

A HYBRID MIMD/DF COMPILER FOR PARALLEL PROCESSING

Noraddin Nakhae B. Tech, M. Sc.

This thesis is submitted to the Council for National Academic Awards in partial fulfilment of the requirements for degree of Doctor of Philosophy.

Nottingham Polytechnic

Department of Computing

March 1992

40 0690186 5



ProQuest Number: 10290201

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



ProQuest 10290201

Published by ProQuest LLC (2017). Copyright of the Dissertation is held by the Author.

All rights reserved.

This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

This copy of the thesis has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the thesis and no information derived from it may be published without the author's prior written consent.

Abstract

A new parallel detection algorithm is devised based on the automatic construction and execution of Petri nets for sequential source programs. The algorithm forms part of a hybrid data-flow and MIMD compiler written in POP-11 and accepts Pascal-S source code.

During the compilation process a Petri net model of the input program is constructed. Execution of the resulting net generates a multi-layered code to reveal the full parallelism inherent in the source program. Each layer consists of several independent parallel statements, which are statically allocated to available processing nodes. The allocator optimizes the communication overhead by using a novel static load balancing technique. Both medium and fine grain parallelism are exploited. Fine grain parallelism is implemented by introducing the co-processor concept.

The implementation offers several other novel features including table-driven analysers (potentially adaptable for different source languages), an algorithm for manipulating symbol tables, and combining parallelism detection and scheduling to eliminate the multiple assignment problem.

Full description of all algorithms including many examples is provided.

Acknowledgments

The author would like to thank Dr. D. Al-Dabass for his guidance and support in completing this research work.

I would also like to thank my wife Roxana and sons Roozbeh and Arash for putting up with me while I was preparing this thesis.

May I also extend my gratitudes to all the people at 3L who gave me encouragement and support. In particular, I am grateful to Dr. Peter Robertson and Mr. Ian Young in providing me with much practical help and advice.

Contents

List of Figures	iv
List of Tables	vii
1 Introduction	1
1.1 Overview	1
1.2 Architectures	4
1.2.1 Data Driven Systems	5
1.2.2 Control Driven systems	10
1.3 Languages	12
1.4 Models	15
1.4.1 DFDs and DFGs	16
1.4.2 Petri Nets	18
1.5 Compilers	19
1.6 Objectives	22
2 Current Research	25
2.1 Overview	25
2.2 Architectures	26
2.2.1 Nottingham Polytechnic's MINNIE	26
2.2.2 Nottingham University MUSE Machine	28
2.2.3 Distributed Data Processor Machines (DDP)	31
2.2.4 The Hughes Data Flow Multiprocessor	32
2.2.5 Manchester Data Flow Machine	35
2.2.6 Other Data Flow Machines	38
2.3 Data Flow Languages	40
2.3.1 Lucid	41
2.3.2 VAL	44
2.3.3 SISAL	50
2.4 Parallelism Detection Techniques	53
2.4.1 Paraphrase	54

2.4.2	Russell's Method	55
2.4.3	Maekawa's Method	55
2.4.4	Bird's Method	58
2.4.5	Interval Analysis Technique	60
2.4.6	The Paralyzer	61
2.4.7	Parallelism Detection for Applicative Languages	62
2.4.8	Parallelisation of CALL statements	63
2.5	Symbol Table Manipulation	64
2.5.1	Use of Hash Tables	64
2.5.2	Binary Tree	67
3	A Hybrid Compiler with Automatic Parallelism Detection	69
3.1	Net Construction	69
3.1.1	Definitions	70
3.2	Net Execution for Parallelism Detection	88
3.2.1	Eliminating Forward Data Dependencies	90
3.2.2	Introducing False Data Dependence	92
3.3	A Hybrid Scheduler	93
3.3.1	Scheduling for MINNIE	96
3.4	Fine Grain Parallelism Detection	101
3.5	Symbol Table	108
3.5.1	A Dynamic Sparse Symbol Table	109
3.6	Variable Storage	113
4	Implementation	116
4.1	The Development Environment	116
4.2	Net Construction	117
4.3	Net Execution and Parallelism Detection	118
4.4	Scheduler	119
4.4.1	Fine Grain Parallelism Detection	120
4.5	Overall Structure of the Compiler	120
4.6	User Interface	122
4.7	Lexical Analyser	123
4.8	Syntax Analyser	125
4.9	Parsing	126
4.9.1	Tree Generation	127
4.9.2	Tree Display	128
4.10	Symbol Table Manipulation	129
4.11	Code Generation	130

5	Results and Discussions	131
5.1	Introduction	131
5.2	Single Node Configuration	132
5.3	Multi-Node Configuration	134
5.3.1	A Worked Example	135
5.3.2	Parallelism Detection for Assignment Statements	146
5.3.3	FOR Loops	150
5.3.4	Mixed Assignment and "For" Loop Statements	156
5.4	Fine Grain Parallelism Detection Test	158
5.5	Benchmark	163
6	Conclusions and Future Work	168
6.1	Conclusions	168
6.2	Possible Future Work	171
	Bibliography	175
A	Introduction to Petri Nets	186
A.1	Historical Background	186
A.2	Petri Net Automata	187
B	The Compiler	191
B..1	The Compiler Structure	193
B..2	A Table-Driven Lexical Analyser (TDLA)	195
B..3	A Table-Driven Parser (TDP)	196
B.1	Translation	198
C	Lexical Rules of Pascal-S	201
D	Syntactic Rules of Pascal-S	203
E	Formal Design of the Compiler Using JSP	208
F	A Sample Test Program for the Code Generator	226
G	A Sample Parallelised Test Program	234
H	Programmers Model of MINNIE	247

List of Figures

1.1	A Simple Data Flow Diagram	7
1.2	Cell Representation of a DFG	7
1.3	Functional Blocks of a DF Machine	8
1.4	Organisation of a processor array multiprocessor	11
1.5	A Branch Node	16
1.6	Showing 'deadlock' and an 'infinite loop' in data flow graphs	17
1.7	The Correct Way of Showing a Loop Construct	18
1.8	A Loop Without Deadlock	20
1.9	A <i>WHILE</i> Loop	20
2.1	The block diagrams showing the architecture of MINNIE.	27
2.2	The structure of the MUSE machine	29
2.3	Table of environments and streams for MUSE	31
2.4	Representation of some constructs as data flow actors	33
2.5	The architecture of Hughes DF machine	34
2.6	Block diagram of Manchester data flow machine	36
2.7	The performance of Manchester DF	39

2.8	A sequential program model and its equivalent parallel model	56
2.9	An augmented data dependency graph	59
2.10	A hashed symbol table organisation.	65
2.11	A typical information field of a symbol-table.	67
3.1	A sample Petri net model and its internal representation	71
3.2	A sample program and its input and output sets	74
3.3	A loosely connected Petri net model	81
3.4	A strongly connected Petri net model	82
3.5	A loop with forward dependence	85
3.6	Model of a loop without forward dependence	87
3.7	A Petri net model showing forward dependence elimination in a loop	91
3.8	A Petri net model showing a loop with forward data dependence in its body	94
3.9	A tree showing the effect of operator precedence.	106
3.10	A tree showing the effect of expression regrouping.	107
3.11	A typical symbol table structure	111
3.12	Entries and the pointers in the dynamic symbol table	112
4.1	Overall structure of the compiler in terms of a Petri net	121
4.2	Internal structure of the lexical analyser in terms of a Petri net . . .	123
4.3	Overall structure of the syntax analyser in terms of a Petri net . . .	125
B.1	The difference between a compiler-compiler and a TDC	192
B.2	Overall structure of the compiler	194

B.3	The block diagram of a TDP	198
B.4	A typical syntax tree	199

List of Tables

5.1	Some test results for variable scoping.	167
5.2	Allocation of iterations of a parallelised for loop in a multi-node system	167

Chapter 1

Introduction

1.1 Overview

In an attempt to reduce program execution time, much attention has been given to the processing in parallel of various program components. These components range from the level of subroutines to the level of micro instructions [Gonzalez 71].

The need for faster computers in solving problems in such diverse applications as weather forecasting and natural language processing has been increasingly apparent [Treleaven 79].

By definition, parallel processing is the simultaneous processing of two (or more) portions of the same program by two (or more) processing units [Baer 73]. Among the latter, I/O processors are excluded; i.e., the overlapping of I/O operations with arithmetic or logical instructions is not considered to be parallel processing. Furthermore, parallel processing should not be confused with multi-programming, which is the time and resource sharing of a computer system by a number of programs resident simultaneously in primary memory.

Technological developments have promoted significant improvements in computer speed, but these advances will approach physical limitations, beyond which improvements in computer architecture and compilation technology are required to solve larger problems [Desrochers 86]. These limitations can be broadly grouped into two areas [Cytron 78] :

1. The physical properties of the devices used to construct the components under consideration.
2. The architectural characteristics of the collected components within a system.

Since the fundamental limitations of electron flow, gate switching and the overall characteristics of the traditional von Neumann architecture necessarily restrict the ultimate speed attainable in executing large programs serially, therefore, parallel processing seems to be the most promising way forward.

Flynn [Flynn 66] outlined four classes for the organisation of high speed computers. These four classes are based upon the nature of the instruction and data streams used. These two streams are classified according to their degree of multiplicity, either single or multiple. Thus there are four possible classifications;

1. single instruction single data (SISD)
2. single instruction multiple data (SIMD)
3. multiple instruction single data (MISD)
4. multiple instruction multiple data (MIMD)

The first organisation is the familiar von Neumann architecture which consists of a processor and some Random Access Memory (RAM) where the instructions and data reside [Ibbett 82]. The second and third class although very powerful in special

applications, e.g. vector processors in class two, they can not be regarded as general purpose computers.

Finally, the last class encompasses a diverse group of computers which are radically different in their behaviour and the style of their programming, but they all share the principle that multiple instructions and data are processed simultaneously on the Processing Units (PU) of the system when possible. For example, ALICE graph reduction machine [Darlington 81], the Manchester data flow machine [Gurd 84] and the Alliant FX series [Lackey 86] are all MIMD machines of different architecture.

This new generation of computers poses a number of challenging questions to computer scientists and the computer industry. Are existing imperative languages, which were designed for sequential machines, suitable for the new computing environments? The answer is perhaps no. As a result, research has concentrated on adapting existing languages for new computers by adding extensions to them [Brinch 75, Meakawa 76]. Such extended languages have been particularly suitable for networked and loosely-coupled systems in which each element is essentially a von Neumann machine, e.g. parallel Pascal.

Another area of research has concentrated on new programming languages specially designed for parallel computers; Occam [Ericson 87] and SISAL [McGraw 83] are two such languages. Occam being tailored for transputer networks and SISAL for data flow machines.

Insights into the nature of parallel processes can be gained from the study of formal models on the one hand and from the detection of potential parallelism in existing sequential programs on the other. These provide a sound basis for the development and evaluation of languages and architectures intended for parallel processing [Volansky 70]. Furthermore, it is economically important to use existing software as much as possible, although in some cases redesign of existing algorithms to new parallel ones may be inevitable.

Another question raised is: are programmers to be responsible for the task of efficiently using the many processors on these parallel systems and indicating the appropriate parallelism in the software? Explicit parallelism, has a number of drawbacks. Firstly, as stated by Maekawa [Maekawa 76] "if GOTO statements are harmful, then the construction of parallel programs by users would be more dangerous". Human nature makes the task of verification of parallel processing programs much more difficult. Automatic detection of parallelism by compilers is thus more desirable for constructing more reliable programs and reducing programmers' workloads and hence increasing their productivity.

Secondly, without careful analysis it is likely that the user will fail to take full advantage of all the parallelism available in the problem. High level parallelism at the process level is easily detectable and leads to concurrent processes being executed in parallel. However, low level parallelism at the instruction level is more tedious to detect and is likely to be ignored [Barrett 85]. It is most unlikely that the explicit approach will allow users to take full advantage of systems with many hundreds or thousands of processors.

Hence automatic support of parallelism has grown rapidly and led to the creation of Data Flow machines and their associated compilers where parallelism at the instruction level is supported automatically. Furthermore, parallelizing compilers allowing detection of coarser grain parallelism have been developed for other types of MIMD computers [Kuck 86].

1.2 Architectures

MIMD machines can broadly be classified into four groups [Duckworth 84]:

1. Demand driven
2. Data driven

3. Control driven
4. Hybrid, a combination of the above.

Types 2 and 3 are of particular interest to our discussion and will be studied in more detail in separate sections.

Demand driven or Reduction systems cause activity to be triggered by the demanding of a result, hence the name. These demands cause further demands for results and eventually, at some point, the demands find actual values rather than more subprograms. These results are then combined operationally providing results which are the values required earlier on. This process eventually provides the overall program results. Demand driven systems are graph based and the demands pass down the graph until they reach nodes that have the required data, at which points the results start moving back upwards through the graph. This approach can be viewed to be a "top-down" method. Normally this method leads to a lazy evaluation strategy, the only results that are calculated being those that are actually needed.

Although demand driven systems are not particularly suitable for arithmetic expression calculations, in contrast to data flow models [Allsopp 86], they have the advantage of having fewer *race condition* possibilities. This type of system proved to be very useful in the implementation of functional and logic languages. For example ALICE [Darlington 81] being developed at Imperial College supports the parallel functional language PARLOG [Clark 86].

1.2.1 Data Driven Systems

In this group of parallel processors it is the sequence of data, not instructions, that controls the execution of the machine. In a data flow architecture, operations on data occur when the operands themselves are ready to be operated on. This

is in contrast to von Neumann architecture in which the sequence of instructions determines when an operation is to be performed. Thus, the data driven systems lack program counters or sequencing facilities.

Furthermore, in a data flow machine there is no global memory in which values can be held for multiple access. Because of this, identifiers must be treated in a different manner to those on conventional computers. Results must be passed to all the instruction nodes which require that result.

In a similar way to demand driven systems, any particular computation is represented in a graphical way by operations as nodes, and data dependencies as arcs. For example, consider the expression :

$$((X * Z) - (4 * 5))/X$$

Figure 1.1 is the data flow representation of this expression which shows the flow of data to each node.

Clearly, the availability of data for operation '*' depends on the previous statements. The value of 'X' is needed in two instances and hence the operation DUP is used to duplicate the value. The only condition needed to perform the operation of a node (firing rule) is the availability of all data for that node.

The abstract form of data flow programs, however, does not provide a good mechanism for physical implementation. For this reason, most data flow architectures utilise packaged units that contain the operation to be performed, the data to be used as operands, and the destination process for the result. These packages are sometimes referred to as *cells* and appear as shown in figure 1.2.

The organisation of a typical data flow machine is depicted in figure 1.3 and is often referred to as *circular pipeline*.

The instruction execution unit executes the operation specified by a node on its as-

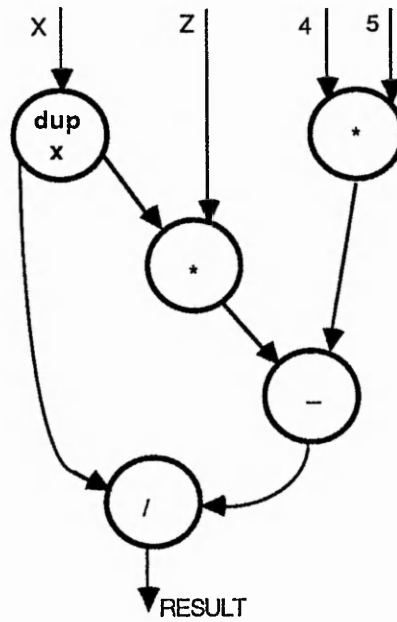


Figure 1.1: A Simple Data Flow Diagram

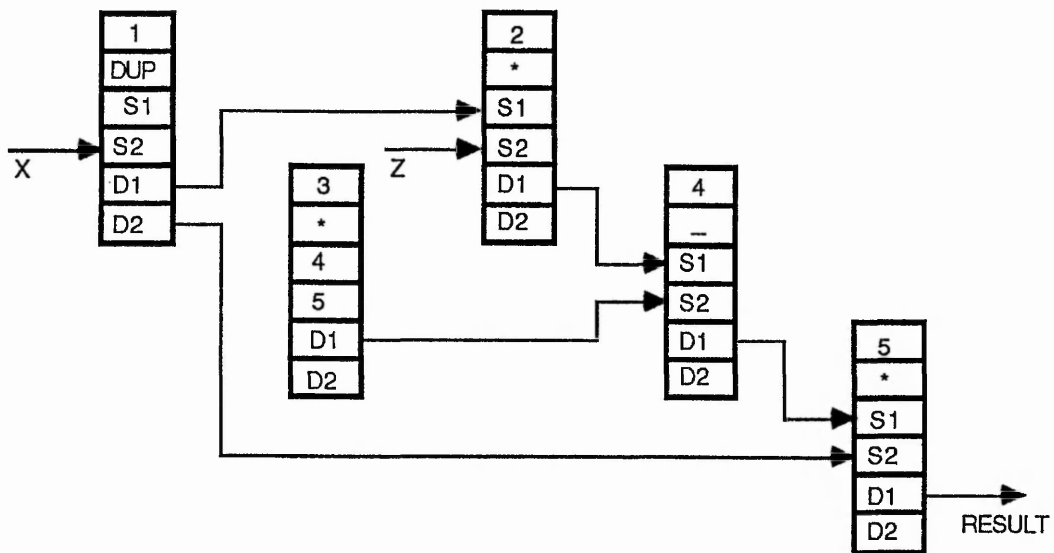


Figure 1.2: Cell Representation of a DFG

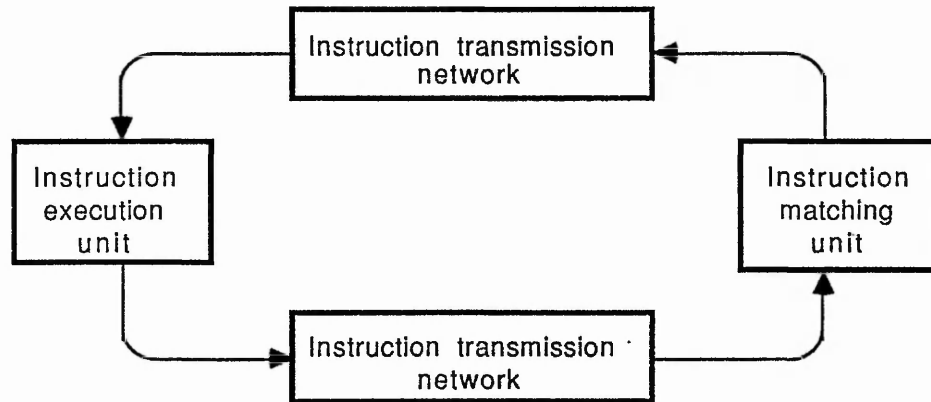


Figure 1.3: Functional Blocks of a DF Machine

sociated input data. The generated result is transmitted by the result transmission network which is then matched to the cells waiting for that result. The instruction transmission network checks the availability of “ready for execution cells” and allocates them to the executing units.

Limitations:

Despite its power, the pure data flow system suffers from a number of drawbacks which makes a hybrid between data flow and other organizations desirable. However, data flow graphs which are purposely designed for pure data flow machines may not be suitable for hybrid machines.

1. The absence of storage (in the form of variables) in data flow requires data items to be copied logically. This causes excessive copying particularly when large data structures are used [Srini 86]. For example, if the value X is to be stored in the i 'th element of an array, then conceptually a new structure has to be created with the value for the i 'th element changed and the rest of the elements having the same set of values as the original array. Clearly a shared memory MIMD machine does not suffer in this way.

2. A major requirement of most programming languages is to support iterative loop constructs and recursive procedure calls. Lack of variable storage does not allow any easy implementation of these language constructs. In static data flow machines the nodes of the graph are created at compile time and are not changed during execution. This leads to a problem in the implementation of reentrant code. Thus, recursive procedure calls are not possible in static data flow machines.
3. To preserve the value of variables generated by the different invocations of a procedure there is a different class of data flow machines called dynamic data flow machines that generates copies of nodes and subgraphs at run time (dynamic code copying). This method has the obvious advantage of supporting reentrant code and specially recursive procedure calls but has the overhead of creating nodes at run time which can be expensive both in terms of hardware and execution time. There is another method called dynamic code labelling [Gurd 85] which labels the tokens flowing through the system. The node execution rules are extended to allow operations to proceed only if operand tokens with the same label values are present.
4. In data flow machines the job of matching data to nodes requiring them is carried out by a functional unit called the *matching store*. A symptom called *starvation* can arise if the number of processing units is increased beyond a certain limit. This is due to the limitations of matching store not being able to generate enough matched cells to keep the majority of the processing units busy. On the other hand, if the number of processing units is not large enough then there can be a *race condition* whereby a pool of instructions is ready for execution but there is not enough processing resource. This means a delicate balancing act has to be done depending on the amount of parallelism present in the particular application.

Furthermore, a total lack of control flow in the execution of programs in data flow machines creates a situation where debugging and tracing of program execution in

the development of a program becomes very difficult, if not impossible.

1.2.2 Control Driven systems

In contrast to data flow machines there are other classes of MIMD machines that are not based on the concept of data flow control. Figure 1.4 depicts the general form of this class of multiprocessors. Cray X-MP is an example of a computer belonging to this group.

Here each processing node is typically a von Neumann machine which executes a set of instructions allocated to it either dynamically at run time or statically at compile time. However, each node is allowed to read or write to a global memory where common data are kept. In order to preserve the data dependency of a program, a number of synchronisation primitives are used.

In Cedar multiprocessor system [Fang 87], each synchronisation variable has two fields: Key and Data, to store synchronisation information and variable value respectively. The format of a Cedar synchronisation instruction is given by :

$$\{X; \text{test on } X.\text{key}; \text{operation on } X.\text{key}; \text{operation on } X.\text{data}\}$$

Here X is the synchronisation variable name. The **test** on X.key specifies the condition to be tested between the key fields of X and a value provided by the instruction. Operations on X.key include increment, decrement, add, fetch, store, fetch and increment, fetch and decrement and No-op. Operations on X.data include fetch, store and no action. The whole synchronisation instruction is executed in each shared memory module and is indivisible.

Cedar synchronisation primitives are very effective in handling low level synchronisations required in numerical computations.

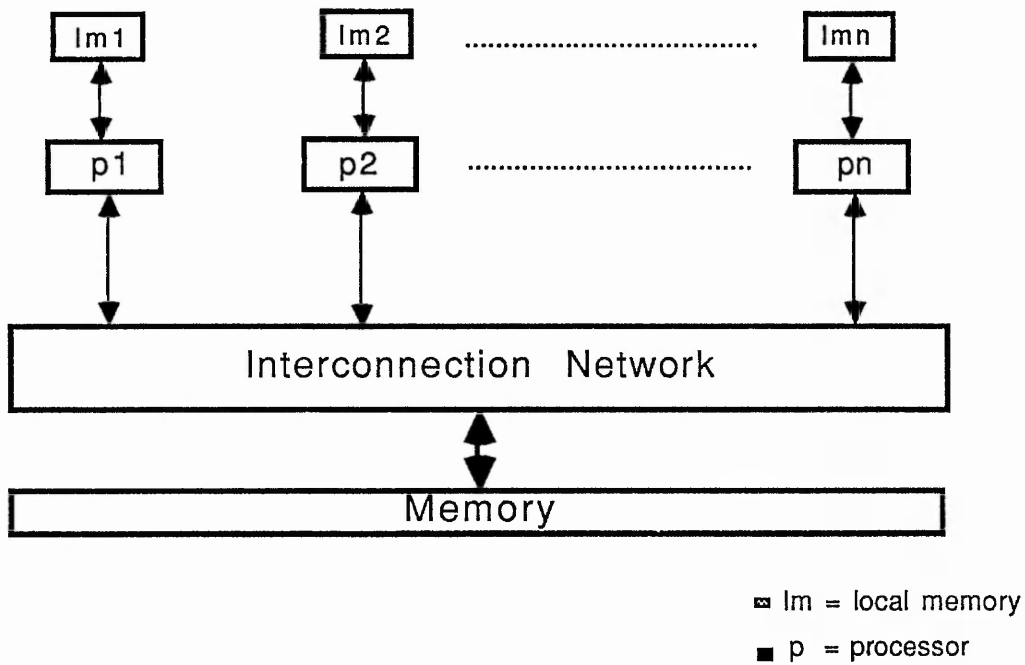


Figure 1.4: Organisation of a processor array multiprocessor

Some of the problems encountered in data flow machines are not present here, however, there is a very large synchronisation overhead that may prove to be critical in cases where there is a large amount of fine to medium grain parallelism present. In general in this type of machine where all the synchronisation is carried out by software, in some cases the overhead becomes unacceptable particularly for fine grain parallelism. Addition of some hardware support for reducing this overhead is highly desirable.

A major drawback of these systems is that the complexity of the interconnection network increases very rapidly with the number of processors. Hence the cost limitation of adding more processors to the system hinders the construction of large multiprocessor systems of this type.

1.3 Languages

To achieve parallelism, algorithms and languages for parallel hardware have developed in a variety of ways. These developments have included new languages, numerical methods and a variety of schemes to analyse programs to exploit simultaneous processing, the latter including both hardware and software devices [Kuck 72].

Utilisation of user defined parallelism (explicit parallelism) in imperative language programs necessitates the introduction of special constructs to represent the new dimension of parallel computing. The basic concept behind these extensions is that of *FORK-JOIN*, *SPLIT-ASSEMBLE* or *START-HALT* primitives [Al-Dabass 87]. Equivalent extensions in higher level languages include;

- DOTOGETHERE - HOLD
- FORK - JOIN - TERMINATE
- PARBEGIN - PAREND

All of these work on a local block basis.

The hazards of explicit parallelism have already been discussed but there exist situations where inherent parallelism in a piece of code cannot be revealed by any other means. Consider the following Pascal code :

```
for I := 1 to 10 do
  begin
    if A(I) = 0
    then
      begin
        J = I;
        goto LABEL
```

```
        end;  
    end;  
LABEL :
```

which represents a sequential process looking for the first null element in an array. Although the last action is purely parallel, no automatic analysis can detect the parallelism inherent in the code. Thus, language extensions described earlier can enhance the power of an automatic analyser.

Compilers with Parallelism Detection:

A different technique for identifying the parallel paths in ordinary sequential programs is the automatic parallelism detection by a compiler. Generally speaking, all of the existing methods are based on some sort of analysis of the various dependencies in the program. These dependencies can be any of flow, data and data output dependencies. A further dependency is called data anti-dependence [Wolfe 82].

If $s1$ and $s2$ represent some statements then $s2$ is data flow dependent on $s1$ if $s1$ can compute a value and store it into a variable x ; and $s2$ might use the value of x later on in program execution. $S2$ is data anti-dependent on $s1$ if $s1$ uses the value of a variable x , and $s2$ might store a new value into x later on. $S2$ is data output-dependent on $s1$ if $s1$ stores a value in a variable x and $s2$ might store a new value in x later on.

Data anti-dependence and data output-dependence are artificial dependencies. They appear in standard computer programming languages, but in some functional and logic languages these kind of data dependencies can not happen, due to rules governing the construction of programs in these languages.

Some of the techniques in parallelism-detection entail the construction of some sort of data dependence graphs (DAGs) based on the various dependencies discussed

above [Kuck 82]. One of the better known of such systems is PARAFRASE developed at the University of Illinois, which is claimed to be a very powerful system in detecting the parallelism in Fortran programs; but even it has difficulty in detecting all the parallelism in a program as shown.

Special Languages:

The first step towards the creation of new programming languages to enhance the power of multiprocessing environments was taken as early as 1962. Brown [Brown 62] proposed a method 'free, at least above a certain minimal level, from the kind of over specification of sequence constraints that characterise present programming'. His ideas, together with the advent of non control-based architectures, led to the development of a new class of programming languages, both functional and logical, based on the single assignment rules. In conventional languages reuse of variables is very common, mainly to save memory space. But if a variable is used in several independent loops as a control variable then it would be difficult to detect that all these loops can be executed in parallel. With this sort of problem in mind, languages that aid rather than hinder parallelism detection were designed. A few examples of the more recent languages in this category will be discussed fully in the next chapter. However, features seen as desirable in this class of language can be listed as [Perrott 86] :

- *No inherent sequentiality*

Statements can be executed in any order without affecting the result of program subject to data dependencies.

- *Clean semantics*

All functions are pure functions, and these can have many separate invocations running in parallel.

- *No side effects*

The prohibition of reassignment, and hence the lack of variables, means that each variable is in fact a name for a value and that value can not be altered anywhere in the program. This prevents the side effects that are prevalent in imperative languages.

- *Increased locality*

This is of great use in systems that have a very large number of processors, as the decreased need for global communication makes it much easier to identify independent sections, and to run these sections on different processors without greatly increasing the communication overhead.

Some of these languages, e.g. Lucid [Ashcroft 77, Ashcroft 85], also facilitate easy program-proving and formal program verification.

1.4 Models

The design of a system, software or hardware, essentially consists of building a model of the system which is then implemented. This model can be analysed to derive important system characteristics and detect design errors before any implementation is attempted. This implies that the form of design implementation is of primary importance in system design [Varadharajan 88]. A number of representation schemes for software systems have been proposed over the last decade, ranging from very graphical to highly analytical forms; e. g., flow charts, structured diagrams, data flow graphs and Petri nets. Some of these methods have dynamic characteristics and can therefore model the dynamic behaviour of a program, e. g. data flow graphs and Petri nets, while others are viewed as static modelling tools.

For automatic parallelism detection, however, the reverse of the design process needs to be carried out. Given a program, a model representing it is built, by say

a compiler, and then analysed to detect the parallel paths in the program.

1.4.1 DFDs and DFGs

Data flow graphs are extensively used for modelling programs for data flow machines. The aim here is to model computational processes in such a way that they closely resemble how computers actually operate [Miller 73]. As there is no control mechanism in data flow machines other than flow of data, data flow graphs that essentially show flow of data are a very suitable choice.

Conditional instructions, however, present a difficulty which requires special nodes to implement controlled branching. The conditional jump of a control flow program is represented in a data flow graph by BRANCH nodes. The most common form is the one depicted in figure 1.5;

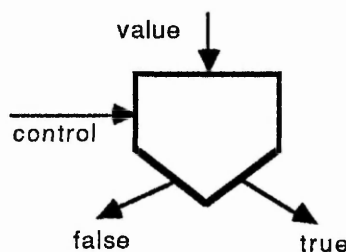


Figure 1.5: A Branch Node

A copy of the token absorbed from the value port is placed on the true or on the false output arc, depending on the value of the control token.

Figure 1.6 illustrates problems that may arise when the graph contains a cycle [Veen 86] associated with the modelling of iterative and recursive constructs. The simple graph on the left will deadlock unless it is possible to initialise the graph with a token on the feedback arc. The node in the graph on the right will never stop firing once started. To prevent these problems special precautions must be taken.

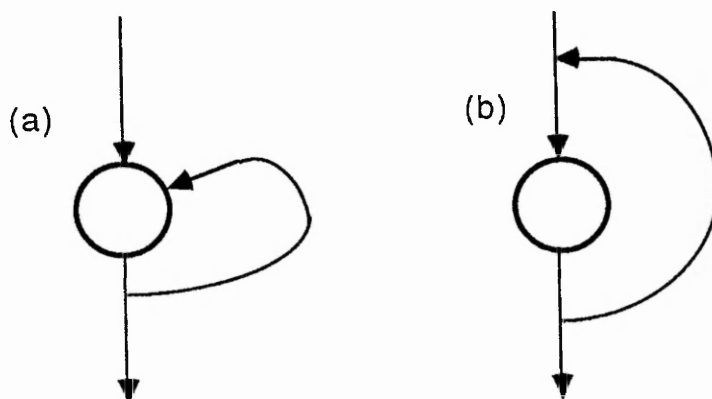


Figure 1.6: Showing 'deadlock' and an 'infinite loop' in data flow graphs

Thus, this maximum concurrency can lead to indeterministic behaviour unless special measures are taken. Reference [Veen 86] gives full coverage of these problems. Figure 1.7 illustrates the correct way of showing the *while* loop represented by the following piece of code:

```
while f(x) do
  begin
    x := g(x);
    y := g(y);
  end;
```

Subgraph *g* is an example of a reentrant graph; its nodes can fire repeatedly while the strict enabling rules ensure that before subgraph *g* is fired the compound branch nodes cannot fire.

In an ordinary program there may be a sequence of I/O statements. For example a number of write statements may simply be outputting a series of messages one after the other according to their sequence in the program. In this situation there is no data dependency, but only control dependency determines the sequence of execution. DFGs are not able to show this sort of dependency.

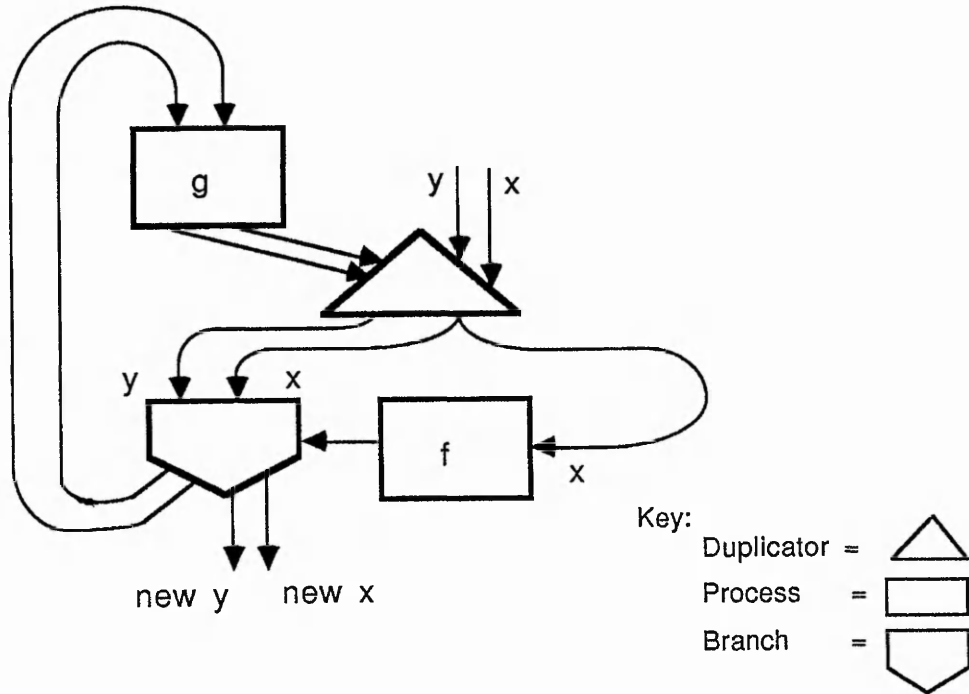


Figure 1.7: The Correct Way of Showing a Loop Construct

1.4.2 Petri Nets

A Petri net is an abstract, formal model of information flow [Peterson 77] which allows one to bring a high degree of mathematical formalism to the modelling process, and hence to the analysis of systems which exhibit parallelism. Systems best suited to Petri net modelling are those with very many interacting components [Peterson 79]. A full treatment of Petri net theory is given in appendix A.

A marked Petri net graph models the static properties of a system, much as a flow chart represents the static properties of a system, be it hardware or software. In addition, a marked Petri net also shows the dynamic behaviour of a system as a result of its execution.

Marked Petri nets are able to model all the common constructs found in programming languages together with any sequence and data dependencies in any program. Figures 1.8 and 1.9 show equivalent diagrams to DF graphs of figure 1.6 (a) and

1.7.

The graph of figure 1.8 will never get into a position of deadlock. As soon as a token is put on the place 1 (p1) the transition can fire and be ready for the next reentrancy, since the token removed by execution of the transition is immediately replaced by the transition itself. The situation depicted in figure 1.6 (b) simply does not arise.

The while loop depicted by figure 1.9 is easier to understand compared to its DFG model and would closely match the way that a computer executes it.

Petri nets are primarily chosen as our modelling technique for extracting parallelism in programs since they satisfy all the features that are required from a scientific modelling tool. These features are listed [Varadharajan 88] as:

- The ability to describe a system at various levels of detail.
- Their graphical and precise nature.
- Their structural generality.
- The existence of analytical tools for determining and verifying the dynamic behaviour of systems from their structure.
- Their ability to represent concurrency.

1.5 Compilers

The design and implementation of compilers and the processes involved is a very well known subject. However, there still remains a number of key questions to be answered when such a problem as parallelism detection is to be tackled.

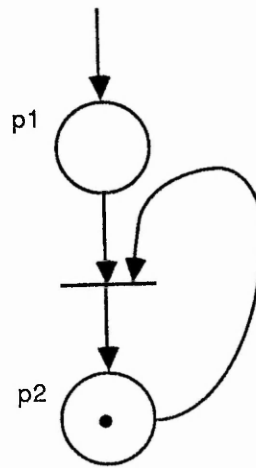


Figure 1.8: A Loop Without Deadlock

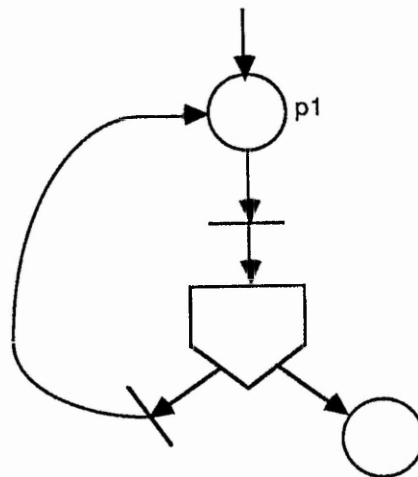


Figure 1.9: A *WHILE* Loop

With the advent of powerful compiler-compilers such as *YACC* [Johnson 86] it is possible to write a compiler for a given language fairly rapidly. The use of such compiler writing tools however, limits the flexibility of the compiler produced. For example, for parallelizing/optimising compilers there are a number of extra processes that need to be embedded in the compilation process, such as modelling of the source code. To incorporate these extra processes in the structure of the program that these compiler-compilers create can be a difficult and time consuming process. Writing a compiler from scratch could also be very expensive, specially if the aim is to generate compilers for different languages quickly and frequently.

These problems lead to the formulation of a new concept in compiler technology that is fundamental to the compiler being discussed in this thesis and will be argued through in the coming chapters. Briefly, the concept is based on the use of table driven lexical and syntax analysis methods, where users supply the tables appropriate to any language using compiler writing tools like *YACC* [Al-Dabass 86, Al-Dabass 86]. This can be made easier by interfacing a table generator to the compiler system.

The packaged system can be viewed as a general purpose compiler, GPC, that allows a compiler writer to quickly configure the system for compiling different languages. Presently, only the front end of the compiler (lexical and syntax analysers) are general purpose. The extra phases of an optimising/parallelizing compiler normally lie between the tree generation and translation phase and are treated separately.

After the early stages of compilation, a tree structure is normally used to represent the source code. Tree construction involves a significant amount of list processing, as trees are invariably modelled as a list of lists. Modelling of the source program from the syntax tree and the subsequent parallelism detection involves further list processing activities. This calls for the use of an implementation language that is powerful in list processing and allows easy use of AI techniques in parallelism detection algorithms. Further requirements of a compiler implementation language could be noted as follows [Nakhaee 85];

- The implementation language and the language being implemented must have similar characteristics even if it is at a lower level. This point is true particularly for implementing interpreters.
- Allowing recursive calls, as many actions of a compiler can be programmed more elegantly using this feature.
- Advanced list processing facilities, specially if a lot of matching and use of AI techniques is envisaged in a specialised compiler.
- String handling facilities.
- Availability of a rich built-in library of functions and procedures.
- It is also useful if the implementation language supports structured programming, e.g. Pascal.

All of the above points are desirable but not essential.

POP-11, an AI language, is particularly suitable for prototyping and is rich in list processing facilities, and was found to fit the project requirements very well [Laventhol 87, Shadbolt 87, Sloman 86]. The use of POP-11 makes it further possible to devise and use new techniques in the storage and access of data in the symbol table. The new technique has a number of advantages over existing methods and is discussed in full in chapter 3.

1.6 Objectives

MINNIE (Multi Interface Node Network for Iterative Environment) is a prototype parallel processing machine which was intended to demonstrate certain principles, namely the associative addressing technique for data communication among a large number of nodes in a parallel system.

In parallel with the design and construction of hardware (MINNIE), work was also started on the creation of a Software Development Tool (SDT) for it. The software development tool is primarily to assist in the testing of MINNIE and also for highlighting limitations in the current version and suggesting enhancements that can be made in future versions of the machine and their SDT.

It was found desirable to develop a *General Purpose Compiler* (GPC) for an ordinary imperative language like Pascal-S, rather than for a data flow type language for the reasons given in the preceding sections.

The developed GPC must carry out the usual compilation processes, e.g. lexical analysis and syntax analysis, and also create a graphical model of the source code using Petri nets. The net is then executed in order to discover the possible parallel paths in the program at any given time.

Having established the parallel paths in the source code, the compiler must then allocate them to different nodes inserting the necessary code for communication and synchronisation. Translation of the source code is into native 6809 assembly language with the embedded instructions for data communications.

As the proposed system can have a large number of processing units it was decided to support fine grain parallelism as well, in cases where an expression could be divided into a number of sub-expressions and there are more processing elements in the system than parallel paths at a higher level. Considerable effort was used in finding a method for optimum decomposition of a long expression; with due consideration to the machine architecture the concept of *co-processors* was devised for implementing fine grain parallelism.

Chapter 2 contains a review of current research in compiler techniques and languages for parallel processing architectures together with associated aspects of modelling and automatic parallelism detection techniques. Chapter 3 puts forward a new combination of concepts to devise a compiler for parallel systems. The next chapter

gives a detailed exposition of the steps taken in the design and realisation of this compiler. Chapter 5 and 6 present results and their interpretation together with conclusions made and recommendations.

Chapter 2

Current Research

2.1 Overview

The process of devising new compilers for parallel processing systems is unlikely to yield successful results without a thorough knowledge of current and past attempts in the field. Three areas of research are seen to be major contributors to new development;

- the architecture of the machines which execute the object code generated by the compiler.
- languages that help the user to explicitly specify the parts of the program that are suitable for parallel processing.
- techniques for parallelism detection embedded in special software processors which remove the burden from the programmer.

The work reported here spans projects that are either currently in progress, or have made significant contributions to parallel processing.

2.2 Architectures

Computer architecture has been dominated by the *Data Flow* concept over the past decade. Data Flow (DF) machines may be broadly divided into two general types, *static* and *dynamic* architectures. The dynamic machines can be further subdivided into code copying and tagged token machines. Data flow machines with potentially the highest level of parallelism are those of the dynamic data flow type. They employ either code copying or tags to protect the reentrant graphs.

2.2.1 Nottingham Polytechnic's MINNIE

The target machine for the compiler described in this thesis is MINNIE (Multi Interface Node Network for Interactive Environment). The hardware and the compiler were developed concurrently. MINNIE [Hammes 89] could be viewed as a global data flow machine where each processing node is based on Von Neumann architecture. Therefore, each processing node is a computing machine in its own right, processing a task and communicating with other tasks being computed in the network of processing elements. Familiarity with the architecture of this machine is essential for the understanding of some of the material being presented in the subsequent chapter.

The MINNIE Architecture

The prototype mark 1 MINNIE consists of 8 processing nodes. Each node has a 6809 microprocessor, local memory (4K RAM and 2K ROM), and an interface. Figure 2.1 shows the block diagram of MINNIE.

All the nodes are connected via a global bus and communication is based on the associative memory addressing technique.

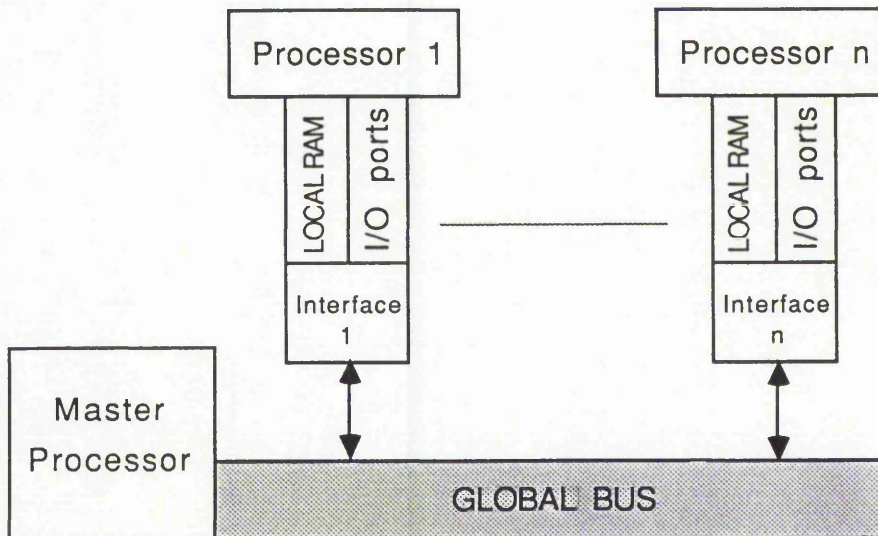


Figure 2.1: The block diagrams showing the architecture of MINNIE.

Any processing node, wishing to send a piece of data, writes the data on to a particular location in the interface (viewed by the processor as another memory location). When the interface obtains control of the bus it places its node address onto the bus combined with the data. All the nodes which contain the address of the node which is sending the data, in their receive registers, will copy the data into one of their input registers.

Similarly for receiving data the processor writes the address of the sending node into a particular location in the interface and whenever that data is broadcast by the sender a copy of it is taken by the interested interfaces. The interface and the processor's work is overlapped. Each node of the current version has one send register and 3 receive registers. In other words each node can only send one piece of data and receive three pieces of data without the danger of any loss of data which may lead to deadlock. Appendix H gives a programmers model of the hardware.

2.2.2 Nottingham University MUSE Machine

This project involves the designing of a static data flow machine. It is based on a structured architecture supporting both serial and parallel processing which allows the abstract structure of a program to be mapped onto the machine in a logical way [Brailsford 85].

The prototype (mark 0) version of the MUSE machine has been built, processing 4 streams and 16 environments. A stream is used, in the MUSE context, to indicate a physical partitioning of the program among the processing resources and environment, or colour, on the other hand identifies different blocks of program code.

The processor for each stream was constructed from AMD bit slice components whilst the sequencer is constructed from SSI and MSI logic circuits. The block diagram in figure 2.2 shows the pipelined ring structure of the MUSE.

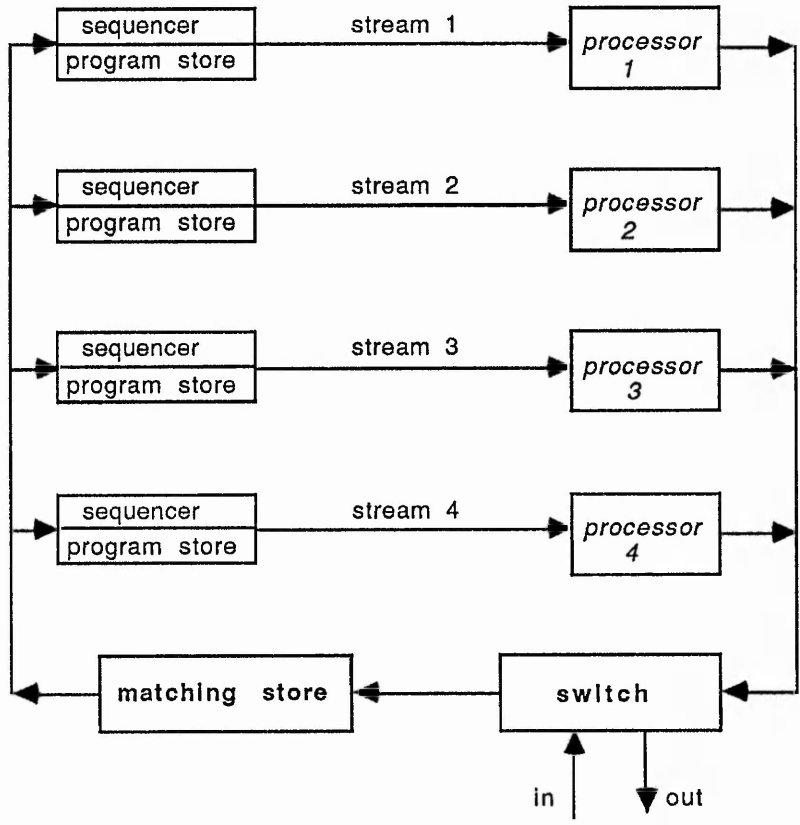


Figure 2.2: The structure of the MUSE machine

The data flow graph representing a program is divided among local memory modules of the nodes or streams. Within each stream further partitioning takes place, by the compiler, to decompose the code into a series of environments.

The processor on each stream accepts a complete node description and carries out the function described by the operation code. The generated result is passed to the switch which handles input and output. If the result is not destined for I/O then it will be passed on to the matching store. This consists of a conventional coordinate addressable memory device which matches tokens destined for the same instruction, i.e., instructions with multiple, and different, operand fields. In contrast to the Manchester machine's matching store, it does not support matching of labels or other fields. When both operands are available for a particular instruction the matching store sends them to the required stream.

The radical diversion of MUSE from other DF machines is the presence of a sequencer at each node which supports demand driven computing within the stream. This gives some measure of control flow to the machine and is similar to the program counter in the Von Neumann model. The lack of support for reentrant code is a major limitation of the system. Also the external bus structure between streams is time multiplexed and may represent a communications bottleneck.

The interesting part of this particular design is the use of what is called *environment switching*. The load associated with each stream is divided into a number of environments. Each environment consists of a set of instructions and a program counter. Whenever computation is held up in any environment on a given stream, an automatic internal interrupt occurs and control is transferred to some other environment where computation can proceed. The different environments are labelled by a number and are referred to by the alternative terminology of *colour*. For example the computations

$$a + b * c - d$$

$$e + f * g/h$$

are divided between the streams and environments as shown in figure 2.3.

		Environments →		
		0	1	2
stream number	0	a; ↓ ₁ ; +; ↓ ₂ ; -	e; ↓ ₁ ; +	
	1	b; c; *; ↑ ₀	↓ ₂ ; h; 1; ↑ ₀	
	2	d; ↑ ₀	f; g; *; ↑ ₁	
	3			

Figure 2.3: Table of environments and streams for MUSE

Down arrows, ↓, show the situation where a result is expected to arrive from another stream, while up arrows, ↑, represent the sending of data to another stream. The send and receive actions must correspond exactly.

2.2.3 Distributed Data Processor Machines (DDP)

This first packet communication data flow machine was built by Texas Instruments in 1980 [Cornish 79].

This machine was designed for executing Fortran programs using the static data flow concept. Although the compiler creates additional copies of a procedure to increase parallelism, this copying occurs statically and DDP uses a locking method to protect reentrant graphs. It is a one-level machine with a ring-structured communication unit, augmented with a direct feedback link for tokens that stay within the same processing unit.

The program graph corresponding to a Fortran program is generated by a compiler in the host processor. Using a cluster detection algorithm the graph is partitioned into a number of subgraphs. The subgraphs are then loaded into the memory units of processing elements [Srini 86].

When a node of the subgraph is enabled, it is executed by the ALU of the processing element. The result of a node execution is forwarded to other nodes, in the processor's memory unit or another processing element's memory unit. Interprocessor communication is via an interconnection network, the E-Bus, carried out as a series of 34 bits packets.

The DDP provides a separate bus, in each processing element, the maintenance bus, to load and dump the contents of memory, monitor the performance of the processor and diagnose the processor when faults occur [Srini 86].

A number of benchmark programs have been run on the processor. The results showed that performance can be improved in a linear manner by adding more processors. The major short-coming is the lack of support for recursive procedure calls. The DDP architecture supports the data types allowed in Fortran iv and a semaphore data type, used for sharing code blocks e. g. subroutines in programs.

2.2.4 The Hughes Data Flow Multiprocessor

This is also a static DF machine developed at the Hughes aircraft company for signal and data processing. According to the original paper [Velder 85] a completed machine was planned for 1985. However in [Veen 86], published in 1986, all results and references to this particular machine were still based on simulation.

A programming environment has been developed which allows high level programming in a language called *HDFL* (Hughes Data Flow Language). HDFL is a functional language similar to VAL, permitting full expression of parallelism. The lan-

guage is value orientated allowing only single assignment to variables. Its features include strong typing, data structures including arrays and records, conditionals, iterations, parallel iterations (forall) and streams.

The compiler developed for HDFL translates a program to a data flow graph form composed of primitive data flow actors. These primitive actors, 39 in all, are implemented directly in hardware. Figure 2.4 shows some of these actors, where figure 2.4(a) is an actor used in the invocation of functions and figure 2.4(b) is a representation of conditionals.

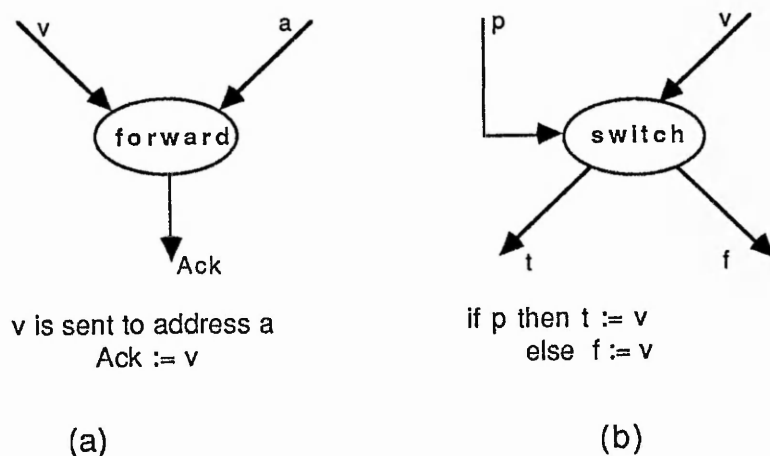


Figure 2.4: Representation of some constructs as data flow actors

The architecture consists of many relatively simple identical processing elements, communicating with each other via a global packet switching network.

The processing elements are arranged on a three dimensional bussed cube network, the maximum distance between any two elements being three. Figure 2.5 shows the physical arrangements of the processing units.

Each processing element consists of two parts, the communications functionality, and a processing engine which performs the primitive actor operations and the data flow sequencing control. Each element has its own local memory which is used for

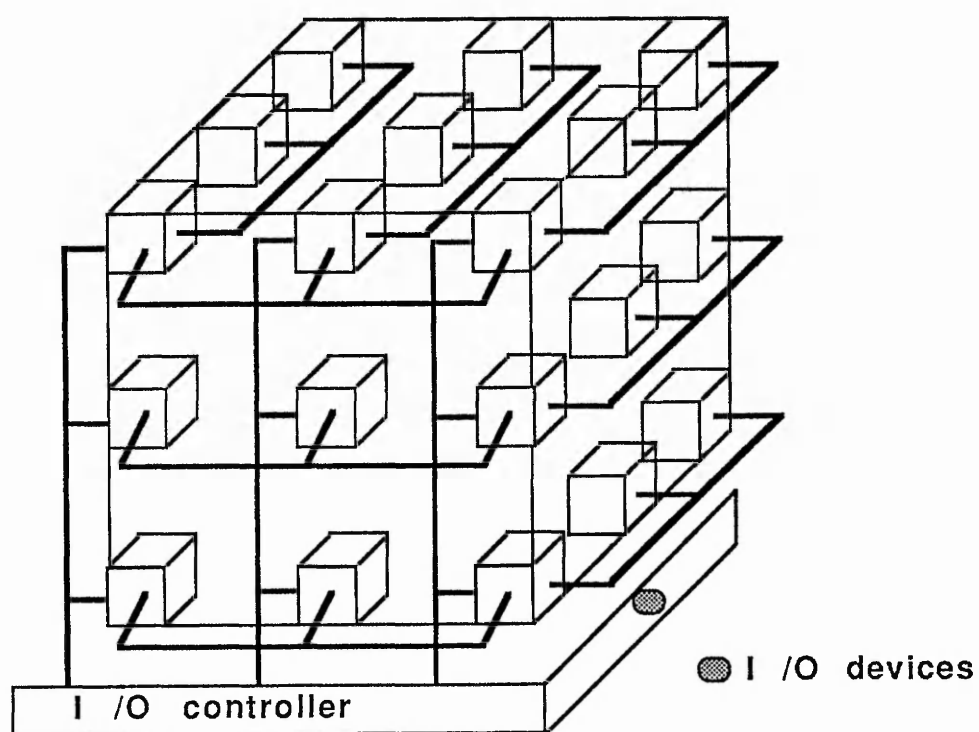


Figure 2.5: The architecture of Hughes DF machine

both program and data storage; there is no global memory. The peak performance is expected to be 2-5 mips.

2.2.5 Manchester Data Flow Machine

This was the first tagged-token dynamic DF machine built by Gurd and Watson at Manchester University [Gurd 85,Gurd 80]. Figure 2.6 shows block diagram of its structure.

The architecture consists of four functional units arranged in a circular pipeline fashion:

- 1. Token queue**

To smooth out the irregular output rates of two other units in the pipeline, the matching unit and the functional unit.

- 2. Matching unit**

To accept tokens from the queue and store either single operands, or form matching pairs of operands. Complete sets of input tokens are sent to the fetching unit.

- 3. Fetching unit**

Combines a set of input tokens with the description of destinations to create an executable code.

- 4. Functional unit**

This unit consists of a number of processors and a distributor which allocates the executable packet to a free node for processing.

All communication paths are parallel, up to 166 bits wide, transmitting a complete packet at a time. Each unit is internally synchronous but communicates with other modules through asynchronous protocols.

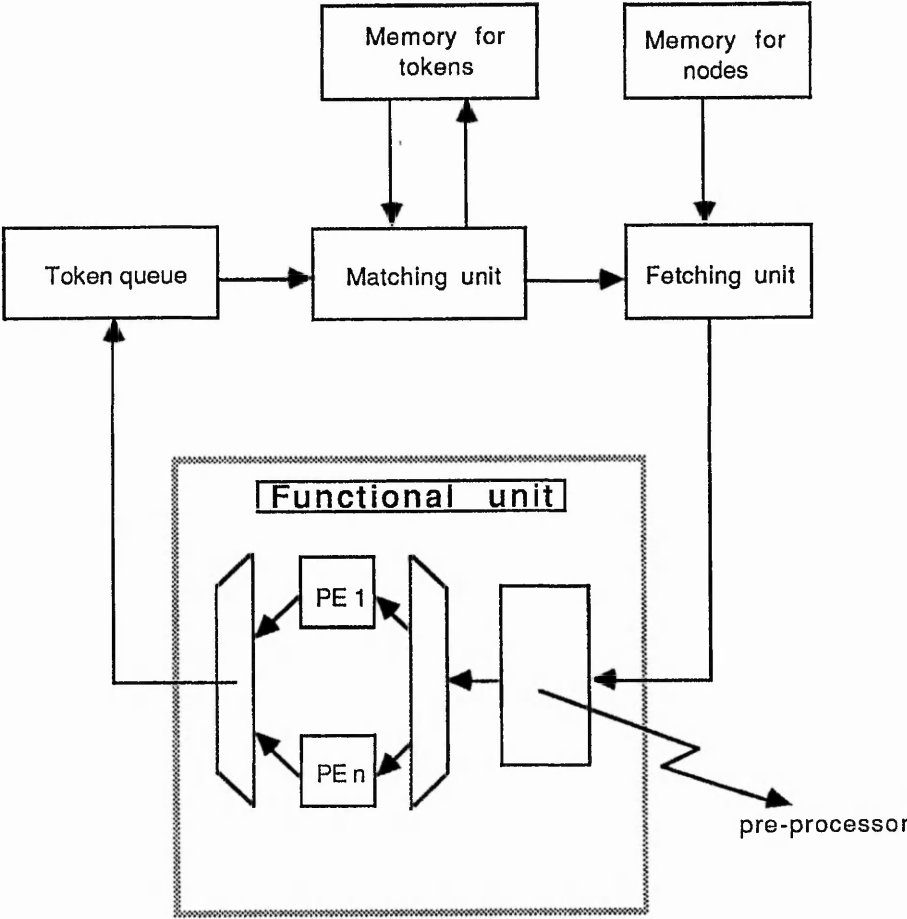


Figure 2.6: Block diagram of Manchester data flow machine

Concurrent reentrancy is supported by means of token labelling. Each activation of a piece of reentrant code generates an associated tag for data values generated. Node firing rules are extended to allow execution only if operands of the same tag-value are present.

Since a further constraint is introduced, that is the presence of the operands of the same tag, the role of the matching store becomes much more critical. To satisfy the additional constraint the matching store must search for a token of equal tag-value to input tokens. If this search is serial then it would severely limit the token throughput rate of the ring.

To increase the matching speed, therefore, an associative check of token tags is required. However it was determined that simulating this by means of a hardware-hashing mechanism is more cost effective [Veen 86].

For programs which do not fill the 16K deep matching store space this mechanism is satisfactory. In cases where a large number of operands are generated before a successful match can take place, the matching store can overflow. In these cases the tokens are forwarded to an overflow unit, with subsequent loss of performance. This symptom is likely to occur when handling large data structures.

In cases where a token does not need to match another, e. g. the input to a monadic operator or a dyadic operator with a literal, the matching operation *BY* (BYpass) is used. *BY* is the simplest matching function, which forces the bypass of the matching unit. As a result, a group containing the token and a piece of data of type *NULL* is constructed and output from the unit. For all other matching functions, the matching store is searched.

Benchmarking the initial model showed a performance of 1-2 mips. However, the prototype is implemented in medium performance technology and an upgrading to around 10 mips is expected by using alternative technologies.

As a result of running a number of test programs, the Manchester group found that

as the number of processing units is increased, the speed of the machine at first increases linearly but then levels off at around 1 mips, figure 2.7. If a program is used that bypasses the matching store then this symptom does not occur. Gurd and Watson concluded that the problem is due to occasional gaps in the sequence of results, causing processors to become idle. The suggested solution is a buffer of matched tokens, the so-called pool, to smooth out fluctuations in the supply of pairs.

2.2.6 Other Data Flow Machines

There are many more data flow projects being carried out around the world. For example the Japanese have embarked on a number of projects in this field [Amamiya 86, Ito 85,

Amamiya and his colleagues designed a data driven system for scientific calculations [Amamiya 86]. The machine was constructed at the electrical communication laboratory of NTT and is optimised for list processing. It contains separate processing and structure elements. Functional units are integrated with the structure elements as well, since many nodes are expected to operate on structures. The constructed experimental system *EDDY* (Experimental system for Data Driven processor array) has a 4 by 4 array of PEs and two broadcast control units to handle program loading and I/O. Amamiya has been able to show, by using *EDDY*, that the design is capable of exploiting parallelism sufficiently to support some 64 functional units on even small programs. This machine is a tagged token DF computer.

The institute for new generation computer architectures (ICOT) initiated research on the parallel execution of logic programs. One result is the design for a parallel inference machine [Ito 85] based on data flow. A distributed mechanism allocates a function invocation on the same processing element, on a neighbouring element or on a distant element, depending on the value of a load factor, which is maintained by periodic exchange of information between processing units.

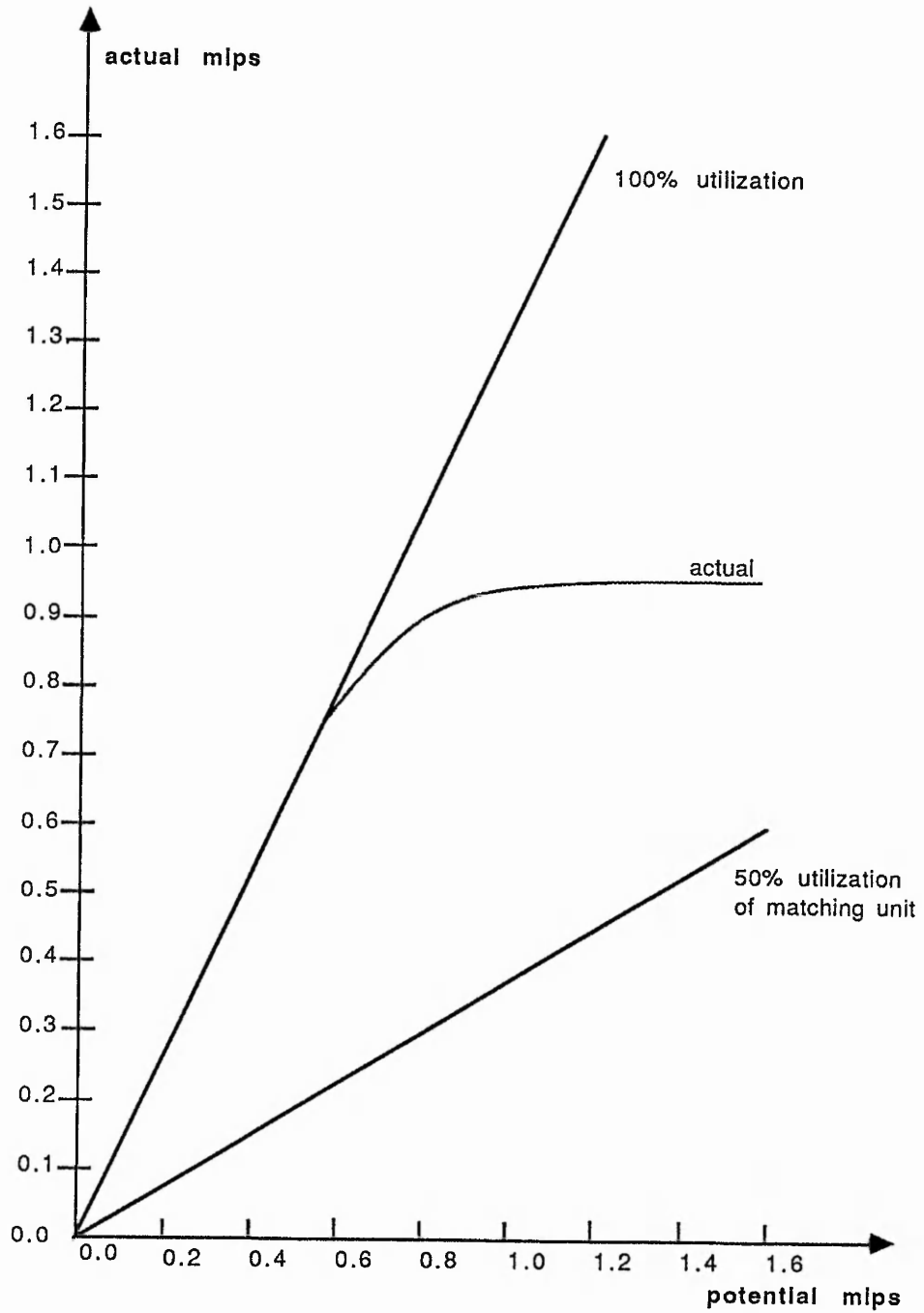


Figure 2.7: The performance of Manchester DF

A prototype has been built consisting of 4 processing elements and 3 structure memories connected by a two-level bus.

In Belgium, a machine was designed (not yet constructed) with elaborate memory management systems [Caluwaerts 82]. Each processing element has its own memory manager, but these managers can also communicate with each other, so that the total memory space is shared. Each invocation of any procedure results in the allocation of a fresh memory area for the tokens belonging to the new invocation. To achieve even load distribution the area is allocated in a neighbouring processing element. Thus, when a node is enabled, its description must be fetched from another processing element. By using caches, local copies are created and in fact memory is paged and complete pages are copied. Data structures are treated in the same way as programs by the memory system managers.

2.3 Data Flow Languages

One of the main advantages of data flow computers is their ability to exploit fine grain parallelism. As already seen in the previous chapter, conventional languages are not able to provide a vehicle to use all the inherent parallelism in a sequential program, despite the development of sophisticated tools such as Parafraze.

The solution adopted by many researchers is to use *Single Assignment Languages* (SAL), first proposed by Tesler and Enea in 1968 [Tesler 86]. The fundamental feature of these languages is that any variable may only be assigned a value once, in the definition of the program or a block. The difficulty arises when iterative constructs are needed. The concept of single assignment means that when an element of a data structure, such as an array, is changed then a brand new array is created and the old array remains the same.

SAL's belong to the class of value oriented , or applicative languages. That is, each data item is considered a value on its own right and is not subject to alterations.

SAL's are also expression oriented; that is, each language construct is an expression which delivers a result. An expression can be part of a larger expression. In this context even a loop construct can deliver a result.

The SAL rule presents a problem for the semantic actions to be taken on statements like $I := I + 1$; within a loop body. Firstly, in each iteration of the loop a new value is assigned to the variable I and secondly because the definition is circular. In SAL languages the problem is solved by insisting that the SAL rule is applied within a loop body and allowing redefinition of variables local to the loop at the boundary between iterations.

Other problems are also caused by the functional semantics of SAL's. For example, lack of history sensitivity makes a pure SAL unsuitable for applications like database management. One solution to this problem is the introduction of concepts like streams. A stream is very much like an array, however, they differ in a number of important details. As an example, the elements of a stream must be accessed in order. That is, an stream is an ordered set of values where each item has an integer index. The result is that it is not possible to access element $n + 1$ before element n . An important property of an stream is that it has no upper bound; ie. there is no limit on the number of elements in the stream.

Because of the importance of SALs to parallel processing some of their major implementations are reviewed below.

2.3.1 Lucid

Lucid can be described as a functional data flow language. That is, it combines the aspects of functional programming with the concept of data flow. By being a functional language, Lucid is a mathematically "pure" language, in which no side effects can be generated [Flaming 86]. Thus, one can express parallel computation, iteration and at the same time utilise the provability properties inherent in

functional languages.

The two main non-mathematical features in programming languages are transfer of control and assignment. However, it is difficult to eliminate them as, iterative constructs seem to make essential use of them [Ashcroft 77].

The way that Lucid brings mathematical respectability to these constructs is by the use of *first* and *next* reserved words. Consider the Fortran statement of: $I = I + 1$. Although this may look like a mathematical assertion, in fact it is mathematically just false. The equivalent statement in Lucid is:

```
first I = value
next  I = I + 1
```

The first statement simply initialises I to a value. The second statement can be viewed as a function with domain $\underline{N} = \{0, 1, \dots\}$, generating an infinite sequence of values for I .

The variables and expressions in Lucid formally denote not single objects, but rather infinite sequence of data objects. The ordinary data operations, relations, and logical connectives work much in the same way. Since in Lucid x denotes a sequence of values $\{x_1, x_2, \dots\}$ and y denotes $\{y_1, y_2, \dots\}$ then $x + y$ would generate $\{x_1 + y_1, x_2 + y_2 + \dots\}$.

In addition Lucid has the binary operator *as soon as* which extracts values from a loop, e. g. ,

```
Next I = I + 1
output = I as soon as condition
```

Therefore in general, a Lucid program is a set of equations specifying a set of variables. A variable may be specified by the following equations where V and V_0 can be arbitrary expressions.

```

first I = V0
next I = V

```

V0 must be syntactically a constant; built up from data constants, terms of the kind first x or x as soon as condition; see examples below:

```

first I = 2
first II = first I
first III = 10 as soon as I eq 3

```

All variables must be specified except "input" and no variables may be specified twice.

Informally, a Lucid program can be regarded as a number of simple loops. If the result of a loop is required in building another loop then the loops are concatenated. For example [Ashcroft 77]:

```

first (i,j) = (1,1);           {1}
next (i,j) = (i+1, (i+1)*j);  {2}
m = j as soon as i equ 10;    {3}
first k = m;                   {4}
next k = k/2;                  {5}

```

The last loop (line 5) needs the value m generated by line 3. Therefore, the result of the (i, j) loop is used to initialise the k loop. So it can not begin until m has been computed. In addition to concatenation, loop nesting is also possible in Lucid.

It is interesting to note that in Lucid, there is no conditional statement, but instead a conditional expression. Therefore, if ... then ... else is also regarded as just another mathematical function. In the following example, the expression iterates x depending on the condition, until x reaches the value of v.

```
next x = if x < v then x+1 else x-1
output = x as soon as x equ v
```

Lucid programs are particularly suitable for representation in a data flow network and execution on a data flow machine [Bush 79]. The advantage of this approach is that independent computations can be carried out in parallel. This type of implementation is favourable as there are no side effects and sequencing other than by data dependence requirements. Other implementations, particularly on sequential machines, have also been reported in the literature [Flaming 86].

2.3.2 VAL

Value oriented Algorithmic Language, VAL, was originally designed by Dennis and Ackerman in 1979 [Dennis 79]. It was primarily devised as a high level language for programming the MIT static data flow machine. VAL has many of its features and concepts based on *ID* and *CLU* [Liskov 77]. *ID* being a SAL and *CLU* a non-SAL language designed more for easy data abstraction.

An area where Val differs markedly from most other languages is its treatment of errors. It supports a large set of error values. For every run-time error that occurs the program returns a value. For example, illegal array access in some languages causes a catastrophic termination of the program where as in Val it merely returns the value *undef*. This philosophy is based on the concept, seen earlier, which requires that all expressions must return a value.

Data Structures and Types

VAL is a strongly typed language, requiring that every value name is declared to be of a specific type. Common types in other programming languages such as integer, real, boolean etc are also found in VAL. Other data structures like records and

arrays are also permitted. User-defined data structures can be created using basic types, through the reserved word *TYPE*.

All structures must have their length defined at compile time and dynamic extension of records or arrays is not possible. Similar to many other scientific languages, both arrays and records come with a set of operators for their manipulation: namely operators exist for creation, element replacement and array concatenation.

VAL also has the capacity for allowing a value to be one of several types, much the same as ALGOL68 *union*, by using the reserved word *oneof*. For example the declaration:

```
Z : oneof [x : real; y : character];
```

is legal and specifies that Z can either take the value of a real or a character but not both.

Assignment

The assignment operator is the familiar “:=” sign. One peculiarity of VAL is its use of concepts like *tuples* and *arity*. A tuple is a collection of values separated by commas whose arity is the number of values it contains. The values in a tuple being independent of each other. Based on this concept, the left hand side in an assignment statement must match the right hand side both in arity and in type. To illustrate, consider the following example;

```
x,y :integer;           %declarations
z :character
x,y := 1,2;            %this is legal
z,y := if condition then
    '2',3
```

```
else
    4,5
endif;
```

The “else” part is not legal since the types mismatch. The concepts outlined above make a program both shorter and neater.

An important semantic observation here is that the values composing a tuple are not dependent on each other and therefore can be evaluated in parallel.

Let Construct

The construct:

```
let
    definitions
in
    expression
endlet
```

is essentially a formalised version of the ID block-expression.

The definitions part can be any collection of statements and the expressions part can be an arbitrary tuple of values. A VAL version of a program to solve a quadratic equation can thus be;

```
r1,r2 : real;
r1,r2 := let
    x : real := sqrt (b*b-4*a*c);
    y : real := 2.0*a;
```

```

in
    (-b+x)/y , (-b-x)/y;
endlet.

```

Note that an object can be declared and defined in either two separate places (as for *r1* and *r2*) or declared and defined at the same place (as for *x* and *y*).

Loops

VAL supports two types of loops; *for loops* and *forall loops*. The *forall* loop is straightforward. The loop takes a range of integer values, and can either create an array using this range, with *construct*, or apply some operator to all the value, using *eval*. *Eval* and *construct* may be mixed in a loop. The following example from [Allsopp 86] is given as an illustration:

```

forall j in [1,n]
    x : real := sqrt (real(j));
    eval plus j*j;
    construct j,x,-x;
endfor;

```

The outputs of this *for* loop are: an integer equal to $\sum_{j=1}^n j^2$, an array of integers from 1 to *n* and two arrays of the positive and negative square roots of those integers in the range 1 to *n*.

In the second form of *for* loop, the reserved word *iter* is used, which acts as a kind of a global *new* to indicate the rebinding of values to loop variables in different iterations. For example, the following piece of code would iterate 100 – *n* times if *n* <= 100.

```
for j := n do
  if j <= 100 then
    j           %exit for loop
  else iter
    j := j+1    %iterate
  enditer
endif
endfor
```

An equivalent sequence, say in Pascal, would be:

```
j := n
while j <= 100 do
  j := j +1;
```

Conditionals

Conditional statement is also supported in VAL:

```
if expression then
  action
elseif expression then
  action
:

else
  action
endif
```


where expression will yield a boolean result and action is a number of statements. The `elseif` and `else` part are optional. The conditional statement can return an arbitrary tuple, but all branches of the conditional must match both in arity and type.

If the predicate of the conditional evaluates to one of the error values, then the whole conditional returns an error. However, if there are a number of boolean expressions forming the predicate, joined by boolean operators, evaluation of the condition ceases as soon as the truth value of the condition is certain. For example, if the first expression in a pair linked by an “and” operator evaluates to false then no further evaluation of the condition is carried out. Although in this case later expressions might be in error, they are ignored.

Functions and Modularity

The definition of a function in VAL is much the same as other languages. It consists of a header showing the values being passed to and returned from a function body. For example, the following function takes two real numbers as input and returns a character.

```
function test(x,y : real returns character);  
    BODY;  
endfun;
```

Functions in VAL are free from *side effects*, using only internal storage. Furthermore recursive function calls and mutually recursive calls are not legal.

Programs written in VAL can be modular, since modular compilation is possible. This allows a programmer to specify the invisibility of certain functions and also allows for the splitting of a program over separate files.

The original proposed language Val had no facility for streams. Furthermore, the loop form using `iter` proved to be difficult in practical use and could not be easily made to accept stream extensions.

2.3.3 SISAL

SISAL (Streams and Iteration in a Single Assignment Language) was designed jointly by Lawrence Livermore National Laboratory, Colorado State University, DEC and Manchester University [ref. manu. 84].

The original motivation for the design of SISAL and the lower level language *IF* were to allow comparison of various parallel processing systems by using a common high level language (SISAL) compiled into a common intermediate format (IF) with a number of machine specific parsers [Allsopp 86]. This approach makes it possible for other languages to be compiled into IF, allowing their use on all the machines.

SISAL is a functional data flow language, designed to express algorithms for execution on computers capable of highly concurrent operations. However the primary targets for translation of SISAL programs are data flow machines. The designers hope that SISAL will evolve into a general purpose language, one suitable for programming future highly parallel computers. SISAL has inherited many of the VAL language constructs. The `let` construct and conditional expressions are exactly the same in both languages. Reference [Chambers 84] gives a good introduction to SISAL.

Types and Data Structures

Having been based on VAL, SISAL has all the basic types and operations of VAL, for example, records, arrays etc. As the name suggests, a further type, *stream*, is also supported. There are a number of operations that can be carried out on

stream types, e.g., create, append, concatenate etc. A stream type consists of two components:

- A range (1,HI) being the inclusive bounds on the defined elements. If HI equals 0 then the stream has no elements.
- A sequence of HI elements of the declared type. Note that the elements of a sequence must be exactly as specified by range.

As with *oneof* in VAL, the type *union* allows a data object to take one of a number of types. Consider the following declaration.

```
union z [n1 :type1; n2 :type2];
```

This specifies that *z* can take either a value of type *typ1* or *typ2*, but not both. User-defined data structures are permitted and can be constructed using the basic types. SISAL is a strongly typed language and type checking is performed by the translator. It ensures that the type of each expression or sub-expression matches the type required by the context in which it appears.

All calculations must produce a result in SISAL even if that result is an error, i.e., *undef*.

Loops

There are two general forms of looping constructs available in SISAL, known as the *product* and the *non-product* form. The product form is a special case of the non-product form. Any loop written using the product form can be rewritten in an equivalent non-product form but not vice-versa. The product form is merely to provide iteration on arrays and streams in a more concise way, while the non-product form is to perform sequential iterations in which one iteration cycle depends on the results of previous cycles.

All loops can return a result tuple of arbitrary type by using *reduction operators*, a generalized form of eval statement from VAL for all loops. There are several reduction operators available ranging from *least*, to specify the return of the smallest value generated in successive loops, to operators altering the order in which the reduction operators affect their operands. For example,

returns value of least $\langle \textit{expression} \rangle$

signifies that the final value returned by *expression* in the loop is the smallest one of them all. Also the *left*, and the *right* tree operators specify the reduction order; for example, if a loop successively returns a value for I from 1 to 5 then:

return value of left sum I

would yield $((((1 + 2) + 3) + 4) + 5)$ and:

returns value of right sum I

yields $(1 + (2 + (3 + (4 + 5))))$ and similarly:

returns value of tree sum I

yields $((1 + 2) + (3 + 4) + (5))$.

Functions and Modularity

Function declaration and invocation is much the same as VAL but with one major difference. VAL does not allow recursive or mutually recursive calls, but on the other hand SISAL allows mutually recursive calls by using a forward definition. The declaration of a function name as forward, before its definition, allows the compiler to gather the type information necessary before fully defining the body of the calling function. A function invocation is itself an expression whose arity and types are the number and types of the values returned by the function.

SISAL allows modular program design and provides function availability over module boundaries by means of *export* and *import* keywords.

One major difference between SISAL and Val is its support for streams. It is possible to return an array or stream using the return constructs `arrays of` and `stream of`. These behave in a manner similar to ID's `return all` clause for loops. They return an array with one element for each iteration of the loop or a stream with at most one element for each iteration. The semantics of streams imply [Sarkar 88] sequential access to the stream elements and allow for non-strict evaluation

2.4 Parallelism Detection Techniques

It has already been shown how parallelism could be explicitly stated by the programmer. Here, attention is focused on available methods for the detection of inherent parallelism in a sequential program. It is worth noting that hardware mechanisms exist to detect possible concurrent execution/loading of instructions at run time, given a sequential stream of instructions [Baer 73]. For example in the IBM 360/91, a hardware mechanism is included for detecting and dispatching floating point instructions to respective functional units. The scheme achieves high level of concurrency based on a tagging process which does not significantly decrease the execution time of the instruction [Tomasulo 67].

There are several different (distinct) processes that the phrase "parallelism-detection" can be applied to. Although all of these processes try to improve the execution time of a program in some way, they can be conceptually very different from each other. For example, the type of parallelism detection a vectorizing compiler carries out is very different to the way a compiler for a data flow machine works. The interest here is mainly on the type of parallelism detection techniques that can be useful for DF-like or processor-array type machines.

Generally speaking, there are two kinds of dependency in a computer program

which hinder parallel execution. *Control dependence*, which is a consequence of the flow of control in a program due to, say, the outcome of a condition test, and *data dependence*, which is a consequence of the flow of data [Wolfe 82].

2.4.1 Parafrase

This is based on the creation of a directed acyclic graph (DAG) to represent a program satisfying the data dependencies seen earlier [Allsopp 86, Kuck 82]. At this point a series of complex processes are carried out to remove as many dependency arcs as possible (restructuring the source code). Finally, several abstract transformations are performed to eliminate cycles in the graph. For example, a flow dependent cycle can be replaced with a recurrence node. Similar operations are performed for cycles of anti and output dependence.

The resulting graph consists of a number of nodes. The nodes in each anti-chain of the DAG can be executed simultaneously. The Parafrase system is designed for Fortran programs and is commercially available. It is particularly powerful for detecting and vectorizing dependencies caused by array indices. Although this system is very powerful it has a number of drawbacks. It is only a restructurer and its output must be compiled by a vectorizing compiler for a particular parallel machine. There are cases where this restructuring reduces performance due to the special features of a compiler and its target machine. Finally Parafrase is mainly concerned with inter loop (Do loop) parallelism and is not particularly useful for data flow applications. The modelling technique, used by Parafrase (DAG), is not capable of representing the dynamic behaviour of a given program and therefore is not suitable for pre-execution (static) analysis of how a program behaves at run time.

2.4.2 Russell's Method

Here again, a directed graph representation of the original program is constructed along with boolean connectivity and precedence matrices [Russel 69]. The feedback arcs of loops are detected and deleted at the time to generate a DAG. By analysing the DAG a parallel equivalent graph and associated boolean matrices are generated, figure 2.8.

Russell's method yields only limited success in detecting parallelism for three reasons [Baer 73]:

- Arrays are treated as one single object, and no subscript analysis is performed.
- There is no attempt to recover hidden parallelism due to variable reuse.
- No attempt is made to replicate loops if possible, where most of the parallelism can be found.

2.4.3 Maekawa's Method

Maekawa [Maekawa 76] proposed a method for detecting the parallelism of computational elements using a generalised form of pipelining. This pipelining method is more flexible than Russell's method but requires queues, thus incurring more overhead. This concept is carried a step further through multi-level pipelining scheme. However, he concludes that this scheme is not effective in small programs due to excessive overhead, even though at a global level the overheads become negligible.

To express multi-level pipelining he introduces the primitive "activate" which is a similar construction to *cobegin* in concurrent Pascal. When an activate primitive is executed by a sender process a message buffer containing all the current values of arguments and the procedure name is placed at the tail of the queue, and the

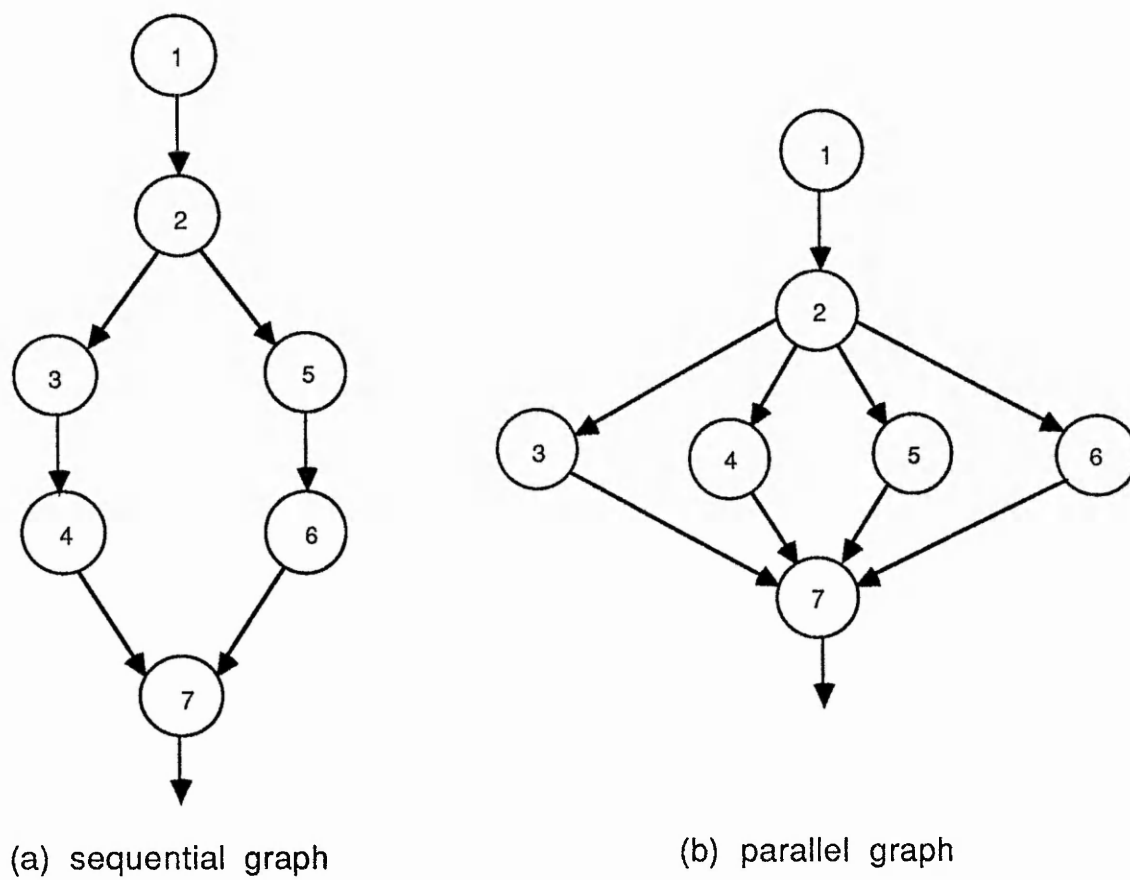


Figure 2.8: A sequential program model and its equivalent parallel model

receiver process is activated. While the queue head is not empty the receiver process continues. To illustrate, consider the following example in Pascal.

```

for i := 1 to n do
  begin
    j := f(i);           {1}
    k(i) := g(j);       {2}
    writeln(k(i));      {3}
  end;

```

Statements 1 and 2 are data dependent on each other and cannot be executed in parallel. However loop replication is possible. The algorithm by Maekawa does not use loop replication but instead recognises that the code above can be decomposed into the equivalent piece of code:

```

for i := 1 to n do
  while queue not empty do
    begin
      j := f(i);
      send(j,i) -----> queue -----> receive(j,i);
      k(i) := g(j);
      writeln(k(i))
    end;

```

The actual algorithm can be found in [Maekawa 76].

2.4.4 Bird's Method

This method uses a topological sort as a vehicle for detecting both intra block and inter block parallelism [Bird 85]. The process is illustrated by way of an example; consider the following piece of code:

```
a :=const;
b := const;
c := a + b;
d := a + const;
```

For each variable or object in the program, two sets of values are computed: the *dependency_count* and *dependent_nodes*, where:

$$\text{dependency_count}(o) = \text{cardinality}(\{o' | o\delta o'\});$$

$$\text{dependent_nodes}(o) = \{o' | o\delta o'\};$$

Note that $(\{o' | o\delta o'\})$ is interpreted as all objects o' that object o is dependent on.

Using the computed values an augmented data dependency graph is constructed. Figure 2.9 shows the graph for the sample program. Each node of the graph has two fields: one containing the *dependency_count* and the other its output. The arcs in the graph represent the sets of *dependent_nodes*.

The topological sort can now be applied to traverse the tree. Suitable candidates for parallel execution are those with zero dependency count. After a node has been released all dependent nodes have their dependency count decremented.

Data dependencies between objects are determined by examining the input set and the output set of an object. All three data dependencies described earlier are examined by this method however, control dependence is not covered.

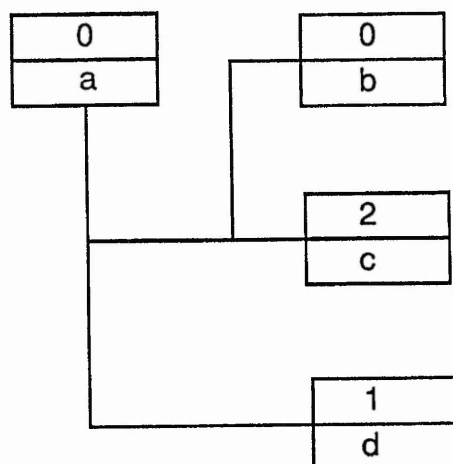


Figure 2.9: An augmented data dependency graph

The method will work if the constructed graph does not have any cycles. As far as intra-block analysis is concerned the computational elements are statements. The statements within a block have a linear order; there cannot be execution cycles within a block. This implies that there is not a data dependency cycle and the graph is a partial order.

For inter-block analysis however, the method only deals with structured flow programs, i.e., those without *goto* statements. There are two other types of control flows that can create data dependency cycles: iterative control structures, e. g. while and for loops; and recursive or mutually recursive function calls. Data dependency cycles in these types of constructs must also be eliminated or their effect on the graph eradicated by encapsulating the cycling portion as a single node before the method works.

2.4.5 Interval Analysis Technique

In his paper [Schneck 72] Schneck puts forward a method for automatic detection of suitable statements in a Fortran program for parallel processing. The output is in the form of a Fortran program which uses extensions to the language for explicitly showing parallelism. Suitable target machines for his technique are listed as ILLIAC-IV, STAR and ASC. In general the method is only suitable for pipeline or vector machines.

The compiler uses "interval analysis" to detect parallelism. It consists of 3 main steps:

- Statement classification

As source statements are read in they are classified as one of 40 Fortran statement types. Appropriate routines for processing each type are then called to transform the statement into an intermediate text representation which is convenient for further analysis. At the same time information regarding program flow is gathered for later use. Each appearance of a variable, constant or label is entered in a "reference table". When all of the statements are read in and processed in this way control is passed to the next phase.

- Flow analysis

Here a model of the program in terms of some "basic blocks" is created. A "basic block" is defined to be a section of code with only one point of entry and one point of exit and no internal flow. Having identified the basic blocks they are then grouped into "intervals". An interval is the maximal set of basic blocks with the property that all the basic blocks within the interval, except for the head, have their predecessors in the interval. The resulting intervals which now have some partial ordering are treated as basic blocks and the process continues until no further intervals could be made.

- Recognition of parallel or vector operations

By examining the variables and constants within each interval an attempt is made to vectorize operations within the interval

One of the major problems with this method is that any calculation which is conditional, and thus loop dependent, cannot be performed in parallel. This would in practice eliminate a great many operations from parallel processing.

2.4.6 The Paralyzer

The work presented here [Presberg 75] is specific to the ILLIAC-IV array processor. The vectorizing process is only applicable to DO loop constructs and can be viewed as a an optimisation step specific to this processor.

The technique used is based on the coordinate and Hyperplane methods. In general terms, the coordinate method determines parallelism which can be expressed in terms of the original DO loops and their index values. The sequential loop data dependencies are preserved by rearranging the loop body statements so that flow of control from statement to statement in the parallel loop compensates for the fact that each statement is executed for all index values before its successor is executed.

The Hyperplane method is applicable only to nests of two or more DO statements. It determines parallel execution which is expressible in a more drastic reordering of the loop index than that given by the coordinate method.

The actions of the technique consists of three distinct parts:

- Setup:

It amounts to cleaning-up of an arbitrary nests of DO loops to create an analysable form.

- Analysis:

This is based on examining the form of array elements within loops. Time

separation values for pairs of access to same memory locations are evaluated. The crucial computation structure to be preserved in the rewriting is the data dependencies among the array references. Thus, an important time relationship exist between generation and use references. This is given by the time separation values which are computed for every array modified by the loop.

- Synthesis:

Once the time separation sets are constructed one of the rewriting methods is employed by the Paralyzer; Coordinate or the Hyperplane method.

The major setbacks of this method are due firstly to its sole concentration on loops and secondly because it is very specific to the target architecture.

2.4.7 Parallelism Detection for Applicative Languages

In this paper [Arvind 80] it is argued that Fortran like languages, with or without parallel extensions, are inadequate to utilise high performance parallel architectures. It is claimed that efficient multiple processor architectures cannot be developed unless it supports the execution of parallel processing language systematically. A method for automatic decomposition of a program written in an applicative language is then presented.

The scheme is useful for applicative programs which have loops as their primary control structure. The operations on bounded-size data structures is shown to be decomposable in 3 steps:

- The nested loop structures are unrolled into a network of computation cells.
- Data structure elements are assigned to the cells.
- And finally the network of computation cells is mapped onto the actual processors of the system.

A computation cell is regarded as a virtual processor to which a program and local data has been assigned. However, the virtual processor program may also refer to data which is not local, in which case a communication between the virtual processors needing and holding the data takes place.

2.4.8 Parallelisation of CALL statements

The research in this area is more concerned to establish the effect of procedure calls on data-dependencies with the view to increase parallelism at the statement and loop levels. Two different sets of work are discussed here.

In a paper [Triolet 86] Triolet presents a method for determining and analysing the effect of a CALL statement. It has four main steps:

- Statement Effect Computation (SEC) which is the effect of computing statements within a procedure, including the effect of any further procedure calls.
- Procedure Effect Computation (PEC) which is the global effect of a procedure call.
- Dependence Graph Computation (DGC) which can be deduced as a result of computing SEC.
- Restructuring and Parallelism Detection (RPD).

The end result of applying this technique to programs is to transform CALL statements to asynchronous CALLs. They call this "Direct Parallelization" as opposed to parallelization of procedures themselves. The DO loops may be transformed into DOALL or DOACROSS as a direct result of applying this technique.

From the result of experiments that they carried out it is concluded that a speed up of around 60% is achievable in certain cases where DO loops contain several CALL

statements. In the prototype model tests not involving `COMPLEX` parameters were considered only.

The method suggested by Burke [Burke 86] is different with that of Triolet in two different ways:

Firstly, this method distinguishes between `CALL` graphs which have static and dynamic cycles. Like the earlier method it rejects graphs with dynamic cycles (recursive `CALLs`) but it does accept static cycles. Secondly, the method is capable of taking into account the effect of variable aliasing.

2.5 Symbol Table Manipulation

The remainder of this chapter describes existing techniques for symbol table construction, storage and access. It also evaluates each method with reference to its speed of access and storage, memory space consumption and finally ease of construction. Chapter 3 presents a new method for construction and manipulation of symbol tables which reduces access time and memory space requirements. Reference [Al-Dabass 86] describes existing methods in a comprehensive way.

2.5.1 Use of Hash Tables

This method is based on a doubly linked list of tables with the use of hashing for indexing. Figure 2.10 shows the topology of a symbol table for a typical program.

At the beginning of compilation a main table with a fixed size is created. The size is based on the maximum number of variables which is envisaged to be declared in a program. The table has three fields: type of variable, an information field which had a tree structure and finally a pointer field.

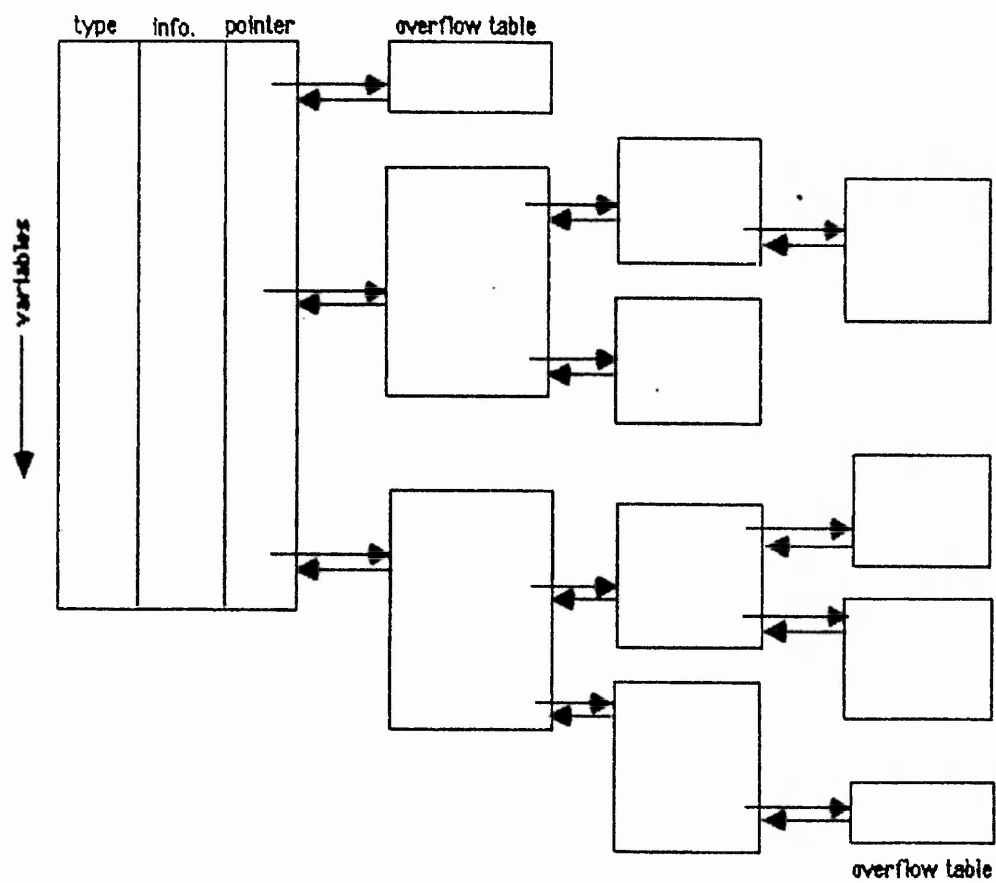


Figure 2.10: A hashed symbol table organisation.

The following operations can be performed using a hash-table:

- *Storage:*

By means of a hashing function, a hash-number is generated when a variable declaration is encountered. The table is indexed by this number to store the variable definition in the table. Before a variable is stored in the table a check is made to ensure that its hash-number was not used previously (collision).

If there is a collision (the entry in the table is already used) then the pointer field would point to another table which is called the overflow table. The new table will hold the information about this new variable if there is no further collisions. In the event of additional collisions the process of using overflow tables is continued.

In order to reduce the number of collisions the table size must be significantly (about 50%) bigger than the actual size thought needed. This would cause a significant rise in the memory space requirement of a symbol table.

As functions and procedures are defined a new table is created for each, containing the local variables and their types. A pointer from parent links, the entry point of the child's name in the parent table, to child's table. This is necessary to keep track of variable scoping.

- *Access:*

To examine the information about a particular variable the table has to be searched. Again the search is carried out by generating a hash-number to index the table. In cases where a collision occurred during storage overflow tables may need to be searched for extraction of the relevant information.

When a variable referenced in the body of a procedure is being examined then the current table is searched. If the variable is not found then the search process continues in the table of the parent, until either the variable is found in one of the parent tables or it reaches the stage that there is no more parent table to be searched.

The information field is usually a tree structure, containing various data depending on the requirement. A typical information field is shown in figure 2.11. This is a compromise between the hash table method and the binary tree method explained below.

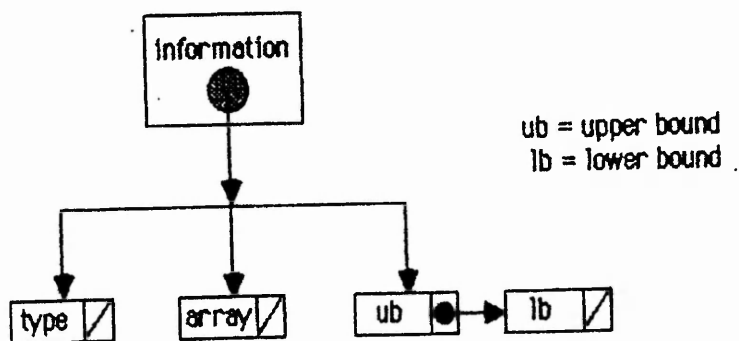


Figure 2.11: A typical information field of a symbol-table.

2.5.2 Binary Tree

This is an alternative data structure for representing symbol tables. Each node of the structure represents a variable and points to an information field. A node also has a left and right pointer which points to other nodes. This ensures that the

constructed tree is a balanced binary tree. This technique offers less performance compared to hash table method, in most cases, but provides less implementation effort and uses less memory space.

Chapter 3

A Hybrid Compiler with Automatic Parallelism Detection

Software written in procedural languages has to be redesigned for parallel computers and rewritten in modern parallel languages which make efficient use of these machines. In the short to medium term, it would be cost effective to utilise existing software but by means of a parallelizing compiler which would generate code for parallel processing systems.

3.1 Net Construction

A program or system may be designed by using a modelling system like Petri nets, specially if the system is likely to exhibit concurrency or the program is to be executed on a parallel machine. This design methodology allows the user to analyse the system before coding, and devise a parallel algorithm for the application. Having generated a Petri net model of the system, the user can code it in an appropriate language. Such a model enables a user to examine the degree of parallelism available in the program at any given time during its execution.

The proposed method is a reversal of this process, i.e., given a piece of source code, the compiler models it by a Petri net and then extracts the parallel paths inherent in the program by executing the net. The model is constructed subject to the following constraints:

1. Any procedure call is treated as one instruction and is thus represented by one transition in the model. A call to the procedure is made on all the nodes that have been allocated to execute a part of the procedure body.
2. Any conditional statement is treated as a block, represented by one transition.
3. Any loop construct is initially treated as one block; loop fission takes place later on during allocation. Only loops without transfers of control in their bodies are analysed for parallelization and decomposition.

3.1.1 Definitions

- **Definition 1: Petri net**

A Petri net is an abstract, formal method of modelling information and control flow (for a full formal treatment see appendix A). It is a powerful way of modelling and analysing systems particularly those that may exhibit asynchronous and concurrent activities. A Petri net graph models the static properties of a system, in much the same way as a flowchart represents the static properties of a computer program. In addition, a Petri net model can represent the dynamic characteristics of a system as a result of its execution.

The graph is made of two types of node circles, called places, and bars, called transitions. These nodes, places and transitions, are connected together by directed arcs, from places to transitions and from transitions to places.

Figure 3.1 represents the Petri net model of the program shown in figure 3.2, and its abstract form stored in a computer for clarification.

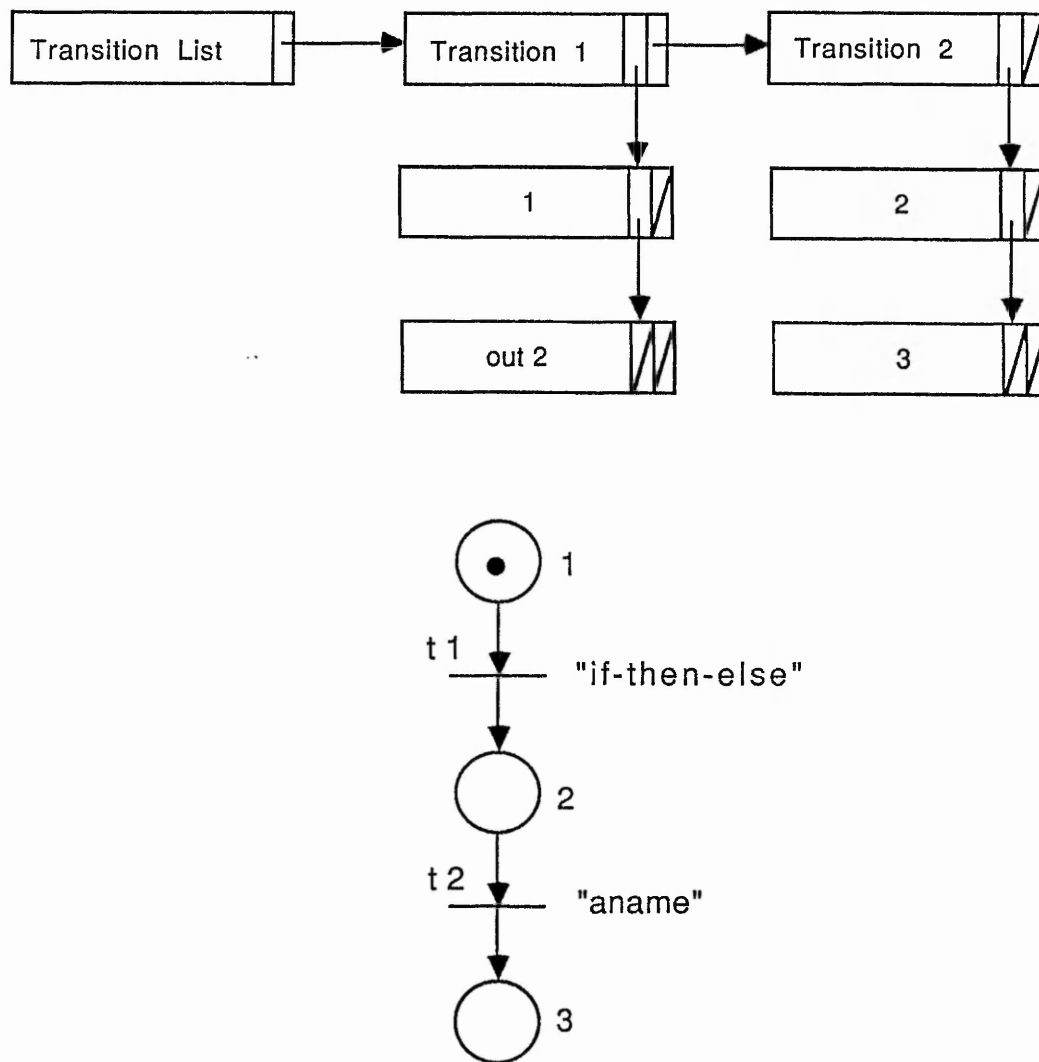


Figure 3.1: A sample Petri net model and its internal representation

As can be seen from the diagram the input to transition 2 (T2) is 2 which corresponds to output of T1. A transition can have any number of input places but only one output place. Statements or blocks that assign to at least one variable always have an output place, but blocks with an empty output variable set do not have an output place. For example, statements like "writeln" never have an output place. In cases where the input variable set is empty for the block then its input place will be 1, which indicates that it is not dependent on any other statement. For example, a statement that outputs a message is not data dependent on any other statement. However if there are a number of such statements, then in order for the message to make sense, there must be a flow dependence between them.

- **Definition 2: input and output variable sets**

If s is a statement or a block of statements then $IN(s)$ will denote the set of input variables of s . Usually, input variables are those to which references are made but whose contents are not modified ¹ (read only).

For an assignment statement:

$$IN(s) = \text{variables that appear on the right hand side}$$

And for a procedure call statement:

$$IN(s) = \text{all the arguments to the procedure and all the global variables to the procedure referenced in the procedure body}$$

And finally for a conditional statement:

$$IN(s) = (\text{variables used in the control part}) \cup (\text{variables referenced in the "then" part}) \cup (\text{variables referenced in the "else" part})$$

¹The exception to this rule is when a statement like $a := a + 3$; has variable a both as input and output.

Similarly the output variables of a statement or block of statement are denoted by $OUT(s)$; where $OUT(s)$ represents all the variables that have been changed (written to). For example, the output variable of an "if" statement can be written as:

$$OUT(s) = (\text{output variables of the "then" part}) \cup (\text{output variables of the "else" part})$$

This is a conservative output variable set for a conditional statement due to the uncertainty of the run time outcome. Figure 3.2 shows a sample program and its statements' input and output variable sets.

- **Definition 3 : data dependence relations**

Given two statements s_j and s_k in this order then the following data dependence relations should hold true between s_j and s_k .

If $X \in OUT(s_j)$ and $X \in IN(s_k)$ then s_k is data flow dependent on s_j denoted by $s_j \delta s_k$.

If, however, $X \in IN(s_j)$ and $X \in OUT(s_k)$ then s_k is data anti-dependence on s_j denoted by $s_j \bar{\delta} s_k$.

Finally, if $X \in OUT(s_j)$ and $X \in OUT(s_k)$ then s_k is data output dependent on s_j denoted by $s_j \delta^o s_k$.

- **Definition 4 : loop fission**

If a loop body consists of a number of independent components or blocks then it is possible to construct a number of independent or communicating loops which can be executed in parallel with each other. The combined effect of these independent or communicating loops is exactly equivalent to the original loop. This action is named loop fission.

Data Flow Dependence

The procedure to construct a Petri net for a source program is developed as follows:

```
program test;

  var i,j,k,l : integer;

  procedure name (var a : integer; b : integer);
  begin
    l := a + b + j;
  end;

begin
  if i < k then i := j else j := l;      1
  aname (i ,k)                            2
end.
```

IN (1) = (i, j, k, l)

IN (2) = (i, j, k)

OUT (1) = (i, j)

OUT (2) = (l, i)

Figure 3.2: A sample program and its input and output sets

1. The syntax tree is first transformed into a suitable form to group all logically related statements together. The tree can then be viewed as a linked list whose elements are the successive statements of the program. Some of these elements can be a collection of statements that form a block. A block can be an entire "for" loop, an "if-then-else" statement or just a procedure call. At this stage no attempt is made to model the data dependencies within a block and therefore no block decomposition is made.
2. This is a top down approach where coarser grain parallelisms are detected first and while allocation of this coarse grain parallelism is being carried out, an attempt is made to detect finer grain parallelism and perform such operations as loop fission if possible. For example, while a block made up of a "for" loop is being allocated to a certain processor, the body of the loop can be modelled by a net and analysed for possible spread over a number of processors, with the communication codes added by another part of the compiler.
3. The fundamental net is cycle free, since all constructs leading to a cycle in the net, e. g. loops, are treated as single blocks. A fundamental net is the net model of the main program body. This approach eradicates the effect of cycles on the net and avoids lengthy procedures needed to overcome their side effect.
4. The process starts with the first block in the main program body and is repeated for every block until every one of them is represented by a transition of the Petri net.
5. The net is represented internally by a linked list whose elements are linked lists too. Successive elements of this list give the net attributes of successive blocks in the program, starting from block one and ending with the last block. Each element of this structure represents a transition in the net providing a list of its input places and its output place.

6. Since all independent transitions have 1 as their input place then theoretically they can all be executed at the same time. To preserve flow dependence, the order of transitions in the net is taken into account by the allocator. Thus all I/O statements, irrespective of their data dependencies, are allocated to node 1 and flow dependence is strictly enforced.
7. In situations where an input variable to a block has not been assigned to a previously uninitialised variable, then the input place to this transition will be 0. When execution takes place this causes the particular transition to be identified as unexecutable, and a warning message is output by the compiler.
8. To construct the net, start with block 1 and extract the input variable set of the block. For each variable in the set, traverse back the syntax tree keeping a cross reference to the transition list built so far. While traversing each node of the tree check if the variable in hand exists in the output variable set of the node. If it does then the number associated with the output place of the current node is assigned to the input place set of the current block. If not, continue traversing until reaching the root, at which point 0 is assigned to the input place set of the current block. Blocks with no input variable set are connected to place 1.

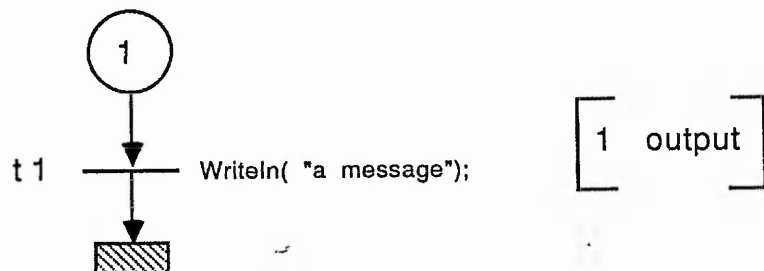
Example 1

```

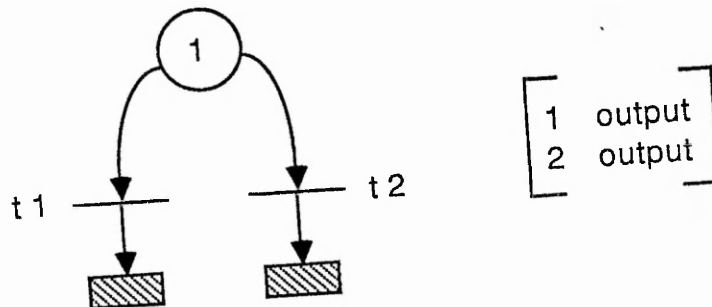
writeln ("this is a message");      {1}
writeln ("input A and B");         {2}
readln (a, b);                     {3}
d := a + b;                         {4}
c := a * b;                         {5}

```

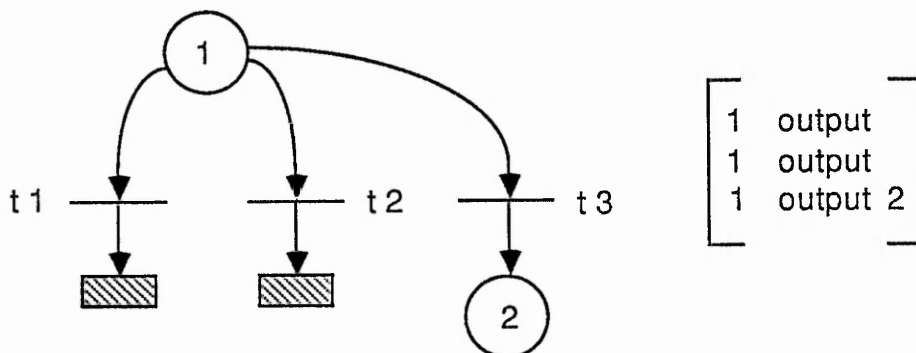
Starting from block 1, there is no input variable set or output variable set for block 1 therefore a transition is created with input place 1 and no output place:



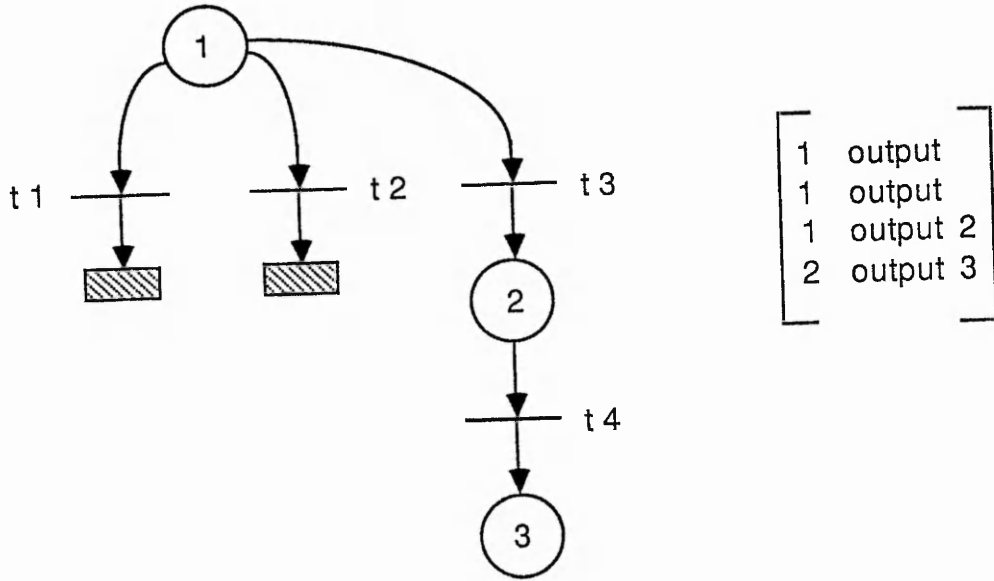
Similarly for block 2 to give:



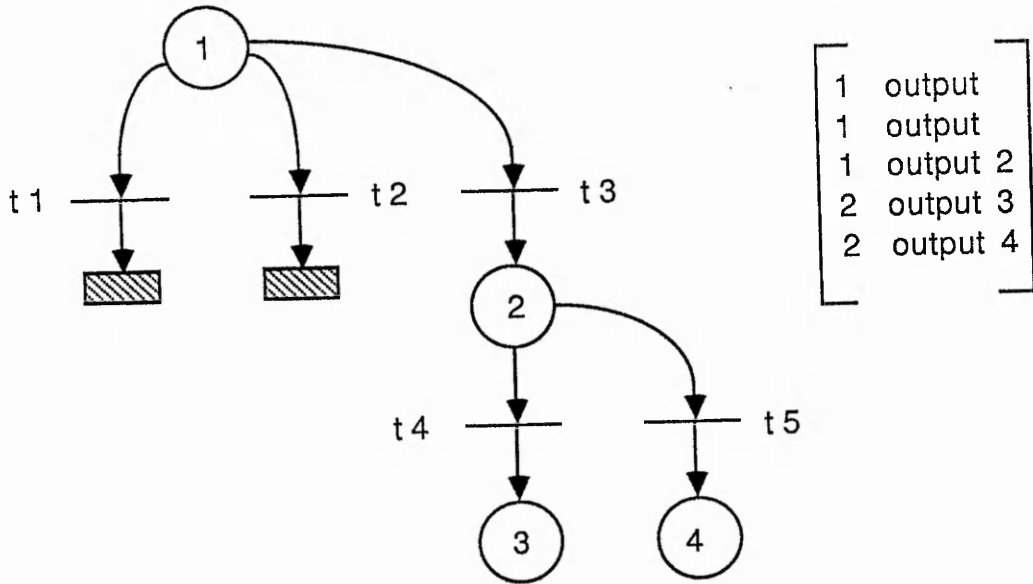
For block 3, there will be an output place but there is no input variable set, therefore the input to its transition is place 1.



Block 4 has an input variable set, consisting of variables a and b, both of which form the output variable set of block 3 whose output place is 2, hence:



Finally adding the net for block 5 gives:



Anti and Output Dependencies

Further tests need to be incorporated in the construction. The necessity for these tests is derived as follows :

1. Consider the following example where s1 and sn are program statements:

```
s1      a := 5;
```

:

```
sn      a := 4;
```

sn is data output dependent on s1, and therefore theoretically these two statements can not be executed in parallel. If a statement between s1 and sn uses the value of variable "a" then sn must wait until the reference by that statement is complete, hence there must be a test to check if the output variable set of sn appears in the input variable set of any block between s1 and sn, and to create the necessary arcs. If however, the value of "a" in s1 is not referenced anywhere in between, then the arc between s1 and sn need not be created. To get round this problem many compilers rename the variable "a" in sn and every use of it after sn until a new assignment to "a" takes place. This means that the source code is internally changed, by those compilers, and different memory locations are used to store the different assignments to the same variable. This is avoided in the proposed compiler.

2. To maximise the number of parallel paths, the above is necessary in architectures where there is only the possibility of storing a variable in a unique memory location accessible by all nodes (global memory architecture), or in distributed memory systems where any node can directly access any memory segment via a switching network.

In data flow systems or message passing machines based on associative memory addressing, different values of the same variable (old and new) can be kept in different cells or local memory locations situated on different nodes. In the latter case during allocation, if two potentially parallel statements store values in the same memory location then they must be allocated to different nodes, providing there are no other detrimental side effects to the efficiency. Hence in the above example the two statements may be executed on different nodes and therefore each assignment to variable "a" may, as a result, be stored in the memory location reserved for that variable on the relevant processing nodes of the machine. The compiler does not necessarily allocate two

parallel statements on different nodes and other considerations will be taken into account, as will be seen later.

This allows use of the maximum number of parallel paths without renaming variables and without any need to consider data output and data anti dependencies. The method also lets the net model the data flow characteristics of a program and hence be easily converted into a real physical representation of a program stored in a computer memory (cell), executable by a true data flow machine.

3. Consider the following piece of code :

```
s1      a := 5;
s2      b := 2;
s3      c := a + 2;
s4      a := a - b;
s5      b := b - 2;
s6      a := 6 + c;
```

The usual way of representing the above example is shown in figure 3.3. This represents a *loosely connected net* which does not enforce the output dependencies present in the original code.

However, figure 3.4 represents a *strongly connected net* for the same example. In this case all dependencies have been taken into account.

Up to transition 3 the net is fairly straightforward to construct. Transition 4 models statement 4 which has {a, b} as its input variable set. In the first instance it looks as though it is dependent on statements s1 and s2. But s4 is assigning a value to variable "a" as well as referencing it, and since "a" is last referenced in s3 therefore s4 is also dependent on s3. But s3 itself is dependent on s1 therefore the link between s1 and s4 is superfluous. Transition 5 is also simple but for transition 6 note that the value of "a" is both referenced and assigned to in transition s4 and hence the link is necessary. This is central to

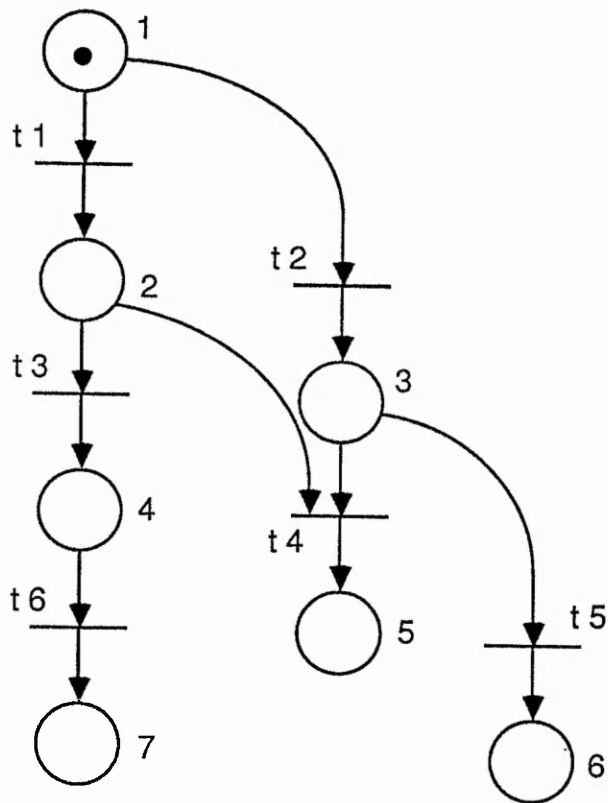


Figure 3.3: A loosely connected Petri net model

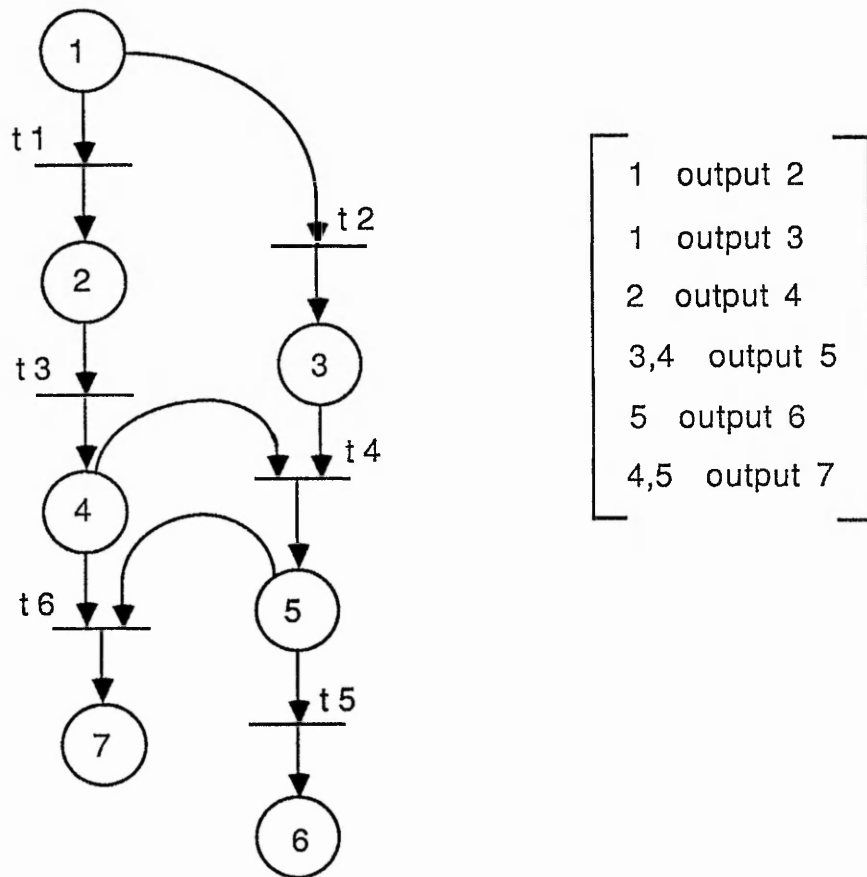


Figure 3.4: A strongly connected Petri net model

keeping all the dependency information without having too many arcs. The flow dependence is particularly important during allocation for non-data flow architectures.

Note that during allocation s_1 and s_6 may not be on the same node and all statements after s_6 will receive the value of "a" from the node holding s_6 , if they require it. If s_6 happens to be allocated on the same node as s_1 then strict flow dependence will be enforced as will be seen in the section describing the functionality of the allocator in this compiler.

4. Consider a simple example:

```
s1      a := 1;
:
sn      a := 2;
```

A problem arises if an instruction s_x , located between s_1 and s_n , references variable "a" and is allocated to the same processor that executes s_n . In this case if the value of variable "a" is communicated to that node, from the node that executes s_1 , then the local value of "a" will be lost ². Two options are available. One is to make sure that during allocation two numbers, a transition number and a processor number associated with each transition set, are stored in the symbol table. Thus in this example "a" will have two sets of numbers in the symbol table associated with it, say {1, 1} and {6, 2}. These numbers show that "a" is represented by two transitions, 1 and 6, at the same time placed on nodes 1 and 2 of the system for execution. From this point if a transition referencing "a" is to be allocated then its transition number is checked against 1 and 6. If it is less than 6 then the transition must not be allocated to processor 2. Therefore, s_3 cannot be allocated to processor

²Parallelism is organised in layers, therefore, in the first instance, statements s_1 and s_n are allocated to two nodes as they can be executed in parallel. In the next layer the allocator may decide to allocate s_x to the same node that executes s_n , thus communication of the value of "a" from the other processor (executing s_1) will overwrite the local value of "a", generated by executing s_n .

2 since its transition number is 3. As soon as a new set of assignments to "a" occurs then the entry in the symbol table changes. This will be referred to as *invocation X reference set*.

The second option is to create a strongly connected net, outlined in 3, and enforce all the dependencies depicted in the model with the resultant loss of some parallelism. In this case S_x will be allocated to the same node that executes s_1 and the allocation process will be a simple mapping of the net to the hardware system.

In cases where the former scheme causes deadlock in allocation then the more conservative method just seen is adapted for deadlock prevention. In systems with a very large number of nodes the possibility of deadlock in allocation is very low and is resolved by the alternative scheme in any case.

Loop Bodies

If the block being scheduled is a loop construct then the loop body is further analysed through its Petri net model. The analysis can result in three distinct outcomes.

However, before considering the result of analysis a further dependence test must be examined. This is required in modelling a loop body or any reentrant piece of code.

Forward Data Dependence: This may exist between blocks in the body of a loop. These dependencies are as a result of different iterations of a loop body. If loop fission is to take place then these tests are necessary to enable the insertion of the correct communication code. Consider a simple loop body where i is a loop index:

```
s1      a := b + a;
s2      b := Rand (i);
```

By inspection it can be deduced that the two statements are independent of each other if they were not part of a loop body. However, s1 is dependent on s2 in the second invocation of the loop body. To account for this type of dependency while modelling loop bodies, the input variable set of any block in the body is tested for possible forward assignment to it. The net representing the above example is shown in figure 3.5

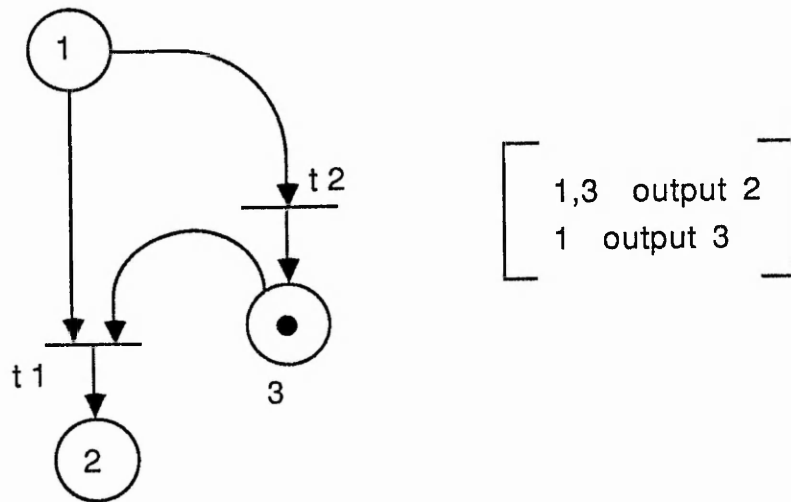


Figure 3.5: A loop with forward dependence

One of the inputs to transition t1 is place 3, which numerically is greater than its own output place, representing a forward data dependence.

Since there is no data dependence, except forward dependency, between the two statements in the loop body then it is possible to allocate each block of the loop to a different processor. To apply loop fission here, it is necessary that the value of variable "b" is sent to other relevant nodes before execution of statement 2 can begin. This would create an extra communication which may not be required if the node executing s1 already has the first value of variable "b" in its local memory. The code with communication instruction for the above example would appear as;

Processor 1	processor 2
label :	label :
receive b	send b
a := b + a;	b := rand (i);
goto label	goto label

Loop Analysis Outcome: The three possible outcomes of the loop body analysis can now be discussed.

1. The first outcome is when nothing can be done to exploit any parallelism in the loop body due to dependencies or existence of transfer of control instructions in the loop.
2. When in an iterative loop, e.g., “for” loop, there are no forward dependencies in the body of the loop then it is possible to spread the iteration space over a number of processors. This is called vertical parallelism. For example, the following loop can be spread over 10 processors, with each processor executing one iteration of the loop. However, the compiler needs to know the bounds of the loop at compile time.

```

for i := 1 to 10 do
  begin
    a := i ** 2;
    b := i ** 0.5;
    b := a - b;
  end;

```

3. The third possibility is when a loop contains some data dependencies and cycles due to forward dependencies. In these cases the loop body can be spread over a maximum number of processors. This maximum number is the

number of parallel paths found in the loop body due to usual dependencies, with communication codes inserted by the compiler to take care of forward dependencies. Consider the following piece of code in a loop body: whose Petri net representation is shown in figure 3.6.

```
s1      a := a + 1;
s2      b := b - 1;
s3      d := a * b;
```

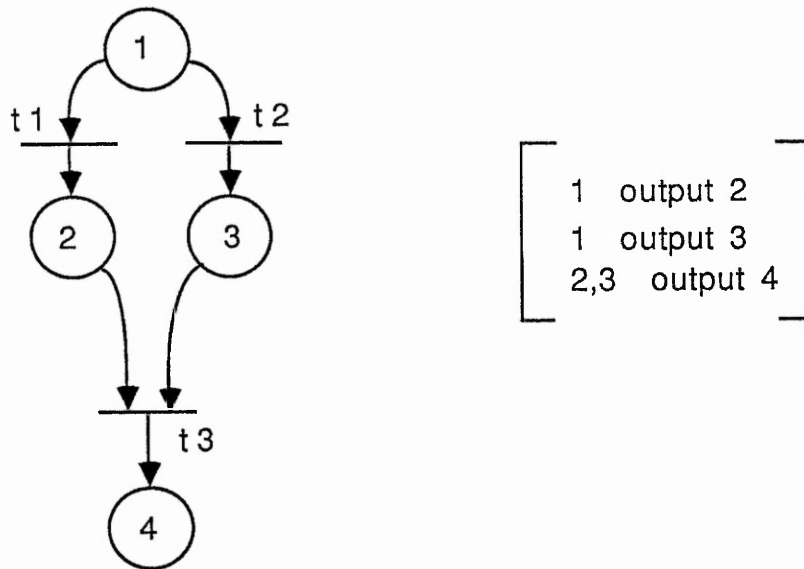


Figure 3.6: Model of a loop without forward dependence

Note that s1 and s2 do not create any forward data dependencies as they refer and assign to variables “a” and “b” respectively. If, however, later on in the loop body variable “a” is assigned to again then a forward dependency will exist if the two statements are allocated to different processing elements.

In this example there is a maximum of two parallel paths and therefore the

loop can be executed by two processors. Statement s3 requires a data communication. Loop fission or splitting could therefore be applied to this example. The same principle can be applied to the modelling and parallelizing of procedure bodies.

The code, including the communication code, for the above net is:

Processor 1	Processor 2
label:	label:
a := a + 1	b := b - 1
receive b	send b
d := a * b	
goto label	goto label

This type of parallelism is termed horizontal parallelism. Vertical parallelism is more desirable as it has no communications between iterations.

3.2 Net Execution for Parallelism Detection

The rules for executing a Petri net are fully covered in appendix A. The procedure for executing a net is fairly straightforward, once a correct net is constructed and the allocation strategy is clear.

Execution begins with assigning a token to the place 1. It is followed by scanning the net structure and disconnecting the arcs that connect any transition to place 1. That is, the input place set of each transition is scanned and if it contains number "1" then it gets deleted. Some of the transitions (at least one) must have an empty input place set. All such transitions are collected together as the first set of parallel independent blocks. This is equivalent to firing all those transitions with only place

1 as their input instantaneously at the same time. The transitions that have just been fired are deleted from the transition list.

These firings must put tokens in other places which in turn enable the firing of some more transitions. From a practical point of view, if any of the transitions just fired has an output place, then these places now have tokens in them and thus a recursive call of the executor procedure and scan of the new transition list can extract the second and subsequent levels of parallelism in the net, provided all the places with a token are identified. Thus this repeated scan must consist of finding and deleting all the input place sets that have any number belonging to the output place set that just have received tokens. It then continues with deleting new set of transitions that have empty input place sets, yielding more parallel paths or in the worst case a single transition for firing (sequential).

The process continues until such time that either the transition list becomes empty, indicating a successful firing of all transitions, or a situation where no more transitions can be fired, for example, due to being connected to place 0 indicating a non initialised variable being referenced, which prompts the user with an appropriate warning message. Another example is when a conditional transfer of control renders some piece of code unexecutable at all times. This leads to *dead code collection*. Again the user must be prompted that some dead portion of code is being eliminated.

A complication that may arise in the execution phase is when a forward data dependency exists in the piece of code being analysed. Depending on the efficiency and speed of the data communication system being used, two approaches can be adopted.

3.2.1 Eliminating Forward Data Dependencies

This approach removes forward data dependencies discovered in a loop body by using loop fission and insertion of extra communication code. This method is especially useful if vertical parallelism can be applied to a large portion of a loop body, however, it must be noted that this scheme generates more communication overhead and can only be evaluated for individual applications.

The main feature of this method is that if the transition creating the forward dependence is allocated to a different node, compared to preceding statements in the loop body, then the output variable set of that transition must be communicated *prior* to its execution and not *after*. Consider the following piece of code representing the body of a loop where i is the loop index. Figure 3.7 shows the Petri net model of the loop below.

```
s1      c := c + 1;
s2      a := a + 1;
s3      b := b + 1 + c;
s4      b := i * 2 + a;
s5      d := c ** a;
```

Assuming two processing nodes in the system, the following piece of code is generated which includes the communication code.

Processor 1	Processor 2
label:	label:
c := c + 1;	a := a + 1;
receive b	send b;
b := b + 1 + c;	send a;
receive a	b := i * 2 + a;

```

d := c ** a;
goto label          goto label

```

Note that on processor 2, "a" is computed and then communicated but "b" is communicated and then computed.

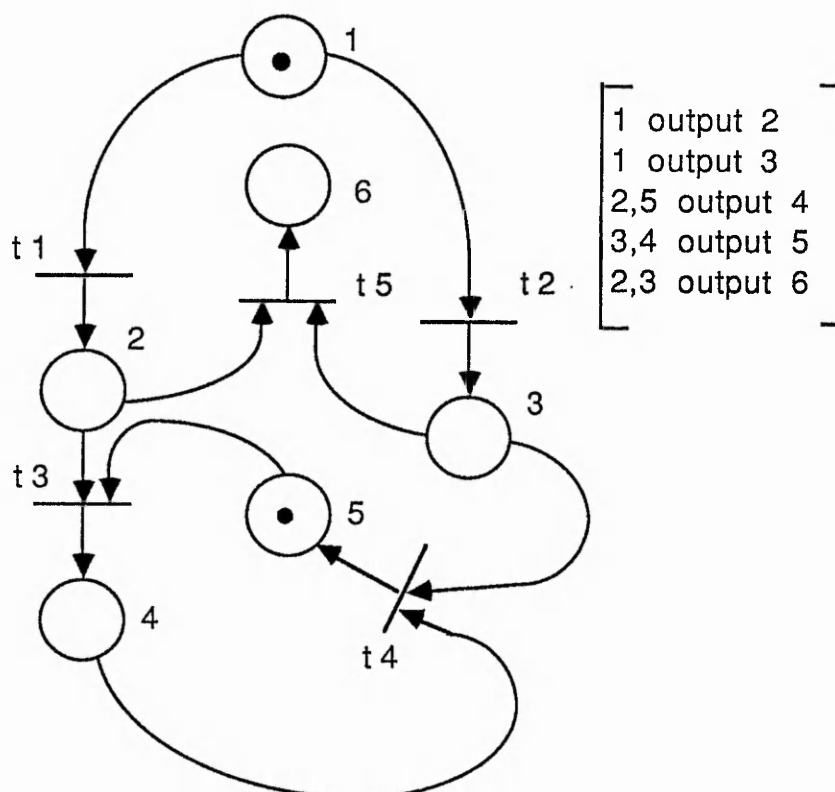


Figure 3.7: A Petri net model showing forward dependence elimination in a loop

To implement this approach the above net execution procedure can be used with some extensions. With reference to figure 3.7, it must be noted that the appearance of digit 5 in the third row of the internal representation of the net (in transition 4) does not constitute an arc. It is there simply to enforce the communication protocol just seen. The original list of places containing a token is thus places 1 and 5. This is to reflect all the forward data dependencies in the loop body. The net execution

procedure thus changes to:

```

for all places in the list of places with token do
  for all the transitions in the net do
    if the input place list includes place with token
      then remove it.
    if the place with token is greater than the output place
      of this transition
    then mark the transition corresponding to a place with
      a token for communication before execution.
  endfor
endfor
allocate all transitions with empty input place set
update the new list of places with tokens
recurse until no more transitions can be executed

```

If there are other blocks in the loop body that require the value of "b" after the execution of the fourth statement, then a further communication is needed after its execution.

3.2.2 Introducing False Data Dependence

The other approach, adopted here, is to force the forward dependent parts of any loop to be allocated on the same processing node. This has the advantage of reducing communication overheads, specially if there are both data and forward dependence between the same blocks in a loop body. Effectively an artificial data dependency is created between the two blocks, which can be viewed as a *critical flow dependence*. For the above example therefore a critical flow dependence can be created between s3 and s4 and the resulting net is shown in figure 3.8. The subsequent execution of the net identifies the parallel paths, while the scheduler,

seen later, ensures that s3 and s4 are on the same processing node because of their forward data dependencies. Hence the following allocation of the loop body takes place which includes a different communication code compared to the last approach.

processor 1	processor 2
label:	label:
c := c + 1;	a := a + 1;
send c	send a
b := b + 1 + c;	receive c
receive a	d := c ** a;
b := i * 2 + a;	
goto label	goto label

If several parallel blocks are forward dependent on another block then their parallelism is ignored and all blocks are allocated to the same processing node that contains the block creating the forward dependence.

3.3 A Hybrid Scheduler

Parallel programming consists of three separate activities; two of them, identification of parallelism and partitioning of the program into layers of parallel paths, have been dealt with. The third activity is concerned with the *scheduling* or placement of the code making up each layer on the multiprocessor system.

Thus, the scheduler or *allocator* is responsible for assigning each block of code to the most suitable processor for subsequent processing. This implies that for any change in the number of processors in the system the program must be recompiled in order to carry out a re-allocation by the scheduler. This action is analogous to the

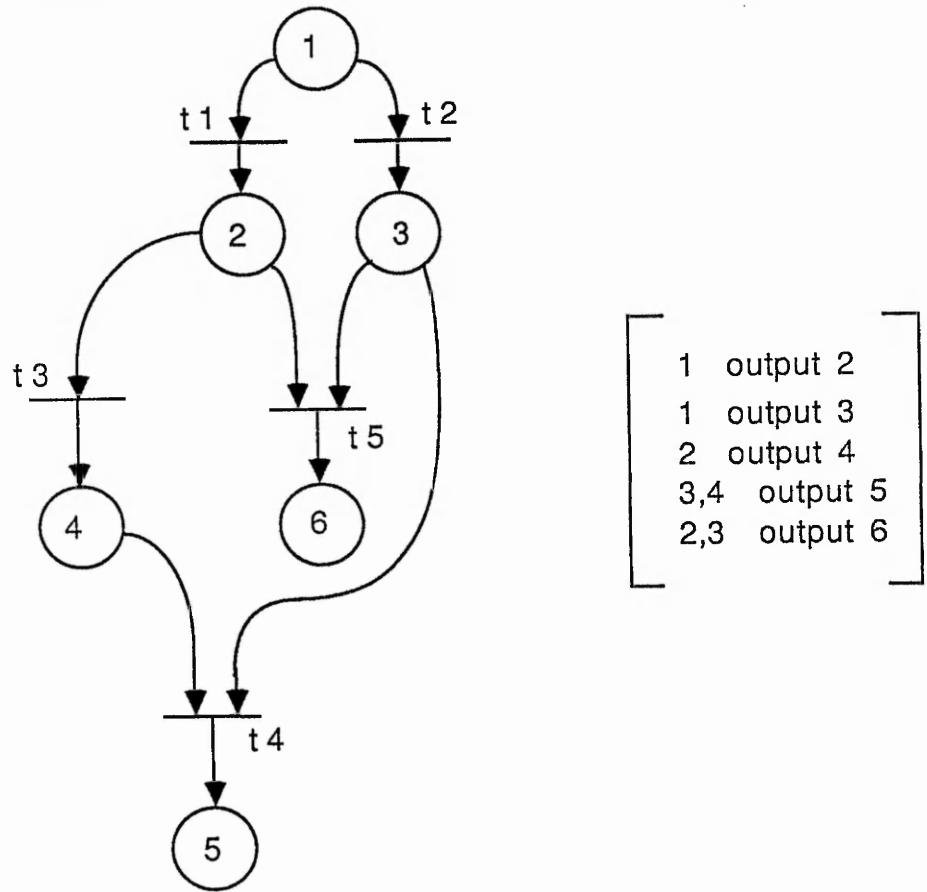


Figure 3.8: A Petri net model showing a loop with forward data dependence in its body

configuration process carried out in systems like a transputer network where there is a notion of connectivity or topology of connection. The allocation is carried out statically and does not change at run time. The terms “scheduler” and “allocator” are used interchangeably throughout.

By executing the net, a new representation of the source program is generated. This new representation is a *multi-layered code*, with each layer containing a number of parallel independent blocks of code. The original sequentiality is lost and the structure does not resemble the old program. The layers must however be executed sequentially.

Although within each layer there is no need for any synchronisation, the transfer from one layer to another requires synchronisation and a number of data communications: *inter-layer communication*. The synchronisation is achieved purely by way of data flow between the processing nodes. A layer would have at least one statement; in this case there is no parallelism in that layer.

Associated with each block is a *send* and *receive* list. At this stage send and receive lists are merely the names of variables that each block has to send or receive and some cross reference information between sends and receives. A typical list at this stage may look like;

```
[ [receive 10 var B integer var c real]
  [d := [b + c]]
  [send 16,18] ]
```

Receive 10 indicates that this is the receive list of statement 10 which receives a value for integer b and one for real c. The send list shows the destination of variable d, that is block 16 and 18. This information is only used by the allocator since the sending node simply puts a value on the bus and does not care about its destination.

To summarize, scheduling is concerned with minimizing the communication over-

head without overloading any one of the processing nodes in the system. Parallelism detection and the subsequent scheduling are both based on static compile time analysis of the Petri net representation of the original sequential code.

3.3.1 Scheduling for MINNIE

This is a much more complex task, compared to scheduling for purely data flow machines, as a number of issues must be taken into account. The first point to be considered is due to the way that a Petri net is constructed by this compiler. It is important to remember that two statements which assign to the same variable can be executed in parallel, that is, they are assigned to two different processing nodes *provided* that all the intermediate statements referencing the variable are not assigned to the same processing node as the last of the two statements was.

Consider the case when in a particular layer there are more such statements than processors in the system. It is evident that the specified condition cannot be satisfied in this case. Therefore, the original net, although correct, would not be suitable for mapping into a particular hardware system. Also it may be that a particular user would like to use only one processing node. Instead of altering the net building mechanism, which reveals the maximum number of parallel paths in the program, the allocation phase can be made to take these conditions into account and adjust the allocation strategy accordingly. The point can be best illustrated through a simple example. Consider the following piece of code:

```
s1 a := 2;  
s2 b := a + 2;  
s3 a := 3;  
s4 b := b / a;  
s5 a := 4;
```

The execution of the resulting net creates the following multi layered parallel paths,

represented in the POP-11 style list of list format;

```

Layer 1:  [ [ [receive 1] [a := 2] [send 2]
            [receive 3] [a := 3] [send 4]
            [receive 5] [a := 4] [send ] ] ]
Layer 2:  [ [receive 2 var a] [b := [a + 2]] [send 4] ]
Layer 3:  [ [receive 4 var a var b] [b := [b / a]] [send ] ] ]

```

The first layer has three independent parallel paths but all those statements assign to the same variable. The second layer is only one block and so is the third layer indicating their sequentiality.

To utilize the maximum degree of parallelism, at least three processors are needed, specially when all three parallel paths have the same output variable set. If the system had three processors or if the output variable sets of the three parallel paths were not the same, then the allocation would be fairly simple. In this case, however, the compiler allocates the first two blocks to the two available processors. The symbol table at this stage holds the following information for variable "a":

```
[2 1] [4 2] {the first figure is the place number}
```

This shows that the latest values assigned to variable "a" are on processor 1 and 2 in the same order. All subsequent layers requiring these values are allocated to the appropriate processor before any other statement assigning to variable "a" is allocated. Further allocation of statements assigning to variable "a" will result in the update of the symbol table. In this example, therefore, statement 2 is allocated before the third element of the first layer could be assigned to a processor.

processor 1

processor 2

```

a := 2;           a := 3;
b := a + 2;      receive b
send b           b := b / a;
a := 4;

```

A New Method for Minimising Interprocessor Communication

The other issue to be addressed is the problem of keeping the communication overhead to a minimum. A new empirical method based on the *greedy* principle is developed allowing the communication requirements to be kept to a minimum. A specification is developed as follows:

1. Each processing node has two values associated with it, a load factor LF , and a communication factor CF . When allocating a block that does not need communication then it is allocated to a node with the least LF . For example, if there are five blocks in the first layer (blocks in the first layer do not receive any data), then they are allocated to nodes 1 to 5, if the system has that many nodes. This brings the load factor of each node to 1 and the total load factor of the system to 5. The communication factor remains zero for all nodes unless the results generated by this layer are required by subsequent layers. Note that if all blocks were allocated to the same node then the total load factor would still be 5 but the total load factor on node 1 ($\sum LF_1$) would be 5 too, with CF remaining zero. Since, in MINNIE's case, execution and communication overlap and take place at the same time, the aim is to keep a balance between LF and CF for each node.
2. Communication can often take longer than execution. To avoid serious performance degradation due to load imbalance, the ratio of CF to LF should be set to 2:3, for example. If this ratio is used to counter the imbalance then in subsequent layers, blocks are allocated one at a time to any processor that offers least $(\sum CF_n * 2) + (\sum LF_n * 3)$. Such a node is excluded from further

allocation until all the nodes have been allocated for a particular layer and there are still more blocks left in the layer for allocation, or a new layer has been started. Both situations lead to the release of all excluded nodes for new allocation.

3. Note that when a block is assigned to a processor then the amount of data it has to receive affects the calculation of CF for another block requiring the data. In other words, the maximum number of communications a layer needs is equal to the number of different variables that exist in the receive lists of all the blocks in the layer. Hence for each layer, where n is the total number of blocks in a layer:

$$Max(\sum CF) = cardinality(\bigcup_{i=1}^n (\text{variables in receive list})_i)$$

4. As soon as a block is assigned to a processing node then its receive list is updated with the address of the sending node, for all the variables in the list. The send list of the sending node is also updated and a record of the communication for each variable is kept in the symbol table as already seen.
5. The pseudo code below shows the structure and functionality of this part of the scheduler which ensures an acceptable level of communication overhead.

While layers exist do

```

for each element of the layer do
  if (element's place number > the symbol
    table entry for its output variable) then
    if (processor number in the
      symbol table = no of processor
      in the system) then
      rearrange the net
    end if
  end if
end if

```

```

    for each node in the system do
        compute the total LF + the total CF for the node
        store the result in an array representing the nodes
    end for
    select the node with the least combined load
    assign the layer element to this node
end for
end while

```

6. For scheduling of conditional blocks: in parallel systems based on associative memory addressing for interprocessor communication, a problem arises when a conditional block is being allocated.

A node requiring a piece of data from another node must have the address of the sending node made available to it at some stage, and in MINNIE's case at compile time. If a variable from the output variable set of a conditional block is required by another block on a different node then the receive list of the receiving block must reflect this. The execution of a conditional block does not update all the variables in its output variable set, and the updating depends on the path that the conditional block takes. The solution is to wait until all the conditional blocks are executed and then send any of the variables needed elsewhere.

Scheduling for Data Flow Machines

Generation of cells for data flow machines from this structure is fairly simple and no complications exist, as shown by the following algorithm:

1. Starting from the first layer, where there is no data to be received from anywhere, create cells for each block and connect the result of each cell to destination cells indicated by the send list.

2. If a destination list has more than two numbers, generate "dup" cells for multiple data flow.
3. Cells created in this way can then be refined down to individual operations level to utilize finer grain parallelism. In effect this involves decomposition of a large cell into a number of smaller sub-cells.

3.4 Fine Grain Parallelism Detection

Parallel computers with many processing elements may best be utilized if fine grain parallelism is exploited in some circumstances too. A scheme for identifying fine grain parallelism in a program and its mapping onto a network of processors is developed based on the following points.

1. If finer grain sub-expressions are to be evaluated by nodes which are also participating in coarse grain computations then this can lead to situations where a processing node may be held up, waiting for the result of a simple calculation. This can happen if the node evaluating the sub-expression is involved in a time-consuming computation, say a loop iteration, prior to computing the sub-expression.
2. Using the communication system provided by MINNIE it is possible to use a different scheme for utilizing fine grain parallelism without introducing long delays in computing sub-expressions. This proposed scheme is based on the idea of using *co-processors* for the computation of simpler fine-grain arithmetic expressions which need to be carried out quite urgently.
3. If the sub-expressions consist of constant values then there is no communication overhead except for sending the result of the computation. Sub-expressions which require the values of variables to be communicated to them,

e.g. $(a * b + 3) * a$, do not usually increase the communication overhead either. This is because of the associative memory addressing technique used. In the example above, it is most probable that variables a and b have to be communicated for other reasons in any case. As a result the node performing the fine grain parallelism will be able to obtain a copy of the value of those variables from the bus at no extra cost.

4. After the coarser grain parallel paths are scheduled ³, there probably remains several other free processing nodes in the system. The remaining free nodes, if any, are reserved for exploiting fine grain parallelism. For example, in a system with 20 processing nodes and an application program with a maximum of 13 parallel paths in one of its layers, 13 processing nodes will be reserved for exploiting coarser grain parallelism and the remaining 7 nodes will be set aside for finer grain parallel computation.
5. The nodes set aside for fine grain parallel computation of sub-expressions may send the result of their computations to the destination node as soon as they are available. Some of these results may even arrive at their destination before being needed.
6. Although the co-processor concept reduces possible delays in receiving the result of sub-expression computation, this can only be guaranteed if fine grain parallelism inside loop bodies are not used at the same time. This is because the sub-expression evaluation for an expression inside a loop has to have the same number of iterations as the other parts of the loop body, in other words, loop constructs are introduced on the nodes that carry out fine grain parallelism. To avoid this either fine grain parallelism inside a loop must be ignored altogether, or such evaluations must be restricted to one per processing node and must be the last block of code to be evaluated on such nodes.

³The number of nodes used for coarse grain parallel paths is equal to the maximum number of parallel paths in the layer with the largest number of elements.

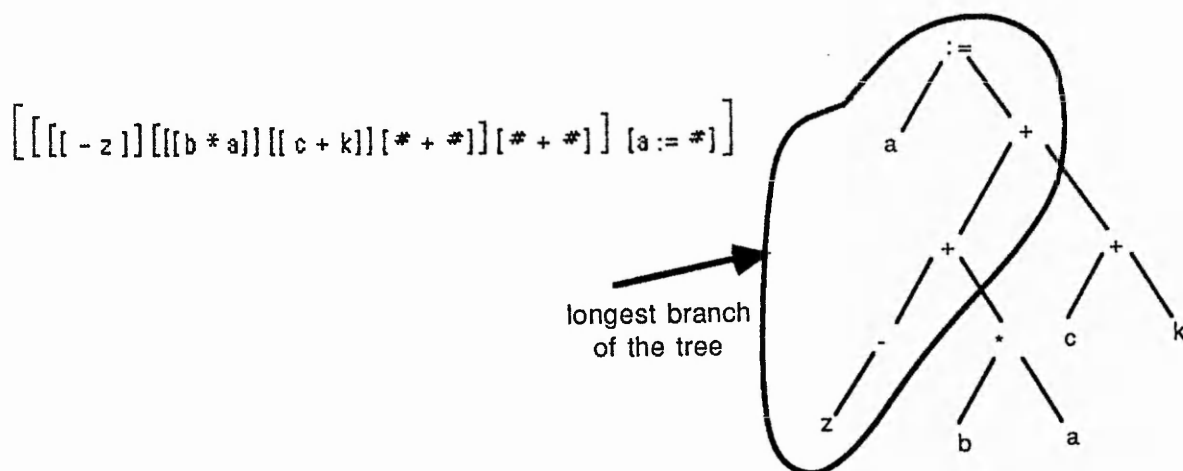
7. Since systems like MINNIE are envisaged to have thousands of processing nodes, it is more efficient to restrict the allocation of such fine grain iterative constructs to one per node without any other load on that node. This eliminates the delay in receiving results of sub-expression evaluations from other nodes, except for the usual availability of data. Fine grain computations without any loops are safely allocated to the same node since such expressions are evaluated very quickly.

Different schemes for exploiting fine grain parallelism in a computational unit will now be considered. Each scheme offers a different number of parallel paths and a certain necessary communication overhead. Examples of these schemes are presented here to show their effectiveness.

Consider the following expression, taken from a test program, and the compiler output which shows its decomposed form. The compiler output is in intermediate code. The character # in the intermediate code tells the translator to take what is on the evaluation stack as an operand for the operation which is to be carried out or it can indicate that the result of a sub-expression is needed.

```
a := - z + b * a + (c + k);
```

The tree representing the entire expression and its decomposed compiler generated internal form are shown below:



In the above internal form, each element represents a node of the tree and the symbol # represents the result of a sub-expression computation. Therefore symbol # represents a non-terminal node.

Four possible schemes are considered for the decomposition and allocation of sub-expressions for exploiting fine grain parallelism. The result of applying each of the four schemes to the expression above is shown below. In each case the possible distribution of the expression over a number of processing nodes is shown together with the communication overhead incurred.

- Scheme 1

output :	processor1	processor2	processor3	processor4
	-z	# + #	c + k	b * a
	# + #			
	a := #			

Comments

Three communications and three parallel paths.

Required processor cycles

82

- Scheme 2

output :	processor1	processor2	processor3	processor4
	# + #	c + k	b * a	
	-z			
	# + #			
	a := #			

Comments

This is the more traditional method which is based on the longest branch of the tree. It involves two communications and two parallel paths.

Required processor cycles

125

- Scheme 3

output :	processor1	processor2	processor3	processor4
	c + k	b * a		
	# + #			
	-z			
	# + #			
	a := #			

Comments

One communication and two parallel paths.

Required processor cycles

150

- Scheme 4

output :	processor1	processor2	processor3	processor4
	-z	c + k	b * a	
	# + #	# + #		
	a := #			

Comments

This involves two communications and offers three parallel paths. It gives the maximum number of parallel paths with the minimum amount of communication overhead.

Required processor cycles

82

The last scheme provides the best possible result and this is the scheme adopted by the compiler. It must be noted that the priority of operators is taken into consideration when allocation takes place. For example, if the above expression was $a := -z + b + a + (c + k)$; then the allocation would be different since the * operator (between b and a) has precedence over the + operator (between b and a).

output :	processor1	processor2	processor3	processor4
	-z	c + k		
	b + a			
	# + #			
	# + #			
	a := #			

The tree representation of the expression above is shown in figure 3.9.

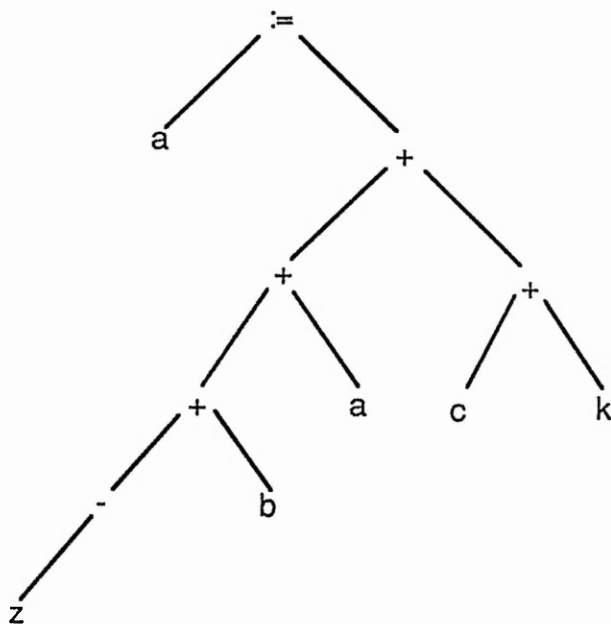


Figure 3.9: A tree showing the effect of operator precedence.

This is an important observation. It shows that a user can assist the compiler in detecting the maximum amount of fine grain parallelism by rearranging an expression. For example, by using brackets and grouping operations together the shape of the tree generated will change and the amount of parallelism detected will differ too. In the above example, if the user rewrites the expression as $a := -z + (b + a) + (c + k)$; then the shape of corresponding tree will be as shown below and will offer greater amount of fine grain parallelism compared to the expression in its original form.

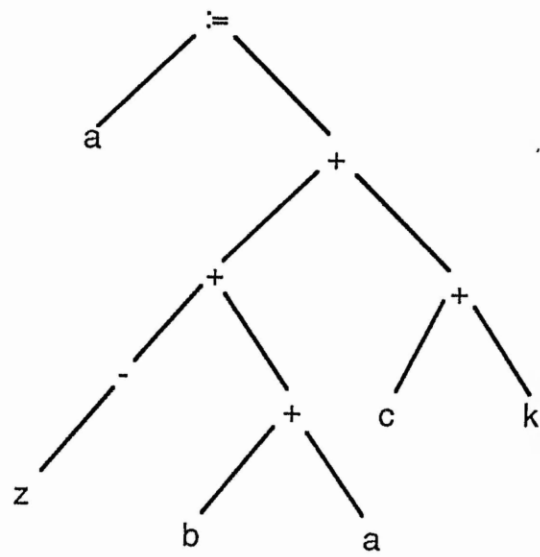


Figure 3.10: A tree showing the effect of expression regrouping.

While the remainder of this chapter introduces a new scheme for symbol-table management and an approach to variable storage in parallel processing environments, appendix B gives an insight into the structure of the compiler and its more traditional components like the lexical analyser.

3.5 Symbol Table

Symbol tables store information about identifiers, i.e., type, value (if a constant), bounds, fields and memory addresses. Scope of variables and declaration points can also be deduced from the table.

Compilers for parallel machines decompose the source code into parallel paths and subsequently allocate them to different nodes, and these make further uses of symbol tables. This can take the form of keeping information like the node address of a variable where its latest value can be found. In some architectures, e.g. MINNIE, a variable can have old and new values on different nodes. This kind of information can help an allocator to distribute the code appropriately and insert the correct communication code.

During the compilation process, storage and retrieval of information on and from the symbol table can take a large part of the whole compilation time, specially if the number of variables is high.

In languages like Fortran and Basic where only a finite number of simple data types exist and no nested procedure definitions is allowed, the actual symbol table can take a very simple form. For other languages with complex data types and nested variable scoping, a more complex method is required.

3.5.1 A Dynamic Sparse Symbol Table

To increase performance and reduce memory space required for symbol table storage, a new method is proposed. The features of the method are developed as follows:

1. The method should allow definition of variables of the same name but different scoping and attributes, as required by languages like Pascal.
2. Figure 3.11 shows the data structure used to store the symbol table. Symbol tables are inherently sparse and therefore, for efficiency considerations a special sparse table storage method should be used. POP-11 has built in sparse table generator routines for this purpose.
3. The constructed table has a row showing the parent of any given procedure or function. Note that the main program has no parent and therefore `table(main, parent)` should return false.
4. Addition of data always occurs at the column that corresponds to the current procedure. The table is horizontally indexed by procedure names and vertically by a variable name.
5. As soon as a procedure definition begins, `table(procedurename, parent)` is updated with the name of the last procedure (`parent`) which could be `main`. To illustrate consider a simple example;

```
program test;
  var a,b : integer;

  procedure A1;
    var a : integer;

    procedure B1;
```

```
        var b : integer;
    begin
        BODY OF PROCEDURE B1;
    end;

begin
    BODY OF PROCEDURE A1;
end;

procedure C1;
begin
    BODY OF PROCEDURE c1;
end;

begin
    MAIN BODY;
end.
```

6. The corresponding entries in the symbol table are shown in figure 3.12. Starting from main, the entry to parent's row is false. The attributes to variables a and b are added under column main. When the definition of A1 starts, the attributes of procedure identifier (pidentifier) A1 are added to the table and immediately under column A1 the parent row is updated to main (the parent of A1). The current procedure name is changed to A1 and as a result addition of new variables occur under column A1 as expected. When definition of A1 is complete, current procedure name is changed to the name of the parent, main in this case. This is done to take care of variable scoping.
7. The search to access data always begins at the current procedure column. Absence of an entry for the variable in the present column leads to a search

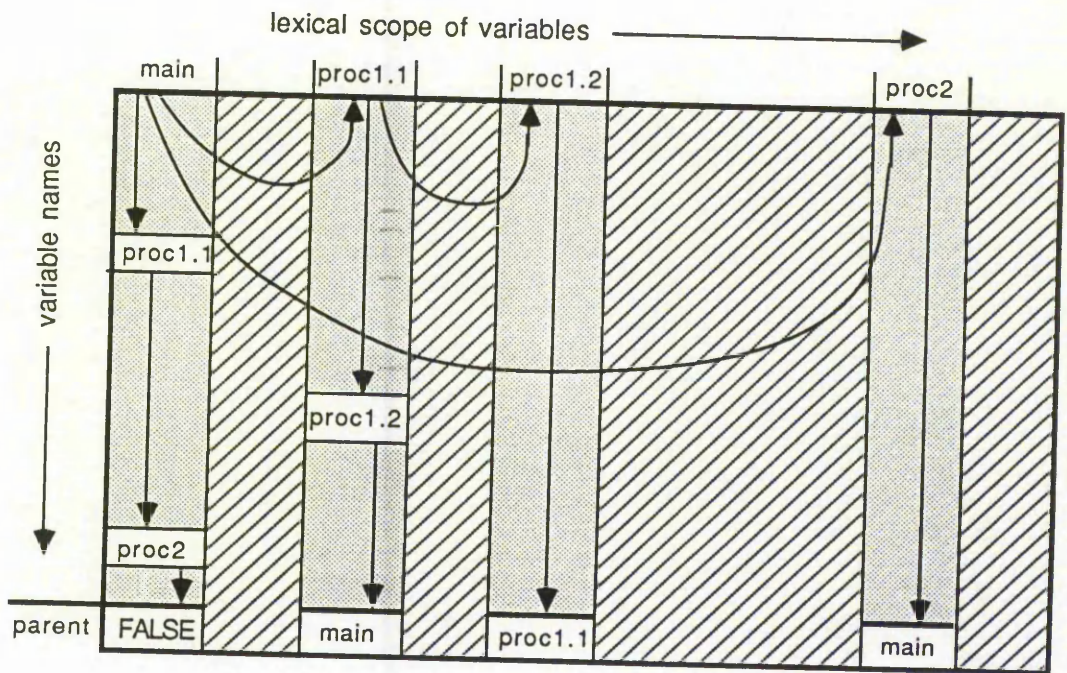
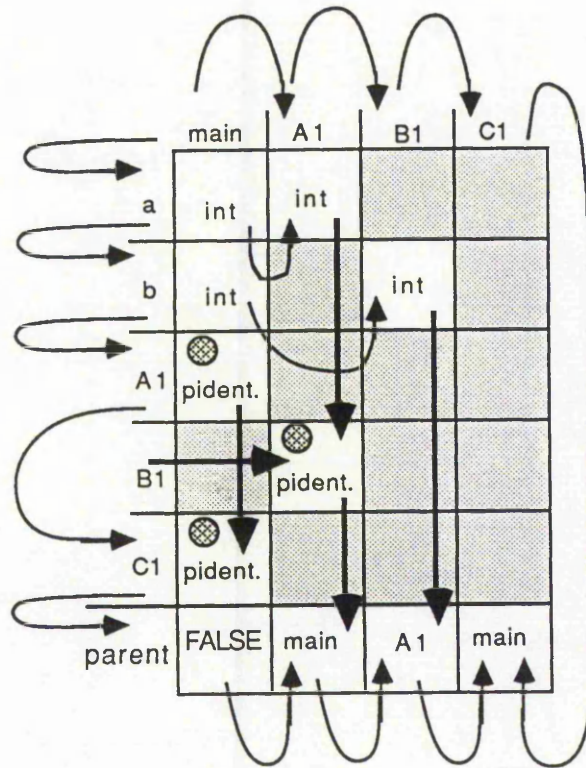


Figure 3.11: A typical symbol table structure



note: only table entries (11 in this case) use memory space

Figure 3.12: Entries and the pointers in the dynamic symbol table

in the parent's column until the relevant information is found or all relevant columns have been searched. As there is no hashing scheme for indexing the table the possibility of collisions is eliminated during storage. The search of the table is fast since the whole structure is linked through pointers. If there is no entry for a particular variable anywhere in the chain then an error condition is declared.

8. To illustrate, assume that variable *b* in the last example is referenced in procedure A1. `Table(A1, b)` returns false thus, if *b* is declared then it must be in the parent procedures or in the main, which renders it a global variable for procedure A1. The search in the main returns *b* as an integer type. The entries in the table are trees for each element which may hold a number of information about any variable. The exact information required must then be extracted from this tree, by a matching process in this case.

3.6 Variable Storage

Storing variables in a distributed memory system is difficult even for languages like Fortran that do not allow recursion. To allow recursive procedure calls and use of reentrant code many parallel computer systems either use a tagging scheme or have hardware support. In conventional computers, the *stack frame* method is used for the storage and referencing of variables to allow dynamic variable scoping.

For any programming language to be useful in practice, it must support data structures, e.g., arrays. A major problem in distributed memory parallel processing systems is the sharing of these data structures between different processing nodes. In the case of an array, for example, if one node alters one of the elements of the array then how is this change going to be reflected on the local copy of the array on another node? As the index to data structures could be decided upon during run time, the simple approach of broadcasting the new value would not work. The

options open are either to broadcast the whole data structure every time that a change is made to one of its elements or communicating the pointer to the element as well as the new value.

The first approach is clearly not practical as the sheer size of data communication overhead could degrade the system performance to an unacceptable level. The other approach is not very attractive either. It means an extra pointer (between 2 to 4 bytes) communication for every data structure update. Until there is more hardware support for solving this problem the only practical approach is the suggestion of using a different programming approach for distributed memory model.

To overcome the problems, a distributed stack frame method is implemented for storing and referencing variables. The proposed scheme is specified as follows:

1. Effectively, each node has its own stack frame mechanism for the variables it is going to use. This means that a node may have an old value of a variable in its local memory if it is not to use that variable again.
2. If a node is to reference a particular variable then a copy of its latest value is taken from the bus when its value is broadcast and the local stack holding the variable is updated.
3. There may be situations where a node has the old value of a variable but does not copy its new value although it is going to reference it later. This can only happen if the variable has another assignment to it before any reference is made. In other words, the node will only update a variable's value when the right assignment to it occurs.
4. As there are no elegant universal solutions to the problem of shared data structures, the following approach was adopted. Data structures *are not shared* between different nodes and as a result all the statements and routines referencing or updating a particular data structure are placed on the

same processing node. This may result in a loss of parallelism detected earlier by the compiler. With more hardware support in a future versions of MINNIE, a different approach by the compiler may be feasible.

If parallel programs were modeled on the principle of *independent communicating processes* or tasks then the problem, discussed above, would be greatly reduced. The programming language Occam is based on this philosophy and similar concepts are used in other programming languages like Fortran, C and Pascal with the aid of extensions and library routines too.

In addition, using suitable applications would reduce the problem even further. As an example, applications based on the master and slaves model where each slave process performs a different set of operations on a given data will run very efficiently on MINNIE. In this sort of applications some data is broadcast by the master process and all of the slave nodes will copy it. Each slave performs its own operations on the data and will inform the master that it has finished. The master will then broadcast the next piece of work packet.

Chapter 4

Implementation

This chapter outlines the major building blocks of the compiler, their design details and any implementation problems faced during coding. Each block of code is detailed by giving its overall structure in terms of Petri nets, where appropriate, and the functionality of its important routines. This takes the form of a guided tour of the program.

A more formal description of the program and its structure is presented in appendix E. The formal design is based on Jackson's Structured Programming or JSP [Storer 91].

4.1 The Development Environment

The compiler was written in the artificial intelligence language POP-11. The language POP-11 is supported in an operating environment called poplog. It offers an integrated development media which consists of an incremental compiler, providing interactive compilation, a powerful editor called ved and some debugging facilities. Within the same environment a version of common Lisp and Prolog are

also available. Facilities for combining routines written in C, Pascal and Fortran with POP-11 programs are also provided.

4.2 Net Construction

The tree generated by the syntax analyser is passed to the procedure `parallelize`. The tree is transformed into a list whose elements are trees representing the successive statements of the source program prior to the invocation of procedure `parallelize`. This is achieved by a call to procedure `del_nested_loop`. It flattens an input tree and removes any nested lists; for example, it transforms a list of the form `[[ab]]` to `[ab]`. Logically related statements are glued together as one item in the list, for example, a whole loop can be one item. The structure `transition_list` at this stage is in reverse order and contains a copy of the flattened list. The tail of this list which is the next statement of the source code is removed. A temporary list containing the original list which does not include the tail is also constructed. These two data structures are passed to the procedure analyser. A for loop construct repeats this processes for all the elements in the list. Variable `index` enables the procedure `allbutlast` to identify the elements of the `transition_list` before the current element. All the output of the for loop are left in a list which would have the same number of elements as the original `transition_list`.

Procedure analyser identifies the type of each construct (statement) under consideration and in turn invokes the appropriate procedure. For example, if the statement under consideration is an assignment statement then procedure `assignment_analyser` is activated. This approach offers a modular scheme which can in turn speed up future additions.

Procedure `assignment_analyser` itself has a number of procedures that it uses to carry out its function. It starts by extracting the variables in the right hand side of the assignment statement, by calling procedure `getalistof_variables` which

returns a list. If this list is empty then there is no data dependence and as a result this node is connected to place one. If the list is not empty then all the variables in the list are compared with the output variables of the statements above to establish data dependence. Data anti dependence is not considered at this stage. The output of the `assignment_analyser` is a list containing a number of integers referring to the statements that create data dependence and an integer referring to the place number of the current statement.

Similar analysis is carried out by other procedures for the other programming constructs. For example, `for_analyser` is for analysing the parallelism in a for loop and `readln_analyser` is for analysing a call to `readln` routine.

4.3 Net Execution and Parallelism Detection

Net execution for parallelism detection is carried out by the procedure `net_executer`. This procedure calls itself recursively until all of the net is executed, ie. the variable `executable_list` is empty or none of its transitions are executable due to nodes with non empty place list. Each layer of the parallel code is detected in one invocation of the procedure. Subsequent layers are identified by a new call to this procedure.

A for loop executes all the nodes that can be fired simultaneously, that is in parallel, and creates a list called `res`. This structure holds pointers to the statement that belong to a particular layer and can be executed in parallel. According to these pointers, a number of statements are grouped together, as a layer of parallelism and put on the stack, variable `res` later on will contain this layer.

The transfer of tokens to places is carried out to generate a new list of places with tokens. This is achieved by calling procedure `getnewpattern` and assigning its output to variable `mypatern`. Procedure `getnewexecutable_list` deletes the nodes just executed and returns the net in the new state assigned to variable

`myexecutable_list`. A new call to the main procedure `net_executer` executes the next set of nodes in the net which are capable of being fired.

After the final call to `net_executer` all the values on the stack are collected together in a list. Each element of this structure is a list itself containing a particular layer of parallel paths. Since all of the outputs of this procedure are left on the stack therefore there is no formal output by the procedure. Input to `net_executer` are three lists, `mytransition_list`, `myexecutable_list`, and `mypattern`. In the first invocation of the procedure, `mypattern` only holds the integer 1 indicating a token in place one.

4.4 Scheduler

This is one of the most important parts of the compiler in its parallel mode. The main routine of this phase is called `allocate`. It calls `allocate_1`, `allocate_2` for each element of the data structure `allocated_list` which holds the parallel paths generated by the net executor. The call to `allocate_1` is for allocation of statements according to *greedy* method to different nodes of the system. As part of this `send` and `receive` primitives will be assigned processor numbers instead of statement numbers. It also allocates I/O statements to processor number 1. As a result of allocation in this phase some of the `send` and `receive` primitive do not have the updated processor nodes for their communication.

Repeated calls to `allocate_2` ensures that all of the statements have a valid communication address. It updates all of the `receive` primitives when it encounters a `send` primitive and vice versa. This process is performed for all of the processors in the system. Procedure `allocate_3` does the final shuffling of the processor loads and updating of the communication primitives. The actions taken are sequential and are fairly clear from the code. Detection and allocation of fine grain parallelism is performed by procedure `allocate` too.

4.4.1 Fine Grain Parallelism Detection

The implementation of fine grain parallelism detection turned out to be quite simple in POP-11. After the initial parallelisation of the source code the data structure `sheduled_for_translation` does not hold any allocation information. Fine grain parallelism is performed at this stage on any statement which has the potential for fine grain parallelism. It is carried out by a call to procedure `fine_grain_detector`. It in turn calls procedure `analyse_set_of_independent_codes` for every layer of the list. This is the routine which identifies the suitability of a statement for fine grain parallelism and to the suitable ones applies the routine `fine_grain_a_statement`. This routine applies the algorithm presented in chapter 3 to all the statements and returns a list structure which contains the statements in the form seen in the examples shown in the previous chapter.

After the allocation of coarser grain parallelism is complete, procedure `allocate` calls the routine `fine_grain_allocator` for utilising the remaining processing nodes for fine grain parallelism. By mutually recursive calls between this routine and procedure `allocate_subexpression` the task of allocation in each case is reduced to the simplest possible form.

4.5 Overall Structure of the Compiler

The compiler consists of several fairly independent but interactive components. Figure 4.1 shows these components and their interaction. The user interface is responsible for error-handling, issuing of advisory messages to the user and requesting input from the user. Lexical analyser is concerned with generation of tokens from the source files whenever a request is made by the syntax analyser. The syntax analyser either calls the parallelism detection phase if there are no detected errors or allows the user interface to halt processing after tidy up. Parallelism detection is carried out if the number of processors is greater than one. The translation

phase is enabled by parallelism detection module n times where n is the number of processors specified by the user. In fact the number of elements in the data structure `allocated_list` determines the number of times which the translator must be called.

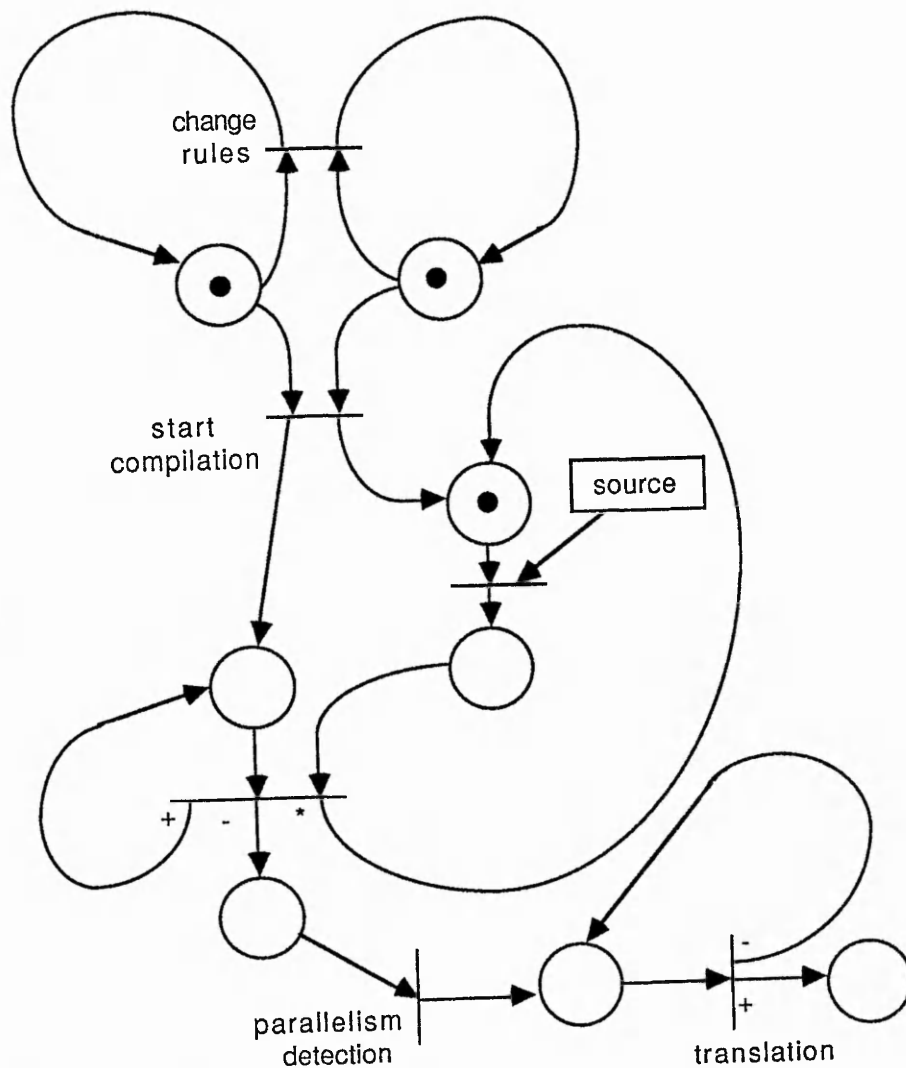


Figure 4.1: Overall structure of the compiler in terms of a Petri net

4.6 User Interface

The user activates the compiler by calling procedure `pascal` and giving it a file name as an input parameter. This is the main procedure of the whole compiler. It checks if the user needs to change any of the lexical or syntactic rules. Rule changes are indicated by issuing `new_rules` instead of a file name as the input parameter to procedure `pascal`.

Procedure `copy_new_rules` is the main procedure which controls and directs a user in changing lexical and syntactic rules. A copy of the new rules are made, on the disk, in files `data.rul`, `psprtab.txt` and `pstab.txt` which are used by the compiler. These files contain list of reserved words for the lexical analyser, production table and state table for LALR parser. This part is particularly user friendly and allows many changes before making a copy of the new rules on the disk.

The procedure `pascal` also sets up all the tables used by the compiler and initialises all the global variables. It calls procedures `initinput` and `initoutput` for opening the source files and the compiler produced listing file. The file `output.list` is produced by the compiler and holds the source code with line numbers and error marking where appropriate. Procedures `shut_input` and `shut_output` are for closing of the open files at the end of compilation or premature termination due to error conditions.

The user must indicate the number of nodes in the hardware so that allocation is carried out accordingly. If a system has only one node then the compiler omits the parallelism detection phase altogether and creates sequential object code. The end of compilation is shown by the message `OK` by the compiler.

Error handling is carried out by procedure `error` which marks the error positions and tidies things up before halting compilation.

4.7 Lexical Analyser

This module generates tokens for the syntax analyser. Figure 4.2 shows the internal structure of the module.

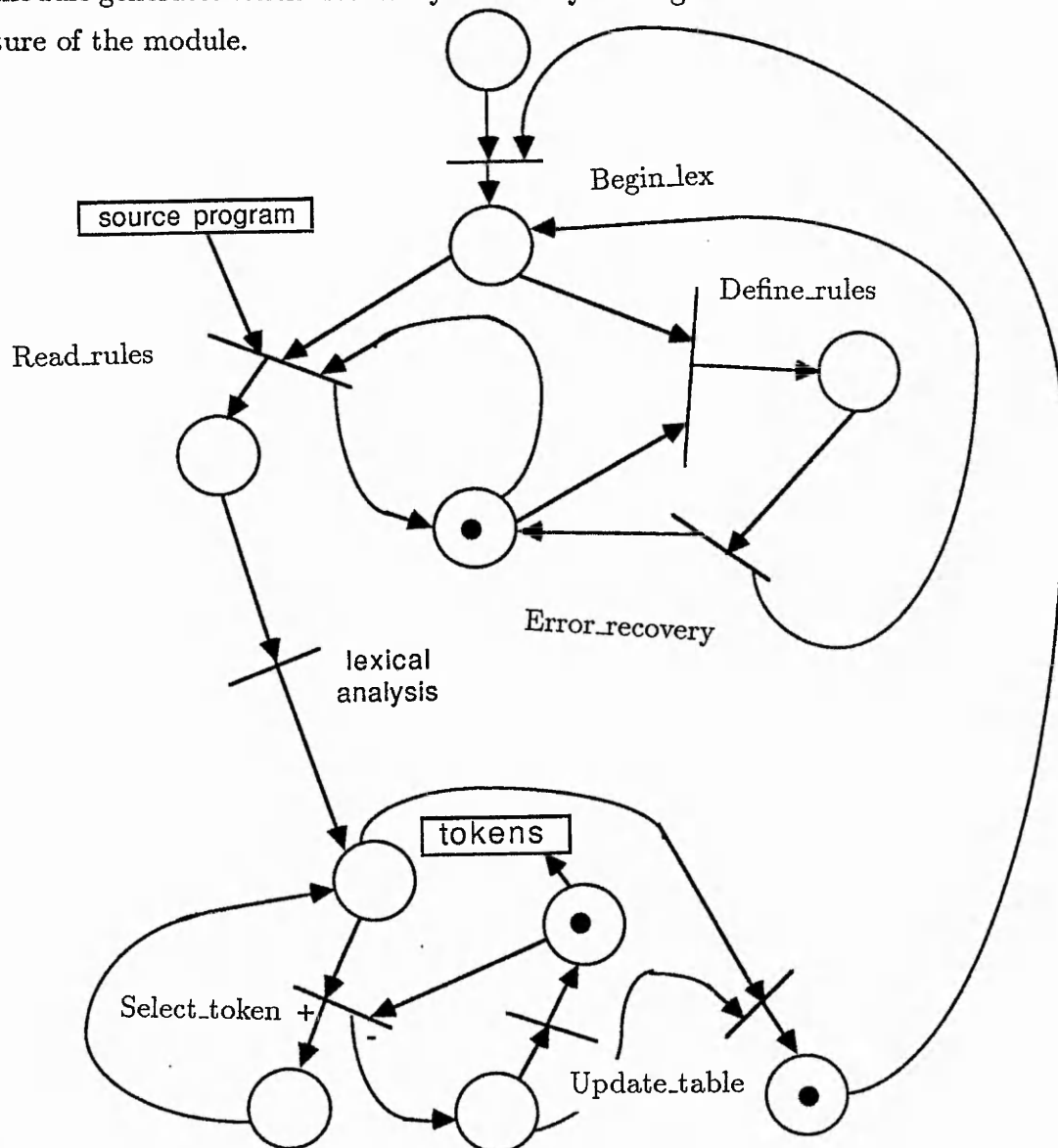


Figure 4.2: Internal structure of the lexical analyser in terms of a Petri net

The main procedure here is `lex`. It calls procedure `lexitem` which returns the next lexical item from the source code file according to current lexical rules. It sets up special flags if the item is a declaration header which is used in the construction of

symbol table.

The produced lexical item is passed to procedure `select` together with the current `flagtype`. The nature of item is determined by this procedure. It first establishes if the item is a number and if not then it tries to determine if it belongs to the set of reserved words and accordingly it returns a token.

A complication arises when a variable is encountered in the statement sections. The variable in question is firstly identified as an identifier by the lexical analyser while the variable could have been declared as a constant identifier (`cidentifier`), a variable identifier (`videntifier`), or a procedure identifier (`pidentifier`). The problem is resolved by the use of the flag set earlier and a call to procedure `lookup_symbol_table`. This procedure calls routine `findsym` which takes the current procedure name and the symbol, as its arguments, and searches the symbol table to return a result of false or type. If false then the symbol is a simple identifier; else, depending on the type returned an appropriate token is returned to the calling procedure. A property table is used to extract the token to be returned according to the type of the item returned.

Before returning control lex notes `lastitemname` and `currentitemname` which are global variables. This procedure never reports an error and always returns a token.

A major change made during development of this module was the omission of procedure `typeofidentifier` and instead making the symbol table search recursive if user defined type was used to determine its nature. This is accomplished by using procedure `isuserdefined` during symbol table search.

The original lexical analyser was very basic and changes were made to it as and when necessary. There was no major implementation problem here except in keeping track of what portion of a program is being processed, for example, type or variable declaration.

4.8 Syntax Analyser

This module consists of a large number of procedures, and not only carries out syntactic check on the source code, but also initiates additions to symbol table and building of a syntax tree. Figure 4.3 shows the overall structure of this module.

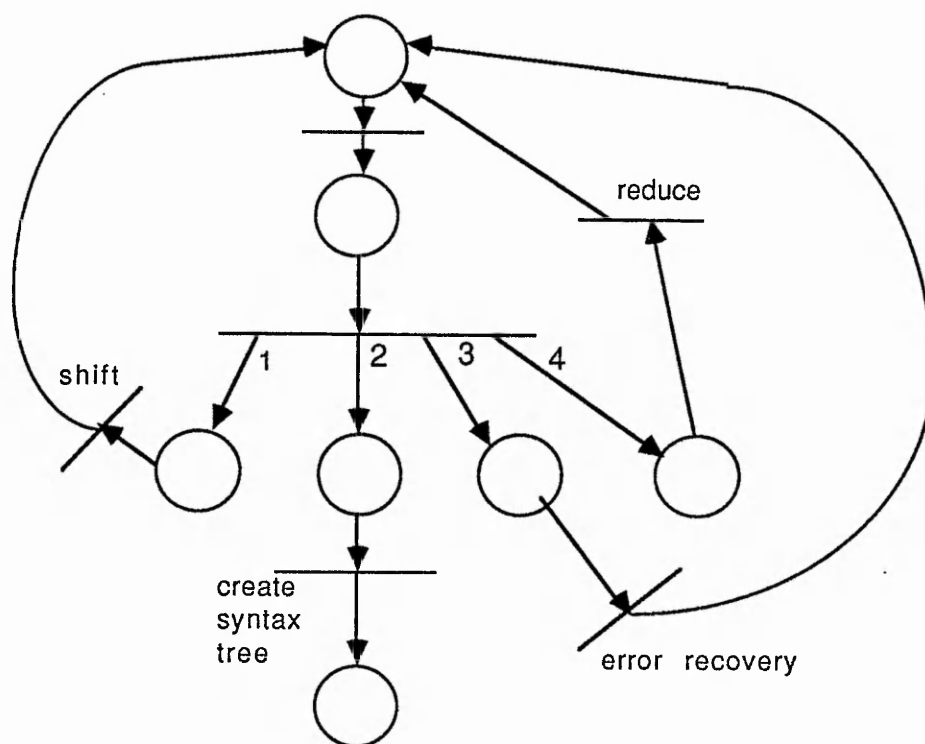


Figure 4.3: Overall structure of the syntax analyser in terms of a Petri net

The main driving procedure of this module is *drive-parse* which is called by procedure *pascal*. It firstly creates the necessary parsing tables and lexical rules by

calling two procedures `create-table` and `make-rules`. These two procedures use the files already mentioned to set up current rules. It then calls `lex` to generate the first token which it assigns to the variable `lat` (look ahead token). The next step is to push a 0 onto the parser stack as a marker using procedure `push` (this only happens once at the beginning of compilation). There are two procedures `push` and `pop` to put or remove tokens from the top of the stack. The final act of this procedure is to call procedure `do-parse` which recursively calls itself until parsing act is complete.

4.9 Parsing

According to the content of variable `lat` and top of the computation stack this procedure (`do-parse`) either calls procedures `shift` or `reduce`. The decision is made by calling the procedure `look-up-table` which returns a pair. The head of this pair is either an `s` (shift), `r` (reduce), `accept`, or `e` (error). The tail of this pair is either a number or `nil`. If head is an `s` then the tail indicates the new state and if head is `r` then tail indicates the rule number by which to reduce.

The action of procedure `shift` is exactly as described in the theory of LALR parsers. But for implementation purposes procedure `reduce` not only reduces a sub-tree according to a preset rule but also initiates addition to symbol table, when appropriate, and adding nodes to the syntax tree by calling procedure `make_tree`. Addition to the tree is initiated after any successful reduction but addition to symbol table only occurs when a complete description of a variable takes place, be it a `const`, `type` or `var` definition. These two procedures always call procedure `do-parse` again before returning control to the caller.

By the time the first invocation of `do-parse` regains control a complete parse tree is available in the variable `tree` or `tree-of-names`. A number of routines are called at this stage to transform this parse tree into a syntax tree. Finally procedure

levellizer is called to insert level numbers at each level of tree for use in tree-display module.

The parser takes a source file and outputs a syntax tree of the source code. When an error is detected procedure error is called to take the necessary actions. The main data structures used in this module are a table, indexed by variable lat and state number, for holding the parse table. Each element of this table is a pair. Also a sparse table is used to hold information about symbols, indexed by symbol name and procedure names. The other main data structure is a tree representing the source code and a list used for keeping partially reduced item(s) for matching purposes.

4.9.1 Tree Generation

The input to this procedure is a tree (root only at the beginning), the number of nodes that has to be used from the tree (n), for matching against the reduction rule being used, and finally the sentences being reduced to (m).

During the construction of the tree, numbers representing state table index are also inserted in the tree. These numbers group a number of elements together. During tree construction a number of elements from the head of the tree is grouped together, according to the value of n . A local node is created by these elements and any state number with any of them is removed. This new node is then given the value of m as the state number and added to the tree as a new node. There is also provision to ignore brackets and any other information not represented in a syntax tree.

When a procedure is being defined, a tree representing its structure is created and when its definition is complete; this tree is analysed and coded before continuing. Procedure definitions are not included in the tree belonging to the main program's body. Despite that, the procedure is represented by a node representing the pro-

duction entity proceduresec. This is necessary for reducing the remainder of the program.

This part was particularly difficult to implement and to gain a clearer picture it is best to study the actual code (see attached disk).

4.9.2 Tree Display

During the course of this research it was felt that it would be helpful to both the research program and future program developers if an automatic syntax tree display and manipulation tool was available. As a result a tree display routine was developed and implemented. As computer aided "parallel processing program development tools" are usually interactive and iterative, provisions for future additions and developments have been taken into account to facilitate this. There are a number of functions displayed in a menu on the right hand side of the display screen that a user can choose from for manipulating the tree. The menu is displayed by the procedure `draw_menu` and the function names are inserted by the procedure `write_menu`.

The tree display is controlled by the procedure `screen_manager`. It starts by clearing the screen, drawing the menu and then displaying the top four layer of the tree. It then starts an infinite loop which is controlled by the user choosing one of the functions on the menu. One of the options is "exit" which terminates the loop.

The main procedures of the tree display are for identifying the nodes and information with regard to each node's neighbour. Procedure `showtree` is the main procedure which calls procedures `shapetree` for establishing the tree shape and displaying the nodes. The relation between nodes is established by procedure `nodefacts`. Procedure `connectup` connects these nodes by using procedures `straightjoin`, `leftjoin` and `rightjoin` according to the position of the two nodes being joined together.

4.10 Symbol Table Manipulation

To create the symbol table, at the beginning of compilation procedure `pascal` calls the routine `setupsparse`. This procedure has no input or output parameters. In turn, it makes a call to the POP-11 routine `newanyparse` which sets up a sparse array returning false for all its empty elements. It also sets the variable `procname` to "main" and assigns the sparse table to the variable `currenttable`. A few rows containing specific information about each procedure are also created. For example, there is a row which stores the parent name of each procedure.

Central to the symbol table operation are the procedures `newproc`, `endproc`, `setmum`, `getmum`, `findsym` and `addsym`. Procedure `newproc` is invoked whenever the definition of a new procedure begins. It first makes a call to the procedure `setmum` to set the name of its parent in the appropriate row. The argument to `setmum` is the name of current procedure given by variable `myname` inside `setmum` and `procname` outside it. It also sets the values of the locations in the table, showing the procedure's number of formal parameters and sets memory size to zero and the value of procedure's nesting level to the variable `levelnumber`. This variable is incremented before this operation.

Procedure `endproc` is called whenever the definition of a procedure is complete. It calls `getmum` to set the variable `procname` to the name of current procedure's parent. This operation ensures the correct lexical scoping when the table is being searched. Procedure `getmum` has one input argument which is the current procedure name; it does not return anything. It also decrements the variable `levelnumber`.

To add any information to the table, a call is made to the procedure `addsym`. It has three input arguments, the value to be added, variable `val`, the table column, indicated by variable `procname` and finally the table row indicated by the variable `sym`. The addition is achieved by assigning the value of variable `val` to `currenttable` at the location specified by the column and row indicators.

Procedure `findsym` is more complex. It takes variable `procname` and the symbol for which information is required as input arguments and generates a result. The result is either `false` or an item which can be any structure. By using an until loop it starts the search in the current procedure's column. If the information is not found it continues to search in the column of the parent by changing the pointer. This process is repeated for each parent until the information is found or if there is no more parents to be searched then the value `false` is returned.

4.11 Code Generation

Translation is into 6809 assembly code which needs to be assembled later using the system's assembler, called Mace.

The main routine of this module is procedure `code_main`. For each of the specified nodes this routine sets up the stack and various pointers which are used for referencing the stack base and top. It then calls routine `pre_coding` which translates the trees into a form suitable for translation. Routine `coding` is then repeatedly called to translate the work load of each processing node. It takes a syntax tree as its argument which represents the work load of a particular node.

Procedure coding is the heart of the translator. It calls itself repeatedly until there are no sub-trees left for translation. Depending on the type of the statement, procedure coding calls one of the routines which have the prefix `plant_`. For example, for a case statement it calls the routine `plant_case` and for a while statement it calls `plant_while`. Note that for built in functions and routines this compiler generates inline code rather than providing a library which would then require a linker.

Chapter 5

Results and Discussions

5.1 Introduction

The most important test a compiler must satisfy is the correctness of the translated machine code, i.e., the translated version must functionally be an exact equivalent to the high level source code. Section 5.2 presents the test results that show the correctness of the generated code for a single node of this computer.

Achieving correctness and testing for it becomes more difficult when a parallel processing machine is being used. Here not only the translated code must be correct, the compiler generated communication codes must also be correct to ensure the integrity of the results generated by the execution of the program. In asynchronous parallel processing systems hardware and software must further ensure that the results are not only correct but are predictable and repeatable too. In the case of MINNIE, synchronization and repeatability are achieved through hardware support for communication primitives generated by the software. The synchronization model used is based on message passing rather than the use of semaphores.

The absence of real working hardware limits the option for actual proper testing of

the compiler output; obviously this would have been the way to test the compiler. In order to be satisfied that the compiler output for a multi node system is both correct and efficient, the next alternative solution is to devise small purpose designed pieces of code and observe that the output code (at a higher level than assembly or machine code) is consistent with the specifications and schemes presented in chapter 3. One of the advantages of this method is to isolate the adverse effect that the communication system may have on the actual timing. Apart from some small examples, which are discussed in general later on in this chapter, a more realistic example is worked out in long-hand and its 6809 assembly code is presented in appendix G.

5.2 Single Node Configuration

When a single node configuration is selected the compilation process speeds up and parallelizing modules are bypassed. The compiler is heavily instrumented for testing and debugging purposes and therefore its current slow speed is not a true indication of its eventual speed. However, because of the many extra information that a parallelizing compiler has to store, and the many more modules that it has, compared to an ordinary compiler it will always be slower.

A full Pascal-S language was implemented for a single node machine and the tests were carried out as the translator was being developed. Appendix F gives one of the test programs used in the verification of this compiler.

The compilation output of the test programs, shown in appendix F, demonstrate that full advantage was taken of flex operating system routines in particular for I/O.

An important area that needed extreme care was the proper operation and management of the stack frame variable storage and thus the correct scoping of variables. As part of the test-set the following simple test program was run and the results

showed that this part of the compiler's translator works correctly.

```
program test(input, output);

    var a, b : integer;

    procedure scope-test ( a : integer; var c: integer);
    begin
        a := a + b;
        if a > 5 then
            writeln ("local value of a = ", a)
        else
            begin
                writeln ("global value of b = ", b);
                c := a;
                writeln ("local variable c = ", c);
            end;
        end;
    end;

begin
    readln (a,b);
    scope-test (a,b);
    writeln ("global a = ", a);
    writeln ("global b = ",b);
end.
```

Table 5.1 shows some of the test results obtained by executing the above program.

Operation of for, while and repeat-until loops were also demonstrated to be correct together with the operation of the conditional if-then-else construct. Nested conditional statements were also used in the test programs and results were

input	output	comments
1,2	global value of b = 2 local variable c = 3 global a = 1 global b = 3	value of b inside the procedure variable c inside the procedure value of a in the main program value of c has been passed to b
2,4	local value of a = 6 global a = 2 global b = 4	value of a inside the procedure no change in the value of a no change in b

Table 5.1: Some test results for variable scoping.

as expected. All operators were used in the various test programs and gave the anticipated results. All the tests presented in this section were carried out on a physical machine, that is, the object codes generated by this compiler were executed on a Windrush 6809 based machine.

5.3 Multi-Node Configuration

The best test would have been to use a large off-the-shelf program in order to measure the usefulness of the compiler's technique for detecting parallelism. There were several fundamental limitations that did not allow such tests to be carried out:

- The compiler is a prototype and does not have all the necessary library and run-time support which may be required for a large complex program. In the given time frame, it was not realistic for one person to develop a complete production compiler and run-time environment, from scratch, quite apart from the extra sophisticated functionality of parallelism detection. It was understood from the beginning that the aim was to prove certain ideas for parallelism detection and not to come up with a complete Pascal compiler. There already exist powerful sequential compilers for Pascal.
- In the thesis much emphasis is put on the possibility of existing software benefiting from parallel processing. Even if the compiler were able to compile

a large application program without any problem it is no measure of the effectiveness of the compiler if the end result of compilation of such a program is a program with little parallelism or even no parallelism. In short, the parallelism detection can only work if there is inherent parallelism in the application program. Also, due to the bus allocation technique used in the hardware, it could be the case that a parallel version of an application would run slower than the sequential form of the same program.

- The hardware platform for which the compiler was designed has never been completed. It is impossible to have a meaningful test without the hardware or a simulator.
- Even if the hardware were complete, because of its limitations, which are discussed in detail in the subsequent pages, it would not be possible to run any useful program on this hardware anyway.
- The compiler generated 6809 assembly code. It is not feasible to look at a large set of assembly code and discover the amount of parallelism detected. Even if this were possible, the only way to be sure that the generated code is correct would be to execute the program on the parallel processing platform. There are complex issues like timing and communications which can only be tested by using the real hardware and not by looking at the assembly code. Due to timing sensitivity even a simulator would be unsuitable to verify the correctness of the generated code.

In view of the above, a realistic example is discussed here, in detail. It includes a description of all the major steps that the compiler would take to arrive at the parallelised code.

Later on, some small test programs are also considered. The small examples show that any large program containing those constructs would benefit from the parallelisation offered by this technique. Because the examples are small it is feasible to

look at the generated code and gain some level of confidence in both the correctness and the effectiveness of the compiler.

5.3.1 A Worked Example

Consider the following sequential Pascal-S program which approximates $\int_{x=a}^b (x^2 + x + 8) dx$, where a and b are values supplied by the user, by evaluating $\sum_{i=1}^{10000} \delta((x + i\delta)^2 + x + i\delta + 8)$, where $\delta = (b - a)/10000$.

```

program area(input, output);
  const steps = 10000;
  var a, b, area, delta, i, strip, funcval : integer;
begin
  writeln (" Please input the value of upper and lower bounds"); {1}
  readln (a,b); {2}
  area := 0; {3}
  delta := (b - a) / steps; {4}
  writeln("iteration step :"); {5}
  for i := 1 to steps do {6}
  begin
    write (i); { i}
    funcval := (a ** 2) + a + 8; { ii}
    strip := funcval * delta; {iii}
    a := a + delta; { iv}
    area := area + strip; { v}
  end;
  writeln; {7}
  writeln ("The approximate value of area = ", area); {8}
end.

```


There are 5 steps to be taken by the compiler:

Step 1:

The front-end of the compiler generates a syntax tree representation of the program and ensures that the source is correct syntactically.

The syntax tree is transformed so that all the logically related statements are grouped together in the form of a list-of-lists (shown below). This is passed to the paralleliser.

```
[
  [writeln [Please input upper and lower bounds]]
  [readln [a , b]]
  [area := 0]
  [delta := [b - a] / 10000]
  [writeln [iteration steps]]
  [ for i := 1 to 10000
    [[write i]
     [funcval := [[a ** 2] + a + 8]]
     [strip := [funcval * delta]]
     [a := [a + delta]]
     [area := [area + strip]]]]]
  [writeln]
  [writeln [The approximate value of area = , area]]
]
```

Step 2

Automatic generation of a Petri net model of the program begins by looking at each statement of the program in turn; starting with the first one which is independent of any other statement. For each of the blocks in the list-of-lists the input and output variable sets are computed. For example, the input and output variable sets to

blocks 4 and 6 are:

IN (4) = (b, a)

IN (6) = (a, area, delta)

OUT (4) = (delta)

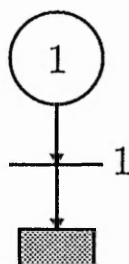
OUT (6) = (a, area, funcval, i,

strip)

In turn, for each of the blocks in the list-of-lists a transition is created and connected to the appropriate places. Now let's consider each block of the above list and generate the net:

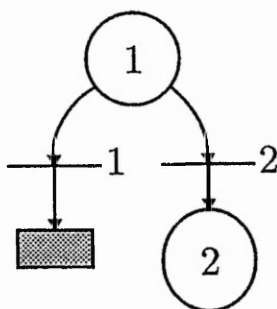
block 1

Empty input and output variable sets; independent of anyother statements and therefore its transition is connected to place 1:



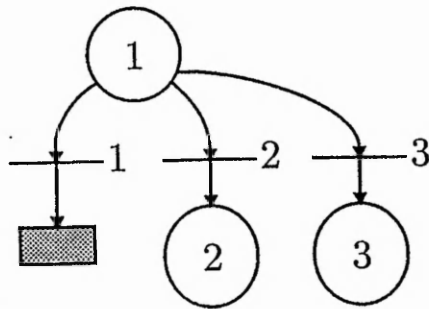
block 2

Empty input but output set contains variables a and b; similarly this block is independent of anyother statements too:

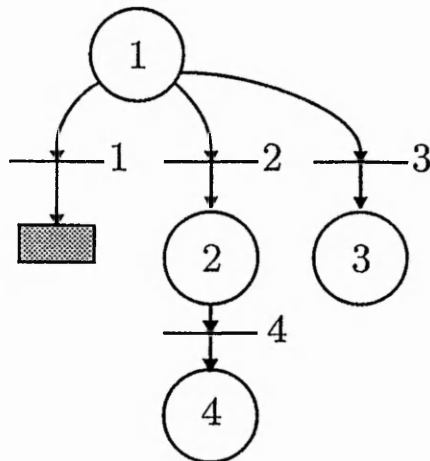


block 3

Empty input but output set contains variable area; independent of any other statements:

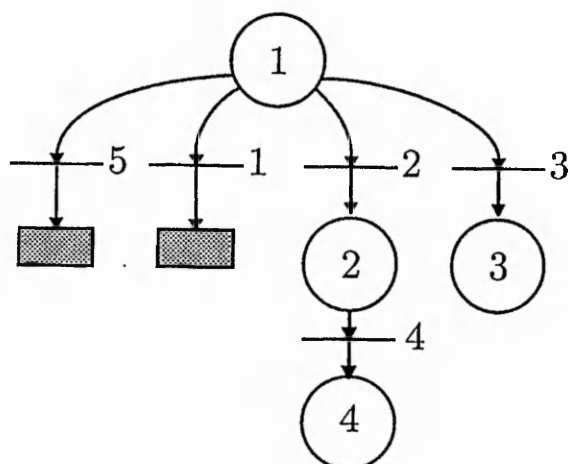
*block 4*

The input variable set contains variables b and a. Note that steps has been replaced by its value which is 10000. The output set contains variable delta which is dependent on block 2:

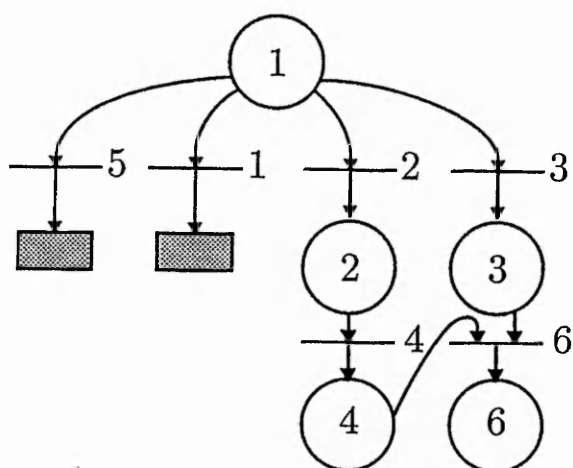


block 5

Empty input and output sets; independent of any other statements:

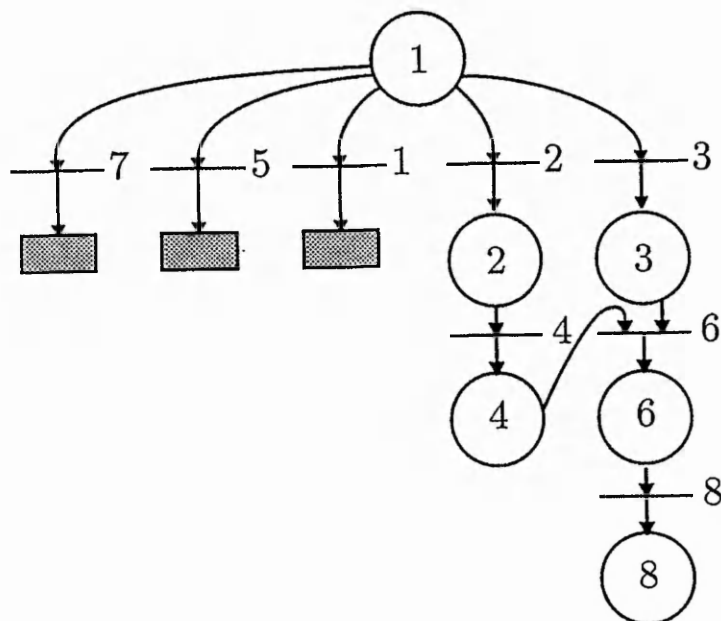
*block 6*

The whole of for loop is considered as one block at this stage. It is dependent on blocks 2, 3 and 4. Note that because block 4 itself is dependent on block 2, dependence of this loop on block 2 is ignored:



blocks 7 and 8

Similarly for the last two blocks:



Step 3

At this point the constructed net is executed and a multi layered parallelised code is generated. The parallelised code does not contain any communication information at this point:

The execution begins with looking at all the transitions which are connected to place 1. Transitions 1,2,3 and 5 are removed from the network and are added to layer 1. A token is put in the places of all the removed transitions which have one. The token from place 1 is removed. At this point all transitions connected to place 2 or place 3 or both are ready for firing. The list contains only one transition which can be fired, transition 4. So the next layer contains only one statement, i.e no parallelism. Similar to the previous stage the token (from place 2) is removed and a token to place 4 is added. The token list now contains 3 and 4 which means that

only these two places contain tokens. Transition 6 is the only one which can now be executed followed by transition 8. The result is that only the first layer contains parallel paths with the remaining part of the code being sequential. However, as will be seen later the loop itself can offer some parallelism which will be investigated at the allocation phase.

```
[
  [
    [writeln [Please input the value of upper and lower bounds]]
    [readln [a , b]]
    [area := 0]
    [writeln [iteration steps]]
    [writeln]]
  [
    [delta := [b - a] / 10000]]
  [
    [ for i := 1 to 10000
      [[write i]
        [funcval := [[a ** 2] + a + 8]]
        [strip := [funcval * delta]]
        [a := [a + delta]]
        [area := [area + strip]]]]]]
  [
    [writeln [The approximate value of area = , area]]]
]
```

As can be seen from the output above there are 4 computational layers where the last 3 layers each contain one block of code.

Step 4

Starting with the first layer, the elements of each layer is looked at in turn and is

allocated to a particular processing node. The allocation is in 4 phases:

phase 1:

Initial receive and send primitives are added to each block. The numbers merely represent the block numbers. This phase ensures that antidependence is resolved.

```
[
  [
    [[receive] [writeln [Please input the...]] [send]]]
    [[receive] [readln [a , b] [send 4, 6]]]
    [[receive] [area := 0] [send 6]]
    [[receive] [writeln [iteration steps] [send]]]
    [[receive] [writeln] [send]]]
  [
    [[receive 2] [delta := [b - a] / 10000] [send 6]]]
  [
    [[receive 2 3 4] [ for i := 1 to 10000
      [[write i]
        [funcval := [[a ** 2] + a + 8]]
        [strip := [funcval * delta]]
        [a := [a + delta]]
        [area := [area + strip]]]]] [send 8]]
  [
    [[receive 6] [writeln [The approximate value of area = , area]]
    [send]]]
]
```

phase 2:

This phase has 5 essential steps:

1. If the block contains I/O then re-order the parallelised code to enforce flow dependence.
2. compute the combined effect of LF and CF for each processor if the current block was to be assigned to it.
3. find the processor with the least combined load and mark it as current processor.
4. assign the block to the current processor.
5. update the work load table.

For assigning the first block it always starts from processor 1. All I/O statements are assigned to processor 1 in the order that they appear in the program irrespective of detected parallelism. The work-load of each processor is 0 at the beginning.

For this example, firstly the parallelised code is re-ordered and block 7 is moved to layer 3.

The parallel paths are then allocated as shown below. Each list contains the work-load of one processor.

```
[
  [[receive] [writeln [Please input the...] [send]]]
  [[receive] [readln [a , b] [send 4, 6]]]
  [[receive] [writeln [iteration steps] [send]]]
  [[receive 2 3 4] [ for i := 1 to 10000
    [[write i]
    [[receive 6-2] [strip := [funcval * delta] [send ]]
    [area := [area + strip]]]] [send 8]]]
  [[receive] [writeln] [send]]]
  [[receive 6] [writeln [The approximate value of area = , area]]
[send]]]
```



```

]
[
  [[receive] [area := 0] [send 6]]
  [[receive 2] [delta := [b - a] / 10000] [send 6]]]
  [[receive 2 3 4] [ for i := 1 to 10000
    [[receive ] funcval := [[a ** 2] + a + 8] [send 6-3]]
    [a := [a + delta]]
  ]
]

```

Note that the for loop is analysed during its allocation. blocks (ii) and (iv) have forward dependence and are allocated to the same processor.

phase 3:

This phase re-orders the work-load of any processor if due to the actions of previous phase a crossed-receive has been generated (not applicable to this example). For example, if in a particular processor load there is the following:

```

[
  some code
  [[receive] a := 2 [send 8]]
  some other code
  [[receive] a := 5 [send 12]]
  [[receive 8] b := a + 3 [send]]
  more code
]

```

Clearly, the second statement must be moved after the third one in order to create the correct flow dependence in the sequential code.

```

[
  some code

```

```

[[receive] a := .2 [send 8]]
some more code
[[receive 8] b := a + 3 [send]]
[[receive] a := 5 [send 12]]
more code
]

```

phase 4:

The final phase of the allocator is to insert the variable names and type for each of the receive lists and also to change the block numbers to corresponding processor numbers. For this example the following will be generated:

```

[
  [[receive] [writeln [Please input the...]] [send]]]
  [[receive] [readln [a , b] [send 2 var a integer, 2 var b integer]]]
  [[receive] [writeln [iteration steps] [send]]]
  [[receive 2 area var integer, 2 delta var integer] [ for i := 1 to
10000
    [[write i]
    [[receive 2 var funcval integer] [strip := [funcval * delta] [send]]
    [[receive] [area := [area + strip]] [send]]
  [[receive] [writeln] [send]]]
  [[receive 6] [writeln [The approximate value of area = , area]]
[send]]]
]
[
  [[receive] [area := 0] [send 1 var area integer]]
  [[receive 1 var a integer, 1 var b integer]
    [delta := [b - a] / 10000] [send 1 var delta integer]]]
  [[receive ] [ for i := 1 to 10000

```

```

[[receive.] funcval := [[a ** 2] + a + 8] [send 1 var funcval
integer]]
    [a := [a + delta]]
]

```

Note that certain variables that are input to a loop body are only communicated once at the initiation of the loop. For example, variables `area` and `delta` are only communicated once from processor 2 to 1 just before the loop body. Also note that variable `a` required by the loop body of processor 2 is not communicated. This is because an earlier statement has already received its latest value as part of its input variable set.

Step 5:

This is the last step that the compiler takes. Here the translator translates each processor load in turn and writes each output to a separate file. Appendix G gives the output of the compiler for each of the processors.

5.3.2 Parallelism Detection for Assignment Statements

The following assignment statement examples were given to the compiler:

Example 1:

Source code

```

a := 2 + 3;
b := a + aconst;
c := a * b - b;

```

Comments:

No parallelism is detected. The translated 6809 assembly code is allocated to node one only.

Required processor cycles

203

Example 2:*Source code*

```

a := 2;
b := 3;
c := a + b;

```

Output:

processor 1	processor 2
a := 2;	b := 3;
receive b2	send b
c := a + b;	

Comments:

Tow parallel paths were detected in the first layer with the second layer only having one instruction. Maximum of two nodes could be used.

Required processor cycles

85

Example 3:*Source code*

```

a := 2 ;
begin
  b := 3;
  c := a + b;
end;

```

Comments:

No parallel paths were detected due to the grouping. The translated code was allocated to node one.

Required processor cycles

94

Example 4:*Source code*

```

1  a := 2;
2  b := 3;
3  c := a + b;
4  a := 3;
5  b := 4;
6  d := a - b;

```

Output:

<u>processor1</u>	<u>processor2</u>	<u>processor3</u>	<u>processor4</u>
a := 2;	b := 3;	a := 3;	b := 4;
receive b2	send b	receive b4	send b
c := a + b;		d := a - b;	

Comments:

First layer of the parallel code includes statements 1,2 , 4, 5. The second layer has two parallel paths, including statements 3, 6. Having more than 4 nodes has no effect on the allocation and parallelism exploitation.

Required processor cycles

95

Example 5:*Source code*

```
a := 2;  
b := 3;  
c := a * b + d;
```

Output:

WARNING: d is referenced before being assigned to.

Comments:

User is warned against the use of variable d, which is not initialized elsewhere before referencing.

Discussion:

- **Single statements**

In the above series of test examples the compiler was able to detect all the parallel paths at any given layer. Communications of variables were consistent with the criteria of allocation according to the number of nodes in the system. The maximum number of parallel paths were observed to be detected, while the number of available processors in the system was only effective in the allocation phase. During parallelism detection, one of the byproducts was the identification of statements that reference uninitialized variables. Appropriate warnings were given when such statements appeared in the program. These statements could not be allocated until corrective measures were taken by the user. Uninitialized variables in programs can give unpredictable results on sequential machines, while their effect in parallel systems can be more drastic.

- **Blocks**

An interesting observation of the results was the effect of grouping a number of assignment statements together by means of begin-end primitives. This forced the group to be treated as a single block. It meant that even if one of the statements in the block was dependent on any statement outside, but before the block, then it would prohibit any parallel execution of other statements in the block which may be independent of a statement outside the block. This phenomenon would limit the number of maximum parallel paths which exist in the source code, however, it may help the programmer in certain situations to have some degree of control. In other words, the standard Pascal begin-end primitives work in the same way as the extended parallel Pascal's par-begin and par-end primitives as far as this compiler is concerned.

One of the most important tests performed was to see the effect of the number of processing nodes in the system on the scheduling of a test program. It was observed that if the number of nodes in the system is more than the number of maximum parallel paths in the program then those extra nodes will not be used unless they can be utilized for finer grain parallelism. Reduction of nodes in the system has the effect of reducing the overall communication overhead but increasing the workload of the remaining processing nodes in the system. The reduction in communication overheads due to the decrease of available processing nodes is not linear and in most cases it should be avoided by using the maximum number of nodes in the system. This is due to the fact that communication in MINNIE is based on associative addressing and therefore a single bus broadcast of a data item can reach many destinations at once. However, this is application dependent and fine tuning may be required to determine the optimum number of processors. The best and most accurate way to establish this optimum number is to use a simulator and execute the particular application program on the simulator using the output of this compiler for different node settings. By measuring the execution times and comparing them the optimum configuration could be deduced for any application program.

5.3.3 FOR Loops

Loops are by far the most important section in any program and can lead to a substantial amount of parallelism and reduction in execution time. The following tests were carried out to show that parallelism in For loop constructs would be automatically detected by the compiler.

Example 1:

Source code

```
readln (n);  
for i := 1 to n do  
    a := rand(i) + i * 2;
```

Comments:

The upper bound of *i*, the loop control variable, is unknown at compile time therefore no parallelism was detected and ordinary sequential code generated for node 1.

Required processor cycles

31 + 83 per iteration.

Example 2:

Source code

```
for i := 1 to 10 do do  
    a := rand(i) + i * 2;
```

Comments:

The bounds of variable *i* were known at compile time. A maximum of ten processors can participate in the parallel execution of loop iterations. Table 5.2 gives the iteration space of each node according to the number of processors available.

Required processor cycles

31 + 83 per iteration. For a 10 node system the total processor cycle would be 114.

number of processors	iteration space allocated to each node
1	node one doing all 10 iterations
2	nodes 1 and 2 each doing 5 iterations
3	nodes 1 and 2 doing 3 and node 3 doing 4
.	.
.	.
.	.
10	each node doing one iteration of the loop.

Table 5.2: Allocation of iterations of a parallelised for loop in a multi-node system

Example 3:*Source code*

```

for i := 1 to n do
begin
  a := a + i;
  b := b - i;
end;

```

Output:

processor1	processor2
for i := 1 to n do	for i := 1 to n do
a := a + i;	b := b - i;

Comments:

The value of n is unknown at compile time, but there exists two parallel paths in the loop body and no forward dependence. The value of n is communicated to either of the processors 1 and 2 or one of them as the case may be.

Required processor cycles

31 + 142 per iteration. In the parallel form, 31 + 86 per iteration.

Example 4:*Source code*

```

for i := 1 to n do
begin
  c := c - i;      1
  a := b + i;     2
  b := a /c;      3
end;
```

Output:

processor1	processor2
for i := 1 to n do	for i := 1 to n do
begin	begin
a := b + i;	c := c - i;
receive c	send c
b := a /c;	
end;	end;

Comments:

Two parallel paths exist in the first layer of the loop body. Forward dependence exists between statement 2 and 3. Two processing nodes can be effectively used. Forward dependence forces the allocator to keep statement 2 and 3 on the same node.

Required processor cycles

31 + 195 per iteration. In the parallel form, 31 + 158 per iteration.

Example 5:*Source code*

```
for i := 1 to n do
begin
  a := a + i;
  b := b - i;
  if a + b > 8 then goto label
end;
label:
```

Comments:

Loop body contains transfer of control statement therefore no parallelism detection is carried out and sequential code was generated.

Discussion:

The above tests were designed to show that the compiler is capable of detecting parallelism inside loop bodies and utilizing them effectively. At a higher level the compiler detects conditions where a loop can be executed in parallel with other loops and blocks, see section 5.3.4. The tests showed the ease with which the Petri net modelling technique detected parallelism both inside and outside loop constructs. Two types of parallelism were exploited by the compiler and are shown by the sample tests (examples 2, 3). Another test (example 4) shows how forward dependencies are treated by the allocator part of the compiler and how it eliminated the extra communication overhead.

Limitations of "For" Loop Tests

Limitations of the hardware and its time of completion prevented a full test of the actual codes generated by the compiler. These limitations are particularly evident when considering a piece of code containing loop constructs or any piece of re-entrant code.

In the present form of the machine, each node has four input (receive) registers and one output (send) register. The source address of the first four data items to be received by a node are loaded in their respective receive address registers at load time. There are two main limitations which hindered a proper test of "For" loop constructs:

- The first limitation is that if a node is to receive two or more data items from another node then when the first item is sent by the source, two or more copies of the data are taken by the receiving node's interface. The receive action generates an interrupt which activates a service routine. The routine clears the flags of registers that have just received a data item to indicate the end of their functionality; that is, when the second data item is broadcast by the sender the receiving node will not copy it as it assumes that it has already been received. This would lead to loss of some data in these situations. The solution is either to change the function of the interrupt service routine or at run-time to re-write the source address in the receive address register after the first data has been received.

With the first suggested solution, the interrupt service routine would avoid changing the flags and hence on both occasions two copies of the data will be taken by the interface of the receiving end. Using two registers for receiving two separate data items from the same node then becomes unnecessary provided there is a receive register for each node of the system and the interrupt service routine adds each received data to the end of a queue depending on the source address. This will eliminate the need for writing the source of data items to registers at run time. As a result, a node expecting data from another node will try to read the data from the head of the queue associated with the source node. An empty queue will indicate that the data has not arrived yet and leads to idling. The processing will resume when the data is in the queue, ready for consumption.

In the second suggested solution, the user program environment on each node

must set the receive flags of all its registers receiving a copy of the same data item to false (data not received) after being set to true by the communication interface. In order for the user program to be aware of the actual received data the interrupt routine must copy the received data to a pre-defined location prior to resetting the receive register flags to false. This will ensure that arrival of the subsequent set of data will not corrupt the previous data before being used.

- The second limitation in the present form of the hardware manifests itself when a node needs to receive more than four pieces of data. By loading the receive address registers at run-time it is possible to receive more than four data items. Consider the case where a data item has to be received by a particular node after the execution of a loop body. Before the node is able to receive the data one of its receive address registers must be loaded with the data source address. The execution time of the loop body may be dependent on the runtime environment. Thus, the sending node may get to a position that it is able to send the data but the receiving end has not been able to load its receive address register with the correct data source address. The data will be lost in this case. There are methods that try to anticipate the execution time of various pieces of code and introduce delays in order to synchronize the exchange of data between sending and receiving nodes to avoid data loss. Although these methods reduce the possibility of such data losses, they are not hundred percent accurate as a lot depends on the runtime environment and the exact input values. In order to guarantee repeatable and predictable results data communications must be made independent of the run-time environment. This necessitates modifications to the hardware. As seen above, the best solution is to have $(n - 1)$ receive registers per node and a data queue per receive register, where n is the number of nodes in the system.

5.3.4 Mixed Assignment and “For” Loop Statements

The following programs containing a mixture of assignment and For loop constructs were submitted to the compiler:

Example 1:

Source code

```

a := 2 + a;
b := b - 2;
for i := 1 to 10 do
  c := c + i;
d := a + b + c;
r := d * d / b * 2;

```

Output:

processor1	processor2	processor3
a := 2 + a;	b := b -2;	for i := 1 to 10 do
receive b	send b	c := c + i;
receive c		send c;
d := a + b + c;		
r := d * d/b * 2;		

Comments:

Parallelism detection generated three layers of parallelism. First layer had three parallel paths; the first layer including the for loop. The loop itself did not offer any parallelism inside its body due to the dependence of iterations on each other. Layers two and three had one statement each, indicating sequentiality.

Required processor cycles

1230 in the sequential mode. For the parallel code 1064.

Example 2:*Source code*

```

a := a + 2;
b := b - 2;
for i := 1 to n do
begin
  c := i * b;
  d := d / i;
  e := c + d + e;
end;
f := e * 2 - a;

```

Output:

processor1	processor2
a := a + 2;	b := b - 2;
receive b	send b
for i := 1 to n do	for i := 1 to n do
begin	begin
c := i * b;	d := d / i;
receive d	send d
e := c + d + e;	
end;	end;
f := e * 2 - a;	

Comments:

First layer has two parallel paths, statement one and two. Second layer holds the for loop. The loop body itself was parallelized as shown in the output below. Third layer has no parallel paths and contains only statement four.

Required processor cycles

244 + 240 per iteration. For the parallel code 208 + 210 per iteration.

Discussion:

The main aim here was to show that the compiler could cope with a mixture of statements. It had to detect the parallelism in the overall main body, as well as within the loop bodies. The results of the experiment showed that it achieves this goal and works as expected. Individual constructs were tested in the previous sections and these tests confirmed that no side effect is created by using a combination of pascal language constructs.

5.4 Fine Grain Parallelism Detection Test

Specification for this part of the compiler was laid down in the previous chapter. Here the tests were simply to confirm that those specifications were met and the detection and allocation strategies were adhered to.

Below are some of the test programs which were used to demonstrate that fine grain parallelism was correctly utilized by the compiler:

Example 1:*Source code*

```
a := 2 + 3 + 4 + 5 + 6;
b := 7 * 8 / 9;
c := 7 * 8 - 9 / 3;
```

Output:

processor1	processor2	processor3	processor4
a := 2 + 3 + 4 + 5 + 6;	b := 7 * 8 / 9;	7 * 8	9 / 3
		receive #	send #
		c := # - #;	

Comments:

There are three parallel paths at statement level. Sufficient number of processors exist for using fine grain parallelism. The first expression can offer low level parallelism only by user assistance and the second expression is simply unsuitable for this purpose. However, the third expression was marked as suitable for expression level parallelism and the correct communication code was generated by the compiler for communicating the result of sub-expressions between nodes.

Example 2:

Source code

```
a := 2;
b := 3 * 4 + 20 / 5 + 3;
c := a + b;
a := 3;
b := 4;
d := a - b;
```

Output:

processor1	processor2	processor3	processor4
a := 2;	b := 3 * 4 + 20 / 5 + 3;	a := 3;	b := 4;
receive b2	send b	receive b4	send b
c := a + b;		d := a - b;	

Comments:

As the number of processors specified (4) was not more than the number of parallel paths found at the statement level therefore fine grain parallelism was not utilized by the compiler.

Example 3:

Source code

```
a := (2 + 3) + (4 + 5) + 6;
```

```
b := 7 * 8 / 9;
```

Output:

processor1	processor2	processor3
2 + 3	b := 7 * 8 / 9;	4 + 5
receive #		send #
# + #		
a := # + 6		

Comments:

The first expression which was found unsuitable for fine grain parallelism previously offered fine grain parallelism after using some brackets to regroup sub-expressions together.

Example 4:

Source code

```
for i := 1 to 10 do
begin
  a := a * i + a ** i;
  writeln ("a = ", a);
end;
```

Output:

processor1	processor2
<pre> for i := 1 to 10 do begin send a a * i receive # a := # + #; writeln("a = ", a); end; </pre>	<pre> for i := 1 to 10 do begin receive a a ** i send # end; </pre>

Comments:

Processor 2 is exclusively assigned in performing fine grain computation for node 1. Value of variable a is communicated before each iteration of the loop.

Discussions:

There were four critical areas that needed to be fully tested:

1. Statements that do not offer fine grain parallelism must be marked as unsuitable for parallel execution and thus translated correctly for a single node computation. Conversely, suitable expressions must benefit from fine grain parallelism. If low level parallelism is used then the compiler generated communication code must be correct (example 1).
2. Expression level parallelism must be ignored if there are insufficient number of processors to meet the number of parallel paths in the medium grain parallelism (example 2).
3. Re-arranging certain expressions must increase the potential for exploiting fine grain parallelism (example 3).
4. If possible, suitable expressions inside loop bodies must benefit from fine grain parallelism. The result of each sub-expression must be correctly communicated for each iteration of the loop (example 4).

The concept of co-processors for fine grain parallelism exploitation was discussed earlier. The tests showed that only if there are more processors in the system than the maximum number of parallel paths in the source code the compiler tries to exploit finer grain parallelism.

The longest branch of any expression tree was allocated to one of the nodes that participated in the coarse grain parallel execution of the program. The rest of the tree (sub-expression) was further divided for allocation on the nodes assigned for fine grain parallelism according to the same longest branch criteria. This part of the fine grain parallelism detector and allocator was proved to be acting correctly and no problems were observed.

Experiments showed that by moving operators with higher priority to the front of the expression, more of the fine grain parallelism inherent in the expression could be detected and used. This is simply because of the way trees are constructed by the compiler. This calls for some re-arrangement of the large expressions by a user to maximize fine grain parallelism.

Careful study of the generated code showed that the communication between sub-expressions were correct and consistent with the algorithm presented in chapter 3. Decomposition of expressions ensured minimum amount of communication overhead relative to the amount of parallelism gained.

Detection of fine grain parallelism for large expressions inside a loop body was also experimented by using the compiler. It duplicated the control structure of the loop for the evaluation of the subexpressions on the coprocessors. This ensures that every iteration of the loop receives the result of a subexpression evaluated for that iteration from a coprocessor participating in the evaluation of that expression. The generated communication codes corresponded with the theoretical expectations. A point to be born in mind is the effect of using fine grain parallelism in loop bodies. It can lead to delays in evaluating other subexpressions whose result is required by other nodes than those executing a loop construct due to the number of iterations

in the loop. Unless a system has a very large number of processing nodes to allow a very thin distribution of fine grain parallelism it is advisable not to use fine grain parallelism inside a loop body.

5.5 Benchmark

For benchmarking this compiler, in its sequential mode, a number of test programs were used. The same tests were carried out on another 8 bit machine (BBC) using its ISO Pascal compiler. The tests were designed to gauge the following additional criteria by which a compiler can be appraised. There was no access to a commercially available compiler for Pascal for micros based on 6809 processor to conduct a more accurate comparison and basically to compare like with like.

- *Automatic parallelism detection:*

The ISO Pascal compiler does not offer any parallelism detection of the source program.

- *Compactness of the generated code:*

The benchmark results compared very closely in favour of the ISO pascal compiler. This could be due to different clock rates and processor types and the quality of the generated code although there is no way of proving this.

- *Quality of error detection and messages:*

All the errors detected by the ISO Pascal compiler were detected by this prototype compiler too. Of course the tests were not exhaustive and more comprehensive tests are required.

- *Availability of optimisation facilities:*

A reachability study for elimination of *dead code* is carried out by this compiler. There was not enough data on any optimization that the ISO Pascal compiler may carry out.

- *Warning on uninitialized variables:*

If a variable is not initialized by the user program before referencing it the parallel compiler then issues a warning message. That is, the compiler does not initialize the variables to zero at run-time.

- *Run-time support:*

The ISO Pascal compiler has a more comprehensive run-time support and a full implementation of the Pascal run-time library. There is a difficulty in providing run-time support in distributed parallel systems particularly if there is no underlying operating system available.

- *Debugging facility:*

A simulator for a four node system was written to help in the testing of the compiler and for using it as a base for a debugging tool in the future. The ISO Pascal compiler did not offer any debugging facility. The Parallel compiler had built in switches which would allow a user to examine the parallelism detection analysis and to see the grouping of parallel codes during compilation. This could be quite a valuable tool in verifying the compiler's assessment of the source code. This tool was developed during compiler's implementation for the compiler's own debugging.

- *Portability of source code:*

As the implemented language is standard Pascal without any extensions at language or run-time library level therefore the code is portable.

- *Compilation speed:*

The ISO Pascal compiler was slower in compiling all test programs when the parallelizing compiler was used in its single node configuration. When the parallelizing mode was selected the ISO Pascal compiler was considerably faster.

The major aim of parallel processing is to gain execution speed at run time. Speed of execution in parallel processing systems not only depends on how well a parallelising

compiler can detect the inherent parallelism in the source code, but also on the sequential program of the particular application and the speed of the communication system used for data transfer. Therefore, the best way to test the performance of the compiler is to compare the execution time using one node of the system and then using the maximum number of nodes that compiler and the application program can effectively use. The gain is not a true representation of the compiler performance as the efficiency of the hardware is very influential. The two measured times can give a ratio for speed up, which is application dependent. This ratio can then be compared with results from other parallel processing systems running the same application program.

Two problems were faced in carrying out these comparisons. Firstly, there is not any commercially available Pascal parallelizing compiler for comparison purposes. Secondly, due to limitations of the machine and the time by which it was developed it was not possible to carry out any realistic test using the actual hardware. These limitations are fully documented in section 5.3.3.

An application program is statically configured by the compiler to use a predefined set of processors. Therefore, to establish the effect of using extra processors on the execution time of an application it is essential to recompile the sources for any particular hardware configuration. It is anticipated that the execution time decreases almost linearly at first, by using more processors, and then to tail off upon reaching a maximum *depending* on the application. The implication is that in some cases this maximum might be 1. That is, no execution speed up can be achieved by having more than one available processor. The fine tuning only requires a simple recompilation and execution of the user program on different number of processors to determine the most cost effective hardware set up.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

A compiler with parallelism detection features was devised to accept sequential programs and generate object codes for parallel machines. A Petri net of the source code is constructed automatically and then executed to reveal all parallel paths. This results in a multi-layered internal representation which can be equally easily mapped onto a data flow or MIMD architectures. A scheduler generates inter-node communication instructions for executing the multiple object codes on a parallel machine. Other features devised during the design and implementation of the compiler included table-driven analysers easily adaptable to cope with different source languages, a new algorithm for manipulating symbol tables and the introduction of the co-processor concept for implementing fine grain parallelism.

1. The use of Petri net in modelling sequential programs and consequently detecting parallelism in the sequential code proved to be a success. The Petri net model generated by the compiler is executed to detect parallelism in the software. The algorithm for executing the generated Petri net was easily im-

plemented. As a by-product it enabled the compiler to detect any reference to uninitialised variables in the user program. Previously, Petri nets have extensively been used for the design of parallel systems but not for the extraction of parallelism from existing sequential software. In addition to showing the flow of data in a program Petri nets also model the flow of control in a program quite easily. The resulting model can be mapped onto data flow or MIMD architectures.

2. The compiler creates a multi-layered parallel version of the original sequential code. Communications are only present between adjacent layers and all blocks within a layer are totally independent of each other and can therefore be executed in parallel. This internal representation of a program is completely independent of the allocation scheme used for MIMD architectures. All inter-layer communications consist of pure data flow between the layers. This has the advantage that the representation mimics the data flow between the levels of an equivalent data flow graph. Thus, by using an interface the output of the compiler can be mapped to a particular data flow architecture; the interface providing the required cells and structures.
3. Fine grain parallelism detection was based on using the longest branch of an expression tree. The result of this method is a reduced communication overhead. The compiler only attempts to use fine grain parallelism if there are more processors in the system than parallel paths at the medium grain level.

In order to reduce the possibility of communication delays while using finer grain parallelism the concept of co-processors was introduced. Theoretical analysis showed that only systems with large numbers of processors are suitable for exploiting fine grain parallelism. However, a special mechanism is needed to overcome the large overheads of inter-node communications, which is available, in this case (MINNIE), through the associative addressing technique.

4. Parallelizing and scheduling a user program results in the generation of several, one per node, high level intermediate codes. These codes are then translated to suitable assembly codes using relevant translators. This modular approach facilitates processor independent analysis of the source code and allows the mixing of different processors in a particular system configuration. One benefit is that more specialised processors can easily co-exist with other types of processors in a system.

The number of available processors is a user supplied parameter. An application program is statically configured at compile time for the number of available processors in the system. Re-configuration for a different hardware set up involves a simple re-compilation only.

5. For scheduling, the allocator uses the internal representation of the application program. It generates workfiles for each node to include the code and communication requirements of a particular node, both of which are independent of processor type. These workfiles are suitable for mapping to any distributed memory MIMD machine. The problem of multiple assignment is normally dealt with by renaming the variables which has its own weakness as it prevents the full exploitation of the parallelism in source programs. In this thesis, the combination of algorithms for parallelism detection and scheduling eliminate the multiple assignment problem directly and hence the disadvantages associated with variable renaming do not arise.

The allocator proved to be a complex piece of software to implement. It has to cope with alteration in the communication structure of a parallelized program. The ability to modify communication structures ensured that optimum results could be achieved while scheduling for MINNIE. Load balancing is achieved by keeping a ratio between computational and communication load of any given processor.

6. Storage of variables was based on the commonly used stack frame method; each node has its own stack frame. A common problem in distributed memory

parallel systems is the difficulty of using recursion. With the distributed stack frame method used recursive constructs in a parallel mode are possible.

7. Although the compiler can detect the inherent parallelism in a piece of sequential code, suitable algorithmic changes in the program or fine tuning by the user can maximise parallelism utilization, i.e., the amount of parallelism discovered by the compiler is application dependent. Clearly, programs designed with parallelism in mind have inherently more detectable parallelism.
8. During the development process a new algorithm for storage and management of symbol tables was devised. The method is applicable to languages with dynamic variable scoping, like Pascal, and less sophisticated languages like Fortran. The technique is based on sparse dynamic tables which is more efficient in memory requirements compared to other methods. It is also very fast in storing or retrieving information relative to other techniques. The table is an abstract data type and the fact that it is a sparse table is transparent to the programmer. Special high level routines were provided for constructing and manipulating this table.

6.2 Possible Future Work

There are several areas in software and architecture which are amenable to further research and development.

Software Enhancements

Enhancements are seen to fall into two groups: those that extend the concepts explored in this thesis (1& 2) and those (3-6) that are specifically applicable to the machine used to test these concepts (MINNIE).

1. Integrated software development system

For parallelism detection the compiler creates a Petri net model of an existing program and then executes the net to extract the parallel paths. This is acceptable for recompilation of existing sequential software for parallel machines.

Development of new parallel programs can also be based on the use of Petri nets as a vehicle for software design. In a similar way to Yourdon and Jackson packages it should be possible to generate code automatically by suitable software development tools based on Petri nets. Such an integrated design tool would relieve the compiler from generating the Petri net model of the program where this is created by the user himself at the design stage.

2. General Purpose Compiler

The concept of *general purpose compilers* was put forward. The idea was appealing since it could drastically reduce the production cost and the development time of compilers for different languages. Subsequently, the front-end of the compiler has been made table-driven to create the right framework for any future work in this direction.

The existing compiler will not be fully general purpose until all its functional units are table-driven. Further work is required to implement a table driven back-end and the right interface between the front and the back-end of the compiler.

For completeness, research is needed to devise a state automaton for the construction and execution of Petri nets for sequential programs. This state machine must construct parallel paths of the input program in the same way as performed by the existing net constructor and executor. Having achieved these aims the compiler would be fully general purpose. Experiments are then needed to test the ease by which the general purpose compiler could be converted for other input languages. The other improvement in this area can be the addition of an interface to YACC so that the table generated by YACC

can be directly used. Alternatively, a parse table generator routine could be added to the compiler to increase its independence.

3. Distributed operating system

A vital line of work is the design and development of an operating system for MINNIE. Without a suitable operating system it is hard to imagine that MINNIE could ever become a general purpose parallel machine. The operating system could provide the basis for writing fault tolerant software systems, dynamic scheduling, load balancing and multiple user process environments.

4. Debugging tools

Debugging of software running on even sequential machines is tedious and difficult. Debugging of software running on a distributed system is much more difficult. Work on the development of a debugger for applications intended to run on MINNIE is essential.

5. Communication library

The basic inter processor communication provided by the hardware is based on a single byte of data. Library routines are required which use this low level byte based communication to provide higher level communication primitives. The communication library must provide routines for the transfer of words (4 bytes) and messages between nodes. The design of the communication protocols and the implementation of the library routines need careful consideration.

6. SIMD applications for MINNIE

Due to the special nature of MINNIE's communication system, SIMD applications seem to be very suitable. In particular, all the nodes could be loaded with a copy of the program at the same time. The master program could broadcast work packets over the bus for processing by any of the processors.

Hardware Enhancements

1. More input registers

There is a major limitation in the present form of the hardware (see previous chapter for details). This limitation needs to be overcome in order to enable users to run meaningful programs. The recommendation is to increase the number of input registers of each node to 63 in a 64 node system. Additionally, the function of the present interrupt service routine needs to be modified and queues used to store the arriving data.

2. Support for pipelined applications

Certain applications require pipelining of data between adjacent processors. Additional hardware support is required to allow neighbouring nodes to pass data to each other without using the data bus. This extension reduces the load of data bus for inter node communications. By using double ported RAMS neighbouring processors can have some shared memory for exchange of data blocks by using block move instructions. This facility would allow large data structures to be communicated between adjacent processors more efficiently than using the bus.

3. Distributed loading

It is envisaged that systems based on MINNIE will have many hundreds if not thousands of processors. If all the nodes were to be loaded by using the data bus then this process might take an unacceptably long time. Research is needed to find ways of using concurrent loading of the image files in each node using special hardware support.

Bibliography

- [Aho 84] A. V. Aho and J. D. Ullman
Principles of compiler design
Addison Wesley 1984
- [Al-Dabass 86] D. Al-Dabass, N. Nakhaee
A general purpose syntax analyser in POP-11
Department of computing, Trent Polytechnic, internal report 2
- [Al-Dabass 86] D. Al-Dabass, N. Nakhaee
A general purpose lexical analyser in POP-11
Department of computing, Trent Polytechnic, internal report 1
- [Al-Dabass 87] D. Al-Dabass, N. Nakhaee
A Petri net approach to parallelism detection
Department of computing, Trent Polytechnic, internal report 5
- [Allsopp 86] D. Allsopp
Compilation techniques for Nottingham MUSE
PhD thesis 1986 University of Nottingham
- [Amamyia 86] M. Amamyia, M. Takesue et al.
Implementaion and evaluation of a list-processing oriented data flow machine
in proc. of the 13th ann. symp. on computer architecture, Tokyo
June 2-5, ACM NY pp. 10-19, 1986

- [Arvind 80] Arvind
Decomposing a program for multiple processor systems
Proc. of the int. conf. on parallel processing, Aug 1980, pp 7-14
- [Ashcroft 77] E. A. Ashcroft, W. W. Wadge
Lucid, a non procedural language with iteration
Com. ACM July 1977
- [Ashcroft 85] E. A. Ashcroft, W. W. Wadge
Lucid, the data flow programming language
Academic Press 1985
- [Baer 73] J.L. Baer
A survey of some theoretical aspects of multi-processing
Computing Surveys, vol. 5, no. 1, March 1973
- [Banerjee 76] U. Banerjee
Data dependence in ordinary programs
M. Sc. thesis, University of Illinois, Urbana Champaign 1976.
- [Barrett 85] N.K. Barrett
S/W tools and a simulation system for the Nottingham multi-stream D/F project
University of Nottingham, UK, PhD Thesis, 1985
- [Bird 85] P. L. Bird
Linear time detection of inherent parallelism in sequential programs
Computer Physics Comm., vol. 37, North-Holland, Amsterdam 1985
- [Brailsford 85] D. Brailsford, R. J. Duckworth
The MUSE machin, An architecture for structured data flow computation
New generation computing, vol. 3, no. 2, pp 811-195, 1985

- [Brinch 75] P. Brinch Hansen
The programming language concurrent pascal
IEEE Transactions on the SW Eng. June 1975
- [Brown 62] G. W. Brown
A new concept in programming
MIT Press 1962
- [Burke 86] Michael Burke and Ron Cytron
Interprocedural dependence analysis and parallelization
proc. of the ACM, 1986 pp 162-175
- [Bush 79] V. J. Bush and J. R. Gurd
A data flow implementation of Lucid
University of Manchester Oct. 1979
- [Caluwaerts 82] L. J. Caluwaerts
*Implementing code reenterancy in functional programs on a data
flow computer system with a paged memory*
in the int. workshop on high-level language and computer archi-
tectures, Dec. 1982
- [Chambers 84] F. B. Chambers, D. A. Duce and G. P. Jones
Distributed computing
Academic Press 1984
- [Clark 86] K. Clark, S. Greogry
Parlog: parallel programming in logic
ACM tran. on prog. lang. and sys. Jan 1986
- [Cornish 79] M. Cornish
*The TI data flow architectures: The power of concurrency for
avionics*
In proc. of the third conf. on digital avionics sys. Nov. 1979, pp.
19-25

- [Cytron 78] R.G. Cytron
Compile time scheduling and optimization for asynchronous machines
computing March 78
- [Darlington 81] J. Darlington, M.Reeve
ALICE a multiple processor reduction machine for the parallel evaluation of applicative languages
proc. ACM conf. on functional languages and computer architectures 1981
- [Dennis 79] J. B. Dennis and W. B. Ackerman
Val: A value oriented language - preliminary reference manual
tech. report TR-218, computation structure group, MIT Cambridge, Mass. June 1979
- [Desrochers 86] George R. Desrochers
Principles of parallel and multi-processing
Addison-wesely 1986
- [Duckworth 84] R. J. Duckworth
Parallel computation on a multi-stream data flow machine
PhD thesis 1984 University of Nottingham
- [Ericson 87] S. Ericson Zenith
Occam 2, aspects of the language and its development
Parallel Processing Specialist Group (PPSG) newsletter, no.2
Jan. 1987
- [Fang 87] Zhixi Fang, Pen-Chung Yew
Dynamic processor self-scheduling for general parallel nested loops
proc. of the international conf. on parallel processing, Aug. 1987

- [Flaming 86] B. L. Flaming
A compiler for Lucid on sequential machines
IEE fifth annual int. Phonix conf. on computers and comm. 1986
pp. 350-354
- [Flynn 66] M.J. Flynn
Very high speed computing systems
proc. of the IEEE Dec. 1966
- [Fortes 86] J. A. B. Fortes
Parallelism detection and transformation techniques useful for VLSI algorithms
Journal of Parallel and Distributed Computing, Aug 85 vol 2, pp 277-301
- [Gonzalez 71] M.J. Gonzalez JR, C.V. Ramamoortly
Program suitability for parallel processing IEEE trans. on computers vol. c-20, no. 6, June 1971
- [Graham 80] Susan L. Graham
Table-Driven Code Generation IEEE trans. on computers August 1980, pp. 25-34
- [Gurd 80] J. R. Gurd and I. Watson
A data driven system for high speed parallel computing
computer design, vol. 9 no. 6 and 7, June 1980, pp. 91-100,97-106
- [Gurd 84] J. Gurd. Edited by F.B. Chambers, D.A. Duce and G.P. Jones
Fundamentals of data flow
1984
- [Gurd 85] J. R. Gurd, C. C. Kirkham and I. Watson
The Manchester prototype data flow computer
comm. of the ACM, vol. 28 no. 1, Jan 1985, pp. 34-52

- [Hammes 89] Mark Hammes
A node interface for parallel processing
PhD thesis, Nottigham Polytechnic, England 1989.
- [Hiraki] K. Hiraki, K. Nishida etal.
Maintenance architecture and its LSI implementation of a data flow computer with a large number of processors
In proc. of int. conf. on parallel processing. IEEE NY pp. 584-591
- [Ibbett 82] Roland N. Ibbett
The architecture of high performance computers
The MacMillan Press limited 1982
- [Ito 85] N. Ito, M. Kishi etal.
The data-flow based parallel inference machine to support two basic languages in KL-1
in IFIP TC-10 working conf. on fifth generation computer architectures, Manchester July 15-18 85, pp. 123-145
- [Johnson 86] Stephen C. Johnson
YACC: Yet Another Compiler-Compiler
Vax Unix manual 86
- [Kluzmiak 86] F. Kluzmiak
Prolog for programmers
Academic Press 1986
- [Kuck 72] D. J. Kuck, Y. Muraoka and Shigh-Cing Chen
On the number of operations simultaneously executable in Fortran-like programs and their resulting speed up
IEE trans. on computers Dec. 1972
- [Kuck 82] D. J. Kuck, edited by D. J. Evans
High speed machines and their compilers
parallel processing systems, Cambridge University Press, 1982

- [Kuck 86] D.J. Kuck, D.H. Lawrie et al.
Parallel super computing today and the Cedar approach
American ass. for the advancement of sci. 1986
- [Lackey 86] S. Lackey, D. Peak
A parallel processing computer with hardware based concurrency control
Alliant computer systems, Acton, MA, USA 1986
- [Laventhol 87] J. Laventhol
Programming in POP-11
BSP professional books 1987
- [Liskov 77] B. Liskov, A. Snyder and R. Atkinson et al.
Abstraction mechanisms in CLU comm. ACM vol. 20 part 8, pp. 564-576 Aug. 1977
- [Maekawa 76] Mamoru Maekawa
Detection of parallelism between statements by decomposing into separate sequential processes
int. journal comp. & inf. science, vol. 5 no. 5, Sep. 1976 pp. 239-255
- [ref. manu. 84] Language reference manual
SISAL: streams and interaction in a single assignment language
M146 University of Manchester, Aug. 1984
- [McGraw 83] J. R. McGraw and S. K. Skedizielewski
Streams and iterations in VAL: Additions to a data flow language
proc. of int. conf. on distributed computing systems, March 1983.
- [McGraw 83] J. McGraw, S. Skedjielewski and S. Allan et al.
SISAL, stream and iterations in a single assignment language
Lawrence Livermore National Lab. University of California July 1983

- [Meakawa 76] Mamoru Meakawa
Detection of parallelism between statements by decomposing into separate sequential processes
int. conf. of comp. and info. sci. vol. 5 no. 3, 1976
- [Miller 73] R. E. Miller
A comparison of some theoretical models of parallel computation
IEEE tran. on comp. vol. c-22, Aug. 73
- [Nakhaee 85] N. Nakhaee
An interpreter for the programming language B
M. Sc. thesis, Dep. of computing, University of technology,
Loughborough, 1985
- [Norman 83] D. A. Norman
On process models of sentence analysis
W. H. Freeman and company 1983
- [Perrott 86] R. H. Perrott
Language structures for parallel processing
conf. proc., Major Advances in Parallel Processing, 9-11 Dec. 86,
pp 217-26
- [Peterson 77] J. L. Peterson
Petri nets
computing surveys, vol 9, no. 3, 1977
- [Peterson 79] J. L. Peterson
Petri net theory and the modelling of systems
North-Holland publications 1979
- [Presberg 75] David L. Presberg and Neil W. Johnson
The PARALYZER: IVTRAN's parallelism analyzer and synthesizer

- sigplan notices, programming languages and compilers for parallel and vector machines, March 1975, vol. 10 part 3, pp 9-16
- [Russel 69] E. Jr. Russel
Automatic program analysis
PhD thesis, Dep. of elec. eng., University of California, 1969
- [Sarkar 86] Vivek Sarkar and John Hennessy
Compile-time partitioning and scheduling of parallel programs
Proc. of the ACM 1986, pp 17-26
- [Sarkar 88] Vivek Sarkar, S. Skedzielewski and Patrick Miller
An automatically partitioning compiler for SISAL
Proc. of CONPAR 88, Sep. 1988 UMIST Manchester, pp 26-33
- [Schneck 72] Paul B. Schneck
Automatic recognition of vector and parallel operations in a higher level language
In proc. of the ACM 25th annual conf. 1972, pp 772-79
- [Shadbolt 87] Shadbolt and M. Burton
PoP-11 programming for artificial intelligence
Addison Wesley 1987
- [Sloman 86] A. Sloman, A. Ramsay and R. Barrett
POP-11 a practical language for artificial intelligence
Ellis Horwood 1986
- [Srini 86] Vason P. Srini
An architectural comparison of data flow systems
Computers, March 1986
- [Storer 91] Ralph Storer
A practical program development using JSP
Blackwell Scientific Publications, 1991

- [Tesler 86] L. J. Tesler and H. J. Enea
A language design for concurrent processes
proc. AFIP 32, 1986, pp. 403-408
- [Tomasulo 67] R. M. Tomasulo
An efficient algorithm for exploiting multiple algorithmic units
IBM jou. of research and development, Jan 67, pp. 25-33
- [Treleaven 79] P.C. Treleaven
Exploiting program concurrency in computing systems
Computer 1979
- [Triolet 86] Remi Triolet
Direct parallelization of CALL statements
University of Illinois, Center for super computing research 1986
- [Varadharajan 88] V. Varadharajan and K. D. Baker
Directed graph based representation for software system design
Software Eng. journal, Jan. 1988, pp. 21-28
- [Veen 86] A. H. Veen
Data Flow machine architecture
ACM computing surveys, vol. 18, no. 4 Dec. 1986
- [Velder 85] Rex Velder, M. Campbell, G. Tucker
The Hughes data flow multiprocessor
proc. of the 5th int. conf. on dist. computing systems, 1985, pp. 2-9
- [Volansky 70] S.A. Volansky
Graph Model Analysis and Implementation of Computational Sequences
University of California, Los Angeles, USA, PhD Thesis 1970

[Wolfe 82]

M. J. Wolfe

Optimizing supercompilers for supercomputers

PhD thesis, University of Illinois, Urbana, USA, 82

Appendix A

Introduction to Petri Nets

A Petri net is an abstract, formal method of modelling systems. It allows a high degree of mathematical formalism to be used in the design of systems in general and those exhibiting parallelism in particular.

A.1 Historical Background

The theory of Petri nets was developed by Carl Adam Petri in 1962. In his Ph. D. thesis, Petri derived a new way for modelling information flow. In addition to having a mathematical representation, the properties of a system could also be illustrated by graphical means when using Petri nets. Both the mathematical and the graphical models could show the relationships between the components of an asynchronous system.

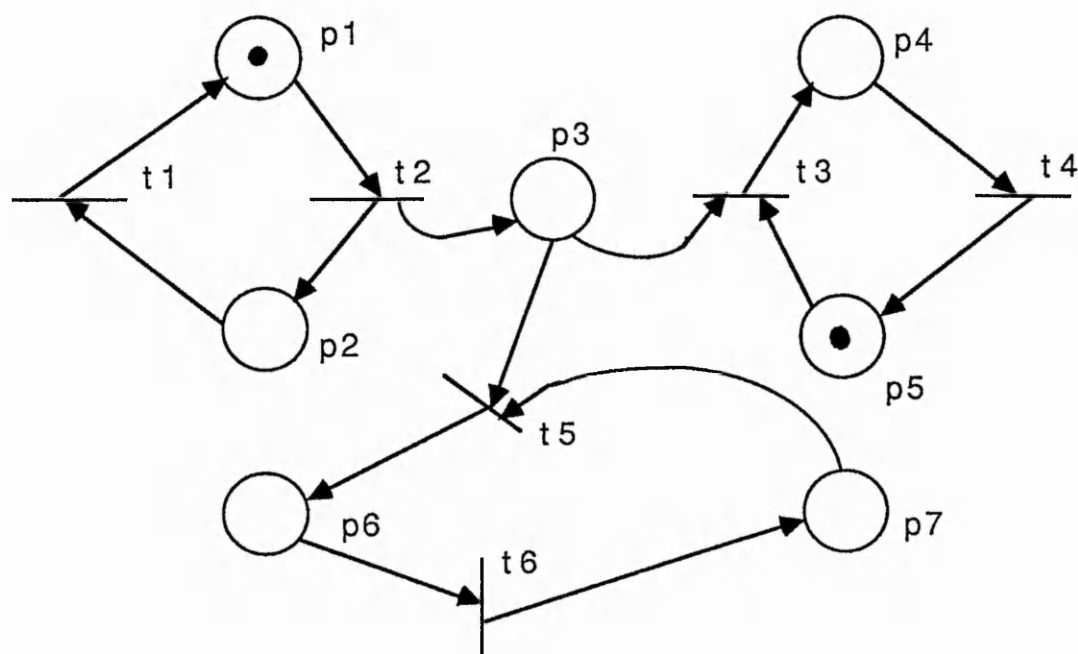
The early works of Petri continued in many directions. Petri himself expanded his original theories on the concepts of information flow, resulting in a form of general systems theory. The original Petri net theory was concerned with the flow of information within a single net only, whereas the general net or system theory is

concerned with the flow of information between nets.

Research in Petri nets has since branched in two directions, pure and applied net theory. The applied net theory is concerned with the applications of Petri nets in the modelling and simulation of systems. It considers ways of generating Petri nets and developing automatic translators for various net representations. Pure net theory, on the other hand, concentrates on the mathematical analysis of nets and aspects close to automata theory. The work here is mainly based on applied Petri net theory rather than on the more complex mathematical approaches to it.

A.2 Petri Net Automata

Petri nets can be represented using two forms of notations: a formal notation and a graphical one. The formal notation describes a Petri net as a mathematical structure, mostly used in theoretical studies. The graphical representation is used for illustrating the basic concepts. Below is a simple Petri net graph:



As seen from the diagram a Petri net graph is made of two types of nodes, places and transitions. Places are represented by *circles* and the transitions with *bars*. Places are connected to transitions with directed arcs and vice-versa.

The graph shows the static properties of the net. In addition, the dynamic properties of the net could be observed by executing the net model. The execution is determined by the black dots or tokens within places (circles). A Petri net with tokens is called a marked Petri net. Tokens move through the net by firing of transitions. In order to fire a transition it must be first enabled. A transition is enabled by having a token corresponding to each of its inputs (places connected to a transition are its input places). A transition fires by removing the tokens from its inputs and depositing tokens into its output places. Notice that for multiple inputs from a place to a transition, or vice versa, there must be at least as many tokens as arcs present in the place. This implies that multiple arcs can exist from a place to a transition and from a transition to a place. The distribution of tokens in a Petri net defines the state of the net and is called its marking.

For the formal representation of Petri nets an extension of set theory called bag theory is used. Mathematically a Petri net is a four tuple (P, T, I, O) where P is the finite set of places, T is the finite set of transitions, I is the input function and finally O is the output function. The above definitions can be shown mathematically as follows:

$$P = \{p_1, p_2, \dots, p_n\} \text{ where } n > 0$$

$$T = \{t_1, t_2, \dots, t_m\} \text{ where } m > 0$$

$$I : T \rightarrow P^\infty \text{ and}$$

$$O : T \rightarrow P^\infty$$

I and O are mapping functions from transitions to bags of places. A Place p_i is an input place of a transition t_j if $p_i \in I(t_j)$ and p_i is an output place if $p_i \in O(t_j)$.

A marking of a Petri net, an assignment of tokens to places, is given by the mapping $\mu : P \rightarrow N$ and can be defined as an n -vector where n is the number of places:

$$\mu = (\mu_1, \mu_2, \dots, \mu_n)$$

Each $\mu_i \in N$ is the number of tokens at place i , that is, $\mu(p_i) = \mu_i$. This implies that a transition t_j is enabled if:

$$\mu(p_i) \geq (p_i, I(t_j))$$

The change in state caused by firing a transition is defined by function δ which when applied to a marking μ and a transition t_j yields a new marking. Thus μ' the new marking can be defined as:

$$\mu' = \delta(\mu, t_j)$$

The next marking μ' is said to be *reachable* from μ . By extending this concept, it is possible to define the set of reachable markings for a given Petri net. The reachability set of a Petri net C with marking μ is defined as $R(C, \mu)$ where R is the set of all markings which are reachable from μ , that is $\mu' \in R(C, \mu)$.

The net shown above can be represented formally as:

$$P = \{p_1, \dots, p_7\}$$

$$T = \{t_1, \dots, t_6\}$$

$$\begin{aligned} I(t_1) &= \{p_2\} & O(t_1) &= \{p_1\} \\ I(t_2) &= \{p_1\} & O(t_2) &= \{p_2, p_3\} \\ I(t_3) &= \{p_3, p_5\} & O(t_3) &= \{p_4\} \\ I(t_4) &= \{p_4\} & O(t_4) &= \{p_5\} \\ I(t_5) &= \{p_3, p_7\} & O(t_5) &= \{p_6\} \\ I(t_6) &= \{p_6\} & O(t_6) &= \{p_7\} \end{aligned}$$

And the marking is:

$$\mu = (1, 0, 0, 0, 1, 0, 0)$$

By executing the net it can be shown that t_5 and t_7 are not reachable while the rest of the net can be executed indefinitely.

Appendix B

The Compiler

Earlier it was argued that it would be cost effective to utilise existing software on parallel processing platforms by means of parallelizing compilers. However, this would require writing compilers for all the major imperative sequential languages, which is in turn expensive and requires many man-years of work. The motivations for the design and realisation of a General Purpose Compiler (GPC) are therefore clear.

It was therefore a design objective to produce the compiler in such a way that in future it could be modified to compile other languages, if necessary. As a result, similar compilers for different sequential imperative languages may be developed quickly. To cater for any future requirements in this direction as many parts of the compiler as possible were made table-driven. The perceived modifications involve changing the tables used by the various components of the compiler to suit the new language. The concept here is different with that of compiler-compilers. In any case, a GPC unlike a compiler-compiler, does not emit code to generate a new compiler, see figure B.1 (a) and (b).

During the course of this research no attempt was made to measure the feasibility/simplicity of modifying the compiler for other languages.

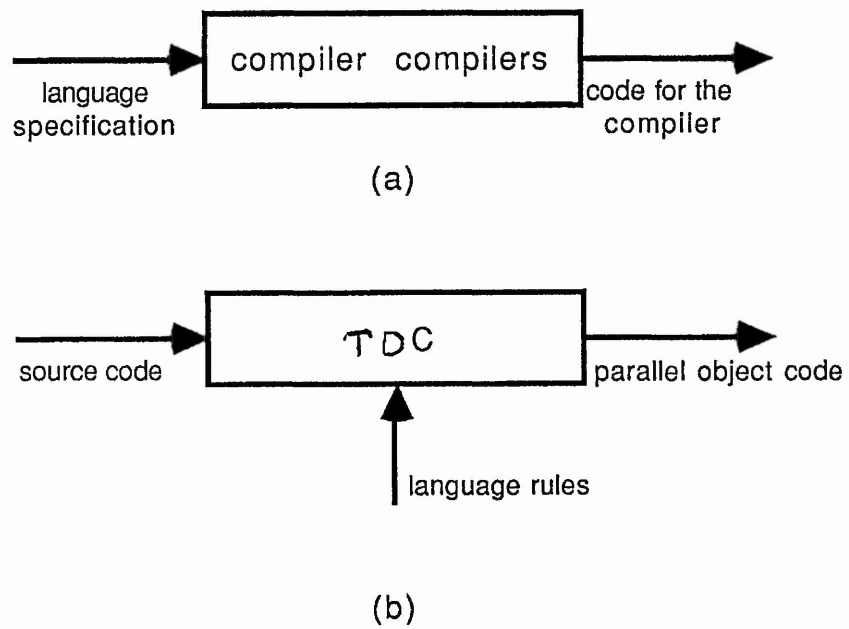


Figure B.1: The difference between a compiler-compiler and a TDC

The lexical and syntax analysis of this compiler are table-driven. The parser produces a parse-tree as it goes through the reduction of syntax rules in the parse table. This tree is then stripped off to generate a syntax-tree (a tree which does not contain the reduction steps).

The semantic actions performed by the back-end could also be made table-driven though, in this case a conventional algorithm was used. If all parts of the compiler were table-driven then it may be possible to consider the compiler as a GPC.

In the case of this compiler, the interface between the front and the back-end may not be adequate if the back-end was also to be made table-driven. A more suitable interface would perhaps be translation to some abstract intermediate form which could be used by a translator based on a table-driven algorithm [Graham 80].

B.1 The Compiler Structure

Traditionally, the process of compilation consists of three main tasks:

- lexical analysis
- syntactic analysis
- translation phase

Compilers for parallel architectures need a new phase to detect and evaluate the inherent parallelism within the source code.

The overall structure of the compiler proposed in this thesis is shown in figure B.2.

For clarity, optimisation, error handling and symbol table management modules have been excluded from the diagram.

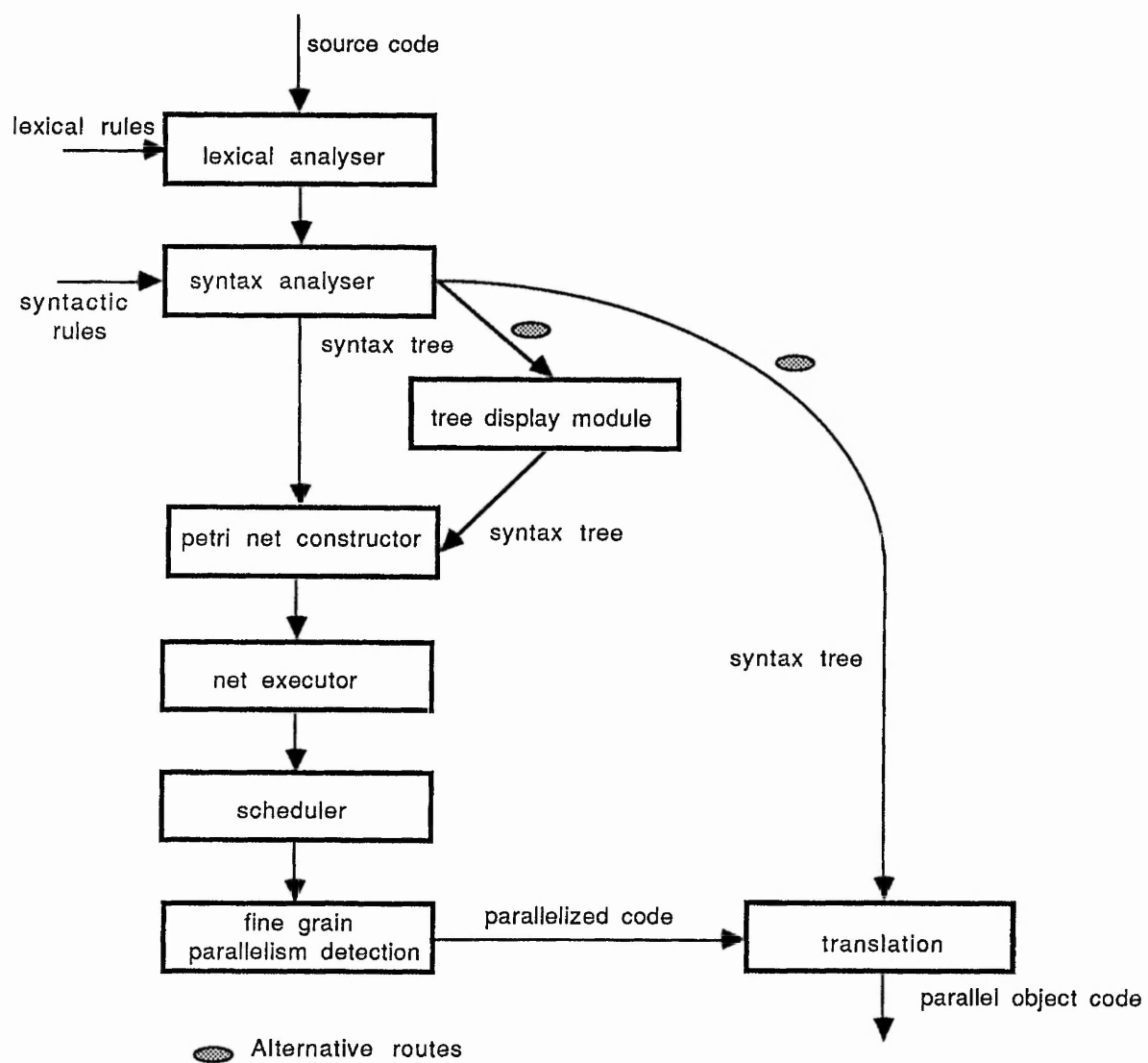


Figure B.2: Overall structure of the compiler

A compilation directive can cause the bypass of the parallelizing phase and hence output sequential object code suitable for execution on one node of the system, marked by the alternative routes on the diagram.

All scheduling is to be carried out statically at compile time and fine grain parallelism analysis takes place during scheduling of individual statements. This analysis leads to the decomposition of a computationally intensive expression into a number of communicating parallel sub-expressions to be evaluated on different nodes of the system in parallel using the concept of *co-processors* introduced later.

B..2 A Table-Driven Lexical Analyser (TDLA)

Any purpose-built lexical analyser has the lexical rules of the language it represents implicitly built into it, so it is difficult to separate the rules from the rest of the program. A specification for a table-driven lexical analyser is developed as follows:

1. The lexical rules of the source language are stored in a file together with a list of reserved and key words. A reserved word can have a number of attributes that will be output by the TDLA when the reserved word is recognised. For example, a reserved word can produce the output of a numerical token defined by a user, e.g. "begin" can be equivalent to integer 1.
2. Whenever no rules are defined by the user, the built in rules of TDLA are applied (see appendix C). Characters are assigned to one of the 12 built-in classes. Assignments can be altered or new classes created at any time. Once assignments and classes have been specified for a particular language the results will be stored and used until change is required.
3. The TDLA's global rules itemise the input stream into the following possible items:
 - (a) word

- (b) string
- (c) an integer
- (d) a decimal

Where a word ¹ is either;

- (a) a sequence of alphabetic or numeric characters
 - (b) a sequence of sign characters
 - (c) a sequence of words produced by (3a) and (3b) joined by underscore “_”
 - (d) a single separator eg “[”
 - (e) a sequence of characters in a new class
4. Classes 10 and 11, (see appendix C) specify the first and second characters of the beginning of a comment line. The global rules of TDLA requires that a comment to be terminated by the reverse of the two characters that started it, e.g. `/* this is a comment */`. In very exceptional cases, the lexical rules of a particular language may need to be slightly modified to conform with the global rules of TDLA. Thus, the comment in an implementation of pascal can be; `(* this is a comment *(` rather than `(* this is a comment*)`.
 5. Rule changes are accomplished by invoking the compiler, using the command line option “new_rules” instead of a source file name. When the routines for changing rules have been activated a user is guided throughout the changing process. Only the rules that need to be changed have to be modified.

B..3 A Table-Driven Parser (TDP)

Features of the proposed table-driven parser are developed as follows:

¹sign characters used in the context of comment descriptor would be automatically excluded from this rule

1. The parser accepts tokens generated by the lexical analyser and checks their syntactic validity.
2. The TDP must be powerful enough to parse the majority of programming language constructs by being given their syntactic rules.
3. Three different parsing methods form likely possibilities :
 - (a) template matching
 - (b) operator precedence
 - (c) look ahead left-right parsers (LALR)

Method 3a [Norman 83, Kluzmiak 86] is best suited for implementing languages like prolog and does not offer any notable advantage over other methods. Methods (3b) and (3c) are both table driven and met the objectives outlined above. However, method (3b) can only be applied to a small class of grammars and may not be effective in parsing all the principal languages that may be used.

4. LALR parsers are table driven, bottom up parsing techniques which scan the input stream from left to right one at a time. The tree is generated from the bottom (leaves) to the top (root). LALR parsers can recognise virtually all programming language constructs for which context free grammar can be written. Because of these characteristics this method is chosen for the new compiler.
5. To generate a parse table for a new language, its syntax rules are submitted to *YACC* in Backus Naur form with option *-v* selected [Johnson 86]. This parse table is then loaded into the parser. The front end for generating this table can be added to the compiler, with conflict detection and resolution capabilities, at a later date.
6. Errors are detected as soon as they occur and invoke the error handling routines to detect their type and issue an appropriate error message.

- The output of the parser is a syntax tree, figure B.3. Procedure definitions are represented by separate trees and are analysed and translated as soon as they are complete, i.e., before the main program begins.

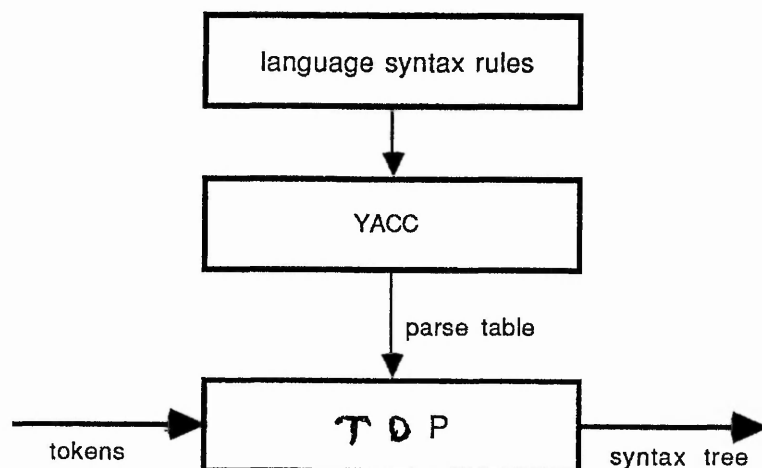


Figure B.3: The block diagram of a TDP

- Symbol table updates are also invoked by the parser whenever a constant, type or variable declaration is detected to be complete.
- The syntax analyser is fully table-driven. It generates a syntax tree and a symbol table for the next phases of the compiler. Figure B.4 shows a typical node of the tree generated.

B.1 Translation

Translation is fairly straightforward and is into 6809 assembly code. The translator is specified as follows:

- Translation is repeated for all nodes that have been allocated a workload.

if a = 2 then write("2") else write("not 2");

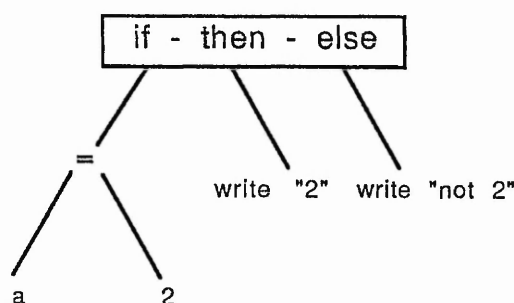


Figure B.4: A typical syntax tree

2. The translation begins with the generation of input/output routines for node one only. These routines handle the I/O operations. No other node has access to these routines since I/O operations only occur at node one. These are low level I/O routines supporting the higher level Pascal input/output requirements. There are a number of routines for arithmetic operations which are present in all the nodes.
3. The object files generated by the compiler are executable. The compilation resolves all the references and calls to the subroutines. In other words, in this experimental version the compiler acts as a linker too. As a result, it will not be possible to link separately compiled modules together. The solution is to combine all the source modules together in one file and then compile this one resulting module. This is not a major problem since programs that need to be run on MINNIE must be recompiled by this compiler in any case, therefore, sources must be made available.
4. As no linking facility is provided, the compiler includes the run-time library and the support code. Loading mechanisms and software were provided by the hardware designer.

5. The translator first translates the procedures as they are defined. Prior to the procedure translation code is generated for an unconditional jump to the label which identifies the beginning of the main program. This is to say that the main entry point to the program is set up.

The translator consists of a number of procedures each of which is responsible for the translation of a particular construct. The tree representing the workload of each node is given in turn to the translator and all the elements of this list are translated one at a time. Each element is a construct and the procedure for the translation of the particular construct is invoked. The construct is then dealt with and nested constructs invoke the main translator procedure. This modular approach reduced the implementation and maintenance time of the translator quite considerably.

Appendix C

Lexical Rules of Pascal-S

The following tokens are generated by the lexical analyser:

```

IDENTIFIER PROGRAM CONST TYPE VAR PROCEDURE BEGIN  END
IF THEN ELSE CASE WHILE DO REPEAT UNTIL FOR TO INTEGER
PIDENTIFIER CIDENTIFIER STRING TIDENTIFIER TRUE  FALSE
AIDENTIFIER RIDENTIFIER FUNCTION FUNCIDENTIFIER OF RECORD
REAL VIDENTIFIER FIDENTIFIER ARRAY
':='
'+ ' '-'
'*' '/' 'div'
UMINUS  /* UNARY MINUS */
'mod' 'and' 'or' 'not'
'=' '<>' '<' '>' '>=' '<='

```

The TDLA uses the following default character assignment classes:

Class	Description
1	Alphabetic; the letters a-z and A-Z.
2	Numeric; the number 0-9.
3	Sign characters; e.g., '+', '\$'. Characters in class 10 and 11 will default to this class if not used in the context of a comment.
4	Underscore; '_'.
5	Separators; e.g., '.', ';', '[', '{'. Control characters are also included in this class (except for those in class 6 and ASCII 128-255).
6	Spaces including white spaces.
7	String quotes " ' ".
8	Character quote "".
9	End-of-line comment character.
10	Bracketed comment or sign; first character.
11	Bracketed comment or sign; second character.
12	Alphabeticiser: This is a special class that forces the next character in the input stream to be of class alphabetic, i.e., class 1.

Appendix D

Syntactic Rules of Pascal-S

The following are the syntax rules of Pascal-S as presented to YACC for generating the parse table.

```
prog: PROGRAM IDENTIFIER '(' idelist ')' ';' block '.' ;
block: constsec typesec varsec proceduresec funcsec
      BEGIN
          statementsequence
      END
      ;
constsec:
  | CONST cbody
  ;
cbody: IDENTIFIER '=' constant ';'
      | IDENTIFIER '=' constant ';' cbody
      ;
typesec:
  | TYPE tbody
```

```

;
tbody: IDENTIFIER '=' typee ';'
      | IDENTIFIER '=' typee ';' tbody
;
varsec:
      | VAR vbody
;
vbody: idelist ':' typee ';'
      | idelist ':' typee ';' vbody
;
proceduresec:
      | PROCEDURE pbody ';' proceduresec
;
funcsec:
      | FUNCTION funcbody ';' funcsec
;
funcbody: IDENTIFIER formalparameterlist ':' TIDENTIFIER ';' block;
pbody: IDENTIFIER formalparameterlist ';' block ;
typee: TIDENTIFIER
      | ARRAY '[' abody ']' OF typee
      | RECORD rbody END
;
abody: constant '.' '.' constant
      | constant '.' '.' constant ',' abody
;
rbody:
      | idelist ':' typee
      | idelist ':' typee ';' rbody
;
formalparameterlist:
      | '(' fbody ')'

```

```

;
fbody: VAR idelist ':' TIDENTIFIER
      | idelist ':' TIDENTIFIER
      | idelist ':' TIDENTIFIER ';' fbody
      | VAR idelist ':' TIDENTIFIER ';' fbody
;
idelist: IDENTIFIER
        | IDENTIFIER ',' idelist
;
statementsequence: statement
                  | statement ';' statementsequence
;
statement:
  | variable ':=' expression
  | FUNCIDENTIFIER ':=' expression
  | PIDENTIFIER
  | PIDENTIFIER actualparameterlist
  | BEGIN statementsequence END
  | IF expression THEN statement
  | IF expression THEN statement ELSE statement
  | CASE expression OF END
  | CASE expression OF casebody END
  | WHILE expression DO statement
  | REPEAT statementsequence UNTIL expression
  | FOR VIDENTIFIER ':=' expression TO expression DO statement
;
casebody: myconstant ':' statement
         | myconstant ':' statement ';' casebody
myconstant: constant
           | constant ',' myconstant
;

```

```
expression: simpleexpression
  | simpleexpression '=' simpleexpression
  | simpleexpression '<>' simpleexpression
  | simpleexpression '<' simpleexpression
  | simpleexpression '>' simpleexpression
  | simpleexpression '>=' simpleexpression
  | simpleexpression '<=' simpleexpression
  ;

simpleterm: term
  | '+' term %prec UMINUS
  | '-' term %prec UMINUS
  ;

simpleexpression: simpleterm subexpression ;

subexpression:
  | '+' term subexpression
  | '-' term subexpression
  | 'or' term subexpression
  ;

term: factor subfactor;

subfactor:
  | '*' factor subfactor
  | '/' factor subfactor
  | 'div' factor subfactor
  | 'mod' factor subfactor
  | 'and' factor subfactor
  ;

factor: constant
  | variable
  | '(' expression ')'
```

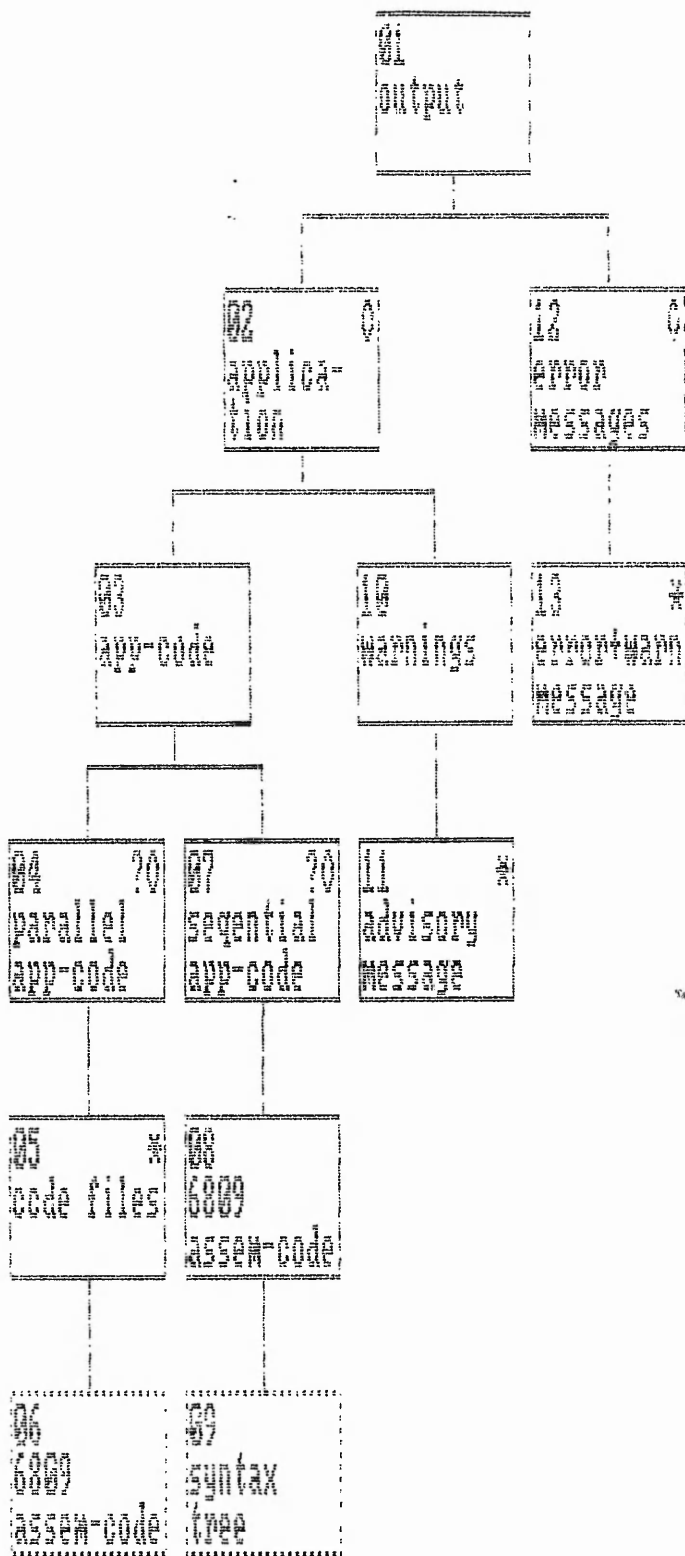
```
    | 'not' factor
    ;
actualparameterlist: '(' aplbody ')' ;
aplbody: expression
    | expression ',' aplbody
    ;
variable: VIDENTIFIER
    | AIDENTIFIER index
    | RIDENTIFIER fields
    ;
index:
    | '[' aplbody ']' index
    ;

fields:
    | '.' FIDENTIFIER fields
    ;
constant: number
    | CIDENTIFIER
    | boolean
    | STRING
    ;
number: REAL
    | INTEGER
    ;
boolean: FALSE
    | TRUE
    ;
```

Appendix E

Formal Design of the Compiler Using JSP

The first diagram and its subtrees show the structure of the data to be processed by the program. The subsequent diagrams show the JSP of functional units within the program.



01
Syntax
Tree

02
Statement

03
tokens

04
Character

05
input

01 3
petri-net

02
syntax
tree

03 *
statement

04 *
tokens

05 *
characters

06
input file

01
drive
compiler

02
set-up

03
process

04
change
rule

06
set-up
rules

07
translate

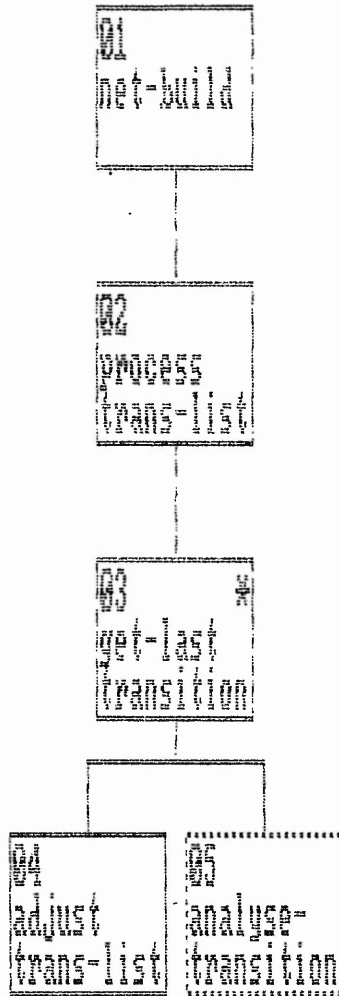
08
unwind

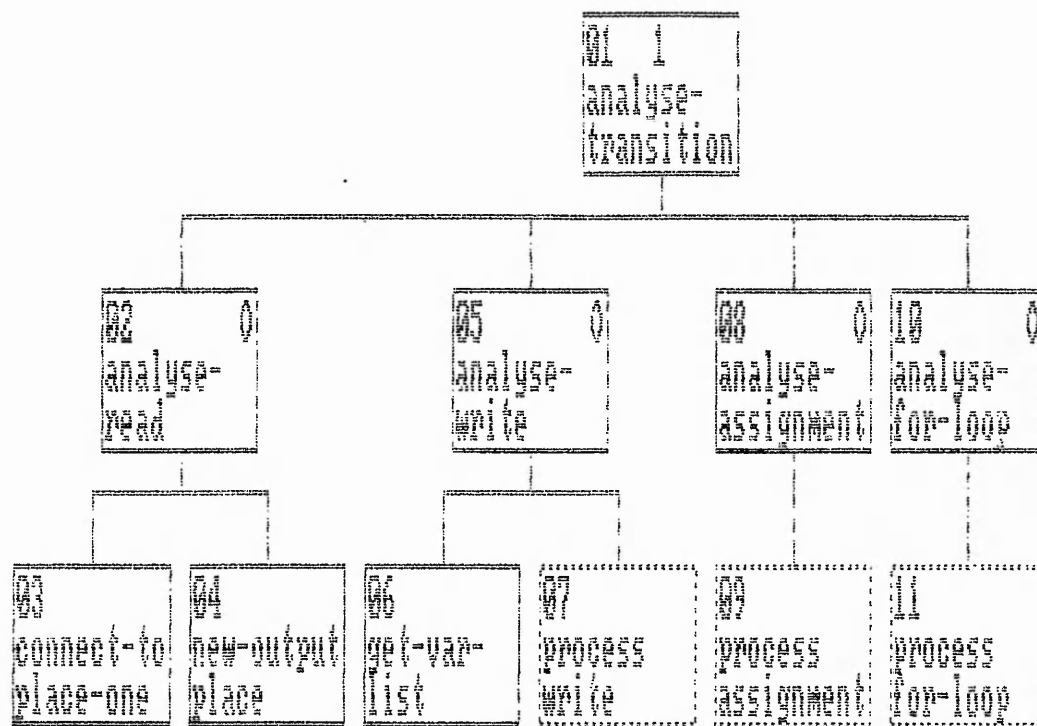
05 *
input-new
rules

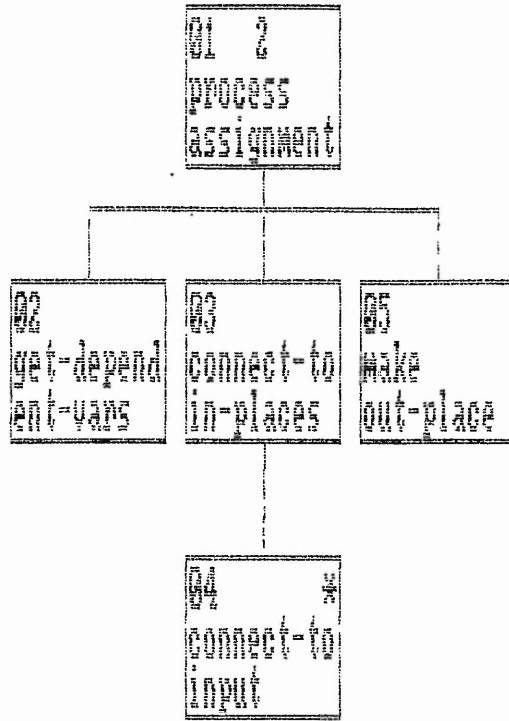
08
drive
PARSER

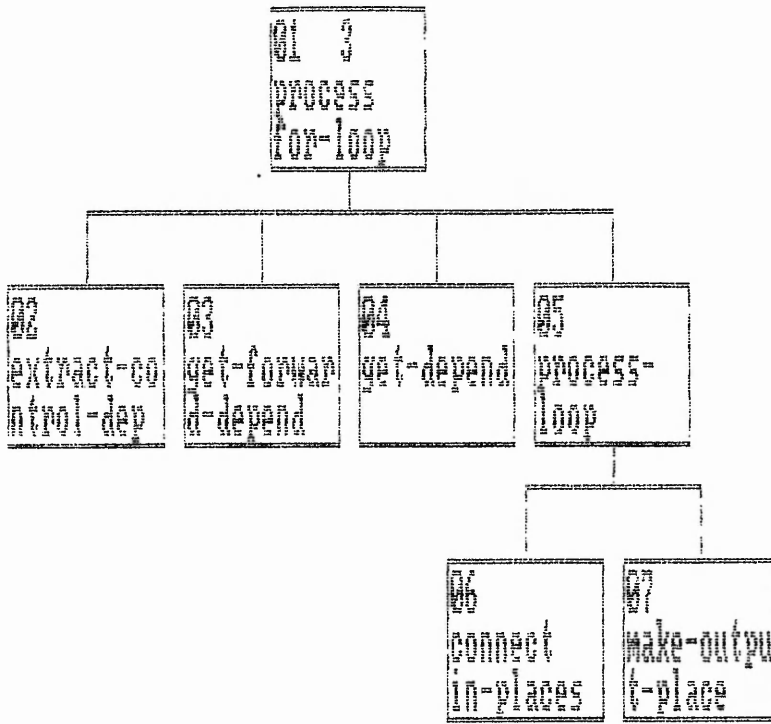
10
parallelize

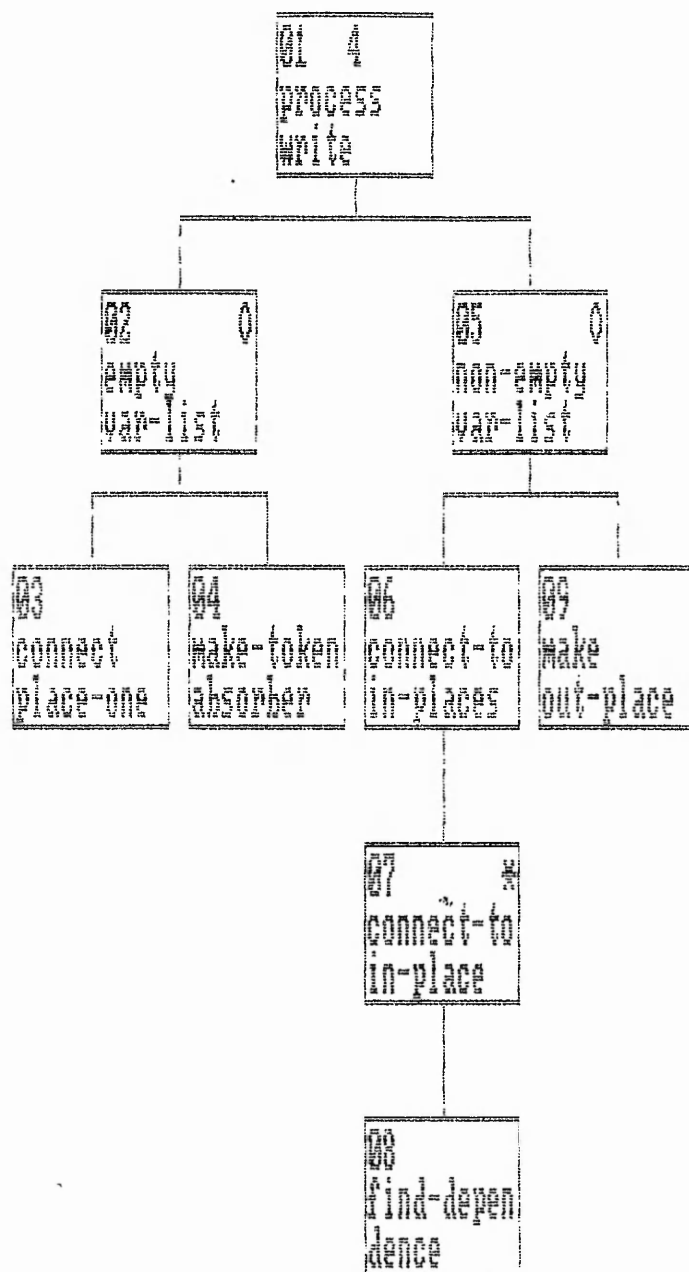
09 *
lex-analyse

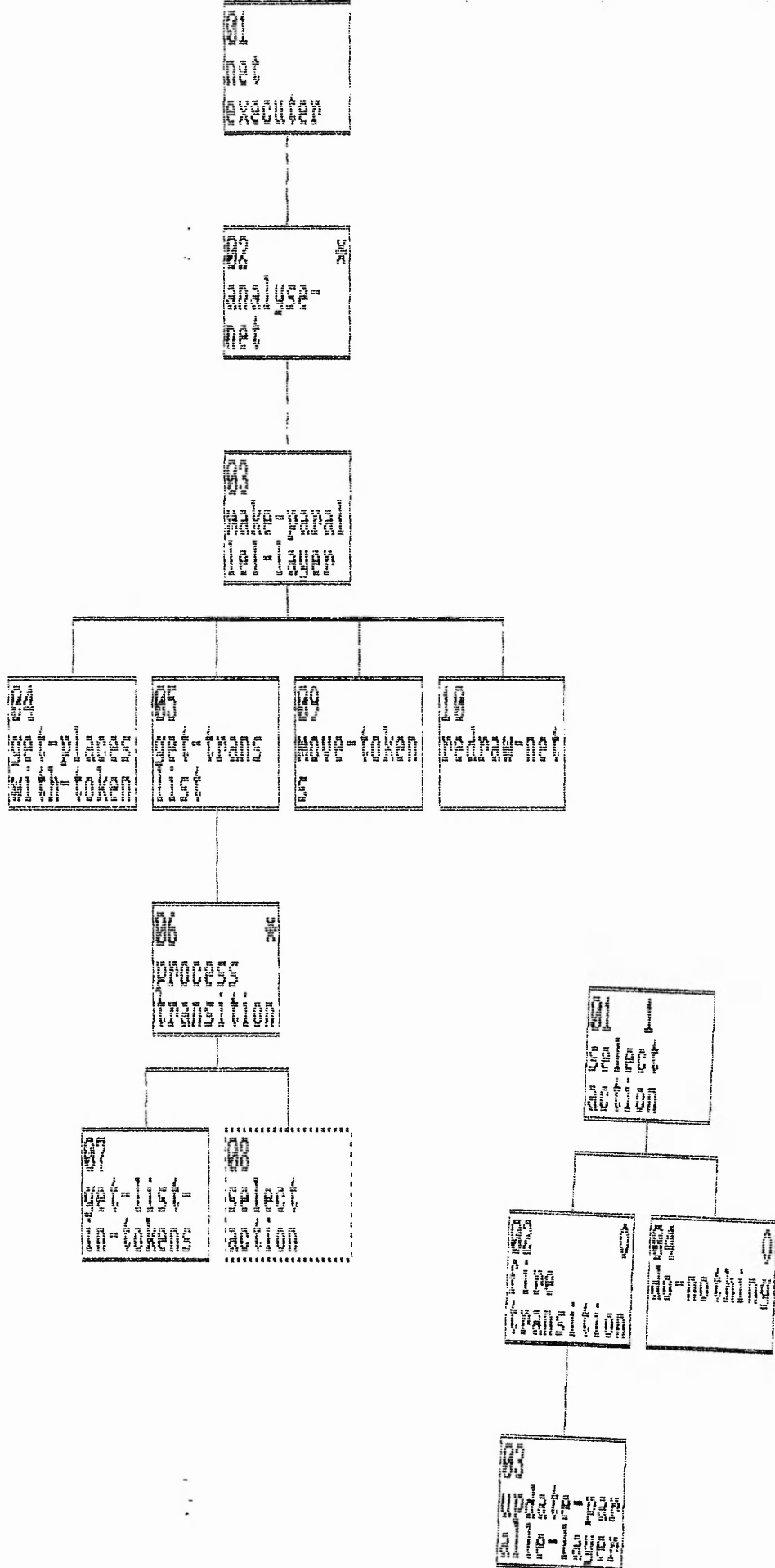


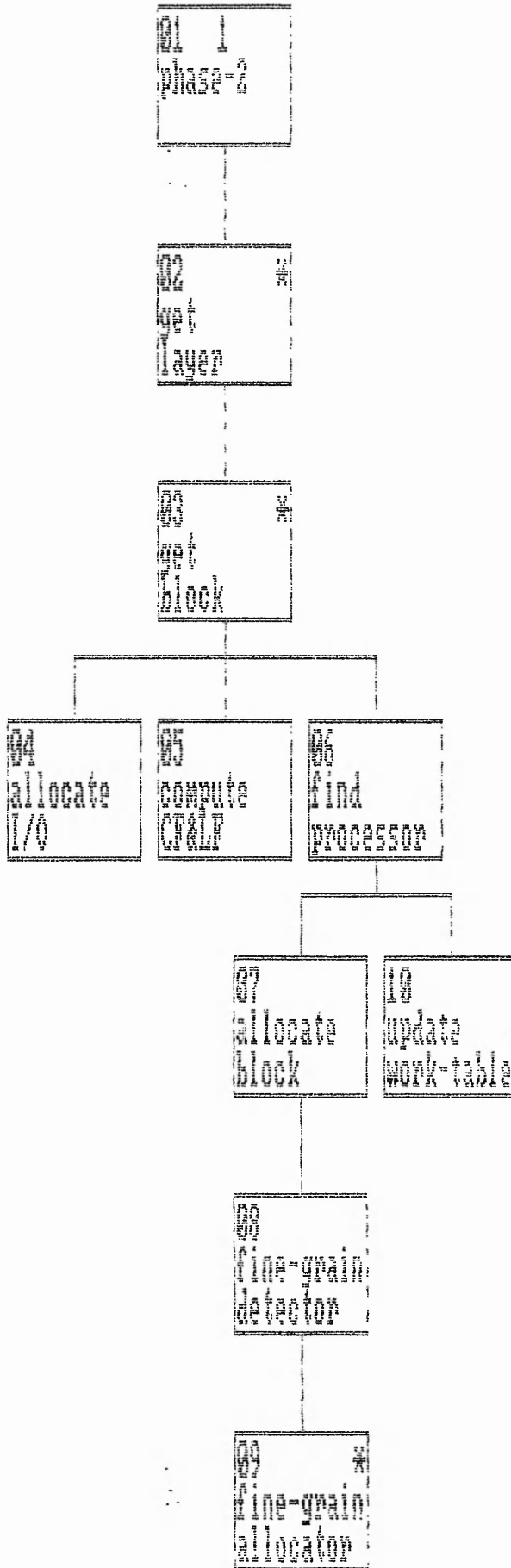


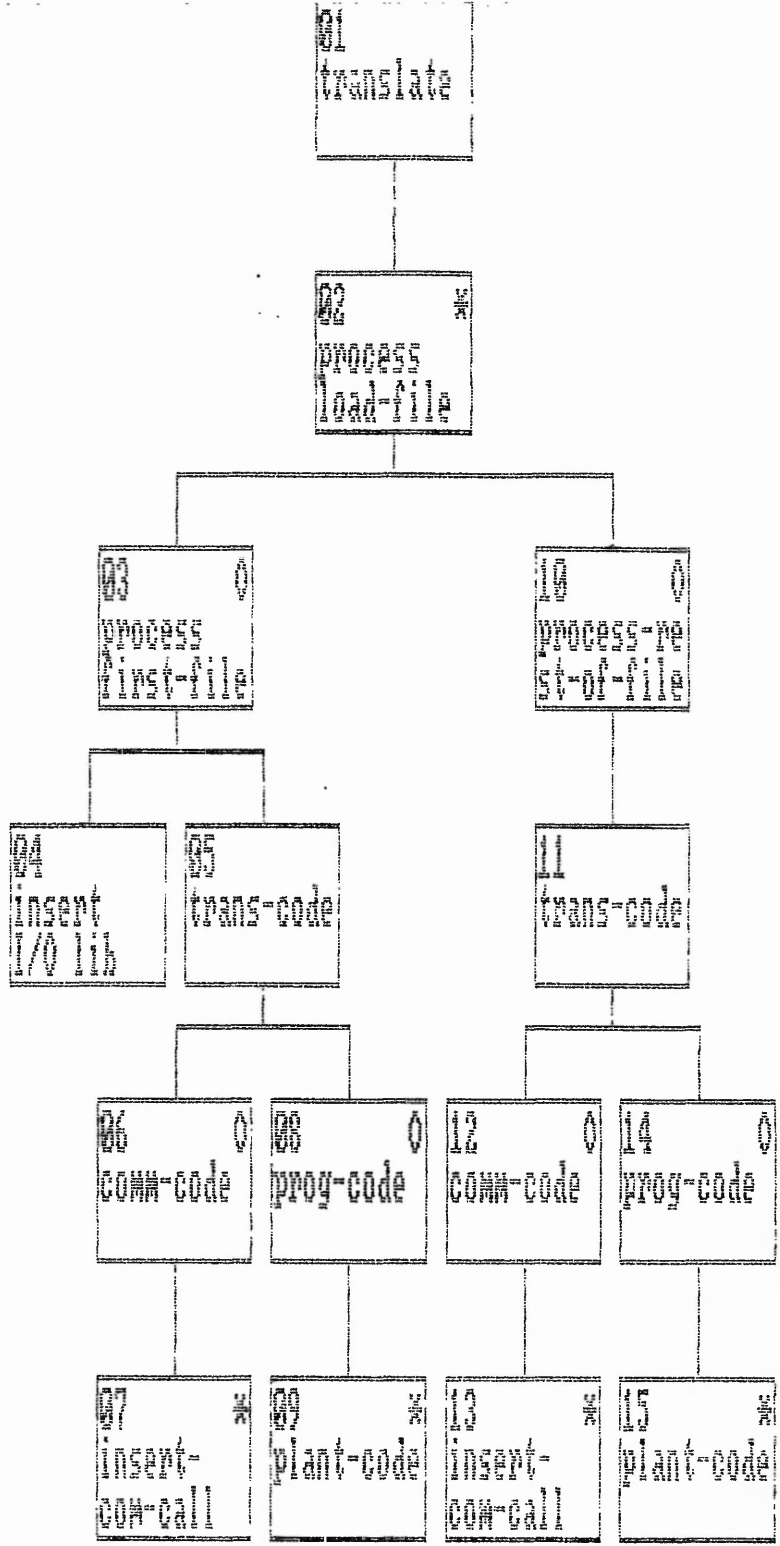


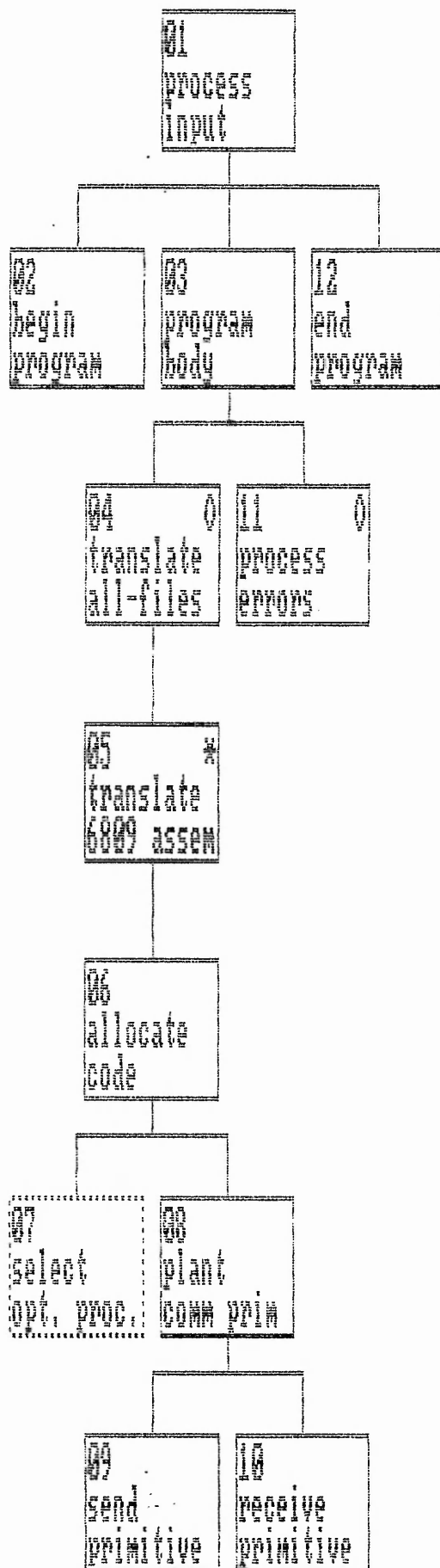


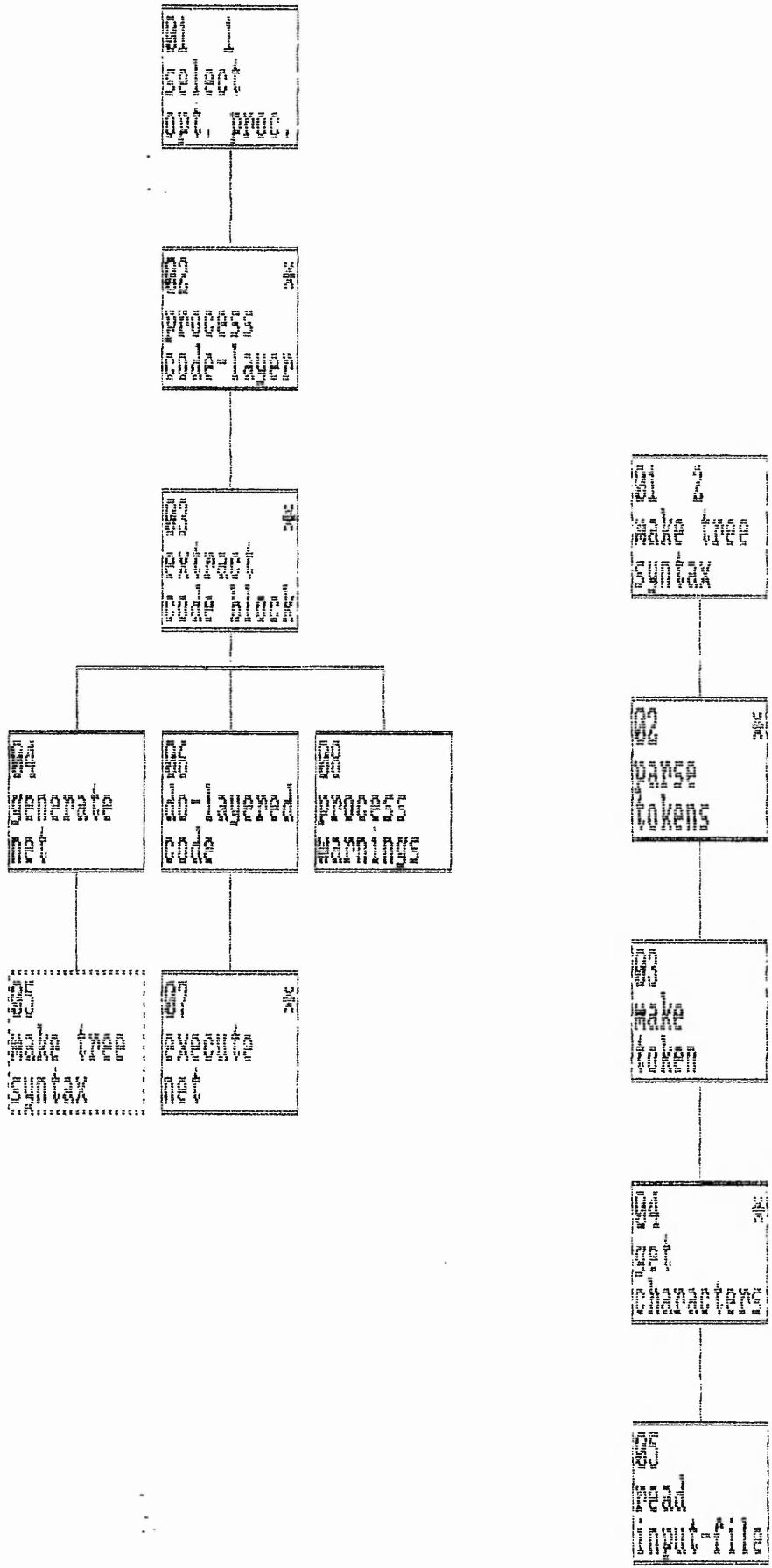


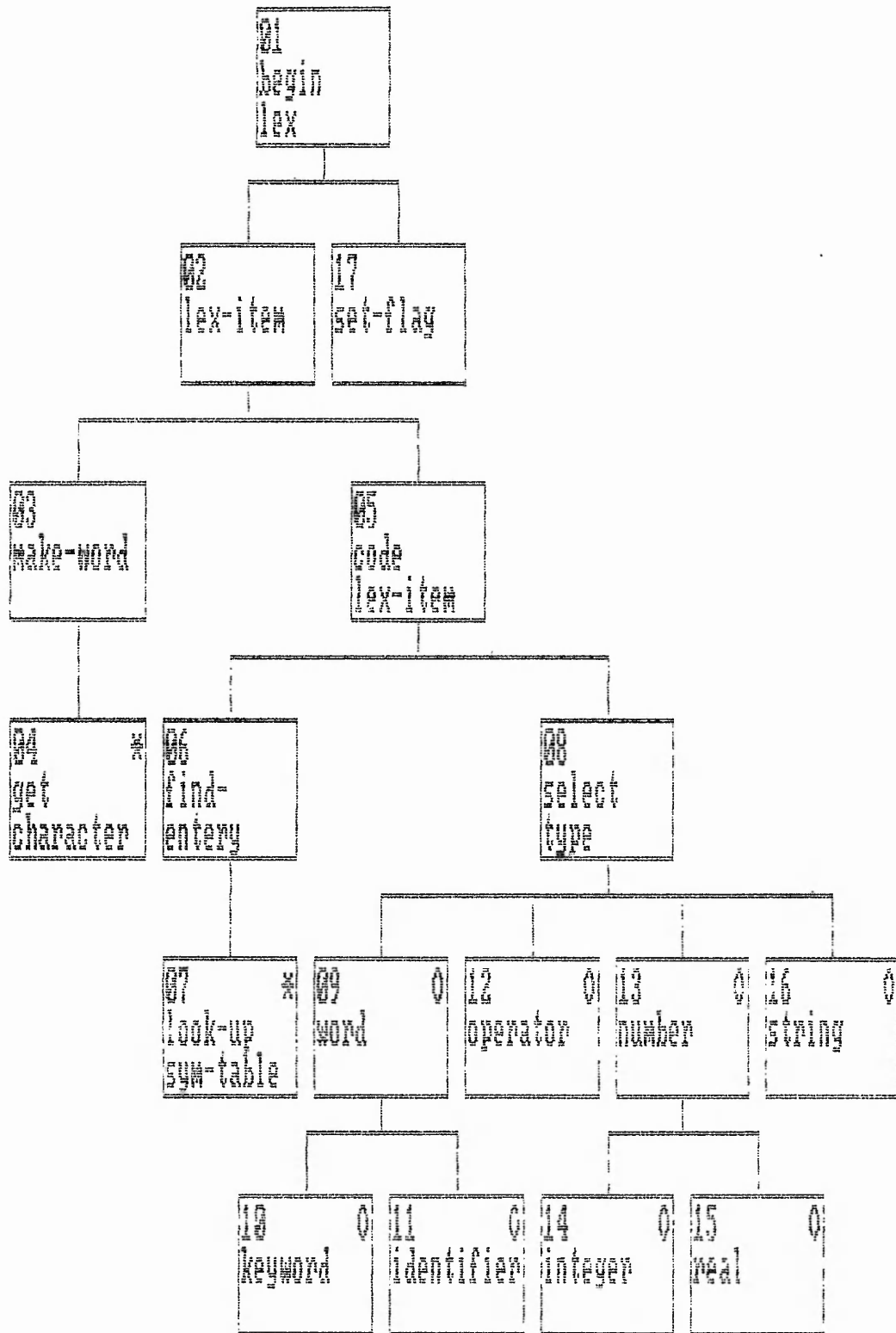


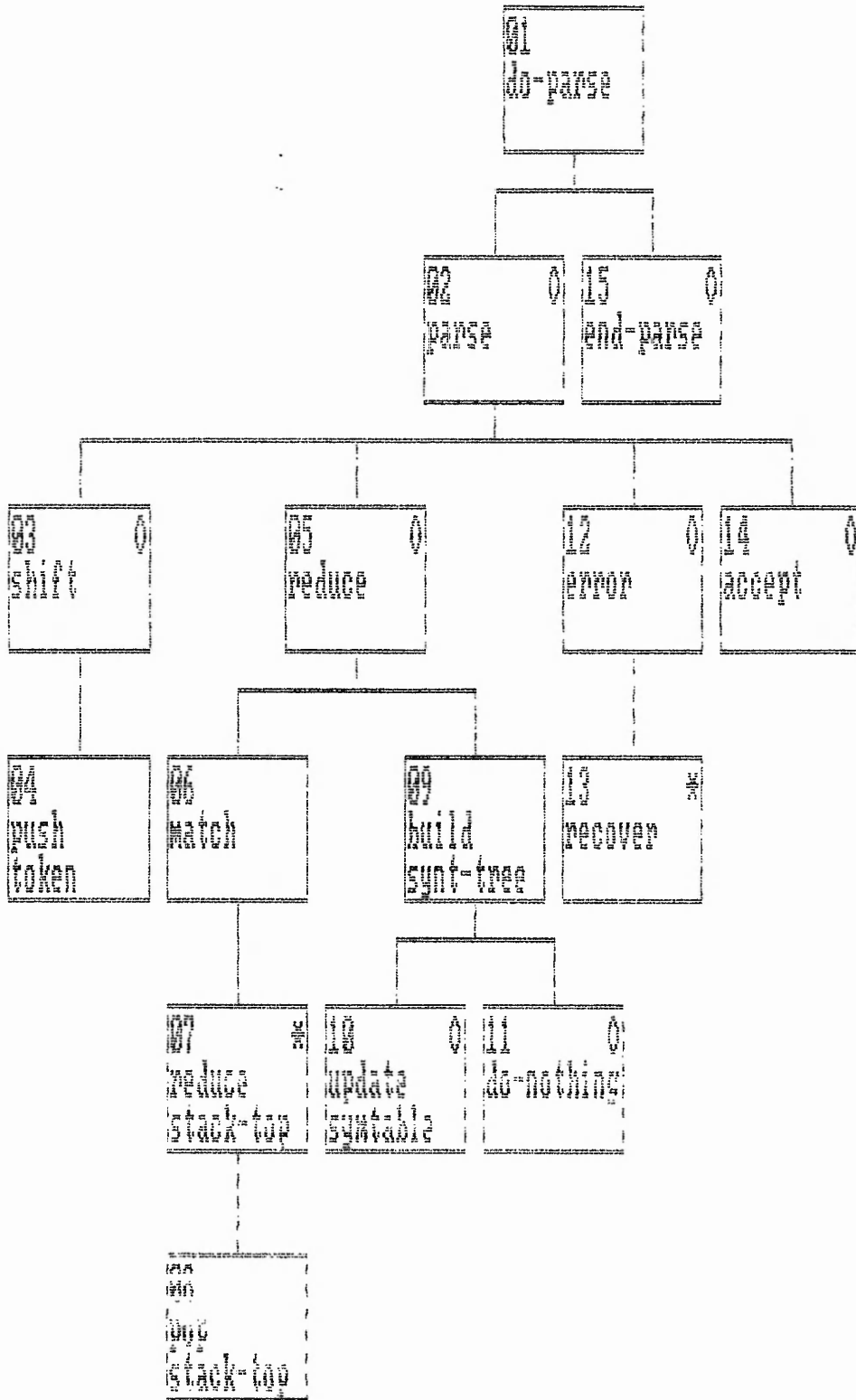












Appendix F

A Sample Test Program for the Code Generator

The sample test program is shown both in its high-level and machine code form. The high-level form is represented by a list of list structure and is generated by the compiler.

```

** [[main] [[ldu ## b000] [[lda ## 0] [sta false] [[lda ## ff] [sta true]]
** [[readk g]
  [readln [g , [c , [new , boolname]]]]
  [write [g , [c , [new , boolname]]]]
  [writeln]
  [c := [c + 1]]
  [g := [[g - c] * g]]
  [writeln [g new = , g]]
  [writeln [c new = , c]]
  [while boolname do
    [[while [not [c < 12]] do [c := [c + 1]] [boolname := false]]]
  [writeln [g , c]]
  [repeat [[c := [c + 1]] [g := [g + 3]]] until [c > 12]]
  [writeln [g , c]]
  [if [not boolname]
    then
      [[c := [c - 6]] [g := [g + c]]]
    else
      [[c := [c + 1]] [c := [c - 2]]]]
  [writeln [g , c]]
  [for c := 3 to 10 do
    [[g := [g + 1]]
    [[for g := 1 to 2 do [writeln g] [writeln [g , c]]]]]
  [readln c]
  [case new of
    [z :
      [case c of [1 : [c := [c + 1]] [2 : [c := [c + 4]]]]]
      [[d , [a , s]] : [g := [g + 1]]]]]
  [writeln [g , c]]
  [g := [- [[- c] + g] * g]]
  [writeln [g , c]]

```



```
** [[bool]]
    [jsr string]
    [andcc ## 0]
    [cmpb ## 74]
    [bne false2]
    [ldb ## ff]
    [pshub]
    [bra incpo]
    [false2 ldb ## 0]
    [pshub]
    [incpo ldd # 5]
    [addd pointer]
    [strd pointer]
    [rts bool]]

** [[output]]
    [andcc ## 0]
    [cmpa ## 0]
    [bne string]
    [std addr1]
    [andcc ## 0]
    [ldx # addr1]
    [tst 1 , x]
    [bpl out]
    [jsr minus]
    [out ldx # addr1]
    [jsr outdec]
    [rts]
    [string jsr patch]
    [finito rts out]]

** [[minus]]
    [andcc ## 0]
    [lda ## 2 d]
    [jsr patch]
    [ldb ## ff]
    [eorb 1 , x]
    [incb]
    [lda ## 0]
    [std addr1]
    [rts minus]]
```

```

** [jsr readk]
** [jsr inbuff]
** [ldb ## 0]
** [jsr input]
** [[pulu b] [stb g]]
** [ldb ## 0]
** [jsr input]
** [[pulu b] [stb c]]
** [ldb ## ff]
** [jsr input]
** [jsr nxtch]
** [[pulu b] [stb new]]
** [ldb ## 11]
** [jsr input]
** [jsr nxtch]
** [[pulu b] [stb boolname]]
** [jsr pcr1f]
** [[lda ## 0] [ldb g] [jsr output]]
** [[lda # 32] [jsr output]]
** [[lda ## 0] [ldb c] [jsr output]]
** [[lda # 32] [jsr output]]
** [[lda new] [jsr output]]
** [[lda # 32] [jsr output]]
** [[andcc ## 0]
    [lda boolname]
    [teta]
    [bne true1]
    [[lda # 102] [jsr output]]
    [[lda # 97] [jsr output]]
    [[lda # 108] [jsr output]]
    [[lda # 115] [jsr output]]
    [[lda # 101] [jsr output]]
    [bra false1]
    [[true1 lda # 116] [jsr output]]
    [[lda # 114] [jsr output]]
    [[lda # 117] [jsr output]]
    [[lda # 101] [jsr output]]
    [false1]]
** [[lda # 32] [jsr output]]
** [jsr pcr1f]
** [[ldb c] [pslu b]]
** [[ldb # 1] [pslu b]]
** [andcc ## 0]
** [[pulu a] [pulu b] [stb addr1] [adda addr1] [pslu a]]
** [[pulu b] [stb c]]
** [[ldb g] [pslu b]]
** [[ldb c] [pslu b]]
** [andcc ## 0]
** [[pulu b] [pulu a] [stb addr1] [suba addr1] [pslu a]]
** [[ldb g] [pslu b]]
** [andcc ## 0]
** [[pulu a] [pulu b] [mul1] [pslu b]]
** [[pulu b] [stb g]]
** [[lda # 103] [jsr output]]
** [[lda # 32] [jsr output]]
** [[lda # 110] [jsr output]]
** [[lda # 101] [jsr output]]
** [[lda # 119] [jsr output]]
** [[lda # 32] [jsr output]]

```

```

** [[lda # 61] [jsr output]]
** [[lda # 32] [jsr output]]
** [[lda ## 0] [ldb g] [jsr output]]
** [[lda # 32] [jsr output]]
** [jsr perlf]
** [[lda # 99] [jsr output]]
** [[lda # 32] [jsr output]]
** [[lda # 110] [jsr output]]
** [[lda # 101] [jsr output]]
** [[lda # 119] [jsr output]]
** [[lda # 32] [jsr output]]
** [[lda # 61] [jsr output]]
** [[lda # 32] [jsr output]]
** [[lda ## 0] [ldb c] [jsr output]]
** [[lda # 32] [jsr output]]
** [jsr perlf]
** [label1]
** [[lda boolname] [pshua]]
** [andcc ## 0]
** [[pulu a] [cmpa ## f?] [ibne label2]]
** [label3]
** [[ldb c] [pshu b]]
** [[ldb # 12] [pshu b]]
** [andcc ## 0]
** [[pula]
    [cmpa , u +]
    [lbgt label4]
    [[lda ## 0] [pshu a]]
    [bra label5]
    [[label4 lda ## f] [pshua]]
    [label5]]
** [[pulu a]
** [[eora ## f] [pshu a]]
** [andcc ## 0]
** [[pulu a] [cmpa ## f] [ibne label6]]
** [[ldb c] [pshu b]]
** [[ldb # 1] [pshu b]]
** [andcc ## 0]
** [[pulu a] [pulu b] [stb addr1] [adda addr1] [pshu a]]
** [[pulu b] [stb c]]
** [[bra label3] [label6]]
** [[ldb false] [pshu b]]
** [[pulu b] [stb boolname]]
** [[bra label1] [label2]]
** [[lda ## 0] [ldb g] [jsr output]]
** [[lda # 32] [jsr output]]
** [[lda ## 0] [ldb c] [jsr output]]
** [[lda # 32] [jsr output]]
** [jsr perlf]
** [label7]
** [[ldb c] [pshu b]]
** [[ldb # 1] [pshu b]]
** [andcc ## 0]
** [[pulu a] [pulu b] [stb addr1] [adda addr1] [pshu a]]
** [[pulu b] [stb c]]
** [[ldb g] [pshu b]]
** [[ldb # 3] [pshu b]]
** [andcc ## 0]
** [[pulu a] [pulu b] [stb addr1] [adda addr1] [pshu a]]

```

```

** [[pulu b] [stb g]]
** [[ldb c] [pspu b]]
** [[ldb # 12] [pspu b]]
** [andcc ## 0]
** [[pulu a]
    [cmpa , u +]
    [lbit label18]
    [[lda false] [pspu a]]
    [lbra label19]
    [[label18 lda true] [pspu a]]
    [label19]]
** [[andcc ## 0] [pulu a] [cmpa true] [bne label7]]
** [[lda ## 0] [ldb g] [jsr output]]
** [[lda # 32] [jsr output]]
** [[lda ## 0] [ldb c] [jsr output]]
** [[lda # 32] [jsr output]]
** [jsr printf]
** [[lda boolname] [pspu a]]
** [pulu a]
** [[leora ## ff] [pspu a]]
** [[pulu a] [andcc ## 0] [cmpa true] [lbra label10]]
** [[ldb c] [pspu b]]
** [[ldb # 6] [pspu b]]
** [andcc ## 0]
** [[pulu b] [pulu a] [stb addr1] [suba addr1] [pspu a]]
** [[pulu b] [stb c]]
** [[ldb g] [pspu b]]
** [[ldb c] [pspu b]]
** [andcc ## 0]
** [[pulu a] [pulu b] [stb addr1] [adda addr1] [pspu a]]
** [[pulu b] [stb g]]
** [lbra label11]
** [label10]
** [[ldb c] [pspu b]]
** [[ldb # 1] [pspu b]]
** [andcc ## 0]
** [[pulu a] [pulu b] [stb addr1] [adda addr1] [pspu a]]
** [[pulu b] [stb c]]
** [[ldb c] [pspu b]]
** [[ldb # 2] [pspu b]]
** [andcc ## 0]
** [[pulu b] [pulu a] [stb addr1] [suba addr1] [pspu a]]
** [[pulu b] [stb c]]
** [label11]
** [[lda ## 0] [ldb g] [jsr output]]
** [[lda # 32] [jsr output]]
** [[lda ## 0] [ldb c] [jsr output]]
** [[lda # 32] [jsr output]]
** [jsr printf]
** [[ldb # 8] [pspu b]]
** [[pulu b] [stb c]]
** [[lda # 10] [pspu a]]
** [[label12 lda c] [andcc ## 0] [cmpa , u] [lbit label13]]
** [[ldb g] [pspu b]]
** [[ldb # 1] [pspu b]]
** [andcc ## 0]
** [[pulu a] [pulu b] [stb addr1] [adda addr1] [pspu a]]
** [[pulu b] [stb g]]
** [[ldb # 1] [pspu b]]

```

```

** [[pulu b] [stb g]]
** [[lda # 2] [pspu a]]
** [[label14 lda g] [andcc # 0] [cmpa , u] [logt label15]]
** [[lda # 0] [ldb g] [jcr output]]
** [[lda # 32] [jcr output]]
** [jcr pcrif]
** [inc g]
** [[bra label14] [label15 dec g] [pulu b]]
** [[lda # 0] [ldb g] [jcr output]]
** [[lda # 32] [jcr output]]
** [[lda # 0] [ldb c] [jcr output]]
** [[lda # 32] [jcr output]]
** [jcr pcrif]
** [inc c]
** [[bra label12] [label13 dec c] [pulu b]]
** [jcr inbuff]
** [ldb # 0]
** [jcr input]
** [[pulu b] [stb c]]
** [jcr pcrif]
** [[lda new] [pspu a]]
** [[lda # 122] [pspu a]]
** [[pulu a] [andcc # 0] [cmpa , u] [lbn label17]]
** [[lda c] [pspu a]]
** [[lda # 1] [pspu a]]
** [[pulu a] [andcc # 0] [cmpa , u] [lbn label19]]
** [[ldb c] [pspu b]]
** [[ldb # 1] [pspu b]]
** [andcc # 0]
** [[pulu a] [pulu b] [stb addr1] [adda addr1] [pspu a]]
** [[pulu b] [stb c]]
** [bra label18]
** [label19]
** [[lda # 2] [pspu a]]
** [[pulu a] [andcc # 0] [cmpa , u] [lbn label20]]
** [[ldb c] [pspu b]]
** [[ldb # 4] [pspu b]]
** [andcc # 0]
** [[pulu a] [pulu b] [stb addr1] [adda addr1] [pspu a]]
** [[pulu b] [stb c]]
** [bra label18]
** [label20]
** [label18]
** [pulu b]
** [bra label16]
** [label17]
** [[lda # 100] [pspu a]]
** [[pulu a] [andcc # 0] [cmpa , u] [lbn label21]]
** [[lda # 97] [pspu a]]
** [[pulu a] [andcc # 0] [cmpa , u] [lbn label21]]
** [[lda # 115] [pspu a]]
** [[pulu a] [andcc # 0] [cmpa , u] [lbn label21]]
** [pspu a]
** [label21]
** [[pulu a] [andcc # 0] [cmpa , u] [lbn label22]]
** [[ldb g] [pspu b]]
** [[ldb # 1] [pspu b]]
** [andcc # 0]

```



```
** [[pulu a] [pulu b] [stb addr1] [adda addr1] [pslu a]]
** [[pulu b] [stb g1]
** [[bra label16]
** [[label22]
** [[label16]
** [[pulu b]
** [[lda ## 0] [ldb g] [jcr output]]
** [[lda # 32] [jcr output]]
** [[lda ## 0] [ldb c] [jcr output]]
** [[lda # 32] [jcr output]]
** [[jcr pcr1f]
** [[lda c] [pslu a]]
** [[pulu a]
** [[negal] [pslu a]]
** [[ldb g] [pslu b]]
** [[andcc ## 0]
** [[pulu a] [pulu b] [stb addr1] [adda addr1] [pslu a]]
** [[ldb g] [pslu b]]
** [[andcc ## 0]
** [[pulu a] [pulu b] [mul] [pslu b]]
** [[pulu a]
** [[negal] [pslu a]]
** [[pulu b] [stb g1]
** [[lda ## 0] [ldb g] [jcr output]]
** [[lda # 32] [jcr output]]
** [[lda ## 0] [ldb c] [jcr output]]
** [[lda # 32] [jcr output]]
** [[jcr pcr1f]
** [[coding done]
:
```

Appendix G

A Sample Parallelised Test Program

The sample test program is shown both in its high-level and machine code form. Note that there are two files, each containing the workload of an individual processor.

```
program area(input, output);

    const steps = 10000;
    var a, b, area, delta, i, strip, funcval : integer;

begin
    writeln (" Please input the value of upper and lower bounds"); {1}
    readln (a,b); {2}
    area := 0; {3}
    delta := (b - a) / steps; {4}
    writeln("iteration step :"); {5}
    for i := 1 to steps do {6}
```

```
begin
    write (i);           { i}
    funcval := (a ** 2) + a + 8;   { ii}
    strip := funcval * delta;      {iii}
    a := a + delta;               { iv}
    area := area + strip;         { v}
end;
writeln;                       {7}
writeln ("The approximate value of area = ", area); {8}
end.
```

Below are the workload of processor 1 both in high-level and assembly format:

```
[
  [[receive] [writeln [Please input...]] [send]]]
  [[receive] [readln [a , b] [send 2 var a integer, 2 var b integer]]]
  [[receive] [writeln [iteration steps] [send]]]
  [[receive 2 area var integer, 2 delta var integer] [ for i := 1 to
10000
    [[write i]
      [[receive 2 var funcval integer] [strip := [funcval * delta] [send]]
      [[receive] [area := [area + strip]] [send]]
    [[receive] [writeln] [send]]]
  [[receive 6] [writeln [The approximate value of area = , area]] [send]]]
]
```

```
[[nam test program]                ;;; setup
[org $4000]
[base rmb 2]                        ;;; holds base of array
[addr1 rmb 2]                       ;;; variables used by translator
[addr2 rmb 2]
[addr3 rmb 2]
[false rmb 1]                       ;;; this locatn. holds value of false
[true rmb 1]                        ;;; similar to above
[display rmb 60]                    ;;; array for stack frame
[label0]
[ ldu #$b000]                       ;;; setting up user stack pointer
[ lda #$00]                         ;;; load false to a
[ sta false]                        ;;; put false in false
[ lda #$ff]                         ;;; load true in a
[ sta true]                         ;;; put true in locatin true
```

```

;;; set up stack frame base for main program
[ ldx #display]   ;;; pointing to display[1]
[ stu ,x]         ;;; store frame base in display[1]
[ tfr u,y]       ;;; current frame base kept in U reg
[outdec equ $cd39]
[indec equ $cd48]   ;;; inputs number from line buffer
;;; takes first char from line buffer and advances
[nxtch equ $cd27]
[inbuff equ $cd1b]   ;;; reads in a line of input
;;; location containing addr to next char in the buffer
[pointer equ $cc14]
[putch equ $cd18]   ;;; outputs char in reg a
[pcrlf equ $cd24] ;;; outputs cr and line feed
[lbra main]        ;;; branch to main program
[label1]           ;;; input retrns next val on stack
[ ldx #pointer]    ;;; ld pointer to buff in x reg
[ andcc #$00]      ;;; clear flags
[ tstb]            ;;; b= 0 signifies integer input
[ beq label2]      ;;; if inte. input bra to num
[ andcc #$00]      ;;; clear flags
[ cmpb #$ff]       ;;; b = $FF means read next char
[ beq label3]      ;;; if b = $ff bra to str
[ jsr label4]      ;;; otherwise read a bool value
[ rts /*bra tmam*/] ;;; rturn from this routine
[label3 jsr label5] ;;; read a char value
[ rts /*bra tmam*/] ;;; return
[label2 jsr label11] ;;; check sign of integer
[ jsr indec]       ;;; read the integer
[ exg x,d]         ;;; put the integer in A reg.
[ bcc label12]     ;;; if valid integer then goto OK
[ bra label13]     ;;; ++++ error condition

```

```

[label12 pulu cc] .   ;;; restore result of sign checking
[ bne label14]      ;;; if pos then goto pos
[ eorb #$ff]       ;;; if neg. then calculate 2's
[ inccb]           ;;; complement of the number
[label14 pshu b]    ;;; put result on the user stack
[tmam rts label1]

;;; checks sign of int and puts flag on the stack. if pos z bit is zero
[label11]
[ lda '[' ,x']' ] ;;; load A with location ptd. by x reg
[ andcc #$00] ;;; clear flags
[ cmpa #$2d] ;;; comp. first char of buff with "-"
[ pshu cc] ;;; push result on the stack
[ bne label15] ;;; if pos branch to return
[ jsr nxtch] ;;; advance buff pointer. jump over "-"
[label15 rts label11]

[label5] ;;; string reads next char in line buffer
[ jsr nxtch] ;;; read char and put in $cc18
[ ldb $cc18] ;;; put char in the b reg
[ pshub] ;;; push char on the stack
[ rts label5] ;;; return
[label4]
[ jsr label5] ;;; read in first character of bool
[ andcc #$00] ;;; clear flags
[ cmpb #$74] ;;; $74 is ascii t for (t)rue
[ bne label6 ] ;;; input must be false
[ ldb #$ff] ;;; true is loaded in the b reg
[ pshub] ;;; value pushed on the user stack
[ bra label7] ;;; branch to increm. line .buff pointer
[label6 ldb #$00];;; false loaded in b reg.
[ pshub] ;;; push value on stack
[label7 ldd #5] ;;; skip the next five chars

```

```

[ addd pointer]
[ strd pointer]
[ rts label4]   ;;; return
[label8] ;;; sub output; output reg b
[ andcc #$00]   ;;; clear flags
[ cmpa #$00]   ;;; check reg a for string code
[ bne label5]   ;;; branch to string for a /= 0
[ std adrr1]    ;;; store 16 bit num in adrr1
[ andcc #$00]   ;;; clear flags
[ ldx #adrr1]   ;;; load addr of num in x reg
[ tst 1,x]     ;;; test if num pointed by x is neg
[ bpl label9]   ;;; branch if positive
[ jsr label10]  ;;; deal with negative nums
[label9 ldx #adrr1];;; put high byte addr of num in x
[ jsr outdec]   ;;; output number
[ rts /*bra finito*/]
[label5 jsr putch];;; output character
[label15 rts label9]
[label10]
[ andcc #$00]   ;;; clear flags
[ lda #$2d]     ;;; load a with char "-"
[ jsr putch]    ;;; print "-"
[ ldb #$ff]    ;;; change num to its proper value
[ eorb 1,x]    ;;; 1's complement
[ incb]        ;;; by two's complementing it
[ lda #$00]    ;;; create 16 bits num from 8bits
[ std adrr1]   ;;; store 16 bits num, ms byte is 0's
[ rts label10]]
[main]
[lda # 112] [jsr output]
[lda # 108] [jsr output]

```

```
[lda # 101] [jsr output]
[lda # 97] [jsr output]
[lda # 115] [jsr output]
[lda # 101] [jsr output]
[lda # 32] [jsr output]
[lda # 105] [jsr output]
[lda # 110] [jsr output]
[lda # 112] [jsr output]
[lda # 117] [jsr output]
[lda # 116] [jsr output]
[lda # 32] [jsr output]
[jsr pcrLf]
[ldb #$0] [jsr input]
[pulu b] [stb va]
[ldb #$0] [jsr input]
[pulu b] [stb vb]
[jsr pcrLf]
[ldb va] [jsr send]
[ldb vb] [jsr send]
[lda # 105] [jsr output]
[lda # 116] [jsr output]
[lda # 116] [jsr output]
[lda # 105] [jsr output]
[lda # 114] [jsr output]
[lda # 97] [jsr output]
[lda # 116] [jsr output]
[lda # 105] [jsr output]
[lda # 111] [jsr output]
[lda # 110] [jsr output]
[lda # 32] [jsr output]
[lda # 115] [jsr output]
```



```
[lda # 116] [jsr output]
[lda # 101] [jsr output]
[lda # 112] [jsr output]
[lda # 115] [jsr output]
[lda # ] [jsr output]
[jsr pcrLf]
[ldb #1] [jsr rec][pulu b]
[stb area]
[ldb #2] [jsr rec][pulu b]
[stb delta]
[ldb #1] [pshu b]
[pulu b] [stb vi]
[lda #10000] [pshu a]
[label16 lda vi] [andcc #$0] [cmpa ,u] [lbgt 17]
[lda #$ 0] [ldb vi] [jsr output]
[ldb #3] [jsr rec][pulu b]
[stb funcval]
[ldb funcval] [pshu b]
[ldb delta] [pshu b]
[pulu a] [pulu b] [mul] [pshu b]
[pulu b] [stb strip]
[ldb area] [pshu b]
[ldb strip] [pshu b]
[pulu a] [pulu b] [stb addr1] [adda addr1] [pshu a]
[pulu b] [stb area]
[inc vi] [bra label16]
[label17 dec vi]
[jsr pcrLf]
[lda # 84] [jsr output]
[lda # 104] [jsr output]
[lda # 101] [jsr output]
```

[lda # 32] [jsr output]
[lda # 97] [jsr output]
[lda # 112] [jsr output]
[lda # 112] [jsr output]
[lda # 114] [jsr output]
[lda # 111] [jsr output]
[lda # 120] [jsr output]
[lda # 105] [jsr output]
[lda # 109] [jsr output]
[lda # 97] [jsr output]
[lda # 116] [jsr output]
[lda # 101] [jsr output]
[lda # 32] [jsr output]
[lda # 118] [jsr output]
[lda # 97] [jsr output]
[lda # 108] [jsr output]
[lda # 117] [jsr output]
[lda # 101] [jsr output]
[lda # 32] [jsr output]
[lda # 111] [jsr output]
[lda # 102] [jsr output]
[lda # 97] [jsr output]
[lda # 114] [jsr output]
[lda # 101] [jsr output]
[lda # 97] [jsr output]
[lda # 32] [jsr output]
[lda # 61] [jsr output]
[lda # 32] [jsr output]
[lda area] [jsr output]
[jsr pcrLf]

The workload of processor 2 both in high-level and assembly format:

```
[
  [[receive] [area := 0] [send 1 var area integer]]
  [[receive 1 var a integer, 1 var b integer]
    [delta := [b - a] / 10000] [send 1 var delta integer]]]
  [[receive ] [ for i := 1 to 10000
    [[receive ] funcval := [[a ** 2] + a + 8] [send 1 var funcval intege
    [a := [a + delta]]
  ]
```

```
[[nam test program]           ;;; setup
[org $4000]
[base rmb 2]                   ;;; holds base of array
[addr1 rmb 2]                   ;;; variables used by translator
[addr2 rmb 2]
[addr3 rmb 2]
[false rmb 1]   ;;; this locatn. holds value of false
[true rmb 1]    ;;; similar to above
[display rmb 60]           ;;; array for stack frame
[label0]
[ ldu #b000]   ;;; setting up user stack pointer
[ lda #00]    ;;; load false to a
[ sta false]  ;;; put false in false
[ lda #ff]   ;;; load true in a
[ sta true]  ;;; put true in locatin true
;;; set up stack frame base for main program
[ ldx #display] ;;; pointing to display[1]
[ stu ,x]      ;;; store frame base in display[1]
[ tfr u,y]    ;;; current frame base kept in U reg
[lbra main]   ;;; branch to main program
```

```

[label1]          .   ;;; input retrns next val on stack
;;; checks sign of int and puts flag on the stack. if pos z bit is zero
[label2]
[ lda '[' ,x' ] ] ;;; load A with location ptd. by x reg
[ andcc #$00 ] ;;; clear flags
[ cmpa #$2d ] ;;; comp. first char of buff with "-"
[ pshu cc ] ;;; push result on the stack
[ bne label3 ] ;;; if pos branch to return
[ jsr nxtch ] ;;; advance buff pointer. jump over "-"
[label3 rts label2]
[label4]
[ jsr labe3 ]    ;;; read in first character of bool
[ andcc #$00 ] ;;; clear flags
[ cmpb #$74 ] ;;; $74 is ascii t for (t)rue
[ bne labe5 ] ;;; input must be false
[ ldb #$ff ] ;;; true is loaded in the b reg
[ pshub ] ;;; value pushed on the user stack
[ bra labe6 ] ;;; branch to increm. line .buff pointer
[labe5 ldb #$00] ;;; false loaded in b reg.
[ pshub ] ;;; push value on stack
[labe6 ldd #5] ;;; skip the next five chars
[ addd pointer ]
[ strd pointer ]
[ rts label4 ]   ;;; return
[label7]
[ andcc #$00 ] ;;; clear flags
[ lda #$2d ]    ;;; load a with char "-"
[ jsr putch ]   ;;; print "-"
[ ldb #$ff ]    ;;; change num to its proper value
[ eorb 1,x ]    ;;; 1's complement
[ incb ]        ;;; by two's complementing it

```

```
[ lda #$00]      ;;; create 16 bits num from 8bits
[ std adrr1]     ;;; store 16 bits num, ms byte is 0's
[ rts label7]]

[main]
[ldb #0][pshu b]
[pulu b][stb area]
[ldb area] [jsr send]
[ldb #1] [jsr rec][pulu b]
[stb va]
[ldb #2] [jsr rec][pulu b]
[stb vb]
[ldb va] [pshu b]
[lda vb] [pshu a]
[pulu b] [pulu a] [stb addr1] [suba addr1] [pshu a]
[pulu b] [stb delta]
[ldb delta] [jsr send]
;;; for loop
[ldb #1] [pshu b]
[pulu b] [stb vi]
[lda #10000] [pshu a]
[label8 lda vi] [andcc #$0] [cmpa ,u] [lbgt 19]
[lda va] [pshu a] [pshu a]
[pulu a] [pulu b] [mul] [pshu b]
[ldb va] [pshu b]
[pulu a] [pulu b] [stb addr1] [adda addr1] [pshu a]
[ldb #8] [pshu b]
[pulu a] [pulu b] [stb addr1] [adda addr1] [pshu a]
[pulu b] [stb funcval]
[ldb funcval] [jsr send]
[ldb va] [pshu b]
[ldb delta] [pshu b]
```

```
[pulu a] [pulu b] [stb addr1] [adda addr1] [pshu a]  
[inc vi] [bra label18]  
[label19 dec vi]
```

Appendix H

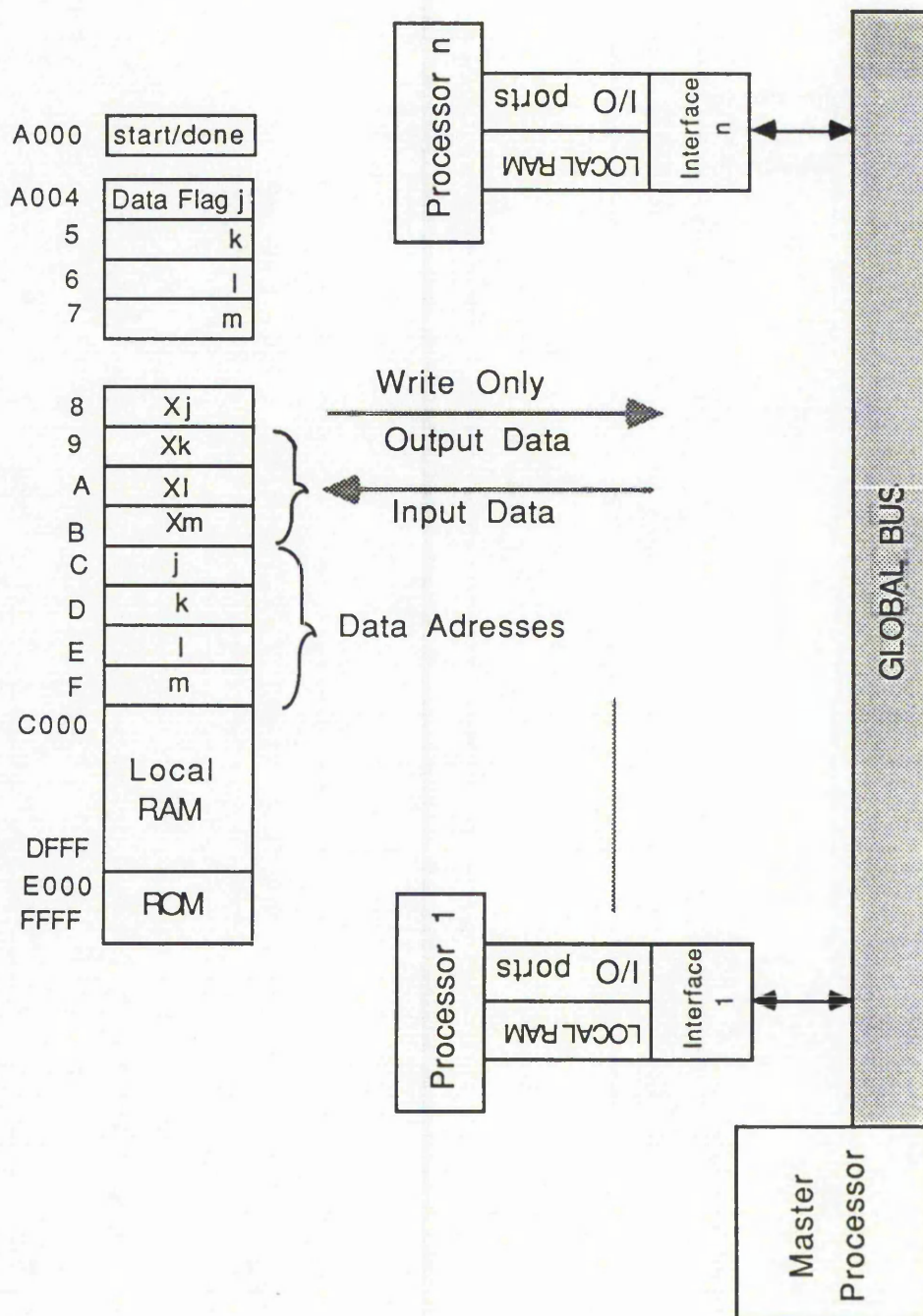
Programmers Model of MINNIE

Master Processor Address Space

Address	Fuction	Description
7000	STP	step for enable
7001	SELECT NA	to r/w NA(Node Address) to node
7002	NAEN	to switch RAM through
7003	RESg	reset of interfaces globaly
7004	START	starting of system
7005	NERES	node enable reset
7006	CY	cyclic
7007	NCY	non-cyclic
7008	GBB	global bus busy
7009	ENABLE	for access of NAs
700A	SA	start allocator

Programmers Model Diagram

The diagram below shows the functional units of MINNIE and the memory map of each node.



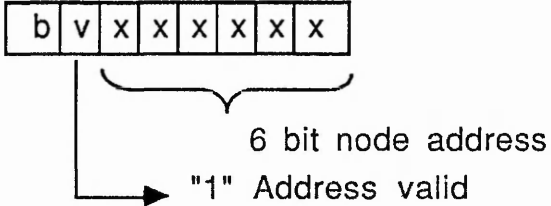
Set-up sequence

1. set GBB to 1
2. set RESg to 0
3. set NAs (see Step 3: Set-up NA)
4. write program/data to node (see Step 4: Program/Data to Node)
5. set SA to 1
6. select cyclic or non-cyclic
7. reset GBB
8. monitor processor "Done" line
9. if cyclic start again, count cycle
10. when cycles are finished or non-cyclic then master is interrupted and reads local memory for data collection (see steps 4.1-4.7).

Step 3: Set-up NA

step	operation
3.1	GBB set
3.2	set ENABLE
3.3	set STP
3.4	reset ENABLE
3.5	read/write NA as data (6-bit node address)
3.6	move to next node (stp 3.3)
3.7	do step 3.5
3.8	repeat 3.6 and 3.7 as necessary
3.9	when done reset ENABLE within system

Step 4: Program/Data to node

step	operation
4.1	GBB set
4.2	switch in memory map to global processor map (FF8C set to 1C)
4.3	set NAEN
	 <p style="text-align: center;">6 bit node address "1" Address valid</p>
4.4	Access memory as usual
4.5	repeat from 4.3 for other nodes
4.6	reset NAEN to switch off all memory switches
4.7	switch memory map off (FF8C set to FC)
4.8	reset GBB