CANADIAN THESES ON MICROFICHE

I.S.B.N.

THESES CANADIENNES SUR MICROFICHE

National Library of Canada
Collections Development Branch

Canadian Theses on
Microfiche Service

Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada
Direction du développement des collections

Service des thèses canadiennes
sur microfiche

NOTICE

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

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

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

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

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

AVIS

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

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

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

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

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

NL-339 (r. 82/08)

Canada

A Distributed High-Level Language System

by

Guy R. Mousseau

A thesis
presented to University of Ottawa
in partial fulfillment of the
requirements for the degree of
Master of Applied Science
in
Electrical Engineering

I hereby declare that I am the sole author of this thesis.

I authorize University of Ottawa to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Guy R. Mousseau

I further authorize University of Ottawa to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Guy R. Mousseau

University of Ottawa requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

ABSTRACT

To efficiently design an up to date general purpose computer, one must take into account the recent software and hardware developments. In this thesis, the various techniques devised over the years to achieve high performance are analysed, in order to propose an efficient general purpose computer.

The suggested architecture is a distributed system that can support concurrent operation at two levels: at the task and at the process levels. Furthermore, to reduce the amount of support software and to efficiently execute procedure oriented language programs, the machine directly supports high-level data and control structures. For this reason, a relatively high-level execution language was defined.

In order to prove the feasibility of this system, two of its critical elements were simulated; the translation unit and the program execution unit. The first simulated the translation from a general high-level language into the proposed execution language, while the second examined the execution of a program in this language.

ACKNOWLEDGEMENTS

I am most grateful to my supervisor, professor Moshe Krieger, for his meaningful advice and guidance during the course of my research. The many helpful discussions confirmed the soundness of the ideas reflected in this thesis.

I would also like to thank my family, friends, and colleagues who made this research more enjoyable. Without them I would not have found the necessary drive.

Furthermore, I wish to acknowledge the financial support of the National Science and Engineering Research Council.

CONTENTS

ABSTRACT iv

ACKNOWLEDGEMENTS v

Chapter page

I. INTRODUCTION 1

II. OVERCOMING THE LIMITATION OF THE VON NEUMANN COMPUTER 7

Introduction 7
Reducing the memory bottleneck 8
Main memory organization 9
Interleaved memory or banking 9
Pipeline memory 10
Intermediate storage provision 11
Cache memory 11
Scratch pad memory 12
Central processor modification 13
Multiaddressing 13
Addition of registers 14
Instruction prefetch 14
Conditional processing (instruction lookahead) 15
Providing concurrent operation 15
Concurrent operation at the instruction level 16
Pipelined architecture 17
Array Processors 21
Associative processors 25
Concurrent operation at the processor level . 28
Von Neumann type 29
Functional language computer 32
Concurrent operation at the task level . . . 39
Computer networks 40
Distributed system 40
Reducing the software overhead 43
Special hardware units 43
Increasing the level of the intermediate language 45
Modifying the data and control structure of the machine 46
Indirect execution computer 47
Direct execution computer 51
Reliability improvements 52

Conclusion 56

III. A DISTRIBUTED COMPUTER SYSTEM 58

Introduction 58
System architecture 60
Virtual language 63
Elements of the virtual language 64
Data entities 66
Operators and expressions 68
Control 69
Language definition 70
Example of a virtual language program 73
System organization 75
Task driven system 75
A virtual language system 77
The elements of the distributed system . . . 79
System operation 85
Minimum system 86
Expanded system 87
Conclusion 88

IV. IMPLEMENTATION DETAILS 90

Introduction 90
Charateristics of the execution language 92
Elements of the execution language 94
Control segment 94
Parameter section 95
Sequence 96
Decision 97
Iteration 98
The control string 99
Control words 101
Statement segment 106
Statement descriptor table 106
Statement section 107
Data segment 110
Data descriptor table 110
Data space 111
Example of an execution language program . 112
Processing of a an execution language process 115
Variable accessing 116
Variable accessing in the main program . 116
Variable accessing in a called procedure 118
Multi-level accessing scheme 121
Process execution 123
Module 0 125
Module 1 125
Module 2 126
Module 3 127
Module 4 128
Module interaction 129
Hardware requirements 133

Conclusion 141

V. PREPROCESSING 143

Introduction 143
Conventional translation 144
Preprocessing steps 147
Step 1: Declare segment preprocessing . . . 149
Step 2: Compute segment preprocessing . . . 151
Conclusion 156

VI. CONCLUSION, ENHANCEMENT AND FUTURE RESEARCH . . . 158

Conclusion 158
Immediate enhancements 163
Future research 167

Appendix page

A. VIRTUAL LANGUAGE SYNTAX 170

B. THE EXECUTION LANGUAGE 179

C. PREPROCESSOR SIMULATION 185

D. TASKMANAGER SIMULATION 207

E. SIMULATION EXAMPLES 225

REFERENCES 243

LIST OF FIGURES

Figure		page
1.	Von Neumann architecture	2
2.	Interleaved memory	9
3.	Pipeline memory	10
4.	Cache memory	12
5.	Scratch pad memory	13
6.	Segmented instruction execution	18
7.	Array processor	23
8.	Associative array processors	27
9.	Word-serial associative processor	28
10.	Tightly coupled multiple processor system	30
11.	Control flow program	33
12.	Data flow graph	36
13.	Data flow computer	37
14.	Reduction of an expression	39
15.	Compiler and interpreter interface	45
16.	Cost decision pyramid	59
17.	Block-structured language tree representation	65
18.	Scope of a variable	66
19.	Flow diagram block	69
20.	Basic structured control constructs	70
21.	Task driven system	76

22. Task dependence 77
23. Task/process relation 78
24. A virtual language system 79
25. Proposed system 81
26. Sharing task executor 83
27. A distributed general purpose computer system . . . 85
28. Virtual language processes execution 91
29. Map of an execution language program 93
30. Detailled map of an execution language program . . 112
31. Main program's sections 116
32. Direct access of variables 117
33. Main program and procedure representation 119
34. Indirect access of variables 121
35. Data space 121
36. Required pointers 125
37. Execution module interaction 130
38. Minimal system 133
39. The control string unit 134
40. Handshake and allocation unit 136
41. Process management unit 137
42. Task executor 140
43. Compilation steps 145
44. Preprocessing function 148
45. Infix to polish transformation 156
46. Flow of variables 172

Chapter I

INTRODUCTION

In a 1946 paper entitled "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument", Burk, Goldstine, and von Neumann outlined the design of an instrument to solve non-linear differential equations (BURK 46). Over the years this architecture has been successfully applied to solve problems in other fields, and is now the basis of most present day computing machines. This architecture has been rightfully called the von Neumann architecture. The traditional von Neumann machine can be decomposed functionally in the following four units: a memory unit to store programs and data, an arithmetic and logic unit (A.L.U.) to manipulate the data according to the operations specified by the program, an input/output unit to communicate with the external world, and a control unit that coordinates the actions of the complete system. Usually the control unit and arithmetic unit are grouped together to form the central processing unit (C.P.U.). The information paths between the units are shown in figure 1 .

In addition, two attributes are implied in the von Neumann architecture. The first attribute is the inclusion of decision making instructions; this gives the capability of

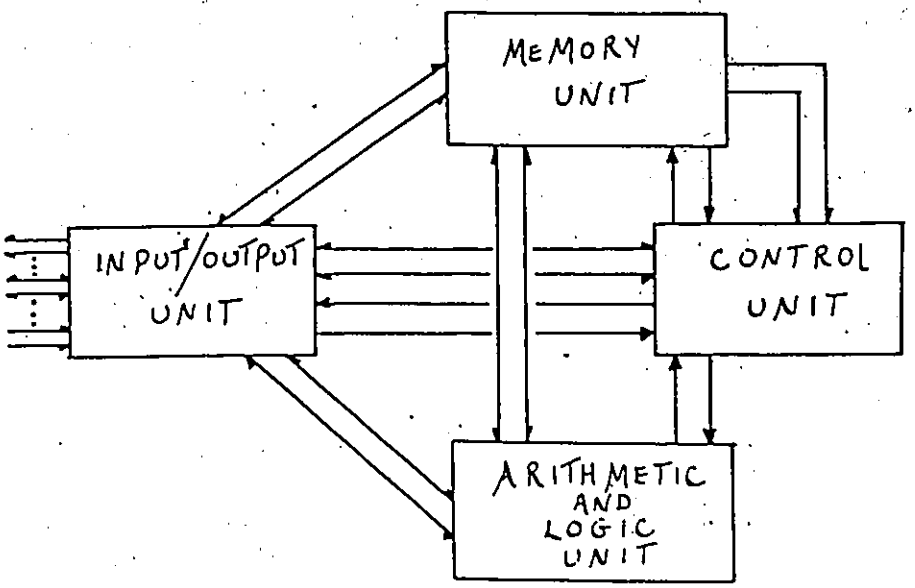


Figure 1: Von Neumann architecture

modifying the sequence of execution of the instructions. The second is the capability to store programs internally in the memory. This implies that instructions can be manipulated like data.

One of the main features of the von Neumann architecture is its simplicity, which was required by the limitation of the technology at that time. It is interresting to note that this simplicity led to the microprocessors and microcomputers of today, which are LSI implementation of the von Neumann architecture. The traditional von Neumann architecture has however a number of problems that affect its overall performance. The following four problems are considered by most as its major limitations:

1. Memory Bottleneck. The traditional von Neumann architecture requires very large number of information exchanges between the central processing unit and the memory. This problem in most systems is further aggravated by the incompatible speed of these two units.

2. Sequential Operation. In its most basic form the von Neumann machine does not support concurrent operations at any level. Instructions are fetched in a sequential manner and executed one at a time.

3. Large Software Overhead. Generally there is very little relation between the machine language and the high-level languages; both with respect to expressions and control operations. In addition the traditional von Neumann machine does not support directly data structures of any complexity as it only manipulates data words.

4. Lack of Reliability. The rapidly changing technology has permitted the incorporation of small, low cost computers in a number of key positions in many systems. The failure of such computers would result, in some cases, in losses of large amounts of capital or in loss of human lives. In these systems, reliability is more important than computing power, therefore the computing elements must be reliable, a characteristic that is lacking in the von Neumann architecture.

The problems of the von Neumann computer arose from the fact that at that time, hardware was very expensive and very unreliable. It is only due to the genius of von Neumann that such architecture was designed and success achieved with so little hardware.

Many techniques have been considered over the years to reduce the effects of the above problems, this is also what we attempt to do here. The major objective of this thesis is to take advantage of both the recent hardware and software advances, in order to propose a highly reliable, high performance, efficient, and easy to use computer. Roughly speaking, reliable operation can be achieved by using a modular design strategy, while high performance can be obtained by providing concurrent exploitation of the system resources. Efficient execution and ease of use can be achieved by supporting some of the functions of present day procedure oriented languages directly in hardware.

In order to provide the basic background for the discussion of an up to date computer architecture, chapter two of this thesis will analyse the work done to the present day to reduce the previously mentioned limitations associated with the von Neumann architecture. This survey classifies the various techniques developed, according to the main problem that they attempt to solve. Although equal details are given for each of the four sections, it should be noted that

the sections on concurrent operation, and on reducing the software overhead contain material more relevant to the developments in this thesis.

In chapter three, a distributed high-level language computer system is developed that is based on the analysis of the techniques described in chapter two. This distributed system was designed to accomodate a virtual language which has the characteristics of present high-level procedure oriented languages. The operation of the complete system is examined with respect to the execution of a virtual language program.

Chapter four examines the actual language that is directly supported by the system. This language is called the execution language, which is a modified form of the virtual language, and was developed in order to obtain a more efficient execution. A detailed description of the processing of an execution language program is also given. To check the feasibility of the execution steps, a simulation program was written in Pascal. In the final section, the hardware organization of the main units of this distributed system is described.

In chapter five, the preprocessing of a virtual language program into an execution language program is analysed. Since the virtual language is related to present day high-level languages, this preprocessing will more or less cor-

respond to what will actually be required in a working system. The algorithm used to perform the preprocessing is described, and a simulation was performed in order to test the feasibility of this algorithm.

Chapter six examines various enhancement to the system, along with a description of future research that must be performed in order to make the system practical.

The present work also includes five appendices that describe the details of the languages used and simulation programs developed. Appendix A defines the syntax of the virtual language, while appendix B presents the characteristics of the execution language. In appendix C, the simulation performed to analyse the preprocessing steps is described in detail, while appendix D analyses the simulation developed to test the processing of an execution language program. In appendix E a few examples of these two simulations are presented.

Chapter II

OVERCOMING THE LIMITATION OF THE VON NEUMANN COMPUTER

2.1 INTRODUCTION

In the introductory chapter of this thesis, the von Neumann architecture was briefly described. Although it is the basis of most present day computing system, it was shown to be hampered with four major problems. These problems are: a memory bottleneck, sequential operation, large software overhead, and lack of reliability.

In this chapter the various techniques suggested to minimize these problems are described and classified according to the main problem that they attempt to solve. This classification scheme was chosen, since most techniques could be easily incorporated into one of these classes.

This chapter describes the most important techniques in each class, in order to provide the proper background needed for the development of a general purpose high performance computer. With respect to the actual work developed in this thesis, one could consider only the new design philosophies, which consists of the various multiple processor system and the system that match software and hardware. These are de-

scribed in the section considering concurrent operation and reducing the software overhead.

2.2 REDUCING THE MEMORY BOTTLENECK

In a von Neumann machine, to execute an instruction generally involves a number of sequential information transfers between the central processing unit and the memory. For example the execution of a double operand instruction, may require the following transfers:

1. The instruction's address is sent from the C.P.U. to the memory.
2. This address is used to locate the instruction, which is then sent from the memory to the C.P.U.
3. When this instruction is decoded, four more transfers may be required to fetch the operands; two addresses, and two data words.
4. Finally to store the result, it may require other transfers; its address, and its value.

Furthermore, more transfers are required if the access is not direct. Therefore the execution of any meaningfull program requires a large number of single word transfers, which, most of the time are not meaningfull data, but addresses of the data. In this section, the various techniques used to minimize the effect of large information tranfers are presented. These techniques can be grouped in the following three categories:

1. Organizing the memory to allow multiple access.
2. Providing high-speed intermediate storage between processor and main memory.
3. Modifying the central processor unit for fast access.

2.2.1 Main memory organization

To mask the speed gap that exists between large main memories and the central processing units, the memories can be reorganized to allow multiple accesses. There are two main organizations.

2.2.1.1 Interleaved memory or banking

An interleaved memory is divided into n banks, where word i of a particular sequence of data or instructions is stored in bank 1, word i+1 in bank 2, and so on, as shown in the following figure.

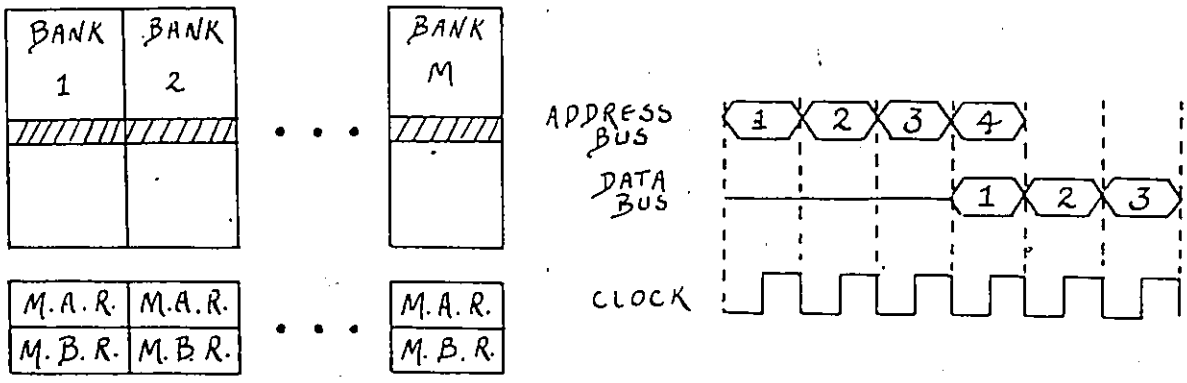


Figure 2: Interleaved memory

This type of memory does not actually reduce the access time but it speeds up the access of a sequence of words. To retrieve a sequence of word, the consecutive addresses are sent sequentially in intervals of T/k, where T is the access time of the memory and k is the number of interleaved banks. The value of k is chosen with respect to the speed of the memory and the processor. The first word is received after time T and the successive words after time T/k, therefore a higher frequency of memory access is possible. If only one word is required from this memory then the access time is the same as for a conventional memory.

2.2.1.2 Pipeline memory

Pipeline memories are similar to interleaved memories except that only one address is supplied to obtain a stream of words, as shown in figure 3 .

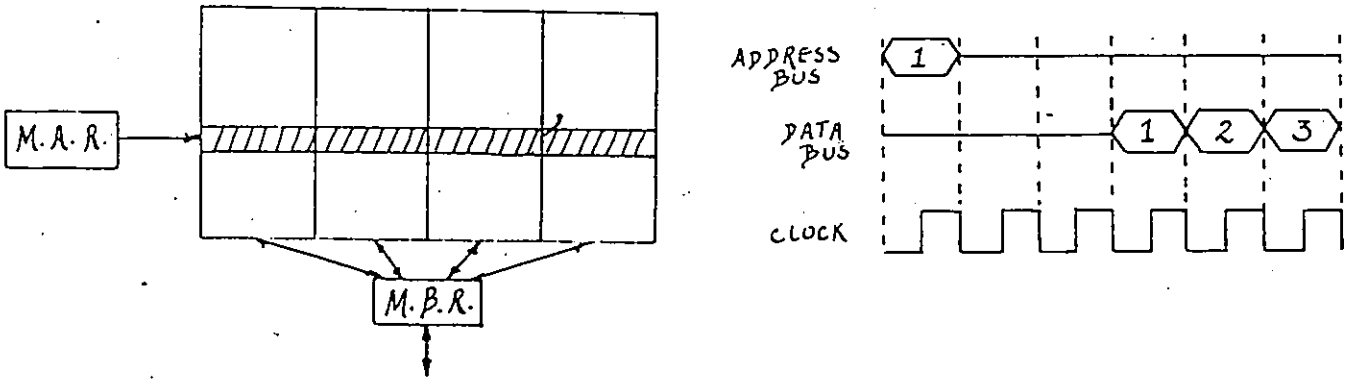


Figure 3: Pipeline memory

Pipeline memories are mostly used in pipeline processor where a string of words (data or instructions) must be fed in a rapid fashion into the pipe.

2.2.2 Intermediate storage provision

An alternate way to reduce the speed mismatch between the central processing unit and the main memory, is by providing small high speed buffer memory between the slower main memory and the central processing unit.

2.2.2.1 Cache memory

The cache memory is a high speed small memory that is used to mask the access time of the main memory (fig. 4). To save memory access time the central processing unit accesses the cache memory instead of the main memory. The problems with using a cache is the amount of transfer required between the main memory and the cache, and the choice of the strategy that ensures that most references are made to the cache memory. The choice of such strategy is not straightforward, since they are highly application dependent. The size of the cache is also a critical factor, because it determines the amount of data that must be transfered.

In most types of cache only the recently accessed data or instructions are stored. Those that have not been recently

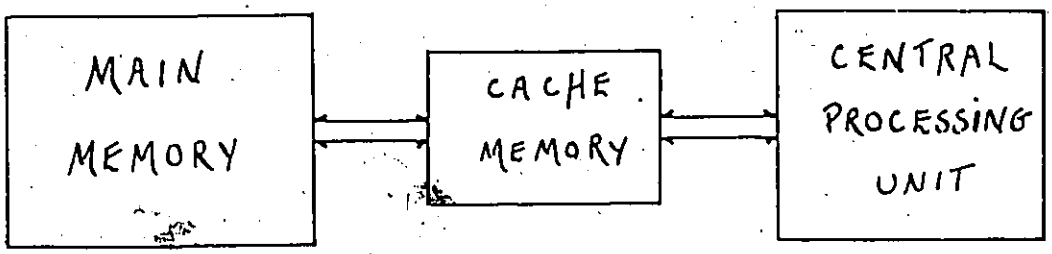


Figure 4: Cache memory

accessed are replaced automatically. A particular example of a cache memory is the associative cache memory. In an associative cache memory the main memory addresses are stored as associative tags to each block of data words so that references to the cache examines these tags to see if the desired information resides there. If the data is not in the cache, then it must be obtained from the main memory.

2.2.2.2 Scratch pad memory

A scratch pad memory is used as a work space to store intermediate result of a computation (fig. 5). It therefore reduces the number of main memory references. The actual control of a scratch pad memory is much simpler than for a cache memory since it only stores data. However there still exists a problem of data transfer between the buffer and main memory.

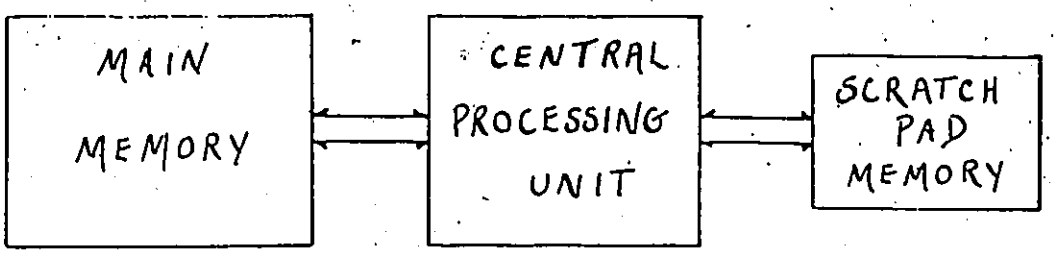


Figure 5: Scratch pad memory

2.2.3 Central processor modification

The previous techniques dealt with the masking of the access time of the main memory in order to reduce the memory bottleneck. An alternative to this, is to modify the central processor to reduce the number of accesses, or to access the memory while the processor is performing other operations.

2.2.3.1 Multiaddressing

By using a large word length, an instruction can specify more than one address, thus reducing the number of fetches in a program. This is practical only if one has to access very small memories. However, using various addressing modes one could reduce the size of the instructions. Alternatively one can have multiword instructions which are fetched automatically in a pipeline fashion.

2.2.3.2 Addition of registers

The addition of general purpose registers can reduce the number of memory references if these registers are used to store information that is frequently referenced by a program segment. By this technique, one can achieve multiaddressing with a reasonable word size. It offers similar advantages and problems as obtained using a scratch pad memory. A side effect of using multi-registers is the increase in the size of the instruction set, since in order to have a symmetric instruction set, one has to be able to address all these registers equally for them to be effective. Another way of adding registers, is using a stack for the operands. In this case, since only the top of the stack is manipulated, addresses are not required to specify the location of the operands. These stacks also find applications in executing high-level languages. For these reasons, new processor architectures support stack directly, either by using special stack pointers, or special addressing modes (i. e. auto-increment, auto-decrement).

2.2.3.3 Instruction prefetch

In instruction prefetch, instructions are fetched anytime the memory bus is idle, therefore instructions to be executed are placed in a high speed FIFO buffer. This technique is now included in some microprocessors to reduce the memory

bottleneck (TOON 81). Furthermore by including additional logic, short loops can be captured and executed at register speed, if the high speed buffer is large enough. Branching outside the range of instructions in the high speed buffer will destroy the advantages associated with this approach. Instruction pre-fetch can also be performed in a block fashion. In this case the processor is halted to fetch a block of instructions, which are placed in a high speed buffer.

2.2.3.4 Conditional processing (instruction lookahead)

Conditional processing is similar to instruction prefetch described above, except that when a conditional branch instruction is encountered, the two possible paths are fetched concurrently until the branch condition is obtained. At this point one of the path is abandoned and execution continues thereon.

2.3 PROVIDING CONCURRENT OPERATION

In its simplest form the von Neumann architecture does not support any type of concurrent operation. Many enhancements and new architectures were developed to remove this drawback by providing concurrent operation at specific levels. To provide a systematic survey, the work done in this area will be considered according to the following three classes:

1. Concurrent operation at the instruction level.
2. Concurrent operation at the processor level.
3. Concurrent operation at the task level.

Note that these levels were not introduced to provide an exact classification, but only to indicate the level of coordination required.

In the following discussion, in order to make it more precise, the terminology introduced by Flynn will be used (FLYNN 66). This terminology is based upon a classification that checks the multiplicity of the hardware to service the instructions and data streams. Using this terminology, the traditional von Neumann machine is a single instruction, single data stream processor, which represents most computing instruments of today.

2.3.1 Concurrent operation at the instruction level

Concurrent operation at the instruction level is generally achieved by processing multiple data streams simultaneously. In a sense these processors allow direct manipulation of more complex data structures. This type of concurrency will be presented under three general headings.

2.3.1.1 Pipelined architecture

In pipeline architectures, the instructions are divided into parts so that either the parts of different instructions, or parts of the same instruction can be executed concurrently. Pipelining is considered in the following three categories: overlapped operation, pipeline processor, and multi ALU system.

a) Overlapped operation

Overlapped operation refers to pipelining at the microinstruction level. At this level, instructions are considered as computational processes that can be segmented into subprocesses. The simplest form consists of overlapping the fetch and execute phase of sequential instructions. In this case while an instruction is executed the next one is fetched and decoded. This is similar to instruction prefetch. Further overlapping can be obtained by segmenting the execution phase. For example, to execute an arithmetic instruction, the required operands must be fetched, which corresponds to the pre-op phase. The op-phase consists of performing the proper operation on the operands. Finally the result is stored, corresponding to the post-op phase. Depending on the instruction one or more of these phases might be missing. The execution of instructions can therefore be segmented as shown in figure 6, where the phases of independent instructions are executed concurrently.

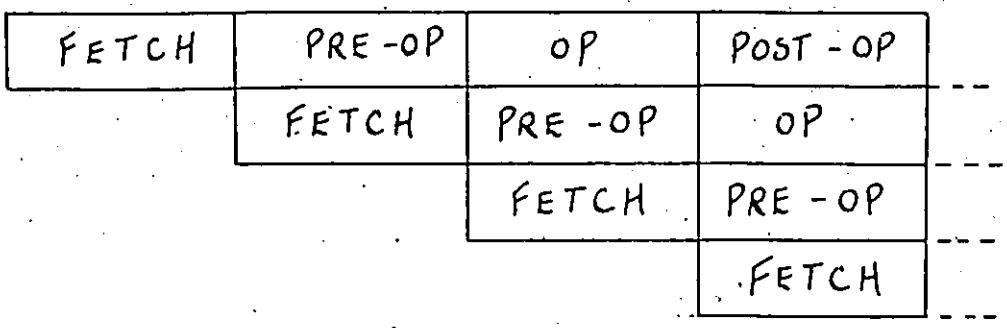


Figure 6: Segmented instruction execution

There exist a number of ways that one can control the concurrency in the execution of the phase of such instructions. One way is to use a separate unit for the control and execution of each phase and to provide a central hardware sequencer. The overall control unit can be segmented so that each unit has its own microprogram controller. This makes the system much simpler and leads to a compaction of microcode (VAIL 80). Furthermore, the units of such a system can be designed, tested, and upgraded separately. Pipelining at this level is usually still considered as a single instruction, single data stream processor, since there is only one CPU and instructions are executed sequentially.

b) Pipeline processor

Pipeline processors, also known as vector processors, refers to pipelining at the computational level. A pipeline computer consists of multiple processing units each responsible for partial processing of the data streams. This involves the segmentation of an operation into independent su-

boperations, so that each unit is independent of the other and performs a specific task on the data stream. These tasks are specified by the instruction being executed. Actually to obtain the required decomposition of the operation, the execution time is generally increased over the nonpipelined processors. Because of this, these processors are not efficient for executing a sequence of different operations on simple data entities.

The throughput of these processors is maximized, if one executes the same operation on a stream of data; this is the case when processing vectors. For vector processing, each element of the vector has to go through a transformation that is independent of the other elements of the vector. The execution of a vector instruction requires: that the instruction be fetched and decoded to set-up the necessary control paths and the function of the pipeline elements, and that the source data be fed to the pipe in an appropriate fashion. This data is usually stored in a pipeline memory for fast access.

The inclusion of these processors in one of Flynn's classes depends upon how one defines the instruction and data stream. Usually they are considered as MISD, that is multiple instruction operating on a single data stream (THUR 75). However, they can be classified as SIMD or even MIMD depending upon the specification of the data and instruction streams (BAER 76).

A pipeline processor can be organized using two types of pipelined schemes (RAMA 77).

i) Unifunctional pipeline

A unifunctional pipeline performs a single dedicated function; this is the simplest form since a minimum of control is required. However, the implementation of a complete processor requires that all functions to be pipelined be implemented separately.

ii) Multifunctional pipeline

A multifunctional pipeline can execute a number of different functions as selected through a number of control lines. These types of pipeline module increase the flexibility and reliability of the system, but they also increase the number of control actions required. A multifunctional pipe can be controlled using two strategies. As a static pipe it retains a fixed configuration during the entire operation, therefore a change of function requires that the pipe be flushed before performing the modifications. A dynamic pipe on the other hand can change configuration before completing the operation on the previous data set. These pipes are more difficult to control but they are more efficient, if the operation must be frequently changed.

c) **Multi ALU system**

In a multi ALU system, the instructions are considered as task that can be executed on dedicated units. The operations specified by these instructions, are used to set up a number of functional units which are computing in parallel. In these systems, data dependencies between the instructions will reduce the efficiency of the system. Instruction in this type of pipelining can be executed out of order, therefore problem such as read before write, write before read, and write after write can occur (RAMA 77).

A SISD processor can have multiple A.L.U. units, such as a floating point unit, a multiplication unit, etc. however in such organization, only one unit is operating at any one time.

For all types of pipeline architecture, branching and interrrupt conditions reduce the efficiency of the pipe, since it is necessary to either flush the pipe and restart or store the information to continue from where the process left off. All of the processors in this class are limited to a single stream of instructions and to a specific class of problems for which concurrency can be exploited.

2.3.1.2 Array Processors

Array processors consist of a single control unit and n interconnected processing elements. A general block diagram of an array processor is shown in figure 7 . The processing elements are independent from each other, and operate on command from a central control unit. A single command is necessary to activate all processing elements, therefore they all perform the same operation. For some applications, only a subset of the processing elements may be activated, while the rest remain idle. The control unit does the I/O, inhibits the operation of the processing elements and controls the sequencing of instructions.

Today more and more array processors are used as add-on devices to increase the performance of existing serial processors (CASP 78). With such peripheral array processors, the host provides the overall system control, and executes simple arithmetic operations, programs, and system control instructions, while the array does the high speed vector calculations. Peripheral array processors are also called signal processors.

With respect to Flynn's classification they are considered as SIMD, that is a single stream of instructions controlling multiple data streams.

Array processors are efficient for solving problems that have the following characteristics.

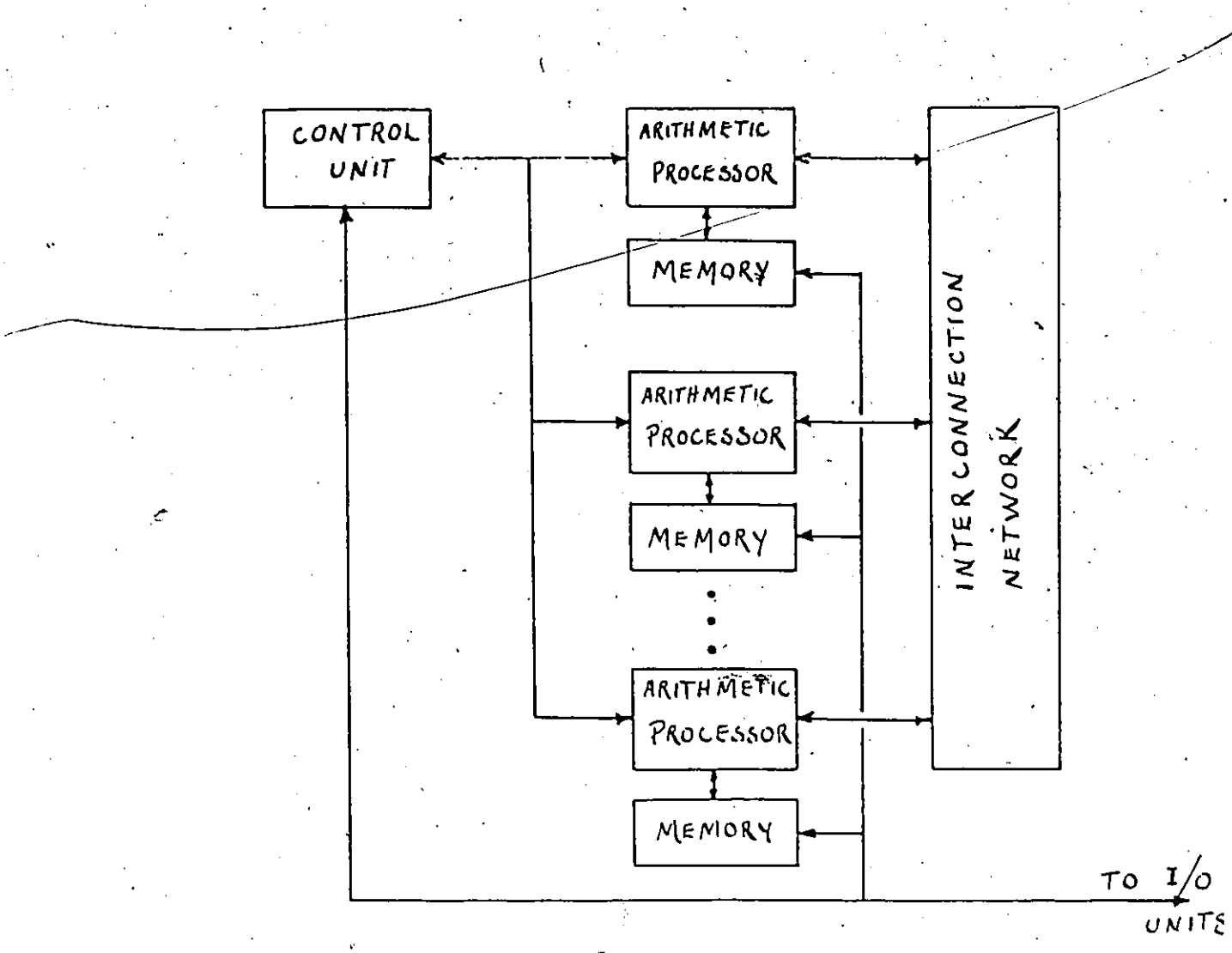


Figure 7: Array processor

1. The computation can be described by vector instructions, such that a majority of the computation time is spent with many identical operations in action simultaneously on different data. Such processes include multiplication, summation and scaling of data arrays or vectors and the computation of logarithms and fast fourier transforms.

2. The data transfer between the processing elements is small. This transfer is accomplished through the interconnection network, which in a sense specifies the range of applications that can be performed efficiently on the system. For example, the following interconnections have been studied: linear array, ring, star, tree, cube, shuffle-exchange, data manipulator, etc. (FENG 81). Each has advantages over the other for some particular applications.

To exploit the power of array processors, they must be programmed to take advantage of the array processor structure. For example the use of such statement as DO PARALLEL, IF ALL, and IF ANY have been studied (BAER 76). Some languages have been designed for such machines, however they are unnatural since they reflect the underlying structure and not the manner in which programmers think about a problem. A simpler approach to programming array processors, is to use a conventional high-level computer language, such as Fortran, and to employ special subroutines that perform the array operations. A compiler can also be designed to detect parallelisms, however this is not a straightforward process (BROD 81). Array processors are therefore efficient in special applications only.

2.3.1.3 Associative processors

Associative processors are special purpose processors used in applications where data bases must be searched. Such operations are cumbersome on a von Neumann computer, since it is not well suited to performing non-numeric processing.

The following three classes of associative processors all perform non-numeric operations in a concurrent fashion. Again, these processors are classified as SIMD using Flynn's classification, since an operation is performed simultaneously on many memory cells.

1) Content addressable memory associative processors

These processors provide a small amount of logic within each memory cell so that simple operations can be executed for all memory locations simultaneously. The main unit of these processors is the content addressable memory, which is also referred to as an associative memory. This type of memory automatically compares an input code to each word in the memory. Output signals are produced to indentify that a match condition has occured. If a complex arithmetic operation is required, it can be performed on a regular A.L.U. circuit by fetching the matched words serially. These processors can therefore be considered as being based on the compare operation, instead of the add operation as in von

Neumann's architecture. The disadvantage of this architecture is that large content addressable memories are expensive. Available content addressable memory intergrated circuit are presently small, for example the 10155 is a 8 words by 2 bits memory, while the F100141 is a 4 words by 4 bits memory.

2) Distributed logic associative processors

Instead of associating the logic with each cell, a number of cells can share the same logic circuitry in order to reduce the complexity of the memory. This however reduces the overall speed of the processor. An example of such a processor is PEPE, a parallel element processing ensemble (YAU 77), where the logic is associated with each character.

An alternate way is found in the STARAN processor (THUR 75, SIEW 82). Here with each one of the 256 memory words is associated a one bit processor (fig. 8). All the processors are interconnected, and they process a one bit-column of all the words at the same time. Processors with this or similar organizations, are referred to as associative array processors, or bit-serial associative processors.

Orthogonal computers are bit-serial processors, that have an associated serial processor that accesses the same memory (HIGB 73). The memory thus allows bit-slice and word access to the memory. This greatly increases the throughput of the

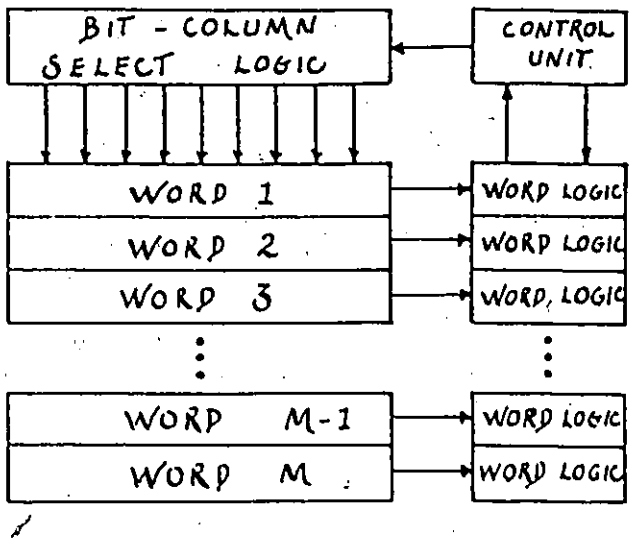


Figure 8: Associative array processors

memory, and permits more natural input/output operations. Furthermore serial and parallel operations can operate concurrently on the data in the memory.

3) Word serial associative processors

These processors can be considered as hardware implementations of a simple program loop that performs a search operation. A circulating memory is used, so that any word in the memory passes through the comparison logic, as shown in figure 9 . When a match occurs the word can be sent to the A.L.U. to be manipulated. A multiplexer is used to select new data or to recirculate the old data.

The circulating memory can be a fixed head disk. In this case, a small processor is placed on each track of the disk, so that all tracks can be searched concurrently. This type

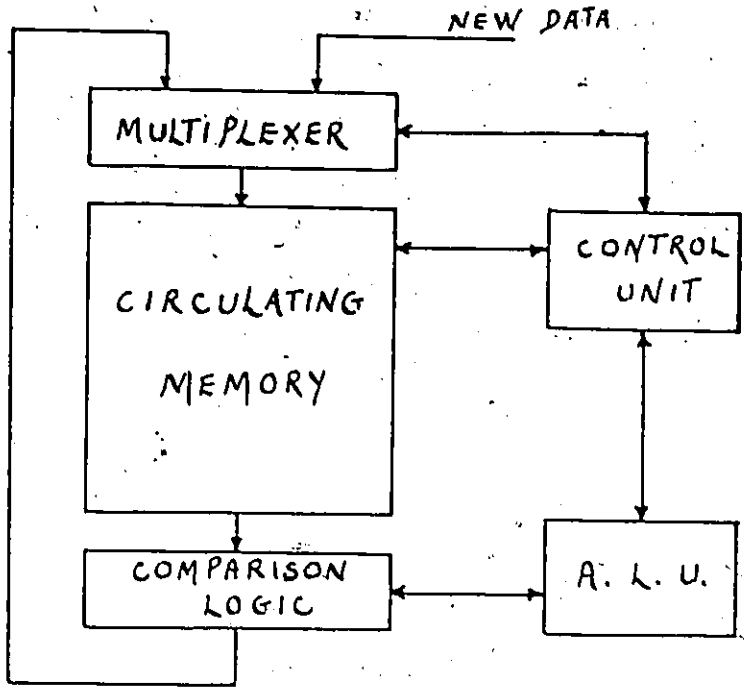


Figure 9: Word-serial associative processor

of organization can be added to a disk to enhance its capabilities, and free the main processor. Other architectures can be devised that implement software searching algorithms in hardware in order to speed up the execution (ALTA 79).

2.3.2 Concurrent operation at the processor level

Under concurrent operation at the processor level, the various types of multiprocessor systems are considered. In these systems, the concurrency of execution can be quite varied, ranging from instructions to program segments. However, compared to the processors in the previous sections, they all belong to the MIMD class of Flynn's. We shall consider these systems under two major classes depending upon their functional similarity to the von Neumann machine.

2.3.2.1 Von Neumann type

The multiprocessors in this class are related to the von Neumann machine with respect to the execution of instructions. In the next section, processor with non-von Neumann characteristics are considered.

a) Tightly coupled system

Tightly coupled systems are characterized by having a single intergrated operating system that coordinates the interaction between the processors, with communication between the processing elements being done via shared memory (ENSL 77). These systems generally also include shared I/O devices. The processors of such systems have comparable capabilities and can execute significant computations on their own. The interaction between the processors can occur at many levels; between complete jobs and tasks or between individual job steps. Therefore, these processors are either executing a common job, or a number of separate jobs simultaneously. An early example of a tighly coupled system, is the use of I/O processors to perform I/O operations concurrently with the execution of a task.

Such systems can be organized in a master/slave mode of operation, where a master processor is in charge of the execution of many slave processors. The master can be a special unit or one of the processors. The latter case produc-

es a more reliable system, since any processor can become the master. In another mode of operation all processors operate at the same level. This requires a more rigid coordination between the processors. An example of a simple tightly coupled system is shown in figure 10, where all shared elements are placed on a time shared bus.

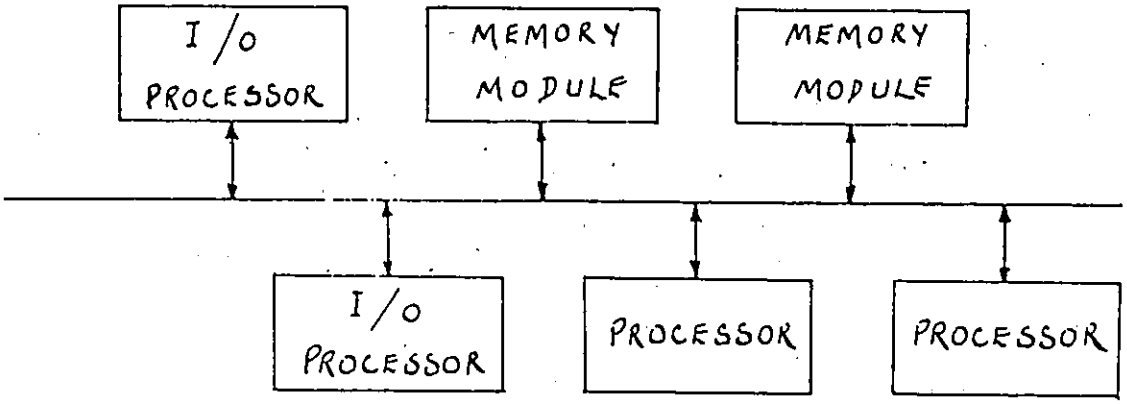


Figure 10: Tightly coupled multiple processor system

The major disadvantage of these systems is that processor interaction is done via shared memory. This creates access conflicts and therefore slows down the execution. Sharing memory implies that a rigid synchronization must be available between the cooperating processors. The number of processors that can be interconnected in such a system depends upon the interconnection scheme between the processors. The time shared bus, shown in figure 10, is the simplest scheme, it however limits the number of processors in the system, to a small number. Other interconnection schemes, such has

multibus, crossbar switch, multiport memory (ENSL 77, THUR 72), segmented bus (FATH 81b), and binary tree (PATT 79) have been designed in order to increase the number of cooperating processors.

Since there is a close interaction between the processors, multiprocessors are efficient for executing a single task by taking advantage of all possible parallelisms. However, in general these systems use conventional high-level language, in which programs and tasks are presented in a sequential format. This makes it difficult to specify the possible parallelism. In general, to be able to take advantage of the total system capabilities, one would include complex preprocessing to isolate the parallelism. The following class of multiprocessor systems use special instructions to implement the required parallelisms, to efficiently execute a single task.

b) Parallel control flow computers

In contrast to sequential control flow computers, such as the traditional von Neumann computer, parallel control flow computers make use of special instructions to support multiple control sequences. In the simplest form only two instructions are needed: FORK and JOIN. The instruction FORK is used to start a number of instruction streams, that are executed concurrently on dedicated processors, while the instruction JOIN, is used to group a number of instruction

streams into a single one (TESL 68). These insructions are sometimes referred to a COBEGIN and COEND, while the parallel instruction streams are referred to as COROUTINES. Programs in such computers are conveniently represented in terms of directed graphs. For example, figure 11 illustrates a control flow program which calculates the factorial of a number.

To make the system more flexible, any instruction can be made to request a number of control signals before it can be activated (TREL 81). This eliminates the FORK and JOIN instructions, since parallel control streams are supported by every instruction.

The drawback with these computers is that usually the programmer must specify where the multiple streams do occur. This is not a straightforward process since we do our problem solving in a sequential manner. Furthermore, parallel control flow multiprocessors are based on the von Neumann principles of linking data to storage locations at run time. This reduces the parallelism due to the storage dependencies.

2.3.2.2 Functional language computer

To overcome the inadequacies in von Neumann related multiprocessor systems, such as the storage dependencies, functional language computers have been developed. These multi-

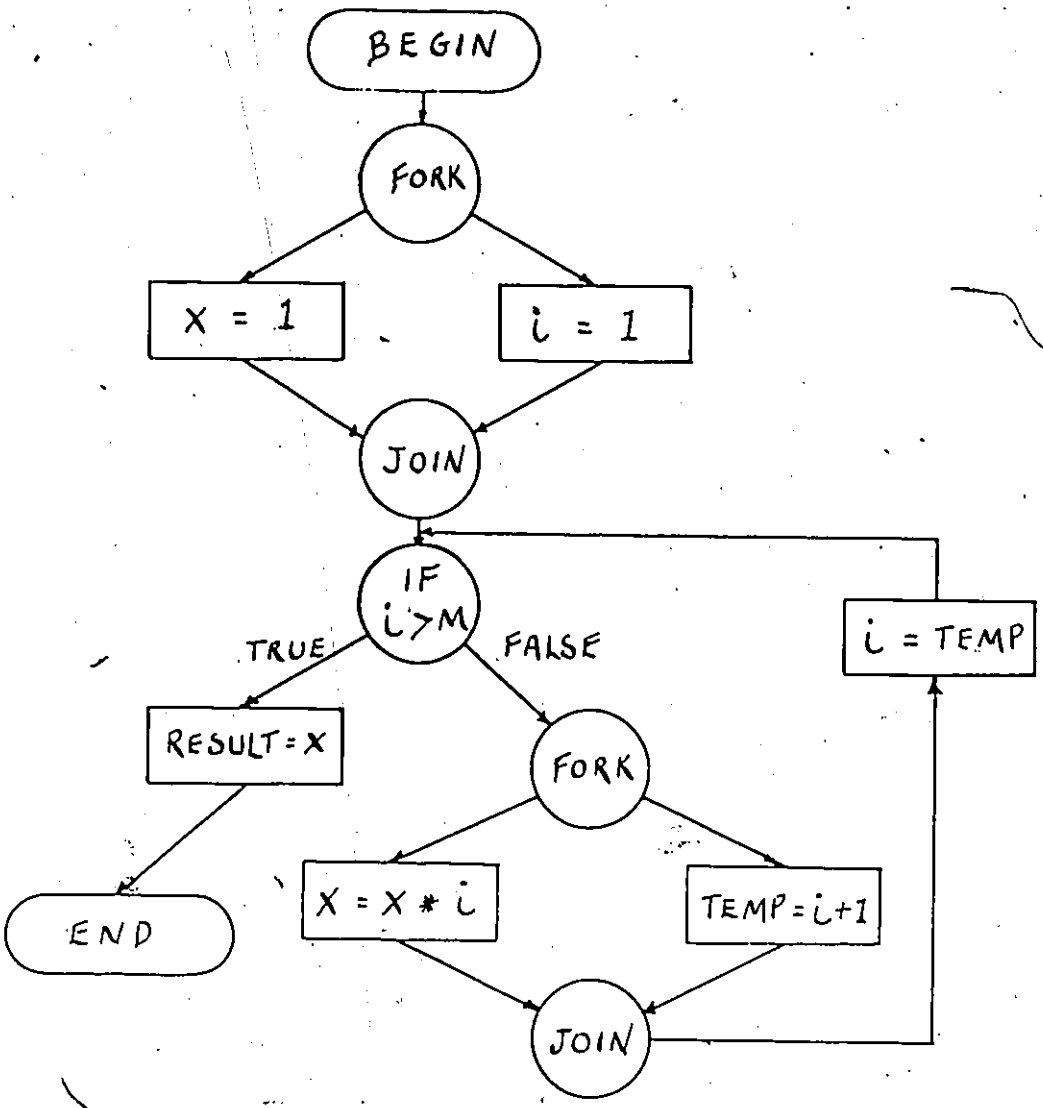


Figure 11: Control flow program

processor systems are based on a functional language that operates by application of functions to values (BACK 78). These languages are directly interpretable, and have no notion of variables (does not specify storage location), that is, a program does not store information that can affect the

behaviour of a later program. Instructions in a functional language program are executed when the data needed by them is available. These instructions produce output values which are used by other instructions. Once used these values are no longer available to other instructions. This permits the execution of many instructions concurrently. Functional language computers are based on a radical approach to program execution, they are referred to as non-von Neumann computers. Here we shall review two types of computers based on these concepts: the data flow and tree processors.

a) Data flow computer

A data flow computer deals only with values and not addresses of values, as in the von Neumann computer. An instruction in such a computer is executed when all its required operands are available. Therefore the concept of a program counter does not exist, since sequencing constraints are imposed by the data dependencies in the algorithm. Executed instructions consume input values and produce a set of output values. The input data once consumed are no longer available to other instructions. Therefore data flow machines are not history sensitive, that is a program segment does not store information that will affect the behaviour of a later segment.

A program for a data flow machine is described in terms of a directed graph that illustrates the flow of data between the instructions. The nodes of this graph represent the operations, while the arcs represent the data dependencies between these operations. Figure 12 illustrates such a data flow program that calculates the factorial of a number. The instruction Merge is executed when one of its input is available, while Copy is used to create a copy of a token.

A data flow processor must recognize which instructions are enabled and dispatch them to an execution unit as soon as the resources are available. Instructions can therefore be executed concurrently. The structure of a data flow computer developed by Dennis is shown in figure 13 (DENN 80).

The update unit places instruction that are ready to execute into the instruction queue which are then executed by the execution units. The update unit also receives the result of an operation, and enters them at the proper location in the instruction store. This architecture is one of many that are presently being investigated (see related articles by P. C. Treleaven).

Data flow processors offer significant advantages in the execution of a single task by exploiting concurrency between individual instructions. Presently there is still some difficulty with these systems, for example they are not efficient for handling complex data structures. Parallel con-

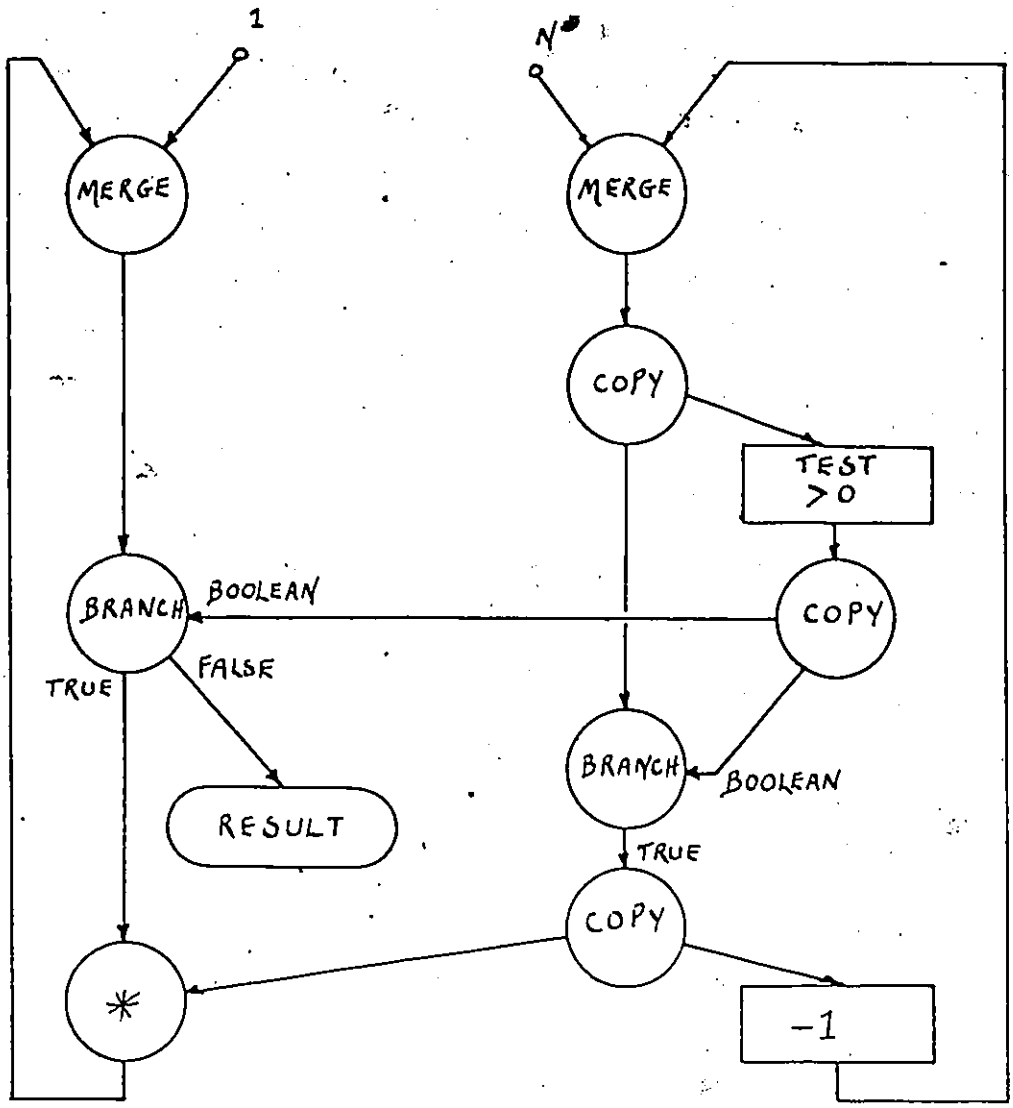


Figure 12: Data flow graph

trol flow computers, on the other hand, can be more easily adapted to solving complex data structures. Furthermore they are not suited to handle complex I/O operations. Presently, most prototypes developed use a host computer to provide the I/O operations. Another point to consider is

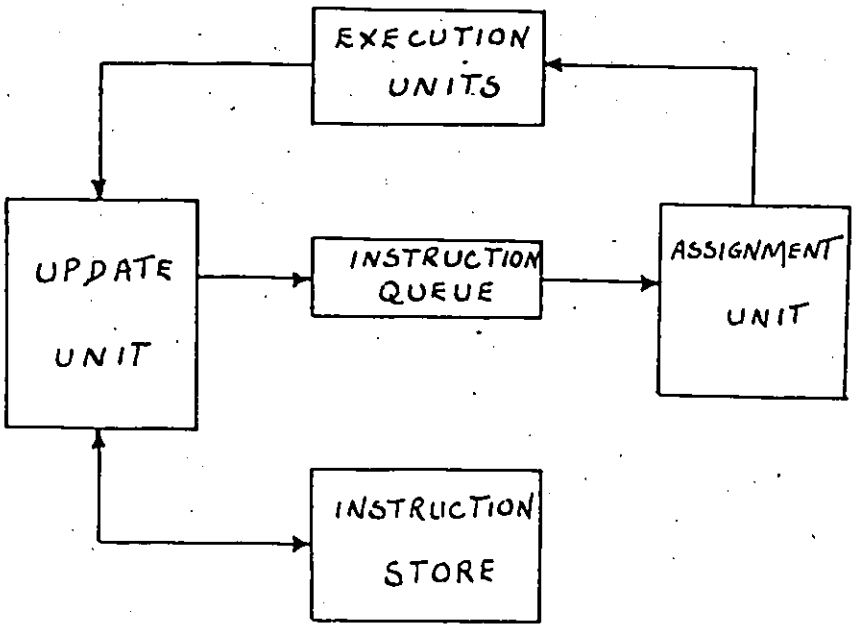


Figure 13: Data flow computer

that they require much user expertise since they are programmed in an unconventional manner.

Multi ALU systems described in the previous section use a form of data flow analysis to obtain concurrency, however since the instructions imply variable locations, the concurrency is limited by the storage dependencies.

b) Tree processor

Tree processors are similar to the data flow computers, except that the programs are described in terms of trees instead of a graphs. The reduction machine (KLUG 79b) is a typical example of such architecture. The principles behind this architecture can be found in BERK 71 and BERK 78.

An expression in a tree processor, is evaluated by a substitution process which in an ordely manner traverses the expression and successively replaces each subexpression by others that have the same meaning. These processors are also referred to as reduction processors, since they are based on a reduction semantic. Reducible expression are recognized by traversing the tree in a pre-ordered fashion. When one such subexpression is encountered it is reduced. The requirements for a machine that would perform in such a fashion are given below.

1. Represent the execution language expression in a suitable storage. For example in a pre-order notation on a push down stack.
2. perform pre-order traversal by a succession of pop operation if a stack is used.
3. recognize at the same time reducible expression
4. execute reduction
5. resume operation.

Figure 14 shows the successive reduction of an expression, in order to obtain the final result.

These operations are similar to the one found in a stack processor, however the subexpressions in this case are executed concurrently and may contain decision making nodes, such as the IF node, which is used to select the proper subtree out of a pair of subtrees.

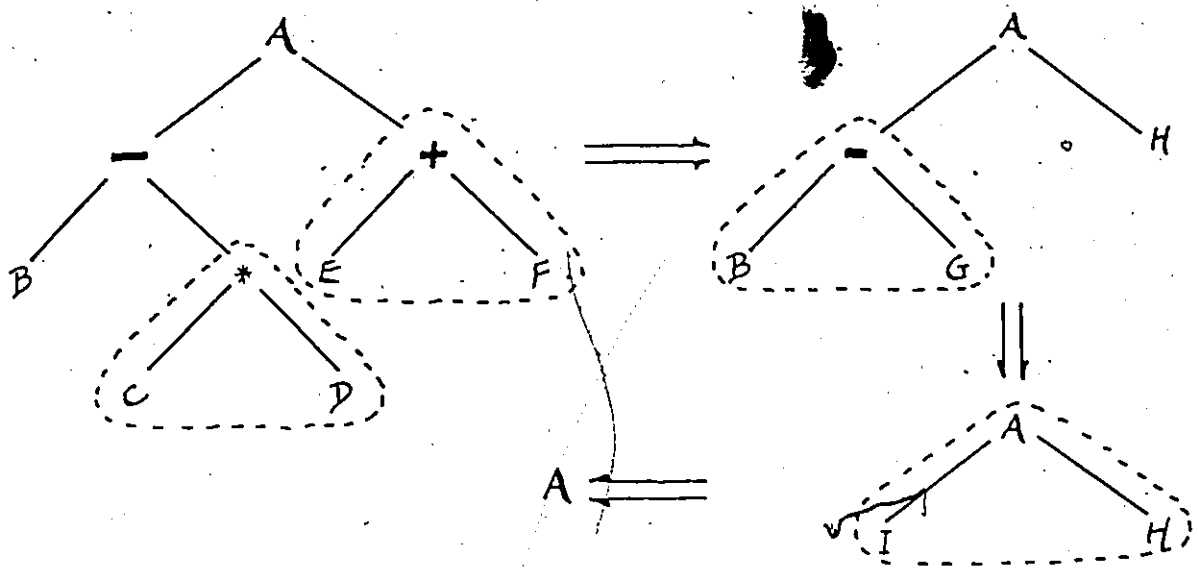


Figure 14: Reduction of an expression

Reduction processors have received less attention than data flow processors, since reduction semantics are an unfamiliar form of program execution for most computer designers. A description of the research in this field can be found in TREL 82a.

2.3.3 Concurrent operation at the task level

In contrast to the tightly coupled systems of the previous section, the present one will describe more loosely coupled systems that support many concurrently executing tasks. These multiple processor systems communicate by a preestablished message protocol, instead of by sharing the main memory. These systems can be subdivided into two classes depending upon the degree of coupling between the processors.

2.3.3.1 Computer networks

Computer networks are loosely coupled multiple processors which consist of several separate and discrete computers. The network interconnecting the computers is used for communication, while the actual computation is done at a single site. The network topology is a critical factor in determining the reliability, and message handling capability of the system. Many interconnection schemes have been devised to minimize the network cost and increase its throughput. For example, ring, star, mesh and tree topologies (GREE 79) have been investigated. Furthermore, the transfer strategy on these networks, has received considerable attention since it also affects the throughput. Presently, packet transmission is predominant in applications where the communication is in a burst mode.

Phylosophically, these systems can be regarded as a group of individual computers, since load sharing cannot be performed. Computer network systems were devised so that users could share data bases and processing resources, in the sense of executing a process on the best suited processors.

2.3.3.2 Distributed system

Distributed computer systems (ENSL 78, RAMA 81, KRIE 81), have a degree of coupling that is situated between the multiprocessors and the computer networks. They are therefore

referred to as moderately coupled multiple processor systems. A distributed system is characterized by the inclusion of a high-level operating system that supervises the operation of the distributed components. The individual processors have their own operating system which supervises the execution of a task. They too, like computer networks, communicate by messages. Unlike the computer networks, the physical and logical resources can be assigned to specific tasks on a dynamic basis. Each processor consists of a central processing unit, local program and data memory, and the required peripheral units. Expensive resources can however be shared by the processors. The topology of the intercommunication network can be arranged like in computer networks, however there is a closer interaction between the processors. Usually in a computer network high speed serial lines are used to interconnect the computers, while in a distributed system parallel busses are used.

For these systems, the workload is partitioned into relatively independent tasks which are then assigned to the various processors for execution. These systems can be organized as homogeneous or heterogeneous systems. In a homogeneous system, general purpose processors are used, in which each is able of supporting a number of varied tasks. For a heterogeneous system, the processors are more specialized and dedicated to performing specific tasks. The task in this case will determine the complexity of the elements.

Since the processors in this case are dedicated to special tasks, dynamic load sharing cannot be performed, therefore proper load balance must be done at the design stage. Usually, to improve reliability, a task should be executable on more than one processor, in order to provide a fail soft mode of operation.

The advantage of distributed data processing systems, is that the workload can be dynamically shared. Furthermore, they are reliable since they are built in a modular fashion, and cooperation between processors is possible to execute complex tasks. The modularity also contributes to flexibility, adaptability, and incremental expansion. Using a terminology in line with Flynn's, a distributed system can be considered as a multiple task, multiple data streams processor, since a number of tasks each utilizing different sets of data are executing concurrently.

According to Enslow, many systems are presently called distributed systems, however they must possess all of the above characteristics to be true distributed systems. In chapter three, the task driven system (KRIE 79) having all of the above characteristics, will be described.

2.4 REDUCING THE SOFTWARE OVERHEAD

The von Neumann architecture which is the basis of most present day computers is based on a bottom-up design approach, of minimizing the cost of the hardware and letting the programmer (user and system programmer) solve all the difficult problems. This approach although important in the 1940s and 1950s, makes little sense today as complete and powerfull systems can be designed with a few LSI building blocks. Furthermore, three decades ago, assembly code was an important programming medium, therefore the machine had to be presented to the programmer in a convenient form. Today this is not the case since we are more concerned about efficient execution of problem oriented language programs.

In this section, the methods devised to execute more efficiently high-level language programs are investigated. The various methods will be classified under three general headings: using special hardware units, increasing the level of the intermediate language and modifying the data and control structure of the machine.

2.4.1 Special hardware units

A straighforward method of executing more efficiently a high-level language program, is to implement some of their functions as special hardware modules. For example in the early days of computing, multiplier hardware, or automatic

floating-point arithmetic units were added to attain higher computation speed. Today, complex data manipulation can be performed efficiently by a hardware module instead of a software module. This therefore simplifies the machine language program that is emulating these high-level language functions.

A typical example, is the use of a tagged architecture that facilitates the representation and manipulation of data structures. In this type of architecture, additional bits are added to each data word to describe its type and structure. Therefore, operations such as type checking can be performed automatically by hardware modules. This architecture also makes easier realization of protection mechanisms, and instruction coding and error detection (FEUS 73).

Special hardware units can also be added to implement some of the functions of the system software. For example, memory management functions are more and more being performed by hardware, in the newer microprocessors.

To a great extent, adding special hardware units is done in a trial and error manner by taking advantage of the present technology. An idea that is presently considered, is not primarily to use more efficiently the hardware resources, but to achieve a more efficient execution.

2.4.2 Increasing the level of the intermediate language

A second approach is to modify the level of the intermediate language. In the traditional von Neumann computer, the compiler generates machine language code whose instructions are executed (interpreted) by the control unit. A present day approach is to compile high-level language program into intermediate level languages. This intermediate level language is then interpreted by the machine (fig. 15). The size of an intermediate language program can be reduced by increasing the complexity of the intermediate language so that it will mirror more closely the high-level language data and control structures. In this case, the complexity of the compiler is reduced, however the size of the interpreter will be increased due to the dissimilarity between the machine and the intermediate language. The machine in this case still retains the characteristics of von Neumann's. This is done for example for Pascal with the use of p-code. The interface level between the compiler and interpreter is arbitrary and its choice usually depends upon the language supported, and upon the underlying machine.

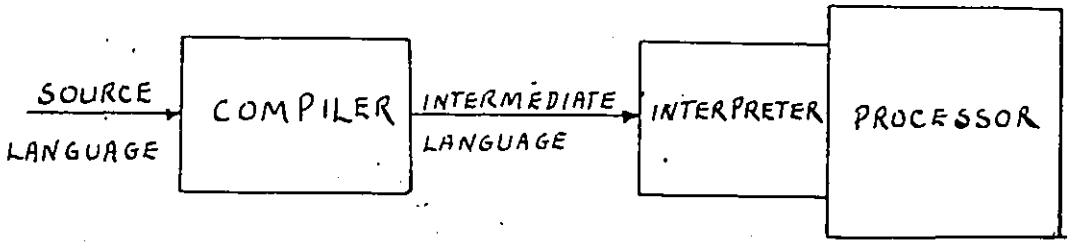


Figure 15: Compiler and interpreter interface

To provide better efficiency, one can have different intermediate languages for different high-level languages. In this way one can match better the specific high-level language with the machine code. In some cases one can also have additional machine language instructions to facilitate the interpretation. Today some intermediate languages are becoming more popular (e.g. p-code) and there are a number of machines that execute them directly. These machines are considered in more detail in the next section.

2.4.3 Modifying the data and control structure of the machine

A more efficient method of executing a high-level language program, is to modify the hardware in order to support directly more complex structures. This is similar to the previous method except that the complexity of the interpreter is reduced, since the hardware directly supports the complex data and control structure of the intermediate language. In the traditional von Neumann computer, complex data and control structure are supported solely by software, and the various high-level language stuctures must be transformed into elementary entities and operations. The large dissimilarities in the von Neumann computer, between the machine language and the high-level language contributes to software unreliability, performance limitations, excessive program size, and compiler complexity. This also implies

that the user must be familiar with the machine language in order to understand the diagnostic messages from the execution of a high-level language program.

Architecture that supports directly more complex data and control structure can be grouped in two types: the indirect execution type and the direct-execution type (CHU 81). In an indirect execution machine, the high-level language is first translated by a compiler into an intermediate language program which is then directly executed by the hardware. The von Neumann computer can be considered as an indirect execution computer where the intermediate language is at a very low level. In a direct execution computer, the high-level language is the machine language, therefore there is no need for compilers or complex support software.

2.4.3.1 Indirect execution computer

In an indirect execution computer the intermediate language generated from the compiler, determines the complexity of the machine. The complexity of the machine increases as the data and control structure supported increase in complexity.

Many types of computers can be included in this class, they depend upon the proximity between the lexical, syntax, and semantic properties of the high-level language and the intermediate language. Another example of an indirect exe-

cution processor at a low level, are the stack computers (MCKE 80). A stack processor stores the operands on a last-in first-out stack and performs the various operations with respect to the top elements of the stack. If the top of the stack is available in high speed registers, then the operations can be executed at register speed. These stack computers are natural for executing polish string expression. The stack can also be used for parameter transfer, recursive subroutine control, procedure call processing, and rapid context switching, which makes them efficient for executing block structured languages. In a sense, these computers increase the proximity with respect to the syntax property, since high-level expressions are efficiently expressed and executed; for example the Burroughs B6500 computer efficiently executes block-structured Algol like languages (BURR 69). Stack processors have also found use in the implementation of compilers and interpreters, and in memory management. A major advantage of stack computer is that they use zero address instructions, thus allowing the packing of instructions. However, stack processors are inefficient for manipulating complex data structures.

The proximity of the lexical and semantic properties between the high-level and execution languages can also be increased in order to obtain a more efficient machine. The Symbol computer (MYER 78, RICE 81), for example, uses a polish string language that is intermixed with delemiters.

Thus the lexical properties, along with the syntax properties of the intermediate language, are more oriented toward those of the high-level language. This implies that the compilation steps are less complex. Furthermore, the Symbol computer has special hardware modules that implement directly such functions as compilation, text editing, memory management and time sharing, arithmetic, and variable sized data structures. The Symbol computer was mostly a research project to show that most functions required when executing a high-level language program could be performed in hardware instead of software. Newer architectures are presently being devised to further reduce the properties gap between the machine and the high-level languages (CARL 75). What is observed is that they are more and more oriented towards the direct manipulation of complex data structures. For example, in Starlet (GILO 75), the basic data structure is that of ordered sets. These sets can consists of the elements of arrays or strings.

As the level of the intermediate language increases, the compilation becomes simple. The underlying hardware, however, becomes more and more complex. The major goal is to obtain an intermediate language that can be used for a number of high-level languages and that facilitates the design of an efficient hardware system.

Supporting more complex data and control structures is now available on some third generation microprocessors; the Z8000 from Ziloq, the MC68000 from Motorola, and the NSC16000 from National semiconductor all have complex instructions that correspond to statements in such high-level languages as Fortran, Pascal, and Ada (BERN 81). Furthermore the NS16000 has symmetrical addressing which allow each instruction operands to be specified by any of the addressing modes (MART 81). This simplifies the design of the compilers, since it can treat registers and memory, as well as source and destination symmetrically (WULF 81).

Fourth generation microprocessors will go beyond this as indicated by the introduction of the Intel iApx 432 micromainframe (RATT 81). This microprocessor is geared toward the efficient execution of Ada programs. Furthermore, its code automatically performs some of the functions usually associated with the software operating system. It should be noted that microprocessors are being developed to support complex software function more rapidely than the large mainframes. This is the case because the large mainframes must remain compatible with the large amount of software that was developed over the years.

This trend of supporting more complex instructions directly by the microprocessor is not however uphelded by everyone. Some computer scientists believe that a more cost

efficient solution would be to improve compilers to simplify the programmer's jobs, while maintaining a simple processor architecture that efficiently supports the basic operation (BERN 81). These two approaches must be considered in the design of new microprocessors, however this is beyond the scope of this thesis.

2.4.3.2 Direct execution computer

The compiling stage of a computer can be totaly removed by directly interpreting the high-level language. This implies that the steps usually performed by the compiler are done in an interactive fashion by the hardware (CHU 81). For example, a typical direct execution computer would have a lexical unit that would scan the source program to identify the various tokens, a syntax unit that would assemble these tokens into meaningfull strings, and an execution unit that would process these strings. A direct execution computer offers many advantages; for example program debugging is simplified since the errors are obtained in an interactive fashion and they can be more easily related to the source program. Execution speed can be high since the machine can directly execute the actions required by the high-level language.

However these computers are inefficient since the steps of lexical and syntax analysis must be performed for all segments executed. Furthermore since the branch addresses are not known at run time, the program must be completely scanned in order to obtain the branch locations. The program size in such a machine is also considerable since symbolic names are used to represent variables, procedures, etc.

As mentioned before, these machines offer the advantage that the program are executed in an interactive fashion, therefore facilitating the debugging stage. For instance, if a user mistypes a token, or violates a semantic rule, the error can be recognized immediately, upon which the execution stops, so that the error can be corrected.

It is questionable that one would implement a general purpose computer using this approach, since they support only one high-level language. It is expected that direct execution computers will find their use mainly as special purpose machines, such as small desk top personal computers.

2.5 RELIABILITY IMPROVEMENTS

The last major problem associated with the von Neumann computer, is its poor reliability. Reliability is a critical factor in such applications as advance guidance systems for aircrafts, air trafic control, nuclear reactor control,

processing plants, life-support equipment, etc. In such applications, fault tolerance is a desired objective, that is the system must overcome software or hardware malfunctions on its own, since a failure might result in serious inconvenience, financial loss or even loss of lives. The von Neumann architecture is not implicitely reliable since a failure of a component results in the failure of the complete system. This can be observed by examining present large and complex von Neumann based maxicomputers, where the mean time before failure is so low, that their speed improvements are nearly cancelled (AVIZ 78). Considering the number of components of a present day computer, only the extremely high reliability of individual components makes the computers operational. Improving the reliability of a system implies that the mean time before failure (MTBF) is increased.

The most basic approach to improving the reliability of a system is by fault-avoidance, which is achieved by choosing and assembling the components in a near perfect manner. This can reduce the occurence of faults, however it cannot eliminate all faults completely, and it produces very costly system. Fault-tolerant techniques use standard components however they increase the reliability by using redundancy. The redundancy can be either temporal or spatial. Temporal redundancy implies the repetition of a calculation in order to eliminate transient fault, permanent fault are not detected in this approach. Transient fault or errors, can

also be detected or corrected by using special codes. An error detecting code is used to perceive that errors have occured in a string of bits, while an error correcting code will identify the exact bit that is in error.

In spatial redundancy, extra hardware or software is used to correct faults. In a spatially redundant fault tolerant system, three functions are required, they are: detection, diagnosis, and correction. The simplest system would only include the detection function, and the diagnosis and correction would be performed by the maintenance staff. However in some applications, as mentioned at the beginning of this section, the function of diagnosis and correction must also be performed by the hardware. The inclusion of these functions can be achieved in a number of ways.

1. Massive redundancy. By using a number of identical units, errors can be masked using a voting circuit. The voting circuit selects the most probable output from the set of outputs of the units. Units that are disagreeing can be disconnected and repaired. This is conceptually is the simplest approach, however it is very expensive, and has a limited fault-tolerance capability. For example if too many units fail, the system might not operate correctly.
2. Selection redundancy. In this approach, a faulty unit is replaced automatically by a stand-by unit. Effective replacement requires that the fault be de-

tected, and contained. Furthermore, system recovery must be performed to restore proper system operation. This approach is limited by the number of stand-by units.

3. Fail-soft. The third approach of improving the reliability, is using a fail-soft mode of operation. Fail-soft operation mostly applies to multiprocessor system or distributed system where the workload of a failed unit can be taken by another unit, therefore producing a graceful degradation of the system. This approach is limited by the number of units capable of performing the same task. Many such system are presently studied, such as the assembly line multiprocessor (NEGR 77), AXE, .C.vmp, FTMP, and SIFT (MALA 80).

4. Forced no response. A fourth approach to implementing the function of detection and correction is forced no response (KRIE 81). In this case, the units are designed with enough intelligence as not to respond in case of failure. Therefore, when an unit fails to respond after a certain time period, its task is assigned to another unit.

The previous discussion considered the detection, diagnosis and correction of hardware faults. However, to obtain a truly reliable system, one must also consider the reliability of the software system (STIF 81). This is not considered

here, however it is of equal importance with the reliability of the hardware. We should mention however that reliable software system can be obtained using a good design strategy, as shown in DIJK 68.

Another issue that is very important today, due to the prolifercation of large data bases, is protection against unauthorized access. This is in a sense related to the reliability of the system, since the data to be error free must be protected against any contamination, be it accidental or intentional. This is a very important research topic, since large data base contain pertinent information relating to governments, industry and even individuals. Under no circumstances does one wish to receive bills for unacquired goods, or have his medical record examined by unauthorized persons.

2.6 CONCLUSION

This chapter presented the major techniques developed over the years to overcome the four main problems associated with the von Neumann architecture. One should note that there are many other thechniques which were developed to overcome problems that occur in specific applications. Good examples, are the various processors that are being developed for picture processing. These were not included here since the goal is to design a general purpose computing system.

This analysis has permitted an understanding of these developpements and a recording of their advantages and disadvantages. This is necessary to achieve the goal of this thesis, which is the design of an efficient general purpose computing system. This system to be efficient must reduce some of the drawbacks associated with the von Neumann architecture.

Chapter III

A DISTRIBUTED COMPUTER SYSTEM

3.1 INTRODUCTION

It has been shown in the previous chapter that over the years, there has been a lot of research to design better and faster computing systems. In obtaining the material for that chapter, the following conclusions have been drawn:

1. There were many succesful designs for special purpose applications.
2. Most of the present general purpose machines, although incorporating some of the mentioned techniques, can still be classified as von Neumann's.

This is the case, since a drastic change in computer architecture requires enormous outlays of software capital, both in terms of reeducating the users and developping completely new system software. In line with this, it is suggested that any new general purpose computer should support the constructs of present day high-level languages, to reduce the software related costs. Secondly, the system architecture should be such that it either leads to simple system software, or implements the system's control functions in hardware.

These two observations can be seen in terms of the cost decision pyramid. This cost decision pyramid depicts the tradeoffs in a system between the hardware, the software and user expertise (fig. 16).

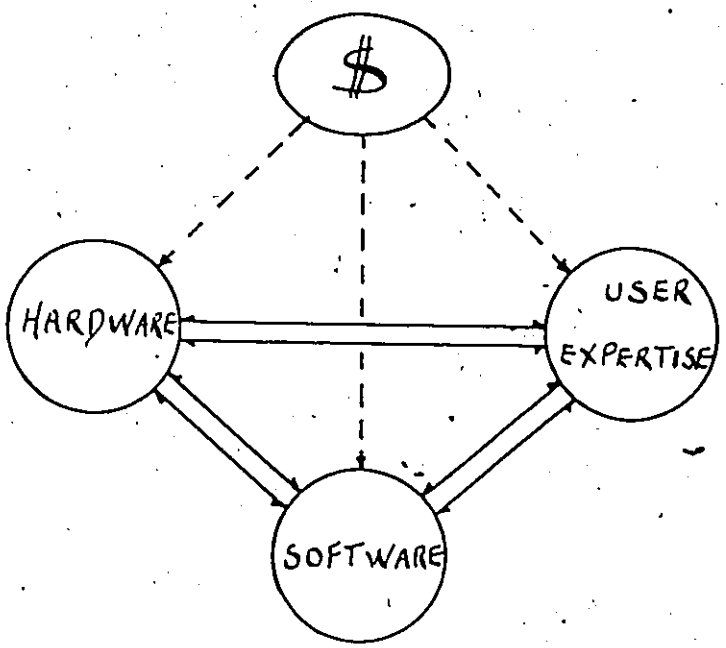


Figure 16: Cost decision pyramid

Obviously today, with the development of LSI and VLSI circuits, where a few of these devices can be used to build complex systems, the trend is to use more hardware oriented systems. This is further emphasized by the high cost of developing and maintaining large software systems. Furthermore, user expertise is at a premium, therefore the system should be user oriented, that is, it should facilitate the editing and debugging of software components. This can be observed in COPE 82, where the implication of having free mass storage is analysed with respect to the human factors.

In this chapter, considering the material presented in chapter two, the architectural characteristics of an efficient general purpose computing system are defined. Following this, a virtual language that has the main characteristics of present day high-level languages is introduced. Finally the system organization and operation, which are based on the previously described virtual language, are presented in detail.

3.2 SYSTEM ARCHITECTURE

The various techniques devised to reduce the problems associated with the von Neumann architecture were analysed in the previous chapter. These techniques were classified according to the main problem that they attempted to solve. The following four classes were described: reducing the memory bottleneck, providing concurrent operation, reducing the software overhead, and improving the reliability of the system. Each of these classes of techniques will be considered separately, to suggest an architecture for an efficient general purpose computing system.

The techniques for reducing the memory bottleneck described in section 2.2 produce limited improvements at best in SISD systems. They will not be considered any further here, since conceptually they do not change the operation of a system, except to mention that they could be added to the

proposed system in various places, to speed-up the transfer of information.

Supporting concurrent operations was described next. Concurrency at the instruction level in a sense increases the throughput by supporting more complex data structures, however these systems are mostly useful for special applications, which is not what we want to achieve here. One should mention that the ideas of overlapped operation and multi-ALU can be introduced relatively easily in most systems, to enhance their performance. Furthermore, complex techniques such as pipeline, array and associative processing can be used to design special add-on units.

The second type of concurrency examined was at the processor level. The von Neumann type multiprocessor systems are faced with the difficulty of expressing the concurrency, and are plagued by memory conflicts since the processors communicate via shared memory. The use of functional languages can solve many of these problems, however they require completely new programming techniques. Programmers are used to thinking in a sequential manner, and in terms of variables, concepts that do not exist in these languages. Therefore functional language systems would require a lot of user expertise, implying that all users of the system would have to be retrained. For these reasons, it was decided not to include any of these ideas in the suggested system.

This leaves us with the third type of concurrency, which is at the task level. Computer networks are not considered here, since they are really an aggregate collection of computers, and not a single system. Distributed computer systems, which are considered next, offer many advantages. The predominant advantage is that they are modular, therefore they facilate the design of reliable systems. Furthermore they are highly hardware oriented, hence the modules can be designed using the present inexpensive LSI technology. They however require complex control, but by using a good design strategy, this control can be distributed and implemented in hardware modules.

The next class of techniques considered the reduction of the software overhead. As mentioned when discussing distributed systems, special hardware units can be used whenever possible to implement the required control functions. Also, by designing the system keeping in mind the basic communication needs, many of the information transfer bottlenecks and control problems can be greatly minimized. Furthermore the complexity of the system software can be reduced significantly by supporting directly the data and control structures of present day high-level languages. This would produce a more user oriented system. Another point worth mentionning, is that with present inexpensive hardware, the system should not emphasize hardware efficiency, but ease of use and simplicity of the system software.

With respect to reliability, distributed systems seem to have the best characteristics, since it was shown that they provide a fail-soft mode of operation (AVIZ 78, KRIE 79). This implies that the tasks of a failed unit can be taken over by other units. Forced no response techniques can also be added easily to a distributed system.

From the above, it is obvious that to design a reliable and efficient general purpose computer, one must use a distributed system. Furthermore, because of the low cost of present day hardware, control should be implemented as much as possible in hardware and the design should be oriented towards control simplicity and not the efficient use of the hardware. Finally the data and control structure of present day high-level languages should be supported directly, in order to reduce the software overhead and to produce a more user oriented system.

3.3 VIRTUAL LANGUAGE

The language which the proposed distributed system supports directly is described in this section. This language is called the virtual language, which is defined has a general plan used to represent the data, the operator, and the control entities that make up any programming language. Therefore, many levels of virtual languages can be defined, each depending upon the complexity of the data, the opera-

tor, and the control entities used. In this thesis a relatively simple virtual language will be defined since only the feasibility aspect of the system is considered.

3.3.1 Elements of the virtual language

In more detail, the major aspect of the virtual language used, is that it should be related to present day high-level languages. This is in terms of the lexical, syntactical and semantic properties. Furthermore, the language should have a well defined clean structure, so that direct execution can be achieved. Such a structure would also contribute to the production of reliable programs.

Furthermore, the language must be flexible and extensible, so that its power can be easily increased. It should be flexible enough so that new data structures, new operators, and new control entities can be easily defined. This would help to assure that the system could be more closely related to any high-level language. Since the hardware organization is based on this language, this extensibility will be reflected in the hardware.

As such, the virtual language proposed is based on the major features of present day block-structured languages. Block-structured high-level languages can be defined by the tree diagram depicted in figure 17 .

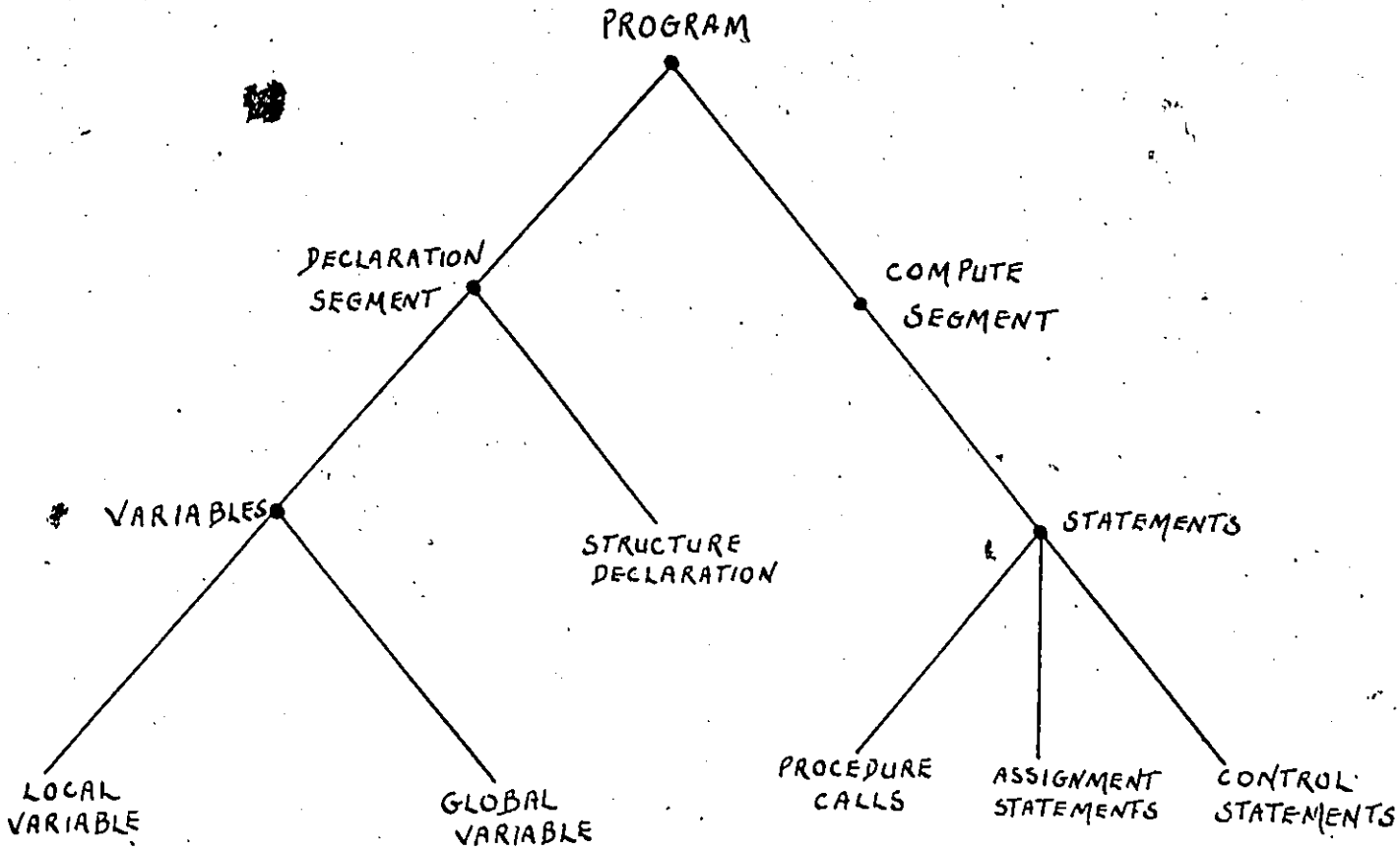


Figure 17: Block-structured language tree representation

From this tree representation, a program therefore consists of two parts: a part in which all the data and their characteristics are specified, and a part in which one specifies the computational sequences. The notion of block structure is implied by using global and local variables. The scope of a variable in a block-structured language, depends on where it is declared local. For example variable "a" in the following figure is local to segment A and global to segment B and C, while variable "b" is local to segment B and global to segment C. This variable is undefined in segment A.

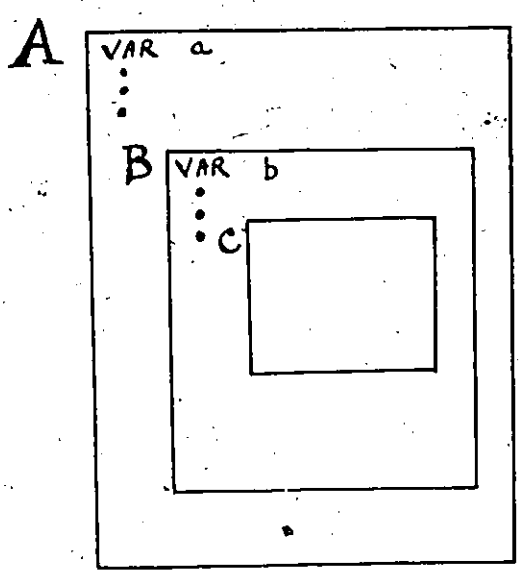


Figure 18: Scope of a variable

It should be noted that we consider blocks to be nested, therefore no overlapping of blocks is allowed. Overlapping of blocks can occur if unrestricted use of goto statements is allowed.

For the description of this virtual language the data, the operator and the control entities will now be considered.

3.3.2 Data entities

The data in present computing systems can be defined according to three levels (SHAN 80). At the hardware level the data structures that are supported directly by the machine are encountered. For example this consist of bits, bytes, fixed and floating-point numbers etc. The second

level is known as the programming level, where more complex data structures are supported by the language. For example arrays, integers, files, records, etc. are supported, in a programming language such as Pascal. These structures are built from the structures available at the hardware level. At the application level, still more complex structures can be defined by the user for some special applications. These structures are themselves defined according to those supported at the programming level.

The actual interface between the three levels is arbitrary and usually depends upon the properties of the language. For example, an assembly language supports only the structures that are directly supported by the machine. Higher level structures must be defined by the user.

The virtual language to be defined should have the capability of expressing a number of structures in an efficient manner. Furthermore, since the hardware system is based on this language we would actually be left with only two levels. The hardware and programming language levels would be the same.

To effectively support complex data structures (queues, trees, graphs, lists, etc.) requires that all operations on the structure and on the elements of the structure be defined. This is beyond the scope of this thesis, however by examining some structures we can see that all operations on

the elements depend on the type of the elements, while operations on the structure can be represented by function and procedure calls. For example, a stack structure requires the following structure operations: Push and Pop. These operations can be considered as procedure calls, and could be implemented in hardware, in a more advanced system.

In order to simplify the overall design, in this thesis only simple data types will be considered. As a first approximation, integer, real and boolean will be supported.

3.3.3 Operators and expressions

Operators that can be defined in a language are those that operate on the elements of a structure and those that manipulate the structure. For example integer addition operates on two integers regardless of their position in a structure. Operations on a structure, such as pop, push, maximum, or sort can be considered as functions or procedures.

In the proposed design all algebraic and boolean operations will be allowed. These operations are performed with respect to the data types described in the previous section.

The operators and data entities lead to the construction of expressions, which can then be used to define the assignment statement.

3.3.4 Control

The control entities of a language are used to dictate the flow of execution of the assignment statements in that language. In traditional systems, the flow of control can be represented by flow diagrams, which make use of the blocks, shown in figure 19 .

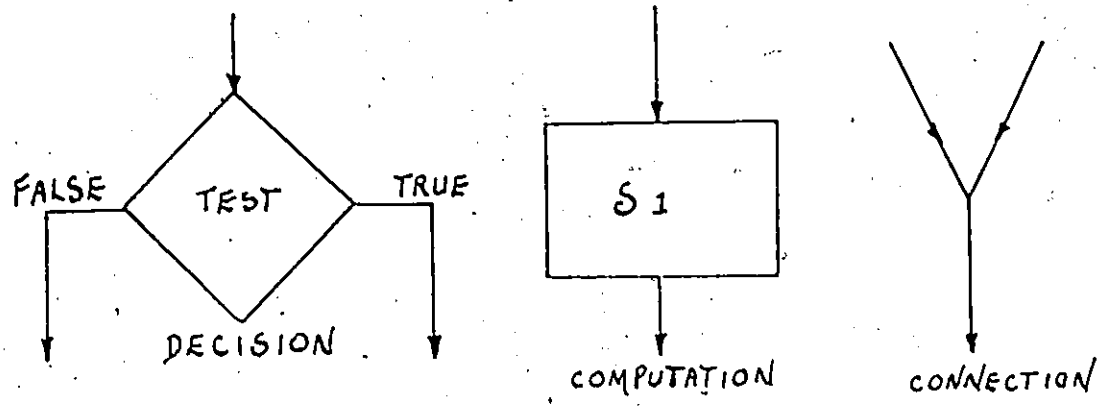


Figure 19: Flow diagram block

The use of these blocks in an unrestricted fashion causes a lot of algorithm to be obscure and unreadable. However, it has been shown (BOHM 66) that the basic constructs: sequence, iteration, and selection, were a sufficient set for representing any algorithm. The use of such a set makes the representation of an algorithm easier to understand, which therefore produces reliable programs. These constructs offer these advantages, since they have a single entry point and a single exit point, they lead to well delimited programs.

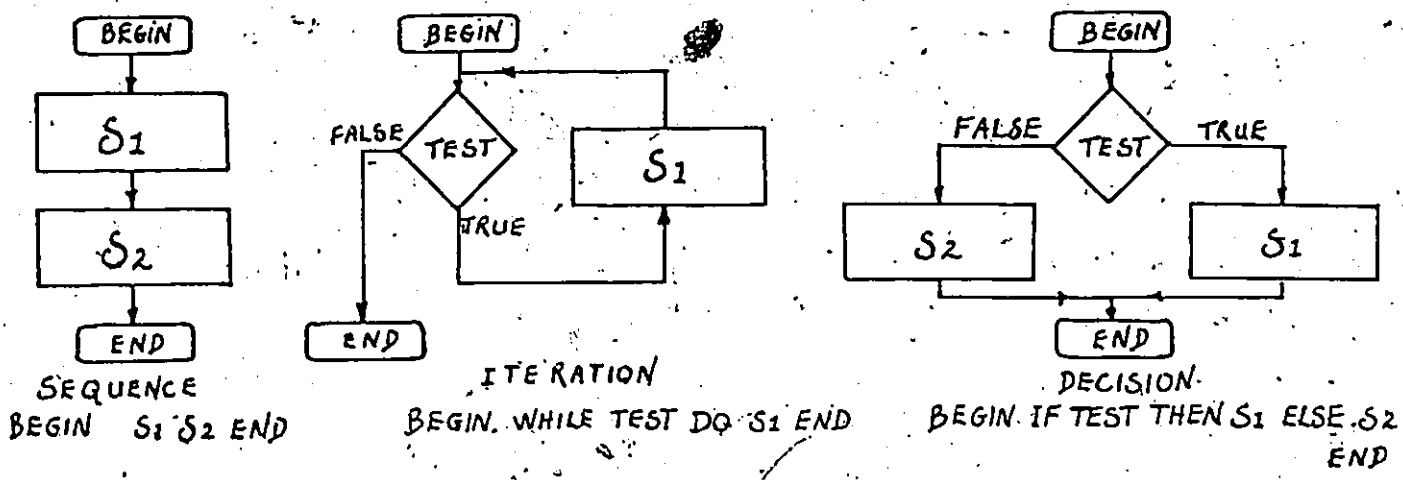


Figure 20: Basic structured control constructs

These three constructs are the basis of structured programming and are the only constructs supported in this virtual language. Structured programming also implies a top down design approach, where one specifies the general aspects of the solution and then refines them in greater and greater details until a form which can be easily coded in a particular computer language is attained. However, this is a programming method and is not reflected in the language.

3.3.5 Language definition

The actual language based on the previously mentioned data, operator, and control entities is described in terms of Backus Naur Form (B.N.F.) notation. This notation uses the following symbols to represent the various entities in a clear and concise fashion.

<xxx> - Denotes a non-terminal symbol that has to be defined elsewhere.

AAA - Denotes a terminal symbol, an atomic element of the language.

::= - Means that the non-terminal symbol on its left can be replaced by what is on its right.

| - Means OR and is used to separate alternatives.

<x1> - Stands for x1 followed by x2, and

<x2> or <x1><x2> is used for sequencing.

The complete description of the language is given in appendix A; here only the important aspect are covered.

Based on the tree structure represented in figure 17 a program in the virtual language has the following form.

<program> ::= <declaration segment>

BEGIN

<compute segment>

END

In the declaration segment, the variables and their characteristics are defined. This segment will include:

1. The name of the program.
2. The variables used in the program. They include the global variables used to interface with the external world, and the local variables used only internally by the program.

3. The characteristics of the variables. This includes the type and structure declarations.

The declare segment, using B.N.F. notation, can be divided into its constituents, as shown below.

<declare segment> ::= <name>(<global variables>)
<local variables>
<type declaration>
<structure declaration>

The compute segment which represents the computational sequence is made up of a sequence of statements, as shown below.

<compute segment> ::= 0|
<statement>
<compute segment>

The statements, as shown in figure 17, can be of three types: assignment, control, and procedure call.

<statement> ::= <assignment statement>|
<control statement>|
<procedure call>

In general the assignment statement is used to link a variable to an expression, where the variables can be of simple types or complex structures.

<assignment statement> ::= <variable type>
<assignment operator>
<expression type>

The type or structure of the variables allowed in an assignment statement, will define the complexity of the executor, since these statements must be supported directly by the hardware system. In this thesis we only consider algebraic and logic expressions operating on simple types, to limit the complexity of the executors.

The control statements are based on the decision and iteration constructs used in structured programming. Sequencing is achieved by ordering the statements in a sequential manner. Other constructs can be easily added if one wishes to relate more closely the virtual language to some procedure oriented language.

Procedures are in a sense used as macro operations, where in a program, a complex operation is replaced by a procedure call which makes the program easier to understand. Furthermore, the software system as a whole is more reliable, since the program is modular and each module can be easily tested and upgraded separately.

The language described in this section will be used to define the structure of the system.

3.3.6 Example of a virtual language program

In this section an example of a virtual language program is given. This program accepts a positive integer value and calculates its factorial using a recursive algorithm.

```
Factorial;
number, result;
(integer : number, result)

BEGIN
   read(number);
   IF number = 0 THEN
      result = 1;
      ELSE fac(number;;result);
   ENDIF
   write(result);
END


fac(input;;output);
temp;
(integer : input,output,temp)

BEGIN
   IF input > 1 THEN
      temp = input - 1;
      fac(temp;;output);
   ELSE output = 1;
   ENDIF
   output = output * input;
END
```

From this example, the syntax of the virtual language can be seen to be similar to present day structured high-level languages (such as Pascal). Structured languages facilitate the understanding of complex algorithms by segmenting the problem into simple modules. It should be understood that the clarity of the algorithm representation depends on whether or not the programmer uses a good design approach, such as a top down approach.

3.4 SYSTEM ORGANIZATION

There are a number of ways to implement a virtual language system. Since one of our goals is to implement a simple expandable system, the task driven architecture was selected. This is a modular top down design, in which each unit can be developed and upgraded separately. Furthermore, the overall control can be done in hardware (FATH 81a).

3.4.1 Task driven system

A distributed system must support the execution of many tasks on the system at the same time. In general, a task is considered to be the smallest logical entity in the system. Task assignment and supervision must be done in a reliable and well coordinated manner. In a task driven system (KRIE 79), the different tasks can be executed concurrently on dedicated executors. Figure 21 illustrates a possible architecture for a task driven multi-processor system.

Two major units are present in the system, the task allocator and the task executor. The task allocator recognizes the various tasks and routes them to a particular task executor. Since each task executor is implemented separately, they can be modified or subdivided independently. Furthermore, a fail-soft mode of operation can be achieved, if all tasks can be executed on more than one executor. However the task allocator must be either duplicated or designed in a fault tolerant fashion since it is a single unit.

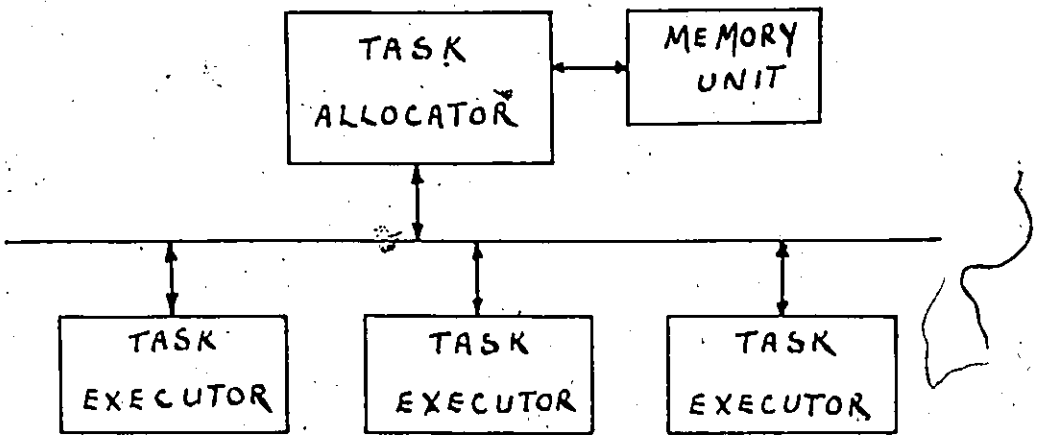


Figure 21: Task driven system

The assignment of the tasks, to the various executors is done by the task allocator by considering the dependency between the tasks. The various dependencies between the tasks are shown in figure 22 . This indicates that some tasks can be started at the same time, while others have to wait until other tasks are partially executed, or completely executed. Dependencies between tasks can be classified into three types: procedural, operational and data. Procedural dependency occurs when the execution of a task specifies the subsequent task to be executed. Operation dependency occurs when the same resource is used simultaneously by a number of different tasks. Data dependencies arise when the execution of a task will affect the source data of another task.

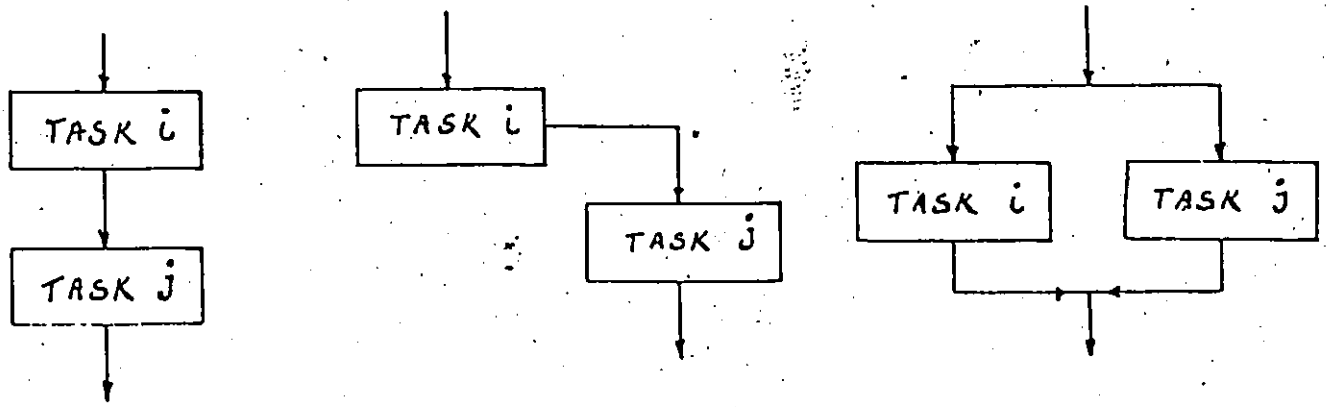


Figure 22: Task dependence

3.4.2 A virtual language system

The previous section discussed the basic concepts of the task driven system. Since the purpose of this thesis is to design a virtual language executor based on these concepts, a task can be defined as a statement in a virtual language program. Therefore, one can refer to the task executors as statement executors, and the task allocator as a statement sequencer. In this case the dependencies will mostly be due to the data dependencies between the statements. The function of the statement sequencer is to scan a virtual language program segment, and route the statements to be executed to the proper statement executors. These statements are therefore executed in a concurrent fashion.

Usually, a process is defined as being composed of many tasks, therefore a process is considered as a virtual lan-

quage program and all the information required to properly execute it. This information includes the system and user procedures called by this program. In other words, a process is a virtual language program, which is composed of many program segments, which are themselves composed of many tasks (statements), as shown in figure 23.

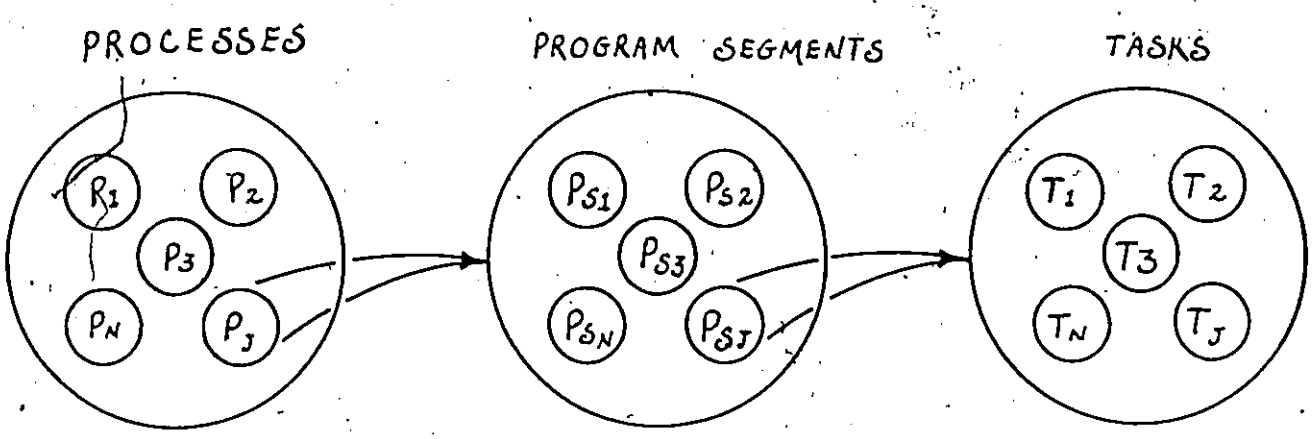


Figure 23: Task/process relation

Previously, it was mentioned that the statements could be executed concurrently by having many statement executors. This can also be accomplished with the program segments, by having many statement sequencers to execute concurrently the program segments of a process. In this case a program manager would be required to keep track of the execution of the various segments.

Furthermore, such a computing system would likely have many processes waiting for execution at one time. Supporting concurrency at the process level would greatly enhance

the capability of the system. This can be accomplished in a fashion similar to the statements and program segments, by using a system manager to assign the processes to the program managers.

The virtual language system described above, is a three level task driven system, which can be represented graphically by a tree, as shown in figure 24 . The concurrency in this system can occur at three levels: at the statement level, at the program segment level, and at the program level.

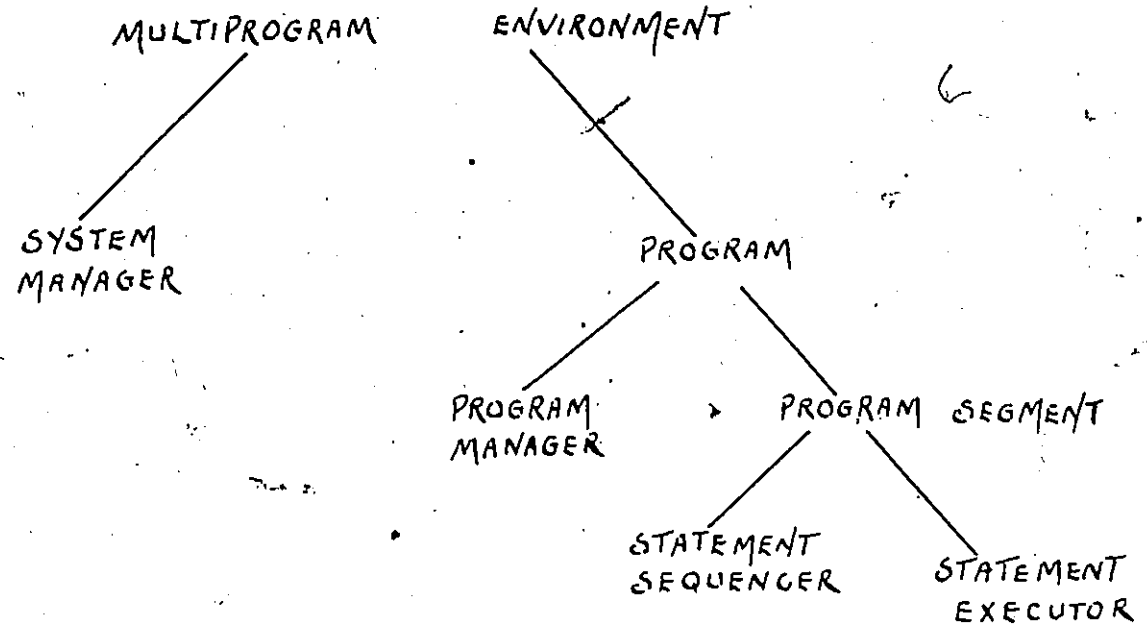


Figure 24: A virtual language system

3.4.3 The elements of the distributed system

If the conceptual three level system introduced above is considered in more detail, it can be observed that at the

program segment level, to have more than one segment executing at the same time, requires special instructions such as FORK (COBEGIN) and JOIN (COEND), or special preprocessing to partition the program into independent segments. However, the use of special instructions requires a lot of user expertise, while special preprocessing requires complex algorithms. Once these independent program segments are identified, conceptually there is no difference between a program manager and a system manager. For these reasons, concurency at the program segment level will not be considered in the following discussion. At the program level, multiple-programs can be executed at the same time. Usually little communication is required between two concurrently executing programs. The third level is at the statement level. This type of concurrency depends on the data shared between the statements. Independent statements are difficult to define due to the sequential nature of the virtual language, and to the use of variables. Data dependence is easier to define if a functional language is used, however these languages are not oriented towards our manner of thinking and programming.

The system suggested therefore consists of a two level task driven system, where at one level a system manager keeps track of the process being executed, while at the other level a task manager is used to scan a virtual language program to locate the tasks (statements) and send them to

the proper task executor. The following figure illustrates the organization of such a system.

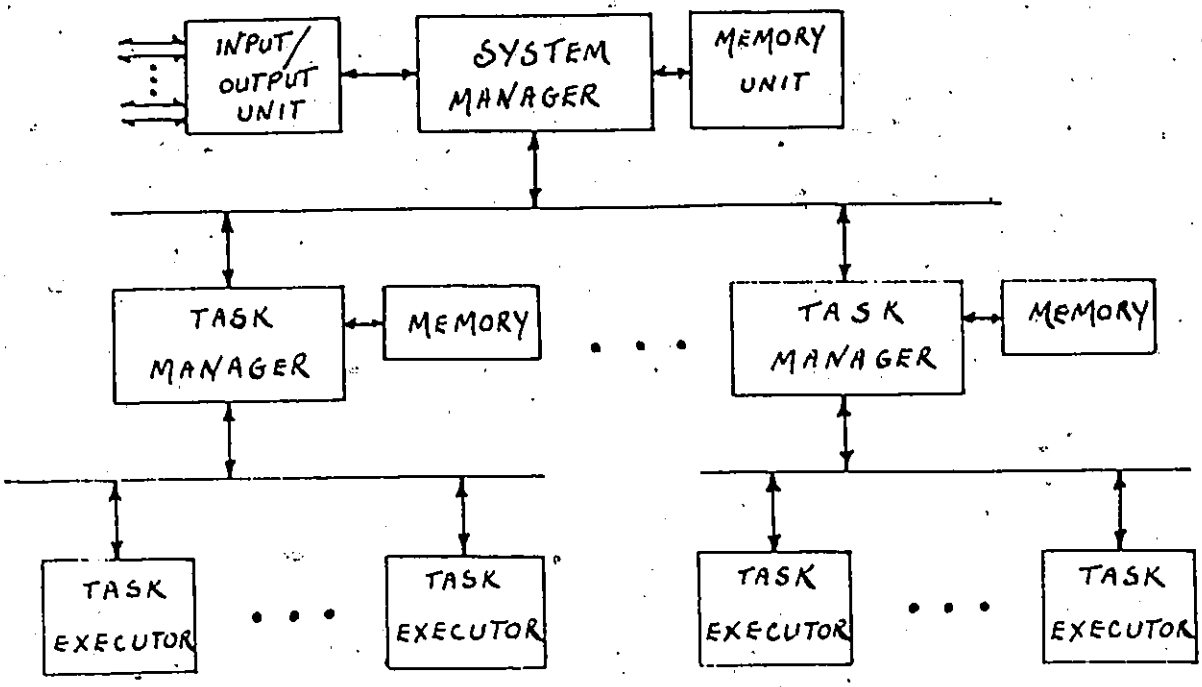


Figure 25: Proposed system

In this system, the task executor can be general purpose or special purpose, this mostly depends on the application. For example, one executor could be oriented toward performing fourier transforms. Furthermore, depending upon the data structures supported by the machine, the executor can be quite complex (i.e. executing vector operations) or relatively simple (i.e. executing scalar operations). In general, the executors would be simple, and would perform all algebraic and logic operations. However if an application

requires a special unit (fourier transform, vector addition etc.), it could be easily added to the system as a special purpose executor without affecting the performance of the entire system.

In a similar fashion, the task managers can be special purpose or general purpose, depending on the control instructions that they support. For example, one task manager could be oriented towards controlling symbol manipulation operations, while another would be better at controlling number crunching operations.

If the execution of a string of processes on this system is examined, it can be clearly seen that in some cases a task manager will require many task executors, while in others only a few will be required. Therefore, in some instances there will exist a surplus of task executors while in others there will be a shortage. Furthermore in the present system, the failure of a single unit can render a number of units unoperational. The performance and reliability of the system can be increased if the task executors are shared between the task managers. The system can thus be reorganized as shown in figure 26 .

The system manager is responsible for assigning the different processes to the task managers and for assigning the task executors to the requesting task manager. Therefore a task manager must request from the system manager the

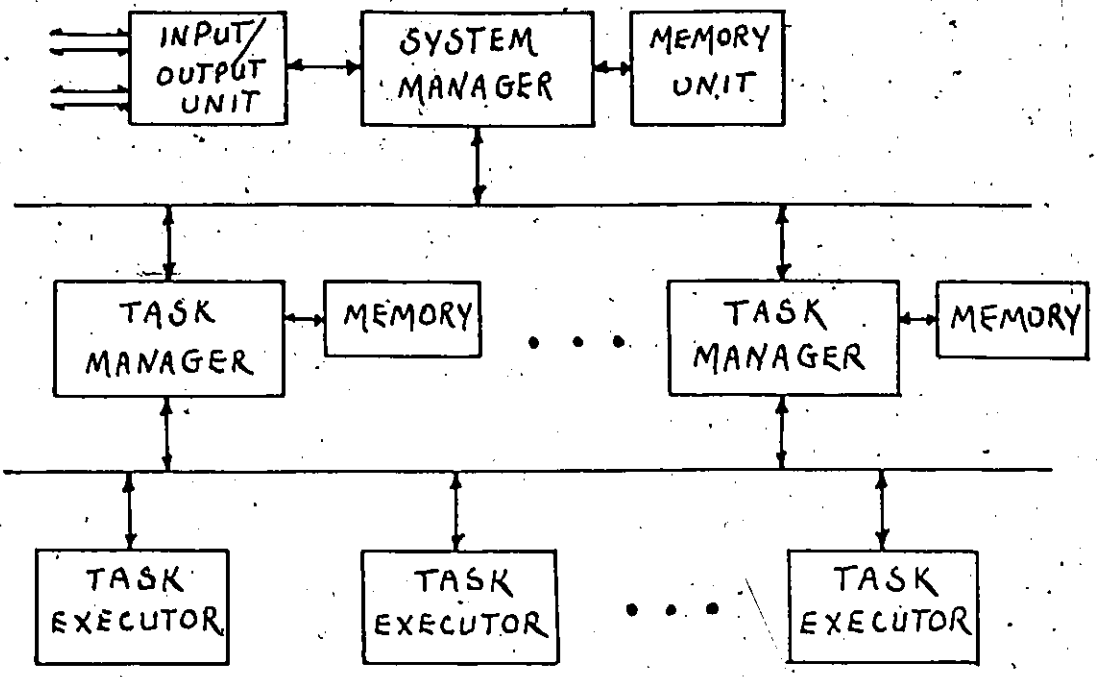


Figure 26: Sharing task executor

executors that it requires for the execution of a process. To reduce the bottleneck that is sure to be encountered every time a task manager requests a task executor, one general purpose executor should be assigned to a task manager on initiation of a process. If multiple executors are requested (n), then n-1 will have to be requested through the system manager.

In the above figure (fig. 26), the memory modules are still dedicated to each task manager, however to increase the reliability and flexibility, a pool of memory modules can be shared between the task managers. Furthermore to reduce the communication between the task managers and the

task executors, the task executors can be made to access the same memory module as the task managers. In this case, all memory modules must be two port memories, where one port is connected to the interconnection network with the task managers, and the other is connected to the network with the task executors. A process would then be assigned a memory module which would be assigned to a task manager, which will in turn be assigned task executors upon request. The management of the memory modules is centralized and controlled by the system manager. The overall system is shown in figure 27 . In this figure, the interconnection networks are represented as time shared busses, however a more complex interconnection scheme can be used if a bottleneck is observed.

This system as a whole offers: reliability, since most unit have spares, and modularity and flexibility since memory modules, task managers and task executors can be easily added to enhance the performance. This system is hierarchically controlled since at the highest level there is a single master, the system manager. However, this is not a rigid hierarchy, since once a task manager is initiated, it then operates on its own. Furthermore, the system manager can itself be a task manager that is controlling the complete system via the interconnection network. This would further increase the reliability, since any task manager could assume the role of the system manager.

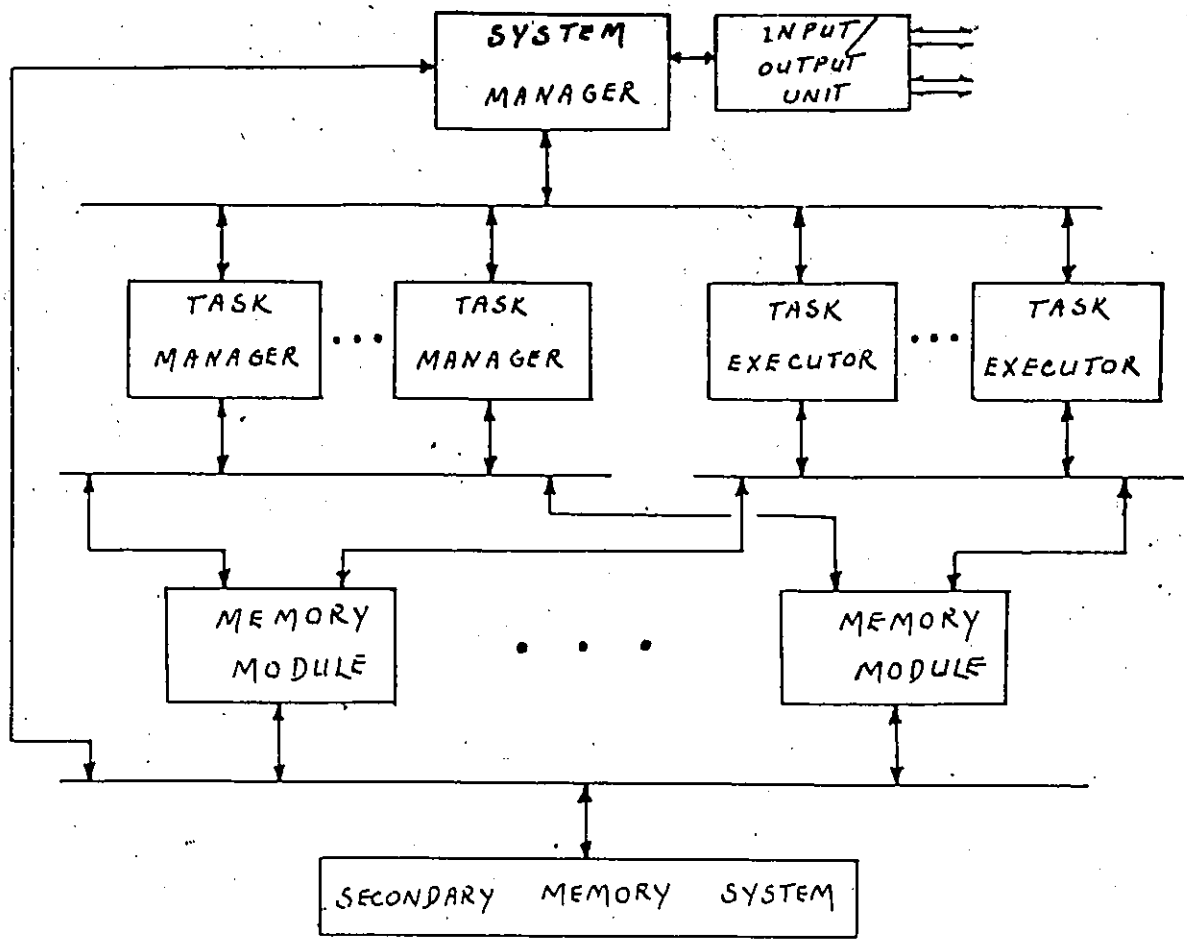


Figure 27: A distributed general purpose computer system

3.5 SYSTEM OPERATION

A brief description of the operation of the system was given when the system's elements were described in the previous section. In this section, the general operation of the system will be analysed, first by considering a minimal system, then by analysing a more complex system.

3.5.1 Minimum system

The minimum requirements for the execution of a process is the availability of a memory module, a task manager and a task executor. The actual execution of a process on such a system is as explained below.

The secondary memory system stores the process that must be executed on the system. If the memory module is free, then the system manager will load it with a process from the secondary memory. Once loaded, the task manager will take control of the memory module and start execution. The process, which is a virtual language program, is scanned by the task manager, and the statements to be executed are sent to the task executor.

The task manager will execute procedure calls and control statements, while the task executor will execute the assignment statements and logic check expressions of the virtual language program. For logic check expressions, to select the next statement to execute, the task executor will return a true or false condition to the task manager.

The execution stops when the END control word of the program is encountered.

3.5.2 Expanded system

In a more advanced system, many task managers, memory modules, and task executors would be supported by the system manager. This implies that many processes and tasks would be executed simultaneously, requiring a more complex control strategy.

In such a system, the processes to be executed would be in the secondary memory, along with the system procedures and functions that are requested by the processes. However, some system procedures can be available as special task executors. For example, a task executor can perform fourier transforms. For such task executors, when its corresponding procedure or function call is encountered in the process, the task manager will request this unit. For reliability purposes, it should be possible, that if such a unit fails another unit can take up its functions using a software routine instead of a hardware routine.

The system manager keeps track of which processes to execute and which are executing. In a sense, it acts as an interface between the processes and the bare machine. The main purpose of the system manager is to allow several processes to utilize the system in an efficient manner. As described in more detail in FATH 81a and FATH 82b, the system manager can be implemented as a table driven system. When a memory module is free the system manager loads it with a

process awaiting execution. The memory module is then assigned to a task manager when one is available. This task manager which should have a task executor assigned to it on initiation, will scan the process in its associated memory module to locate executable expressions.

Other task executor can be requested by examining the dependency between the expressions, thus obtaining concurrency between the expressions. (see chapter 6).

A process is terminated when the program's END control word is encountered. At this point the proper information is stored in the secondary memory and the units are freed to execute other processes.

3.6 CONCLUSION

This chapter presented the overall organization and operation of a distributed system that supports directly the entities found in present day high-level languages. In considering the feasibility of implementing the above suggested system three main aspects must be investigated: the overall system control, the execution of the high-level language process, and the memory management aspect.

With respect to the overall system control, the investigation in FATH 81a indicated that considering present day technology, the required control actions could be implemented in hardware.

With respect to the execution of high-level language processes, the design of the task manager and the task executor will be analysed in some detail in the following chapter. The other units of the system will only be examined when necessary.

The memory management aspect of such a distributed system is a major research topic of its own and is being investigated in parallel. To keep in line with the goal of minimizing the system software, the memory management function should use hardware that directly assists in the storage, retrieval and management of the data. A hardware system that directly supports these functions is described in DeMA 76.

Chapter IV

IMPLEMENTATION DETAILS

4.1 INTRODUCTION

Chapter three described a two level task driven computing system that is oriented toward the execution of virtual language processes. It was shown that the control of the execution of a process is performed by the task manager, while the actual execution of the various tasks in this process is performed by the task executors. These two actions although separated in the system, are linked in the virtual language program representation. A more direct execution would be achieved if these actions were also separated in the language used.

Furthermore in chapter 2 it was shown that the direct execution of a high-level language program can be quite inefficient, since the branch addresses required in a control statement are not present. Without these branch addresses, to locate a jump location requires that the program be completely scanned. This could be overcome by modifying the language so that branch addresses are inserted after the proper control words.

Another improvement, would be to speed-up the execution of the various expressions. Since only simple data types are considered, execution would be simplified if these expressions were written in a reverse polish notation (suffix notation).

The language that is directly supported by the system will support these enhancements. This language is called the execution language, and is a preprocessed version of the virtual language. Its syntax and semantic properties follow closely those of the virtual language. The execution of a virtual language process on this system is shown in figure 28 .

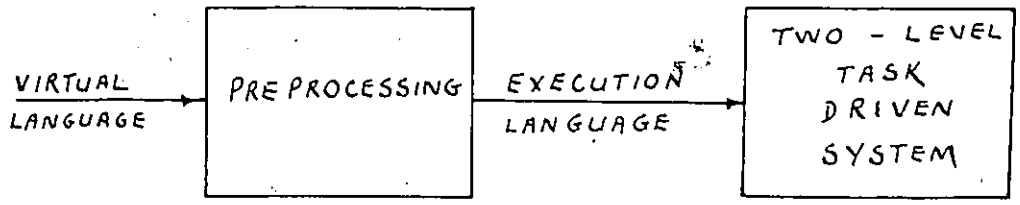


Figure 28: Virtual language processes execution

It should be stated, that another approach would have been to start immediately with the execution language, but this would have been contrary to the stated goal, that the language should be related to present day high-level languages. Because of the simplicity of the preprocessing, it is felt that this approach represent a better compromise

between using a high-level language, and simplicity of execution.

In this chapter the execution language is first described in detail, followed by the analysis of the processing of an execution language process. An execution language process is the preprocessed version of a virtual language process, therefore it contains a main execution language program and all the called execution language procedures. Finally the hardware requirements of the task manager and task executor that support directly the execution language, will be briefly analysed.

4.2 CHARATERISTICS OF THE EXECUTION LANGUAGE

The execution language, to be oriented toward the system, will contain three segments. A control segment used for proper sequencing of the various expressions is followed by a statement segment that includes the expressions to be executed, which is followed by a data segment that contains the characteristics of the variables and their values. Figure 29 illustrates the three execution language segments required for every program, and procedure.

As indicated in the previous chapter a virtual language program has the following syntax.

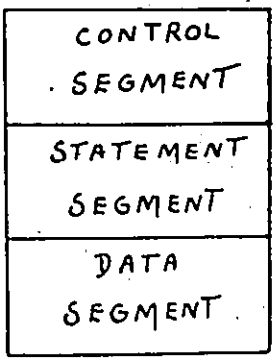


Figure 29: Map of an execution language program

```
<virtual language program> ::= <declare segment>
                               BEGIN
                                  <compute segment>
                               END
```

Separating the control of the process from the execution of the tasks requires the following changes to the virtual language representation.

```
BEGIN                          <control segment>
   <compute segment>
END                            <statement segment>
```

Here the control segment drives the task manager, while the statement segment includes the statements to be executed and information about these statements. It should be noted here that the goal is to obtain a representation that will retain all of the information of the virtual language program yet be more efficient to execute.

Furthermore, the declare segment is used to build the data segment which holds the information about all variables used in the program plus their actual value.

<declare segment> <data segment>

Therefore a program in virtual language is transformed into three segments as shown below.

<virtual language program> <E.L. program>|
<control segment>
<statement segment>
<data segment>

A procedure written in virtual language will be transformed in the same fashion.

4.3 ELEMENTS OF THE EXECUTION LANGUAGE

The different segments of the execution language are now described in some detail in the following subsections, keeping in mind the arguments presented above.

4.3.1 Control segment

In order to seperate the control of the process from the execution of the tasks, all assignment statements, logic check expressions and procedure calls in the compute segment

of a virtual language program or procedure are replaced by descriptors. These descriptors indicate which statements, in the statement segment, to execute. For explanation purposes, these descriptors are numbers representing the ordering of the statements. Separating the control and execution steps, removes the necessity of performing the step of lexical and syntactical analysis for every segment of code during execution, since all statements are no longer intermixed.

The control segment includes two sections: a parameter section to specify the global variables accessed, and a control string that dictates the flow of control as determined by the control constructs.

<control segment> ::= <parameter section>
<control string>

4.3.1.1 Parameter section

The control segment contains a parameter section if the program or procedure accesses any global variable. This parameter section holds the list of the formal parameters, and is placed before the BEGIN control word, as shown below. This section is required so that the formal parameters of the called procedure can be easily related to the actual parameters of the calling program or procedure.

1 PARA
<formal parameters>

```
1 BEGIN
....
```

The three constructs that form the basis of structured programming, hence that are incorporated in the virtual language, are: sequence, decision, and iteration. The transformation of these three constructs is now analysed.

4.3.1.2 Sequence

A sequence of statements in a virtual language program will be transformed into a sequence of descriptors, where each descriptor will point to the statement it replaces. The following example clarifies this point.

Given the virtual language program.

```
BEGIN
   A = A + B;
   C = D;
   E = F;
END
```

The resulting execution language representation is

```
BEGIN
{0}
{1}
{2}
END
```

Where descriptor {0}, {1} and {2} point to the statements A = A + B, C = D, and E = F respectively. These statements are now located in the statement segment.

4.3.1.3 Decision

The decision construct given below, is transformed in a similar fashion except that the branch addresses are also supplied in order to increase the execution speed.

```
IF {logic check}
   THEN {compute segment}
   ELSE {compute segment}
ENDIF
```

This string will be transformed into:

```
IF
{logic expression descriptor}
<displacement to else>
{statement 1 descriptor}
....
{statement i descriptor}
ELSE
<displacement to endif>
{statement i+1 descriptor}
....
ENDIF
```

where { } indicates the descriptors pointing to a statement in the statement segment, and < > indicates a displacement that is added to the program counter to execute the proper jump.

Another form of the decision construct can be encountered in which the ELSE part is missing. Its transformation is similar to the full decision construct except that the ELSE control word is not present in the string as shown below:

```
IF {logic check}
   THEN {compute segment}
ENDIF
```

is transformed into:

```
IF
{logic expression descriptor}
<displacement to endif>
{statement 1 descriptor}
....
{statement n descriptor}
ENDIF
```

4.3.1.4 Iteration

For the iteration construct the transformation is similar to those above. The required branch addresses is also supplied.

Given the virtual language iteration construct:

```
WHILE {logic check} DO
```

{compute segment}

ENDWHILE

its resulting execution language counterpart is:

WHILE

{logic expression descriptor}

<displacement to endwhile>

{statement 1 descriptor}

....

{statement n descriptor}

ENDWHILE

<displacement to while>

In this case the <displacement to while> is subtracted from the program counter in order to jump back to the while control word.

4.3.1.5 The control string

Considering the above transformation, it is easy to see that when translating a virtual language program into the execution language, one obtains a control string that contains the following elements.

- control words

  BEGIN, END, IF, ELSE, ENDIF, WHILE, ENDWHILE

- displacements

  <xxx>

- descriptors

  {xxx}

To simplify control, the words in the control string are tagged. A 1 is appended to control words, while a 0 is appended to displacements and descriptors. This tag also assures that a descriptor or a displacement will never be interpreted has a control word. Such a problem is common in most present day computers since there exists no way of differentiating between data and instructions. Note that there is no need to distinguish between displacements and descriptors, since displacements can occur only at specific locations.

The following is an example of the transformation of the compute segment of a virtual language program into the control string of an execution language program.

```
BEGIN                                1 BEGIN
   READ(number);                     0 {0}
   IF number = 0                     1 IF
      THEN result = 1;               0 {1}
      ELSE FAC(number;;result);      0 <1>
   ENDIF                             0 {2}
   WRITE(result);                    1 ELSE
END                                  0 <1>
                                     0 {3}
                                     1 ENDIF
                                     0 {4}
                                     1 END
```

The logic check expression, the assignment statement, and the three procedure calls present in this virtual language representation, are replaced by descriptor {0} to {4}, in the execution language representation. In this example three procedure calls were encountered. Two of these were system calls (READ and WRITE), while the other one was user defined. In this thesis only the READ and WRITE system procedures are considered. These two procedures have a single parameter, and are used to input and ouput values to or from a program or procedure.

This is a simple example to indicate the format of the control string. As will be shown later, the programs can include nested structures.

4.3.1.6 Control words

A word in the control segment is identified as a control word if its associated tag bit is one. The task manager will scan the control segment, and perform the action required by the various control words. Based on the use of the control constructs (sequence, decision and iteration), the following eight control words can be encountered.

PARA:

Control word PARA indicates the start of the parameter section in the control segment. This section holds the list

of the global variables accessed by this program. This section is required to properly access the global variables.

BEGIN:

This control word indicates the beginning of the control string of an execution language program. An error can be flagged if this control word is not encountered after the parameter section.

END:

Indicates the end of a program. When this control word is reached we must either return to the calling program or procedure, since it indicates the end of a procedure, or stop execution since it can also indicate the end of an execution language program. In this case the task manager will inform the system manager that the process is completed.

Previously, we mentioned that nested structures can be included in a program. To coordinate the flow of control in nested structures, a single flag will be used to record the status obtained from the execution of logic check expressions.

IF:

When this control word is encountered, the following actions are initiated:

1. Send information about the logic check expression pointed to by the descriptor following the IF control word to the task executor so that it can be executed.
2. Wait for the logic condition returned by the task executor, after the execution of the logic check expression. This condition is used to set or reset the status flag, since the condition obtained will be needed later on by the ELSE control word. The status flag is set if the logic condition is false.
3. If the status flag is reset, then skip the displacement <else> or <endif> and continue from there. In this case the THEN segment is executed.
4. If the status flag is set, add displacement <else> or <endif> to the program counter. The addition of this displacement can transfer control to either a control word such as ELSE or ENDIF, or to the first descriptor following these control words. To add error checking capability to the system, the control will always jump to a control word.

ELSE:

The ELSE control word requires that the following actions be undertaken:

1. If the status flag at this instant is reset, then the displacement <endif> is added to the program counter. This implies that the THEN segment was just executed

and that the ELSE segment must be skipped. In this case the control will be transferred to the ENDIF control word.

2. If the status flag is set, then the displacement <endif> is skipped, therefore the ELSE segment is executed.

ENDIF:

This control word will reset the status flag, indicating that the present decision construct is terminated. This is necessary since nested IFs are allowed. This control word can also be used for error checking since we must land on it when adding address <endif> to the program counter.

WHILE:

This control word indicates the start of an iteration construct and requires the following actions:

1. Send information about the logic check expression pointed to by the descriptor following the WHILE to the task executor for execution. A logic check expression can only follow IF's and WHILE's control words; an error occurs if they are encountered in any other circumstances.
2. Wait for the logic condition returned by the task executor and modify the status flag accordingly. It is set if the condition is false.

3. If the status flag is reset, then the displacement <endwhile> is skipped and the execution continues from there. This implies that the statements in the loop are executed.
4. If the status flag is set, then the displacement <endwhile> is added to the program counter. This implies that the loop is terminated. An error is flagged if the jump does not transfer control to the ENDWHILE control word.

ENDWHILE:

This control word requires the following actions:

1. If the status flag is set, then skip the displacement <while> and continue. The status flag must also be reset since nested loops are allowed.
2. If the status flag is reset then substract the displacement <while> from the program counter, which will jump back to the WHILE control word.

Considering the constructs of structured programming, the use of a single status flag is sufficient to allow nesting of control constructs at any level. If one uses restricted goto statements (still structured), then this flag is still sufficient. However with unrestricted use of goto statements this flag would be inadequate.

4.3.2 Statement segment

The statement segment contains all the executable statements, that is the assignment statements, the logic check expressions, and the procedure calls that are accessed by the descriptors in the control string. To allow for simple translation, and future enhancements, the statements in the statement segment are accessed in an indirect fashion. A statement descriptor table is used, which contains information about the statements, and pointers to the statements which are placed in the statement section. Therefore the statement segment is divided into a statement descriptor table and a statement section, as shown below.

<statement segment> ::= <statement descriptor table>
<statement section>

4.3.2.1 Statement descriptor table

A descriptor in the control string points to an element in the statement descriptor table. This table specifies the class of this statement (ASSIGN, PROCESS, LOGIC) and where it is found in the statement section.

<statement desc. elements> ::= 0|
<class><location>
<statement desc. elements>

More information about a statement could be easily added to this table to increase the performance of the system. For example information about concurrent operations and special task executors required to execute a statement could be included. Furthermore, this statement descriptor table can be used to access a single copy of repeated statements.

4.3.2.2 Statement section

Before describing in details the statement section, the execution strategy is briefly outlined. In the case of assignment statements and logic check expressions, the task manager will awaken the proper executor to process the statement. With respect to procedure calls, two types can be encountered: user and system procedures. For user procedures, the task manager will save the proper information about the calling program or procedure, then take over the execution of the called procedure. In the case of system procedures, such as I/O or special functions (fourier transforms), the task manager will awaken a special task manager or executor, depending on the complexity of the procedure. The execution details of the above strategy will be described further on.

It should be noted that functions are not supported, however they could be added to the virtual language and translated to procedures during preprocessing. In this case, the

name of the function would correspond to one of the parameters passed to the procedure.

As was stated before, all expressions are translated into reverse polish notation, since this form can be easily interpreted using a stack mechanism. (The translation from infix notation to suffix notation is described in chapter 5.)

In the virtual language, all executable statements contain symbolic names representing the variables, where each variable can assume various types and structures. When executing an expression the values of these variables must be obtained and their classes must be known. The class of a variable can be one of the following four: INPUT, INOUT, OUTPUT, or LOCAL. Symbolic names are difficult to manipulate unless an associative memory is used. Since associative memories are considered to be still too cumbersome, all variables are replaced by descriptors, as was done for the statements. A descriptor for a variable will point to its value in the data segment. Constants will also be replaced by descriptors, which will point to the value of this constant in the data segment.

The statement section of the execution language has a format similar to the control string. A tag bit is used to distinguish between descriptors for data, and operators. For example, the translation of an assignment statement in

the virtual language format into the execution language format, is shown below.

Virtual language	Suffix notation	Execution language
A = B + (C - D)	BCD-+A=	0 {0} 0 {1} 0 {2} 1 - 1 + 0 {3} 1 = 1 S

where {0} is a descriptor for B, {1} for C, {2} for D, and {3} for A. Operator 'S' is used to indicate the end of a statement. Note that for assignment statements, only the right hand side (the expression) is translated into suffix notation, since it is the only part that needs computation. The left hand side indicates the location where the result has to be stored.

Virtual language procedure calls are transformed as follows, and placed in the statement section.

<name>(<procedure variables>);	1 <name> 0 {xx} .... 1 S

The name of the procedure is given first followed by the descriptors of the actual parameters. The actual parameters correspond to the variables that are passed to this procedure. Again operator 'S' is used to indicate the end of this statement. A descriptor of a variable in the procedure call statement of the calling program or procedure will correspond to a descriptor given in the parameter section of the called procedure.

4.3.3 Data segment

In the previous section it was shown that the symbolic name for variables as well as constants were replaced by descriptors in the executable statements. Here again to provide for further enhancements, such as error checking, the variables are accessed in an indirect fashion. A data descriptor table is used to give the information about a variable, and where it can be located in the data space. Therefore the data segment can be divided into a data descriptor table and a data space.

<data segment> ::= <data descriptor table>
<data space>

4.3.3.1 Data descriptor table

An entry in this table is pointed to by a descriptor in one of the statements in the statement section, or by a de-

scriptor in the parameter section of a called procedure. The data descriptor table holds the information about the variable, such as:

1. Its type (integer, real, etc.)
2. The class of this variable (input, output, const, local, inout)
3. The location of the present value of this variable

<data descriptor elements> ::= 0|
<class><type><location>
<data descriptor elements>

For debugging purposes the name of the variable can also be included in this table so that it can be printed when an error occurs, or when an input or output procedure is called.

4.3.3.2 Data space

The data space contains the actual data that is used by this segment. Of the six sections of the execution language, this is the only one that will be modified as the program is executed. The other five sections are execution invariant. When executing a process, the data space of the called procedure is appended to the one of the calling program or procedure in order to share the variables. Therefore, there exists a single data space when executing a process. How this is accomplished is described in the next section.

Every virtual language procedure and program is translated into these six sections by the preprocessing unit prior to execution. An execution language is completely relocatable since no addresses are specified, only displacements.

The following figure illustrates the map of the various sections of an execution language program.

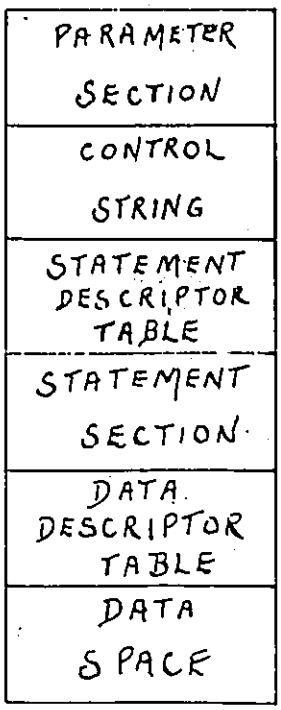


Figure 30: Detailled map of an execution language program.

4.3.4 Example of an execution language program

Using the example of section 3.2.6, of a virtual language program that calculates factorial in a recursive fashion, its equivalent execution language form is developed from the

above discussion. The following two columns represent the transformation of the main program "factorial". The left column contains its virtual language representation, while the right column lists the equivalent execution language representation.

```
Factorial;
number, result;
(integer : number, result)

BEGIN
   read(number);
   IF number = 0 THEN
      result = 1;
      ELSE fac(number;;result);
   ENDIF
   write(result);
END
```

```
1 BEGIN
0 {0}
1 IF
0 {1}
0 <1>
0 {2}
1 ELSE
0 <1>
0 {3}
1 ENDIF
0 {4}
1 END

STATEMENT DESC TABLE

PROCESS    0
LOGIC      3
ASSIGN     7
PROCESS   11
PROCESS   15

STATEMENTS

1 READ
0 {0}
1 S
0 {0}
0 {2}
1 =
1 S
0 {3}
0 {1}
1 =
1 S
1 FAC
0 {0}
0 {1}
1 S
1 WRITE
0 {1}
1 S

DATA DESC TABLE
```

```
LOCAL   INTEGER   1
LOCAL   INTEGER   2
CONST   INTEGER   0
CONST   INTEGER   1
```

In a similar fashion the transformation of procedure "fac" can be obtained. In this case, the left column represents the virtual language form, while the right one represents its equivalent execution language listing. Note that procedure "fac" has a parameter section, since it accesses global variables.

```
fac(input;;output);
temp;
(integer : input,output,temp)

BEGIN
   IF input > 1 THEN
      temp = input - 1;
      fac(temp;;output);
   ELSE output = 1;
   ENDIF
   output = output * input;
END
```

```
1 PARA
0 {0}
0 {1}
1 BEGIN
1 IF
0 {0}
0 <2>
0 {1}
0 {2}
1 ELSE
0 <1>
0 {4}
1 END

STATEMENT DESC TABLE

LOGIC     0
ASSIGN    4
PROCESS  10
ASSIGN   14
ASSIGN   18

STATEMENTS

0 {0}
0 {3}
1 >
1 S
0 {0}
0 {3}
1 -
0 {2}
1 =
1 S
1 FAC
```

```
0 {2}
0 {1}
1 S
0 {3}
0 {1}
1 =
1 S
0 {1}
0 {0}
1 *
1 =
1 S

DATA DESC TABLE

INPUT   INTEGER   1
OUTPUT  INTEGER   2
LOCAL   INTEGER   3
CONST   INTEGER   1
```

4.4 PROCESSING OF A AN EXECUTION LANGUAGE PROCESS

The representation and characteristics of the execution language was described in the previous sections. In the present section, the processing of an execution language process is discussed. This analysis is made in terms of a simulation program written in Pascal.

One of the major requirements for efficient execution lies in the way the variables are accessed in a program or procedure. The way this is achieved in this system is first analysed, followed by the overall processing of an execution language process. Finally, a rough specification of the hardware organization of the task manager and task executor is given.

4.4.1 Variable accessing

In this section, the manner in which the local and global variables are accessed during the execution of a program or procedure, is described. This will be presented in three steps by analysing the accessing of variables, first in the main program, next in a procedure called by the main program and finally in a procedure called by another procedure.

4.4.1.1 Variable accessing in the main program

In the main program of a process, all variables are considered as local. As outlined earlier, the main program has the section shown in figure 31 .

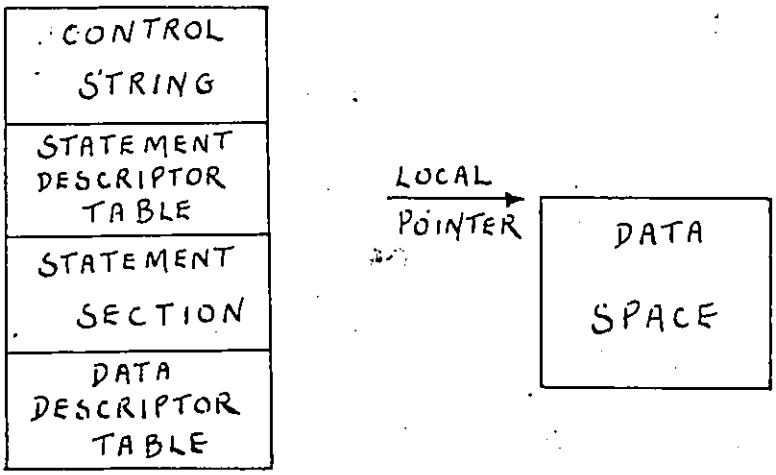


Figure 31: Main program's sections

Since the main program does not access any global variables, it does not possess a parameter section. In this figure, the data space is separated from the other sections, since it is the only time variant section of the execution language representation. To access the value of a variable that is in the data space, a local pointer points to its first element. Therefore a descriptor in a statement points to an entry in the data descriptor table which will contain, among other things, the position of the data in the data space as a relative address that is added to the local pointer. In this case the access is direct as shown in the following figure.

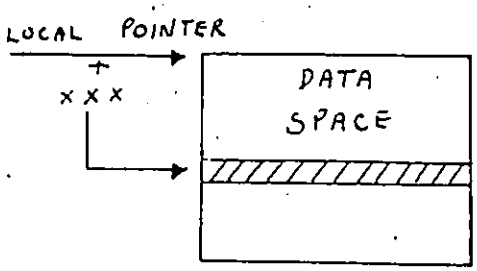


Figure 32: Direct access of variables

Relative addressing also allows easy relocation of the data space. Constants encountered in the main program can be treated in two fashions: their values can be in the data space therefore they are treated as local variables, or they can be stored directly in the data descriptor table. In

this system they are placed in the data descriptor table, since this implies that the data space of a program or procedure is empty if the program or procedure is not executing. This greatly simplifies the management of the data space.

4.4.1.2 Variable accessing in a called procedure

The accessing of variables in a procedure called by the main program is now analysed. An execution language procedure call statement is located in the statement section of the calling program. Information about this statement is stored in the statement descriptor table.

A called procedure has a parameter section, where each descriptor, corresponds to a descriptor in the procedure call statement. The descriptors in the procedure call statement are referred as the actual parameters, while those in the parameter section of the called procedure are referred as the formal parameters, as shown below. In this case, since the procedure call statement is in the main program, the actual parameters are all of class LOCAL.

```
1 <name>                    1 PARA
0 {actual parameters}        0 {formal parameters}
....                         ....
1 S                          1 BEGIN
```

calling program procedure called

The execution language representation of the main program and the called procedure are shown in the following figure.

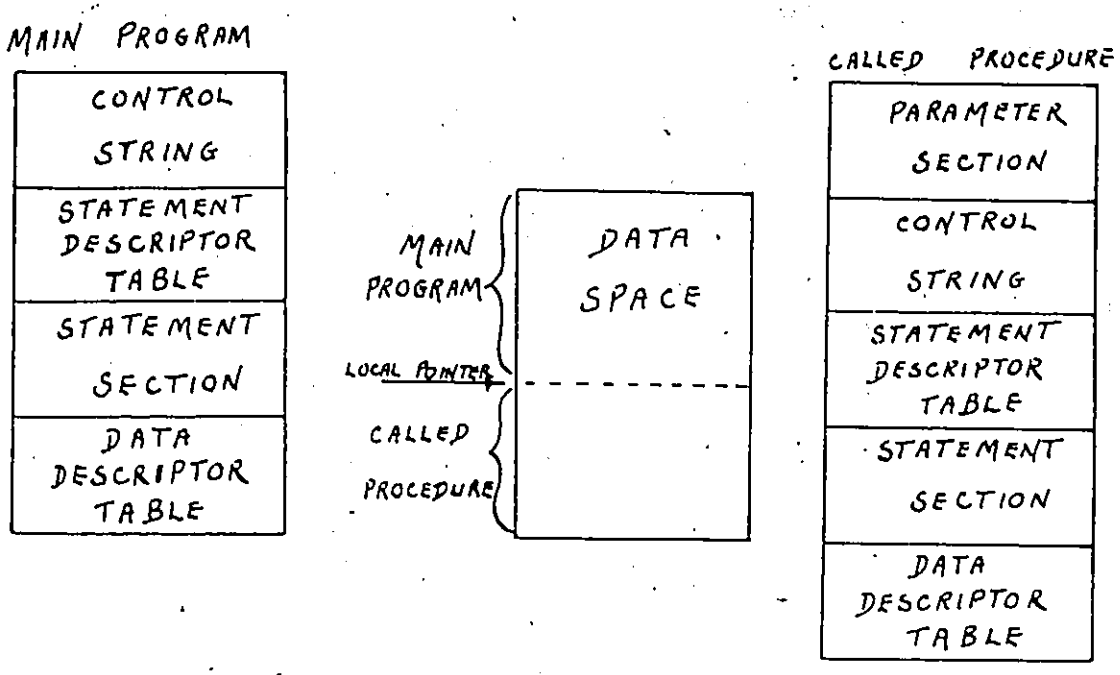


Figure 33: Main program and procedure representation

On executing a procedure call statement, the formal and actual parameters must be linked in a straightforward fashion. This is accomplished by appending the data space of the called procedure to the one of the main program, and by modifying the local pointer so that it will point to the local section of the data space used by the procedure.

Local variables in the procedure are accessed as previously mentioned. A relative address is stored in the data descriptor table which when added to the local pointer will give the address of the value of the variable.

For global variables, this is different since their values are not in the local section of the data space used by the procedure, but in the local section of the main program. Therefore, the following actions must be performed before control is passed to the procedure; if the class of the formal parameter is INPUT, then the value of the actual parameter is copied at the proper location in the local section used by the called procedure. This is necessary since the procedure must not change the value of the actual parameter, in the main program. However, if the class of the formal parameter is OUTPUT or INOUT, the address of the actual parameter in the data space is placed in the local section used by the called procedure, at the location indicated in the data descriptor table.

Once this is performed for all variables, control is passed to the procedure. INPUT variables are accessed as if they were LOCAL variables, since their values have been copied. Variables of class INOUT and OUTPUT are accessed indirectly using the addresses previously stored; that is, the relative address obtained from the data descriptor table is added to the local pointer giving the location of the address of the value of the variable. This is shown graphically below.

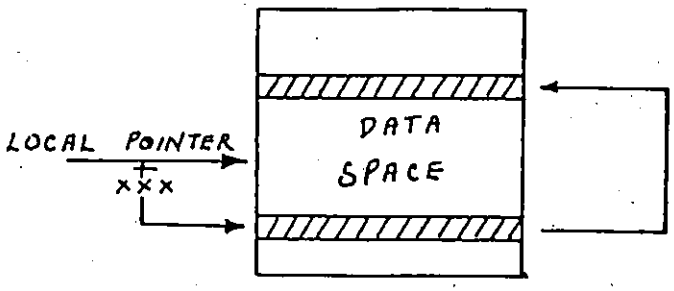


Figure 34: Indirect access of variables

4.4.1.3 Multi-level accessing scheme

The following figure, illustrates the data space after a few calls have been performed. It should be understood that when a procedure terminates, the local pointer returns to the value it possessed before the procedure call.

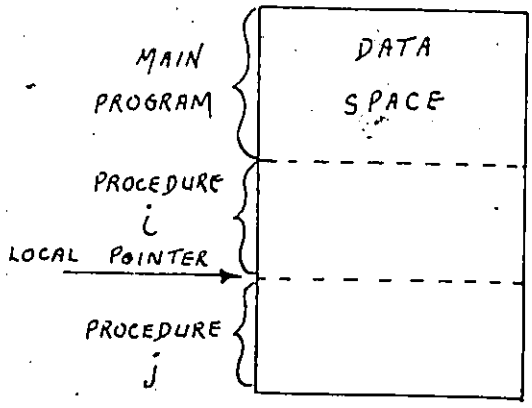


Figure 35: Data space

The proper addresses and values that must reside in the local section of an executing procedure to properly access

the global variables, are placed there when the procedure is called. For multi-level accessing, the actual parameters can be global variables, instead of only LOCAL. This implies that four different types of parameter linkages are possible, as described below.

If the class of the actual parameter is either LOCAL or INPUT, and the class of the formal parameter is INPUT then the following action is required: the value of the actual parameter is copied at the proper location in the local section of the called procedure. The way this is accomplished will be described when the hardware of the system will be presented.

Similarly, if the class of the actual parameter is OUTPUT, or INOUT, and the class of the formal parameter is INPUT, then the address of the actual parameter in the local section of the calling program or procedure, is used to obtain the value of this parameter, which is then copied in the local section of the called procedure.

If the class of the actual parameter is LOCAL or INPUT, and the class of the formal parameter is OUTPUT or INOUT, then the address of the actual parameter in the data space, is placed in the local section of the called procedure at the location indicated by the descriptor of the formal parameter.

Finally if the class of the actual parameter is OUTPUT or INOUT and the class of the formal parameters is OUTPUT or INOUT, then the location in the local section of the calling program or procedure pointed to by the descriptor of the actual parameter, contains an address pointing to the actual value of the variable. This address is simply copied to the local section of the called procedure at the appropriate location indicated by the descriptor of the formal parameter.

Once the parameters are linked, control is passed to the called procedure. In this procedure the parameters are accessed as described before. This accessing scheme can be direct or indirect, depending on the class of the formal parameter.

This method of accessing the variables requires at most two references to the data space to obtain the value of a variable, even for recursive algorithms.

4.4.2 Process execution

Having covered the mechanisms of variable accessing, the details of the processing of an execution language process can be analysed. To see the problems involved, the process execution was simulated in Pascal.

To execute a program or procedure in the execution language, the location of its various sections must be known.

In the simulation, this is accomplished by adding an information section to every execution language program and procedure. This section contains the name of the program or procedure, plus the pointers to some execution language sections, as shown below.

```
<info section> ::= <name>
                   <base pointer 1>
                   <base pointer 2>
                   <base pointer 3>
                   <local end>
```

Where <name> gives the name of the program, <base pointer 1> points to the start of the statement descriptor table, <base pointer 2> points to the first executable statement in the statement section, and <base pointer 3> points to the start of the data descriptor table. <local end> indicates the number of data elements contained in the local section of this program or procedure. This value is required to update the local pointer when a procedure is called or terminated.

Furthermore, when executing a program or procedure, a control string pointer is required to scan the control string. The relation between these pointers and the execution language sections is shown in the figure 36 .

If the execution of a process is examined, one can specify a set of modules such that each module is in charge of a portion of the execution. Roughly speaking, module 0 is in

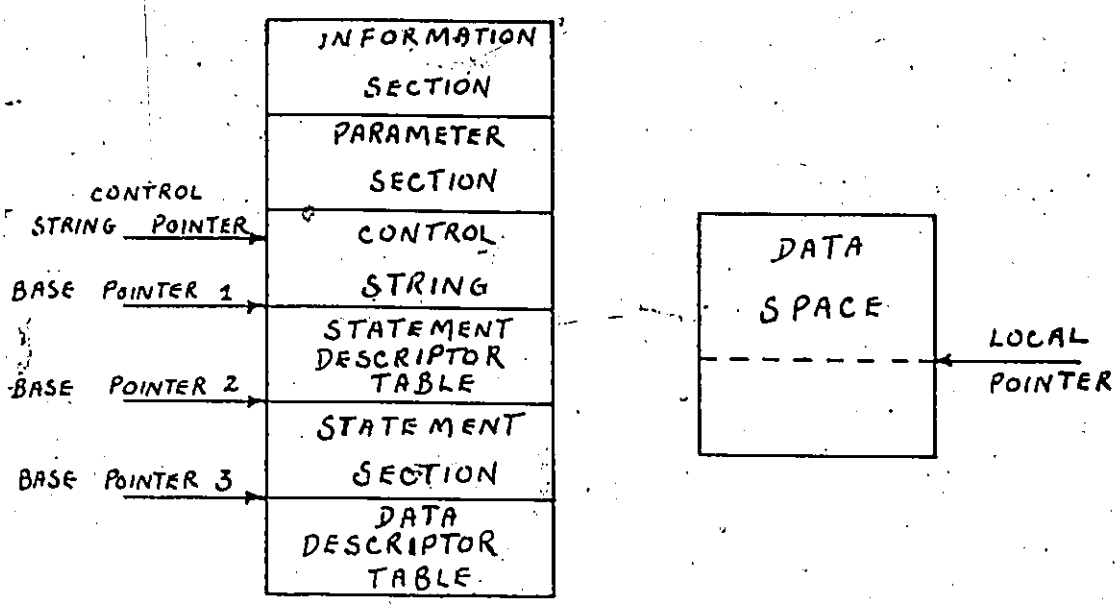


Figure 36: Required pointers

charge of reading the information section, while module 1 and 2 analyse the control string and activate either module 3, the expression executor, or module 4, the process management unit.

4.4.2.1 Module 0

This module is in charge of obtaining the various pointers from the information section. This module must be activated before the main program can be executed, since the various pointers are required to properly access the execution language sections.

4.4.2.2 Module 1

Module 1 is in charge of scanning through the control string to pick up the descriptors of the executable statements. The output of this module is therefore a queue of descriptors that must be executed. This module must also perform the various jumps as indicated by the control words. These various control words are identified by the tag bit, which in this case is set to one. As outlined in section 4.3.1.6, the actions required are very simple, since they mostly consist of either skipping a displacement or adding it to (or subtracting it form) the program counter. The actions are selected with respect to the value of the status flag, which is modified by specific control words, and by the result of a logic check expression.

4.4.2.3 Module 2

This module accepts the descriptors from module 1 and sends information to the proper module for execution. The information in the statement descriptor table is used to select the proper module. The address of the logic check expression or assignment statement that must be executed is sent to module 3, which would correspond to a task executor in the real system. Procedure calls on the other hand are sent to module 4, which coordinates the passing of parameters, and saves the required information in order to return to the calling program or procedure.

4.4.2.4 Module 3

Assignment statements and logic check expressions that were sent here by module 2 are executed using a stack mechanism since the expressions are written in a reverse polish notation, and only manipulate simple data types. To execute such expressions, this module must access the statement section to obtain the statement, the data descriptor table to obtain information about the variables and constants in an expression, and the data space to obtain the actual value of the variables and constants.

The rules for evaluating a reverse polish expression can be summarized as follows:

1. Scan the expression from left to right
2. When a descriptor of an operand is encountered, obtain its value by first obtaining its location from the data descriptor table, and then retrieving it from the data space; this value is pushed on the operand stack. Since only integers are allowed, the type and structure of a variable is not considered. In a more advanced system the task executor would have to be modified to handle more complex data structures.
3. When an operator is encountered, that is if the tag bit is one, the operation is performed with respect to the two operands on top of the stack. The two op-

erands are then discarded and the result is placed on top of the stack. Some operators might affect only one operand, in this case the top element of the stack is used.

4.4.2.5 Module 4

When a procedure call statement is encountered, as outlined in section 4.4.1, the parameters must be passed between the two procedures, before the control is passed to the called procedure. This is accomplished by appending the data space of the called procedure to the one of the calling program, and transferring into it the required values and addresses. As stated before, the actual description of the passing of parameters will be described when the hardware of the various units will be analysed. In the simulation, the constants are kept in the data descriptor table in order to simplify the management of the data space.

This module also stores the information about the calling program on a stack in order to return to it when the called procedure terminates.

Program returns are also executed by this module when the END control word is located by module 1.

4.4.2.6 Module interaction

In the most general form, the execution of a process requires complex interactions between these modules. A number of operations, such as scanning the control string and activating the task executor, can be performed concurrently thus increasing the processing speed. However, this is not apparent in the simulation where all operations are performed sequentially.

Using the module interaction flowchart of figure 37, the actual execution of a process in the execution language can be described. The first step is to obtain the information about the main program, this is done by module 0. After this is accomplished, the control string is scanned by module 1 to obtain the first word. If this word is a control word then the proper actions are initiated by this module. However, if this word is a descriptor, it is placed in the descriptor queue where it will be accessed by module 2. Module 2 accesses the descriptor queue sequentially and activates the proper module to execute the particular statement or procedure call. The actual execution of an assignment statement or a logic check expression is done by module 3. The sequence of operation, module 1, module 2, module 3, module 1, ... is broken only if the END control word is located or if a procedure call is encountered. In this case, module 4 is activated, and the sequence module 1, module 2, module 3, ... is resumed for the procedure.

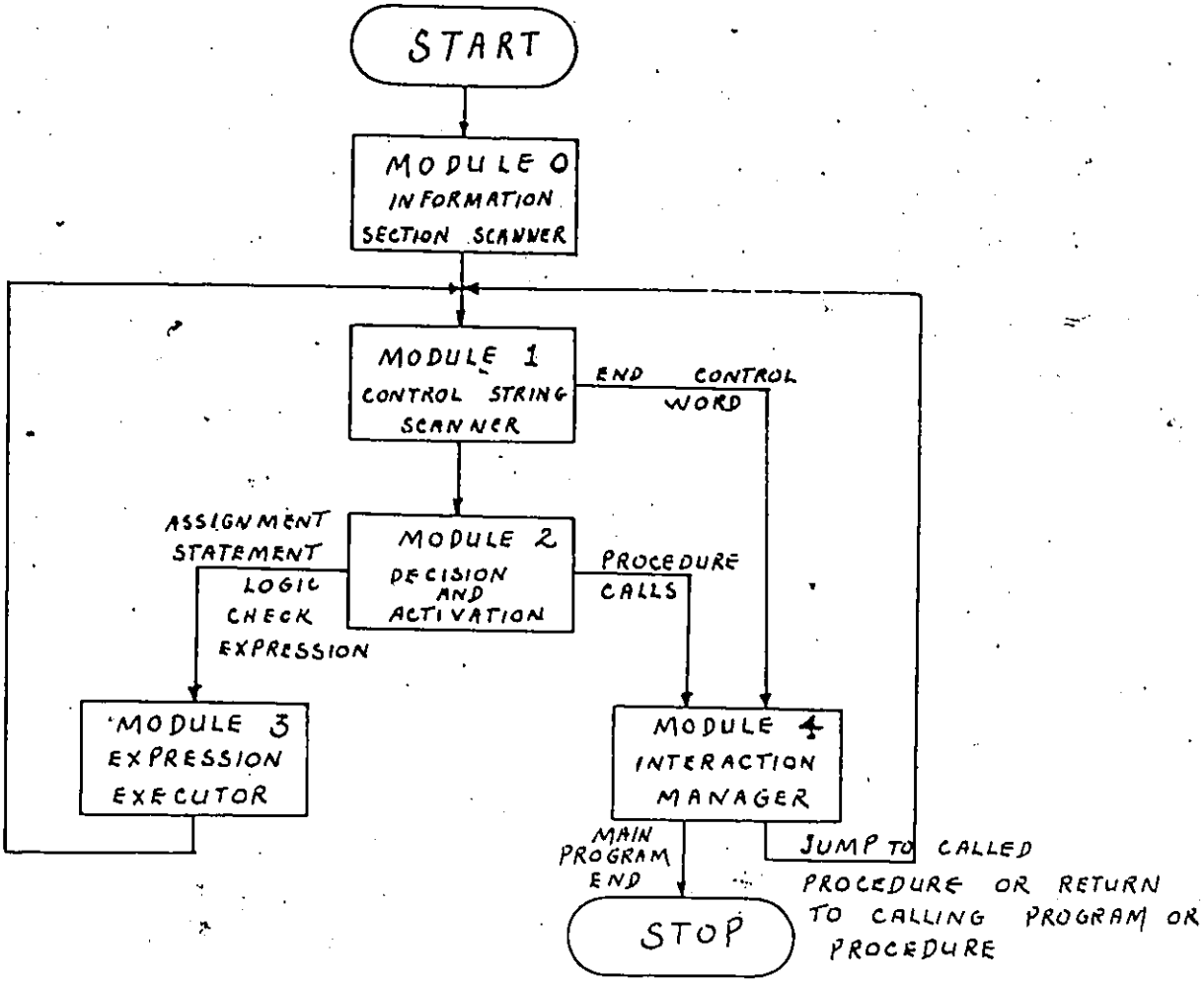


Figure 37: Execution module interaction

For END control words, two possible actions can be requested, since this control word can correspond to the end of the main program or to the end of a procedure. If the END control word is of the main program, then the processing stops. However, if it corresponds to the end of a procedure, module 4 will use the information that was pushed on the control stack to return to the calling program or proce-

dure. The information section of the calling program or procedure is read to update the various pointers.

If a procedure call is encountered, module 4 will store the information about the calling program or procedure on the stack, plus the location of the actual parameters, and then jump to the called procedure. This is accomplished by reading in the information section of the called procedure, which corresponds to the actions of module 0. Once in the called procedure, the actual parameters and formal parameters are related, so that the called procedure will access the proper values. Processing continues for this procedure in the manner described above.

While executing a process, many errors can easily be detected. This is a major advantage and stems from the use of a higher level machine language. For example:

1. Type mismatch are easily detected, since information about variables is stored in the data descriptor table.
2. While jumping in a program the control can only be transferred to one of the following control words: ELSE, ENDIF, ENDWHILE, WHILE.
3. Logic check expressions can only follow WHILE and IF control words.

In developing the simulation programs, these checks were included to test for proper operation. Once the programs

were proven correct, these checks were dropped for the sake of simplicity.

Using the example of the program that calculates the factorial of a number in a recursive fashion, the list of statements executed as obtained from the simulation is shown below.

```
INPUT VALUE OF number
   5
0    2    =    S
FAC    S
0    3    >    S
0    3    -    2    :    S
FAC    S
0    3    >    S
0    3    -    2    :    S
FAC    S
0    3    >    S
0    3    -    2    :    S
FAC    S
0    3    >    S
0    3    -    2    :    S
FAC    S
0    3    >    S
3    1    :    S
1    0    *    1    :    S
1    0    *    1    :    S
1    0    *    1    :    S
1    0    *    1    :    S
1    0    *    1    :    S
OUTPUT VALUE OF output
   120
```

In appendix E, other examples of this simulation are presented.

4.5 HARDWARE REQUIREMENTS

In this section, the hardware organization corresponding to a minimum complexity system is presented. A minimum complexity system, that executes a single process, has three elements as shown in figure 38 .

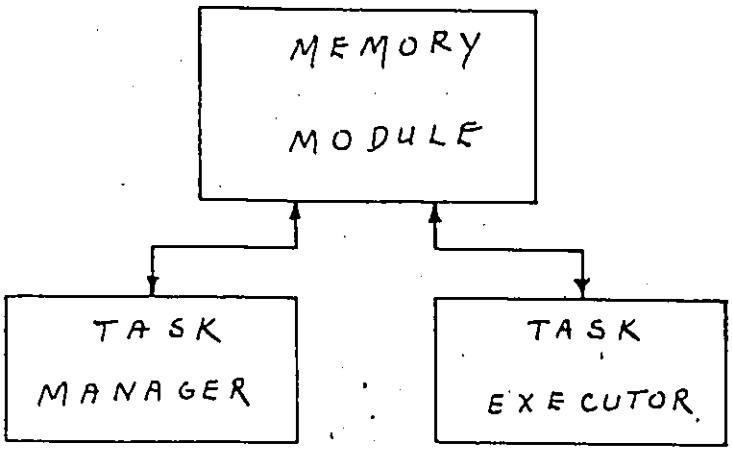


Figure 38: Minimal system

In this figure, the system manager is not represented, since only the execution of a single process is considered. From the definition of the different modules in the previous section, it can be seen that the task manager performs the action of module 0, 1, 2, and 4, while the task executor corresponds to module 3.

With respect to the task manager, three subunits can be specified that perform the required actions. These subunits are: the control string unit, the handshake and allocation

unit, and the process management unit. Note that in the following block diagrams, the organization is given in terms of the tables or sections that the units access.

The control string unit is in charge of scanning the control string, that is, it performs the actions required by module 1. A block diagram of this unit is shown in the following figure.

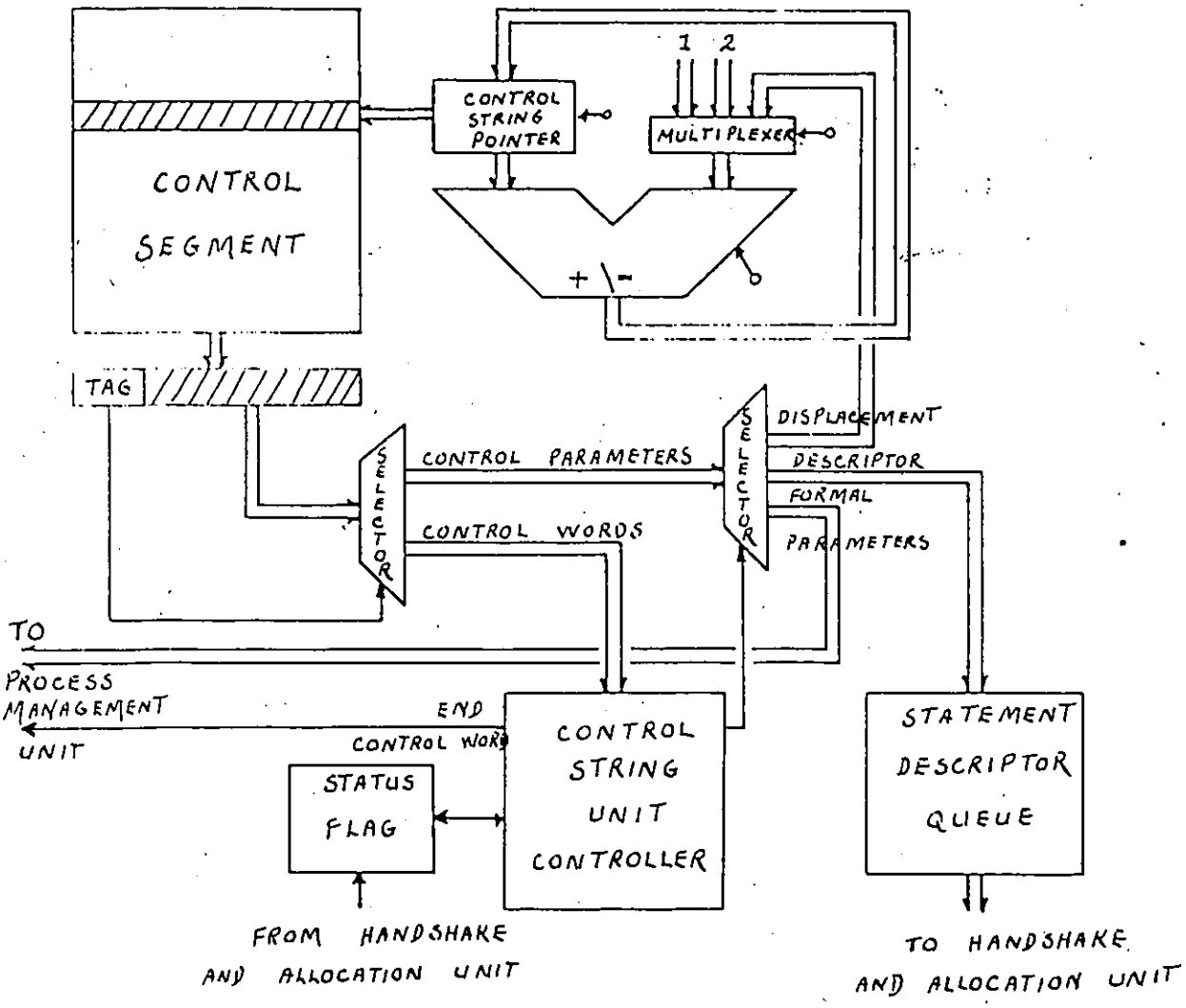


Figure 39: The control string unit

This scanning is performed by using a control string pointer which is manipulated according to some control words, by a simple adder/subtractor circuit. The output of this unit is a queue of statement descriptors pointing to the statements that must be executed. To coordinate the flow of descriptors, this unit has a status flag that is used to save the status of the executed logic check expressions. As stated before, this status flag is also manipulated by some control words. A small control unit is required to decode the control words and initiate the required actions. The microprogram in such a unit would be small, since the actions required by the control words are simple, and mainly consist of skipping or adding a displacement to the control string pointer. This unit also activates the process management unit if an END control word is encountered.

The handshake and allocation unit reads the descriptors that are in the descriptor queue and activates the proper units to execute the corresponding statements. The organization of this unit is shown below.

For logic check expressions and assignment statements, the task executor is activated by sending it the address of the statement to execute. This address is obtained from the statement descriptor table by adding the descriptor to pointer 1, obtaining the entry corresponding to this statement.

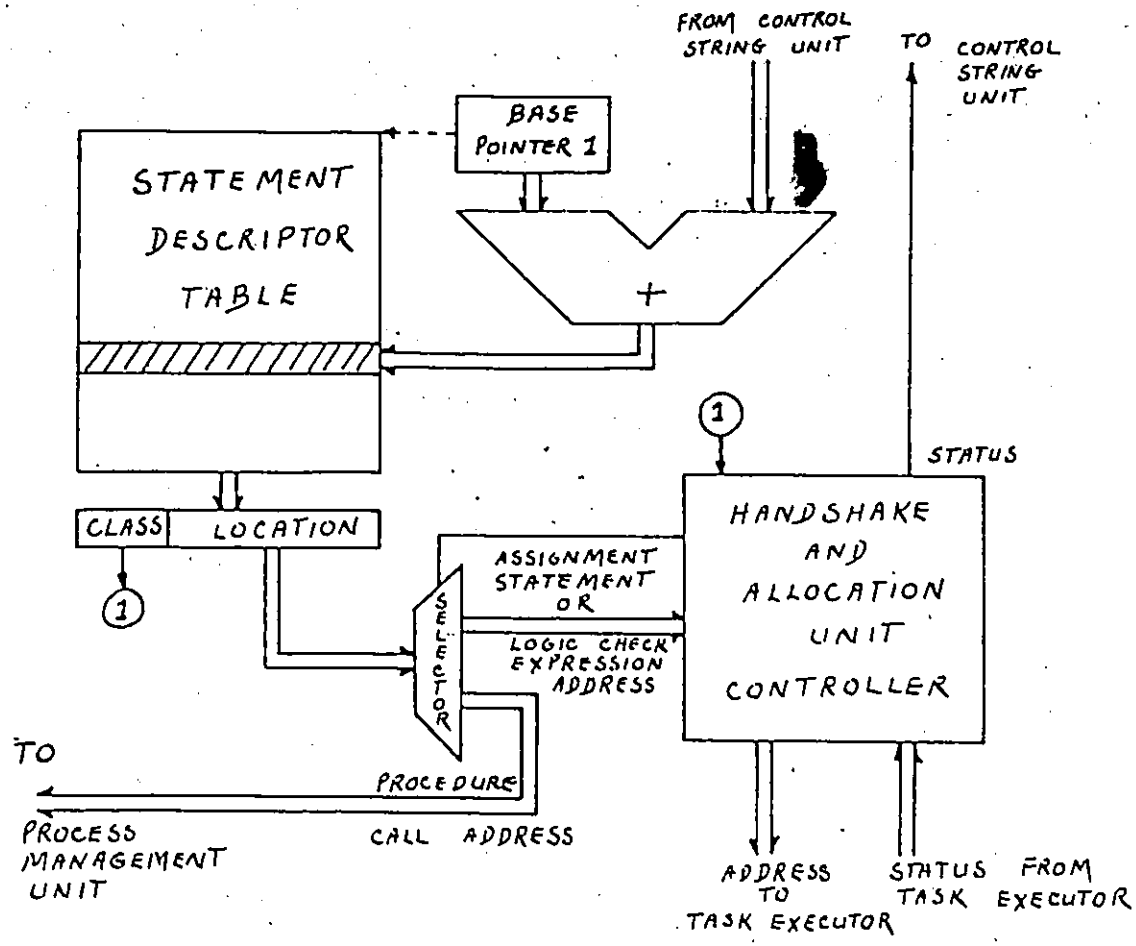


Figure 40: Handshake and allocation unit

The handshake and allocation unit, will also activate the process management unit if a procedure call is encountered. This is done by sending to the process management unit, the address of the procedure call statement that must be executed.

The process management unit performs the actions required by module 4 and 0. It is activated at the start of a process and when a procedure call statement is encountered by the handshake and allocation unit, or when an END control word is encountered by the control string unit. The block diagram of this unit is given below.

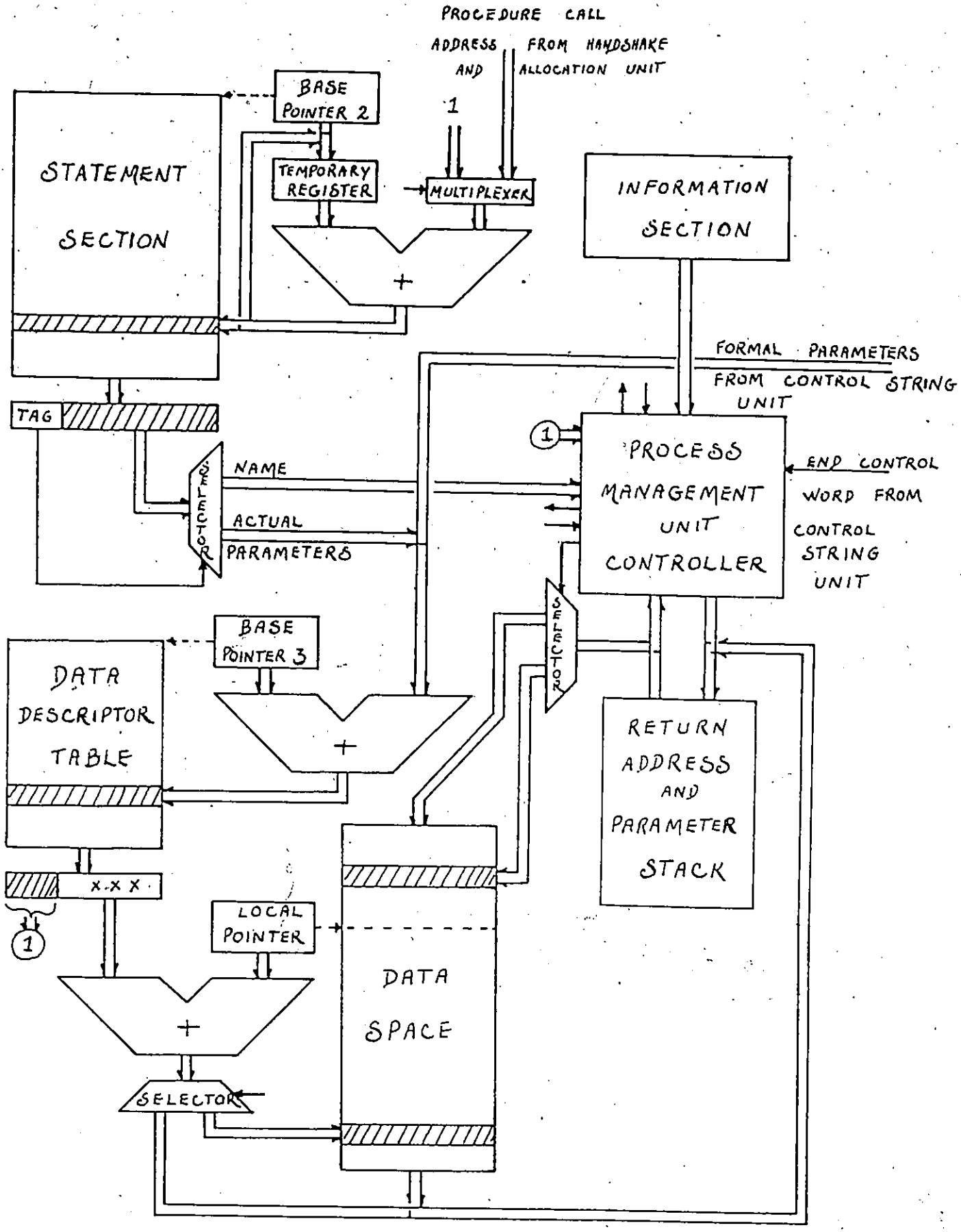


Figure 41: Process management unit

At the start of a process, the information section of the main program is read in, in order to initialize the various pointers. If a procedure call statement is encountered, the control string pointer and the name of the calling program or procedure will be saved on the control stack. This stack is also used to pass the parameters to the called procedure. This is accomplished in the following manner:

1. First the addresses of the values of the actual parameters in the data space are placed on the control stack. For LOCAL and INPUT variables, the relative address obtained from the data descriptor table is added to the local pointer and this value is pushed on the control stack. However, for INOUT and OUPUT variables, the relative address is added to the local pointer, and this computed address is used to obtain a value from the data space. This value is then pushed on the control stack.
2. Once this is accomplished for all actual parameters, the information section of the called procedure is read to update the various pointers. The location of the information section of a called procedure can be obtained by looking for the name of the procedure in a table. However in the simulations, since the procedure called were all on a sequential file, the location of the procedure information section was placed in the procedure call statement, following the name of the procedure.

3. Finally, the addresses of the actual parameters on the stack are used to fill the local section of the data space used by the called procedure, with the proper values and addresses. This is done by reading in the parameter section of the called procedure. For formal parameters of class INPUT, the address on the stack is used to access the value of the variable which is then copied in the local section of the called procedure. For formal parameters of class INOUT or OUTPUT, the address on the stack is directly placed in the local section of the called procedure, at the proper location.

When an END control word is encountered, the information on the control stack is used to return to the calling program or procedure. That is, the information of the calling program or procedure will be read in, in order to initialize the pointers, then the control string unit will be activated to continue processing where it left off.

The process management unit, when modifying the various pointers, must inform all other units. This includes the task executor, since it uses pointer 2 to access the statement section, pointer 3 to access the data descriptor table, and the global and local pointers to access the data space.

The task executor is responsible for executing logic check expressions and assignment statements. Its corresponding block diagram is given in the following figure.

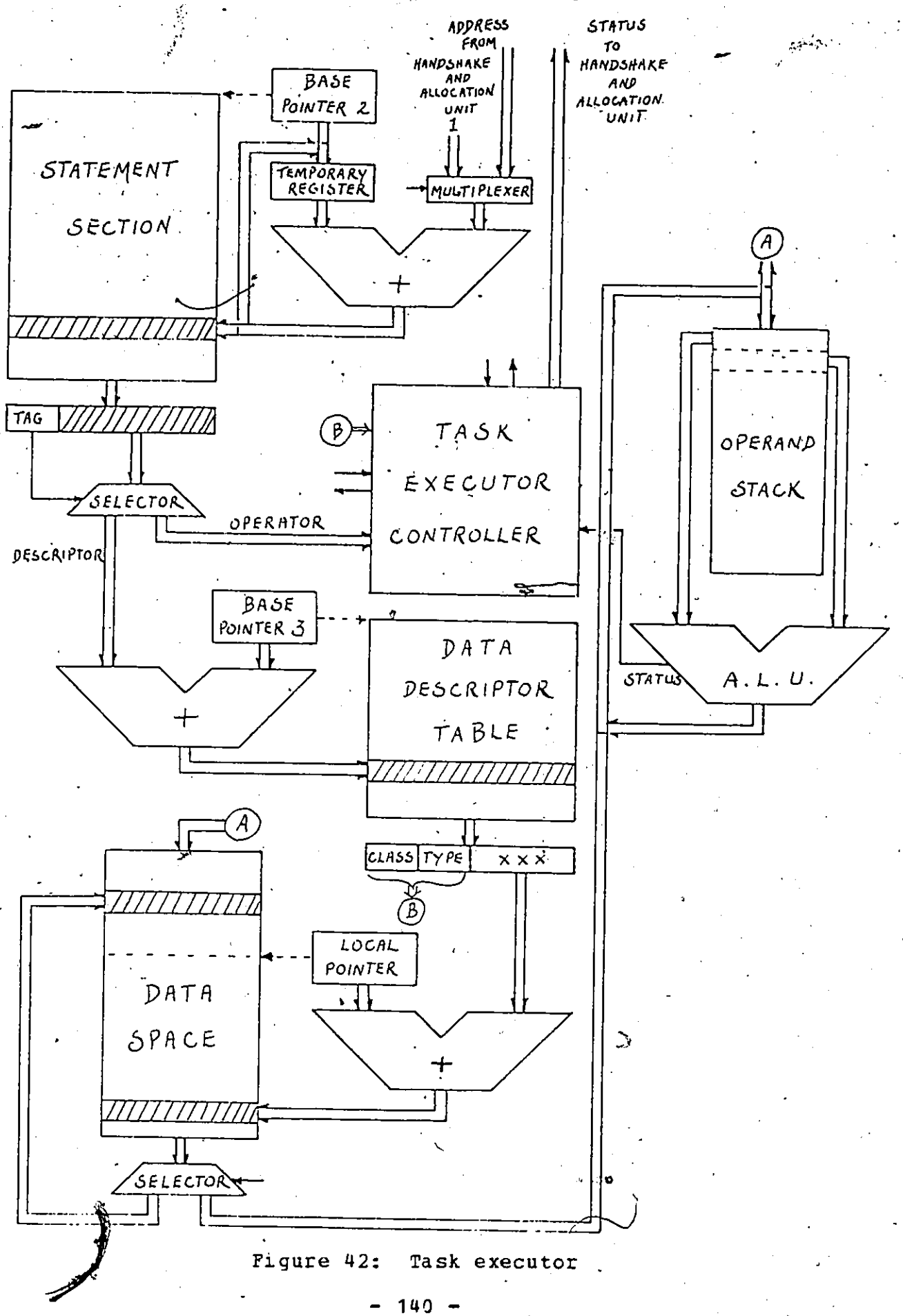


Figure 42: Task executor

The task executor uses a stack mechanism since the expressions are written in a reverse polish notation, and only simple types are allowed. As mentioned before, the task executor will be activated when it receives the address of an assignment statement or a logic check expression from the task manager. This address is a relative address, which is combined with the base pointer 2 to obtain the actual address of the statement. This statement is then executed by pushing the proper values onto the operand stack and executing the operations with respect to the top of the stack. The operands are either obtained directly or indirectly from the data space by examining the class of the variables. Logic check expressions will return a TRUE or FALSE condition to the task manager, which is used to update the status flag. The expression is completely processed when operator 's' is encountered.

4.6 CONCLUSION

In this chapter, the direct execution of a modified high-level language called the execution language was presented. The modification introduced provided a functional separation of the control aspects and processing aspects of any high-level language. This separation reduces the amount of information transfer required and greatly simplifies the implementation of the overall system.

The actual implementation of the units, as indicated by the simulation, can be done in a segmented way which permits each unit to be designed and upgraded separately. This also allows for high reliability of the system.

The execution language was chosen so that it can be representative of present day high-level languages in terms of its syntactical and semantic properties. The system facilitates the editing and debugging of high-level language programs, since errors can be easily identified during execution.

Chapter V

PREPROCESSING

5.1 INTRODUCTION

The purpose of this chapter is to define the preprocessing functions involved in the translation from the virtual language to the execution language. This preprocessing function was briefly introduced in the previous chapter (fig. 28). As indicated in chapter four, an execution language program or procedure has three segments: a control segment, a statement segment and a data segment.

In addition, one needs to include an information section, to facilitate the processing of an execution language program or procedure. This information section is accessed before a program or procedure is executed, and contains the necessary pointers for proper execution. In more detail, the execution language can be separated in the following seven sections.

1- Information section

2- Parameter section

3- Control string

4- Statement descriptor table

5- Statement section

6- Data descriptor table

7- Data space

The generation of these sections is described in detail in this chapter. However, before analysing the preprocessing function, the conventional translation from a high-level language source program to a von Neumann based machine language will be described.

5.2 CONVENTIONAL TRANSLATION

In most present day von Neumann computer, before execution, a high-level language source program has to be compiled using a software compiler. This compilation can be subdivided as shown in figure 43 to simplify its complete understanding (AHO 79). Each of these steps is now briefly described.

The lexical analyser scans the characters of the source program one at a time and assembles them into meaningfull tokens. The tokens can be identifiers, operators, reserved words, or delimiters. The output of the lexical analyser is passed to the syntax analyser, which groups these tokens into syntatic structures. These two steps can be done sequentially (in two passes), where the lexical analyser places its output on an intermediate file, which in turn is accessed by the syntax analyser. However, as done in most modern compilers, these two steps are grouped together on

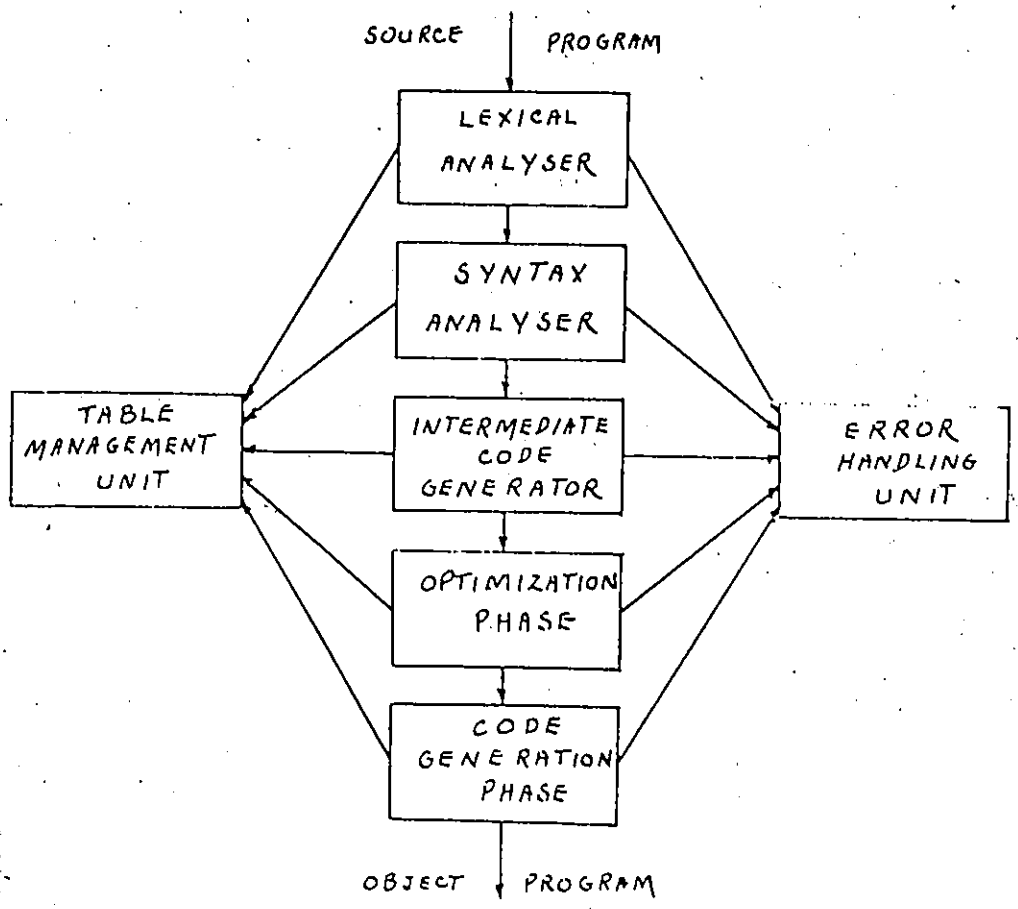


Figure 43: Compilation steps

the same pass, with the lexical analyser being activated by the syntax analyser. The lexical analyser therefore acts as a subroutine to the syntax analyser.

Code generation is usually performed in two steps to simplify translation. The intermediate code generation step, transforms the syntatic structure into an intermediate language representation of the source program. The intermediate language constructs used, depend on the properties of the high-level language and on the underlying machine. Most

intermediate languages are based on a polish string representation since this can be easily obtained from the syntactic structure. The syntactic structure is itself a parse tree, representing the link between the various tokens. Since most present day computers do not directly process polish strings, an additional translation step is needed.

The optimization phase, as available on most compilers, transforms the output of the intermediate code generator into an intermediate language version from which a faster or smaller object language program can be derived. This optimization must take into account the peculiarities of the given machine, both with respect to the hardware (registers, flags, etc.) and the instruction set.

The final phase is the code generation phase, which converts the intermediate code into a sequence of machine instructions.

As indicated in figure 43, two other units are usually present in a compiler. An error handling unit is required to detect flaws in the source program, and a table management unit is used to keep track of the indentifiers used in the program.

Code optimization and code generation are the most difficult steps to implement if one wishes to produce truly efficient object programs. For example code generation is not

straightforward, since this unit must keep track of the run-time contents of the registers in order to generate load and store machine instructions only when necessary. These two steps can be removed by directly interpreting the intermediate language. However, in this case the steps of code optimization and generation are performed by the interpreter, which contributes to its complexity.

5.3 PREPROCESSING STEPS

In this section, translation between the virtual language and the execution language is investigated. Thus the intermediate language corresponds to the execution language which is directly executed, therefore the steps of optimization and generation are eliminated. The translation required has thus the elements shown in figure 44 . Roughly speaking, the lexical and syntactical analysis phases are the same as on present day computers, but the intermediate code generation phase is simpler because of the functional separation of the data, statement and control.

The actual steps required by the preprocessing function were simulated using the language Pascal. This was done, not only to indicate the feasibility, but also to show that the translation is a very simple process.

The lexical analysis stage scans the source program character by character and assembles the tokens. In order to

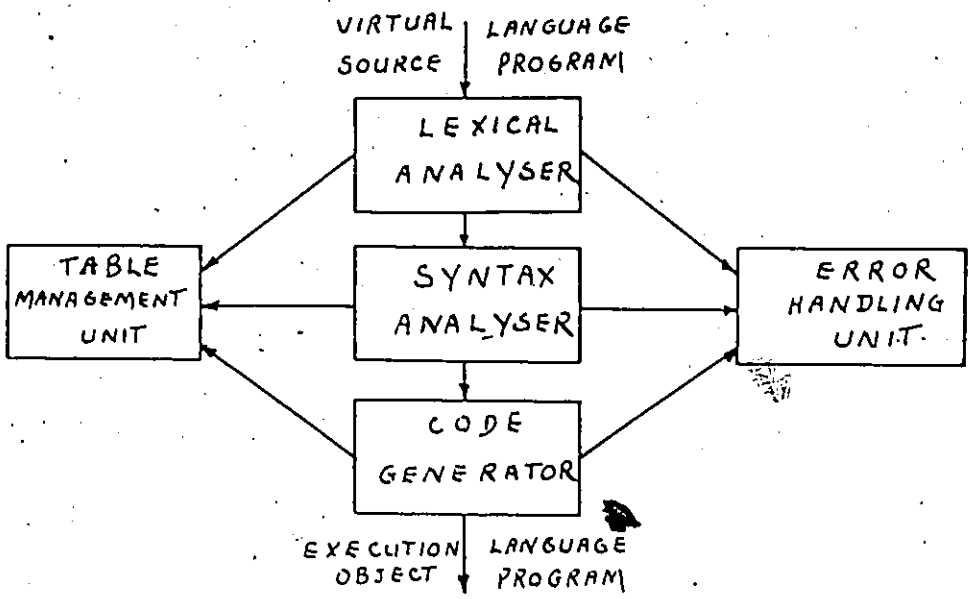


Figure 44: Preprocessing function

simplify the lexical analysis stage, the virtual language was defined so that each token is separated from the others by delimiters. Having to put delimiters makes programming a bit cumbersome, but it greatly simplifies the lexical analysis phase. An alternate method requires that transition diagrams be built to consider all possible occurences of character strings in the language. In the virtual language the following delimiters are available.

<delimiter> ::= ; | , | b | (|)

An exact syntactical definition of the virtual language is given in appendix A, using Backus Naur Form notation. This definition takes into account the separability of all the tokens.

The declare segment, in order to separate all entities has the following syntax: a semicolon is added after the global and local variables to separate these two sections.

<name>(<global variables>);
<local variables>;
<type declaration>

Assignment statements and procedure calls are also delimited by a semicolon to ensure their proper processing.

<assigment statement>;
<procedure call>;

The preprocessing function is divided into two steps. The first one will manipulate the declare segment, while the second will manipulate the compute segment. Each of these steps is in charge of generating the various sections of the execution language program.

5.3.1 Step 1: Declare segment preprocessing

The declare segment preprocessing involves the following four steps:

1. The name of the virtual language program or procedure is located and saved. This name is added to the information section of the execution language representation.

2. The names of the global variables that follow the program's name are placed in the symbol table along with their class specifications, such as INPUT, OUTPUT, or INOUT. The symbol table holds all information about the variables and, in a sense, corresponds to the data descriptor table. At the same time, a parameter section that holds the descriptors of the global variables is generated.
3. The local variables which follow the global variables are placed in the symbol table with their class specification as LOCAL.
4. Next, the type declarations are scanned, and the proper type is assigned to each variable in the symbol table. Since all of the information about the variables is available, the addresses of the variables in the local section of the data space are calculated at this point. These addresses are calculated with respect to the type and class of the variables, that is, a variable in the symbol table will be assigned an address that depends upon the space required to store the previous variable. For integers, the space required corresponds to one word, however for reals two words might be necessary.

Using the information that was present in the declare segment, step 1 generated the data descriptor table. Declaring the variables before using them, reduces the number

of passes required for preprocessing, since the symbol table is filled before the compute segment is processed.

5.3.2 Step 2: Compute segment preprocessing

The compute segment preprocessing locates the various statements in order to generate the control string, the statement descriptor table, and the statement section. The data descriptor table is also manipulated to include the characteristics and values of constants.

As described in chapter three, three different types of statements can be encountered, they are: assignment statements, control statements, and procedure calls statements. These statements must be effectively recognized in order to build the control string. Once the beginning of a statement is located, its type must be identified. The first token of an assignment statement will correspond to the symbolic name of a variable that can be found in the symbol table, while the first token of a control statement will correspond to a control word. Procedure calls on the other hand are all the statements where the first token does not correspond to the identifier of a variable or to a control word. It should be noted that a variable name or a control word should not be used as a procedure name since the procedure call statement will then be misinterpreted. However, by modifying the preprocessing function, this could be avoided. The following

steps indicate the different actions initiated for each of the three types of statements.

1. For an assignment statement a descriptor representing this statement is placed in the control string. As described before, the tag bit associated with this descriptor is set to zero. This statement is then transformed into a reverse polish notation and placed in the statement section. The infix to suffix transformation is described later on.
2. For control statements, different actions are initiated depending upon the control word. When a control word is located, it is placed in the control string with its associated tag bit set to one. For IF and WHILE control words, the logic expression that follows is transformed into a reverse polish notation and placed in the statement section. A descriptor is then placed in the control string representing this expression. The THEN and DO control words are used to limit a logic check expression as shown below. These two control words are not included in the control string.

   IF <logic check> THEN

   WHILE <logic check> DO

   The ELSE, ENDIF and ENDWHILE control words indicate the end of a particular virtual language seg-

ment, therefore when encountered, the required jump addresses must be placed at the appropriate location in the control string. These addresses are calculated by using a stack. This stack is loaded with the control string pointer when the IF, ELSE or WHILE control word is encountered. The control string pointer points to the present control string element being accessed. This counter is used to simulate the operation of a hardware instruction counter as though the program was executed. Therefore when the control words ELSE, ENDIF or ENDWHILE are encountered, the top of the stack is subtracted from the present value of the control string pointer, and this value which corresponds to the required displacement, is placed at the proper position in the control string.

3. For procedure call statements, a descriptor is placed in the control string representing this statement, then it is placed in the statement section in the appropriate format. In this simulation only calls by name are allowed.

At the same time, the statement descriptor table is generated, which contains the statement's class and address in the statement section. This address is a relative address, which must be combined with the base pointer 2 to obtain the location of the statement.

The transformation from infix to suffix notation is performed in the standard form, by scanning the expression from left to right, and using a stack to manipulate the operators. The following steps are required for the translation.

1. If an identifier of a variable is encountered, its descriptor is placed in the statement section, with the tag bit equal to zero. Note that the address of the first element of a statement in the statement section is included in the statement descriptor table.
2. Similarly if a constant is found, an entry is created in the data descriptor table to store information about it. A descriptor for this constant is then placed in the statement section. The value of the constant, for the simulation, is placed in the data descriptor table, however, in a more sophisticated system it would be placed in the data space of this program or procedure. To simplify preprocessing, the constants could be declared in the declaration segment. The expressions in this case would contain symbolic names representing the constants.
3. When an operator is encountered, the top of the operator stack is examined in order to select the proper action. Left and right parantheses are used to remove ambiguities in the order of evaluation. If two operators have the same precedence, the expression is

evaluated from left to right. (The precedence of the operators is given in appendix C, under the description of FUNCTION LEVEL).

a) If the stack is empty, the operator is placed on top of the stack.

b) If the operator is a left parantheses "(", it is placed on top of the stack regardless of the content of the stack.

c) If the operator has a lower precedence than the one on top of the stack, then the operator on top of the stack is placed in the statement section, with its tag bit equal to one. This continues until the stack is empty or until the operator on top of the stack has a lower precedence. Then if the operator is not a right parantheses ")", it is pushed on the stack, if it is, then the stack is popped. In other words, a right paranthesis will cancel a left paranthesis that was previously pushed on.

Processing of the string continues until the token indicating the end of the expression is encountered. For assignment statements this token is ";", while for logic check expressions it is either THEN or DO.

Figure 45 illustrates the movement of tokens required to perform the translation.

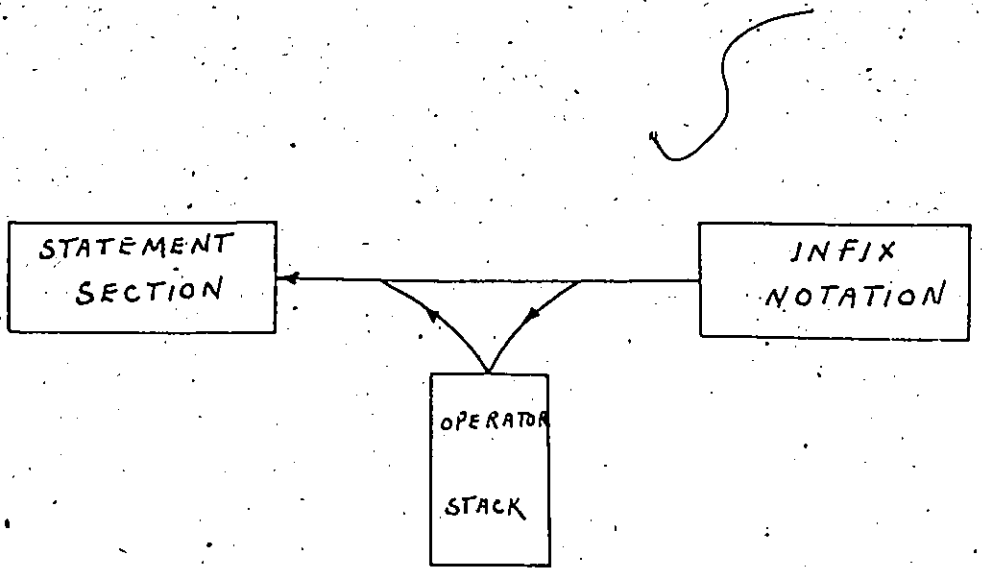


Figure 45: Infix to polish transformation

Preprocessing stops when the END control word is encountered. At this point the information section required to properly execute the program is built. The pointers in this section are defined when the execution language representation is placed on the output file. One should note that in the preprocessing simulation, only simple error detection was incorporated.

5.4 CONCLUSION

Due to the close relation between the properties of the virtual language and the execution language, the preprocessing function has been shown to be relatively simple. This simplicity makes it attractive to implement this preprocessing in hardware. This unit could have an organization similar to the one used in the Symbol computer (LALI 73), since their operations are similar. The definition of such a hardware preprocessor was not attempted here, because in

order to specify a meaningful hardware system, the following issues must be considered.

1. error diagnostics and error recovery.
2. operating environments.
3. amount of storage available.
4. use of a virtual language with more complex data and control structures.
5. structure of the symbol table.
6. etc.

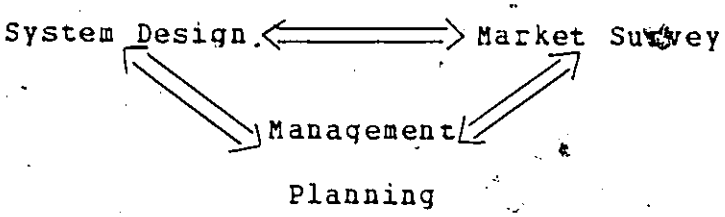
In addition, a preprocessor could also consider the detection of concurrencies, however this was not attempted here. One could deduce from a superficial examination that this would be simple because of the separation of the data, the statements, and the control during preprocessing.

Chapter VI

CONCLUSION, ENHANCEMENT AND FUTURE RESEARCH

6.1 CONCLUSION

Whenever one considers the possible implementation of a complex engineering product, a number of preliminary studies must be performed. These studies can be roughly subdivided into three classes



where market survey includes the survey relating to demand and product acceptance, while management planning includes, planning of production, planning of distribution, planning of consumption, and planning of retirement. Finally the system design consists of a feasibility study, a preliminary design, and a detailed design.

In this thesis, the feasibility study for a distributed high-level language system was demonstrated. Because of the positive result obtained, a preliminary design could now be undertaken. This would require going in depth in the overall system operation.

The feasibility study was based on a top down design approach, by performing the following five steps.

1. Specification of the design objectives. The design objectives, described at the beginning of chapter 3, were defined by examining the techniques devised to solve the four main problems associated with the von Neumann computer, as surveyed in chapter two. This analysis lead to the choice of a distributed system that supports directly the constructs of present day high-level languages.

2. Environment specifications. This step involved the specification of a virtual language (section 3.3), as a typical high-level language. The virtual language chosen is related, in terms of its data, control and operation entities, to present day procedure oriented languages.

3. Specification of system organization. From the defined virtual language, the specification of a task and process was defined. A process corresponds to a program, while a task corresponds to an executable statement in this program. These tasks and processes can be supported in a concurrent fashion by the two level task driven system developed, in order to achieve high performance. The three main units of this system are: the system manager, responsible for managing the resources and the flow of processes in the

system, the task manager, which performs a task similar to the system manager but at the task level, and the task executor, which executes the required tasks.

This system is inherently modular, which implies that many units can be easily added to increase its performance. For example special task executors can be added to handle complex functions. This modularity also enhances the system reliability, since the simplicity of the module makes it easy to detect and isolate faults. A fail-soft mode of operation is also possible, since many task managers and task executors are available. However, the system manager must have some form of fault tolerance since it is not duplicated. The operations of the system manager could be performed by one of the task managers, in order to increase the reliability of the system.

4. Specification of the execution language. The next step, was to examine the actual execution of a process on this system. It was shown that an increase in execution speed could be achieved by modifying the language. The execution language was therefore developed from the virtual language by performing some relatively simple preprocessing. This preprocessing is very simple due to the small property gap between these two languages.

Since the execution language is directly related to present day high-level languages, many errors can be detected by the hardware during processing. This produces a user oriented system, as an error can easily be associated with the proper high-level language source statement. Furthermore, this execution language does not restrict the number of high-level languages supported on the system, since it can be easily obtained from many high-level languages.

5. Specification of unit design. The task manager and executor units were then designed to support directly the execution language. Examining the processing of an execution language process, the various functions of the units were specified and their basic organization was defined.

To indicate that the overall system performs as described, three simulation examples are given in appendix B. The first example computes the correct moves to solve the tower of Hanoi game. This program was used because of its recursive algorithm in order to test the accessing of variables in the procedures. The second example, calculates the determinant of a matrix, and indicated the correctness of the reverse polish transformation algorithm and of the execution of such a transformed expression.

In the previous two examples, only simple types were allowed. The third example considers a sorting program, in

which sorting is performed on a vector. The required changes to the two simulation programs to support vectors, are explained along with this example. In each of these, the virtual language and the execution language representations are given. This is followed by the output obtained from the processing of the execution language representation.

The design cost for large distributed systems can be prohibitive, since they require a separate design for each processor. Present designers want to minimize this cost. However for this design the units are simple and the software design cost is reduced due to the specialization of the units.

For such a system to be cost effective it must be built from VLSI elements. Using standard VLSI components such as microprocessor to build a complex distributed system results in a system with little practical value since these microprocessors have the same problems that are associated with the von Neumann computer (KLUG 79). One should design new architectures that are based on special purpose VLSI elements, since these elements are designed with respect to the complete system. For example X-Tree (PATT 79) is a multiprocessor system based on special VLSI processors called X-Node. In this distributed high-level language system, the simplicity of the elements makes them ideal for implementation using LSI components.

A disadvantage of our proposed system, is the use of descriptors. The time needed to fetch the information pointed to by them can be large. A possible solution to this is to keep the descriptor tables in high speed storage. However the use of descriptors offers many advantages, such as an easy representation of the type and structure of a variable. This has not been examined as such in this thesis, however some research has been done in this area, for example in TANE 76, the representation of arrays using descriptors is analysed.

6.2 IMMEDIATE ENHANCEMENTS

In this section, some enhancements directly related to the system described in chapter three and four are presented. This system was kept simple by limiting the complexity of the data and control structures of the virtual language.

With respect to control structures, only the basic constructs of structured programming were supported. Supporting more complex control structures requires that the task managers be modified to recognize these structures. More complex structures can also be transformed into the basic three control structures during preprocessing. For example REPEAT/UNTIL and FOR statements can be easily transformed into WHILE statements, while a CASE statement can be transformed into a sequence of IFs; however the direct support of

these constructs would greatly enhance the system, and simplify preprocessing.

Concerning the data structures supported, only simple types were allowed. This greatly restricts the range of applications of the system, as was observed when defining some meaningful examples to test the simulation programs. Therefore the possibility of adding vectors to the system was investigated. A structure declaration for a vector can have the following syntax.

(VECTOR (<index>) OF <data type> : <variable>)

where <index> is a constant that specifies the size of the vector, and <data type> specifies the type of the elements of this vector.

For a first approximation, vectors can be treated as follows. During preprocessing, when a vector declaration statement is encountered, a type of VECTINT will be added to all of the corresponding variables in the data descriptor table. This specifies that the variable is a vector of type integer. The size of this vector will determine the amount of place reserved for it in the data space.

A vector in a statement or expression has the following format.

<name>(<index variable>)

where <name> is the name of the vector, and <index variable> is a variable that is used to access the proper element of this vector. When a vector is encountered in a statement, it is replaced by its descriptor, and placed at the proper location in the statement section. The index variable is also replaced by its corresponding descriptor, and placed in the statement section following the descriptor of the vector.

During the processing of an execution language program, if a descriptor of a vector is encountered, the descriptor following it, in the statement section, is used to access the proper element. This is accomplished by taking the value of the index variable and adding it to the address of the first element of the vector. The address of the first element of the vector is given in the data descriptor table. The two simulation programs were modified to incorporate these changes. Since these changes were not completely tested they are covered in appendix E, where a program that manipulates vectors is analysed.

Supporting more complex data structures requires that the complexity of the task executor be increased in order to manipulate them efficiently. In this case the task executor could support concurrent execution at the processor level, to efficiently manipulate the structures. Furthermore, these structures must be represented in a proper format in

the data descriptor table and statement section. Much more reseach is needed in this field to determine the usefulness of the system.

With respect to the various sections of the execution language, additional fields can be added to increase the performance of the system. For example, the statement descriptor table in its present form, does not contain any pertinent information. The overhead of storing and retrieving statement descriptor is not justified, since the information contained in the table is small. However, additional fields can be added to enhance the capability of the system, such as fields specifying which task executor is required for the efficient execution of a task. Furthermore, details of statement concurrency can be added to this table during preprocessing. Executing statements concurrently requires that the data dependencies between the statements be small.

Concurrent execution of program segments, as described earlier, can be supported if such instructions as FORK and JOIN are available. One requires a program manager which will coordinate the execution of the various segments. However, since the operations of the system manager and program manager are similar, the system manager could perform directly these operations; this would imply that more than one task manager would be assigned to a process. This is practical only if the inclusion of these instructions is per-

formed by the preprocessor. Too much expertise is required when the user must include these instructions himself. (This however depend on the applications).

6.3 FUTURE RESEARCH

The most fundamental research required at this time, involves the inclusion of the various enhancements described in section 6.2. The efficient inclusion of these enhancements will determine the usefulness of the system. However, before the system becomes totally operational, the following research topics must also be investigated.

The overall control and operation of a system with many task managers and executors must be analysed. This would require the specification of the operation of the system manager. These operations include the loading of processes in the memory modules, and the allocation of the task managers and task executors on demand. As described in FATH 81a, the control of a task driven system can be performed in a table driven manner. These ideas must be adapted to the system described in this thesis, which is a two level task driven system. It is interesting to note that the execution of a process and the control of the system can both be performed using a table driven approach.

The memory management function of this system must also be analysed. This would probably require the specification

of a hardware unit to aid in the retrieval and storage of the data. This memory management is required at two levels: the secondary memory level, and the main memory level. At the secondary memory level, the functions to properly access and store complex data organizations must be specified. At the main memory level the same functions must be specified, however more complex structure manipulations must also be available. Usually it is difficult to specify which functions must be performed on the secondary memory, and which function on the main memory.

Once the foregoing research subjects have been investigated, one should consider the reliability aspects of the system. It was shown that the system could produce a fail-soft mode of operation since the tasks of a failed unit can be taken over by other units. This implies that the functions of detection, diagnosis, and correction must be performed by the hardware. Reliability can also be improved by adding enough intelligence to every unit to incorporate forced-no-response capability into the system.

As was mentioned in the conclusion, this distributed system, because of the simplicity of its elements, is ideal for VLSI implementation. Furthermore, these elements were all shown to be based upon various table manipulations, therefore one could design a general table manipulator chip that could be used as a building block for all the units of the

system. This building block would also be used in the control of the overall system.

Appendix A

VIRTUAL LANGUAGE SYNTAX

The virtual language which is used to develop the overall organization of the distributed system, was briefly introduced in chapter 3, by defining its control, operator and data entities. In this appendix its complete syntax will be described using Backus normal form notation. This indepth description is required to define the preprocessing function.

Program parts

A program has two parts:

1. A declaration part where one specifies all variables and their characteristics.
2. A computational part where one specifies the computational sequences.

```
<program> ::= <declare segment>
              BEGIN
                 <compute segment>
              END
```

The declare segment includes:

- The name of the program.

- Variables; the global variables used to interface the program with the external world and the local variables used internally by the program.

- Characteristics of the variables; type and structure declaration.

<declare segment> ::= <name>(<global variables>);
<local variables>;
<type declaration>
<structure declaration>

<name> ::= ... name of the program.

Variables.

<global variables> ::= <input vars>;<inout vars>;
<output vars>

<input vars> ::= ... list of variables whose value is given externally to the program.

<inout vars> ::= ... list of variables whose value is given to, changed by an returned from the program.

<ouput vars> ::= ... list of variables whose value is determined by the program and returned from it.

<local variables> ::= ... list of variables used internally by the program.

The following figure illustrates the scope of a variable in a program segment.

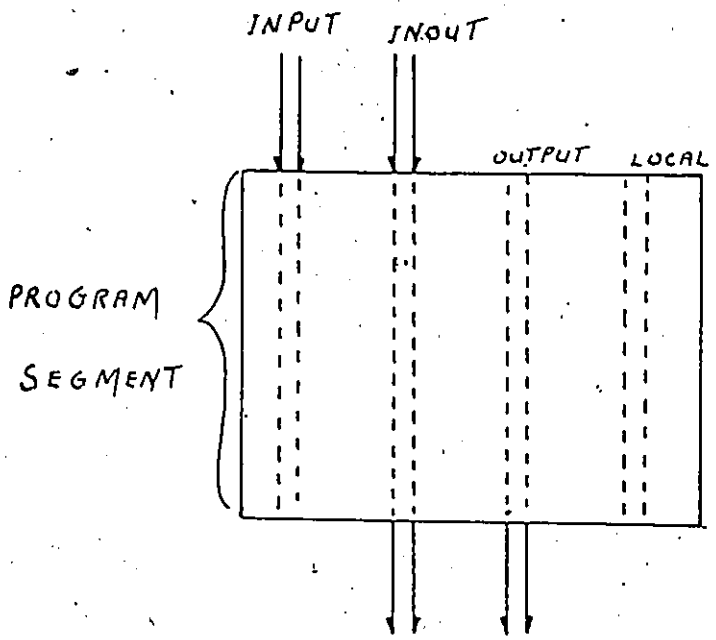


Figure 46: Flow of variables

Variables characteristics.

<type declaration> ::= 0|
(<data type> : <vars>)
<type declaration>

<data type> ::= ... the type of allowed data, ex: integer, real etc.

<vars> ::= ... list of variables associated with each data type.

<structured declaration> ::= ... the specification and characteristics of allowed data structure, ex: array, list, binary tree, etc.

Compute segment.

The compute segment is a list of statements, which can be of the following three types: assignment statements, control statements, and procedure call statements.

<compute segment> ::= 0|
<statement>
<compute segment>

<statement> ::= <assignment statement>;|
<control statement>|
<procedure call>;

In general the assignment statement is used to link a variable to an expression, that is:

<assignment statement> ::= <variable type>
<assignment operator>
<expression type>

Here we shall consider only arithmetic and logic expressions to limit the complexity of the task executors.

<assignment statement> ::= <arith variable> =
<arith expression>|
<logic variable> =
<logic expression>

Arithmetic expression

Arithmetic expressions are used to specify all arithmetic computations. The result of an expression is a single numeric value which is assigned to the associated arithmetic variable.

<arith variable> ::= ... a variable declared to be arithmetic. (Real, Integer, etc.)

<arith expression>| ::= <arith operand>
(<arith expression>)|
<arith expression>|
<arith expression>
<arith operand>
<arith expression>

<arith operand> ::= <arith constant>|
<arith variable>|
<arith function call>

<arith constant> ::= ... a number

<arith operator> ::= + | - | * | /

<arith function call> ::= <name>(<vars>)

<name> ::= ... name of the arithmetic function.

<vars> ::= ... the variables of the arithmetic function.

Logic expressions

A logic expression is used for testing purposes and contains relational and logical operators. The result of a logic expression is either TRUE or FALSE which is assigned to the associated logical variable.

<logic variable> ::= ... a variable declared logic. (Boolean)

<logic expression> ::= <logic operand>|
(<logic expression>)|
NOT <logic expression>|
<logic expression>
<logic operator>
<logic expression>

<logic operand> ::= <logic constant>|
<logic variable>|
<relational expression>|
<logic function call>

<logic constant> ::= TRUE | FALSE

<relational expression> ::= <arith expression>
<relational operator>
<arith expression>

<relational operator> ::= = | <> | >= | <= | > | <

<logic operator> ::= AND | OR | XOR | EQU

<logic function call> ::= <name>(<vars>)

<name> ::= ... name of the logic function.

<vars> ::= ... the variables of the logic function.

Control statements

The control statements used in this virtual language, are the ones employed in in structured programming.

<control statement> ::= IF <logic expression>
THEN <compute segment>
ELSE <compute segment>
ENDIF|
IF <logic expression>
THEN <compute segment>
ENDIF|
WHILE <logic expression>
DO <compute segment>
ENDWHILE

Procedure call.

<procedure call> ::= <name>(<procedure variables>)

<name> ::= ... name of the procedure.

<procedure variables> ::= <input vars>;<inout vars>;
<output vars>

Procedure and function declaration.

A procedure is just a program.

```
<procedure> ::= <name>(<global variables>);
                <local varaibles>;
                <type declaration>
                <structure declaration>
                BEGIN
                   <compute segment>
                END
```

Functions can be considered as undefined operators. Therefore a function has a single output, its name, which is returned as a value and a number of input variables which have to be returned unchanged to the program.

```
<function> ::= FUNCTION <name><input vars>;
               <local variables>;
               <type declaration>
               <structure declaration>
               BEGIN
                  <compute segment>
               END
```

Functions are discussed here yet they will not be considered as such in the system. However, as mentioned before, if they are used, they can be transformed into procedures during preprocessing. For example:

```
Number = result + SQR(result);
```

could be transformed into

```
SQR(result;;sqr);
number = result + sqr;
```

during preprocessing. In this case a new variable is created (sqr). Supporting function directly in the system could be performed by creating a new section in the execution language representation. This section or table would hold the operators and functions used in this process, and would be accessed when processing a statement. However this seems to offer more overhead than transforming functions into procedures.

Appendix B

THE EXECUTION LANGUAGE

A program or procedure in virtual language is transformed into an execution language program, by the preprocessing function. The steps required by this function are described in chapter five. An execution language program or procedure is build up of the following four sections.

```
<E.L. program> ::= <info section>
                   <control segment>
                   <statement segment>
                   <data segment>
```

In this appendix, the syntax of these sections is given using backus normal form notation.

Information Section

The information section contains the pointers necessary to access the required tables and sections. This section is required by the simulation program.

```
<info section> ::= <name>
                   <pointer 1>
                   <pointer 2>
                   <pointer 3>
                   <local end>
```

<name> ::= ... the name of the program

<pointer 1> ::= ... points to the start of the statement descriptor table

<pointer 2> ::= ... points to the start of the statement section

<pointer 3> ::= ... points to the start of the data descriptor table

<local end> ::= ... is equal to the number of elements in the local section of this program

Control Segment

The control segment contains the descriptors of the executable statements plus the control word that dictates the flow of these descriptors. It also contains the descriptor representing the global variables that are accessed by this segment.

<control segment> ::= <parameter section>
<control string>

The parameter section contains a list of the formal parameters.

<parameter section> ::= 0|
1 PARA
<formal parameters>

<formal parameters> ::= <tag bit><data descriptor>|
<tag bit><data descriptor>
<formal parameters>

<data descriptor> ::= ... a number that points to an entry in the data descriptor table

<control string> ::= 0|
<tag bit><word>
<control string>

<tag bit> ::= 0 | 1

<word> ::= <control word>|
<statement descriptor>|
<displacement>

<control word> ::= IF | ELSE | ENDIF | WHILE | ENDWHILE

<statement descriptor> ::= ... a number that points to an entry in the statement descriptor table

<displacement> ::= ... a number that is added or substracted to/from the program counter when a branch occurs.

Statement Segment

The statement segment is made up of the statement descriptor table and the statement section.

<statement segment> ::= <statement descriptor table>
<statement section>

The statement descriptor table contains information about all the executable statement in the program or procedure.

<statement descriptor table> ::= <statement desc. elements>

<statement desc. elements> ::= 0|
<class><location>
<statement desc. elements>

<class> ::= ... the class of the statement (ASSIGN, PROCESS, LOGIC)

<location> ::= ... where this statement can be found in the statement section. This address is a relative address.

The statement section holds the actual statement that must be executed by the system.

<statement section> ::= 0|
<executable statement>
<statement section>

<executable statement> ::= <assignment>|
<logic check>|
<procedure call>

<assignment> ::= 0|
<tag bit><word>
<assignment>

<word> ::= <operator>|
<data descriptor>

<operator> ::= ... a valid operator described in the virtual language

<logic check> ::= <assignment>

<procedure call> ::= 1 <name>
<actual parameters>
1 S

<actual parameters> ::= 0|
<tag bit><data descriptor>
<actual parameters>

Data Segment

The data segment is composed of the data descriptor table and the data space. The data space contains the actual values of the variables.

<data segment> ::= <data descriptor table>
<data space>

The data descriptor table contains the information about a variable such as its type and class.

<data descriptor table> ::= <data descriptor elements>

<data descriptor elements> ::= 0|
<class><type><location>
<data descriptor elements>

<class> ::= ... the class of the variable (CONST, LOCAL, INPUT, OUTPUT, INOUT)

<type> ::= ... the type of the variable (integer, boolean, real ...)

<location> ::= ... where this variable can be found in the data space. This is also a relative address.

Appendix C

PREPROCESSOR SIMULATION

In order to show that an execution language program can be obtained in a straightforward fashion from a virtual language program, due to the close relation between both languages, a simulation using the language Pascal was performed.

This program (PREPROCESSOR) takes an input source program (file VLFILE) written in the virtual language (described in appendix A) and translates it into an execution language object program (file ELFILE). The execution language representation is described in details in chapter 4, and in appendix B.

A description of the actual program written in Pascal will now follow. A few examples of its operation are given in appendix E.

```
PROGRAM PREPROCESSOR(INPUT/, VLFILE, ELFILE, OUTPUT);
```

The main segment of this program is first described. It is responsible for implementing the steps required by the preprocessing function. All the procedures called are described in details further on.

```
BEGIN
   RESET(VLFILE);
   REWRITE(ELFILE);
```

These two Pascal procedures are used to preprared file VLFILE for reading and file ELFILE for writing.

```
   INITVAL;
```

This procedure is required to initialize some of the pointers and counters used in the simulation program.

```
   SKIPBLANK;
```

This procedure is called when the next non blank token is requested from file VLFILE. A token in a virtual language program can be:

1. a delemiter
   ; | , | (|) | b | :

2. an operator
   * | - | + | / | < | > | >= | <= | <> | = | AND | NOT | OR | XOR | EQU

3. an identifier

4. a control word
   BEGIN | END | IF | THEN | ELSE | ENDIF | WHILE | DO | ENDWHILE

5. a constant
   TRUE | FALSE | a number

```
REPEAT
   SAVENAME;
```

This procedure locates the name of the program and saves it.

```
GLOBALDEC;
```

This procedure places the global variables in the symbol table. The symbol table holds all information about the variables and is used to build the data descriptor table.

```
LOCALDEC;
```

This procedure places the local variables in the symbol table.

```
DATATYPE;
```

This procedure assigns the type to the various variables in the symbol table.

```
DATALOCATION;
```

Procedure DATALOCATION is used to add the information about the location of the data in the data space, in the symbol table.

```
COMPUTESEG;
```

The previous procedures implemented step 1 of the preprocessing function. Procedure COMPUTESEG performs step 2, which manipulates the compute segment. This procedure is the most complex since it must fill the control string, the statement descriptor table, and the statement section from the information contained in the compute segment of a virtual language source program or procedure.

```
FINISH;
```

The FINISH procedure writes the different sections of the execution language representation, in the proper format on file ELFILE. This file can then be directly executed by the simulation representing the task manager, which is described in appendix D.

```
      INITVAL;
      SKIPBLANK;
   UNTIL TOKEN = 'STOP    ';
END. {PREPROCESSOR}
```

This segment is used to repeat the preprocessing steps for all programs and procedures included in file PCFILE. Reading in a program name equal to STOP, halts the preprocessing.

The main program to be preprocessed is placed first in the file, followed by the procedures that are called. Token STOP is placed after the last procedure to halt preprocessing.

The following variables are required for the proper operation of the simulation. The four following arrays represent the sections of the execution language program that are generated by program PREPROCESSOR.

DATABLE - Represents the symbol table used in the preprocessing function, which is used to build the data descriptor table. This table is an array of records which have the following fields:

- (name) - name of the variable
- (class) - class of the variable (INPUT, OUTPUT, INOUT, LOCAL)
- (sorte) - type (integer, etc.)
- (position) - a relative address pointing to a position in the local section of the data space.

A record in this array is located using pointer DTPOINT.

CONTABLE - The control segment is also an array of records having the following fields:

- (tagbit) - a tag bit is associated with each word.
- (progcont) - holds a control word when (tagbit) = 1
- (desctor) - holds a displacement or a statement descriptor when (tagbit) = 0.

An element in this array is accessed using pointer CSPOINT.

STASPACE - The statement section is an array of records with the same fields as CONTABLE.

- (tagbit) - a tag bit is associated with each word.
- (progcont) - holds an operator when (tagbit) = 1

(desctor) - holds a descriptor for a variable when (tagbit) = 0

It is accessed using pointer PLACE.

STATABLE - The statement descriptor table is an array of records which contains the following fields.

(class) - class of the statement (PROCESS, ASSIGN, or LOGIC).

(location) - where this statement can be found in the statement section (STATSPACE)

An element in this array is accessed using pointer STPOINT.

The other variables will be described when encountered. The following is the variable declaration segment of program PREPROCESSOR.

```
CONST
   MAXLENGTH = 8;
   MAXDEPTH = 20;
   MAXSIZE = 100;
   MAXBLOCK = 80;

TYPE
   STRING = PACKED ARRAY(.1..MAXLENGTH.) OF CHAR;
   STACK = ARRAY(.1..MAXDEPTH.) OF INTEGER;

   DATADESC =
      RECORD
         NAME : STRING;
         CLASS : STRING;
         SORTE : STRING;
         POSITION : INTEGER;
      END;

   CONTROLX =
      RECORD
         CONTBIT : CHAR;
         PROGCONT : STRING;
         DESCTOR : INTEGER;
      END;

   STATDESC =
      RECORD
         CLASS : STRING;
         LOCATION : INTEGER;
      END;

   CWORDTYPE = (FIRST, SECOND, THIRD, FOURTH, FIFTH, SIXTH);

VAR
   I, VALUE, DTPOINT, LOCALPOINT, CSPOINT : INTEGER;
   PLACE, TOPS, POINTER1, POINTER2, POINTER3, : INTEGER;
   STPOINT, OPTOP, TEMP1 = INTEGER;
   SWITCH, FOUND, HIGHER : BOOLEAN;
```

```
   CARD : CHAR;
   VLFILE, ELFILE : TEXT;
   TOKEN, NAME, CLASS, LIMITE : STRING;
   DATABLE : ARRAY(.1..MAXSIZE.) OF DATADESC;
   CONTABLE : ARRAY(.1..MAXSIZE.) OF CONTROLX;
   STATABLE : ARRAY(.1..MAXSIZE.) OF STATDESC;
   CWORD : CWORDTYPE;
   STATSPACE : ARRAY(.1..MAXSIZE.) OF CONTROLX;
   COSTACK : STACK;
   OPSTACK : ARRAY(.1..MAXDEPTH.) OF STRING;
```

The interaction between the main program and the various procedures called is given in the following flowchart, which represent the preprocessing of a single virtual language program or procedure. The various procedure called in the simulation, are described following this figure.

```
PROCEDURE ERROR(NUMBER : INTEGER);
```

This procedure was mostly used in the debugging stage, it is called when an improper token is read. For example it will be called in some cases when a delimiter is not found.

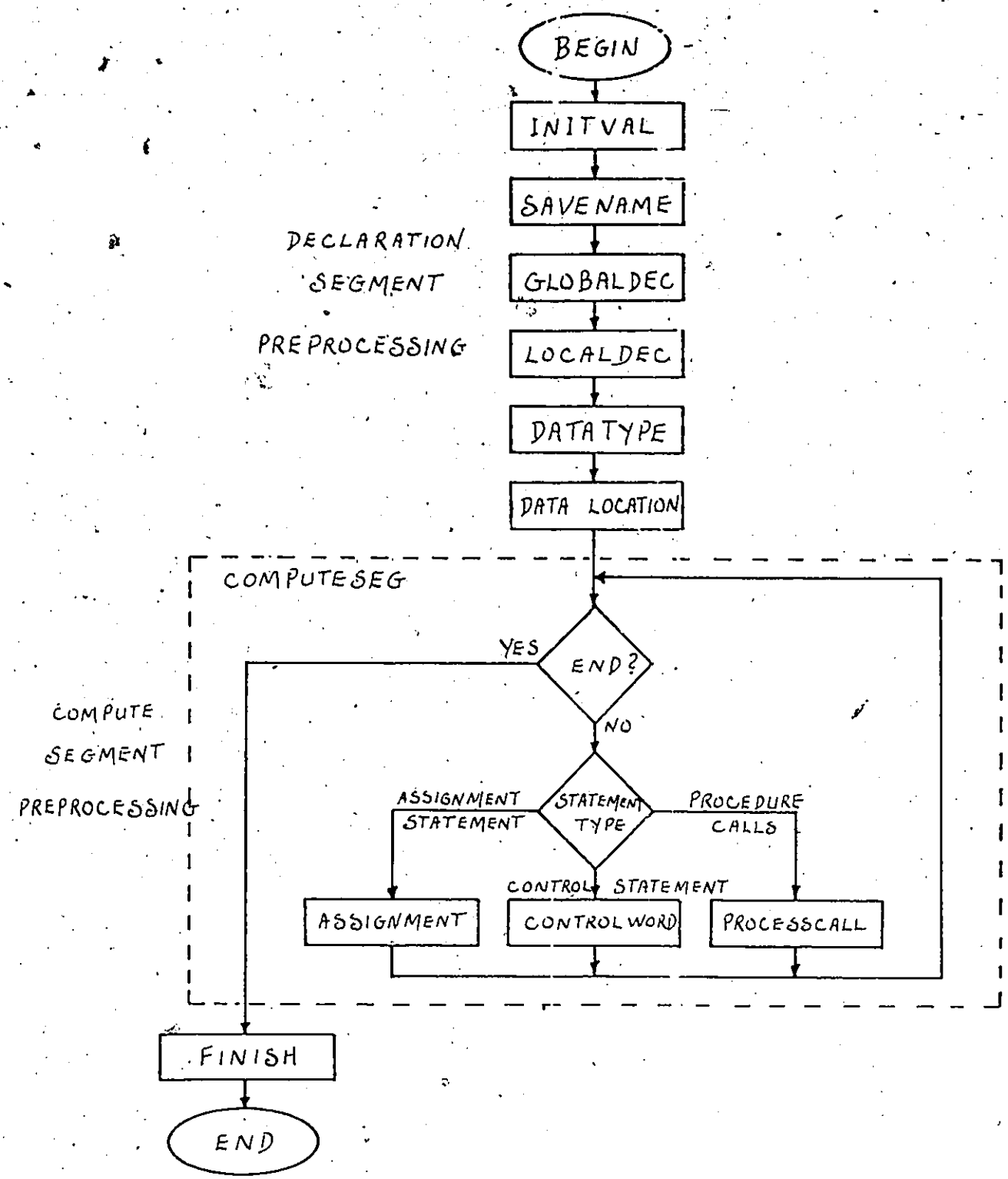
```
BEGIN
   WRITELN(' ERROR OCCURED NUMBER ', NUMBER)
END; {ERROR}
```

```
PROCEDURE INITVAL;
```

Procedure INITVAL resets the pointers for the data descriptor table (DTPOINT), the statement descriptor table (STPOINT), the control string (CSPOINT), and the statement section (PLACE). It also resets the stack pointer used in the reverse polish transformation (OPTOP) and the pointer used to store the position of a displacement in the control string (TOPS).

```
BEGIN
   LINE := 1;
   DTPOINT := 1;
   CSPOINT := 1;
   STPOINT := 1;
   PLACE := 1;
   TOPS := 1;
   OPTOP := 1;
   SWITCH := FALSE;
END; {INITVAL}
```

```
PROCEDURE SKIPBLANK;
```



Procedure skipblank returns the first nonblank token read. It obtains a token using procedure lexical.

```
BEGIN
   REPEAT
      LEXICAL(TOKEN);
   UNTIL TOKEN <>  '          ';
END; {SKIPBLANK}
```

PROCEDURE LEXICAL(VAR TOKEN : STRING);

This procedure scans the virtual language source program character by character (using var. CARD) and assembles them into tokens. When a token is found it is placed in variable TOKEN. Tokens larger than 8 characters are truncated. All token in the file must be seperated by delimiters. The presence of a delimiter is found using function SEPERATOR. This procedure performs the steps associated with the lexical analysis phase.

```
VAR
   SYMBOL : STRING;

PROCEDURE GETTOKEN;

VAR
   J, I :INTEGER;

BEGIN
   I := 1;
   REPEAT
      READ(VLFILE, CARD);
      IF NOT SEPERATOR(CARD) THEN
         BEGIN
            SYMBOL(.I.) := CARD;
            I := I + 1;
         END
      ELSE
         BEGIN
            FOR J := I TO 8 DO
               SYMBOL(.J.) := ' ';
               I := 9
         END;
   UNTIL I = 9;
   WHILE NOT SEPERATOR(CARD) DO
      READ(VLFILE, CARD);
END; {GETTOKEN}


BEGIN
   IF NOT SWITCH THEN
      BEGIN
         GETTOKEN;
         IF SYMBOL = '          ' THEN
            BEGIN
               SYMBOL(.1.) := CARD;
```

```
            TOKEN := SYMBOL;
          END
       ELSE
          BEGIN
             TOKEN := SYMBOL;
             SWITCH := TRUE
          END
    END
 ELSE
    BEGIN
       SYMBOL := '          ';
       SYMBOL(.1.) := CARD;
       TOKEN := SYMBOL;
       SWITCH := FALSE
    END;
END; {LEXICAL}
```

```
FUNCTION SEPERATOR(CARD : CHAR) : BOOLEAN;
```

This function will indicate if the character read in is a delimiter.

```
BEGIN
   IF CARD = ';' THEN
      SEPERATOR := TRUE
   ELSE IF CARD = ',' THEN
      SEPERATOR := TRUE
   ELSE IF CARD = ' ' THEN
      SEPERATOR := TRUE
   ELSE IF CARD = '(' THEN
      SEPERATOR := TRUE
   ELSE IF CARD = ')' THEN
      SEPERATOR := TRUE
   ELSE SEPERATOR := FALSE;
END; {SEPERATOR}
```

```
PROCEDURE SAVENAME;
```

The first token read must be the name of the program. This name will be placed in the information section of the execution language representation. It is placed in variable (NAME).

```
BEGIN
   NAME := TOKEN;
END; {SAVENAME}
```

```
PROCEDURE GLOBALDEC;
```

This procedure reads in the global variables which are used by this program or procedure. A place is reserved in the symbol table (DATABLE) for each of these variables. If global variables are present in the program or procedure, then the parameter section of the control string is build up. This is done by calling procedure PARAMETER.

```
VAR
   SYMBOL : STRING;

BEGIN
   SKIPBLANK;
   IF TOKEN = '(       ' THEN
      BEGIN
         CONTABLE(.CSPOINT.).CONTBIT := '1';
         CONTABLE(.CSPOINT.).PROGCONT := 'PARA    ';
         CSPOINT := CSPOINT + 1;
         SKIPBLANK;
         WHILE TOKEN <> ';       ' DO
            BEGIN
               IF TOKEN = ',       ' THEN
                  SKIPBLANK
               ELSE
                  BEGIN
                     DATABLE(.DTPOINT.).NAME := TOKEN;
                     DATABLE(.DTPOINT.).CLASS := 'INPUT   ';
                     DTPOINT := DTPOINT + 1;
                     PARAMETER;
                     SKIPBLANK;
                  END;
            END;
         SKIPBLANK;
         WHILE TOKEN <> ';       ' DO
            BEGIN
               IF TOKEN = ',       ' THEN
                  SKIPBLANK
               ELSE
                  BEGIN
                     DATABLE(.DTPOINT.).NAME := TOKEN;
                     DATABLE(.DTPOINT.).CLASS := 'INOUT   ';
                     DTPOINT := DTPOINT + 1;
                     PARAMETER;
                     SKIPBLANK;
                  END;
            END;
         SKIPBLANK;
         WHILE TOKEN <> ')       ' DO
            BEGIN
               IF TOKEN = ',       ' THEN
                  SKIPBLANK
               ELSE
                  BEGIN
                     DATABLE(.DTPOINT.).NAME := TOKEN;
                     DATABLE(.DTPOINT.).CLASS := 'OUTPUT  ';
```

```
                    DTPOINT := DTPOINT + 1;
                    PARAMETER;
                    SKIPBLANK;
                 END;
           END;
        SKIPBLANK;
     END;
  IF TOKEN <> ';       ' THEN
     ERROR(1);
END; {GLOBALDEC}
```

```
PROCEDURE PARAMETER;
```

Procedure PARAMETER builds up the parameter section of the control segment when global variables are present. If global variables are not accessed, then the parameter section is not built, therefore control word PARA is not included in the control segment.

```
BEGIN
  CONTABLE(.CSPOINT.).CONTBIT := '0';
  CONTABLE(.CSPOINT.).DESCTOR := DTPOINT -2;
  CSPOINT := CSPOINT + 1;
END; {PARAMETER}
```

```
PROCEDURE LOCALDEC;
```

Filing the symbol table with the local variables is the responsibility of this procedure.

```
BEGIN
  SKIPBLANK;
  IF TOKEN <> '(       ' THEN
     BEGIN
        WHILE TOKEN <> ';       ' DO
           BEGIN
              IF TOKEN = ',       ' THEN
                 SKIPBLANK
              ELSE
                 BEGIN
                    DATABLE(.DTPOINT.).NAME := TOKEN;
                    DATABLE(.DTPOINT.).CLASS := 'LOCAL   ';
                    DTPOINT := DTPOINT + 1;
                    SKIPBLANK
                 END;
           END;
        SKIPBLANK;
     END;
END; {LOCALDEC}
```

```
PROCEDURE DATATYPE;
```

Procedure DATATYPE is called when procedure LOCALDEC is terminated. This procedure will assign the various types to the variables in the symbol table, by scanning the type declaration statements. Variable CLASS holds the type of the variables read in.

```
VAR
   SYMBOL : STRING;

BEGIN
   WHILE TOKEN = '(        ' DO
      BEGIN
         SKIPBLANK;
         CLASS := TOKEN;
         SKIPBLANK;
         IF TOKEN <> ':        ' THEN
            ERROR(5);
         SKIPBLANK;
         REPEAT
            LOCATE(VALUE);
            IF NOT FOUND THEN
               ERROR(444);
            DATABLE(.VALUE.).SORTE := CLASS;
            SKIPBLANK;
            SYMBOL := TOKEN;
            SKIPBLANK;
         UNTIL SYMBOL = ')        ';
      END;
END; {DATATYPE}
```

PROCEDURE LOCATE(VAR VALUE : INTEGER);

Procedure LOCATE will find an entry in the symbol table corresponding to the token read in. It is used by procedure DATATYPE to assign the various type to the right variables. If one is not found then variable FOUND remains false. Variable VALUE will hold the position of the variable in the symbol table.

```
VAR
   I : INTEGER;

BEGIN
   FOUND := FALSE;
   I := 1;
   FOR I := 1 TO DTPOINT - 1 DO
      BEGIN
         IF DATABLE(.I.).NAME = TOKEN THEN
            BEGIN
               VALUE := I;
               FOUND := TRUE;
            END
      END;
```

```
END; {LOCATE}
```

PROCEDURE DATALOCATION;

Procedure DATALOCATION will reserve a place for all the variables contained in the symbol table, in the data space. The size of the local section for this program or procedure is also calculated and placed in variable LOCALPOINT. This variable is required to update the local pointer during a procedure call and return. The value of this variable will be placed in the information section.

```
VAR
   I, X : INTEGER;

BEGIN
   VALUE := 1;
   FOR I := 1 TO DTPOINT - 1 DO
      BEGIN
         IF DATABLE(.I.).SORTE = 'INTEGER ' THEN
            BEGIN
               DATABLE(.I.).POSITION := VALUE;
               X := 1;
            END
         VALUE := VALUE + X;
      END;
   LOCALPOINT := VALUE - X;
END; {DATALOCATION}
```

PROCEDURE COMPUTESEG;

This procedure manipulates the compute segment of a virtual language program, in order to fill the control string, the statement descriptor table and the statement section. The first token read in must be BEGIN, an error is flagged if it is not located. Procedure COMPUTESEG is exited when token END is located, indicating that the compute segment has been totally preprocessed.

A token read from this procedure will point to one of three types of statements. Procedure LOCATE is used to see if the token exist in the symbol table. If it does then this imply that this statement is an assignment statement, and that it should be processed by procedure ASSIGNMENT. If not we check if this token corresponds to a control word using function CONTWORD. If it does correspond to one, then this control statement is processed using procedure CONTROLWORD. If it is not identified at this point, then it is considered as a procedure call statement, which is processed using procedure PROCESSCALL.

```
BEGIN
```

```
   IF TOKEN <> 'BEGIN    ' THEN
      ERROR(555);
   CONTABLE(.CSPOINT.).CONTBIT := '1';
   CONTABLE(.CSPOINT.).PROGCONT := TOKEN;
   CSPOINT := CSPOINT + 1;
   SKIPBLANK;
   WHILE TOKEN <> 'END      ' DO
      BEGIN
        LOCATE(VALUE);
        IF FOUND THEN
           ASSIGNMENT
        ELSE IF CONTWORD THEN
           CONTROLWORD
        ELSE PROCESSCALL;
        SKIPBLANK;
      END;
   CONTABLE(.CSPOINT.).CONTBIT := '1';
   CONTABLE(.CSPOINT.).PROGCONT := TOKEN;
   CSPOINT := CSPOINT + 1;
END; {COMPUTESEG}
```

```
PROCEDURE ASSIGNMENT;
```

Procedure ASSIGNMENT puts a descriptor in the control string (CONTABLE), representing the assignment statement. Then the statement descriptor table is filled with information about this statement (class of statement = ASSIGN, and location of this statement in the statement section). Procedure RPOLISH is then called to transform this statement in a reverse polish notation, which is placed in the statement section.

```
BEGIN
   STATABLE(.STPOINT.).CLASS := 'ASSIGN  ';
   STATABLE(.STPOINT.).LOCATION := PLACE - 1;
   STPOINT := STPOINT + 1;
   CONTABLE(.CSPOINT.).CONTBIT := '0';
   CONTABLE(.CSPOINT.).DESCTOR := STPOINT - 2;
   CSPOINT := CSPOINT + 1;
   TEMP1 := VALUE;
   SKIPBLANK;
   IF TOKEN <> '=        ' THEN
      ERROR(LINE);
   LIMITE := ';        ';
   RPOLISH;
   VALUE := TEMP1;
   STATSPACE(.PLACE.).CONTBIT := '0';
   STATSPACE(.PLACE.).DESCTOR := VALUE - 1;
   PLACE := PLACE + 1;
   STATSPACE(.PLACE.).CONTBIT := '1';
   STATSPACE(.PLACE.).PROGCONT := ':       ';
   PLACE := PLACE + 1;
   STATSPACE(.PLACE.).CONTBIT := '1';
```

```
    STATSPACE(.PLACE.).PROGCONT := 'S      ';
    PLACE := PLACE + 1;
END; {ASSIGNMENT}
```

PROCEDURE RPOLISH;

Procedure RPOLISH transforms an expression written in an infix notation into one written in a reverse polish notation (suffix notation). This translation is performed using a stack for the operators (OPSTACK). The infix notation is read in from file VLFILE and the translated string is placed in the statement section (STATSPACE). The algorithm for this transformation is:

1) If an identifier for a variable is found its descriptor is placed in the statement section (STATSPACE).

2) Similarly if a constant is found, information about it is placed in the data descriptor table (DATABLE), then a descriptor for this constant is placed in the statement section. In this simulation the actual value of the constants are kept in the data descriptor table.

3) Two other kinds of token can be encountered, they are:
   - blanks
   - operators

Due to the way procedure LEXICAL was written, blanks must be used to seperate indentifier, constants and operators. Left and right parantheses are used to remove ambiguity and modify the order of evaluation. If two operators have the same precedence, the expression is evaluated from left to right.

When an operator is encountered it must inquire about the top element of the stack (OPSTACK). The following actions are performed:

i) If the stack is empty, the operator is placed on top of the stack.

ii) If the operator is a left paranthese "(", it is placed on top of the stack regardless of the content of the stack.

iii) If the operator has a lower precedence than the one on top of the stack, then the operator on top of the stack is placed in the statement section. This continues until the stack is empty or until the operator on the stack has a lower precedence. Then if the operator is not a right paranthese it is place on top of the stack, if it's one then the stack is popped. In otherword a right paranthese will cancel a left paranthese that was previously pushed on.

Processing of the string continues until the token indicating the end (LIMITE) of the expression is encountered. It can be ";", "THEN", or "DO" depending on the expression processed.

```
BEGIN
```

```
SKIPBLANK;
WHILE TOKEN <> LIMITE DO
   BEGIN
      LOCATE(VALUE);
      IF FOUND THEN
         BEGIN
            STATSPACE(.PLACE.).CONTBIT := '0';
            STATSPACE(.PLACE.).DESCTOR := VALUE - 1;
            PLACE := PLACE + 1;
            SKIPBLANK;
         END
      ELSE IF OPERATOR THEN
         BEGIN
            IF OPTOP = 1 THEN
               BEGIN
                  OPSTACK(.OPTOP.) := TOKEN;
                  OPTOP := OPTOP + 1;
               END
            ELSE
               BEGIN
                  IF TOKEN(.1.) = '(' THEN
                     BEGIN
                        OPSTACK(.OPTOP.) := TOKEN;
                        OPTOP := OPTOP + 1;
                     END
                  ELSE
                     BEGIN
                        PRECEDENCE(TOKEN,OPSTACK(.OPTOP-1.));
                        WHILE (NOT HIGHER) DO
                           BEGIN
                              STATSPACE(.PLACE.).CONTBIT:='1';
                              OPTOP := OPTOP - 1;
                              STATSPACE(.PLACE.).PROGCONT
                                          :=OPSTACK(.OPTOP.);
                              PLACE := PLACE + 1;
                              IF OPTOP = 1 THEN
                                 HIGHER := TRUE
                              ELSE
                                PRECEDENCE(TOKEN,OPSTACK(.OPTOP-1.));
                           END;
                        IF TOKEN(.1.) = ')' THEN
                           OPTOP := OPTOP - 1
                        ELSE
                           BEGIN
                              OPSTACK(.OPTOP.) := TOKEN;
                              OPTOP := OPTOP + 1;
                           END;
                     END
               END
            SKIPBLANK;
         END
      ELSE
         BEGIN
            STATSPACE(.PLACE.).CONTBIT := '0';
```

```
                STATSPACE(.PLACE.).DESCTOR := DTPOINT - 1;
                PLACE := PLACE + 1;
                DATABLE(.DTPOINT.).NAME := TOKEN;
                DATABLE(.DTPOINT.).CLASS := 'CONST   ';
                DATABLE(.DTPOINT.).SORTE := 'INTEGER ';
                DTPOINT := DTPOINT + 1;
                SKIPBLANK;
             END;
      END;
   WHILE OPTOP <> 1 DO
      BEGIN
         STATSPACE(.PLACE.).CONTBIT := '1';
         STATSPACE(.PLACE.).PROGCONT:=OPSTACK(.OPTOP-1.);
         PLACE := PLACE + 1;
         OPTOP := OPTOP - 1;
      END;
END; {RPOLISH}


FUNCTION OPERATOR: BOOLEAN;
```

This function is used to identify an operator in an expression. OPERATOR becomes true if an operator is found. This is necessary to distinguish betweem variable, constants, and operators.

```
BEGIN
   IF TOKEN(.1.) = '+' THEN
      OPERATOR := TRUE
   ELSE IF TOKEN(.1.) = '-' THEN
      OPERATOR := TRUE
   ELSE IF TOKEN(.1.) = '*' THEN
      OPERATOR := TRUE
   ELSE IF TOKEN(.1.) = '/' THEN
      OPERATOR := TRUE
   ELSE IF TOKEN(.1.) = '=' THEN
      OPERATOR := TRUE
   ELSE IF TOKEN(.1.) = '>' THEN
      OPERATOR := TRUE
   ELSE IF TOKEN(.1.) = '<' THEN
      OPERATOR := TRUE
   ELSE IF TOKEN(.1.) = '(' THEN
      OPERATOR := TRUE
   ELSE IF TOKEN(.1.) = ')' THEN
      OPERATOR := TRUE
   ELSE OPERATOR := FALSE;
END; {OPERATOR}


PROCEDURE PRECEDENCE(INCOMING, TOPSTACK : STRING);
```

This procedure will compare the precedence of two operators. Variable HIGHER will become true if the precedence of

the operator read in is lower than the one of the operator on the stack (OPSTACK).

```
BEGIN
   IF LEVEL(INCOMING) < LEVEL(TOPSTACK) THEN
      HIGHER := TRUE
   ELSE HIGHER := FALSE;
END; {PRECEDENCE}
```

```
FUNCTION LEVEL(TEMPOR : STRING): INTEGER;
```

Function LEVEL will return the precedence of an operator.

```
BEGIN
   IF (TEMPOR(.1.) = '*') OR (TEMPOR(.1.) = '/') THEN
      LEVEL := 1
   ELSE IF (TEMPOR(.1.) = '+') OR (TEMPOR(.1.) = '-') THEN
      LEVEL := 2
   ELSE IF(TEMPOR='<       ')OR(TEMPOR='<=      ')THEN
      LEVEL := 3
   ELSE IF(TEMPOR='<>      ')OR(TEMPOR='=       ')THEN
      LEVEL := 3
   ELSE IF(TEMPOR='>       ')OR(TEMPOR='>=      ')THEN
      LEVEL := 3
   ELSE IF (TEMPOR = 'NOT     ') THEN
      LEVEL := 4
   ELSE IF (TEMPOR = 'AND     ') THEN
      LEVEL := 5
   ELSE IF (TEMPOR = 'OR      ') THEN
      LEVEL := 6
   ELSE IF (TEMPOR(.1.) = ')') THEN
      LEVEL := 7
   ELSE IF (TEMPOR(.1.) = '(') THEN
      LEVEL := 8
END; {LEVEL}
```

```
FUNCTION CONTWORD: BOOLEAN;
```

If the statement is not an assignment statement, then this function is called to see if it's a control statement. This function will compare the token with all the control words presently available. Variable CWORD will hold a scalar value identifying the control word that was found.

```
BEGIN
   IF TOKEN = 'IF      ' THEN
      BEGIN
         CONTWORD := TRUE;
         CWORD := FIRST
      END
   ELSE IF TOKEN = 'THEN    ' THEN
      BEGIN
```

```
            CONTWORD := TRUE;
            CWORD := SECOND
         END
      ELSE IF TOKEN = 'ELSE    ' THEN
         BEGIN
            CONTWORD := TRUE;
            CWORD := THIRD
         END
      ELSE IF TOKEN = 'ENDIF   ' THEN
         BEGIN
            CONTWORD := TRUE;
            CWORD := FOURTH
         END
      ELSE IF TOKEN = 'WHILE   ' THEN
         BEGIN
            CONTWORD := TRUE;
            CWORD := FIFTH
         END
      ELSE IF TOKEN = 'ENDWHILE' THEN
         BEGIN
            CONTWORD := TRUE;
            CWORD := SIXTH
         END
         ELSE CONTWORD := FALSE;
END; {CONTWORD}
```

PROCEDURE CONTROLWORD;

If the statement is identified as a control word using the previous function, then procedure CONTROLWORD is called. The following control words are valid:

- IF, ELSE, ENDIF, WHILE, ENDWHILE

A case statement is used to initiate the action required by the different control words. For all control words the control string (CONTABLE), the statement descriptor table (STATABLE), and the statement section (STATSPACE) are updated accordingly. These control words are placed in the control string, with their tag bit equal to one. For IFs and WHILEs control words, the logic expression that follows is transformed in a reverse polish notation, using procedure RPOLIH. It is then placed in the statement section. The statement descriptor table is filled with information about this expressions, such as its class (LOGIC), and its location in the statement section. Furthermore, a descriptor representing this expression is placed in the control string. In order to calculate the displacement that must be introduced after the control word IF, ELSE, WHILE, and ENDWHILE, a stack (COSTACK) is used to save the control string pointer (CSPOINT).

```
BEGIN
   CONTABLE(.CSPOINT.).CONTBIT := '1';
   CONTABLE(.CSPOINT.).PROGCONT := TOKEN;
```

```
CSPOINT := CSPOINT + 1;

CASE CWORD OF
   FIRST:{IF}
      BEGIN
         STATABLE(.STPOINT.).CLASS := 'LOGIC   ';
         STATABLE(.STPOINT.).LOCATION := PLACE - 1;
         STPOINT := STPOINT + 1;
         CONTABLE(.CSPOINT.).CONTBIT := '0';
         CONTABLE(.CSPOINT.).DESCTOR := STPOINT - 2;
         CSPOINT := CSPOINT + 1;
         CONTABLE(.CSPOINT.).CONTBIT := '0';
         COSTACK(.TOPS.) := CSPOINT;
         TOPS := TOPS + 1;
         CSPOINT := CSPOINT + 1;
         LIMITE := 'THEN    ';
         RPOLISH;
         STATSPACE(.PLACE.).CONTBIT := '1';
         STATSPACE(.PLACE.).PROGCONT := 'S       ';
         PLACE := PLACE + 1;
      END; {IF}

   SECOND:{THEN}
      ERROR(LINE);

   THIRD:{ELSE}
      BEGIN
         TOPS := TOPS - 1;
         CONTABLE(.COSTACK(.TOPS.).).DESCTOR
                        :=CSPOINT-COSTACK(.TOPS.)-2;
         COSTACK(.TOPS.) := CSPOINT;
         TOPS := TOPS + 1;
         CONTABLE(.CSPOINT.).CONTBIT := '0';
         CSPOINT := CSPOINT + 1;
      END; {ELSE}

   FOURTH:{ENDIF}
      BEGIN
         TOPS := TOPS - 1;
         CONTABLE(.COSTACK(.TOPS.).).DESCTOR
                        :=CSPOINT-COSTACK(.TOPS.)-2;
      END; {ENDIF}

   FIFTH:{WHILE}
      BEGIN
         STATABLE(.STPOINT.).CLASS := 'LOGIC   ';
         STATABLE(.STPOINT.).LOCATION := PLACE - 1;
         STPOINT := STPOINT + 1;
         CONTABLE(.CSPOINT.).CONTBIT := '0';
         CONTABLE(.CSPOINT.).DESCTOR := STPOINT - 2;
         CSPOINT := CSPOINT + 1;
         CONTABLE(.CSPOINT.).CONTBIT := '0';
         COSTACK(.TOPS.) := CSPOINT;
         TOPS := TOPS + 1;
```

```
            CSPOINT := CSPOINT + 1;
            LIMITE := 'DO      ';
            RPOLISH;
            STATSPACE(.PLACE.).CONTBIT := '1';
            STATSPACE(.PLACE.).PROGCONT := 'S       ';
            PLACE := PLACE + 1;
         END; {WHILE}

      SIXTH:{ENDWHILE}
         BEGIN
            TOPS := TOPS - 1;
            CONTABLE(.COSTACK(.TOPS.).).DESCTOR
                        :=CSPOINT-COSTACK(.TOPS.)-2;
            CONTABLE(.CSPOINT.).CONTBIT := '0';
            CONTABLE(.CSPOINT.).DESCTOR:=CSPOINT-COSTACK(.TOPS.)+3;
            CSPOINT := CSPOINT + 1;
         END; {ENDWHILE}
   END;
END; {CONTROLWORD}

PROCEDURE PROCESSCALL;
```

If the token read in does not correspond to an assigment statement or a control statement, then a procedure call statement is assumed. Procedure PROCESSCALL will first fill the statement descriptor table (STATABLE) with the proper information (type of statement = PROCESS and its location in the statement section), and then place a descriptor in the control string representing this statement. The statement section (STATSPACE) will also be filled with the information about this procedure, which include:

- its name
- location of this procedure (LEVEL) in file ELFILE
- and the actual parameter passed to this procedure.

```
VAR
   SYMBOL : STRING;
   I      : INTEGER;

PROCEDURE STORE;

BEGIN
   LOCATE(VALUE);
   STATSPACE(.PLACE.).CONTBIT := '0';
   STATSPACE(.PLACE.).DESCTOR := VALUE - 1;
   PLACE := PLACE + 1;
   SKIPBLANK;
END; {STORE}


BEGIN
   STATABLE(.STPOINT.).CLASS := 'PROCESS ';
   STATABLE(.STPOINT.).LOCATION := PLACE - 1;
   STPOINT := STPOINT + 1;
```

```
   CONTABLE(.CSPOINT.).CONTBIT := '0';
   CONTABLE(.CSPOINT.).DESCTOR := STPOINT - 2;
   CSPOINT := CSPOINT + 1;
   STATSPACE(.PLACE.).CONTBIT := '1';
   STATSPACE(.PLACE.).PROGCONT := TOKEN;
   PLACE := PLACE + 1;
   STATSPACE(.PLACE.).CONTBIT := '1';
   STATSPACE(.PLACE.).PROGCONT := 'LEVEL    ';
   PLACE := PLACE + 1;
   SKIPBLANK;
   IF TOKEN = '(       ' THEN
      BEGIN
         SKIPBLANK;
         WHILE TOKEN <> ')        ' DO
            BEGIN
               IF(TOKEN = ';        ')
                              OR(TOKEN = ',        ') THEN
                  SKIPBLANK
               ELSE STORE;
            END;
         SKIPBLANK;
      END;
   IF TOKEN <> ';       ' THEN
      ERROR(234);
   STATSPACE(.PLACE.).CONTBIT := '1';
   STATSPACE(.PLACE.).PROGCONT := 'S        ';
   PLACE := PLACE + 1;
END; {PROCESSCALL}
```

PROCEDURE FINISH;

This procedure writes the result of the preprocessing function on file ELFILE. POINTER1, POINTER2, and POINTER3 represent the pointers to the statement descriptor table, the statement section, and the data descriptor table respectively.

```
VAR
   I : INTEGER;

BEGIN
   WRITELN(ELFILE, NAME);
   POINTER1 := CSPOINT + 7;
   POINTER2 := CSPOINT + STPOINT + 9;
   POINTER3 := CSPOINT + STPOINT + PLACE + 11;
   WRITELN(ELFILE, POINTER1);
   WRITELN(ELFILE, POINTER2);
   WRITELN(ELFILE, POINTER3);
   WRITELN(ELFILE, LOCALPOINT);
   FOR I:= 1 TO CSPOINT - 1 DO
      BEGIN
         WRITE(ELFILE, CONTABLE(.I.).CONTBIT);
         IF CONTABLE(.I.).CONTBIT = '1' THEN
```

```
            WRITELN(ELFILE, CONTABLE(.I.).PROGCONT)
         ELSE WRITELN(ELFILE, CONTABLE(.I.).DESCTOR);
      END;
   WRITELN(ELFILE);
   WRITELN(ELFILE, 'STATEMENT DESCRIPTOR TABLE');
   WRITELN(ELFILE);
   FOR I := 1 TO STPOINT - 1 DO
      BEGIN
         WRITE(ELFILE, STATABLE(.I.).CLASS);
         WRITELN(ELFILE, STATABLE(.I.).LOCATION);
      END;
   WRITELN(ELFILE);
   WRITELN(ELFILE, 'STATEMENTS');
   WRITELN(ELFILE);
   FOR I := 1 TO PLACE - 1 DO
      BEGIN
         WRITE(ELFILE, STATSPACE(.I.).CONTBIT);
         IF STATSPACE(.I.).CONTBIT = '1' THEN
            WRITELN(ELFILE, STATSPACE(.I.).PROGCONT)
         ELSE WRITELN(ELFILE, STATSPACE(.I.).DESCTOR);
      END;
   WRITELN(ELFILE);
   WRITELN(ELFILE, 'DATA DESCRIPTOR TABLE');
   WRITELN(ELFILE);
   FOR I:= 1 TO DTPOINT - 1 DO
      BEGIN
         WRITE(ELFILE, DATABLE(.I.).NAME);
         WRITE(ELFILE, DATABLE(.I.).CLASS);
         WRITE(ELFILE, DATABLE(.I.).SORTE);
         IF DATABLE(.I.).CLASS = 'CONST    ' THEN
            WRITELN(ELFILE, DATABLE(.I.).NAME)
         ELSE WRITELN(ELFILE, DATABLE(.I.).POSITION);
      END;
   FOR I:=1 TO MAXBLOCK-(CSPOINT+STPOINT+PLACE+DTPOINT+10) DO
      WRITELN(ELFILE);
END; {FINISH}
```

Appendix D

TASKMANAGER SIMULATION

To understand the processing of an execution language process and to analyse the feasibility of the design, a second simulation was performed.

The simultation program TASKMANAGER executes directly a program written in the execution language from input file ELFILE. The execution language representation is obtained from program preprocessor described in appendix C. The only modification that must be performed to the execution language representation is to add the location of the called procedure, following the name of the procedure in the statement section.

The output of this program is a list of the statements executed. This output is place on file OUTFILE. The execution language procedure READ and WRITE are used to input a value from the keyboard, and output the value of a variable to file OUTFILE respectively.

This program scans the control string and initiates the required actions depending on the content of the word read in. If the tag bit associated with a word in the control string is one then the word is a control word. The following control word can be encountered:
BEGIN, END, IF, ELSE, ENDIF, WHILE, ENDWHILE.

The action requested by each of these control words was described in detail in chapter 4. If the tag bit is zero, then the word can be a displacement or a statement descriptor. Displacement are encountered only after control words. When a descriptor for a statement is located, the statement is sent to be executed.

A description of the program will now follow. The main program is first described, followed by a list of the relevant variables used and a description of the various procedure called.

```
PROGRAM TASKMANAGER(INPUT/,ELFILE,OUTFILE,OUTPUT);
```

The main portion of this program scans the control string and initiates the required action has described in chapter 4.

```
BEGIN
   INITVAL;
```

This procedure will initialize some of the counters and pointers used in this program.

```
   RESET(ELFILE);
   REWRITE(OUTFILE);
```

These 2 pascal procedures are used to prepare file ELFILE for reading and file OUTFILE for writing.

```
   GETPOINTER;
```

This procedure gets the pointer required for this program. That is, it scans the information section that is added to every execution language program and procedure.

```
   READ(ELFILE, TAGBIT);
   PCOUNTER := PCOUNTER + 1;
   IF TAGBIT <> '1'
      THEN ERROR(0);
   READLN(ELFILE, CWORD);
   IF CWORD <> 'BEGIN   'THEN
      ERROR(1);
```

This segment reads in the first word in the control string which must be BEGIN. An error is flagged if this is not the first word encountered. It should be noted that the first program does not access any global variables, since all variables are local to this program.

```
   WHILE NOT EOEXECUT DO
```

Variable EOEXECUT will indicate the end of the processing of the main program.

```
      BEGIN
         STOP := FALSE;
         WHILE NOT STOP DO
```

Variable STOP will indicate the end of the processing of a program or procedure. Therefore when STOP is true we must return to the calling program or procedure. If there is none then EOEXECUT becomes true.

```
            BEGIN
               READ(ELFILE, TAGBIT);
               PCOUNTER := PCOUNTER + 1;
               IF TAGBIT = '1'
                  THEN CONTROLWORD
                  ELSE STATDESC;
```

A word in the control string is either a control word or a descriptor depending on the tag bit associated with it. Procedure CONTROLWORD is called for control words, while procedure STATDESC is called for statement descriptors.

```
            END;
        PROCESSRET;
```

Procedure PROCESSRET will return to the calling program or procedure at the end of the called procedure.

```
     END;
END. {TASKMANAGER}
```

The following variables are required in the program.

Array DATASPACE is the data space that holds the actual value of the variable used in the program. The other variables will be described when encountered.

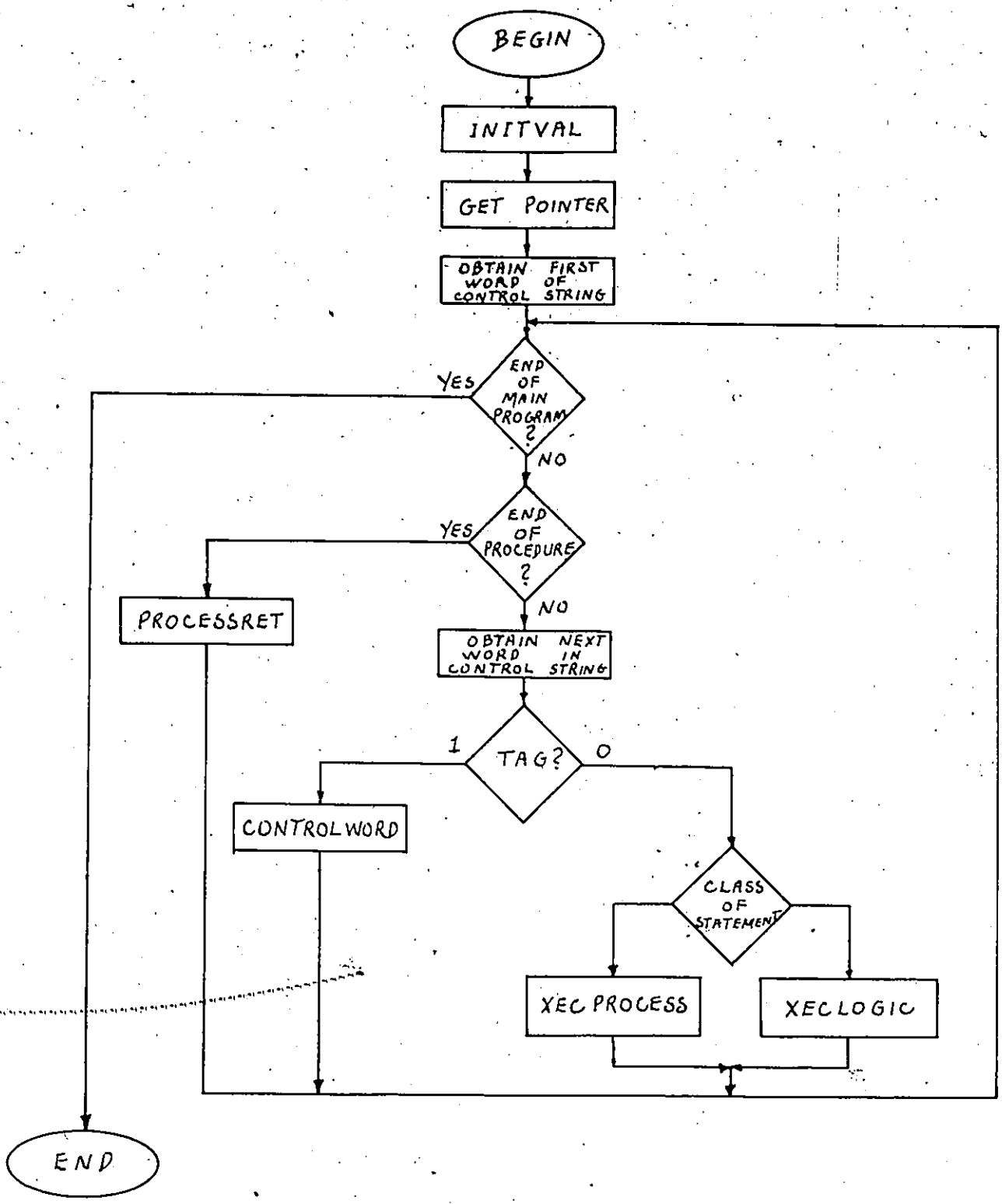
```
CONST
   MAXDEPTH = 25;
   MAXLENGTH = 8;
   MAXBLOCK = 80;

TYPE
   STACK = PACKED ARRAY (.1..MAXDEPTH.) OF BOOLEAN;
   STRING = PACKED ARRAY (.1..MAXLENGTH.) OF CHAR;
   CWORDTYPE=(FIRST,SECOND,THIRD,FOURTH,FIFTH,SIXTH,SEVENTH);
   ARITHST = ARRAY (.1..MAXDEPTH.) OF INTEGER;

VAR
   STPOINTER, PCOUNTER, DISPLACEMENT, ADDRESS : INTEGER;
   STPLOGIC, STPARITH, POINTER3, VALUE, DATA : INTEGER;
   STPCONT, LEVEL, J, LOCALPOINT, LOCALEND : INTEGER;
   POINTER1, POINTER2, DESCRIPTOR : INTEGER;
   TEMPA, QUEPOINT :INTEGER;
   STOP, STATUS, CHECK, EOXPR, EOEXECUT : BOOLEAN;
   CWORD, CLASS, NAME, SORTE : STRING;
   LOGICST, DATABOOL : STACK;
   TAGBIT : CHAR;
   ELFILE, OUTFILE : TEXT;
   CWORDNUM : CWORDTYPE;
   ARITST, DATASPACE, CONTST : ARITHST;
```

The following flowchart depicts the interaction between the various Pascal procedures during the processing of an execution language process. This flowchart is followed by a description of the Pascal procedure called.

```
PROCEDURE ERROR(NUMBER : INTEGER);
```



This procedure was used at the debugging stage to find some transient errors. It is still used in some places to indicate errors in the execution.

```
BEGIN
   WRITE(' ERROR OCCURRED,    NUMBER');
   WRITELN(NUMBER)
END; {ERROR}


PROCEDURE INITVAL;
```

The following value are initialized:

1. The pointer to the status stack (STPOINTER)

2. End Of EXECUTion indicator (EOEXECUT)

3. Location of the starting program segment (LEVEL)

4. The pointer to the top of the control stack (STPCONT)

5. The program counter which points to the next word to be executed in the control string (PCOUNTER)

6. The pointer to the start of the present local section in the data space (LOCALPOINT). Note the global pointer is always considered as being equal to zero, which in a sense points to the first element of array DATASPACE.

```
VAR
   J : INTEGER;

BEGIN
   FOR J := 1 TO MAXDEPTH DO
      DATASPACE(.J.) := 0;
   STPOINTER := 1;
   EOEXECUT := FALSE;
   LEVEL := 0;
   STPCONT := 1;
   PCOUNTER := 0;
   LOCALPOINT := 0;
END; {INITVAL}


PROCEDURE GETPOINTER;
```

The information section must be read before the control segment can be processed. The value read in are:

- name of the program (NAME)
- pointer to the statement descriptor table (POINTER1)
- pointer to the statement section (POINTER2)
- pointer to the data descriptor table (POINTER3)
- size of data space segment required (LOCALEND)

```
BEGIN
```

```
   READLN(ELFILE, NAME);
   READLN(ELFILE, POINTER1);
   READLN(ELFILE, POINTER2);
   READLN(ELFILE, POINTER3);
   READLN(ELFILE, LOCALEND);
END; {GETPOINTER}
```

PROCEDURE CONTROLWORD;

If the tag bit associated with a word in the control string is one then the word corresponds to a control word. Procedure CONTROLWORD will execute the action required by all control words as described in chapter 4. Variable CWORD holds the control word. The action required are very simple and mostly consist of skipping a displacement, or adding it to the program counter.

```
BEGIN
   READLN(ELFILE, CWORD);
   TRANSFORM;
   CASE CWORDNUM OF
      FIRST:{BEGIN}
         ERROR(2);
      SECOND:{END}
            STOP := TRUE;
      THIRD:{IF}
         BEGIN
            LOGICCHECK;
            IF STATUS
               THEN SKIP
               ELSE JUMP
         END;
      FOURTH:{ELSE}
         BEGIN
            IF STATUS
               THEN JUMP
               ELSE SKIP
         END;

      FIFTH:{ENDIF}
         STATUS := TRUE;

      SIXTH:{WHILE}
         BEGIN
            LOGICCHECK;
            IF STATUS
               THEN SKIP
               ELSE JUMP
         END;

      SEVENTH:{ENDWHILE}
         BEGIN
            IF STATUS
```

```
                THEN
                   BEGIN
                      READ(ELFILE, TAGBIT);
                      PCOUNTER := PCOUNTER + 1;
                      IF TAGBIT ='1'
                         THEN ERROR(3);
                      READLN(ELFILE, DISPLACEMENT);
                      PCOUNTER := PCOUNTER + DISPLACEMENT;
                      RESUME;
                   END
                ELSE SKIP;
             STATUS := TRUE;
          END;
   END;
END; {CONTROLWORD}
```

PROCEDURE RESUME;

This procedure is used to return control to the next word in the control string. This is necessary since all segment of the execution program are stored on a sequential file, therefore we must move within this file to access the various sections.

```
VAR
   I : INTEGER;

BEGIN
   RESET(ELFILE);
   FOR I := 1 TO PCOUNTER + 5 + MAXBLOCK * LEVEL DO
      READLN(ELFILE)
END; {RESUME}
```

PROCEDURE TRANSFORM;

Procedure TRANSFORM is used to transform from type character (var. CWORD) to type scalar (var. CWORDNUM) in order to use a case statement in procedure CONTROLWORD.

```
BEGIN
   IF CWORD = 'BEGIN    '
      THEN CWORDNUM := FIRST
   ELSE IF CWORD = 'END      '
      THEN CWORDNUM := SECOND
   ELSE IF CWORD = 'IF       '
      THEN CWORDNUM := THIRD
   ELSE IF CWORD = 'ELSE     '
      THEN CWORDNUM := FOURTH
   ELSE IF CWORD = 'ENDIF    '
      THEN CWORDNUM := FIFTH
   ELSE IF CWORD = 'WHILE    '
      THEN CWORDNUM := SIXTH
```

```
   ELSE IF CWORD = 'ENDWHILE'
      THEN CWORDNUM := SEVENTH
      ELSE ERROR(5)
END; {TRANSFORM}
```

PROCEDURE LOGICCHECK;

Following the IF and WHILE control words there exist a descriptor pointing to a logic check expression. This procedure will execute such an expression. The major part of the expression is executed by procedure XECDESC which is described further on. A status flag (STATUS) is used to save the value (CHECK) returned. This status flag is also manipulated by some control words.

```
VAR
   LOGIC : INTEGER;

BEGIN
   READ(ELFILE, TAGBIT);
   PCOUNTER := PCOUNTER + 1;
   IF TAGBIT = '1'
      THEN ERROR(4);
   READLN(ELFILE, LOGIC);
   XECDESC( 1, LOGIC, CHECK);
   STATUS := CHECK;
END; {LOGICCHECK}
```

PROCEDURE SKIP;

This procedure is used to skip one word in the control string. It is used to skip an unwanted displacement.

```
BEGIN
   READ(ELFILE, TAGBIT);
   PCOUNTER := PCOUNTER + 1;
   IF TAGBIT = '1'
      THEN ERROR(6);
   READLN(ELFILE)
END; {SKIP}
```

PROCEDURE JUMP;

Procedure JUMP will add the present value of the control string pointer (DISPLACEMENT) to the program counter and continue processing from there.

```
VAR
   I : INTEGER;

BEGIN
```

```
   READ(ELFILE, TAGBIT);
   PCOUNTER := PCOUNTER + 1;
   IF TAGBIT = '1'
      THEN ERROR(7);
   READLN(ELFILE, DISPLACEMENT);
   PCOUNTER := PCOUNTER + DISPLACEMENT;
   FOR I := 1 TO DISPLACEMENT DO
      READLN(ELFILE)
END; {JUMP}


PROCEDURE STATDESC;

   Executes a descriptor (DESCRIPTOR)  for an assignment or
procedure call statement.


BEGIN
   READLN(ELFILE, DESCRIPTOR);
   XECDESC( 0, DESCRIPTOR, CHECK)
END; {STATDESC}


PROCEDURE XECDESC(I, DESCRIPTOR : INTEGER;
                  VAR CHECK : BOOLEAN);

   This procedure is used to execute the three kinds of exe-
cutable statements  that can be  encountered in  the control
string.  These executable statements are:
     - logic check expressions
     - assignment statements
     - procedure calls

   Procedure XECLOGIC  will execute the  first two  kinds of
statements,  while procedure XECPROCESS  will execute proce-
dure calls.  Procedure XECLOGIC therefore corresponds to the
actions  required by  the  task  executor,  while  procedure
XECPROCESS correspond to  the action of the  process manage-
ment unit.

VAR
   J : INTEGER;

BEGIN
   RESET(ELFILE);
   FOR J:=1 TO POINTER1 + MAXBLOCK * LEVEL + DESCRIPTOR DO
      READLN(ELFILE);
   READ(ELFILE, CLASS);
   READLN(ELFILE, ADDRESS);
   IF (I=1 ) AND ( CLASS = 'LOGIC   ')
      THEN XECLOGIC(ADDRESS, CHECK, I)
   ELSE IF (I=0 ) AND (CLASS = 'ASSIGN  ')
      THEN XECLOGIC(ADDRESS, CHECK, I)
   ELSE IF (I=0 ) AND (CLASS = 'PROCESS ')
```

```
      THEN XECPROCESS(ADDRESS)
      ELSE ERROR(8)
END; {XECDESC}


PROCEDURE XECLOGIC(VAR ADDRESS : INTEGER;
                   VAR CHECK : BOOLEAN;
                       I : INTEGER);
```

This procedure will execute logic expressions and assignment statements. Since the expressions are written in reverse polish notation, a stack (ARITST) is used for the execution. The pointer to the top of the stack is STPARITH. A logic stack (LOGICST) is used to store logic result, it is accessed using pointer STPLOGIC. Procedure DATUM will push an operand on the stack, while procedure OPERATOR will execute an operation with respect to the top of the stack. Variable ADDRESS will point to the next token in the statement section.

```
VAR
   J : INTEGER;

BEGIN
   STPARITH := 1;
   STPLOGIC := 1;
   EOXPR := FALSE;
   CHECK :=FALSE;
   WHILE NOT EOXPR DO
      BEGIN
         RESET(ELFILE);
         FOR J:=1 TO POINTER2+MAXBLOCK*LEVEL+ADDRESS DO
            READLN(ELFILE);
         READ(ELFILE, TAGBIT);
         ADDRESS := ADDRESS+1;
         IF TAGBIT = '1'
            THEN OPERATOR
            ELSE DATUM;
      END;
   IF I=1
      THEN CHECK := LOGICST(.STPLOGIC-1.);
   RESUME;
END; {XECLOGIC}


PROCEDURE OPERATOR;
```

This procedure will perform the proper operation called for by the operator read in. All operations are performed with respect to the top of the stack (ARITST). When operator 'S' is encountered it indicates the end of the expression by setting variable EOXPR true. (note: not all operations available in the virtual language have been implemented)

```
VAR
   DATAOP : PACKED ARRAY (.1..2.) OF CHAR;

BEGIN
   READLN(ELFILE, DATAOP);
   IF DATAOP <> 'S' THEN
      WRITE(OUTFILE, DATAOP);
   IF DATAOP = '+ '
      THEN
         BEGIN
            ARITST(.STPARITH-2.):=ARITST(.STPARITH-2.)
                                  +ARITST(.STPARITH-1.);
            STPARITH := STPARITH - 1
         END
   ELSE IF DATAOP = '- '
      THEN
         BEGIN
            ARITST(.STPARITH-2.):=ARITST(.STPARITH-2.)
                                  -ARITST(.STPARITH-2.);
            STPARITH := STPARITH -1
         END
   ELSE IF DATAOP = ': '
      THEN
         BEGIN
            STPARITH := STPARITH - 1;
            DATASPACE(.VALUE.) := ARITST(.STPARITH-1.);
         END
   ELSE IF DATAOP = '= '
      THEN
         BEGIN
            LOGICST(.STPLOGIC.):=ARITST(.STPARITH-2.)
                                 =ARITST(.STPARITH-1.);
            STPLOGIC := STPLOGIC + 1;
            STPARITH := STPARITH - 2
         END
   ELSE IF DATAOP = 'S ' THEN
      BEGIN
         EOXPR := TRUE;
         WRITELN(OUTFILE, DATAOP);
      END
   ELSE IF DATAOP = '> ' THEN
      BEGIN
         LOGICST(.STPLOGIC.):=ARITST(.STPARITH-2.)
                             >ARITST(.STPARITH-1.);
         STPLOGIC := STPLOGIC + 1;
         STPARITH := STPARITH - 2
      END
   ELSE IF DATAOP = '< ' THEN
      BEGIN
         LOGICST(.STPLOGIC.):=ARITST(.STPARITH-2.)
                             <ARITST(.STPARITH-1.);
         STPLOGIC := STPLOGIC + 1;
         STPARITH := STPARITH - 2;
```

```
      END
   ELSE IF DATAOP = '* ' THEN
      BEGIN
         ARITST(.STPARITH-2.):=ARITST(.STPARITH-2.)
                               *ARITST(.STPARITH-1.);
         STPARITH := STPARITH - 1
      END
   ELSE IF DATAOP = '/ ' THEN
      BEGIN
         ARITST(.STPARITH-2.):=ARITST(.STPARITH-2.)
                               DIV ARITST(.STPARITH-1.);
         STPARITH := STPARITH - 1;
      END
   ELSE ERROR(10)
END; {OPERATOR}


PROCEDURE DATUM;

   Procedure DATUM will push the data called for by the ex-
pression on the execution stack (ARITST). Variable CLASS,
SORTE, and VALUE are used to access the data descriptor ta-
ble. CLASS will hold the class of this variable (INPUT,
OUTPUT, LOCAL, CONST, INOUT), SORTE will hold the type of
this variable (integer, etc.), and variable VALUE will hold
the relative address used to access an entry in the local
section of this procedure or program. Depending on the
class of the variable, the access is either direct or indi-
rect.

VAR
   J, DATAP : INTEGER;

BEGIN
   READ(ELFILE, DATAP);
   WRITE(OUTFILE, DATAP);
   RESET(ELFILE);
   FOR J := 1 TO POINTER3 + MAXBLOCK * LEVEL + DATAP DO
      READLN(ELFILE);
   READ(ELFILE, NAME);
   READ(ELFILE, CLASS);
   READ(ELFILE, SORTE);
   READ(ELFILE, VALUE);
   IF SORTE <> 'INTEGER ' THEN
      ERROR(20);
   IF (CLASS = 'OUTPUT  ') OR (CLASS = 'INOUT   ') THEN
      BEGIN
         VALUE := LOCALPOINT + VALUE;
         VALUE := DATASPACE(.VALUE.);
         ARITST(.STPARITH.) := DATASPACE(.VALUE.);
         STPARITH := STPARITH + 1;
      END
   ELSE IF (CLASS = 'LOCAL   ') OR (CLASS = 'INPUT   ') THEN
      BEGIN
```

```
            VALUE := VALUE + LOCALPOINT;
            ARITST(.STPARITH.) := DATASPACE(.VALUE.);
            STPARITH := STPARITH + 1;
         END
      ELSE IF CLASS = 'CONST   ' THEN
         BEGIN
            ARITST(.STPARITH.) := VALUE;
            STPARITH := STPARITH + 1;
         END;
   END; {DATUM}


   PROCEDURE XECPROCESS(ADDRESS : INTEGER);

      Procedure XECPROCESS is used to transfer control to the
   called procedure.  A stack (CONTST) is used to save the pro-
   gram counter (PCOUNTER) and the location of the calling pro-
   gram or procedure (LEVEL).  If the procedure called is READ
   then procedure READDATA is called,  if it is WRITE then
   procedure WRITEDATA is called.  If it is neither then the
   procedure called is a user procedure, therefore the required
   parameters are passed to it using procedure PUSHPARA and
   POPPARA.

   VAR
      J : INTEGER;

   BEGIN
      CONTST(.STPCONT.) := PCOUNTER;
      STPCONT := STPCONT + 1;
      CONTST(.STPCONT.) := LEVEL;
      STPCONT := STPCONT + 1;
      QUEPOINT := STPCONT;
      RESET(ELFILE);
      FOR J := 1 TO POINTER2 + MAXBLOCK * LEVEL + ADDRESS DO
         READLN(ELFILE);
      READ(ELFILE, TAGBIT);
      READLN(ELFILE, NAME);
      ADDRESS := ADDRESS + 1;
      WRITE(OUTFILE, NAME);
      IF NAME = 'READ    ' THEN
         READDATA
      ELSE IF NAME = 'WRITE   ' THEN
         WRITEDATA
      ELSE
         BEGIN
            PUSHPARA;
            POPPARA;
         END;
   END; {XECPROCESS}


   PROCEDURE READDATA;
```

Procedure READDATA will read a value from the keyboard and place it in the proper location in the data space. This simulate the READ procedure of the virtual language.

```
VAR
   J : INTEGER;

BEGIN
   READLN(ELFILE);
   READ(ELFILE, TAGBIT);
   READLN(ELFILE, DESCRIPTOR);
   READLN(ELFILE);
   RESET(ELFILE);
   FOR J := 1 TO POINTER3 + MAXBLOCK * LEVEL + DESCRIPTOR DO
       READLN(ELFILE);
   READ(ELFILE, NAME);
   READ(ELFILE, CLASS);
   READ(ELFILE, SORTE);
   READ(ELFILE, DESCRIPTOR);
   WRITELN(OUTFILE, 'INPUT VALUE OF', NAME);
   READLN;
   READ(VALUE);
   IF (CLASS = 'LOCAL   ') OR (CLASS = 'INPUT   ') THEN
      DATASPACE(.DESCRIPTOR + LOCALPOINT.) := VALUE
   ELSE
      BEGIN
         DESCRIPTOR := DATASPACE(.DESCRIPTOR + LOCALPOINT.);
         DATASPACE(.DESCRIPTOR.) := VALUE
      END;
   SYSTEMRET;
END; {READDATA}
```

PROCEDURE WRITEDATA;

Procedure WRITEDATA will write to file OUTFILE the proper value from the data space. This procedure simulate the virtual language WRITE procedure.

```
VAR
   J : INTEGER;

BEGIN
   READLN(ELFILE);
   READ(ELFILE, TAGBIT);
   READLN(ELFILE, DESCRIPTOR);
   READLN(ELFILE);
   RESET(ELFILE);
   FOR J := 1 TO POINTER3 + MAXBLOCK * LEVEL + DESCRIPTOR DO
       READLN(ELFILE);
   READ(ELFILE, NAME);
   READ(ELFILE, CLASS);
   READ(ELFILE, SORTE);
   READ(ELFILE, DESCRIPTOR);
```

```
   WRITELN(OUTFILE, ' OUTPUT VALUE OF ', NAME);
   IF (CLASS = 'LOCAL    ') OR (CLASS = 'INPUT    ') THEN
      WRITELN(OUTFILE, DATASPACE(.DESCRIPTOR + LOCALPOINT.))
   ELSE
      BEGIN
         DESCRIPTOR := DATASPACE(.DESCRIPTOR + LOCALPOINT.);
         WRITELN(OUTFILE, DATASPACE(.DESCRIPTOR.))
      END;
   SYSTEMRET;
END; {WRITEDATA}


PROCEDURE PUSHPARA;

   This procedure is used to push the parameter that are
passed to the procedure on the control stack (CONTST). The
position of the parameter in the data space (DATASPACE) is
actually pushed on the stack.  This insure that there will
be at most 2 references to the data space to obtain the cor-
rect data.

VAR
   J : INTEGER;

BEGIN
   READ(ELFILE, TAGBIT);
   READLN(ELFILE, TEMPA);
   EOXPR := FALSE;
   ADDRESS := ADDRESS + 2;
   WHILE NOT EOXPR DO
      BEGIN
         READ(ELFILE, TAGBIT);
         IF TAGBIT = '1' THEN
            OPERATOR
         ELSE
            BEGIN
               READLN(ELFILE, DESCRIPTOR);
               WRITE(OUTDEC , DESCRIPTOR);
               ADDRESS := ADDRESS + 1;
               RESET(ELFILE);
               FOR J:=1 TO POINTER3+MAXBLOCK*LEVEL+DESCRIPTOR DO
                  READLN(ELFILE);
               READ(ELFILE, NAME);
               READ(ELFILE, CLASS);
               IF(CLASS='LOCAL    ')OR(CLASS='INPUT    ')THEN
                  BEGIN
                     READ(ELFILE, SORTE);
                     READ(ELFILE, VALUE);
                     CONTST(.QUEPOINT.) := LOCALPOINT+VALUE;
                     QUEPOINT := QUEPOINT + 1;
                  END
               ELSE IF(CLASS='OUTPUT   ')OR(CLASS='INOUT    ')THEN
                  BEGIN
                     READ(ELFILE,SORTE);
```

```
                    READ(ELFILE, VALUE);
                    CONTST(.QUEPOINT.):=DATASPACE(.VALUE+LOCALPOINT.);
                    QUEPOINT := QUEPOINT + 1;
                  END;
               RESET(ELFILE);
               FOR J:=1 TO POINTER2+MAXBLOCK*LEVEL+ADDRESS DO
                  READLN(ELFILE);
            END;
      END;
END; {PUSHPARA}
```

PROCEDURE POPPARA;

Procedure POPPARA will fill the local section in the data space of the called procedure, with the values and addresses required to access the global variables. The parameter passed, which are on the control stack (CONTST), are accessed using pointer QUEPOINT, that points initially to the first parameter passed.

```
VAR
   J : INTEGER;

BEGIN
   LEVEL := TEMPA;
   RESET(ELFILE);
   FOR J:=1 TO MAXBLOCK * LEVEL DO
      READLN(ELFILE);
   LOCALPOINT := LOCALPOINT + LOCALEND;
   GETPOINTER;
   PCOUNTER := 0;
   QUEPOINT := STPCONT;
   READ(ELFILE, TAGBIT);
   PCOUNTER := PCOUNTER + 1;
   READLN(ELFILE, CWORD);
   IF CWORD = 'PARA    ' THEN
      BEGIN
         READ(ELFILE, TAGBIT);
         PCOUNTER := PCOUNTER + 1;
         WHILE TAGBIT = '0' DO
            BEGIN
               READLN(ELFILE, DESCRIPTOR);
               RESET(ELFILE);
               FOR J:=1 TO POINTER3+MAXBLOCK*LEVEL+DESCRIPTOR DO
                  READLN(ELFILE);
               READ(ELFILE, NAME);
               READ(ELFILE, CLASS);
               IF(CLASS='LOCAL   ')OR(CLASS='CONST   ') THEN
                  ERROR(11)
               ELSE IF CLASS = 'INPUT   ' THEN
                  BEGIN
                     READ(ELFILE, SORTE);
                     READ(ELFILE, VALUE);
```

```
                    VALUE := VALUE + LOCALPOINT;
                    DATASPACE(.VALUE.):=DATASPACE(.CONTST(.QUEPOINT.).);
                    QUEPOINT := QUEPOINT + 1;
                 END
              ELSE IF(CLASS='INOUT   ')OR(CLASS='OUTPUT  ')THEN
                 BEGIN
                    READ(ELFILE, SORTE);
                    READ(ELFILE, VALUE);
                    VALUE := VALUE + LOCALPOINT;
                    DATASPACE(.VALUE.):=CONTST(.QUEPOINT.);
                    QUEPOINT := QUEPOINT + 1;
                 END;
              RESUME;
              READ(ELFILE, TAGBIT);
              PCOUNTER := PCOUNTER + 1;
           END;
        READLN(ELFILE, CWORD);
        IF CWORD <> 'BEGIN   ' THEN
           ERROR(12);
     END;
END; {POPPARA}
```

PROCEDURE SYSTEMRET;

This procedure is called when the execution of a system procedure (WRITE or READ) is terminated. It will return execution to the calling program or procedure.

```
BEGIN
   STPCONT := STPCONT - 1;
   LEVEL := CONTST(.STPCONT.);
   STPCONT := STPCONT - 1;
   PCOUNTER := CONTST(.STPCONT.);
   RESUME;
END; {SYSTEMRET}
```

PROCEDURE PROCESSRET;

This procedure is called when the execution of a procedure has terminated, it will return to the calling procedure or program. The information that was pushed on the control stack (CONTST) is used to return to the statement following the procedure call. If the control stack is empty this indicates that the main program is terminated and that processing should halt.

```
VAR
   J : INTEGER;

BEGIN
   IF STPCONT > 1 THEN
      BEGIN
```

```
            STPCONT := STPCONT - 1;
            LEVEL := CONTST(.STPCONT.);
            STPCONT := STPCONT -1;
            PCOUNTER := CONTST(.STPCONT.);
            RESET(ELFILE);
            IF LEVEL <> 0 THEN
              BEGIN
                  FOR J:= 1 TO MAXBLOCK * LEVEL DO
                    READLN(ELFILE);
              END;
            GETPOINTER;
            LOCALPOINT := LOCALPOINT - LOCALEND;
            FOR J := 1 TO PCOUNTER DO
              READLN(ELFILE);
        END
      ELSE EOEXECUT := TRUE
END; {PROCESSRET}
```

Appendix E

SIMULATION EXAMPLES

In this appendix some examples of the two simulations that were performed are analysed. These examples are very simple since only variables of type integer are supported in the virtual language defined.

The following virtual language program gives a list of the correct move that must be performed to solve the tower of Hanoi game. This game consists of three needles, and a number of disks. The disk must be move from the first needle to the third needle, under the condition that no larger disk may rest on a smaller disk.

```
HANOI;
TOTAL,ONE,THREE,TWO;
(INTEGER : ONE,TWO,THREE,TOTAL)

BEGIN
   READ(TOTAL);
   ONE = 1;
   TWO = 2;
   THREE = 3;
   MOVETOWER(TOTAL,ONE,THREE,TWO);
END

MOVETOWER(HEIGHT,FROMNEEDLE,TONEEDLE,USINGNEEDLE;;);
TEMP;
(INTEGER : HEIGHT,FROMNEEDLE,TONEEDLE,USINGNEEDLE,TEMP)

BEGIN
   IF HEIGHT > 0 THEN
      TEMP = HEIGHT - 1;
      MOVETOWER(TEMP,FROMNEEDLE,TONEEDLE,USINGNEEDLE;;);
      WRITE(FROMNEEDLE);
      WRITE(TONEEDLE);
      MOVETOWER(TEMP,USINGNEEDLE,TONEEDLE,FROMNEEDLE;;);
   ENDIF
END

STOP
```

This virtual language program is transformed into the following execution language program using the simulation

program PREPROCESSOR (appendix C). The left column is the transformation of program Hanoi, while the right column corresponds to the transformation of procedure Movetower.

```
HANOI
15
23
49
 4
1 BEGIN
0 0
0 1
0 2
0 3
0 4
1 END

STATEMENT DESC. TABLE

PROCESS   0
ASSIGN    4
ASSIGN    8
ASSIGN   12
PROCESS  16

STATEMENTS

1 READ
1 0
0 0
1 S
0 4
0 1
1 =
1 S
0 5
0 3
1 =
1 S
0 6
0 2
1 =
1 S
1 MOVETOWE
1 1
0 0
0 1
0 2
0 3
1 S

DATA DESCRIPTOR TABLE

TOTAL   LOCAL   INTEGER 1
```

```
MOVETOWE
24
33
68
 5
1 PARA
0 0
0 1
0 2
0 3
1 BEGIN
1 IF
0 0
0 5
0 1
0 2
0 3
0 4
0 5
1 ENDIF
1 END

STATEMENT DESC. TABLE

LOGIC     0
ASSIGN    4
PROCESS  10
PROCESS  17
PROCESS  21
PROCESS  25

STATEMENTS

0 0
0 5
1 >
1 S
0 0
0 6
1 -
0 4
1 =
1 S
1 MOVETOWE
1 1
0 4
0 1
0 3
0 2
1 S
```

```
ONE       LOCAL    INTEGER  2  1 WRITE
THREE     LOCAL    INTEGER  3  1 0
TWO       LOCAL    INTEGER  4  0 1
1         CONST    INTEGER  1  1 S
2         CONST    INTEGER  2  1 WRITE
3         CONST    INTEGER  3  1 0
                               0 2
                               1 S
                               1 MOVETOWE
                               1 1
                               0 4
                               0 3
                               0 2
                               0 1
                               1 S

DATA DESCRIPTOR TABLE

HEIGHT    INPUT    INTEGER  1
FROMNEED  INPUT    INTEGER  2
TONEEDLE  INPUT    INTEGER  3
USINGNEE  INPUT    INTEGER  4
TEMP      LOCAL    INTEGER  5
0         CONST    INTEGER  0
1         CONST    INTEGER  1
```

Once preprocessed this program can be executed directly using the second simulation program (TASKMAN). Program TASKMAN will give a list of the statements that were executed, along with the value outputed by the write procedure. For example using 3 disks we would obtain:

```
INPUT VALUE OF total
   3
4   1   :   S
5   3   :   S
6   2   :   S
MOVETOWE    S
0   5   >   S
0   6   -   4   :   S
MOVETOWE    S
0   5   >   S
0   6   -   4   :   S
MOVETOWE    S
0   5   >   S
0   6   -   4   :   S
MOVETOWE    S
0   5   >   S
OUTPUT VALUE OF fromneed
   1
OUTPUT VALUE OF toneedle
   3
MOVETOWE    S
```

```
0   5   >   S
OUTPUT VALUE OF fromneed
   1
OUTPUT VALUE OF toneedle
   2
MOVETOWE    S
0   5   >   S
0   6   -   4   :   S
MOVETOWE    S
0   5   >   S
OUTPUT VALUE OF fromneed
   3
OUTPUT VALUE OF toneedle
   2
MOVETOWE    S
0   5   >   S
OUTPUT VALUE OF fromneed
   1
OUTPUT VALUE OF toneedle
   3
MOVETOWE    S
0   5   >   S
0   6   -   4   :   S
MOVETOWE    S
0   5   >   S
0   6   -   4   :   S
MOVETOWE    S
0   5   >   S
OUTPUT VALUE OF fromneed
   2
OUTPUT VALUE OF toneedle
   1
MOVETOWE    S
0   5   >   S
OUTPUT VALUE OF fromneed
   2
OUTPUT VALUE OF toneedle
   3
MOVETOWE    S
0   5   >   S
0   6   -   4   :   S
MOVETOWE    S
0   5   >   S
OUTPUT VALUE OF fromneed
   1
OUTPUT VALUE OF toneedle
   3
MOVETOWE    S
0   5   >   S
```

The second example calculates the determinant of a matrix. We can see that without supporting complex data structures, this program is cumbersome. In the third example, addition of vectors to the system will be analysed.

```
DETER;
A11,A12,A13,A21,A22,A23,A31,A32,A33,SUM;
(INTEGER : A11,A12,A13,A21,A22,A23,A31,A32,A33,SUM)

BEGIN
   READ(A11);
   READ(A12);
   READ(A13);
   READ(A21);
   READ(A22);
   READ(A23);
   READ(A33);
   READ(A32);
   READ(A33);
   SUM = A11 * (A22 * A33 - A32 * A23)
         - A12 * (A21 * A33 - A31 * A23)
         + A13 * (A21 * A32 - A31 * A22);
   WRITE(SUM);
END

STOP
```

The program is translated in an execution language representation using the simulation program PREPROCESSOR.

```
DETER
21
35
110
10
1 BEGIN
0 0
0 1
0 2
0 3
0 4
0 5
0 6
0 7
0 8
0 9
0 10
1 END

STATEMENT DESCRIPTOR TABLE

PROCESS   0
PROCESS   4
```

```
PROCESS   8
PROCESS  12
PROCESS  16
PROCESS  20
PROCESS  24
PROCESS  28
PROCESS  32
ASSIGN   36
PROCESS  68

STATEMENTS

1 READ
1 0
0 0
1 S
1 READ
1 0
0 1
1 S
1 READ
1 0
0 2
1 S
.....
1 READ
1 0
0 8
1 S
0 0
0 4
0 8
1 *
0 7
0 5
1 *
1 -
1 *
0 1
0 3
0 8
1 *
0 6
0 5
1 *
1 -
1 *
1 -
0 2
0 3
0 7
1 *
0 6
0 4
```

```
1 *
1 -
1 *
1 +
0 9
1 =
1 S
1 WRITE
1 0
0 9
1 S
```

```
DATA DESCRIPTOR TABLE

A11       LOCAL    INTEGER    1
A12       LOCAL    INTEGER    2
A13       LOCAL    INTEGER    3
A21       LOCAL    INTEGER    4
A22       LOCAL    INTEGER    5
A23       LOCAL    INTEGER    6
A31       LOCAL    INTEGER    7
A32       LOCAL    INTEGER    8
A33       LOCAL    INTEGER    9
SUM       LOCAL    INTEGER   10
```

The preprocessed version of this program can be executed by program TASKMAN, which will give the following output.

```
INPUT VALUE OF a11
   1
INPUT VALUE OF a12
   3
INPUT VALUE OF a13
   4
INPUT VALUE OF a21
   8
INPUT VALUE OF a22
   5
INPUT VALUE OF a23
   2
INPUT VALUE OF a31
   6
INPUT VALUE OF a32
   7
INPUT VALUE OF a33
   1
0   4   8   *   7   5   *   -   *
1   3   8   *   6   5   *   -   *
-   2   3   7   *   6   4   *   -
*   +   9   :   S
OUTPUT VALUE OF sum
   107
```

The previous examples dealt only with variable of type integer, this greatly limits the number of applications. For example trying to sort a number of elements is nearly impossible without vectors. Therefore we looked at the possibility of adding vectors to the system (see chapter 6). The following example performs a sort on a vector of 3 elements.

```
SORT;
TOP,LIST,ELEM,TEMP;
(INTEGER : TOP,ELEM,TEMP)
(VECTOR (3) OF INTEGER : LIST)

BEGIN
   TEMP = 1;
   WHILE TEMP < 4 DO
      READ(ELEM);
      LIST(TEMP) = ELEM;
      TEMP = TEMP + 1;
   ENDWHILE
   TOP = 1;
   WHILE TOP < 3 DO
      TEMP = LIST(TOP);
      ELEM = TOP;
      WHILE ELEM < 3 DO
         ELEM = ELEM + 1;
         IF LIST(ELEM) < TEMP THEN
            LIST(TOP) = LIST(ELEM);
            LIST(ELEM) = TEMP;
            TEMP = LIST(TOP);
         ENDIF
      ENDWHILE
      TOP = TOP + 1;
   ENDWHILE
   TEMP = 1;
   WHILE TEMP < 4 DO
      ELEM = LIST(TEMP);
      WRITE(ELEM);
      TEMP = TEMP + 1;
   ENDWHILE
END

STOP
```

In order to preprocess this program, a few changes must be made to program PREPROCESSOR. The first change required concerns procedure DATATYPE. This procedure must be modified to recognize the vector declaration statement. Once a vector declaration statement is encountered, a type of VECTINT is assigned to the corresponding variables. These changes are shown below. It should be noted that only integer vectors are allowed.

```
PROCEDURE DATATYPE;
```

```
VAR
   SYMBOL : STRING;
   INDEX : INTEGER;

BEGIN
   WHILE TOKEN = '(        ' DO
      BEGIN
         SKIPBLANK;
         CLASS := TOKEN;
         IF CLASS = 'INTEGER ' THEN
            BEGIN
               SKIPBLANK;
               IF TOKEN <> ':        ' THEN
                  ERROR(5);
               SKIPBLANK;
               REPEAT
                  LOCATE(VALUE);
                  DATABLE(.VALUE.).SORTE := CLASS;
                  SKIPBLANK;
                  SYMBOL := TOKEN;
                  SKIPBLANK;
               UNTIL SYMBOL = ')        ';
            END
         ELSE IF CLASS = 'VECTOR  ' THEN
            BEGIN
               SKIPBLANK;
               IF TOKEN <> '(        ' THEN
                  ERROR(50);
               READ(VLFILE, INDEX);
               SKIPBLANK;
               IF TOKEN <> ')        ' THEN
                  ERROR(51);
               SKIPBLANK;
               SKIPBLANK;
               IF TOKEN = 'INTEGER ' THEN
                  CLASS := 'VECTINT '
               ELSE ERROR(52);
               SKIPBLANK;
               IF TOKEN < ':         ' THEN
                  ERROR(53);
               SKIPBLANK;
               REPEAT
                  LOCATE(VALUE);
                  DATABLE(.VALUE.).SORTE := CLASS;
                  DATABLE(.VALUE.).LENGTH := INDEX;
                  SKIPBLANK;
                  SYMBOL := TOKEN;
                  SKIPBLANK;
               UNTIL SYMBOL = ')        ';
            END;
      END;
END;
```

The second change required concerns procedure DATALOCATION, since we must reserve the appropriate number of locations in the data space for this vector.

```
PROCEDURE DATALOCATION;

VAR
   I, X : INTEGER;

BEGIN
   VALUE := 1;
   FOR I := 1 TO DTPOINT - 1 DO
      BEGIN
         IF DATABLE(.I.).SORTE = 'INTEGER ' THEN
            BEGIN
               DATABLE(.I.).POSITION := VALUE;
               X := 1;
            END
         ELSE IF DATABLE(.I.).SORTE = 'VECTINT ' THEN
            BEGIN
               DATABLE(.I.).POSITION := VALUE;
               X := DATATABLE(.I.).LENGTH;
            END;
         VALUE := VALUE + X;
      END;
   LOCALPOINT := VALUE - X;
END;
```

Next, procedure ASSIGNMENT must be modified to recognize vectors that appear on the left hand side of an assignment statement.

```
PROCEDURE ASSIGNMENT;

VAR
   TEMP2 : INTEGER;
   VECT : BOOLEAN;

BEGIN
   VECT := FALSE;
   STATABLE(.STPOINT.).CLASS := 'ASSIGN  ';
   STATABLE(.STPOINT.).LOCATION := PLACE - 1;
   STPOINT := STPOINT + 1;
   CONTABLE(.CSPOINT.).CONTBIT := '0';
   CONTABLE(.CSPOINT.).DESCTOR := STPOINT - 2;
   CSPOINT := CSPOINT + 1;
   TEMP1 := VALUE;
   SKIPBLANK;
   IF TOKEN = '(       ' THEN
      BEGIN
         SKIPBLANK;
         LOCATE(VALUE);
         TEMP2 := VALUE;
         VECT := TRUE;
```

```
            SKIPBLANK;
            SKIPBLANK;
         END;
      IF TOKEN <> '=       ' THEN
         ERROR(23);
      LIMITE := ';       ';
      RPOLISH;
      VALUE := TEMP1;
      STATSPACE(.PLACE.).CONTBIT := '0';
      STATSPACE(.PLACE.).DESCTOR := VALUE - 1;
      PLACE := PLACE + 1;
      IF VECT = TRUE THEN
         BEGIN
            STATSPACE(.PLACE.).CONTBIT := '0';
            STATSPACE(.PLACE.).DESCTOR := TEMP2 -1;
            PLACE := PLACE + 1;
         END;
      STATSPACE(.PLACE.).CONTBIT := '1';
      STATSPACE(.PLACE.).PROGCONT := ':       ';
      PLACE := PLACE + 1;
      STATSPACE(.PLACE.).CONTBIT := '1';
      STATSPACE(.PLACE.).PROGCONT := 'S';
      PLACE := PLACE + 1;
   END;
```

Finally procedure RPOLISH must be modified to properly translate expressions into a reverse polish notation. When a symbolic name of a vector is encountered, it will be replaced by a descriptor along with the descriptor of its index variable.

```
   PROCEDURE RPOLISH;

   BEGIN
      SKIPBLANK;
      WHILE TOKEN <> LIMITE DO
         BEGIN
            LOCATE(VALUE);
            IF FOUND THEN
               BEGIN
                  STATSPACE(.PLACE.).CONTBIT := '0';
                  STATSPACE(.PLACE.).DESCTOR := VALUE - 1;
                  PLACE := PLACE + 1;
                  SKIPBLANK;
                  IF TOKEN = '(       ' THEN
                     BEGIN
                        SKIPBLANK;
                        LOCATE(VALUE);
                        STATSPACE(.PLACE.).CONTBIT := '0';
                        STATSPACE(.PLACE.).DESCTOR:=VALUE-1;
                        PLACE := PLACE + 1;
                        SKIPBLANK;
                        SKIPBLANK;
                     END;
```

```
        END
    ELSE IF OPERATOR THEN
.....
```

With these modifications, program SORT can be translated as follows. It should be noted that these changes do not permit the passing of vectors between procedures.

```
SORT
50
74
177
6
1 BEGIN
0 0
1 WHILE
0 1
0 3
0 2
0 3
0 4
1 ENDWHILE
0 8
0 5
1 WHILE
0 6
0 16
0 7
0 8
1 WHILE
0 9
0 8
0 10
1 IF
0 11
0 3
0 12
0 13
0 14
1 ENDIF
1 ENDWHILE
0 13
0 15
1 ENDWHILE
0 21
0 16
1 WHILE
0 17
0 3
0 18
0 19
0 20
1 ENDWHILE
0 8
```

```
1 END

STATEMENT DESCRIPTOR TABLE

ASSIGN        0
LOGIC         4
PROCESS       8
ASSIGN       12
ASSIGN       17
ASSIGN       23
LOGIC        27
ASSIGN       32
ASSIGN       36
LOGIC        40
ASSIGN       44
LOGIC        50
ASSIGN       55
ASSIGN       61
ASSIGN       66
ASSIGN       71
ASSIGN       77
LOGIC        81
ASSIGN       85
PROCESS      90
ASSIGN       94

STATEMENTS

0 4
0 3
1 =
1 S
0 3
0 5
1 <
1 S
1 READ
1 0
0 2
1 S
0 2
0 1
0 3
1 =
1 S
0 3
0 4
1 +
0 3
1 =
1 S
0 4
0 0
1 =
```

```
1 S
0 0
0 6
1 <
1 S
0 1
0 0
0 3
1 =
1 S
0 0
0 2
1 =
1 S
0 2
0 6
1 <
1 S
0 2
0 4
1 +
0 2
1 =
1 S
0 1
0 2
0 3
1 <
1 S
0 1
0 2
0 1
0 0
1 =
1 S
0 3
0 1
0 2
1 =
1 S
0 1
0 0
0 3
1 =
1 S
0 0
0 4
1 +
0 0
1 =
1 S
0 4
0 3
1 =
```

```
1 S
0 3
0 5
1 <
1 S
0 1
0 3
0 2
1 =
1 S
1 WRITE
1 0
0 2
1 S
0 3
0 4
1 +
0 3
1 =
1 S
```

```
DATA DESCRIPTOR TABLE

TOP       LOCAL     INTEGER       1
LIST      LOCAL     VECTINT       2
ELEM      LOCAL     INTEGER       5
TEMP      LOCAL     INTEGER       6
1         CONST     INTEGER       1
4         CONST     INTEGER       4
3         CONST     INTEGER       3
```

Once translated, this program can be executed by program TASKMAN. However, to accomplish this a few modifications must be performed to this simulation program. This requires that procedure DATUM be modified, since it is in charge of pushing the appropriate values on the execution stack. The required changes are given below.

```
PROCEDURE DATUM;

VAR
   J, TEMP1, DATAP : INTEGER;

BEGIN
   READ(ELFILE, DATAP);
   WRITE(OUTFILE, DATAP);
   RESET(ELFILE);
   FOR J := 1 TO POINTER3 + MAXBLOCK * LEVEL + DATAP DO
      READLN(ELFILE);
   READ(ELFILE, NAME);
   READ(ELFILE, CLASS);
   READ(ELFILE, SORTE);
   READ(ELFILE, VALUE);
   IF SORTE = 'INTEGER ' THEN
```

```
      BEGIN
         IF(CLASS='OUTPUT  ')OR(CLASS='INOUT   ')THEN
            BEGIN
               VALUE := LOCALPOINT + VALUE;
               VALUE := DATASPACE(.VALUE.);
               ARITST(.STPARITH.) := DATASPACE(.VALUE.);
               STPARITH := STPARITH + 1;
            END
         ELSE IF (CLASS='LOCAL   ')OR(CLASS='INPUT   ')THEN
            BEGIN
               VALUE := VALUE + LOCALPOINT;
               ARITST(.STPARITH.) := DATASPACE(.VALUE.);
               STPARITH := STPARITH + 1;
            END
         ELSE IF CLASS = 'CONST   ' THEN
            BEGIN
               ARITST(.STPARITH.) := VALUE;
               STPARITH := STPARITH + 1;
            END;
      END
   ELSE IF SORTE = 'VECTINT ' THEN
      BEGIN
         TEMP1 := VALUE + LOCALPOINT;
         RESET(ELFILE);
         FOR J:=1 TO POINTER2+MAXBLOCK*LEVEL+ADDRESS DO
            READLN(ELFILE);
         READ(ELFILE, TAGBIT);
         ADDRESS := ADDRESS + 1;
         IF TAGBIT = '1' THEN
            ERROR(100);
         READ(ELFILE, DATAP);
         RESET(ELFILE);
         FOR J:=1 TO POINTER3+MAXBLOCK*LEVEL+DATAP DO
            READLN(ELFILE);
         READ(ELFILE, NAME);
         READ(ELFILE, CLASS);
         READ(ELFILE, SORTE);
         READ(ELFILE, VALUE);
         VALUE := DATASPACE(.VALUE.) + TEMP1;
         ARITST(.STPARITH.) := DATASPACE(.VALUE.);
         STPARITH := STPARITH + 1;
      END
   ELSE ERROR(20);
END;
```

From these changes, the following output would be obtained from the execution of this sorting program. This output corresponds to the statements executed.

```
4   3   :   S
3   5   <   S
INPUT VALUE OF list
   4
2   1   :   S
```

```
3   4   +   3   :   S
3   5   <   S
INPUT VALUE OF list
   9
2   1   :   S
3   4   +   3   :   S
3   5   <   S
INPUT VALUE OF list
   2
2   1   :   S
3   4   +   3   :   S
3   5   <   S
4   0   :   S
0   6   <   S
1   3   :   S
0   2   :   S
2   6   <   S
2   4   +   2   :   S
1   3   <   S
1   1   :   S
3   1   :   S
1   3   :   S
2   6   <   S
2   4   +   2   :   S
1   3   <   S
2   6   <   S
0   4   +   0   :   S
0   6   <   S
1   3   :   S
0   2   :   S
2   6   <   S
2   4   +   2   :   S
1   3   <   S
1   1   :   S
3   1   :   S
1   3   :   S
2   6   <   S
0   4   +   0   :   S
0   6   <   S
0   6   <   S
4   3   :   S
3   5   <   S
1   2   :   S
OUTPUT VALUE OF list
   2
3   4   +   3   :   S
3   5   <   S
1   2   :   S
OUTPUT VALUE OF list
   4
3   4   +   3   :   S
3   5   <   S
1   2   :   S
OUTPUT VALUE OF list
```

```
    9
  3   4   +   3   :   S
  3   5   <   S
```

REFERENCES

AHO 79 Aho, A. V., and J. D. Ullman, "Principles of Compiler Design", Addison-Wesley, Reading, Mass., 1979.

ALTA 79 Altaber, J., F. Beck, B. Mears, and R. Rausch, "List Search Hardware for Interactive Software", Microprocessors and their applications, Euromicro, 1979.

AVIZ 78 Avizienis, A., "Fault-Tolerance: The Survival Attribute of Digital Systems", Proceedings of the IEEE, vol. 66, no. 10, oct. 1978, pp. 1109-1125.

BACK 78 Backus, J., "Can Programming Be Liberated from the von Neumann Style? a Functional Style and Its Algebra of Program", comm. ACM, vol. 21, no. 8, aug. 1978, pp. 613-641.

BAER 76 Baer, J. L., "Multiprocessing Systems", IEEE trans. on computers, vol. C-25, no. 12, dec. 1976, pp. 1271-1277.

BERK 71 Berkling, K., "A Computing Machine Based on Tree Structures", IEEE trans. on computers, vol. C-20, no. 4, april 1971, pp. 404-418.

BERK 78 Berkling, K., "Computer Architecture for Correct Programming", IEEE 5th annual symp. on computer architecture, april 1978, Palo Alto, CA, pp. 78-84.

BERN 81 Berhard, R., "More Hardware Means Less Software", IEEE spectrum, vol. 18, no. 12, dec. 1981, pp. 30-37.

BOHM 66 Bohm, C., and J. Jacopini, "Flow Diagrams, Turing Machines and Languages With Only Two Formation Rules", Comm ACM, vol. 9, no. 5, may 1966, pp. 366-371.

BOUL 77 Boulaye, G. G., and D. W. Venin, "Computer Architecture", D. Reidel publishing co., 1977.

BRIN 75 Brinch Hansen P., "The Programming Language Concurrent Pascal", IEEE trans. on soft. eng., vol. SE-1, no. 2, june 75, pp. 199-207.

BROD 81 Brode, B., "Precompilation of Fortran Programs to Facilitate Array Processing", computer magazine, vol. 14, no. 9, sept. 1981, pp. 46-51.

BURK 46 Burks, A. W., H. H. Goldstine, and J. von Neumann, "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument", in "The Origins of Digital Computers", B. Randell, Springer-Verlag, New York, 1975, pp. 371-385.

BURR 69 Burroughs B6500 Information Processing System, Reference Manual, 1969.

CARL 75 Carlson, C. R., "A Survey of High-Level Language Computers", in "High-Level Computer Architecture", edited by Y. Chu, Academic Press, 1975.

CASP 78 Caspe, R. A., "Array Processors", Mini-Micro Systems, july 1978.

COPE 82 Copeland, G., "What if Mass Storage Were Free?", computer magazine, vol. 15, no. 7, july 1982, pp. 27-35.

CHU 75 Chu, Y., "High-Level Language Computer Architecture", Academic Press, New York, 1975.

CHU 79 Chu, Y., "Architecture of a Hardware Interpreter", IEEE trans. on computers, vol. C-28, no. 2, feb. 1979, pp. 101-109.

CHU 81 Chu, Y., and M. Abrams, "Programming Languages and Direct-Execution Computer Architecture", computer magazine, vol. 14, no. 7, july 1981, pp. 22-32.

DeMA 76 DeMartinis, M., G. J. Lipovski, S. Y. W. Su, and J. K. Watson, "A Self Managing Secondary Memory System", the 3rd annual symp. on computer architecture, 1976, pp. 186-194.

DENN 80 Dennis, J. B., "Data Flow Supercomputers", computer magazine, vol. 13, no. 11, nov. 1980, pp. 48-56.

DIJK 68 Dijkstra, E. W., "The Structure of the 'THE' Multiprogramming System", comm. ACM, vol. 11, no. 5, may 1968, pp. 341-346.

ENSL 77 Enslow, P. H., "Multiprocessor Organization - A Survey", A CM computing surveys, vol. 9, no. 1, mar. 1977, pp. 103-129.

ENSL 78 Enslow, P. H., "What is a Distributed Data Processing System?", computer magazine, vol. 11, no. 1, jan. 1978, pp. 13-21.

FALK 74 Falk, H., "Hard-Soft Tradeoffs", IEEE spectrum, vol. 11, no. 2, feb. 1974, pp. 34-39.

FATH 81a Fathi, E. T., "Task-Driven Multi-Microcomputer System", M.A.Sc. thesis, University of Ottawa, 1981.

FATH 81b Fathi, E. T., and M. Krieger, "Centrally Controlled Segmented Bus Architecture", 19th annual Allerton Conference on Communication, Control, and Computer, Monticello, Illinois, 1981.

FATH 82a Fathi, E. T., and M. Krieger, "W-3 of Multiple Microprocessor Systems", to appear.

FATH 82b Fathi, E. T., and M. Krieger, "Executive For Task-Driven Multi-Microcomputer Systems", to appear.

FENG 81 Feng, T.-Y., "A Survey of Interconnection Networks", computer magazine, vol. 14, no. 12, dec. 1981, pp. 12-27.

FEUS 73 Feustel, E. A., "On the Advantage of Tagged Architecture", IEEE trans. on computer, vol. C-22, 1973, pp. 644-652.

FLYN 66 Flynn, M. J., "Very High-Speed Computing System", Proceedings of the IEEE, vol. 54, no. 12, dec. 1966, pp. 1901-1909.

FLYN 72 Flynn, M. J., "Some Computer Organization and Their Effectiveness", IEEE trans. on computers, vol. C-21, no. 9, sept. 1973, pp. 948-960.

FLYN 80 Flynn, M. J., "Directions and Issues in Architecture and Language", computer magazine, vol. 13, no. 10, oct. 1980, pp. 5-20.

FREE 75 Freeman, P., "Software System Principles : A Survey", Science Research Associates, Inc., 1975.

GARO 74 Garon, G., "The Fast Processing of a Single Stream of Instructions", M.A.Sc. thesis, University of Ottawa, april 1974.

GILO 75 Giloi, W. K., and H. Berg, "STARLET - A Computer Based on Ordered Sets as Primitive Data Types", Proc. 2nd Annual Symp. on Computer Architecture, Houston 1975, pp. 201-206.

GONZ 78 Gonzalez, M. J., "Future Directions in Computer Architecture", computer magazine, vol. 11, no. 3, march 1978, pp. 54-62.

GREE 79 Green, P. E., "An Introduction to Network Architectures and Protocols", IBM systems journal, vol. 18, no. 2, 1979, pp. 202-222.

GROG 80 Grogono, P., "Programming in PASCAL", Addison-Wesley publ. co., Inc., 1980.

HIGB 73 Higbie, L. C., "Supercomputer Architecture, A Tutorial", computer magazine, dec. 1973, pp. 48-58.

HOEV 73 Hoevel, L. W., "Ideal Directly Executed Languages: An Analytical Argument for Emulation", IEEE trans. on computer, vol. C-23, no. 8, aug. 1978, pp. 750-767.

HOMM 80 Hommes, F., W. Kluge, and H. Schlutter, "A Reduction Architecture and Expression Oriented Editing", Gesellschaft Fur Mathematik (GMD), ISF-Report 80.04, 1980.

JENS 81 Jensen, R. W., "Structured Programming", computer magazine, vol. 14, no. 3, march 1981, pp. 31-48.

KLUG 79a Kluge, W. E., "Some Less Enthusiastic Remarks on the Impact of New Technologies", Issues in Data Base Management, North-Holland publ. co., 1979.

KLUG 79b Kluge, W. E., "The Architecture of a Reduction Language Machine Hardware Model", Gesellschaft Fur Mathematik (GMD), ISF-Report 79.03, aug. 1979.

KRIE 79 Krieger, M., "Task-Driven Multimicroprocessor System", Proceedings of the first canadian workshop on the design and development of computer systems, may 1979, pp. 81-88.

KRIE 81 Krieger, M., and E. T. Fathi, "Design Aspects of a Simple Distributed Microprocessor System", International Conference on Communications, Circuits and Systems, Judavpor University, Calcutta, Dec. 1981.

KRIE 82 Krieger, M., and N. Santoro, "Some Open Questions in (Complex) Computer System Design - A Cry for Help", to appear.

LALI 73 Laliotis, T. A., "Implementation Aspects of the Symbol Hardware Compiler", Proc. First Annual Symp. Computer Architecture, ACM, New York, 1973, pp. 111-115.

LECO 79 Lecouffe, M. P., and B. Toursel, "The Possible Impact of Software Trends on Microcomputer Architecture", microprocessors and their applications, Euromicro, North-Holland publ. co., 1979, pp. 139-146.

LIPO 78 Lipovski, G. J., and K. L. Doty, "Developments and Directions in Computer Architecture", computer magazine, vol. 11, no. 8, aug. 1978, pp. 54-67.

MART 81 Martin, G. R., "Micro Architecture Benefits High-Level Languages", Electronics design, nov. 26, 1981, pp. 219-226.

McKE 80 McKeenan, "Stack Computers", in "Introduction to Computer Architecture", editor H. S. Stone, S.R.A., 1980.

MYER 78 Myers, G. J., "Advances in Computer Architecture", John Wiley & sons Inc., New York, 1978.

MALA 80 Malaiya, Y. K., and S. Y. H. Su, "Fault-Tolerance in Multiple Processor Systems", IEEE 7th annual symp. on computer architecture, 1980, pp. 710-716.

NEGE 77 Negini, R., and S. Sami, "The Assembly-Line Multiprocessor Structure", microcomputer architecture, Euromicro, North-Holland publ. co., 1977, pp. 266-275.

NORI 79 Norin, R., and T. Pettibone, "Programming Array Processors", Mini-Micro systems, aug. 1979, pp. 59-71.

PATT 79 Patterson, D. A., E. Fehr, and C. H. Sequin, "Design Considerations for the VLSI Processor of X-Tree", IEEE computer architecture 6th annual symposium, 1979.

RAMA 77 Ramamoorthy, C. V., and H. F. Li, "Pipeline Architecture", A.C.M. computing surveys, vol. 9, no. 1, mar. 1977, pp. 61-102.

RAMA 81 Ramamoorthy, C. V., S. T. Dong, S. L. Ganesh, C.-H. Jen, and W.-T. Tsai, "Architectural Issues in Distributed Systems", International Conference on Communications, Circuits and Systems, Jadovpur University, Calcutta, Dec. 1981.

RAND 75 Randell, B. (editor), "The Origins of Digital Computers", Springer-Verlag, New York, 1975.

RATT 81 Rattner, J., and W. W. Lattan, "ADA Determines Architecture of 32-bit Microprocessor", Electronics, vol. 54, no. 4, feb. 24, 1981.

RICE 81 Rice, R., "The Chief Architect's Reflections on Symbol IIR", computer magazine, vol. 14, no. 7, july 1981, pp. 49-54.

SHAN 80 Shankar, K. S., "Tutorial : Data Stuctures, Types, and Abstractions", computer magazine, vol. 13, no. 4, april 1980, pp. 67-77.

SIEW 82 Siewiorek, D. P., C. G. Bell, and A. Newell, "Computer Structures: Principles and Examples", New York, McGraw Hill, 1982.

SNYD 81 Snyder, F. G., "Modular Fault Tolerance Keeps Computer Systems Reliable", Electronic design, may 14, 1981, pp. 163-167.

STIF 81 Stiffler, J. J., W. Mosteller, "How Computer Fail", IEEE spectrum, special issue on reliability, vol. 18, no. 10, oct. 1981, pp. 43-46.

STON 80 Stone, H. S., "Introduction to Computer Architecture", Science Research Associates, Inc., 1980.

TANE 76 Tanebaum, A. S., "Structured Computer Organization", Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

TANE 78 Tanenbaum, A. S., "Implications of Structured Programming for Machine Architecture", comm. A.C.M., vol. 21, no. 3, mar. 1978, pp. 237-246.

TESL 68 Teslel, L. G., and H. J. Enea, "A Language Design for Concurrent Processes", Spring Joint computer conferences, 1968, pp. 403-408.

THEI 74 Theis, D. J., "Vector Supercomputers", computer magazine, april 1974, pp. 52-61.

THUR 72 Thurber, K. J., et al., "A Systematic Approach to the Design of Digital Bussing Structures", Fall Joint computer conference, 1972, pp. 719-740.

THUR 75 Thurber, K. J., and L. D. Wald, "Associative an Parallel Processors", A.C.M. computing surveys, vol. 7, no. 4, dec. 1975, pp. 215-255.

TOON 81 Toong, H.-M. D., and A. Gupta, "An Architectural Comparaison of Contemporary 16-bit Microprocessors", IEEE Micro, vol. 1, no. 2, may 1981, pp. 26-37.

TREL 79 Treleaven, P. C., "Exploiting Program Concurrency in Computing Systems", computer magazine, vol. 12, no. 1, jan. 1979, pp. 42-50.

TREL 81 Treleaven P. C., and R. P. Hopkins, "Decentralized Computation", Proc. of the 8th international symp. on computer architecture, 1981, pp. 279-270

TREL 82a Treleaven, P. C., D. R. Brownridge, and H. P. Hopkins, "Data-Driven and Demand-Driven Computer Architecture", A.C.M. computing surveys, vol. 14, no. 1, march 1982, pp. 93-142.

TREL 82b Treleaven, P. C., "VLSI Processor Architecture", computer magazine, vol. 15, no. 6, june 1982, pp. 33-45.

VAIL 80 Vaillancourt, S., "A Segmented Microprogrammed Processor", M.A.Sc. thesis, University of Ottawa, 1980.

VAIL 82 Vaillancourt, S., and M. Krieger, "Segmented Microprogramming", 1982 Princeton Conference on Information Sciences and Systems, Princeton, New Jersey, March 1982.

WELC 76 Welch, T. A., "An Investigation of a Descriptor Oriented Architecture", the 3rd annual symp. on computer architecture, 1976, pp. 141-146.

WULF 81 Wulf, W. A., "Compilers and Computer Architecture", computer magazine, vol. 14, no. 7, july 1981, pp. 41-47.

YAU 77 Yau, S. S., and H. S. Fung, "Associative Processor Architecture - A Survey", A.C.M. computing surveys, vol. 9, no. 1, mar. 1977, pp. 3-27.