

LOUGHBOROUGH
UNIVERSITY OF TECHNOLOGY
LIBRARY

AUTHOR/FILING TITLE

MUSLIM, OK

ACCESSION/COPY NO.

03352602

VOL. NO.

CLASS MARK

LOAN COPY

- 5 JUL 1991

12 NOV 1990

- 5 JUL 1991

- 1 NOV 1990

- 5 JUL 1991

003 3526 02



THE INSTRUCTION SYSTOLIC ARRAY (ISA)
AND SIMULATION OF PARALLEL ALGORITHMS

BY

OSSAMA KADOM MUSLIH, B.Sc., M.Ph.

A Doctoral Thesis

submitted in partial fulfilment of the requirements

for the award of Doctor of Philosophy

of the Loughborough University of Technology

September, 1989.

Supervisor: PROFESSOR D.J. EVANS, Ph.D., D.Sc.,

Department of Computer Studies.

© OSSAMA KADOM MUSLIH, 1989.

Loughborough University of Technology Library	
Date	Sept 90
Class	
Acc No.	03852602

49315516

*To the memory of my
father and mother,*

my wife, INTIHA,

and my children, OMAR and OLLA.

ACKNOWLEDGEMENTS

The author wishes to express his sincere thanks to Professor D.J. Evans for his guidance, suggestions and advice throughout the course of the research and preparation of this thesis.

The author also acknowledges the Iraqi Government (Ministry of Higher Education and Scientific Research) for their financial support.

Thanks also to my dearest wife, Intiha, for her constant support during the course of this work.

Finally, thanks to Dr. G.M. Megson, for his help at the beginning of the research.

THE INSTRUCTION SYSTOLIC ARRAY (ISA)

AND SIMULATION OF PARALLEL ALGORITHMS

ABSTRACT

Systolic arrays have proved to be well suited for Very Large Scale Integrated technology (VLSI) since they:

- Consist of a regular network of simple processing cells,
- Use local communication between the processing cells only,
- Exploit a maximal degree of parallelism.

However, systolic arrays have one main disadvantage compared with other parallel computer architectures: they are special purpose architectures only capable of executing one algorithm, e.g., a systolic array designed for sorting cannot be used to form matrix multiplication.

Several approaches have been made to make systolic arrays more flexible, in order to be able to handle different problems on a single systolic array.

In this thesis an alternative concept to a VLSI-architecture the Soft-Systolic Simulation System (SSSS), is introduced and developed as a working model of virtual machine with the power to simulate hard systolic arrays and more general forms of concurrency such as the SIMD and MIMD models of computation.

The virtual machine includes a processing element consisting of a soft-systolic processor implemented in the virtual machine language. The processing element considered here was a very general element

which allows the choice of a wide range of arithmetic and logical operators and allows the simulation of a wide class of algorithms but in principle extra processing cells can be added making a library and this library be tailored to individual needs.

The virtual machine chosen for this implementation is the Instruction Systolic Array (ISA). The ISA has a number of interesting features, firstly it has been used to simulate all SIMD algorithms and many MIMD algorithms by a simple program transformation technique, further, the ISA can also simulate the so-called wavefront processor algorithms, as well as many hard systolic algorithms. The ISA removes the need for the broadcasting of data which is a feature of SIMD algorithms (limiting the size of the machine and its cycle time) and also presents a fairly simple communication structure for MIMD algorithms.

The model of systolic computation developed from the VLSI approach to systolic arrays is such that the processing surface is fixed, as are the processing elements or cells by virtue of their being embedded in the processing surface.

The VLSI approach therefore freezes instructions and hardware relative to the movement of data, with the virtual machine and soft-systolic programming retaining the constructions of VLSI for array design features such as regularity, simplicity and local communication, allowing the movement of instructions with respect to data. Data can be frozen into the structure with instructions moving systolically. Alternatively both the data and instructions can move systolically around the virtual processors, (which are deemed fixed relative to the underlying architecture).

The ISA is implemented in OCCAM programs whose execution and output implicitly confirm the correctness of the design.

The soft-systolic preparation comprises of the usual operating system facilities for the creation and modification of files during the development of new programs and ISA processor elements. We allow any concurrent high level language to be used to model the soft-systolic program. Consequently the Replicating Instruction Systolic Array Language (RISAL) was devised to provide a very primitive program environment to the ISA but adequate for testing. RISAL accepts instructions in an assembler-like form, but is fairly permissive about the format of statements, subject of course to syntax.

The RISAL compiler is adopted to transform the soft-systolic program description (RISAL) into a form suitable for the virtual machine (simulating the algorithm) to run.

Finally we conclude that the principles mentioned here can form the basis for a soft-systolic simulator using an orthogonally connected mesh of processors. The wide range of algorithms which the ISA can simulate make it suitable for a virtual simulating grid.

CONTENTS

	<u>PAGE</u>
ACKNOWLEDGEMENTS	i
ABSTRACT	ii
<u>CHAPTER 1:</u> FUNDAMENTALS OF PARALLEL COMPUTER ARCHITECTURE	
1.1 Introduction	1
1.2 Main Motivations	4
1.3 Design Classifications	6
1.3.1 Flynn's High-Speed Parallel Computers Classification	6
1.3.2 Shore's Classification	13
1.3.3 Other Classification Approaches	16
1.4 Pipelined Computers	20
1.5 Data-Flow Computers	27
1.6 Array Processors	33
<u>CHAPTER 2:</u> PARALLEL ARCHITECTURES - A VLSI APPROACH	
2.1 Introduction to the VLSI Technology Paradigm	38
2.2 Fundamental Architectural Concepts in Designing Special Purpose VLSI Computing Structures	41
2.2.1 Systolic Arrays	45
2.2.2 Wavefront Arrays	49
2.3 VLSI-Oriented Architectures	52
2.3.1 The WARP Architecture	52
2.3.2 The CHIP Architecture	53
2.3.3 INMOS Transputers and OCCAM	58
2.3.4 Simulation of Systolic Arrays	61

	<u>PAGE</u>
2.4 MIMD Architecture Design - The Sequent Balance System	67
2.4.1 MIMD Hardware Organisation	67
2.4.2 The Sequent Balance 8000 System	75
 <u>CHAPTER 3:</u> THE INSTRUCTION SYSTOLIC ARRAY (ISA) - A PARALLEL ARCHITECTURE FOR VLSI	
3.1 The Instruction Systolic Array (ISA)	80
3.2 The Instruction Systolic Array and Its Relation to Other Models of Parallel Computers	85
3.2.1 Basic Definitions	85
3.2.2 A Simple Example Program	91
3.2.3 Relationships Between ISA, IBA, and PA	97
3.2.4 Relationship of ISA to Standard Models of Parallel Computers	104
3.3 A Comparison-Based Instruction Systolic Array	107
3.3.1 ISA Construction	107
3.3.2 Example Programs on the ISA	110
3.4 To Sort by the Instruction Systolic Array	115
3.4.1 Introduction	115
3.4.2 One Dimensional Sorting Methods	116
3.4.3 The Two Dimensional Sorting Algorithm	121
3.4.4 The Algorithm on the ISA	122
3.5 Additional Algorithms Solution by Using the Instruction Systolic Array (ISA)	127
3.5.1 Finding the Generalized Matrix Inversion	127
3.5.2 Top-Down Designs of Instruction Systolic Arrays for Polynomial Interpolation and Evaluation	128

	<u>PAGE</u>
3.5.3 Finding Transitive Closure	131
3.5.4 Finding all Cut-Points	131
3.6 The Singe Instruction Systolic Array (SISA) - Variants of the ISA Model	133
<u>CHAPTER 4:</u> THE SOFT-SYSTOLIC SIMULATION SYSTEM (SSSS)	
4.1 Basic Definitions of the System	136
4.2 System and Machine Preparation	139
4.3 The Virtual Machine	146
4.3.1 The Instruction Systolic Array (ISA) Network	147
4.3.2 The Processing Element (PE)	159
<u>CHAPTER 5:</u> THE IMPLEMENTATION OF THE REPLICATING INSTRUCTION SYSTOLIC ARRAY LANGUAGE (RISAL) AND SYSTEM TESTING	
5.1 Introduction	168
5.2 Language Design Principles	171
5.3 The RISAL Compiler	181
5.4 Soft-Systolic Simulation Architecture and Testing	193
<u>CHAPTER 6:</u> THE SOFT-SYSTOLIC SIMULATION SYSTEM (SSSS) FOR VARIOUS ALGORITHMS	
6.1 Basic Mathematics	207
6.2 Matrix Applications Using (SSSS)	217
6.2.1 4*4 Matrix Transpose	217
6.2.2 4*4 LU Decomposition	221
6.2.3 Matrix-Vector Multiplication	225
6.2.4 Matrix-Matrix Multiplication	228

	<u>PAGE</u>
6.3 The Solution of Linear Systems Using SSSS	235
6.4 Finding the Generalized Inverse of a Rectangular Matrix Using SSSS	243
6.5 Some Applications to the Generalized Inverse of a Rectangular Matrix Using SSSS	248
6.5.1 The Solution of a Homogeneous System of Equations	249
6.5.2 The Most General Solution of a System of Equations	252
6.6 Deletion from a Heap Sort Using SSSS	257
6.7 Hermite Polynomial Interpolating and Evaluation Using SSSS	264
<u>CHAPTER 7:</u> SUMMARY AND CONCLUSIONS	280
REFERENCES	289
APPENDICES:	
APPENDIX I: LOUGHBOROUGH OCCAM COMPILER VERSION 5.0 DOCUMENTATION	298
APPENDIX II: THE SOFT-SYSTOLIC SIMULATION SYSTEM (SSSS) PROGRAM LISTINGS	317
APPENDIX III: RISAL PROGRAM LISTINGS	332

CHAPTER 1

FUNDAMENTALS OF PARALLEL COMPUTER

ARCHITECTURE

1.1 INTRODUCTION

The information revolution, has had the most tremendous impact on both technology and our society. This fast developing revolution has just recently started to migrate towards a new era - the knowledge revolution, by giving birth to the Fifth Generation of Super Computers (FGSC). These have in fact changed our lifestyles, our educational programs and most of all many professional careers.

Amongst the huge numbers of computer applications which range from the simple personal computer games to the weather forecasting calculation and satellite transmission programs, there are many that require the use of large amounts of computational time. In an attempt to meet the challenging problem of providing fast and economical computation, Large-Scale Parallel Computers were developed. In fact, until recently computational speed was derived only from the development of faster electronic devices.

In the late 1960s, Integrated Circuits (ICS) were used in computer design and were followed by Large Scale Integrated (LSI) techniques. The Very Large-Scale Integrated Circuits (VLSI), developed seven years ago, are currently being used in the design of very high speed special and general purpose computer systems.

Until seven years ago, the current state of electronic technology was such that all factors affecting computational speed were almost minimised and any further computational speed increase could only be achieved through both increased switching speeds and increased circuit density.

Due to the physical laws, the intended breakthrough seemed unlikely to be achieved mainly because we are fast approaching the

limits of optical resolution. Hence, even if switching times are almost instantaneous, distances between any two points may not be small enough to minimise the propagation delays and thus improve computational speed. Therefore, the achievement of even faster computers is conditional by the use of new approaches that do not depend on breakthrough in device technology but rather on imaginative applications of the skills of computer architecture.

Obviously one approach to increasing speed is through parallelism. The ideal objective is to create a system containing P processors, connected in some cooperating fashion, so that it is P times faster than a computer with a single processor. These parallel computer systems or multiprocessors as they are commonly known, not only increase the potential processing speed, but they also increase the overall throughput, flexibility, reliability and provide for the tolerance of processor failures.

Hockney and Jesshope [Hockney 1981] summarised the principle ways of introducing parallelism at the hardware level of the computer architectures as:

1. The application of pipelining - assembly line - techniques in order to improve the performance of the arithmetic or control units. A processor is decomposed into a certain number of elementary subprocesses each of which being capable of executing on dedicated autonomous units.
2. The provision of several independent units, operating in parallel, to perform some basic fundamental functions such as logic, addition or multiplications.

3. The provision of an array of processing elements performing simultaneously the same instruction on a set of different data where the data is stored in the processing elements (PE) private memory.
4. The provision of several independent processors, working in a co-operative manner towards the solution of a single task by communicating via a shared or common memory, each one of them being a complete computer, obeying its own stored instructions.

The following sections will cover a wide selection of the principle significant parallel computer architectures, which differ sufficiently from each other, the pipeline, SIMD, MIMD, data-flow and VLSI systems, to illustrate alternative hardware and software approaches.

1.2 MAIN MOTIVATIONS

During the last decade the multiple processor approach has tailored a set of long sought after motivating goals in order to satisfactorily meet many of the challenging system design requirements. In reviewing some aspects of parallel processing systems, one finds that while the hardware is improving at a fast rate, the software tools to take advantage of the provided benefits are only slowly forthcoming; a fact that affects the design motivations mentioned below.

Since the early developed multiple processing systems, the system characteristics that have motivated the continued development in this field have not changed much. The most significant of these are increased throughput, improved flexibility and reliability. Since none of these goals is numerically specified (i.e. they are all qualitative goals), it is not surprising that the design of the future "supercomputers" will also be motivated by the same objectives as today's parallel computers. However, the improvements of some or all of these specifications must ultimately result in an improved overall system performance, usually measured on the basis of cost effectiveness.

The system throughput can be used to mean several different characteristics such as the potential number of bits processed per time-unit, the number of memory transfers per time unit or the maximal number of programs that can be handled at the same time. However, it is usually used nowadays to describe the long-turnaround of a program in a multiprocessing environment. The multiple processor approach is a cost-effective solution to the achievement of most of these goals. The use of several cooperating processing units can considerably increase the system throughput which could not be matched

by a uniprocessor system with enhanced logic circuitry.

Literally, flexibility means the ease in changing the system configuration to suit new conditions and the use of more than one processor has greatly increased the system potential flexibility since it offers the ability to expand the memory space, the number of processing units and even the software facilities in order to meet the new demands. This flexibility may also be used to justify the increased reliability of the system.

Broadly speaking, the reliability is related to two different system aspects required by different applications. The first one is the system availability which is defined by the requirement that the system should remain available even in the case of a malfunctioning unit. An example of this is the computer controlled telephone switching board. The system integrity is the second one and it is defined by the requirement that the information contained within should be "protected" against any defection or corruption (e.g. in a banking system).

Concluding, since all the system characteristics that have motivated the development of the parallel processor computers are not described quantitatively, any new major system concept has been claimed by its proponents as the ultimate solution to achieving these motivating goals. In fact, the same motives were behind the follow-up to the parallel processing systems, the VLSI architectures.

1.3 DESIGN CLASSIFICATIONS

As a result of the introduction of various forms of parallelism which has proved to be an effective approach for increasing computational speed, several competitive computer architectures were constructed but there was little evidence as to which design was superior, nor was there sufficient knowledge on which to make a careful evaluation. Researchers helped the study of high-speed parallel computers by attempting to classify all the proposed computer architectures, or at least those which have been already well established. A brief presentation of the concepts of the architectural taxonomy given by different researchers, especially by the two pioneers, Flynn [Flynn 1966] and Shore [Shore 1973], follows below. However Flynn's classification scheme is too broad: since it lumps all parallel computers except the multiprocessor into the SIMD class and draws no distinction between the pipelined computer and the processor array which have entirely different computer architectures. These classifications have been widely referenced and their corresponding terminology has greatly contributed to the formation of the Computer Science vocabulary.

1.3.1 Flynn's High-Speed Parallel Computer Classification

Based on the dependent relation between instructions that are propagated by the computer and the data being processed, Flynn explored theoretically some of the organisational possibilities for large scientific computing machinery before attempting to classify them into four broad classes. We shall briefly review his theoretical concepts leading to the actual grouping of the high-speed parallel computers.

For convenience, he defined the instruction stream as a sequence of instructions to be processed by the computer and the data stream as a set of operands, including input and partial or temporary results. Also, two additional useful concepts were adopted, bandwidth and latency. By bandwidth he expressed the time-rate of occurrences, and latency is used to express the total time between execution of response of a computing process on a particular data unit. Particularly, for the former notion, computational or execution bandwidth is the number of instructions processed per second and storage bandwidth is the retrieval rate of the data and instruction from the store (i.e. memory words per second).

By using the two former definitions, Flynn categorized the almost theoretically defined computer organisations depending on the multiplicity of the hardware provided to service the instruction and data streams. The word "multiplicity", which was intentionally used to avoid the ubiquitous and ambiguous term "Parallelism", refers to the maximum number of simultaneous instructions or data in the same phase of execution at the most constrained component of the organisation.

Flynn observed that as a consequence of the above definitions four classes emerged naturally, being characterized from the multiplicity or not of the instruction and data streams:

- i) Single Instruction Stream - Single Data Stream (SISD)
- ii) Single Instruction Stream - Multiple Data Stream (SIMD)
- iii) Multiple Instruction Stream - Single Data Stream (MISD)
- iv) Multiple Instruction Stream - Multiple Data Stream (MIMD)

The SISD computer [e.g. most of the general purpose machines such as

IBM STRETCH, DEC PDP-11 (Serial or unpipelined) and CDC 6600 series, IBM 360/90 series pipelined], is nothing more than the ordinary serial computer (the von-Neumann type computer). Even though, the CDC 6600 and IBM 360/90 series achieve their power by overlapping various sequential decision processes which make up the execution of the instruction (confluent SISD), there still remains an essential constraint of this type of organisation, namely the decoding of one instruction per unit time. In Figures 1.1 and 1.2 we see a SISD organisation, and the concurrency and instruction processing respectively.

The SIMD type structure, proposed by [Unger 1958], Slotnick [Slotnick 1962] is created by replicating the data stream on which the single instruction stream acts simultaneously thus theoretically increasing the throughput by a factor almost equal to the number of data streams. Several factors, such as data conflict and data communication problems tend to degrade the expected performance. Solomon and ILLIAC IV are two examples of such a computer.

The third, MISD type class of parallel computers, the organisation of which is outlined in Figure 1.3, is by all means the least realistic one compared to the others since no examples of any well established organisation have yet been proposed. In this class, a forwarding procedure of data flowing through the Execution Units was forced. Thus, the data stream presented to Execution Unit 2 is the resultant of Execution Unit 1 operating its instruction on the source data stream. The instruction performed on any Execution Unit can be one of the three following types: fixed, semi-fixed or variable. It may be fixed such that the interconnection of units must be flexible

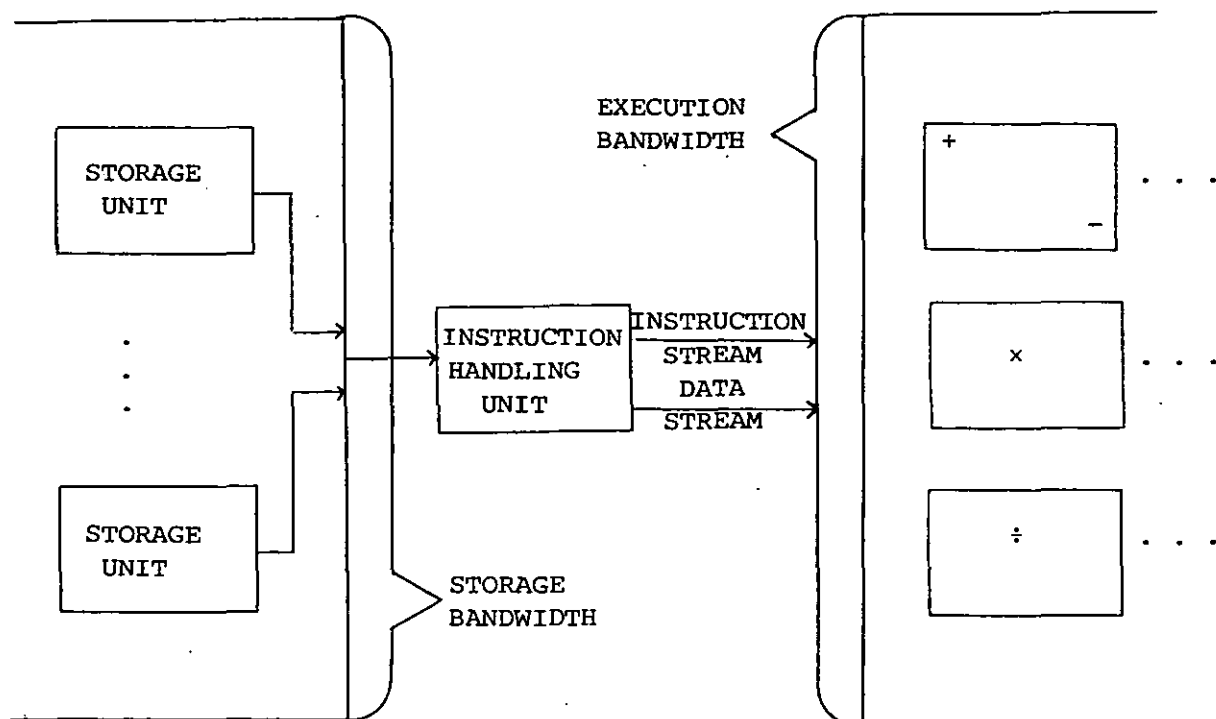


FIGURE 1.1: Flynn's SISD Computer Organisation

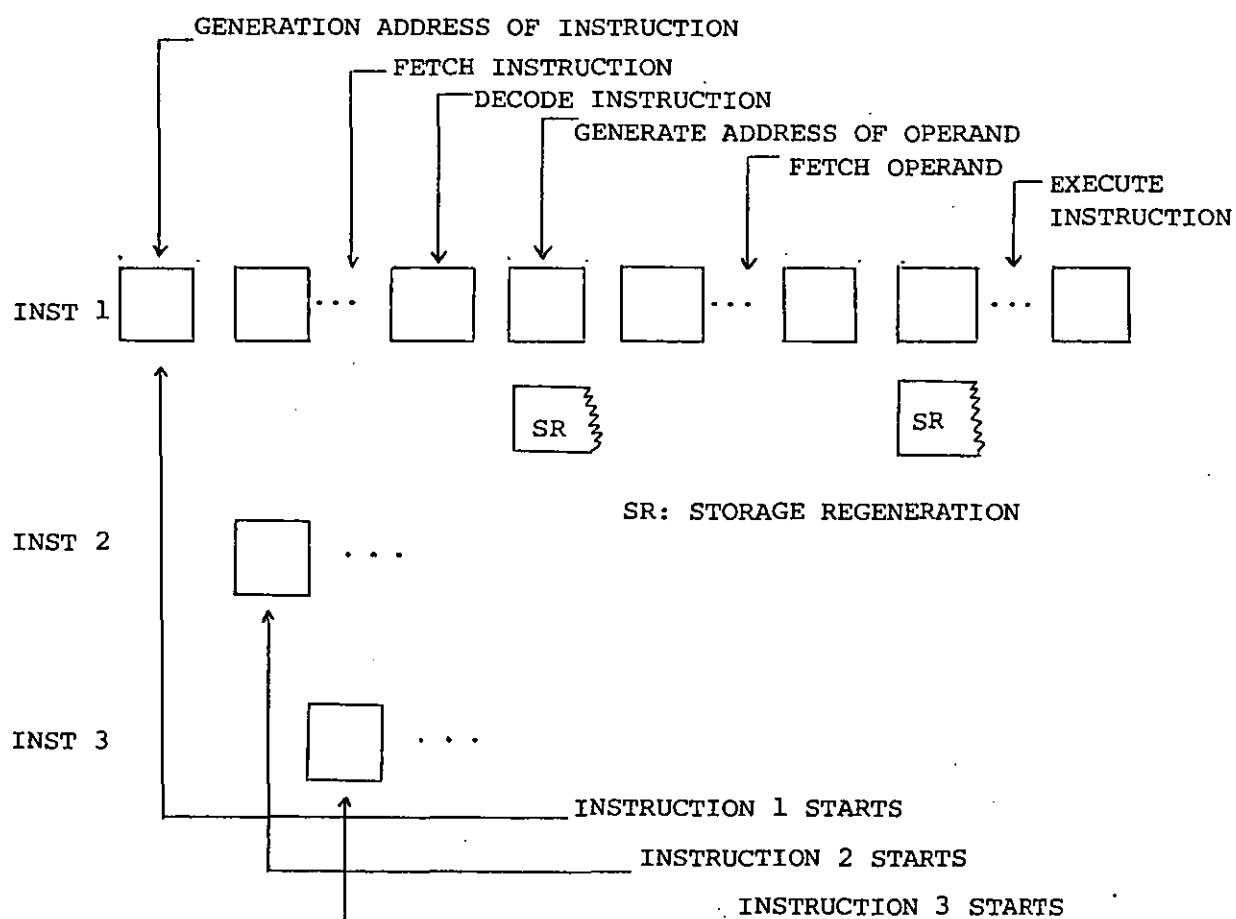


FIGURE 1.2: Concurrency and Instruction Processing

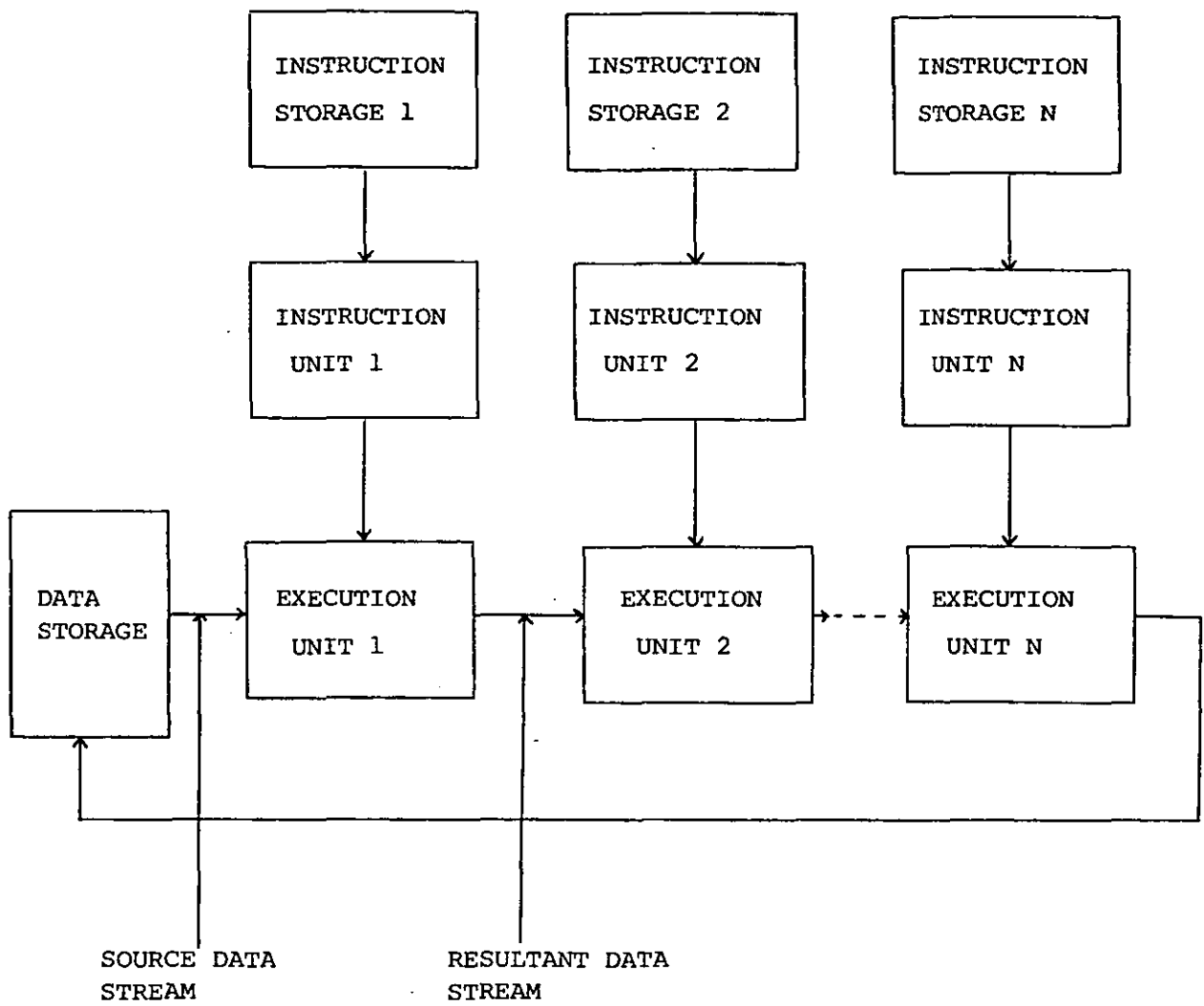


FIGURE 1.3: A MISD Organisation

semi-fixed such that the function of any unit is fixed for one pass of the data or variable meaning that the execution of a stream of instructions can take place at any point on the single data stream. Consequently this arrangement suggests that only the first processing component faces the source data stream whereas the remaining Units are processing derivations of the data from previous components. By combining parallelism in both the instruction and data streams a MIMD type of structure is thus obtained. This computer possesses N independent executing units (processors), each of which is a complete computer on its own (has arithmetic and logic capabilities and local data storage), with processors connected together to provide means for cooperation during a computation phase.

Most serial main frames could be classified as MIMD computers since they include many data channels, such as Direct Memory Access (DMA) which are, in a sense, independent processors. Thus, a computer with one or two data channels is indeed a MIMD parallel computer, but the MIMD is commonly accepted to refer to large computers with possibly several identical processors such as Cmp [Wulf 1972], Cm* [Swan 1977]. Of particular interest, the Balance 8000 parallel computer system which is in the Department of Computer Studies at Loughborough University of Technology is an example of this class, this machine is described in detail in Chapter 2.

Resuming, Flynn classified computer systems into four broad classes (Figure 1.4) depending on the multiplicity or not of the instruction stream and data stream. Due to the fact that the actual architectural details of the machines were not taken into account, his taxonomy was somehow obscure since one finds that there is no

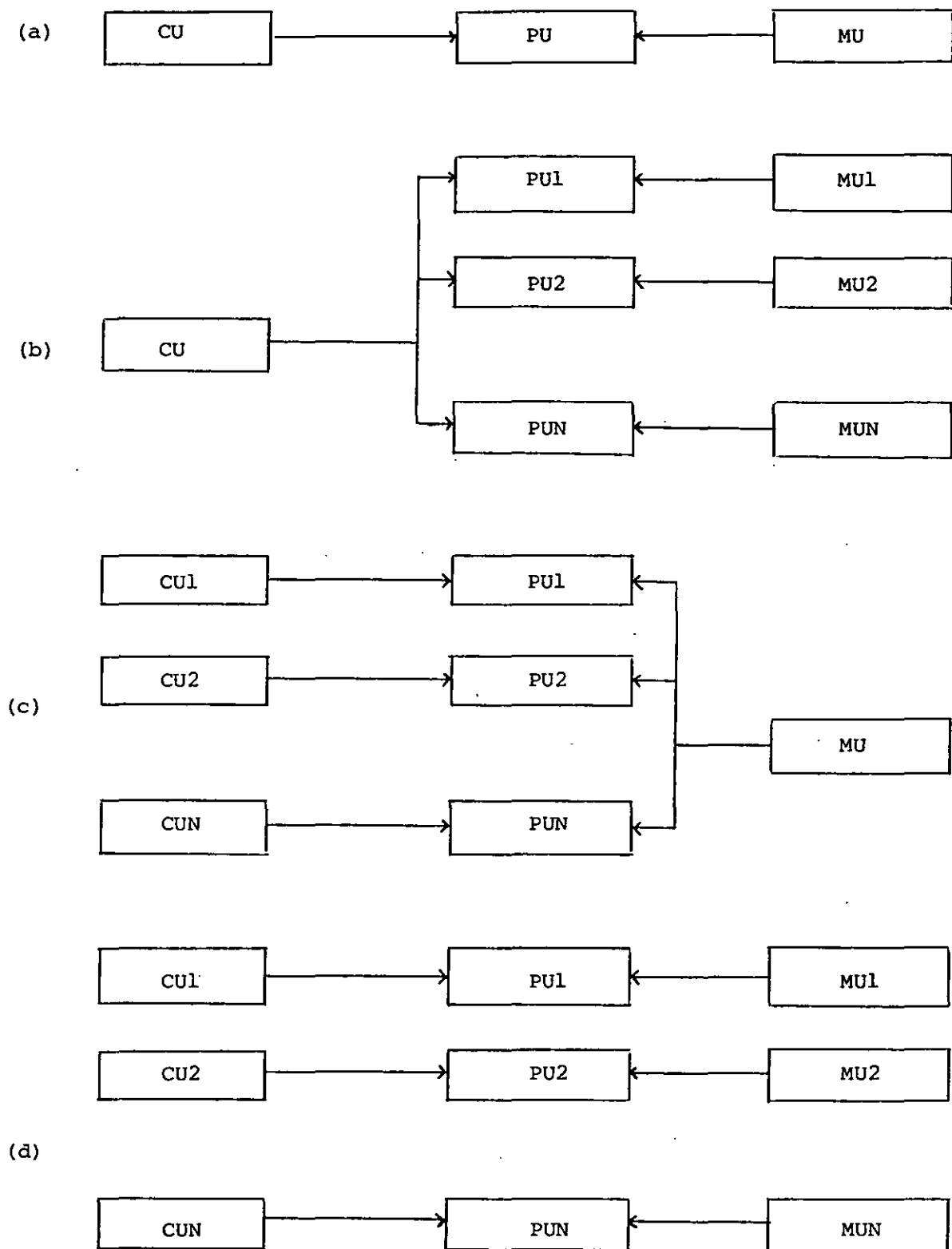


FIGURE 1.4: Flynn's Computer Organisation Classes:

(a) SISD, (b) SIMD, (c) MISD, and (d) MIMD

where CU, PU, and MU refer to control, processing and memory unit respectively

apparent distinctive differences between classes (MIMD class exempted). Consequently, pipelined and array processor computers are considered similar, although they are two completely different architectures.

Also, the meaning of the data streams, as used by Flynn, has caused many ambiguities due to the fact it does not make a distinctive difference between a single stream of vectorised data and a multiple scalar stream.

Consequently, in the sections, the SIMD and pipelined computers are considered to be two distinct classes along with the multiprocessor category.

1.3.2 Shore's Classification

Classification of parallel computer systems based on their constituent hardware components was observed by Shore [Shore 1973]. Accordingly, all current existing computer architectures were categorised into six different classes which are schematically shown in Figure 1.5.

The first machine (I), [e.g. CDC 7600 a pipelined scalar computer, CRAY 1, a pipelined vector computer] which is the conventional serial Von-Neuman-type organisation, consists of an Instruction Memory (IM), a single Control Unit (CU), a Processing Unit (PU) and a Data Memory (DM). The main source of power increase comes from the processing unit which may consist of several functional units, pipelined or not and all bits of a single word are read in order to be processed simultaneously (Horizontal PU).

A second alternative machine (II) is obtained from the first one

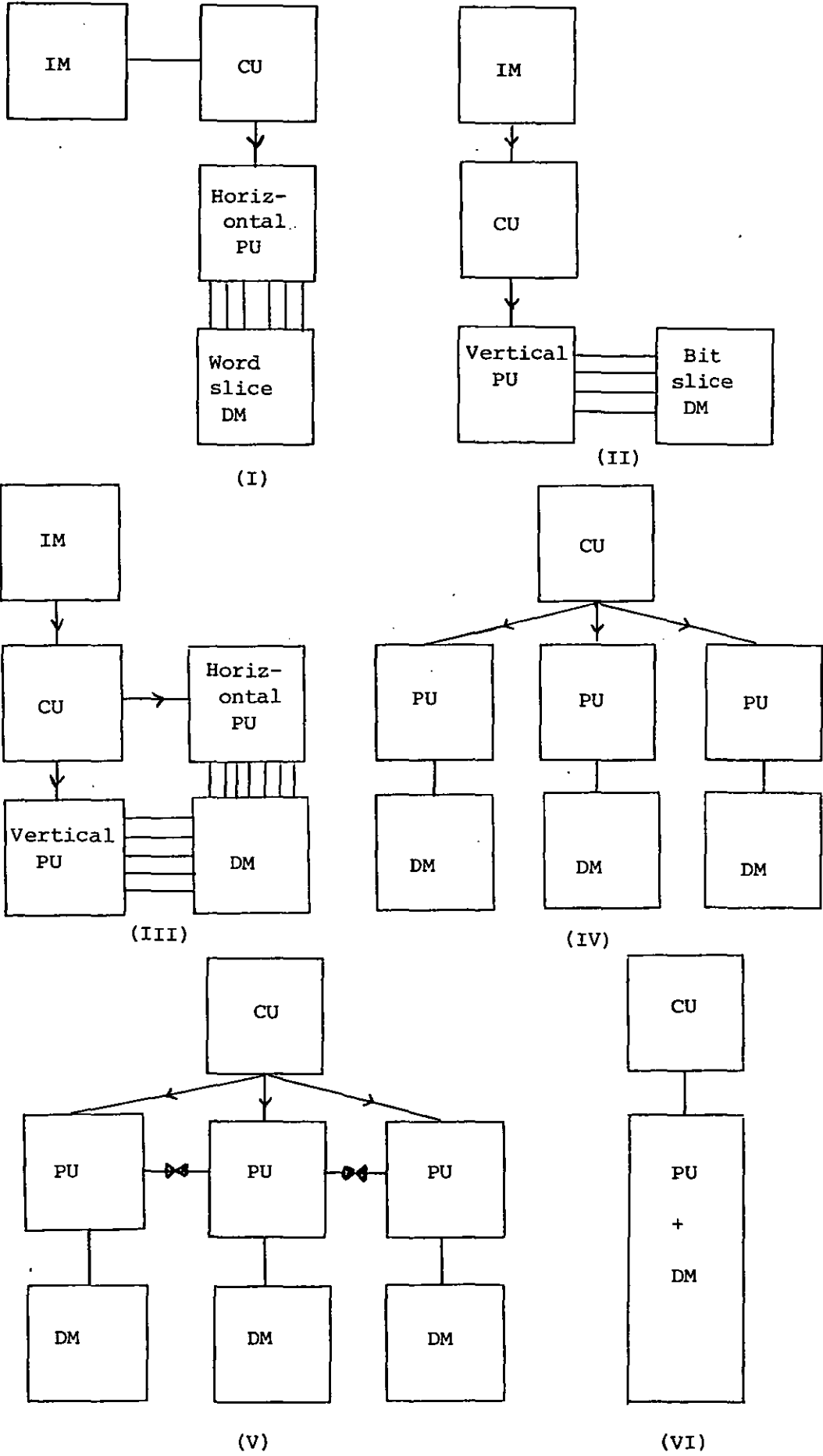


FIGURE 1.5: The Configuration of the Six Machine Classes

by simply changing the way data is read from the data memory. Instead of reading all bits of a single word as (I) does, machine (II) reads a bit from every word in the memory, i.e. bit serially, but word processing is parallel. In other words, if the memory area is considered as a two dimensional array of bits, with each word occupying an individual row, then machine (I) reads horizontal slices whereas machine (II) reads vertical slices.

A combination of the two above machines yields machine (III). This means that machine (III) has two processing units, a horizontal and a vertical one and is capable of processing data in either of the two directions. The ICL DAP could have been a favourable candidate for this class if only it had separate processing units to offer this capability. An example of this organisation is the Sanders Associates OMEN 60 Series of computer [Higbie 1972].

Machine (IV) consists of a single control unit and many independent processing elements, each of which has a processing unit and a data memory. Communication between these components is restricted to take place only through the control unit. A good example of this machine is the PEPE system.

If however, additional limited communication is allowed to take place among the processor elements in a nearest-neighbour fashion, then machine (V) is conceived. Thus, communication paths between the linearly connected processors offer for any processor in the array the possibility to access data from its immediate neighbour's memories, as well as its own. An example of this machine type is the ILLIAC IV, which provides a short cut communication to every eight surrounding

processing elements.

The Logic-In-Memory-Array (LIMA) is Shore's last class of computer organisation. The main difference in machine (VI) and the previous one is that the processing unit and the data memory are no longer two individual hardware components, but instead they are constructed on the same IC board. Examples range from simple associative memories to complex associative processors.

It is observed that, generally speaking, Shore's classification, compared with Flynn's, does not offer anything new, but only a sub-categorisation of the obscure SIMD class given by Flynn, except for machine (I) which is an SISD-type computer. Again, as with Flynn's categorisation, pipelined computers do not belong to a well specified class, that represents their hardware characteristics, but on the contrary they are mixed up with unpipelined scalar computers.

1.3.3 Other Classification Approaches

This paragraph gives a brief note on some other classification approaches of less significant importance compared to the former two and which are based mainly on the concept of parallelism.

One of the taxonomies, based on the amount of parallelism involved in the control unit, data streams and instruction units was suggested by Hobbs et al [Hobbs 1970] in 1970. They distinguished parallel computers into multiprocessors, associative processors, array processors and functional processors.

Another classification, due to Murtha and Beadles [Murtha 1964] was based upon the parallelism properties. An attempt to underline

the main significant differences between the multiprocessors and highly parallel organisations was appreciated. Three main classes for parallel processor systems were identified and they are general-purpose network computers, special-purpose network computers characterised by global parallelism and finally non-global, semi-independent network computers with local parallelism. Furthermore, all these classes, but the last one, were further subcategorised into two subclasses each. Whereas, the first class, the general-purpose one, was subdivided into the general-purpose network computers subclass with centralised common control and the general-purpose network computers subclass, with many identical processors, each being capable of, independent from the others, executing instructions from its own local storage, the second class identified the pattern processors and associative processors subclasses.

Hockney and Jesshope [Hockney 1981] formulated a taxonomy scheme for both serial and parallel computers. The main subdivisions are shown in Figures 1.6 and 1.7 together with a well-known example in each class. Their taxonomy was more detailed than that of Flynn or Shore and took implicit account of pipelined structures. Therefore, the Multiple Instruction class was not considered for further categorisation as with the pipelined and array processor computers. Nevertheless, this scheme if coupled with that of Flynn could well be suited for a general classification of parallel computers.

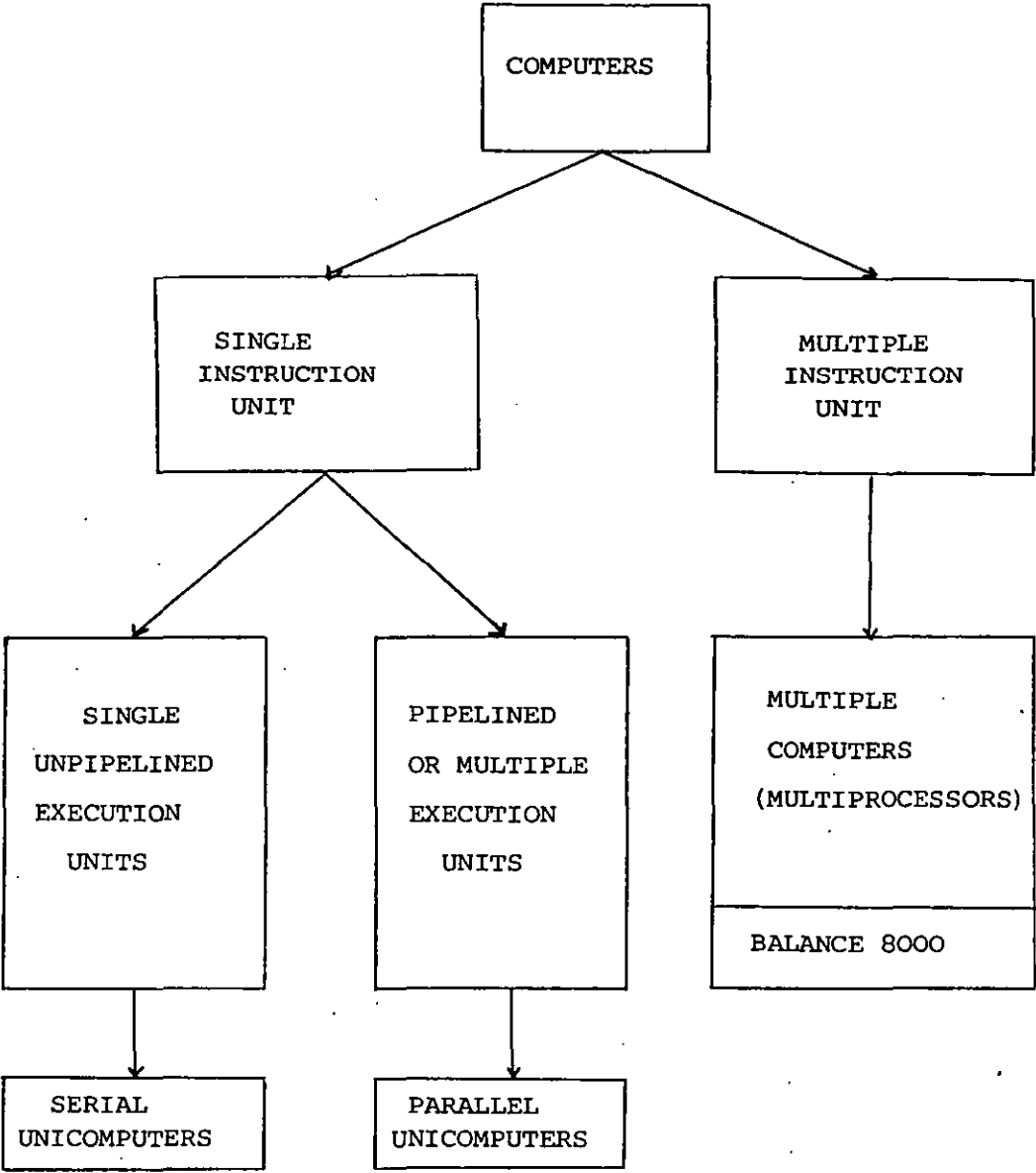


FIGURE 1.6: Structural Classification of Computers

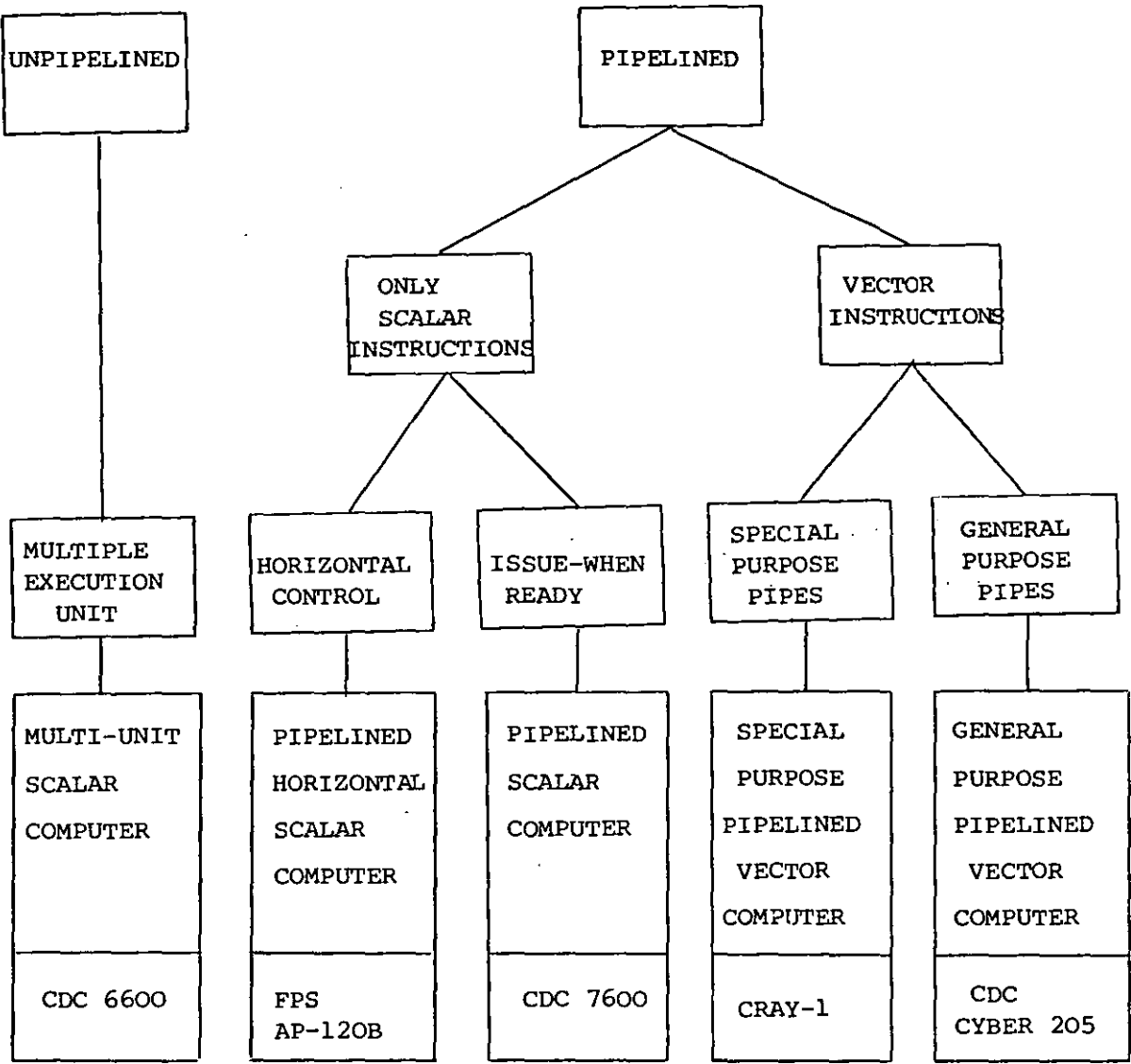


FIGURE 1.7: Parallel Computer Systems Based on Functional Parallelism

1.4 PIPELINED COMPUTERS

The pipeline or vector notion, generally included in the parallelism notion, has been widely exploited since the 1960's when the need for faster and more cost-effective computer systems became critical. Pipelining, a novel architectural design approach, is one form or technique of embedding parallelism or concurrency in a computer system. Although, essentially sequential, this type of computer helps to match the speeds of various subsystems without duplicating the cost of the entire system involved. It also improves system availability and reliability by providing several copies of dedicated subsystems.

In principle, the pipeline is closely related to an industrial assembly line. As in the assembly line, procedure is automatically observed, but it takes time to fill the pipeline before full efficiency per cycle is reached and time to drain the pipeline completely as the last trailing results are collected.

Figure 1.7 depicts the sequential and vector processing taxonomy derived from pipeline computers together with examples of some well known and commercially available computer systems. Although the pipelined computer architectures present somewhat different organisational characteristics when compared to SIMD and MIMD computer architectures, they are of significant interest because of the close connection between algorithms best suited for SIMD and those which achieve great performance on a pipelined computer system.

Pipelined computers achieve an increase in computational speed by decomposing every process into several sub-processes which can be executed by special autonomous and concurrently operating hardware unit. Furthermore pipelining can be introduced at more than one level

in the design of computers. Ramamoorthy [Ramamoorthy 1977] distinguished two pipeline levels, the system level for the pipelining of the processing unit and the subsystem level for the arithmetic pipelining. Particularly Handler [Handler 1982] introduced a third level and distinguished them under the names: macro-pipelining for the program level, instruction pipelining for the instruction level and the arithmetic pipelining for the word level. Others distinguished the instruction pipelining, depending on the control structure in the system, to strict and relax pipelining. A pipe can be further distinguished by its design configurations and control strategies into two forms; it can be either a static or dynamic pipe. Sometimes a pipelined structure is dedicated to a single function, e.g. a pipelined adder or multiplier. In this case it is termed a unifunctional pipe with static configuration. On the other hand, a pipelined module can serve several different functions. Such a pipe is called a multifunctional pipe which can be static or dynamic depending on the number of active configurations (interconnections). If only one configuration is active at any one time, then the pipe is said to be static. Thus any overlapping of operations has to involve the same configuration. However, in a dynamic multifunctional pipe, more than one configuration can be active at any one time, thus permitting a synchronous overlapping on different interconnections.

The simplified model of a general pipelined computer is shown in Figure 1.8 where the processor unit is segmented into M modules, each of which performs its part of the processing and the result appears at the end of the M th segment.

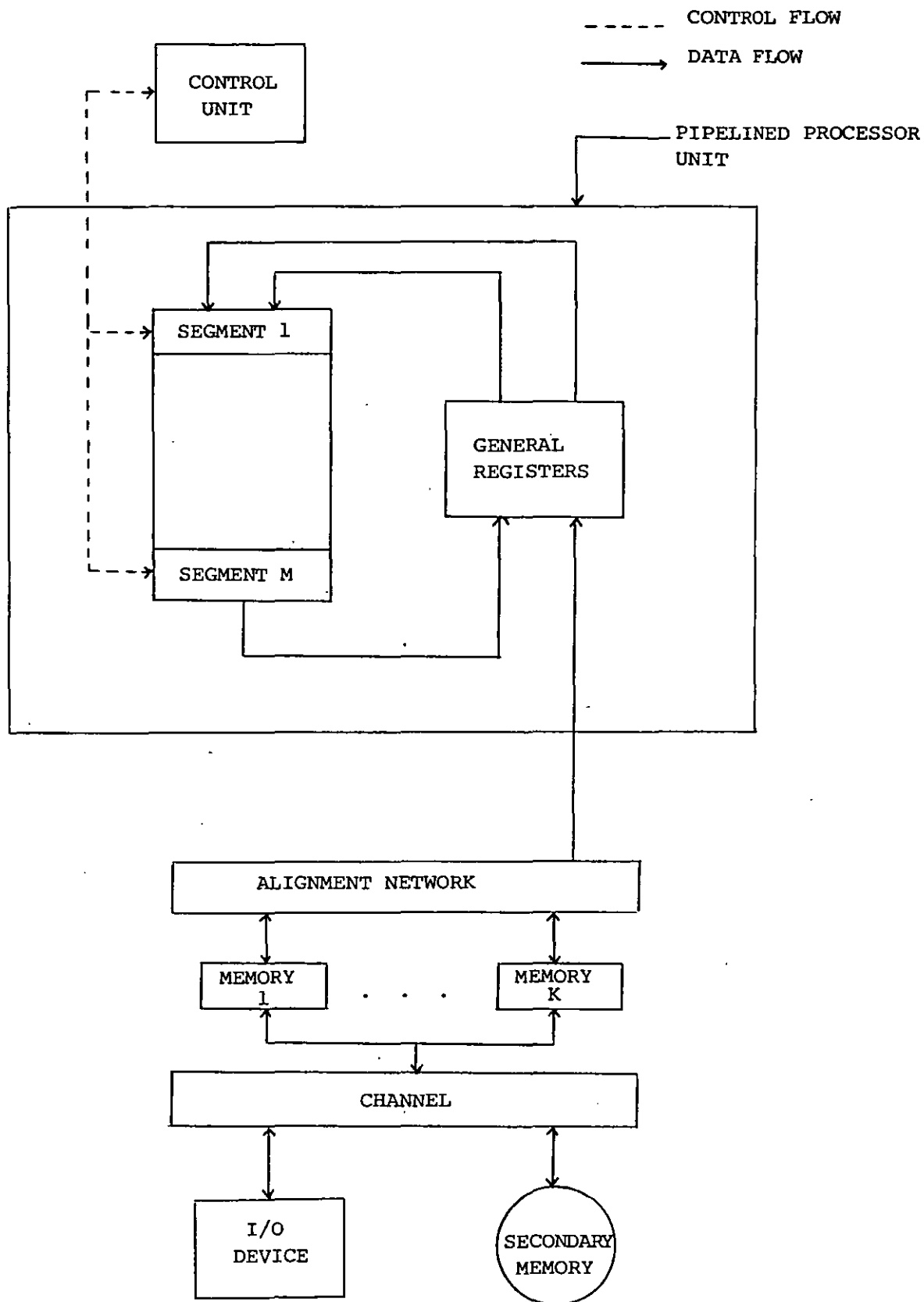


FIGURE 1.8: A Parallel Processor System

The pipelined concurrency, a main characteristic of the simplest pipelining, is exemplified by the process of executing instructions. In Figure 1.9, we considered four modules: Instruction Fetch (IF), Instruction Decode (ID), Operand Fetch (OF) and Execution (E), obtained when segmenting the process of processing instructions. Consequently, if the process is decomposed into four subprocesses and executed on the four-module pipelined system as defined above, then four successive instructions may execute in parallel and independently of each other but at different execution stages: the first instruction is in the execution phase, the second one is in the operand fetching stage, the third is in the instruction decoding phase and lastly, the fourth instruction is in the fetching stage. The overlapping procedure among these individual modules is depicted in Figure 1.10.

However the expected full-potential computation speed increase is not always achieved mainly due to some design and operational problems. These are buffering, busing structure, branching and interrupt handling. A brief discussion of these major design constituents along with the pipelining of the arithmetic functions is included. Their importance and effects which can actually decide the efficiency and performance of the resulting design are also outlined.

Buffering, an essential process to ensure a continuous smooth flow of data through the pipeline segments in the case where variable speed occurs, is virtually a process of storing the results of a segment temporarily before sending them to the next segment. Similar to an industrial assembly line, a segment may occasionally be slowed down for one of many reasons which could prevent the continuous input to the next station. To remedy this problem, a sufficient storage

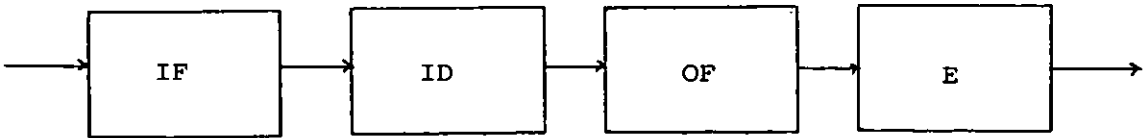


FIGURE 1.9: The Modules of a Pipelined Processor

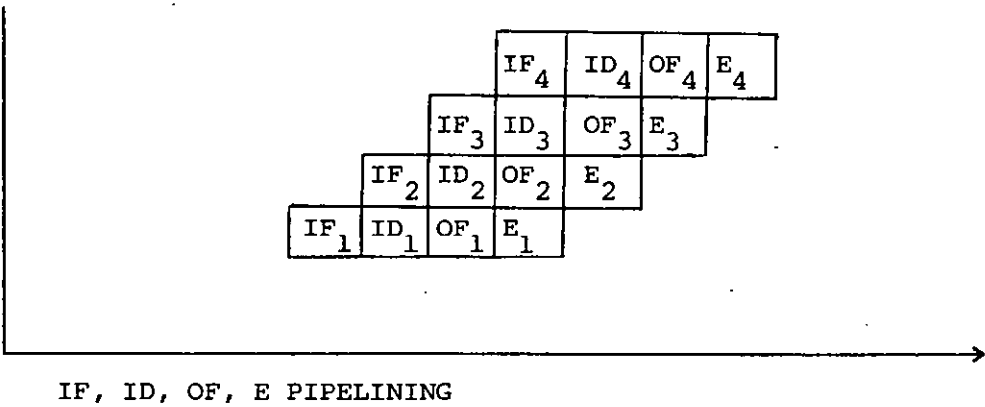
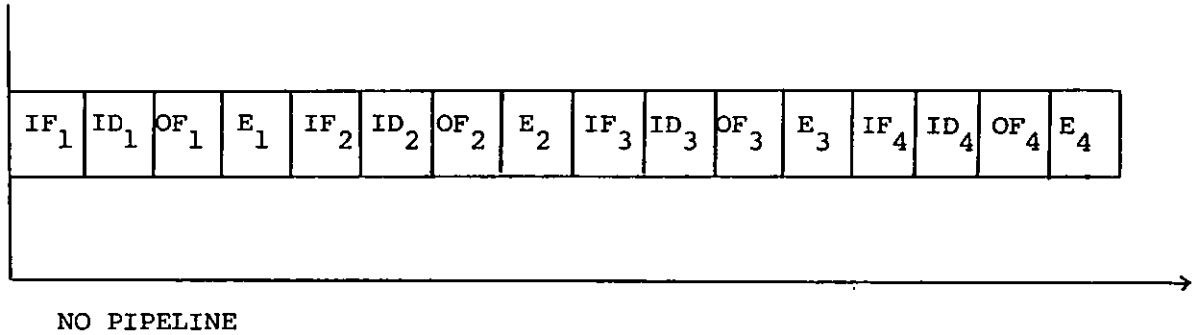


FIGURE 1.10: Space-Time Diagram

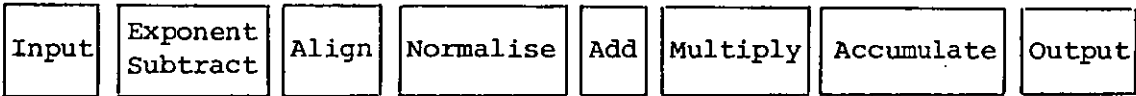


FIGURE 1.11: Modules of an Arithmetic Pipelined Processor

space or buffer is included between this segment and its processor, the latter can continue its operation on other results and transfer them to the provided buffer until it is full.

When the slowing down segment resumes normal service, it clears out its buffer. Perhaps at a faster speed. Consequently buffering may be needed before and after a segment with variable processing time. The inclusion of buffering between segments in a pipelined structure makes the system perform at a relatively constant rate rather than at the speed of slowest component. However full-speed is not always expected to be achieved since buffers have to be stabilised prior to any transfer activity.

In addition to the architectural features of the pipelined processor, the busing structure is equally important in deciding the efficiency of an algorithm to be executed on such a system. Pipelining in essence, refers to the concurrent processing of independent instructions though they may be in different stages of execution due to overlapping. In real life, often, pipelined computers have to deal with dependent or intermixed instructions. With dependent tasks, their input and traversal through the pipe have to be paused before the dependency is tackled. The internal busing structure serves this purpose by routing the results to the requesting segment efficiently, thus reducing the adverse effect of instruction dependency, but still leaving a great burden on the programmer. However, in the case of intermixed instructions, more concurrent processing can take place since the resulting dependency is hidden behind the processing of independent tasks.

Another damaging factor to the pipeline performance, even more than the instruction dependency is branching. The encounter of a conditional branch not only delays further executions but affects the performance of the entire pipe since the exact sequence of instructions to be followed is hard to foretell until the deciding results becomes available at the output. To alleviate the effects of branching, several techniques have been employed to provide mechanisms through which processing can resume safely even if an incorrect branch occurs which may create a discontinuous supply of instructions.

A similar degrading effect to the conditional branching is caused by interrupts which disrupt the continuity of the instruction stream through the pipeline. Interrupts must be serviced before any action can be applied to the next instruction. In the case that the cost of a recovery mechanism for processing to proceed after an unpredictable interrupt occurs (while instruction *i* is the next one to enter the pipe), is not exceedingly substantial, sufficient information is saved for the eventual recovery. Otherwise these two instructions, the interrupt instruction and instruction *i*, have to be executed sequentially which is in fact, not aimed at by the pipelining principle.

Finally, one of the most beneficial applications of overlapped processing in order to increase the total throughput has been the execution of arithmetic functions. Specially, the advantages of pipelining are greatly enhanced when floating point operations are being considered since they represent quite a lengthy process. Again, until all modules in the pipe are excessively used, full speed is not obtained. For example, the TI ASC arithmetic pipelined processor is made up of eight modules, as shown in Figure 1.11.

1.5 DATA-FLOW COMPUTERS

A common feature for all the high-speed parallel computer architectures is that, due to the basic linearity of the program, the use of implicit sequencing of the instructions is possible. This is a von-Neumann characteristic which means that the order of execution of the instructions is determined by the order in which they are stored in the memory with branches used to break this implicit sequencing at selective points. An alternative form of instruction controlling is the explicit sequencing which is basically the principal concept exploited by the data-flow machines to provide the maximum possibilities for concurrency and speed-up. However, this concept has a significant impact on the architecture of such machines, the program representation, and the synchronisation overheads.

In a data-flow architecture the algorithm is represented by a graph where the nodes correspond to the computations and the arcs describe the flow of data or operands, from the node producing the data (as a result) to the node using it as an operand [Dennis 1980]. In addition to the nodes describing the basic operations, there are nodes which are used to control the routing of data. Thus, the execution of any instruction is determined by the availability of all its operands resulting in a more complex control due to the high overheads involved in routing the data. With the use of the above graph representation, the data-flow concept encounters some problems when the algorithm contains loops or subroutine calls, in which case the same instruction is executed several times. Basically, the implementation of the data-flow computers can be grouped into two main classes, the static and dynamic structures, depending on how

this problem is tackled. In the first class, the static one, the loops and subroutine calls are unfolded at compile time so that each instruction is executed only once. Consequently, the implementation of the sequencing control is made simple since it directly follows that of the graph. On the other hand, in the dynamic case, the operands are labelled so that a single copy of the same instruction can be used several times for different instances of the loop (or subroutine). For this type of architecture, it is necessary to match all the operands with the same label before issuing the single copy of the instruction, the implementation of the control is significantly more complex in comparison with that of the previous class. However, the dynamic approach which allows a compact representation of large programs, can effectively exploit the concurrency that appears during execution (for example, recursive calls or data-dependent loops).

An example of the static approach is the MIT Data-Flow machine (Figure 1.12) which consists of the following main components: a store that contains the instruction cells or packets having space for the operation, operands and for pointers to the successors, and a set of operating units to perform the operations. These two components are connected by the two interconnection networks, one to send ready-to-execute instruction packets to the operating units and another to send results back from the operating units to the instructions that use them as operands. The system has to be carefully designed so as to prevent any bottleneck from occurring and to provide means for the full exploitation of all the concurrency.

In such a system, the maximum throughput is determined by the speed and number of the operating units, the memory bandwidth and by

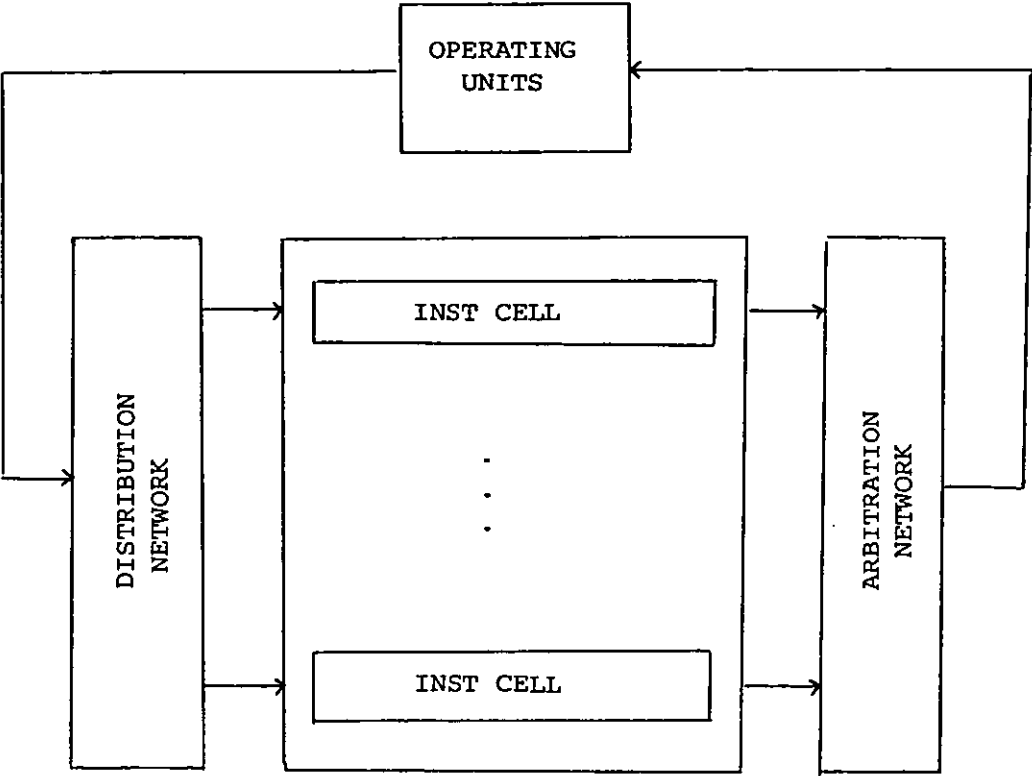


FIGURE 1.12: The Static Data-Flow Machine

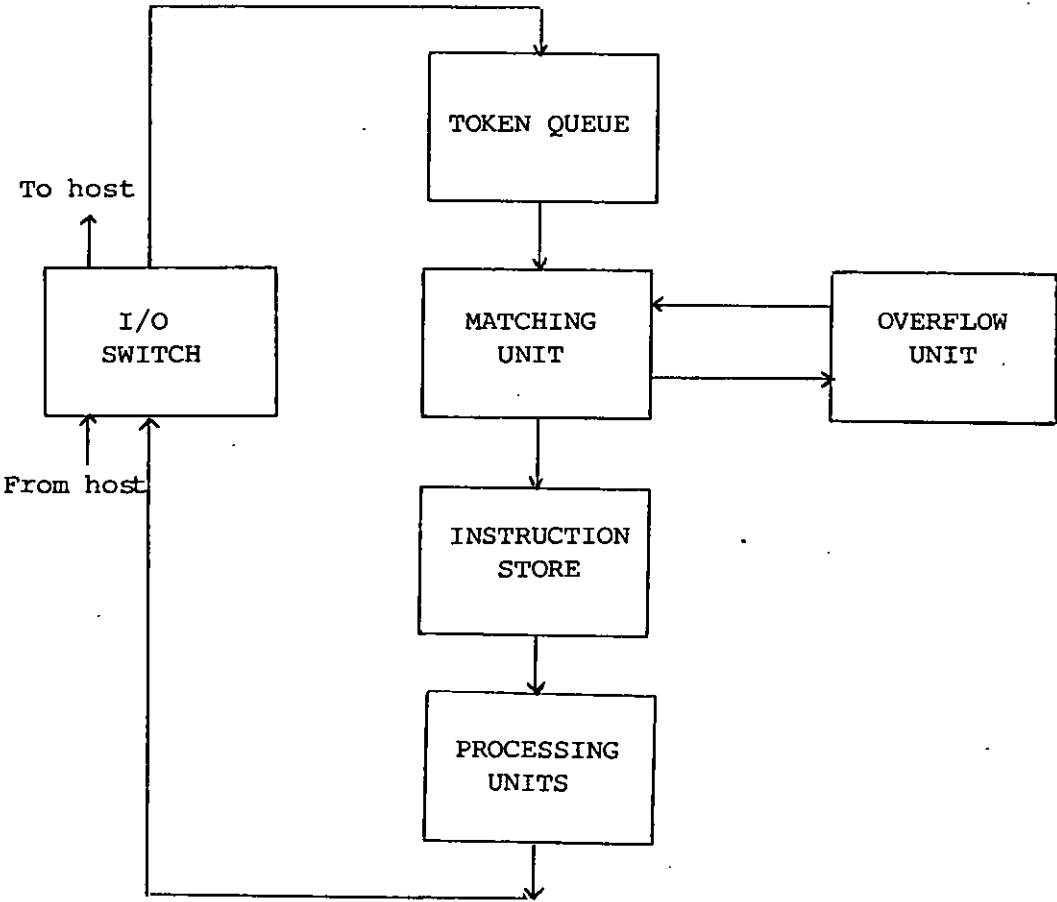


FIGURE 1.13: The Dynamic Data-Flow Machine

the interconnection system. As in the other organisations, several degradation factors reduce the effective throughput. The most significant are the degree of concurrency available in the program, the memory access and the interconnection network conflicts, and the broadcasting of results, all of which except the last one are similar to the other systems. Sometimes an instruction has several successors, so that the result has to be sent, or broadcast, to all of them and this introduces significant overheads in the case when the number of destination pointers present in an instruction cell is limited.

Examples of the dynamic approach include the U-Interpreter Machine [Arvind 1982] and the Manchester Dataflow Machine [Gurd 1985]. The main components of the latter (see Figure 1.13) are the token queue that stores computed results, the token matching unit that combines the corresponding tokens into instruction arguments, the instruction store that holds the ready-to-execute instructions, the operating units, and the I/O switch for communication with the host. The degradation factors are similar to those of the static case except the additional overhead in token label matching. Due to the above mentioned degradation factors, data flow machines are only attractive for cases in which the concurrency exhibited is of several hundred instructions.

Another problem in the use of the dataflow approach is the lack of any data structure definition, in fact only scalar operations were first utilised in the attempt to maximise the amount of concurrency and this had significant limitations in terms of the modularity of the programs. The inclusion of data structures in the graph representation requires that the dataflow concept be extended and operations on them

be defined [Davis 1982]. From the operational point of view, the most straightforward solution is to treat the data structure as an atomic operand, requiring the structure to be sent as a whole to the operating units even though only few elements are operated on. This can be performed by sending to the operating unit a pointer to the data structure instead of its value. However the disadvantage with this is that the whole data structure has to be copied when any of its elements is modified resulting in a heavy transfer rate between the memory and the operating units. To avoid this copying overhead, Dennis [Dennis 1974] has proposed a tree structure to store arrays and operations such as select and append to modify parts of the array. However, Dennis' proposal does not solve the limitation that the elements of the array have to be modified in a sequential manner, which increases the overhead for the select and append operations. To reduce this overhead Gandiot and Evcegovac [Gandiot 1982] proposed the introduction of macro-actors to perform more complex updating. To eliminate the sequential nature of the modifications, Arvind and Thomas [Arvind 1980] introduced I-structures that allow concurrent writes and reads by adding to each element a tag indicating if the element has already been written and a list of pending reads to the reads queue to arrive before the element has been written.

One of the most significant advantages of the data-flow machines, as claimed by its proponents, is the exploitation of the concurrency at a low level of the execution hierarchy since it allows the maximum utilisation of all the available concurrency. However, some researchers argued that the overhead with this unstructured low-level concurrency

is too high and have proposed the use of a hierarchical approach in which different types of concurrency can be exploited at different levels.

Finally, the dataflow organisation which is still in an experimental stage, has recently received considerable researchers' attention. Several prototype systems have been built or simulated and are being evaluated.

1.6 ARRAY PROCESSORS

The early interest in the parallel processor area initially appeared in the investigation of machines that were arrays of processors connected in a four-nearest-neighbour manner "N,E,S,W" such as the Von Neumann's Cellular Automate [Von Neumann 1968] and the Holland machine [Holland 1959]. Eventually, as a result of the growing interest in this form of a computer, parallel processors with a central control mechanism that controlled the entire array and operating in a SIMD manner began to emerge.

All the systems in the array processor class can be identified by their major components, structured in a number of various and different ways:

A number of identical Processor Elements (PE's) synchronously operating on different data streams proliferating from a number of memory banks not necessarily equal to the number of the PE's through a communication network with some form of local control and finally some form of global control. A simple array computer is shown in Figure 1.14.

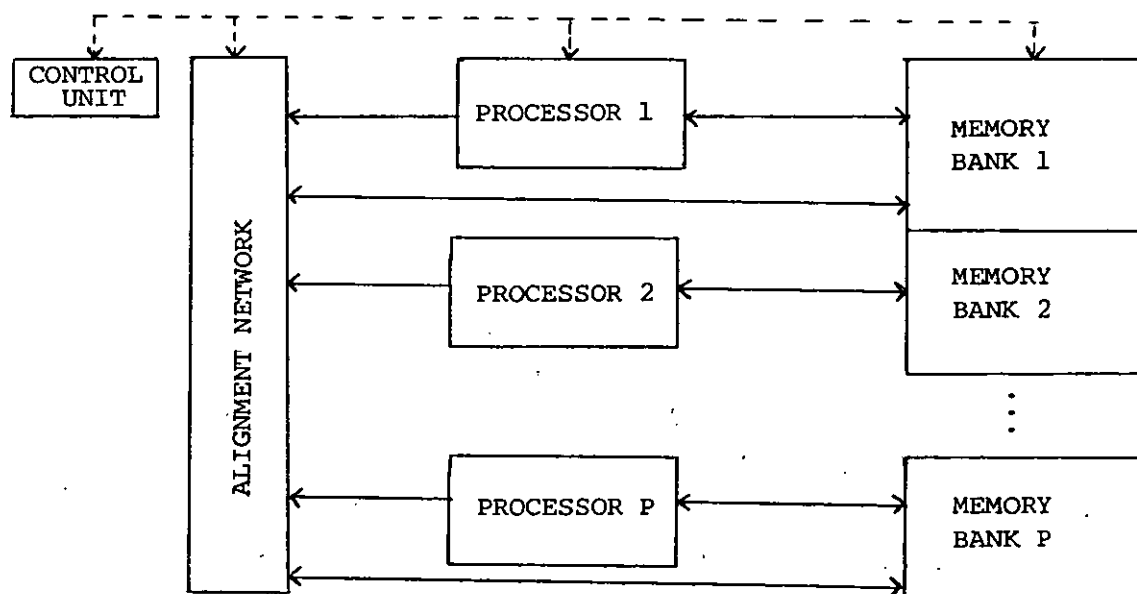


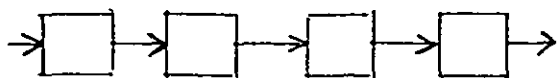
FIGURE 1.14: A General SIMD Architecture

The control unit which is usually a computer itself with its own arithmetic and logic unit, memory and registers, differs from the other processors in that it can execute scalar and control instructions (including conditional branch instructions). The processor elements which lack this ability since they must all be kept in synchronisation, do not generate their own instructions, but they all receive the same sequence of vector instruction from the control unit. A local on-off control unit is used to permit processors to either execute or ignore certain broadcast vector instructions.

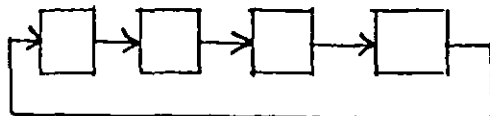
One of the most currently active research areas in computer architecture is the interconnection networks since they represent the accumulation of a large number of design decisions made before the implementation of the actual architecture.

The interconnection networks can be generally distinguished into two types, the bus and the alignment networks with basic differences between them: while the former allows only a single one-to-one communication to take place at any given time, the latter allows several one-to-one (parallel data and control transfer) or one-to-many (allowing one unit to broadcast to many units in parallel) communication. It follows that the bus network is less expensive but a slower network than the other.

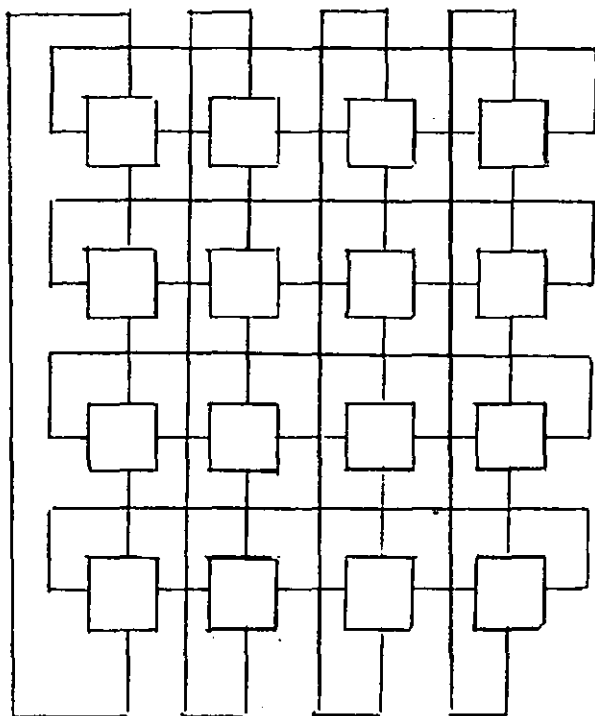
Furthermore, the alignment networks can be topographically sub-categorised into static and dynamic networks. A static network is characterised by the required dimensions for layout. Examples range from one-dimensional structures to hypercube networks. In Figure 1.15, we can see examples of one, two and three-dimensional networks. On



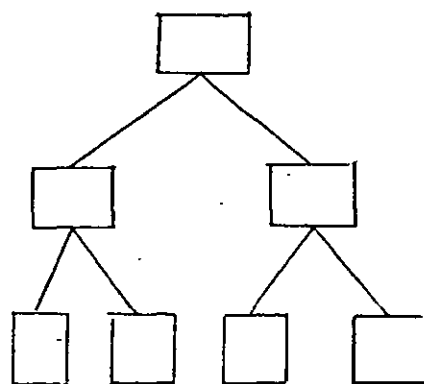
(a) Linear array network



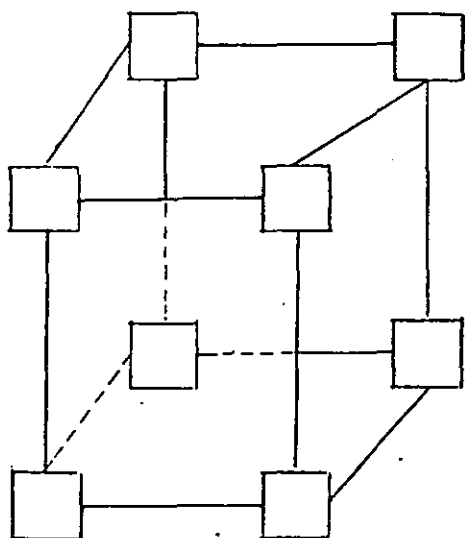
(b) Ring network



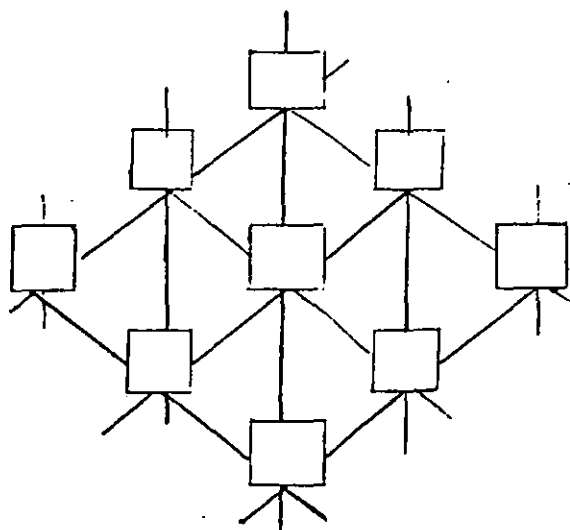
(c) Four-neighbour network



(d) Tree



(e) 3D-cube



(f) Systolic array

FIGURE 1.15: Example of 1,2 and 3-Dimensional Interconnection System

the other hand, the dynamic networks are distinguished into the single-stage, multiple-stage and crossbar types of networks. The single-stage network consists of a single stage of switches. The nearest neighbour network and the perfect shuffle networks are examples of this type of network (see Figure 1.16). A more generalised connection network, where every input is connected to every output channel through a crosspoint is the crossbar switch. Figure 1.17 shows two representations of the crossbar switch from four inputs to four outputs. Finally, the multi-stage networks which can provide a cheaper alternative to the complete connection as offered by the crossbar switches are based upon a number of interconnected 2×2 crossbar networks organised into several stages. In Figure 1.18 we can see two multi-stage networks, the binary Bene's and the indirect binary n -cube networks. An example of the parallel or array processors is ILLIAC IV [Barnes 1968].

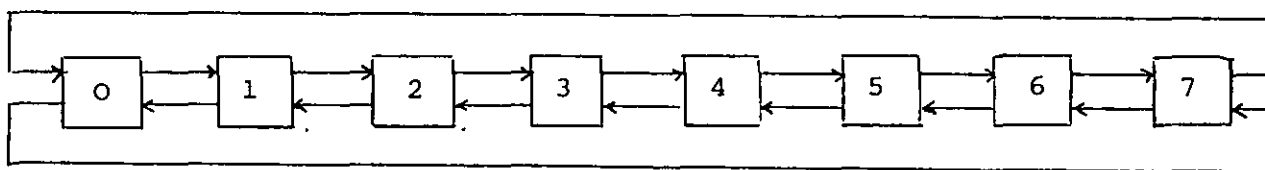


FIGURE 1.16(a): The Nearest-Neighbour Network

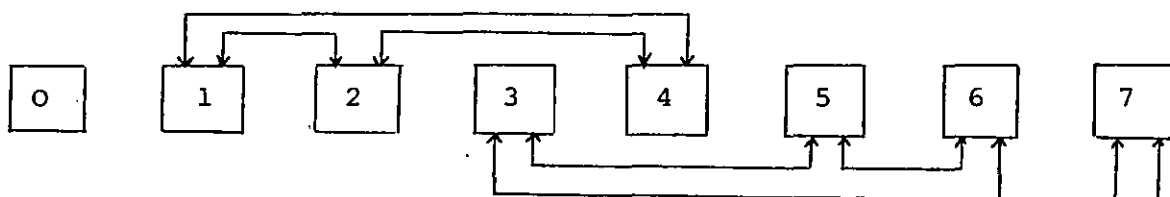


FIGURE 1.16(b): Perfect-Shuffle Network

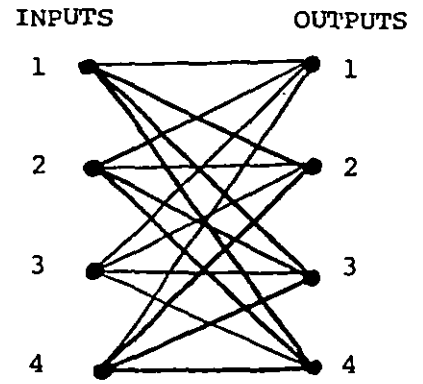
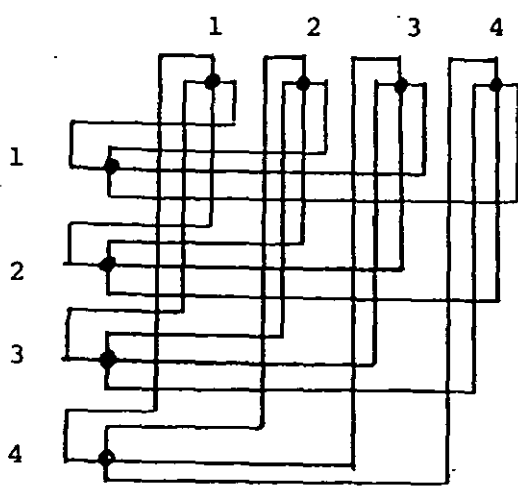


FIGURE 1.17: Two Representations of the Crossbar Switch from Four Inputs to Four Outputs

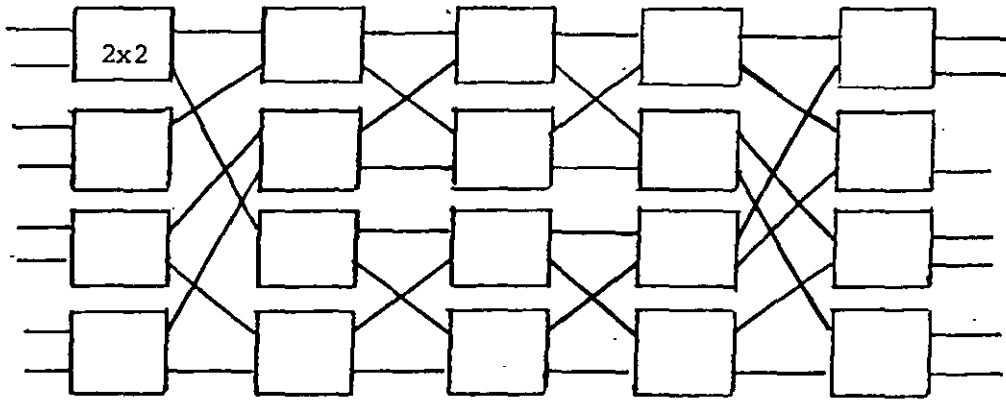


FIGURE 1.18(a): The Binary Benes Network Using 2x2 Crossbar Switches

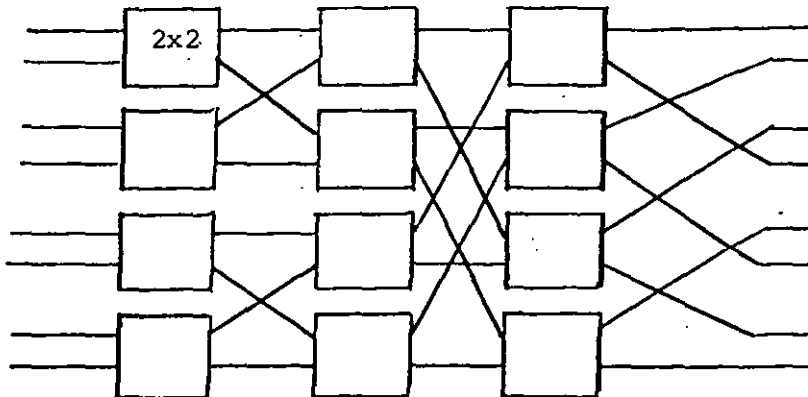


FIGURE 1.18(b): The Indirect Binary n-Cube Network

CHAPTER 2

PARALLEL ARCHITECTURES - A VLSI APPROACH

2.1 INTRODUCTION TO THE VLSI TECHNOLOGY PARADIGM

There has been a rapid growth of computing technology that has followed the invention of transistors in the late 1940's. (The first transistor was invented in 1948 at the Bell Telephone Laboratories) and integrated circuits in the late 1960's. Through developments in transistors, new families of small computers (i.e. minicomputers) began to emerge on the market. As a result, thousands of transistor elements were assembled on minute chips of silicon. The race for smaller and faster computing machines has developed ever since. A mainframe computer built using the original thermionic valves had weighed more than thirty tons and required a room of 60×25 square feet to hold it; a computer of superior capability could, by 1971, be accommodated on a sliver of silicon.

The migration of IC to large scale integration (LSI) technology allowed tens of thousands of electronic components to fit on to a single chip. Following the rapid advances in LSI technology, the Very Large Scale Integration (VLSI) circuits have been developed with which enormously complex digital electronic systems can be fabricated on a single chip of silicon, one-tenth the size of a postage stamp. In fact, it is foreseen that the number of components that a VLSI chip could accommodate would be increased by a multiplier factor of ten to one hundred in the next two decades [Mead 1980]. Devices which once required many complex components can now be built with just a few VLSI chips, reducing the difficulties in reliability, performance and heat dissipation that arise from standard SSI and MSI components [Kung 1979].

As computer applications still require faster and more powerful

computer architectures than these which are currently available and as we are migrating from the information processing era towards "knowledge" based systems which characterise the projected fifth generation of computers, the research in computer technology has been widened more than ever before. H.T. Kung was the first to realise that the rapidly developing chip industry together with automata theory could be the key success to constructing fast, highly parallel computer structures at low cost. Until the advent of VLSI, the development of parallel computers with a large number of processors had been limited by the unaffordable high costs of manufacture. Existing machines had been improved by tinkering with the traditional Von Neumann architecture, for instance cycle stealing, direct memory access (DMA) and pipelining of fetch and execute operations. As such, parallel machines were confined only to research purposes or military operations.

The development of new manufacturing techniques for fabrication of small, dense and inexpensive semi-conductor chips created a unique circumstance in the computer industry. With the use of VLSI in circuits, size and cost of processing elements and memory was considerably reduced and it became feasible to combine the principles of automation theory with the pipeline concepts. The combination was especially attractive since device manufacture costs remained constant relative to circuit complexity, with most time and money invested in design and testing.

In relation with what was said above, approaches to device designs have progressed so significantly to the point that hardware design now relies heavily on software techniques, i.e. special rules

for circuit layout and high level design languages (e.g. geometry languages, stick languages, register transfer languages, etc.) [Mead 1981]. In fact, some of these languages offer the powerful chip fabrication capability directly from a design they express.

Illustrative of this trend is the term silicon compiler utilised by the hardware designers to refer to computer-aided design systems currently under development. Analogous to a conventional software compiler, the silicon compiler will convert linguistic representations of hardware components into machine code, which can be stored and subsequently utilised in computer-assisted fabrication.

However, VLSI presents some problems, as the size of wires and transistors approach the limits of photolithographic resolution for it becomes literally impossible to achieve further miniaturisation and actual circuit area becomes a key issue. In addition, the chip area is also limited in order to maintain high chip yield and the number of pins (through which the chip communicates with the outside world) is limited by the finite size of the chip perimeter. These restrictions form the basis of the VLSI paradigm.

For a newly developed technology or product to survive in a highly competitive industry there must be sufficient demand for it. The emergence and subsequent success of VLSI oriented computing systems is not due only to H.T. Kung's foresight but also to the timeliness. At the same time Kung revealed the systolic concept, the idea of using VLSI for signal processing was the major focus of attention in governmental, industrial and university research establishments.

2.2 FUNDAMENTAL ARCHITECTURAL CONCEPTS IN DESIGNING SPECIAL PURPOSE VLSI COMPUTING STRUCTURES

High-performance special-purpose VLSI oriented computer systems are typically used to meet specific applications, or to off-load computations that are especially taxing to general-purpose computers. However since most of these systems are built on an adhoc basis for specific tasks, methodological work in this area is rare. In an attempt to assist in correcting this adhoc approach, some general design concepts will be discussed, while in the following paragraph the particular concept of systolic and wavefront array architectures, two general methodologies for mapping high-level computation problems into hardware cellular structures, will be introduced.

The problem of embedding a network of processors and memories into a set of VLSI chips is similar to that of embedding graphs whose nodes are computers, or gates, onto grids so as to minimise area. Most of the researchers exploring this problem usually make certain assumptions; for example, they assume that wires run and devices are oriented in only horizontal and vertical directions, everything is embedded on a square grid, all device nodes are at the same layer.

The computational power of a chip is often measured by the number of transistors it contains. However, this is quite a misleading approach for the organisation of a chip's circuitry has a very strong effect. In general, regular chip designs make more efficient utilisation of silicon area, which is a more natural measurement factor for the circuit size than the number of transistors. Such designs utilise less area for the wiring amongst transistors, leaving

more space for transistors themselves.

From the memory capacity point of view, the number of bits has been quadrupling every few years; in the mid-1970's technology passed through the era of 1K, 4K and 16K bits memory chips. In 1981 the memory size was expanded to 32K bits and a 64K bit is predicted.

Particularly for the design of special-purpose VLSI oriented computer machines, cost effectiveness has always been a major concern; their fabrication must be low enough to justify their specialised, and consequently, limited applicability. Cost can be distinguished in non-recurring design and recurring parts costs. Any fall of the latter's cost is equally applied for the merit of both special-purpose and general-purpose computer systems. Furthermore this cost is even less significant than the design cost, since the production of special-purpose computer systems in large quantities is quite a rare phenomenon. Hence, the design of such a system should be relatively small for it to become more attractive compared to a general-purpose computer and this can be achieved by the utilisation of appropriate architectures. More specifically, if the decomposition of a structure into a few types of simple substructures which are repetitively utilised with simple and regular interfaces is feasible, then significant savings are most likely to be achieved.

In addition, special-purpose computer systems based on simple and regular designs are likely to be modular and consequently adjustable to various performance goals, i.e. system costs may be made analogous to the performance required. This fact reveals that achieving the architectural challenge for simple and regular design, yields cost-

effective special-purpose computer systems.

Since such VLSI computing structures can function as peripheral devices, attached to conventional host computer, receiving data and control signals and outputting results, at a computation rate, which will balance the available I/O bandwidth with the host, is the ultimate performance goal of a special-purpose computer system. Therefore the likely modular attribute of such a concept is highly necessary, since it allows the flexibility of the structure to match a variety of I/O bandwidths; and since an accurate a priori estimate of available I/O bandwidths in complex systems is often possible.

However this problem becomes especially severe when a very large computation is performed on a relatively small special-purpose computer system. In this case the computation must be decomposed.

In fact one of the major challenging research items becomes the development of algorithms that could be mapped into and executed efficiently by a special-purpose computer system. This implies that algorithms should decompose into modules, that map compactly into one VLSI chip (or a module of chips), and modules should be interconnected in an efficient manner. These algorithms must support high degrees of concurrency and employ a simple, regular data and control flow to enable an efficient implementation [Dew, 1984].

To conclude we mention that special-purpose VLSI oriented computing structures can be either a single chip, built from a replication of simple cells, or a system built from identical chips, or even a combination of these two approaches. Figure 2.1 summarises the principle stages and tasks interdependencies involved in the design of a VLSI chip (see Foster and Kungs' paper, [Foster 1980]).

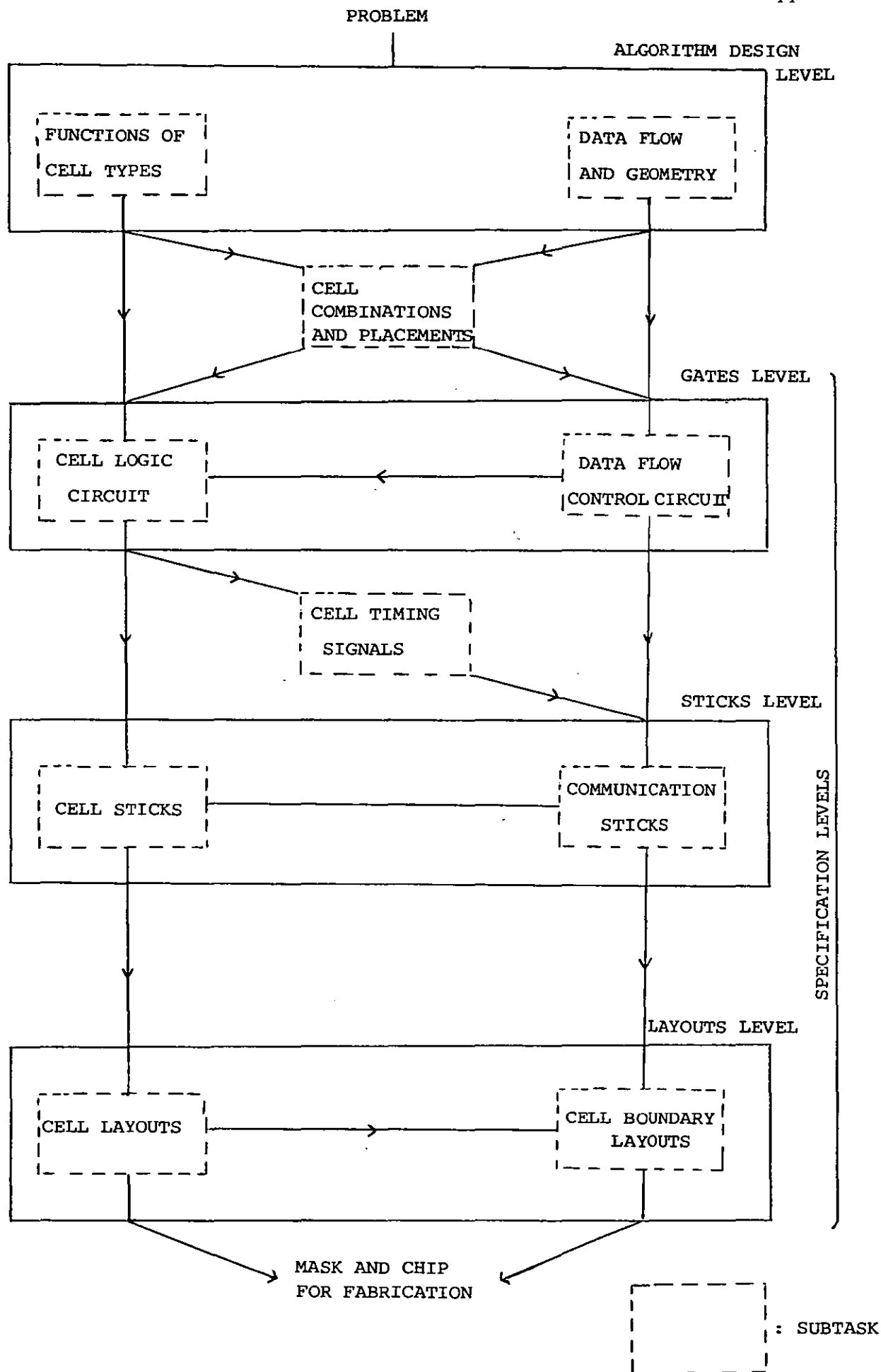


FIGURE 2.1: The Design Stages of a Special-Purpose VLSI Chip

In fact in the environment of VLSI systems design, the boundary between software and hardware has become increasingly vague.

2.2.1 Systolic Arrays

The concept of systolic architectures, pioneered by H.T. Kung, which has been successfully shown to be suitable for VLSI implementation is basically a general methodology of directly mapping algorithms onto an array of processor elements. It is especially amenable to a special class of algorithms, taking advantage of their regular, localised data flow.

The word 'systole' was borrowed from physiologists who used it to describe the rhythmically recurrent contraction of the heart and arteries which pulse blood through the body. By analogy, the function of a cell in a systolic computing system is to ensure that data and control are pumped in and out at a regular pulse, while performing some short computation [Kung 1978], [Dew, 1986].

Thus, a systolic array is a network of processing elements, usually arranged in a regular pattern and locally linked by communication channels. Operands are pumped through the array at a regular pulse. Everything is planned in advance so that all inputs to a cell arrive at just the right time before they are consumed. Intermediate results are passed on immediately to become the inputs for further cells. A steady stream flows at one end of the array which is said to consume data and produce results 'on the fly'. For instance, by locally connecting a few basic cells, known as Inner Product Steps 'IPS' - each performing the operation $C=C+A*B$ - leads to a fundamental network capable of performing computation - intensive

algorithms, such as digital filtering, matrix multiplication, and other related problems (see Table 2.1 for a more comprehensive list of potential systolic applications).

The systolic array systems feature the important properties of modularity regularity local interconnection, a high degree of pipelining and highly synchronised multiprocessing. Such features are particularly more interesting in the implementation of compute-bound algorithms, rather than input/output - 'I/O' - bound computations. In a compute-bound algorithm, the number of computing operations is larger than the total number of I/O elements, otherwise the problem is termed I/O-bound. Illustrative of these concepts are the following matrix-matrix multiplication and addition examples. An ordinary algorithm, for the former, represents a compute-bound task, since every entry in the matrix is multiplied by all the entries in some row or column of the other matrix - i.e. $O(n^3)$ multiply-add steps, but only $O(n^2)$ I/O elements. The addition of two matrices, on the other hand, is an I/O bound task. Since the total number of adds is not larger than the total number of I/O operations, i.e. $O(n^2)$ add steps and $O(n^2)$ I/O elements.

It is apparent that any attempt to speed-up an I/O-bound computation must rely on an increase in memory bandwidth (the so-called 'Von Neumann' bottlenecks). Memory bandwidths can be increased by the utilisation of either fast components, which may be quite expensive, or interleaved memories, which may create complex memory management problems. However, the speed-up of a compute-bound computation may often be achieved in a relatively simple and inexpensive manner, that is by the systolic architectural approach.

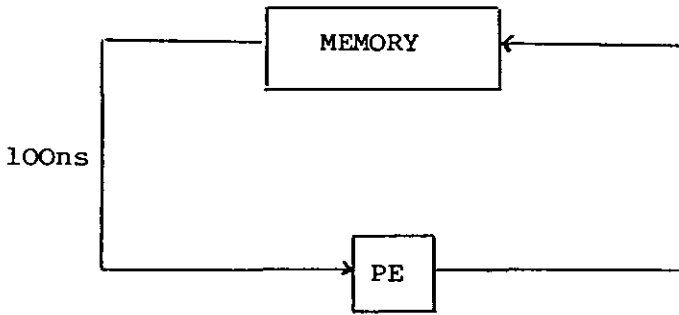
'SYSTOLIC' PROCESSOR ARRAY STRUCTURE	PROBLEM CASES
1-D linear arrays	: FIR-filter, convolution, 'Discrete Fourier Transform' - DFT, matrix-vector multiplication, recurrence evaluation, solution of triangular linear systems, carry pipelining, cartesian product, odd-even transposition sort, real-time priority queue, pipeline arithmetic units.
2-D square arrays	: Dynamic programming for optimal parenthesization, image processing, pattern matching, numerical relaxation, graph algorithms involving adjacency matrices.
2-D hexagonal arrays	: Matrix problems (matrix multiplication), LU decomposition by Gaussian elimination without pivoting, QR-factorization, transitive closure, relational database operations, DFT.
Trees	: Searching algorithms (queries on nearest neighbour, rank, etc., systolic search (tree), recurrence evaluation.
Triangular arrays	: Inversion of triangular matrix, formal language recognition.

TABLE 2.1: The Potential Utilization of 'Systolic' Array Configurations

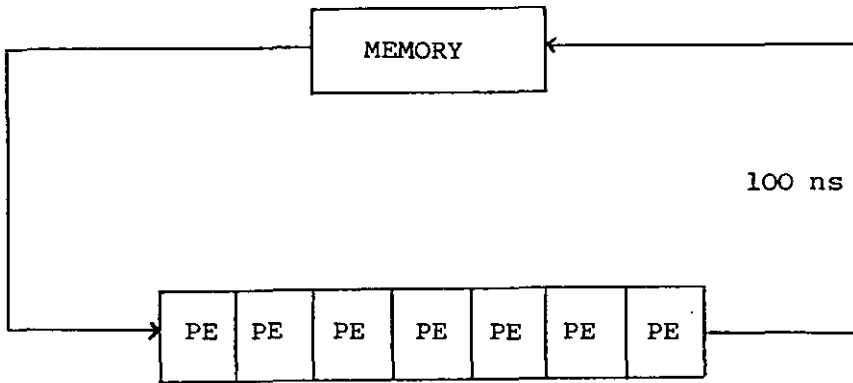
The fundamental principle of a systolic architecture, a systolic array in particular is illustrated in Figure 2.2. By replacing a single processing element with an array of PEs, a higher computation throughput can be achieved without increasing memory bandwidth. This is apparent if we assume that the clock period of each PE is 100ns; then the conventional memory-processor organisation (a) has at most 5 MOPS performance, while with the same clock rate, the systolic array (b) will result in a possible 35 MOPS performance.

Finally, this approach of utilising each input data item a number of times, thus achieving a high computation throughput with only a modest memory bandwidth, is just one of the advantages of the systolic concept. Other equally significant criteria and advantages include modular expansibility, utilisation of simple, uniform cells, extensive concurrency and fast response time.

However, one problem associated with systolic arrays is that the data and control movements are controlled by global timing-reference beats. In order to synchronise the cells, extra delays are often used to ensure correct timing. More critically, the burden of having to synchronise the entire network will eventually become intolerable for very large or ultra large scale arrays [Dew, 1984].



(a) The Conventional Organisation



(b) A Systolic Processor Array

FIGURE 2.2: Systolic Design Principle

2.2.2 Wavefront Arrays

A solution to the above mentioned problems, as suggested by S.Y. Kung [Kung 1985], is to take advantage of the data and control flow locality, inherently possessed by most algorithms. This permits a data-driven, self-timed approach to array processing. Conceptually such an approach substitutes the requirement of correct 'timing' by correct 'sequencing', this concept is used extensively in data flow computers and wavefront arrays.

Basically the derivation of a wavefront process consists of the three following steps:

- a) the algorithms are expressed in terms of a sequence of recursions;
- b) each of the above recursions is mapped to a corresponding computation wavefront; and
- c) the wavefronts are successively pipelined through the processor array.

Based on this approach, S.Y. Kung introduced the Wavefront Array Processor (WAP) which consists of an $N \times N$ processing element with a regular connection structure, a program store and memory buffering modules as illustrated in Figure 2.3. The processor grid acts as a wave propagating medium using handshaking protocols.

Each processor performs a limited number of computations and is controlled by a program loaded in the program store. Data is stored in memory modules around the boundary and extra time must be allowed to set up a computation. An algorithm is executed by a series of wavefronts moving across the grid with processors computing whenever its data and instructions are available. Processors are assumed to support pipelining of waves and the spacing of waves (T) is determined by the availability of data and the execution of the basic operation. The speed of wavefront A is equivalent to the data transfer time.

Summarising, the wavefront approach combines the advantages of data flow machines with both the localities of data flow and control flow inherent in a certain class of algorithms. Since the burden of synchronising the entire array is avoided, a wavefront array is architecturally 'scalable'.

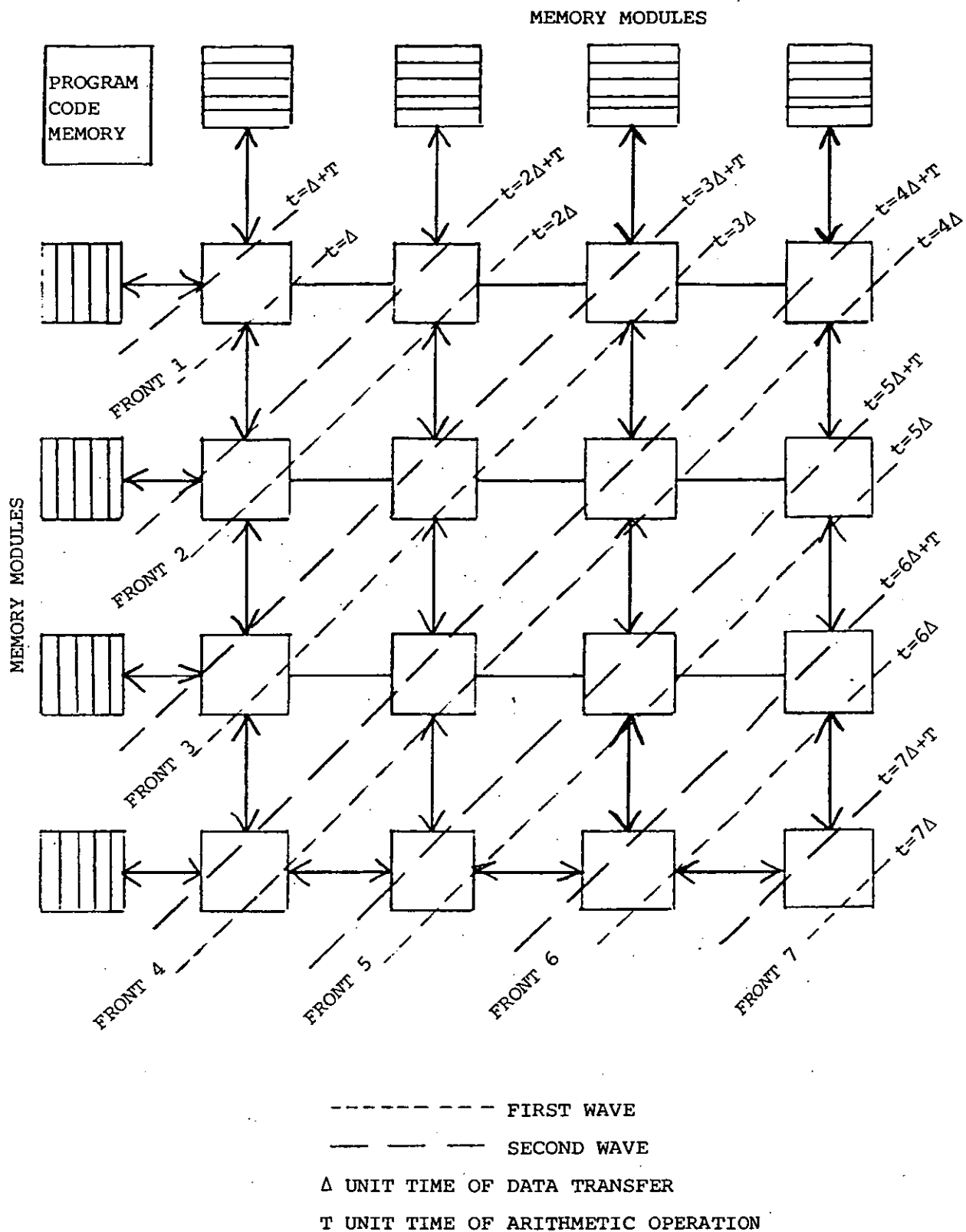


FIGURE 2.3: The Wavefront Array Processor

2.3 VLSI-ORIENTED ARCHITECTURES

For large applications it may not be feasible to design a single chip implementation of an array, especially when balance between flexibility, efficiency, performance and implementation cost is essential. An alternative approach is to implement basic cells at the board level using a set of 'off-the-shelf' components which are widely available as chip packages from various manufacturers.

Systolic arrays achieve high performance and efficiency by considering only restricted problem classes, at the expense of flexibility and implementation cost. For a more economical solution, arrays must be constructed with many incorporated features so as to handle a large number of systolic algorithms. In this section, we shall briefly review the main contenders of VLSI-Oriented computing systems which have received attention to date.

2.3.1 The WARP Architecture

The WARP architecture, one of the most advanced VLSI-oriented systems, was developed at Carnegie Mellon University (CMU) by H.T. Kung and his associates for purely systolic algorithms. Initially, the design began with a preliminary study of different architectures based on general purpose microprocessors which could implement a variety of systolic algorithms efficiently. The study resulted in the Programmable Systolic Chip (PSC) discussed in [Fisher 1984] and prompted research into cell structures for high performance systolic arrays in a particular area (signal processing).

The WARP architecture is a 1-D linear systolic array with data and control flowing in one direction (with input at one end of the

array and output at the other). From the preceeding discussion we observe that the design allows easy implementation, synchronisation by a simple global clock mechanism, minimum input/output requirements and the use of efficient fault tolerance techniques for malfunction.

The basic WARP cell is constructed from a collection of chips as is illustrated in Figure 2.4, its main characteristics being the pipelining of data and control. Weitek 32-bit floating point multiplier (MPY) and ALU's perform operations and can be used in a pipeline mode to improve throughput by two level pipelining. The MPY and ALU register files use Weitek register file chips and can compute approximate functions like inverse square roots using look-up facilities. The X,Y and address-files are also register files but this time they are used to implement delays for synchronising data paths, and can be used as extra registers for book-keeping operations, while the data memory is used to reduce the input/output bandwidth by implementing tables of data and storing intermediate results. It can also be used to implement multiple cells on the same processor and hence 2-D arrays. The crossbar and input multiplexors (muxes) provide communication between the individual elements and can be reconfigured by control signals. The muxes permit two-directional dataflow and ring set-ups. A ten-cell prototype has been built at CMU and tested on a number of example arrays discussed in H.T. Kung [Kung 1984].

2.3.2 The CHIP Architecture

In order to derive a more flexible VLSI-oriented computing system than the special-purpose computers, where the same hardware would be

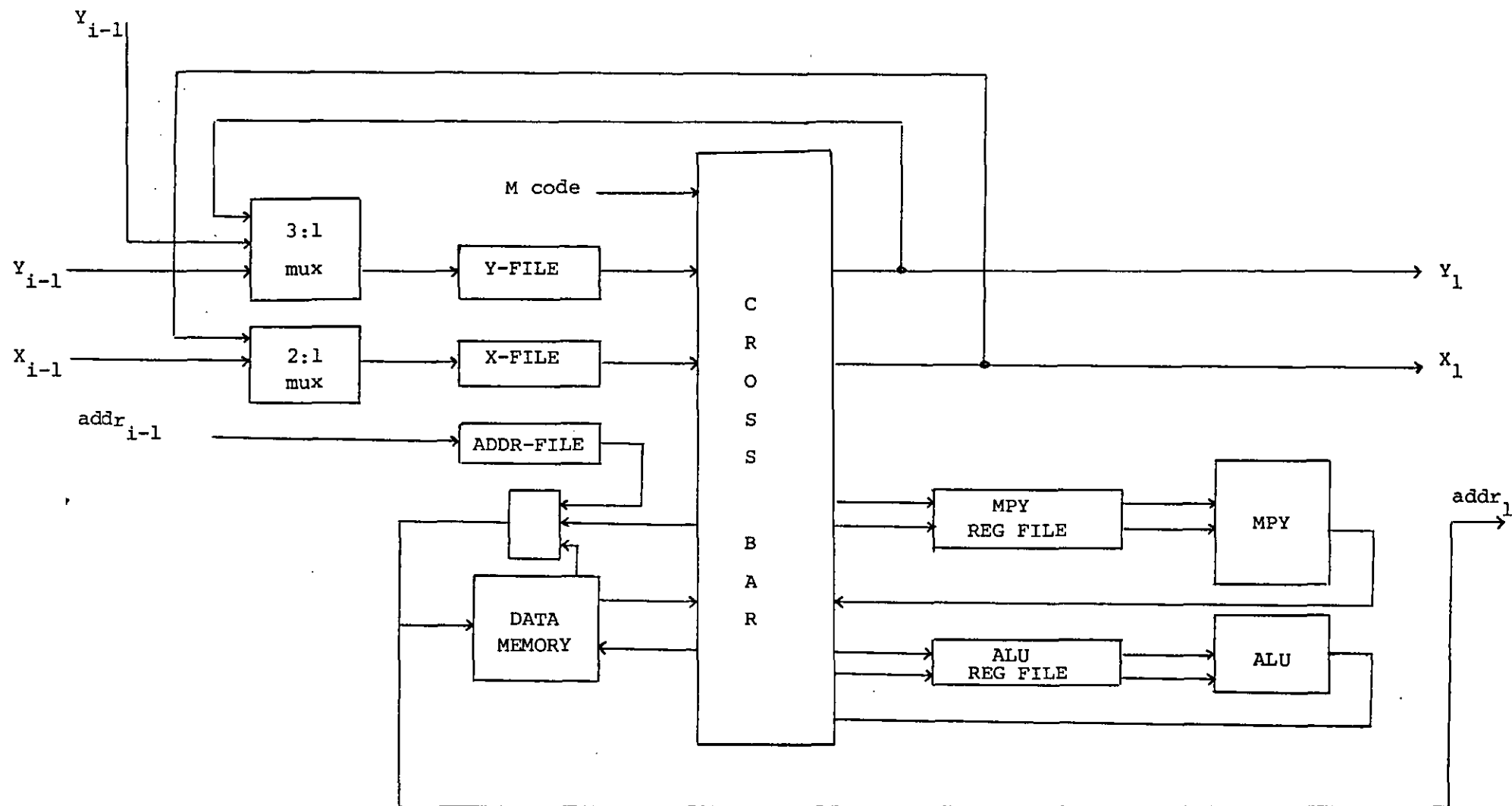


FIGURE 2.4: Data Paths for the WARP Cell

used to solve several different problems, L. Snyder suggested the design of the configurable, highly parallel architecture 'CHIP' [Snyder 1982] based on the configurability principle. Conceptually, the chip represents a family of systems, each built out of three major components: a set of processing elements (PE's), a switch lattice and a controller. The lattice, the most important component of a chip, is a 2-D structure of programmable switches connected by data paths. The PEs are placed at regular intervals. Figure 2.5 shows two examples where squares represent PEs, circles represent switches and lines represent data paths. Note that the PEs are not directly connected to each other, but rather are connected to switches.

The processing elements are microprocessors each coupled with several kilo-bytes of RAM used as local storage. Data can be read or written through any of the eight data paths or ports connected to the PE. Generally, the data transfer unit is a word, though the physical data path may be narrower. The PE's operate synchronously and systolically.

Each programmable switch contains a small amount (around 16 words) of local RAM which is used to store instructions (one instruction per word) called configuration settings. Each configuration setting specifies pairs of data paths to be connected. When executed, each pair which is also known as a crossover level, establishes a direct, static connection across the switch that is independent of the others. The data paths are bidirectional and fully duplex, i.e. data movements can take place in either direction simultaneously. Now, executing a configuration settings program causes the specified connections to be

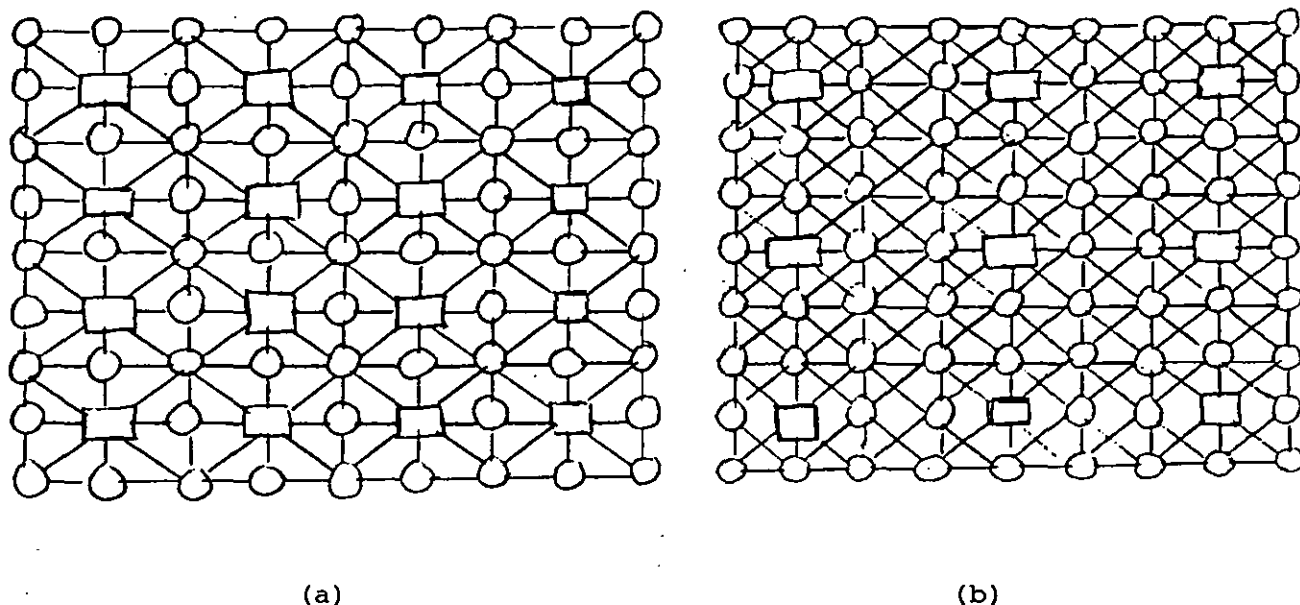
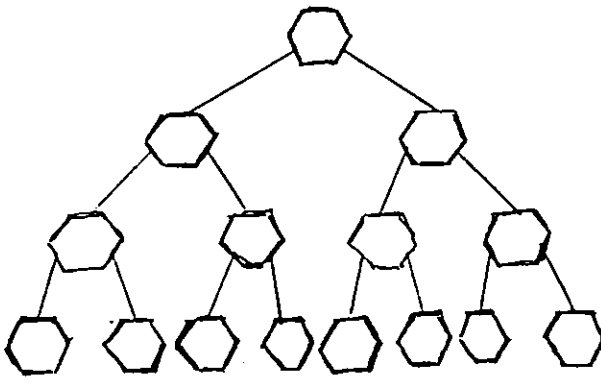


FIGURE 2.5: Two Lattice Structures

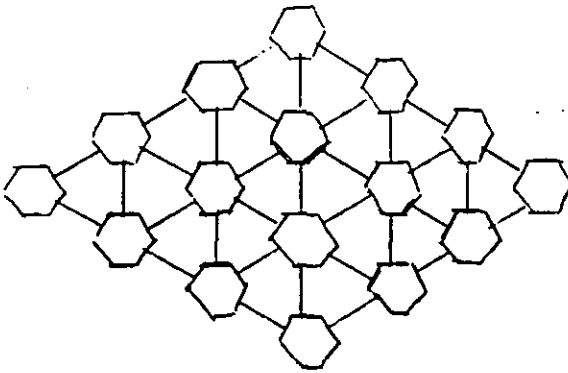
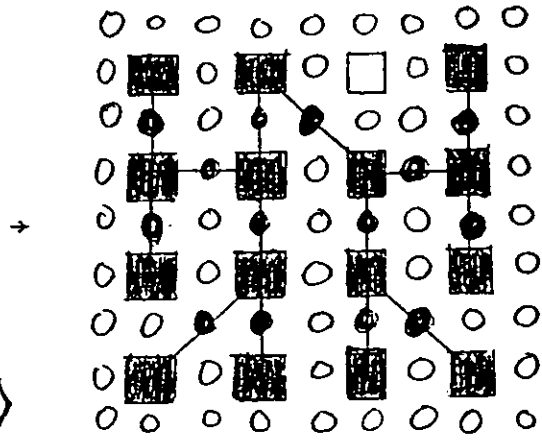
established and to persist over time, e.g. over the execution of an entire algorithm.

The processing elements can be connected together to form a particular structure by directly configuring the lattice. That is, the programmer sets each switch such that collectively they implement the desired processor interconnection graph. Figure 2.6 illustrates three examples of how the lattice of Figure 2.5(a) might be configured to implement some commonly used interconnection schemes.

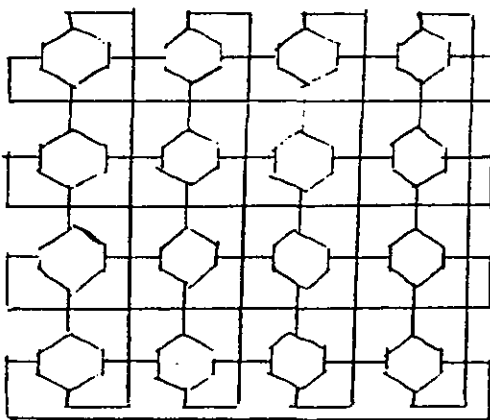
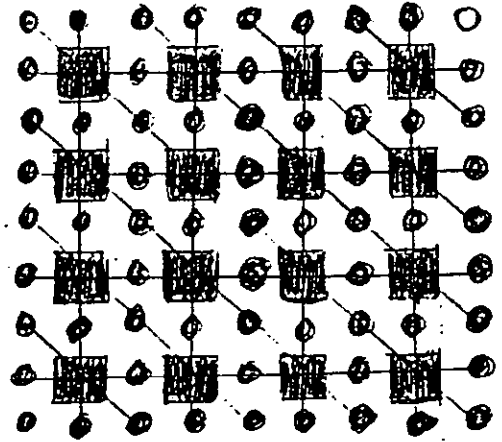
In addition to the lattice, a controller is also provided, and is responsible for loading programs and configuration settings into PE and switch memories respectively. This task is performed through an additional data path network, called 'skeleton'.



(a) Binary tree



(b) Systolic array



(c) Four-neighbour network

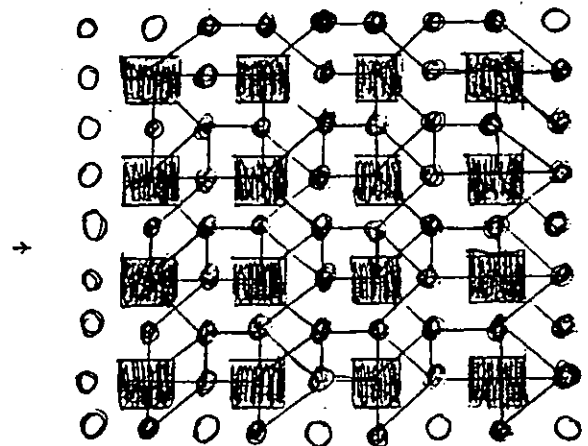


FIGURE 2.6: Embedding Graphs into the Lattice of Figure 2.5

From the functional point of view, CHIP processing starts with the controller broadcasting a command to all switches to invoke a particular configuration setting; for example to implement a mesh pattern. The established configuration remains during the execution of a particular phase of an algorithm. When a new phase of processing, requiring different configuration settings is to begin, the controller broadcasts a command to all switches so that they invoke the new configuration setting; for example, a structure implementing a tree. With the lattice thus restructured, the PE's resume processing, having taken only a single logical step in reconfiguring the structure.

In conclusion, the chip computer which is a highly parallel computing system, providing a programmable interconnection structure integrated with the processor elements, is well suited for VLSI implementation. Its main objective is to provide the flexibility needed in order to solve general problems while retaining the benefits of regularity and locality.

2.3.3 INMOS Transputers and OCCAM

A third possibility is the INMOS transputer, a single chip micro-processor containing a memory, processor and communication links for connection to other transputers, which provides direct hardware support for the parallel language OCCAM. The structure of a transputer is given in Figure 2.7.

The transputer and OCCAM were designed in conjunction and all transputers include special instructions and hardware which provide optimal implementations of the OCCAM model of concurrency and

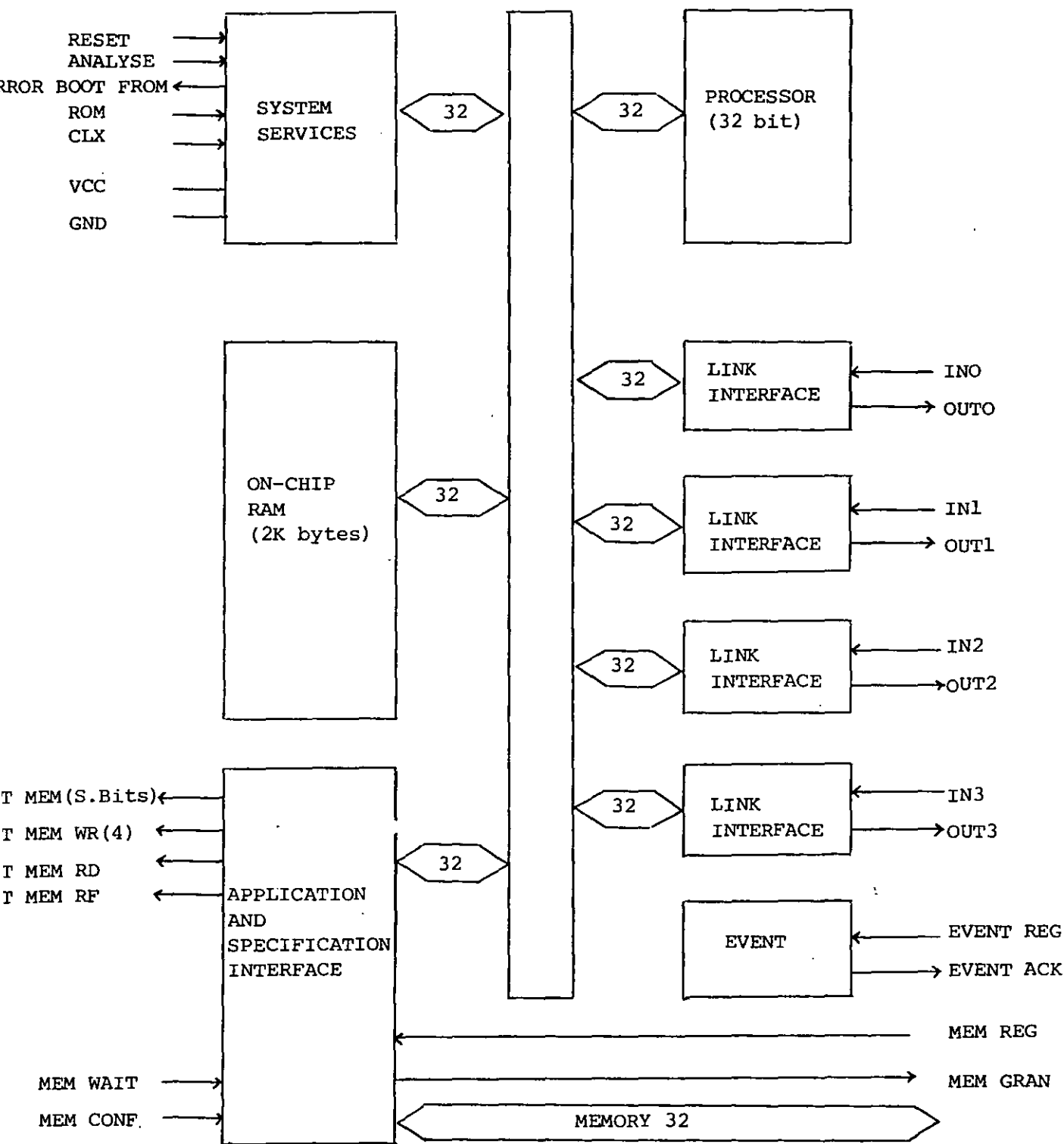


FIGURE 2.7: Transputer Architecture

communication. Different types of transputers can have different instruction sets depending on the required balance between cost, performance, internal concurrency and hardware, without altering the users view of OCCAM. Hence the transputer is a Reduced Instruction Set Computer (RISC).

The processor contains a scheduler which enables any number of processes to run on a single transputer sharing processing time, while each link provides two unidirectional channels for point to point communication synchronised by a handshaking protocol. Communication on any link can occur concurrently with communication on other links and with program execution.

OCCAM itself is based on communicating sequential processors [Hoare 1978] where parallel activities are viewed as black boxes with internal states, called processes, and which communicate with each other using a one-way channel. Communication is achieved by sending a message down a channel between two processes; one process sends a message and another reads it from the channel.

As every transputer implements OCCAM, an OCCAM program can be executed on a single transputer or a network of transputers. In the former case, parallel processes share the processor time and channel communication is simulated by moving data in memory. For a transputer network processes are distributed among transputers and channels allocated to links.

The main characteristic of the OCCAM language is its simplicity which makes it an appealing prospect for proving the correctness of the processes. It has fewer than thirty keywords, and only a small

number of constructors. Although each process used destructive assignments, the use of channels for interprocess communication makes it entirely consistent with data flow and graph reduction computer architectures. OCCAM was designed with computer architectures of this nature in mind, and with a view towards fifth generation applications. Together with the Inmos transputers, it provides a modular hardware/software component of the type which is essential in the construction of highly parallel computer systems.

However, its lack of a powerful data structure and its closeness to the hardware, means that OCCAM is likely to be the low-level language of fifth generation systems with applications possibly written in a more abstract language.

2.3.4 Simulation of Systolic Arrays

We use the fact that OCCAM programs can be divorced from transputer configurations by using the language as a simulation tool throughout the development of our simulation system in this research. A brief summary of the OCCAM language is given in Chapter 4. The general structure of OCCAM programs which represent the simulation of systolic arrays is shown in Figure 2.8, where branching indicates parallel execution. The construction of programs follows ideas developed by G.M. Megson [Megson 1984]. Consequently OCCAM programs simulate the formal proofs by replacing I/O descriptions by actual results. Although the simulation does not guarantee correctness it is nevertheless a less time consuming approach which does not result in unsolvable equations. Furthermore, a working OCCAM program retains the possibility

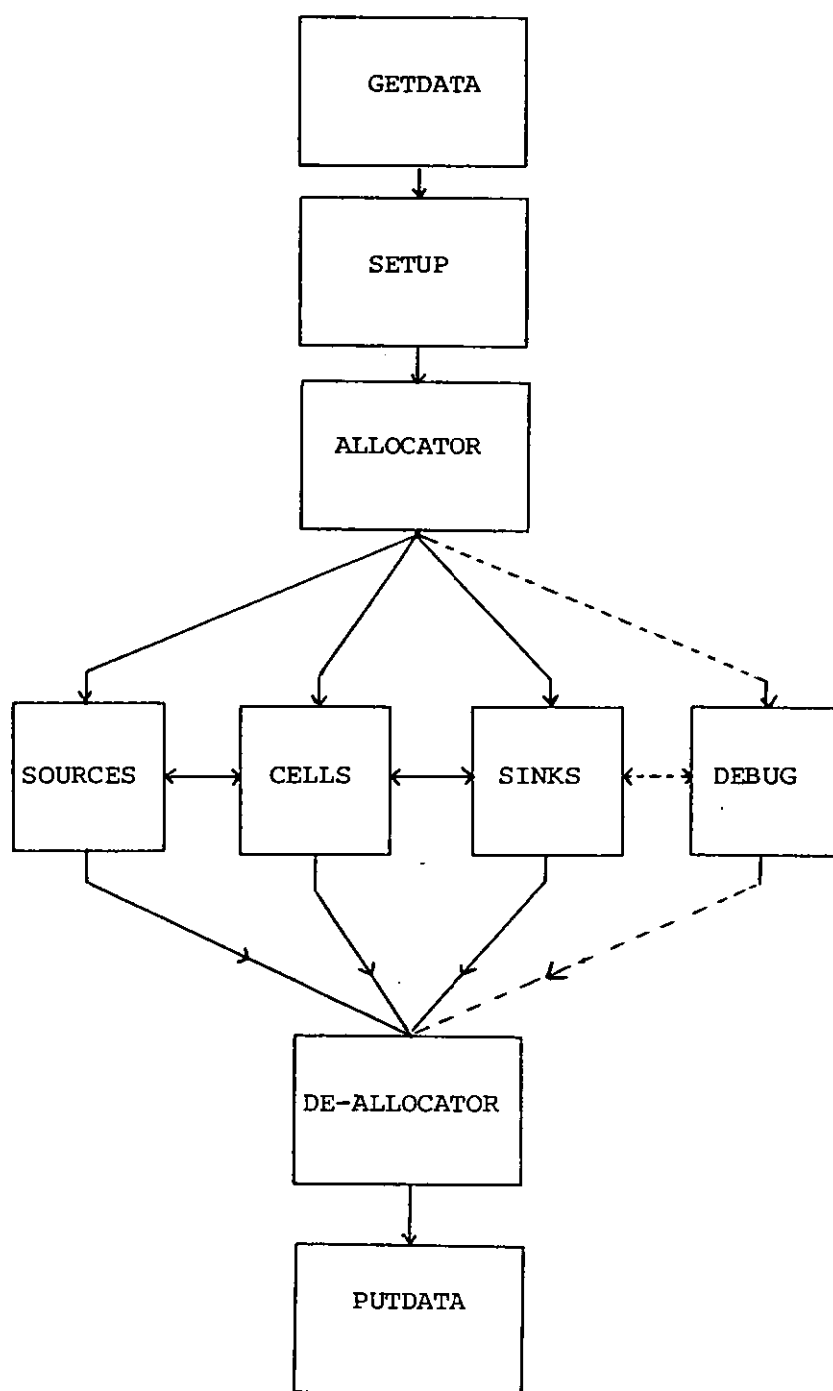


FIGURE 2.8: Structure of OCCAM Program for Simulating Systolic Arrays

of actual transputer implementation and so solves two problems in one attempt.

The getdata and putdata sections of Figure 2.8 which represent the host machine interface, are responsible for receiving and sending data and control to and from the program. Each routine contains enough memory to store the initial array input data and the final output data corresponding to the global input and output sequences of the model. In principle, the two routines can be run in parallel with each other and the array, but generally they are sequential, in order to emphasise the parallel operation of the array. The actual host can be predefined I/O files or simply the terminal. The former method is useful for buffering and throughput testing, while the latter helps with debugging and interactive array performance. The routines can be augmented with user friendly features directing the program use, the collection of data necessary for the array construction and formatting of results.

The setup routine is a key section of the algorithm which computes array dependent quantities. More specially, it performs many necessary calculations whose values are useful in defining the structure of the array. These structural values are more important as the array becomes more complex.

Sources, sinks and cells are OCCAM procedures that define the network model. A source is located initially with a vector from getdata representing its associated bounded data sequence, together with additional values from the set-up routine. Sinks are analogous to sources except they work in inverse by placing real values into data vectors which are then passed to putdata for output. The cell

procedures implement the n-ary sequence operators. Generally there is one procedure for each type of cell, and the programming task is simplified for homogeneous networks. The I/O sequences are represented by OCCAM channels appearing as actual parameters in the procedure headings. Where cell definitions are only marginally different, extra switches and flags can be added to a procedure heading so it can set up the correct cell type. This collapses a number of definitions onto a single generic one. Extra parameters can also be used for preloading array values.

A cell definition is divided into three sections, initialization, communication and computation. Initialization is performed only once and allows cells to be cleared before use or predetermined values to be set up. In particular, initialization defines neutral element quantities which can be used in communication before real data reaches the cell and is essential to maintain dataflow in OCCAM programs. The communication and computation sections of the cell are performed many times and are enclosed in a loop for iteration, and are performed sequentially one after the other. All communication is performed in parallel and computation is mainly sequential. The allocator routine is called after setup and is supplied with parameters about the array dimensions, synchronisation details of the total number of cycles in the algorithm, if a loop scheme is used, and data sequence sizes. The allocator is simply a set of parallel loops which specify and start-up the computational graph by connecting corresponding procedures using OCCAM channels as arcs and allocating channels accordingly. To achieve setup, the graph is mapped onto a grid of points whose points and

hence arcs can be recovered from a simple address type calculation. The simpler the array the easier are the mapping functions, and the result is an allocation similar to the VLSI grid model. Once started the sources and sinks control the computation, and the allocator only terminates when all the graph cell procedures have terminated. Termination of procedures is assumed to be globally synchronised if a for-loop is used in cells and asynchronous if while-loops are incorporated. As OCCAM is an asynchronous communication language, for-loops tend to be messy requiring some additional computation after the loop to clear all the channels - hence avoiding deadlock. While-loops are better suited to the model of concurrency and when augmented with systolic control sequences can be used to selectively close down cells, input and output channels. Consequently array cells can be switched off or de-allocated by a wavefront progression or pipelined approach from sources to sinks.

An additional procedure for debugging purposes can be added which runs in parallel with graph networks, and is mainly a screen/file mixer routine. The allocator sets up the procedure and network cells are augmented with an additional channel each, which the debug routine uses to analyse cells. Debug channels are allocated from a pool of channels and require an ordering of network cells for correct indexing. When the indexing function is simple, debug can be used to output snapshots in a sequential cell-ordering and the additional debug channel communication must be placed carefully in cell definitions.

Finally, the techniques described above have been used successfully to implement designs in OCCAM by G.M. Megson [Megson 1987], but can in

principle be extended to any parallel language provided channels and cells can be modelled.

In fact Brent, Kung and Luk [Brent 1983] used an extended version of Pascal. ADA also seems a likely candidate as ADA rendezvous is very similar to channel communication both being based on CSP. The adoption of OCCAM offers more direct hardware support for special purpose designs as well as common architectures.

2.4 MIMD ARCHITECTURE DESIGN - THE SEQUENT BALANCE SYSTEM

2.4.1 MIMD Hardware Organisation

One of the motivations of the MIMD computer design is the increase in computational speed-up by the concurrent execution of instructions, organised in several sequential streams with infrequent dependencies among them, by a large pool of processors with approximately similar capabilities. Of importance to this type of structure is the mechanism to synchronise and communicate between processors. Specifically the used mechanisms can be classified into two classes, those that use a shared memory, and those that use passing messages (see [Baer 1976], [Enslow 1977] and [Stone 1980]). The use of the shared memory which might be a multiported main memory, cache memory or a multiported disk, results in a faster mechanism but requires all the processors to access the shared memory. Consequently, this limits the total number of processors that the system can effectively handle. On the other hand, the mechanism based on messages has a large overhead so that it is only useful when synchronisation and communication are very infrequent [Gehrig 1982].

The general class of MIMD computers was distinguished into two main classes, the tightly-coupled and the loosely-coupled systems depending on the amount of interactions between the processing elements (see [Hayes 1978]). In the case of tightly coupled processors, as shown in Figure 2.9, (i.e. a large number of processors sharing a common parallel memory via a high-speed multiplexed bus), the processors operate under the strict control of the bus assignment scheme which is implemented in hardware at the bus/processor interface. On the other hand, in a system with loosely-coupled processors the communication and

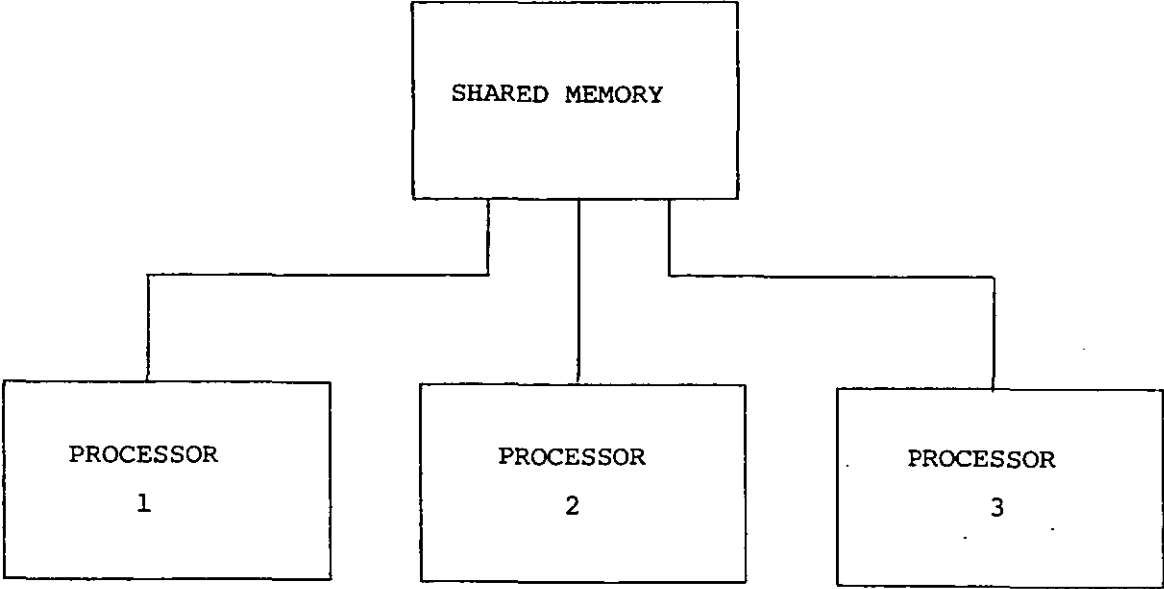


FIGURE 2.9: Tightly-Coupled Multiprocessor System

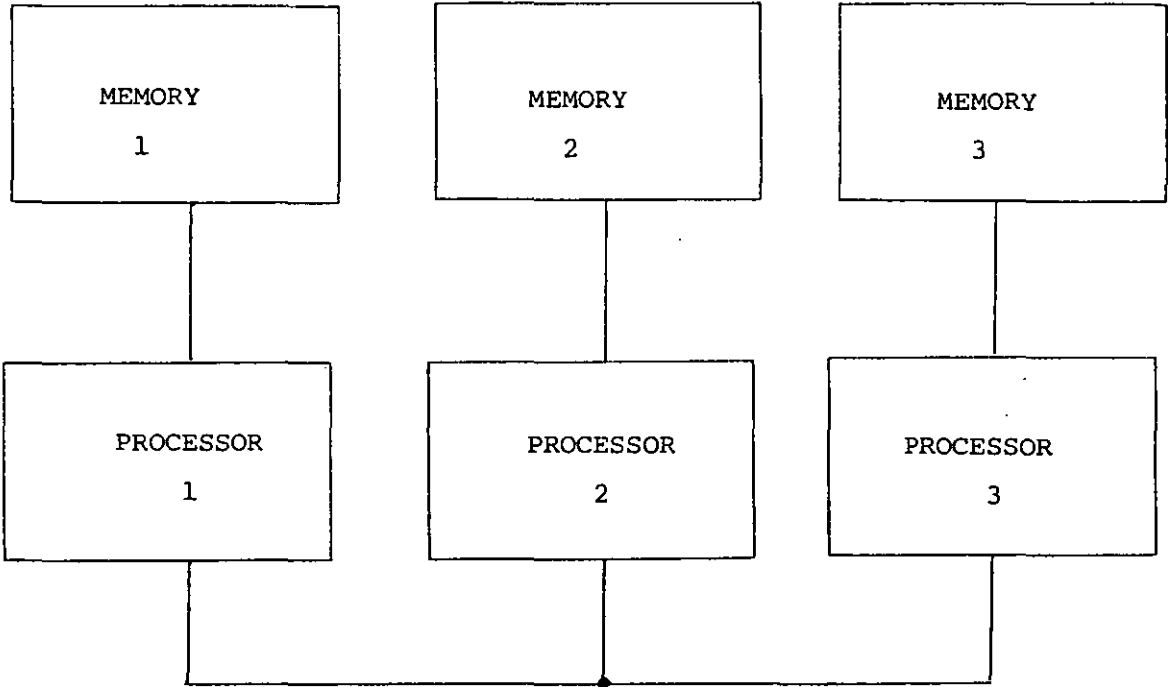


FIGURE 2.10: Loosely-Coupled Multiprocessor System

interaction takes place on the basis of information exchange. Figure 2.10 shows a general architecture of a loosely coupled system where each processor has its own local memory. Comparing the above two classes of multiprocessor systems, the main difference lies in the organisation of the memory and the bandwidth of the interconnection network.

Several interconnection networks with different characteristics such as bandwidth, delay and cost, ranging from the shared common bus to the crossbar switch have been proposed.

However, Enslow identified three fundamentally different organisations, namely the time-shared common bus, the multiport memory and the crossbar switch.

The time-shared common bus interconnection scheme, as illustrated in Figure 2.11, represents the simplest form of connecting all the functional units using a single bus which incorporates some arbitration logic associated with every bus/unit interface to resolve the bus request contention since only one transfer can take place at any given time. Thus, the unit wishing to initiate a transfer, a processor or an I/O unit, must first determine the availability state of the bus, then address the receiving unit as well as determining its availability and capability to receive the transfer.

By its nature, such a system is quite reliable and its cost is relatively low, however several limitations are introduced that can have serious damaging effects on both the system, since a malfunction of any unit interface causes a system failure, and the total overall transfer rate.

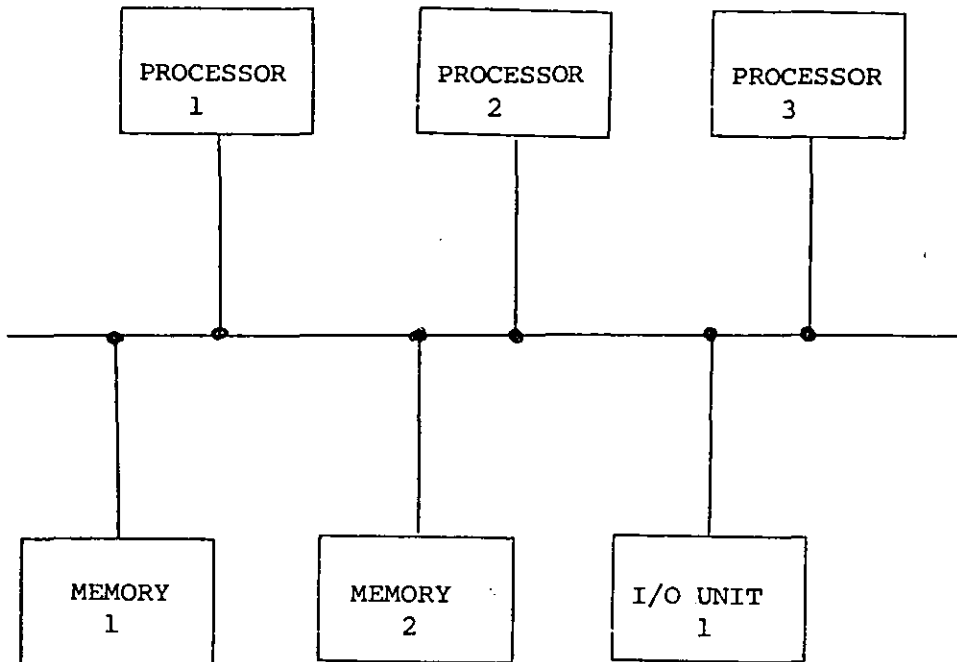


FIGURE 2.11: The Time-Shared Common Bus Interconnection System

Several interconnection systems such as the use of two one-way paths and multiple two-way buses have been provided in an attempt to solve this problem of a single transfer. The former example which does not increase system complexity or diminish reliability has a comparable performance with its predecessor since a single transfer requires the use of both paths. On the other hand with the latter technique multiple simultaneous transfers are possible but at additional system complexity.

The most extensive and expensive interconnection network providing a separate path for every processor, memory module and I/O unit is the crossbar switch (see Figure 2.12).

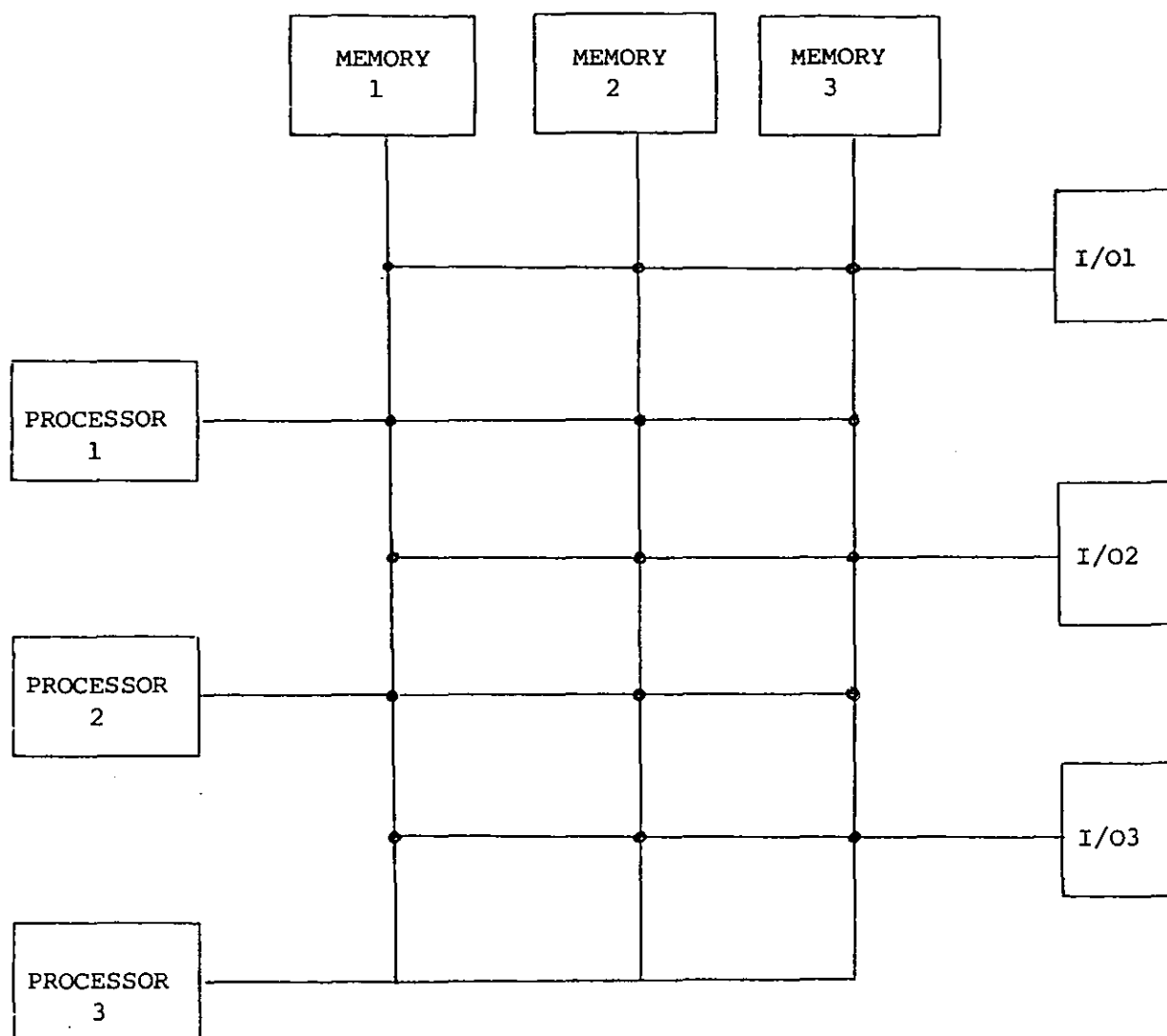


FIGURE 2.12: The Crossbar Switch System

In the case that the multiprocessor system contains p processors and m memories, the crossbar requires $p \times m$ switches, each of which is capable of switching parallel transfers and arbitrating conflicting requests. In this system, the bus-interface logic required by the functional units is kept at the lowest level since some of the functions, i.e. transfer recognition and conflicts resolution, which are performed at every bus-unit interface, are assumed by the switch

matrix. Consequently, such an interconnection is very complex (exponential growth for large p and m), expensive and physically large. However the important characteristics of this system which is shown in Figure 2.12, are the extreme simplicity of the switch-functional unit interfaces and the ability to support concurrent transfers for all memory modules.

The interconnection of the control, switching and priority arbitration logic, which are distributed throughout the crossbar switch matrix, at the interface to the memory modules leads to the multiport memory organisation, as shown in Figure 2.13, where every processor has a private bus to every passive unit, i.e. memory and I/O units. The multiple ports of every passive unit, one for each connection to a processor, are assigned fixed priorities through which arising conflicts are resolved.

This organisation offers a high potential transfer rate within the system at a comparable hardware complexity with that of the crossbar switch except for the localised logic, but with a severe constraint on the number of processors imposed by the number and type of the memory ports.

Besides these three presented interconnection networks, there are many others which can be valuable for the multiprocessor organisation such as the Omega network [Lawrie 1975] and the Delta network [Patel 1981] and the Augmented Data Manipulator [Siegel 1979].

The interference or conflict, produced in the accessing of a shared memory in a multiprocessor system, which is one of the factors that degrade the overall performance of the system has been

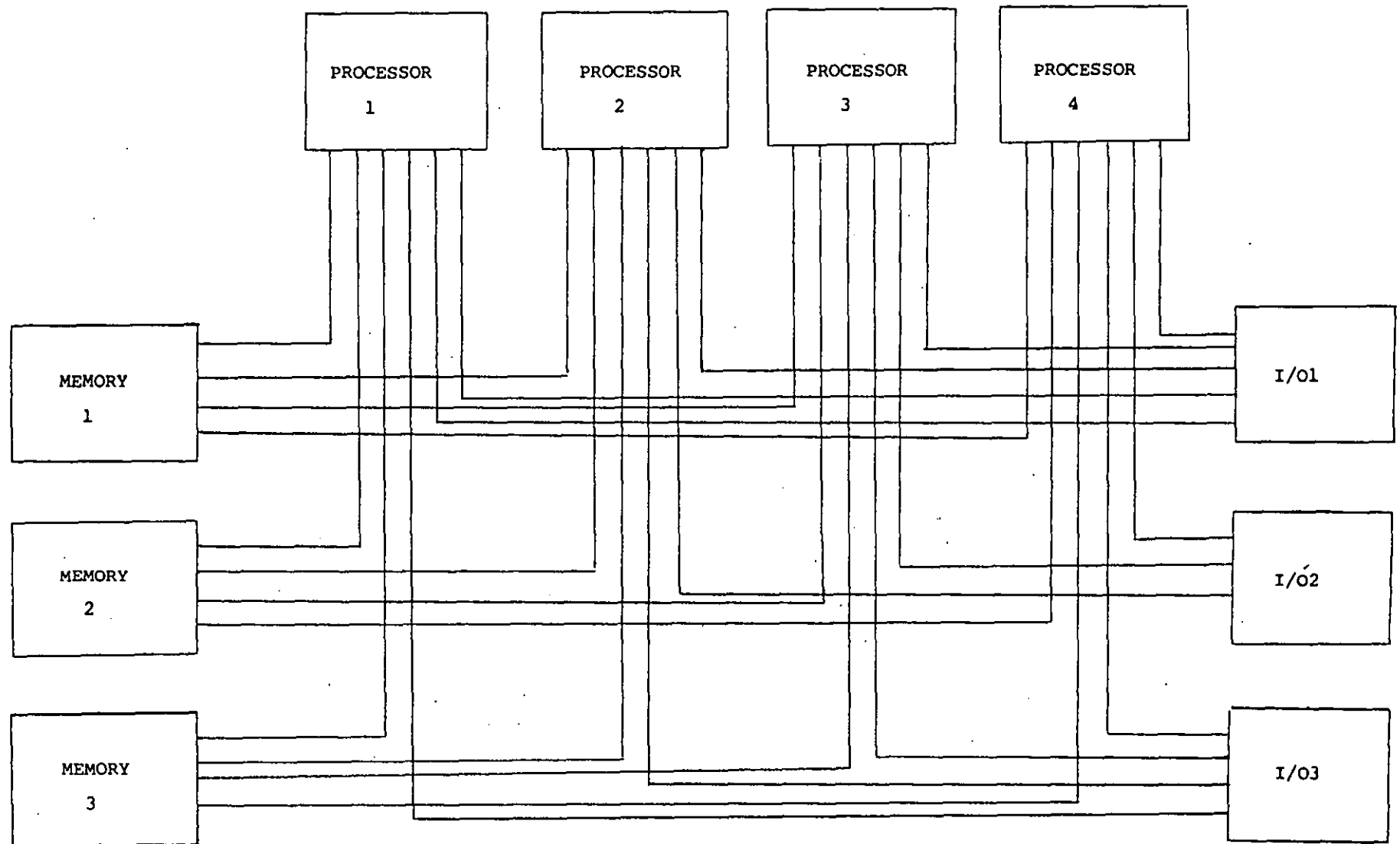


FIGURE 2.13: The Multi-port Memory Interconnection System

investigated extensively, resulting in some exact and approximate modules under various assumptions [Chang 1977], [Janek 1981], [Janek 1982], [Lillevik 1984] and [Basket 1976]. These interferences can be generally classified into two types: software and hardware types.

The first memory conflict is caused by a processor attempting to use a data set while it is currently being accessed by another processor which has eventually activated a software 'lock' mechanism to prevent any other processor from accessing the same data set. Thus, although this action forces serial manipulation of some sensitive data sets through a software mechanism, called critical region it ensures data integrity in a multiple processor environment.

On the other hand, the second type of memory conflict is caused when two or more processors attempt to access the same memory module simultaneously, i.e. more than one request is made to the same module during a single memory cycle by different processors. Therefore, all but one request must wait to be served sequentially since only one access can be made per memory cycle. Thus, programs with a large number of these conflicts have greater degradation in their overall performance.

A way to reduce the processor interconnection network and the interference in the memory is to have a cache memory associated to each processor. The main difficulty with this approach is the coherence problem that appeared when shared data is present simultaneously in several caches. Another solution to this problem is to partition the physical memory into local memories while keeping the uniform access at the virtual level. To reduce even further the

cost of the interconnection network, it is useful to divide the processors into clusters and have a slower interconnection between clusters. This approach is implemented in the Cm* [Gehrig 1982].

2.4.2 The Sequent Balance 8000 System

The Balance 8000 which was developed by Sequent Computer System Inc., Oregon, using a new processor pool architecture was installed in Loughborough University, Computer Studies Department in 1986. This system dynamically shares its load among twelve architecturally similar processing units and operates under a single copy of a Unix-based operating system, known as DYNIX, capable of delivering up to 5 MIPS. The pool processing organisation requires dynamic balancing of the system workload among the processors with an effective use of all resources in general. Consequently the system automatically and continuously assigns tasks to run on any processor that is currently idle or busy with a lower priority task, meaning that a process does not necessarily run to completion on the same processor but on the contrary it may involve several processors. This balancing process is carried out transparently; neither the user nor the programmer need to be aware that the system supports multi-tasking operations.

From the hardware point of view, the Balance 8000 consists of a pool of two to twelve processors, a bandwidth bus, up to 28 Mbytes of main memory, a diagnostic processor up to four high-performance I/O channels and up to four IEEE-796 (multibus) bus couplers. Figure 2.14 shows the main functional blocks of the Balance 8000 system.

Each processor is a subsystem containing three VLSI components:

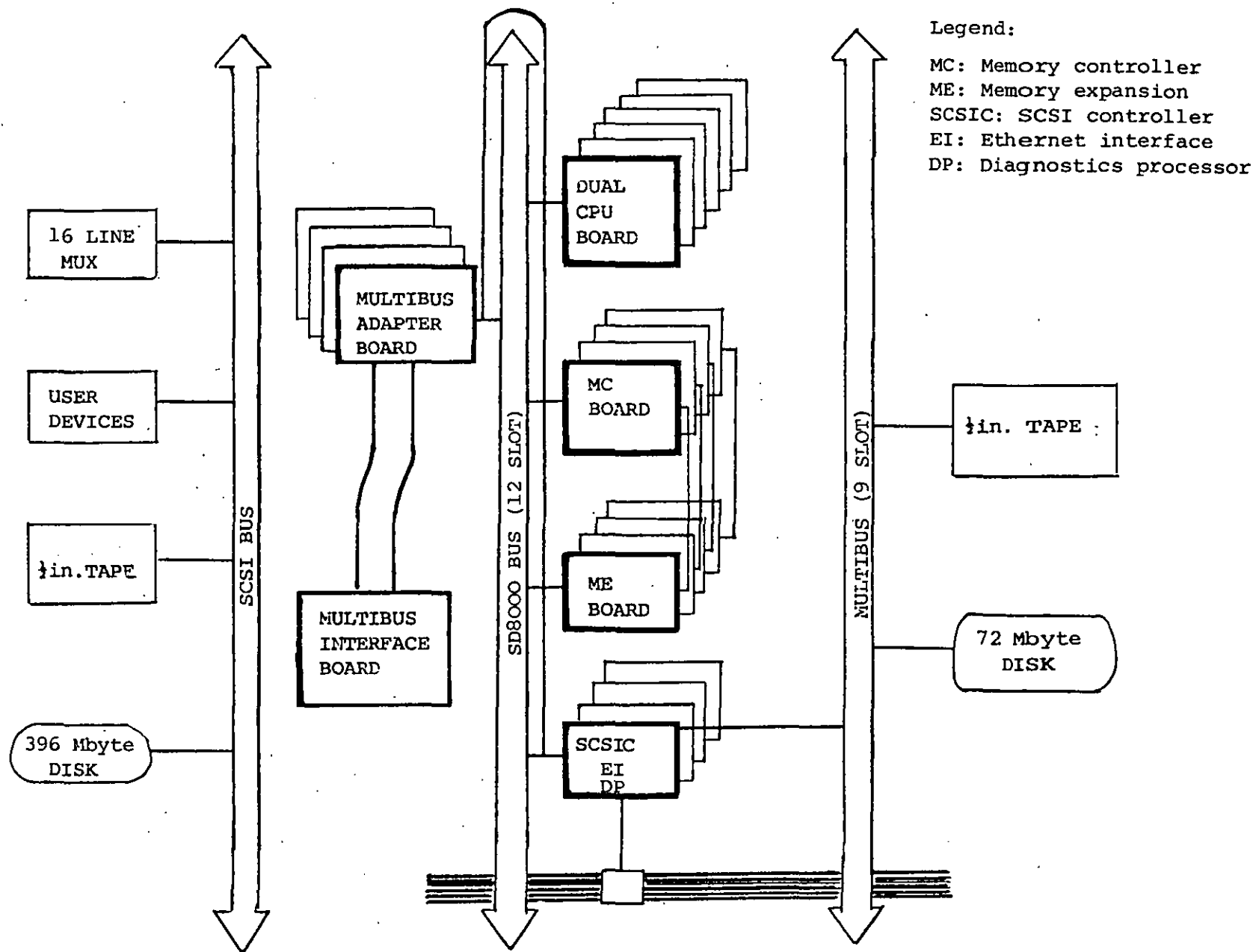


FIGURE 2.14 The Balance 8000 System Configuration

a 32-bit processing unit, a hardware floating-point unit and a paged virtual memory management unit. Two such subsystems are on one circuit board (see Figure 2.15 which shows the major units of a dual processor board). Also each processor contains a cache memory that almost reduces to zero all the processor waiting periods and minimises the bus traffic. The two-way set-associative cache consists of 8 Kbytes of very high speed memory and stores recently accesses instructions and data, so subsequent requests for the same data are satisfied from the cache, rather than from the main memory.

However, with the use of these cache memories two coherence problems arise, mainly the coherence of the data between the main memory and the caches on each processor and the coherence of the data between the caches themselves. For the former problem, a write-through mechanism is utilised in order to keep the main memory up-to-date with all the eventual changes made in every processor's cache. In addition to the update of the appropriate cache, this mechanism would allow the same write cycle to pass to the bus and memory. In the latter case, the answer is provided by the bus watching logic implemented in every cache. Consequently, all the write cycles on the bus are monitored and the addresses are compared with those in the cache, so whenever the contents of the cache are altered, the cache invalidates the entry in question.

Significant processing time is saved by including a write-buffer in each processor which can proceed immediately after issuing a write cycle letting the buffer wait for the memory cycle to complete.

Finally, to complete the description of the components found in

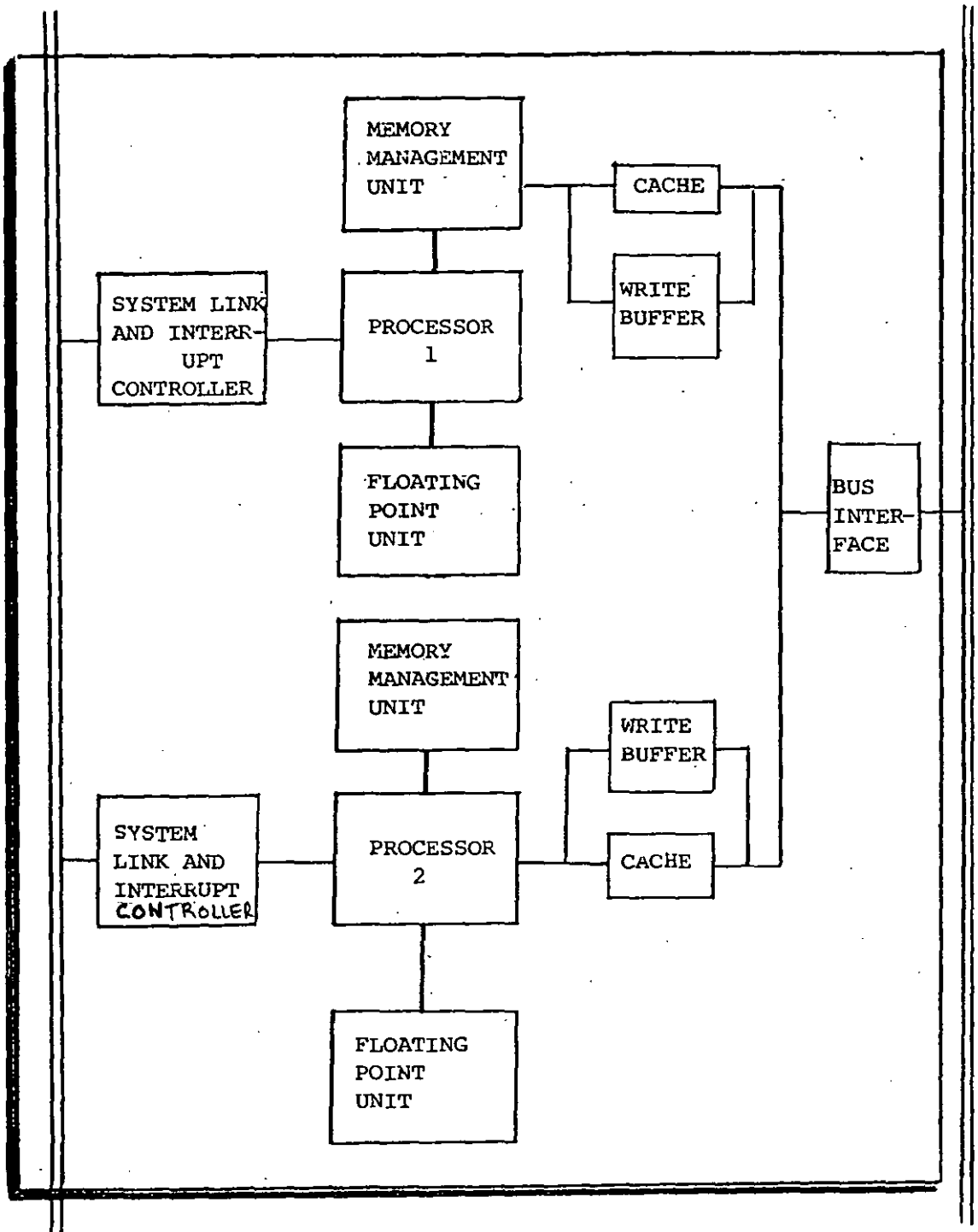


FIGURE 2.15: Configuration of a CPU Board with 3 VLSI Components Attached to Each Processor

the processor subsystem we need to refer to the system "System Link Interrupt Controller" (SLIC) which is a chip, one for each processor and for every other board, attached to the SB 8000 bus. This SLIC chip manages interprocessor communication, synchronised access to shared data structures, distribution of interrupts among the processors, and diagnostics and configuration control. The SLIC bus which is a part of the SB 8000 system bus provides an inter-connection for communication among the SLIC chips.

The SB 8000 system bus is a 32-bit wide, pipelined, packet bus supporting multiple overlapped memory and I/O transactions and capable of achieving a throughput rate of 26 Mbyte/sec. It also supports several packet lengths and checks parity to aid in error detection.

This system provides up to 28 Mbytes of principal memory, a 4 Mbytes I/O address space that can be shared by all the processors and a 16 Mbyte virtual memory address space for each process. The Balance 8000 supports up to four memory controllers, each with an optional expansion board, reducing memory contention among processors. It also supports standard I/O throughout the system, and permits several instances of each interface to increase the I/O bandwidth. More specifically this system supports a SCSI interface for disc and tape I/O, a Multibus interface for serial communications, large disc and tape support, and user-added devices, and finally an Ethernet local area network for communication amongst systems.

CHAPTER 3

THE INSTRUCTION SYSTOLIC ARRAY (ISA) -

A PARALLEL ARCHITECTURE FOR VLSI

3.1 THE INSTRUCTION SYSTOLIC ARRAY (ISA)

Systolic arrays have proved to be well suited for VLSI technology since they:

- consist of a regular network of simple processing cells,
- use local communication between the processing cells only,
- exploit a maximal degree of parallelism.

However, systolic arrays have one main disadvantage compared with parallel computer architectures: They are special purpose architectures only capable of executing one algorithm, (or a collection of related problems in a generic array) i.e., a systolic array designed for sorting cannot multiply matrices, whereas a systolic array for matrix multiplication cannot solve pattern matching problems and so on.

Several approaches have been made to make systolic arrays more flexible, in order to be able to handle different problems on a single systolic array. In Hans-Werner Lang [Lang 1985] the instruction systolic array (ISA) has been suggested as a new architecture for parallel computation which meets the requirements of VLSI and be capable of efficiently executing a large variety of parallel algorithms.

The basic idea of this concept is illustrated in Figure 3.1. Instead of pumping data through the array of processing cells which can execute only one fixed instruction (as in a systolic array), the ISA moves the instructions through the array of processing cells. In addition to the vertical stream of instructions, a horizontal stream of selector bits is introduced (Figure 3.2). An instruction is

executed if it meets a selector bit '1', whereas execution of an instruction is suppressed if the selector bit is '0'. So the instructions may be executed only in certain rows of the processor array.

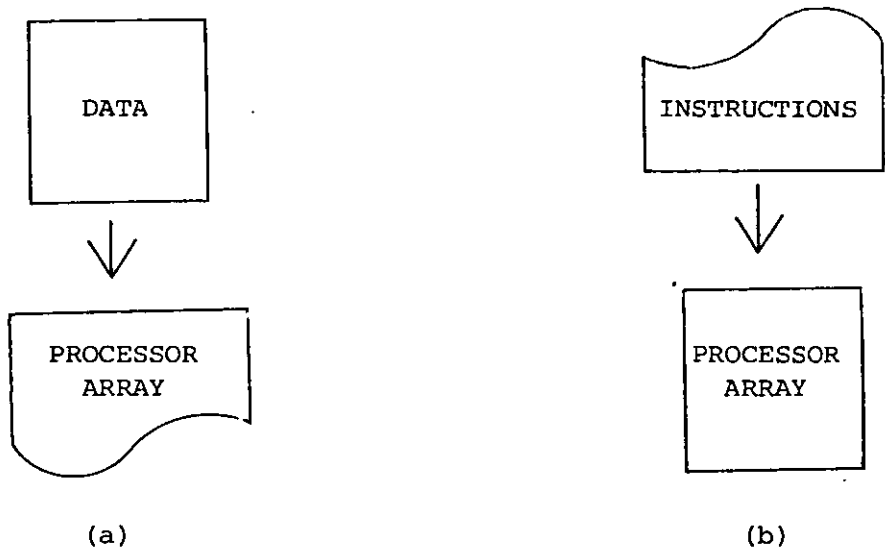


FIGURE 3.1: (a) Data is Shifted Through the Systolic Array
(b) Instructions are Shifted Through the ISA

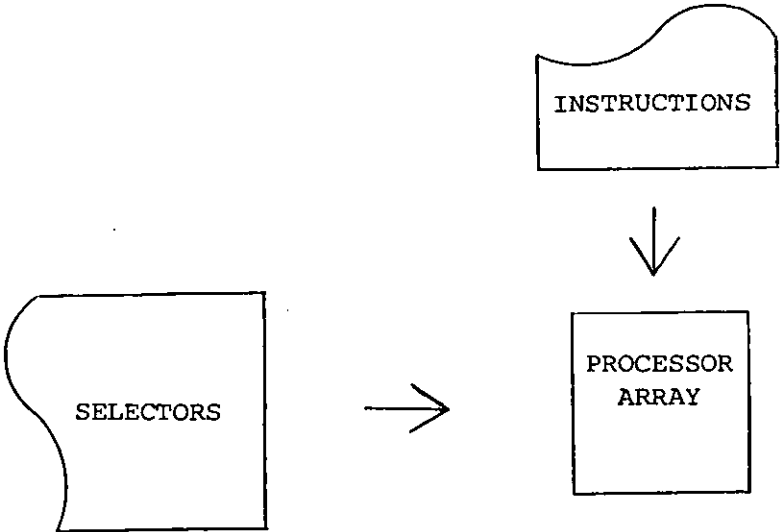


FIGURE 3.2: The Vertical Instruction Stream is Combined with a Horizontal Stream of Selector Bits

Our basic model of a parallel computer is a mesh-connected $n \times n$ -array of $N=n^2$ identical processors (Figure 3.3). The processors can execute instructions from a small instruction set. The processor array is synchronized by a global clock and the execution of every instruction is assumed to take the same time.

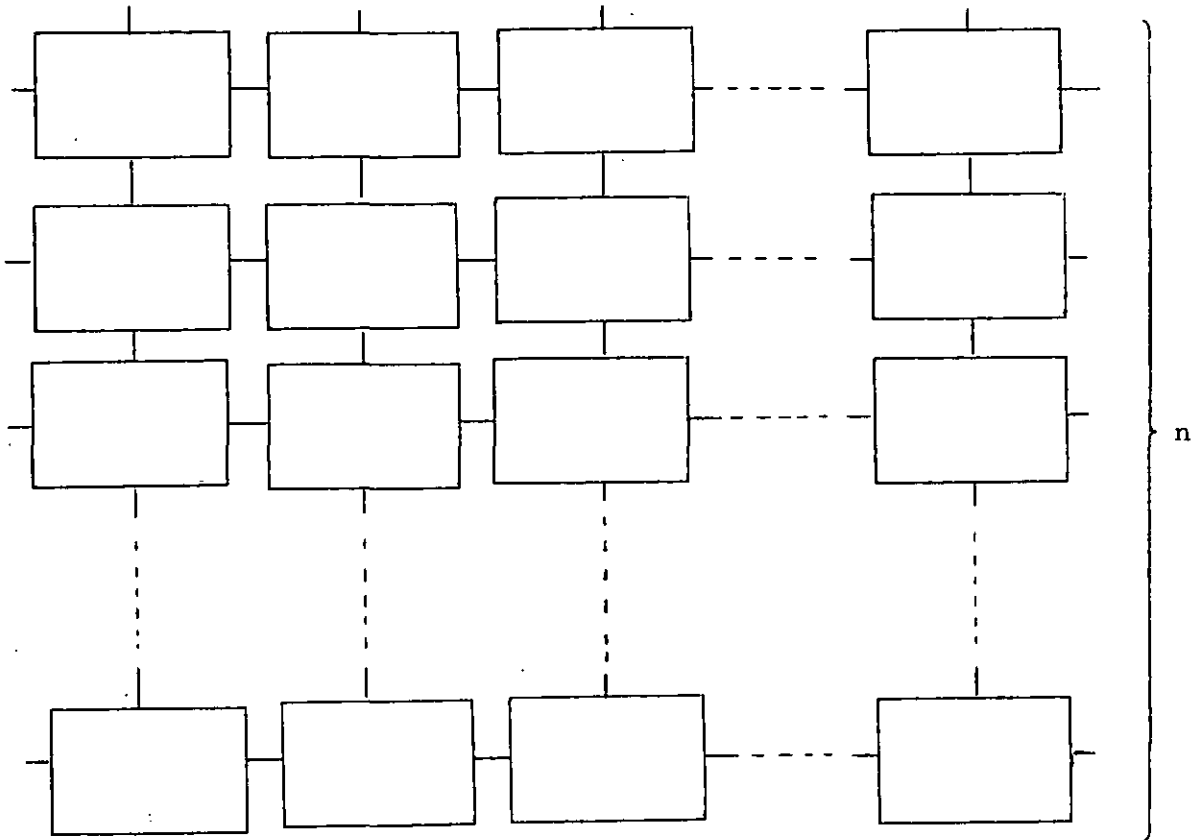


FIGURE 3.3: A Mesh-Connected Processor Array

Each processor has some data registers including a designated communication register (CR). Communication between two processors P and Q is done in the following way:

If a data item is to be sent from P to Q, P writes the data item

into its communication register. In the next instruction cycle Q reads the contents of P's communication register.

Each processor can write only into its own communication register, but it can read from the communication registers of its four direct neighbours. It is allowed that two or more processors read from the same communication register simultaneously. In order to avoid read/write conflicts we assume that reading from a register is done during the first half of the execution of an instruction and writing into a register during the second half (Figure 3.4), or any equivalent mechanism: it must be guaranteed that reading from a register always yields its 'old' contents (of a previous instruction cycle).

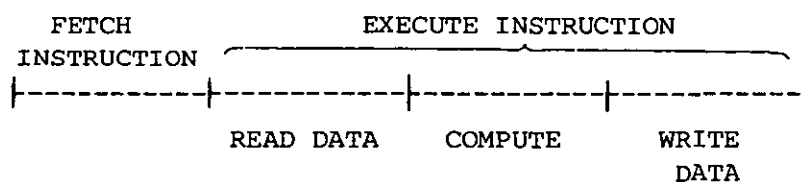


FIGURE 3.4: Instruction Cycle

The processors do not have their own control units but are supplied with instructions from outside. Each processor has an instruction register. At the beginning of each instruction cycle each processor fetches the instruction from the instruction register of its top neighbour. This is done synchronously, so that, by this mechanism, rows of instructions are shifted through the processor array from the top to the bottom. The processors in the top row of the array are supplied with instructions from a memory outside. In an analog way columns of selector bits are simultaneously shifted

through the array from left to right. A processor executes its instruction if and only if its selector bit is '1' otherwise it performs 'no operation' (no-op). More formally we define:

A program on an ISA consists of a sequence $p^{(1)}, \dots, p^{(r)}$ of n -tuples over the instruction set I and a sequence $s^{(1)}, \dots, s^{(r)}$ of n -tuples over $\{0,1\}$. For every $i, j \leq n$ and $t \leq r$, $p^{(t)}$ is the row of instructions which enters the i th row of the ISA at time $t+i-1$, and $s^{(t)}$ is the column of selector information which enters the j th column of the ISA at time $t+j-1$. That means, the instruction executed by processor (i,j) is:

$$p(i,j) = \begin{cases} p_j^{(t+i-1)} & \text{iff } s_i^{(t-j+1)} = 1 \\ \text{no-op} & \text{otherwise} \end{cases}$$

Input and output of data to the processor array is done via the open-ended processor links at the boundary of the array. The ISA is assumed to be embedded in an environment that is capable of:

- supplying the ISA with instructions and selectors,
 - supplying the ISA with input data and sorting its output data,
- both at the speed determined by the clock of the ISA chip.

The length of a program on an instruction systolic array does not affect its area requirements, whereas the complexity of many systolic algorithms is proportional to their time complexity [Ullman 1984]. The reduced area requirements imply that on the fixed area (of a chip) larger problem sizes can be treated than on comparable standard systolic architectures.

3.2 THE INSTRUCTION SYSTOLIC ARRAY AND ITS RELATION TO OTHER MODELS OF PARALLEL COMPUTERS

3.2.1 Basic Definitions

In this section we study the feasibility of the ISA concept by comparing it to other parallel computer concepts based on mesh-connected arrays (Figure 3.3).

Now, in the MIMD concept of parallelism, all the processors of a given array (denoted PA) can execute different instructions. That means the array consists of n^2 independent processors having their own control store. Similar to the ISA the PA uses local communication only, which makes it suitable for VLSI but the processors have to be much larger than the ISA. Therefore, on the same area the ISA concept can realise a larger degree of parallelism than the PA. Furthermore, a PA program (see later) may consist of up to n^2 different programs for the individual processors which have to be distributed over the array before the PA program can be executed as it filters through the array, whereas in the ISA the program is executed while it is moved through the array. Consequently it is easier to execute a pipelined sequence of different programs on the ISA than on the PA.

As mentioned in Section 3.1, the basic model parallel computer is a mesh-connected $n \times n$ array of n^2 identical processors (Figure 3.3), which is synchronized by a global clock. The processors can execute instructions from some instruction set, where the execution time of all instructions is the same. Each processor has some local memory including a designated Communication Register (CR). The communication between processors is done in the following way:

If a processor needs data from one of its four direct neighbours, it reads that neighbour's communication register. This means that at most five processors can read from a communication register simultaneously (including the processor itself). Reading and writing is done in the same manner as described in Figure 3.4. This timing assures a mutual exclusion of reading and writing in the communication register. The open-ended data links of the processors at the boundaries of the array are used for external input and the output of data.

Now, in the SIMD concept of parallelism, at every time unit, all active processors of the array execute the same instruction. The instructions are broadcast by a central control unit to all the processors. Since this involves signal propagation on long wires this concept is not suitable for implementation in VLSI technology.

A control structure in between the ISA and the SIMD array is the Instruction Broadcasting Array (IBA). Here, as in the ISA, new instructions are fed into the array at every step, but these instructions are broadcast to all the processors of a column and not pumped through the array. Because of the broadcasting, this model is less suitable for VLSI than the ISA. But since it is conceptually simpler than the ISA we include it in our comparative investigation.

The three architectures we consider in the following differ in how the control information is supplied to the processors [Kunde, Lang, Schimmler, Schmeck, Schroder 1986]:

(i) The Processor Array (PA)

Where each processor has its own control store, Figure 3.3.

(ii) The Instruction Broadcast Array (IBA):

This computer's structure is depicted in Figure 3.5. The processors need only a very simple control unit without a control store. Instructions are broadcast to all the processors of a column. In addition, selector information ('0' or '1') is broadcast to all the processors of a row. A '0' means that all the processors of this row are inactive, i.e. they execute a "no-operation"-instruction. A '1' means that all the processors of this row are active and execute the instructions that have been broadcast. If I_j is the instruction of column j , and s_i is the selector of row i processor p_{ij} performs operations according to:

$$p_{ij} = \begin{cases} I_j & \text{iff } s_i=1 \\ \text{no-op} & \text{iff } s_i=0 \end{cases}$$

(iii) The Instruction Systolic Array (ISA):

This computer's structure is depicted in Figure 3.6. It is identical to the IBA except that the instructions and the selectors information are not broadcast but pumped systolically through the array of processors. The instructions move row-wise north-south, and the selectors move column-wise west-east.

Corresponding to the informal description given above we now define the notion of a program on a PA_n , IBA_n , or ISA_n (whenever the side length n of the underlying array is relevant we write it as a subscript):

Let I be the set of instructions the processors can execute, where no-op is an operation contained in the instruction set which

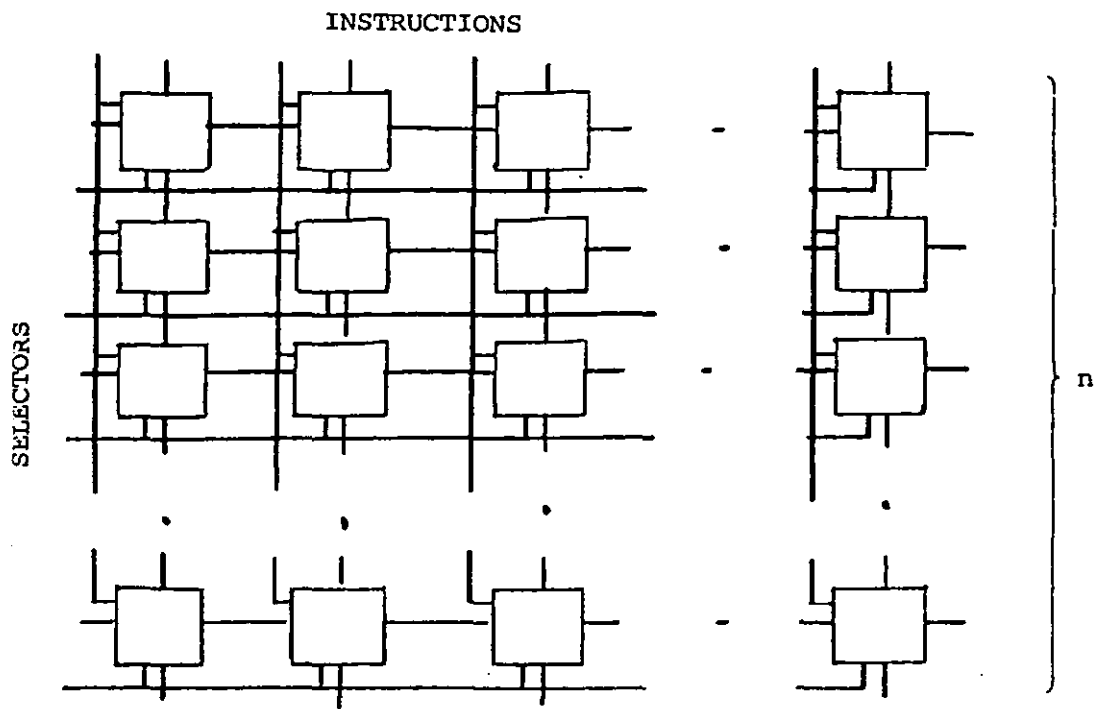


FIGURE 3.5: Instruction Broadcast Array (IBA)

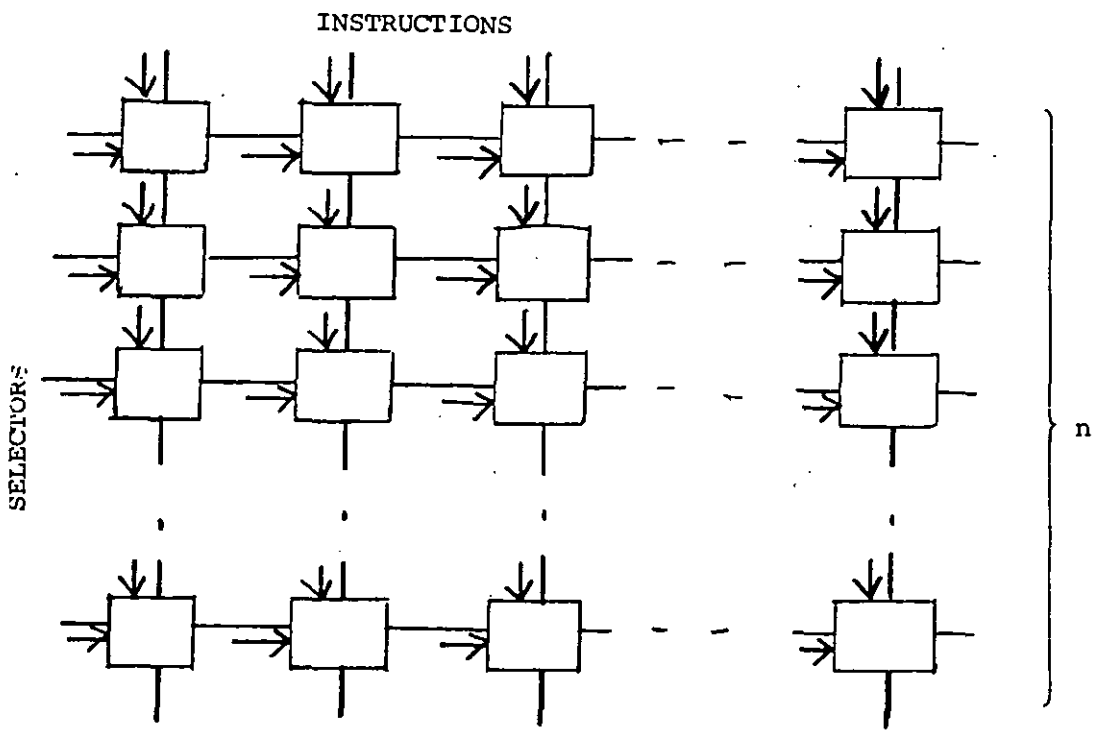


FIGURE 3.6: Instruction Systolic Array (ISA)

does not modify the processors memory contents.

A program on a PA_n : is a sequence $p^{(1)}, p^{(2)}, \dots, p^{(r)}$ of $n \times n$ matrices over I , such that for all $i, j \leq n$ and $t \leq r$ the instruction executed by processor (i, j) at time t is $p_{ij}^{(t)}$.

A program on an IBA_n : is a sequence $p^{(1)}, p^{(2)}, \dots, p^{(r)}$ of n -tuples (vectors) over I and a sequence $s^{(1)}, \dots, s^{(r)}$ of n -tuples (vectors) over $\{0, 1\}$ such that for all $i, j \leq n$ and $t \leq r$, $p_j^{(t)}$ is the instruction broadcast to all the processors of column j and $s_i^{(t)}$ is the selector information broadcast to all the processors of row i at time t .

Alternatively processor $p(i, j)$ executes according to:

$$p(i, j) = \begin{cases} p_j^{(t)} & \text{iff } s_i^{(t)} = 1 \\ \text{No-op} & \text{otherwise} \end{cases}$$

A program on an ISA_n : is a sequence $p^{(1)}, \dots, p^{(r)}$ of n -tuples over I and a sequence $s^{(1)}, \dots, s^{(r)}$ of n -tuples over $\{0, 1\}$. For every $i, j \leq n$ and $t \leq r$, $p^{(t)}$ is the row of instructions which enters the i th row of the ISA_n at time $t+i-1$ and $s^{(t)}$ is the column of selector information which enters the j th column of the ISA at time $t+j-1$. This means that the instruction executed by processor (i, j) at time t is:

$$p(i, j) = \begin{cases} p_j^{(t+i-1)} & \text{iff } s_i^{(t-j+1)} = 1 \\ \text{No-op} & \text{otherwise.} \end{cases}$$

Finally, the execution of an ISA program terminates after the last row of instructions $p^{(r)}$ has entered the first row of processors. Therefore, if the last instruction row is supposed to be moved down to the last row of the array, it has to be followed by $n-1$ rows of no-ops.

REMARK:

The definitions are easily extended to rectangular grids denoted $PA_{m,n}$, $IBA_{m,n}$ and $ISA_{m,n}$ with simple modifications to the i, j indices, where $m \neq n$. If p is a program on a PA_n , IBA_n or ISA_n , then $T(p)$ denotes the execution time of p , which is equal to the length of the program. Let $C = (C_{ij})$ be an $n \times n$ -matrix, where C_{ij} are the contents of the communication register of processor (i, j) of a PA_n , IBA_n , or ISA_n before the execution of program p . Then C_p denotes the corresponding contents of the communication registers after the execution of p (i.e. at time $T(p)+1$).

The input of a program p occurs every time a processor on the boundary reads from the communication register of a non-existent neighbour, i.e. everytime a processor uses one of the open-ended data links. From now on these "non-existent" communication registers are called Input Registers. The input of a program is specified by defining $E(p)$, the Environment of p , to be a $4n$ -tuple of strings of values that are read from the input registers during the execution of p . The contents of the communication registers of processors on the boundary of the array are viewed as "Potential Output". Therefore, these communication registers are called Output Registers of the array. The output of a program p then is a subsequence of the sequence of values of some output registers during the execution of p .

Finally, we define the equivalence of programs. Among the various conceivable notions we choose the following: Programs p and q on a PA_n , IBA_n or ISA_n are called equivalent, if for all identical environments $E(p)$ and $E(q)$, for every initial contents C , and for

every interpretation of the instructions occurring in p or q we have $C_p = C_q$. This type of equivalence could also be called Internal in contrast to an External Equivalence which could be defined with respect to output sequences instead of the final contents of the communication registers.

3.2.2 A Simple Example Program

To illustrate some of the basic definitions mentioned above, a simple parallel algorithm for merging two sorted data sets is implemented on each of the three models.

Algorithm Merge

Input: Two $\frac{n}{2} \times n$ arrays of data, the upper one sorted in right-to-left row-major order, the lower one sorted in left-to-right row-major order (Figure 3.7a).

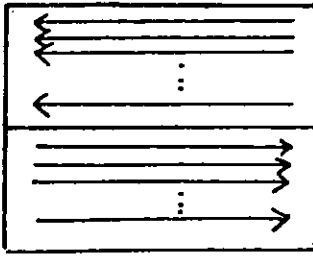
Output: One $n \times n$ array sorted in left-to-right row-major order.

Step 1: Sort all columns of $n \times n$ array by odd-even-transposition sort (Figure 3.7b).

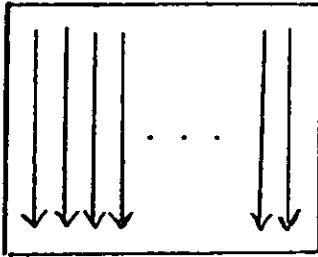
Step 2: Sort all rows of the $n \times n$ array by odd-even transposition sort (Figure 3.7c).

The validity of this algorithm is easily seen using the 0-1 principle [Knuth 1973]: If a sorting network sorts all sequences of 0's and 1's then it will also sort any sequence of elements chosen from an arbitrary ordered set.

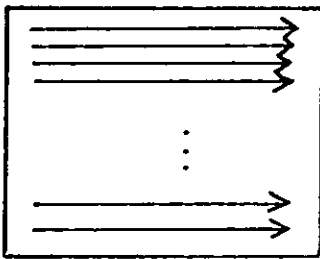
Thus, we may assume the sorted $\frac{n}{2} \times n$ -arrays to consist of 0's and 1's only (Figure 3.8a). After step 1 at most one row of the array consists of both 0's and 1's (Figure 3.8b). Therefore, step 2 yields the completely sorted $n \times n$ array (Figure 3.8c).



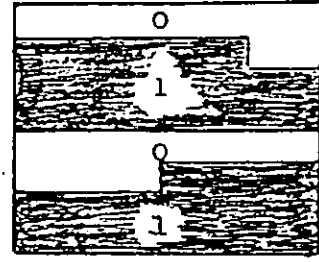
(a)



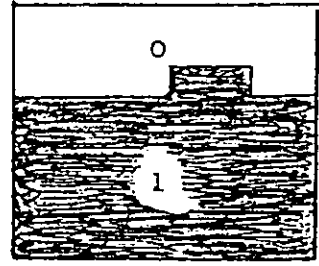
(b)



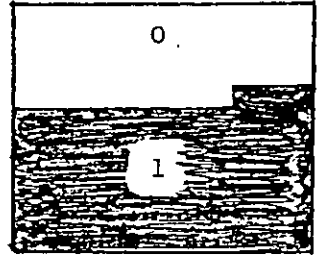
(c)



(a)



(b)



(c)

FIGURE 3.7: Merge Algorithm

FIGURE 3.8: Merging Two Arrays of 0's and 1's

The Merge Programs on PA_4 , IBA_4 and ISA_4

A PA_4 program for the merge algorithm is illustrated in (Figure 3.9a). Each square symbol of these matrices represents an instruction. The meaning of the instruction symbols is given in Figure 3.9b. By a simultaneous execution of a max and a min instruction of this kind a comparison-exchange of the two communication

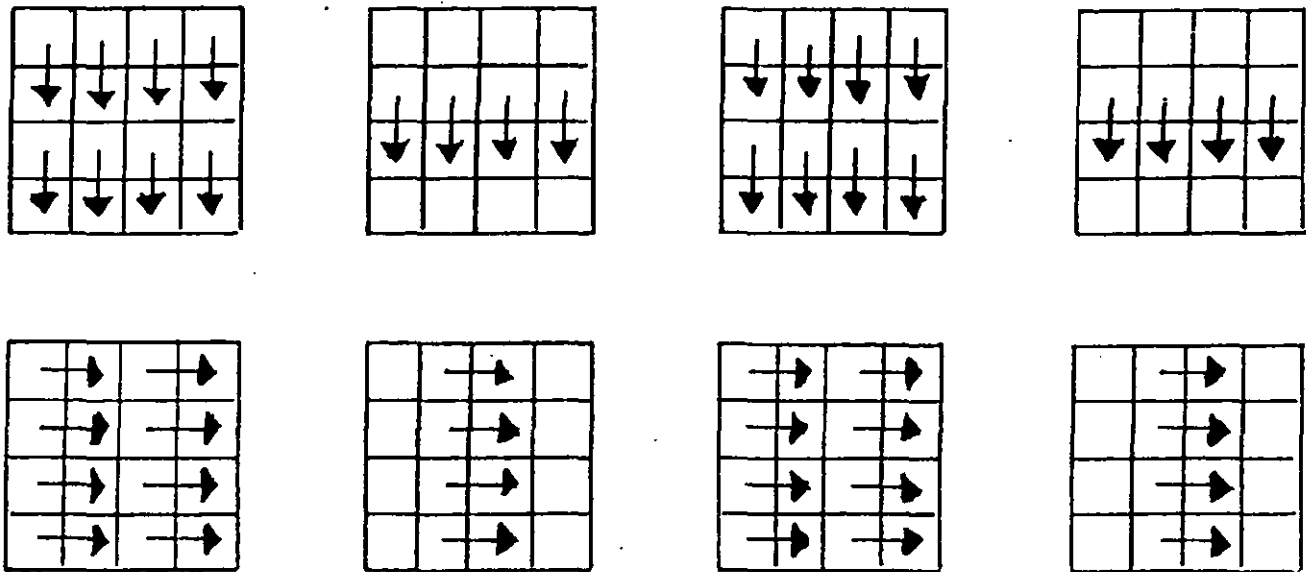


FIGURE 3.9a: PA_4 Program for Merge Algorithm

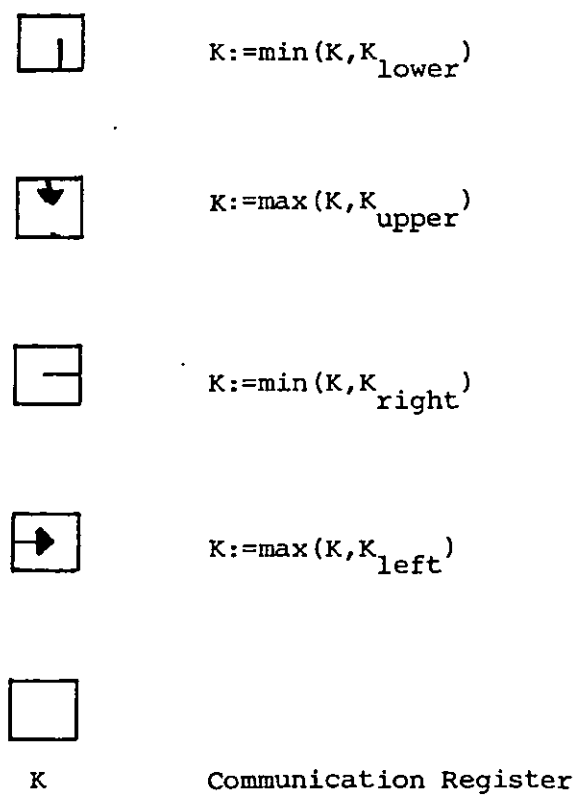
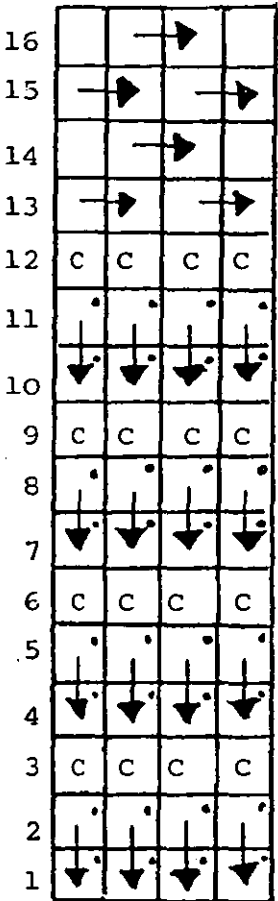


FIGURE 3.9b: Instruction Symbols and its Meaning

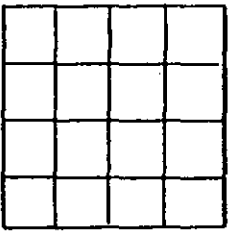
register contents is achieved. Matrices 1 through 4 correspond to step 1 of the algorithm and matrices 5 through 8 to step 2. Thus, the time of the program is 8 steps.

Figure 3.10a shows the IBA_4 version of the merge algorithm. Instruction rows 1 through 12 (together with the corresponding selector column 1 through 12) from step 1 of the algorithm, rows 13 through 16 step 2. Due to the broadcasting of instructions along the columns, in the IBA model the simultaneous execution of different instructions in one column is not possible. Therefore, modified instructions for the vertical comparisons are used [(Figure 3.10b, for details, see paragraph 3.2.3(b)]. This causes a delay factor of 3 in the execution time of step 1, leading to a time of 16 steps for the execution of the entire program.

The ISA_4 program for the merge algorithm is given in Figure 3.11. Now the meaning of the instruction symbols is again the same as in the PA program (Figure 3.8b). Both the instruction and the selector part of an ISA program can be viewed as diamond shaped, consisting of diagonals of instructions and selector bits, respectively. The i th selector bit in a selector diagonal tells whether the corresponding diagonal of instructions is to be executed in row i of the ISA or not. Diagonals 1 through 6 now correspond to step 1 of the merge algorithm and diagonals 7 through 12 to step 2. Note that this ISA program is not simply the skewed version of the IBA program from above, because on the ISA it is possible (and generally the case) that different instructions are executed simultaneously in one column. The time of the program is 16 steps.



1	1	1	1	1			1	1		1			1	1	
1	1	1	1	1	1		1		1	1	1		1		1
1	1	1	1	1			1	1	1		1		1	1	1
1	1	1	1	1			1		1	1			1		1



IBA

FIGURE 3.10a: IBA₄ Version of the Merge Algorithm

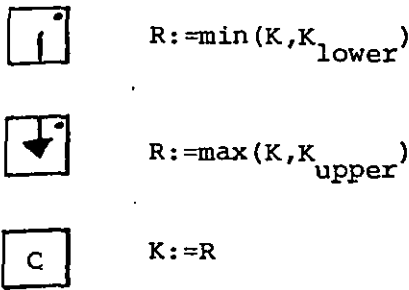


FIGURE 3.10b: Vertical Comparison Symbols and its Meaning

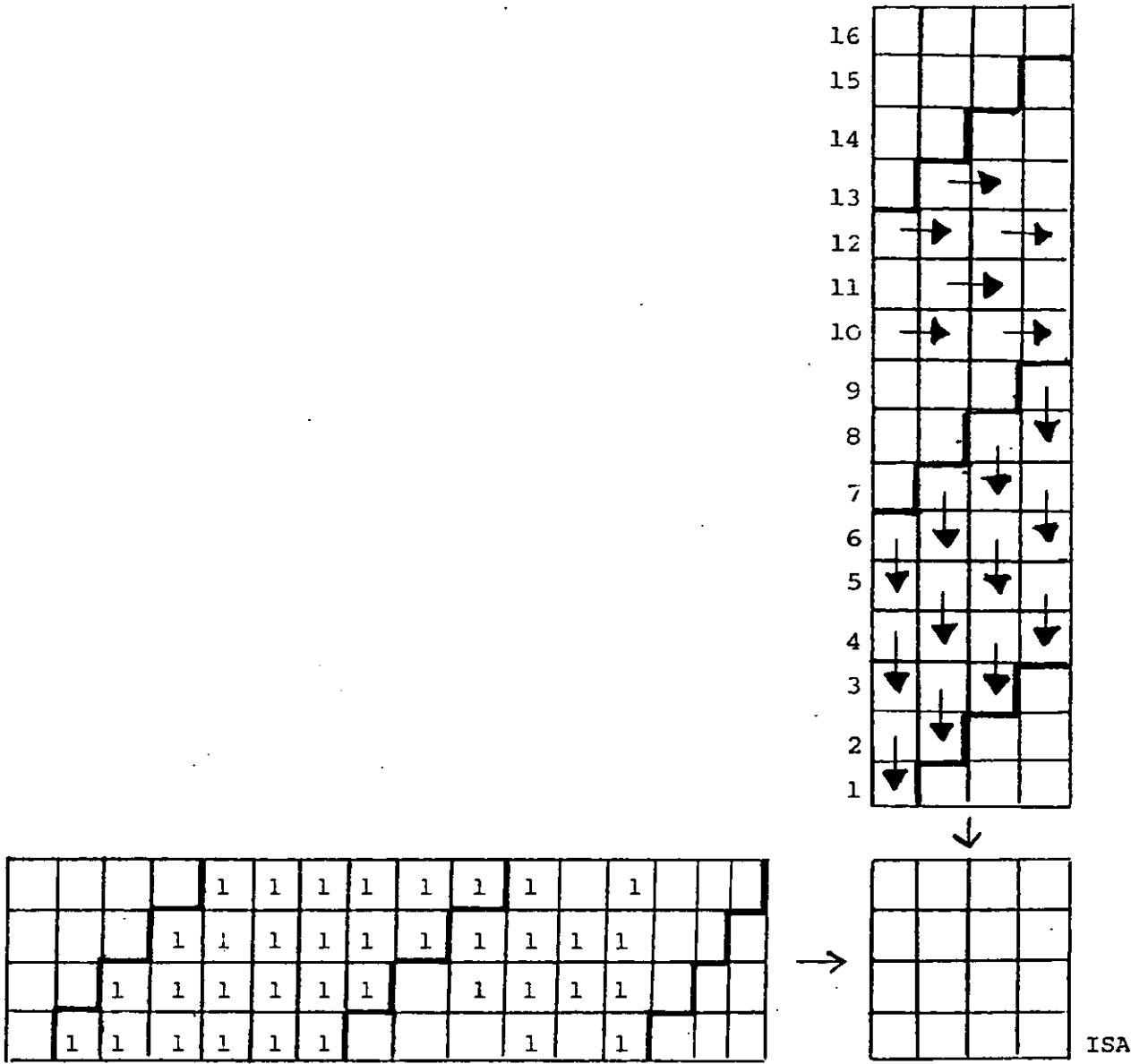


FIGURE 3.11: ISA Version of the Merge Algorithm

3.2.3 Relationships Between ISA, IBA AND PA

In this paragraph we study the relationship between our three different mesh-connected processor arrays, the PA, the IBA, and the ISA. For every pair of these different models we determine tight bounds on the worst-case delay introduced by a transformation of a program on one parallel computer model into a program on the other:

(a) ISA, IBA \rightarrow PA

Obviously, for every program p on an ISA or an IBA there is an equivalent program q on a PA such that $T(p)=T(q)$. Although there are programs that can be simulated by much faster programs on the PA, these are simple worst cases where no speed-up is possible. Therefore we get

Proposition 1:

Every program p on an ISA_n or an IBA_n can be simulated by an equivalent program q on the PA_n such that

$$T(q) = T(p) .$$

In general, it is not possible to achieve a speed-up.

(b) PA \rightarrow IBA

A simple way of simulating a program p on a PA by a program on an IBA would be to simulate every step $p^{(t)}$ of p by n steps on an IBA: In the i th of these n steps the i th row of $p^{(t)}$ is broadcast and only the i th row of the array is selected. However this may lead to problems, whenever an instruction of row i of $p^{(t)}$ reads from the communication register of a processor of row $i-1$, because in the suggested simulation it would read the possibly new contents of the communication register

after the execution of the instructions of row $i-1$ of $p^{(t)}$. Therefore, to produce an equivalent program on the IBA it is necessary to save the old contents of the communication register of a processor at least until all its four neighbours have read this information. This is achieved as follows:

Every processor in the IBA is augmented with a new internal register R and a flag F . Every instruction b on the PA is replaced with an instruction b' on the IBA which is identical to b , but instead of writing into the communication register it writes it into R , sets F , and leaves the contents of the communication register unchanged. In addition, a special copy instruction C is introduced which copies the contents of R into the communication register, if the flag F is set, and resets F . All other instructions do not change the value of F . Obviously, the effect of the sequence of instructions b' and C is the same as the effect of b . In all further statements on the equivalence of programs it is assumed that the instructions, their primed versions, and the special instructions C and no-op are always interpreted exactly as defined above.

Now $p^{(t)}$ can be simulated correctly by $n+1$ steps on the IBA (Figure 3.12).

In the i th of the first n steps the "primed" version of the i th row of $p^{(t)}$ is broadcast and only the i th row of the IBA is selected. In the $(n+1)$ st step an n -tuple of C 's is broadcast and all the rows of the IBA are selected.

Hence we get,

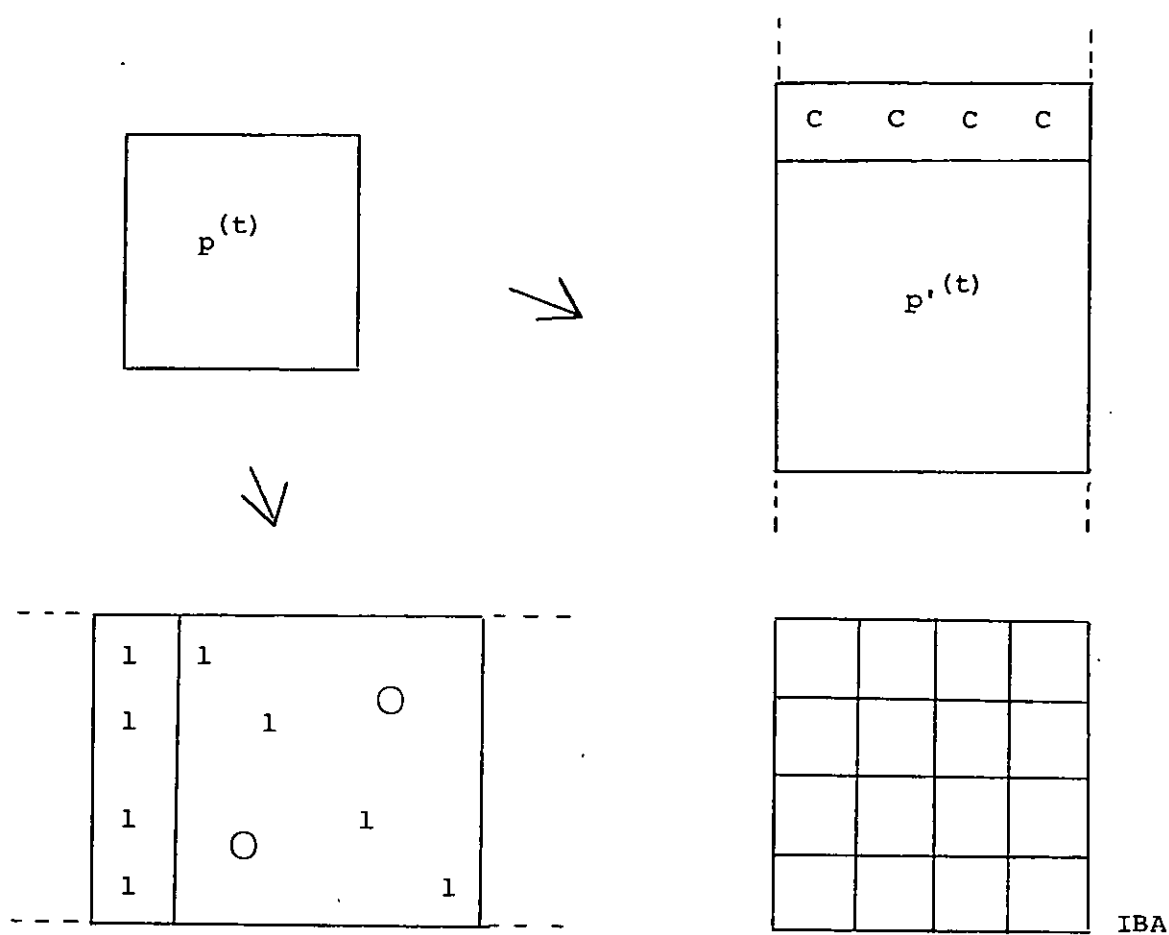


FIGURE 3.12: Program Simulation on IBA

Proposition 2:

For every program p on a PA_n there exists an equivalent program q on an IBA_n such that $T(q) \leq (n+1)T(p)$, for a lower bound on the worst-case delay that can occur, if a program on a PA is simulated by an equivalent program on an IBA.

Proposition 3:

For every $r > 0$ there exists a program p on a PA_n with $T(p) = r$ such that for any equivalent program q on an IBA_n we have $T(q) \geq (n+1)T(p)$.

An outline of the proof can be illustrated by the program p of length 1 which is shown in Figure 3.13.

```

a - - - - - a b
a - - - - - a b a
- - - - - b - - - -
- - - - -
a b a - - - - a
b a - - - - - a

```

FIGURE 3.13: p Program

p consists of n "linearly independent" rows of instructions which cannot be simulated by fewer than n steps on an IBA_n . Assuming that a or b cause a processor to read from its neighbours communication registers, the simulation has to use the primed versions of a and b . Therefore, one additional step is needed to broadcast the copy instruction C .

(c) $IBA \rightarrow ISA$

Let p be a program on an IBA with selector sequence s and $T(p)=r$. Obviously, it is not possible to obtain an equivalent program on an ISA by simply moving the rows of instructions of p and column of selector information of s through the array simultaneously, since this would lead to incorrect combinations of selectors and instructions. The correct combinations could be achieved in a program q on the ISA with selector sequence s' if, for example, for every $i \leq n$ $p_i^{(t)}$ and $s_i^{(t)}$ appeared in the $(t+i-1)$ th row of q and $(t+i-1)$ th column of s' , respectively. This skewed input of p and s would lead to the same problem as in part (b): Instructions that are executed in neighbouring

processors of the IBA simultaneously would be executed at different times on the ISA. Therefore, the same construction as in part (b) has to be used. Every processor is augmented with a new internal register R and flag F , every instruction b on the IBA is replaced with its primed version b' on the ISA, and an additional copy instruction C is introduced. The transformation of p and s into an equivalent program q with selector sequence s' on the ISA can now be done in the following way (Figure 3.14):

p and s are transformed into sequences p' and s'' of length $3r$ by replacing every n -tuple of p with its primed version followed by an n -tuple of C 's and an n -tuple of $no\text{-}ops$, and by inserting after every n -tuple of s two n -tuples of 1 's. q and s' are defined to be the skewed versions of p' and s'' , respectively, (of length $3r+n-1$) followed by $n-1$ n -tuples of $no\text{-}ops$. More formally, for every $t \leq 3r+2n-2$ and $i \leq n$, $q_i^{(t)}$ and $s_i^{(t)}$ are $p_i^{(t-i+1)}$ and $s_i^{(t-i+1)}$, respectively if $1 \leq t-i+1 \leq 3r$ and $no\text{-}op$ otherwise. The final $n-1$ rows of no-operation instructions in q are necessary because it takes that many steps to move the instruction $p_n^{(r)}$ from the first to the last row of the ISA. We thus obtain,

Proposition 4:

For every program p on an IBA_n there exists an equivalent program q on an ISA_n such that:

$$T(q) \leq 3T(p) + 2n - 2$$

Hence, the asymptotic time complexity of p and q is the same, if $T(p)$ is in $\Omega(n)$. It can easily be shown that it is not possible to do much better.

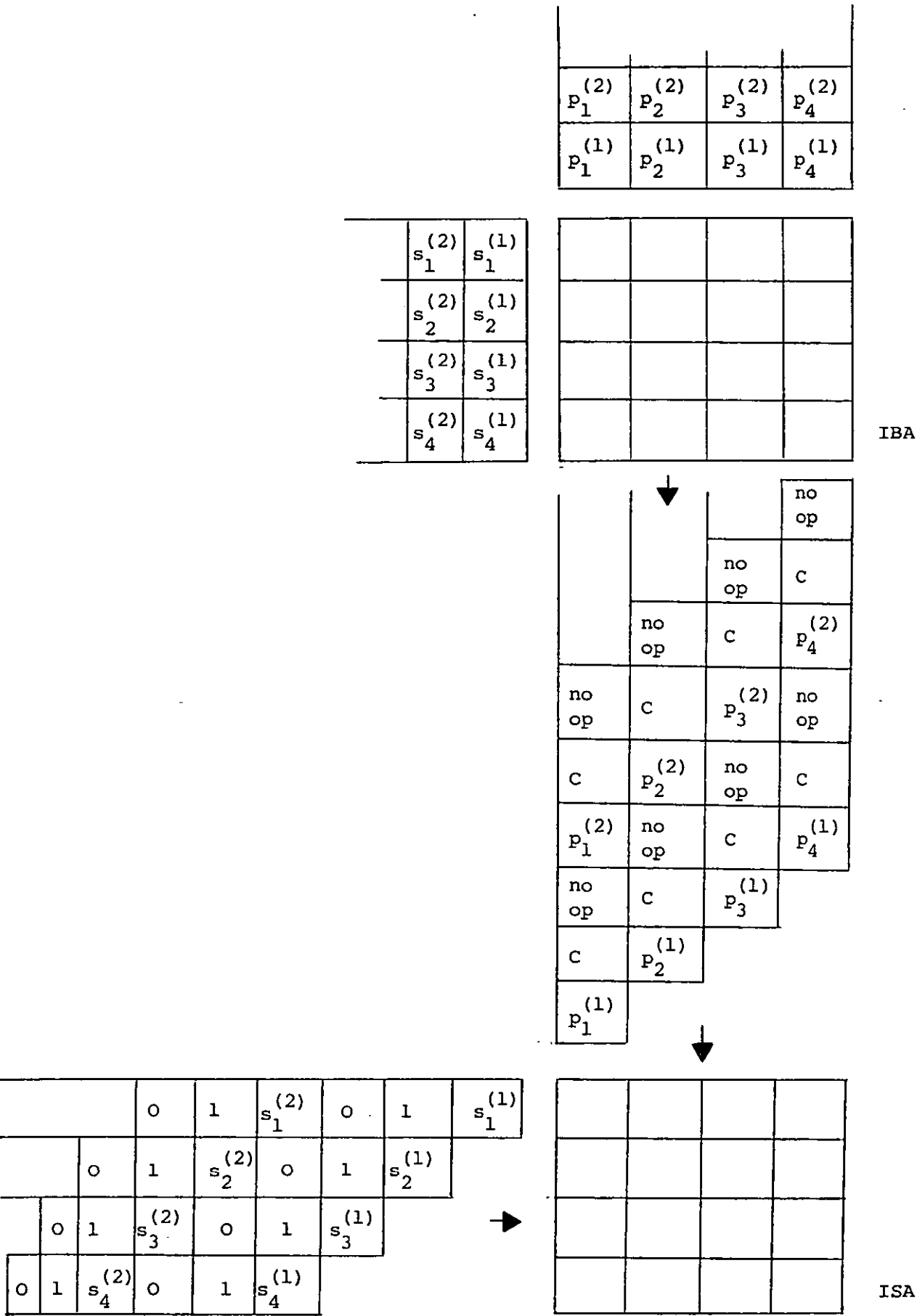


FIGURE 3.14: Transformation Program from IBA to ISA

Proposition 5:

For every $r > 0$ there exists a program p on an IBA_n with $T(p) = r$ such that for any equivalent program q on an ISA_n $T(q)$ is in $\Omega(T(p))$.

(d) $PA \rightarrow ISA$

To simulate a program p on a PA by a program q on an ISA , similar constructions as in (b) and (c) can be used resulting in the following propositions.

Proposition 6:

For every program p on a PA_n there exists an equivalent program q on an ISA_n such that $T(q) \leq (n+2)T(p) + 2n - 2$.

Proposition 7:

For every $r > 0$ there exists a program p on a PA_n with $T(p) = r$ such that $T(q) \geq (n+2)T(p)$ for any equivalent program q on an ISA_n .

(e) $ISA \rightarrow IBA$

The results of parts (a) and (b) immediately provide an upper bound on the worst case effect of simulating a program on an ISA by a program on an IBA . The initial n steps of the execution of a program on an ISA_n can be simulated somewhat faster. In step t ($t \leq n$) there are at most t active rows in the array of processors, thus $t+1$ steps on the IBA suffice to simulate step t . The analog holds for the last n steps. Therefore, to simulate the first and the last n steps of an ISA_n program of length $r \geq 2n$ on the IBA , $n(n+3)$ steps are needed.

Proposition 8:

For every program p on an ISA there exists an equivalent program

q on an IBA_n such that $T(q) \leq (n+1)T(p) - n^2 + n$, if $T(p) \geq 2n$.

The example of a program shown in Figure 3.13 can be used to produce programs of arbitrary length on an ISA such that their simulation on an IBA causes a delay of $\Omega(n)$.

Proposition 9:

For every $r \geq n$ there exists a program p on the ISA_n with $T(p) = r$ such that for any equivalent program q on an IBA_n we have $T(q) = \Omega(n)T(p)$.

3.2.4 Relationship of ISA to Standard Models of Parallel Computers

If we consider the taxonomy of parallel computers as introduced by Flynn [Flynn 1972], all three different types of processor arrays under consideration have to be characterized as MIMD-machines, since several different instructions can be executed simultaneously on different rows and columns hence data streams of the mesh. Obviously the processor array of type PA is closest to the commonly assumed structure of an MIMD-machine. Since the processors in an IBA or an ISA do not have their own control store, the processor arrays of type IBA or ISA are more similar to the array-type SIMD-machines which consist of a mesh-connected $n \times n$ -array of processors receiving their instructions via broadcasting from a central control unit.

Since there exists a large variety of programs based on array-type SIMD-machines [Flynn 1972], [Rodrigue 1982], it is of interest to know how these programs can be simulated on our types of processor arrays. We consider programs for SIMD-machines as special cases of programs for the PA. A full SIMD-program on a PA is a sequence of instructions matrices, each consisting of only identical instructions.

Of course, these programs can be simulated on the IBA and ISA in a much simpler and faster way than ordinary programs.

Proposition 1: [Knude, Lang, Schimmler, Schmeck, Schroder (1985)]

For every full SIMD-program on a PA there is an equivalent program on an IBA having the same time complexity.

The proof is simple: Each program vector $p^{(t)}$ in the IBA program is simply a repetition of the instruction occurring in step t , all selectors are 1. We also refer to the kind of programs on the IBA as full SIMD-programs.

If SIMD-programs on a PA or an IBA have to be simulated on an ISA, we have to deal with the same problem as in the case of arbitrary programs, because instructions executed simultaneously by neighbouring processors of the PA or the IBA will be executed in consecutive steps on the ISA. Therefore we get,

Proposition 2: [Knude, Lang, Schimmler, Schmeck, Schroder (1985)]

For each full SIMD-program p on a PA_n or an IBA_n there is an equivalent program q on the ISA with $T(q) \leq 3T(p) + 2n - 2$.

To capture the situation where instructions of SIMD-machines may be executed by only some of the processors, we define a partial SIMD-program on a PA or an IBA to be a program such that all the instructions executed simultaneously by active processors (i.e. all instructions that are not no-operation instructions O's) are identical.

An example of a partial SIMD-program on a PA is:

a	O	O	O
a	a	O	O
O	a	a	O
O	O	a	a

It is a slightly changed version of the example used in the proof of Proposition 3 in the previous paragraph.

Proposition 3:

For every r there is a partial SIMD-program p on a PA_n with $T(p)=r$ such that for any equivalent program q on an IBA_n we have $T(q) \geq (n+1)T(p)$.

Although we just saw that, in general, partial SIMD-programs on a PA cannot be simulated on an IBA faster than arbitrary programs, there is a large sub-class of partial SIMD-programs which can be simulated as fast as full SIMD-programs: A partial SIMD-program on a PA is termed vector oriented, if in every instruction matrix $p^{(t)}$ the no-operation instruction 0 occurs only in complete rows or columns of no-op's. As an example, the SIMD program for a PA is vector oriented:

0	b	0	b
0	0	0	0
0	b	0	b
0	b	0	b

Proposition 4: [Kunde, Lang, Schimmler, Schmeck, Schroder (1985)]

For every vector-oriented SIMD-program on a PA there is an equivalent partial SIMD-program on an IBA having the same time complexity.

In order to transform a PA program step $p^{(t)}$ with an instruction b occurring in it into an equivalent step on the IBA, we set,

$$p_j^{(t)} = \begin{cases} b, & \text{if column } j \text{ in } p^{(t)} \text{ is not a complete no-op column} \\ \text{no-op}, & \text{otherwise} \end{cases}$$

and put,

$$s_i^{(t)} = \begin{cases} 1, & \text{if row } i \text{ is not a complete no-op row} \\ 0, & \text{otherwise.} \end{cases}$$

3.3 A COMPARISON-BASED INSTRUCTION SYSTOLIC ARRAY

3.3.1 ISA Construction

In Section 3.1 a realisation of an instruction systolic array was described. Now we consider an ISA, the processors of which are capable of executing a small set of one and two operand instructions for comparing and exchanging data items of adjacent processors. Examples of simple programs on the ISA for determining the maximum value or for performing a perfect shuffle are given.

In order to construct an instruction systolic array (ISA) capable of executing algorithms using comparison and exchange operations, the following set of instructions have been developed by Hartmut Schmeck [Schmeck 1986]. In the following K' denotes the communication register of one of the adjacent processors in the north, east, south, or west. In the case of a processor on the boundary of the array, K' may denote an I/O-pad.

$\text{read}(K')$: K gets the value of K'

pictorial representation



$\text{min}(K, K')$: K gets the minimum of its own value and that of K' .

pictorial representation



$\text{max}(K, K')$: K gets the maximum of its own value and that of K' .

pictorial representation



neg : K is negated, i.e. it gets the one-complement of its own value.

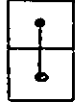
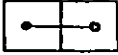
pictorial representation.



no-op: The value of K is not changed, the register is only refreshed.

pictorial representation



Appropriate combinations of read instructions( , ) can

be used to execute an exchange operation between adjacent processors.

In the same way, a combination of min and max instructions can be used to execute a comparison-exchange. Although the effect of a comparison-exchange will always be the same while it is being moved through the array, a preceding use of the neg instruction in some rows of the array can lead to comparison-exchanges in the opposite direction. This is desirable, if e.g. the rows of the array are to be sorted in alternating order (ascending and descending).

All the instructions are executed bit-serially. Otherwise the number of wires and I/O pads and therefore the area of the ISA would be too large. Because of the bit-serial mode of operation, the communication registers are implemented as shift-registers.

The block structure of an ISA-processor realizing the 14 different instructions is shown in (Figure 3.15). The functional unit essentially consists of a bit-level comparison-exchange unit. Depending on the control signals received, it provides K , K' , $-K$, $\min(K, K')$ or $\max(K, K')$ as output. To allow for a very simple decoding the instructions are encoded using 5 instead of 4 bits. The input of instructions into the array is done bit-serially, to save I/O pads. However, they are passed from row to row in a bit-parallel way. The execution of an instruction takes $k+1$ clock cycles where K is the

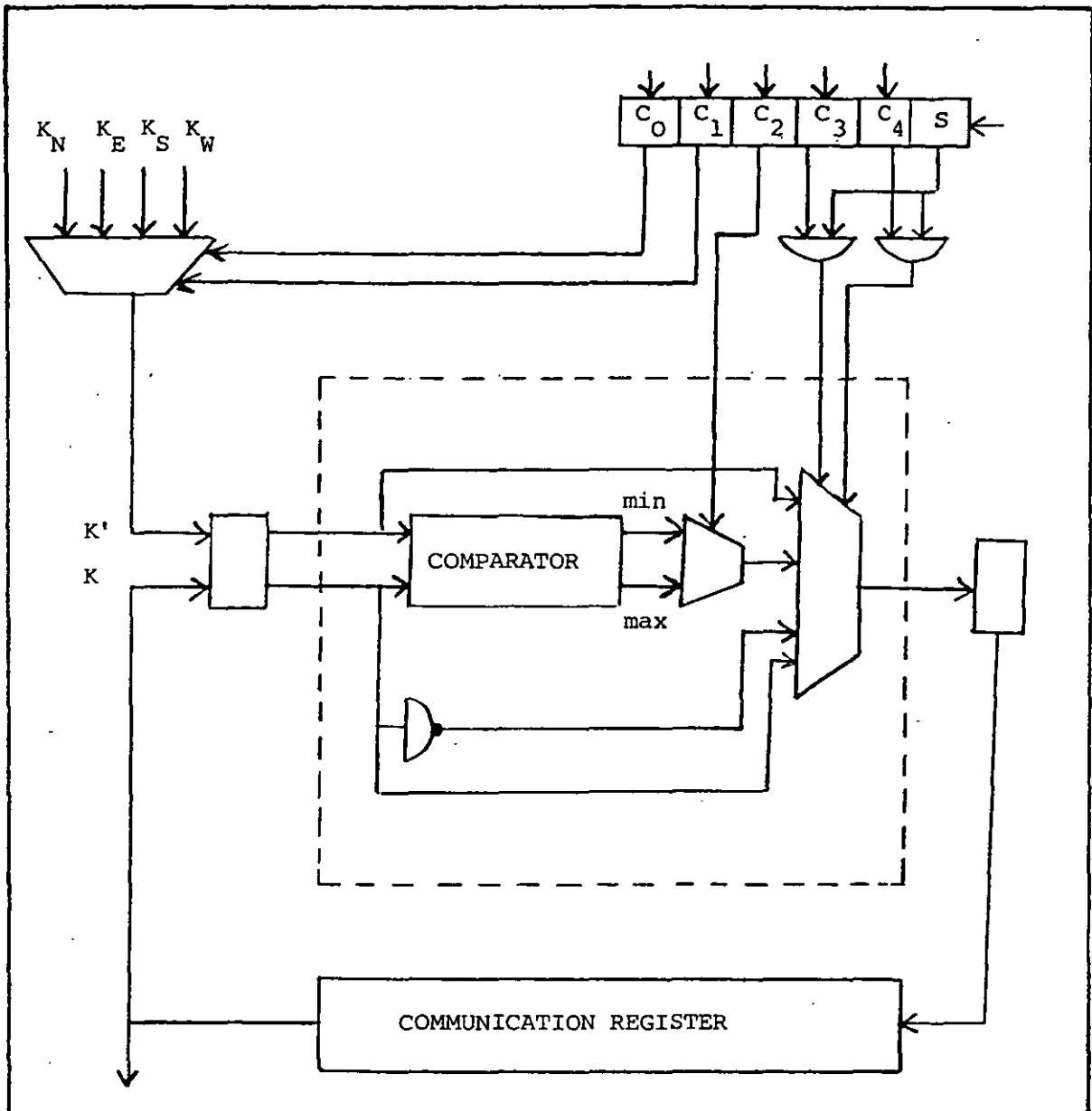


FIGURE 3.15: Block Structure of a Comparison Based ISA-Processor

length of the data items. If instructions were passed bit-serially, either the execution time would be longer or an extra register for sorting the instruction would be necessary.

3.3.2 Example Programs on the ISA

(i) Input and Output of Data Items

To initialize the communication registers of the Instruction Systolic Array, the program depicted in Figure 3.16_a may be used which moves n^2 data items from the left into the array. The execution of this program could be overlapped with the execution of the program of Figure 3.16_b which moves the contents of the communication registers to the right and out of the array. Since the instructions of both programs have to be executed in every processor the selector sequence consists of ones only.

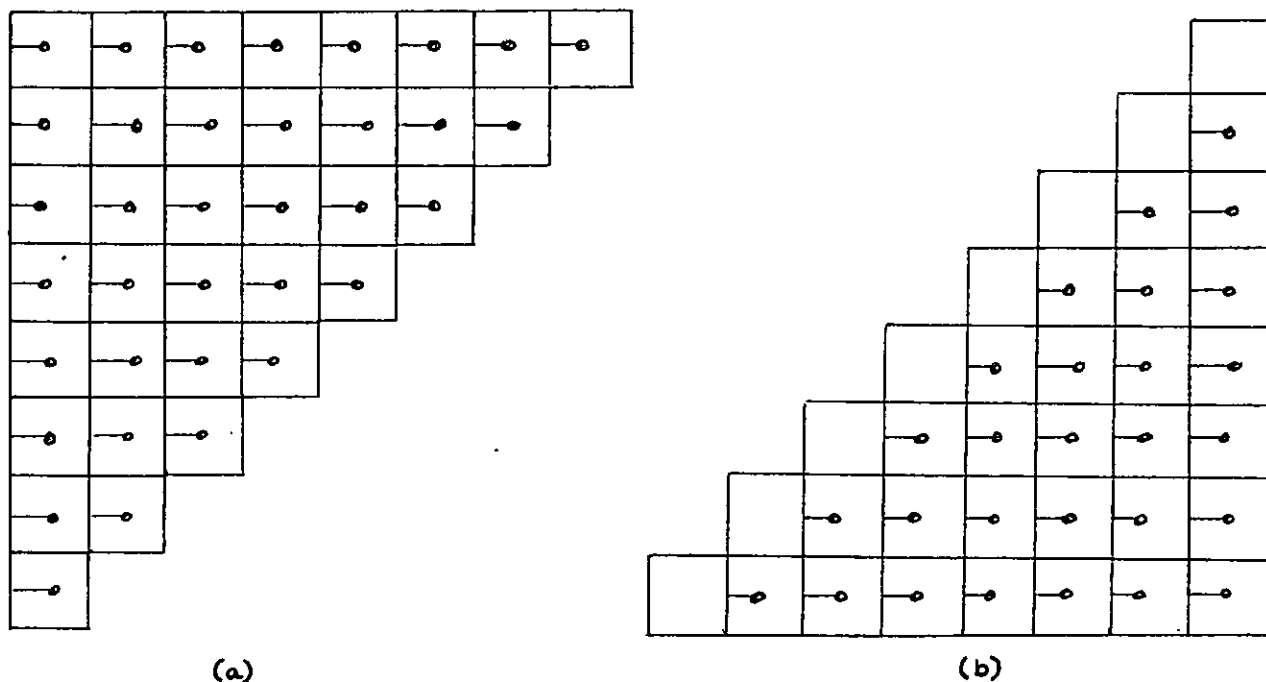


FIGURE 3.16: (a) Input into and (b) Output from an 8x8-ISA

(ii) To Determine the Maximum

One way of determining the maximum of the n^2 items stored in the ISA is to move the maximal data item to a fixed location, e.g. to the processor at the lower right corner. In the worst case this requires at least $2n-2$ local exchanges. Two simple ISA-programs for achieving this are given in Figure 3.17a and 3.17b. Both programs have only two diagonals of instructions. After execution of the program shown in Figure 3.17b, we do not only have the maximum of all the data items in the lower right corner, but for every $i \leq n$ we have the maximum of the first i rows (columns) in the i -th processor of the rightmost column (bottom row).

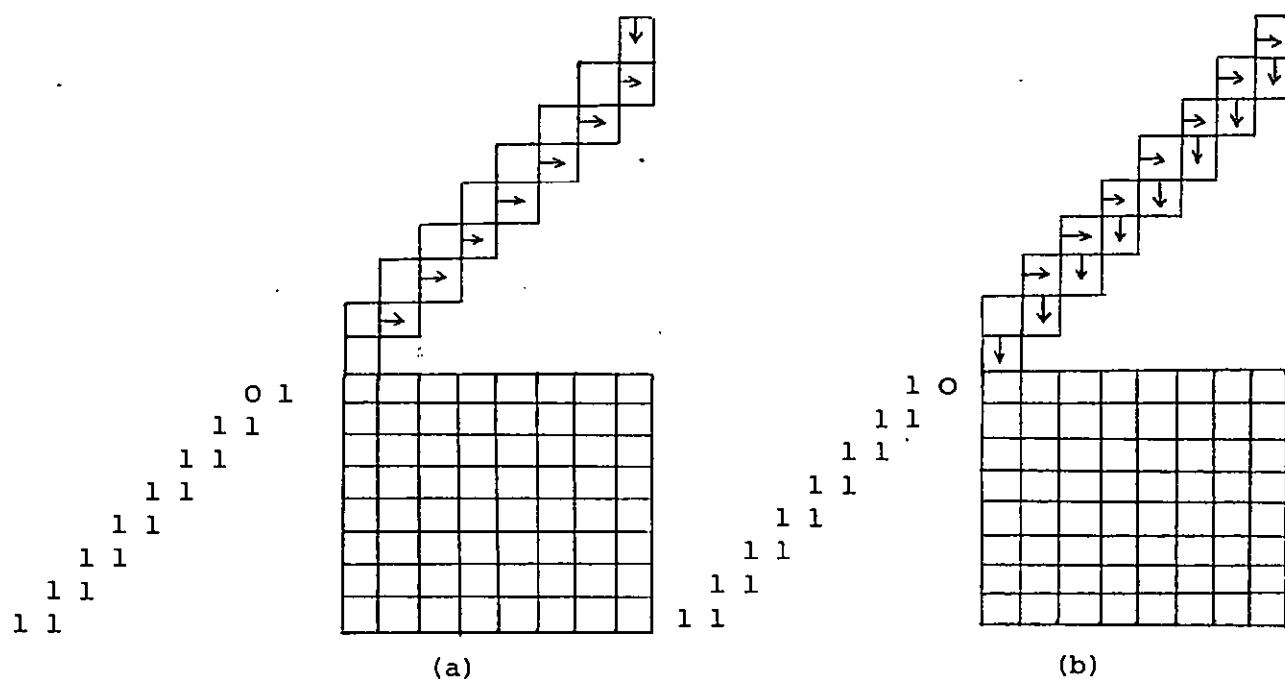


FIGURE 3.17: Programs for Determining the Maximum on an 8x8-ISA

If it were not acceptable to change the set of data items stored in the array, one could extend the programs by two diagonals using comparison-exchanges instead of the simple max instructions. Obviously

the minimum may be determined completely analogously, unless it has to be moved to the upper left corner of the array. It is easy to notice that it is much harder to move a data item from the right to the left or from the bottom to the top than in the opposite direction. A program for moving the bottom row of data items to the top row is shown in Figure 3.18. It has $2n-n$ diagonals of instructions and selectors. The corresponding program for moving the top row to the bottom would consist of two diagonals only.

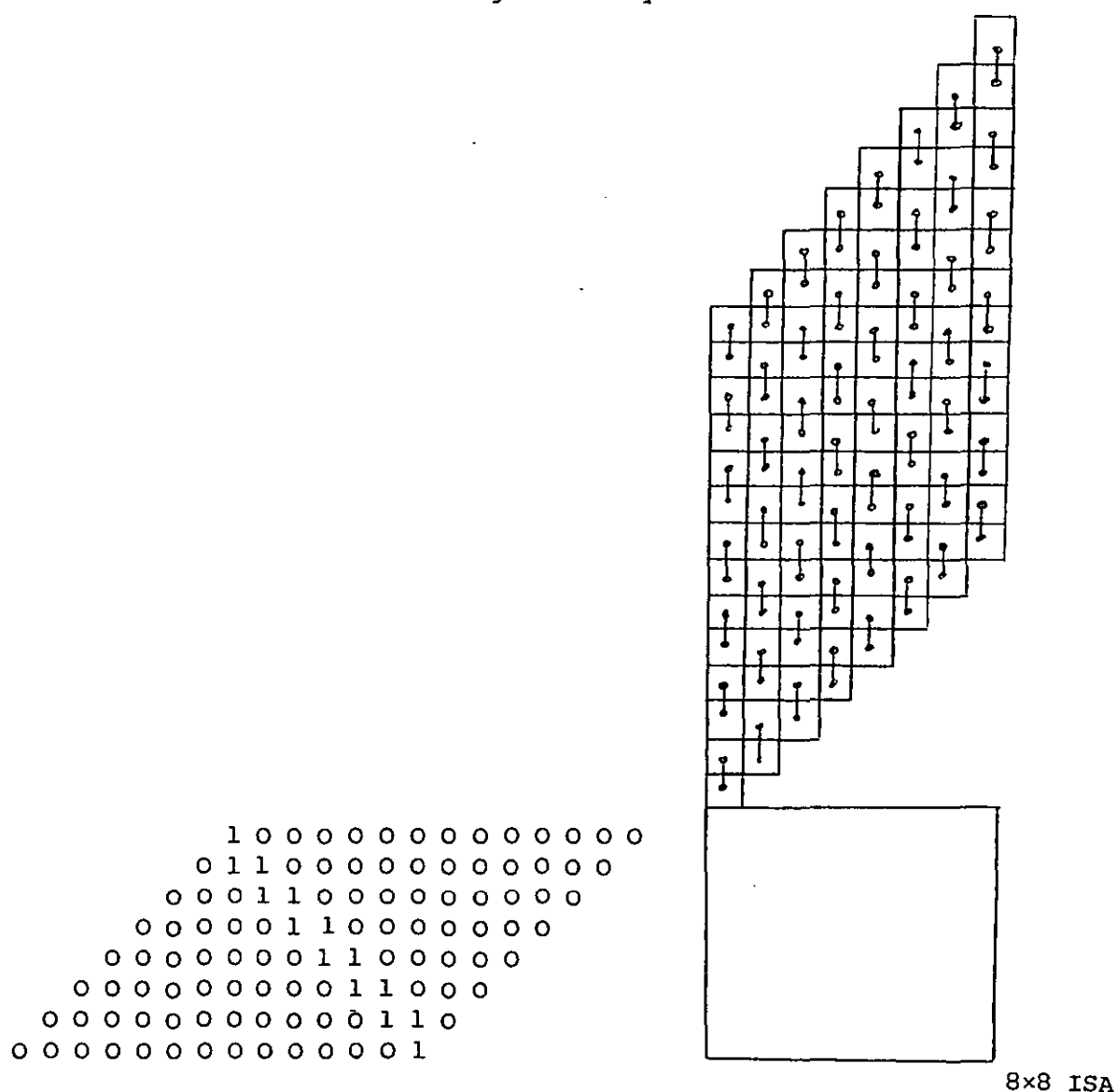


FIGURE 3.18: Program for Moving the Bottom Row to the Top Row of an 8x8-ISA

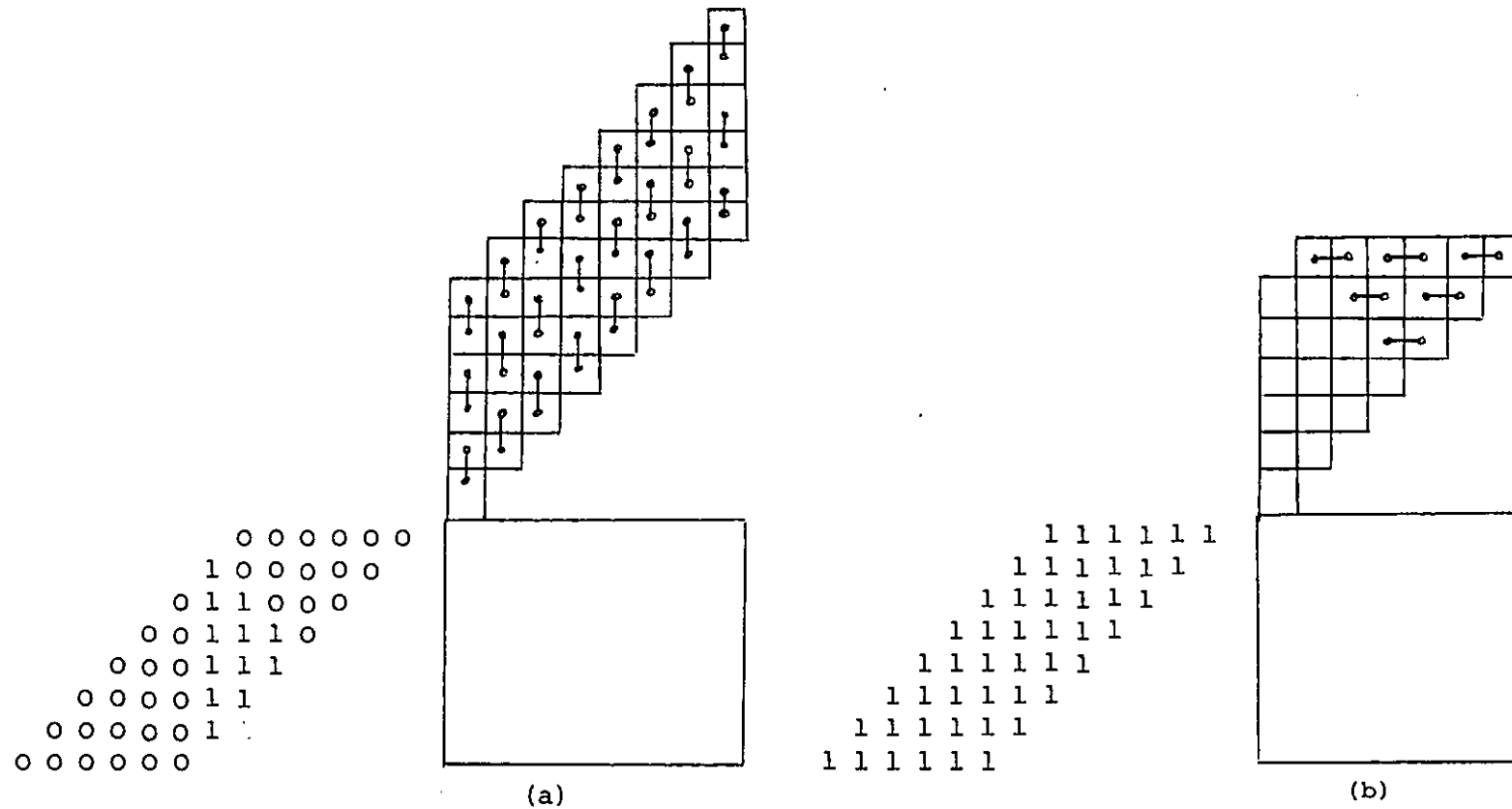


FIGURE 3.19: Perfect Shuffle on (a) the Rows and (b) the Columns of an 8x8-ISA.

(iii) To Perform the Perfect Shuffle

The perfect shuffle is an operation that is used in many different algorithms [Stone 1971]. It transforms the sequence $a_1, \dots, a_k, b_1, \dots, b_k$ into a sequence $a_1, b_1, \dots, a_k, b_k$. Programs for performing the perfect shuffle on all the rows or all the columns of the ISA are depicted in Figure 3.19. Both programs have $n-2$ diagonals of instructions. The program for the perfect shuffle on the rows is somewhat simpler, since only $n/2-1$ of its rows contains instructions that are different from no-op.

(iv) Sort Program

Figure 3.20 shows two simple-ISA programs for sorting a 2×2 -array.

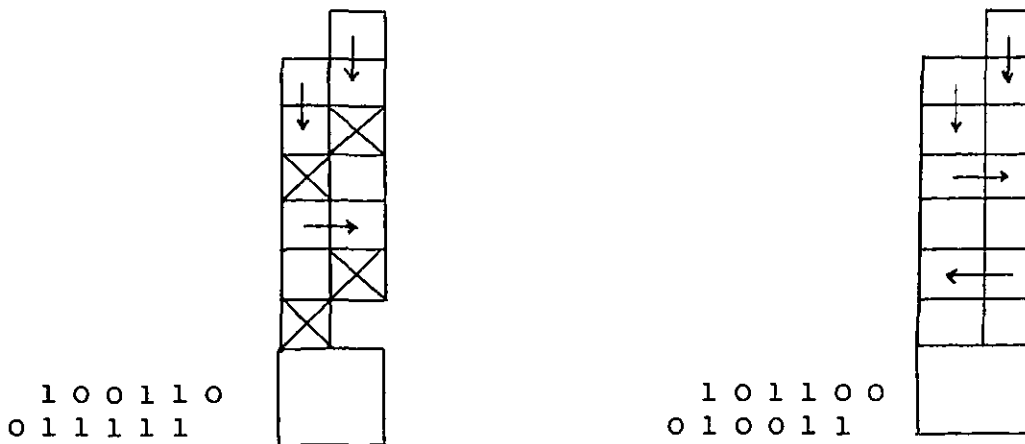


FIGURE 3.20: Two Simple ISA-Programs for Sorting a 2×2 -Array

3.4 TO SORT BY THE INSTRUCTION SYSTOLIC ARRAY

3.4.1 Introduction

In this section an algorithm for sorting n^2 items on a $n \times n$ mesh-connected processor array (Figure 3.3) in time $O(n)$ is presented. It is very simple in its structure and can be implemented easily on a systolic array. In particular, it can be written as a simple program for the ISA.

There is a lot of previous work done in closely related fields. In [Thompson, Kung 1977], [Nassimi, Shani 1979], sorting algorithms with time complexity $O(n)$ are presented. This time performance is asymptotically optimal for mesh-connected architectures. Another algorithm with the asymptotic time complexity can be found in [Lang, Schimmler, Schmeck, Schroder 1983] together with a systolic array to sort n^2 items in $8n$ steps. (The constants are very interesting in this context since there is no difference in the asymptotic behaviour).

Naturally, the power of the ISA depends on the power of the individual processors. The restriction to a rather simple instruction set seems desirable to keep the processors small enough, in order to allow integration of many processors on a single chip.

Formally, the operation executed by processor (i,j) at time t is defined in the following way:

For every $i, j \leq n$ and $t \leq r \cdot p^{(t)}$ is the row of instructions which enters the i -th row of the ISA at time $t+i-1$, and $s^{(t)}$ is the column of selector information which enters the j -th column of the ISA at time $t+j-1$. This means, that the instruction executed by processor (i,j) at time t is:

$$p(i,j) = \begin{cases} p_j^{(t+i-1)} & \text{iff } s_i^{(t-j+1)} = 1 \\ \text{no-op} & \text{otherwise} \end{cases}$$

Hence we assume each processor to have only one register (communication register) and to be able to execute the following instructions chosen by Manfred Schimmler [Schimmler 1986]:

SYMBOL

MEANING


 $C := \max(C_{\text{left}}, C)$

(Read the contents of the left neighbour's register, compare it with its own register, and store the maximum in its own register).


 $C := \max(C_{\text{right}}, C)$

 $C := \max(C_{\text{upper}}, C)$

 $C := \min(C_{\text{left}}, C)$

 $C := \min(C_{\text{right}}, C)$

 $C := \min(C_{\text{lower}}, C)$

 $C := \bar{C}$

(Invert all bits of the contents of C)



NO-OP (no-operation)

3.4.2 One Dimensional Sorting Methods

Here we introduce two parallel sorting methods for one dimensional sequences which we use as part of the algorithm (for two dimensional arrays). For a better understanding we represent them as sorting nets

in the manner shown by [Knuth 1973]. For example, the odd-even-transposition sort for $n=6$ items is illustrated in (Figure 3.21). Comparator models are represented by vertical arrows between two lines. The number enters at the left and each comparator causes an interchange of its inputs, if necessary, so that the larger number appears on the line of the arrow head after passing the comparator. At the right of the diagram all numbers are in order from top to bottom.

(i) The K-Triangle Merger:

The K-triangle merge net (Figure 3.22) consists of $\frac{K}{2}$ parallel comparison steps, where the i -th comparison step consists of the comparisons $[i+j-1 : i+j]$, $j=0,2,4,\dots,k-2i$ ($[l:m]$ denotes a comparison-exchange between the elements on line l and line m).

It turns out to be a sorting net if the input consists of two sorted sequences (of length $\frac{K}{2}$) that have been concatenated and perfectly shuffled afterwards:

Lemma 1:

Given an even number K and a sequence a_0, a_1, \dots, a_{k-1} of items to be sorted, where a_0, a_2, \dots, a_{k-2} and a_1, a_3, \dots, a_{k-1} are already sorted in non-decreasing order, the K-triangle merger sorts the whole sequence into non-decreasing order.

Proof:

Using the 0-1-principle for sorting nets [Knuth 1973] we can assume the input elements a_i to be 0's or 1's only. We will prove the Lemma by induction on K : $K=2$: A 2-triangle merger consists of exactly one comparator between line 0 and line 1, so obviously every sequence of length two will be sorted by a 2-triangle merger. $K>2$: Assume

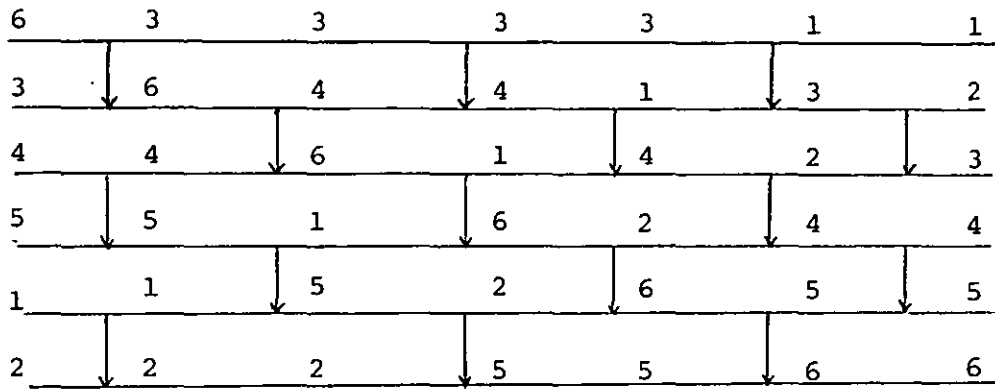


FIGURE 3.21: Odd-Even-Transposition Sorting Net for $n=6$ Items

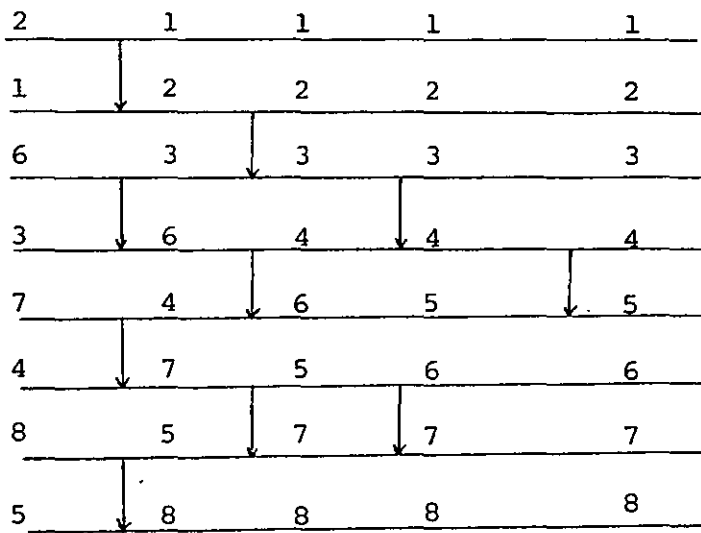


FIGURE 3.22: The 8-Triangle Merger

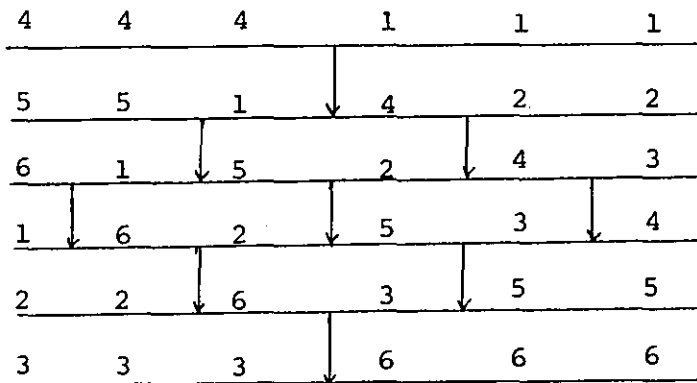


FIGURE 3.23: The 6-Diamond Merger

Lemma 1 to be true for the K-triangle merger. We consider the sequence a_0, a_1, \dots, a_{k-1} as input into the K-triangle merger. We have to distinguish four cases:

$$(1) \quad a_0=0, a_1=0$$

The sequence a_2, a_3, \dots, a_{k-1} is sorted by the K-2-triangle merger obtained by removing the first comparator of each step of the k-triangle merger. Thus, the whole sequence a_0, a_1, \dots, a_{k-1} is sorted by the K-triangle merger, since the inputs into the additional comparators are already sorted.

$$(2) \quad a_0=0, a_1=1$$

$a_1=1$ implies $a_{2i+1}=1$ for $i=0, 1, \dots, \frac{k}{2}-1$. Thus the inputs into all the comparators of the first step of the K-triangle merger are already sorted. The remaining comparators form exactly a K-2 triangle merger that sorts the sequence a_1, \dots, a_{k-2} . Since $a_0=0$ and $a_{k-1}=1$ the whole sequence is sorted after passing the k-triangle merger.

$$(3) \quad a_0=1, a_1=0$$

$a_0=1$ implies $a_{2i}=1$ for $i=0, 1, \dots, \frac{k}{2}-1$. Thus, initially we have $a_{2i} \geq a_{2i+1}$ for every i . After passing the first step of the K-triangle merger the sequence is transformed to $a_1, a_0, a_3, a_2, \dots, a_{k-1}, a_{k-2}$. As in Case 2 the whole sequence is sorted by the remaining K-2-triangular merger.

$$(4) \quad a_0=1, a_1=1$$

This implies that the whole sequence consists of ones only and it is obviously sorted before and after passing the merger.

(ii) The K-Diamond Merger

The K-diamond merge net (Figure 3.23) consists of $k-1$ parallel comparison steps, where for $i=1, 2, \dots, \frac{k}{2}$ the i th comparison step consists of the comparators $[\frac{k}{2} - i + 2j : \frac{k}{2} - i + 2j + 1]$, $j=0, 1, \dots, i-1$, and for $i=\frac{k}{2} + 1, \frac{k}{2} + 2, \dots, k-1$ it consists of the comparators $[i - \frac{k}{2} + 2j : i - \frac{k}{2} + 2j + 1]$, $j=0, 1, \dots, k-i-1$.

Lemma 2:

Given an even number K and a sequence a_0, a_1, \dots, a_{k-1} of items to be sorted, where $a_0, a_1, \dots, a_{k/2-1}$ and $a_{k/2}, a_{k/2+1}, \dots, a_{k-1}$ are each sorted in non-decreasing order, the K -dimensional merger sorts the whole sequence into non-decreasing order.

Proof:

The 0-1-principle again allows us to restrict the inputs to 0's and 1's only. We prove Lemma 2 by induction on K : $K=2$: A 2-diamond merger consists of one comparator that obviously sorts any sequence of length two. $K>2$: Assume Lemma 1 to be true for the $K-2$ -diamond merger. Let a_0, a_1, \dots, a_{k-1} be the input sequence. The K -diamond merger consists of a $K-2$ -diamond merger followed by the comparators $[j-1 : j]$ and $[k-j-1 : k-j]$, $j=1, 2, \dots, \frac{k}{2}$. The $K-2$ -diamond merger sorts the sequence a_1, a_2, \dots, a_{k-2} by the induction hypothesis, leaving only a_0 and a_{k-1} to be inserted. If a_0 is 0, it is already in its final position. If a_0 is 1 its final position is somewhere between line 0 and line $\frac{k}{2}$, since $a_0=1$ implies $a_i=1$ for $i=0, 1, \dots, \frac{k}{2}-1$. Thus the remaining comparator diagonal $[j-1 : j]$, $j=1, 2, \dots, \frac{k}{2}$, will insert a_0 into its correct position. The dual argument shows that a_{k-1} finds its

position by the comparator diagonal $[k-j-1 : k-j]$, $j=1, \dots, 2, \dots, \frac{k}{2}$.

3.4.3 The Two Dimensional Sorting Algorithm

We consider a simple model of an MIMD computer: a mesh-connected $n \times n$ array of identical processors (Figure 3.3). Every processor has some local memory including a designated communication register. It can execute a small number of instructions and is capable of generating its own instruction sequence. The processor array is synchronized by a global clock, and the execution of every instruction is assumed to take the same time. During the sorting process every processor contains one data item in its communication register. Observe that there are situations, where two elements initially loaded at the opposite corner processors have to be transposed during the sorting. It is easy to argue that even for this simple transposition at least $2n-n$ local interchange steps are needed. This implies that no algorithm on such a mesh-connected processor array can sort n^2 elements in less than $O(n)$ steps.

The following algorithm is to be executed on the processor array:

Algorithm Merge:

Input: Four $\frac{n}{2} \times \frac{n}{2}$ arrays, the upper sorted in left to right row major order, the lower ones in right to left row major order.

Output: One $n \times n$ array sorted in row major order.

1. Merge all columns of the $n \times n$ array by the $n-1$ steps of an n -diamond merger.
2. Sort all rows by n -steps of the odd-even-transposition sort, the odd rows in left-to-right direction, the even rows in right-to-left direction.

3. Merge all columns by the $\frac{n}{2}$ steps of an n -triangle merger.
4. Sort the rows by n -steps of the odd-even-transposition sort.

The validity of the algorithm may be seen again by the use of the 0-1-principle: the initial configuration is that of (Figure 3.24a), four sorted $\frac{n}{2} \times \frac{n}{2}$ subarrays, where the white regions denote 0's and black regions 1's. After step 1 of the algorithm the array consists of two $n \times \frac{n}{2}$ arrays and in both halves there is only one row still unsorted (Figure 3.24b). This is called the critical row. Step 2 sorts all rows in alternating directions and can so produce two different situations:-

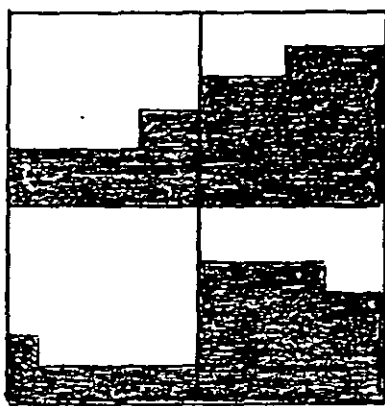
- a. the critical rows are still in different halves (Figure 3.24c)
- or b. the critical rows are in the same half (e.g. the left as in Figure 3.24d). In both cases step 3 results in a situation where there is only one single unsorted row in the whole $n \times n$ array (Figure 3.24e). Obviously step 4 suffices to complete the sort (Figure 3.24f).

For the complexity analysis we assume one elementary comparison-exchange step to require the time t_c . Step 1 needs $(n-1)t_c$. For Step 2 we need nt_c and Step 3 requires $\frac{n}{2}t_c$. Step 4 again needs nt_c and so the time $T_M(c)$ for the merge stage of four $\frac{n}{2} \times \frac{n}{2}$ arrays is:

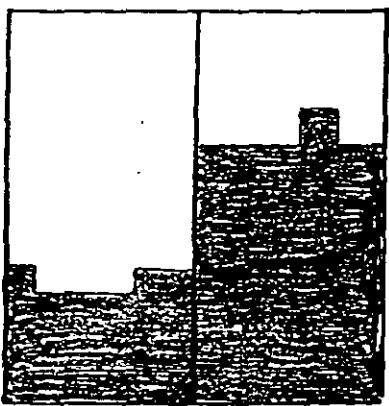
$$T_M(c) = (3\frac{1}{2}n-1)t_c.$$

3.4.4 The Algorithm on the ISA

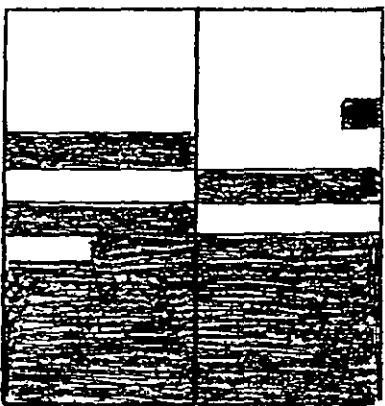
In this paragraph we give a program for the ISA that realizes the algorithm MERGE. We assume that the items to be sorted are loaded in the $n \times n$ array before starting the sorting process, each one in the



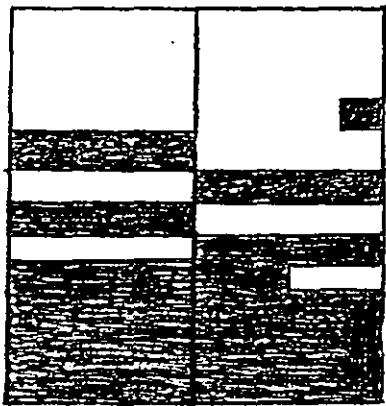
(a)



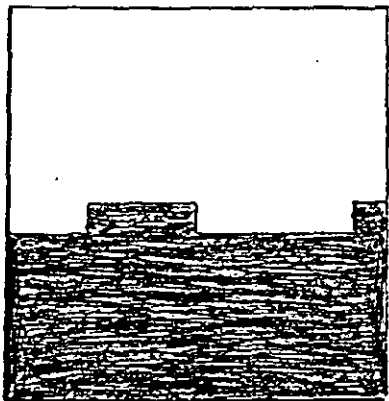
(b)



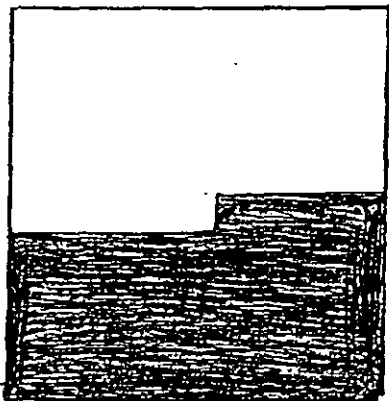
(c)



(d)



(e)



(f)

FIGURE 3.24: Sorting an Array of 0's and 1's;
White Regions are 0's, Black Regions are 1's.

communication register of one processor. We give the program as an instruction stripe and a selector stripe, where the i -th row of the instruction stripe is the i -th instruction n -tuple (counted from the bottom to the top) and the i -th column of the selector stripe is the i -th selector n -tuple (counted from the right to the left). The meaning of the instruction symbols has been explained in paragraph 3.4.1. As an example (Figure 3.25) shows the program for a 6-triangle merger for the columns of a 6×6 ISA. The complete program of the last merge stage of an 8×8 sorter is depicted in (Figure 3.26).

The length of the sorting program is the number of instruction n -tuples of the ISA program. The program is composed of diagonals of either identical instructions or no-ops and instructions of the same type. So it is easy to count the number of instruction diagonals (ND) for the logical blocks of the algorithm and sum these up to compute the length of the whole sorting program [Lang 1987]. The merging of two half-columns of an $K \times K$ array with K -diamond merger requires K instruction diagonals. One diagonal is used to invert every bit of the elements of the even rows of the array. Now we can sort all rows (in $K-2$ diagonals) of the array and again invert the elements of every even row. The effect is that the even rows are sorted in right-to-left order by this. Sorting the columns with a K -triangle merger again needs K diagonals and sorting the rows afterwards needs $2+K-2$ diagonals. Since the sorting direction on this last step alternates (according to the position of the $K \times K$ subarray in the whole $n \times n$ array) we need two more diagonals of invert operations, one before and after the final sorting of the rows.

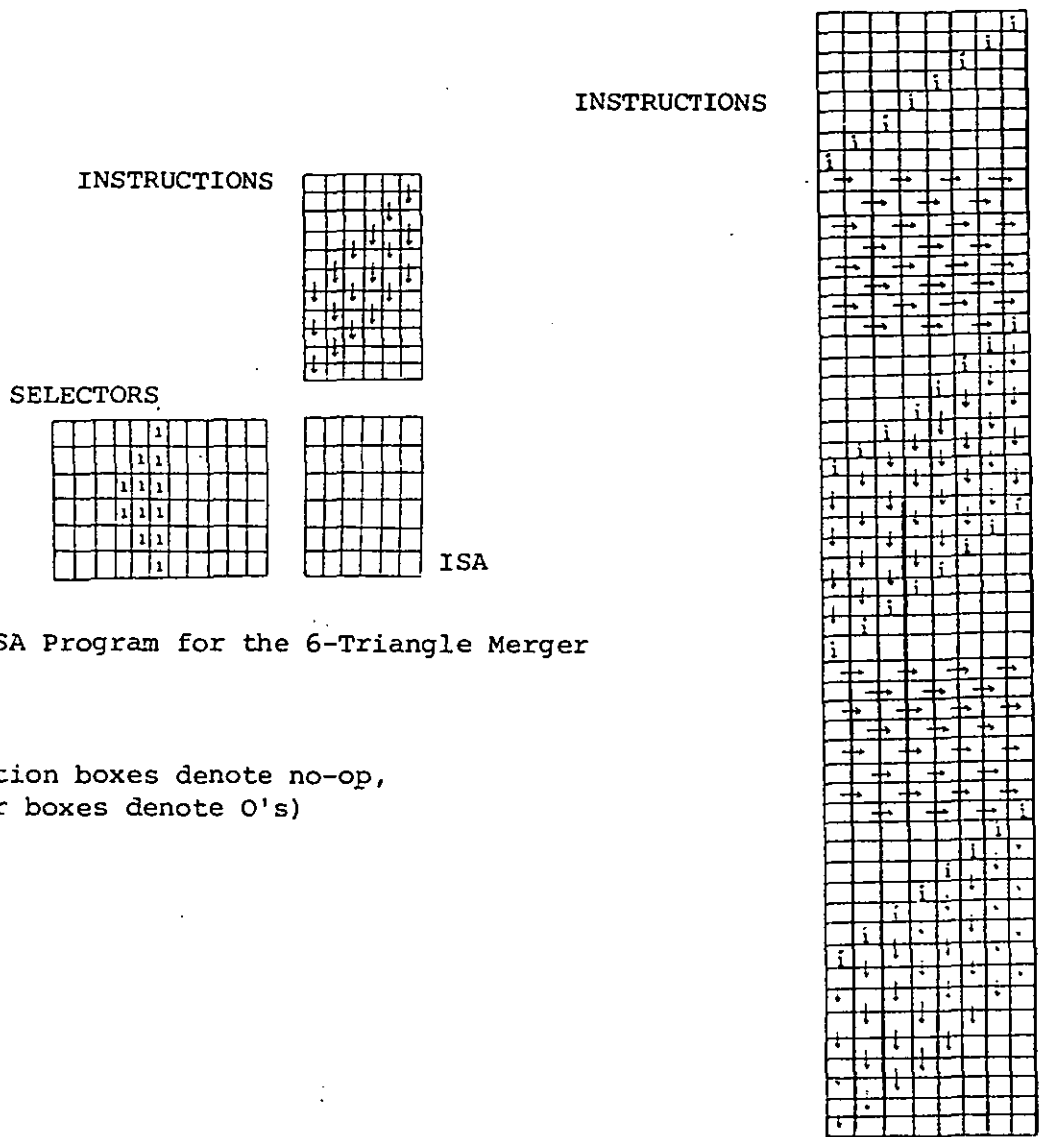


FIGURE 3.25: ISA Program for the 6-Triangle Merger

(Empty instruction boxes denote no-op,
empty selector boxes denote 0's)

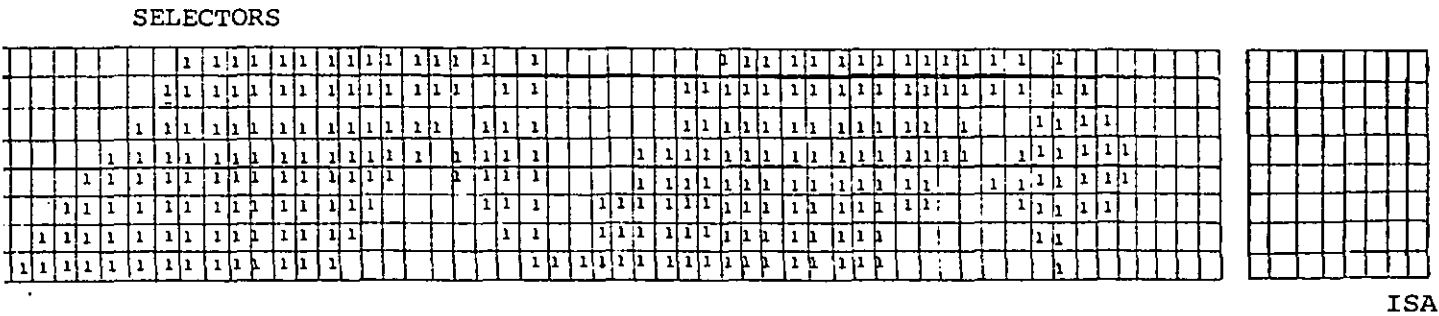


FIGURE 3.26: ISA Program for the Last Merge Stage of 8x8 Sorter

If we sum up the number of instruction diagonals for the $(\log K)$ th merge step of the sorting program on the ISA we get:

$$ND_M(K) = 6 * K$$

and for the whole sorting algorithm

$$ND_S(K) = ND_S\left(\frac{K}{2}\right) + ND_M(K)$$

Since $ND_S(1)=0$, we get,

$$ND_S(n) = 12 * n - 12 .$$

To complete the ISA program we have to add a number of no-ops instructions to fill up the empty spaces under the first and on top of the last instruction diagonals, and we must add n instruction rows containing no-ops only at the end of the program to ensure that the last significant instruction is executed by the processor (n,n) when the program is terminated. So the length of the sorting program on an $n \times n$ ISA is:

$$L_S(n) = ND_S(n) + 2n - 1$$

$$L_S(n) = 14 * n - 13 .$$

3.5 ADDITIONAL ALGORITHMS SOLUTION BY USING THE INSTRUCTION SYSTOLIC ARRAY (ISA)

3.5.1 Finding the Generalized Matrix Inversion

Matrix inversion is a very important operation in scientific computation and for this reason there is a large and growing body of literature concerned with the design of parallel processor networks for this purpose. H. Schroder and E.V. Krishnamurthy [Schroder, Krishnamurthy 1988] used two different implementations for Greville's algorithm [Gregory, Krishnamurthy 1984] on the new concept of parallel computation is the Instruction Systolic Array. A matrix of instruction codes (called left program (LP)) is pumped through the array of processors in a systolic fashion, in addition a boolean matrix TP (top program) is also pumped through the array of processors. If a (zero) 0 meets an instruction in a processor then it prevents the instruction to be executed, while a (one) 1 enables its execution. Thus, an ISA-program consists of a pair (LP,TP).

The first implementation of the g-inversion presented uses an $n \times (m+1)$ matrix of mesh connected processors P_{ij} with $i=1$ to n , $j=0$ to m ; and an extra processor to carry out the division operations.

Let A be an $m \times n$ matrix, A_n^+ is generated iteratively in n cycles (counting the basis as one cycle). A_i^+ is generated in cycle i . The ISA-program represents the systolic execution of cycle i . In the execution of cycle i only the upper i rows of the ISA grid are engaged. The second implementation uses for the execution of cycle i only a linear array of p_1, \dots, p_i (column for each step), and a device to execute the division. The matrices A_i and A_i^+ can either be stored outside the chip or in cyclic shift registers on the chip and be

supplied to the processors on demand.

The two different implementations of Greville's algorithm mentioned above permit a free choice of algorithmic mode, also they require a smaller set of instructions and a smaller capacity local memory for the processors, thereby facilitating massive parallelism in a smaller area of the VLSI chip.

3.5.2 Top-Down Designs of Instruction Systolic Arrays for Polynomial Interpolation and Evaluation

In [McKeown 1986] a parallel version of Aitken's method of iterated linear interpolation is presented. Its execution time is $2n$ and its period is n using n processing elements. An elementary step in their implementation consists of two additions (subtractions), two multiplications and one division (which has to be executed after the multiplications are finished).

In [Schroder 1988] a generalized version of a design method for systolic arrays due to S.Y. Kung [Kung 1987] was presented. As in [Kung 1987] Schroder starts with a locally recursive algorithm and generates the Dependence Graph. A geometric layout of the dependence graph is then projected onto an array of processors. Now in the design process of systolic arrays this projection can only be done along an axis of shift-invariance, designing instruction systolic arrays allows projections of the dependence graph in different directions, which allows optimisation of the implementation under other aspects than the execution time. This method results in a design which is optimal in execution time and period. This can easily be

seen using standard network planning techniques. The parallel interpolation algorithm presented has the advantage over the algorithm in [McKeown 1986] that a single step consists of either two multiplications or one division only. Another advantage of the implementations in this method of using the concept of instruction systolic arrays leads to a significant flexibility. Here the evaluation program can be started right after the interpolation program.

The Dependence Graph for a set of recursive equations contains the information which evaluations of variables (tasks) have to precede with. Its geometrical layout is not unique and has a major impact on the quality of the resulting implementation.

There are different techniques to generate valid and optimal schedules from dependence graphs [Moldovan 1983], [Mongenot, Perrin 1987]. Mongenet and Perrin show how transformations in the 'time-space' can be used to ensure that coefficients are not used before they are produced. Those techniques would have to be modified to ensure that the coefficients are read before they are overwritten.

The next step in the top-down design is projecting the geometric layout of the dependence graphs on to a 1-dimensional array of processing elements. Designing systolic arrays would have to project along directions of Shift Invariance [Kung 1987] (i.e. all tasks following this specific direction are the same). Since each processing element is capable of executing one operation only.

Designing Instruction Systolic Array programs have several choices for the projection direction. The implementations are based on vertical projection which in the case of the interpolation algorithm is not along the direction of Shift Invariance (i.e. along the diagonal

from the north-west to the south-east). Vertical projection for the evaluation algorithm has been used here in order to achieve a short period. Horizontal projection of the evaluation algorithm would enable the evaluation with just one or two processors, with a period proportional to n .

The hardware requirement here is a $n \times n$ connected processor array, each processor having nine registers. The ISA program terminates when the last instruction row has entered the first processor row. A $1 \times n$ instruction systolic array is used, selectors are not needed and are thus omitted. Let a polynomial of degree 3 be given by x_0, r_0 and x_1, r_1 and x_2, r_2 and x_3, r_3 . Figures 3,27a,b represent an ISA-program for interpolation and evaluation.

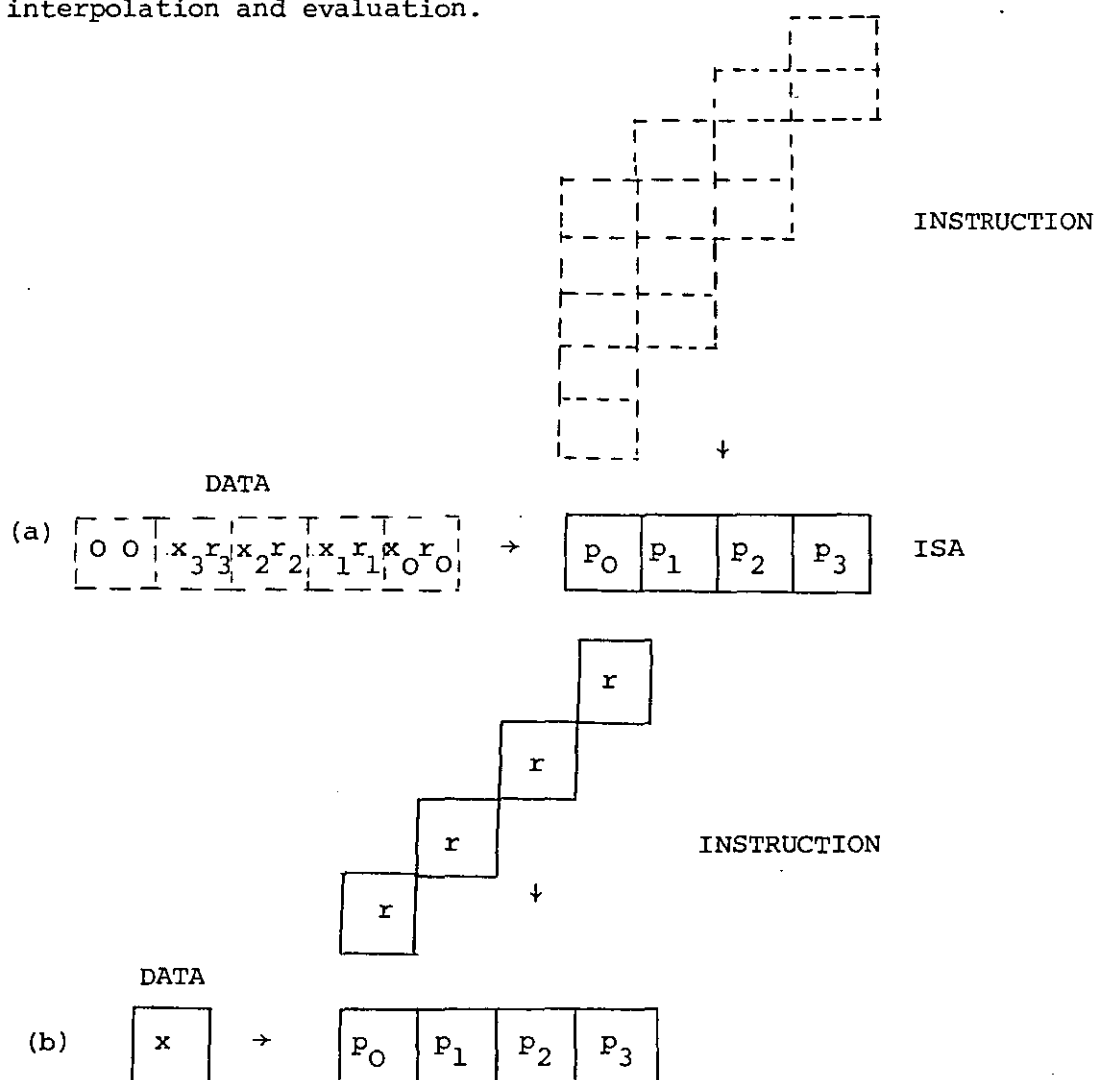


FIGURE 3.27: (a) The ISA-Program for Interpolation, $n=3$
(b) The ISA-Program for Evaluation, $n=3$

3.5.3 Finding Transitive Closure

Finding the transitive R^+ of a binary relation R over a finite set M is fundamental in computing. The well-known Warshall Algorithm [Warshall 1962] solves this problem on a sequential machine in $O(n^3)$ steps, where $n=|M|$. Several algorithms for parallel computers and especially for systolic arrays, which solve the transitive closure problem or related problems in time $O(n)$, are known. In [Guibas, Kung 1979] a systolic algorithm for computing the transitive closure is given. In [Kung, Lo, Lewis 1987] systolic solutions for the transitive closure and the shortest path problem are presented. The problem of finding the connected components of a graph is solved by S.E. Hambrusch [Hambrusch 1983] on a mesh-connected processor array. Y. Robert and D. Trystram [Robert, Trystram 1986] and G. Rote [Rote 1985] give systolic arrays for the algebraic path problem, which is a generalisation of the transitive closure problem.

Hans-Werner Lang [Lang 1987] presents a parallel implementation of the Warshall algorithm on an instruction systolic array. The transitive closure problem may be generalised to the algebraic path problem. So the ISA program given is to implement a generalized closure algorithm for solving the algebraic path problem. A decomposition technique is also given in order to map arbitrary large problem instances onto a processor array of fixed size.

3.5.4 Finding All Cut-Points

A cut-point (articulation point, cut vertex, cut node) of a graph is a vertex whose omission increases the number of connected

components. Finding all cut-points in a connected graph is an important problem with numerous applications as e.g. in network flow theory. M. Schimmmler and H. Schroder [Schimmmler, Schroder 1987] present a method to find all cut-points of an undirected connected graph in time $O(|V|\log|V|)$ on an ISA. It is based on an ISA suitable version [Lang 1987] of Warshall's transitive closure algorithm [Warshall 1962] which is used to check for every vertex whether its removal produces more than one connected component. This algorithm does not meet the lower bound $O(n)$ on the time complexity as it is achieved by M.J. Atallah and S.R. Kosaraju [Atallah, Kosaraju 1984]. Its main advantage compared to the algorithm presented in [Atallah, Kosaraju 1984] is its simplicity.

3.6 THE SINGLE INSTRUCTION SYSTOLIC ARRAY (SISA) VARIANTS OF THE ISA MODEL

The asymmetry of the flow of control information in the ISA-m-bit instruction codes from the top, 1-bit selectors from the left could be resolved by:

- i) breaking the control code into two equal sized parts.

Instruction Prefixes that are shifted through the processor array from the top and Instruction Suffixes that are shifted through from the left. Inside the array prefixes and suffixes recombine to form instructions (including no-op).

- ii) using column selectors in addition to the row selectors and shifting single instructions in diagonal wavefronts through the array, starting at the upper left corner (Figure 3.28).

Now the instructions are executed only if both row and column selector bits are '1'.

The advantage of this variant which will be referred to as SISA (for Single Instruction Systolic Array) is the reduction of the overall amount of code by a factor of approximately $m/2$, m being the length of an instruction code.

Hans-Werner Lang [Lang 1987] sketched the relationships between SISA and other models of mesh-connected processor arrays (ISA and SIMD). The results are summarized here as:

- i) $SISA \leftrightarrow ISA$

A SISA program can directly be transformed into an ISA program by replacing each '1' in a column selector diagonal by the corresponding instruction, each '0' by 'no-op'.

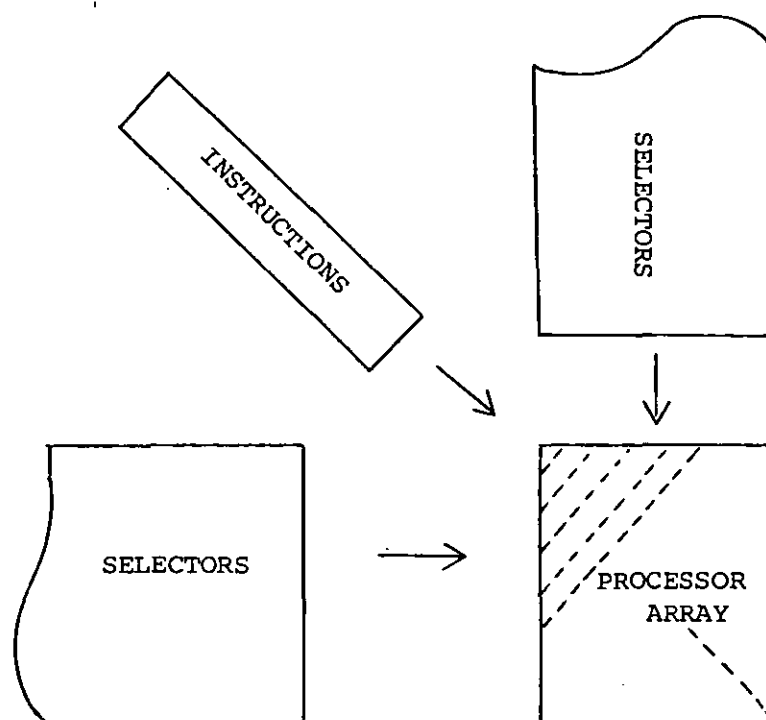


FIGURE 3.28: Single Instruction Systolic Array (SISA)

In the converse direction, only such ISA programs can directly be transformed into SISA programs, where in each instruction diagonal there occur only identical instructions other than no-op. This is quite frequently the case in practical applications.

ii) SISA \leftrightarrow SIMD

One observes a strong similarity between the control structures of the SISA and the SIMD-type mesh-connected processor array. In fact, a SIMD array whose masking mechanism is not more powerful than the SISA masking mechanism, can be simulated by the SISA within a delay factor of 3. The corresponding program transformation consists of replacing each SIMD instruction by a sequence of at most three modified

instructions and "skewing" the masking information appropriately. However, simply "unskewing" a SISA program to get an equivalent SIMD program is in general not possible. It is an open question whether the SISA is more powerful than the SIMD processor array or not!

On the other hand, there are operations like "broadcasting" a data item from the left or from the top across the whole array, or like ring shifting a row or a column of data, that can be realized on the SISA with constant period. On the SIMD array these operations take time $\Omega(n)$, but n of these operations still take only time $O(n)$. Thus, if all these n operations are meaningful, a constant period is achieved also. But it is not clear if, when trying to transform a SISA program to an SIMD program, these meaningful operations can be found. On the other hand, the similarity of control structures of the two models suggests that they are equally powerful.

CHAPTER 4

THE SOFT-SYSTOLIC SIMULATION

SYSTEM (SSSS)

4.1 BASIC DEFINITIONS OF THE SYSTEM

The basic design problem for a general systolic array simulator is to provide a fixed architecture which is capable of simulating the arbitrary graph structure of an array, while also mapping parallel processors to achieve parallelism. In Chapter 2 we have envisaged systolic arrays as a systolic program written in OCCAM language with the implicit understanding that OCCAM can be executed effectively on transputer networks to provide parallelism. The problem with this scheme is that it may be better to write a dedicated transputer based version of a method rather than simulate a systolic array version of the algorithm. Thus, as we accept the idea of programmable arrays the effectiveness of the special purpose systolic approach to specific algorithms is not so important. The essential problem is the emphasis placed on dataflow which demands a different OCCAM program structure for each design. The ISA on the other hand as we mentioned in Chapter 3 places emphasis on the systolic movement of instructions fixing the data communication and processor structure, and the chances of producing a fast and an economic systolic simulator, with an alternative perspective on the meaning of a "SYSTOLIC COMPUTER".

In this chapter we consider a soft-systolic simulation system implemented on the Balance 8000 Sequent Computer System running under DYNIX operating system, at Loughborough University of Technology, Computer Studies Department, and solve a number of common problems to demonstrate its flexibility. The system can be used to develop special purpose algorithms with a regular form and opens up the possibility of a systolic design workstation for development of simple systolic processing systems.

An overview of the system is shown in Figure 4.1, and the main sections of our soft-systolic simulation system are:

* System and Machine Preparation:

comprises the operating system facilities, programming language, and the compiler.

* Replication Instruction Systolic Array Language (RISAL) and RISAL compiler:

which comprises the virtual machine language and the adopted RISAL compiler.

* Virtual Machine:

which consists of an Instruction Systolic Array (ISA) network, a set of virtual spoolers, and a collection of processing elements.

* Virtual to Real Mapping:

Here we define a library of processor plugs which allow a number of virtual processors to be essentially plugged into a single real processing element of the underlying architecture. Thus, allowing a large virtual grid to be mapped onto a smaller real grid.

* The Real Architecture:

For simplicity we assume that this is a square orthogonally connected grid of processors such as a transputer network, capable of executing any of the virtual PE's and mapping plugs.

To demonstrate the feasibility of the system we concentrate in this chapter on the system and machine preparation, and the virtual

machine. In the following chapter we will describe the Replication Instruction Systolic Array Language (RISAL) and its RISAL compiler. However, the virtual machine and RISAL and the compiler will form the core of the design.

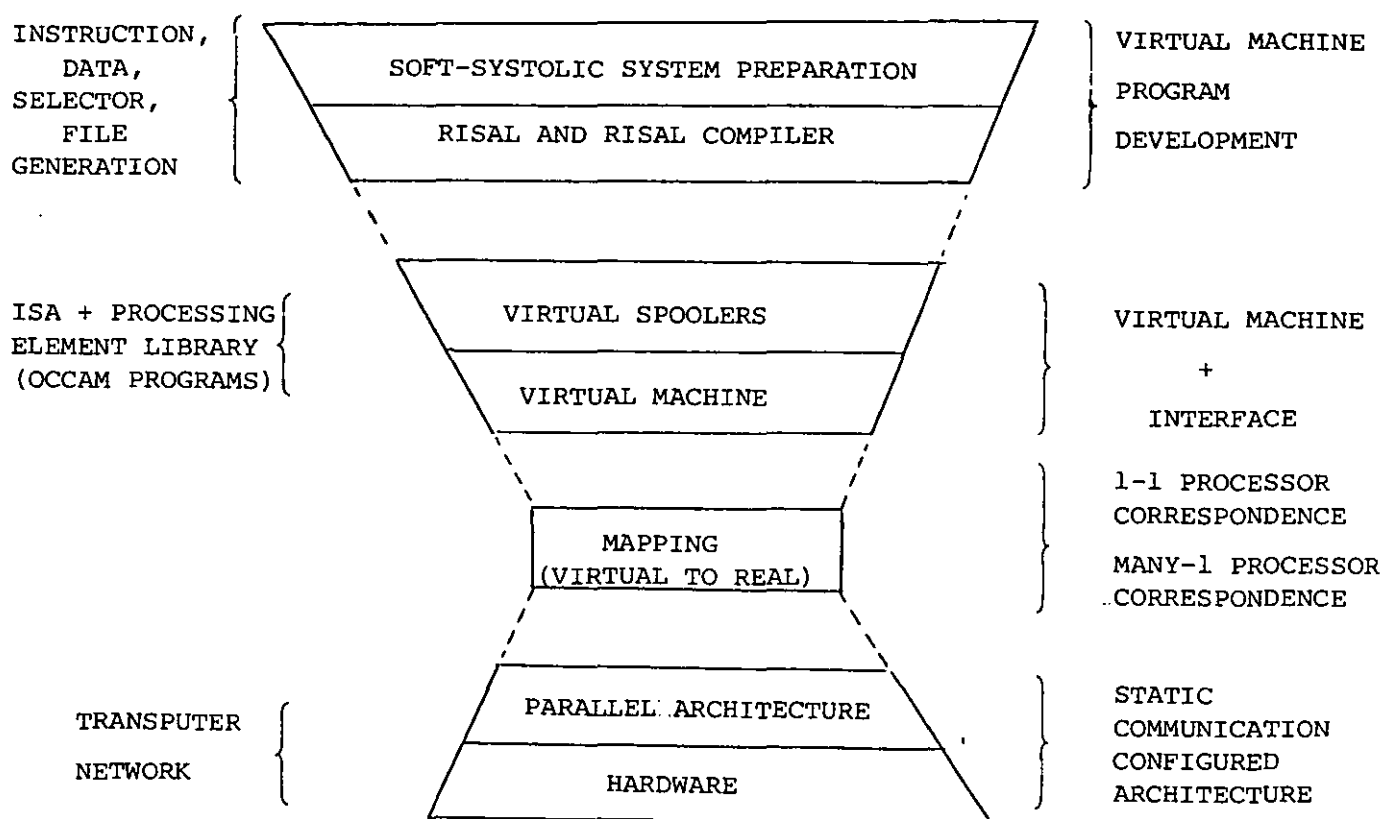


FIGURE 4.1: Organisation of Soft-Systolic Simulation System

4.2 SYSTEM AND MACHINE PREPARATION

The soft-systolic system preparation section comprises of the usual operation system facilities for the creation and modification of files during the development of new programs and ISA processor elements. We allow here any concurrent high level language to be used to model the soft-systolic system.

We develop our soft-systolic system on the Balance 8000 Sequent system, the system operates under the powerful DYNIX operating system which is based on the UNIX uniprocessor operating system with several significant enhanced features to support multitasking. The DYNIX kernel or executive has been made shareable, so that all the processors can execute the same system calls and other kernel code simultaneously. The DYNIX system schedules the processes to execute on the processor such that the workload is well balanced. This means that any user or system-defined process can run on any processor at any time and may involve several processors to complete. The DYNIX determines the minimum and maximum amount of physical memory that a given process can consume, then adjusts the memory allocation for each process between these two bounds to maintain each processor's paging rate and tune the virtual memory performance for the entire system.

The Balance 8000 system provides an excellent environment for software development. Program support tools include the standard UNIX utilities for creation and manipulation, program development, performance analysis, text editing, and document preparation.

The system implemented in OCCAM (Loughborough University Implementation) was implemented by R.P. STALLARD in the Computer Studies

Department. The main features of the OCCAM language are briefly reviewed as: an OCCAM program is written in terms of concurrent processes, communicating via channels. Individual processes operate mostly in independence of each other. Hence, a design problem can be expressed as a hierarchical structure. Groups of processes connected by communication channels can be conceived of as individual processes in their own right, with inter-process connections of the group ignored (a process known as hiding), as a result complex systems can be built with only a few processes under consideration at any one time.

In an OCCAM program, each channel provides a one-way connection between two concurrent processes; one of the processes may output to the channel, and the other may input from it. The two concurrent processes must communicate by using input and output primitives. Input and output are synchronised, and an input will not complete execution until an output on the same channel is also executed, and so on. For example, we have two concurrent processes, initially both will start executing in parallel; after some time one of them will reach its input process. Now, this process has to wait until it receives a value down the channel it is waiting on. There are two possibilities, either the second process is at the output process already (for the same channel) or the second process has not reached its output. In the first case, the process waiting for input immediately receives the value and so does not wait any further. In the second case the input waits and eventually the second process outputs a value, thus completing the input wait of the other process. The key thing is that eventually the communication is made and the

processes continue executing in parallel.

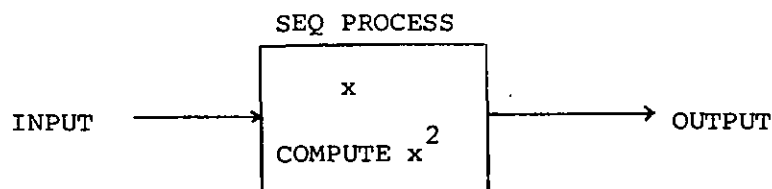
The basic data type in OCCAM is the WORD. The word can be used to represent CHARACTERS, NUMBERS, and BOOLEAN values, as well as BIT PATTERNS. These basic types can be manipulated using a wide variety of arithmetic, logical and bitwise operations (e.g. shift left, shift right, exclusive or).

Values can also be declared as VECTORS, with appropriate subscripting available. VALUES can be declared (i.e. created) by definitions. Definitions are introduced by the keywords CHAN, VAR, DEF. The keyword CHAN, VAR, DEF, and PROG (for named process substitution) is followed by a ":" on the last line of the declaration. The process code must follow on the next line after the declaration at the same level of indentation.

e.g. VAR declaration

```
VAR x:
SEQ
  Input ? x
  Output ! x*x
```

represent



This declares the identifier x to hold a value within the SEQ process.

The keyword CHAN is used to declare a channel used for communication; it is declared in the same way as a VAR except the word CHAN is used.

OCCAM programs are built from a small number of primitive processes. These are:

- (i) ASSIGNMENT (denoted by symbol :=)
- (ii) INPUT (denoted ?)
- (iii) OUTPUT (denoted !)
- (iv) WAIT (denoted WAIT)

Complex programs are built by constructing complex processes by connecting these primitive processes together using constructors.

There are five types of constructors:

- (i) SEquence (SEQ)
- (ii) PARallel (PAR)
- (iii) Conditional (IF)
- (iv) ALternative (ALT)
- (v) Repetition (WHILE)

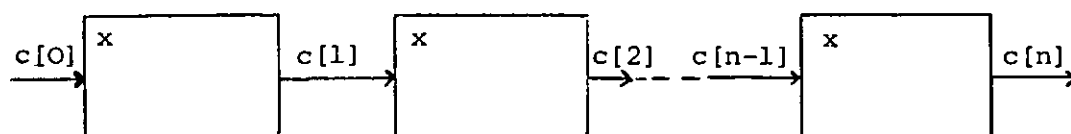
The first four constructors can be accompanied by a replicater, which replicates the component processes to which it is attached a specified number of times, e.g. Creation of n processes performing infinite while-loop with variable x

```

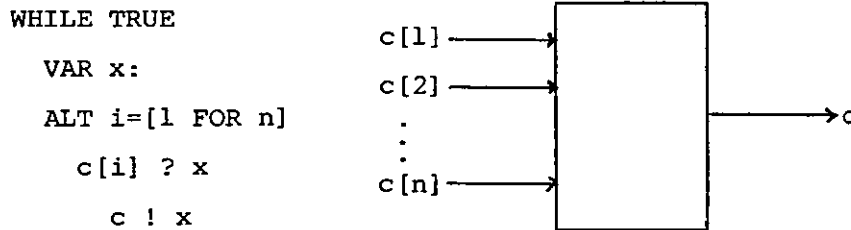
CHAN c[n+1]:
PAR i=[0 FOR n]
  WHILE TRUE
    VAR x:
    SEQ
      c[i] ? x
      c[i+1] ! x

```

represent



e.g. selecting first available input from n-channels and route to output channel simulates a mux or merger.



Each primitive process (input, output,... etc.) is a single line of code, and generally constructs have their component process indented by two spaces from the first letter of the constructor. All processors at the same level of indentation are assumed to be components of the same constructor. Declarations are at the same level of indentation as the constructs they are attached to (note that procedures are declarations). The number of channels declared is fixed in the program text (i.e. constant bounds on vectors) this limits the amount of parallelism in a program definition.

The OCCAM compiler has an improved method of calling routines from the library routines, and provide commonly used routines to read and write to the keyboard and screen channels e.g.

```
EXTERNAL PROC num.from.keyboard (VARn):
```

This means read a number from the keyboard and assign to variable 'n'.

```
EXTERNAL PROC num.from.chan (CHAN c, VAR n):
```

Read a number from a channel 'c'. If 'c' is the keyboard this is equivalent to calling 'num.from.keyboard'. The routines are written in C language and OCCAM. Also provided are general routines for use for pause or to abort a program as well as to use the 'c' random

number routines. These routines are available by default to all programs unless the `-s` compiler flag is used to override their inclusion. Also provided are routines to perform floating point input/output operations. They are available by giving the compiler flag `'-F'` when linking an OCCAM program, which we commonly used in our implementation, floating point value can be assigned and transmitted via channels just like normal integer values, e.g.

```
EXTERNAL PROC fn.num.from.keyboard (VAR FLOAT f):
```

means to read in a floating point number. The number is expected to begin with a digit or `'.'` (indicating 0.), leading spaces are ignored. The number ends with a non-digit and this character will not be available to subsequent reads from the keyboard channel, e.g.:

```
EXTERNAL PROC fp.num.from.chan (CHAN c, VAR FLOAT f):
```

This means to read a floating point number from a channel `'c'`. If the channel is a keyboard this is equivalent to `'fp.num.from.keyboard'` external procedure.

One of the most important features of the OCCAM compiler (5.0 version implemented by R.P. STALLARD) is the Execution Trace. When an OCCAM program is compiled and run with the `'-e'` flag, it produces a trace history of all the synchronization events of all the processes. When compiled the object code includes updates of a timing variable to model the execution of an actual parallel computer. The operation times can be given with the `'-T'` flag, by default they resemble those of the transputer. The trace history file can later be inspected with the `'tracer'` utility, specific time periods can be analysed, discovering where idle time is incurred or tracing the behaviour up to the time of

a fault. The run time system keeps the processes running as if in parallel and not in a round-robin priority fashion so that the program may well behave differently depending on the setting of the '-e' flag. A 'timerbuild' utility is available to construct the user's own timing profile for a target parallel system. The system currently has a number of limitations, it assumes all 'PAR' processes are executing on separate parallel computers and that all channel intercommunication is on direct identical 'links'.

The OCCAM compiler implements the OCCAM language as defined in the OCCAM programming manual published by INMOS Limited subject to a few restrictions and extensions that are described in Appendix I (which comprises a listing of the online documentation for the Loughborough OCCAM 5.0 compiler). These differences are intended to make the transform of OCCAM programs from different implementations feasible.

4.3 THE VIRTUAL MACHINE

A part of the virtual machine chosen for this implementation is the Instruction Systolic Array (ISA). As we have mentioned in Chapter 3, the ISA has a number of interesting features. Firstly it has been used to simulate all SIMD algorithms and many MIMD algorithms by a simple program transformation technique. Further, the ISA can also simulate the so-called wavefront processor algorithms, as well as many hard systolic algorithms, hence allowing the gap between systolic and other needs of computation to be bridged. The ISA removes the need for the broadcasting of data which is a feature of SIMD algorithms (limiting the size of the machine and its cycle time) and also presents a fairly simple communication structure for MIMD algorithms. The model of systolic computation developed from the VLSI approach to systolic arrays is such that the processing surface is fixed, as are the processing elements or cells by virtue of their being embedded in the processing surface.

The VLSI approach therefore freezes instructions and hardware relative to the movement of data, with the virtual machine and soft-systolic system retaining the constraints of VLSI for array design of regularity, simplicity and local communication, allowing the movement of instructions with respect to data. Data can be frozen in the structure with instructions moving systolically or both the data and instructions can move systolically around the virtual processor (which are deemed fixed relative to the underlying architecture). Our virtual machine can thus implement time-static and time-dynamic systolic algorithms, allowing the virtual machine to be fixed (static)

during the time of computing as for hard systolic algorithms or dynamically changing from one systolic configuration to another on the virtual processing surface with time.

The virtual machine consists of an ISA network of data and control paths, and a set of virtual spoolers for driving the ISA computation and opening up the communication bandwidth of the array, and a collection of processing elements (PE) descriptions for creating specific ISA grids. In the following two paragraphs we will describe the basic sections of the virtual machine.

4.3.1 The Instruction Systolic Array (ISA) Network

The Instruction Systolic Array is an orthogonal grid of processing elements. Each processing element executes a number of simple operations, and includes memory for intermediate results and registers for communication with other processing cells and a save register holding results until the neighbouring PE's have had a chance to read the communication\$. Each PE is activated by a combination of an instruction and selector. If the selector entering the cell is high (1) it executes the instruction which also entered the cell. Otherwise the cell remains inactive if the selector is low (0). The selectors move through the ISA column by column, while the instructions move row by row as shown in Figure 4.2.

The systolic movement of instructions and selectors is reminiscent of the wavefront processor and obviates the need of control store in the PE as is required for designs like the 'PSC WARP' device. Additional data inputs on the boundary make it easy to simulate a wavefront processor.

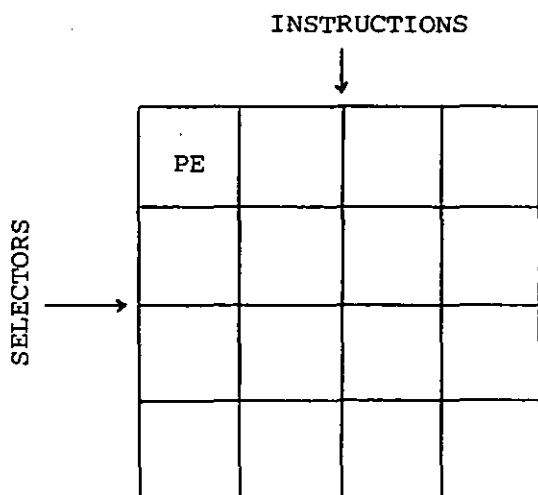


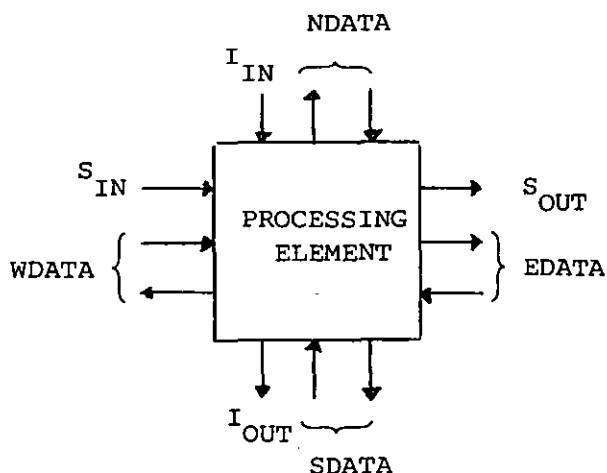
FIGURE 4.2: ISA Processors (Grid 4x4)

To retain the possibility of a straightforward mapping of virtual machine to real processor architecture we implement the ISA in OCCAM, using the powerful system features of DYNIX coupled with Loughborough OCCAM, the ISA was easily specified as a two part design consisting of:

1. PE library files
2. Grid architecture and virtual spoolers.

The virtual spoolers played the role of buffers for the ISA array interface with higher levels of the system, allowing the bandwidth of the input to meet that of the ISA. The grid architecture was a simple specification of network connections between processors, the PE libraries simply containing cell descriptions which responded to ISA instructions with different characteristics. Loughborough OCCAM allows the precomputation of library PE's and the grid connection network, which could be simply linked when the virtual machine was required to run effectively plugging in the correct PE's. Thus, a user of the

system can develop programs and new PE's with only an abstract working knowledge of the ISA grid. The virtual grid architecture is shown in Figure 4.4 based on the cell structure for a 4x4 case as shown in Figure 4.3.



NDATA = NORTH INPUT/OUTPUT DATA

EDATA = EAST INPUT/OUTPUT DATA

SDATA = SOUTH INPUT/OUTPUT DATA

WDATA = WEST INPUT/OUTPUT DATA

S_{IN}, S_{OUT} = SELECTOR INPUT/OUTPUT

I_{IN}, I_{OUT} = INSTRUCTION INPUT/OUTPUT

FIGURE 4.3: The Cell Structure

Included with the ISA grid specification is the data and instruction spooler code. The spoolers are concurrent processes representing buffers for data and instruction input to the boundary cells of the grid. The spoolers also include data output and instruction/selectors garbage collection for values falling off the grid. The interface between the virtual machine and the program/PE

CHANNEL SPECIFICATIONS
 OF THE ISA GRID:

INS - INSTRUCTION NORTH
 SOUTH

ANS - 'A' NORTH SOUTH

BNS - 'B' " "

SEL - SELECTOR WEST
 EAST

AWE - 'A' WEST EAST

BWE - 'B' " "

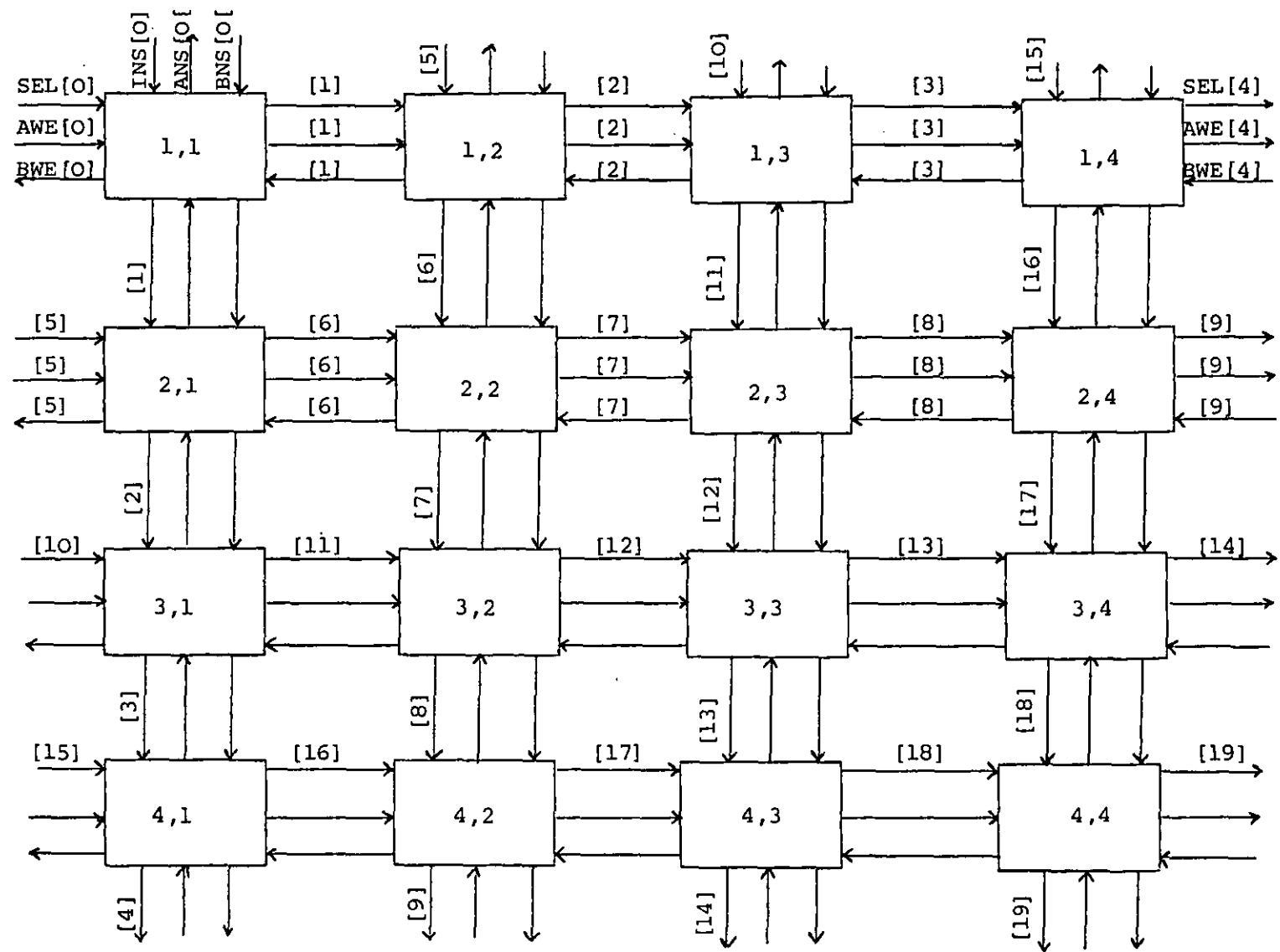


FIGURE 4.4: The Virtual Grid Architecture

development section is assumed to be of narrow bandwidth. In fact all data and instructions are assumed to be placed in three files denoted DATAIN, SELECTOR, INSTRUCT, and the output is dumped in DATAOUT to represent virtual spool files. The virtual spoolers (shown in Figure 4.5) read these sequentially and convert the input into a parallel form for the ISA. Likewise for the ISA output the spooler converts the output back into a single stream output sequentially to DATAOUT. The reading of input and writing of output data is performed in parallel with the ISA execution. Clearly this is the place where any bottlenecks are likely to occur especially for large n . The spoolers can also be used to pad out unused cells with dummy values, when the ISA program running is smaller than the total number of virtual processors. Hence the system with a bounded number of processors can simulate smaller networks without difficulty, [Muslih and Evans, 1987].

To allocate the channel to the virtual grid architecture, the correct channels can be hooked up by a simple computation using the grid PE position of the form,

```
PROC LOC (VALUE i,j, VAR V)=
  SEQ
  r:=(((i-1)*(n+1))+j)-1:
```

The PE to fit the locations is called as a library routine,

```
EXTERNAL PROC PE(CHAN wn,we,ws,ww,rn,re,rs,rw,in,is,sw,se)=
```

and the library PE section uses the PE definitions:

```
LIBRARY PROC PE(CHAN wn,we,ws,ww,rn,re,rs,rw,in,is,sw,se)=
```

... code for the cell.

The external environment communicates with the grid (processing elements network) by passing the data, instruction and selector to be

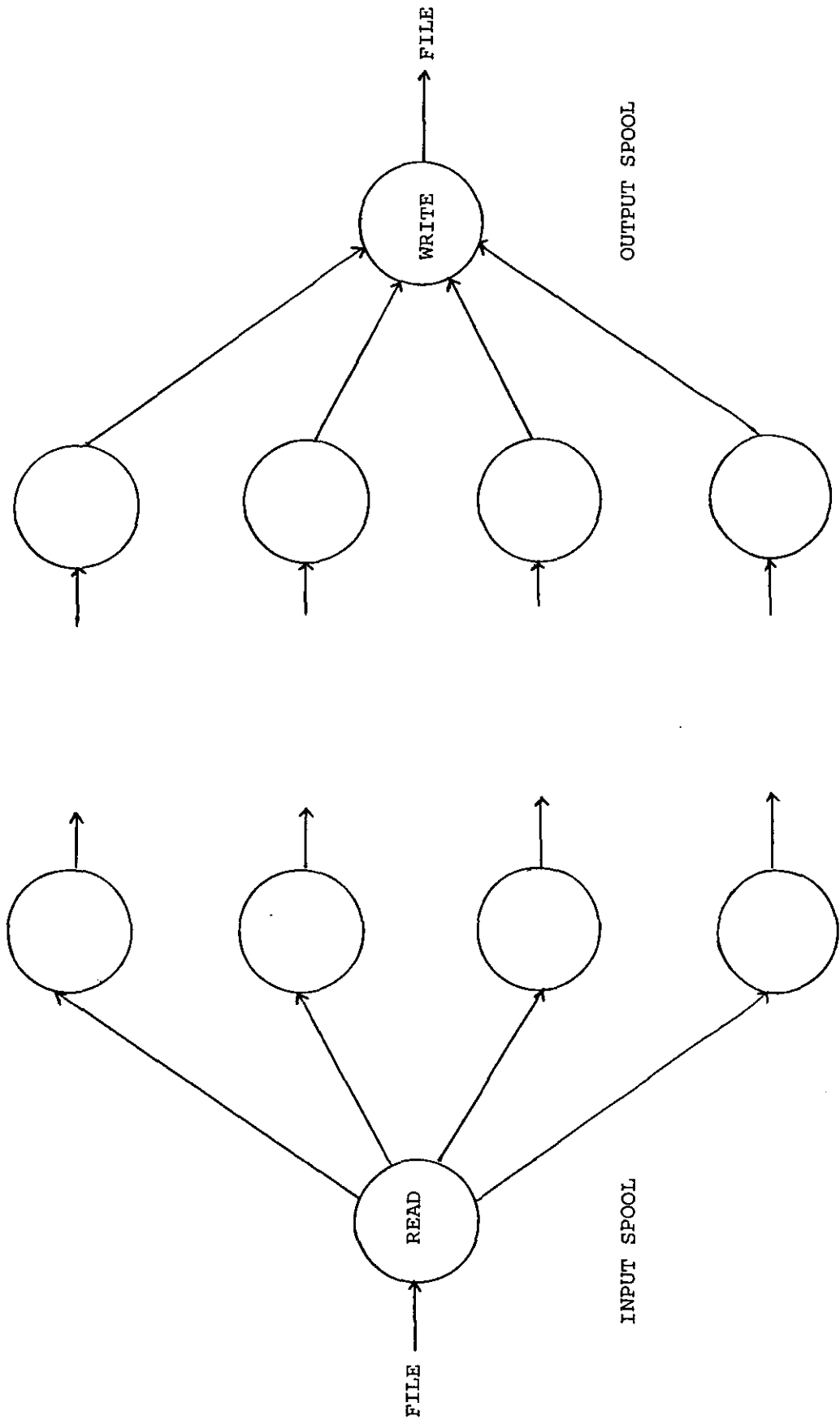


FIGURE 4.5: Virtual Spoolers (for N=4)

sent to sources. Sources communicate the signals directly to the grid interfaces through the virtual spoolers, by pumping the signals into the grid, this serves the purpose of each source modelling itself as a process.

Information sent down the source interface channels can be classified into two categories:

- a) control - (instruction, and selector) for directing the computation.
- b) data - for use in computation.

For simplicity, we can consider separate sources for the data and control (instruction and selector). The merging of sources for data can be performed, likewise merging of the control (instruction, and selector) sources can be performed.

To implement this concept in OCCAM, the generic source for the instruction and selector files, which is sequential to parallel program bus expander is:

```
PROC SOURCE (CHAN OUT [], Link, value t)=
  VAR k,i,j, buffer [n]:
  CHAN ptr
  SEQ
  IF
    t=0
    open.file ("selector","r",ptr)
  TRUE
    open.file ("instruct","r",ptr)
```

open the selector input file if t=0, otherwise open the instruct input file.

To read the next input line from the selector or instruct input

file and pump in parallel into the grid, we write:

```

num.from.chan(ptr,k)
link ! k
SEQ i=[1 FOR k]
  SEQ
    IF
      i>k
        PAR j=[1 FOR n]
          VAR t1:
            SEQ
              loc (j,l,t1)
              OUT [t1] ! 0
        TRUE
          SEQ
            SEQ j=[1 FOR n]
              num.from.chan (ptr,buffer[j-1])
            PAR j=[1 FOR n]
              VAR t1:
                SEQ
                  loc (j,l,t1)
                  OUT [t1] ! buffer [j-1]
close.file(ptr)
str.to.screen ("*n source closed")
link ! 0

```

The opposite of a source process is the sink process and the sink cannot perform any output to the grid. Data and control (instruction and selector) leaving the grid through the virtual spoolers directly enters the sink, where it is routed to relevant areas of surrounding outer environment. In this light the sink corresponds to the output, it also collects all the data and control (instruction, and selector) pumped into the grid by the source.

As for sources we allow multiple sinks processes, corresponding to different stages of output. So the purpose of the garbage collector (sink process) is to collect the instructions and selectors and output them from the grid, i.e.,

```
PROC SINK (CHAN in[],link)=
  VAR i,j,k
  SEQ
    link ? k
    SEQ i=[1 FOR k]
      PAR j=[1 FOR n]
        VAR t1:
          SEQ
            loc (j,n,t1)
            in [t1+1] ? any
str.to.screen ("*n sink closed")
link ? any :
```

The generic source for the DATA file (data bus expander), to open the data input file "datain" and decide the number of lines in the file for each input line is:

```
PROC data.source (CHAN ans[ ], bns[ ], awe[ ], bwe[ ],link)=
  DEF n1=2*n, n3=3*n:
  VAR k,i,j,t:
  VAR FLOAT buffer [4*n]:
  CHAN ptr
  SEQ
    open.file ("datain","r",ptr)
    num.from.chan (ptr,k)
    link ! k
    str.to.screen ("*nk=")
    num.to.screen (k)
```

To read the north, east, south and west boundaries of the grid is:

```

SEQ i=[1 FOR k]
  SEQ
    str.to.screen ("*ni=")
    num.to.screen (i)
    SEQ j=[0 for 4]
      IF
        i<=k
          SEQ
            num.from.chan (ptr,t)
            IF
              t<0
                SEQ z=[0 for n]
                  buffer [(j*n)+z]:=0.0
                TRUE
                  SEQ z=[0 FOR n]
                    fp.num.from.chan (ptr,buffer [(j*n)*z])

```

To pump all the data elements around the boundaries into the ISA grid in parallel is:

```

TRUE
  SEQ z=[0 for n]
    buffer [(j*n)+z]:=0.0
PAR j=[1 FOR n]
  VAR t1,t2:
  SEQ
    loc (j,1,t1)
    loc (j,n,t2)
    t2:= t2+1
  PAR
    bns[t1] ! buffer [(j-1)]
    bwe[t2] ! buffer [n+(j-1)]
    awe[t1] ! buffer [n3+(j-1)]
    ans[t2] ! buffer [n2+(j-1)]

```

To close the input file "DATAIN" is:

```

close.file (ptr)
str.to.screen ("*n DATA source closed")
link ! 0:

```

To open the output file "DATAOUT" and read all the boundaries in parallel (parallel to sequential bus condenser) is:

```

PROC data.sink (CHAN ans[ ],bns[ ],awe[ ],bwe[ ],link)=
  DEF n2=2*n, n3=3*n
  VAR k,i,j:
  VAR FLOAT buffe[4*n]:
  CHAN ptr:
  SEQ
    open.file ("dataout","w",ptr)
    num.from.chan(ptr,k)
    link ? k
    SEQ i=[1 FOR k]
      SEQ
        PAR j=[1 FOR n]
          VAR t1,t2
          SEQ
            loc (j,1,t1)
            loc (j,n,t2)
            t2:=t2+1
          PAR
            ans[t1] ? buffer [j-1]
            awe[t2] ? buffer [n+(j-1)]
            bns[t2] ? buffer [n2+(j-1)]
            bwe[t1] ? buffer [n3+(j-1)]

```

The output is sequential to the dataout, i.e.,

```

SEQ
  SEQ j=[0 for 4]
    SEQ
      str.to.chan (ptr, "*n")
      SEQ z=[0 FOR n]
        SEQ

```

```

fp.num.to.chan (ptr,buffer [j*n+z])
str.to.chan (ptr, " ")
str.to.chan (ptr, "*n")

```

To close the output file is:

```

close.file (ptr)
str.to.screen ("*n data sink closed")
link ? any
abort.program:

```

To define the ISA grid (network), which is the main procedure comprising the setups and to start the ISA grid is:

```

DEF size=n*(n+1)
CHAN ans[size],bns[size],awe[size],bwe[size],sel[size],ins[size]:
CHAN link[3]
VAR i,j:
PAR
  PAR i=[1 FOR n]
    PAR j=[1 FOR n]
      VAR t1,t2,t3,t4:
      SEQ
        loc (i,j,t1)
        loc (j,i,t2)
        t3:=t1+1
        t4:=t2+1
        plug (ans[t2],awe[t3],bns[t4],bwe[t1],bns[t2],bwe[t3],
              ans[t4],awe[t1],ins[t2],ins[t4],sel[t1],sel[t3])

```

The interface program which will connect the selector file (source and the selector file (sink) is:

```

source (sel,link[0],0)
sink (sel,link[0])

```

The interface program which will connect the instruction file (source) with the instruction file (sink) is:

```

source (ins,link[1],1)
sink (ins,link[1])

```

The interface program will connect the data file (source) with the data file (sink) is:

```

data.source (ans,bns,awe,bwe,link[2])
data.sink (ans,bns,awe,bwe,link[2])

```

The dimension of the array in the case of a 4x4 grid: DEF n=4, and if the user wishes to change the dimension of the array we only need to change the value of n.

To run the ISA grid program described above we need to use the interface routines which are called from the library routines. The interface routines used here are shown in the complete code of the ISA grid in Appendix II.

4.3.2 The Processing Element (PE)

The processing element (PE) considered here in our implementation is a very general element which allows the choice of a wide range of arithmetic and logical operators, and allows the simulation of a wide class of algorithms without the need to develop more special purpose PE's immediately. As the design unfolds it becomes apparent that highly specialized processors can be developed by reducing the number of instructions implemented by the PE's, [Muslih and Evans, 1987].

The PE design indicates the type of program required to deal with the movement of instructions and selectors through the array which will in the main be a generic form for all library PE's.

The PE to be developed is fairly complex and is shown in Figure 4.6. It consists of a central processing element which is enabled by

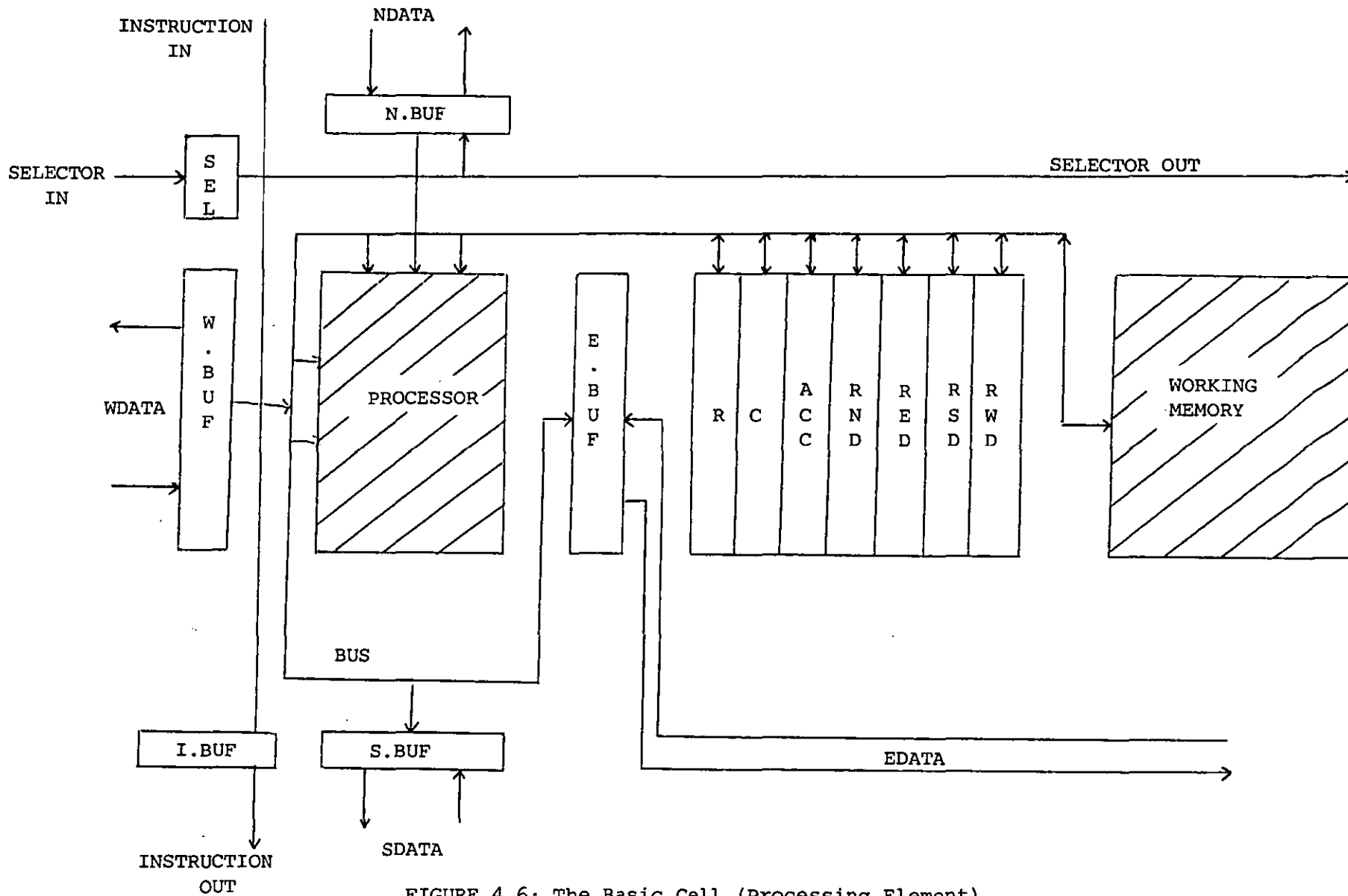


FIGURE 4.6: The Basic Cell (Processing Element)

a selector high signal and any instruction except the NO.OP. A simple bus connects the port input buffers to the memory (Figure 4.7), which contain the port value storage registers (RND, RED, RSD, RWD) as well

		A	R	R	R	R	AUXILIARY MEMORY FOR TEMPORARY VARIABLES + DATA
R	C	C	N	E	S	W	
		C	D	D	D	D	

R: RESULT REGISTER

C: COMMUNICATION REGISTER

ACC: SECONDARY ACCUMULATOR

RND: REGISTER NORTH DATA

RED: REGISTER EAST DATA

RSD: REGISTER SOUTH DATA

RWD: REGISTER WEST DATA

FIGURE 4.7: The Memory Organization

as working memory for data and temporary results and variables, and R which acts as an accumulator and holds the results of the computation until C has been read and ACC which is a secondary accumulator for complex computation, and C which is the communication register (the current output of the cell).

This processing element embodies all the principles of the ISA cell. Communication can be achieved by first loading the output buffers with C, and then taking the input and output in parallel. The input buffers are then read sequentially to memory to complete the communication phase and various masks can be made on the input buffers so as to prevent the overwriting of RND, RED, RSD or

RWD and so avoid unnecessary movement of data in the memory, when a previous input is to be retained. The port mask is defined as part of the processor instruction which is a four field instruction. For simplicity there is the need to keep the bandwidth narrow. The instruction is represented as an 8 digit integer (Figure 4.8) with each field 2-digits wide to allow the possible implementation of 100

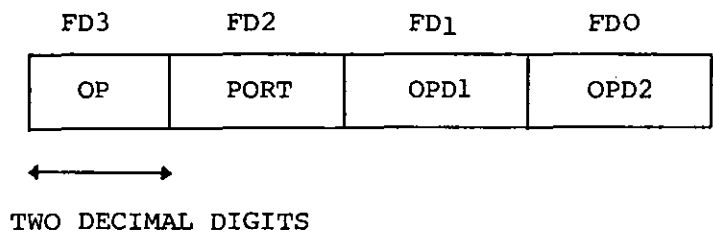


FIGURE 4.8: The Instruction Format

instructions and an internal memory address space of 100 instructions. The port specification allows 100 combinations of Input/Output but only the first 16 have been used here. One possible extension is to utilise the extra slots to allow multiple communication registers in each cell.

REMARK: These operations can be implemented more effectively by using bit logic and slices, but the Loughborough OCCAM is restricted in this respect. Furthermore, a 2-digit field also allows a wide range of library PE's to be developed.

The processor operation codes (Figure 4.9) and the port controllers (Figure 4.10) indicate the instruction to be implemented. The definitions of the read masks using the port instruction field are a high bit indicates that the value of what input port will be copied to memory, while a low bit indicates that the value is not transferred.

OP	CODE	COMMENT
00	NULL	NO OPERATION
01	COPY	MOV R TO C
02	ADD	R:=A+B
03	SUB	R:=A-B
04	MULT	R:=A*B
05	DIV	R:=A/B
06	MIN	R:=MIN(A,B)
07	MAX	R:=MAX(A,B)
08	DATA	C:=A
09	MOV	MEM[FDO] := A

A=MEM[FD1], B=MEM[FDO]

FIGURE 4.9: The Processor Operation Code

The instruction format allows two address fields OPD1 and OPD2 which can be used for memory referencing, including RND, RED, RSD, RSW, R,C, and ACC hence quite general data manipulation can be formed. Originally an extra result field was intended but would not fit into a single integer sized data item.

The resulting instructions are easily decoded by the following OCCAM code:

```
SEQ j=[0 FOR 4]
  SEQ
    fd[j]:=i\100
    i:=i\100          i=Instruction Integer
```

W	S	E	N	INPUT VALID	
0	0	0	0	NO VALID DATA	
0	0	0	1	N	VALID
0	0	1	0	E	VALID
0	0	1	1	N,E	VALID
0	1	0	0	S	VALID
0	1	0	1	S,N	VALID
0	1	1	0	S,E	VALID
0	1	1	1	S,E,N	VALID
1	0	0	0	W	VALID
1	0	0	1	W,N	VALID
1	0	1	0	W,E	VALID
1	0	1	1	W,E,N	VALID
1	1	0	0	W,S	VALID
1	1	0	1	W,S,N	VALID
1	1	1	0	W,S,E	VALID
1	1	1	1	W,S,E,N	VALID

FIGURE 4.10: The Port Controllers

and the port mask with `port:=fd[2]`

```
SEQ i=[0 FOR 4]
```

```
SEQ
```

```
  p[i]:=port\2
```

```
  port:=port/2
```

To define the size of the processing element and the external interface in OCCAM:

```
LIBRARY PROC PE(CHAN wn,we,ws,ww,rn,re,rs,rw,in,is,sw,se)=
```

```
  DEF msize=10:
```

```
  VAR FLOAT a,b, mem[msize],c,i.o.buf[4]:
```

```
  VAR i,j,s,port,p[4],fd[4],op,old.i,old.s:
```

```
  VAR running
```

where mem[msize] is the internal memory cell, i.o.buf[] is what was input from adjacent cells, and c contains the value which the cell is outputting. Finally old.i and old.s is last instruction and last selector signal.

To initialise the cell memory and switch on the cell to make it ready to start and fetch instruction is:

```
SEQ
```

```
  running:=true
```

```
  mem[1]:=0.0
```

```
  mem[0]:=0.0
```

```
  old.i:=0
```

```
  old.s:=0
```

```
  WHILE running
```

```
    SEQ
```

```
      c:=mem[1]
```

To read instruction, selector and data, and send the old instruction, old selector and the value of c through the channels is:

```
PAR
```

```
  in ? i
```

```
  is ! old.i
```

```
  sw ? s
```

```
  se ! old.s
```

```
  wn ! c
```

```
  we ! c
```

```
  ws ! c
```

```
  ww ! c
```

```
  rn ! i.o.buf[0]
```

```

re ! i.o. buf[1]
rs ! i.o. buf[2]
rw ! i.o. buf[3]
old.s:=s
old.i:=i

```

The next stage is the decoding of the instruction and the ports as described above, and then copying the valid data is:

```

SEQ i=[0 FOR 4]
  IF
    p[i]=1
    mem[i+3]:=i.o.buf[i]

```

To execute the instruction considering the processor operation code is:

```

a:=mem[fd[1]]
b:=mem[fd[0]]
IF
  (s<>0) AND (op<>0)
  IF
    op=1
      mem[1]:=mem[0]  (null operation)
    op=2
      mem[0]:=a+b      (add operation)
    op=3
      mem[0]:=a-b      (sub operation)
    op=4
      mem[0]:=a*b      (mult operation)
    op=5
      mem[0]:=a/b      (div operation)
    op=6
      SEQ
      IF
        a>b
          mem[0]:=a
        TRUE      (get min operation)
          mem[0]:=b

```

```

op=7
  SEQ
    IF
      a>b
        mem[0] := a
      TRUE      (get max operation)
        mem[0] := b
op=8
  mem[1] := mem[fd[1]]      (get data operation)
op=9
  mem[fd[0]] := a :      (moving data operation)

```

The full processing element (PE) OCCAM coding is given in Appendix II.

CHAPTER 5

THE IMPLEMENTATION OF THE REPLICATING

INSTRUCTION SYSTOLIC ARRAY LANGUAGE

(RISAL) AND SYSTEM TESTING

5.1 INTRODUCTION

While a great deal of programming language design has progressed, much of it has been at cross purposes. On the one hand the designer has been trying to facilitate the messy process of human understanding; on the other hand he has had to insure efficient use of modern computers. These difficulties constitute the impedance match between grossly different representations. In some sense the designer has been limited to the top of a tower of languages that starts at bits in a computer memory and builds up through the stages to his higher level language. Between each stage there must be an automatic translation program. As might be expected, there is only a limited amount of variation possible under these constraints. The major concepts that have arisen are the variables and structures composed of variables which are, in fact, ways of using computer memory; finite functions over data structures; and sequence control. The fact that programming costs now exceed computer costs has forced the language designer to concentrate more on structuring the programming process than the program itself. There is as much to save by reducing the pre-inspiration flailings of a programmer as there is in eliminating a redundant STORE in the inner loop.

Two additional levels of language appear to be forming on top of the more traditional programming structures (Figure 5.1). One is characterized by a top-down analysis of the program structure. The other is characterized by predicates over various abstract data structures. At the highest level we now see we have statements of things that must be true, perhaps at specific points in the computation.

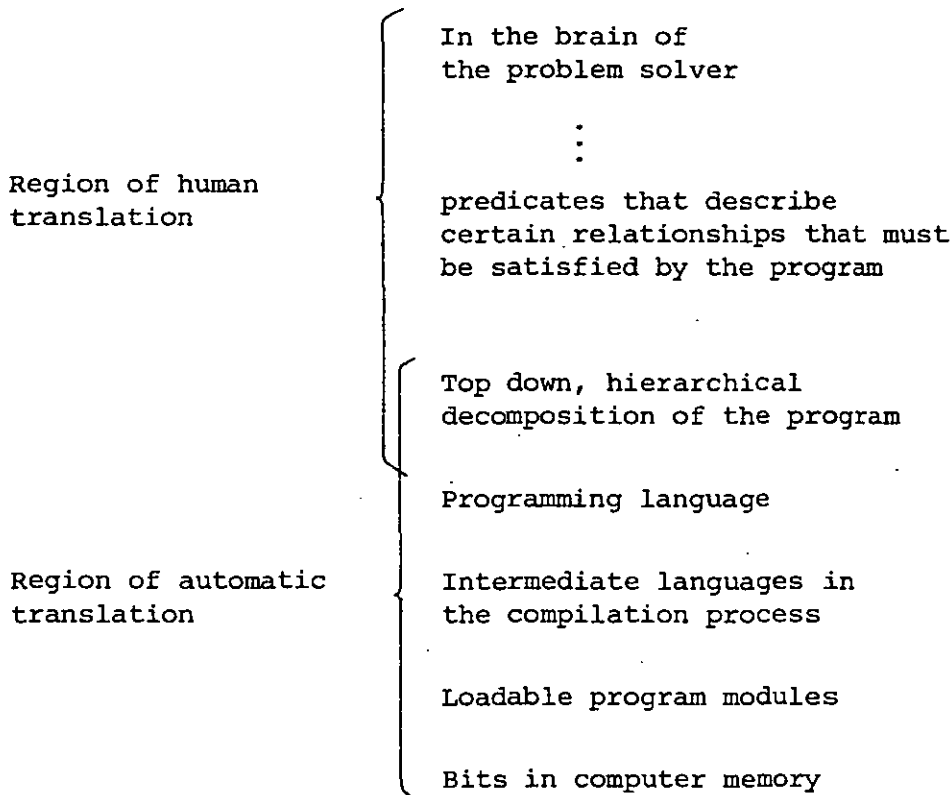


FIGURE 5.1: Levels in the Program Solving Language Tower

Once we have established these restrictions, we fragment the program hierarchically into the most natural units possible. Only then do we map the program onto machine-related constructs. These topmost mappings are probably not done automatically; it is easier to do them by hand than to formalize the mapping process. Again, since it is the programming process that is being facilitated, we observe that progress down the tower of abstraction may well run into problems, causing lower level obstacles to be solved by changing higher level descriptions. It is an iterative process involving all levels of abstraction.

The substantive questions are what structures are useful at each of the various levels of abstraction. The new viewpoint is that it is

not necessary to mix all the levels into one notation. To put it differently, it was a mistake to assume we could think effectively in a (single) programming language.

5.2 LANGUAGE DESIGN PRINCIPLES

There are many motives for the design of computer languages, but the point of view expressed here is that there is a special application area and a special machine which needs a special language. The first question, is how to do it? The first rule, is to keep it simple. The problem then reduces to achieving the necessary features in a consistent manner. The simplest way to proceed is to write some programs. A small program will generally exercise a large part of the language. Then attempt to use the grammars to specify the language concisely. The restrictive form of definition will surely suggest changes in the language, then, in turn, changes in the sample programs. We iterate the process until both the programs and the language description are elegant and understandable.

One might suspect that the language would not improve by having to conform to a restrictive defining tool. But experience shows that it does. In some sense there is no art unless there is a restriction of the medium. In some perverse way, the human mind, in coping with the restrictions, produces its best results, and grammars, the very formalization of nested definition, are a rich medium.

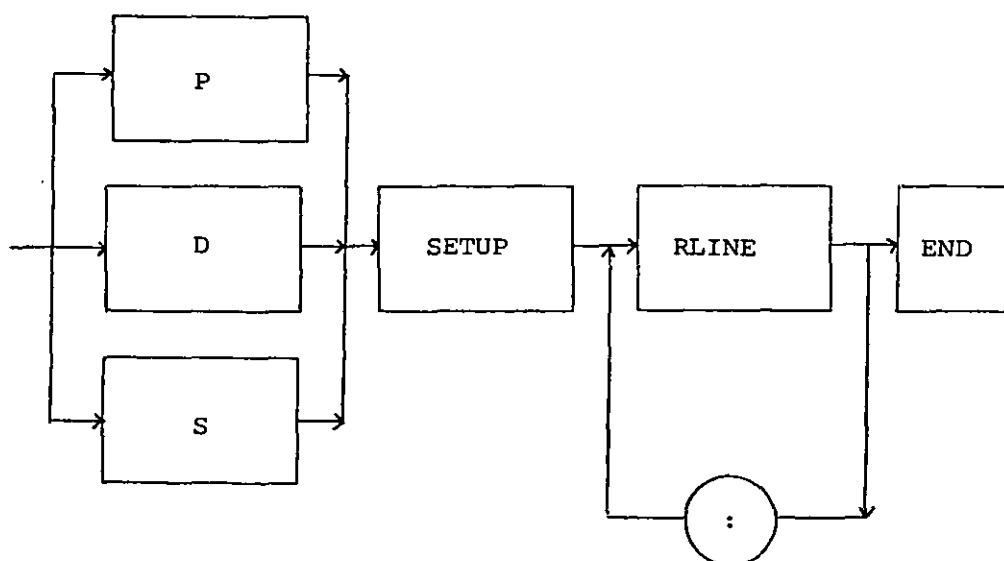
Orthogonality is a desirable property to the language. The facilities that are there should be highly independent, e.g. if there are features for sequence control, then there should not be an additional set of sequence controlling features down inside the instructions.

Adequacy is also a desirable property. It should be able to express the solutions to all the algorithms to be solved in it, but

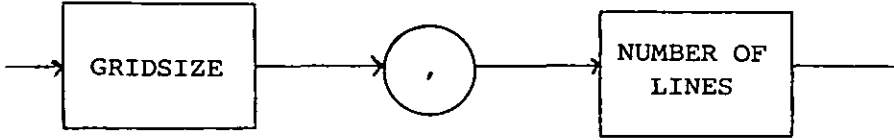
that is not the same as generality, or completeness. There is no reason to be able to compute an arbitrary function if we know ahead of time that only certain simple classes of functions are going to be used. Translatability is a desirable property for the language. There is not much point in designing a neat language that cannot be translated, eventually, to a form acceptable by the machine.

Given the ISA grid and the processing element (PE) to plug into the grid points, we require a suitable medium in which to prepare and debug the ISA control programs, and a method for generating the necessary form of instructions for the ISA. Early test programs were developed in a format akin to a machine code and were difficult to modify and relate to the abstract algorithms. The RISAL compiler (see later) was developed and introduced to allow a simple but adequate design environment. RISAL accepts instructions in an assembler like form, but is fairly permissive about the format within the constraints of syntax. The syntax of RISAL is:

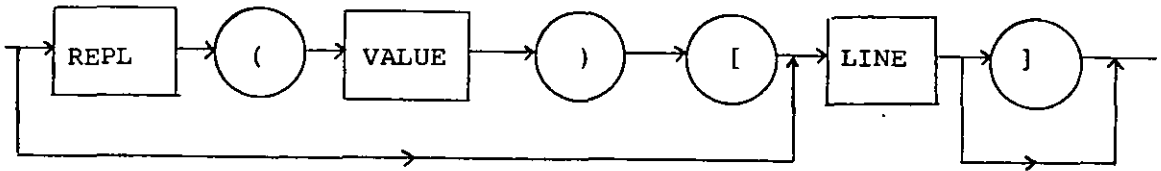
1. RISAL FILE



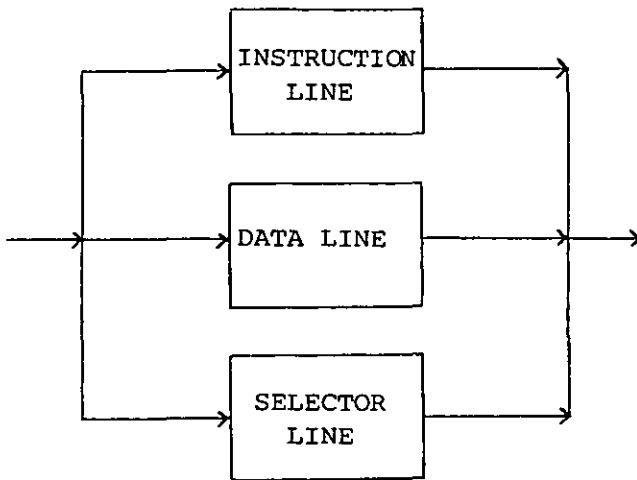
2. SETUP



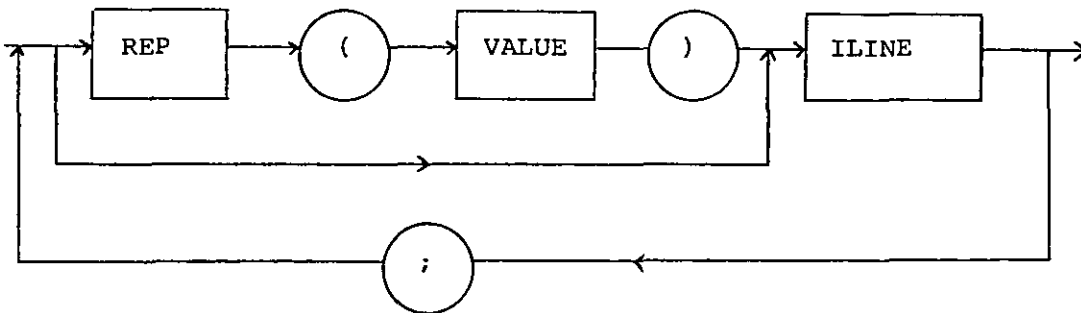
3. RLINE



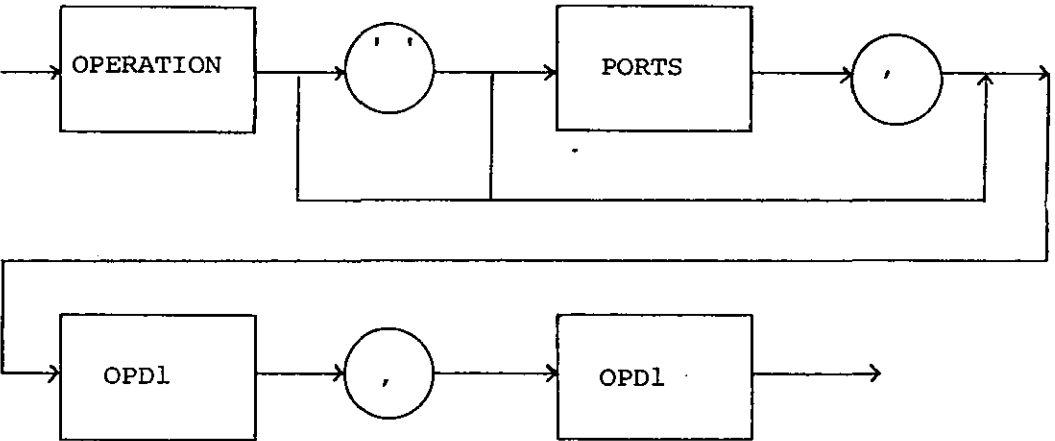
4. LINE



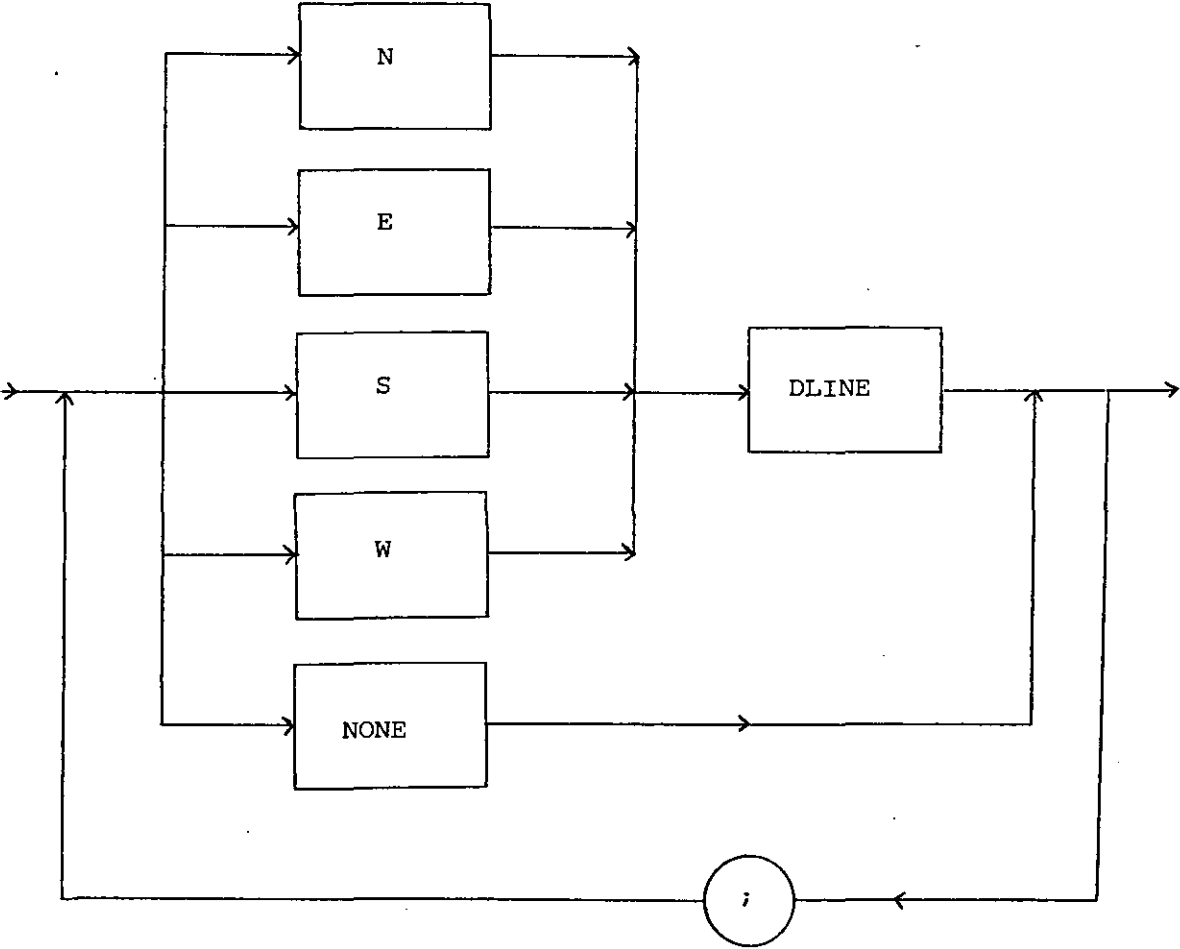
5. INSTRUCTION LINE

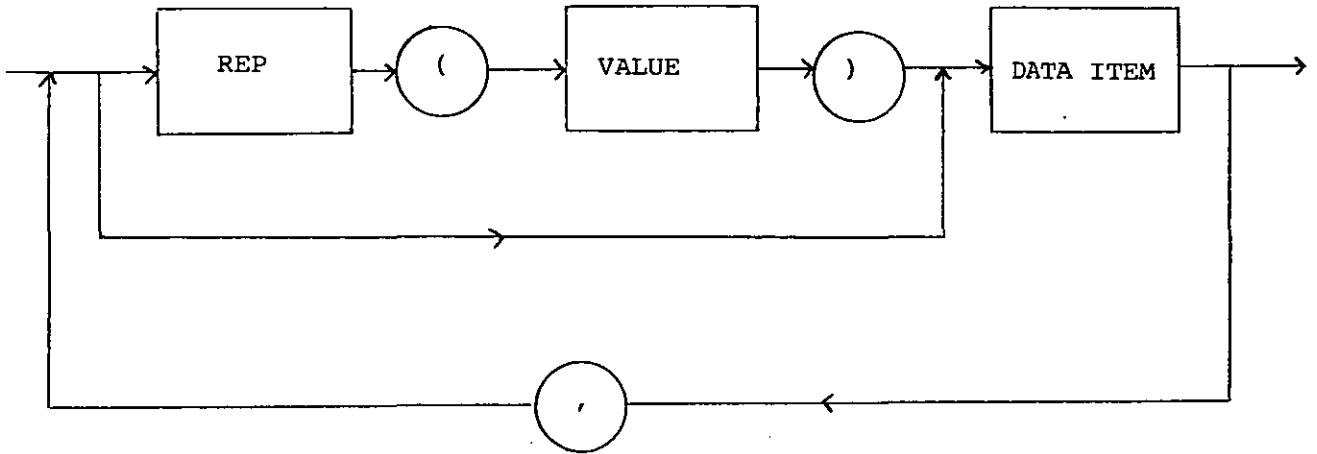
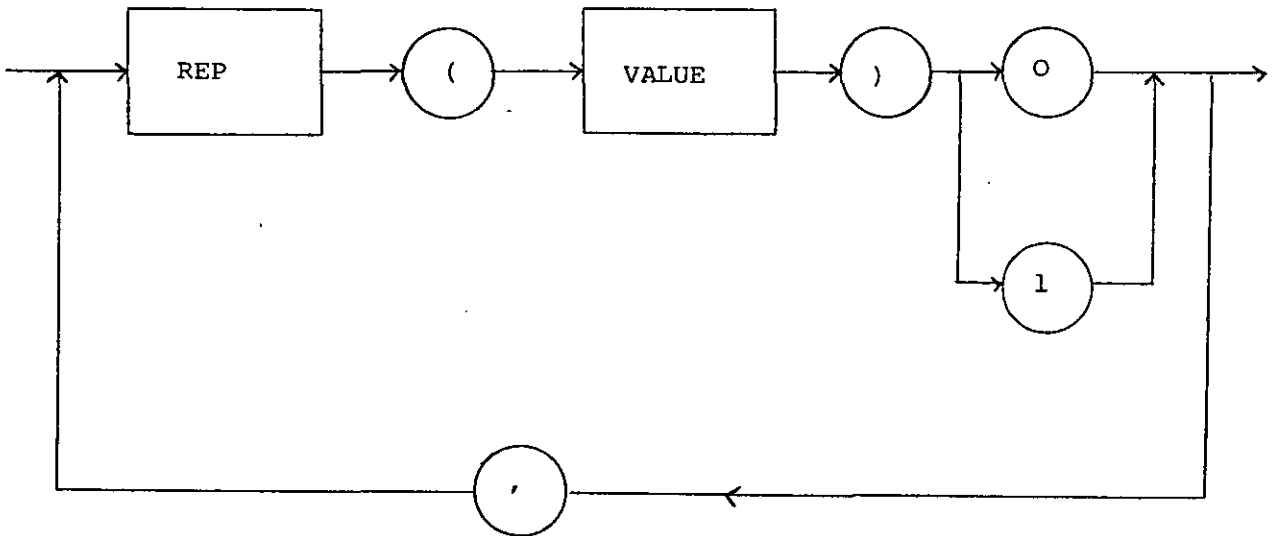


6. ILINE



7. DATALINE



8. DLINE9. SELECTOR LINE

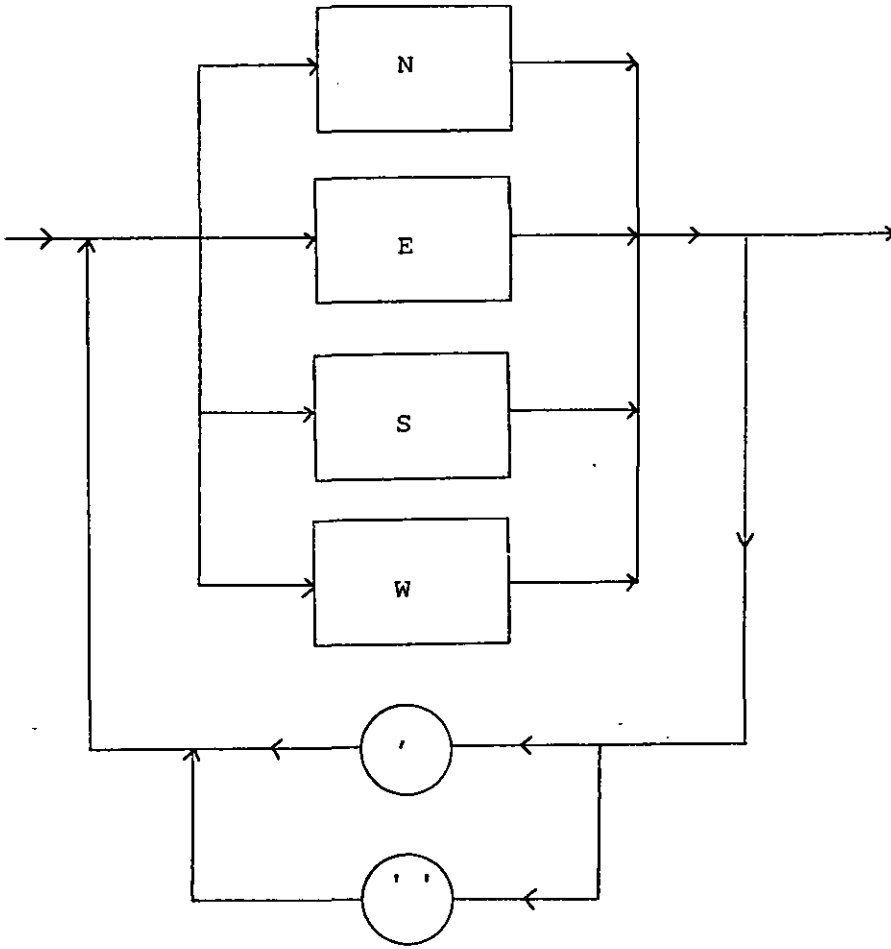
10. VALUE = INTEGER < GRIDSIZE

11. GRIDSIZE = MAXIMUM NUMBER OF COLUMNS OR ROWS OF PROCESSORS

12. DATA ITEM = REAL (BUT CAN BE EXTENDED TO OTHER MORE COMPLEX TYPES)

13. OPERATION = RESERVED (MNEMONIC) KEYWORD FOR OPERATION

14. PORTS



15. OPD1 } INTEGERS IN RANGE 0...MSIZE-1
 OPD2 } (MSIZE = SIZE OF PE PRIVATE MEMORY)

RISAL contains a proportion of semantic rules not indicated in the syntax and allows programs (instruction, data and selector files) to be produced using the same syntax and compiler. Instruction, data, or selector files can be prefixed with a replicating command which will generate the following instruction by a specified number (e.g. REP(4)), also can be prefixed with a replicating command which will generate the following lines of instruction by a specified number

(e.g. REPL(20)), checks are made to ensure that enough data, instructions or selectors are given to control the selected gridsize. The start of a file (instruction, data, or selector) must identify three things:

1. Instruction (p), data (d), and selector (s).
2. The size of the grid, the instruction and selectors can be different giving rectangular grids.
3. The number of rows in the file. This provides the OCCAM ISA with a primitive shut-down capacity and could be removed on a real machine where a reset is available. The choice of p, d or s directs the RISAL compiler to fix the syntax for a particular type of file, preventing the mixing of instructions, data, and selectors in one file, and giving useful error messages as to malformed constructions in a file (see later).

THE DATA FILE

The data file is more complex than the rest, as it requires the specification of input for the possible four boundaries of the ISA grid. The current implementation does not expose all the inherent parallelism in collecting the boundary data, as we can define four files one for each boundary, and then use the buffers in parallel, however, there is a considerable expense in checking that enough boundary data is available which requires the specification of four separate files. Here we define only one file and sequentially buffer the boundary input and output, this makes the checks easier and the setting up of a data input sequence is more easily related to the algorithms being simulated. For large grids however this method will become impractical and adding a pre-processor to the ISA, to separate out the data into temporary files

seems the best alternative. RISAL contains a certain amount of semantics to check that data boundaries are not confused, and replicators do not generate too much data so a special command NONE is also available which allows a complete boundary to be masked out. The data must always be input in order N,E,S,W (Figure 5.2), and the RISAL compiler will check this.

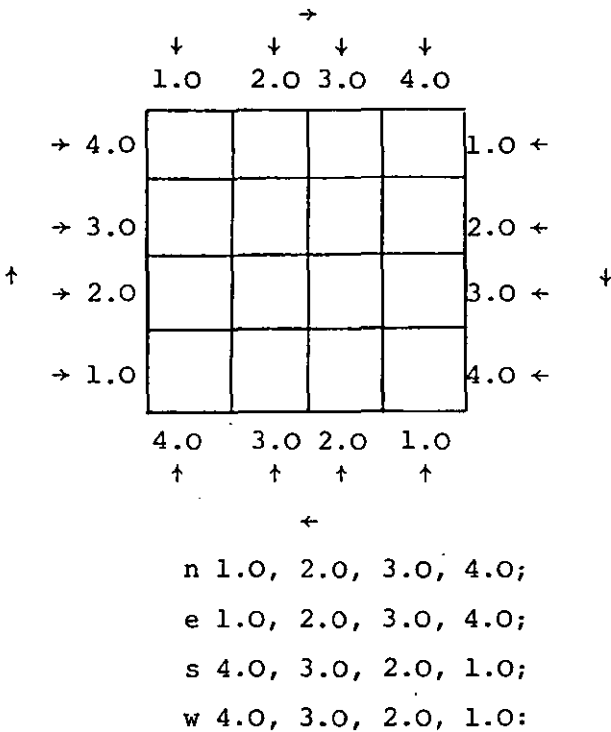
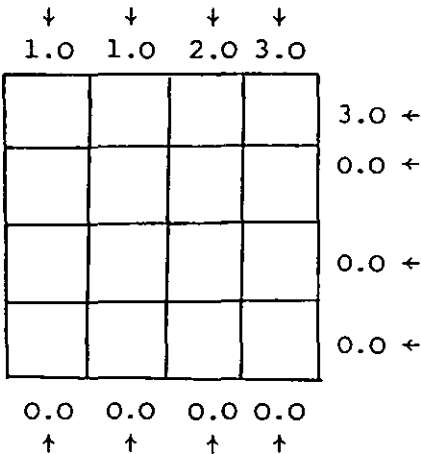


FIGURE 5.2: The Input to the ISA Boundaries

Example for Data statement:

```
n 1.0, 1.0, 2.0, 3.0;  
e 3.0, rep(3) 0.0;  
s rep(4) 0.0;  
none:
```

no data
masked out



The north boundary of the 4x4 ISA grid receives 1.0, 1.0, 2.0, 3.0 as data, while the east boundary receives 3.0 and zeros for the remaining inputs, with the south boundary inputting 4 zeros and finally the west boundary is masked out (no data).

THE SELECTOR FILE

Selectors are Boolean values and can be specified similarly, e.g. to select all the cells in column one of a 4x4 ISA grid, we would send 1, 1, 1, 1: or equivalently to rep(4)1: into the first column.

SELECTORS	→	1			
	→	1			
	→	1			
	→	1			

On the next step, if we sent 1, 1, 0, 0: or equivalently to rep(2)1; rep(2)0: into the first column the picture will be:

SELECTORS	→	1	→ 1		
	→	1	→ 1		
	→	0	→ 1		
	→	0	→ 1		

THE INSTRUCTION FILE

The instructions enter the ISA grid from the north moving across to the south row by row, and each PE in the grid is activated by a combination of an instruction and a selector. The selectors enter the grid from the west moving across to the east column by column. Below is a list of operation codes which represent all the operations occurred in the PE if the selector entered is a high signal:

```

null
data
copy
mov
add
sub
mult
div
min
max

```

An example of an instruction: DATA n, O3, OO means read the north data port and move the value into the communication register for the PE defined previously in Chapter 4.

DATA n, O3, OO; DATA n, O3, OO; DATA n, O3, OO; DATA n, O3, OO: issues the same command to 4 columns of a 4×4 grid simultaneously and is equivalent to the replicated form

```
REP(4) DATA n, O3, OO:
```

5.3 THE RISAL COMPILER

The compiler is a program written in the implementation language, accepting text in a source language and producing text in a target language. Language description languages are used to define all these languages and themselves as well. The source language is an algorithmic language to be used by programmers. The target language is suitable for execution by some particular computer. If the source and target are reasonably simple, and well matched to each other, the compiler can be short and easy to implement. The more complex the requirements become, the more elaborate the compiler must be and, the more elaborate the compiler, the higher the payoff in applying the techniques of structured programming.

Compilers can and have been written in almost every programming language, but the use of structured programming techniques is dependent upon the implementation language being able to express structure. Today there are some existing languages which were explicitly designed for the task of compiler writing. The criterion for choosing an implementation language is quite straight forward, it should minimize the implementation effort and maximize the performance of the compiler being written [Alfred, Jeffrey, 1977].

Each compiler is developed in a particular environment in response to certain needs, and that the environment will shape not only the form of the completed compiler, but also the process by which it is developed. This brings into discussion the nature of the target machine, which in our case is the specially designed virtual machine. The PASCAL language was used to develop and test the compiler whose job was to read the Replicated Instruction Systolic Array Language elements (RISAL)

and transform it into a form suitable for the virtual machine to run. The general phases of the RISAL compiler are shown in Figure 5.3.

Explicit goals should be formulated at the outset of any compiler development, although they may change with time, they provide guidelines for the major decisions and are the basis for evaluation. The typical compiler goals:

- correctness, it should give the correct outputs for each possible input. This is what we mean by a program 'works'. If a program does not work, measures of efficiency, of adaptability, or production costs have no meaning. One goal of every compiler is to correctly translate all correct input programs and correctly diagnose all incorrect ones. However, compilers are seldom absolutely correct: perhaps "reliability" is a more feasible goal, i.e. keeping the number of errors encountered acceptably small.
- availability, even a correct compiler that cannot be run is of little use. Thus, a very important aspect of any compiler development is its schedule and it must run on the right machine in the right configuration with the right operating system.
- generality and adaptability, although some special-purpose compilers are produced to compile single programs, most compilers must be planned to handle a large number of programs. During the life-time of a compiler, requirements and specifications may change many times (often, even before it is completed). Unless special care is taken during its

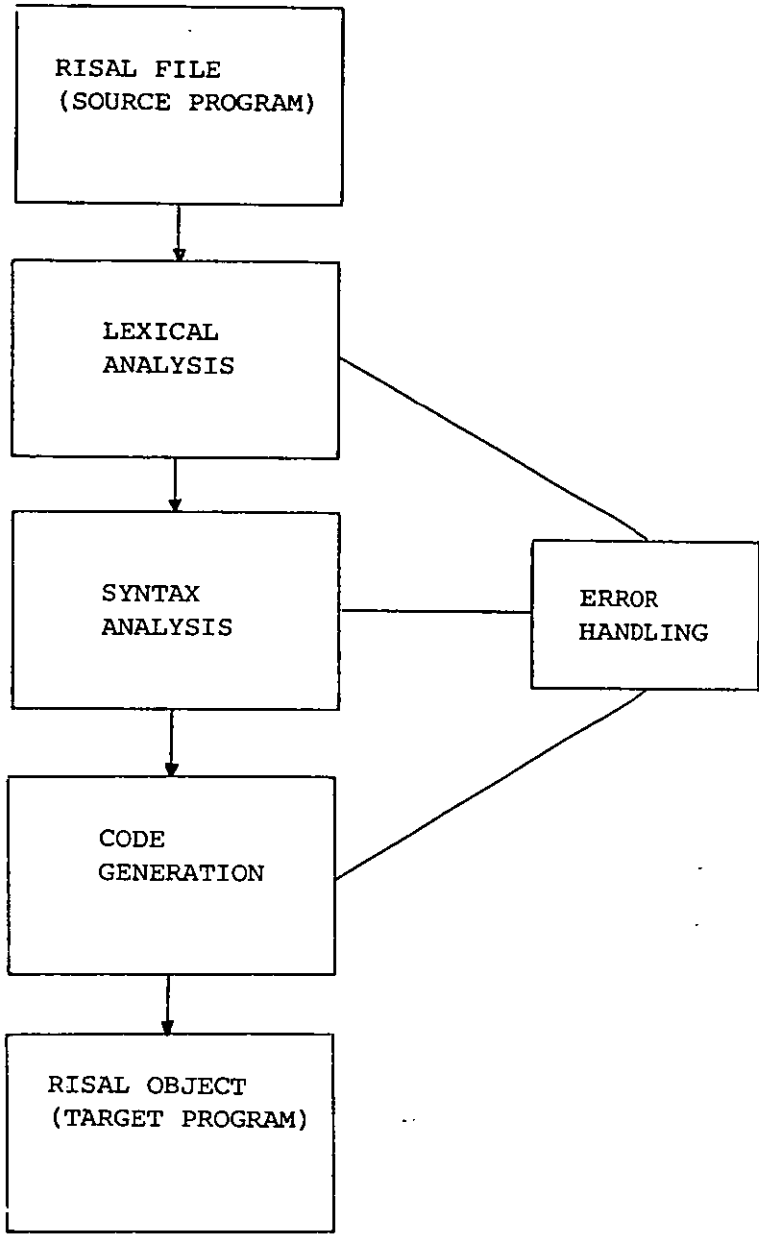


FIGURE 5.3: Phases of RISAL Compiler

construction to ensure adaptability, responding to these changes may be both traumatic and expensive.

- helpfulness, the kind and amount of help that is most appropriate will depend on the intended users: beginners need careful explanations of simple errors in small programs, while system programmers are more concerned with the detection of subtle errors in large programs, or the location of efficiency "bottlenecks".
- efficiency, there are several dimensions of efficiency to be taken into account: efficiency of compiler development process, efficiency of program development using the compiler (including efficiency of compilation), efficiency of target programs produced by the compiler.

To develop the RISAL compiler, it is not intended here, nor appropriate, to demonstrate state-of-the-art techniques in compiler writing for parallel processing, but rather to provide a practical way of how the virtual machine would handle a subset of possible operations to solve various algorithms. The construction of the RISAL compiler involves several conceptually distinct processes.

SPECIFICATION

With our design aims chosen then to solve the problem of generating the three object files (INSTRUCT, DATA, and SELECTOR) which are used to control the performance of the virtual machine (ISA grid), the RISAL compiler specification document includes:

- a precise specification of the source language (Section 5.2).
- design target for the compiler size.

- a choice of the language in which the RISAL compiler is to be written (PASCAL language).

DESIGN

The design of the RISAL compiler was started before the specification and continued well into the implementation phase. The RISAL compiler was structured into major components (procedures, modules), and we allocated functions and responsibilities amongst them, and the definition of their interfaces.

IMPLEMENTATION

Regardless of the design technique used, at some point the RISAL compiler must be written in an already implemented language translated into machine code and executed. As we mentioned above the PASCAL language was chosen to write the RISAL compiler, because it is easily readable and understandable, with appropriate data objects, in addition it has a simple yet powerful control and data structure, with enough redundancy for substantial compile-time checking.

The RISAL compilation process is partitioned into a series of subprocesses called procedures as shown in Figure 5.4.

In the initialisation procedure the process of setting up the current keywords are assigned and initialise the values. The size of the keywords file is 20 and contains the following keywords:

<u>Keyword</u>	<u>Code representation</u>
add	2
copy	1
d (data file)	105
data	8

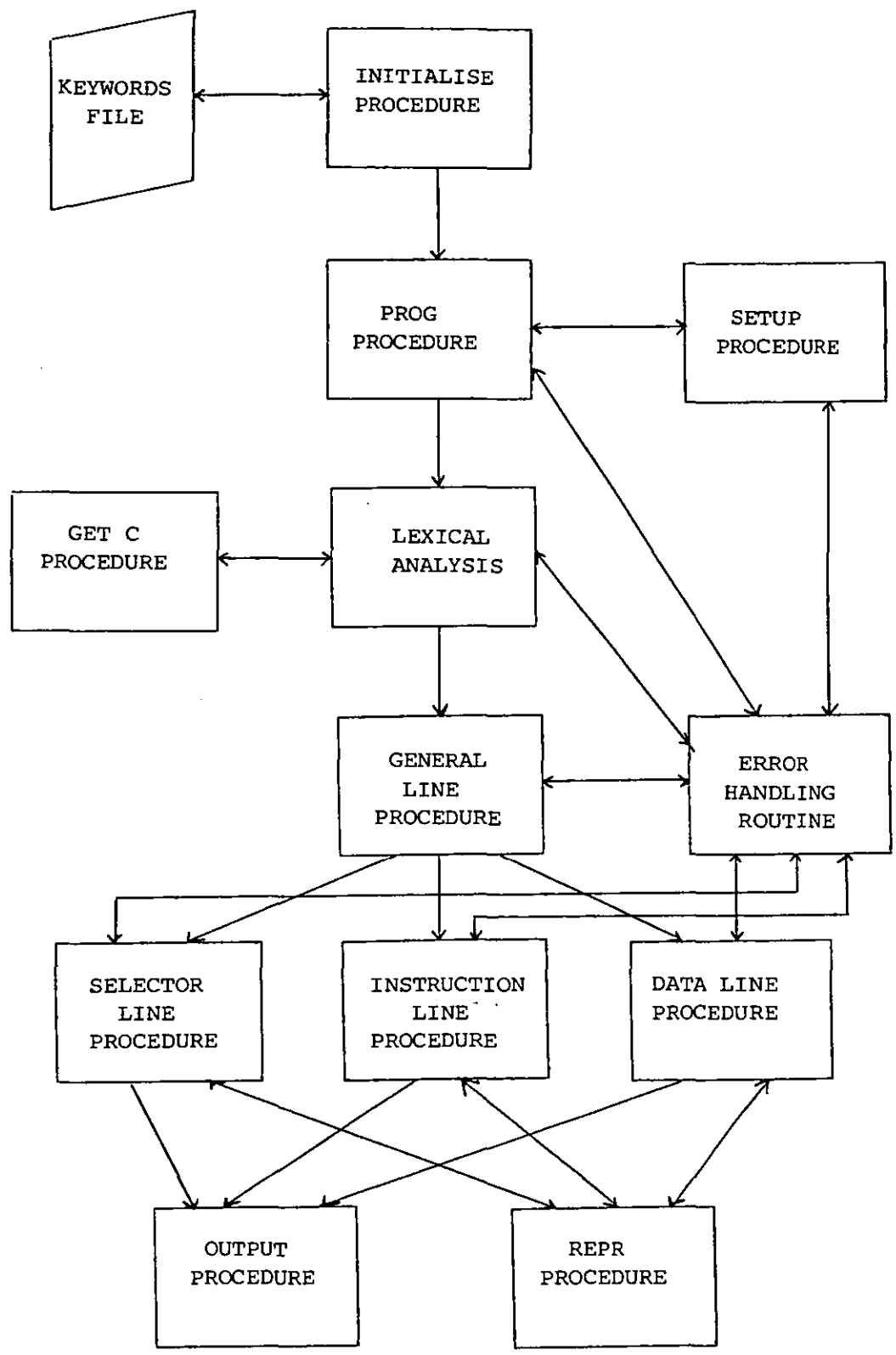


FIGURE 5.4: The RISAL Compilation Process

<u>Keyword</u>	<u>Code representation</u>
div	5
e (east)	2
end	101
max	7
min	6
mov	9
mult	4
n (north)	1
none	102
null	0
p (program)	104
rep	103
repl	106
s (south)	4
sub	3
w (west)	8

The prog procedure is to decide the input file type, whether Instruction, Data, or Selector file, and to process this input file. The checking of the dimension of the ISA grid will occur in this procedure. The input file must start with header T (value1,value2). This means,

T type of input file

T=p for instruction

T=d for data

T=s for selector

value1 the dimension of the ISA grid (e.g. value 1=4 in the case of 4×4 ISA grid

value2 the number of lines ended by (:) in the input file

The setup procedure is to decipher the input file header. The lexical analyzer is the interface between the RISAL source program and the RISAL compiler. The lexical analyzer reads the source program one character at a time, carving the source program into a sequence of atomic units called tokens.

A classical lexical analysis was used to develop the scanning and screening functionality for reading the RISAL statements. Early in the compilation process the source file (RISAL statements) appear as a stream of characters. By scanning them finds substrings of characters that constitute the textual elements (words, punctuation, operators, comments, spaces, etc.) and classifies each as to which sort of textual element it is. The screening process discards some of the textual elements (spaces, comments, etc.) while recognizing reserved symbols and generating the token stream for parsing. Below we summarize the scanning taken by the lexical analyzer of the RISAL compiler when processing a RISAL statement:

- make sure there is a token and it can be recognized.
- skipping leading blanks.
- skipping comments - all comments starting with { .
- skipping trailing blanks
- find token
- collect identifier
- searching for keywords and locate them
- convert to token value
- convert to a number
- convert all the integers to a numeric value, and the reals remain as strings.

The purpose of the `getc` procedure is to maintain a buffer of characters, keep the buffer filled and to skip the blank spaces. In addition, book keeping will occur in this procedure.

The line procedure is to process a general line, checking will be made first to decide whether the line is from the instruction, data or selector. There are three separate procedures to implement the instruction, data and selector line.

The output procedure is to construct the `instruct`, `datain`, or selector object files.

Finally, the error handling routine is to print out error messages in output file called `ERROR` file in case of any fault in the RISAL program. The RISAL compiler attempts to detect and report as many errors as possible. Below are the following error messages provided by the RISAL compiler:

- program must start with p,d or s
- expected (but found)
- expected) but found (
- expected :, , , ;, end,]
- too many data elements
- incorrect data boundary spec
- expected integer arguments
- errors detected =
- no errors detected
- expected integer operands in instruction
- should be real value in data expression
- require integer in rep count parameter
- attempt to read past end of file

- alphabetic string found require keyword
- invalid character
- selector should be 0 or 1
- malformed expression

OBJECT FILES

We now turn to the code generation routine, the final phase of the compilation process. Good code generation is difficult, and it depends on the construction of the virtual machine we are using. We initially developed a straightforward algorithm to generate code from a sequence of statements. The algorithm was used successfully to produce an ISA form placed in three files (INSTRUCT, DATAIN, and SELECTOR). To show the picture of generating the ISA form, below is an example of three input RISAL files to calculate the value of $x = (A+B) * (C-D) / E$, and the picture of the ISA form after we compile them by the RISAL compiler (INSTRUCTION, DATAIN, SELECTOR).

INSTRUCTION FILE

```
p(4,16)
data n,3,0; rep(3) null n,0,0:
mov ,0,7; data n,3,0; rep(2) null n,0,0:
data n,3,0; mov ,0,7; data n,3,0; null n,0,0:
mov ,0,8; data n,3,0; mov ,0,7; null n,0,0:
add ,7,8; mov ,0,8; rep(2) null ,0,0:
copy ,0,0; sub ,7,8; rep(2) null ,0,0:
null ,0,0; mov ,0,7; rep(2) null ,0,0:
null ,0,0; data w,6,0; rep(2) null ,0,0:
null ,0,0; mov ,0,8; rep(2) null ,0,0:
null ,0,0; mult ,7,8; rep(2) null ,0,0:
null ,0,0; copy ,0,0; rep(2) null ,0,0:
rep(2) null ,0,0; data w,6,0; null ,0,0:
rep(2) null ,0,0; mov ,0,8; null ,0,0:
```



```

rep(2) null ,0,0; div ,8,7; null ,0,0:
rep(2) null ,0,0; copy ,0,0; null ,0,0:
rep(4) null ,0,0

```

```
end
```

```
SELECTOR FILE
```

```
s(4,16)
```

```
repl(16) [1,rep(3)0]
```

```
end
```

```
DATA FILE
```

```
d(4,16)
```

```
n 4.0, 0.0, 0.0, 0.0; none; none; none:
```

```
n 0.0, 5.0, 0.0, 0.0; none; none; none:
```

```
n 6.0, 0.0, 0.0, 0.0; none; none; none:
```

```
n 0.0, 3.0, 0.0, 0.0; none; none; none:
```

```
repl(12) [rep(4)none]
```

```
end
```

```
INSTRUCT
```

```
16
```

08010300	10000	10000	10000
09000007	08010300	0	0
08010300	09000007	08010300	0
09000008	08010300	09000007	0
02000708	09000008	0	0
01000000	03000708	0	0
0	09000007	0	0
0	08080600	0	0
0	09000008	0	0
0	04000708	0	0
0	01000000	0	0
0	0	08080600	0
0	0	09000008	0
0	0	05000807	0
0	0	01000000	0
0	0	0	0

SELECTOR

16

1	0	0	0	0
1	0	0	0	0
1	0	0	0	0
1	0	0	0	0
1	0	0	0	0
1	0	0	0	0
1	0	0	0	0
1	0	0	0	0
1	0	0	0	0
1	0	0	0	0
1	0	0	0	0
1	0	0	0	0
1	0	0	0	0
1	0	0	0	0
1	0	0	0	0
1	0	0	0	0

DATAIN

16

0	4.0	0.0	0.0	0.0
-1				
-1				
-1				
0	0.0	5.0	0.0	0.0
-1				
-1				
-1				
0	6.0	0.0	10.0	0.0
-1				
-1				
-1				
0	0.0	3.0	0.0	0.0
-1				
-1				
:				
:	to the end of 16 lines			

The complete actual code for the RISAL compiler is given in Appendix II.

5.4 SOFT-SYSTOLIC SIMULATION ARCHITECTURE AND TESTING

From the previous sections in Chapter 4 and this chapter a number of components are readily identified which need connecting: the RISAL source files, the RISAL compiler, the compiled object files, the virtual machine (the ISA grid, processing element, and the virtual spoolers), and the resulting dataout file.

The components mentioned above are serially linked together as shown in Figure 5.5. The virtual spooling section of the ISA grid, expects to find instructions in a file called 'INSTRUCT', with selector information in a file called "SELECTOR" and data from file "DATAIN". The RISAL compiler allows the output of generated ISA items to be directed to any of these files or temporary files as requested by the user using the DYNIX file in direction commands. It is up to the user to ensure that the spoolers have the correct data and program = (instruction, selector).

A typical program specification is as follows:

- i) Develop three files
 - I1 = instructions
 - D1 = data
 - S1 = selectors
- ii) Check syntax with RISAL compiler, generating the files INSTRUCT, SELECTOR, DATAIN.
- iii) All bugs are now semantic errors in the ISA program
 - Compile ISA.OCC (virtual) grid if not compiled
 - Compile PE.OCC processing element
 - Link the two above programs (plugs in PE)
- iv) Execute the virtual machine in (iii), the results will be placed in the "DATAOUT" file.

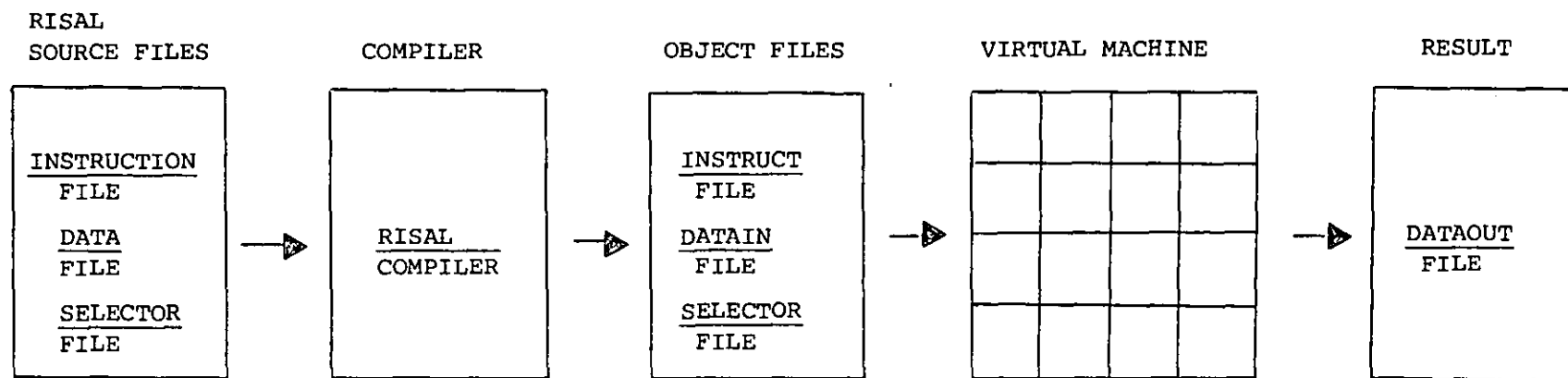


FIGURE 5.5: Soft-Systolic Simulation Architecture

This is how the ISA.OCC virtual ISA can be used as a simulation architecture to solve soft systolic algorithms, the process is quite simple and requires only the RISAL source files.

SAMPLE PROGRAMS

To examine the performance of the solution architecture, we will illustrate first the use of each operation code mentioned previously in the instruction file.

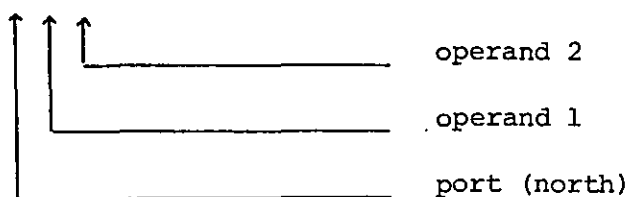
1. NULL operation code

e.g. null ,0,0;

means that there is no operation, even if there is a selector high signal.

2. DATA operation code

e.g. data n,3,0;



meaning read the north data port and move the value into the communication register C, for the PE defined previously.

data n,3,0; data n,3,0; data n,3,0; data n,3,0;

this would issue the same instruction for 4 cells of the 4*4 grid and is equivalent to the replicated form:

rep(4) data n,3,0;

Example 1

To write a RISAL program to read the data 10,20,30,40 from the north data port and move the value into the communication register (C) for the 4 cells in the first row of the 4*4 grid, and let the data

move across the grid row by row

```
p(4,7)
rep(4) null ,0,0:
rep(4) null ,0,0:
rep(4) null ,0,0:
rep(4) data n,3,0:
rep(4) null ,0,0:
rep(4) null ,0,0:
rep(4) null ,0,0
end
```

this is equivalent to:

```
p(4,7)
repl(3)[rep(4) null ,0,0]:
rep(4) data n,3,0:
repl(3)[rep(4) null ,0,0]
end
```

```
s(4,7)
1,rep(3),0:
rep(2)1,rep(2)0:
rep(3)1,0:
rep(4) 1:
0,rep(3)1:
rep(2)0,rep(2)1:
rep(3)0,1
end
```

```
d(4,7)
repl(3)[rep(4)none]:
n 10.0, 20.0, 30.0, 40.0;
none; none; none:
repl(3)[rep(4) none]
end
```

INSTRUCTIONS

null	null	null	null
null	null	null	null
null	null	null	null
datan	datan	datan	datan
null	null	null	null
null	null	null	null
null	null	null	null

SELECTORS

0	0	0	1	1	1	1
0	0	1	1	1	1	0
0	1	1	1	1	0	0
1	1	1	1	0	0	0

↓

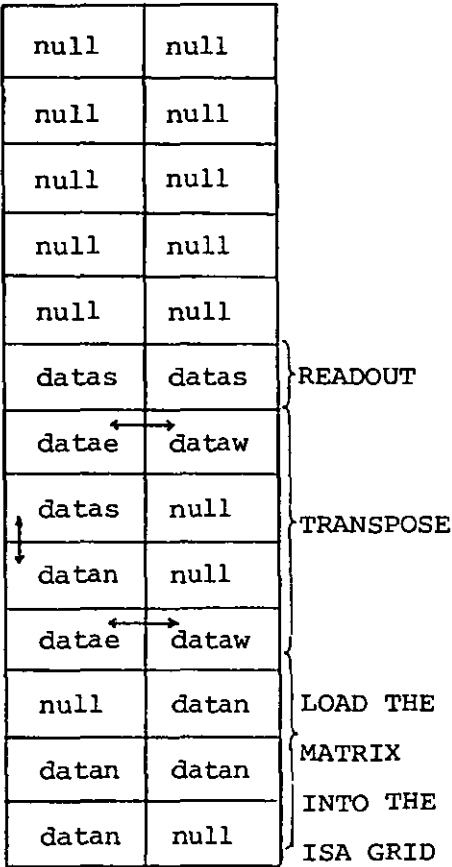
4*4 ISA grid

Example 2

2*2 matrix transpose (see the definitions in Section 6.1).

Transpose the following matrix,

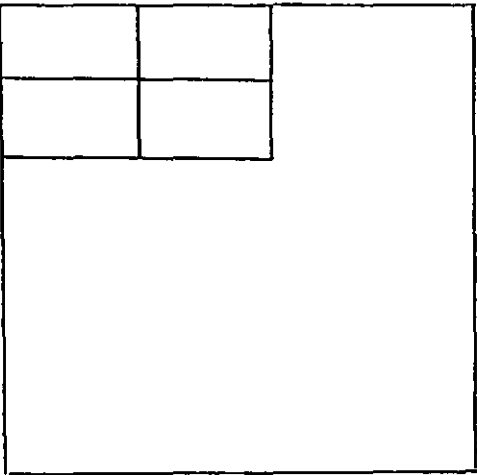
$$\begin{bmatrix} 8 & 5 \\ 6 & 2 \end{bmatrix} \rightarrow \begin{bmatrix} 8 & 6 \\ 5 & 2 \end{bmatrix}$$



SELECTORS

0	1	1	1	1	0	1	1	0	1	1	1	1
0	0	0	0	1	0	0	1	0	0	0	1	0

```
p(4,13)
{load matrix}
data n,3,0; rep(3) null n,0,0:
rep(2) data n,3,0; rep(2) null n,0,0:
null n,0,0; data n,3,0; rep(2) null ,0,0:
{Transpose}
data e,4,0; data w,6,0; rep(2) null ,0,0:
```



4*4 ISA grid

```

data n,3,0; rep(3) null ,0,0:
data s,5,0; rep(3) null ,0,0:
data e,4,0; data w,6,0; rep(2) null ,0,0:
{readout}
rep(2) data s,5,0; rep(2) null ,0,6:
repl(5) [rep(4) null ,0,0]
end

```

```

d(4,13)
n 6.0, 0.0, 0.0, 0.0; none; none; none:
n 8.0, 2.0, 0.0, 0.0; none; none; none:
n 0.0, 5.0, 0.0, 0.0; none; none; none:
repl(10) [rep(4) none]
end

```

```

s(4,13)
1, rep(3)0:
rep(2)1, rep(2)0:
1,rep(3)0:
1,rep(3)0:
rep(4)0:
rep(2) 1,rep(2)0:
1, rep(3)0:
rep(4)0:
rep(2)1,rep(2)0:
repl(3) [1,rep(3)0]:
rep(4)0
end

```

3. COPY operation code

e.g. copy ,0,0;

means copying the value from the result register (R) after the computation has been made to the communication register (C).

When the store is in the communication register, this means that the value is ready to be read by the neighbouring cell.

4. MIN operation code

e.g. min e,4,1;

The minimum operation code above means read data from the east port data and compare it with the value in the register east data

addressed by operand 1 and put the minimum value in the communication register (C) addressed by operand 2.

5. MAX operation code

e.g. max w,6,1;

The maximum operation code above means read data from the west port data and compare it with the value in the register west data addressed by operand 1, and put the maximum value in the communication register (C) addressed by operand 2.

The operation codes minimum and maximum could form the basis for a comparison based cell in their own right, possibly augmented with EQ (equals) and so provide a simpler PE for sorting, and searching ISA algorithms.

Example 3: Sorting a list of 4 numbers using 4*4 ISA grid

```
p(4,13)
repl(3)[rep(4) null n,0,0]:
rep(4) data n,3,0:
min e,4,1; max w,6,1; min e,4,1; max w,6,1:
rep(4) copy ,0,0:
null ,0,0; min e,4,1; max w,6,1; null ,0,0:
null ,0,0; rep(2) copy ,0,0; null ,0,0:
min e,4,1; max w,6,1; min e,4,1; max w,6,1:
rep(4) copy ,0,0:
null ,0,0; min e,4,1; max w,6,1; null ,0,0:
null ,0,0; rep(2) copy ,0,0; null ,0,0:
rep(4) null ,0,0
end
```

```
s(4,13)
rep(13)[1,rep(3)0]
end
```

```
d(4,13)
rep(3)[rep(4) none]:
n 4.0,3.0,2.0,1.0; none; none; nine:
repl(9)[rep(4) none]
end
```

INSTRUCTIONS

1 null	2 null	3 null	4 null
null	copy	copy	null
1 null	3 mine ← 2 maxw		4 null
copy	copy	copy	copy
3 mine ← 1 maxw		4 mine ← 2 maxw	
null	copy	copy	null
3 null	4 mine ← 1 maxw		2 null
copy	copy	copy	copy
4 mine ← 3 maxw		2 mine ← 1 maxw	
datan	datan	datan	datan
null	null	null	null
null	null	null	null
null	null	null	null

LOAD
NUMBERS

SELECTORS

1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0

→

4*4 ISA grid

6. MOV operation code

e.g. `mov ,0,7;`

The mov operation code above means move the data in the result register addressed by operand 1 and put it in the auxiliary memory of the PE which is addressed by operand 2.

We can use the mov operation code to move the data from any register or auxiliary memory to any register in the memory organization of the PE defined previously by giving the right addresses in operand 1 and operand 2. -

Example 4: Data array movement

To write a RISAL program to read data from the north port data for the first row in the 4*4 ISA grid and move the data to the auxiliary memory of these cells, and re-read them again to the north.

```
p(4,9)
repl(3) [rep(4) null n,0,0]:
rep(4) data n,3,0:
rep(4) mov ,1,7:
rep(4) mov ,7,0:
rep(4) copy ,0,0:
rep(4) data s,5,0:
rep(4) null ,0,0
end

s(4,9)
repl(9) [1,rep(3)0]
end

d(4,9)
repl(3) [rep(4) none]:
n 2.0,4.0,6.0,8.0; none; none; none:
repl(5) [rep(4) none]
end
```

7. ADD operation code

e.g. `add ,7,0;`

The operation code add above means add the value in the auxiliary

memory addressed by operand 1 to the value in the result register addressed by operand 2, and the result will be held in the result register (R).

We can use this operation code to add the value in any two registers in the memory organization of the PE, and the result will be held in the result register (R).

Example 5: Summation calculation

To write a RISAL program to add data from the north port data and the west port data and add them with another set of data from the same ports for the first cell of the first row of 4*4 ISA grid.

```
p(4,6)
add n w ,3,6; rep(3) null ,0,0:
mov ,0,7; rep(3) null ,0,0:
add n w ,3,6; rep(3) null ,0,0:
add ,0,7; rep(3) null ,0,0:
copy ,0,0; rep(3) null ,0,0:
rep(4) null ,0,0
end

s(4,6)
repl(6) [1,rep(3)0]
end

d(4,6)
n 10.0,0.0,0.0,0.0; none; none; w 4.0,0.0,0.0,0.0:
rep(4) none:
n 20.0,0.0,0.0,0.0; none; none; w 7.0,0.0,0.0,0.0:
repl(3) [rep(4) none]
end
```

The result is 41 which is placed in dataout file after reading it from the result register.

8. SUB operation code

e.g. sub ,7,0;

The operation code sub above means subtract the value in the

result register addressed by operand 2 from the value in the auxiliary memory addressed by operand 1, and keep the result in the result register. We can use this operation code to subtract the value in any register in the memory organization from any value in another register and the result of the subtraction will be held in the result register (R).

Example 6:

To write a RISAL program to add data from the north port data and the west port data and subtract them from the addition of another set of data which reads from the same ports of the first cell of the first row of the 4*4 ISA grid.

```
p(4,6)
add n w ,3,6; rep(3) null ,0,0:
mov ,0,7; rep(3) null ,0,0:
add n w ,3,6; rep(3) null ,0,0:
sub ,0,7; rep(3) null ,0,0:
copy ,0,0; rep(3) null ,0,0:
rep(4) null ,0,0
end
```

```
s(4,6)
repl(6) [1, rep(3) 0]
end
```

```
d(4,6)
n 10,0,0,0,0,0,0,0; none; none; w 4,0,0,0,0,0,0,0:
rep(4) none:
n 20,0,0,0,0,0,0,0; none; none: w 7,0,0,0,0,0,0,0:
repl(3) [rep(4) none]
end
```

The result is 13 which is placed in dataout file after reading it from the result register.

9. MULT operation code

e.g. mult ,0,7;

The operation code (mult) above means multiply the value in the

result register addressed by operand 1, by the value in the auxiliary memory of the PE addressed by operand 2, and the result will be held in the result register.

We can use the operation code (mult) to multiply the value in any register in the memory organization by the value in another register and the result will be held in the result register (R).

The mult operation is used also to read the data from two different ports, multiply them and hold the result in the result register (R).

Example 7: Multiplication of data

To write a RISAL program to read two sets of data from the north port data and multiply them, and read the result

```
p(4,9)
relp(3) [rep(4) null n,O,O]:
rep(4) data n,3,O:
rep(4) mov ,O,7:
rep(4) data n,3,O:
rep(4) mult ,O,7:
rep(4) copy ,O,O:
rep(4) null ,O,O
end

s(4,9)
repl(9) [1,rep(3)O]
end

d(4,9)
repl(3) [rep(4) none]:
n 2.0,4.0,6.0,8.0; none; none; none:
rep(4) none:
n 3.0,5.0,7.0,9.0; none; none; none:
repl(3) [rep(4) none]
end
```

The result is 6.0, 20.0, 42.0, 72.0 which is placed in the dataout file after reading all these values from the result register.

Example 8: Inner product calculation

To write a RISAL program to read from the north port data and the west port data for the first cell in the first row of 4*4 ISA grid, multiply them and store the result in the auxiliary memory, repeat this process and add the results of the multiplication and read the result.

```

p(4,6)
mult n w ,3,6; rep(3) null ,0,0:
mov ,0,7; rep(3) null ,0,0:
mult n w ,3,6; rep(3) null ,0,0:
add ,0,7; rep(3) null ,0,0:
copy ,0,0; rep(3) null ,0,0:
rep(4) null ,0,0
end

s(4,6)
repl(6) [1,rep(3)0]
end

d(4,6)
n 2.0,0.0,0.0,0.0; none; none; w 4.0,0.0,0.0,0.0:
rep(4) none:
n 3.0,0.0,0.0,0.0; none; none; w 5.0,0.0,0.0,0.0:
repl(3) [rep(4) none]
end

```

The result is 23 which is placed in the dataout file.

10. DIV operation code

e.g. div ,7,3;

The operation code div above means divide the value in the auxiliary memory of the PE addressed by operand 1 by the value in the register north data addressed by operand 2, and the result to be held in the result register (R). Also we can use the operation code (div), to divide the value of any register in the memory organization of the PE by the value in another register, and the result will be held in the result register (R).

Example 9: Division of two numbers

To write a RISAL program to add two numbers read from the north and west port data, store the result into the auxiliary memory and add another two numbers which is read from the same ports and divide them by the data stored in the memory and read the result.

```
p(4,6)
add n w ,3,6; rep(3) null ,0,0:
mov ,0,7; rep(3) null ,0,0:
add n w ,3,6; rep(3) null ,0,0:
div ,0,7; rep(3) null ,0,0:
copy ,0,0; rep(3) null ,0,0:
rep(4) null ,0,0
end
```

```
s(4,6)
repl(6) [1,rep(3)0]
end
```

```
d(4,6)
n 5.0,0.0,0.0,0.0; none; none; w 2.0,0.0,0.0,0.0:
rep(4) none:
n 10.0,0.0,0.0,0.0; none; none; w 11.0,0.0,0.0,0.0:
repl(3) [rep(4) none]
end
```

The result is 3 which is placed in the dataout file.

CHAPTER 6

THE SOFT-SYSTOLIC SIMULATION SYSTEM (SSSS)

FOR VARIOUS ALGORITHMS

6.1 BASIC MATHEMATICS

In this section some basic mathematical definitions and concepts are given. The material presented is necessary for the description of algorithms used later in this chapter. First of all, vectors and matrices are defined together with some relevant properties and relations. These are then used to discuss methods for solving linear systems, matrix-vector multiplication, matrix-matrix multiplication, matrix transpose and LU decomposition. The generalized matrix inversion is defined next, and then the Soft-Systolic Simulation (SSSS) is used to solve all these problems.

Matrices and Vectors:

Matrices are important to numerical algorithms because they provide a concise method for specifying manipulating large numbers of linear equations. A system of m linear equations in n unknown has the general form,

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n &= b_2 \\ \dots & \\ a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n &= b_m \end{aligned} \tag{6.1.1}$$

The coefficients of the (6.1.1) above form a matrix, which we denote A or (a_{ij}) of order $m \times n$, where $(i=1, \dots, m; j=1, \dots, n)$, and b_i ($i=1, \dots, m$) are given numbers. If A is an $n \times n$ matrix, that means A is a square matrix of order n . If the matrix has only one column or only one row, the matrix is called column vector, or row vector,

$$b = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_m \end{bmatrix} \quad x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} \quad (6.1.2)$$

We say that b is an m -vector, and x is an n -vector. If $A=(a_{ij})$ and $B=(b_{ij})$ are both matrices, then we say that A equals B , or $A=B$, provided A and B have the same order and $a_{ij}=b_{ij}$, all i and j . In the terminology so far introduced, (6.1.1) states that the matrix A combined in a certain way with the one-column matrix, or vector, x should equal the one-column matrix, or vector, b .

The process of combining matrices involved here is called matrix multiplication and is defined in general, as follows:

Let $A=(a_{ij})$ be an $m \times n$ matrix, $B=(b_{ij})$ an $n \times p$ matrix; then matrix $C=(c_{ij})$ is the (matrix) product of A with B (in that order), or $C=AB$, provided C is of order $m \times p$ and,

$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}, \text{ for } i=1, \dots, m; j=1, \dots, p \quad (6.1.3)$$

In other words, the (i,j) entry of the product $C=AB$ of A with B is calculated by taking the n entries of row i of A and the n entries of column j of B , multiplying corresponding entries, and summing the resulting n products.

With this definition of matrix product and definitions in (6.1.1) and (6.1.2), we can write our system of equations (6.1.1) simply as:

$$Ax = b. \quad (6.1.4)$$

Matrix multiplication does not behave like multiplication of numbers, for example, it is possible to form the product of the matrix A with

the matrix B only when the number of columns of A equals the number of rows of B . Hence, even when the product AB is defined, the product of B with A need not be defined. Further, even when both AB and BA are defined, they need not be equal.

If $A = (a_{ij})$ is a square matrix of order n , then we call its entries $a_{11}, a_{22}, \dots, a_{nn}$ the diagonal entries of A , and call all other entries off-diagonal. All entries a_{ij} of A with $i < j$ are called superdiagonal, all entries a_{ij} with $i > j$ are called subdiagonal. If all off-diagonal entries of the square matrix A are zero, we call A a diagonal matrix. If all subdiagonal entries of the square matrix A are zero, we call A an upper (or right) triangular matrix, while if all superdiagonal entries of A are zero, then A is called lower (or left) triangular. Clearly, a matrix is diagonal if and only if it is both upper and lower triangular.

If a diagonal matrix of order n has all its diagonal entries equal to 1, then we call it the Identity Matrix of order n and denote it by I or I_n if the order is important. The name identity matrix was chosen for this matrix because:

$$I_n A = A \text{ for all } n \times p \text{ matrices } A$$

$$B I_n = B \text{ for all } m \times n \text{ matrices } B$$

i.e., the matrix I acts just like the number 1 in ordinary multiplication.

Inversion and Generalized Inversion of Matrices:

Division of matrices is, in general, not defined. However, for square matrices, we define a related concept, matrix inversion. We say that the square matrix A of order n is invertible provided there is a square matrix B of order n such that:

$$AB = I = BA \quad (6.1.5)$$

The matrix $A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$, for instance, is invertible since,

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

On the other hand, the matrix $A = \begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix}$ is not invertible. For if B were a matrix such that $BA=I$, then it would follow that:

$$\begin{bmatrix} b_{11}+2b_{12} & 2b_{11}+4b_{12} \\ b_{21}+2b_{22} & 2b_{21}+4b_{22} \end{bmatrix} = BA = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Hence we should have $b_{11}+2b_{12}=1$ and, at the same time, $2(b_{11}+2b_{12})=2b_{11}+4b_{12}=0$, which is impossible. We note that (6.1.5) can hold for at most one matrix B . For if,

$$AB = I, \text{ and } CA = I,$$

where B and C are square matrices of the same order as A , then,

$$C = CI = C(AB) = (CA)B = IB = B$$

showing that B and C must then be equal. Hence, if A is invertible, then there exists exactly one matrix satisfying (6.1.5). This matrix is called the inverse of A and is denoted by A^{-1} .

It follows at once from (6.1.5) that if A is invertible, then so is A^{-1} , and its inverse is A ; that is

$$(A^{-1})^{-1} = A. \quad (6.1.6)$$

Further, if both A and B are invertible square matrices of the same order, then their product is invertible and

$$(AB)^{-1} = B^{-1}A^{-1}. \quad (6.1.7)$$

It is well known that every non-singular real or complex (square) matrix A has a unique inverse which has the property that,

$$AA^{-1} = A^{-1}A = I . \quad (6.1.8)$$

This guarantees that the system of linear equations $Ax=b$ has the unique solution,

$$x = A^{-1}b . \quad (6.1.9)$$

A matrix has an inverse only if it is square, and a square matrix A has an inverse if and only if it is nonsingular, that is, if and only if,

- (i) $\det A \neq 0$, or
- (ii) the columns of A are linearly independent, or
- (iii) the rows of A are linearly independent,

where each of these three properties implies the other two.

If a matrix, rectangular or square is singular, it does not have an inverse. However it does have a generalized inverse, called a g -inverse, which has the following properties:

- (i) a g -inverse exists for a class of matrices larger than the class of nonsingular matrices,
- (ii) a g -inverse has some of the properties of the ordinary matrix inverse, and
- (iii) a g -inverse reduces to the ordinary matrix inverse, if A is square and nonsingular.

If A is an $m \times n$ matrix and G is a g -inverse of A , then G is an $n \times m$ matrix defined as follows:

DEFINITION: Consider the matrix equations:

- (a) $AGA = A$

- (b) $GAG = G$
- (c) $(AG)^H = AG$
- (d) $(GA)^H = GA$,

where the subscript H denotes the complex conjugate transpose. The matrix G is called:

- (i) a g-inverse of A, denoted by A^{-1} , if (a) holds,
- (ii) a reflexive g-inverse of A, denoted by A_R^- , if both (a) and (b) hold,
- (iii) a least-squares g-inverse of A, denoted by A_L^- , if both (a) and (c) hold,
- (iv) a minimum-norm g-inverse of A, denoted by A_M^- , if both (a) and (d) hold, and
- (v) the Moore-Penrose g-inverse of A, denoted by A^+ , if (a), (b) and (d) all hold.

Matrix Transpose:

There is an operation on matrices which has no parallel in ordinary arithmetic, the formation of the transposed matrix. If $A=(a_{ij})$ and $B=(b_{ij})$ are matrices, we say that B is the transpose of A, or $B=A^T$, provided B has as many rows as A has columns and

$$b_{ij} = a_{ji} \text{ all } i \text{ and } j .$$

In other words, one forms the transpose A^T of A by

"reflecting A across the diagonal"

If $A^T = A$

then A is said to be symmetric.

One easily verifies the following rules regarding transposition:

- (i) If A and B are matrices such that AB is defined, then $B^T A^T$ is defined and $(AB)^T = B^T A^T$.
- (ii) For any matrix A , $(A^T)^T = A$.
- (iii) If the matrix A is invertible, then so is A^T , and $(A^T)^{-1} = (A^{-1})^T$.

Solution of Linear System:

Consider the linear system,

$$Ax = b ,$$

where A is a square ($n \times n$) matrix, b is a given right hand side vector, and x is an unknown vector. It will be assumed that A is non-singular, hence A^{-1} exists and there is a unique solution \underline{x} . The choice of solution method depends on a number of factors including the structure and size of the matrix A , the number of arithmetic operations required, and the control of the rounding error growth (or stability). There are two general classes of methods, direct and iterative methods. As regards the matrix size and structure, direct methods, are used mainly when the matrix A is small, dense or banded. Direct methods cannot, in general, be used for large sparse matrices because of the problem of fill-ins which occurs during the elimination process. For large sparse matrices we normally use the iterative methods since these will not alter the structure of the original matrix and therefore preserve sparsity. However, there are special cases where pivoting techniques can alleviate the fill-in problem of direct methods.

Herein a brief introduction on the direct method is presented which is used later on by our simulation system (SSSS) and to calculate the generalized inverse of a matrix.

The direct method concerned factoring a matrix A in terms of a lower triangular matrix L and an upper triangular matrix U . In Burden, Faires and Reynolds [1981] it was shown that this factorization existed whenever the linear system $Ax=b$ could be solved uniquely by Gaussian elimination (this method is generally used to solve a system of linear equations) without row or column interchanges. The system $LUx=Ax=b$ could be transformed into the system $Ux=L^{-1}b$ and, since U is upper triangular, backward substitution could be applied. Although the specific form of L and U can be obtained from the Gaussian elimination process, it is desirable to find a more direct method for their determination, so that, if many systems are to be solved using A , only a forward and backward substitution need to be performed. To illustrate a procedure for the calculation of the entries of these matrices, we consider that a general matrix ($n \times n$) A can be factored in the form,

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & & a_{nn} \end{bmatrix} = LU$$

where,

$$L = \begin{bmatrix} l_{11} & & & \\ l_{21} & l_{22} & & \\ \vdots & \vdots & \ddots & \\ l_{n1} & \cdots & l_{n,n-1} & l_{nn} \end{bmatrix} \quad \text{and} \quad U = \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ & u_{22} & & u_{2n} \\ & & \ddots & \\ & & & u_{nn} \end{bmatrix}$$

For a (4×4) , the 16 known entries can be used to partially determine

the ten unknown entries in L and the same number in U. If a procedure leading to a unique solution is desired, however, four additional conditions on the entries of L and U are needed. The method to be used in this example arbitrarily requires that $L_{11}=L_{22}=L_{33}=L_{44}=1$; this is known as Doolittle's method. The multiplication of L by U,

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 \\ l_{31} & l_{32} & 1 & 0 \\ l_{41} & l_{42} & l_{43} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix}$$

To calculate all the unknown entries in L and U in the case of a matrix A (n×n), we can use the following algorithm:

Step 1: Select l_{11} and u_{11} satisfying $l_{11}u_{11}=a_{11}$.

Step 2: Generate the entries in the first column of L by the condition:

$$l_{j1} = \frac{a_{j1}}{u_{11}}, \text{ for each } j=2,3,\dots,n.$$

Step 3: Generate the entries in the first row of U by the condition:

$$u_{1j} = \frac{a_{1j}}{l_{11}}, \text{ for each } j=2,3,\dots,n.$$

Step 4: Set $i=2$.

Step 5: Select l_{ii} and u_{ii} satisfying,

$$l_{ii}u_{ii} = a_{ii} - \sum_{k=1}^{i-1} l_{ik}u_{ki}.$$

Step 6: If $i < n$, goto step 7.

If $i=n$, goto step 10.

Step 7: Generate the entries in the i^{th} column of L by the condition:

$$l_{ji} = \frac{1}{u_{ii}} \left[a_{ji} - \sum_{k=1}^{i-1} l_{jk}u_{ki} \right], \text{ for each } j=i+1, i+2, \dots, n.$$

Step 8: Generate the entries in the i th row of U by the condition,

$$u_{ij} = \frac{1}{l_{ii}} \left[a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj} \right], \text{ for } j=i+1, i+2, \dots, n.$$

Step 9: Add 1 to i and goto step 5.

Step 10: The procedure is complete when all entries of L and U have been determined.

A difficulty which can arise when using the algorithm above to obtain the factorization of the coefficient matrix of a linear system of equations is caused by the fact that no pivoting is used to reduce the effect of round-off error. The round-off error can be quite significant when finite digit arithmetic is used and any efficient algorithm must take this effect into consideration. The material of this section is obtained from [Burden, Faires and Reynolds, 1981], [Deboor, 1972].

6.2 MATRIX APPLICATIONS USING SSSS

In the following paragraphs of this section the solution of some matrix applications by using the soft-systolic simulation system is presented.

6.2.1 4*4 Matrix Transpose

This is a slightly more complex transposition problem incorporating the use of the 2x2 problem which was defined earlier. Consider the matrix,

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

Problem: Trace a RISAL program to ensure that:

$$A^T = \begin{bmatrix} a_{11} & a_{21} & a_{31} & a_{41} \\ a_{12} & a_{22} & a_{32} & a_{42} \\ a_{13} & a_{23} & a_{33} & a_{43} \\ a_{14} & a_{24} & a_{34} & a_{44} \end{bmatrix}$$

To write a RISAL program to transpose the matrix above we implement the following steps:

REMARK: null=null ,0,0, datan=data n,3,0, datas=data s,5,0, dataw=data w, 6,0, and datae=data e,4,0.

Step 1: By reading the matrix elements from the north into the ISA grid, each matrix element will be stored in a processor, as shown in Figure 6.1.

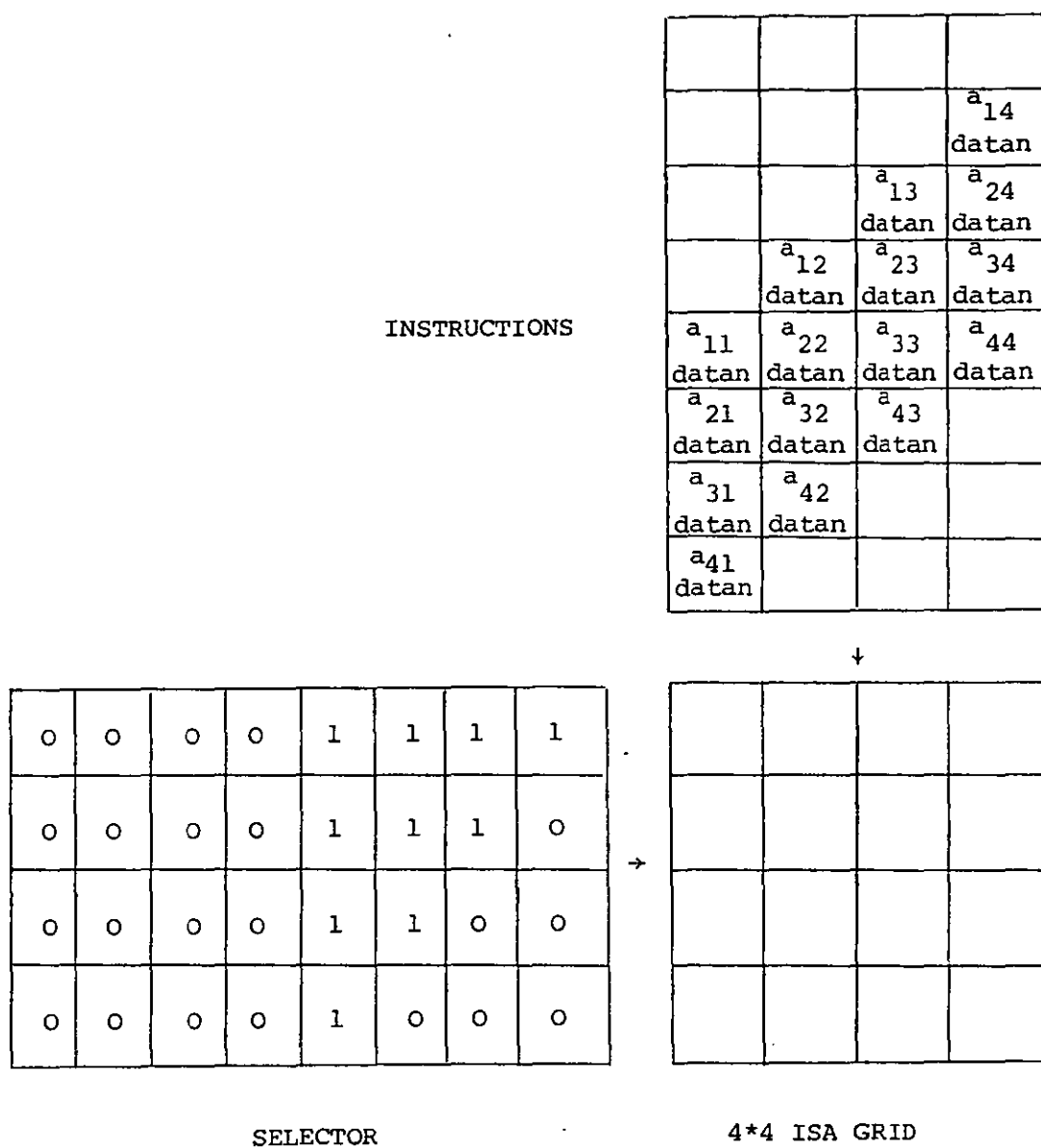


FIGURE 6.1: Reading the 4x4 Matrix Elements from the North into the 4*4 ISA Grid.

Step 2: Start to transpose the matrix elements as shown in Figure 6.2.

INSTRUCTION

	null	datae	dataw
null	datae	dataw	null
datae	dataw	datae	dataw
null	datae	dataw	null
datae	dataw	null	null
null	datas	null	null
datas	datan	null	null
datan	datas	null	null
datas	datan	datae	dataw
datan	datae	dataw	null
datae	dataw	datae	dataw
null	datae	dataw	null
datae	dataw	datae	dataw
null	null	datas	null
datae	dataw	datan	null
datas	null	datae	dataw
datan	null	null	null
datae	dataw	null	null
null	null	null	null

SELECTOR

0	1	1	1	1	1	1	0	1	0	1	1	1	1	1	0	1	1
0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	0	0
0	0	0	0	1	1	1	1	0	0	0	0	1	1	0	1	1	0
0	0	0	0	1	0	1	0	0	0	0	0	0	1	0	0	0	0

→

a ₁₁	a ₁₂	a ₁₃	a ₁₄
a ₂₁	a ₂₂	a ₂₃	a ₂₄
a ₃₁	a ₃₂	a ₃₃	a ₃₄
a ₄₁	a ₄₂	a ₄₃	a ₄₄

4*4 ISA GRID

FIGURE 6.2: Transpose RISAL Program for 4*4 Matrix

Step 3: Read the matrix elements from the south to the north of the ISA grid as shown in Figure 6.3.

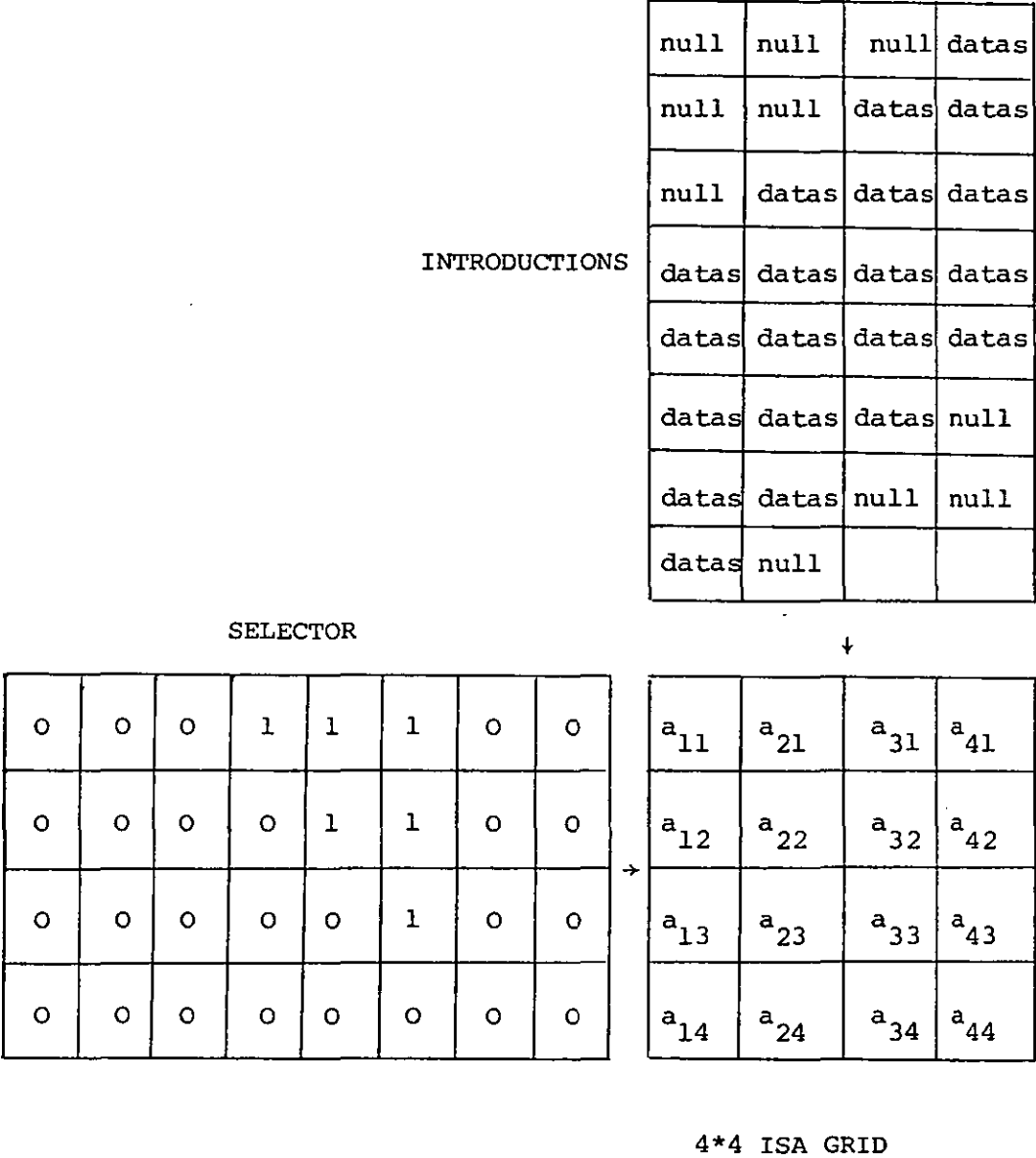


FIGURE 6.3: Reading the Matrix Elements 4×4 from the South to the North of the 4*4 ISA Grid.

EXAMPLE: Given the matrix,

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}^T = \begin{bmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{bmatrix}$$

6.2.2 4×4 LU Decomposition

Given a 4×4 matrix,

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

To factorize this matrix into a lower triangular matrix L and an upper triangular matrix U as defined previously in Section 6.1, we obtain,

$$L = \begin{bmatrix} 1 & & & \\ \ell_{21} & 1 & & \\ \ell_{31} & \ell_{32} & 1 & \\ \ell_{41} & \ell_{42} & \ell_{43} & 1 \end{bmatrix} \quad U = \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ & u_{22} & u_{23} & u_{24} \\ & & u_{33} & u_{34} \\ & & & u_{44} \end{bmatrix}$$

To determine the unknown entries of L and U we obtain by comparing terms:

U entries

$$u_{11} = a_{11}$$

$$u_{12} = a_{12}$$

$$u_{13} = a_{13}$$

$$u_{14} = a_{14}$$

$$u_{22} = a_{22} - l_{21} u_{12}$$

$$u_{23} = a_{23} - l_{21} u_{13}$$

$$u_{24} = a_{24} - l_{21} u_{14}$$

$$u_{33} = a_{33} - (l_{31} u_{13} + l_{32} u_{23})$$

$$u_{34} = a_{34} - (l_{31} u_{14} + l_{32} u_{24})$$

$$u_{44} = a_{44} - (l_{41} u_{14} + l_{42} u_{24} + l_{43} u_{34})$$

L entries

$$l_{21} = \frac{a_{21}}{u_{11}}$$

$$l_{31} = \frac{a_{31}}{u_{11}}$$

$$l_{41} = \frac{a_{41}}{u_{11}}$$

$$l_{32} = \frac{a_{32} - l_{31} u_{12}}{u_{22}}$$

$$l_{42} = \frac{a_{42} - l_{41} u_{12}}{u_{22}}$$

$$l_{43} = \frac{a_{43} - (l_{41} u_{13} + l_{42} u_{23})}{u_{33}}$$

To write a RISAL program to determine the matrix entries of L and U above we have to implement the following steps:

Step 1: Read the matrix elements A into the 4x4 ISA grid. See 6.2.1, Step 1.

Step 2: Start to factorize the matrix A into L and U by tracing the RISAL program shown in Figure 6.4. As a result, the elements of L will be held in the processing elements $P_{21}, P_{31}, P_{32}, P_{41}, P_{42}, P_{43}$ and the elements of U will be held in the processing elements $P_{11}, P_{12}, P_{13}, P_{14}, P_{22}, P_{23}, P_{24}, P_{33}, P_{34}$ and P_{44} .

Step 3: Read the L and U elements from the south to the north of the ISA grid. See 6.2.1, step 3.

EXAMPLE:

Given a 4*4 matrix

$$A = \begin{bmatrix} 2 & 3 & 3 & 2 \\ 4 & 1 & 2 & 3 \\ 2 & 2 & 5 & 1 \\ 3 & 4 & 1 & 2 \end{bmatrix}$$

By tracing the matrix through the RISAL program in Figure 6.4, we obtain:

$$L = \begin{bmatrix} 1 & & & \\ 2 & 1 & \bigcirc & \\ 1 & 0.2 & 1 & \\ 1.5 & 0.1 & -1.107143 & 1 \end{bmatrix} \quad U = \begin{bmatrix} 2 & 3 & 3 & 2 \\ & -5 & -4 & -1 \\ & & 2.8 & -0.8 \\ & \bigcirc & & -1.785714 \end{bmatrix}$$

REMARK:

null = null ,0,0
datan = data n,3,0
datas = data s,5,0
dataw = data w,6,0
datae = data e,4,0
copy = copy ,0,0

INSTRUCTIONS

FIGURE 6.4: 4×4 LU Decomposition RISAL Program.

REPEATED EIGHT TIMES	REPEATED EIGHT TIMES	REPEATED FIVE TIMES			
+	+	+			
0	0	0	0	0	0
0	0	1	1	1	0
0	1	1	1	0	0
1	1	1	0	0	0
SELECTOR					

null	null	null	copy
null	null	null	sub ,7,6
null	null	null	mult ,3,6
null	null	null	dataw
null	null	copy	null
null	null	div ,7,3	datan
null	null	datan	mov 1,7,
null	null	mov ,1,7	copy
null	null	copy	sub ,7,0
null	null	sub ,7,0	mult ,3,6
null	null	mult ,3,6	dataw
null	null	dataw	null
null	copy	null	datan
null	div ,3,7	datan	mov ,1,7
null	datan	mov ,1,7	copy
null	mov ,1,7	copy	sub ,7,0
null	copy	sub ,7,0	mult ,3,6
null	sub ,7,0	mult ,3,6	dataw
null	mult ,3,6	dataw	null
null	dataw	null	datan
copy	null	datan	mov ,1,7
div ,3,7	datan	mov ,1,7	
datan	mov ,1,7		
mov ,1,7			

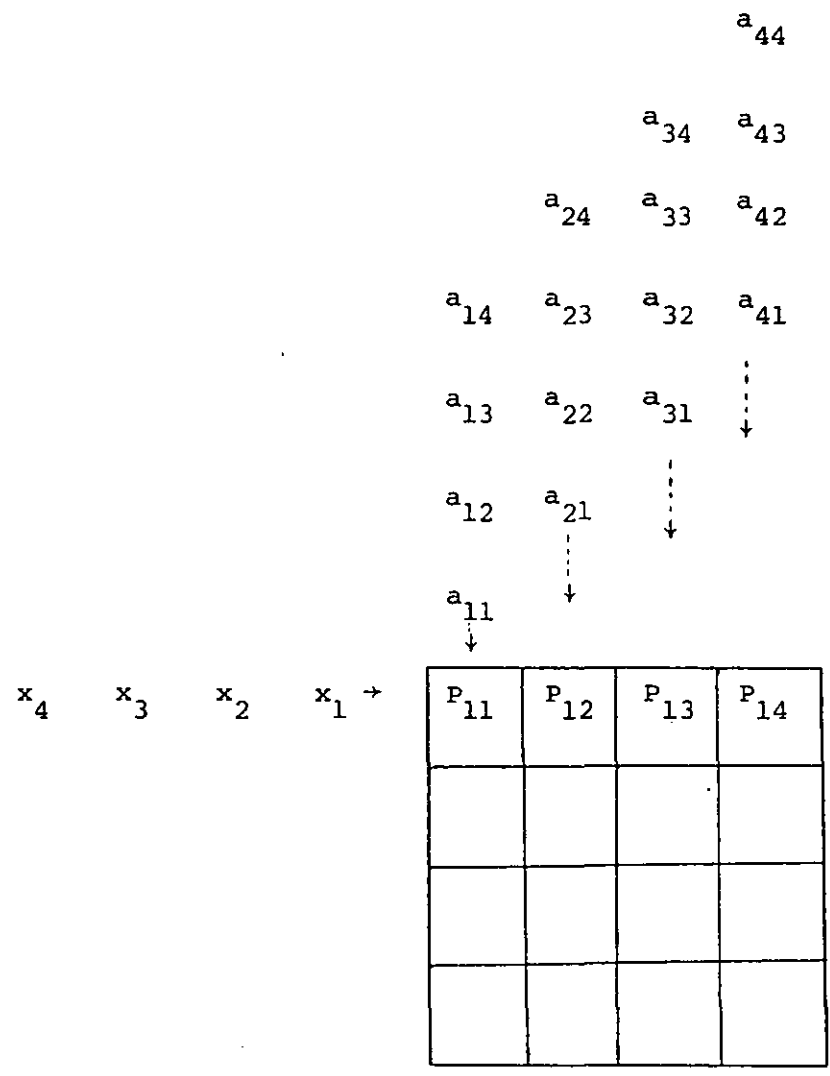
a ₁₁	a ₁₂	a ₁₃	a ₁₄
a ₂₁	a ₂₂	a ₂₃	a ₂₄
a ₃₁	a ₃₂	a ₃₃	a ₃₄
a ₄₁	a ₄₂	a ₄₃	a ₄₄
4×4 ISA GRID			

6.2.3 Matrix-Vector Multiplication

Consider the matrix multiplication by vector, i.e. $y=Ax$, as defined in Section 6.1, where A is a $(n \times n)$ matrix and x, y are $(n \times 1)$ vectors. Each component of y is produced by adding the multiplication of a row of A by x . More formally, the recurrence relation can take the form,

$$\begin{aligned} y_i^{(1)} &= 0 \\ y_i^{(k+1)} &= y_i^{(k)} + a_{ik} x_k \\ y_i &= y_i^{(n+1)}, \quad i, k=1, 2, \dots, n. \end{aligned}$$

For $n=4$, Figures 6.5 and 6.6 show the implementation of this algorithm by using the ISA grid. It is based on the engagement of the processing elements in the first row of the ISA grid. Each processing element in this row will implement just multiply and addition instructions and then move the result of the addition into a storage register in the auxiliary memory of the processing element. The data sequences from the north consist of the rows of the matrix A , while the data from the west are the components of the vector x . Finally, each element of the resulting vector is accumulated into $P_{11}, P_{12}, P_{13}, P_{14}$ simultaneously.



4x4 ISA GRID

FIGURE 6.5: Data Moving From the North and the West into the ISA Grid.

In Figure 6.6, each Y instruction represents the following instructions:

```
mult n w,3,6;  
add ,7,0;  
mov ,0,7;  
mov ,6,1;
```

and datas means data s,5,0;

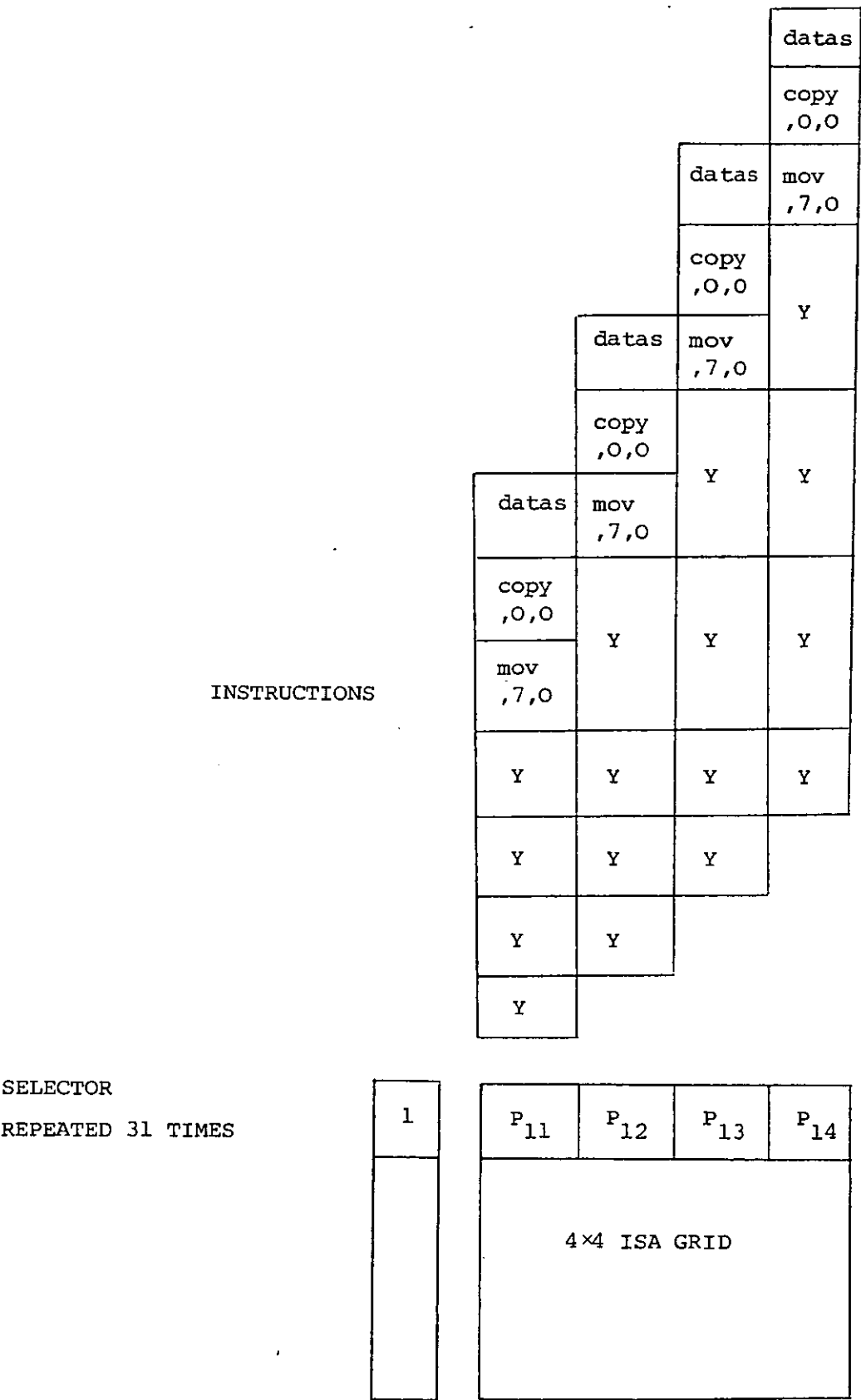


FIGURE 6.6: RISAL Program for the 4*4 Matrix Multiplication by Vector

EXAMPLE:

Given $y = Ax$,

where the matrix,

$$A = \begin{bmatrix} 2.8 & 3 & 2 & 5.1 \\ 3.6 & 4.8 & 6 & 8 \\ 4 & 3 & 2.2 & 6.1 \\ 4.2 & 1 & 0 & 9 \end{bmatrix}$$

and the vector

$$x = \begin{bmatrix} 2.1 \\ 3 \\ 5 \\ 6.6 \end{bmatrix}$$

By using the RISAL program mentioned previously we obtain the vector,

$$y = \begin{bmatrix} 58.540001 \\ 104.759995 \\ 68.659996 \\ 71.219994 \end{bmatrix}$$

6.2.4 Matrix-Matrix Multiplication

Another problem to be discussed is the multiplication of two $(n \times n)$ matrices, $C=AB$, as defined in Section 6.1, again each component of matrix C is produced by adding the multiplication of each row of matrix A and each column of matrix B . More formally, the recurrence,

$$c_{ij}^{(1)} = 0$$

$$c_{ij}^{(k+1)} = c_{ij}^{(k)} + a_{ik}b_{kj}$$

$$c_{ij} = c_{ij}^{(n+1)}, \quad i, j, k=1, 2, \dots, n.$$

The formula can be seen as a set of n matrix by vector multiplications as defined in the previous paragraph. To solve this problem for $n=4$ and by using the ISA grid, it is again seen as based on the engagement of the processing elements in the first row of the ISA grid. The data sequences from the north represents the matrix B elements, while the data from the west represents the matrix A elements. By repeating the same process as in the previous paragraph, the processing element $(P_{11}, P_{12}, P_{13}, P_{14})$ will implement the multiply and addition instructions and then move the result of the addition into a storage register in the auxiliary memory of the processing element for every column of the matrix C. Finally, the elements of the resulting C are accumulated as follows:

The first column in P_{11} registers 7,8,9 and 10

"	second	"	"	P_{12}	"	"	"	"	"
"	third	"	"	P_{13}	"	"	"	"	"
"	fourth	"	"	P_{14}	"	"	"	"	"

So, given $C=A*B$, we have,

$$C = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}$$

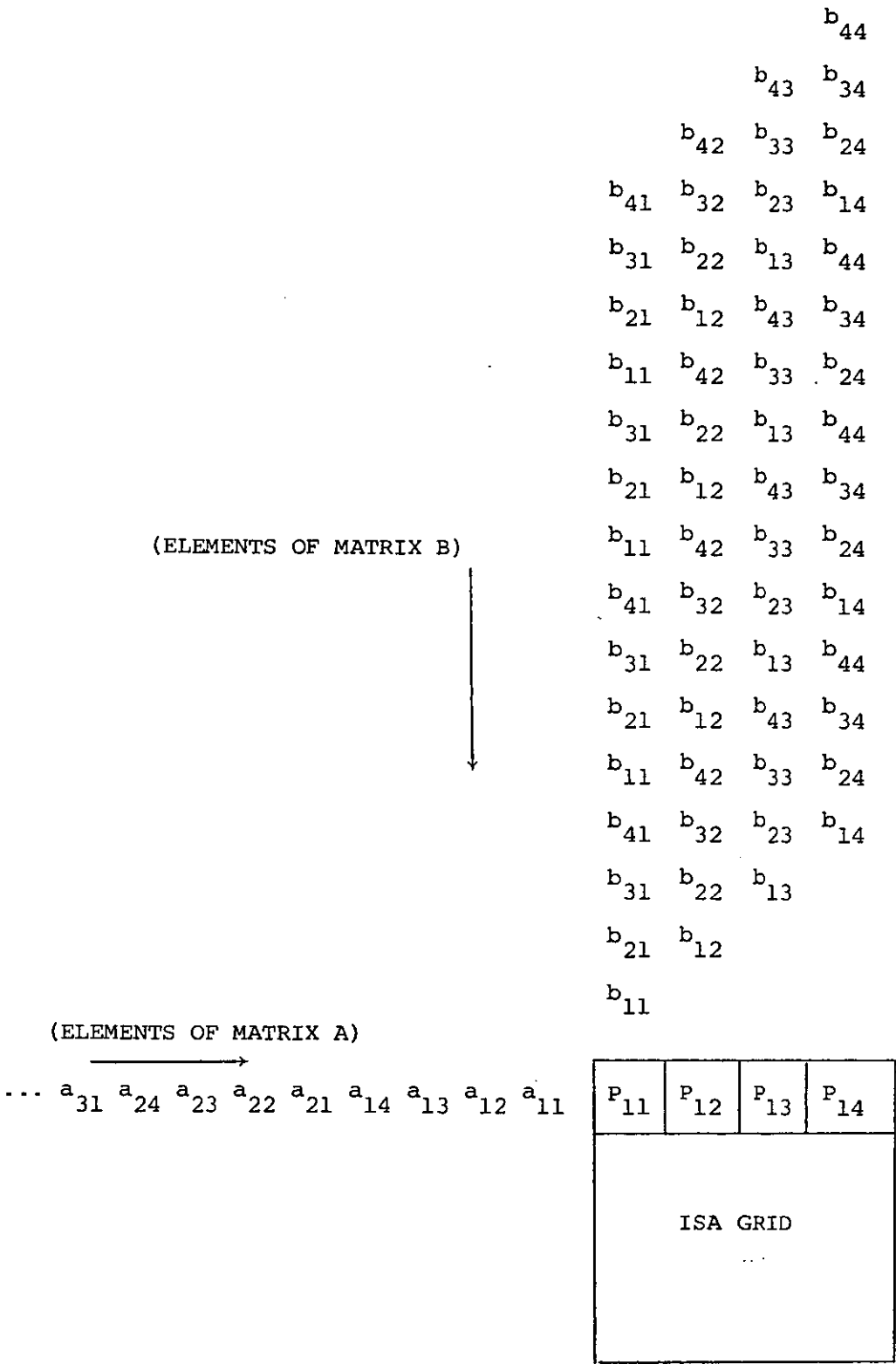


FIGURE 6.7: Data of Matrices A and B Moving from the North and the West into the ISA Grid.

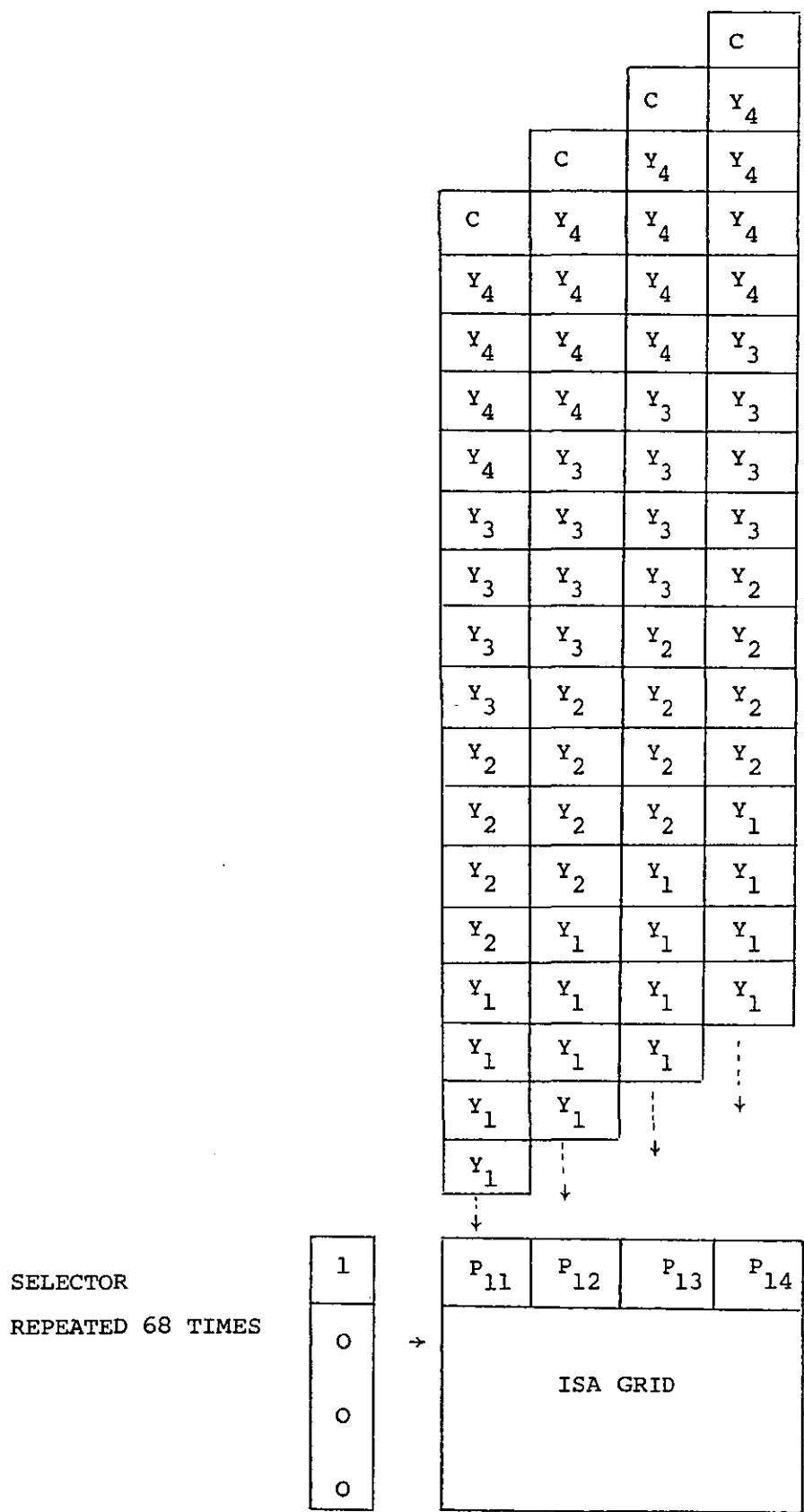


FIGURE 6.8: RISAL Program for 4x4 Matrix Multiplication

Figure 6.7 shows the elements of matrix B moving from the north, and the elements of matrix A moving from the west into the ISA grid.

Figure 6.8 shows the RISAL program for the matrix-matrix multiplication process. The symbols Y_1, Y_2, Y_3, Y_4 and C represents the following instruction,

$$Y_1 = \left\{ \begin{array}{l} \text{mult n w,3,6;} \\ \text{add ,7,0;} \\ \text{mov ,0,7;} \\ \text{mov ,6,1;} \end{array} \right\}$$

$$Y_2 = \left\{ \begin{array}{l} \text{mult n w,3,6;} \\ \text{add ,8,0;} \\ \text{mov ,0,8;} \\ \text{mov ,6,1;} \end{array} \right\}$$

$$Y_3 = \left\{ \begin{array}{l} \text{mult n w,3,6;} \\ \text{add ,9,0;} \\ \text{mov ,0,9;} \\ \text{mov ,6,1;} \end{array} \right\}$$

$$Y_4 = \left\{ \begin{array}{l} \text{mult n w,3,6;} \\ \text{add ,10,0;} \\ \text{mov ,0,10;} \\ \text{mov ,6,1;} \end{array} \right\}$$

and

$$C = \left\{ \begin{array}{l} \text{mov ,7,0;} \\ \text{copy ,0,0;} \\ \text{data s,5,0;} \\ \text{mov ,8,0;} \\ \text{copy ,0,0;} \\ \text{data s,5,0;} \\ \text{mov ,9,0;} \\ \text{copy ,0,0;} \\ \text{data s,5,0;} \\ \text{mov ,10,0;} \\ \text{copy ,0,0;} \\ \text{data s,5,0;} \end{array} \right\}$$

EXAMPLE:

Obtain $C=AB$, where,

$$A = \begin{bmatrix} 2.8 & 3 & 2 & 5.1 \\ 3.6 & 4.8 & 6 & 8 \\ 4 & 3 & 2.2 & 6.1 \\ 4.2 & 1 & 0 & 9 \end{bmatrix}$$

and

$$B = \begin{bmatrix} 2.1 & 0 & 2.3 & 1.8 \\ 3 & 1 & 5 & 6.1 \\ 5 & 1.2 & 2 & 3.3 \\ 6.6 & 2.2 & 0 & 3.6 \end{bmatrix}$$

By using the multiplication process shown in Figures 6.7 and 6.8 we obtain,

$$C = \begin{bmatrix} 58.540001 & 16.620001 & 25.439999 & 48.299999 \\ 104.759995 & 29.600000 & 44.279999 & 84.360001 \\ 68.659996 & 19.060001 & 28.600000 & 54.720001 \\ 71.219994 & 20.800001 & 14.659999 & 46.059998 \end{bmatrix}$$

6.3 THE SOLUTION OF LINEAR SYSTEMS USING SSSS

Given the linear system,

$$Ax = b, \quad (6.3.1)$$

as defined earlier in section 6.1, where A is a given square matrix of order n, b a given n-vector. We wish to solve the linear system above for the unknown n-vector x.

To obtain the vector x we have,

$$x = A^{-1}b. \quad (6.3.2)$$

However A^{-1} is difficult to obtain, so we need to factorize the matrix A into LU factors because L and U are easily inverted systems. So we have,

$$LUx = b.$$

Let $Ux = y$,

so the system breaks down into 2 triangular systems,

$$Ly = b, \quad (6.3.3)$$

$$\text{and} \quad Ux = y \quad (6.3.4)$$

For n=4, the matrix,

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

$$= \begin{bmatrix} 1 & & & \\ l_{21} & 1 & & \\ l_{31} & l_{32} & 1 & \\ l_{41} & l_{42} & l_{43} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ & u_{22} & u_{23} & u_{24} \\ & & u_{33} & u_{34} \\ & & & u_{44} \end{bmatrix}$$

To obtain y from (6.3.3) we have,

$$\begin{bmatrix} 1 & & & \\ \ell_{21} & 1 & & \\ \ell_{31} & \ell_{32} & 1 & \\ \ell_{41} & \ell_{42} & \ell_{43} & 1 \end{bmatrix} * \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} \quad (6.3.5)$$

From (6.3.5) above, we obtain,

$$\begin{aligned} y_1 &= b_1 \\ y_2 &= b_2 - \ell_{21}y_1 \\ y_3 &= b_3 - \ell_{31}y_1 + \ell_{32}y_2 \\ y_4 &= b_4 - \ell_{41}y_1 + \ell_{42}y_2 + \ell_{43}y_3 \end{aligned}$$

To obtain x from (6.3.4), we have,

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ & u_{22} & u_{23} & u_{24} \\ & & u_{33} & u_{34} \\ & & & u_{44} \end{bmatrix} * \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} \quad (6.3.6)$$

From (6.3.6) above we obtain,

$$\begin{aligned} x_4 &= \frac{y_4}{u_{44}} \\ x_3 &= \frac{y_3 - u_{34}x_4}{u_{33}} \\ x_2 &= \frac{y_2 - (u_{23}x_3 + u_{24}x_4)}{u_{22}} \\ x_1 &= \frac{y_1 - (u_{12}x_2 + u_{13}x_3 + u_{14}x_4)}{u_{11}} \end{aligned}$$

To write a RISAL program to solve this problem we implement the following steps:

Step 1: Read the matrix A into the ISA grid, and factorize it by

using the same concepts used in step 1 and step 2 in paragraph 6.2.2. The elements of L and U will be held in the ISA grid, each element in a processing element.

Step 2: As shown in Figure 6.9.

- move the elements of L into register 7 of the processing elements $P_{21}, P_{31}, P_{32}, P_{41}, P_{42}$, and P_{43} .
- move the elements of U into register 8 of the processing elements $P_{11}, P_{12}, P_{13}, P_{14}, P_{22}, P_{23}, P_{24}, P_{33}, P_{34}$ and P_{44} .
- read one's (1) from the north into register 7 of the processing elements P_{11}, P_{22}, P_{33} and P_{44} .

Step 3: As shown in Figure 6.10, by reading the value of the vector b (b_1, b_2, b_3 and b_4) respectively from the north into the columns of the ISA grid, we obtain the values of y_1, y_2, y_3 and y_4 as follows:

- in P_{11} we obtain the value of y_1 ($y_1 = b_1$).
- read the value of y_1 from P_{11} throughout P_{21}, P_{31} and P_{41} and multiply y_1 by the value in register 7 of these processing elements, and move the result of the multiplication into the communication registers, and read them to the west neighbouring processing elements P_{22}, P_{32} and P_{42} (register 6).
- in P_{12} no operation.
- in P_{22} , we obtain the value of y_2 by the subtraction of the value in register 6 from the value of b_2 and move the value of y_2 into the communication register.
- read the value of y_2 throughout the processing elements P_{32}, P_{42} and multiply y_2 by the value in register 7 of these processing elements, and add the result to the value in

register 6. Move the result into the communication registers and read them to the west neighbouring processing elements P_{33} and P_{43} .

- in P_{13} and P_{23} no operation.
- in P_{33} , we obtain the value of y_3 by the subtraction of the value in register 6 from the value of b_3 , and move the result (which is y_3) into the communication register.
- read y_3 into P_{43} , and multiply y_3 by the value in register 7, add the result to the value in register 6, and move the result into the communication register, then read it to the west neighbouring processing element P_{44} .
- in P_{14} , P_{24} and P_{34} no operation.
- in P_{44} , we obtain the value of y_4 by subtracting the value in register 6 from the value of b_4 .

Step 4: From Step 3 above we have obtained the values of y_1, y_2, y_3 and y_4 stored in the processing elements P_{11}, P_{22}, P_{33} and P_{44} .

To determine the values of x_1, x_2, x_3 and x_4 , we implement the procedure as shown in Figure 6.11.

- in P_{44} , we divide the value of y_4 by the value in register 8 to obtain x_4 .
- read the value of x_4 to the north throughout the processing elements P_{34}, P_{24} , and P_{14} and multiply x_4 by the values in register 8 of these processing elements, move the results into the communication registers, and finally read them to the east neighbouring processing elements P_{33}, P_{23} and P_{13} (register 4).

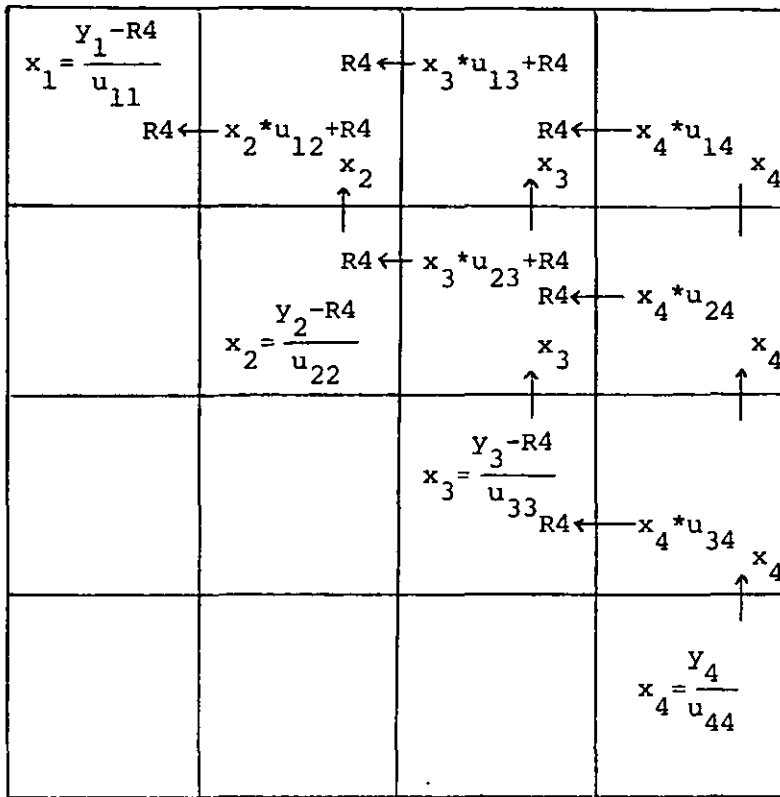


FIGURE 6.11: Determination of the Values of x by Using the ISA Grid

- in P_{33} , we subtract the value in register 4 from the value of y_3 , and divide the result by the value in register 8 to obtain x_3 .
- read the value of x_3 to the north throughout the processing element P_{23}, P_{13} , and multiply the value of x_3 by the value in register 8 of these processing elements; add the result to the value in register 4, and move the result into the communication registers. Finally, read them to the east neighbouring P_{22} and P_{12} (register 4).
- in P_{22} , we subtract the value in register 4 from the value of y_2 , and divide the result by the value in register 8 to obtain x_2 .

- read the value of x_2 to the north into the processing element P_{12} and multiply it by the value in register 8; add the result to the value in register 4, and move the result into the communication register. Finally, read it to the east neighbouring P_{11} (register 4).
- in P_{11} , we subtract the value in register 4 from the value of y_1 , and divide the result by the value in register 8 of this processing element to obtain the value of x_1 .

Step 5: read the values of x_1, x_2, x_3 and x_4 from the south of the first row of the ISA grid.

EXAMPLE:

Given the linear system,

$$Ax = b,$$

where,

$$A = \begin{bmatrix} 2 & 3 & 3 & 2 \\ 4 & 1 & 2 & 3 \\ 2 & 3 & 6 & 1 \\ 2 & 3 & 3 & 3 \end{bmatrix}$$

and the vector,

$$b = \begin{bmatrix} 10 \\ 10 \\ 12 \\ 11 \end{bmatrix}.$$

By using the steps mentioned above we factorize the matrix A, to obtain,

$$L = \begin{bmatrix} 1 & & & \\ 2 & 1 & \bigcirc & \\ 1 & 0 & 1 & \\ 1 & 0 & 0 & 1 \end{bmatrix} \quad \text{and} \quad U = \begin{bmatrix} 2 & 3 & 3 & 2 \\ & -5 & -4 & -1 \\ & & 3 & -1 \\ \bigcirc & & & 1 \end{bmatrix}.$$

To obtain the value of y from (6.3.5), we have,

$$\begin{bmatrix} 1 & & & \\ 2 & 1 & & \bigcirc \\ 1 & 0 & 1 & \\ 1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} 10 \\ 10 \\ 12 \\ 11 \end{bmatrix}$$

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} 10 \\ -10 \\ 2 \\ 1 \end{bmatrix}$$

To obtain the value of x from (6.3.6) we have,

$$\begin{bmatrix} 2 & 3 & 3 & 2 \\ & -5 & -4 & -1 \\ & & \bigcirc & 3 \\ & & & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 10 \\ -10 \\ 2 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

6.4 FINDING THE GENERALIZED INVERSE OF A RECTANGULAR MATRIX USING SSSS

Our aim in this section is to find the optimal solution \bar{x} to an inconsistent linear system,

$$Ax = b$$

where A any coefficient matrix and b any right side.

First, we have to find a rule that specifies \bar{x} . Suppose we know the value of $A\bar{x}$, for every x , Ax is necessarily in the column space of A; it is a combination of the columns, weighted by the components of x . Therefore the optional choice $A\bar{x}$ is the point p in this column space closest to the given b . This choice minimizes the error $E = \|Ax - b\|$. In other words, we project b onto the column space,

$$A\bar{x} = p = pb \quad (6.4.1)$$

The equation above is enough to determine \bar{x} itself. It is another form of the normal equation,

$$A^T A \bar{x} = A^T b \quad (6.4.2)$$

Certainly \bar{x} is determined when there is only one combination of the columns of A that will produce P ; the weights in this combination will be the components of \bar{x} .

We know several equivalent conditions for the equation $A\bar{x} = p$ to have only one solution:

- (i) The columns of A are linearly independent.
- (ii) The null space of A contains only the zero vector.
- (iii) The rank of A is n .
- (iv) The square matrix $A^T A$ is invertible.

In such a case, the only solution to (6.4.1) is,

$$\bar{x} = [(A^T A)^{-1} A^T] b \quad (6.4.3)$$

This formula, which is comparatively simple, includes the simplest case of all when A is actually invertible. Then \bar{x} coincides with the one and only solution of the original system $Ax=b$: $\bar{x}=A^{-1}(A^T)^{-1}A^Tb=A^{-1}b$. This suggests another way of describing our aim: we are trying to define the pseudo inverse A^+ of a matrix which may not be invertible.

REMARK: A^+ is also called the Moore-Penrose inverse, after its discoverers, or more commonly known as a generalized inverse of A , as defined earlier in Section 6.1. But a great many other matrices, sharing some but not all of the properties we intend for A^+ , have also been described as a generalized inverse.

When the matrix is invertible, that means $A^+=A^{-1}$. When the matrix satisfies the condition (i)-(iv) listed above, the pseudoinverse is the left inverse which appears in the formula (6.4.3),

$$A^+ = (A^T A)^{-1} A^T . \quad (6.4.4)$$

But when the conditions (i)-(iv) do not hold, and \bar{x} is uniquely determined by $A\bar{x}=p$, the pseudo inverse remains to be defined. We have to choose one of the many vectors that satisfy $A\bar{x}=p$ and that choice will be, by definition the optimal solution $\bar{x}=A^+b$ to the inconsistent linear system $Ax=b$. To solve the problem above and calculate the generalized inverse, we have the rectangular linear system

$$Ax = b , \quad (6.4.5)$$

which after multiplying by A^T , we obtain

$$A^T Ax = A^T b , \quad (6.4.6)$$

$$\tilde{A}x = A^T b . \quad (6.4.7)$$

Then, by factorizing \tilde{A} into L and U elements, we have,

$$LUx = A^T b .$$

$$\text{Let} \quad Ux = y, \quad (6.4.8)$$

we obtain,

$$Ly = A^T b, \quad (6.4.9)$$

From (6.4.9) we determine the value of y , and by substituting the value of y in (6.4.8) we obtain the final value of x .

To write a RISAL program to solve this problem and calculate the value of x in the case of an $(n \times m)$ matrix A when $n=3$ and $m=4$, we implement the following steps:

- Step 1:
- Read the matrix A elements from the north into the ISA grid.
 - Transpose the matrix A by using the same concept in paragraph 6.2.1.
 - multiply the matrix A^T by A by using the concept used in paragraph 6.2.4. The result will be \tilde{A} and it will be held in the ISA grid, each element in a processor.
- Step 2: Factorize the matrix \tilde{A} into L and U components by using the same concept mentioned in paragraph 6.2.2 Step 2.
- Move the L and U elements into a different register as has been done earlier in Step 2 in Section 6.3.
- Step 3: To calculate the values of y and then x 's which is the generalized inverse, we have to implement Step 3 and Step 4 in Section 6.3, by reading the elements of A^T (right hand side) from the north into the ISA grid, instead of the vector b mentioned in that section. By repeating this process four times, we will obtain the values of x .
- Step 4: Read the values of x from the south to the north of the ISA grid.

EXAMPLE:

Given the rectangular linear system in (6.4.5), where,

$$A = \begin{bmatrix} 1 & 4 & 3 \\ 2 & 0 & 1 \\ 3 & 2 & 2 \\ 4 & 1 & 1 \end{bmatrix}$$

To determine the value of x , we have to implement the RISAL program mentioned earlier in Step 1, Step 2, Step 3 and Step 4. First by applying equation (6.4.6) we have,

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 4 & 0 & 2 & 1 \\ 3 & 1 & 2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 4 & 3 \\ 2 & 0 & 1 \\ 3 & 2 & 2 \\ 4 & 1 & 1 \end{bmatrix} * x = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 4 & 0 & 2 & 1 \\ 3 & 1 & 2 & 1 \end{bmatrix} * b$$

$$\begin{bmatrix} 30 & 14 & 15 \\ 14 & 21 & 17 \\ 15 & 17 & 15 \end{bmatrix} * x = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 4 & 0 & 2 & 1 \\ 3 & 1 & 2 & 1 \end{bmatrix}$$

To factorize \tilde{A} in equation (6.4.7) we obtain,

$$\begin{bmatrix} 1 & & & \\ 0.466667 & 1 & & \\ 0.5 & 0.691244 & 1 & \end{bmatrix} \begin{bmatrix} 30 & 14 & 15 \\ 14.466667 & 10 & \\ & 0.587558 & \end{bmatrix} * x =$$

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 4 & 0 & 2 & 1 \\ 3 & 1 & 2 & 1 \end{bmatrix}$$

To calculate the values of y , we have (4.6.9),

$$\begin{bmatrix} 1 & & & \\ 0.466667 & 1 & & \\ 0.5 & 0.691244 & 1 & \end{bmatrix} \begin{bmatrix} y_{11} & y_{12} & y_{13} & y_{14} \\ y_{21} & y_{22} & y_{23} & y_{24} \\ y_{31} & y_{32} & y_{33} & y_{34} \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 4 & 0 & 2 & 1 \\ 3 & 1 & 2 & 1 \end{bmatrix}$$

$$\begin{bmatrix} y_{11} & y_{12} & y_{13} & y_{14} \\ y_{21} & y_{22} & y_{23} & y_{24} \\ y_{31} & y_{32} & y_{33} & y_{34} \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 3.533333 & -0.933333 & 0.600002 & -0.866664 \\ 0.057608 & -0.645159 & 0.085253 & 0.400925 \end{bmatrix}$$

To calculate x's from equation (6.4.8), we have,

$$\begin{bmatrix} 30 & 14 & 15 \\ & 14.466667 & 10 \\ & & 0.587558 \end{bmatrix} \begin{bmatrix} x_{11} & x_{12} & x_{13} & x_{14} \\ x_{21} & x_{22} & x_{23} & x_{24} \\ x_{31} & x_{32} & x_{33} & x_{34} \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 3.533333 & 0.933333 & 0.600002 & -0.866664 \\ 0.057608 & -0.645159 & 0.085353 & 0.400925 \end{bmatrix}$$

From the above we obtain,

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} & x_{14} \\ x_{12} & x_{22} & x_{23} & x_{24} \\ x_{31} & x_{32} & x_{33} & x_{34} \end{bmatrix} = \begin{bmatrix} -0.098039 & -0.098039 & 0.054902 & 0.0282353 \\ 0.176471 & -0.823529 & -0.058824 & 0.411764 \\ 0.098039 & 1.098039 & 0.145098 & -0.682353 \end{bmatrix}$$

which is the generalized inverse of the rectangular matrix A in (6.4.5).

6.5 SOME APPLICATION TO THE GENERALIZED INVERSE OF A RECTANGULAR MATRIX USING SSSS

Consider the system of m linear algebraic equations in n unknowns,

$$a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1$$

$$a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2$$

.....

$$a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n = b_m$$

which we write in the form,

$$Ax = b. \quad (6.5.1)$$

Here A is an $m \times n$ matrix, x is an n -vector, and b is an m -vector.

We shall solve this system of equations again in terms of the generalised inverse of A .

THEOREM:

Let A^- be any generalized inverse of the coefficient matrix A in (6.5.1). Then (6.5.1) is consistent if and only if,

$$AA^{-1}b = b.$$

In which case the most general solution is,

$$x = A^-b + (I - A^-A)z, \quad (6.5.2)$$

where z is an arbitrary n -vector, and I the identity, [Gregory, Krishnamurthy, 1984].

REMARK: If the system of equations is homogeneous, that means if $b=0$, the x becomes,

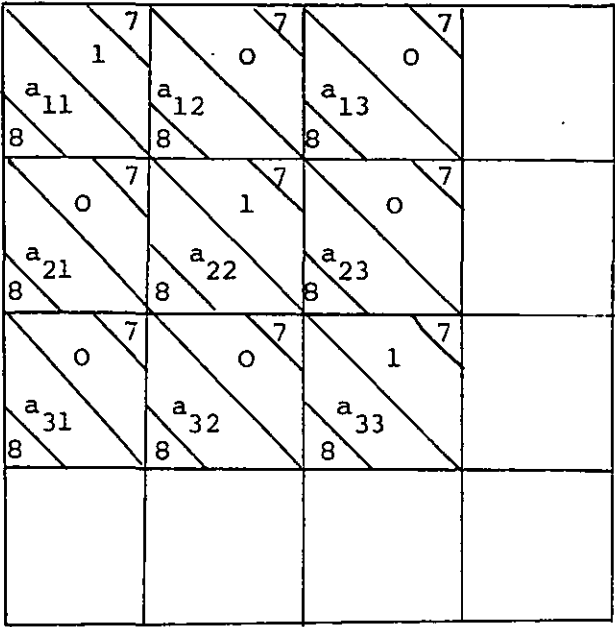
$$x = (I - A^-A)z. \quad (6.5.3)$$

If A is a square matrix and nonsingular, $A^- = A^{-1}$ and, in this special case $x=0$ is the only solution.

6.5.1 The Solution of a Homogeneous System of Equations

First we will solve the equation in (6.5.3) in the case of the system of equation being homogeneous, and $n=3$ and $m=4$. To write a RISAL program for calculating the value of x , we implemented the following steps:

- Step 1: Calculate the generalized inverse of the matrix A by using the same steps used in the previous Section 6.4.
- Step 2: Multiply the generalized inverse A^{\sim} by the matrix A by using the same concepts used previously in paragraph 6.2.4. The result will be a matrix. Move each element of this matrix to register 8 of the processing elements of the ISA grid.
- Step 3: Read the identity matrix I from the north into the ISA grid, and move the elements of the identity into register 7 of the processing elements of the ISA grid, as shown in Figure 6.12.
- Step 4: Subtract the value in register 8 from the value in register 7, and move the result into register 9.
- Step 5: As shown in Figure 6.13, read z_1 from the north throughout the first column of the ISA grid, multiply the value of z_1 by the value in register 9, and move the result into the communication register. Then read them by the second column of the ISA grid, so that they will be held in register 6 (west data input register).
- Read z_2 from the north throughout the second column of the ISA grid, and multiply the value of z_2 by the value in register 9 of this column, and add the result of multiplication to the value in register 6. Move the result to the communication register to read them by the third column of the ISA grid (register 6).



4x4 ISA GRID

FIGURE 6.12: The Identity Elements and the Matrix A Elements Stored in Register 7 and 8 of the ISA Grid.

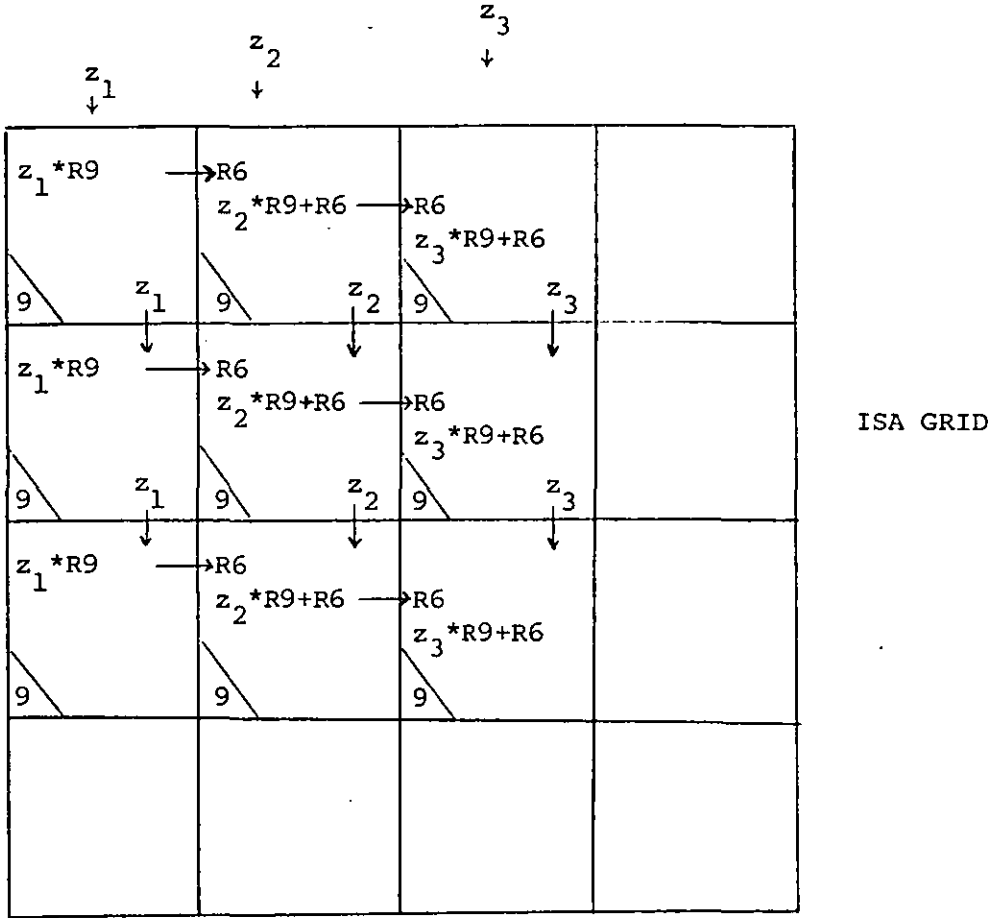


FIGURE 6.13: Determination of the Values of x's (Step 4)

- Read z_3 again from the north throughout the third column, and multiply the value of z_3 by the values held in register 9. Add the result to the values held in register 6, and move the result into the communication register.

Step 6: Read the values in the communication register of the processing elements P_{13}, P_{23} and P_{33} from the south to the north of the ISA grid. The values of x_1, x_2 and x_3 which is the solution of a homogeneous system of equations.

EXAMPLE:

Given the linear system of equation in (6.5.1) where,

$$A = \begin{bmatrix} 1 & 4 & 3 \\ 2 & 0 & 1 \\ 3 & 2 & 2 \\ 4 & 1 & 1 \end{bmatrix}$$

By implementing Step 1 of the RISAL program, the generalized inverse of the matrix A is,

$$A^- = \begin{bmatrix} -0.098039 & -0.098039 & 0.054902 & 0.0282353 \\ 0.176471 & -0.823529 & -0.058824 & 0.411764 \\ 0.098039 & 1.098039 & 0.145098 & -0.682353 \end{bmatrix}$$

By implementing Step 2 in the RISAL program (A^-A) , we obtain

$$\begin{bmatrix} 1.000024 & 0.000036 & 0.000036 \\ 0.000027 & 1.000008 & 0.000017 \\ 0.000011 & 0.000002 & 1.000002 \end{bmatrix}$$

By implementing Step 3 and Step 4 of the RISAL program $(I-A^-A)$, we obtain,

$$\begin{bmatrix} -0.000024 & -0.000036 & -0.000036 \\ -0.000027 & -0.000008 & -0.000017 \\ -0.000011 & -0.000002 & -0.000002 \end{bmatrix}$$

Now, if $b=0$, and vector $z = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$, by substituting in equation (6.5.3)

and implementing Step 5 and Step 6 of the RISAL program, we obtain

$$x = \begin{bmatrix} -0.000024 & -0.000036 & -0.000036 \\ -0.000027 & -0.000008 & -0.000017 \\ -0.000011 & -0.000002 & -0.000002 \end{bmatrix} * \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

$$x = \begin{bmatrix} -0.000060 \\ -0.000044 \\ -0.000013 \end{bmatrix}$$

6.5.2 The Most General Solution of a System of Equations

Now we turn to the most general solution of a system of equations mentioned earlier in (6.5.2). To write a RISAL program to determine the value of x in (6.5.2) we implemented the following steps:

Step 1: Implement Step 1-Step 5 in the previous paragraph 6.5.1, and that means we have calculated the generalized inverse of the matrix A , and the value of x 's in case of $b=0$ (homogeneous system of equations), at the end of Step 5 mentioned above, the value of x 's are stored in the communication register of the processing elements, P_{13}, P_{23} , and P_{33} .

- Move the value of x 's to register 10 of P_{13}, P_{23} and P_{33} .

- Step 2: Read the generalized inverse of A into the ISA grid, and move its elements to register 7 of the processing elements.
- Step 3: As shown in Figure 6.14, and by implementing the same concept in Step 5, in 6.5.1 by reading the values of the vector b from the north, instead of the vector z. At the end of the multiplication process of $(A^{-1}b)$, add the result of this multiplication to register 10 in P_{12}, P_{23}, P_{33} and move the result into the communication registers of P_{14}, P_{24} and P_{34} .
- Step 4: Read the value of x's from the communication register of P_{14}, P_{24} and P_{34} from the south to the north of the ISA grid. These values are the most general solution of a system of equations in terms of the generalized inverse.

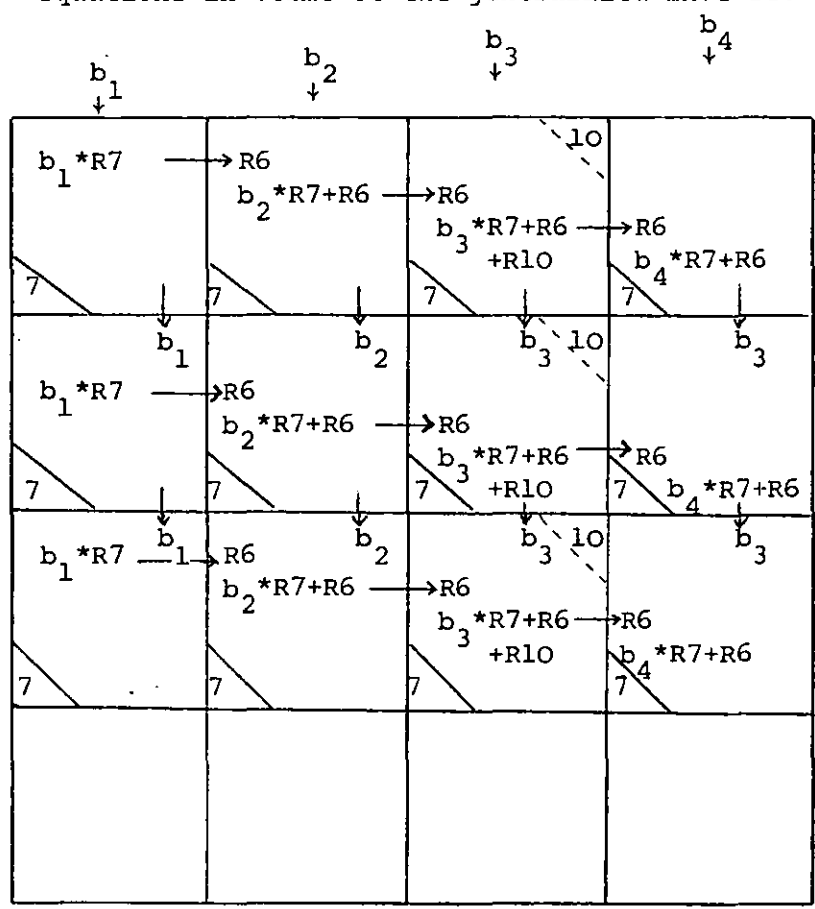


FIGURE 6.14: Determination of the value of x's (Step 3)

EXAMPLE:

Given the system of equations in (6.5.1), where,

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

By implementing Step 1 of the RISAL program, and by applying (6.4.6), we obtain,

$$\begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} * x = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

From (6.4.7)

$$\begin{bmatrix} 2 & 1 & 2 \\ 1 & 2 & 2 \\ 2 & 2 & 4 \end{bmatrix} * x = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

By factorizing, we obtain,

$$\begin{bmatrix} 1 & & & \\ 0.5 & 1 & & \\ 1 & 0.666667 & 1 & \end{bmatrix} \begin{bmatrix} 2 & 1 & 2 \\ & 1.5 & 1 \\ & & 1.333333 \end{bmatrix} * x = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

By applying equation (6.4.9), we obtain,

$$\begin{bmatrix} 1 & & & \\ 0.5 & 1 & & \\ 1 & 0.666667 & 1 & \end{bmatrix} \begin{bmatrix} y_{11} & y_{12} & y_{13} & y_{14} \\ y_{21} & y_{22} & y_{23} & y_{24} \\ y_{31} & y_{32} & y_{33} & y_{34} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

$$\begin{bmatrix} y_{11} & y_{12} & y_{13} & y_{14} \\ y_{21} & y_{22} & y_{23} & y_{24} \\ y_{31} & y_{32} & y_{33} & y_{34} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ -0.5 & 1 & 0.5 & 0 \\ 0.333333 & 0.333333 & -0.333333 & 1 \end{bmatrix}$$

By applying equation (6.4.8), we obtain,

$$\begin{bmatrix} 2 & 1 & 2 \\ & 1.5 & 1 \\ & & 1.333333 \end{bmatrix} \begin{bmatrix} x_{11} & x_{12} & x_{13} & x_{14} \\ x_{21} & x_{22} & x_{23} & x_{24} \\ x_{31} & x_{32} & x_{33} & x_{34} \end{bmatrix} \begin{bmatrix} 1 & 0 & 1 & 0 \\ -0.5 & 1 & 0.5 & 0 \\ 0.333333 & 0.333333 & -0.333333 & 1 \end{bmatrix}$$

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} & x_{14} \\ x_{21} & x_{22} & x_{23} & x_{24} \\ x_{31} & x_{32} & x_{33} & x_{34} \end{bmatrix} = \begin{bmatrix} 0.5 & -0.5 & 0.5 & -0.5 \\ -0.5 & 0.5 & 0.5 & -0.5 \\ 0.25 & 0.25 & -0.25 & 0.75 \end{bmatrix}$$

which is the generalized inverse of the matrix A. To verify this result by multiplying $A^{-}A$, we obtain,

$$\begin{bmatrix} 0.5 & -0.5 & 0.5 & -0.5 \\ -0.5 & 0.5 & 0.5 & -0.5 \\ 0.25 & 0.25 & -0.25 & 0.75 \end{bmatrix} \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

To calculate the value of x in case of $b=0$, and $z = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$, we obtain,

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} - \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Multiplying by the vector z, we obtain,

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix},$$

The value of x , where $b = \begin{bmatrix} 2 \\ 2 \\ 2 \\ 2 \end{bmatrix}$, and $z = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$, is obtained by

implementing Step 2, Step 3 and Step 4 of the RISAL program. Finally we obtain,

$$\begin{bmatrix} 0.5 & -0.5 & 0.5 & -0.5 \\ -0.5 & 0.5 & 0.5 & -0.5 \\ 0.25 & 0.25 & -0.25 & 0.75 \end{bmatrix} \begin{bmatrix} 2 \\ 2 \\ 2 \\ 2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 2 \end{bmatrix}$$

$$\begin{bmatrix} 0 \\ 0 \\ 2 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 2 \end{bmatrix}$$

which is the most general solution of a system of equations.

6.6 DELETION FROM A HEAP SORT USING SSSS

The heap sort algorithm uses a data structure called a heap, which is a binary tree with some special properties. The definition of a heap includes a description of the structure and a condition on the data in the nodes. Informally, a heap structure is a complete binary tree with some of the rightmost leaves removed. (See Figure 6.15 for illustrations).

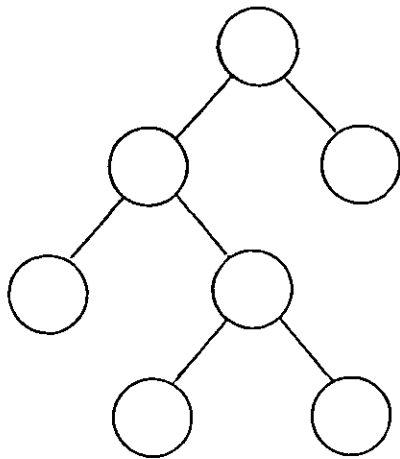
DEFINITION:

Let S be a set of keys with a linear ordering and T be a binary tree with depth d whose nodes contain the elements of S . T is a heap if and only if it satisfies the following conditions:

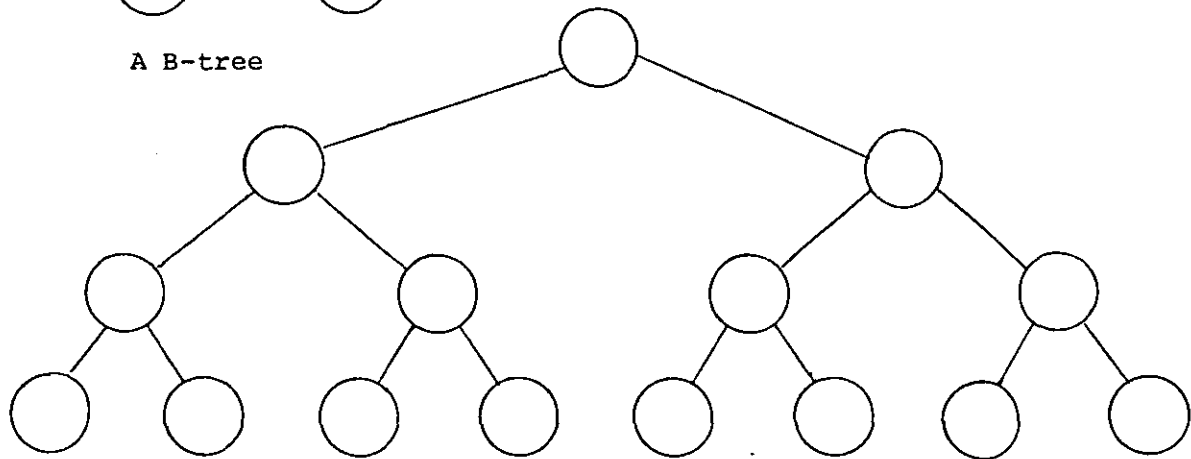
1. All internal nodes (with one possible exception) have degree 2, and at level $d-1$ the leaves are all to the right of the internal nodes. The rightmost internal node at level $d-1$ may have degree 1 (with no right child).
2. The key at any node is greater than or equal to the keys at each of its children (if it has any).

We will use the term heap structure to describe a binary tree that satisfies condition (1). Observe that a complete binary tree is a heap structure. When new nodes are added to a heap they must be added left to right at the bottom level, and if a node is removed, it must be the rightmost node at the bottom level if the resulting structure is still to be a heap. Note that the root must contain the largest key in the heap.

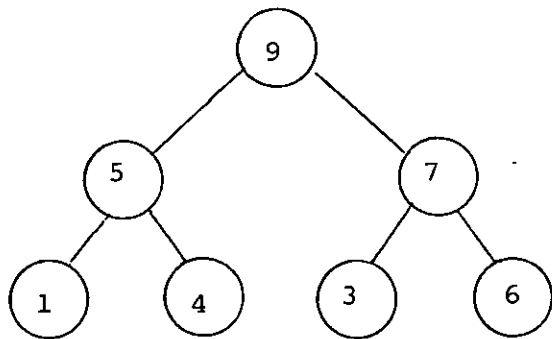
Deletion from the heap means removing the key at the root, the largest key in the heap, and rearranging the nodes so that the heap



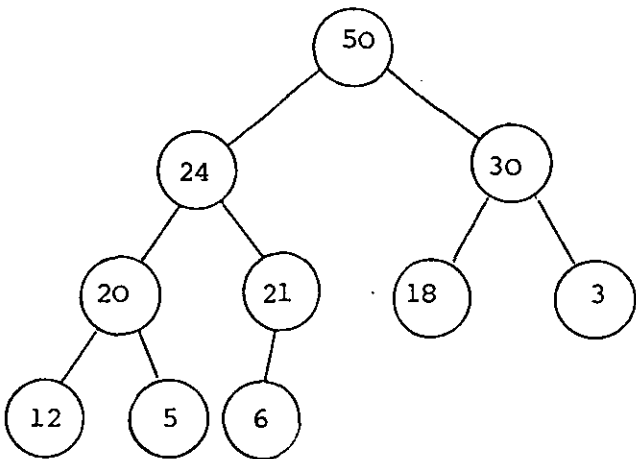
A B-tree



A Complete Binary Tree



Heap 1



Heap 2

FIGURE 6.15: B-Tree, Complete Binary Tree, and Heaps

properties are still satisfied. Structurally, the node to be removed is the rightmost leaf at the bottom level. The key, say, K , from that leaf must be placed elsewhere. The only vacant node is the root, so we begin there and let the key K filter down to its correct position. At its final position, K must be greater than or equal to each of its children, so at each step K is compared to the larger of the children of the currently vacant node. If K is larger (or equal) it can be inserted, otherwise the larger child is moved up to the vacant node and the process is repeated, [Baase, 1978].

The deletion algorithm assumes that there are at least two nodes in the heap. The algorithm is illustrated in Figure 6.16, by taking the heap 2 in Figure 6.15 as an example.

To implement the algorithm above using our simulation system, we need first of all to find a way to embed the heap structure onto the ISA grid, and rearranging the nodes so that the heap properties still satisfy the conditions mentioned above.

The binary trees are generally implemented as linked structures; so that the heap can be stored efficiently in an array in such a way that accessing a child of a node is quite easy, e.g. the heap 1 in Figure 6.15 can rearrange its nodes into an H-shape, and then we can embed it into the ISA grid as shown in Figure 6.17.

To write a RISAL program to solve this problem, we implemented the following steps:

Step 1: Read the H-shape structure from the north into the ISA grid.

The root will be held in the processing element P_{22} , and its children in P_{21} and P_{23} , and their leaves in P_{13} , P_{33} , P_{11} and P_{31} respectively.

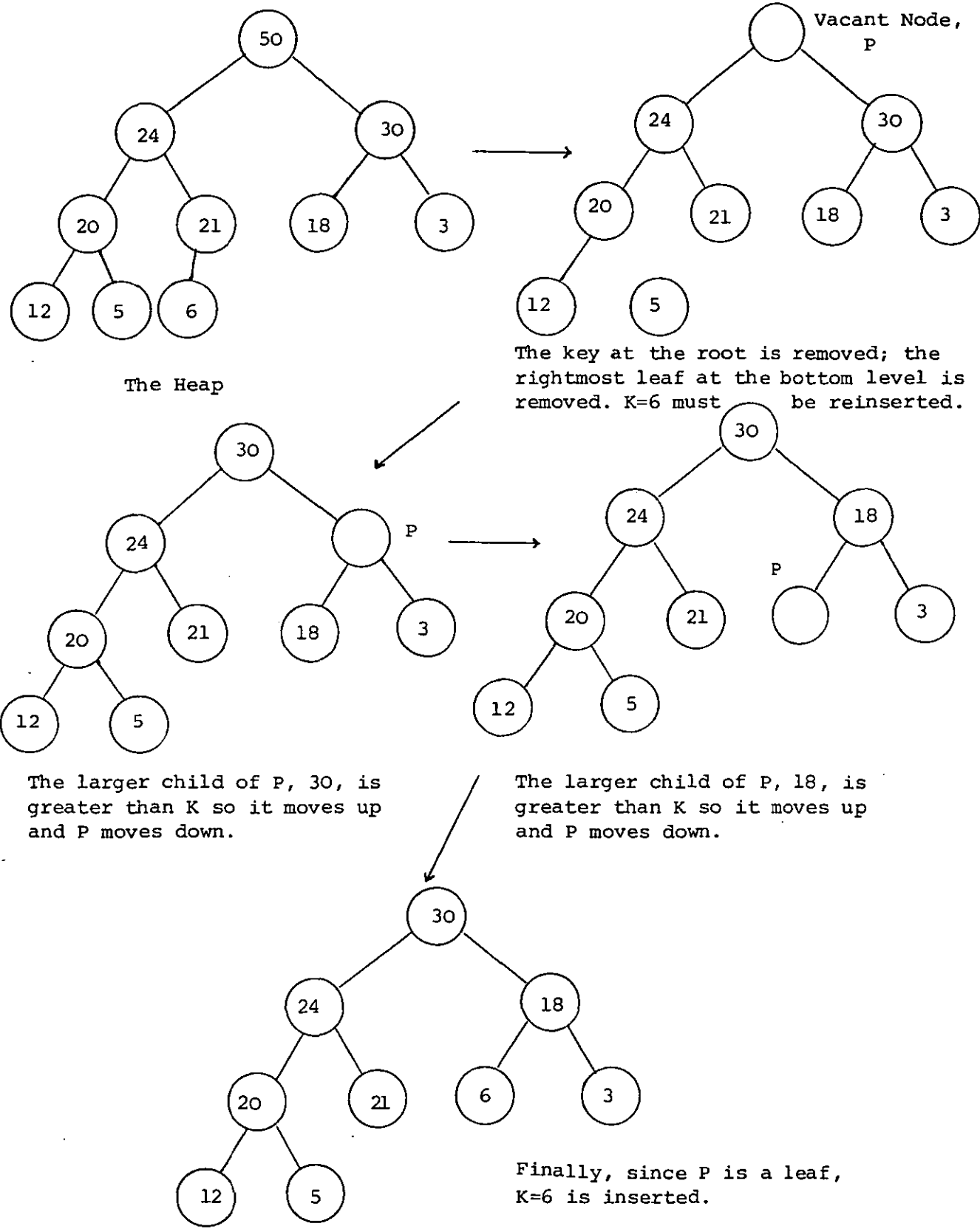


FIGURE 6.16: Deletions from a Heap.

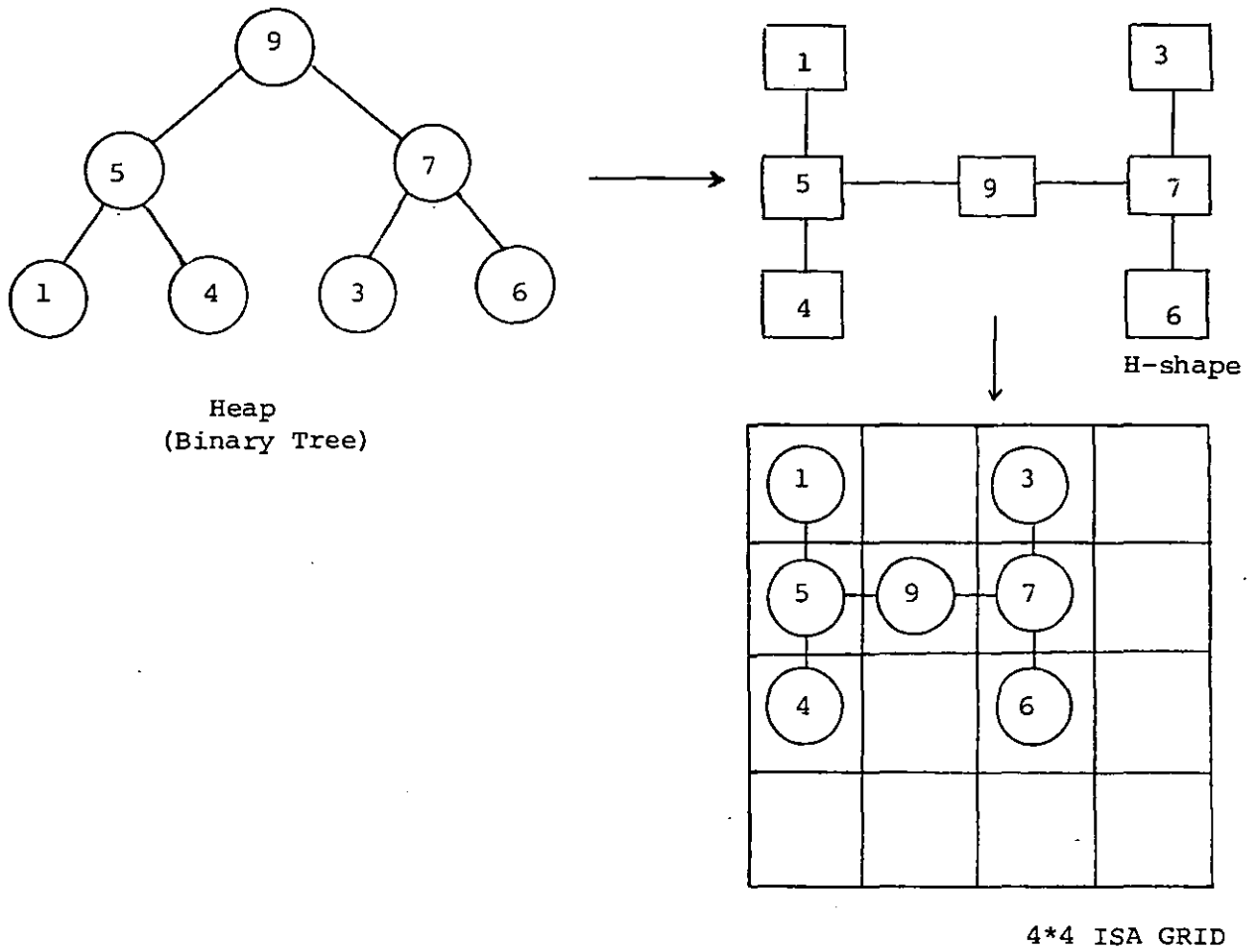


FIGURE 6.17: The H-Shape Embedded into the ISA Grid.

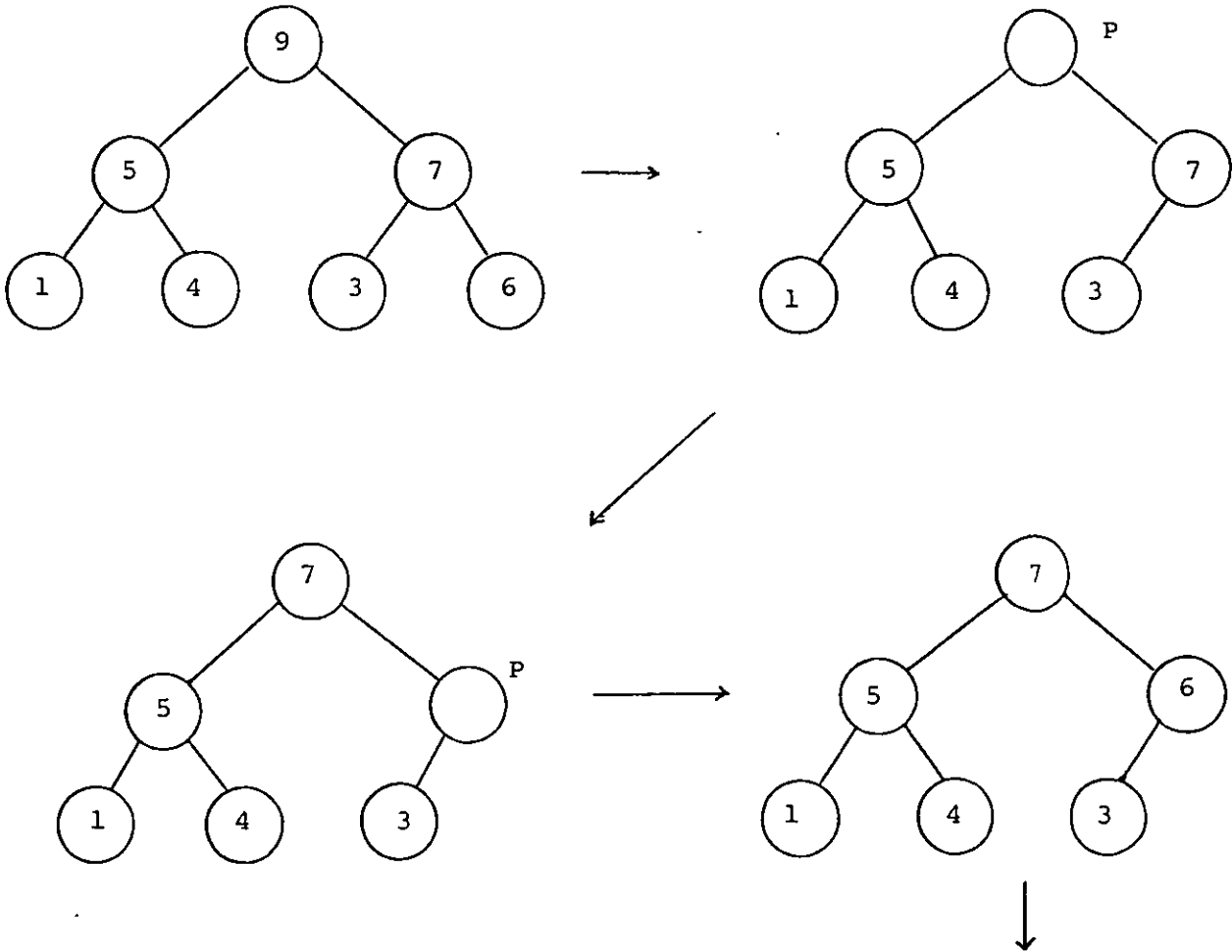
- Step 2:** Read the value in P_{22} to the north of the ISA grid, which represents the key in the root.
- Step 3:** Move the value in P_{23} into P_{22} .
- Step 4:** Compare the value in P_{21} with the value in P_{22} , and store the largest in P_{22} .
- Step 5:** Move the value in P_{33} into P_{23} .
- Step 6:** Compare the value in P_{22} with the value in P_{23} , and store the largest in P_{22} , and then implement steps 2,3 and 4.
- Step 7:** Move the value in P_{13} into P_{23} , and implement Step 6 and Step 2.

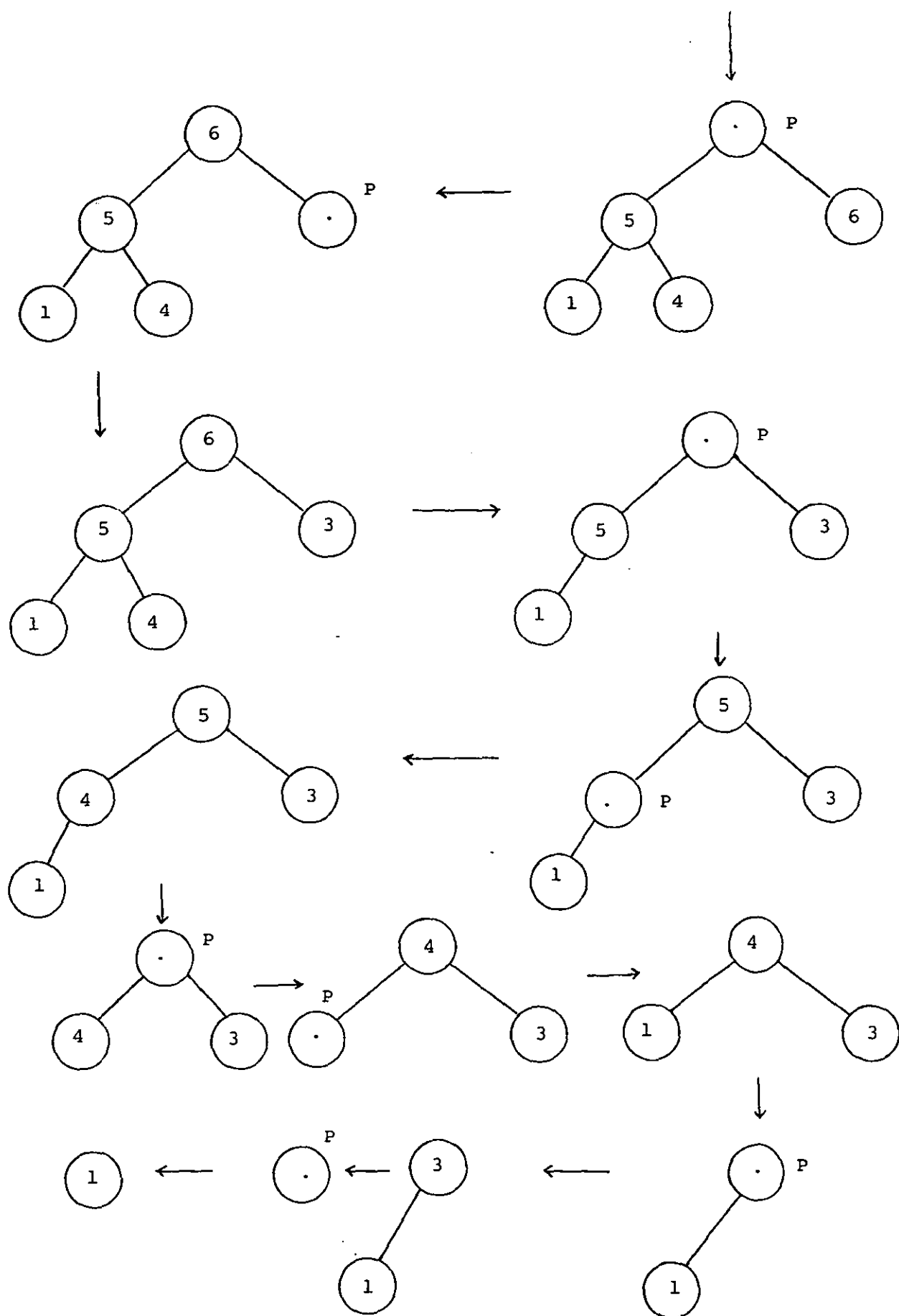
Step 8: Move the value in P_{21} into P_{22} , and implement Step 6.

Step 9: Move the value in P_{31} into P_{21} , and implement Steps 4,2,9, and 6 respectively.

Step 10: Move the value in P_{11} into P_{21} , and implements Steps 4,2,3,4, 2,9, and 2 respectively.

By using the RISAL program above, the deletion process from heap 1 shown in Figure 6.17, is illustrated below:





6.7 HERMITE POLYNOMIAL INTERPOLATION AND EVALUATION USING SSSS

Interpolation is the process of "reading between the lines" of a table or the fitting of a smooth curve to a limited set of data. We take it up first for a number of reasons, the most obvious of which is that interpolation is frequently used for estimating quantities from tabulated data. A more important reason is that many numerical differentiation and integration procedures are derived by using interpolation to find a smooth approximation and then differentiating or integrating the result.

There are two kinds of interpolation, depending on the type of data provided and the kind of result wanted. In the standard type of interpolation we are given a set of data points and require a curve that passes smoothly through them. In least squares interpolation generally the data has some uncertainty associated with them and we want to find a smooth curve that passes sufficiently near the data points. In standard interpolation the equation of the approximation curve must have as many parameters as there are data points; in least squares fitting the number of parameters typically is much smaller than the number of data points.

The basic problem of interpolation may be stated as follows. Given a set of data (x_i, y_i) , $i=1,2,\dots,n$, find a smooth curve $f(x)$ that passes through the data. We require the following criteria of the interpolating curve.

1. From the problem statement, we must have,

$$f(x_i) = y_i, \quad i=1,2,\dots,n,$$

2. The function should be easy to evaluate.
3. It should also be easy to integrate and differentiate.

4. It should be linear in the adjustable parameters (to simplify the problem of finding them).

The choice of the interpolating function depends on what one means by smoothness and on the function to be approximated. Many functions have been used, the most common of which are polynomials of various kinds because they satisfy criteria (2) and (3) above better than any other type of function. Even among polynomial interpolations there are a number of classes, i.e. Lagrange interpolation, Taylor interpolation, Hermite interpolation and others.

Our aim in this section is to solve Hermite interpolation problem, but first we will begin with the simplest class, the Lagrange form which leads us to calculate the Hermite interpolation form.

In Lagrange we consider the problem of determining a polynomial of degree 1 which passes through the distinct points (x_0, y_0) and (x_1, y_1) . This problem is the same as approximating a function f , for which $f(x_0) = y_0$ and $f(x_1) = y_1$ by means of a first-degree polynomial interpolating, or agreeing with, the values of f at the given points. See Figure 6.18.

If

$$p(x) = a_0 + a_1 x, \quad (6.7.1)$$

is the polynomial, then a_0 and a_1 must satisfy,

$$y_0 = p(x_0) = a_0 + a_1 x_0,$$

$$y_1 = p(x_1) = a_0 + a_1 x_1,$$

Solving these equations for a_0 and a_1 , we obtain:

$$a_1 = \frac{y_0 - y_1}{x_0 - x_1}$$

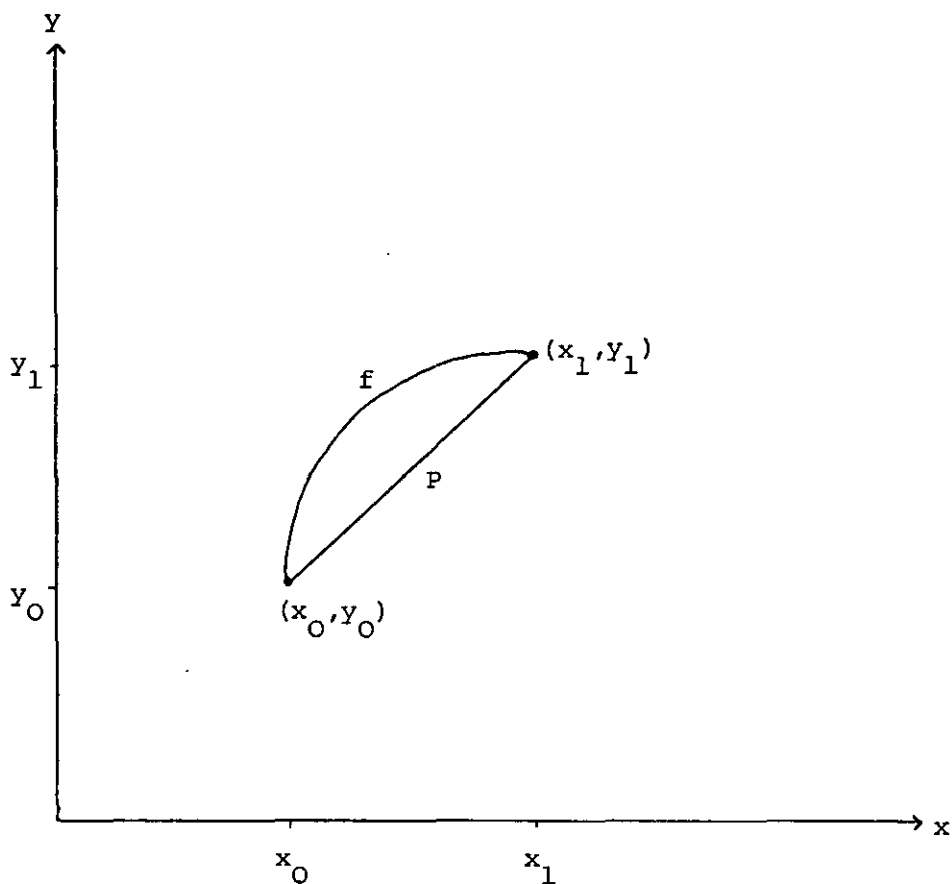


FIGURE 6.18: The Values of f at Points (x_0, y_0) and (x_1, y_1) .

and
$$a_0 = y_1 - a_1 x_1 = y_1 - \left(\frac{y_0 - y_1}{x_0 - x_1} \right) x_1 .$$

Substituting the values of a_0 and a_1 into equation (6.7.1) we obtain,

$$\begin{aligned} p(x) &= y_1 - \left(\frac{y_0 - y_1}{x_0 - x_1} \right) x_1 + \left(\frac{y_0 - y_1}{x_0 - x_1} \right) x \\ &= \frac{y_1(x_0 - x_1) - x_1(y_0 - y_1) + x(y_0 - y_1)}{x_0 - x_1} \end{aligned}$$

$$\begin{aligned}
&= \frac{y_1(x_0 - x_1) - x_1(y_0 - y_1) + x(y_0 - y_1)}{x_0 - x_1} \\
&= \frac{y_1(x_0 - x_1 + x_1 - x)}{x_0 - x_1} + \frac{y_0(-x_1 + x)}{x_0 - x_1} \\
&= \frac{(x - x_1)}{(x_0 - x_1)} y_0 + \frac{(x - x_0)}{(x_1 - x_0)} y_1 .
\end{aligned}$$

To generalize the concept of linear interpolation, consider finding a polynomial of degree at most n which passes through $(n+1)$ given points. This can be viewed as an approximation technique in that, given a function f , we find a polynomial P which agrees with the values of the function at certain specified points, and the polynomial P is then used to approximate f at other points.

THEOREM:

If x_0, x_1, \dots, x_n are $(n+1)$ distinct points and f is a function whose degree is at most n , with the property that,

$$f(x_k) = p(x_k) \text{ for each } k=0,1,\dots,n$$

this polynomial is given by,

$$p(x) = f(x_0)L_{n,0}(x) + \dots + f(x_n)L_{n,n}(x) \quad (6.7.2)$$

$$= \sum_{k=0}^n f(x_k)L_{n,k}(x) ,$$

where,

$$L_{n,k}(x) = \frac{(x-x_0)(x-x_1)\dots(x-x_{k-1})(x-x_{k+1})\dots(x-x_n)}{(x_k-x_0)(x_k-x_1)\dots(x_k-x_{k-1})(x_k-x_{k+1})\dots(x_k-x_n)} \quad (6.7.3)$$

$$= \prod_{\substack{i=0 \\ i \neq k}}^n \frac{(x-x_i)}{(x_k-x_i)} , \text{ for each } k=0,1,\dots,n \quad (6.7.4)$$

We will write $L_{n,k}(x)$ simply as $L_k(x)$ when there can be no confusion as to its degree. Proof of this theorem is given in [Burden, Faires, Reynolds, 1978].

There are occasions when we want smoothness beyond that provided by the Lagrange interpolation. Such smoothness can be obtained in a number of ways. One of the simplest is to provide not only the value of the function at each point, but also the value of its derivative. We then have Hermite polynomial interpolation.

THEOREM:

If $f \in C'[a,b]$ and $x_0, \dots, x_n \in [a,b]$ are distinct, the unique polynomial of least degree agreeing with f and f' at x_0, \dots, x_n is given by,

$$H_{2n+1}(x) = \sum_{j=0}^n f(x_j) H_{n,j}(x) + \sum_{j=0}^n f'(x_j) \hat{H}_{n,j}(x), \quad (6.7.5)$$

where,

$$H_{n,j}(x) = [1 - 2(x - x_j)L'_{n,j}(x_j)] L_{n,j}^2(x), \quad (6.7.6)$$

and,

$$\hat{H}_{n,j}(x) = (x - x_j) L_{n,j}^2(x) \quad (6.7.7)$$

In this context, $L_{n,j}$ denotes the j th Lagrange coefficient polynomial of degree n defined by (6.7.3). Moreover,

if $f \in C^{(2n+2)}[a,b]$ then,

$$f(x) - H_{2n+1}(x) = \frac{(x - x_0)^2 \dots (x - x_n)^2}{(2n+2)!} f^{(2n+2)}(\bar{\xi})$$

for some point $\bar{\xi}$, with $a < \bar{\xi} < b$. The proof of this theorem is given in [Burden, Faires, Reynolds, 1978].

To solve the Hermite polynomial interpolation by using our simulation system, we consider the polynomial of least degree which agrees with the data listed in the table below for the Bessel function of the first kind of order zero, to find an approximation of $f(x)$.

k	x_k	$f(x_k)$	$f'(x_k)$
0	x_0	$f(x_0)$	$f'(x_0)$
1	x_1	$f(x_1)$	$f'(x_1)$
2	x_2	$f(x_2)$	$f'(x_2)$

By substituting in equation (6.7.3) we obtain,

$$L_{2,0}(x) = \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)}$$

$$L'_{2,0}(x) = \frac{(x-x_2)+(x-x_1)}{(x_0-x_1)(x_0-x_2)}$$

$$L_{2,1}(x) = \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)}$$

$$L'_{2,1}(x) = \frac{(x-x_0)+(x-x_2)}{(x_1-x_0)(x_1-x_2)}$$

$$L_{2,2}(x) = \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}$$

$$L'_{2,2}(x) = \frac{(x-x_0)+(x-x_1)}{(x_2-x_0)(x_2-x_1)}$$

By substituting in equation (6.7.6) we obtain,

$$H_{2,0}(x) = [1-2(x-x_0)L'_{2,0}(x)]L_{2,0}^2(x)$$

$$H_{2,1}(x) = [1-2(x-x_1)L'_{2,1}(x)]L_{2,1}^2(x)$$

$$H_{2,2}(x) = [1-2(x-x_2)L'_{2,2}(x)]L_{2,2}^2(x)$$

Further by substituting in equation (6.7.7), we obtain,

$$\hat{H}_{2,0}(x) = (x-x_0)L_{2,0}^2(x)$$

$$\hat{H}_{2,1}(x) = (x-x_1)L_{2,1}^2(x)$$

$$\hat{H}_{2,2}(x) = (x-x_2)L_{2,2}^2(x) \quad .$$

Finally by substituting in equation (6.7.5), we obtain,

$$\begin{aligned} H_5(x) = & f(x_0)H_{2,0}(x) + f(x_1)H_{2,1}(x) + f(x_2)H_{2,2}(x) \\ & + f'(x_0)\hat{H}_{2,0}(x) + f'(x_1)\hat{H}_{2,1}(x) + f'(x_2)\hat{H}_{2,2}(x) \end{aligned}$$

The value of $H_5(x)$ is accurate to the places listed above.

To write a RISAL program to calculate the value of $H_{5,0}(x)$, we implemented the following steps:

Step 1: First we determine the values of $L_{2,0}(x)$, $L_{2,1}(x)$ and $L_{2,2}(x)$ in the processing element P_{11} , and $L'_{2,0}(x)$, $L'_{2,1}(x)$, and $L'_{2,2}(x)$ in the processing element P_{12} , as follows:

- Read the values of x , x_0 , x_1 , and x_2 from the north into the P_{11} and P_{12} of the ISA grid, and store them in register 7,8,9, and 10 respectively. We obtain,
- (x_0-x_1) and (x_0-x_2) by subtracting the values in registers 9 and 10 from the value in register 8, and store the results in registers 11 and 12.
- $(x_0-x_1)(x_0-x_2)$ by multiplying the values in registers 11 and 12, and store the result in register 13.
- (x_1-x_0) and (x_1-x_2) by subtracting the values in registers 8 and 10 from the value in register 9, and store the result in registers 14 and 15.

- $(x_1 - x_0)(x_1 - x_2)$ by multiplying the values in registers 14 and 15, and store the result in register 16.
- $(x_2 - x_0)$ and $(x_2 - x_1)$ by subtracting the values in register 8 and 9 from the value in register 10, and store the results in registers 17 and 18.
- $(x_2 - x_0)(x_2 - x_1)$ by multiplying the values in registers 17 and 18 and store the result in register 19.

Step 2: In P_{11} , we obtain:

- $(x - x_1)$ and $(x - x_2)$ by subtracting the values in registers 9 and 10 from the value in register 7 and store the results in registers 11 and 12.
- $(x - x_1)(x - x_2)$ by multiplying the values in registers 11 and 12, and store the result in 14.
- $(x - x_0)$ by subtracting the value in register 8 from the value in register 7, and store the result in register 15.
- $(x - x_0)(x - x_2)$ by multiplying the values in registers 15 and 12, and store the result in register 17.
- $(x - x_0)(x - x_1)$ by multiplying the values in register 15 and 11, and store the result in register 18.
- $L_{2,0}(x)$ - by dividing the value in register 14 by the value in register 13, and store the result in 13.
- $L_{2,1}(x)$ by dividing the value in register 17 by the value in register 16, and store the result in 14.
- $L_{2,2}(x)$ by dividing the value in register 18 by the value in register 19, and store the result in 16.
- $L_{2,0}^2(x)$ by multiplying the value in register 13 by itself, and store the result in 13.

- $L_{2,1}^2(x)$ by multiplying the value in register 14 by itself, and store the result in 14.
- $L_{2,2}^2(x)$ by multiplying the value in register 16 by itself, and store the result in 16.

Step 3: In P_{12} , we obtain:

- $(x_0 - x_2)$ and $(x_0 - x_1)$ by the subtraction of the values in registers 10 and 9 from the value in register 8 and store the results in registers 11 and 12.
- $(x_0 - x_2) + (x_0 - x_1)$ by the addition of the values in register 11 and 12, and store the result in register 14.
- $(x_1 - x_2)$ and $(x_1 - x_0)$ by the subtraction of the values in register 10 and 8 from the value in register 9, and store the results in registers 15 and 17.
- $(x_1 - x_2) + (x_1 - x_0)$ by the addition of the values in registers 15 and 17, and store the result in register 18.
- $(x_2 - x_1)$ and $(x_2 - x_0)$ by the subtraction of the values in registers 9 and 8 from the value in register 10, and store the results in registers 11 and 12.
- $(x_2 - x_1) + (x_2 - x_0)$ by the addition of the values in registers 11 and 12, and store the result in register 17.
- $L_{2,0}'(x)$ by the division of the value in register 14 by the value in register 13.
- $L_{2,1}'(x)$ by the division of the value in register 18 by the value in register 16.
- $L_{2,2}'(x)$ by the division of the value in register 17 by the value in register 19.

At this stage of the RISAL program, the values of $L_{2,0}^2(x)$, $L_{2,1}^2(x)$, and $L_{2,2}^2(x)$ are held in the processing element P_{11} , and the values of $L'_{2,0}(x)$, $L'_{2,1}(x)$ and $L'_{2,2}(x)$ are held in the processing element P_{12} .

Step 4: To determine the values of $H_{2,0}(x)$, $H_{2,1}(x)$, $H_{2,2}(x)$, $\hat{H}_{2,0}(x)$, $\hat{H}_{2,1}(x)$ and $\hat{H}_{2,2}(x)$:

- the values in registers 13, 14 and 16 of P_{12} were moved to registers 17, 18 and 19 respectively of P_{11} .
- read 1 and 2 from the north into register 7 and 8 of P_{11} .

Step 5: In P_{11} , we obtain:

- $2(x-x_0)$ by the multiplication of the value in register 8 by the value in register 15, and store the result in register 9.
- $2(x-x_0)L'_{2,0}(x)$ by the multiplication of the value in register 9 by the value in register 17, and store the result in register 9.
- $2(x-x_1)$ by the multiplication of the value in register 8 by the value in register 11, and store the result in register 10.
- $2(x-x_0)L'_{2,1}(x)$ by multiplying the value in register 10 by the value in register 18, and store the result in register 10.
- $2(x-x_2)$ by the multiplication of the value in register 8 by the value in register 12, and store the result in register 17.
- $2(x-x_2)L'_{2,2}(x)$ by the multiplication of the value in register 17 by the value in register 19, and store the result in register 17.

- $[1-2(x-x_0)L'_{2,0}(x_0)]$ by the subtraction of the value in register 9 from the value in register 7, and store the result in register 9.
- $[1-2(x-x_1)L'_{2,1}(x_1)]$ by the subtraction of the value in register 10 from the value in register 7, and store the result in register 10.
- $[1-2(x-x_2)L'_{2,2}(x_2)]$ by the subtraction of the value in register 17 from the value in register 7, and store the result in register 17.
- $[1-2(x-x_0)L'_{2,0}(x_0)]L_{2,0}^2(x)$ by the multiplication of the value in register 9 by the value in register 13, and store the result in register 7, which is equal to $H_{2,0}(x)$.
- $[1-2(x-x_1)L'_{2,1}(x_1)]L_{2,1}^2(x)$ by the multiplication of the value in register 10 by the value in register 14, and store the result in register 8, which is equal to $H_{2,1}(x)$.
- $[1-2(x-x_2)L'_{2,2}(x_2)]L_{2,1}^2(x)$ by the multiplication of the value in register 17 by the value in register 16, and store the result in register 9, which is equal to $H_{2,2}(x)$.
- $(x-x_0)L_{2,0}^2(x)$ by the multiplication of the value in register 15 by the value in register 13, and store the result in register 10, which is equal to $\hat{H}_{2,0}(x)$.
- $(x-x_1)L_{2,1}^2(x)$ by the multiplication of the value in register 11 by the value in register 14, and store the result in register 11, which is equal to $\hat{H}_{2,1}(x)$.
- $(x-x_2)L_{2,2}^2(x)$ by the multiplication of the value in register 12 by the value in register 16 and store the result in register 12, which is the value of $\hat{H}_{2,2}(x)$.

At this stage of the RISAL program the values of $H_{2,0}(x)$, $H_{2,1}(x)$, $H_{2,2}(x)$, $\hat{H}_{2,0}(x)$, $\hat{H}_{2,1}(x)$ and $\hat{H}_{2,2}(x)$ are held in the processing element P_{11} in registers 7,8,9,10,11,12 and 13 respectively.

By reading the values of $f(x_0)$, $f(x_1)$, $f(x_2)$, $f'(x_0)$, $f'(x_1)$ and $f'(x_2)$ from the north of the ISA grid and store them in P_{11} in registers 13,14,15,16,17 and 18 respectively, we obtain;

- $f(x_0)H_{2,0}(x)$ by the multiplication of the value in register 13 by the value in register 7, and store the result in register 7.
- $f(x_1)H_{2,1}(x)$ by the multiplication of the value in register 14 by the value in register 8, and store the result in register 8.
- $f(x_2)H_{2,2}(x)$ by the multiplication of the value in register 15 by the value in register 9, and store the result in register 9.
- $f'(x_0)\hat{H}_{2,0}(x)$ by the multiplication of the value in register 16 by the value in register 10, and store the result in register 10.
- $f'(x_1)\hat{H}_{2,1}(x)$ by the multiplication of the value in register 17 by the value in register 11, and store the result in register 11.
- $f'(x_2)\hat{H}_{2,2}(x)$ by the multiplication of the value in register 18 by the value in register 12, and store the result in register 12.

Finally, to obtain the value of $H_5(x)$, we added the values in registers 7,8,9,10,11 and 12.

EXAMPLE:

Consider the problem described above to find an approximation of $f(1.5)$. The data is:

\underline{k}	$\underline{x_k}$	$\underline{f(x_k)}$	$\underline{f'(x_k)}$
0	1.3	0.620086	-0.522023
1	1.6	0.455402	-0.569895
2	1.9	0.281818	-0.581157

By implementing Step 1, Step 2, and Step 3 of the RISAL program mentioned above, we obtain:

$$L_{2,0}(x) = \frac{2}{9}$$

$$L'_{2,0}(x) = -5$$

$$L_{2,1}(x) = \frac{8}{9}$$

$$L'_{2,1}(x) = 0$$

$$L_{2,2}(x) = -\frac{1}{9}$$

$$L'_{2,2}(x) = 5$$

By implementing Step 4, we obtain:

$$H_{2,0} = \frac{4}{27}$$

$$H_{2,1} = \frac{64}{81}$$

$$H_{2,2} = \frac{5}{81}$$

$$\hat{H}_{2,0} = \frac{4}{405}$$

$$\hat{H}_{2,1} = -\frac{32}{405}$$

$$\hat{H}_{2,2} = -\frac{2}{405}$$

By implementing Step 5, we obtain:

$$\begin{aligned}
 H_5(1.5) &= 0.620086\left(\frac{4}{27}\right) + 0.455402\left(\frac{64}{81}\right) + 0.281818\left(\frac{5}{81}\right) - \\
 &\quad 0.522023\left(\frac{4}{405}\right) - 0.569895\left(-\frac{32}{405}\right) - 0.581157\left(-\frac{2}{405}\right) \\
 &= 0.511827.
 \end{aligned}$$

The result above is accurate to the places listed above.

Parallel Polynomial Evaluation:

To locate approximate roots of a polynomial P , it is necessary to evaluate P and its derivative at specified values. If the n th-degree polynomial $p(x) = p_0 + p_1x + p_2x^2 + \dots + p_nx^n$, therefore, to evaluate $p(x)$, it requires $(2n-1)$ multiplications and n additions.

To write a RISAL program to solve the problem mentioned above, we consider the polynomial,

$$p(x) = p_0 + p_1x + p_2x^2 + p_3x^3 + p_4x^4 + p_5x^5 + p_6x^6 + p_7x^7$$

By rearranging we obtain,

$$= (p_0 + p_1x) + x^2(p_2 + p_3x) + x^4(p_4 + p_5x) + x^6(p_6 + p_7x)$$

Figure 6.19 illustrates the polynomial above in terms of a balanced tree.

First of all, we have to evaluate the powers of $x^2(1, x^2, x^4, x^6)$ and place it in each processor, and evaluate $p_i + p_{i+1}x$ in each processor. To achieve this concept we implemented the following steps:

- Step 1: Read the value of x from the north into the first row of the ISA grid, and store it in register 7.
- In P_{12}, P_{13} and P_{14} , multiply the value in the result register by the value in register 7.

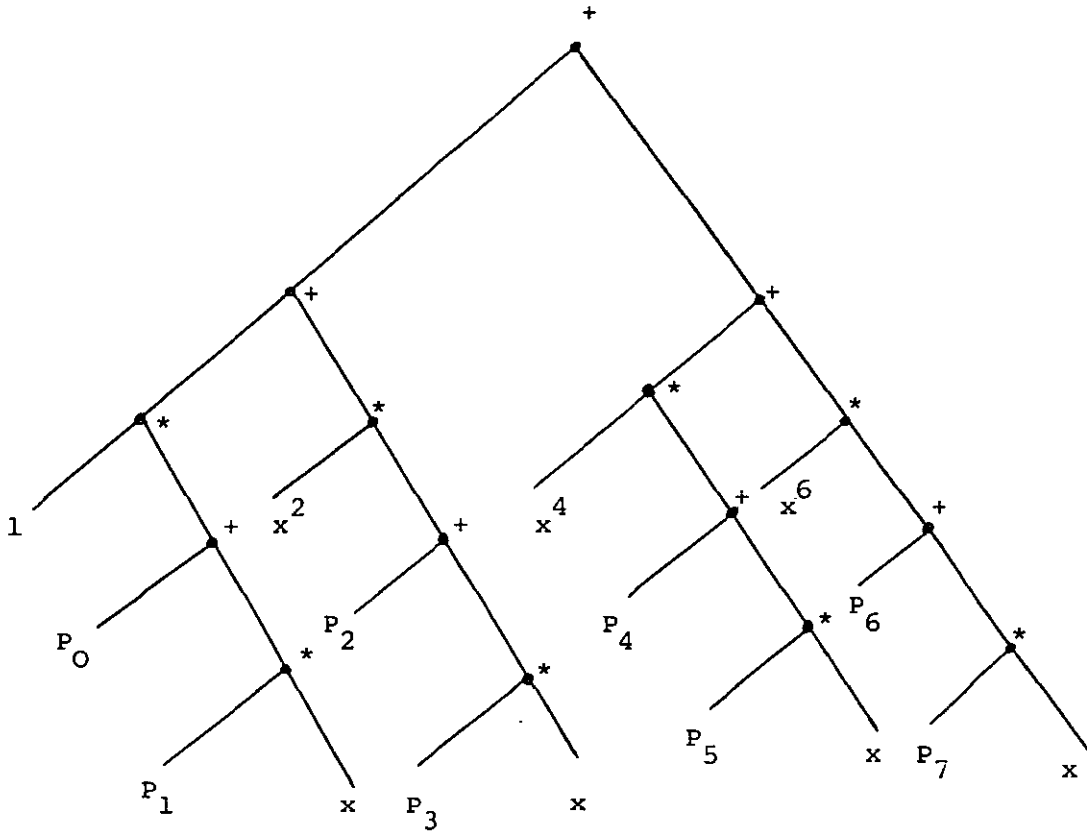


FIGURE 6.19: Balanced Tree Representation of Polynomial Evaluation Equation

- In P_{12} , move the value in the result register into register 8.
- In P_{13} and P_{14} multiply the value in the result register by the value in register 7.
- In P_{13} , move the value in the result register into register 8.
- In P_{14} , multiply the value in the result register by the value in register 7, and move the result into register 8.

Step 2: Read the value of P_1, P_3, P_5 and P_7 from the north (register 3)

- multiply the value in register 3 by the value in register 7, and store the result in register 9.
- read the value of P_0, P_2, P_4 and P_6 from the north (register 3).
- Add the value in register 3 to the value in register 9.
- In P_{12}, P_{13} and P_{14} move the value in the result register into register 10, and multiply the value in register 10 by the value in register 8. Finally, store the result in register 10.

Step 3: Add the value in result register of P_{11} , to the register 10 of P_{12}, P_{13} and P_{14} to obtain the final result.

EXAMPLE:

Given the polynomial,

$$p(x) = P_0 + P_1x + P_2x^2 + P_3x^3 + P_4x^4 + P_5x^5 + P_6x^6 + P_7x^7,$$

where $P_0 = 3, P_1 = 4, P_2 = 5, P_3 = 6, P_4 = 7, P_5 = 8, P_6 = 9, P_7 = 10$, and $x = 2$.

By implementing Step 1, Step 2, and Step 3 of the RISAL program, we obtain that,

$$p(x) = 2303.$$

CHAPTER 7

SUMMARY AND CONCLUSIONS

In this thesis an alternative concept to a VLSI-architecture, the Soft-Systolic Simulation System (SSSS) is introduced and developed as a working model of a virtual machine with the power to simulate hard systolic arrays and more general forms of concurrency such as the SIMD and MIMD models of computation. An overall system structure was defined and the virtual machine discussed in detail. A primitive assembler/compiler for a special language the Replicating Instruction Systolic Array Language (RISAL) was devised for experimentation with the machine.

In the first three introductory chapters, a brief and disciplined state-of-the-art survey was compiled with up-to-date information on the present parallel computing environment.

More analytically, in Chapter 1, we have discussed the main motivations that led to the "parallel way of thinking" and presented several different forms of exploiting this novel idea. Although several attempts (at least three of them were presented in this thesis) have been made to classify these various architectural designs, none of them seem to succeed in providing a clear distinction between the classes since sometimes the intersection of two classes is not empty.

Of the architectures designed for highly parallel processing we presented the pipeline, data-flow computer, and array processors. In the pipeline computer the sequential, vector processing taxonomy, some well known and commercially available computers were discussed.

The data-flow computers are grouped into two classes (static and dynamic), i.e. for the static approach is the MIT data-flow, and for the dynamic approach, the U-interpreter machine and Manchester data-

flow machine are presented. The general SIMD architecture is also presented as an example of the array processor. Also, the inter-connection networks are discussed as a most currently active research area in computer architecture.

In Chapter 2, we presented the VLSI technology as a substantial contender to the achievement of very high-performance, cost effective computing systems for the future decade. We also presented its fundamental concepts such as regularity, planarity, use of pipelining and concurrency, in designing special-purpose and general-purpose computing structures. For the special-purpose class of VLSI-oriented systems we established two main contenders which are the systolic arrays as suggested by H.T. Kung and the wavefront arrays resulting from the work of S.Y. Kung. Although these systems are cost-effective they are however specially designed for one particular class of problems. In order to increase flexibility, the general-purpose computing structures such as the Warp, built by H.T. Kung and the chip of L. Snyder can be used to solve a predefined set of algorithms. Also, a possibility is the Inmos Transputer which is a single chip processor. The Transputer and Occam language were designed in conjunction and all transputers include special instructions and hardware which provides optimal implementations of the Occam model of concurrency and communication. Following these substantial benefits, a research program was carried out in the Department of Computer Studies, at Loughborough University to investigate the simulation of systolic arrays, by using the fact that Occam programs can be divorced from transputer configurations and using the language as a simulation

tool. The general structure of Occam programs which represent the simulation of systolic arrays is introduced, and the techniques described have been used successfully to implement designs in Occam.

The adoption of Occam offers more direct hardware support for special purpose designs as well as common architectures. We concluded this chapter by introducing the MIMD architecture design, and described the Sequent Balance system installed in the Computer Studies Department, Loughborough University in 1986. This system was used to develop and implement the simulation system presented in this thesis.

In Chapter 3, the Instruction Systolic Array (ISA) was introduced as a highly parallel computer architecture that combines the advantages of systolic arrays with the idea of a universal machine, which is capable of solving a large variety of problems.

The analysis of the relationship between the MIMD type mesh-connected parallel computer or Processor Array (PA), and the Instruction Systolic Array (ISA), (1.1.1) shows that programs on a PA can be simulated by equivalent programs on either of the two other models such that the delay is at most proportional to the square root of the number of processors. Asymptotically, the same delay occurs in the simulation of programs on an ISA by equivalent programs on an IBA whereas in the opposite direction we have only constant factor delays. Therefore, with respect to this worst case analysis the ISA is superior to the IBA. Since no instructions have to be broadcast, there is only local information transfer in the ISA. This property is especially advantageous with respect to its realization using VLSI technology. The only main advantage of the IBA over the other two models is its conceptual simplicity: it is much easier to design and understand

programs on an IBA than on an ISA.

Although the PA is the most powerful of the three types of parallel architectures, its main disadvantage is that each of its processors needs its own program store and has to be individually programmable. Because of this increased complexity of the processors the area of a PA is much larger than the area of an IBA or ISA which makes it less suitable for VLSI. The comparison with MIMD- and SIMD-machines shows that the instruction systolic arrays are at least as powerful as array type SIMD-machines. Thus, the large variety of programs on SIMD-machines is easily simulated on ISA. While in many applications the high degree of independence of the individual processors of MIMD-machines is not exploited, the SIMD concept seems to be too restrictive.

In the remainder of this chapter there are many algorithms solved by using the ISA which shows that the ISA is a flexible and powerful parallel architecture well suited for VLSI.

In Chapters 4 and 5, using the flexible architecture (ISA) as a virtual machine programmed in Occam, we developed and implemented a soft-systolic simulation system (SSSS) where the emphasis was on executing programs systolically rather than systolic movement of data. An overall system structure was defined. We demonstrate the feasibility of the system by concentrating on the System and Machine Preparation, Virtual Machine, and Replicating Instruction Systolic Array Language (RISAL), and its RISAL compiler.

More analytically, in Chapter 4, the system and machine preparation used to develop the system was discussed. We reported

that the Balance 8000 Sequent computer system running under the Dynix operating system at Loughborough provides an excellent environment for software development parallel program using support tools for i.e., creation and manipulation and parallel program development. The main features of Loughborough Occam were introduced and used to develop the system.

The virtual machine introduced has three basic sections:

- a) An ISA network of data and control paths.
- b) A set of virtual spoolers for driving the ISA computation and opening up the communication bandwidth of the array.
- c) A collection of processing elements (PE) descriptions for creating specific ISA grids.

The Instruction Systolic Array was introduced as an orthogonal grid of processing elements. Each processing element executes a number of simple operations, and includes memory for intermediate results and registers for communication with other processing cells. Each PE is activated by a combination of an instruction and selector.

We introduced the virtual spoolers which played the role of buffers for the ISA array interface with higher levels of the system, allowing the bandwidth of the input to meet that of the ISA. The grid architecture was a simple specification of network connections between processors, the PE libraries simply containing cell descriptions which responded to ISA instructions with different characteristics.

Using Loughborough Occam, we described the implementation of the virtual machine in detail. We concluded this chapter by describing the processing element (PE) considered in our simulation system which

was a very general element which allows the choice of a wide range of arithmetic and logical operators, and allows the simulation of a wide class of algorithms without the need to develop more special purpose PE's immediately. The structure of the processing element is described in detail, and the implementation process using Loughborough Occam is also presented in detail.

Chapter 5 constitutes a complement to the implementation of the simulation system introduced in Chapter 4. The Replicating Instruction Systolic Array Language (RISAL) is presented here as a suitable medium in which to prepare and debug the ISA control programs, and a method for generating the necessary form of instructions for the ISA. Also, the RISAL compiler was introduced to allow a simple but adequate design environment. RISAL accepts instructions in an assembler-like form, but is fairly permissive about format within the constraints of the syntax. The syntax of RISAL is described in detail. RISAL contains a proportion of semantic rules not indicated in the syntax and allows programs (instruction, selector and data files) to be produced using the same syntax and compiler. The instruction, selector, and data files are described in detail, and can be prefixed with a replicating command which will generate the following instruction by a specified number, and also prefixed with another command to replicate the following lines by a specified number. We used these replicating commands extensively to achieve a reduction of RISAL program coding.

The Pascal language was used to develop and test the RISAL compiler whose task was to read the replicating instruction systolic

array language elements and transform them into a form suitable for the virtual machine to run. The specification and implementation process of the RISAL compiler is described in detail. By then, a number of components are identified and connected serially to form the Soft-Systolic Simulation System (SSSS). This chapter was concluded by testing the simulation system to examine the performance of the solution architecture.

In Chapter 6, the Soft-Systolic Simulation System (SSSS) was used to solve a wide range of algorithms. We have shown the simplicity of this implementation which emphasises the practical significance of the ISA as a flexible array processor architecture. These implementations required a small set of instructions and smaller capacity local memory for the processor, thereby facilitating massive parallelism in a smaller area. While preserving the advantages of the systolic array, namely local communication, regularity and identical simple cell, the ISA concept used in our simulation system overcomes the main disadvantages of the systolic arrays, namely their lack of flexibility. Another important aspect of the ISA is their fault tolerance. If defective processors can be bypassed, a large part of the remaining array may still be used by adjusting the programs to an array of smaller size.

Naturally the power of our simulation system depends on the size of the instruction set. We feel that the restriction to such a simple instruction set is necessary to keep the processors small enough, in order to allow integration of many processors on a single chip.

Although RISAL is very primitive it has been useful in illustrating the ISA's capabilities and has suggested some improvements

to the design of the PE, the interfacing arrangements such as spooling for the virtual grid and a number of additional features to produce a more robust version of RISAL itself.

To allow a wide flexibility in PE development it was observed that reading operation definitions from a file (in alphabetical order) including operation codes allowed new commands to be enlisted easily inside RISAL and permitted the same code for different operations in alternative PE's. We remark here that care must be taken in using duplicate codes but no real problems were encountered.

For RISAL, two main constructs suggest themselves as follows:

- i) Replicated line section (REPS): For example, data n,O3,00;
data n,O3,00; REPS(count)[null ,O,O;data n,O3,00];null ,O,O;
which would repeat the section of the line in brackets count times.
The main difficulty in implementing this statement is keeping track of REP nesting and checking that the correct number of instructions are generated.
- ii) Replicated line shift (REPLS): of the form,
REPLS(count,shift)[line]:
Here a specified line is replicated count times and on each replication is shifted right or left. 'Shift' places according to the sign of the shift. Instructions falling off the end of a line must be neglected and spare places filled with a default operation like null.

Many variations to these basic constructions such as cyclic line shifting, shifting of line sections, and conditional line shifting are also apparent - but amount only to improving the readability of the ISA program.

In the ISA design which is introduced in this thesis, the ISA instruction is represented as an 8 digit integer with each field being 2-digits wide to allow for the possible implementation of 100 instructions and an internal memory address space of 100 instructions. The port specifications also allows 100 combinations of input/output but only the first 16 have been used.

To improve the communication between the registers it is necessary to utilise the extra slots to allow for multiple communication registers in each cell. These operations can be implemented more effectively by using bit logic and slices, but Loughborough OCCAM is restricted in this respect. Furthermore, a 2-digit field also allows a wide range of library PE's to be developed.

The data file introduced here is more complex than the instruct and selector file, as it requires the specification of input for the four possible boundaries of the ISA grid. The current implementation does not expose all the inherent parallelism in collecting the boundary data, as we can define four files one for each boundary, and then use the buffers in parallel. However, there is a considerable overhead in checking that sufficient boundary data is available. This requires the specification of four separate files. In our implementation, one file is defined and the boundary input and output is sequentially buffered. This makes the checking and the setting up of the data input sequence easier and more related to the algorithms being simulated. For large grids however this method will become impractical and adding a pre-processor to the ISA to separate out the data into temporary files seems the best alternative.

In general, the reading of input and writing of output data is

performed in the ISA in parallel with the ISA execution. Clearly this is the place where any bottlenecks are likely to occur especially for large n (large grid size). It is clear that the matrix size and structure of the ISA is useful mainly when the matrix is small, dense or banded. However, the ISA operation will become slow for the case of large sparse matrices, hence the present system design with a bounded number of processors can simulate smaller networks without difficulty.

There are a number of issues concerning the problem of transforming programs that were originally designed for different models of parallel computers into ISA programs. Among these different models are the MIMD and SIMD type mesh-connected processor arrays and some variants of the ISA. The main result is that an arbitrary program that runs on an $(n \times n)$ mesh-connected parallel computer in k steps can be transformed into an ISA program having $O(nk)$ steps. In many cases, however, especially when the original machine is of SIMD-type, this transformation introduces only a delay of $O(1)$. Often it is possible, e.g. in the case of two-dimensional systolic algorithms, to re-design an existing algorithm. Sometimes it will be necessary to develop a new algorithm in order to meet the requirements of the ISA.

Naturally, the power of the ISA depends on the size and complexity of the instruction set. This leads to the use of complex processors which will inevitably slow down the whole system. Besides, the integration of many different types of processors is not easy, especially if other types of problems are to be considered i.e. graph theoretic problems. Thus, our Instruction Systolic Array having a different and more powerful set of instructions would need to be much more complex again.

Finally, the ISA design can be considered analogous to the choreography of the ballet which consist in fitting a story to both music and scenario. In the ISA, the problem corresponds to the story, the algorithm to the ballet, the timing steps to the dance movements and the VLSI layout to the scenario. Thus, the instruction stream from the top of the ISA and the selector stream from the left of the ISA are analogous and should be planned in advance so that each instruction when executed meets a required selector bit. In view of these considerations, the ISA seems to be a candidate for the realization of a highly parallel VLSI computer design.

REFERENCES

ALFRED, V. AHO, JEFFREY, D. ULLMAN [1977]: *"Principles of Compiler Design"*, Addison-Wesley Series in Computer Science and Information Processing, Bell Laboratories, Murray Hill, New Jersey, 1977.

ARVIND, and GOSTELOW, K.P. [1982]: *"The U-Interpreter"*, IEEE Computers, Feb. 1982, pp.42-49.

ARVIND, and THOMAS, R.E. [1980]: *"I: Structures: An Efficient Data Type for Functional Languages"*, MIT/LCS/TMN-178, Sept. 1980.

ATALLAH, M.J., KOSARAJU, S.R. [1984]: *"Graph Problems on a Mesh-Connected Processor Array"*, JACM, Vol. 31, No.3, Jul. 1984.

BAASE, S. [1978]: *"Computer Algorithms: Introduction to Design and Analysis"*, Addison-Wesley Series in Computer Science, 1978.

BARNES, G.H. ET AL [1968]: *"The Illiac IV Computer"*, IEEE Trans. Comput., Vol. C-17, No.8, Aug. 1968, pp.746-757.

BASKET, F. and SMITH, A.J. [1976]: *"Interference in Multiprocessor Computer Systems with Interleaved Memory"*, Communications of the ACM, Vol. 19, No.6, June, 1976.

BEAR, J.L., [1976]: *"Multiprocessing Systems"*, IEEE Trans.Comput., Vol. C-25, No.12, Dec. 1976, pp.1271-1277.

BRENT, KUNG, H.T. and LUK [1983]: *"Some Linear Time Algorithms for Systolic Arrays"*, CMU-ROL-83 and Invited Paper 9th World Computer Congress, Paris, 1983.

- BURDEN, R.L., FAIRES, J.D., REYNOLDS, A.C. [1981]: *"Numerical Analysis"*, Prindle, Weber and Schmidt Incorporated, 20 Newbury Street, Boston, Massachusetts 02116, 1981.
- CHANG, D.Y. ET AL [1977]: *"On the Effective Bandwidth of Parallel Memories"*, Vol.C-26, No.5, May 1977, pp.480-490.
- DE BOOR, C. [1972]: *"Elementary Numerical Analysis: An Algorithmic Approach"*, McGraw-Hill Kogakusha Ltd., Second Edition, 1972.
- DENNIS, J.B. [1980]: *"Data Flow Supercomputer"*, IEEE Computer, Nov. 1980, pp.48-56.
- DENNIS, J.B. [1974]: *"First Version of a Data Flow Procedure Language"*, Computer Science, Vol.19, Springer-Verlag, 1974, pp.362-376.
- DEW, P.M. [1984]: *"VLSI Architectures for Problems in Numerical Computation"*, Workshop on Progress in the Use of Vector and Array Processors, eds. Paddon D.J. and Pryce, J.D., pp.1-24.
- DEW, P.M., MANNING, L.J., MCEVOY, K. [1986]: *"A Tutorial on Systolic Array Architectures for High Performance Processors"*, 2nd Int. Electronic Image Week, Nice, 1986, and Report No.205, Leeds University.
- ENSLOW, P.H. [1977]: *"Multiprocessor Organisation - A Survey"*, Comp. Surveys, Vol.9, No.1, March 1977, pp.103-129.
- FISHER [1984]: *"Implementation Issues for Algorithmic VLSI Processor Arrays"*, Ph.D. Thesis, 1984, CMU, Pittsburgh.

- FLYNN, M.J. [1972]: *"Some Computer Organizations and Their Effectiveness"*, Trans. Comput. C-21, 1972.
- FLYNN, M.J. [1966]: *"Very High-Speed Computing Systems"*, Proc. of the IEE, Vol.54, No.12, Dec. 1966, pp.1901-1909.
- FOSTER, M.J. and KUNG, H.T. [1980]: *"The Design of Special-Purpose VLSI Chips"*, Computer, Vol.13, No. 1, Jan. 1980, pp.26-40.
- GANDIO, J.L. and ERCEGOVAC, M.D. [1982]: *"A Scheme for Handling Arrays in Data Flow Systems"*, Proc. 3rd Int.Conf. Distributed Computing Systems, 1983, pp.235-242.
- GEHRIG, E. ET AL [1982]: *"The CM* Testbed"*, IEEE Computer. Oct.1982, pp.40-53.
- GREGORY, R.T., KRISHNAMURTHY, E.V. [1984]: *"Methods and Applications of Error-Free Computation"*, Springer Verlag, New York, 1984.
- GUIBAS, L.J., KUNG, H.T., THOMPSON, C.D. [1979]: *"Direct VLSI Implementation of Combinatorial Algorithms"*, Proc. Caltech Conf. on VLSI, Californian Institute of Technology, Pasadena, 1979, pp.509-525.
- GURD, J.R., KIRKHAM, C.C. and WATSON, I. [1985]: *"The Manchester Prototype Data-Flow Computer"*, Comm. ACM, No. 1, Jan. 1985, pp.34-52.
- HAMBRUSCH, S.E. [1983]: *"VLSI Algorithms for Connected Component Problem"*, SIAM J. Computing, Vol.12, Nr.2, 1983.
- HANDLER, W. [1982]: *"Innovation Computer Architectures - How to Increase Parallelism But Not Complexity"*, in Parallel Processing Systems, Evans, D.J. (Ed.), Cambridge University Press, 1982, GB, pp.1-41.

- HAYES, J.P. [1978]: *"Computer Architecture and Organisation"*, McGraw-Hill, Kogakusha Ltd., Japan, 1978.
- HIGBIE, L.C. [1972]: *"The Omen Computers: Associative Array Processors"*, IEEE Comp.Conf., 1972, Digest, pp.287-290.
- HOARE, C.A. [1978]: *"Communicating Sequential Processors"*, Communications of the ACM, Vol.21, No.8, Aug. 1978, pp.666-677.
- HOBBS, L.C. and THESIS, D.J. [1970]: *"Survey of Parallel Processor Approaches and Techniques"*, in *Parallel Systems: Technology and Applications*, Hobbs et al (Eds.), Spartan Books, New York, 1970, pp.3-20.
- HOCKNEY, R.W. and JESSHOPE, C.R. [1981]: *"Parallel Computers - Architecture: Programming and Algorithms"*, Adam Hilger Ltd., Bristol, England 1981.
- KNUDE, M., LANG, H.W., SCHIMMLER, M., SCHMECK, H., SCHRODER, H. [1985]: *"The Instruction Systolic Array and Its Relation to Other Models of Parallel Computer"*, Proc.Int.Conf. Parallel Computing 85, North Holland 1985.
- KNUTH, D.E. [1973]: *"The Art of Computer Programming"*, Vol. 3: Sorting and Searching, Addison Wesley, 1973.
- KUNG, S.Y. [1985]: *"VLSI Array Processors"*, IEE ASSP Magazine, July 1985, pp.5-22.
- KUNG, S.Y., LO, S.C., LEWIS, P.S. [1987]: *"Optimal Systolic Design for the Transitive Closer and the Shortest Path Problems"*,

IEEE Trans. Comput. Vol. C-36, No.5, 1987, pp.603-614.

KUNG, S.Y. [1987]: *"VLSI Array Processors"*, in *Systolic Arrays*,
Will Moore et al (eds.), Adam Hilger, Bristol and Boston, 1987.

KUNG, H.T. [1984]: *"Systolic Algorithms for the CMM Warp Processor"*,
CMUC-CSA-84-158 (7th Int.Conf.).

KUNG, H.T. [1979]: *"Let's Design Algorithms for VLSI System"*, Proc.
Conf. Very Large Scale Integration: Architecture, Design,
Fabrication, California Institute of Technology, Jan. 1979,
pp.65-90.

KUNG, H.T. and LEISERSON, C.E. [1978]: *"Systolic Arrays (for VLSI)"*,
in Proc. Sparse Matrix Symp. (SIAM), 1978, pp.256-282.

LANG, H.W. [1985]: *"The Instruction Systolic Array, a Parallel
Architecture for VLSI"*, Integration, the VLSI Journal 4, 1986,
pp.65-74.

LANG, H.W. [1987]: *"Transitive Closure on the Instruction Systolic
Array"*, Technical Rep. Institut for Informatic and Praktische
Mathematik, Keil Univ., F.R. Germany, 1987.

LANG, H.W. [1987]: *"ISA and SISA: Two Variants of a General Purpose
Systolic Array Architecture"*, Proc. Second Int.Conf. on
Supercomputing, Vol.1, 1987, pp.460-465.

LANG, H.W., SCHIMMLER, M., SCHMECK, H., SCHRODER, H. [1983]: *"A Fast
Sorting Algorithm for VLSI"*, LNCS 154, Springer-Verlag, 1983.

- LAWRIE, D.H., LAYMAN, T., BEAR, D., RANDAL, J.M., [1975]: *"GLIPNIR - A Programming Language for ILLIAC IV"*, Comm.ACM, Vol.18, March 1975, pp.157-164.
- LILLEVIK, S.L., EASTERDAY, J.L. [1984]: *"Throughput of Multiprocessor with Replicated Shared Memories"*, National Computer Conference, 1984, pp.51-56.
- McKEOWN, G.P. [1986]: *"Iterated Interpolation using Systolic Arrays"*, ACM Trans.Math. Software, Vol.12, 1986, pp.162-170.
- MEAD, C.A. [1981]: *"VLSI and Technological Innovations"*, in VLSI 81, Proceedings of the 1st Int. Conf. on Very Large Scale Integration, Univ. of Edinburgh, Aug. 1981, J.P. Gray (ed.), Academic Press, London, U.K., pp.3-11.
- MEAD, C.A. and CONWAY, L.A. [1980]: *"Introduction to VLSI System"*, Addison-Wesley, Reading, Mass. 1980.
- MEGSON, G.M. [1984]: *"Implementing Systolic Algorithms in OCCAM"*, Technical Report, Leeds University, 1984.
- MEGSON, G.M. [1987]: *"Novel Algorithms for the Soft-Systolic Paradigm"*, Ph.D. Thesis, OL4518102, L.U.T., 1987.
- MOLDOVAN, D.I. [1983]: *"On the Design of Algorithms for VLSI Systolic Arrays"*, Proceedings of the IEEE, Vol. 71, No.1, Jan. 1983, pp.113-120.
- MONGENET, C., PERRIN, G.R. [1987]: *"Synthesis of Systolic Arrays for Inductive Problems"*, Lecture Notes in Computer Science 258, Springer-Verlag, 1987, pp.260-277.

- MURTHA, J. and BEADLES, R. [1964]: *"Survey of the Highly Parallel Information Processing Systems"*, Prepared by the Westinghouse Electric Corp., Aerospace Division, ONR, Rept. No. 4755, Nov.1964.
- MUSLIH, O.K., EVANS, D.J. [1987]: *"Simulation of Soft-Systolic Programs"*, Int.Rept. No. 408, Dept. of Computer Studies, L.U.T. 1987.
- NASSIMI, D., SAHNI, S. [1979]: *"Bitonic Sort on a Mesh-Connected Parallel Computer"*, IEEE Trans. Computers, Vol. C-28, 1979.
- PATEL, J.H. [1981]: *"Performance of Processor-Memory Interconnections, for Multiprocessors"*, IEEE Trans.Comp., Vol. C-31, No.10, Oct. 1981, pp.771-780.
- RAMAMOORTHY, C.V. and LI, H.F. [1977]: *"Pipeline Architecture"*, Computer Survey Vol.9, No.1, March 1977, pp.61-102.
- ROBERT, Y., TRYSTRAM, D. [1986]: *"Systolic Solution of the Algebraic Path Problem"*, in W. Moore, A. McCabe, R. Urquhart (eds.): *Systolic Arrays*, Adam Hilger, Bristol, 1986, pp.171-180.
- RODRIGUE, G.H., [1982]: *"Parallel Computations"*, Vol. 1, Academic Press, New York, 1982.
- ROTE, G. [1985]: *"A Systolic Array Algorithm for Algebraic Path Problem (Shortest Path; Matrix Inversion)"*, Computing 34, 1985, pp.191-219.
- SCHIMMLER, M. [1987]: *"Fast Sorting on the Instruction Systolic Array"*, Inst. Fur Informatik U.P.M., Universitat Kiel, West Germany,

Tech.Rept. No.8705, 1987.

SCHIMMLER, M., SCHRODER, H. [1987]: *"Finding all Cut-Points on the Instruction Systolic Array"*, Computer Science Laboratory, Australian National University, Canberra, ACT, 2601, Australia, 1987.

SCHMECK, H. [1986]: *"A Comparison-Based Instruction Systolic Array"*, in *Parallel Algorithms and Architectures*, M. Cosnard et al., Editors, Elsevier Science Publishers B.V., North-Holland, 1986, pp.281-292.

SCHRODER, H. [1988]: *"Top-Down Designs of Instruction Systolic Arrays for Polynomial Interpolation and Evaluation"*, Computer Science Laboratory, Australian National University, Canberra, ACT 2601, Australia, Feb. 1988.

SCHRODER, H., KRISHNAMURTHY, E.V. [1988]: *"Generalized Matrix Inversion using Instruction Systolic Arrays"*, Computer Science Laboratory, Australian National University, Canberra, ACT 2601, Australia, March, 1988.

SEIGEL, H.J. [1979]: *"Interconnection Networks for SIMD Machines"*, IEEE Computer, June 1979, pp.57-65.

SHORE, J.E. [1973]: *"Second Thoughts on Parallel Processing"*, Comput. Elect.Eng., pp.95-109, 1973.

SLOTNICK, D.L., BORCH, W.C., McREYNOLDS, R.C. [1962]: *"The SOLOMON Computer"*, 1962 Fall Joint Computer Conf., American AFIPS Proc.

Vol.22, Washington, Spartan, D.C. 1962, pp.97-107.

SNYDER, L. [1982]: *"Introduction to the Configurable Highly Parallel Computer"*, IEEE, Comput. 1982 (15), pp.47-56.

STONE, H.S. [1971]: *"Parallel Processing with the Perfect Shuffle"*, IEEE Trans.Comput. Vol. C-20, 1971.

STONE, H.S. [1980]: *"Parallel Computers"*, in Introduction to Computer Architectures, Stone, H.S. (ed.), SRA, Chicago, USA, 1975, pp.318-374.

SWAN, R.J., FULLER, S.H., SIEWIOREK, D.P. [1977]: *"CM - A Modular Microprocessor"*, National Computer Conference, 1977, pp.637-646.

THOMPSON, C.D., KUNG, H.T. [1977]: *"Sorting on a Mesh-Connected Parallel Computer"*, CACM, Vol.20, 1977.

ULLMAN, J.D. [1984]: *"Computational Aspects of VLSI"*, Computer Science Press, 1984.

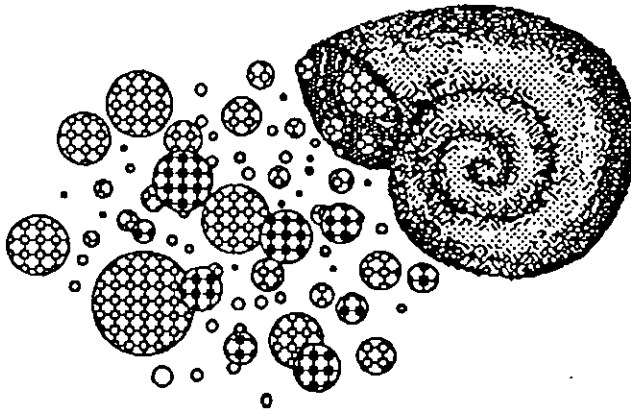
UNGER, S.H. [1958]: *"A Computer Oriented Toward Spatial Problems"*, Proc. IRE, Oct. 1958, pp.17-44.

WARSHALL, S. [1962]: *"A Theorem on Boolean Matrices"*, J.Assoc.Comput. Mach. 9, 1962, pp.11-12.

WULF, W., BELL, C.G. [1972]: *"Cmmp a Multi-Mini-Processor"*, AFIPS Fall Joint Computer Conf. American AFIPS Proc. Vol. 22, Washington, D.C., Spartan, 1962, pp.97-107, 1972, FJOC, pp.765-777.

APPENDIX I

Loughborough OccamTM Compiler Version 5.0 Documentation



Help for running the occam compiler

A source 'occam' file (OCCAM and INMOS are trademarks of the INMOS group of companies) must be of the form '*.occ', to compile it to form an 'a.out' command file use the default options. For example to compile 'my_first.occ' :-

```
occam my_first.occ
```

An executable object 'a.out' is produced. As a shortcut you can omit the '.occ' affix and just say 'occam my_first', the compiler will add on the affix for you. If a program is split into several files these can be separately compiled and linked together using the occam compiler and built in linker. Each previously compiled occam program is specified in the command line in the form '*.o' e.g. :-

```
occam main.occ numericlib.o screenlib.o
```

This will compile the source of 'main' and link it in with the pre compiled library occam files 'numericlib.occ' 'screenlib.occ'. The -l option is used to generate new versions of library file objects. Various switch options are provided, mainly for compiler debugging. Flags can either be put separately ('-g -l') or together and in any order ('-lg', '-gl'). The following switches may be useful :-

-g :

```
occam -g fast.occ
```

Compile the occam program as before but run the resulting program immediately (a compile,load and go option). If flag options are specified that apply to the run of the program these will be passed on as in 'occam -gqc fast'.

-l :

```
occam -l new_lib
```

Compile the program and produce object but do not link the object files together to produce an object program. This option is used for bulding up libraries of routines or to cut down the compilation time for compiling one long program.

-o :

```
occam keep_it -o saverun
```

Compile the program as normal but place the object program in the file 'saverun' rather than the default 'a.out'. Useful for saving several occam object files at the same time.

-x :

```
occam -x old_fashioned.occ
```

Compile according to the strict Inmos occam specification, LUT extensions (see file 'occamversion') currently include :-

Multiple source file cross linking.

Dynamic features.

Variable PAR replicator counts.

Floating point arithmetic.

-c :

```
a.out -c
```

Run the object program with cursor addressable facilities enabled, the standard library procedures 'goto.x.y' and 'clear.screen' require these facilities.

-G :

```
occam -G error_prone
```

Compiles the file as normal but generates a symbol file as well (in this case it would be 'error_prone.sym'), this is used by the run-time system to inspect the values of variables.

-q :

```
a.out -q
```

Run the object program without producing any characters to the screen other than those output by the program (unless CTRL c used). This enables occam programs to dump output that can be processed by other occam programs.

-F and -M :

```
occam -F num.occ
```

'-F' Includes the floating point library routines to provide a simple real number arithmetic capability. '-M' includes both the floating point and mathematical library routines to provide mathematical library routines.

-I :

This provides the features of the Inmos proto-occam definition (see 'occam_version') such as STOP and TIME, it should be used where possible as it is closer to the occam-2 definition.

Full list of compiler option flags

The full (often cryptic) range of switch options are as follows. Several switch flags can be given, in any order and either separately or together. The mnemonic character giving the switch is highlighted by a capital letter. They are divided into sections - user defined flags, and system defined options, which are selected by prefixing with '%'.

User Flags

- % The next flag(s) are system flags - switch flag mode.
- c Run the program with Cursor addressable options enabled. The library routines 'clear.screen' and 'goto.x.y' need this flag set. If used for the compiler must also give the -g option.
- e Produce object/run object for Execution tracing. The resulting object file is then run with the '-e' option. This utility is described in 'tracerinfo'.
- f Force full occam semantic check on use of variables.
A variable (not vectors though) can not be set within a PAR construct if the declaration is outside the PAR. This applies equally to procedure calls that change global variables.
- g Run the resulting object file if compilation succeeded.
The program Goes immediately it is ready to.
- h Print out this 'Help' information.
- i Force an Interrupt immediately before start of execution - immediately displays the debug help menu. This enables break and trace points to be setup prior to anything being executed.
- l Compile but do not link the occam source. Needed when using multiple occam source Library files.
- m Check that every channel Match properly on execution, channels can have only one input and one output process during execution.
- o Produce an Object program with name given by the non-switch argument following this switch. Enables you to choose an object file name other than 'a.out'.
- q Run the program without outputting some non occam program produced messages - e.g. 'OCCAM Start Run'. Must give -g option as well 'q' stands for Quiet.

Useful when producing output to be piped or processed by other programs.

- w Suppress the Warning messages from the compiler - when you have seen these warnings once you may find it less irritating to suppress them on subsequent compilations - does not affect error reporting or any other compiler action.
- x Do not permit any local LUT eXtensions in the source text. See 'occinfo' for information about these - for example recursion and EXTERNAL procedure definitions. Useful if moving an occam program for use on another occam compiler system.
- F Include the standard Floating point library routines. Provides routines to read or write floating point routines to channels.
- G Produce a symbol table file (with affix '.sym') for use with the 'm' option in the dynamic debugger for symbol value examination.
- I Permit the use of INMOS proto-occam version 2. These changes include the use of 'TIME' instead of 'NOW', the 'STOP' primitive and the use of 'Stopping IF' - an alternative without any TRUE conditions will STOP.
- L Use Long winded load, all the 'C' libraries are added at the last moment rather than using the pre-linked object, this may be useful if a user occam/C library calls a 'C' routine that is not used in the occam run time system. See 'libraryhelp' for more info.
- M Include the Mathematical library and floating point routines.
- O Produce optimized object. May improve run time by 20%.
- R Use Randomized scheduling when running the program - the same scheduler choices will not be made on separate executions. This gives non-deterministic execution and will be slightly slower but may be useful occasionally.
- S Do not include the Standard I/O routines with the object. This library is included by default, there is no reason not to want to include it unless you want to devise a totally new one.
- T The next argument is a Timing definition file built by the 'timebuild' utility to be used in conjunction with the '-e' option, supplying '-T' automatically selects '-e'. If this option is not selected the execution timings are taken from the source library file 'times'. Look at the 'timerinfo' help file for more details.
- V The compiler will normally desist reporting errors and warnings after the first fifty or

anything - like '-n' in the UNIX 'make' command. Useful when options start getting complicated. A No operation facility.

- Q Undocumented feature under test.
- S Do not apply some Simplifying transformations on the program. These currently remove constructs with no processes in them and redundant SEQ and PAR headers. These save a small amount of space and time at run and compile time and there is little point in turning off this option.
- X Print out the procedures that have been defined in the link files but has not been referenced - detects eXtra procedures defined across files but not used.
- Y Produce the linker assembler output in a permanent file rather than in a temporary file on '/tmp'. Enables the output from the linker to be debugged.
- Z Get the linker to print out all the definitions it is told about.

Description of the library routines

Standard Library

Provide commonly used routines to read and write to the keyboard and screen channels. The routines are written in 'C' and occam and use standard C or 'curses' I/O routines. There are also general routines for use to pause or abort a program as well as to use the 'C' random number routines. They are available by default to all programs unless the -S compiler flag is used to override their inclusion.

EXTERNAL PROC str.to.screen (VALUE s []) :

Output the string s (a byte array with byte 0 as the length). The whole string is guaranteed to be printed in one sequence, two concurrent calls to str.to.screen will not interleave. Equivalent to the program fragment :-

```
PROC str.to.screen (VALUE s []) =
  SEQ n = [1 for s [BYTE 0]]
    screen ! s [BYTE n] :
```

EXTERNAL PROC num.to.screen (VALUE n) :

Output a number to the screen. The number can be signed, and uses the minimum number of characters (no leading spaces). Equivalent to the 'C' language 'printf ("%d",n);' statement.

EXTERNAL PROC str.to.chan (CHAN c,VALUE s []) :

Output the string s to a channel 'c'. The call 'str.to.chan (screen,"fred")' is identical to 'str.to.screen (fred)'. Useful for string output to files.

EXTERNAL PROC num.to.chan (CHAN c,VALUE n) :

Output ascii string for the number 'n' to channel 'c'. Like 'str.to.chan' but for numbers not channels.

EXTERNAL PROC num.to.screen.f (VALUE n,d) :

Output a number to the screen in a field of width 'd'. If the number is too big for the field the number is written out in full regardless, the routine call num.to.screen.f (n,1) is equivalent to num.to.screen (n). The routine uses the 'C' language printf format %nd where n is the field width.

EXTERNAL PROC goto.x.y (VALUE x,y) :

Use the 'curses' package to implement a cursor 'goto' facility. No error checking is made that the move is within the screen area. The x-axis is across the screen and y-axis down, co-ordinate (0,0) is in the top left hand corner of the screen. The first line is used by the run time system to print messages.

EXTERNAL PROC clear.screen :

Use curses to clear the screen,if cursor addressable option not used this will still try to clear the screen using the curses "CL" termcap defined string.

EXTERNAL PROC num.from.keyboard (VAR n) :

Read a number from the keyboard and assign to variable 'n'. The routine is not very sophisticated. It will read negative numbers (start '-') and ignore any leading 'space' characters. The number must be followed by a non-digit, this character is read by the routine and not available on a subsequent 'Keyboard ? ch' process. There is no check that the number is too big for the number range. It will expect at least one digit otherwise it will give an error message.

EXTERNAL PROC num.from.chan (CHAN c,VAR n) :

Read a number from a channel 'c'. If 'c' is the keyboard this is equivalent to calling 'num.from.keyboard'.

EXTERNAL PROC abort.program :

Force the program to abort execution. An explanatory message is printed so that the cause will be known.

EXTERNAL PROC force.break :

Perform the same action as if 'CTRL-C' was pressed at the terminal. The user interface routines can then be run under the menu selection facility provided.

EXTERNAL PROC random (VALUE d,VAR n) :

Return a pseudo random number in the range 0 to d-1 by using the 'C' 'random ()' function in the variable n. The VALUE of d must not be zero. The sequence of random numbers will be modified if the '-R' run option is used.

EXTERNAL PROC init.random (VALUE n) :

Initialise the seed for the random number generator for subsequent calls to the procedure 'random'. Uses the 'C' language routine 'srandom ()'.

EXTERNAL PROC trace.value (VALUE n) :

Print out the integer value of 'n' on the screen with the prefix string 'Trace value : ' - this makes debugging a little easier.

EXTERNAL PROC open.file (VALUE path.name [],access [],CHAN io.chan) :

Connect the channel 'io.chan' to a UNIX file. The procedure must be provided with the pathname of the file as a string, and the access mode ("r" read access,"w" write access,"a" append access). Subsequent input or output on 'io.chan' will fetch/put a single character from/to the file. Attempts to input past the end of file will receive the value -1.

EXTERNAL PROC close.file (CHAN io.chan) :

Cease connection of the channel with its currently open file.

EXTERNAL PROC open.pipe (VALUE command.name [],access [],CHAN io.chan) :

Connect the channel 'io.chan' to a UNIX pipe running command 'command.name'. The procedure must be provided with the UNIX command name and 'r' to read from it, or 'w' to write to it). Subsequent input or output on 'io.chan' will fetch/put a single character from/to the file. Attempts to input past the end of file will receive the value -1.

EXTERNAL PROC close.pipe (CHAN io.chan) :

Cease connection of the channel with its currently active command.

EXTERNAL PROC system.call (VALUE command [],VAR code) :

Execute the UNIX command contained in the string 'command' and return the value in 'code' TRUE if the command succeeded without error and FALSE otherwise.

EXTERNAL PROC set.timers (VALUE init.value) :

Set up the interval timers ITIMER_REAL,ITIMER_VIRTUAL to the given start value. These are used for timing sections of code on the VAX. Uses 'setitimer' call. Note that

using 'WAIT' primitive will reset the timer so it can only be used for simple sections of code. It should also be noted that it times the whole program and not a single occam process.

EXTERNAL PROC get.real.timer (VAR secs,micro.secs) :

Get the current elapsed timer values in seconds and microseconds. Timers count downwards and are not especially accurate. Uses 'getitimer' call.

EXTERNAL PROC get.cpu.timer (VAR secs,micro.secs) :

Get the current executed CPU timer values in seconds and microseconds. Timers count downwards and are not especially accurate.

Floating Point Library

Routines to perform floating point input/output. They are available by giving the compiler flag '-F' when linking an occam program. Floating point value can be assigned and transmitted via channels just like normal integer values, see the file 'occamversion' for details as to the language extensions introduced to support them.

Input/Output Routines

EXTERNAL PROC fp.num.to.screen (VALUE FLOAT f) :

Print out the floating point number in 'C' language float format "%6.6f". If the number is too small or too big the standard 'C' action will be taken.

EXTERNAL PROC fp.num.to.screen.f (VALUE FLOAT f,VALUE w,d) :

Print out the floating point number in 'C' real format "%w.df". If the number is too small or too big problems will arise.

EXTERNAL PROC fp.num.to.screen.g (VALUE FLOAT f) :

Print out the floating point number in 'C' real format "%g". This will use the most appropriate format - exponent form if necessary.

EXTERNAL PROC fp.num.to.chan (CHAN c,VALUE FLOAT f) :

Write a number to a channel. If channel is 'screen' this is equivalent to 'fp.num.to.screen'. Useful for writing data to files.

EXTERNAL PROC fp.num.from.keyboard (VAR FLOAT f) :

Read in a floating point number. The number is expected to begin with a digit or '.' (indicating 0.), leading spaces are ignored. The number ends on a non-digit and this character will not be available to subsequent reads from the keyboard channel. The following are valid input numbers followed by the interpreted value for the input.

45.35 (45.35) 0.0004 (0.0004) .0 (0.0) 1. (1.0) 124 (124.0)

EXTERNAL PROC fp.num.from.chan (CHAN c, VAR FLOAT f) :

Read a floating point number from a channel 'c'. If channel is keyboard this is equivalent to 'fp.num.from.keyboard'.

Mathematical Routine Library

Mathematical routines from the UNIX '-lm' library. These are included by specifying the '-M' flag. They are all in single precision even though double precision 'C' routines are called.

EXTERNAL PROC fp.sine (VALUE FLOAT a, VAR FLOAT res) :

Return the sine of 'a' in 'res'. Angles are in radians.

EXTERNAL PROC fp.cosine (VALUE FLOAT a, VAR FLOAT res) :

Return the cosine of 'a' in 'res'. Angles are in radians.

EXTERNAL PROC fp.arc.sine (VALUE FLOAT a, VAR FLOAT res) :

Return the arc sine of 'a' in 'res'. Angles are in radians.

EXTERNAL PROC fp.arc.cosine (VALUE FLOAT a, VAR FLOAT res) :

Return the arc cosine of 'a' in 'res'. Angles are in radians.

EXTERNAL PROC fp.arc.tan (VALUE FLOAT a, VAR FLOAT res) :

Return the arc tangent of 'a' in 'res'. Angles are in radians.

EXTERNAL PROC fp.exp (VALUE FLOAT a, VAR FLOAT res) :

Return e to the power 'a' in 'res'.

EXTERNAL PROC fp.log (VALUE FLOAT a, VAR FLOAT res) :

Natural logarithm of 'a' in 'res'.

EXTERNAL PROC fp.sqrt (VALUE FLOAT a, VAR FLOAT res) :

Square root of 'a' in 'res'. Returns an occam error if 'a' is negative.

The run time system

As you might hope when an occam program is executed it will follow the program execution until one of three things happen.

- 1] The program terminates
- 2] CTRL-C is pressed on the keyboard
- 3] An error is detected.

In the case of (2) and (3) a debug option will be displayed, this allows you to abort the program, ignore the interrupt (continue), and to restart the program again. Other options control the '-e' trace output, provide a 'system' debug option (which is only really useful to someone who knows their way around the compiler), an option to specify which source file you want to debug and the 'screen animated debug'. This latter option should be of most use and is described in detail in the next section.

Errors come in two types 'Fatal Errors' and just 'Errors', it is not possible (or wise) to continue execution after the former, but the latter may be ignored if the symptom is expected.

The run time display debugger

This utility that runs under the run time system enables users to look at the status of the processes during execution of a program.

The utility requires the use of a cursor addressable terminal. The system provides selective display of the source file(s) that were compiled to form the program together with a column showing the currently existing processes on those particular lines of the source file.

When initially entered by pressing 'CTRL-C' the program execution will be halted, the execution can be restarted in 'stepped mode' so that the display will be updated every occam scheduler action.

Breakpoints and trace points can be added at selected line numbers. Break points cause the debug display to be automatically entered when any of the process executes any of the source lines on which a break point is set. Trace points cause temporary entry into the debug display before resuming normal execution after five seconds pause.

If a file has been compiled with the '-G' flag then the value of occam variables and the status of channels can be printed. Because an occam program can have several processes running with different values to the same identifiers (e.g. within `PAR n = [0 FOR 7]`, 'n' has a different value for each separate process) a single process must be selected as before this

facility can be used. When selected a second window within the debug display is opened and the values printed by the program are placed within it.

Straightforward use of the debug display will normally entail running a program and pressing CTRL-C when a dubious section of code is about to be executed and entering the debug display ('z' command). Thereafter the commands 'p' to find the next process, 'f' and 'b' might be used to see whereabouts the process is executing. The program can then be single stepped through using the 'r' command to start execution and 's' command to stop execution. Eventually exit of the debug display can be made with the 'x' command.

There are two special markers that are used, '>' on a line indicates the currently selected line and '-' the currently selected process.

The commands where practical have been made similar to those in UNIX 'vi'. (UNIX is a trademark of A.T. & T.).

Available commands

Moving about within the file

^D- Move forward half a page of source text.

^F- Move forward a page of source text.

^U- Move backward half a page of source text.

^B- Move backward a page of source text.

:<number> - Move to given line <number> in file.

k - (or ^K) Move down one line.

j - (or ^J) Move up one line.

/<string> - Find given <string> in file from current position.

n - Find next string occurrence for match string selected by '/' command.

p - Find the next process in the file.

Trace/Breakpoints

b - Add breakpoint at currently selected line.

t - Add tracepoint at currently selected line.

d - Delete the trace/break point at the selected line.

c - Delete all the points in the current file.

C - Delete all the points in all the files.

P - Print process status of the currently selected process .

D - Deselect the current debug occam process.

S - Select the current debug occam process.

N - Select next process on the same line, if there are several processes that are shown as executing on the same line then 'S' will make an arbitrary choice, 'N' can be used to override this and step through the processes until the one that is desired is selected.

Symbol inspection

m - Select a symbol to display, if no symbols have been selected before then the symbol window is opened and the value of the variable or the status of a channel.

M - Repeat the previous 'm' command. To find the value of the same variable name again.

Execution control

a - Abort the run.

r - Run debug display if a debug process is selected the debug display will be re-entered every time that process is run, otherwise the debug display will be run each time any process is run.

> - Execute in single step mode. Only a single step is executed.

s - Stop the debug display from running temporarily after a 'r' or 'x' command.

u - Change display step interval (initial step interval is 1), this permits the location of processes to be seen after 'n' steps rather than after each and every time it is executed. Not particularly useful.

x - Exit display debugger, program will proceed normally until a trace/break point is found or '^C' is pressed.

X - Exit to main '^C' menu so that program restart, abort, file selection or system debug can be done. Used when you wish to debug a different file or to set things going again after setting up breakpoints.

Miscellaneous

? - Print out this help information.

^L- (or ^R) Redraw the current displayed information.

i - Buffer keyboard channel input text for the program.

O - Print overall data about the processes currently executing - how many are in each process status, stack use and clock time.

V - Display the occam program's current screen output temporarily .

v - Invoke the 'view' command on the occam source file (this is just like 'vi' but with read only access to the file - This can be used to provide more powerful string search facilities when debugging.

Display key

The column between the line number and the text is used to display the number and status of processes executing on that line. Because of the compilation these may be out by a line or two in some circumstances. Most sequential code will be executed as a single block - so a process will not move through a SEQ block one step at a time necessarily.

The special symbol 'P' does not represent a process, it indicates that a procedure has been called at that point. 'P' therefore represents the 'call point' of the procedure.

The following symbols are used to represent the various process stati :-

- * - An active process - may be chosen for execution at any time.
- a - Process waiting for one or more ALT guards to become TRUE.
- w - Process waiting for a clock time or for input/output.
- c - Process is waiting for one or more child PAR processes to terminate.

In addition break and trace points are indicated in the column by giving a 'T' for a trace point and 'B' for a break point.

So a display of :-

```
316:3*w      : occam.s ? razor
```

indicates that there are three active processes and one process waiting input on line 316.

Keyboard and Screen input/output

Because the debug display routine is fully interactive the screen and keyboard data from the program can not be handled in the same manner as normal. Input for the keyboard must be input using the 'i' command - a whole line can be input and will be buffered up for program input in this way. Screen output should be displayed as it is produced (but a copy of it will be sent to the screen image that will redisplayed on exit from the display debugger) or the 'V' command. Strings can have escapes in them '*n' means newline, '*r' carriage return and '**' space.

Non standard occam features

This compiler to the best of my knowledge (Mr.R.P. Stallard of the Department of Computer Studies, Loughborough University of Technology, U.K.) implements the occam language as defined in the occam programming manual published by INMOS limited subject to a few restrictions and extensions that are described in this file. These differences are intended to make transfer of occam programs from different implementations feasible. It is intended to be compatible to the INMOS booklet version and the Prentice Hall book definition. OCCAM, INMOS and Transputer are registered trademarks of the INMOS Group of Companies.

INMOS proto-occam language revisions

The following additional features introduced into INMOS occam products can now be selected by the compiler flag option '-I'.

STOP primitive.

TIME channel.

IF on finding none of the conditions TRUE STOPs.

Restrictions

These restrictions are either optional features as described in the published language definition or compiler restrictions unlikely to limit ordinary use of occam.

No configuration section rules.

The operator '>>' uses VAX shift right operator.

No prioritized PAR, all parallel processes have equal priority.

Number of arguments to a procedure limited to 255 maximum.

AFTER returns a time difference not a boolean value.

Extensions

PAR replicator count and base can be variables

A variable number of processes can be created by replicated PAR.

Recursive calls to procedures permitted

A procedure can call itself.

Screen channel can be used by more than one process

The special screen channel can be accessed by any number of different occam processes. This facilitates debugging of occam programs and is not difficult to implement.

Multiple source file compilation

Procedures and Variables can be defined in one file and referenced in another. The definition is preceded by the new keyword 'LIBRARY' before 'PROC' and the definition must be at the outer level of program nesting.

References to procedures in other files are defined by preceding 'PROC' by

'EXTERNAL' and replacing the '=' start of procedure definition by ':' to indicate end of definition.

e.g.

File main.occ

File sub.occ

EXTERNAL PROC f (value n) :	LIBRARY PROC f (value n) =
SEQ	SEQ
f (27)	num.to.screen (n*102)
	str.to.screen ("Enter next");

The two files can be compiled by :-

occam main.occ sub.occ	to compile both together
occam sub.occ -l	to compile sub.occ separately
occam main.occ sub.o	to link in the pre-compiled sub.occ file

In 5.0 this has been extended to variables and channels, in the case of vectors of variables and channels the size need not be specified but the type must be :-

Defining file :-

```
LIBRARY CHAN network,comms [56] :
LIBRARY VAR blot [BYTE 4],spot [42] :
LIBRARY VAR FLOAT hyper,bolic [2],active [17] :
```

Referring file :-

```
EXTERNAL CHAN network,comms [] :
EXTERNAL VAR blot [BYTE],spot [],bolic [FLOAT] :
EXTERNAL VAR FLOAT hyper,active [] :
```

Floating point arithmetic

The compiler permits the use of floating point numbers and arithmetic operators. The compiler uses 32 bit VAX floating point throughout.

Floating point numbers are declared by following VAR by the new keyword float :-

VAR FLOAT x,y,factor :	-- Floating point number declaration
VAR num,ply :	-- Normal occam variables.

Floating point number constants are supported these may be in two forms with decimal point or with decimal point and exponent :-

x := 1.45

```
y := 2.3e-23 + 3.4e+1      -- Note that the exponent must be given a sign
```

The following operators may be used on floating point numbers (both operands must be floating point)

```
+ - * / < > <= >= = <> - (monadic minus)
```

```
x := 1.3 + (y * factor)
```

```
IF
```

```
  x > 67.8
```

```
    y := -3.4      -- Note use of monadic minus.
```

Parameters to procedures must also have type set to VAR FLOAT or VALUE FLOAT - the actual parameters must be of the same type.

```
PROC sum (VALUE FLOAT a [],b [],VAR FLOAT res [],VALUE n) =
```

```
  PAR i = [0 FOR n]
```

```
    res [i] := a [i] + b [i] :
```

```
VAR FLOAT t [23],s [45],w [32] :
```

```
--
```

```
--
```

```
sum (t,s,w,12)
```

Floating values may be transmitted along channels - but there are no checks that the sender and receiver both expect floating point values. Input of floating point numbers can be carried out by calling the library routine 'fp.num.from.keyboard' and output by the routine 'fp.num.to.screen'.

Interconversion of floating point and integers is performed by the assignment operator :-

```
num := x      -- Convert floating 'x' to integer 'num'
```

```
y := num      -- Convert integer 'num' to floating 'y'
```

Attempts to use logical and shift operators on floating point numbers are flagged as errors.

APPENDIX II

THE SOFT-SYSTOLIC SIMULATION SYSTEM

(SSSS) PROGRAM LISTINGS

1. ISA
2. PROCESSING CELL
3. PLUG
4. RISAL COMPILER


```

        PAR j=[1 for n]
            VAR t1 :
                SEQ
                    loc(j,1,t1)
                    out[t1]!0
TRUE
    SEQ
        SEQ j=[1 for n]
            SEQ
                num.from.chan(ptr,buffer[j-1])
--            IF
--                t = 0
--                SEQ
--                    num.to.screen(buffer[j-1])
--                    str.to.screen(" ")
        PAR j=[ 1 for n]
            VAR t1 :
                SEQ
                    loc(j,1,t1)
                    out[t1]!buffer[j-1]
close.file(ptr)
str.to.screen("*n Source closed")
link!0 :

-- Garbage collector.

PROC sink( CHAN in[], link ) =
    VAR i,j, k :
    SEQ
        link?k
        SEQ i=[1 for k]
            PAR j =[1 for n]
                VAR t1 :
                    SEQ
                        loc(j,n,t1)
                        in[t1+1]?any
            str.to.screen("*nSink closed")
            link?any :

-- Data bus expander.

PROC data.source( CHAN ans[],bns[],awe[],bwe[],link ) =
    DEF n2=2*n,n3=3*n :
    VAR k,i,j,t :
    VAR FLOAT buffer[4*n] :
    CHAN ptr :
    SEQ
        open.file("datain","r",ptr)
        num.from.chan(ptr,k)
        link!k
        str.to.screen("*nk = ")
        num.to.screen(k)
        SEQ i=[1 for k ]
            SEQ
                str.to.screen("*ni = ")
                num.to.screen(i)
                SEQ j=[ 0 for 4]
                    IF
                        i <= k
                        SEQ
                            num.from.chan(ptr,t)
--                            str.to.screen("*n")
                            IF
                                t < 0
                                SEQ z =[0 for n]

```

```

TRUE
  SEQ z =[ 0 for n]
  SEQ
    fp.num.from.chan(ptr,buffer[(j*n)+z])
    fp.num.to.screen(buffer[(j*n)+z])
--
  TRUE
    SEQ z=[0 for n]
    buffer[(j*n)+z] := 0.0
  PAR j=[1 for n]
    VAR t1,t2 :
    SEQ
      loc(j,1,t1)
      loc(j,n,t2)
      t2 := t2 + 1
    PAR
      bns[t1]!buffer[j-1]
      bwe[t2]!buffer[n+(j-1)]
      awe[t1]!buffer[n3+(j-1)]
      ans[t2]!buffer[n2+(j-1)]
  close.file(ptr)
  str.to.screen("*n Data Source closed")
  link!0.0 :

```

-- Parallel to sequential bus condenser.

```

PROC data.sink( CHAN ans[],bns[],awe[],bwe[], link) =
  DEF n2=2*n, n3=3*n :
  VAR k,i,j :
  VAR FLOAT buffer[4*n] :
  CHAN ptr :
  SEQ
    open.file("dataout","w",ptr)
    link?k
    SEQ i=[1 for k]
      SEQ
        PAR j=[1 for n]
          VAR t1,t2 :
          SEQ
            loc(j,1,t1)
            loc(j,n,t2)
            t2 := t2 + 1
          PAR
            ans[t1]?buffer[j-1]
            awe[t2]?buffer[n+(j-1)]
            bns[t2]?buffer[n2+(j-1)]
            bwe[t1]?buffer[n3+(j-1)]
        SEQ
          SEQ j=[0 for 4]
            SEQ
              str.to.chan(ptr,"*n")
              SEQ z=[0 for n]
                SEQ
                  fp.num.to.chan(ptr,buffer[(j*n)+z])
                  str.to.chan(ptr," ")
              str.to.chan(ptr,"*n")
            close.file(ptr)
            str.to.screen("*n Data sink closed")
            link ? any
            abort.program :

```

-- Main program.

-- Setups and starts the isa grid.

```

DEF size = n*(n+1) :
CHAN ans[size],bns[size],awe[size],bwe[size],sel[size],ins[size]:

```

```
CHAN link[3] :  
VAR i,j :  
PAR
```

320

```
-- The grid.
```

```
PAR i=[1 for n]  
  PAR j=[1 for n]  
    VAR t1,t2,t3,t4 :  
      SEQ  
        loc(i,j,t1)  
        loc(j,i,t2)  
        t3 :=t1+1  
        t4 := t2 + 1  
        plug(ans[t2],awe[t3],bns[t4],bwe[t1],bns[t2],bwe[t3],  
            ans[t4],awe[t1],ins[t2],ins[t4],sel[t1],sel[t3])
```

```
-- Program interface.
```

```
source(sel,link[0],0)  
sink(sel,link[0])  
  
source(ins,link[1],1)  
sink(ins,link[1])
```

```
-- Data input/output.
```

```
data.source(ans,bns,awe,bwe,link[2])  
data.sink(ans,bns,awe,bwe,link[2])
```



```
-- PROGRAM NO. 4.3.2.
```

```
-- THE PROCESSING ELEMENT.
```

```
-- Notes:
```

```
-- General processor to illustrate the development of
-- a PE for the ISA grid, it is placed in the grid by a
-- plug procedure which allows the same definition to
-- implement a grid of processors and is control by a
-- assembler program generated by the RISAL.P compiler.
```

```
EXTERNAL proc num.to.screen(value n) :
EXTERNAL proc fp.num.to.screen(value float f) :
EXTERNAL proc str.to.screen(value s[]) :
```

```
LIBRARY PROC PE(CHAN wn,we,ws,ww,rn,re,rs,rw,
                in,is,sw,se      )=
```

```
DEF msize = 20:
```

```
VAR FLOAT a,b, mem[msize],c, i.o.buf[4] :
```

```
VAR i,j,s,port,p[4],fd[4],op,old.i,old.s :
```

```
VAR running :
```

```
SEQ
```

```
    running := true
```

```
    mem[1] := 0.0
```

```
    mem[0] := 0.0
```

```
    old.i := 0
```

```
    old.s := 0
```

```
    WHILE running
```

```
        SEQ
```

```
            -- Fetch instruction.
```

```
            c := mem[1]
```

```
            PAR
```

```
                in?i
```

```
                is!old.i
```

```
                sw?s
```

```
                se!old.s
```

```
                wn!c
```

```
                we!c
```

```
                ws!c
```

```
                ww!c
```

```
                rn?i.o.buf[0]
```

```
                re?i.o.buf[1]
```

```
                rs?i.o.buf[2]
```

```
                rw?i.o.buf[3]
```

```
            old.s := s
```

```
            old.i := i
```

```
            -- Decode instruction.
```

```
        SEQ
```

```
            SEQ j =[0 for 4]
```

```
                SEQ
```

```
                    fd[j] := i\100
```

```
                    i := i/100
```

```
            port := fd[2]
```

```
            op := fd[3]
```

```
        -- Communication enable.
```

```
    SEQ
```

```
        SEQ i=[0 for 4]
```

```
            SEQ
```

```

        p[i] := port\2
        port := port/2
SEQ i=[0 for 4]
    IF
        p[i] = 1
        mem[i+3] := i.o.buf[i]

-- Execute instruction.

a := mem[fd[1]]
b := mem[fd[0]]
IF
    (s<>0) AND (op <> 0)
    IF
        op = 1
            mem[1] := mem[0]
        op = 2
            mem[0] := a + b
        op = 3
            mem[0] := a - b
        op = 4
            mem[0] := a * b
        op = 5
            mem[0] := a / b
        op = 6
            SEQ
                IF
                    a < b
                        mem[0] := a
                    TRUE
                        mem[0] := b
        op = 7
            SEQ
                IF
                    a > b
                        mem[0] := a
                    TRUE
                        mem[0] := b
        op = 8
            mem[1] := mem[fd[1]]
        op = 9
            mem[fd[0]] := a

```

-- PROGRAM NO. 4.3.1.B.

-- THE PLUG PROCEDURE.

-- Notes:

-- Single processor plug use to plug a processor
-- into the grid.

EXTERNAL proc PE(CHAN wn,we,ws,ww,rn,re,rs,rw,in,is,sw,se):

LIBRARY PROC plug(CHAN wn,we,ws,ww,rn,re,rs,rw,in,is,sw,se)=
SEQ

PE(wn,we,ws,ww,rn,re,rs,rw,in,is,sw,se) :

```
(* PROGRAM NO. 5.3. *)
```

```
(* THE RISAL COMPILER *)
```

```
(* This compiler is used to compile the RISAL language *)
(* and develop the ISA programs *)
```

```
program lan(input,output,error,keywords);
label 99 ; (* for abort *)
const
  asig= 4; nsig=8;
  size = 20; bufsize = 80;
  com= 44; sem = 59; col = 58;
  rbk = 41; lbk = 40; aend = 101; rep =103; repl=106; p=104;
  d = 105; s = 4; none = 102;
  srbk = 93; slbk = 91;
```

```
type
```

```
  words = array[1..asig] of char;
```

```
var
```

```
  tk,f,n,k,count,statcount,rpos,cpos,int,lim:integer;
  mwords,linecount,errs,sindx : integer;
  progm,data,selector,eflag,drepflg : boolean;
  number :array[1..nsig] of char;
  saveline:array[1..2000] of integer ;
  kword:array[1..size] of words;
  tval:array[1..size] of integer;
  buf:array[1..bufsize] of char;
  error,keywords:text;
```

```
procedure intialise;
```

```
(* set up current keywords and intial values *)
```

```
var i,j : integer;
```

```
    ch : char;
```

```
begin
```

```
  (* get keywords *)
```

```
  reset(keywords);
```

```
  read(keywords,mwords); readln(keywords);
```

```
  if mwords > size then
```

```
    begin
```

```
      writeln('Too many words in vocabulary');
```

```
      goto 99
```

```
    end;
```

```
  for i := 1 to mwords do
```

```
    begin
```

```
      for j := 1 to asig do
```

```
        read(keywords,kword[i][j]);
```

```
        repeat read(keywords,ch) until ch = ' ';
```

```
        read(keywords,tval[i]); readln(keywords);
```

```
      end;
```

```
  rpos := 0; cpos:= 0; linecount := 0; errs := 0;
```

```
  eflag := false; drepflg := false;
```

```
  rewrite(error);
```

```
(* keywords array now contains keywords *)
```

```
end;
```

```
procedure m( i,j:integer);
```

```
(* error message routine *)
```

```
var k :integer;
```

```
begin
```

```

(* book keeping *)
for k := 1 to 11 do write(error, ' ');
if eflag then
  begin
    for k := 1 to rpos do write(error, buf[k]);
    writeln(error);
    eflag := false;
  end;
(* format message *)
if (i<>9) and (i<>10) then
  begin
    errs := errs + 1;
    write(error, linecount, ':');
    for k := 1 to (cpos - 1) do write(error, '-');
    write(error, '^'); writeln(error);
  end;
case i of
  1:writeln(error, 'program must start with p,d, or s');
  2:writeln(error, 'expected''(''but found'', buf[cpos], ''')');
  3:writeln(error, 'expected''(''but found'', buf[cpos], ''')');
  4:writeln(error, 'expected''::'', ''', ''', or''end''or''')');
  5:writeln(error, 'too many data elements');
  6:writeln(error, 'incorrect data boundary spec');
  7:writeln(error, 'expected integer arguments');
  8:writeln(error, 'expected ''::'', ''', ''');
  9:writeln(error, 'errors detected = ', errs);
  10:writeln(error, 'no errors detected');
  11:writeln(error, 'expected integer operands in instruction');
  12:writeln(error, 'should be real value in data expression');
  13:writeln(error, 'require integer in rep count parameter');
  14:writeln(error, 'more than required number of statements');
  15:writeln(error, 'attempt to read past end of file');
  16:writeln(error, 'alphabetic string found require keyword');
  17:writeln(error, 'invalid character');
  18:writeln(error, 'selector should be ''0'', or ''1'');
  19:writeln(error, 'malformed expression');
  20:writeln(error, 'expected ''[''', or ', ''')');
end;
(* fatal message abort *)
if j = 1 then goto 99
end;

procedure getc(var ch:char);
(* maintain buffer of characters *)
begin
  if rpos=cpos then
    if eof then m(15,1)
    else
      begin
        (* fill buffer *)
        cpos := 0; rpos := 0;
        while not eoln do
          begin
            (* skip white space *)
            rpos := rpos + 1;
            repeat
              read(buf[rpos]);
            until not (ord(buf[rpos]) in [0..9,14..31]);
          end;
          (* book keeping *)
          eflag := true; linecount := linecount + 1;
          readln;
          rpos := rpos + 1;
          buf[rpos] := ' ';
        end;
        cpos := cpos + 1;
      end;

```



```

    if ch = '-' then sign := -1;
  repeat
    if i <= nsig then
      begin
        number[i] := ch;
        int := int*10 +(ord(ch)-ord('0'));
        getc(ch);
        i := i + 1;
      end
    until not ( ch in digit);
    int := int * sign;
    if ch = '.' then
      begin
        f := 1;
        repeat
          if i <= nsig then number[i] := ch;
          getc(ch);
          i := i + 1;
        until not(ch in digit);
      end;
    cpos := cpos - 1;
    token := i-1;
    (* integers are converted to numeric value
       reals remain as strings *)
  end
else if ch in punc then
  (* punctutaion symbols *)
  begin
    f := 3;
    token := ord(ch);
  end
else m(17,0)
end;

```

```

procedure outpt(lim,com,typ :integer);
(* construction of data,program or selector file *)
var i,j : integer;
begin
  (* decides on replicated construct and checks
     for more data than specified *)
  if (lim > n-count) and (not drepflg) then m(6,1)
  else if lim = 0 then
    begin
      if typ = 1 then
        for i:= 1 to com do
          write(number[i])
        else
          begin sindx := sindx + 1;
            saveline[sindx] := com;
            write(com);
            end;
        count := count + 1;
        if count < n then write(' ')
        else writeln;
      end
    else
      for i := 1 to lim do
        begin
          if typ=1 then
            for j := 1 to com do
              write(number[j])
            else
              begin sindx := sindx + 1;
                saveline[sindx] := com;
                write(com);
                end;
          end
        end
      end;

```

```

        count := count + 1;
        if count < n then write(' ')
        else writeln;
    end;
end;

procedure repr( var lim:integer);
(* process replicator *)
begin
    if token(f) <> lbk then m(2,0); tk := token(f);
    if f=0 then lim := int else m(13,1);
    if token(f) <> rbk then m(3,0); tk := token(f);
end;

procedure sline;
(* line of selector file *)
begin
    repeat
        if (f=0) and ((int = 0) or (int=1)) then outpt(0,int,0)
        else if tk = rep then
            begin
                repr(lim);
                if (f=0) and ((int=0) or (int=1)) then
                    outpt(lim,int,0)
                else m(18,1);
            end
        else m(19,1);
        tk := token(f);
        if (tk<>col) and (tk<>com) and (tk <> aend)
            and (tk <> srbk) then m(4,0);
        if (tk = com) or (tk = srbk) then tk := token(f);
    until (tk = col) or (tk = aend) ;
end;

procedure dline;
(* line of data *)
var save : integer;
begin
    if tk = none then
        begin
            save := count; count := n;
            outpt(0,-1,0);
            count := save;
            tk := token(f);
        end
    else
        begin
            if tk <> rep then
                outpt(0,0,0); count := count - 1;
            while (tk <> sem) and (tk <> col) and (tk<>aend) do
                begin
                    if (tk = rep) then
                        begin
                            repr(lim);
                            save := count; count := n;
                            drepflg := true;
                            outpt(lim,-1,0);
                            drepflg := false;
                            count := save;
                        end
                    else begin tk := token(f); if f= 1
                        then outpt(0,tk,1) else m(12,0);
                        end;
                    tk := token(f);
                    if tk = srbk then tk := token(f)
                end;
            end;
        end;
end;

```


end;

```

function inst: integer;
(* check instruction format *)
var coma,port,j : integer;
begin
  (* decipher communication ports *)
  port := 0; j:=1; coma := tk;
  repeat
    tk := token(f);
    if (tk in [1,2,4,8]) and (j<=4)then
      begin
        port := port + tk; j := j + 1;
      end
    else
      if (j > 4) and (tk in [1,2,4,8] ) then m(5,1);
    until tk = com;
    (* construct ISA assembler instruction *)
    coma := coma * 100 + port ;tk := token(f);
    if f=0 then coma := coma*100 + int else m(11,0);
    if token(f) <> com then m(4,0); tk:= token(f);
    if f=0 then coma := coma*100 + int else m(11,0);
    inst := coma;
  end;
end;

```

```

procedure instruction;
(* instruction line *)
var lim : integer;
begin
  lim := 0;
  if tk <> rep then outpt(lim,inst,0)
  else
    begin
      repr(lim); outpt(lim,inst,0);
    end;
  end;
end;

```

```

procedure line ;
(* process a general line *)
var j,c1 ,i,l: integer;
begin
  j := 9;
  if progrm then
    (* line is in program *)
    repeat
      tk := token(f);
      if tk = repl then
        begin
          sindx := 0;
          repr(c1);
          if tk <> slbk then m(20,0);
          tk := token(f); instruction ;
          for i := 1 to (c1-1)
            do begin for l := 1 to sindx do
              begin
                write(saveline[l]);
                if l < sindx then
                  write(' ');
              end;
            writeln
          end;
          tk := token(f);
          if tk <> srbk then m(20,0);
        end
      else instruction;
      tk := token(f);
    end;
  end;
end;

```

```

    if (tk <> col) and (tk<>sem) and
        (tk <> aend) then m(4,0);
    until (tk = col) or (tk=aend)
else if data then
    (* line is from data *)
    repeat
        count := 0;
        tk := token(f);
        if j > 8 then j := 1;
        if (tk<>j) and (tk<>none) and (tk<>rep)
            and (tk <> repl) then m(8,0);
        if tk = repl then
            begin sindx := 0;
                repr(c1);
                if tk <> slbk then m(20,0);
                tk := token(f);dline;
                for i := 1 to c1-1 do
                    begin
                        for l := 1 to sindx do
                            writeln(saveline[l]);
                    end
                end
            end
        else dline;
        j := j * 2;
        if (tk<>col) and (tk<>sem)
            and (tk <> aend) then m(4,0);
    until (tk=col) or (tk = aend)
else if selector then
    (* line is from selector *)
    begin
        tk := token(f);
        if tk = repl then
            begin
                sindx := 0;
                repr(c1);
                if tk <> slbk then m(20,0);
                tk := token(f); sline;
                for i := 1 to c1-1 do
                    begin for l := 1 to sindx do
                        begin write(saveline[l]);
                            if l < sindx then
                                write(' ');
                        end;
                    end;
                    writeln
                end;
            end
        else sline;
    end
    else m(1,1);
end;

procedure setup;
(* decipher file header*)
begin
    tk := token(f);
    if f=0 then n:= int else m(7,1);
    if token(f) <> com then m(4,0);
    tk := token(f);
    if f=0 then k := int else m(7,1);
    write(k);writeln;
end;

procedure prog;
(* process input file *)
begin
    prog := false; data:= false; selector := false;

```

```

tk := token(f);
(* decide file type *)
if (tk<>p) and (tk<>s) and (tk<>d) then m(1,1)
else
  case tk of
    p : progm := true;
    d : data := true;
    s : selector:=true;
  end;
(* dimensions of ISA *)
if token(f) <> lbk then m(2,0);
setup;
if token(f) <> rbk then m(3,0);
statcount := 0;
(* process lines of file *)
repeat
  count := 0;
  line;
  if (tk<>aend) and (tk<>col) then m(4,0);
  statcount:= statcount + 1;
  if statcount>k then m(14,0);
until tk = aend;
{if errs = 0 then m(10,0) else m(9,0)}
end;

(* main program *)

begin
  initialise;
  prog;
99 :end.

```

APPENDIX III

RISAL PROGRAM LISTINGS

```

p(4,34)
{ N0. 6.2.1 }
{ Program for matrix transpose 4*4 }
data n,3,0; rep(3) null ,0,0:
rep(2) data n,3,0; rep(2) null ,0,0:
rep(3) data n,3,0; null ,0,0:
rep(4) data n,3,0 :
null ,0,0; rep(3) data n,3,0:
rep(2) null,0,0; rep(2) data n,3,0:
rep(3) null ,0,0; data n,3,0:
rep(4) null ,0,0:
data e,4,0; data w,6,0; rep(2) null ,0,0:
data n,3,0; rep(3) null,0,0:
data s,5,0; null,0,0; data e,4,0; data w,6,0:
data e,4,0; data w,6,0; data n,3,0; null ,0,0:
rep(2) null,0,0; data s,5,0; null ,0,0:
data e,4,0; data w,6,0; data e,4,0; data w,6,0:
null ,0,0; data e,4,0; data w,6,0; null ,0,0:
data e,4,0; data w,6,0; data e,4,0; data w,6,0 :
data n,3,0; data e,4,0; data w,6,0; null ,0,0:
data s,5,0; data n,3,0; data e,4,0; data w,6,0:
data n,3,0; data s,5,0; rep(2) null ,0,0:
data s,5,0; data n,3,0; rep(2) null ,0,0:
null ,0,0; data s,5,0; rep(2) null ,0,0:
data e,4,0; data w,6,0; rep(2) null ,0,0:
null ,0,0; data e,4,0; data w,6,0; null ,0,0:
data e,4,0; data w,6,0; data e,4,0; data w,6,0:
null ,0,0; data e,4,0; data w,6,0; null ,0,0:
data s,5,0; null ,0,0; data e,4,0; data w,6,0:
rep(2) data s,5,0; rep(2) null ,0,0:
rep(3) data s,5,0; null ,0,0:
repl(2)[rep(4) data s,5,0]:
null ,0,0; rep(3) data s,5,0:
rep(2) null ,0,0; rep(2) data s,5,0:
rep(3) null ,0,0; data s,5,0:
rep(4) null ,0,0
end

```

```

d(4,34)
{ N0. 6.2.1 }
{ Data file for matrix transpose 4*4 }
n 13.0,0.0,0.0,0.0;
none;none;none:
n 9.0,14.0,0.0,0.0;
none;none;none:
n 5.0,10.0,15.0,0.0;
none;none;none:
n 1.0,6.0,11.0,16.0;
none;none;none:
n 0.0,2.0,7.0,12.0;
none;none;none:
n 0.0,0.0,3.0,8.0;
none;none;none:
n 0.0,0.0,0.0,4.0;
none;none;none:
repl(27)[rep(4) none]
end

```

```

s(4,34)
{ NO. 6.2.1 }
{ Selector file for matrix transpose 4*4 }
1,0,0,0:
1,1,0,0:
1,1,1,0:
1,1,1,1:
repl(3)[0,0,0,0]:
repl(2)[1, rep(3) 0]:
0,0,1,0:
rep(3) 1, 0:
1, rep(3) 0 :
1,0,1,1:
rep(3) 1, 0:
repl(2)[1,1,0,0]:
0,1,0,0:
1,1,0,0:
0,1,1,0:
rep(4)1:
rep(3) 1,0:
rep(4) 1 :
repl(3)[1,1,0,0]:
repl(2)[rep(4)0]:
rep(3) 1,0:
1,1,0,0:
1,0,0,0:
repl(4)[rep(4) 0]:
end

```

```

p(4,38)
{ NO. 6.2.2 }
{ Program for LU decomposition 4*4 matrix }
{ load matrix}
data n,3,0; rep(3) null n,0,0 :
rep(2) data n,3,0; rep(2) null n,0,0:
rep(3) data n,3,0; null n,0,0 :
rep(4) data n,3,0:
{ start factorisation}
mov s,1,7; rep(3) data n,3,0:
data n,3,0; mov s,1,7; rep(2) data n,3,0:
div ,7,3; data n,3,0; mov s,1,7; data n,3,0:
copy ,0,0; null ,0,0; data n,3,0; mov s,1,7:
null ,0,0; data w,6,0; null ,0,0; data n,3,0 :
null ,0,0; mult ,3,6; data w,6,0; null ,0,0:
null ,0,0; sub ,7,0 ; mult ,3,6; data w,6,0 :
null ,0,0; copy ,0,0; sub ,7,0; mult ,3,6 :
null ,0,0; mov s,1,7; copy ,0,0; sub ,7,0 :
null ,0,0; data n,3,0; mov s,1,7; copy ,0,0 :
null ,0,0; div ,7,3; data n,3,0; mov s,1,7:
null ,0,0; copy ,0,0; null ,0,0; data n,3,0:
rep(2) null ,0,0; data w,6,0; null ,0,0:
rep(2) null ,0,0; mult ,3,6; data w,6,0:
rep(2) null ,0,0; sub ,7,0; mult ,3,6:
rep(2) null ,0,0; copy ,0,0; sub ,7,0:
rep(2) null ,0,0; mov s,1,7; copy ,0,0:
rep(2) null ,0,0; data n,3,0; mov s,1,7:
rep(2) null ,0,0; div ,7,3; data n,3,0 :
rep(2) null ,0,0; copy ,0,0; null ,0,0 :
rep(3) null ,0,0; data w,6,0:
rep(3) null ,0,0; mult ,3,6:
rep(3) null ,0,0; sub ,7,0:
rep(3) null ,0,0; copy ,0,0:
{ read result}
data s,5,0 ; rep(3) null ,0,0:
rep(2) data s,5,0; rep(2) null,0,0:
rep(3) data s,5,0; null,0,0:
repl(3)[rep(4) data s,5,0]:
null ,0,0; rep(3) data s,5,0:
rep(2) null ,0,0; rep(2) data s,5,0:
rep(3) null ,0,0; data s,5,0:
rep(4) null ,0,0
end

```

```

d(4,38)
{ NO. 6.2.2 }
{ Data file for LU decomposition 4*4 matrix }
n 3.0,0.0,0.0,0.0;
none; none; none:
n 2.0,4.0,0.0,0.0;
none; none; none:
n 4.0,2.0,1.0,0.0;
none; none; none :
n 2.0,1.0,5.0,2.0;
none; none; none :
n 0.0,3.0,2.0,1.0;
none; none; none:
n 0.0,0.0,3.0,3.0;
none; none; none:
n 0.0,0.0,0.0,2.0;
none; none; none:

```

```
repl(31)[rep(4) none]
end
```

335

```
s(4,38)
{ NO. 6.2.2 }
{ Selector file for LU decomposition 4*4 matrix }
1, rep(3)0 :
rep(2)1, rep(2)0:
rep(3)1, 0:
rep(4)1 :
rep(4)0 :
0 ,1,rep(2)0:
0 ,1,1,0:
repl(5)[0, rep(3)1]:
repl(8)[rep(2) 0, rep(2)1]:
repl(8)[rep(3) 0, 1]:
repl(3)[rep(4) 0]:
rep(4) 1:
rep(3) 1,0:
rep(2) 1, rep(2) 0 :
1,rep(3) 0:
repl(3)[rep(4) 0]
end
```



```

p(4,34)
{ NO. 6.2.3 }
{ Program for 4*4 matrix-vector multiplication }
{ calculation }
rep(4) mult n w,3,6:
rep(4) add ,7,0:
rep(4) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,7,0:
rep(4) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,7,0:
rep(4) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,7,0:
rep(4) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,7,0:
rep(4) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,7,0:
rep(4) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,7,0:
rep(4) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mov ,7,0:
rep(4) copy ,0,0:
repl(4)[rep(4) null ,0,0]
end

```

```

d(4,34)
{ NO. 6.2.3 }
{ Data file for 4*4 matrix-vector multiplication }
n 2.8,0.0,0.0,0.0;none;none;w 2.1,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 3.0,3.6,0.0,0.0;none;none;w 3.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 2.0,4.8,4.0,0.0;none;none;w 5.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 5.1,6.0,3.0,4.2;none;none;w 6.6,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 0.0,8.0,2.2,1.0;none;none;w 0.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 0.0,0.0,6.1,0.0;none;none;w 0.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 0.0,0.0,0.0,9.0;none;none;w 0.0,0.0,0.0,0.0:
repl(9)[rep(4) none]
end

```

```
s(4,34)
{ NO. 6.2.3 }
{ Selector file for 4*4 matrix-vector multiplication }
repl(34)[1,rep(3)0]
end
```

```

p(4,85)
{ NO. 6.2.4 }
{ Program for 4*4 matrix-matrix multiplication }
rep(4) mult n w,3,6:
rep(4) add ,7,0:
rep(4) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,7,0:
rep(4) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,7,0:
rep(4) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,7,0:
rep(4) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
add ,8,0;rep(3) add ,7,0:
mov ,0,8;rep(3) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(2) add ,8,0;rep(2) add ,7,0:
rep(2) mov ,0,8;rep(2) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(3) add ,8,0;add ,7,0:
rep(3) mov ,0,8;mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,8,0:
rep(4) mov ,0,8:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
add ,9,0;rep(3) add ,8,0:
mov ,0,9;rep(3) mov ,0,8:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(2) add ,9,0;rep(2) add ,8,0:
rep(2) mov ,0,9;rep(2) mov ,0,8:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(3) add ,9,0;add ,8,0:
rep(3) mov ,0,9;mov ,0,8:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,9,0:
rep(4) mov ,0,9:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
add ,10,0;rep(3) add ,9,0:
mov ,0,10;rep(3) mov ,0,9:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(2) add ,10,0;rep(2) add ,9,0:
rep(2) mov ,0,10;rep(2) mov ,0,9:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(3) add ,10,0;add ,9,0:
rep(3) mov ,0,10;mov ,0,9:

```

```

rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,10,0:
rep(4) mov ,0,10:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,10,0:
rep(4) mov ,0,10:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,10,0:
rep(4) mov ,0,10:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,10,0:
rep(4) mov ,0,10:
rep(4) mov ,6,1:
rep(4) mov ,7,0:
rep(4) copy ,0,0:
rep(4) mov ,8,0:
rep(4) copy ,0,0:
rep(4) mov ,9,0:
rep(4) copy ,0,0:
rep(4) mov ,10,0:
rep(4) copy ,0,0:
rep(4) null ,0,0
end

```

```

d(4,85)
{ N0. 6.2.4 }
{ Data file for the 4*4 matrix-matrix multiplication }
n 2.1,0.0,0.0,0.0;none;none;w 2.8,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 3.0,0.0,0.0,0.0;none;none;w 3.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 5.0,1.0,2.3,0.0;none;none;w 2.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 6.6,1.2,5.0,1.8;none;none;w 5.1,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 2.1,2.2,2.0,6.1;none;none;w 3.6,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 3.0,0.0,0.0,3.3;none;none;w 4.8,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 5.0,1.0,2.3,3.6;none;none;w 6.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 6.6,1.2,5.0,1.8;none;none;w 8.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 2.1,2.2,2.0,6.1;none;none;w 4.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 3.0,0.0,0.0,3.3;none;none;w 3.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 5.0,1.0,2.3,3.6;none;none;w 2.2,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 6.6,1.2,5.0,1.8;none;none;w 6.1,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 2.1,2.2,2.0,6.1;none;none;w 4.2,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 3.0,0.0,0.0,3.3;none;none;w 1.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 5.0,1.0,2.3,3.6;none;none;w 0.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 6.6,1.2,5.0,1.8;none;none;w 9.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 0.0,2.2,2.0,6.1;none;none;w 0.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:

```

```
n 0.0,0.0,0.0,3.3;none;none;w 0.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 0.0,0.0,0.0,3.6;none;none;w 0.0,0.0,0.0,0.0:
repl(12)[rep(4)none]
end
```

340

```
s(4,85)
{ NO. 6.2.4 }
{ Selector file for the 4*4 matrix-matrix multiplication }
repl(85)[1,rep(3)0]
end
```

```

p(4,147)
{ NO. 6.3 }
{Program for the solution of linear systems }
data n,3,0; rep(3) null n,0,0 :
rep(2) data n,3,0; rep(2) null n,0,0:
rep(3) data n,3,0; null n,0,0 :
rep(4) data n,3,0:
{ start factorisation}
mov s,1,7; rep(3) data n,3,0:
data n,3,0; mov s,1,7; rep(2) data n,3,0:
div ,7,3; data n,3,0; mov s,1,7; data n,3,0:
copy ,0,0; null ,0,0; data n,3,0; mov s,1,7:
null ,0,0; data w,6,0; null ,0,0; data n,3,0 :
null ,0,0; mult ,3,6; data w,6,0; null ,0,0:
null ,0,0; sub ,7,0 ; mult ,3,6; data w,6,0 :
null ,0,0; copy ,0,0; sub ,7,0; mult ,3,6 :
null ,0,0; mov s,1,7; copy ,0,0; sub ,7,0 :
null ,0,0; data n,3,0; mov s,1,7; copy ,0,0 :
null ,0,0; div ,7,3; data n,3,0; mov s,1,7:
null ,0,0; copy ,0,0; null ,0,0; data n,3,0:
rep(2) null ,0,0; data w,6,0; null ,0,0:
rep(2) null ,0,0; mult ,3,6; data w,6,0:
rep(2) null ,0,0; sub ,7,0; mult ,3,6:
rep(2) null ,0,0; copy ,0,0; sub ,7,0:
rep(2) null ,0,0; mov s,1,7; copy ,0,0:
rep(2) null ,0,0; data n,3,0; mov s,1,7:
rep(2) null ,0,0; div ,7,3; data n,3,0 :
rep(2) null ,0,0; copy ,0,0; null ,0,0 :
rep(3) null ,0,0; data w,6,0:
rep(3) null ,0,0; mult ,3,6:
rep(3) null ,0,0; sub ,7,0:
mov s,1,8; rep(2) null ,0,0; copy ,0,0:
rep(4) null ,0,0:
null ,0,0; mov s,1,8; rep(2) null ,0,0:
rep(4) null ,0,0:
mov s,1,7; null ,0,0; mov s,1,8; null ,0,0:
rep(4) null ,0,0:
null ,0,0; mov s,1,7; null ,0,0; mov s,1,8:
rep(4) null ,0,0:
rep(2) null ,0,0; mov s,1,7; null ,0,0:
{ read result}
data s,5,0 ; rep(3) null ,0,0:
rep(2) data s,5,0; rep(2) null,0,0:
rep(3) data s,5,0; null,0,0:
repl(3)[rep(4) data s,5,0]:
null ,0,0; rep(3) data s,5,0:
rep(2) null ,0,0; rep(2) data s,5,0:
rep(3) null ,0,0; data s,5,0:
rep(4) null ,0,0:
data n,3,0; rep(3) null ,0,0:
rep(4) null ,0,0:
null ,0,0; data n,3,0; rep(2) null ,0,0:
rep(4) null ,0,0:
mov s,1,7; null ,0,0; data n,3,0; null ,0,0:
rep(4) null ,0,0:
null ,0,0; mov s,1,7; null ,0,0; data n,3,0:
rep(4) null ,0,0:
rep(2) null ,0,0; mov s,1,7; null ,0,0:
rep(4) null ,0,0:
data n,3,0; rep(2) null ,0,0; mov s,1,7:
mult ,3,7; rep(3) null n,0,0:
copy ,0,0; rep(3) null n,0,0:

```

```

null n,0,0; data w,6,0; rep(2) null n,0,0:
null n,0,0; data n,3,0; rep(2) null n,0,0:
rep(4) null n,0,0:
null n,0,0; sub ,3,6; rep(2) null n,0,0:
null n,0,0; copy ,0,0; rep(2) null n,0,0:
null n,0,0; data n,3,0; rep(2) null n,0,0:
rep(4) null n,0,0:
null n,0,0; mult ,3,7; rep(2) null n,0,0:
null n,0,0; add ,6,0; rep(2) null n,0,0:
null n,0,0; copy ,0,0; rep(2) null n,0,0:
rep(2) null n,0,0; data w,6,0; null n,0,0:
rep(2) null n,0,0; data n,3,0; null n,0,0:
rep(4) null n,0,0:
rep(2) null n,0,0; sub ,3,6; null n,0,0:
rep(2) null n,0,0; copy ,0,0; null n,0,0:
rep(2) null n,0,0; data n,3,0; null n,0,0:
rep(4) null n,0,0:
rep(2) null n,0,0; mult ,3,7; null n,0,0:
rep(2) null n,0,0; add ,6,0; null n,0,0:
rep(2) null n,0,0; copy ,0,0; null n,0,0:
rep(3) null n,0,0; data w,6,0:
rep(3) null n,0,0; data n,3,0:
rep(4) null n,0,0:
rep(3) null n,0,0; sub ,3,6:
rep(3) null n,0,0; copy ,0,0:
rep(3) null n,0,0; data s,5,0:
rep(3) null n,0,0; data s,5,0:
rep(3) null n,0,0; data s,5,0:
rep(2) null ,0,0; rep(2) data s,5,0:
rep(2) null ,0,0; rep(2) data s,5,0:
null n,0,0; rep(3) data s,5,0:
rep(4) data s,5,0:
repl(4)[rep(4) null n,0,0]:
rep(3) null n,0,0; div ,0,8:
rep(3) null n,0,0; copy ,0,0:
rep(4) null n,0,0:
rep(3) null n,0,0; data s,5,0:
rep(3) null n,0,0; mult ,5,8:
rep(3) null n,0,0; copy ,0,0:
rep(2) null n,0,0; data e,4,0; null n,0,0:
repl(2)[rep(4) null n,0,0]:
rep(2) null n,0,0; sub ,0,4; null n,0,0:
rep(2) null n,0,0; div ,0,8; data s,5,0:
rep(2) null n,0,0; copy ,0,0; null n,0,0:
rep(4) null n,0,0:
rep(2) null n,0,0; data s,5,0; null n,0,0:
rep(2) null n,0,0; mult ,5,8; data s,5,0:
rep(2) null n,0,0; copy ,0,0; mult ,5,8:
rep(2) null n,0,0; mov s,1,8; copy ,0,0:
rep(2) null n,0,0; data e,4,0; null ,0,0:
rep(2) null n,0,0; add ,4,8; null n,0,0:
rep(2) null n,0,0; copy ,0,0; null n,0,0:
null n,0,0; data e,4,0; rep(2) null n,0,0:
rep(4) null n,0,0:
rep(3) null n,0,0; data s,5,0:
null ,0,0; sub ,0,4; rep(2) null ,0,0:
null ,0,0; div ,0,8; data s,5,0; null ,0,0:
null ,0,0; copy ,0,0; rep(2) null ,0,0:
rep(4) null ,0,0:
null ,0,0; data s,5,0; rep(2) null ,0,0:
null ,0,0; mult ,5,8; data s,5,0; null ,0,0:
null ,0,0; copy ,0,0; mult ,5,8; data s,5,0:
null ,0,0; mov s,1,8; copy ,0,0; mult ,5,8:
rep(2) null ,0,0; mov s,1,8; copy ,0,0:
rep(2) null ,0,0; data e,4,0; null ,0,0:
rep(2) null ,0,0; add ,4,8; null ,0,0:

```

```

rep(2) null ,0,0; copy ,0,0; null ,0,0:
null ,0,0; data e,4,0; rep(2) null ,0,0:
null ,0,0; add ,4,8; rep(2) null ,0,0:
null ,0,0; copy ,0,0; rep(2) null ,0,0:
data e,4,0; rep(3) null ,0,0:
rep(4) null ,0,0:
sub ,0,4; rep(3) null ,0,0:
div ,0,8; rep(3) null ,0,0:
copy ,0,0; rep(3) null ,0,0:
repl(3)[rep(4) null ,0,0]:
rep(3) null ,0,0; data s,5,0:
rep(2) null ,0,0; rep(2) data s,5,0:
rep(2) null ,0,0; rep(2) data s,5,0:
null ,0,0; rep(3) data s,5,0:
null ,0,0; rep(3) data s,5,0:
rep(4) data s,5,0
end

```

```

d(4,147)
{ NO. 6.3 }
{ Data file for the solution of linear systems }
n 2.0,0.0,0.0,0.0;
none; none; none:
n 2.0,3.0,0.0,0.0;
none; none; none:
n 4.0,3.0,3.0,0.0;
none; none; none :
n 2.0,1.0,6.0,3.0;
none; none; none :
n 0.0,3.0,2.0,1.0;
none; none; none:
n 0.0,0.0,3.0,3.0;
none; none; none:
n 0.0,0.0,0.0,2.0;
none; none; none:
repl(39)[rep(4) none]:
n 1.0,0.0,0.0,0.0; none;none;none:
none; none; none; none:
n 0.0,1.0,0.0,0.0; none;none;none:
none; none; none; none:
n 0.0,0.0,1.0,0.0; none;none;none:
none; none; none; none:
n 0.0,0.0,0.0,1.0; none;none;none:
repl(3)[rep(4) none]:
n 10.0,0.0,0.0,0.0; none;none;none:
repl(3)[rep(4) none]:
n 0.0,10.0,0.0,0.0; none;none;none:
repl(9)[rep(4) none]:
n 0.0,0.0,12.0,0.0; none;none;none:
repl(9)[rep(4) none]:
n 0.0,0.0,0.0,11.0; none;none;none:
repl(66)[rep(4) none]
end

```

```

s(4,147)
{ NO. 6.3 }
{ Selector file for the solution of linear systems }
1, rep(3)0 :
rep(2)1, rep(2)0:
rep(3)1, 0:
rep(4)1 :
rep(4)0 :
0 ,1,rep(2)0:
0 ,1,1,0:

```



```

repl(5)[0, rep(3)1]:
repl(8)[rep(2) 0, rep(2)1]:
repl(7)[rep(3) 0, 1]:
1,0,0,1:
1,0,0,0:
repl(2)[1,1,0,0]:
repl(2)[0,1,1,0]:
repl(2)[0,0,1,1]:
repl(2)[0,0,0,1]:
repl(2)[rep(4) 0]:
rep(4) 1:
rep(3) 1,0:
rep(2) 1, rep(2) 0 :
1,rep(3) 0:
repl(3)[rep(4) 0]:
repl(2)[1,0,0,0]:
repl(2)[1,1,0,0]:
1,1,1,0:
0,0,1,0:
0,1,0,1:
0,0,0,0:
0,0,1,0:
0,0,0,0:
1,0,0,1:
1,1,0,0:
1,1,1,0:
repl(2)[rep(4)1]:
rep(3)0,1:
repl(2)[0,1,rep(2)0]:
rep(4)0:
0,1,1,0:
rep(4)0:
0,0,1,1:
1,0,1,1:
rep(4)1:
0, rep(3)1:
rep(4)0:
repl(2)[0,0,1,0]:
rep(4)0:
0,0,1,1:
rep(4)0:
1,0,0,1:
1,1,0,1:
0,rep(3)1:
0,0,1,1:
rep(4)0:
repl(2)[rep(3)0,1]:
rep(4)1:
repl(4)[rep(3)1,0]:
1,1,0,0:
1,rep(3)0:
repl(4)[rep(4)0]:
repl(2)[rep(3)0,1]:
0,0,1,1:
0,0,1,0:
0,1,1,0:
1,0,1,0:
rep(3)1,0:
0,1,1,0:
rep(4)0:
0,0,1,1:
0,0,1,0:
repl(2)[0,1,1,0]:
repl(7)[0,1,0,0]:
1,1,1,0:
1,1,0,0:

```

```
rep(4)0:
0,1,1,0:
0,1,0,0:
repl(2)[1,1,0,0]:
repl(16)[1,0,0,0]:
repl(3)[rep(4)0]:
rep(4)1:
1,1,1,0:
repl(2)[1,1,0,0]:
repl(2)[1,rep(3)0]
end
```

```

p(4,300)
{ NO. 6.4 }
{ Program for finding the g-inverse of a rec-matrix }
data n,3,0; rep(3) null ,0,0:
rep(2) data n,3,0; rep(2) null ,0,0:
rep(3) data n,3,0; null ,0,0:
rep(4) data n,3,0 :
null ,0,0; rep(3) data n,3,0:
rep(2) null,0,0; rep(2) data n,3,0:
rep(3) null ,0,0; data n,3,0:
rep(4) null ,0,0:
data e,4,0; data w,6,0; rep(2) null ,0,0:
data n,3,0; rep(3) null,0,0:
data s,5,0; null,0,0; data e,4,0; data w,6,0:
data e,4,0; data w,6,0; data n,3,0; null ,0,0:
rep(2) null,0,0; data s,5,0; null ,0,0:
data e,4,0; data w,6,0; data e,4,0; data w,6,0:
null ,0,0; data e,4,0; data w,6,0; null ,0,0:
data e,4,0; data w,6,0; data e,4,0; data w,6,0 :
data n,3,0; data e,4,0; data w,6,0; null ,0,0:
data s,5,0; data n,3,0; data e,4,0; data w,6,0:
data n,3,0; data s,5,0; rep(2) null ,0,0:
data s,5,0; data n,3,0; rep(2) null ,0,0:
null ,0,0; data s,5,0; rep(2) null ,0,0:
data e,4,0; data w,6,0; rep(2) null ,0,0:
null ,0,0; data e,4,0; data w,6,0; null ,0,0:
data e,4,0; data w,6,0; data e,4,0; data w,6,0:
null ,0,0; data e,4,0; data w,6,0; null ,0,0:
data s,5,0; null ,0,0; data e,4,0; data w,6,0:
rep(2) data s,5,0; rep(2) null ,0,0:
rep(3) data s,5,0; null ,0,0:
rep(4) data s,5,0:
rep(4) data s,5,0:
null ,0,0; rep(3) data s,5,0:
rep(2) null ,0,0; rep(2) data s,5,0:
rep(3) null ,0,0; data s,5,0:
rep(4) null ,0,0:
{matrix multiplication 4x4}
repl(3){rep(4) null n,0,0}:
{calculation}
rep(4) mult n w,3,6:
rep(4) add ,7,0:
rep(4) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,7,0:
rep(4) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,7,0:
rep(4) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,7,0:
rep(4) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
add ,8,0; rep(3) add ,7,0:
mov ,0,8; rep(3) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(2) add ,8,0; rep(2) add ,7,0:

```

```

rep(2) mov ,0,8;rep(2) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(3) add ,8,0;add ,7,0:
rep(3) mov ,0,8;mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,8,0:
rep(4) mov ,0,8:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
add ,9,0;rep(3) add ,8,0:
mov ,0,9;rep(3) mov ,0,8:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(2) add ,9,0;rep(2) add ,8,0:
rep(2) mov ,0,9;rep(2) mov ,0,8:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(3) add ,9,0;add ,8,0:
rep(3) mov ,0,9;mov ,0,8:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,9,0:
rep(4) mov ,0,9:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,9,0:
rep(4) mov ,0,9:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,9,0:
rep(3) copy ,0,0; null ,0,0:
data n,3,0; rep(3) null ,0,0:
mov s,1,7; data n,3,0; rep(2) null ,0,0:
mov ,8,0; mov s,1,7; data n,3,0; null ,0,0:
copy ,0,0; mov ,8,0; mov s,1,7; null ,0,0:
data n,3,0; copy ,0,0; mov ,8,0; null ,0,0:
mov s,1,7; data n,3,0; copy ,0,0; null ,0,0:
mov ,7,0; mov s,1,7; data n,3,0; null ,0,0:
copy ,0,0; mov ,7,0; mov s,1,7; null ,0,0:
data n,3,0; copy ,0,0; mov ,7,0; null ,0,0:
div ,7,3; data n,3,0; copy ,0,0; null ,0,0:
copy ,0,0; null ,0,0; data n,3,0; null ,0,0:
null ,0,0; data w,6,0; rep(2) null ,0,0:
null ,0,0; mult ,3,6; data w,6,0; null ,0,0:
null ,0,0; sub ,7,0 ; mult ,3,6; null ,0,0:
null ,0,0; copy ,0,0; sub ,7,0; null ,0,0:
null ,0,0; mov s,1,7; copy ,0,0; null ,0,0:
null ,0,0; data n,3,0; mov s,1,7; null ,0,0:
null ,0,0; div ,7,3; data n,3,0; null ,0,0:
null ,0,0; copy ,0,0; rep(2) null ,0,0:
rep(2) null ,0,0; data w,6,0; null ,0,0:
rep(2) null ,0,0; mult ,3,6; null ,0,0:
rep(2) null ,0,0; sub ,7,0; null ,0,0:
mov s,1,8; null ,0,0; copy ,0,0; null ,0,0:
rep(4) null ,0,0:
null ,0,0; mov s,1,8; rep(2) null ,0,0:
rep(4) null ,0,0:
mov s,1,7; null ,0,0; mov s,1,8; null ,0,0:
rep(4) null ,0,0:
data n,3,0;mov s,1,7; rep(2) null ,0,0:
rep(4) null ,0,0:
null ,0,0; data n,3,0; rep(2) null ,0,0:
rep(4) null ,0,0:
mov s,1,7; null ,0,0; data n,3,0; null ,0,0:

```

```

rep(4) null ,0,0:
null ,0,0; mov s,1,7; rep(2) null ,0,0:
rep(4) null ,0,0:
data n,3,0; null ,0,0; mov s,1,7; null ,0,0:
{ start factorisation}
mult ,3,7; rep(3) null n,0,0:
copy ,0,0; rep(3) null n,0,0:
null n,0,0; data w,6,0; rep(2) null n,0,0:
null n,0,0; data n,3,0; rep(2) null n,0,0:
null n,0,0; sub ,3,6; rep(2) null n,0,0:
null n,0,0; copy ,0,0; rep(2) null n,0,0:
null n,0,0; data n,3,0; rep(2) null n,0,0:
null n,0,0; mult ,3,7; rep(2) null n,0,0:
null n,0,0; add ,6,0; rep(2) null n,0,0:
null n,0,0; copy ,0,0; rep(2) null n,0,0:
rep(2) null n,0,0; data w,6,0; null n,0,0:
rep(2) null n,0,0; data n,3,0; null n,0,0:
rep(2) null n,0,0; sub ,3,6; null n,0,0:
rep(2) null n,0,0; copy ,0,0; null n,0,0:
rep(2) null n,0,0; div ,0,8; null ,0,0:
rep(2) null n,0,0; copy ,0,0; null n,0,0:
rep(4) null n,0,0:
rep(2) null n,0,0; data s,5,0; null n,0,0:
rep(2) null n,0,0; mult ,5,8; null ,0,0:
rep(2) null n,0,0; copy ,0,0; null ,0,0:
null n,0,0; data e,4,0; data s,5,0; null n,0,0:
null ,0,0; sub ,0,4; rep(2) null ,0,0:
null ,0,0; div ,0,8; rep(2) null ,0,0:
null ,0,0; copy ,0,0; rep(2) null ,0,0:
rep(4) null ,0,0:
null ,0,0; data s,5,0; rep(2) null ,0,0:
null ,0,0; mult ,5,8; data s,5,0; null ,0,0:
null ,0,0; copy ,0,0; mult ,5,8; null ,0,0:
null ,0,0; mov s,1,9; copy ,0,0; null ,0,0:
null ,0,0; data e,4,0; rep(2) null ,0,0:
null ,0,0; add ,4,9; rep(2) null ,0,0:
null ,0,0; copy ,0,0; rep(2) null ,0,0:
data e,4,0; rep(3) null ,0,0:
sub ,0,4; rep(3) null ,0,0:
div ,0,8; rep(3) null ,0,0:
copy ,0,0; rep(3) null ,0,0:
rep(4) null ,0,0:
rep(4) null ,0,0:
rep(2) null ,0,0; data s,5,0; null ,0,0:
null ,0,0; rep(2) data s,5,0; null ,0,0:
null ,0,0; rep(2) data s,5,0; null ,0,0:
rep(3) data s,5,0; null ,0,0:
data n,3,0; rep(3) null ,0,0:
{ start factorisation}
mult ,3,7; rep(3) null n,0,0:
copy ,0,0; rep(3) null n,0,0:
null n,0,0; data w,6,0; rep(2) null n,0,0:
null n,0,0; data n,3,0; rep(2) null n,0,0:
null n,0,0; sub ,3,6; rep(2) null n,0,0:
null n,0,0; copy ,0,0; rep(2) null n,0,0:
null n,0,0; data n,3,0; rep(2) null n,0,0:
null n,0,0; mult ,3,7; rep(2) null n,0,0:
null n,0,0; add ,6,0; rep(2) null n,0,0:
null n,0,0; copy ,0,0; rep(2) null n,0,0:
rep(2) null n,0,0; data w,6,0; null n,0,0:
rep(2) null n,0,0; data n,3,0; null n,0,0:
rep(2) null n,0,0; sub ,3,6; null n,0,0:
rep(2) null n,0,0; copy ,0,0; null n,0,0:
rep(2) null n,0,0; div ,0,8; null ,0,0:
rep(2) null n,0,0; copy ,0,0; null n,0,0:
rep(4) null n,0,0:

```

```

rep(2) null n,0,0; data s,5,0; null n,0,0:
rep(2) null n,0,0; mult ,5,8; null ,0,0:
rep(2) null n,0,0; copy ,0,0; null ,0,0:
null n,0,0; data e,4,0; data s,5,0; null n,0,0:
null ,0,0; sub ,0,4; rep(2) null ,0,0:
null ,0,0; div ,0,8; rep(2) null ,0,0:
null ,0,0; copy ,0,0; rep(2) null ,0,0:
rep(4) null ,0,0:
null ,0,0; data s,5,0; rep(2) null ,0,0:
null ,0,0; mult ,5,8; data s,5,0; null ,0,0:
null ,0,0; copy ,0,0; mult ,5,8; null ,0,0:
null ,0,0; mov s,1,9; copy ,0,0; null ,0,0:
null ,0,0; data e,4,0; rep(2) null ,0,0:
null ,0,0; add ,4,9; rep(2) null ,0,0:
null ,0,0; copy ,0,0; rep(2) null ,0,0:
data e,4,0; rep(3) null ,0,0:
sub ,0,4; rep(3) null ,0,0:
div ,0,8; rep(3) null ,0,0:
copy ,0,0; rep(3) null ,0,0:
rep(4) null ,0,0:
rep(4) null ,0,0:
rep(2) null ,0,0; data s,5,0; null ,0,0:
null ,0,0; rep(2) data s,5,0; null ,0,0:
null ,0,0; rep(2) data s,5,0; null ,0,0:
rep(3) data s,5,0; null ,0,0:
data n,3,0; rep(3) null ,0,0:
{ start factorisation}
mult ,3,7; rep(3) null n,0,0:
copy ,0,0; rep(3) null n,0,0:
null n,0,0; data w,6,0; rep(2) null n,0,0:
null n,0,0; data n,3,0; rep(2) null n,0,0:
null n,0,0; sub ,3,6; rep(2) null n,0,0:
null n,0,0; copy ,0,0; rep(2) null n,0,0:
null n,0,0; data n,3,0; rep(2) null n,0,0:
null n,0,0; mult ,3,7; rep(2) null n,0,0:
null n,0,0; add ,6,0; rep(2) null n,0,0:
null n,0,0; copy ,0,0; rep(2) null n,0,0:
rep(2) null n,0,0; data w,6,0; null n,0,0:
rep(2) null n,0,0; data n,3,0; null n,0,0:
rep(2) null n,0,0; sub ,3,6; null n,0,0:
rep(2) null n,0,0; copy ,0,0; null n,0,0:
rep(2) null n,0,0; div ,0,8; null ,0,0:
rep(2) null n,0,0; copy ,0,0; null n,0,0:
rep(4) null n,0,0:
rep(2) null n,0,0; data s,5,0; null n,0,0:
rep(2) null n,0,0; mult ,5,8; null ,0,0:
rep(2) null n,0,0; copy ,0,0; null ,0,0:
null n,0,0; data e,4,0; data s,5,0; null n,0,0:
null ,0,0; sub ,0,4; rep(2) null ,0,0:
null ,0,0; div ,0,8; rep(2) null ,0,0:
null ,0,0; copy ,0,0; rep(2) null ,0,0:
rep(4) null ,0,0:
null ,0,0; data s,5,0; rep(2) null ,0,0:
null ,0,0; mult ,5,8; data s,5,0; null ,0,0:
null ,0,0; copy ,0,0; mult ,5,8; null ,0,0:
null ,0,0; mov s,1,9; copy ,0,0; null ,0,0:
null ,0,0; data e,4,0; rep(2) null ,0,0:
null ,0,0; add ,4,9; rep(2) null ,0,0:
null ,0,0; copy ,0,0; rep(2) null ,0,0:
data e,4,0; rep(3) null ,0,0:
sub ,0,4; rep(3) null ,0,0:
div ,0,8; rep(3) null ,0,0:
copy ,0,0; rep(3) null ,0,0:
rep(4) null ,0,0:
rep(4) null ,0,0:
rep(2) null ,0,0; data s,5,0; null ,0,0:

```

```

null ,0,0; rep(2) data s,5,0; null ,0,0:
null ,0,0; rep(2) data s,5,0; null ,0,0:
rep(3) data s,5,0; null ,0,0:
data n,3,0; rep(3) null ,0,0:
{ start factorisation}
mult ,3,7; rep(3) null n,0,0:
copy ,0,0; rep(3) null n,0,0:
null n,0,0; data w,6,0; rep(2) null n,0,0:
null n,0,0; data n,3,0; rep(2) null n,0,0:
null n,0,0; sub ,3,6; rep(2) null n,0,0:
null n,0,0; copy ,0,0; rep(2) null n,0,0:
null n,0,0; data n,3,0; rep(2) null n,0,0:
null n,0,0; mult ,3,7; rep(2) null n,0,0:
null n,0,0; add ,6,0; rep(2) null n,0,0:
null n,0,0; copy ,0,0; rep(2) null n,0,0:
rep(2) null n,0,0; data w,6,0; null n,0,0:
rep(2) null n,0,0; data n,3,0; null n,0,0:
rep(2) null n,0,0; sub ,3,6; null n,0,0:
rep(2) null n,0,0; copy ,0,0; null n,0,0:
rep(2) null n,0,0; div ,0,8; null ,0,0:
rep(2) null n,0,0; copy ,0,0; null n,0,0:
rep(4) null n,0,0:
rep(2) null n,0,0; data s,5,0; null n,0,0:
rep(2) null n,0,0; mult ,5,8; null ,0,0:
rep(2) null n,0,0; copy ,0,0; null ,0,0:
null n,0,0; data e,4,0; data s,5,0; null n,0,0:
null ,0,0; sub ,0,4; rep(2) null ,0,0:
null ,0,0; div ,0,8; rep(2) null ,0,0:
null ,0,0; copy ,0,0; rep(2) null ,0,0:
rep(4) null ,0,0:
null ,0,0; data s,5,0; rep(2) null ,0,0:
null ,0,0; mult ,5,8; data s,5,0; null ,0,0:
null ,0,0; copy ,0,0; mult ,5,8; null ,0,0:
null ,0,0; mov s,1,9; copy ,0,0; null ,0,0:
null ,0,0; data e,4,0; rep(2) null ,0,0:
null ,0,0; add ,4,9; rep(2) null ,0,0:
null ,0,0; copy ,0,0; rep(2) null ,0,0:
data e,4,0; rep(3) null ,0,0:
sub ,0,4; rep(3) null ,0,0:
div ,0,8; rep(3) null ,0,0:
copy ,0,0; rep(3) null ,0,0:
rep(4) null ,0,0:
rep(4) null ,0,0:
rep(2) null ,0,0; data s,5,0; null ,0,0:
null ,0,0; rep(2) data s,5,0; null ,0,0:
null ,0,0; rep(2) data s,5,0; null ,0,0:
rep(3) data s,5,0; null ,0,0
end

```

```

d(4,300)
{ NO. 6.4 }
{ Data file for finding the g-inverse of a rec-matrix }
n 4.0,0.0,0.0,0.0;
none;none;none:
n 3.0,1.0,0.0,0.0;
none;none;none:
n 2.0,2.0,1.0,0.0;
none;none;none:
n 1.0,0.0,2.0,0.0;
none;none;none:
n 0.0,4.0,1.0,0.0;
none;none;none:
n 0.0,0.0,3.0,0.0;
none;none;none:
n 0.0,0.0,0.0,0.0;

```

```

none;none;none:
repl(27)[rep(4) none]:
repl(3)[rep(4) none]:
n 1.0,0.0,0.0,0.0;none;none;w 1.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 2.0,4.0,0.0,0.0;none;none;w 2.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 3.0,0.0,3.0,0.0;none;none;w 3.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 4.0,2.0,1.0,0.0;none;none;w 4.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 1.0,1.0,2.0,0.0;none;none;w 4.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 2.0,4.0,1.0,0.0;none;none;w 0.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 3.0,0.0,3.0,0.0;none;none;w 2.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 4.0,2.0,1.0,0.0;none;none;w 1.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 1.0,1.0,2.0,0.0;none;none;w 3.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 2.0,4.0,1.0,0.0;none;none;w 1.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 3.0,0.0,3.0,0.0;none;none;w 2.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 4.0,2.0,1.0,0.0;none;none;w 1.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 0.0,1.0,2.0,0.0;none;none;w 0.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 0.0,0.0,1.0,0.0;none;none;w 0.0,0.0,0.0,0.0:
repl(30)[rep(4) none]:
n 1.0,0.0,0.0,0.0; none;none;none:
none; none; none; none:
n 0.0,1.0,0.0,0.0; none;none;none:
none; none; none; none:
n 0.0,0.0,1.0,0.0; none;none;none:
repl(3)[rep(4) none]:
n 1.0,0.0,0.0,0.0; none;none;none:
repl(3)[rep(4) none]:
n 0.0,4.0,0.0,0.0; none;none;none:
repl(7)[rep(4) none]:
n 0.0,0.0,3.0,0.0; none;none;none:
repl(30)[rep(4) none]:
n 2.0,0.0,0.0,0.0; none;none;none:
repl(3)[rep(4) none]:
n 0.0,0.0,0.0,0.0; none;none;none:
repl(7)[rep(4) none]:
n 0.0,0.0,1.0,0.0; none;none;none:
repl(30)[rep(4) none]:
n 3.0,0.0,0.0,0.0; none;none;none:
repl(3)[rep(4) none]:
n 0.0,2.0,0.0,0.0; none;none;none:
repl(7)[rep(4) none]:
n 0.0,0.0,2.0,0.0; none;none;none:
repl(30)[rep(4) none]:
n 4.0,0.0,0.0,0.0; none;none;none:
repl(3)[rep(4) none]:
n 0.0,1.0,0.0,0.0; none;none;none:
repl(7)[rep(4) none]:
n 0.0,0.0,1.0,0.0; none;none;none:
repl(30)[rep(4) none]
end

```



```

s(4,300)
{ NO. 6.4 }
{ Selector file for finding the g-inverse of a rec-matrix }
1,0,0,0:
1,1,0,0:
1,1,1,0:
1,1,1,1:
repl(3)[rep(4)0]:
repl(2)[1,rep(3)0]:
0,0,1,0:
rep(3) 1, 0:
1, rep(3) 0 :
1,0,1,1:
rep(3) 1, 0:
repl(2)[1,1,0,0]:
0,1,0,0:
1,1,0,0:
0,1,1,0:
rep(4)1:
rep(3) 1,0:
rep(4) 1 :
repl(3)[1,1,0,0]:
repl(2)[rep(4)0]:
rep(3) 1,0:
1,1,0,0:
1,0,0,0:
repl(4)[rep(4) 0]:
repl(57)[1,rep(3)0]:
1,0,0,0:
0,0,0,0:
0,1,0,0:
1,0,1,0:
1,0,1,0:
0,0,0,0:
0,1,0,0:
1,1,0,0:
1,0,0,0:
0,0,0,0:
0,1,0,0:
0,1,1,0:
repl(4)[0, rep(2)1,0]:
repl(7)[rep(2) 0, 1,0]:
1,0,1,0:
1,0,0,0:
repl(2)[1,1,0,0]:
repl(2)[0,1,1,0]:
repl(2)[1,0,1,0]:
1,1,0,0:
0,1,0,0:
1,0,1,0:
0,0,0,0:
0,1,0,0:
0,0,0,0:
1,0,1,0:
1,1,0,0:
repl(2)[1,1,1,0]:
0,1,1,0:
repl(2)[0,1,0,0]:
rep(4)0:
repl(2)[0,0,1,0]:
1,0,1,0:

```

```

0,1,1,0:
repl(5)[0,0,1,0]:
0,1,1,0:
repl(7)[0,1,0,0]:
1,1,0,0:
repl(11)[1,0,0,0]:
0,0,0,0:
0,1,0,0:
repl(5)[1,0,0,0]:
1,1,0,0:
repl(2)[1,1,1,0]:
0,1,1,0:
repl(2)[0,1,0,0]:
rep(4)0:
repl(2)[0,0,1,0]:
1,0,1,0:
0,1,1,0:
repl(5)[0,0,1,0]:
0,1,1,0:
repl(7)[0,1,0,0]:
1,1,0,0:
repl(11)[1,0,0,0]:
0,0,0,0:
0,1,0,0:
repl(5)[1,0,0,0]:
1,1,0,0:
repl(2)[1,1,1,0]:
0,1,1,0:
repl(2)[0,1,0,0]:
rep(4)0:
repl(2)[0,0,1,0]:
1,0,1,0:
0,1,1,0:
repl(5)[0,0,1,0]:
0,1,1,0:
repl(7)[0,1,0,0]:
1,1,0,0:
repl(11)[1,0,0,0]:
0,0,0,0:
0,1,0,0:
repl(5)[1,0,0,0]:
1,1,0,0:
repl(2)[1,1,1,0]:
0,1,1,0:
repl(2)[0,1,0,0]:
rep(4)0:
repl(2)[0,0,1,0]:
1,0,1,0:
0,1,1,0:
repl(5)[0,0,1,0]:
0,1,1,0:
repl(7)[0,1,0,0]:
1,1,0,0:
repl(11)[1,0,0,0]:
0,0,0,0:
0,1,0,0:
repl(4)[1,0,0,0]
end

```

```

p(4,300)
{NO. 6.5.1 part-1 }
{ Program for solution of a homogenous system of eqs. }
data n,3,0; rep(3) null ,0,0:
rep(2) data n,3,0; rep(2) null ,0,0:
rep(3) data n,3,0; null ,0,0:
rep(4) data n,3,0 :
null ,0,0; rep(3) data n,3,0:
rep(2) null,0,0; rep(2) data n,3,0:
rep(3) null ,0,0; data n,3,0:
rep(4) null ,0,0:
data e,4,0; data w,6,0; rep(2) null ,0,0:
data n,3,0; rep(3) null,0,0:
data s,5,0; null,0,0; data e,4,0; data w,6,0:
data e,4,0; data w,6,0; data n,3,0; null ,0,0:
rep(2) null,0,0; data s,5,0; null ,0,0:
data e,4,0; data w,6,0; data e,4,0; data w,6,0:
null ,0,0; data e,4,0; data w,6,0; null ,0,0:
data e,4,0; data w,6,0; data e,4,0; data w,6,0 :
data n,3,0; data e,4,0; data w,6,0; null ,0,0:
data s,5,0; data n,3,0; data e,4,0; data w,6,0:
data n,3,0; data s,5,0; rep(2) null ,0,0:
data s,5,0; data n,3,0; rep(2) null ,0,0:
null ,0,0; data s,5,0; rep(2) null ,0,0:
data e,4,0; data w,6,0; rep(2) null ,0,0:
null ,0,0; data e,4,0; data w,6,0; null ,0,0:
data e,4,0; data w,6,0; data e,4,0; data w,6,0:
null ,0,0; data e,4,0; data w,6,0; null ,0,0:
data s,5,0; null ,0,0; data e,4,0; data w,6,0:
rep(2) data s,5,0; rep(2) null ,0,0:
rep(3) data s,5,0; null ,0,0:
rep(4) data s,5,0:
rep(4) data s,5,0:
null ,0,0; rep(3) data s,5,0:
rep(2) null ,0,0; rep(2) data s,5,0:
rep(3) null ,0,0; data s,5,0:
rep(4) null ,0,0:
{matrix multiplication 4x4}
repl(3){rep(4) null n,0,0}:
{calculation}
rep(4) mult n w,3,6:
rep(4) add ,7,0:
rep(4) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,7,0:
rep(4) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,7,0:
rep(4) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,7,0:
rep(4) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
add ,8,0; rep(3) add ,7,0:
mov ,0,8; rep(3) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(2) add ,8,0; rep(2) add ,7,0:

```

```

rep(2) mov ,0,8;rep(2) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(3) add ,8,0;add ,7,0:
rep(3) mov ,0,8;mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,8,0:
rep(4) mov ,0,8:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
add ,9,0;rep(3) add ,8,0:
mov ,0,9;rep(3) mov ,0,8:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(2) add ,9,0;rep(2) add ,8,0:
rep(2) mov ,0,9;rep(2) mov ,0,8:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(3) add ,9,0;add ,8,0:
rep(3) mov ,0,9;mov ,0,8:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,9,0:
rep(4) mov ,0,9:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,9,0:
rep(4) mov ,0,9:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,9,0:
rep(3) copy ,0,0; null ,0,0:
data n,3,0; rep(3) null ,0,0:
mov s,1,7; data n,3,0; rep(2) null ,0,0:
mov ,8,0; mov s,1,7; data n,3,0; null ,0,0:
copy ,0,0; mov ,8,0; mov s,1,7; null ,0,0:
data n,3,0; copy ,0,0; mov ,8,0; null ,0,0:
mov s,1,7; data n,3,0; copy ,0,0; null ,0,0:
mov ,7,0; mov s,1,7; data n,3,0; null ,0,0:
copy ,0,0; mov ,7,0; mov s,1,7; null ,0,0:
data n,3,0; copy ,0,0; mov ,7,0; null ,0,0:
div ,7,3; data n,3,0; copy ,0,0; null ,0,0:
copy ,0,0; null ,0,0; data n,3,0; null ,0,0:
null ,0,0; data w,6,0; rep(2) null ,0,0:
null ,0,0; mult ,3,6; data w,6,0; null ,0,0:
null ,0,0; sub ,7,0 ; mult ,3,6; null ,0,0:
null ,0,0; copy ,0,0; sub ,7,0; null ,0,0:
null ,0,0; mov s,1,7; copy ,0,0; null ,0,0:
null ,0,0; data n,3,0; mov s,1,7; null ,0,0:
null ,0,0; div ,7,3; data n,3,0; null ,0,0:
null ,0,0; copy ,0,0; rep(2) null ,0,0:
rep(2) null ,0,0; data w,6,0; null ,0,0:
rep(2) null ,0,0; mult ,3,6; null ,0,0:
rep(2) null ,0,0; sub ,7,0; null ,0,0:
mov s,1,8; null ,0,0; copy ,0,0; null ,0,0:
rep(4) null ,0,0:
null ,0,0; mov s,1,8; rep(2) null ,0,0:
rep(4) null ,0,0:
mov s,1,7; null ,0,0; mov s,1,8; null ,0,0:
rep(4) null ,0,0:
data n,3,0;mov s,1,7; rep(2) null ,0,0:
rep(4) null ,0,0:
null ,0,0; data n,3,0; rep(2) null ,0,0:
rep(4) null ,0,0:
mov s,1,7; null ,0,0; data n,3,0; null ,0,0:

```

```

rep(4) null ,0,0:
null ,0,0; mov s,1,7; rep(2) null ,0,0:
rep(4) null ,0,0:
data n,3,0; null ,0,0; mov s,1,7; null ,0,0:
mult ,3,7; rep(3) null n,0,0:
copy ,0,0; rep(3) null n,0,0:
null n,0,0; data w,6,0; rep(2) null n,0,0:
null n,0,0; data n,3,0; rep(2) null n,0,0:
null n,0,0; sub ,3,6; rep(2) null n,0,0:
null n,0,0; copy ,0,0; rep(2) null n,0,0:
null n,0,0; data n,3,0; rep(2) null n,0,0:
null n,0,0; mult ,3,7; rep(2) null n,0,0:
null n,0,0; add ,6,0; rep(2) null n,0,0:
null n,0,0; copy ,0,0; rep(2) null n,0,0:
rep(2) null n,0,0; data w,6,0; null n,0,0:
rep(2) null n,0,0; data n,3,0; null n,0,0:
rep(2) null n,0,0; sub ,3,6; null n,0,0:
rep(2) null n,0,0; copy ,0,0; null n,0,0:
rep(2) null n,0,0; div ,0,8; null ,0,0:
rep(2) null n,0,0; copy ,0,0; null n,0,0:
rep(4) null n,0,0:
rep(2) null n,0,0; data s,5,0; null n,0,0:
rep(2) null n,0,0; mult ,5,8; null ,0,0:
rep(2) null n,0,0; copy ,0,0; null ,0,0:
null n,0,0; data e,4,0; data s,5,0; null n,0,0:
null ,0,0; sub ,0,4; rep(2) null ,0,0:
null ,0,0; div ,0,8; rep(2) null ,0,0:
null ,0,0; copy ,0,0; rep(2) null ,0,0:
rep(4) null ,0,0:
null ,0,0; data s,5,0; rep(2) null ,0,0:
null ,0,0; mult ,5,8; data s,5,0; null ,0,0:
null ,0,0; copy ,0,0; mult ,5,8; null ,0,0:
null ,0,0; mov s,1,9; copy ,0,0; null ,0,0:
null ,0,0; data e,4,0; rep(2) null ,0,0:
null ,0,0; add ,4,9; rep(2) null ,0,0:
null ,0,0; copy ,0,0; rep(2) null ,0,0:
data e,4,0; rep(3) null ,0,0:
sub ,0,4; rep(3) null ,0,0:
div ,0,8; rep(3) null ,0,0:
copy ,0,0; rep(3) null ,0,0:
rep(4) null ,0,0:
rep(4) null ,0,0:
rep(2) null ,0,0; data s,5,0; null ,0,0:
null ,0,0; rep(2) data s,5,0; null ,0,0:
null ,0,0; rep(2) data s,5,0; null ,0,0:
rep(3) data s,5,0; null ,0,0:
data n,3,0; rep(3) null ,0,0:
mult ,3,7; rep(3) null n,0,0:
copy ,0,0; rep(3) null n,0,0:
null n,0,0; data w,6,0; rep(2) null n,0,0:
null n,0,0; data n,3,0; rep(2) null n,0,0:
null n,0,0; sub ,3,6; rep(2) null n,0,0:
null n,0,0; copy ,0,0; rep(2) null n,0,0:
null n,0,0; data n,3,0; rep(2) null n,0,0:
null n,0,0; mult ,3,7; rep(2) null n,0,0:
null n,0,0; add ,6,0; rep(2) null n,0,0:
null n,0,0; copy ,0,0; rep(2) null n,0,0:
rep(2) null n,0,0; data w,6,0; null n,0,0:
rep(2) null n,0,0; data n,3,0; null n,0,0:
rep(2) null n,0,0; sub ,3,6; null n,0,0:
rep(2) null n,0,0; copy ,0,0; null n,0,0:
rep(2) null n,0,0; div ,0,8; null ,0,0:
rep(2) null n,0,0; copy ,0,0; null n,0,0:
rep(4) null n,0,0:
rep(2) null n,0,0; data s,5,0; null n,0,0:
rep(2) null n,0,0; mult ,5,8; null ,0,0:

```

```

rep(2) null n,0,0; copy ,0,0; null ,0,0:
null n,0,0; data e,4,0; data s,5,0; null n,0,0:
null ,0,0; sub ,0,4; rep(2) null ,0,0:
null ,0,0; div ,0,8; rep(2) null ,0,0:
null ,0,0; copy ,0,0; rep(2) null ,0,0:
rep(4) null ,0,0:
null ,0,0; data s,5,0; rep(2) null ,0,0:
null ,0,0; mult ,5,8; data s,5,0; null ,0,0:
null ,0,0; copy ,0,0; mult ,5,8; null ,0,0:
null ,0,0; mov s,1,9; copy ,0,0; null ,0,0:
null ,0,0; data e,4,0; rep(2) null ,0,0:
null ,0,0; add ,4,9; rep(2) null ,0,0:
null ,0,0; copy ,0,0; rep(2) null ,0,0:
data e,4,0; rep(3) null ,0,0:
sub ,0,4; rep(3) null ,0,0:
div ,0,8; rep(3) null ,0,0:
copy ,0,0; rep(3) null ,0,0:
rep(4) null ,0,0:
rep(4) null ,0,0:
rep(2) null ,0,0; data s,5,0; null ,0,0:
null ,0,0; rep(2) data s,5,0; null ,0,0:
null ,0,0; rep(2) data s,5,0; null ,0,0:
rep(3) data s,5,0; null ,0,0:
data n,3,0; rep(3) null ,0,0:
mult ,3,7; rep(3) null n,0,0:
copy ,0,0; rep(3) null n,0,0:
null n,0,0; data w,6,0; rep(2) null n,0,0:
null n,0,0; data n,3,0; rep(2) null n,0,0:
null n,0,0; sub ,3,6; rep(2) null n,0,0:
null n,0,0; copy ,0,0; rep(2) null n,0,0:
null n,0,0; data n,3,0; rep(2) null n,0,0:
null n,0,0; mult ,3,7; rep(2) null n,0,0:
null n,0,0; add ,6,0; rep(2) null n,0,0:
null n,0,0; copy ,0,0; rep(2) null n,0,0:
rep(2) null n,0,0; data w,6,0; null n,0,0:
rep(2) null n,0,0; data n,3,0; null n,0,0:
rep(2) null n,0,0; sub ,3,6; null n,0,0:
rep(2) null n,0,0; copy ,0,0; null n,0,0:
rep(2) null n,0,0; div ,0,8; null ,0,0:
rep(2) null n,0,0; copy ,0,0; null n,0,0:
rep(4) null n,0,0:
rep(2) null n,0,0; data s,5,0; null n,0,0:
rep(2) null n,0,0; mult ,5,8; null ,0,0:
rep(2) null n,0,0; copy ,0,0; null ,0,0:
null n,0,0; data e,4,0; data s,5,0; null n,0,0:
null ,0,0; sub ,0,4; rep(2) null ,0,0:
null ,0,0; div ,0,8; rep(2) null ,0,0:
null ,0,0; copy ,0,0; rep(2) null ,0,0:
rep(4) null ,0,0:
null ,0,0; data s,5,0; rep(2) null ,0,0:
null ,0,0; mult ,5,8; data s,5,0; null ,0,0:
null ,0,0; copy ,0,0; mult ,5,8; null ,0,0:
null ,0,0; mov s,1,9; copy ,0,0; null ,0,0:
null ,0,0; data e,4,0; rep(2) null ,0,0:
null ,0,0; add ,4,9; rep(2) null ,0,0:
null ,0,0; copy ,0,0; rep(2) null ,0,0:
data e,4,0; rep(3) null ,0,0:
sub ,0,4; rep(3) null ,0,0:
div ,0,8; rep(3) null ,0,0:
copy ,0,0; rep(3) null ,0,0:
rep(4) null ,0,0:
rep(4) null ,0,0:
rep(2) null ,0,0; data s,5,0; null ,0,0:
null ,0,0; rep(2) data s,5,0; null ,0,0:
null ,0,0; rep(2) data s,5,0; null ,0,0:
rep(3) data s,5,0; null ,0,0:

```

```

data n,3,0; rep(3) null ,0,0:
mult ,3,7; rep(3) null n,0,0:
copy ,0,0; rep(3) null n,0,0:
null n,0,0; data w,6,0; rep(2) null n,0,0:
null n,0,0; data n,3,0; rep(2) null n,0,0:
null n,0,0; sub ,3,6; rep(2) null n,0,0:
null n,0,0; copy ,0,0; rep(2) null n,0,0:
null n,0,0; data n,3,0; rep(2) null n,0,0:
null n,0,0; mult ,3,7; rep(2) null n,0,0:
null n,0,0; add ,6,0; rep(2) null n,0,0:
null n,0,0; copy ,0,0; rep(2) null n,0,0:
rep(2) null n,0,0; data w,6,0; null n,0,0:
rep(2) null n,0,0; data n,3,0; null n,0,0:
rep(2) null n,0,0; sub ,3,6; null n,0,0:
rep(2) null n,0,0; copy ,0,0; null n,0,0:
rep(2) null n,0,0; div ,0,8; null ,0,0:
rep(2) null n,0,0; copy ,0,0; null n,0,0:
rep(4) null n,0,0:
rep(2) null n,0,0; data s,5,0; null n,0,0:
rep(2) null n,0,0; mult ,5,8; null ,0,0:
rep(2) null n,0,0; copy ,0,0; null ,0,0:
null n,0,0; data e,4,0; data s,5,0; null n,0,0:
null ,0,0; sub ,0,4; rep(2) null ,0,0:
null ,0,0; div ,0,8; rep(2) null ,0,0:
null ,0,0; copy ,0,0; rep(2) null ,0,0:
rep(4) null ,0,0:
null ,0,0; data s,5,0; rep(2) null ,0,0:
null ,0,0; mult ,5,8; data s,5,0; null ,0,0:
null ,0,0; copy ,0,0; mult ,5,8; null ,0,0:
null ,0,0; mov s,1,9; copy ,0,0; null ,0,0:
null ,0,0; data e,4,0; rep(2) null ,0,0:
null ,0,0; add ,4,9; rep(2) null ,0,0:
null ,0,0; copy ,0,0; rep(2) null ,0,0:
data e,4,0; rep(3) null ,0,0:
sub ,0,4; rep(3) null ,0,0:
div ,0,8; rep(3) null ,0,0:
copy ,0,0; rep(3) null ,0,0:
rep(4) null ,0,0:
rep(4) null ,0,0:
rep(2) null ,0,0; data s,5,0; null ,0,0:
null ,0,0; rep(2) data s,5,0; null ,0,0:
null ,0,0; rep(2) data s,5,0; null ,0,0:
rep(3) data s,5,0; null ,0,0
end

```

```

d(4,300)
{ NO. 6.5.1 part-1 }
{ Data file for solution of a homogenous system of eqs. }
n 4.0,0.0,0.0,0.0;
none;none;none:
n 3.0,1.0,0.0,0.0;
none;none;none:
n 2.0,2.0,1.0,0.0;
none;none;none:
n 1.0,0.0,2.0,0.0;
none;none;none:
n 0.0,4.0,1.0,0.0;
none;none;none:
n 0.0,0.0,3.0,0.0;
none;none;none:
n 0.0,0.0,0.0,0.0;
none;none;none:
repl(27)[rep(4) none]:
repl(3)[rep(4) none]:
n 1.0,0.0,0.0,0.0;none;none;w 1.0,0.0,0.0,0.0:

```

```

repl(3)[rep(4) none]:
n 2.0,4.0,0.0,0.0;none;none;w 2.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 3.0,0.0,3.0,0.0;none;none;w 3.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 4.0,2.0,1.0,0.0;none;none;w 4.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 1.0,1.0,2.0,0.0;none;none;w 4.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 2.0,4.0,1.0,0.0;none;none;w 0.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 3.0,0.0,3.0,0.0;none;none;w 2.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 4.0,2.0,1.0,0.0;none;none;w 1.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 1.0,1.0,2.0,0.0;none;none;w 3.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 2.0,4.0,1.0,0.0;none;none;w 1.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 3.0,0.0,3.0,0.0;none;none;w 2.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 4.0,2.0,1.0,0.0;none;none;w 1.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 0.0,1.0,2.0,0.0;none;none;w 0.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 0.0,0.0,1.0,0.0;none;none;w 0.0,0.0,0.0,0.0:
repl(30)[rep(4) none]:
n 1.0,0.0,0.0,0.0; none;none;none:
none; none; none; none:
n 0.0,1.0,0.0,0.0; none;none;none:
none; none; none; none:
n 0.0,0.0,1.0,0.0; none;none;none:
repl(3)[rep(4) none]:
n 1.0,0.0,0.0,0.0; none;none;none:
repl(3)[rep(4) none]:
n 0.0,4.0,0.0,0.0; none;none;none:
repl(7)[rep(4) none]:
n 0.0,0.0,3.0,0.0; none;none;none:
repl(30)[rep(4) none]:
n 2.0,0.0,0.0,0.0; none;none;none:
repl(3)[rep(4) none]:
n 0.0,0.0,0.0,0.0; none;none;none:
repl(7)[rep(4) none]:
n 0.0,0.0,1.0,0.0; none;none;none:
repl(30)[rep(4) none]:
n 3.0,0.0,0.0,0.0; none;none;none:
repl(3)[rep(4) none]:
n 0.0,2.0,0.0,0.0; none;none;none:
repl(7)[rep(4) none]:
n 0.0,0.0,2.0,0.0; none;none;none:
repl(30)[rep(4) none]:
n 4.0,0.0,0.0,0.0; none;none;none:
repl(3)[rep(4) none]:
n 0.0,1.0,0.0,0.0; none;none;none:
repl(7)[rep(4) none]:
n 0.0,0.0,1.0,0.0; none;none;none:
repl(30)[rep(4) none]
end

```

```

s(4,300)
{ NO. 6.5.1 part-1 }
{ Selector file for solution of a homogenous system of eqs.}
1,0,0,0:
1,1,0,0:
1,1,1,0:

```



```

1,1,1,1:
repl(3)[rep(4)0]:
repl(2)[1,rep(3)0]:
0,0,1,0:
rep(3) 1, 0:
1, rep(3) 0 :
1,0,1,1:
rep(3) 1, 0:
repl(2)[1,1,0,0]:
0,1,0,0:
1,1,0,0:
0,1,1,0:
rep(4)1:
rep(3) 1,0:
rep(4) 1 :
repl(3)[1,1,0,0]:
repl(2)[rep(4)0]:
rep(3) 1,0:
1,1,0,0:
1,0,0,0:
repl(4)[rep(4) 0]:
repl(57)[1,rep(3)0]:
1,0,0,0:
0,0,0,0:
0,1,0,0:
1,0,1,0:
1,0,1,0:
0,0,0,0:
0,1,0,0:
1,1,0,0:
1,0,0,0:
0,0,0,0:
0,1,0,0:
0,1,1,0:
repl(4)[0, rep(2)1,0]:
repl(7)[rep(2) 0, 1,0]:
1,0,1,0:
1,0,0,0:
repl(2)[1,1,0,0]:
repl(2)[0,1,1,0]:
repl(2)[1,0,1,0]:
1,1,0,0:
0,1,0,0:
1,0,1,0:
0,0,0,0:
0,1,0,0:
0,0,0,0:
1,0,1,0:
1,1,0,0:
repl(2)[1,1,1,0]:
0,1,1,0:
repl(2)[0,1,0,0]:
rep(4)0:
repl(2)[0,0,1,0]:
1,0,1,0:
0,1,1,0:
repl(5)[0,0,1,0]:
0,1,1,0:
repl(7)[0,1,0,0]:
1,1,0,0:
repl(11)[1,0,0,0]:
0,0,0,0:
0,1,0,0:
repl(5)[1,0,0,0]:
1,1,0,0:
repl(2)[1,1,1,0]:

```

```

0,1,1,0:
repl(2)[0,1,0,0]:
rep(4)0:
repl(2)[0,0,1,0]:
1,0,1,0:
0,1,1,0:
repl(5)[0,0,1,0]:
0,1,1,0:
repl(7)[0,1,0,0]:
1,1,0,0:
repl(11)[1,0,0,0]:
0,0,0,0:
0,1,0,0:
repl(5)[1,0,0,0]:
1,1,0,0:
repl(2)[1,1,1,0]:
0,1,1,0:
repl(2)[0,1,0,0]:
rep(4)0:
repl(2)[0,0,1,0]:
1,0,1,0:
0,1,1,0:
repl(5)[0,0,1,0]:
0,1,1,0:
repl(7)[0,1,0,0]:
1,1,0,0:
repl(11)[1,0,0,0]:
0,0,0,0:
0,1,0,0:
repl(5)[1,0,0,0]:
1,1,0,0:
repl(2)[1,1,1,0]:
0,1,1,0:
repl(2)[0,1,0,0]:
rep(4)0:
repl(2)[0,0,1,0]:
1,0,1,0:
0,1,1,0:
repl(5)[0,0,1,0]:
0,1,1,0:
repl(7)[0,1,0,0]:
1,1,0,0:
repl(11)[1,0,0,0]:
0,0,0,0:
0,1,0,0:
repl(4)[1,0,0,0]
end

```

```

p(4,91)
{ N0. 6.5.1 part-2 }
{ Program for solution of a homogenous system of eqs. }
repl(3)[rep(4) null n,0,0]:
{calculation}
rep(4) mult n w,3,6:
rep(4) add ,7,0:
rep(4) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,7,0:
rep(4) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,7,0:
rep(4) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,7,0:
rep(4) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
add ,8,0;rep(3) add ,7,0:
mov ,0,8;rep(3) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(2) add ,8,0;rep(2) add ,7,0:
rep(2) mov ,0,8;rep(2) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(3) add ,8,0;add ,7,0:
rep(3) mov ,0,8;mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,8,0:
rep(4) mov ,0,8:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
add ,9,0;rep(3) add ,8,0:
mov ,0,9;rep(3) mov ,0,8:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(2) add ,9,0;rep(2) add ,8,0:
rep(2) mov ,0,9;rep(2) mov ,0,8:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(3) add ,9,0;add ,8,0:
rep(3) mov ,0,9;mov ,0,8:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,9,0:
rep(4) mov ,0,9:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,9,0:
rep(4) mov ,0,9:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,9,0:
rep(4) mov ,0,9:
rep(4) mov ,6,1:
rep(4) mov ,7,0:
rep(4) copy ,0,0:
rep(4) mov ,8,0:

```

```

rep(4) copy ,0,0:
rep(4) mov ,9,0:
rep(4) copy ,0,0:
rep(4) null ,0,0:
data n,3,0; rep(3) null ,0,0:
rep(2) data n,3,0; rep(2) null ,0,0:
rep(3) data n,3,0; null ,0,0:
mov s,1,7; rep(2) data n,3,0; null ,0,0:
data n,3,0; mov s,1,7; data n,3,0; null ,0,0:
rep(2) data n,3,0; mov s,1,7; null ,0,0:
rep(3) data n,3,0; null ,0,0:
mov s,1,8; rep(2) data n,3,0; null ,0,0:
sub ,7,8; mov s,1,8; data n,3,0; null ,0,0:
mov ,0,9; sub ,7,8; mov s,1,8; null ,0,0:
data n,3,0; mov ,0,9; sub ,7,8; null ,0,0:
mult ,3,9; data n,3,0; mov ,0,9; null ,0,0:
copy ,0,0; mult ,3,9; data n,3,0; null ,0,0:
null ,0,0; mov ,0,10; mult ,3,9; null ,0,0:
null ,0,0; data w,6,0; mov ,0,10; null ,0,0:
null ,0,0; add ,6,10; rep(2) null ,0,0:
null ,0,0; copy ,0,0; rep(2) null ,0,0:
rep(2) null ,0,0; data w,6,0; null ,0,0:
rep(2) null ,0,0; add ,6,10; null ,0,0:
rep(2) null ,0,0; copy ,0,0; null ,0,0:
rep(2) null ,0,0; data s,5,0; null ,0,0:
rep(2) null ,0,0; data s,5,0; null ,0,0:
rep(2) null ,0,0; data s,5,0; null ,0,0:
rep(2) null ,0,0; data s,5,0; null ,0,0:
rep(2) null ,0,0; data s,5,0; null ,0,0:
end

```

```

d(4,91)
{ N0. 6.5.1 part-2 }
{ Data file for solution of a homogenous system of eqs. }
repl(3)[rep(4) none]:
n 1.0,0.0,0.0,0.0;none;none;w -0.098039,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 2.0,4.0,0.0,0.0;none;none;w -0.098039,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 3.0,0.0,3.0,0.0;none;none;w 0.054902,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 4.0,2.0,1.0,0.0;none;none;w 0.282352,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 1.0,1.0,2.0,0.0;none;none;w 0.176471,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 2.0,4.0,1.0,0.0;none;none;w -0.823529,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 3.0,0.0,3.0,0.0;none;none;w -0.058824,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 4.0,2.0,1.0,0.0;none;none;w 0.411764,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 1.0,1.0,2.0,0.0;none;none;w 0.098039,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 2.0,4.0,1.0,0.0;none;none;w 1.098039,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 3.0,0.0,3.0,0.0;none;none;w 0.145098,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 4.0,2.0,1.0,0.0;none;none;w -0.682353,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 0.0,1.0,2.0,0.0;none;none;w 0.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 0.0,0.0,1.0,0.0;none;none;w 0.0,0.0,0.0,0.0:
repl(10)[rep(4)none]:
n 0.0, 0.0, 0.0, 0.0;
none;none;none:

```

```

n 0.0, 0.0, 0.0, 0.0;
none;none;none:
n 1.0, 1.0, 1.0, 0.0;
none;none;none:
n 0.0, 0.0, 0.0, 0.0;
none;none;none:
n 0.000011, 0.0, 0.0, 0.0;
none;none;none:
n 0.000027, 0.000002, 0.0, 0.0;
none;none;none:
n 1.000024, 1.000008, 1.000002, 0.0;
none;none;none:
n 0.0, 0.000036, 0.000017, 0.0;
none;none;none:
n 0.0, 0.0, 0.000036, 0.0;
none;none;none:
none;none;none;none:
n 1.0, 0.0, 0.0, 0.0;
none;none;none:
n 0.0, 0.0, 0.0, 0.0;
none;none;none:
n 0.0, 0.0, 1.0, 0.0;
none;none;none:
repl(12)[rep(4) none]
end

```

```

s(4,91)
{ NO. 6.5.1 part-2 }
{ Selector file for solution of a homogenous system of eqs. }
repl(66)[1,rep(3)0]:
1,rep(3)0:
1,1,0,0:
1,1,1,0:
1,rep(3)0:
1,1,0,0:
rep(3)1,0:
rep(3)1,0:
1,rep(3)0:
1,1,0,0:
repl(9)[rep(3)1,0]:
0,1,1,0:
0,0,1,0:
1,1,1,0:
1,1,0,0:
repl(2)[1,0,0,0]:
0,0,0,0
end

```

```

p(4,91)
{ NO. 6.5.1 part-2 }
{ Program for solution of a homogenous system of eqs. }
repl(3)[rep(4) null n,0,0]:
{calculation}
rep(4) mult n w,3,6:
rep(4) add ,7,0:
rep(4) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,7,0:
rep(4) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,7,0:
rep(4) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,7,0:
rep(4) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
add ,8,0;rep(3) add ,7,0:
mov ,0,8;rep(3) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(2) add ,8,0;rep(2) add ,7,0:
rep(2) mov ,0,8;rep(2) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(3) add ,8,0;add ,7,0:
rep(3) mov ,0,8;mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,8,0:
rep(4) mov ,0,8:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
add ,9,0;rep(3) add ,8,0:
mov ,0,9;rep(3) mov ,0,8:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(2) add ,9,0;rep(2) add ,8,0:
rep(2) mov ,0,9;rep(2) mov ,0,8:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(3) add ,9,0;add ,8,0:
rep(3) mov ,0,9;mov ,0,8:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,9,0:
rep(4) mov ,0,9:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,9,0:
rep(4) mov ,0,9:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,9,0:
rep(4) mov ,0,9:
rep(4) mov ,6,1:
rep(4) mov ,7,0:

```

```

rep(4) copy ,0,0:
rep(4) mov ,8,0:
rep(4) copy ,0,0:
rep(4) mov ,9,0:
rep(4) copy ,0,0:
rep(4) null ,0,0:
data n,3,0; rep(3) null ,0,0:
rep(2) data n,3,0; rep(2) null ,0,0:
rep(3) data n,3,0; null ,0,0:
mov s,1,7; rep(2) data n,3,0; null ,0,0:
data n,3,0; mov s,1,7; data n,3,0; null ,0,0:
rep(2) data n,3,0; mov s,1,7; null ,0,0:
rep(3) data n,3,0; null ,0,0:
mov s,1,8; rep(2) data n,3,0; null ,0,0:
sub ,7,8; mov s,1,8; data n,3,0; null ,0,0:
mov ,0,9; sub ,7,8; mov s,1,8; null ,0,0:
data n,3,0; mov ,0,9; sub ,7,8; null ,0,0:
mult ,3,9; data n,3,0; mov ,0,9; null ,0,0:
copy ,0,0; mult ,3,9; data n,3,0; null ,0,0:
null ,0,0; mov ,0,10; mult ,3,9; null ,0,0:
null ,0,0; data w,6,0; mov ,0,10; null ,0,0:
null ,0,0; add ,6,10; rep(2) null ,0,0:
null ,0,0; copy ,0,0; rep(2) null ,0,0:
rep(2) null ,0,0; data w,6,0; null ,0,0:
rep(2) null ,0,0; add ,6,10; null ,0,0:
rep(2) null ,0,0; copy ,0,0; null ,0,0:
rep(2) null ,0,0; data s,5,0; null ,0,0:
rep(2) null ,0,0; data s,5,0; null ,0,0:
rep(2) null ,0,0; data s,5,0; null ,0,0:
rep(2) null ,0,0; data s,5,0; null ,0,0:
rep(2) null ,0,0; data s,5,0; null ,0,0:
end

```

```

d(4,91)
{ N0. 6.5.1 part-2 }
{ Data file for solution of a homogenous system of eqs. }
repl(3)[rep(4) none]:
n 1.0,0.0,0.0,0.0;none;none;w -0.098039,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 2.0,4.0,0.0,0.0;none;none;w -0.098039,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 3.0,0.0,3.0,0.0;none;none;w 0.054902,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 4.0,2.0,1.0,0.0;none;none;w 0.282352,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 1.0,1.0,2.0,0.0;none;none;w 0.176471,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 2.0,4.0,1.0,0.0;none;none;w -0.823529,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 3.0,0.0,3.0,0.0;none;none;w -0.058824,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 4.0,2.0,1.0,0.0;none;none;w 0.411764,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 1.0,1.0,2.0,0.0;none;none;w 0.098039,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 2.0,4.0,1.0,0.0;none;none;w 1.098039,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 3.0,0.0,3.0,0.0;none;none;w 0.145098,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 4.0,2.0,1.0,0.0;none;none;w -0.682353,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 0.0,1.0,2.0,0.0;none;none;w 0.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 0.0,0.0,1.0,0.0;none;none;w 0.0,0.0,0.0,0.0:
repl(10)[rep(4)none]:

```

```

n 0.0, 0.0, 0.0, 0.0;
none;none;none:
n 0.0, 0.0, 0.0, 0.0;
none;none;none:
n 1.0, 1.0, 1.0, 0.0;
none;none;none:
n 0.0, 0.0, 0.0, 0.0;
none;none;none:
n 0.000011, 0.0, 0.0, 0.0;
none;none;none:
n 0.000027, 0.000002, 0.0, 0.0;
none;none;none:
n 1.000024, 1.000008, 1.000002, 0.0;
none;none;none:
n 0.0, 0.000036, 0.000017, 0.0;
none;none;none:
n 0.0, 0.0, 0.000036, 0.0;
none;none;none:
none;none;none;none:
n 1.0, 0.0, 0.0, 0.0;
none;none;none:
n 0.0, 0.0, 0.0, 0.0;
none;none;none:
n 0.0, 0.0, 1.0, 0.0;
none;none;none:
repl(12)[rep(4) none]
end

```

```

s(4,91)
{ NO. 6.5.1 part-2 }
{ Selector file for solution of a homogenous system of eqs. }
repl(66)[1,rep(3)0]:
1,rep(3)0:
1,1,0,0:
1,1,1,0:
1,rep(3)0:
1,1,0,0:
rep(3)1,0:
rep(3)1,0:
1,rep(3)0:
1,1,0,0:
repl(9)[rep(3)1,0]:
0,1,1,0:
0,0,1,0:
1,1,1,0:
1,1,0,0:
repl(2)[1,0,0,0]:
0,0,0,0
end

```



```

p(4,300)
{ NO. 6.5.2 part-1 }
{ Program for the most general solution of a system of eqs. }
data n,3,0; rep(3) null ,0,0:
rep(2) data n,3,0; rep(2) null ,0,0:
rep(3) data n,3,0; null ,0,0:
rep(4) data n,3,0 :
null ,0,0; rep(3) data n,3,0:
rep(2) null,0,0; rep(2) data n,3,0:
rep(3) null ,0,0; data n,3,0:
rep(4) null ,0,0:
data e,4,0; data w,6,0; rep(2) null ,0,0:
data n,3,0; rep(3) null,0,0:
data s,5,0; null,0,0; data e,4,0; data w,6,0:
data e,4,0; data w,6,0; data n,3,0; null ,0,0:
rep(2) null,0,0; data s,5,0; null ,0,0:
data e,4,0; data w,6,0; data e,4,0; data w,6,0:
null ,0,0; data e,4,0; data w,6,0; null ,0,0:
data e,4,0; data w,6,0; data e,4,0; data w,6,0 :
data n,3,0; data e,4,0; data w,6,0; null ,0,0:
data s,5,0; data n,3,0; data e,4,0; data w,6,0:
data n,3,0; data s,5,0; rep(2) null ,0,0:
data s,5,0; data n,3,0; rep(2) null ,0,0:
null ,0,0; data s,5,0; rep(2) null ,0,0:
data e,4,0; data w,6,0; rep(2) null ,0,0:
null ,0,0; data e,4,0; data w,6,0; null ,0,0:
data e,4,0; data w,6,0; data e,4,0; data w,6,0:
null ,0,0; data e,4,0; data w,6,0; null ,0,0:
data s,5,0; null ,0,0; data e,4,0; data w,6,0:
rep(2) data s,5,0; rep(2) null ,0,0:
rep(3) data s,5,0; null ,0,0:
rep(4) data s,5,0:
rep(4) data s,5,0:
null ,0,0; rep(3) data s,5,0:
rep(2) null ,0,0; rep(2) data s,5,0:
rep(3) null ,0,0; data s,5,0:
rep(4) null ,0,0:
{matrix multiplication 4x4}
repl(3)[rep(4) null n,0,0]:
{calculation}
rep(4) mult n w,3,6:
rep(4) add ,7,0:
rep(4) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,7,0:
rep(4) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,7,0:
rep(4) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
add ,8,0; rep(3) add ,7,0:
mov ,0,8; rep(3) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(2) add ,8,0; rep(2) add ,7,0:

```

```

rep(2) mov ,0,8;rep(2) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(3) add ,8,0;add ,7,0:
rep(3) mov ,0,8;mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,8,0:
rep(4) mov ,0,8:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
add ,9,0;rep(3) add ,8,0:
mov ,0,9;rep(3) mov ,0,8:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(2) add ,9,0;rep(2) add ,8,0:
rep(2) mov ,0,9;rep(2) mov ,0,8:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(3) add ,9,0;add ,8,0:
rep(3) mov ,0,9;mov ,0,8:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,9,0:
rep(4) mov ,0,9:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,9,0:
rep(4) mov ,0,9:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,9,0:
rep(3) copy ,0,0; null ,0,0:
data n,3,0; rep(3) null ,0,0:
mov s,1,7; data n,3,0; rep(2) null ,0,0:
mov ,8,0; mov s,1,7; data n,3,0; null ,0,0:
copy ,0,0; mov ,8,0; mov s,1,7; null ,0,0:
data n,3,0; copy ,0,0; mov ,8,0; null ,0,0:
mov s,1,7; data n,3,0; copy ,0,0; null ,0,0:
mov ,7,0; mov s,1,7; data n,3,0; null ,0,0:
copy ,0,0; mov ,7,0; mov s,1,7; null ,0,0:
data n,3,0; copy ,0,0; mov ,7,0; null ,0,0:
div ,7,3; data n,3,0; copy ,0,0; null ,0,0:
copy ,0,0; null ,0,0; data n,3,0; null ,0,0:
null ,0,0; data w,6,0; rep(2) null ,0,0:
null ,0,0; mult ,3,6; data w,6,0; null ,0,0:
null ,0,0; sub ,7,0 ; mult ,3,6; null ,0,0:
null ,0,0; copy ,0,0; sub ,7,0; null ,0,0:
null ,0,0; mov s,1,7; copy ,0,0; null ,0,0:
null ,0,0; data n,3,0; mov s,1,7; null ,0,0:
null ,0,0; div ,7,3; data n,3,0; null ,0,0:
null ,0,0; copy ,0,0; rep(2) null ,0,0:
rep(2) null ,0,0; data w,6,0; null ,0,0:
rep(2) null ,0,0; mult ,3,6; null ,0,0:
rep(2) null ,0,0; sub ,7,0; null ,0,0:
mov s,1,8; null ,0,0; copy ,0,0; null ,0,0:
rep(4) null ,0,0:
null ,0,0; mov s,1,8; rep(2) null ,0,0:
rep(4) null ,0,0:
mov s,1,7; null ,0,0; mov s,1,8; null ,0,0:
rep(4) null ,0,0:
data n,3,0;mov s,1,7; rep(2) null ,0,0:
rep(4) null ,0,0:
null ,0,0; data n,3,0; rep(2) null ,0,0:
rep(4) null ,0,0:
mov s,1,7; null ,0,0; data n,3,0; null ,0,0:

```

```

rep(4) null ,0,0:
null ,0,0; mov s,1,7; rep(2) null ,0,0:
rep(4) null ,0,0:
data n,3,0; null ,0,0; mov s,1,7; null ,0,0:
mult ,3,7; rep(3) null n,0,0:
copy ,0,0; rep(3) null n,0,0:
null n,0,0; data w,6,0; rep(2) null n,0,0:
null n,0,0; data n,3,0; rep(2) null n,0,0:
null n,0,0; sub ,3,6; rep(2) null n,0,0:
null n,0,0; copy ,0,0; rep(2) null n,0,0:
null n,0,0; data n,3,0; rep(2) null n,0,0:
null n,0,0; mult ,3,7; rep(2) null n,0,0:
null n,0,0; add ,6,0; rep(2) null n,0,0:
null n,0,0; copy ,0,0; rep(2) null n,0,0:
rep(2) null n,0,0; data w,6,0; null n,0,0:
rep(2) null n,0,0; data n,3,0; null n,0,0:
rep(2) null n,0,0; sub ,3,6; null n,0,0:
rep(2) null n,0,0; copy ,0,0; null n,0,0:
rep(2) null n,0,0; div ,0,8; null ,0,0:
rep(2) null n,0,0; copy ,0,0; null n,0,0:
rep(4) null n,0,0:
rep(2) null n,0,0; data s,5,0; null n,0,0:
rep(2) null n,0,0; mult ,5,8; null ,0,0:
rep(2) null n,0,0; copy ,0,0; null ,0,0:
null n,0,0; data e,4,0; data s,5,0; null n,0,0:
null ,0,0; sub ,0,4; rep(2) null ,0,0:
null ,0,0; div ,0,8; rep(2) null ,0,0:
null ,0,0; copy ,0,0; rep(2) null ,0,0:
rep(4) null ,0,0:
null ,0,0; data s,5,0; rep(2) null ,0,0:
null ,0,0; mult ,5,8; data s,5,0; null ,0,0:
null ,0,0; copy ,0,0; mult ,5,8; null ,0,0:
null ,0,0; mov s,1,9; copy ,0,0; null ,0,0:
null ,0,0; data e,4,0; rep(2) null ,0,0:
null ,0,0; add ,4,9; rep(2) null ,0,0:
null ,0,0; copy ,0,0; rep(2) null ,0,0:
data e,4,0; rep(3) null ,0,0:
sub ,0,4; rep(3) null ,0,0:
div ,0,8; rep(3) null ,0,0:
copy ,0,0; rep(3) null ,0,0:
rep(4) null ,0,0:
rep(4) null ,0,0:
rep(2) null ,0,0; data s,5,0; null ,0,0:
null ,0,0; rep(2) data s,5,0; null ,0,0:
null ,0,0; rep(2) data s,5,0; null ,0,0:
rep(3) data s,5,0; null ,0,0:
data n,3,0; rep(3) null ,0,0:
{ start factorisation}
mult ,3,7; rep(3) null n,0,0:
copy ,0,0; rep(3) null n,0,0:
null n,0,0; data w,6,0; rep(2) null n,0,0:
null n,0,0; data n,3,0; rep(2) null n,0,0:
null n,0,0; sub ,3,6; rep(2) null n,0,0:
null n,0,0; copy ,0,0; rep(2) null n,0,0:
null n,0,0; data n,3,0; rep(2) null n,0,0:
null n,0,0; mult ,3,7; rep(2) null n,0,0:
null n,0,0; add ,6,0; rep(2) null n,0,0:
null n,0,0; copy ,0,0; rep(2) null n,0,0:
rep(2) null n,0,0; data w,6,0; null n,0,0:
rep(2) null n,0,0; data n,3,0; null n,0,0:
rep(2) null n,0,0; sub ,3,6; null n,0,0:
rep(2) null n,0,0; copy ,0,0; null n,0,0:
rep(2) null n,0,0; div ,0,8; null ,0,0:
rep(2) null n,0,0; copy ,0,0; null n,0,0:
rep(4) null n,0,0:
rep(2) null n,0,0; data s,5,0; null n,0,0:

```

```

rep(2) null n,0,0; mult ,5,8; null ,0,0:
rep(2) null n,0,0; copy ,0,0; null ,0,0:
null n,0,0; data e,4,0; data s,5,0; null n,0,0:
null ,0,0; sub ,0,4; rep(2) null ,0,0:
null ,0,0; div ,0,8; rep(2) null ,0,0:
null ,0,0; copy ,0,0; rep(2) null ,0,0:
rep(4) null ,0,0:
null ,0,0; data s,5,0; rep(2) null ,0,0:
null ,0,0; mult ,5,8; data s,5,0; null ,0,0:
null ,0,0; copy ,0,0; mult ,5,8; null ,0,0:
null ,0,0; mov s,1,9; copy ,0,0; null ,0,0:
null ,0,0; data e,4,0; rep(2) null ,0,0:
null ,0,0; add ,4,9; rep(2) null ,0,0:
null ,0,0; copy ,0,0; rep(2) null ,0,0:
data e,4,0; rep(3) null ,0,0:
sub ,0,4; rep(3) null ,0,0:
div ,0,8; rep(3) null ,0,0:
copy ,0,0; rep(3) null ,0,0:
rep(4) null ,0,0:
rep(4) null ,0,0:
rep(2) null ,0,0; data s,5,0; null ,0,0:
null ,0,0; rep(2) data s,5,0; null ,0,0:
null ,0,0; rep(2) data s,5,0; null ,0,0:
rep(3) data s,5,0; null ,0,0:
data n,3,0; rep(3) null ,0,0:
mult ,3,7; rep(3) null n,0,0:
copy ,0,0; rep(3) null n,0,0:
null n,0,0; data w,6,0; rep(2) null n,0,0:
null n,0,0; data n,3,0; rep(2) null n,0,0:
null n,0,0; sub ,3,6; rep(2) null n,0,0:
null n,0,0; copy ,0,0; rep(2) null n,0,0:
null n,0,0; data n,3,0; rep(2) null n,0,0:
null n,0,0; mult ,3,7; rep(2) null n,0,0:
null n,0,0; add ,6,0; rep(2) null n,0,0:
null n,0,0; copy ,0,0; rep(2) null n,0,0:
rep(2) null n,0,0; data w,6,0; null n,0,0:
rep(2) null n,0,0; data n,3,0; null n,0,0:
rep(2) null n,0,0; sub ,3,6; null n,0,0:
rep(2) null n,0,0; copy ,0,0; null n,0,0:
rep(2) null n,0,0; div ,0,8; null ,0,0:
rep(2) null n,0,0; copy ,0,0; null n,0,0:
rep(4) null n,0,0:
rep(2) null n,0,0; data s,5,0; null n,0,0:
rep(2) null n,0,0; mult ,5,8; null ,0,0:
rep(2) null n,0,0; copy ,0,0; null ,0,0:
null n,0,0; data e,4,0; data s,5,0; null n,0,0:
null ,0,0; sub ,0,4; rep(2) null ,0,0:
null ,0,0; div ,0,8; rep(2) null ,0,0:
null ,0,0; copy ,0,0; rep(2) null ,0,0:
rep(4) null ,0,0:
null ,0,0; data s,5,0; rep(2) null ,0,0:
null ,0,0; mult ,5,8; data s,5,0; null ,0,0:
null ,0,0; copy ,0,0; mult ,5,8; null ,0,0:
null ,0,0; mov s,1,9; copy ,0,0; null ,0,0:
null ,0,0; data e,4,0; rep(2) null ,0,0:
null ,0,0; add ,4,9; rep(2) null ,0,0:
null ,0,0; copy ,0,0; rep(2) null ,0,0:
data e,4,0; rep(3) null ,0,0:
sub ,0,4; rep(3) null ,0,0:
div ,0,8; rep(3) null ,0,0:
copy ,0,0; rep(3) null ,0,0:
rep(4) null ,0,0:
rep(4) null ,0,0:
rep(2) null ,0,0; data s,5,0; null ,0,0:
null ,0,0; rep(2) data s,5,0; null ,0,0:
null ,0,0; rep(2) data s,5,0; null ,0,0:

```

```

rep(3) data s,5,0; null ,0,0:
data n,3,0; rep(3) null ,0,0:
mult ,3,7; rep(3) null n,0,0:
copy ,0,0; rep(3) null n,0,0:
null n,0,0; data w,6,0; rep(2) null n,0,0:
null n,0,0; data n,3,0; rep(2) null n,0,0:
null n,0,0; sub ,3,6; rep(2) null n,0,0:
null n,0,0; copy ,0,0; rep(2) null n,0,0:
null n,0,0; data n,3,0; rep(2) null n,0,0:
null n,0,0; mult ,3,7; rep(2) null n,0,0:
null n,0,0; add ,6,0; rep(2) null n,0,0:
null n,0,0; copy ,0,0; rep(2) null n,0,0:
rep(2) null n,0,0; data w,6,0; null n,0,0:
rep(2) null n,0,0; data n,3,0; null n,0,0:
rep(2) null n,0,0; sub ,3,6; null n,0,0:
rep(2) null n,0,0; copy ,0,0; null n,0,0:
rep(2) null n,0,0; div ,0,8; null ,0,0:
rep(2) null n,0,0; copy ,0,0; null n,0,0:
rep(4) null n,0,0:
rep(2) null n,0,0; data s,5,0; null n,0,0:
rep(2) null n,0,0; mult ,5,8; null ,0,0:
rep(2) null n,0,0; copy ,0,0; null ,0,0:
null n,0,0; data e,4,0; data s,5,0; null n,0,0:
null ,0,0; sub ,0,4; rep(2) null ,0,0:
null ,0,0; div ,0,8; rep(2) null ,0,0:
null ,0,0; copy ,0,0; rep(2) null ,0,0:
rep(4) null ,0,0:
null ,0,0; data s,5,0; rep(2) null ,0,0:
null ,0,0; mult ,5,8; data s,5,0; null ,0,0:
null ,0,0; copy ,0,0; mult ,5,8; null ,0,0:
null ,0,0; mov s,1,9; copy ,0,0; null ,0,0:
null ,0,0; data e,4,0; rep(2) null ,0,0:
null ,0,0; add ,4,9; rep(2) null ,0,0:
null ,0,0; copy ,0,0; rep(2) null ,0,0:
data e,4,0; rep(3) null ,0,0:
sub ,0,4; rep(3) null ,0,0:
div ,0,8; rep(3) null ,0,0:
copy ,0,0; rep(3) null ,0,0:
rep(4) null ,0,0:
rep(4) null ,0,0:
rep(2) null ,0,0; data s,5,0; null ,0,0:
null ,0,0; rep(2) data s,5,0; null ,0,0:
null ,0,0; rep(2) data s,5,0; null ,0,0:
rep(3) data s,5,0; null ,0,0
end

```

```

d(4,300)
{ N0. 6.5.2 part-1 }
{ Data file for the most general solution of a system of eqs. }
n 0.0,0.0,0.0,0.0;
none;none;none:
n 1.0,0.0,0.0,0.0;
none;none;none:
n 0.0,1.0,1.0,0.0;
none;none;none:
n 1.0,1.0,1.0,0.0;
none;none;none:
n 0.0,0.0,1.0,0.0;
none;none;none:
n 0.0,0.0,1.0,0.0;
none;none;none:
n 0.0,0.0,0.0,0.0;
none;none;none:
repl(27)[rep(4) none]:
repl(3)[rep(4) none]:

```

```

n 1.0,0.0,0.0,0.0;none;none;w 1.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 0.0,0.0,0.0,0.0;none;none;w 0.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 1.0,1.0,1.0,0.0;none;none;w 1.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 0.0,1.0,1.0,0.0;none;none;w 0.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 1.0,0.0,1.0,0.0;none;none;w 0.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 0.0,0.0,1.0,0.0;none;none;w 1.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 1.0,1.0,1.0,0.0;none;none;w 1.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 0.0,1.0,1.0,0.0;none;none;w 0.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 1.0,0.0,1.0,0.0;none;none;w 1.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 0.0,0.0,1.0,0.0;none;none;w 1.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 1.0,1.0,1.0,0.0;none;none;w 1.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 0.0,1.0,1.0,0.0;none;none;w 1.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 0.0,0.0,1.0,0.0;none;none;w 0.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 0.0,0.0,1.0,0.0;none;none;w 0.0,0.0,0.0,0.0:
repl(30)[rep(4) none]:
{new data}
n 1.0,0.0,0.0,0.0; none;none;none:
none; none; none; none:
n 0.0,1.0,0.0,0.0; none;none;none:
none; none; none; none:
n 0.0,0.0,1.0,0.0; none;none;none:
repl(3)[rep(4) none]:
n 1.0,0.0,0.0,0.0; none;none;none:
repl(3)[rep(4) none]:
n 0.0,0.0,0.0,0.0; none;none;none:
repl(7)[rep(4) none]:
n 0.0,0.0,1.0,0.0; none;none;none:
repl(30)[rep(4) none]:
n 0.0,0.0,0.0,0.0; none;none;none:
repl(3)[rep(4) none]:
n 0.0,1.0,0.0,0.0; none;none;none:
repl(7)[rep(4) none]:
n 0.0,0.0,1.0,0.0; none;none;none:
repl(30)[rep(4) none]:
n 1.0,0.0,0.0,0.0; none;none;none:
repl(3)[rep(4) none]:
n 0.0,1.0,0.0,0.0; none;none;none:
repl(7)[rep(4) none]:
n 0.0,0.0,1.0,0.0; none;none;none:
repl(30)[rep(4) none]:
n 0.0,0.0,0.0,0.0; none;none;none:
repl(3)[rep(4) none]:
n 0.0,0.0,0.0,0.0; none;none;none:
repl(7)[rep(4) none]:
n 0.0,0.0,1.0,0.0; none;none;none:
repl(30)[rep(4) none]
end

```

```

s(4,300)
{ NO. 6.5.2 part-1 }
{ Selector file for the most general solution of a system of eqs.}
1,0,0,0:

```

```

1,1,0,0:
1,1,1,0:
1,1,1,1:
repl(3)[rep(4)0]:
repl(2)[1,rep(3)0]:
0,0,1,0:
rep(3) 1, 0:
1, rep(3) 0 :
1,0,1,1:
rep(3) 1, 0:
repl(2)[1,1,0,0]:
0,1,0,0:
1,1,0,0:
0,1,1,0:
rep(4)1:
rep(3) 1,0:
rep(4) 1 :
repl(3)[1,1,0,0]:
repl(2)[rep(4)0]:
rep(3) 1,0:
1,1,0,0:
1,0,0,0:
repl(4)[rep(4) 0]:
repl(57)[1,rep(3)0]:
1,0,0,0:
0,0,0,0:
0,1,0,0:
1,0,1,0:
1,0,1,0:
0,0,0,0:
0,1,0,0:
1,1,0,0:
1,0,0,0:
0,0,0,0:
0,1,0,0:
0,1,1,0:
repl(4)[0, rep(2)1,0]:
repl(7)[rep(2) 0, 1,0]:
1,0,1,0:
1,0,0,0:
repl(2)[1,1,0,0]:
repl(2)[0,1,1,0]:
repl(2)[1,0,1,0]:
1,1,0,0:
0,1,0,0:
1,0,1,0:
0,0,0,0:
0,1,0,0:
0,0,0,0:
1,0,1,0:
1,1,0,0:
repl(2)[1,1,1,0]:
0,1,1,0:
repl(2)[0,1,0,0]:
rep(4)0:
repl(2)[0,0,1,0]:
1,0,1,0:
0,1,1,0:
repl(5)[0,0,1,0]:
0,1,1,0:
repl(7)[0,1,0,0]:
1,1,0,0:
repl(11)[1,0,0,0]:
0,0,0,0:
0,1,0,0:
repl(5)[1,0,0,0]:

```

```

1,1,0,0:
repl(2)[1,1,1,0]:
0,1,1,0:
repl(2)[0,1,0,0]:
rep(4)0:
repl(2)[0,0,1,0]:
1,0,1,0:
0,1,1,0:
repl(5)[0,0,1,0]:
0,1,1,0:
repl(7)[0,1,0,0]:
1,1,0,0:
repl(11)[1,0,0,0]:
0,0,0,0:
0,1,0,0:
repl(5)[1,0,0,0]:
1,1,0,0:
repl(2)[1,1,1,0]:
0,1,1,0:
repl(2)[0,1,0,0]:
rep(4)0:
repl(2)[0,0,1,0]:
1,0,1,0:
0,1,1,0:
repl(5)[0,0,1,0]:
0,1,1,0:
repl(7)[0,1,0,0]:
1,1,0,0:
repl(11)[1,0,0,0]:
0,0,0,0:
0,1,0,0:
repl(5)[1,0,0,0]:
1,1,0,0:
repl(2)[1,1,1,0]:
0,1,1,0:
repl(2)[0,1,0,0]:
rep(4)0:
repl(2)[0,0,1,0]:
1,0,1,0:
0,1,1,0:
repl(5)[0,0,1,0]:
0,1,1,0:
repl(7)[0,1,0,0]:
1,1,0,0:
repl(11)[1,0,0,0]:
0,0,0,0:
0,1,0,0:
repl(4)[1,0,0,0]
end

```



```

p(4,108)
{ NO. 6.5.2 part-2 }
{ Program for the most general solution of a system of eqs. }
repl(3)[rep(4) null n,0,0]:
{calculation}
rep(4) mult n w,3,6:
rep(4) add ,7,0:
rep(4) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,7,0:
rep(4) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,7,0:
rep(4) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,7,0:
rep(4) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
add ,8,0;rep(3) add ,7,0:
mov ,0,8;rep(3) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(2) add ,8,0;rep(2) add ,7,0:
rep(2) mov ,0,8;rep(2) mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(3) add ,8,0;add ,7,0:
rep(3) mov ,0,8;mov ,0,7:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,8,0:
rep(4) mov ,0,8:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
add ,9,0;rep(3) add ,8,0:
mov ,0,9;rep(3) mov ,0,8:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(2) add ,9,0;rep(2) add ,8,0:
rep(2) mov ,0,9;rep(2) mov ,0,8:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(3) add ,9,0;add ,8,0:
rep(3) mov ,0,9;mov ,0,8:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,9,0:
rep(4) mov ,0,9:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,9,0:
rep(4) mov ,0,9:
rep(4) mov ,6,1:
rep(4) mult n w,3,6:
rep(4) add ,9,0:
rep(4) mov ,0,9:
rep(4) mov ,6,1:
rep(4) mov ,7,0:

```

```

rep(4) copy ,0,0:
rep(4) mov ,8,0:
rep(4) copy ,0,0:
rep(4) mov ,9,0:
rep(4) copy ,0,0:
rep(4) null ,0,0:
data n,3,0; rep(3) null ,0,0:
rep(2) data n,3,0; rep(2) null ,0,0:
rep(3) data n,3,0; null ,0,0:
mov s,1,7; rep(2) data n,3,0; null ,0,0:
data n,3,0; mov s,1,7; data n,3,0; null ,0,0:
rep(2) data n,3,0; mov s,1,7; null ,0,0:
rep(3) data n,3,0; null ,0,0:
mov s,1,8; rep(2) data n,3,0; null ,0,0:
sub ,7,8; mov s,1,8; data n,3,0; null ,0,0:
mov ,0,9; sub ,7,8; mov s,1,8; null ,0,0:
data n,3,0; mov ,0,9; sub ,7,8; null ,0,0:
mult ,3,9; data n,3,0; mov ,0,9; null ,0,0:
copy ,0,0; mult ,3,9; data n,3,0; null ,0,0:
null ,0,0; mov ,0,10; mult ,3,9; null ,0,0:
null ,0,0; data w,6,0; mov ,0,10; null ,0,0:
null ,0,0; add ,6,10; rep(2) null ,0,0:
null ,0,0; copy ,0,0; rep(2) null ,0,0:
rep(2) null ,0,0; data w,6,0; null ,0,0:
rep(2) null ,0,0; add ,6,10; null ,0,0:
data n,3,0; null ,0,0; mov ,0,10; null ,0,0:
rep(2) data n,3,0; rep(2) null ,0,0:
rep(3) data n,3,0; null ,0,0:
mov s,1,7; rep(3) data n,3,0:
data n,3,0; mov s,1,7; rep(2) data n,3,0:
mult ,3,7; data n,3,0; mov s,1,7; data n,3,0:
copy ,0,0; mult ,3,7; data n,3,0; mov s,1,7:
null ,0,0; mov ,0,8; mult ,3,7; data n,3,0:
null ,0,0; data w,6,0; mov ,0,8; mult ,3,7:
null ,0,0; add ,6,8; null ,0,0; mov ,0,8:
null ,0,0; copy ,0,0; rep(2) null ,0,0:
rep(2) null ,0,0; data w,6,0; null ,0,0:
rep(2) null ,0,0; add ,6,8; null ,0,0:
rep(2) null ,0,0; add ,0,10; null ,0,0:
rep(2) null ,0,0; copy ,0,0; null ,0,0:
rep(3) null ,0,0; data w,6,0:
rep(3) null ,0,0; add ,6,8:
rep(3) null ,0,0; copy ,0,0:
rep(3) null ,0,0; data s,5,0:
rep(3) null ,0,0; data s,5,0:
rep(3) null ,0,0; data s,5,0:
rep(3) null ,0,0; data s,5,0:
rep(3) null ,0,0; data s,5,0
end

```

d(4,108)

```

{ N0. 6.5.2 part-2 }
{ Data file for the most general solution of a system of eqs. }
repl(3)[rep(4) none]:
n 1.0,0.0,0.0,0.0;none;none;w 0.5,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 0.0,0.0,0.0,0.0;none;none;w -0.5,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 1.0,1.0,1.0,0.0;none;none;w 0.5,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 0.0,1.0,1.0,0.0;none;none;w -0.5,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 1.0,0.0,1.0,0.0;none;none;w -0.5,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 0.0,0.0,1.0,0.0;none;none;w 0.5,0.0,0.0,0.0:

```

```

repl(3)[rep(4) none]:
n 1.0,1.0,1.0,0.0;none;none;w 0.5,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 0.0,1.0,1.0,0.0;none;none;w -0.5,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 1.0,0.0,1.0,0.0;none;none;w 0.25,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 0.0,0.0,1.0,0.0;none;none;w 0.25,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 1.0,1.0,1.0,0.0;none;none;w -0.25,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 0.0,1.0,1.0,0.0;none;none;w 0.75,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 0.0,0.0,1.0,0.0;none;none;w 0.0,0.0,0.0,0.0:
repl(3)[rep(4) none]:
n 0.0,0.0,1.0,0.0;none;none;w 0.0,0.0,0.0,0.0:
repl(10)[rep(4)none]:
n 0.0, 0.0, 0.0, 0.0;
none;none;none:
n 0.0, 0.0, 0.0, 0.0;
none;none;none:
n 1.0, 1.0, 1.0, 0.0;
none;none;none:
n 0.0, 0.0, 0.0, 0.0;
none;none;none:
n 0.0, 0.0, 0.0, 0.0;
none;none;none:
n 0.0, 0.0, 0.0, 0.0;
none;none;none:
n 1.0, 1.0, 1.0, 0.0;
none;none;none:
n 0.0, 0.0, 0.0, 0.0;
none;none;none:
n 0.0, 0.0, 0.0, 0.0;
none;none;none:
n 1.0, 0.0, 0.0, 0.0;
none;none;none:
n 0.0, 1.0, 0.0, 0.0;
none;none;none:
n 0.0, 0.0, 1.0, 0.0;
none;none;none:
repl(6)[rep(4) none]:
n 0.25, 0.0, 0.0, 0.0;
none;none;none:
n -0.5, 0.25, 0.0, 0.0;
none;none;none:
n 0.5, 0.5, -0.25, 0.0;
none;none;none:
n 0.0, -0.5, 0.5, 0.75;
none;none;none:
n 2.0, 0.0, 0.5, -0.5;
none;none;none:
n 0.0, 2.0, 0.0, -0.5;
none;none;none:
n 0.0, 0.0, 2.0, 0.0;
none;none;none:
n 0.0, 0.0, 0.0, 2.0;
none;none;none:
repl(15)[rep(4) none]
end

```

```

s(4,108)
{ NO. 6.5.2 part-2 }
{ Selector file for the most general solution of a system of eqs.}
repl(66)[1,rep(3)0]:
1,rep(3)0:
1,1,0,0:
1,1,1,0:
1,rep(3)0:
1,1,0,0:
rep(3)1,0:
rep(3)1,0:
1,rep(3)0:
1,1,0,0:
repl(9)[rep(3)1,0]:
0,1,1,0:
1,0,1,0:
1,1,0,0:
1,1,1,0:
1,0,0,0:
1,1,0,0:
repl(10)[rep(3)1,0]:
0,1,1,0:
0,0,1,0:
1,1,1,0:
1,1,0,0:
repl(2)[1,0,0,0]:
repl(2)[0,0,0,0]
end

```

```

p(4,39)
{ NO. 6.6 }
{ Program for the deletion from the heap sort }
{loading data}
data n,3,0; rep(3) null ,0,0:
data n,3,0; rep(3) null ,0,0:
rep(3) data n,3,0; null ,0,0:
null ,0,0; rep(2) data n,3,0; null ,0,0:
rep(2) null ,0,0; data n,3,0; null ,0,0:
{calculation}
rep(2) null ,0,0; data s,5,0; null ,0,0:
null ,0,0; data e,4,0; data s,5,0; null ,0,0:
null ,0,0; data s,5,0; rep(2) null ,0,0:
min e,4,1; max w,6,1; rep(2) null ,0,0:
null ,0,0; max e,4,1; min w,6,1; null ,0,0:
null ,0,0; data s,5,0; rep(2) null ,0,0:
rep(4) null ,0,0:
null ,0,0; data e,4,0; rep(2) null ,0,0:
rep(2) null ,0,0; data n,3,0; null ,0,0:
min e,4,1; max w,6,1; rep(2) null ,0,0:
null ,0,0; max e,4,1; min w,6,1; null ,0,0:
null ,0,0; data s,5,0; rep(2) null ,0,0:
data s,5,0; data w,6,0; rep(2) null ,0,0:
null ,0,0; max e,4,1; min w,6,1; null ,0,0:
rep(4) null ,0,0:
min e,4,1; max w,6,1; rep(2) null ,0,0:
rep(4) null ,0,0:
null ,0,0; data s,5,0; rep(2) null ,0,0:
null ,0,0; data w,6,0; rep(2) null ,0,0:
data n,3,0; rep(3) null ,0,0:
null ,0,0; max e,4,1; min w,6,1; null ,0,0:
rep(4) null ,0,0:
min e,4,1; max w,6,1; rep(2) null ,0,0:
rep(4) null ,0,0:
null ,0,0; data s,5,0; rep(2) null ,0,0:
rep(4) null ,0,0:
null ,0,0; data e,4,0; rep(2) null ,0,0:
min e,4,1; max w,6,1; rep(2) null ,0,0:
rep(4) null ,0,0:
null ,0,0; data s,5,0; rep(2) null ,0,0:
null ,0,0; data w,6,0; rep(2) null ,0,0:
null ,0,0; data s,5,0; rep(2) null ,0,0:
null ,0,0; data s,5,0; rep(2) null ,0,0:
rep(4) null ,0,0
end

```

```

d(4,39)
{ NO. 6.6 }
{ Data file for the deletion from the heap sort }
n 4.0,0.0,0.0,0.0;rep(3) none:
n 5.0,0.0,0.0,0.0;rep(3) none:
n 1.0,9.0,6.0,0.0;rep(3) none:
n 0.0,0.0,7.0,0.0;rep(3) none:
n 0.0,0.0,3.0,0.0;rep(3) none:
repl(34)[rep(4) none]
end

```

```

s(4,39)
{ NO. 6.6 }
{ Selector file for the deletion from the heap sort }
repl(3)[rep(3) 1, 0]:
repl(2)[rep(4) 0]:
0, rep(2) 1, 0:
rep(2) 1, rep(2) 0:
rep(4) 0:
0, 1, rep(2) 0:
rep(2) 1, rep(2) 0:
rep(4) 0:
0, 1, rep(2) 0:
1, 1, rep(2) 0:
rep(4) 0:
0, 1, rep(2) 0:
rep(2) 1, rep(2) 0:
rep(4) 0:
0, 1, rep(2) 0:
0, rep(2) 1, 0:
repl(2)[0, 1, rep(2) 0]:
rep(2) 1, rep(2) 0:
rep(4) 0:
repl(2)[0, 1, rep(2) 0]:
rep(2) 1, rep(2) 0:
rep(4) 0:
0, 1, rep(2) 0:
rep(2) 1, rep(2) 0:
rep(4) 0:
repl(3)[0, 1, rep(2) 0]:
rep(2) 1, rep(2) 0:
rep(4) 0:
repl(3)[rep(2) 1, rep(2) 0]:
rep(3) 1, 0
end

```

```

p(4,121)
{ NO. 6.7 HPI }
{ Program for Hermite Polynomial Interpolation }
data n,3,0; rep(3) null ,0,0:
mov s,1,7; rep(3) null ,0,0:
rep(2) data n,3,0; rep(2) null ,0,0:
rep(2) mov s,1,8; rep(2) null ,0,0:
rep(2) data n,3,0; rep(2) null ,0,0:
rep(2) mov s,1,9; rep(2) null ,0,0:
rep(2) data n,3,0; rep(2) null ,0,0:
rep(2) mov s,1,10; rep(2) null ,0,0:
rep(2) sub ,8,9; rep(2) null ,0,0:
rep(2) mov ,0,11; rep(2) null ,0,0:
rep(2) sub ,8,10; rep(2) null ,0,0:
rep(2) mov ,0,12; rep(2) null ,0,0:
rep(2) mult ,11,12; rep(2) null ,0,0:
rep(2) mov ,0,13; rep(2) null ,0,0:
rep(2) sub ,9,8; rep(2) null ,0,0:
rep(2) mov ,0,14; rep(2) null ,0,0:
rep(2) sub ,9,10; rep(2) null ,0,0:
rep(2) mov ,0,15; rep(2) null ,0,0:
rep(2) mult ,14,15; rep(2) null ,0,0:
rep(2) mov ,0,16; rep(2) null ,0,0:
rep(2) sub ,10,8; rep(2) null ,0,0:
rep(2) mov ,0,17; rep(2) null ,0,0:
rep(2) sub ,10,9; rep(2) null ,0,0:
rep(2) mov ,0,18; rep(2) null ,0,0:
rep(2) mult ,17,18; rep(2) null ,0,0:
rep(2) mov ,0,19; rep(2) null ,0,0:
sub ,7,9; sub ,8,10; rep(2) null ,0,0:
rep(2) mov ,0,11; rep(2) null ,0,0:
sub ,7,10; sub ,8,9; rep(2) null ,0,0:
rep(2) mov ,0,12; rep(2) null ,0,0:
mult ,11,12; add ,11,12; rep(2) null ,0,0:
rep(2) mov ,0,14; rep(2) null ,0,0:
sub ,7,8; sub ,9,10; rep(2) null ,0,0:
rep(2) mov ,0,15; rep(2) null ,0,0:
mult ,15,12; sub ,9,8; rep(2) null ,0,0:
rep(2) mov ,0,17; rep(2) null ,0,0:
mult ,15,11; add ,15,17; rep(2) null ,0,0:
rep(2) mov ,0,18; rep(2) null ,0,0:
div ,14,13; sub ,10,9; rep(2) null ,0,0:
mov ,0,13; mov ,0,11; rep(2) null ,0,0:
div ,17,16; sub ,10,8; rep(2) null ,0,0:
mov ,0,14; mov ,0,12; rep(2) null ,0,0:
div ,18,19; add ,11,12; rep(2) null ,0,0:
mov ,0,16; mov ,0,17; rep(2) null ,0,0:
mult ,13,13; div ,14,13; rep(2) null ,0,0:
rep(2) mov ,0,13; rep(2) null ,0,0:
mult ,14,14; div ,18,16; rep(2) null ,0,0:
rep(2) mov ,0,14; rep(2) null ,0,0:
mult ,16,16; div ,17,19; rep(2) null ,0,0:
rep(2) mov ,0,16; rep(2) null ,0,0:
data n,3,0; mov ,13,0; rep(2) null ,0,0:
mov s,1,7; rep(3) null ,0,0:
data n,3,0; copy ,0,0; rep(2) null ,0,0:
mov s,1,8; rep(3) null ,0,0:
data e,4,0; mov ,14,0; rep(2) null ,0,0:
mov s,1,17; copy ,0,0; rep(2) null ,0,0:
data e,4,0; mov ,16,0; rep(2) null ,0,0:
mov s,1,18; copy ,0,0; rep(2) null ,0,0:
data e,4,0; rep(3) null ,0,0:

```

```

mov s,1,19; rep(3) null ,0,0:
mult ,8,15; rep(3) null ,0,0:
mov ,0,9; rep(3) null ,0,0:
mult ,9,17; rep(3) null ,0,0:
mov ,0,9; rep(3) null ,0,0:
mult ,8,11; rep(3) null ,0,0:
mov ,0,10; rep(3) null ,0,0:
mult ,10,18; rep(3) null ,0,0:
mov ,0,10; rep(3) null ,0,0:
mult ,8,12; rep(3) null ,0,0:
mov ,0,17; rep(3) null ,0,0:
mult ,17,19; rep(3) null ,0,0:
mov ,0,17; rep(3) null ,0,0:
sub ,7,9; rep(3) null ,0,0:
mov ,0,9; rep(3) null ,0,0:
sub ,7,10; rep(3) null ,0,0:
mov ,0,10; rep(3) null ,0,0:
sub ,7,17; rep(3) null ,0,0:
mov ,0,17; rep(3) null ,0,0:
mult ,9,13; rep(3) null ,0,0:
mov ,0,7; rep(3) null ,0,0:
mult ,10,14; rep(3) null ,0,0:
mov ,0,8; rep(3) null ,0,0:
mult ,17,16; rep(3) null ,0,0:
mov ,0,9; rep(3) null ,0,0:
mult ,15,13; rep(3) null ,0,0:
mov ,0,10; rep(3) null ,0,0:
mult ,11,14; rep(3) null ,0,0:
mov ,0,11; rep(3) null ,0,0:
mult ,12,16; rep(3) null ,0,0:
mov ,0,12; rep(3) null ,0,0:
data n,3,0; rep(3) null ,0,0:
mov s,1,13; rep(3) null ,0,0:
data n,3,0; rep(3) null ,0,0:
mov s,1,14; rep(3) null ,0,0:
data n,3,0; rep(3) null ,0,0:
mov s,1,15; rep(3) null ,0,0:
data n,3,0; rep(3) null ,0,0:
mov s,1,16; rep(3) null ,0,0:
data n,3,0; rep(3) null ,0,0:
mov s,1,17; rep(3) null ,0,0:
data n,3,0; rep(3) null ,0,0:
mov s,1,18; rep(3) null ,0,0:
rep(2) data n,3,0; rep(2) null ,0,0:
mult ,13,7; rep(3) null ,0,0:
mov ,0,7; rep(3) null ,0,0:
mult ,14,8; rep(3) null ,0,0:
mov ,0,8; rep(3) null ,0,0:
mult ,15,9; rep(3) null ,0,0:
mov ,0,9; rep(3) null ,0,0:
mult ,16,10; rep(3) null ,0,0:
mov ,0,10; rep(3) null ,0,0:
mult ,17,11; rep(3) null ,0,0:
mov ,0,11; rep(3) null ,0,0:
mult ,18,12; rep(3) null ,0,0:
add ,0,11; rep(3) null ,0,0:
add ,0,10; rep(3) null ,0,0:
add ,0,9; rep(3) null ,0,0:
add ,0,8; rep(3) null ,0,0:
add ,0,7; rep(3) null ,0,0:
copy ,0,0; rep(3) null ,0,0:
rep(4) null ,0,0
end

```



```

d(4,121)
{ NO. 6.7 HPI }
{ Data file for Hermite Polynomial Interpolation }
n 1.5, 0.0, 0.0, 0.0; none; none; none:
rep(4) none:
n 1.3, 1.3, 0.0, 0.0; none;none;none:
rep(4) none:
n 1.6, 1.6, 0.0, 0.0; none;none;none:
rep(4) none:
n 1.9, 1.9, 0.0, 0.0; none;none;none:
repl(43)[rep(4) none]:
n 1.0, 0.0, 0.0, 0.0; none; none; none:
rep(4) none:
n 2.0, 0.0, 0.0, 0.0; none; none; none:
repl(37)[rep(4) none]:
n 0.620086, 0.0, 0.0, 0.0; none; none; none:
rep(4) none:
n 0.455402, 0.0, 0.0, 0.0; none; none; none:
rep(4) none:
n 0.281818, 0.0, 0.0, 0.0; none; none; none:
rep(4) none:
n -0.522023, 0.0, 0.0, 0.0; none; none; none:
rep(4) none:
n -0.569895, 0.0, 0.0, 0.0; none; none; none:
rep(4) none:
n -0.581157, 0.0, 0.0, 0.0; none; none; none:
rep(4) none:
n 0.0, 0.0, 0.0, 0.0; none; none; none:
repl(18)[rep(4) none]
end

s(4,121)
{ NO. 6.7 HPI }
{ Selector file for Hermite Polynomial Interpolation }
repl(121)[1,0,0,0]
end

```

```

p(4,29)
{ NO. 6.7 PE }
{ Program for parallel evaluation }
rep(4) null ,0,0:
rep(4) null ,0,0:
rep(4) null ,0,0:
rep(4) data n,3,0:
rep(4) mov ,3,7:
null ,0,0; rep(3) mult ,3,7:
null ,0,0; mov ,0,8; rep(2) mult ,0,7:
rep(2) null ,0,0; rep(2) mult ,0,7:
rep(2) null ,0,0; mov ,0,8; mult ,0,7:
rep(3) null ,0,0; mult ,0,7:
rep(3) null ,0,0; mov ,0,8:
rep(4) data n,3,0:
rep(4) mult ,3,7:
rep(4) mov ,0,9:
rep(4) data n,3,0:
rep(4) add ,3,9:
copy ,0,0; rep(3) mov ,0,10:
null ,0,0; rep(3) mult ,8,10:
null ,0,0; rep(3) mov ,0,10:
null ,0,0; data w,6,0; rep(2) null ,0,0:
null ,0,0; add ,6,10; rep(2) null ,0,0:
null ,0,0; copy ,0,0; rep(2) null ,0,0:
rep(2) null ,0,0; data w,6,0; null ,0,0:
rep(2) null ,0,0; add ,6,10; null ,0,0:
rep(2) null ,0,0; copy ,0,0; null ,0,0:
rep(3) null ,0,0; data w,6,0:
rep(3) null ,0,0; add ,6,10:
rep(3) null ,0,0; copy ,0,0:
rep(4) null ,0,0
end

```

```

d(4,29)
{ NO. 6.7 PE }
{ Data file for parallel evaluation }
repl(3)[rep(4) none]:
n 2.0,2.0,2.0,2.0;none;none;none:
repl(7)[rep(4) none]:
n 4.0,6.0,8.0,10.0;none;none;none:
rep(4) none:
rep(4) none:
n 3.0,5.0,7.0,9.0;none;none;none:
repl(14)[rep(4) none]
end

```

```

s(4,29)
{ NO. 6.7 PE }
{ Selector file for parallel evaluation }
repl(29)[1,0,0,0]
end

```

