected to the queue, the host, and the data structure modules.

Figure 3.5 shows the structure of the sub-net. The network consists of a number of arbitration switches and a square switch. The arbitration switches funnel the flow of tokens into the square switch. Each arbitration switch accepts tokens at its input terminal and outputs them one at
a) Square Switch

b) Arbitration Switch

c) Distribution Switch

Figure 3.4: Different types of switching elements.
Figure 3.5: The structure of the arbitration network.
a time. It uses the information in the 3-bit control field to give the tokens destined for the E-units and the FU's the highest priority and the host and data structure destined tokens the lowest priority. This priority scheme stems from the data flow semantics which require a higher and faster level of communication for intra-block operations as opposed to inter-block operations. Furthermore, the tokens destined for the E-units contain some result values which may fire many waiting E-units. Thus, delaying these tokens will delay the instructions whose execution is pending the arrival of the tokens.

The two stages of the square switches distribute the tokens to the queue or one of the two output ports in accordance with the token's 3-bit control field.

There are many ways to implement an arbitration network. Two of the most frequently used arbitration networks are shown in Figure 3.6 [44]. In the first method, the square switches are placed before the arbitration switches. Thus, the packets are first distributed among several streams and each stream is then funneled into a single output stream. This method has two disadvantages. First, the network area is increased. Second, the crossing of a large number of links between switches may pose some technological impossibilities during the implementation. This is because all the communication links are to be carried out at the metal layer during the VLSI implementa-
Figure 3.6: Nonsquare packet switching networks.
tion. In the second method which is similar to the sub-net structure, the arbitration switches are placed before the square switches. Therefore, the area is decreased and the problem of the cross-over lines is resolved.

In addition to the above considerations, Dias and Jump [44] have shown that the use of data buffers between the stages of packet switching networks can improve their performance. In fact they have concluded that buffering makes the performance of this class of networks comparable to that of crossbar switches which are considerably more expensive. Therefore, in the implementation of the sub-net we will use at least one level of buffering between the stages.

3. The Functional Units:

The Functional Units (FU) are a collection of processing units, each capable of performing one complex operation such as floating-point addition, subtraction, multiplication, etc. As an example of these units, we have introduced the architecture of a fast systolic multiplier unit in [77,80].

Figure 3.7 depicts the general organization of a functional unit. The input queue performs two functions. First, it checks the control bits of the input token to see if it is an operation token. In case of a mismatch, the
Figure 3.7: The overall organization of a functional unit.
token is ignored. Otherwise, the FU-addr field of the operation token is compared against the FU's ID number. A match indicates that the operation is to be performed by this unit; otherwise, the token is ignored. The second function of the input queue is to simply queue the operation tokens since an FU may receive tokens in a faster rate than it can process them. The processing unit uses the operand values provided in the operation token to execute the instruction. The generated result is augmented with the destination addresses, obtained from the operation token, to form a data token which is output to the sub-net.

It may seem advantageous to directly assign the complex operations to the FU's when a block is assigned to a processing module. In that case, only one hardware resource is used for each complex operation and only one pass through the network is sufficient to route the result to its destination. However, such an organization has several serious drawbacks. First, several instructions may share the same FU. Hence, an FU must be capable of detecting the firing condition for each of the instructions. This is not only time-consuming, but it also increases the complexity of the FU's. Second, the addressing scheme becomes complicated since each address must consist of the block name, instruction address within the FU's, and the operand number within the instruction. Third, the complexity of the compiler operations is greatly increased since the compiler
must separate the simple and complex operations and set the destination addresses accordingly.

4. **The Queue:**

The queue is used to regulate the flow of tokens to/from the E-units and the FU's. The need for the queuing arises from the fact that the transmission time through the sub-net is less than the execution time of the processing elements (the timing of the units are discussed in Chapter VI). Therefore, the output tokens from the sub-net must first be queued and then placed on the common bus one at a time. The interval between broadcasting the tokens on the common bus must be equal to the time required by an E-unit or an FU to check the token's destination address(es).

The simulation results presented in Chapter VII indicate that the maximum queue length ranges from 1/3 to 1/2 of the number of E-units. Since we are expecting the number of E-units to be in the range of 32 to 64 (refer to Chapter VI, where the geometry area of the wafers are discussed), the size of the queue will be well within an acceptable range. Nevertheless, as a safety measure, the units on a processing module operate in a handshaking fashion. Thus, if the queue becomes full, the flow of tokens through the sub-net is temporarily suspended.
5. The AID Unit:

The Active/Inactive Detector (AID) unit determines the status of a processing module to the host module. A processing module becomes active when it begins the execution of a program block. An active processing module becomes inactive when it calls another block and the execution of some instructions within the block is pending upon the return result from the called block; i.e. there are no instructions which can proceed their execution.

An AID unit keeps a count of the active elements on a processing module. When the count becomes zero, the processing module has become inactive. Each element of a processing module sends an active or inactive signal to the AID unit at n units-of-time intervals; where, n is the units of time required by the AID unit to update its counter and check for the inactivity condition. Hence, if there are no active units on a wafer in two consecutive intervals, the unit is inactive. In that case, the AID unit sends an appropriate signal to the host module. The inactivity condition should be checked in two consecutive intervals to allow enough time for token transmissions among the units (e.g. from the E-units to the sub-net, from the sub-net to the queue, etc.)
The Host and the Data Structure Modules

The details of the organization and the operations of the host module are discussed in Chapter IV. Similarly, the data structure module is covered in Chapter V. However, in the remainder of this section we will briefly introduce a general view of the functions of these two modules.

The Host Module:

This module is composed of a host computer and a main memory unit (Figure 4.3). The host computer contains a cache memory and a number of tables for performing memory management tasks. In addition, the host provides I/O communication to the outside world. The memory management tasks include memory hierarchy operations, handling of the procedure call/return mechanism, and allocation of the enabled blocks to free processing modules. The main memory holds program blocks as they are generated by the compiler.

The system uses a 3-level memory hierarchy organization, with the main memory at the bottom and the processing modules at the top of the hierarchy. The main memory is a mass storage core memory which holds passive program blocks. The processing modules execute the active blocks which have been fired. The inactive blocks are stored in the cache memory awaiting the result values to be
returned from the called blocks.

**The Data Structure Module:**

The implementation of the data structures in a data flow environment poses some problems since conceptually there is no shared memory. For example, if an operation node only modifies a single element of a data structure, the entire structure must be passed to the succeeding node, with that element modified. In WDDM, the data structure module is used to facilitate the implementation of the data structures.

The module is composed of a data structure memory unit and a data structure controller. The data structure memory is an interleaved memory which holds the data structures. It provides simultaneous accesses to several data items and allows parallel transmission of the elements of a data structure to a processing module. The data structure controller is responsible for carrying out the data structure operations which include duplication of the data structures, management of the data structure memory, and execution of the constructs such as FORALL. In a FORALL operation, a single instruction is applied to every element of a data structure in a SIMD fashion.
2. SUMMARY

This chapter introduced the design of a new data driven multiprocessor which incorporates the characteristics of the third class of the data driven machines as well as the block driven principles. The proposed architecture eliminates some of the shortcomings of the existing machines, takes advantage of the current technology, and achieves parallelism at the data, block, and instruction levels. In addition, the semantic gap between the data flow languages and the underlying architecture is narrowed by: i) allowing concurrent execution of program block activations and ii) providing shorter and faster communication lengths for the intra-block versus a longer one for the inter-block communications.

In order to comply with the current technology, the proposed model is composed of a few building block elements which are replicated across the system. These elements consist of the E-units, the FUs, the switching elements, and the buffering units. Furthermore, it is suggested that the processing modules be implemented on silicon wafers using the wafer-scale integration technology. This
technique eliminates the delays and the I/O pin limitation problems of multichip systems.

The general organization and functions of the system units were discussed in this chapter. The host module is studied in details in Chapter IV, while the data structure module is covered in Chapter V. The VLSI design and the timing/geometry of the system units are introduced in Chapter VI.
CHAPTER IV

MEMORY MANAGEMENT OPERATIONS

This chapter discusses issues related to the management of the memory and the execution of program blocks in WDDM as well as some of the existing data flow machines. The term "memory management" in this manuscript has a broader meaning than what has been defined in the conventional systems. In addition to the usual memory management operations, the term includes the memory hierarchy operations, allocation of enabled blocks or instructions to the processing modules, and handling of the procedure call/return mechanisms.

One of the major problems of memory management in a data flow environment is the optimal assignment of operators (or program blocks) to the processing elements in order to minimize the execution time. Mundell, et al. [106] have indicated that the problem of obtaining optimal allocations is NP-complete and hence, the cost of doing so is prohibitive. This is due to the fact that several factors affect the overall execution time of a program. These factors include computation time of the operations,
transmission time of the operation and the result packets, and concurrent and asynchronous nature of the operations.

However, in order to improve the performance, many measures may be taken into consideration in the design of processor allocation schemes. For example, one must take full advantage of the inherent concurrency in the system. Furthermore, the amount of interprocessor communication should be reduced wherever possible. Finally, the amount of run-time analysis for allocation must be minimal.

In this chapter we will first cover the previous work in this area and then discuss the memory management operations in WDDM.
1. THE PREVIOUS WORK

Unfortunately, most of the published literature in the data flow area is either related to the architecture of the data flow machines or the data flow languages. In this section, we will represent the memory management operations in two of the existing data flow machines; namely the MIT machine [41] and the Data Flow Multiprocessor proposed by Rumbaugh [120]. The architectures of these machines were discussed in Chapter II. We assume that the reader is already familiar with the organization and function of the major units of these two computers. Some other material related to the memory management operations in a data flow environment can also be found in [10, 19, 28, 87, 106].

Example 1: The MIT machine:

Figure 4.1 represents the incorporation of a 2-level memory hierarchy in the organization of the MIT machine as proposed by Dennis and Misunas [41]. The main memory is composed of a number of cell blocks such that each block contains a number of instruction cells. The cell blocks act
Figure 4.1: Organization of the MIT machine with auxiliary memory.
as a cache for the most active instructions of a data flow program. Therefore, individual instructions are retrieved from the auxiliary memory as they are needed by the execution of the program. Likewise, they are returned to the auxiliary memory when the cell blocks holding them are required for more active parts of the program.

The Auxiliary Memory: The auxiliary memory is a string of memory cells such that each cell can hold one instruction of a data flow program. The unit has two input ports (command and store) and one output port (retrieve). The instructions are stored in the memory through the store port and retrieved through the retrieve port. The actual communication is carried out via instruction packets which contain the address and the contents of the cell to be stored or retrieved.

The command port is used to receive memory command packets which are used to control the flow of instructions into and out of the auxiliary memory. For example, a memory command packet \([a, \text{retr}]\) presented to the command port, requests the retrieval of an instruction packet \([a, x]\), where \(a\) is the address and \(x\) is the content of the instruction. Hence, the instruction packet is delivered at the retrieve port of the auxiliary memory.

Similarly, an instruction packet \([a, x]\) presented at the store port of the auxiliary memory requests the storage of the cell contents \(x\) at the address \(a\). However, the
storage is not effective until a memory command packet \([a, \text{store}]\) is received at the command port AND all prior retrieval requests have been honored. Of course, retrieval requests are not honored until prior storage requests for a cell have taken effect.

The need for the memory command packets stems from the envision of the designers for an auxiliary memory organization which is capable of handling multiple storage and retrieval requests concurrently. Therefore, the command packets are used to enforce the dependencies among the storage/retrieval requests pertaining to a cell, while the independent storage/retrieval operations can be carried out concurrently.

**Memory Management Operations:** In order to apply the cache principle to the memory organization, each instruction memory address (in the main memory) is divided into a major address and a minor address. The major addresses are used to identify the cell blocks, while the minor addresses are used to locate the instruction within the cell block. Therefore, all the instructions having the same major address are processed by the same cell block.

Figure 4.2 represents the structure of a cell block. The instruction cells hold the instructions whose major address is that of the cell block. Because of the 2-level memory hierarchy, there are more instructions sharing a major address than there are instruction cells in a cell.
Figure 4.2: The organization of a cell block.
block. Therefore, the cell block must return some of the inactive instructions to the auxiliary memory whenever there is a request for an instruction which is not in the cell block. The requested instruction is then brought from the auxiliary memory into the cell block.

The association table is used to keep track of the status of each instruction cell in a cell block. There is an entry for each instruction cell in the association table, consisting of a pair \([m, i]\), where \(m\) is the minor address of the cell and \(i\) is a cell status indicator with the following interpretations:

- \(i = "free"\): the cell is not assigned to any instruction.
- \(i = "engaged"\): the cell has been engaged for an instruction having minor address \(m\).
- \(i = "occupied"\): the cell is occupied by an instruction with minor address \(m\).

The meaning of the "free" and "occupied" status is clear. The "engaged" status arises when a data or control packet arrives at a cell block, but it is addressed to an instruction which is not currently in that block. The cell is marked "engaged" until the instruction is fetched from the auxiliary memory.

The stack elements of the cell blocks define an ordering for the instruction cells whose contents can be sent to the auxiliary memory to be replaced by the newly
activated instructions. Only the cells in "occupied" status are candidates for displacement.

The memory management operations are divided into two groups: one activated by the arrival of a data or control packet and one initiated by the arrival of an instruction packet from the auxiliary memory.

**Procedure 1:** Arrival of a data or control packet \([n, y]\), where \(n\) is a minor address and \(y\) is the packet contents.

**step 1:** Does the association table have an entry with minor address \(n\)? If so, let \(p\) be the cell corresponding to the entry, and go to step 5. Otherwise, continue with step 2.

**step 2:** If the association table shows that no instruction cell is free, go to step 3. Otherwise, let \(p\) be a cell with status "free". Let the association table entry for \(p\) be \([m, \text{free}]\); go to step 4.

**step 3:** Use the stack to choose a cell \((p)\) in occupied status for preemption; let the association table entry for \(p\) be \([m, \text{occupied}]\); transmit the contents \((z)\)
of cell p as an instruction packet \[m, z\] to the auxiliary memory; transmit the memory command packet \[m, store\] to the auxiliary memory.

**step 4:** Make an entry \[n, engaged\] for cell p in the association table; transmit the memory command packet \[n, retr\] to the auxiliary memory.

**step 5:** Update the operand registers of cell p having minor address n according to the content y of the data or control packet.

**step 6:** If the cell is occupied and the instruction is fired, transmit an operation or decision packet to the operation or decision units; leave cell p in "occupied" status in the association table; and change the order of cells in the stack to make cell p the last candidate for displacement. If the cell is engaged, update the operands and leave the status "engaged" until the instruction arrives from the auxiliary memory.
Procedure 2: Arrival of an instruction packet \([n, x]\) with minor address \(n\) and content \(x\).

**step 1:** Let \(p\) be the instruction cell with entry \([n, \text{engaged}]\) in the association table.

**step 2:** Update the contents of cell \(p\) according to \(x\).

**step 3:** Change the association table entry for cell \(p\) from \([n, \text{engaged}]\) to \([n, \text{occupied}]\).

**step 4:** If the instruction in \(p\) is fired, transmit an operation or decision packet to the operation or decision units; leave cell \(p\) in occupied status; and change the order of cells in the stack to make cell \(p\) the last candidate for displacement.

Example 2: The Data Flow Multiprocessor

The organization of the data flow multiprocessor proposed by Rumbaugh [119] was represented in Figures 2.7(a)
and 2.7(b). The program memory, the SWAP memory, and the SWAP network are used to manage the memory organization by swapping active/inactive procedures between the processors and the memory. The scheduler handles procedure calls and intercommunication among the procedures.

The activity count unit of the processors keeps track of the procedure call and returns. The count is decremented on procedure calls and incremented on returns. Therefore, if the count is non-zero AND there are no active functional units, then the procedure must be waiting for the return results from a called procedure. In that case, it can be transmitted to the SWAP memory, remaining dormant until the return value is arrived. At that point, the procedure is assigned to a free processor and its execution continues.

The scheduler is responsible for providing communication among a program's procedures. It maintains three tables:

1. **The Call Queue:** It is a list of calls waiting to be processed by the scheduler. Each entry in the queue has the following format:

   (Caller's processor ID, Called procedure name, Input arguments, Point of call)

2. **The Procedure Table:** This table is used to identify procedures which are currently in process. The format of the entries is:

   (Procedure name, Processor ID to which it is assigned)
3. **The Activation Table:** It identifies the caller and the point of call for each called procedure. The entries have the format below:

(Procedure name, Caller's processor ID, Point of call)

The flow of the scheduler's operations is defined in the following paragraphs. The operations are described for both the procedure calls and returns:

(a) **Procedure calls:**

i) A call request is converted to an entry in the call queue list.

ii) When a processor becomes free, an entry from the call queue is retrieved and assigned to that processor.

iii) The procedure name and the processor ID are entered in the processor table.

iv) The procedure name, caller's processor ID, and the point of call are stored in the activation table.

(b) **Procedure returns:**

i) When a procedure terminates, it signals the scheduler.

ii) The caller's processor ID and the point of return are retrieved from the activation table.

iii) The processor table is searched to see if the calling procedure is in a processor. If so, the results are returned. Otherwise, the calling procedure has to be reinstalled in a processor and then the results are returned.
2. MEMORY MANAGEMENT OPERATIONS IN WDDM

As addressed in Chapter III, WDDM has a 3-level memory hierarchy organization consisting of the processing modules, the cache memory, and the main memory. The host module is responsible for the management of the memory hierarchy, allocation of the enabled blocks to the processing modules, and communication between blocks of a program. In addition, it must detect the firing condition for a block and keep track of the free modules. In our terminology all of these operations are referred to as "memory management".

The host module is composed of a main memory unit and a host computer, as depicted in Figure 4.3. The main memory holds the program blocks as they are generated by the compiler. The host computer consists of a cache memory and a number of tables (TABLE\textsubscript{i} in Figure 4.3) which are used to carry out memory management tasks.

In this section, we will first discuss the memory management policy in WDDM. Second, the compiler role in the memory management operations is introduced. Third, communication among the blocks (i.e. procedure call/return) will be covered. Fourth, the organization of the host
Figure 4.3: The overall organization of the host module.
computer is studied, including the structure and the functions of different tables. Fifth, some performance improvement measures are proposed to increase the performance of the host computer. And finally, the flow of memory management operations in the host module is presented.

**Memory Management Policy in WDDM:**

In the 3-level memory hierarchy of WDDM, the compiler first stores the generated blocks in the main memory. The enabled blocks are then assigned to the processing modules, where their execution begins. Finally, the cache memory contains those blocks which have already begun their execution, but are temporarily inactive due to calling another block and waiting for its result(s).

In WDDM, a **fixed block-size** memory management technique is utilized to implement the memory hierarchy organization. In this technique, each program block size does not exceed a fixed, predetermined size. In the case of WDDM, the fixed block size is equivalent to the number of the E-units in a processing module. The reasons for selecting such a policy are as follows:

1. The underlying machine architecture imposes utilization of a fixed block size. This is due to the fact that the number of E-units in a processing module is fixed.
Therefore, a processing module can only handle blocks of the size smaller or equivalent to its capacity. Furthermore, the architecture of WDDM is designed based on the data flow principle that the degree of intra-block communication is much more than inter-block communication. Thus, if a dynamic block-size organization is utilized, a large program block must be assigned to many processing modules at once, requiring a high degree of communication among the processing modules.

(ii) The fixed block-size organization simplifies the memory management tasks which in turn reduces the overhead of the host computer. This reduction in overhead is critical in improving the system performance due to the fact that the host module is a "bridge" providing communication among the processing modules. Therefore, it can well become the bottleneck of the system if it suffers from large overhead operations. For example, in case of a dynamic block-size system, the main and the cache memories will require a segmentation memory management system with an overhead to overcome the external fragmentation and the garbage collection problems.

Nevertheless, we suggest the further evaluation of the merits and demerits of a dynamic block-size organization in the future projects.
The Compiler Role in the Memory Management Operations:

We will consider the compiler role in the memory management operations in WDDM with regard to four areas: block decomposition, block labelling management, block "return" instruction, and miscellaneous details. The interested readers may refer to [4, 24, 84] for more information about this subject.

(a) Block Decomposition: It is our assumption that the data flow language implemented on WDDM is highly structured or block oriented (e.g. VAL [95]). Therefore, the program blocks which are marked by constructs such as "begin-end", can be easily detected by the compiler. If a program block size is less than or equal to the predetermined fixed block-size (a processing module's capacity), then it can simply be compiled and the object module will be stored in the main memory. Otherwise, the compiler must decompose the block into a number of blocks each of the size smaller or equivalent to the fixed block-size. For example, let the capacity of a processing module be (k) bytes. If a program block is k bytes or less, it simply occupies a fixed block of k bytes in the main or in the cache memory. On the other hand, if the block size is (nk) bytes, the compiler will try to decompose the block into m (m > n) blocks, each as close as possible to size k. In that case, these blocks must be processed in sequence
such that block (i) will pass its arguments to block (i+1), for $1 \leq i < m - 1$. The arguments can be a list of all the variables used in the block (a multi-pass compiler can detect and mark the arguments and the variables of a block during its first pass). Hence, during the execution, each block segment uses a part of the variables in the argument list and updates those variables which are to be used by subsequent block segments.

(b) Block Labelling Management: A data flow environment allows concurrent execution of activations of the same block (as discussed in Chapters II and III). Thus, a dynamic block labelling scheme is used to facilitate implementation of multiple activations of a block. Each block has a label consisting of two parts ($\alpha . B$), where $\alpha$ is the static label and $B$ is the dynamic label. A static label, assigned to a block at the compile time, represents a block's identification (or name). A dynamic label is assigned to a block at the execution time. They are used to distinguish different activations of a block. For example, if different iterations of a loop are independent, they can all be processed simultaneously. In that case, for each activation of the loop, the $\alpha$ part of the label identifies the loop and the $B$ part represents the iteration.

It is the responsibility of the compiler to associate each block name (static labels) to its corresponding
location in the main memory. Therefore, the compiler must generate a table similar to the one depicted in Figure 4.4. The block label column contains static block labels and the address column represents the \textit{beginning} address of the block in the main memory. Notice that it is not necessary to keep track of the block size since it is fixed. In the remaining of this chapter we use a block's "static label" and its "beginning address" in the main memory as interchangeable terms.

\textbf{(c) Block "Return" Instruction:} Each program block ends with a "RETURN" instruction which has two functions. It is used as a signal to indicate the completion of the execution of a block. In addition, it holds the return address(es) for the block; i.e. the address(es) of the instruction(s) within the blocks to which the output of this block is passed. The format of the return instruction is as follows:

\begin{verbatim}
(opcode, return block label(s), return instruction address(es), list of arguments)
\end{verbatim}

The opcode indicates that the instruction is "RETURN". The return block label(s) consist of both the static and the dynamic labels of the block(s) to which the result(s) must be returned to. The return instruction address(es) are the address(es) of the E-unit(s) to which the output of the block is returned. These two components constitute the return address(es) for a block. The list of arguments
Block label table

<table>
<thead>
<tr>
<th>Label</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.4: The organization of the block label table.
indicates the output of the block.

In order to simplify the procedure call/return during the execution, the compiler performs two functions with regard to the return instruction. First, it always assigns the return statement to the last position in a block (i.e. last E-unit in a processing module). During the execution, when a block calls another block, the point of return must be assigned to the return instruction of the called block. By fixing the position of the return statement, this instruction can be easily accessed during the execution.

The second function of the compiler is to assign a default dynamic label (e.g. all 1's) to the return block label segment of the return instructions. This default dynamic label is used during the execution to differentiate between the blocks which are called by another block during the execution and those which are simply fired by their predecessors and will return their output to their successors. Therefore, if a block's successor is known during the compilation, the compiler will insert the successor's static label and the default dynamic label in the return instruction. Otherwise, the return block label segment of the return instruction is left blank.

(d) Miscellaneous Details: In a data flow environment, two operation nodes communicate to each other via "tokens". In WDDM, a token is either a single value or a variable-name identifying a data structure (a pointer to a
data structure). Each token is thus augmented with a tag bit indicating if it is a single value or a data structure name. The tag bit is used during the execution to manipulate data structure operations. The compiler is responsible for setting this tag bit.

In addition to the above functions, the compiler is also responsible for the generation of a symbol table to keep track of the addresses of the variables. Therefore, during the execution, the address(es) of the instruction(s) which use the input argument(s) are known by referring to the name of the arguments. This information is used to assign the input arguments to a block in their perspective addresses during the execution.

**Communication among the blocks**

In WDDM, the communication among the blocks is divided into dynamic and static communications. A dynamic communication occurs when a block calls another block during the execution time and receives the result(s) of the called block. The static communication happens when a block receives its input argument(s) from its predecessor block(s) and passes its output to the successor block(s), as detected during the compilation.

The double star interconnection network of Figure 3.1 reveals that all the communications (static or dynamic)
among the blocks are handled through the host module. The details of these operations are discussed in the flow of memory management operations section. However, generally speaking, during a dynamic communication, a block calls another block by sending a "CALL" token to the host computer. The host assigns the point of return to the return instruction of the called block and fires it for execution. When the called block completes its execution, a "RETURN" token is sent to the host, where the output of the called block is assigned to the point of return in the calling block. The static communication is handled in exactly the same manner except that a block is fired by receiving its input argument(s) from its predecessor(s) instead of being called. A CALL token has the following format:

[called block label, arguments, point of return].

The point of return is composed of: i) the calling block's label (static and dynamic); and, ii) the addresses of the instructions within the calling block to which the result(s) of the call are to be returned. Naturally, the instruction addresses consist of an E-unit's ID and the operand number within the instruction. The point of return is stored in the return instruction of the called block. When the return instruction of the called block is executed, a RETURN token is sent to the host computer with the following format:

[result(s), point of return],
where the point of return has the same format explained for a CALL token. The host uses the point of return address to assign the result(s) to its destination within the calling block.

In the case of static communication, the RETURN token is recognized by the host since the dynamic label of the calling block label contains the default value. Therefore, the result(s) are simply assigned to the entry point of the successor block whose name appears in the static block label segment.

The Host Computer Organization

In order to handle the memory management operations, the host computer consists of a number of components whose organizations and functions are discussed in the following paragraphs.

(1) The Cache Memory: The cache memory is the intermediate level in the memory hierarchy organization used in WDDM. The main memory contains "passive" blocks which are not yet enabled for execution. The processing modules hold "active" or enabled blocks whose execution have begun. Finally, the cache memory is a temporary storage for "inactive" blocks.

An active (executing) block becomes inactive when it has not yet been completed, but it is unable to continue its
execution. As discussed in Chapter III, the AID unit of a processing module is responsible for the detection of an inactive block. The inactivity condition arises when a block calls another block (or itself) and must wait for the return value from the called block. Naturally, after a block is called, the execution of the instructions in the calling block will continue until the execution of the remaining instructions is pending upon the return value from the called block. At that point, the block becomes inactive and must be saved in the cache memory so that the processing module can be assigned to some other enabled block.

(2) The Cache Table: The cache table is used to keep track of the location of inactive blocks in the cache memory. Figure 4.5 depicts the organization of the cache table. The "block label" column contains the run-time block names; i.e. static and dynamic labels. The "address" column holds the beginning address of the block in the cache. Since the size of the blocks is fixed, it is not necessary to keep track of the block size.

When a block becomes inactive, the AID unit of the processing module sends a signal to the host computer. The host then transfers the block to the cache and adds an entry pertaining to this block to the cache table. When the called block is executed, it sends a RETURN token to the host which contains the point of return and the value to be returned. At this point, the host searches the cache table
Cache table

<table>
<thead>
<tr>
<th>Block label</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

Figure 4.5: Organization of the cache table.
for the return block label and uses its beginning address to assign the result to its destination. In addition, the block's label is added to the (enabled blocks list) to continue its execution as soon as a processing module is available.

In order to speed up the operations of the cache table, an associative memory with read, write, and search capability is used to implement this table. We have discussed the details of a VLSI implementable associative memory with such capabilities in [74,75,78,122].

(3) **The Block Assignment Table:** The function of this table is to determine the status of a block by indicating whether or not it is active. Figure 4.6 presents the organization of the block assignment table. The block label consists of both the static and the dynamic labels. When an enabled block is assigned to a free processing module, the block name and the corresponding processing module number are added to the block assignment table. Upon the completion of a block or as soon as it becomes inactive, the entry for that block is removed from this table. Thus, the presence of a block's label in this list is an indication of the "activeness" of the block and vice-versa. When a called block returns its result, the calling block may be in a processing module if it is continuing its execution, or it may be in the cache memory if it becomes inactive. A search of the block assignment table can easily indicate the
Block assignment table

<table>
<thead>
<tr>
<th>Block label</th>
<th>Processing module number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.6: The organization of the block assignment table.
location of the calling block. If there is an entry for the block in the table, then it must be active and resident in a processing module whose tag number appears in the entry. Otherwise, the block is inactive and can be found by a search of the cache table.

The operations performed on the block assignment table include: writing/removing an entry into/from the table, reading the information related to an entry, and searching for a particular entry. The nature of these operations implies the utilization of associative memories for achieving higher performance. Thus, an associative memory organization such as the one proposed in [74,75,78] can be used to implement the block assignment table.

It should be noted that one can combine the cache and the block assignment tables by adding a tag field which distinguishes the status of a block (active/inactive). This scheme slightly reduces the memory requirements for these two tables, but increases the search delay since the size of the combined table will be larger. Therefore, we will proceed with the former two-table scheme.

(4) The Free Module List: This unit is simply a list of processing modules each with an associated tag. The 1-bit tag is used to indicate whether or not the processing module is free. Figure 4.7 depicts this organization. In order to assign the enabled blocks to the processing modules, the host first searches the tag field for a free
### Free module table

<table>
<thead>
<tr>
<th>Processing module #</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>▪</td>
<td>▪</td>
</tr>
<tr>
<td>▪</td>
<td>▪</td>
</tr>
<tr>
<td>▪</td>
<td>▪</td>
</tr>
<tr>
<td>▪</td>
<td>▪</td>
</tr>
</tbody>
</table>

Figure 4.7: Organization of the free module table.
module (tag = 0). If one is found, then the block is assigned to that module and its tag is marked occupied (tag = 1). If no free modules are found, then the request is queued in the enabled blocks table. A processing module becomes free either when it is inactive or when the block occupying the module completes its execution. The inactiveness condition was defined in the discussion of the cache memory organization. The block completion condition occurs when a RETURN token is received by the host computer. As previously mentioned, the execution of a "return" instruction indicates the completion of the block.

The operations on the free module list are searching for a free tag, setting the tag to free/occupied, and reading the module number of the free modules. Once again, an associative memory similar to the one discussed in [74, 75, 78] can be utilized to carry out these operations efficiently.

(5) The Firing Condition Table: The functions of this table are to keep track of the input arguments to a block and to detect the firing condition for a block. The organization of the firing condition table is shown in Figure 4.8. The link field for each block is a pointer to a linked list in which the input arguments are kept. Each argument field in the linked list consists of the value of the input argument and its relative position in the input argument list for that block. As an argument is made
Figure 4.8: The organization of the firing conditions table.
available to a block, it is added to the corresponding linked list and the number of unavailable input arguments is decremented. When this number reaches zero, the block is enabled for execution. In that case, an entry for the block is added to the enabled blocks queue.

A block is enabled either statically (following the program structure) or dynamically (being called by another block during the execution). The tag field is used to distinguish these two cases. The information in the tag field is later used to assign an enabled block to a processing module.

The point of return field is left blank for the blocks which are statically enabled (since their point of return is known during the compilation). However, when a block is called, the dynamic label of the caller is added to the called block's label and the point of return is stored in the point of return field.

(6) The Enabled Blocks Queue: The main function of this table is to queue the enabled blocks so that they can be assigned to a processing module, when one becomes available. The enabled blocks queue receives entries either from the firing condition table or the cache table. Therefore, each entry is augmented with a tag field indicating from where the entry has been received. In addition, the tag distinguishes the statically and dynamically enabled blocks (from the firing condition
An entry from the cache table contains the label of an inactive block which has received its called value and now can become active. When a return token is received which enables an inactive block, the cache table is searched for the beginning address of the block. The return value is assigned to its destination(s) in the cache and a corresponding entry is added to the enabled blocks queue.

When a free processing module is available and the entry at the front of the queue has been received from the cache table, the block is simply assigned to the processing module to continue its execution. Otherwise, the following sequence of operations are carried out: i) the enabled block is retrieved from the main memory; ii) the input arguments from the firing condition table are added to the block; iii) if block is dynamically called, the point of return address is stored in the return instruction of the block; iv) if the block is statically enabled, a dynamic label is added to the block's label; and v) the block is assigned to the processing module to commence its execution. It should be noted that these different cases are detected by checking the tag field associated to each entry in the enabled blocks queue.

(7) **The Label Generator Counter:** The function of this unit is to generate unique dynamic labels for each block as they are needed during the execution. Figure 4.9 presents a
<table>
<thead>
<tr>
<th>Block label (static)</th>
<th>Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>•</td>
<td>•</td>
</tr>
</tbody>
</table>

Figure 4.9: Conceptual view of the label generator counter unit.
conceptual view of such a unit. Associated to each static block label there is a counter which generates a unique number (dynamic label) each time that one is needed. One way to implement this unit is to use a simple accumulator (buffer) as a counter. Each time a dynamic label is needed, the host searches the table for the block label and increments the corresponding accumulator. This method is simple and inexpensive but provides an overhead for the host due to the search and incrementing operations. An alternative is to use an associative processor with a static label field and an accumulator field. The associative processor can search for a static label and return the corresponding accumulator field as a dynamic label. It can then increment the accumulator field to generate the next dynamic label. Therefore, the operations of this unit can be overlapped with the host's operations.

(8) The Command Queue: The host computer concurrently receives different commands from the modules in an asynchronous fashion. These commands consist of procedure calls, return commands, and inactivity signals. In order to process these commands in an orderly fashion, they are first queued in the command queue and then processed sequentially.

Performance Improvement Considerations

It is obvious that the host computer can become the
bottleneck of the system since all the communication paths among the blocks must pass through this unit. However, several measures may be taken to improve the performance of the host computer and to remove the possibility of creating a bottleneck. For example, to speed up the transfer of the blocks and communication between the host and the processing modules, a buffering (spooling) technique can be used. This technique can be similar to the ones used in conventional systems for handling multiple I/O devices by one CPU [58, 110]. In an alternative scheme, an interleaved memory organization can be used for the main memory. Therefore, different blocks are assigned to different memory banks and can be transmitted to the processing modules simultaneously.

Further performance improvements are possible by the utilization of separate associative memories for the cache table, the block assignment table, the free module list, and the label generator counter. In this case, not only search operations are carried out more efficiently by hardware, but also the different host operations can be overlapped.

A third possibility for improving the performance may arise in the execution of recursive procedures. One should take advantage of the fact that a copy of the block is already in a processing module and there is no need to transfer the block to the modules for future iterations. This consideration will result in a decrease in the amount and time of transferring blocks between the host and the
processing modules. However, at the same time, the overhead of the host operations may be increased due to the complexities involved in the implementation of such an approach. For example, a large block is decomposed into several segments and only one segment will already be in a processing module. Furthermore, there is the problem of reinitializing the subsequent iterations to the state in which their execution may begin. Nevertheless, it is our intention to investigate the advantages and disadvantages of such a possibility in the future projects.

**The Flow of Memory management Operations**

So far, the memory management operations have been sporadically introduced during the discussion of different elements of the host computer. We will now merge these operations by proposing an algorithm which is presented in the form of a flowchart in Figure 4.10. Different segments of the flowchart are marked by an identifying number. The function of each segment is as follows:

1) This segment attempts to assign an enabled block to a free processing module.

2) When a processing module is available, the block at the front of the enabled blocks queue will be assigned to it. If the enabled block is a passive block in the main memory, the input argument is assigned to its destination
Figure 4.10(a): The flow of operations in the host module.
Figure 4.10(b): The flow of operations in the host module.
within the block and the return instruction is updated. If the enabled block is an inactive block in the cache memory, it is simply assigned to the processing module since it has already been updated in segment 5. In either case, the free module table and the block assignment table are updated.

3) The command queue is checked for the presence of a command. If the command is a CALL token, then a dynamic label is generated and an entry for the called block is added to the enabled blocks queue. In case of an inactive signal (left portion of the third segment), the inactive block is transferred to the cache memory and the cache table, the free module table, and the block assignment table are adjusted accordingly.

4) This segment deals with the RETURN tokens in which the dynamic label of the return address is the default value. This case arises when a block simply passes its output to its successor blocks. Therefore, the entry for the completed block is removed from the block assignment table and a new entry for the successor block is added to the enabled blocks queue.

5) Segment 5 is executed when the result of a called block is returned to the host to be assigned to its calling block. If the calling block is still active in a processing module, then the result is simply assigned to it. Otherwise, the block must be inactive in the cache memory. In that case, the return value is assigned to its
destination within the calling block and an entry for the block is added to the enabled blocks queue.
3. SUMMARY

In this chapter the memory management operations in some of the existing data flow machines and the proposed architecture were discussed. These operations include the management of the memory hierarchy, procedure (block) communication mechanism, handling of the concurrent activations of the same block, resource management, etc.

In WDDM, a fixed block-size memory management policy is employed due to the underlying machine architecture and in an attempt to reduce the host computer's overhead. Moreover, the program blocks communicate through a procedure call/return mechanism which must pass through the host computer.

The host computer uses a number of elements to carry out the memory management tasks. These elements include: the cache memory, the cache table, the block assignment table, the free module list, the firing condition table, the enabled blocks queue, the dynamic label generator counter, and the command queue. In addition, this chapter contains a discussion of the compiler role in the memory management operations, measurements for improving the performance of
the host computer, and details of the memory management operations.
In this chapter we will discuss the representation and manipulation of data structures in a data flow environment in general and in WDDM in particular. Data flow computations conceptually prohibit the use of shared or global memory. Therefore, in such an environment, neither single data items nor data structures can be shared among a program's instructions. This implies that the scope of any data item is limited to the instruction that "consumes" it. Thus, when the control is passed from one instruction to another, new copies of the data item must be generated rather than updating of the same memory area. Such a scheme eliminates the side-effects resulting from global memory access.

However, during the implementation, in order to improve the performance and make effective use of the resources, other factors such as cost (space), performance, and practicality must also be considered. Basically, two methods for implementation of data structures in a data flow
system have been proposed in the literature [121]:

(1) The entire data structure is copied from one node to the next. This approach maximizes the parallelism by providing multiple copies of a data structure and eliminating the side-effects. However, it is not cost effective since it reduces the memory utilization and generally degrades the performance due to transmission of data structures within the system. For example, if a node modifies a single element of an array, then a copy of the entire array must be passed to the successor node, with the particular element updated.

(2) The data structures are stored in the memory and only a pointer to them is passed from one node to the next. Naturally, this method increases the memory efficiency by allowing the substructures to be shared. However, the side-effects due to the shared memory and the overhead of the data structure operations (keeping track of the pointers and garbage collection) reduces the system performance.

It is obvious that both methods suffer from some disadvantages and problems. As a result, there is an ongoing research effort in this area in order to find efficient methods for handling the data structures in a data flow environment [2,5,26,103]. In WDDM, an attempt is made to take advantage of the merits of both approaches. In other words, data structures are copied to increase parallelism and speed, but whenever possible, they are shared to increase the efficiency of the memory.
1. THE PREVIOUS WORK

In this section, the data structure manipulation in the MIT [2,43,103], the Manchester [133], and the Utah [35] machines are discussed. This study reveals different approaches for implementation of data structures in a data driven architecture. The advantages and the disadvantages of each approach is also discussed.

The MIT Machine

The data structures in the MIT machine [43,121] are binary trees whose nodes are either primitive or pointers to other nodes. In other words, a data structure is a set of nodes (<selector, value> pairs), where selector is an integer or a string indicating the node's position relative to the root of the tree and value is a data flow value (primitive or structured). The selector can be an integer indicating the index of the node in an array or a bit string defining the position of the node in the tree. A structured value is represented by a token containing a unique pointer to the root of the structure. Figure 5.1 depicts an example
Figure 5.1: An example of a data structure in the MIT machine.
of a data structure:

\[ a = \langle S_1, V_1 \rangle, \langle S_2, (\langle S_3, V_3 \rangle, \langle S_4, V_4 \rangle) \rangle, \ldots, \langle S_n, V_n \rangle \],

where, "a" is the root and \( S_i \) denotes the selector for value \( V_i \) \((1 \leq i \leq n)\). Two of the major data structure operations are SELECT and APPEND [2]:

**SELECT:** The select operation takes a data structure and a selector and returns the value at the specified selector. As an example, select \( (a, S_3) \) results in retrieval of the value \( V_3 \) from the data structure of Figure 5.1. If the data structure does not contain the selector component in the select operator, the result is the value **undefined**.

**APPEND:** This operator takes a structure, selector, and a value and returns a structure identical to the original structure, except that it contains the given value at the position indicated by the selector. An example of the append operation is represented in Figure 5.2. After the execution of this operation, whatever value which was previously at the selector position will be absent in the resulting structure. Thus, to delete part of a structure, one can append \((\text{nil})\) in its place. Similarly, to create a structure, simply use \((\text{nil})\) as the initial structure and append values to it.

Clearly, the MIT machine allows substructures to be shared between structures without violating data flow semantics. As a result of sharing substructures, a node of a structure may be referenced many times. This makes the
Figure 5.2: An example of Append operation in the MIT machine.

\[
\text{Append } (a, s, y) \rightarrow \beta
\]
memory management tasks rather complex. For example, the data structure controller of the system must keep a reference count for each node of a structure for garbage collection purposes. This technique is similar to the mechanisms developed in the conventional systems to remove dangling references [112]. The reference count indicates the number of pointers that refer to a node. When a node's reference count becomes zero, two actions take place. First, the node is returned to free storage since it is no longer being accessed. Second, all the successors of the freed node must have their reference count decremented by one.

A special case arises when a node's reference count is equal to one; i.e. only one pointer to the node exists. In this case, it is possible to update the node in place (no copying) since the node is not shared with other structures.

It is crucial that the data structure controller ensures that no cycles are created as the result of the structure operations. Such a situation is depicted in Figure 5.3(a). The root of this structure has its reference count equal to two. When the structure is no longer needed, the pointer to the root is deleted and its reference count is decremented by one, as in Figure 5.3(b). However, the structure is never returned to the free storage since all of its nodes have a reference count greater than zero.

The manipulation of data structures in the MIT machine
Figure 5.3(a): The structure is being accessed.

Figure 5.3(b): The structure is not accessible.
suffers from several drawbacks:

i) The overhead resulting from management of the reference counts, garbage collection, and ensuring acyclic trees degrades the overall performance of the system.

ii) In spite of sharing the substructures, the amount of structure duplication can be substantial if small substructures are shared among large structures.

iii) Since the structure memory consists of a single unit, the shared substructures must be accessed in sequence, reducing parallelism in data structure operations.

As an extension to the above organization, Rumbaugh has proposed utilization of separate structure memory and controller for each processor [120]. As a result, conflicts in requests for accesses to a structure are avoided and structure operations are overlapped. The clear drawback of such a system is that it becomes prohibitively expensive.

**The Utah Machine**

In the Utah machine, data structures are represented through the list structures [35,121]. A list is a recursive structure consisting of a field which by itself may be composed of other well-nested fields. Each field is delimited by a set of left and right parentheses. The first subfield within a field is the descriptor for the rest of the subfields. Figures 5.4(a) and (b) depict list structure
Figure 5.4(a): Representation of a vector in DDM1.

```
((ordered vector) (V_1)(V_2) \ldots (V_n))
```

Figure 5.4(b): Representation of a matrix in DDM1.

```
((ordered matrix)
  ((ordered row 1) (V_1)(V_2) \ldots (V_n))
  ((ordered row 2) (V_1)(V_2) \ldots (V_n))
    \ldots
    \ldots
  ((ordered row m) (V_1)(V_2) \ldots (V_n))
)
```
representation of a vector and an mXn matrix, respectively.

The fields may be ordered or unordered. An unordered field is accessed by a unique name associated to that field, while ordered fields are accessed through their indices. Furthermore, a field may be simple or complex. In the case of the complex fields, the field itself is indexed by an access vector whose elements denote the subfields. For example, in Figure 5.4(b), the access vector is the set of row descriptors identifying different subfields (rows).

The system utilizes variable length fields. Therefore, a dynamic memory management is required to allow a location-independent scheme for dealing with variable length fields. Even though such an organization offers greater flexibility in comparison to fixed length environments, the overhead resulting from dynamic memory management is its drawback.

The Manchester Machine

The Manchester machine utilizes several data structures including arrays, records, and streams [121,133]. Such a variety of structures have been motivated by the viewpoint that Pascal-like data structures have been successful in representation of real-world applications.

i) Arrays: The token labelling scheme utilized by the Manchester machine [68] allows the arrays to be represented
as a fixed number of tokens travelling on an arc, with index fields specifying a token's position in the array. The array operations are performed through such constructs as "for...use..." and "as...with...". The "for...use..." construct is used to define an entire array. For example, in \( A := \text{for I use } e(I) \), 'e' is an expression whose value depends on I, and it defines the value of the \( I^{\text{th}} \) element of array A. The range of I depends on the number of elements in A. The expression 'e' is evaluated for each index (I-value) in parallel. Thus, this construct is used for defining all array elements at the same time. The "as...with..." construct, however, allows updating of one or several elements of an array. For example, in \( A := \text{as B with I := } e \), arrays A and B are the same except that the \( I^{\text{th}} \) element of A has the value of 'e'.

\( \text{ii) Records: A record consists of a number of fields, where each field is simply a named value. Records are implemented in a manner similar to the arrays. In other words, one arc of the directed graph is designated for each record field to allow transmission of records between instructions. Variant records (records with variable number of fields) can also be represented in the same way. However, it must be ensured that the number of arcs carrying a variant record is greater than or equal to the maximum number of record components.} \)
iii) Streams: A stream is a structure whose elements have a total linear ordering but may not all coexist since the stream elements are generated in a pipelined fashion. Concurrency can be exploited because the process of producing and consuming streams can be overlapped in a parallel, pipelined manner. Since the Manchester machine is a backend pipelined data flow computer [133], the utilization of streams in this architecture is a natural choice.

The organization of a stream is similar to a vector in the sense that a stream is implemented as a set of tokens with a unique index representing each stream element. The stream tokens travel on a common arc in sequence. The size of a stream is variable, requiring a special end-of-stream token to signal the end of a stream.

The stream operations are similar to operations performed on lists. For example, FIRST(S) returns the first element of the stream S. While, REST(S) returns a stream composed of all the elements of S, but the first one.

The major problem associated with the manipulation of data structures in the Manchester machine is that the structure elements are processed sequentially as streams of tokens travelling on an arc. Such an approach is not universal, but only suitable for the Manchester machine since it has a pipelined architecture. Therefore, the same method implemented on other data flow machines may pose a serious performance degradation.
2. DATA STRUCTURE MANIPULATION IN WDDM

The data structure module is used to facilitate the manipulation of data structures. As depicted in Figure 3.1, this unit is shared among the processing modules. The data structure module is composed of a Data Structure Processor (DSP) and a Data Structure Memory (DSM) unit. The DSP is a dedicated processor designed for the manipulation of data structures and control of the data structure memory. The DSM has an interleaved organization and holds the data structures. By freeing the host module from handling the data structures and providing overlapped operations in the host and the data structure modules, the overall system performance is improved.

Data Structure Processor (DSP)

The DSP is used as an interface between the instructions of a program and the data structure memory. It provides smooth access to the data structures, manages the data structure memory, and initiates constructs such as FORALL [95]. Specifically, the DSP functions are composed
of the following:

i) Queuing the input/output tokens: DSP receives data structure tokens (refer to Chapter III) simultaneously from several processing modules. These tokens must first be queued at DSP and then processed by this unit. Similarly, the results of the data structure operations are queued as they become available and then routed to their destinations.

ii) Initiating data structure operations: DSP initiates the execution of constructs such as FORALL and OVAL (which will be discussed shortly). In addition, DSP can fetch or store individual data structure elements.

iii) Maintaining and updating data structure descriptors: In order to facilitate the manipulation of data structures, DSP maintains and updates a table of descriptors for the data structures. The specific structure of this table is to be decided during the design of the operating system for WDDM. However, some of the major components of the table include: The type of the data structure, the location in the memory (e.g. the beginning address), the data structure size, a reference count, the type of elements, etc. The information in the table of descriptors is used for accessing the data structures and managing the data structure memory.

iv) Managing the data structure memory: DSP is responsible to allocate space for the data structures, duplicate the data structures, and delete the data
structures. Each of these operations involves the updating of the table of descriptors. When a data structure is allocated or duplicated, an entry for that structure is added to the table of descriptors. Whenever a data structure is deleted, its reference count is decremented.

**Data Structure Memory (DSM)**

The DSM holds the data structures used in the data flow programs. As a major component of the data structure module, it is shared among the system's processing modules. Each DSM word is augmented with a 2-bit tag field. The first tag bit is used to define the type of the DSM word; i.e. either a data value or a pointer/link value. The function of the second tag bit will be discussed in the next section.

The advantages of a tagged architecture are conclusively discussed in [55,107,108]. One of the advantages with regard to our architecture is that such a tagging scheme provides a high degree of flexibility in the representation of the data structures. For example, Figure 5.5 depicts the internal representation of a vector and a two-dimensional matrix.
Figure 5.5: Examples of data structure representation in WDDM.

(a) Vector representation

\[ V_1, V_2, \ldots, V_n \]

(b) Matrix representation

\[ V_{11}, V_{12}, \ldots, V_{1n} \]

\[ V_{21}, V_{22}, \ldots, V_{2n} \]

\[ \vdots \]

\[ V_{m1}, V_{m2}, \ldots, V_{mn} \]

\[ V_i = \text{value}_i \]

\[ P_j = \text{pointer}_j \]
The Memory Organization:

The processing capability of WDDM is distributed among the processing modules. This processing power can be used either to process a data structure in a SIMD parallel fashion or to manipulate individual data structure elements. Therefore, there will be many occasions in which several processing modules require access to the data structure memory simultaneously. Furthermore, for parallel SIMD operations, a data structure may be distributed among several processing modules.

In order to improve the performance of DSM, an interleaved memory [14] organization can be used for the implementation of this unit. An n-way interleaved memory is composed of n memory modules numbered 0, 1, ..., n-1, where words at address (i) are in module number (i mod n). In general, if the n memory modules can be operated independently, the data transfer rate (bandwidth) between the memory unit and the processing unit can be up to n times faster than a single component memory system. Thus, by sequentially storing the elements of a data structure in separate modules, it is possible to have access to many of its elements concurrently.

In terms of parallel data structure operations, it now becomes possible to simultaneously load different segments of a data structure from different memory modules into
different processing modules.

In terms of accessing the elements of the data structures, any two access requests can be carried out in parallel provided that the requested elements are in different memory modules. Therefore, it is possible for several blocks to simultaneously access different elements of the same or different data structures.

Data Structure Manipulation Policy in WDDM

The general policy for the manipulation of the data structures is essentially based on the duplication of data structures through the application of the single assignment philosophy. Of course, the cost of the memory overhead due to the duplication of data structures is justifiable because of the advances in technology [18] and the performance improvements gained by parallel operations on data structures. Nevertheless, in the interest of practicality and cost-effectiveness, unnecessary duplications of the data structures are avoided. Furthermore, to reduce the communication overhead, only the data structure names (pointers to structures) are passed from one instruction to another. The actual data structures are kept in the data structure memory. The manipulation of data structures in WDDM can be discussed in two different contexts as follows:
1. Inter-block data structure operations:

In a block-oriented, data driven organization, the independent program blocks are executed in parallel. Such an environment imposes the duplication of data structures among the independent blocks in order to remove the possibility of side-effects among the blocks. Therefore, a data structure used as an input argument to a block is copied for each independent block. The duplicated structures are then used as the input tokens within the blocks. Such a scheme guarantees parallelism, without side-effects, among independent blocks.

Naturally, whenever possible, unnecessary duplications are avoided. If a data structure is only used as a source in several blocks, the descriptor entry for that structure can be easily marked by the compiler to prevent its unnecessary duplication. Consider a case in which a data structure (as the output of a block) is to be routed to several blocks. A multi-pass compiler can find whether or not the data structure is used as a destination within the succeeding blocks during one of its passes. It can then mark the output data structure accordingly during a subsequent pass.

Due to the functionality of a block driven program, a block "consumes" its input arguments and "produces" an output result(s). Hence, when a block completes its
execution, all the input and generated structures which are not passed to the succeeding blocks can be deleted. Therefore, each entry of a data structure in the table of descriptors contains a field which holds the label of the block in which the data structure is generated. Upon the completion of a block, a message containing the block label and the name(s) of any output data structure(s) is routed to the data structure module. This unit then deletes all the data structures related to that block except the output data structure(s).

In some cases, it may be necessary to generate a data structure once and use it as a source among several blocks or among several activations of the same block. In such an occasion, the data structure is declared to have such a property in the program. The compiler can then mark the descriptor for that data structure accordingly. Therefore, these data structures are not deleted at the completion of the execution of a block.

2. Intra-block data structure operations:

Within a program block, the Single Assignment Rule [95, 125] is used to determine when a data structure needs to be duplicated. This rule implies that an identifier, including a data structure or an element of it, may be updated only once within the scope of a block. Therefore, if two
instructions update a data structure or one of its components, the single assignment rule forces the programmer to duplicate the data structure for each instruction. In that case, the two instructions may process different copies of the data structure simultaneously, without any side-effects. However, if different elements of an array are updated only once by many instructions, it is not necessary to duplicate the array for each instruction.

The single assignment rule must be enforced both at the compilation and the execution times. During the compilation, a multi-pass compiler can keep track of the number of times a data structure name is used as the destination. Moreover, the compiler can count the number of times each data structure element is used as the destination, provided that the subscripts of the elements are known during the compilation.

However, when the subscript(s) of an element is not known during the compilation, the single assignment rule must be enforced during the execution. The second tag bit of the data structure memory words is used for this purpose. When the space for a data structure is allocated, this tag bit is set to zero for each element. As soon as a value is assigned to an element whose tag is zero, the tag is changed to one. Any attempt to update an element whose tag is one results in a violation of the single assignment rule.
Data Structure Operations

Since each data structure may be processed as one unit in parallel SIMD operations or through its individual elements, there are four possible data structure operations:

Type 1: An entire structure is addressed as a source.
Type 2: An entire structure is addressed as a destination.
Type 3: A structure element is addressed as a source.
Type 4: A structure element is addressed as a destination.

Clearly, Type 1 and Type 3 operations do not imply duplication of the data structures since they do not update the data structure memory. The Type 2 and Type 4 operations may or may not induce data structure duplication according to our discussion of the single assignment rule.

Processing Data Structure Elements:

If a data structure element is addressed as a source, a data structure token is routed to the data structure module. This token contains a fetch command, name of the data structure, subscript information, and the module number and the E-unit ID of the initiating instruction. The data structure processor uses the table of descriptors to locate the element. It will then send a result token (composed of a value and destination address) to the requesting instruction.
If a data structure element is updated by an instruction, a data structure token is routed to the data structure module containing: an opcode, a value, the structure name, and the subscript information. The data structure module checks the tag bit of the structure element. If the element has not been updated, the value is stored in the memory word and the tag bit is changed. Otherwise, an execution time error has occurred.

**Processing Entire Data Structures:**

The WDDM's architecture provides a suitable environment for the execution of the parallel SIMD type data structure operations. This is due to the fact that the E-units and the functional units distributed across the system could be utilized for processing the data structure elements in parallel. The details of these operations should be worked out during the operating system design phase. Therefore, such a discussion is out of the scope of this project. However, we will briefly discuss the general steps involved in carrying out the parallel SIMD type data structure operations in WDDM. Several examples of these operations are presented next. However, it should be noted that the following discussion is not a complete enumeration of the data structure operations in WDDM. This study can be further developed in the future to include the manipulation
of pointers, linked lists, trees, streams, etc.

**Example 1: FORALL.opcode construct:**

Suppose it is necessary to multiply every element of an array by a constant \( c \). Instead of sequentially executing the operation for each element, one can use a FORALL construct to carry out the operation in parallel. The FORALL command consists of the application of a single operation (opcode) to each element of a structure in parallel. The execution of a FORALL command consists of the following general steps:

1. Send a data structure token to the data structure module containing: the opcode, the FORALL operation to be performed, name of the data structure(s) involved, and the point of return (module number and E-unit number).

2. The data structure processor updates the table of descriptors for the structures and sends a message to the host, requesting access to a processing module. The message consists of the name of the data structures, their size, their type, and the operation involved.

3. When a free processing module is available, the host allocates that module to this operation. Each E-unit is set to the FORALL operation opcode and each destination address consists of the name of the result structure and the corresponding subscript.
(4) The involved data structure elements are assigned to the E-units and the execution is begun. If the size of the data structure exceeds the capacity of a processing module, the operation is carried out in several stages or by several processing modules.

(5) The results of the operation are routed to the data structure module and stored in the prospective elements of the resultant structure.

(6) When the operation is completed a token is sent back to the initiating instruction to acknowledge the completion of the operation.

Example 2: OVAL.opcode Construct:

An OVAL operation is the application of an operation with associativity property to the elements of a data structure in a Binary Tree fashion. Figure 5.6 depicts an example in which the elements of an array are added up. The OVAL operations consist of: AND, OR, PLUS, TIMES, MAX, and MIN. The first three steps in the execution of an OVAL command are the same as those discussed in the FORALL command. The subsequent steps are as follows:

(i) Assign pairs of the structure elements to the E-units. Thus, the operations in the lowest level of the binary tree are carried out in parallel. This will produce a temporary array containing the elements for the next level
Figure 5.6: An example of an OVAL operation.
of the tree. Repeat this procedure $\log_2 n$ times until the final result is obtained ($n$ is the size of the data structure).

(ii) The result of the operation is sent to the data structure module. It is then augmented with its destination address and routed to the initiating instruction.

**Example 3: Copy Construct:**

The copy construct "copy ... EXCEPT i", copies a data structure into another data structure except for the $i^{th}$ element. For example, $(A := \text{Copy B EXCEPT 1})$, results in copying all the elements of B into A except the first one. The resulting data structure has $(n-1)$ elements, where $n$ is the size of B. When the EXCEPT option is omitted, the entire data structure is copied. This construct is useful in explicit duplication of arrays as well as handling streams and lists. For example, LISP's CDR function [135] can be readily carried out in a manner identical to the previous example.

**Example 4: Matrix Multiplication:**

In order to demonstrate the use of the parallel SIMD type operations, the problem of matrix multiplication is discussed here. Table 5.1 shows an algorithmic approach for
Table 5.1: Matrix multiplication in WDDM.

Comment: Transpose matrix B

For i = 1 to m
    For j = 1 to p
        D(i,j) = B(j,i)
    end
end

Comment: Apply FORALL and OVAL operations

i represents a row and j a column of C

For i = 1 to n
    For j = 1 to m
        E(i) = FORALL.MULTIPLY (A(i),D(j))
        C(i,j) = OVAL.ADD(E(i))
    end
end
multiplication of two matrices (e.g. \( A_{nxp} \) and \( B_{pxm} \)). In this algorithm, matrices \( A, B, C, D, \) and \( E \) are assumed to be stored as depicted in Figure 5.5 (b). In such a storage scheme, \( A(i) \) (\( 1 \leq i \leq n \)) refers to the \( i^{th} \) row of array \( A \), while \( A(i,j) \) (\( 1 \leq i \leq n, \ 1 \leq j \leq p \)) refers to the element of \( A \) positioned at the \( i^{th} \) row and \( j^{th} \) column. The time complexity of this algorithm is \( O(n^2) \) because of the transpose operation. If we only consider the addition and multiplication operations, the time complexity will be \( O(\log_2 n) \). This is because all the multiplications are carried out in parallel in a constant time and the OVAL.ADD operation requires a \( \log_2 n \) time complexity. Therefore, the time complexity of matrix multiplication can be reduced by at least one order of magnitude in comparison to a conventional matrix multiplication (\( O(n^3) \)).
3. SUMMARY

This chapter discussed the representation and manipulation of data structures in some of the existing data driven computers as well as the WDDM. The study showed that in a data flow environment, data structures may either be duplicated from one node to the next to improve the system's performance or they may be shared to increase the memory efficiency. The former case suffers from poor utilization of memory resources, while the latter case poses some performance degradation problems.

In WDDM, the data structure module is used to facilitate the manipulation of data structures. This module consists of a data structure processor and a data structure memory. The processor holds a table of descriptors for the data structures and performs memory management operations such as allocation of data structures, deletion of the data structures, and garbage collection. The data structure memory has an interleaved organization, allowing simultaneous accesses to several elements.

The data structure manipulation policy is based on the single assignment rule. This rule enforces the duplication
of the data structures only in the interest of improving the performance. However, whenever possible, unnecessary duplications are avoided.
In contrast to some of the proposed data driven machines, WDDM is designed according to the constraints imposed by the VLSI technology. This means that WDDM consists of only a few building block elements which are duplicated many times across the system [52, 54, 56, 99, 127]. In addition, in order to solve the I/O pin limitation problem [21, 57], we have utilized one of the most recent developments in the VLSI technology, namely wafer-scale integration [25, 30, 48, 59, 70, 85, 94]. This technique uses the entire wafer, instead of dicing, to condense more functionality into a single device. The cable connector delays of multi-chip systems are therefore eliminated and packaging problems are reduced. The disadvantages associated with this technology, such as isolation of faulty elements, higher power requirement, and potential timing problems, are expected to be resolved through the advances of technology in the near future [34, 60, 94, 96, 97].

By placing each processing module on a wafer, three
advantages have become possible: i) independent program blocks can be assigned to different modules and processed simultaneously; ii) the instruction level communication within a block (wafer) is carried out at the on-chip-level speed; and iii) the VLSI I/O pin limitation problem is solved for both the local and the global communications.

The purpose of this chapter is to discuss the VLSI design and the time and space complexities of the major components of WDDM. Section 1 of this chapter studies the characteristics of the wafer-scale integration and its advantages and disadvantages. In section 2, the VLSI design of the major components of WDDM along with their time and space complexities are discussed. Finally, section 3 is the summary of the chapter.
1. WAFER-SCALE INTEGRATION

The Wafer-Scale Integration (WSI) technology can be traced back to nearly two decades ago, when Texas Instruments used full wafers to implement the first large-scale integrated circuits [94]. But due to the expensive fabrication and packaging process, WSI did not emerge as a practical technology until recently. Today, the development of economical "superchips" with the size of a person's hand is becoming a reality. These circuits, encompassing an entire wafer, will contain millions of logic gates. Therefore, WSI allows more functionality to be condensed into a single device. Furthermore, replacing a multi-chip system on a printed-circuit board with one integrated wafer eliminates interchip connections and increases the reliability of the system.

Two factors have been the driving force for the emergence of WSI: i) evolution of new designs, fabrications, and testing techniques and ii) the market demand for more complex systems. Due to physical laws, the shrinkage of circuits is approaching its limitations. Thus, the alternative is to increase the device size.
The size of a silicon wafer may range from 2 to 8 inches in diameter. Depending on the technology used (nMOS, cMOS, λ size), a wafer may contain up to 100 integrated circuits or chips. For example, it is possible to implant 25 to 100 microprocessors with the size and capability of an Intel 8086 on a wafer [94]; or, with an .8-micrometer cMOS process, a wafer can hold a 20 megabyte dynamic memory [94].

The wafer-scale integration research is under development in the industrial and academic laboratories in the United States, Japan, and Europe. The Trilogy System Corporation (formed by Sperry, Digital Equipment, and Honeywell Corporations) has announced a 40-wafer mainframe computer with a performance of 32 MIPS [25]. The National Telegraph and Telephone (NTT) of Japan has made a 4-megabit WSI ROM which holds the entire Kanji alphabet [94]. Other institutions currently involved in the WSI research include: Mosaic Systems Incorporated, Wafer Scale Integration Incorporated, Inmos Corporation, Sinclair of England, MIT, Stanford, Rensselaer Polytechnic Institute, etc. [30,85,94].

Advantages of Wafer-Scale Integration:

1) The WSI technology eliminates the cable and connector delays of multi-chip systems and increases the reliability of the system. Off-chip communications are often long, noisy, power-hungry, and slow. Also, a large area of
the chip is taken up by drivers which push signals off the chip onto the package pins. These drivers require large amounts of current and power. Furthermore, the connection pins are often vulnerable to mechanical failure.

In contrast, on-wafer communications are short, fast, and require low power consumption. There is also an extensive research being conducted for improving the on-wafer communications. For example, at the Stanford University amplifiers are added to the communication lines to speed up the communication delay [94]. Cooling the wafer is another method of improving the communication. At the subfreezing temperatures, the propagation delay approaches the gate delay (.5 ns to propagate a 4-inch long line) [85].

2) WSI eliminates the VLSI I/O pin limitation problem. Within a wafer, the entire communication is performed at the on-chip level, requiring no external pins. In addition, there is no need for large I/O drivers for the internal communications since the signals are not pushed off the chips. The inter-wafer communication is also improved since the number of I/O pins can be increased due to the large size of the wafers. A wafer can accommodate up to 840 I/O pads [85] and up to 2000 surface contacts [94].

3) WSI provides opportunities for a more economical technology [34]. Mosaic System's Wafer Hybrid Interconnection Packaging (WHIP) technology allows 32 wafers to be packaged in an 8-inch cubed volume with a high level of
interconnection at a minimal cost [85]. For example, a mainframe with 500,000 gates can be implemented on 5 wafers. The cost of this system will be the cost of the chips plus the cost of the interconnection and packaging which would be less than $20,000 if WHIP technology is used [85].

The Problems of the WSI technology:

At the present time, the WSI technology suffers from some problems and limitations. However, they are expected to be resolved through the ongoing research which is being conducted to overcome these problems. In the remaining of this section we will discuss these problems and their possible solutions.

1) Heat dissipation: The power density on a single integrated wafer is much higher than a similar IC system. Thus, a wafer may generate as much as 1000 watts of heat [94]. There are many ways to remove the heat dissipated from a wafer. To begin with, the cMOS technology should be used. A cMOS IC would dissipate two orders of magnitude less power than an equivalent nMOS IC operating at the same clock rate [94]. This is because cMOS has almost no power dissipation when a gate is off, while nMOS requires more power when a gate is off. Other attempts to remove the heat from a wafer include the Trilogy System's water cooling system and the Mosaic System's packaging which immerses the
wafer into liquid nitrogen.

2) Redundancy [60,96,97]: Unlike a faulty chip, a wafer is too costly to simply discard when some of its components are faulty. Therefore, some means for repairing faulty wafers is needed. A solution is to introduce redundant circuitry so that a defective chip can be replaced by its clone. In general, there are two types of defects on a wafer: i) random defects which can be overcome by routing the logic around them and ii) clustered defects which can be reduced by improving the manufacturing process.
2. VLSI DESIGN AND TIME/SPACE COMPLEXITY

This section discusses the VLSI time and space complexities of WDDM through the analysis of some of its major units. Due to the highly modular organization of the system, we can take advantage of the existing designed and tested components for different system modules. In addition, we have proposed a number of VLSI designs for some special-purpose components. It should be noted that the figures presented for the time and space complexities are "rough estimates" in the sense that they are a function of the advances in technology and development of "suitable" VLSI algorithms.

The Memory Units

The memory components of WDDM are composed of the main, queue, cache, and data structure memories. Because of the modularity of the system, the memory units are physically independent of other system units. Therefore, we can take advantage of the recent developments in the memory technology to implement these units. As an example of a
VLSI implementable dynamic memory cell, Figure 6.1 represents the transistor diagram of a basic 3-transistor RAM cell designed in [109]. In this Figure, the Bio (Bit I/O) indicates the input/output to/from a cell, the R and W are clocked Read and Write control lines, and GND is the ground line. In order to write into the cell, Bio is driven to the desired value and W is clocked. To read from the cell, Bio is precharged, R is clocked, and the inverted value appears on Bio.

The size of a RAM cell is 18.5 \( \lambda \times 28 \lambda \) [109]. However, pairs of memory cells can share the ground line, resulting in a 4\( \lambda \) overlap. Thus, the dimension of a pair of RAM cells is 33\( \lambda \times 28 \lambda \). With \( \lambda = 2.5 \mu m \) [109], the cell size is .05 mm \( \times \).07 mm, allowing about 24K bits to be packed in a 1 cm \( \times \) 1 cm area. As previously mentioned, a silicon wafer can hold up to 20 megabytes of dynamic RAM [94].

**The E-unit Design**

The major component of each E-unit is the elementary processor which performs simple logic and arithmetic operations. These simple operations include integer addition, subtraction, negation, logical operation, etc. In order to accommodate such capabilities into an E-unit, we will take advantage of the OM2 Data Path ALU [99]. The
Figure 6.1: A 3-transistor RAM cell.
details of the design and the operations of OM2 are described in [99]. However, we will briefly discuss those segments of the OM2 design which are utilized in WDDM.

The function of the elementary processor is to generate different logic combinations of the two input signals (e.g. A and B). Figure 6.2 shows the block diagram of a 1-bit OM2 ALU. The ALU consists of three function blocks (P, K, and R) and a carry propagate block (C). The inputs to the ALU are: i) the input operands and their complements (A, B, \(^{\bar{A}}\), \(^{\bar{B}}\)); ii) a 12-bit control signal (\(P_0-P_3\), \(K_0-K_3\), and \(R_0-R_3\)); iii) a precharge signal; and iv) the carry in signal (Cin). The ALU's outputs are the carry-out to the next bit and the result of the operation (out). The 12-bit control signals are used to determine the type of the operation to be performed by the ALU. Table 6.1 enumerates the control codes and the corresponding operations which can be performed by this unit.

The Functions of the P, K, and R Blocks:

a) The K function block generates the carry-kill control signal to the carry function block (c). The inputs to this block are the operands (A and B), and its output (K) is used to kill (ground) the carry in the carry chain. This signal is necessary since the carry chain is precharged before the ALU operations and must be drained (as needed)
Figure 6.2: The block diagram of a 1-bit OM2 ALU.
Table 6.1: ALU operation list\(^{[106]}\).

<table>
<thead>
<tr>
<th>OP</th>
<th>K</th>
<th>P</th>
<th>R</th>
<th>LSB Cin</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>A±B</td>
<td>14</td>
<td>9</td>
<td>6</td>
<td>-</td>
<td>Add with carry</td>
</tr>
<tr>
<td>A⇒B</td>
<td>11</td>
<td>6</td>
<td>6</td>
<td>1</td>
<td>A⇒B+1</td>
</tr>
<tr>
<td>B⇒A</td>
<td>13</td>
<td>6</td>
<td>6</td>
<td>1</td>
<td>B⇒A+1</td>
</tr>
<tr>
<td>A⇒Cin</td>
<td>10</td>
<td>5</td>
<td>6</td>
<td>-</td>
<td>Increment A if LSB Cin=1</td>
</tr>
<tr>
<td>B⇒Cin</td>
<td>12</td>
<td>3</td>
<td>6</td>
<td>-</td>
<td>Increment B if LSB Cin=1</td>
</tr>
<tr>
<td>A⇒Cin</td>
<td>5</td>
<td>10</td>
<td>9</td>
<td>-</td>
<td>Decrement A if LSB Cin=1</td>
</tr>
<tr>
<td>B⇒Cin</td>
<td>3</td>
<td>12</td>
<td>9</td>
<td>-</td>
<td>Decrement B if LSB Cin=1</td>
</tr>
<tr>
<td>A and B</td>
<td>0</td>
<td>8</td>
<td>3</td>
<td>0</td>
<td>Bit-wise AND</td>
</tr>
<tr>
<td>A or B</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>Bit-wise OR</td>
</tr>
<tr>
<td>A xor B</td>
<td>0</td>
<td>6</td>
<td>3</td>
<td>0</td>
<td>Bit-wise XOR</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>5</td>
<td>3</td>
<td>0</td>
<td>Not A</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>Not B</td>
</tr>
<tr>
<td>A or B</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Bit-wise NOR</td>
</tr>
<tr>
<td>A xor B</td>
<td>0</td>
<td>9</td>
<td>1</td>
<td>0</td>
<td>XNOR</td>
</tr>
<tr>
<td>B⇒A</td>
<td>0</td>
<td>2</td>
<td>12</td>
<td>0</td>
<td>B implies A</td>
</tr>
<tr>
<td>A⇒B</td>
<td>0</td>
<td>2</td>
<td>12</td>
<td>0</td>
<td>A implies B</td>
</tr>
<tr>
<td>A &amp; B</td>
<td>0</td>
<td>8</td>
<td>12</td>
<td>0</td>
<td>Bit-wise NAND</td>
</tr>
<tr>
<td>Zero</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>Zero function</td>
</tr>
<tr>
<td>One</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>One function</td>
</tr>
<tr>
<td>A⇒B</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>A⇒B</td>
</tr>
<tr>
<td>B⇒A</td>
<td>0</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>B⇒A</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>10</td>
<td>3</td>
<td>0</td>
<td>Just A</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>12</td>
<td>3</td>
<td>0</td>
<td>Just B</td>
</tr>
</tbody>
</table>
during the operation.

b) The P function block has the operands (A and B) as its inputs and P as its output. The carry-Propagate signal (P) generated by this block has two functions. First, it is used to control the propagation of the carry in the C function block. Second, it is used to form the output of the ALU in the R function block. Thus, the P signal propagates through the carry block and is used as an input to the R function block.

c) The R function block takes Cin and P as its inputs and generates the output for the operation (out). Cin is the Carry-in to the bit and P is a logic-function of the two input operands A and B as generated by the P function block.

The Structure of the P, K, and R Blocks:

These three function blocks share the same architecture and can fully generate the logic combinations of their input signals. Figure 6.3 shows the block diagram of such a function block. The input signals are the two input operands (X and Y) and the 4-bit control signal (C0-C3). The output signal (f) is a function of X and Y. Figure 6.4 depicts the details of the design of the function block of Figure 6.3. This circuitry implements the sixteen logic functions of the two input variables. In order to generate any logic function of the two input variables, it is
Figure 6.3: Block diagram of a general function block.
Figure 6.4: Circuit diagram of the general function block of Figure 6.3.
sufficient to place the truth table entry for that function on the vertical control lines. For example, if A Exclusive-OR B is desired, C₁ and C₂ should be high, and C₀ and C₃ should be low. To generate A AND B, a logic-0 will be applied to C₀, C₁, and C₂, and a logic-1 will be applied to C₃.

The Function and Structure of the Carry Block:

The function of this block is to generate the carries for some of the ALU operations such as addition. The produced carries are used in the R function block to form the result of the operations.

As depicted in Figure 6.2, the inputs to this block are a precharge signal, carry-in, and \( \overline{K} \) and \( \overline{P} \) which are generated by the K and P function blocks, respectively. The outputs of the block are carry-out to the next bit and p and Cin which are routed to the R function block.

In the OM2 design, simulation of several carry look-ahead circuits has indicated that they increase the complexity of the system without significant gains in performance [99]. Thus, a Manchester-type carry chain [99] was selected for carry propagation circuitry. In nMos technology, a Manchester-type carry chain propagation of high carry signals are much slower than low carry signals. For this reason, a precharge signal is used to drive the
carry-out signal high during clock period $\phi 1$. At $\phi 2$, when the output (carry-out) is to be generated, the carry-out may remain high without any delay or brought to low very rapidly. This scheme is represented in the carry block transistor diagram of Figure 6.5.

The $P$, $\bar{P}$, Cin, and $\bar{\text{Cin}}$ output signals are simply generated by passing input signals $\bar{P}$ and Cin through an appropriate number of inverters. The Carry-out signal is controlled by three gates labelled $G_1$ through $G_3$ in Figure 6.5. $G_1$ simply precharges the Carry-out during $\phi 1$. $G_2$ generates the carry-kill signal ($K$) from its input ($\bar{K}$). When $K$ is high, the pull-down transistor of $G_1$ becomes closed and the Carry-out signal is grounded during $\phi 2$; i.e. Carry-out is killed. $G_3$ simply forms the carry-propagation signal ($P$) from its input ($\bar{P}$). If $P$ is low, then the pass transistor on the carry chain remains open. In this case, the Carry-out will be decided by the carry-kill signal which will either ground the Carry-out or allow it to remain high. If $P$ is high, the pass transistor on the carry chain is closed and Carry-out becomes equal to Cin.

It should be noted that in $G_2$ and $G_3$, the outputs are generated by two NOR gates that have $\bar{K}$ and $\bar{P}$ as one input and precharge as the second input. This scheme is used to assure that the kill and propagate signals are disabled during $\phi 1$ period when precharging takes place.
Figure 6.5: The stick diagram of the carry chain block.
The Implementation of the Control Signals:

The control signals (P₀-P₃, K₀-K₃, and R₀-R₃) are used to control the operations of the ALU. They are generated by the E-unit's controller through decoding the instruction's opcode. Therefore, they determine the type of the operation to be performed by the ALU.

In order to propagate these control signals among the ALU bits, control line drivers are required. The drivers for P, K, and R control lines have three functions:

a) During \( \phi 1 \), they must receive the decoded opcode specifying the state of each control line.

b) This input control signal must be latched while ALU is being precharged (\( \phi 1 \)).

c) The control signal must be applied to the P, K, and R control lines as soon as the ALU is activated (\( \phi 2 \)).

Figure 6.6 depicts the driver function block for the control lines. Again, the output of this block, consisting of pass transistors, is more effectively driven low than high. For this reason, the output of the drivers are precharged to high during \( \phi 1 \). At \( \phi 2 \), when the control signals are to be activated, they may remain high or be driven low very rapidly depending on the state of the control signal.

In the driver function block of Figure 6.6, the opcode is first latched through a pass transistor whose gate is
Figure 6.6: A control line driver.
connected to $\phi 1$. Thus, during $\phi 1$, the opcode is valid as one of the inputs of a NOR gate whose other input is $\bar{\phi} 2$. This scheme guarantees that the output of the NOR gate is low during the $\phi 1$ period. Since the output of the NOR gate is fed to an inverting superbuffer, the output of the driver is guaranteed to be high during $\phi 1$. At the start of $\phi 2$, the output of the NOR gate will depend on the opcode alone. This output then determines the output of the driver during $\phi 2$.

The Geometry/Timing of the ALU:

Figure 6.7 represents the transistor diagram of the 1-bit ALU whose block diagram was depicted in Figure 6.2. For an $n$-bit ALU, $n > 1$, the control lines need not be generated for each bit slice, but they may be generated once at the top or bottom of the array and shared among all the bits. The system buses, connecting the registers in the controller to the ALU, are run horizontally on the polysilicon level. The control lines along with the power, ground, and clock lines are run vertically on the metal level. Each ALU bit occupies an area of 0.1 mm x 0.6 mm [109]. Considering the area needed for the drivers of the control lines [109], the total area for a 32-bit ALU is estimated to be 1500 $\lambda$ x 400 $\lambda$ or 3.8 mm x 1 mm.
Figure 6.7: The transistor diagram of the ALU[109].
The minimum estimated clock phase periods are $\phi_1 = 50 \tau$ and $\phi_2 = 100 \tau$ for each block of 4 ALU bits [99], where $\tau$ is the basic inverter transit time. With $\tau = .3$ ns, the minimum total clock period for 32 bits is $850 \tau$ or 255 ns.

So far, we have been concerned with the design of the ALU section of an E-unit. The other elements of an E-unit are the input/output ports and the controller. The basic components of these elements are: i) a set of registers (buffers) used in the I/O ports and the controller (opcode, operands, destinations); ii) a comparator for matching the address of the input token against the E-unit's tag number; and iii) a decoder for decoding the opcode.

i) The registers: These temporary buffers may be constructed from the basic 3-transistor RAM cell which was previously discussed. According to that design, an n-bit register will occupy an area equal to $16.5n \times 28 \lambda$. The buffer area required by the controller is about $.5 \times 1.3$ mm for 2 operands, 3 destination addresses, opcode, and an extra register to keep the count of the unavailable operands.

ii) The input port: This unit is responsible for matching the address of the token on the common bus against the E-unit's tag number. The design and the time/space complexity of a match unit for this purpose will be discussed later on in the design of an associative memory.
utilized in the host computer. However, it should be mentioned that based on this design, it takes $100 \tau$ (or 30 ns) to perform the comparison for 32 bits, and that the total area for the input port is estimated to be .5 mm X .5 mm.

iii) The decoder: The transistor diagram of a NOR-gate implemented 1-of-4 decoder is depicted in Figure 6.8. This unit requires an estimated area of 40 $\lambda$ X 100 $\lambda$, or .1 mm X .25 mm. The decoder delay is $1\tau$ (.3 ns) for each $Z_i$, $0 \leq i \leq 3$.

In conclusion, the total geometry area of an E-unit is about 4 mm X 4 mm. This area includes the input port, the controller and its associated registers, the ALU, and the output port.

The Sub-net

As previously discussed, the sub-net is an arbitration network providing communication among the units of a processing module. It consists of $(\log_2 n)$-2 stages of standard 2-by-1 arbitration switches followed by two stages of 2-by-2 square switches. Figure 6.9 depicts a 2-by-1 "selector" unit which can be used as an arbitration switch. The (C) control line is generated by a function block which gives the tokens destined for E-units the highest priority and the host destined tokens the lowest priority. Figure
Figure 6.8: The transistor diagram of a 1-of-4 decoder.
Figure 6.9: Selector transistor diagram.
6.10 shows the block diagram and the truth table for a 4-input priority generator function block. The transistor diagram of this function block is represented in Figure 6.11.

The area of an arbitration switch is \( 30 \lambda \times 100 \lambda \) for routing one bit of the input at a time. If \( n \) bits are to be propagated in parallel, the area of a switch will be \( 30n \lambda \times 100 \lambda \). For \( n=64 \), the switch area is about \( 2000 \lambda \times 100 \lambda \). The data propagation delay in an arbitration switch is about 10 ns, including the time to latch the input signals, generate the priority control signals, and propagate the data. For the square switches, a design similar to the one proposed in [93] can be used. The total propagation delay of this switch is 45 ns [93].

Let \( n \) be the total number of the E-units, functional units, and the two input ports on a processing module. The length of the first stage of the network will then be \( 2000 \lambda (n/2) \). With \( n=32 \), the length of the network is 32000 \( \lambda \) or 8 cm. The network requires \( (\log_2 n)-2 \) stages of arbitration and 2 stages of square switches. If each stage is 200 \( \lambda \) wide (including the switch and the communication lines), the network width is 200 \( \lambda (\log_2 n) \). Again with \( n=32 \), the network width is 1000 \( \lambda \) or 2.5 mm.

The network delay for a token of 64 bits is \( 10((\log_2 n)-2)+90 \) ns if there is no conflict with the passage of the other tokens. With \( n=32 \), this delay is 120 ns.
Figure 6.10: The block diagram and the truth table of a priority generator function block.
Figure 6.11: The transistor diagram of the priority generator function block.
The Functional Units

These units are responsible for the execution of such complex operations as the floating point addition/subtraction, multiplication, division, etc. Recent developments in the VLSI technology have been the major motivation for the design of the "suitable" Functional Units (FU) for implementation by the current technology. A suitable FU reduces communication as well as computation based on the replication of a basic function in space or time. As an example of a VLSI implementable FU, we have introduced a 2's complement systolic multiplier unit in [77, 80]. The architecture and the VLSI layout of this multiplier unit is discussed in the remainder of this sub-section.

In the traditional multipliers the operation is performed either in bit-oriented or word-oriented fashions. The bit-oriented method is the hardware implementation of a software multiplier where, in each iteration, a bit of the multiplier determines whether or not a copy of the multiplicand should be accumulated. Due to the concept of carry propagation, this approach is too lengthy.

In the word-oriented approach, first an n X n matrix \( M \) (matrix of partial products) is generated and then elements in each column are added. Even though \( M \) is generated in parallel, the second step is the main bottleneck in this
process:
\[ m_{ij} \in M = \begin{cases} 1 & \text{iff } X_i = Y_j = 1 \\ 0 & \text{otherwise} \end{cases} \quad (1) \]

Such a bottleneck is due to the dependency between the operations within the columns and across the columns. Figure 6.12 depicts this process. Without loss of generality, we assume that M has a parallelogram rather than a rectangular shape, and the columns are numbered from right to left. The result of the multiplication \( Z_{2n}Z_{2n-1}...Z_1 \) is then defined as:

\[
Z_k = S_k \mod 2 \quad 1 \leq k \leq 2n
\]

where

\[
S_k = (\sum_{i=1}^{k} X_i Y_{k+1-i}) + C_{k-1} \quad X_j = Y_j = 0 \text{ for } j > n \quad (2)
\]

and

\[
C_{k-1} = \lfloor S_{k-1}/2 \rfloor \quad 2 \leq k \leq 2n
\]

\[ C_0 = 0 \]

From equation 2, we can conclude that: i) zero elements \((X_iY_j)\) in column k do not affect \( Z_k \) and ii) each pair of 1's in column k contributes a carry to the column k+1. Hence, \( Z_k \) is a function of the elements in column 1,2,...,k. Therefore, the execution time of the whole process is a function of the generation and propagation of carries and the summation of the elements in each column.

Recent developments in technology, namely VLSI, have steered the design of multiplier units in two directions: a) the adaptation of some of the previously proposed algorithms for VLSI implementation \([22,27]\) and b) the design of new
Figure 6.12: A traditional word-oriented multiplication.
algorithms based on the constraints imposed by current technology [69, 77, 80]. The investigations in the first direction have resulted in some VLSI multiplier units with reasonable time and space complexities. However, such efforts are still "hard" to implement since the originators of these algorithms did not consider the constraints imposed by current technology in their architectures. The research for the generation of new algorithms has resulted in a new class of multiplier units with systolic organization in mind [89]. A systolic organization offers simplicity, modularity, and high performance, the characteristics which are demanded by VLSI technology. Therefore, a multiplier unit based on systolic organization should offer a regular, fast and cost-effective unit for VLSI implementation. Next, we will introduce a multiplier unit with such characteristics.

The Proposed Scheme

Equation (2) indicates that: i) $Z_k$ is either 0 or 1; ii) zero elements $X_iY_j$ do not affect $S_k$; and iii) every pair of 1's in \( \sum_{i=1}^{k} X_i Y_{k+1-i} \) contributes a 1 to $C_{k+1}$ (1 ≤ k ≤ 2n). These observations are the basis for our scheme. First, we remove all the zero elements from the matrix of partial products. Second, for each pair of 1's in column k, a 1 is propagated to column k+1. Third, if the number of 1's in
column $k$ is an odd number, then $z_k$ is 1; otherwise, it is 0. The proposed architecture performs these operations in a pipelined fashion.

Figure 6.13 depicts an example of our scheme. The bold-faced 1's in the intermediate part represent the propagated carries. In our hardware model, the partial products are generated serially (one row at each pipeline beat). This might imply a lengthy operation. However, because of the overlap and parallelism which have been incorporated in our design, such an approach does not degrade the performance. During each pipeline beat, one output bit is generated. Such a strategy has three advantages: 1) it reduces the number of columns in the matrix of partial products; 2) it reduces the number of input/output ports (pins) by a factor of 2; and 3) it reduces the depth of each column $k$ ($1 \leq k \leq n$) to $k$ rather than $n$. The overall organization of the model is depicted in Figure 6.14. The architecture is composed of three modules: set of registers, shift controller, and output generator.

**Set of Registers:** is a collection of latches ($R_i$) ($1 \leq i \leq n$) which holds the non-zero elements of the matrix of partial products. During each pipeline beat, non-zero elements of a bit pattern representing a partial product ($p = p_1 p_2 \ldots p_n$) are pushed into these latches, while the contents of the latches are inched one position to the right. Such constant movement to the right, delivers one
Figure 6.13: An example of the proposed scheme.
Figure 6.14: The overall architecture of the proposed multiplier unit.
column of the partial products to the shift controller unit in each pipeline beat.

**Shift Controller:** consists of a set of \((n-1)\) latches \((R_i)\) \((n+1<i<2n-1)\) similar to those in the set of registers. During each pipeline beat, as data flows through these latches, the carries are determined and propagated appropriately from a register to its left neighbor. The bits in the even positions are used to determine the carries. This is possible because of the novel arrangement of the bit pattern in each column. The movement of data in this module is the same as the movement of data in the set of registers module, except that the contents of \(P_i\)'s \((n+1<i<2n-2)\) are determined by the carries. It is notable that in each beat, no more than one carry is propagated per column. This provides a uniform timing sequence in different modules and smooths the flow of the data throughout the pipe.

**Output Generator:** counts the number of 1's in the last column of the shift controller and generates the final result accordingly. Because of the unique format of the bit patterns in each column, the "oddity" or "evenness" of 1's in a column is determined by the position of the last 1 in that column.

Table 6.2 shows the sequence of the operations in the proposed model. The multiplication of two \(n\)-bit numbers will be carried out in \(3n\) beats. During each beat, three
Table 6.2: Sequence of the operations in the proposed model.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R_{i,j} \leftarrow 0 )</td>
<td>( 1 \leq i \leq 2n-2 )</td>
<td>Initialization of ( R )'s and ( P ) registers</td>
</tr>
<tr>
<td>( P_i \leftarrow X_i A Y_{2n} )</td>
<td>( 1 \leq i \leq n )</td>
<td>( n+1 \leq i \leq 2n-2 )</td>
</tr>
<tr>
<td>( P_i \leftarrow 0 )</td>
<td>( n+1 \leq i \leq 2n-2 )</td>
<td></td>
</tr>
</tbody>
</table>

For \( k = 1 \) To \( 2n \) Do

Coebegin

Coebegin

\( P_i \leftarrow X_i A Y_{2n-k} \) \( 1 \leq i \leq n \)

\( P_i \leftarrow R(i, 2(i-n)) \) \( n+1 \leq i \leq 2n-2 \)

Coend

Coebegin

\( R(1,1) \leftarrow P_1 \)

if \( P(i+1) = 1 \) then

\( R(i+1, j+1) \leftarrow R(i,j) \)

\( R(i+1, 1) \leftarrow P(i+1) \)

else

\( R(i+1, j) \leftarrow R(i,j) \)

Coend

Begin

\( O_e \leftarrow S_{2e-1} A S_{2e} \) \( 1 \leq e \leq 2n-2 \)

\( \text{Out}_k \leftarrow V \ 0_e \)

\( k = 1 \)

end

Coend

Coend

I and \( Y \) have been assumed to be positive numbers. In addition, we assumed \( Y \) has been padded by \( n \) zeroes.
operations occur concurrently: i) the contents of the latches ($R_i$'s), $1 \leq i \leq 2n-1$, are inched to the right under the control of the P register; ii) the content of the P register is generated for the next beat; and iii) one output bit is generated. It should be noted that the first bit of the result will be generated after a start-up period of $(n-1)$ beats. This is due to the initial flow of the data through the shift controller unit. However, for a series of multiplications, this cold start-up cost is incurred only once. Thus, a new multiplication can be initiated in the pipe after each $2n$ beats.

The VLSI layout of the proposed architecture is a replication of 2 basic cells (Type-1, Type-2) in a two-dimensional space. Type-1 cells are utilized in the set of registers module and Type-2 cells are the basic block of the shift controller module. Figures 6.15 and 6.16 depict the stick diagram of the Type-1 and Type-2 basic cells, respectively. The first output bit is available to the output generator after a delay of $5(n-1)\tau$ (start-up period), while the subsequent bits are available at $5\tau$ intervals, where $\tau$ is the basic inversion transit time. Disregarding the start-up delay, the execution time would be $10n\tau$. The geometry area of each cell is estimated at $70 \lambda \times 80 \lambda$. This will result in a total geometry of $140n\lambda \times 160(n-1)\lambda$. Table 6.3 shows the execution time and estimated geometry area of the proposed scheme for $\tau = .3$ ns and $\lambda = 2.5 \mu m$ [109].
Figure 6.15: Stick diagram of the basic cell type 1.
Figure 6.16: Stick diagram of the basic cell type 2.
Table 6.3: Execution and geometry for different configurations.

<table>
<thead>
<tr>
<th>n</th>
<th>Execution time (( \mu \text{sec} ))</th>
<th>Estimated Geometric Area (( \text{Cm}^2 ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>21</td>
<td>0.26 x 0.26</td>
</tr>
<tr>
<td>12</td>
<td>36</td>
<td>0.45 x 0.48</td>
</tr>
<tr>
<td>32</td>
<td>96</td>
<td>1.21 x 1.34</td>
</tr>
<tr>
<td>64</td>
<td>192</td>
<td>2.42 x 2.72</td>
</tr>
</tbody>
</table>
2's Complement Multiplication

One of the drawbacks of the proposed multiplier unit is that it only handles unsigned numbers. We will now introduce a modified version of this unit which is capable of handling positive and negative numbers in 2's complement notation. The proposed unit also reduces the time complexity as well as the hardware requirements of the operation. Such a scheme is accomplished through incorporation of the modified Booth's algorithm in the design of the previous multiplier.

Compared to the classical add and shift method for multiplication, Booth's algorithm [20] (known as a decoding multiplication method) increases the speed of the operation by reducing the number of additions. The algorithm examines two bits of the multiplier at each iteration and eliminates the need for addition for continuous strings of 1's and 0's in the multiplier. The algorithm can be further improved by examining more than two bits of the multiplier at a time. In that case, the number of iteration cycles would be reduced. Modified Booth's algorithm [118] examines 3 bits of the multiplier at each iteration and performs one of the 8 operations according to the decoding scheme represented in table 6.4. Thus, not only are the iteration cycles cut in half, but the operation is also speeded up in the presence of strings of consecutive 0's or 1's in the multiplier. In
Table 6.4: The 3-bit pattern of modified Booth's algorithm.

<table>
<thead>
<tr>
<th>Bit pattern</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>No operation, shift 2 positions to the right.</td>
</tr>
<tr>
<td>001</td>
<td>Add the multiplicand, shift 2 positions to the right.</td>
</tr>
<tr>
<td>010</td>
<td>Same as above</td>
</tr>
<tr>
<td>011</td>
<td>Add twice the multiplicand, shift 2 positions to the right.</td>
</tr>
<tr>
<td>100</td>
<td>Add twice the two's complement of the multiplicand, shift 2 positions to the right.</td>
</tr>
<tr>
<td>101</td>
<td>Add two's complement of the multiplicand, shift 2 positions to the right.</td>
</tr>
<tr>
<td>110</td>
<td>Same as above</td>
</tr>
<tr>
<td>111</td>
<td>K: operation, shift 2 positions to the right.</td>
</tr>
</tbody>
</table>
our discussion, we assume that the multiplier has an even
number of bits. However, a multiplier with an odd number of
bits can be easily adjusted by extending its sign bit.

Figure 6.17 presents the structure of the new proposed
architecture. In order to incorporate the modified Booth's
algorithm in the previous model, the original design has
been modified as follows:

1) The Decoder: At each pipeline beat, register $P_i$, $1 \leq i \leq 2n$, is set to one of 0, X, $\bar{X}$, 2X, or 2$\bar{X}$ according to the
decoding scheme of table 6.4, where X is the multiplicand
and $\bar{X}$ stands for 2's complement of X.

ii) The PP Section: The Partial Products (PP) section
is similar to the set of registers. However, in order to
propagate the sign for negative numbers, the number of
columns in this section has been extended by n. Obviously,
the multiplicand (X) is also extended by its sign bit to 2n
bits. For a more efficient geometry area, we can use the
following configuration for the registers in the PP section:

\[
\begin{align*}
|R_i| &= (i+1)/2 \\
|R_{i+1}| &= (i+1)/2
\end{align*}
\]

and

\[
R_i = n/2 \quad \text{for } n+1 \leq i \leq 2n,
\]

During each pipeline beat, the contents of the latches
are shifted 2 positions to the right. Such a movement
between $R_i$ and $R_{i+2}$ is under the control of $P_{i+1}$:
Figure 6.17: The overall organization of the proposed multiplier.
\[1 \leq i \leq 2n\] and
\[1 \leq j \leq R_1\]

\[
\begin{align*}
R(i,j) & \rightarrow R(i+2,j+1) \quad \text{if } P_{i+1} = 1 \\
R(i,j) & \rightarrow R(i+2,j) \quad \text{otherwise}
\end{align*}
\]

**iii) The Carry Section:** This section has the same function as the shift controller. It is a collection of \((n-2)\) registers \(R_i\) \((2n+1 \leq i \leq 3n-2)\) which generates and propagates carries properly. Again, for a more cost effective system, the length of each \(R_i\) in this section is defined as:

\[
|R_i| = \frac{n}{2} + \frac{i-2n+1}{2}
\]

\[
i = 2n+1, 2n+3, \ldots, 3n-1
\]

\[
|R_{i+1}| = \frac{n}{2} + \frac{i-2n+1}{2}
\]

The bits at the even positions in each register determine the carries to that register's left neighboring column. According to the entries in Table 6.4, during each pipe beat the contents of the PP and carry sections are shifted two positions to the right. This implies the generation and propagation of two carries in each pair of columns. These carries, namely the implicit and explicit carries, are held in the P register in the carry section. Consider 2 bits \(R(i,j)\) and \(R(i+1,j)\) in the carry section \((i=n+1, n+3, \ldots, 3n-3\) and \(j=2, 4, \ldots, n)\). The implicit carry from \(R_{i+1}\) to \(R_i\) is determined by the contents of \(R(i+1,j)\). However, the explicit carry from \(R_i\) to \(R_{i-1}\) is generated if \((R(i,j)=1) \text{ OR } (R(i,j-1)=1 \text{ AND } R(i+1,j)=1)\). In the latter case, the effect of the implicit carry from \(R_{i+1}\) to \(R_i\) has
been taken into consideration. In general:

Implicit carry: \( P_{2n+i} \leftarrow R(2n+i,i) \) \( i=2,4,6,\ldots,n-2 \)

Explicit carry: \( P_{2n+i} \leftarrow R(2n+i,i+1) \lor (R(2n+i,i) \land R(2n+i+1,i+1)) \) \( i=1,3,\ldots,n-1 \)

The contents of \( P_i \)'s in this section determine the direction of data movement among the latches:

\[ 1 \leq i < n-1 \]

\[ R(2n+i,j) \rightarrow R(2n+i+2,j+1) \] if \( P_{2n+i} = 1 \)

and

\[ 1 \leq j < R_{2n+i} \]

\[ R(2n+i,j) \rightarrow R(2n+i+2,j) \] otherwise

At the end of the pipeline beat, \( P_{2n+i} \) is saved in \( R(2n+i+1,j) \) if \( P_{2n+i} = 1 \), \( 1 \leq i < n-2 \).

iv) The Output Generator: This unit consists of 2 exact replicas of the output generator in the original system. Since the new multiplier unit produces 2 output bits at each pipeline beat, one output generator is attached to \( R_{3n-1} \) and one to \( R_{3n} \). Each output generator uses the position of the last 1 in \( R_{3n-1} \) or \( R_{3n} \) to determine the "oddity" or the "evenness" of the number of 1's in these registers. It then generates a 1 for the odd number of 1's and a 0 for the even number of 1's.

Table 6.5 presents the sequence of the operations in the proposed system. During a pipeline beat, three operations are carried out simultaneously: i) the contents of the registers in the PP section and the carry section are shifted 2 positions to the right under the control of the \( P \) register; ii) the contents of the \( P \) register are generated
Table 6.5: Sequence of the operations in the proposed model.

\[ R_{(i,j)} = 0 \quad 1 \leq i \leq 3n-2 \quad 1 \leq j \leq |R_i| \] /Initialization/

For \( k = 1 \) To \((3n/2)\) Do /\( n\) is even/

Cobegin

Cobegin

\[ P_1 = \text{decoder output} \quad 1 \leq i \leq 2n \]
\[ P_{(2m+1)} = R_{(2m+1,1)} \quad m = 1, 2, \ldots, n/2 \]
\[ P_{(2m+1)} = R_{(2m+1,1)} \lor (R_{(2m+1,1)} \land R_{(2m+1,1+1)}) \quad m = 1, 2, \ldots, n/2 \]

Cobegin

If \( P_1 = 1 \) Then \( R_{1,1} = P_1 \)
If \( P_{(i+1)} = 1 \) Then

\[ R_{(i+1,1)} = R_{(i+2,1)} \]
\[ P_{(i+1)} = P_{(i+2,1)} \]
else
\[ R_{(i+1,1)} = R_{(i+2,1)} \]
endif

Cobegin

begin

\[ O_e = R_{(3n-1,2e-1)} \lor R_{(3n-1,2e)} \]
\[ \text{Out}_k = \lor_{k=1}^{n/2} O_e \]
end

begin

\[ O_e = R_{(3n,2e-4)} \lor R_{(3n,2e)} \]
\[ \text{Out}_k = \lor_{k=1}^{n/2} O_e \]
end

Cobegin

Coend

end

* We assume \( Y \) has been padded by \( n \) zeros to the left and one zero to the right.
for the next beat; and iii) two bits of the result are generated.

**Time and Space Analysis of the Proposed Model**

According to the entries in Table 6.5, the first column is available to the output generator in \( \frac{n}{2} \) pipeline beats. The subsequent output bits are generated in the next \( n \) iterations, 2 bits per cycle. This implies a time complexity of the order \( O(1.5n) \), which is faster than the previous model by a factor of 2. Figures 6.18(a) and 6.18(b) depict the overall geometry area of the two models. The area complexity of the older model is \( O(2n^2) \), while the new model utilizes a geometry area of the order \( O(1.5n^2) \). This is a 25% improvement over the previous model. However, we admit the fact that the overhead of the decoder and two output generators overrides the above gain.

**VLSI Design of the Proposed Architecture**

The proposed scheme is composed of two basic blocks. First, the VLSI layout for each basic block is presented and then their time and space complexities are discussed. In order to reduce the duration of each pipeline beat and the space requirements, the registers between the pipeline stages are eliminated. This has been achieved by the
Figure 6.18(a): Geometry area of the previous model.

Figure 6.18(b): Geometry area of the proposed architecture.
synchronization of the propagation of the signals among all the stages in the pipe. A pipeline stage consists of an array of a maximum of \( n \) basic cell blocks, where \( n \) is the size of the operands. All the pipeline stages in the PP section and the carry section utilize the basic cell blocks depicted in Figures 6.19(a) and 6.19(b), respectively. The \( P \) and \( R \) signals are the same as those defined in Table 6.5, the delay elements are used for synchronization purposes, and the \( P^+_{i+1} \) and \( P^-_{i-1} \) signals in Figure 6.19(b) stand for implicit and explicit carries, respectively.

Figures 6.20(a) and 6.20(b) depict the stick diagram of the cells defined in Figures 6.19(a) and 6.19(b). The VDD and GND are shared among the cells of a row by running them horizontally in metal. The \( P \) and \( \overline{P} \) control signals are run vertically at the diffusion level and thus can be shared among the cells in the same stage.

The execution time of the system is estimated based on the Type-2 cell block (Figure 6.19(b)). In this cell, the output is available in \( 4\tau \) (2 stages of NAND gates). However, the explicit carry and its complement require an additional \( 3\tau \) delay (2 stages of NOR gate and one NOT gate). Therefore, the effective pipeline beat is \( 7\tau \). The system requires \( 3/2n \) pipeline beats in order to multiply two \( n \)-bit operands (with \( n/2 \) cycles for the start-up periods). Thus, the total execution time will be \( 21/2n\tau \). In the case of a series of multiplications, the effective execution time
a) Cell type 1.

b) Cell type 2

Figure 6.19: The basic cell blocks.
Figure 6.20(a): Stick diagram of the basic cell type 1.
Figure 6.20(b): Stick diagram of the basic cell type 2.
will be $7n \tau$. Table 6.6 shows the execution time and geometric area of this model against models proposed in [69, 77]. The geometry area of the unit is estimated to be the same as the Hurson's model (previous multiplier). Even though a 25\% space improvement was obtained for the PP section and the carry section, the extra space is used for the decoder, the additional output generator, and the routing of the signals between each pair of stages.

**The Host Computer**

The major function of the host computer is to carry out the memory management tasks. To accomplish this, the host is required to manipulate the tables previously introduced in Chapter IV. For example, the cache table is used to keep track of the blocks which are stored in the cache memory. Thus, the host is required to add an entry to the cache table for each inactive block, to search for a block's identification, and to read an entry's information from the table. In order to facilitate these operations and to speed up the execution, associative memories with READ, WRITE, and COMPARISON capabilities can be used. We have previously designed a VLSI implementable associative memory with such capabilities. The details of the design can be found in [74, 75, 78]. Nevertheless, the overall VLSI design of the system will be briefly discussed in the following paragraphs.
Table 6.6: Comparison with other multiplier units.

<table>
<thead>
<tr>
<th>n</th>
<th>Proposed model+</th>
<th>Hurson's model*</th>
<th>Harata's model**</th>
<th>Proposed model+</th>
<th>Hurson's model+</th>
<th>Harata's model**</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>33.6</td>
<td>48</td>
<td>120</td>
<td>≈ .6 x .65</td>
<td>.6 x .65</td>
<td>.58 x .63</td>
</tr>
<tr>
<td>32</td>
<td>67</td>
<td>96</td>
<td>140</td>
<td>≈1.3 x 1.5</td>
<td>1.3 x 1.5</td>
<td>1.1 x 1.1</td>
</tr>
<tr>
<td>64</td>
<td>134</td>
<td>192</td>
<td>---</td>
<td>≈2.7 x 3</td>
<td>2.7 x 3</td>
<td>------</td>
</tr>
</tbody>
</table>

* Reference 77.
** Reference 69.
+ 2 Multiplier units on a single chip.
Figure 6.21 depicts the design of a comparand cell and its interconnection to the memory cells. The organization of a memory cell is shown in Figure 6.22. Each memory and comparand bit is a dynamic register which repeatedly refreshes itself. Allowing a fan-in of three, the comparison of a bit in the associative memory and the corresponding bit in the comparand register requires a delay time of 6\( \tau \). In order to search a field, the first NAND operation is carried out in parallel for all of the bits of that field. Therefore, the total search delay time is \((3 + 3n)\tau + \tau\), where \(n\) is the number of bits participating in the search.

Figures 6.23 and 6.24 depict the stick diagram of the logic circuits of Figures 6.21 and 6.22, respectively. The VDD, GND, \(\phi 1\), \(\phi 2\), control lines, and the comparand contents are run vertically in metal. The mismatch signals and the tag lines are run horizontally at the diffusion level.

A simplified version of the tree structured multi-match resolver defined by Anderson [6] is used to detect the multi-matches. The relation \(A_j \wedge \overline{I}_j\) \((1 \leq j \leq n)\) suffices to find the first match. In this relation, \(A_j\) and \(I_j\) represent the \(j\)th bit of the tag register and \(I\) vector, respectively (Figure 6.25), and \(n\) is the number of rows in the associative memory. The \(I\) vector is a string of 0's followed by a string of 1's, one bit for each row of the
Figure 6.21: A comparand cell and its associated circuitry.
Figure 6.22: A memory cell and its associated circuitry.
Figure 6.23: Stick diagram of a comparand cell.
Figure 6.24: Stick diagram of a memory cell.
Figure 6.25: I-generator block.
memory. The first 1 in the jth bit of the I vector indicates that the (j-1)th word is the first match. Figure 6.26 depicts the stick diagram of the I-generator block defined in Figure 6.25. With a maximum fan-in of four, the time required to locate the first match is $5(2\log_2 n-1)\tau$.

The memory words can be read by ORing the contents of the words in a tree fashion. With a fan-in of four, it takes $14\tau$ to read a field of 8 bits [78]. The total delay time for a write operation is $4\tau$ [78].

For an associative memory of 64 words (each 16 bits long), it takes 1.2 ns to copy the comparand into a memory word, 4.2 ns to read a field, 7.5 ns to detect a match, and 10 ns to perform a search ($\tau = 0.3$ ns). The geometry area is estimated to be 10 mm X 5 mm [78].
Figure 6.26: Stick diagram of the I-generator block.
3. SUMMARY

In this chapter, a general discussion of the overall VLSI design of the WDDM has been introduced. Due to the modularity of the design, we were able to evaluate each module separately. In order to comply with the VLSI constraints and to facilitate the implementation of the system, wafer-scale integration is suggested to be used for fabrication of the processing modules. This technique uses the entire wafer area to implant a complete system as opposed to multi-chip systems. Therefore, the functionality of a single device is increased and the communication delays are reduced.

The VLSI layout of the system's major components along with their timing and geometry analysis have also been presented. The result of this analysis reveals that our proposal for implementation of the processing modules on silicon wafers is well within the limits of current technology. For example, with a conservative $\lambda$ parameter (2.5 micron), a system of 32 E-units can be easily implemented on a wafer. This size is sufficient enough to handle the majority of loops and small procedures.
Naturally, as the λ size is shrunk due to the advances in technology, the number of E-units on a wafer can be increased. For example, with λ = .8 micron, a 64-E-unit processing module will be reasonably practical.

It should be mentioned that many details of the VLSI design are left for completion in the future projects. However, the results obtained in this chapter can well serve as a guideline for development of such projects. Furthermore, the timing analysis of the WDDM components is used in the simulation of the system in the next chapter.
CHAPTER VII

THE SYSTEM PERFORMANCE EVALUATION

The performance of the WDDM as a general-purpose machine is highly dependent on the characteristics and the dynamics of the various applications which are run on the system. In fact, one of the major critiques of the data driven architecture is based on its poor performance on naturally sequential applications [62]. However, there exist many applications which are inherently parallel and carry a high degree of embedded parallelism. Some examples include weather forecasting, dynamic fluid simulations, pattern recognition, etc. [13,39,51]. It is not the intention of this chapter to debate such an issue. Accepting the usefulness of the data driven architecture, the major goal of this chapter is to evaluate the performance of the proposed architecture by simulating its behavior. Furthermore, since WDDM is based on the current technology, we are interested to evaluate the effects of the improvements of technology on our model.

In addition to the technology trend, we would like to
study the effects of the program dynamics such as data structure operations and procedure calls in our system. These operations directly affect the performance of the data driven systems since they represent non-primitive functions requiring massive data transmissions and memory space [62, 92].

Briefly speaking, in this chapter the performance of WDDM is studied through a simulation of the system behavior [123]. The purpose of this evaluation can be summarized as:

1) A study of the effects of the dynamics of the application programs: some of the parameters giving a dynamic nature to an application program are:
   i) Degree of parallelism at the instruction level.
   ii) Degree of parallelism at the program block level.
   iii) Data Structure operations.
   iv) Procedure calls.
We would like to investigate the system behavior with regard to these dynamic parameters.

2) A study of the different system configurations: This evaluation is based on the effects of the technology on the performance of the proposed model. For example, as the technology improves we can incorporate more E-units and functional units on a wafer. Moreover, the reduction in the hardware cost could enable one to increase the number of the processing modules in the system.
The major criteria in our evaluation will be the number of instructions which are executed in a unit of time; i.e. MIPS - Millions of Instructions Per Second. This parameter is a function of the turn-around time (time required to complete a program). In addition to the MIPS, we would also measure the processing module utilization and the queue length in different system configurations.

The system is simulated as a single-program organization in which the application programs are processed sequentially. This is because we are more interested in studying the effects of dynamics of application programs and system configurations rather than measuring the pure MIPS performance of the machine. Naturally, in a multi-programming environment the idle processors can be assigned to other programs and the performance is expected to increase. Therefore, the MIPS measurements of the system should be higher than those presented in this study. The simulation of the system as a multi-programming organization is to be carried out as one of the future projects.
1. THE SIMULATION LANGUAGE

The WDDM emulator is implemented using the SLAM [113] simulation language. SLAM is a general purpose simulation language which allows simulation of network, discrete event, and continuous models. The network model can be used to set up a network of queues and servers in which entities, carrying different attributes, flow from one node to another. Furthermore, the model allows parallel servers with identical service capability. The network model is especially suitable for emulation of systems such as WDDM. This is because one can easily map the processors to servers, the queues to registers, and the packets to entities. The different attributes of an entity (packet) can be representative of opcode, operands, and destination addresses. Associated to each entity in the network is an array called (ATRIB) which holds the attributes of that entity. Thus, ATRIB(i) represents the ith attribute of an entity.
2. THE SIMULATION MODEL

The simulation project consists of two phases. First, we have simulated the behavior of an individual processing module. Then, the whole system, including the host and data structure modules, is simulated.

Phase 1) A Processing Module Simulation:

Figure 7.1 presents the operations of an E-unit within the simulator. The input to an E-unit is an entity (packet) whose attributes are as follows: ATRIB(2) represents the type of result packet (0 = operation, 1 = data), ATRIB(3) holds the number of operands, ATRIB(4) and ATRIB(5) contain the addresses of up to two destination addresses, and ATRIB(6) is the opcode.

The output of an E-unit is either an operation or a data packet which is routed to the sub-net. The sub-net is simulated exactly as the arbitration network represented in Figure 3.5. Both the arbitration and the square switches are replaced by a set of queues. Each queue has buffering capability and routes its incoming entities to the next
Figure 7.1: A model for the processing module operations.
level of the network. At the last stage, the entities are either sent to one of the E-units or one of the functional units according to the value of $\text{ATRIB}(2)$ (i.e. type of the packet).

The operation entities (packets) arrive to different functional units according to $\text{ATRIB}(6)$. Each functional unit is simulated via a queue and a server. The incoming tokens are buffered in the queue and processed by the server. The output of a functional unit is sent to the sub-net.

The behavior of a processing module can be simulated by setting the destination addresses ($\text{ATRIB}(4)$ and $\text{ATRIB}(5)$) and the number of operands for the succeeding instructions. The opcode (and thus type of the result token) is automatically generated by the simulator using a distribution which is explained in the next section.

Phase 2) System Simulation:

The primary observation collected from Phase 1 is the execution time of a program block. This information is used in simulating the system. Figure 7.2 depicts the operations of the system at the host and data structure modules' level. Two types of entities flow through the system, one representing a program block and the other representing a SIMD type data structure operation.
Figure 7.2(a): A model for the proposed system operations.
Figure 7.2(b): A model for the proposed system operations.
The input to the simulator is an entity representing the main program block. At the first step, the characteristics of a block are assigned to the attributes of the entity. These characteristics include the block execution time, obtained from Phase 1, and the program dynamics such as procedure calls, recursive calls, and data structure operations. The latter parameters are randomly set by the simulator.

Once a block completes its execution, it may enable up to n other blocks. The enables blocks are assigned to the enabled blocks queue and the process continues.

It should be noted that if a block contains a data structure operation, it resides in a processing module until the result of the operation is available. Such a scheme is employed in an attempt to reduce the traffic to/from the host and to decrease the overhead of the host operations.
3. DATA ACQUISITION AND ASSUMPTIONS

1) Assumptions:

In order to make the simulation manageable according to the resources (such as manpower and time), one must often define a framework for the simulator model. The boundaries of this framework are defined via certain assumptions and simplifications. The assumptions in this project are enumerated below:

i) The complexity and the time involved in the compilation of the programs are not considered. The program blocks are assumed to be in the object-code form and ready for execution.

ii) As discussed in Chapter IV, the size of the blocks is fixed and the compiler optimizes the size of a block to the maximum possible fixed block size.

iii) The simulator is used to study different system configurations. For example, the effect of increasing the number of processing modules is examined. Such changes in the system configuration also require some minor changes in the execution time of the host operations. For example, the
duration of the associative search of the free processing module list is directly related to the number of processing modules in the system; i.e., a search for a system of 48 processing modules is longer than that of a system of 8 processing modules.

2) Data Acquisition:

i) Execution times: The execution time of most operations were obtained from Chapter VI where the VLSI timing of these operations is discussed. These timings include the execution of simple operations within an E-unit, the delay of switching elements of the arbitration network, the associative search of the tables in the host, etc.

The execution time of the functional units is represented in Table 7.1. These figures represent the best execution time of these operations among VAX 11/780 [73], IBM 370 [124], and Sperry Univac 1100/80 [73].

The block allocation/deallocation is computed as the process of sequentially loading the instructions into the host module input port of a processing module and feeding its content to the sub-net in a pipelined fashion.

ii) Distributions: The distribution of arithmetic operations was obtained from the study discussed in [88]. This distribution is represented in Table 7.1. As shown in this table, about 55% of the operations are simple enough to
Table 7.1: The distribution and the execution time of arithmetic operations.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Distribution (%)</th>
<th>Execution Time (nsec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple</td>
<td></td>
<td></td>
</tr>
<tr>
<td>compare</td>
<td>13.1</td>
<td></td>
</tr>
<tr>
<td>shift/boolean</td>
<td>20.8</td>
<td>250</td>
</tr>
<tr>
<td>+/- fixed</td>
<td>21.0</td>
<td></td>
</tr>
<tr>
<td>complex</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+/- floating</td>
<td>24.0</td>
<td>850</td>
</tr>
<tr>
<td>* fixed</td>
<td>2.0</td>
<td>140</td>
</tr>
<tr>
<td>/ fixed</td>
<td>0.6</td>
<td>3700</td>
</tr>
<tr>
<td>* floating</td>
<td>13.1</td>
<td>1700</td>
</tr>
<tr>
<td>/ floating</td>
<td>5.4</td>
<td>4800</td>
</tr>
</tbody>
</table>
be carried out directly within an E-unit.

With regard to the degree of parallelism, a wide spectrum of applications with varied degrees of parallelism was run by the simulator. The obtained average execution times were then used in the simulation of the system.
4. STRATEGIC AND TACTICAL PLANNING

In order to insure the independence of the collected observations, the seeds to the random number generator were randomly changed for different runs. In addition, to collect more meaningful observations, each case study was run several times and the results were averaged. Thus, each MIPS, execution time, queue length, or processor utilization figure presented in the next section is the average of the results obtained from multiple runs of each case.

The strategy for collecting observations during Phase 1 was to: i) fix the number of E-units; ii) program the processing module for an application program; and iii) collect the execution time of that program, the average waiting time for the queue, and the maximum queue length. This process was repeated many times by changing the number of E-units and the degree of parallelism within a program. For example, the number of E-units were changed from 8 to 48 in multiples of 4. The case of less than 8 E-units was not considered since the block size would be too small. We discussed in Chapter VI that with the present technology it is possible to implant up to 32 E-units on a wafer.
Therefore, in addition to a 32-E-unit processing module, we also considered a 48-E-unit model as a representative of the future technology.

The strategy for Phase 2 was to collect MIPS and processing module utilization parameters for different system configurations and application programs. Since we were to simulate a single-program environment, the strategy was to begin the simulation with the main program and observe the system for a period of time. At the end of this period no new block is enabled, but the simulation would continue until all the remaining blocks in the enabled blocks queue are processed. When the simulation is over, the execution time of the program and the number of blocks and the number of data structure operations are collected. This information is then used in calculating the MIPS for the system. In addition to the above observations, the average number of processing modules utilized during the execution is also collected.

The simulation was run for different system configurations ranging from 4 to 256 processing modules (changing in powers 2). For each configuration the effects of data structure operations and the block size are also studied.
5. THE SIMULATION RESULTS

1) Phase 1:

The purpose of this phase of the project is three-fold. First, the effect of parallelism in a program block is studied. Second, the average waiting time in the queue and the maximum queue length information are collected. Third, the average execution times for blocks of different sizes are collected and used in the next phase.

Figure 7.3 plots the execution time of a block of size eight versus the number of computational steps involved in completing the block. It is apparent that the higher the degree of parallelism or the less the number of steps is, the lower the execution time becomes.

The average waiting time in the queue and the maximum queue length are plotted against different block sizes in Figures 7.4 and 7.5, respectively. In order to show the sensitivity of the system to the type of operations, the experiment was conducted for both the distribution of Table 7.1 and a uniform distribution. Both the queue length and its waiting time increase sharply as the block size is
Figure 7.3: The effect of parallelism in a program.
Figure 7.4: Average waiting time in the queue.
MAXIMUM QUEUE LENGTH

Type 1 -> Kuck's Distribution
Type 2 -> Uniform Distribution

Figure 7.5: Maximum queue length.
incremented. The maximum queue length ranges from $1/3$ to $1/2$ of the number of E-units in a processing module.

Figure 7.6 depicts a plot of the average execution time of a block against the block size for a uniform distribution as well as the distribution of Table 7.1 for the type of operations. The case of uniform distribution resulted in a longer execution time since the number of complex operations are increased. For each block size, a spectrum of degrees of parallelism is simulated and the average of these are presented here. This plot shows that the average execution time is increased as the block size is increased, but at a much lower rate. This is due to the expectation that, on the average, the number of parallel instructions should increase as the block size increases. However, at the same time the average number of steps to complete the program should also increase.

2) Phase 2:

One of the first problems that we needed to resolve was the incorporation of parallelism among the program blocks. For that purpose, we assumed that each block may randomly enable from 1 to $N$ other blocks at the end of its execution. Figure 7.7 shows the MIPS performance of a 64 processing module (each 8 E-units) system against different values of $N$. Obviously, the performance is improved as the value of $N$
Figure 7.6: Average block execution time.
Figure 7.7: Effect of parallelism among the blocks.
is increased. However, the improvement is not proportional to the increase in N since the overhead of the host computer is also increased. For this particular project we assume that the value of N is 5; i.e. on termination of a block, up to 5 blocks may be enabled.

In order to study the effect of SIMD type data structure operations, we included such operations in one experiment and removed them in another experiment. In either case, procedure calls and reference to individual data structure elements are allowed.

The MIPS performance of the system for different system configurations and different number of E-units (block sizes) is presented in Figure 7.8. In this experiment, SIMD type data structure operations are not included. It is noticeable that the performance increases as the block size (or number of E-units) is increased. However, for a particular block size the performance saturates at about 128 processing modules. This is because we are simulating a single-program environment, and a single process on the average requires a limited number of resources. This can be verified by the processing modules utilization which is depicted in Figure 7.9. Naturally, in a multiprogramming environment the utilization and the performance are expected to increase.

Similar experiments were carried out while allowing SIMD type data structure operations. These results are
NUMBER OF E-UNITS IS VARIABLE
DATA STRUCTURE ACCESS EXCLUDED

Figure 7.8: The MIPS performance of the system.
NUMBER OF E-UNITS = 32
DATA STRUCTURE ACCESS EXCLUDED

Figure 7.9: The average processing module utilization.
shown in Figures 7.10 and 7.11. A comparison of these experiments reveals that the inclusion of SIMD type data structure operations causes a reduction in both the MIPS performance and the processing modules utilization. Figure 7.12 depicts the MIPS performance and Figure 7.13 shows the processing utilization in the two experiments.

The main reason for this difference in performance and utilization is that when a block initiates a SIMD data structure operation, it remains in the processing module until the result of the operation is returned. The alternative is to reassign the processing module to another block while the data structure operation is carried out. In that case, the overhead of the host is increased due to the swapping process and keeping track of the association between the blocks and the data structure operations.

An interesting observation in Figure 7.10 is that there is a tighter gap in performance for lower numbers of processing modules than for higher numbers of these units. When the number of processing modules is increased, the system can still allocate the blocks to the processing elements, even though a subset of them are unproductively remaining occupied.

In order to draw conclusive results from the figures presented in the previous plots, one must perform a test of hypothesis for each pair of the results [45,113]. Such a study is out of the scope of this manuscript. However, a
Figure 7.10: The MIPS performance of the system.
NUMBER OF E-UNITS = 32
DATA STRUCTURE ACCESS INCLUDED

Figure 7.11: The average processing module utilization.
NUMBER OF E-UNITS = 32

TYPE = 1 --> DATA STRUCTURE ACCESS EXCLUDED
TYPE = 2 --> DATA STRUCTURE ACCESS INCLUDED

Figure 7.12: The effect of data structure operations on MIPS measurement.
Figure 7.13: The effect of data structure operations on average processing module utilization.
sample is presented below:

Consider Figure 7.8, when the block size is 32, the performance has dropped as the number of processing modules is increased from 128 to 256. Let $A = 58.78$ be the average MIPS for the system of 128 modules and $B = 56.77$ be the average MIPS for the 256-module system as obtained from sample data. In this case $\mu_A$ and $\mu_B$ will represent the average MIPS obtainable from the populations. We assume that the standard deviation of the population MIPS are unknown and different (i.e. $\sigma_A \neq \sigma_B$). We will perform a one sided test of hypothesis at a 5% significance level. The null hypothesis is that $\mu_A$ and $\mu_B$ are equal ($H_0 : \mu_B = \mu_A$). The alternate hypothesis is that $\mu_B$ is less than $\mu_A$ ($H_1 : \mu_B < \mu_A$). Our objective is to determine whether or not we can reject the null hypothesis.

The decision criterion is established by constructing a test statistic whose distribution is "Student's t" [45,113]. This test statistic is calculated from the sample data as follows:

$$t = \frac{A - B}{\sqrt{\frac{S_A^2}{N} + \frac{S_B^2}{N}}}$$

where, $S_A^2 = \frac{S_A^2}{N}$ and $S_B^2 = \frac{S_B^2}{N}$. $S_A^2$ and $S_B^2$ are the variances of sample data collected and $N$ is the sample size. Plugging in the values of these parameters, we have $S_A^2 = 59.01$, $S_B^2 = 46.52$, and $t = -.20$.

In order to find the corresponding percentile of the Student's t distribution, we need to calculate the degrees
of freedom:

\[ df = \frac{\left( \frac{S_A^2 + S_B^2}{N+1} \right)^2}{\frac{3^4}{N+1} + \frac{S_g^4}{N+1}} - 2 \]

Replacing the values of the parameters, we have df = 12 and \( t_{5\%, 12} = 1.782 \).

The null hypothesis rejection rule is \( t < -t_{5\%, 12} \). Since -.20 < -1.782, we fail to reject the null hypothesis. Therefore, with a 5% significance level, there is not sufficient evidence to reject the null hypothesis (\( \mu_A = \mu_B \)).
6. VERIFICATION

Three steps are used in order to verify the correctness of this simulation program. These steps are as follows:

i) SLAM provides a MONITOR/TRACE facility which is used by the programmers for debugging purposes and to verify the correctness of the operations. This feature dumps all the attributes of each entity after each operation as the entities pass through the network. Thus, one can monitor every operation and entity during the course of the simulation. The monitor feature was turned on for this simulator and the results were exactly as expected; i.e. the simulator was behaving the way it was supposed to.

ii) The obtained results are logically acceptable in the sense that they are correlated with what one would have logically expected them to be. In other words, there was no evidence of gross miscalculations in the results.

iii) Finally, we tested the simulator with a set of test data for which we had manually calculated the results. The test data were chosen from the two extreme cases and an average case. The simulation results matched the calculated results. For example, consider the operations of a
processing module with 8 E-units. Let's assume that the application program is completely sequential with six instructions being simple operations, one instruction being a fixed point multiplication, and a return statement. Each simple operation requires 260 nsec to complete and form the result packet. The network delay (including the sub-net, the common bus, and the match of the token addresses by the E-units) is 320 nsec. The multiplication requires two packet formation delays, two network delays, and one operation delay, totaling 800 nsec. Finally, the return statement requires one packet formation delay and a network delay for 330 nsec. Therefore, the total execution time is 4610 nsec which was the same as the result produced by the simulator.
7. SUMMARY

The performance of the proposed system was evaluated in this chapter. Since the architecture is general purpose, its performance depends on the type of the application programs. For this reason, the system behavior was simulated so that a wide spectrum of applications can be tested on the simulator.

The following conclusions can be drawn from the simulation results:

1) The system performance is improved as the degree of parallelism in the application programs is increased.

2) The system performance is improved as the number of processing modules is increased. However, for a single-process operation environment the performance saturates at some point.

3) The processing utilization is decreased as the number of processing modules is increased.

4) The system performs better for applications which do not include the SIMD type data structure operations.

For block sizes in excess of 24 and processing modules in excess of 32, the system performance is superior to some
of the existing data flow machines [102,133]. However, this comparison is meaningful only if the same simulation parameters and application programs are used for each system. Such a study is to be undertaken in one of the future projects.
CHAPTER VIII

CONCLUSION AND FURTHER DEVELOPMENTS

In this manuscript we have introduced a new multi-processor system based on the principles of the data and block driven computations. Chapter I discussed the motivations behind this project and outlined the dissertation objectives. Chapter II introduced a new classification for the data driven computers while indicating the trend in the design of the future architectures of this kind. The overall organization and the structure of each system unit was presented in Chapter III. Chapter IV was a discussion of the memory management issues such as memory hierarchy organization, processor allocation, procedure call/return mechanism, etc. The manipulation of data structures was covered in Chapter V. The VLSI design of the system units and their timing/geometry evaluation was discussed in Chapter VI. Finally, Chapter VII presented the system performance evaluation through the simulation of the system behavior.

The proposed architecture has achieved three major
milestones:

i) Elimination of some of the shortcomings of the existing data driven systems: WDDM distributes the computational power over the storage cells. This scheme eliminates the memory access overhead and the network delays between the memory and the processing elements during the execution. Furthermore, it introduces a more regular organization consisting of basic storage cells as the building block elements.

In addition to the distribution of the processing power, the proposed architecture closes the semantic gap between the data driven applications and the underlying architecture. This has been accomplished by providing communication paths of different lengths for inter- and intra-block communications in accordance with the data driven semantics.

ii) Ease of implementation by current technology: In order to comply with the VLSI technology constraints, the system has a highly regular and modular organization. The utilization of Wafer-Scale Integration improves intra-block communication and eliminates the VLSI I/O pin limitation problem.

iii) Achieving parallelism at the program block level, instruction level, and data level: The proposed system utilizes a collection of processing modules to execute multiple program blocks in parallel. The instructions
within each block can be carried out in parallel in a data driven fashion. In addition, the processing modules can be used for implementation of the SIMD type data structure operations.

The simulation results have indicated that the performance of the system improves either by increasing the number of E-units on the processing modules or by increasing the number of processing modules in the system. These conditions (increasing the number of processors) are in accordance with the current trends of the technology. However, for a uni-program environment, the performance saturates at some point due to the fact that only one application program is carried out at a time. Therefore, the processing module utilization decreases by increasing the number of processors. One can expect better utilization and performance in a multiprogramming environment.
FURTHER DEVELOPMENTS

This project can be further pursued in many directions ranging from the actual implementation of the proposed architecture to developing a software system for it. In the following we discuss some directions which one can take to further develop this project. However, it should be noted that by no means is this an exhaustive enumeration of such possibilities.

1) Expansion and modification of WDDM: One can evaluate many variations of the proposed system. For example, if we replace the E-units and the functional units by a set of microprocessors, then the extra pass through the sub-net is eliminated for the complex operations. However, the execution time of all the operations is increased [76].

Another interesting variation is to replace the sub-net, which is an arbitration network, by an NXN packet switching network. In this case the bottle-neck of the queue and the common bus are eliminated. However, the traffic through the network is increased, the tokens must be queued at the E-units, and problems such as the cross-over communication lines must be resolved.

286
The simulation can be expanded for a multiprogramming environment. In that case, one expects a better processing module utilization and hence, a better system performance. However, the overhead on the host module in such an environment should be investigated.

2) Implementation of a prototype system: The VLSI design of the system components along with their timing and geometry analysis is already presented. Thus, one can continue the project by designing suitable algorithms for other complex operations. This can lead to fabrication and testing of the required chips in order to build a prototype model.

3) Comparison of the system performance: One can compare the performance of the proposed machine against some of the existing data driven architecture. Naturally, the same set of parameters and application programs should be used in the simulation of each system.

4) Design of the operating system: In this project, we discussed some of the operating system issues such as memory management, processor allocation, procedure call/return, data structure operations, etc. Such discussion can be further developed for the complete design of an operating system capable of handling a multi-programming organization in a data driven environment.

5) Programming language issues: The implementation of one of the existing data driven languages on WDDM should be
considered. The design and development of a new suitable language for WDDM could also be investigated. In addition, the compiler role (e.g. decomposition of large blocks into smaller fixed size blocks) and its implementation in our environment require further research.

6) Interrupts and I/O operations: As part of the implementation of a prototype system, one must also address the issues of I/O operations and program and system interrupts.

7) Non-numeric operations: Data driven architectures are most suitable for numeric applications requiring a large degree of number crunching. An interesting project will be the application of the data flow principles to non-numeric operations such as text retrieval and data base applications.
BIBLIOGRAPHY


