An Intelligent Cell Memory System For Real Time Engineering Applications

by

Kam-Fai Wong, B.Sc.(Hons)

Doctor of Philosophy

Edinburgh University

May, 1987

Dedication

To my wife Alley and daughter Carmen

Without their constant encouragement and support, this work would never have been accomplished.

Table of Contents

Chapter III ....Garbage Collection

Chapter IV.....The Intelligent Cell Memory System

Chapter V .....Construction I: Hardware

Chapter VI.....Construction II: Microprogramming

Chapter VII... Verification and Results

Chapter VIII.Conclusion

APPENDICES

Acknowledgement

The author wishes to express his gratitude to Dr. John H. Hannah for his constructive criticisms and invaluable advice throughout the past two years. Despite his work load as an associate dean of the university, Dr. Hannah has exercised a good deal of patience and spent a great amount of time in proof reading intermediate drafts, which has helped me develop the thesis to its present state.

The author is much indebted to his ex-supervisor Dr. George G. Coghill for his useful insight and sound guidance which helped a lot with the initial formulation of the project.

Sincere thanks are due to Mr. Alex F.L. Wong, Mr. Cyril H.F. Chan, Mr. C.H. Lau and Mr. Hamish Taylor for their continuous comments, assistance and encouragement.

Last but not least, the author would like to express his appreciation and love to his wife Alley. Her patience, constant care, support and encouragement have withstood the difficult times and she has helped in every way she could to make this work all worthwhile.

Declaration

This thesis is composed by the author and the work described herein is original unless otherwise indicated.

The author,

Kam-Fai Wong BSc

Abstract

There is a growing interest in the application of Artificial Intelligence (AI) techniques in engineering. Existing AI systems are not suitable for many applications because of their unacceptable real time response and complexity. An Intelligent Cell Memory System (ICMS) which is capable of providing improved real time performance is proposed in this thesis. The ICMS adopts a novel mode of garbage collection which guarantees a bounded separation time. The garbage collector is an optimised hardware realisation of Baker's algorithm. A prototype of the ICMS which has been constructed using microprogrammed bit-slice processors is described. It is configured as a coprocessor module and loosely coupled to a M68000 microprocessor via a standard VME bus interface. Experiments on the ICMS prototype have revealed more than one order of magnitude improvement in system performance. Suggestions for further development of the ICMS approach are presented.

List of Figures

List of Photographs

List of Tables

CHAPTER I
Introduction

"... *There are several bottlenecks within typical expert systems that could prevent successful monitoring of complex, real-time events.* ..... *A traditional LISP machine is designed to be a thinking machine, not a real-time controller.*" [1]

1.1 Background

Artificial Intelligence AI† is a science which attempts to provide machines (artificial entities) with human-like behaviour : the ability to store and acquire knowledge and to reason and act on deductions as a human being would. Unfortunately, after decades of development, very few practical AI systems are suitable for engineering applications. This lack of success is largely a result of two major shortcomings of existing AI technology :

(1) *Complexity of system hardware.*

AI software using conventional programming languages, is large and complicated, hard to comprehend and difficult to implement. Efforts have therefore been made by researchers to produce special AI languages with highly efficient compilers. The nature of this AI software leads to inefficient processing by conventional computers. Some novel machines have been designed to improve processing efficiency. Very Large Scale Integration (VLSI) - design at chip level has produced special purpose AI processors which often involve the use of non Von Neumann architectures. Other machines based on existing technology often use coprocessor systems. Most AI machines however are large, complex and costly and have a system interface which is non-standard and unsuitable for engineering applications.

† a formal definition will be given in chapter two.

(2) *Poor real time performance.*

Programming languages with special features are necessary to solve AI problems. By far their most important feature is *symbolic manipulation*. Computer intelligence implies reasoning with knowledge or "rules of thumb". Each piece of knowledge, internally, is an object represented by a symbol. In programming terms, objects have lists of properties and values associated with them. Functionally, reasoning is done by making heuristic inferences according to the information obtained from objects' properties and the relationships between them. The most widely used language is LISP, a language originally designed for symbolic manipulation.

During the process of reasoning, units of memory are dynamically being consumed and released. With a finite storage system, memory exhaustion is bound to occur unless the released memory can be re-used. The process of memory reorganisation so as to reclaim reusable units is called *Garbage Collection*.

Garbage Collection is usually considered to be the responsibility of the host processor. Previously, to run such a process, the host would suspend all active jobs and dedicated itself solely to retrieving re-usable memory. This was acceptable in the research and development environments of the past. Nevertheless, the complexity of present AI software has rendered this approach inadequate, especially, in practical engineering systems. To have to stop in the middle of a continuous task could be costly and dangerous and is completely impossible in a real time system.

1.2 Garbage Collection

Currently, three practical approaches to improve real time AI performance are being investigated:

(1) software garbage collection algorithms with real time capability;

(2) special architecture AI machines with microcoded garbage collection schemes; and

(3) a dedicated hardware garbage collector in a multiprocessor environment.

The first of these provides the basis for the rest which are largely hardware realisations of the software scheme. At present, the software solution is still practical in some situations. However, it is only a partial remedy marginally acceptable for systems with a modest response time, e.g. a single user interactive programming environment.

The second approach is most efficient and provides the fastest response. The processors used in these machines are mostly based on semi-custom Very Large Scale Integrated (VLSI) technology. Garbage collection schemes are usually microcoded and completely transparent to users. However, these processors are usually part of complex systems which have their own specifications and are difficult to integrate into engineering systems.

The performance offered by the third solution could be comparable with the second, providing processors are configured in the proper way. Concurrent processing techniques have long been employed to remove the burden from host processors. In this case, a second processor or special purpose piece of hardware is dedicated for garbage collection; thus removing the problem of memory exhaustion from the host processor. As hardware costs decrease, the multiprocessing approach becomes increasingly promising. However, present systems are far from perfect : they are

- **difficult to implement**: crucial inter-processor communication protocols are required, e.g. dead-lock avoidance using semaphores. As a consequence of

this, debugging and maintenance problems become significant;

- **lacking in flexibility**: the garbage collection hardware is constructed around specific processors; and worst of all interfaces are largely based on non-standard specifications.

The multiprocessor approach is superior to the rest, despite its disadvantages, because of its low cost, ease of expansion and flexibility. The software approach is becoming obsolete; nevertheless, it is still viable if a quick and cheap solution demanding only a moderate response is required. The special purpose processor with its dedicated architecture is promising, but not until some standardised architecture with an internationally accepted interface standard has been achieved. In the mean time, the multiprocessor approach will remain the dominant practical solution for improving real time AI system performance.

1.3 The Project

The aim of the project is to provide a simple yet efficient solution to improve an AI system's real-time performance. This is achieved by the design of a special purpose coprocessor system known as ICMS - Intelligent Cell Memory System to remove the burden of garbage collection from the host. A novel operational mode of garbage collection is introduced. An ICMS prototype has been constructed. Experiments on the prototype have shown that the ICMS hardware can provide at least an order of magnitude increase in execution speed for the five basic list processing primitives: car, cdr, replaca, replacd and cons.

The ICMS has been designed to meet the need for low cost, portable, intelligent instruments for engineering applications. These systems would have wide practical potential, such as monitoring, control, manufacturing, etc. The final design is based

on the VME bus specification (IEEE 1014 standard multiprocessor bus), a popular de facto industrial standard. This makes the overall system easily expandable and not confined to a specific type of processor.

The project was divided into three stages: a literature survey, the design and implementation of supporting software and the design and construction of the ICMS system. During the literature survey stage, efforts were made to identify the most important features of AI languages and processors. In addition, a survey of garbage collection schemes was carried out. Although the project is mainly hardware orientated, several pieces of software were produced in stage two, to make the final system operational and to assess its comparative performance. This included the design and implementation of a LISP dialect *simpleLISP* and its interpreter and a microprogram assembler. In the third stage an ICMS prototype was constructed using bit slice technology and by microprogramming. The five basic list processing primitives were included in the design. This means that, effectively, the coprocessor may be regarded as a list processing accelerator. Finally the overall system performance was studied.

1.4 Chapter Summary

Because of the unique nature of AI problems, special features are required for programming languages. These features are identified and a survey of common AI languages is given in the first half of chapter two. To enhance language performance, dedicated hardware systems are being developed. Current trends in hardware design are reviewed and examples of existing practical machines are described at the end of chapter two. Classical garbage collection schemes, current state of the art techniques

and desirable characteristics of a real time garbage collection system are described in chapter three.

Chapter four reviews the functionality of the ICMS, the ICMS system interface and the operational details of the adopted garbage collection algorithm. The design details of the ICMS prototype to achieve the desirable features outlined in chapter three are also described.

The construction of the system is described in chapter five and six. Chapter five reviews the hardware construction details. The ICMS is constructed using bit-slice devices, and internally it can be divided into several functional units, the operation of which is depicted. In chapter six, the implementation of the micro-assembler software, the field assignment of a microinstruction word and the ICMS microinstruction routines are explained.

In chapter seven, the performance of a system using the ICMS and one without is compared.

Finally, the thesis concludes with ideas leading to possible improvements of the prototype system. Suggestions are also given for advanced realisations of the ICMS philosophy in the future.

CHAPTER II
AI Systems

In this chapter, a formal definition of Artificial Intelligence is given and the advantages of applying AI to engineering problems are outlined. Also, the two main factors which have been limiting the widespread application of this approach are identified and described.

2.1 Background

Artificial Intelligence (AI) originated in the fifties when several problems were partly solved with computers using automatic deduction techniques. Ever since there has been a rapidly increasing interest in research on the subject worldwide. The term AI is formally defined as "... the part of computer science that is concerned with the symbol-manipulation processes that produce intelligent action. By 'Intelligent action' is meant an act of decision that is goal-oriented, arrived at by an understandable chain of symbolic analysis and reasoning steps, and is one in which knowledge of the world informs and guides the reason "[2]. Alty[3], with his automation model, figure 2.1, asserts that a problem is always divided into two parts : one is automated (by machines) and the other involves human activity. The objective of the application of AI is to shift the boundary in the problem space. As AI technology improves, the boundary is shifting in the direction of the arrow and gradually encompassing more and more of the human activity.

In practice, AI is implemented as a set of advanced computer software applicable to problems such as natural language understanding, image recognition, expert systems, knowledge acquisition and representation, heuristic search, deductive reason-

ing and planning. A rich set of literature exists on the topic. Winston [4], Gevarter[5] and Nilsson[6] are all excellent introductory sources. Also in Brown [2], Barr et al [7] and [8] overviews of the current status and future research possibilities of AI are given.

In recent years, AI technology has been taken from laboratories to be employed in practical areas. With constantly decreasing hardware costs and increasing computing power, this trend will accelerate. Many new projects and research programmes are scheduled and all of them involve significant funding. Notable among these are the 10-year Fifth Generation Computing programme in Japan [9], the ESPRIT programme in Europe, the Strategic Computing project sponsored by the Department of Defence in the USA [2] and the Information Technology boost supported by the Alvey Commission in the UK [10].

2.1.1 AI for Engineering

The potential of AI has also been realised in many fields of engineering. Practical systems have been built for a wide range of applications - such as fault diagnosis for machines and equipment [11, 12], engineering consultancy [13], factory automation [14], testing of electronic systems [15], process control [16] and VLSI design [17, 18].

The application of AI techniques to engineering overcomes the disadvantages of normal human and machine approaches. A fully automated system is an engineering design paradigm. Automation guarantees precision, fast response and robustness. It alleviates the physical limitation of human engineers; especially when they are subjected to psychological influences - boredom, weariness and depression.

In the first stage of the design of automated systems, the process of the applica-

tion area concerned is analytically modelled using complicated mathematics. Unfortunately, under operational circumstances, these models fail because they are usually constructed under several ideal assumptions. At this point, human effort inevitably intervenes. The complication of the analytical models frequently demands fully-trained personnel or even sometimes the design engineers themselves in order to regulate "abnormal" behaviour of the systems.

The introduction of AI technology into engineering makes human expertise available at machine level. Intelligent engineering systems are based on *causality*. They emulate the procedure of problem-solving adopted by human experts who investigate the cause of the problems. Engineering "know-how" is electronically encoded and subsequently retrieved by heuristic searching strategies. Moreover, AI can provide a friendly Man Machine Interface (MMI) using natural language understanding and/or speech recognition techniques. A less obvious benefit of AI for engineering exists in training and education. A friendly MMI exposes engineering knowledge to non-technical users readily and comprehensibly. Summarising, AI technology, and its application to engineering in particular, can "preserve otherwise perishable human expertise; distribute otherwise scarce expertise; reduce the cost of mediocre or poor human performance and provide help to humans trying to access information and employ computers."[19]

2.2 AI Language Features

There are several common characteristics [20] which high level programming languages applicable to large complex systems should possess. These include

- support of a variety of data types to describe all kinds of information structures;
- support of flexible control structures, such as recursion and iteration;

- support of parallel control structure, e.g. fork()/wait() in C;
- ability to decompose the system into small, understandable chunks so that it is possible to alter one part of the system without disturbing other parts.

In addition, individual languages also possess local sets of higher level functional primitives (environment) which expedite problem solving in a specific application domain. For example, Cobol and Fortran are widely used for business and number-crunching, respectively.

AI problems are large, complex and unique, therefore special high level languages are required to facilitate efficient information handling and control. On top of the characteristics above, languages for AI programming should exhibit the following features [7, 21] :

2.2.1 Data Abstraction

i) *Symbolic manipulation and list processing.*

Information to be manipulated by AI programs is suitably represented by symbols. Symbols are named entities, resembling a physical representation of objects, which have lists of associated properties, e.g. if "square" is a symbol, it could have properties as {"a polygon", "four vertices",etc.}. The capability of symbolic manipulation is essential for languages to be used in AI applications. Association or properties of symbols are represented in list structures. Efficient handling of lists facilitates the process of machine reasoning.

ii) *Dynamic and late binding.*

Symbols are arbitrary defined and could stand for anything, numbers or characters or names, etc. Their actual semantics are revealed upon functional invocation.

iii) *Functional/Applicative programming.*

AI programs consist of functions defined in mathematical formats. Effectively, data and functions are all symbols, which makes it possible for them to be intermixed giving referential transparency.

iv) *Dynamic storage allocation.*

The basic storage unit of a list is a cell. The simplicity of the cell data type assists the creation of complicated information structures. The non-deterministic nature of AI problems, e.g the next move of a chess game, renders predefined variables inapplicable. The advantage of dynamic storage allocation is the ability to "create" cells on demand (when needed).

2.2.2 Control Structures

i) *Pattern matching.*

It is used to identify specific symbols and to determine control. In practice, most existing AI systems are pattern directed e.g. Hearsay-II[22], for speech understanding.

ii) *Logic programming.*

One type of pattern-invoked control mechanism of particular interest is production rules. Typically of the form :

IF *condition* **THEN** *primitive action.*

The *condition* is usually a set of predicates which examines properties about the current state and the *primitive action* is some simple operation that changes the current state. If the predicates are positive then the *primitive action* is executed. Systems using production rules to transform their state are sometimes known as **state automata**†.

† Automata usually refers to the class of systems that operate on discrete data representations in discrete time intervals.

iii) *Deduction mechanism.*

The provision of forward- and/or backward-chaining capability aids the process of deduction. Forward and backward chaining are steps taken to solve a problem. If reasoning starts from a set of conditions and moves toward some conclusion, the method is known as forward chaining. If the conclusion is well defined but the path to the conclusion is unknown then backward chaining is employed to retrace the route of reasoning. For a state automaton, forward chaining implies that the conjunction of predicate *conditions* is used to drive the *primitive action* (**antecedent driven** [23]); on the other hand, the system states (i.e. *primitive action*) are known in a backward chaining system, and they are used to derive the *condition* or goals (**goal driven**).

No existing AI language provides all of these properties. Some languages do better than others. Chapter six of Barr et al [7] gives a more detailed description of the desirable AI language features with reference to several practical systems. Corlett [24] suggests similar language features for AI programming. Moreover, from a software engineer's point of view, Corlett also identifies features for a complete AI programming environment essential for software development. There are many practical AI languages; but by far the most popular are LISP and Prolog. Table 2.2, (a part extraction from [24]), compares the features of them with other common conventional languages.

2.2.3 LISP

LISP [25] is one of the oldest AI languages which was designed and implemented by John McCarthy at the MIT in 1960. It is a functional programming

language. The idea of functional programming is borrowed from mathematics theory. Its mechanism is simply to transform the *range* (or arguments) into the corresponding *domain* (or solution) under some predefined mapping function. The characteristics of LISP are:

- As its name suggests, **LIST** Processing, LISP is extensively based on list processing.
- Simple but flexible data structuring is achieved by LISP using two data types atom and list.
- It supports a small but well defined set of functional primitives, namely: CAR, CDR, REPLACAR, REPLACDR, CON, ATOM and EQ.
- Dynamic and late bindings of functional parameters at run time and referential transparency allowing intermix of program and data.

LISP was originally designed as a purely functional language but throughout the years imperative features have been included into various LISP dialects. Some well known LISP dialects are MACLISP the nearest kin to the original pure LISP by McCarthy, FranzLISP [26] written in C by the University of California at Berkeley which is operational on many UNIX hosts and CommonLISP the proposed standard by the Defense Advanced Research Projects Agency (Darpa) of the U.S.A. The differences that exist between each dialect are due partly to the variation in computer hardware and the host operating systems†.

---

† Control flow in imperative languages (e.g. PASCAL, C) is explicit and is governed by the content of the program counter. They can be regarded as the high level representation of the Von Neumann computer. In functional languages (LISP, FP) control is based on graph or string reduction.

2.2.4 Prolog

Prolog [27, 28] was invented jointly by Colmerauer and Kowalski in 1972. Since then several implementations have appeared, each with a different syntax. Some examples are micro-Prolog implemented at Imperial College, for Z80 based microcomputers with CP/M operating systems, and C-Prolog [29] written in C and portable to most UNIX systems. The concept of Prolog is based on **PRO**gramming in **LOG**ic. The original objective was to facilitate automatic theorem proving using logic deduction. Unlike LISP in which programs are formed out of collections of functions, a Prolog program consists of a sequence of relations and rules describing the problem domain. These form a database of information that can be queried or added to. A mixed tracking mode is in built. Consider the rule :

X is_true_if (Y and Z are both true).

All capital letters are unknowns. In this case, the left hand side i.e. X is called the goal and on the right hand side Y and Z are considered as subgoals. The criterion for rule evaluation in Prolog is :

> To reach a goal, all subgoals have to be reached or satisfied. When a subgoal cannot be reached, backtracking is performed to search for another statement that matches an earlier subgoal.

Therefore, control flow is governed by pattern matching - technically known as *unification* in Prolog. In practice besides pattern matching, parameters are bound during the unification process. Programming in Prolog is hierarchical and it is done in small, self contained, individually testable chunks. List processing is also supported using explicit list data definition (i.e. [<atoms>] in C-Prolog).

2.2.5 Software Tools

The trend in AI software technology is moving toward the design of high level development tools. The role of these tools is to provide a rich programming environment as suggested in Corlett [24] : interactive environment, incremental coding, integrated editor, crash proof systems, powerful debugging aids, pretty printing, automatic filing, program understanding aids and extensibility, thus reducing the time and effort of software development.

The technological achievement of software tools is assessed by Hayes-Roth [30] using the graph shown in figure 2.2. Under his assessment productivity (in terms of engineering hours per rule) has approximately doubled annually over the last 15 years. This trend is likely to continue and more sophisticated tools will be created and constantly improved under practical feedback. In the figure five successive technological advancements are identifiable. They are "programming languages, such as Lisp; programming environments, such as InterLisp; research tools, such as Emycin; commercial tools, such as S.1; and anticipated generic knowledge systems that incorporate a user's own knowledge into a prefabricated heuristic problem-solving package like a personal planner." [19] The design and development of AI tools and their applications are expected to dominate the industrial/commercial market in the short term future. By 1990, as predicated by Suydam [31], a total amount of over 800 million dollars will be the value of the market.

AI languages and high level software tools expedite the process of problem solving. This is, however, achieved at the expense of degraded real time performance.

2.3 Garbage Collection

The major function of AI systems is to emulate the human procedure of deduction/reasoning. Internally, the process of deduction/reasoning requires storage cells which act as vehicles to transport rules and ideas. These cells are "created" dynamically. They are mostly empirical entities and are often abandoned immediately after their utilisation. Dynamic cell allocation heavily degrades system performance. Physically, storage media have finite capacity - allowing memory cells to be allocated continuously, would eventually result in storage being exhausted. The process of continuous memory allocation is prevented due to congestion created by useless cells "living" in systems. Book-keeping strategies are specially devised to identify useless cells, or garbage, and recycle them for later usage. These strategies are technically known as *garbage collection.* Although garbage collection successfully relieves storage congestion, it creates a problem which affects the overall system performance. High level processing is frequently forced to halt in order to await the completion of garbage collection. This unpredictable process interruption can make AI software unsuitable for real time applications, for which most engineering systems are designed.

2.4 AI Hardware

The potential of AI techniques is gradually being realised not merely in research and development but also in office and factory applications. Increasing market demands for fast systems have urged the launch of a whole new range of AI machines. The design of these machines is based on one of the following four system configurations: VLSI processors, LSI systems, coprocessor systems and non Von Neuman architectures. In the following sub-sections, examples of each

configuration are given.

2.4.1 VLSI AI processors

One major trend in processor design is the emphasis on targeting at and support for high level languages. Under this trend, a whole new range of VLSI AI processors are being designed to facilitate AI programming. The design approach for these processors focuses on two methods:

i) design and implementation of special architectures to support AI language features efficiently. Some desirable AI architectures are suggested by Deering [32]; they are

- Tagged architectures to enable dynamic type checking at run time. Functions and data are intermixed, at the language level they are indistinct and known as symbols. At machine level, however, steps have to be taken to reveal their types prior to passing them into another function. Conventionally, this is performed sequentially and thus creates considerable delays. The concept of a tagged architecture is to sacrifice a few bits of a word for type identification. Parallelism is exploited by monitoring these bits with dedicated hardware.
- Generic machine instructions which make use of the results of dynamic type checking. Only one instruction is necessary for more than one type of data. The resultant signal from type checking is used to direct the execution unit to the appropriate microcode. This minimises the instruction set thus simplifying AI programming.
- Use of associative memory. This is particular essential for executing Prolog software to expedite the unification process. Instead of memory accessing as in conventional devices using addresses, associative memory is accessed by

content - Content Addressable Memory (CAM). The roles of the processor with the CAM are to read/write to/from the memory, to supply the required searching keys and to interact in case of multiple hits. All these can be achieved with a proper I/O interface.

- External communication is necessary for message passing and to interface with other devices - memory, similar processors, coprocessor systems, etc.

ii) adoption of conventional computer design techniques to increase processor throughput and to reduce design time. A widely used approach is to emulate a Reduced Instruction Set Computer (RISC) [33]. The salient features of the RISC are

- a simple instruction set with fixed length instructions reduces decoding time;
- extensive use of internal registers to reduce off-chip communication; in fact, the only instructions which communicate off-chip are LOAD and STORE;
- register windowing which permits internal registers to be employed for temporary storage (e.g. local variables);
- single cycle execution is possible because operands are internal;
- writable control store for directly compiled object code;
- pipeline architecture with delayed branches - the instruction after a branch is not executed until the branch destination is resolved.

2.4.1.1 Examples - SCHEME and SOAR

SCHEME81[34, 35] is a VLSI processor chip, designed at Massachusetts Institute of Technology (MIT), which endeavoured to map a LISP variant, SCHEME, onto silicon. Both direct interpretation and compilation of the language are supported. A two level microcode is used for program execution and garbage collection.

Garbage collection is based on the mark and sweep strategy. Special architectural features of SCHEME81 are assignment of particular registers to specified hardware functions, concurrent invocation of functional registers, dynamic type checking using extra tag bits to type each data item and support of multiple SCHEME environments.

The SOAR processor[36] is another example of a VLSI machine for AI applications. It was designed at the University of Berkeley to support the language Smalltalk on a RISC. The target of the design is to execute the language efficiently. This is done by concentrating on three areas :

i) Dynamic type checking allows instruction execution and type checking to be performed in parallel.

ii) A register windowing scheme (a RISC feature) is used for procedural calls.

iii) Hardware support for the garbage collection process [37].

Both SCHEME81 and SOAR have made valuable contributions to the evolution of VLSI AI processors. Texas Instruments Inc. (TI) is one of the pioneers of commercial VLSI AI systems. A LISPchip is expected to be launched in mid 1986 [38]. It is implemented in sub-2-μ CMOS technology and contains half a million transistors. The internal clock speed of the chip is 40 MHz with a power consumption of one watt. The hardware characteristics of the chip are : 1K internal word general purpose scratch pad memory, a push down list buffer and 100K bits of RAM. The design is based on the TI's Explorer LISPMachine [39]. It is claimed that the LISP chip's performance is five times better than the Explorer. The processor chip, at completion, will be used as a CPU for a complete workstation - Compact LISP Machine (CLM) (cf Section 2.4.2).

2.4.2 LSI Systems

LSI systems are machines with proper organisation using existing hardware. These machines are designed for specific languages and based on two approaches :

i) use of conventional processors operating with a fast clock rate; and

ii) use of customised VLSI processors (Section 2.4.1).

AI workstations are grouped under this category. They are self-contained and mainly designed for debugging, testing and development of AI software. There are growing numbers of commercial AI workstations. Some examples are :

• *AI workstations based on conventional processors.*

The Tektronix 4400 series LISP machines are constructed using conventional Motorola 68010 and 68020 microprocessors. The performance of LISP (both Franz and Common) on these machines is between Franz LISP on a VAX and on a Symbolics 3600. The Tektronix target is the low-end of the AI market, to supply a range of low cost AI machines with reasonable system performance. There are basically three machines in the 4400 family : the 68010-based 4404, the 68020-based 4405 and the 4405 upgrade 4406 which is a 68020 based machine running with a 10MHz clock with additional floating point and graphics capabilities. [31]

The DEC AI VAXstation is based on the company's MICROVAX II central processor. The local LISP dialect is VAXLISP and the machine runs on a VMS host. Another AI software package supported is OPS5 a VAX/VMS version of OPS, an expert system shell. [40]

• *AI workstations based on VLSI processors.*

The LISP Machine is a high performance personal computer that is microcoded to handle LISP operations, such as list manipulation and function calls, efficiently.

The idea of a LISP Machine originated from MIT where the first prototype was constructed [41]. An interesting feature of this machine is the existence of a two-bit CDR code for each list cell. The advantage of CDR coding [42, 43] is that it is possible to save a word of space in any cell in which the CDR (the tail element) is either an end or points to the subsequent cell. Commercial machines have been constructed [31] adopting MIT's design philosophy.

Symbolic Inc. delivered the 3600 computer initially in December 1982 [44] ; and most recently, an updated version 3610AE has been announced [45]. In contrast with its predecessor and other contemporary machines, the 3610AE is targeted not just at research and development but also at office and factory applications. Low-cost and small size are the key design issues. The machine is implemented with seven 2-μm CMOS gate arrays on one board. On top of this, the cabinet houses three other hardware boards: the input/output board, the console board and the Random Access Memory (RAM) board. Technically, there are two main attractive hardware features: tagged architecture and an "ephemeral garbage collector"[45] (a variant of Ungar's algorithm[37]). The local programming dialect is Symbolics LISP .

Under the sponsorship of the Defence Advanced Research Projects Agency (Darpa) in the USA, a design project for a Compact LISP Machine (CLM) [46] is scheduled by Texas Instruments Inc.. CLM is an updated TI's Explorer workstation and consists of four module types:

i) a 32 bit VLSI processor (c.f. Section 2.4.1);

ii) a memory array made up on 72 256×1 dynamic RAMs, giving a total capacity of two megabytes;

iii) a Multibus interface module; and

iv) a cache/mapper module which contains a high speed data cache to maximise processor to memory bandwidth and minimise bus traffic; and also, an address mapper which translates virtual addresses.

Again the main design goal is to inject AI technology into practical fields. The machine is expected to be fully operational by September 1986.

2.4.3 Coprocessor Systems

The design objective of the coprocessor approach is to transform conventional general purpose processors into reasonable AI machines. A coprocessor system incorporates special purpose hardware for dedicated functions. This approach is frequently adopted in memory management and arithmetic acceleration. Two examples of existing AI coprocessors are given in the following paragraphs.

In some AI systems, reasoning is achieved by using fuzzy logic. Fuzzy logic assigns degrees of certainty for a given decision. On these values logic operations are applied to derive a final conclusion. AT&T Bell laboratories have announced a fuzzy-logic inference engine on a single chip [47]. The chip is initially designed in 2.5 μm CMOS technology and in the future it will be refined to 1.25 μm. The chip is claimed to be able to process 16 inferences simultaneously and put out 80,000 inferences per second. The application in mind is for personal computer based robotic control.

A project of the Alvey programme has produced the Generic Associative Memory (GAM) chip [48]. The design and development are being carried out in Strathclyde University, Scotland. The function of GAM is to achieve content addressing, in which a symbol can be accessed by its attributes/properties. It is

intended to assist AI workers in vision processing and natural language systems. A prototype has been implemented using NMOS and it will be converted to high speed CMOS in the near future.

2.4.4 Non Von Neumann Architectures

Von Neumann machines are sequential and control driven - program execution is implicitly governed by a program counter. Conventional imperative languages [49], e.g. Fortran and Pascal, are designed to be used on these sequential machines. They are restrictive in that they provide minimal parallelism. Concurrent control flow structures are possible in some systems, such as the fork()/wait() function pair in C. Nevertheless, they still have to be called explicitly.

Special AI language features are unsuited to conventional architecture. A new generation of computer architectures is being considered to facilitate the application of AI languages. Demand flow, data flow[50] and logic flow are all potential architectures for the next generation of computers.

2.4.4.1 Demand Flow Architecture

Demand flow (or reduction) architecture is based on the recognition of reducible expressions and the transformation of these expressions. Program execution is achieved by traversing the program and replacing reducible expressions by others that have the same meaning; until an indivisible literal expression representing the result of the program is reached. Therefore, the execution sequence is governed by the reducibility of an expression.

Typical demand flow machines are based on distributed computing with multiple processing elements (PE). Each PE has its own memory, execution unit and communication interface module. Program execution is achieved by message

passing between PEs which are connected via the communication interface modules (c.f. the INMOS Transputer [51]). The connection network, i.e. topology, of PEs is process dependent. Popular topologies are tree structures† and vectors††.

The most well known project for demand flow architecture in Britain is ALICE [52] - Applicative Language Idealised Computing Engine - at Imperial College, London. Functionally, ALICE is a reduction machine featuring parallel language evaluation, packet graph representation and lazy evaluation. It is implemented with multiple INMOS Transputers. Very recently, extra government support has been approved to develop ALICE. A project, officially known as Flagship [53], is running jointly between ICL, Plessey and Imperial college. The objective is to construct a fifth generation computer based on ALICE using multiple INMOS Transputers.

In fact before ALICE, a similar philosophy was adopted at the University of Kent at Canterbury where a loosely coupled multiprocessor system, based on M6800 microprocessors - PLEAIDES [54], was implemented. The prototype verified the feasibility of distributed processing for knowledge manipulation.

2.4.4.2 Data Flow Architecture

The sequencing constraints of data flow architectures are tied to the flow of data. Instructions are typically packets consisting of slots for the required number of arguments, the destination addresses and instructions' identities. Destination addresses are specific argument slots of other instructions. During execution, instructions remain idle until the required arguments have arrived. Results are, at the end, placed in each destination address for further computation. Enhanced system

---

† PE are linked like binary trees; with each PE having a head and a tail neighbour.

†† PEs are connected in series.

throughput is achieved in two ways :

i) Instructions with independent arguments are executed in parallel with multiple processors.

ii) For a stream of instructions, whose argument dependence is accumulative, a pipeline configuration is adopted.

Commercial cascadable data flow processor chips are starting to appear. At present, their applications are mainly in number crunching applications, such as μPD7281 designed for image processing [55]. Their AI potential, however, remains [49].

Basically, a data flow machine consists of three functional units : a matching unit, a fetch/update unit and an execution unit. Operation is based on token passing and packet communication. Initially, the matching unit ensures that the required arguments have arrived for an instruction. When matched, a token is passed down stream to activate the fetch/update unit. The latter sends a packet to the appropriate processing element, within the execution unit, which executes the instruction and returns the results to the destination addresses through the matching unit where the computation cycle resumes.

2.4.4.3 Logic Flow Architecture

Another possible candidate for the architecture of fifth generation computers is logic flow (or pattern driven). The concept is derived from

```
IF <condition_1> AND
  <condition_2> AND
      .
      .
      .
  <condition_n>

THEN <execution>
```

control structures. An execution is fired only when the conditions (or required patterns) are satisfied (matched). This forms the basis for Prolog machines.

2.4.4.4 An AI Machine of the Future

Computers with novel non Von Neumann architectures are most suitable for AI programming. These machines are tailored to support AI languages best; contrasting with classical imperative languages which are designed to suit the underlying sequential architecture.

Novel architectures, however, are less general purpose. They are designed to fit a particular class of language or application. The reduction machine efficiently executes functional/applicative languages (e.g. LISP), the demand flow architecture is best suited to single assignment languages (e.g. LUCID) and logic flow architecture is designed to facilitate logic programming (e.g. Prolog).

In practice, problems frequently arise which are best tackled by a combination of the above language types. It is conceivable that a distributed computing system is the best solution. The computing system could consist of a top level controller and a bank of heterogeneous execution units. The controller could monitor the incoming instructions and allocate them to the appropriate execution element.

2.5 Hardware Complexity

At present, sequential machines are firmly established in engineering; most

working systems are based on this approach. On the other hand, the state of technology for novel architectures is still in its development stage. Until the technology of novel machines is fully developed and their concepts are well understood, Von Neumann architecture based machines will persist for engineering applications.

Two different design approaches exist for AI systems based on the Von Neumann architecture.

1) VLSI AI processors and LSI systems both consist of customised standalone hardware. Low power consumption, increasing bandwidth and compactness, together with decreasing design and fabrication costs have made the VLSI design approach most promising for the future. Nevertheless, as for novel architectures, VLSI AI processors are still in their infancy. More experience has to be gained, before they would be accepted by engineers. LSI systems are based on the proper configuration of existing technologies. These systems are meant to be design aids for AI software programmers, but they are too bulky to be used for many engineering applications.

2) Coprocessor systems are hardware translations of frequently used software functions. By interfacing these to existing Von Neumann computers, system performance is improved. For commercial reasons, vendors often produce coprocessor systems compatible with their own range of computers. Engineers who wish to employ specific coprocessors would have to install the complete computer system. In addition, the running and replacement costs of the systems could be very high.

Summarising, there are two shortcomings in existing AI technology which have been hindering engineers from using them in practical real-time applications. They are identified as :

- Garbage Collection which causes variable system interruption;
- the bulk and complexity of existing AI hardware.

The aim of this project is to provide a simple yet efficient solution to these problems using existing standard hardware. Before proceeding any further, the garbage collection problem must be studied in some depth with emphasis on its significance for real-time engineering applications. This is considered further in the next chapter.

	LISP	PROLOG	FORTRAN	C	BASIC
User Defined Data Structures	√	√	x	√	x
Procedures	√	√	√	√	x
Recursion	√	√	x	√	x
Symbol Manipulation	√	√	x	x	x
List Processing	√	√	x	x	x
Dynamic Memory Allocation	√	√	x	x	x
Pattern Matching	x	√	x	x	x
Function as Data	√	√	x	√	x
Function/Data Intermix	√	x	x	x	x

Table 2.2: Comparison of Features of Various Languages [24].

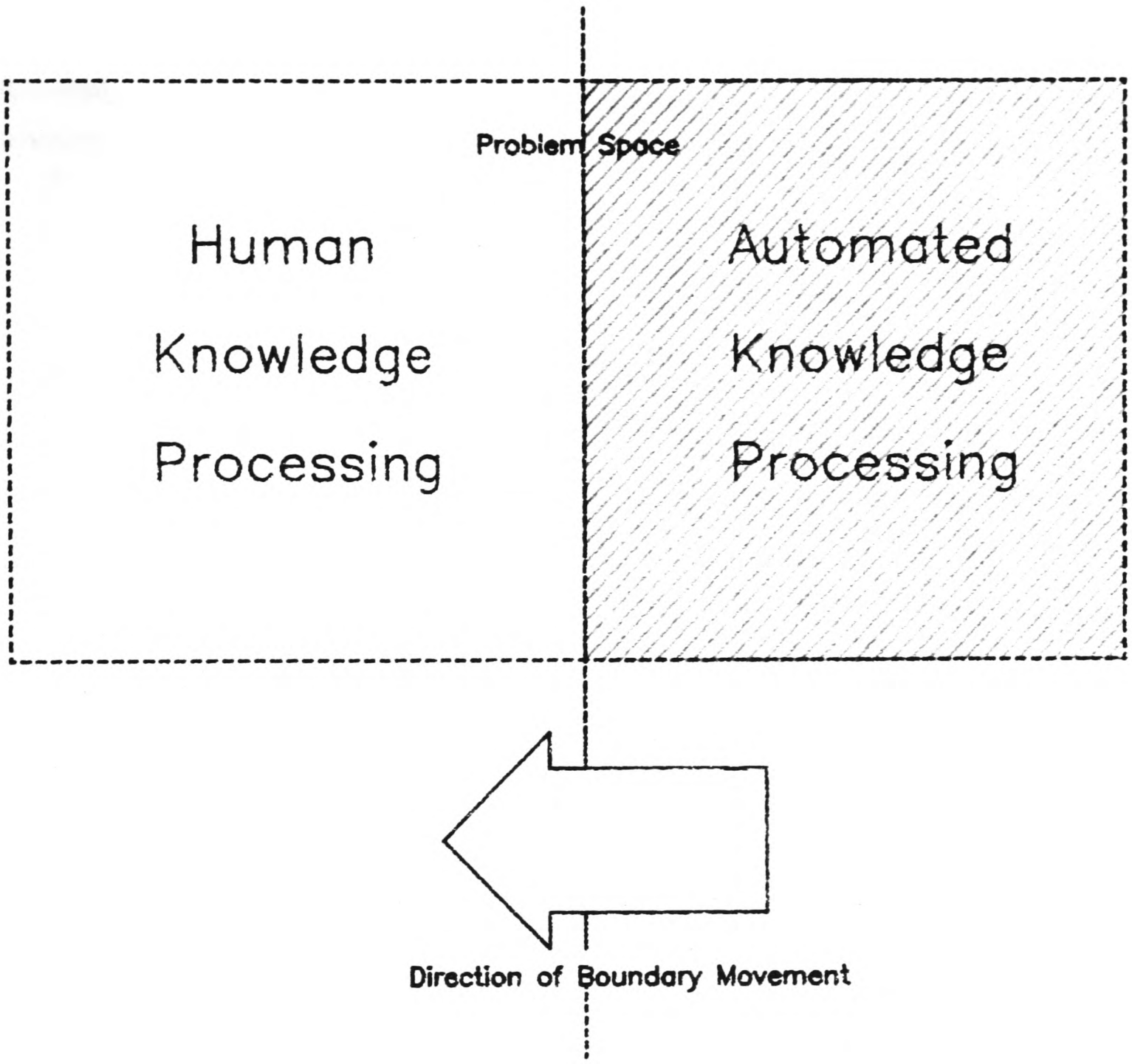


Figure 2.1: Alty's Automation Model [3].

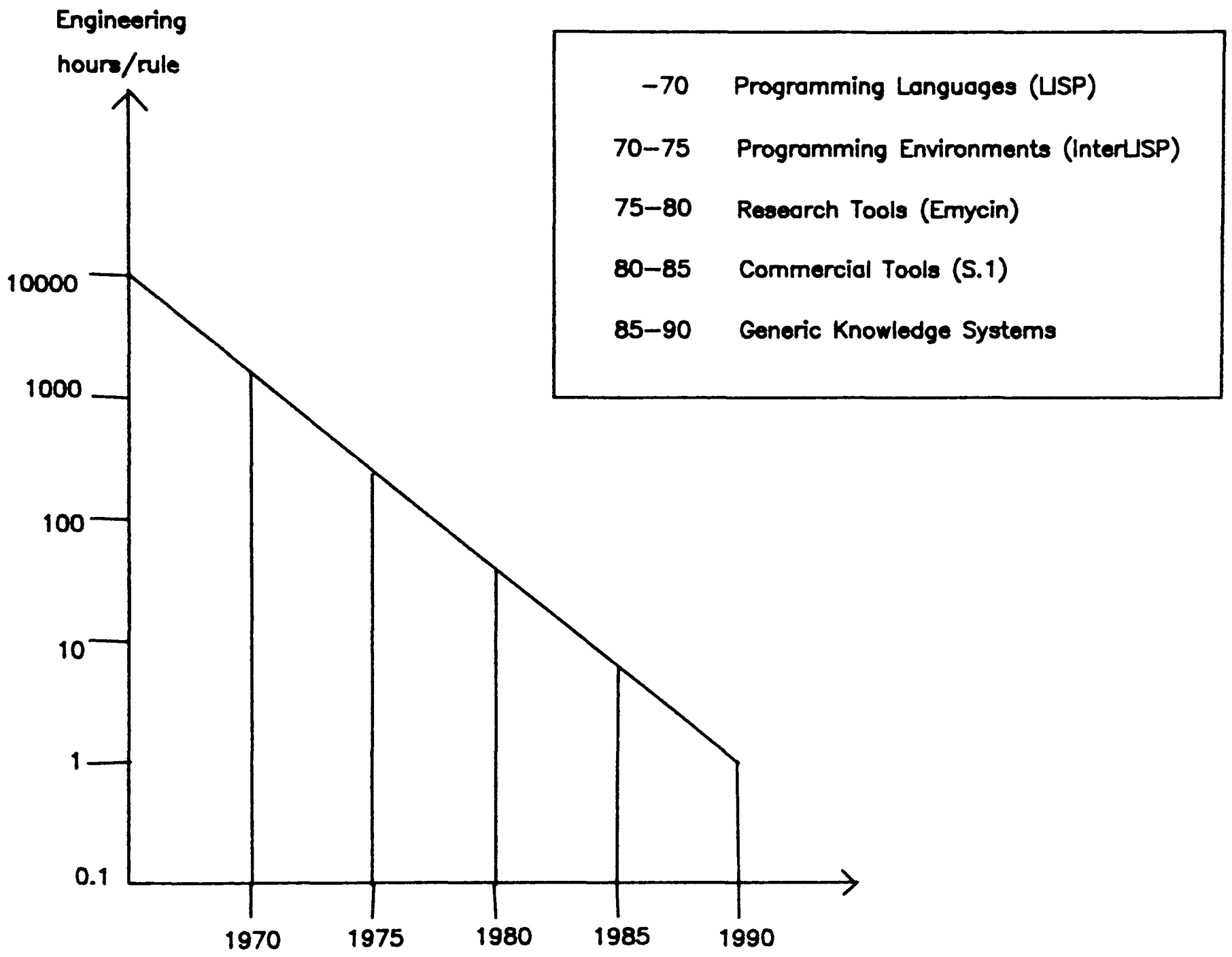


Figure 2.2: The Technological Advancement of AI Software [30].

CHAPTER III
Garbage Collection

Poor real-time response has made AI systems unsuitable for many engineering applications. This is caused by the unpredictable and variable interruptions incurred by the *garbage collection* process. This was identified in Section 2.3. The aim of this project is to provide a simple yet efficient solution to improve an AI system's real-time performance. In this chapter, an overview of classical garbage collection strategies is presented. Their relative merits are considered and from time distribution graphs, desirable features of a practical algorithm suitable for real time applications are identified.

3.1 Background

Variables in AI programming are represented as Symbols. Each symbol has properties and/or associations which are represented by list data structures. Execution of AI software is extensively based on list manipulation. The elementary storage unit of lists is a *cell*. A cell, figure 3.1a, consists of a HEAD, a TAIL and several TAG bits. HEADs and TAILs may contain either pointers to other cells (cell pointers) or *atoms*. The amount of storage required for a HEAD/TAIL is machine dependent. Literally, an atom is an indivisible entity which can either be a number, a word or a NIL, a special atom which marks an end of a list. TAG bits serve various purposes depending on the application. They can contain information for type identification, status of garbage collection (mark and relocate details, c.f. Section 3.2), compaction (CDR-coding), capabilities and "age", of a cell. Using cells as building blocks complicated information frameworks, such as lists, can be con-

structed. For example, LISP [25] a widely used AI language is designed for LISt Processing. A list is a data structure, widely employed in AI, which enables association of properties and values with symbols. The syntax definition of a list and an example of a symbol with its associated list structure are shown in figures 3.1b and 3.1c,

During the course of reasoning cells are dynamically being "created". These cells are used to transport ideas and "thoughts" and they become useless after their roles have been fulfilled. If unwanted cells are not released the finite physical storage of the system would be exhausted. These useless cells are called garbage. The process of *Garbage Collection* (GC) is to identify garbage cells and recycle them for future usage. The rest of this chapter is devoted to providing a survey of garbage collection methods.

3.1.1 A Theoretical Machine

A theoretical machine can be used to assist understanding of the various garbage collection schemes. It is depicted in figure 3.1.1. It consists of two parts : the processor (P) and the cell memory (CM). Within P, there are several internal registers, called roots (R0,R1,....Rn). Each root is a pointer to the origin (the first cell) of a list in the CM. For instance, it may point at a property list, an association list, a local environment, etc. The CM is a reservoir of cells. Graphically, a cell is labelled (00,01,02...) and it consists of two outgoing arcs - the HEAD (left) and TAIL (right). Above the cell label, tag bits are situated. Dynamic memory allocation is performed upon these cells. Garbage cells are created by pointer redirection or root redirection. In either case, the list which was previously linked to the pointer or root is no longer accessible by the system (assuming it was not pointed to by any other pointers or

roots). By definition, the constituent cells of an inaccessible list are garbage - the target of the GC process.

3.2 Classical Algorithms

The idea of garbage collection is closely associated with AI programming. In conventional programming languages, dynamic memory allocation is achieved by explicit function calls at run time; such as, alloc() in C and new() in PASCAL. Users are responsible for ensuring redundant memory is returned to the operating system by subsequent function calls, e.g. free() and dispose() for C and PASCAL, respectively. The lack of full automation and user transparency in memory allocation are key factors rendering imperative languages unsuitable for AI applications.

Garbage collection has always been one of the active research topics in AI studies. Knuth[56] and Standish[57] are excellent introductory sources and in Cohen[58] a thorough overview of existing theory and bibliography on the subject is provided. Three of the classical algorithms are described below.

3.2.1 Reference Count Method

In this case, the TAG bits of a cell contain a count of the number of cell pointers which are pointing to it at any instant. When a link is created between two cells, the TAG of the pointer is incremented. Conversely, the TAG is decremented when a cell is discarded. There is a special list, the FREE_LIST, which chains together all unused cells in the system. The process of allocation extracts a fresh cell from the FREE_LIST. When the TAG count is zero, implying a cell is not associated with any lists, it is then returned to the FREE_LIST.

Referring to example GC1, figure 3.2.1, originally, a list is composed of cells 00, 05 and 09 whose root is R0 and the rest of the cells are chained together in the

FREE_LIST. At some time, cell 09 is dissociated (the dotted arc) and its TAG count falls to zero. Subsequently, this cell will be placed at the end of FREE_LIST and pointed to by 03.

Because of its simplicity, the reference count approach is a favourite solution for many systems. Nevertheless, the major disadvantage of this method is that it cannot cater for recursive or self-referential lists. Recursive lists are widely used in practice (e.g. iteration). They are formed by cells pointing backward to their fellow constituent cells of the same list, forming entangling circular structures. In such cases, the reference counts can never drop to zero.† To be able to recognise and recycle recursive lists, practical systems include a second garbage collection scheme - mark/sweep or copying. Normally, cells are collected "on-the-spot", immediately after they are discarded. In addition, when the population of the FREE_LIST falls below a preset level, the second scheme is invoked.

3.2.2 Mark and Sweep Method

When there are no more free cells, i.e. the FREE_LIST is empty, the process of GC is invoked. GC is performed in two phases :

i) Marking - initially, all roots are marked by setting a TAG bit. Traversing all lists reachable from the roots, constituent cells are marked. The mark phase is terminated when all accessible cells are marked.

ii) Sweeping - unmarked cells are identified, reclaimed to the FREE_LIST and finally, marked cells are unmarked preparing them for the next cycle.

Example GC2 (figure 3.2.2) shows snap shots just at the end of the two phases.

There are various ways to realise the mark and sweep algorithm [58]. Effectively, they are all identical in function and differ only in the marking schemes

† Example: The list formed by cells 02 and 08 in figure 3.2.1.

used. In large systems which support virtual memory, two more phases are included - Relocation and Update. These phases help to maintain cell locality thus increasing the hit ratio, which in turn reduces the rate of fragmentation and thrashing. Generally, the overall collection time (G) is assumed to be $\alpha N + \beta M$, where N is the number of useful cells, M is the total number of cells, and α and β are constants. α is dependent on the time required to mark a cell and subsequently unmark it in the sweep phase and β is dependent on the time to put an inaccessible cell into the FREE_LIST. In practice, α is much bigger than β, thus $G = \alpha N$ is a close approximation. This implies that as the number of useful cells increases, the systems' performance will be degraded by the frequent and time-consuming GC process. Moreover, another problem of memory utilisation results from the large amount of memory which has to be reserved for stacking during the course of list traversal. In the worst case the stack may amount to M words.

The simplest form of mark and sweep collector is the one proposed by Winston [59]. This algorithm traces the complete list structure by first marking the tail branches until it comes to a halt by an atom or a marked node; then it resumes tracing the heads from where it originally branched off. The system stack is extensively used for storing return addresses and each node is visited twice: once before marking the tail and once before marking the head. The major drawback of this type of collector is that it uses extra storage for stacking return addresses. In the extreme case, if a list space of N list cells has to be completely marked, the size of the stack required would be that of the list space itself (i.e. N). Provision of such a huge additional amount of memory can greatly reduce the effectiveness of garbage collection.

Deutsch, Schorr and Waite [60] had independently designed a similar marking algorithm without incorporating a stack. This algorithm was later refined by Knuth

(algorithm E in [56]). The main idea of this method is that the nodes of a tree can be inspected by reversing successive links, using three pointers (p,q,t), until leaves (i.e. atoms) or already visited nodes are found. The link reversal can then be undone to restore the original structure of the tree. Initially, p is set to point at the root of the tree, q is set to point at the contents of the head of this root and t is set to NIL (a special LISP atom). The first cell is marked, t is placed in the head of the cell: this acts as the termination marker during the reverse trace; and p is set equal to q. This algorithm then moves to the next cell by setting q to the head of p and t equal to p. When the bottom of the tree is reached the same procedure is carried out in reverse until the NIL termination marker at the root cell is encountered. At any node, the program has a choice of two routes: either via the head or the tail. To distinguish the routes that it has taken, the numeric atom bit is used. On branching from the head, the numeric atom bit is set. In this way, all the numeric atom bits will have been set on tracing the head links and these must be cleared during the reversal. Each time the head of any node has been traced, the tail of that node is immediately traced by the same method.

Although, the link-reversal algorithm has been widely accepted as a standard mark/sweep garbage collector, it is by no means ideal. There are two main identifiable drawbacks:

D1 Each node is being visited three times and extra effort is required to set the numeric atom bit, thus rendering the technique less efficient.

D2 Because contents of nodes are modified during link-reversal, it is strictly required that garbage collection is completed before memory accessing can

resume. This is annoying and is most apparent in multi-user systems where users may experience interruptions lasting minutes. In extreme cases, successive collections may take place with little actual program execution between them, making continued computation impractical.

To overcome the second drawback (D2), Dijkstra et al [61] and Kung et al [62] have separately come up with a parallel mark and sweep scheme. Both schemes use two processors, one known as the mutator and the other the collector. The collection algorithm used is called *shading* or *colouring* which requires two extra bits per cell. The information stored in those bits may be thought of as representing three colours white, grey and black†. Initially, all cells are white and marking begins by stopping all processes and shading the ROOT nodes grey. Grey nodes are shaded by performing the two following indivisible operations. Firstly, the offspring are shaded grey and second, the already grey parent cell is shaded black. After the marking phase has begun, any processes which have pointers to at least one black node may be restarted. When the marking phase can find no more grey cells, the mark phase has ended and the collection of all white cells still left in the system can begin. Note that it is illegal for black cells to point to white ones. Therefore should a running process attempt to perform an operation which would violate this principle, it must first shade the offending white node grey. After the collection process has been completed the interpretation of the colours is reversed, thus saving the system from having to clear the mark bits of all white cells before the next cycle begins.

† Algorithm designed by Kung [62] based on four colours. Besides white, grey and black an additional colour 'off-white' was defined which is possessed by all free cells.

3.2.3 Copying Method

No FREE_LIST is required with copying type collectors. Memory is divided into two semispaces: the TOSPACE and FROMSPACE [63]. List manipulation and memory allocation always operate on the TOSPACE. A special pointer (B) marks the next available free cell. When the content of B is NIL, free cells have been exhausted thus GC is automatically initiated. Firstly, the two semispaces are flipped - the previous FROMSPACE becomes the present TOSPACE and the previous TOSPACE becomes the present FROMSPACE. Cells accessible from all the roots are useful by definition, and they get copied into the present TOSPACE. GC is finished when all accessible cells are copied over.

Figure 3.2.3 is an example of a copying collector. There are a few notable differences from the previous examples:

- Extra storage space is required†.
- Free cells are allocated to incremental addresses.
- The order of a list is altered although its information is retained.

The copying scheme moves reachable objects from OLD to NEW and ignores the others. The collection time is simply given by γN; where γ is a constant related to the time for testing a cell, deciding that it resides in the FROMSPACE and later moving it into the TOSPACE. In common with the schemes already described, when the number of reachable cells has grown substantially, the copying scheme takes embarrassingly long for collection. Inevitably, this is usually the case because intelligent systems are constantly being 'educated' or updated with new knowledge. To overcome this drawback, researchers [37,64] have further amended this scheme

† Compaction techniques [43] have been designed to reduce the amount of storage required.

by differentiating between reachable cells on the basis of their period of existence. By doing so, the longer a cell has existed the less frequently it is subjected to garbage collection. In particular, once a cell has persisted for several cycles of garbage collection, it is considered old and archived in a permanent region [37].

The idea of copying type garbage collection was first suggested by Fenichel [65] and Cheney [66]. The most promising algorithm which was designed to overcome the previously stated drawbacks (D1,D2 of the mark and sweep method) is the one originally used by Fenichel and Yochelson in an early Multics LISP [65], elegantly refined in [66], and applied by Arnborg to SIMULA [67] (a LISP system). The principle is as previously described, i.e. it bisects the list space into two semispaces; and the moving algorithm works as follows :

During the execution of the user program, all list cells are located in the FROMSPACE. When garbage collection is invoked, all accessible cells are traced, and instead of simply being marked, they are moved to the other semispace. A forwarding address is left at the old location, and whenever an edge is traced which points to a cell containing a forwarding address, the edge is updated to reflect the move. The end of tracing occurs when all accessible cells have been moved into the TOSPACE and all edges have been updated. Since the TOSPACE now contains only accessible cells and the FROMSPACE contains only garbage, the collection is done and the program execution can resume with cell allocation proceeding in the former FROMSPACE.

This method is simple and elegant because :

- it requires only one pass instead of two as in the mark/sweep collector;
- it requires no collector stack.

The stack is avoided through the use of two pointers, B and S. B points to the first free word (the bottom) of the free area, which is always in TOSPACE. B is incremented by the procedure COPY, which transfers old cells from FROMSPACE to the bottom of the free area, and by CONS which allocates new cells. S scans the cells in the TOSPACE which have been moved, and updates them by moving the cells they point to. S is initialised to point to the beginning of TOSPACE at every flip of the semispaces and is incremented when the cell it points to has been updated. At all times, then, the cells between S and B have been moved, but their heads and tails have not been updated. Thus when S=B all accessible cells have been moved into TOSPACE and their outgoing pointers have been updated. This method of pointer updating is equivalent to using a queue instead of a stack for marking, and therefore traces a spanning tree of the accessible cells in breadth first order.

Although no collector stack is required, the efficiency of memory usage is only ½ since 2N cells are required for only N useful cells. Because of this storage inefficiency, at one stage, copying collection had nearly disappeared from the research scene. Baker's [63] analysis of his own algorithm has proved that the storage utilisation of copying collectors compares rather favourably with other algorithms, especially when compaction (e.g CDR coding) techniques are employed. This can reduce the amount of memory required considerably. Moreover, Baker has also proposed an incremental version of his algorithm which allows garbage collection to be performed in 'real-time' even on a serial computer. The principle is to interleave garbage collection with list processing.

Today, the design of novel garbage collectors is almost entirely Baker-oriented. One most promising implementation is the idea proposed by Liberman [64] : the *Generation Garbage Collector*. Studies have revealed that many

temporary cells exist in AI programs and they generally have short lifetimes. Exploiting this observation, Liberman's algorithm segregates cells into generations. Physically, a generation is a small region of memory the size of which is a crucial design factor. Each generation has its own pair of semispaces. Generations can be individually collected without disturbing older ones; thus younger generations are collected more often. This greatly reduces the time spent collecting older, more permanent cells. Moreover, the processing of garbage collection of one generation does not inhibit the instantiation of the others. Several individual generation collections can occur at different times creating a wave of garbage collecting processes.

Liberman drew an analogy of the generation collection scheme with 'renting/buying'. His algorithm ".. can be thought of as 'renting' memory space where the storage management cost for an object is proportional to the time during which the object is used. Traditional methods are more like 'buying' memory space, since the cost for an object is paid once and is always the same, regardless of how much the object is used. When large numbers of objects are used, although each object may be used only for a short period of time, the renting strategy will cost less overall than the buying strategy." [64] The simplicity and effectiveness of this algorithm has made it widely accepted. This algorithm has formed the basis of the garbage collector in the MIT LISP machines and also its commercial offsprings e.g. SYMBOLICS 3600.

3.2.4 Discussion

Despite its inability to identify recursive list structures, the reference count method is used in small systems because of its ease of implementation. For large practical systems, both the copying and mark and sweep collection schemes are

widely used. Each of them have many variants [58] which have their individual pros and cons. No formal comparison between the performance of the two methods has ever been published. The mark and sweep scheme is slightly more complicated and takes a longer time for collection. This is due to the additional effort required to maintain an explicit stack. In larger systems, which incorporate relocation and update phases, the situation becomes worse. On the other hand, copying schemes inter-mingle these phases and employ an implicit 'stack' (the FROMSPACE - functionally, it is in fact a queue). By doing so, the collection time is generally shorter. The penalty paid is that a larger amount of memory is required - normally twice that of the original cell space. Paradoxically, it is shortage of memory which demands garbage collection in the first place! Data compaction techniques are frequently employed to alleviate this predicament. For example, Baker [63] has used cdr-coding with a copying collector and the storage overhead has been verified to be minimal. Recent work on GC on the basis of "objects' life times" [64] was directed towards reducing storage overheads. By archiving permanent cells, the cell memory is less congested - thus leaving the list processor more "room to breathe" and invoking GC less frequently.

3.3 Garbage Collection in the Time Domain

In engineering design, timing is frequently a crucial design factor. Before selecting the most suitable collection scheme, it is instructive to study the time distribution graphs, shown in figure 3.3. They reveal the distribution of time on a machine shared between GC and list processing. These graphs are produced under the following assumptions:

1 When a system has been operating for a long period of time, it is safe to

assume that the average rate at which new cells are added to a list and the average rate at which cells are released will be equal [68]. Under this condition, the system is in a steady state.

2 For simplicity, it is further assumed that the system is continuously requesting new cells (CONS).

In terms of time, garbage collection schemes can be classified into the following three categories:

3.3.1 Non-Real-Time

The process of GC is completely independent from main program execution. When the memory space is exhausted, the main program has to be suspended until garbage collection has taken place. The time distribution graph of this class of collector is illustrated in figure 3.3a. As shown, the garbage collection time (G) follows immediately after the list processing time (A). In LISP terminology, A is the total time for CONS operations (assumption 2) before the next memory exhaustion. The fraction of useful work done (FUWD) is defined as the ratio of the processing time spent on memory allocation to the overall machine time (M), i.e.

$$\text{FUWD} \equiv \frac{\text{total allocation time}}{\text{overall machine time}} = \frac{A}{M} \qquad (3.3.1.1)$$

In the steady state (assumption A1), A is equal to G†, thus

$$\text{FUWD} = \frac{A}{M} = \frac{A}{A+G} = \frac{A}{2A} = \frac{1}{2} \qquad (3.3.1.2)$$

Both the copying and mark and sweep schemes have non-real-time variants. Whichever algorithm is adopted, the main program will have to be stopped during garbage collection and for large systems (i.e. with a large N) the time taken for this activity might well be hours. In most engineering applications, such long interruptions are

† Assume that the time to create a cell equals the time to recover a cell.

intolerable.

3.3.2 Real-Time

Garbage collection schemes in this category mostly work in multiprocessor environments under mark and sweep strategies. The simplest configuration is a dual processor system in which one processor is responsible for list processing (the *mutator* [61]) and the other dedicated solely to garbage collection (the *collector*). The two processors are running in parallel and theoretically, with no interaction. In practice, the mutator is occasionally suspended depending on what state the collector is in when the former makes a request for a new cell. For example in [61], when the free cell list is exhausted and the collector is still in the mark phase, the mutator will be forced to wait until the first free cell becomes available. The exact time of the pause varies depending upon how deep into active list space the collector has traversed.

Figure 3.3c depicts this category in the time domain. Parallelism is achieved in the overlapped portions of G and A. Ultimately, the goal of the design of this type of collector system is to maximise the G/A overlap ratio. The aim is to achieve the perfect situation of hiding G completely under A; thus making the collection process totally transparent with respect to the mutator. Also notice that due to overlapping, M is greatly reduced. There is occasional separation between G and A, in the portions marked t_1 and t_2. They are the idle states of the mutator and their actual value varies, with the worst case approaching G. At steady state, A and G are equal as before, but M is no longer equal to their sum (usually much less), then the fraction of useful work done is :

$$\text{FUWD} = \frac{A}{M} \geq \frac{1}{2} \qquad (3.3.1.3)$$

A FUWD greater than a half implies the mutator is more efficient and so can do more work than in the non-real-time case.

The major difficulty in the implementing of this type of collector is the coordination between processors. Reported implementations [61, 69, 62, 70] have their own customised architectures with complicated protocols for traffic control which improve the overall performance, but make programming very complex. Such machines have been mainly used for research purposes and none have ever been implemented commercially.

3.3.3 Virtual-Real-Time

The real-time performance of collectors in this category lies in the middle of the previous two. In this case the collector deceives the list processor into believing that the collection is always finished whenever the latter tries to make an access into the cell space although in fact, garbage cells still exist in the system. This is achieved by breaking up the overall garbage collection process into short bursts. Each burst corresponds to the collection of one or more cells. Garbage collection is, effectively, interleaved with list processing. Technically, this is known as incremental or "on-the-fly" collection†. In terms of time, it is illustrated in figure 3.3b.

As before, A and G are assumed to be equal. Incremental collection divides G into small bursts, each with a small time interval g which is related to G as : $G = \kappa g$, where κ is the number of reachable cells collected in one burst. Similarly, allocation is done in bursts. The overall machine time remains unchanged as in the non-real-

† Reference count systems are analogous to this approach. De-referencing a cell and reclaiming it if its count drops to zero is equivalent to one collection burst. However, this time burst is variable and depends on the depth into a list that the cell is located. Inevitably, this results in the same unpredictability as a parallel collector.

time case. The only difference is simply the pattern of distribution. Not surprisingly, FUWD is the same as the non-real-time case.

Intuitively, this type of collector seems to offer few advantages. Nevertheless, it is widely adopted in many practical systems. (e.g. the current version of the M.I.T. LISP machine is based on [64]). Frequently, A.I. systems function in an interactive mode and interleaving garbage collection and list processing avoids intolerable pauses. Another less apparent feature is the consistency of the collection burst time g. This is an advantage over real-time collectors whose variable pauses could cause unpredictability in system response. On the other hand incremental collectors enforce a short but fixed upper bound on timing which ensures predictability thus making the system more applicable to engineering situations.

3.3.4 A Practical Choice for Engineering Applications

From the foregoing discussion, it can be concluded that a practical garbage collection system should exhibit the desirable properties of both the real-time and the virtual-real-time schemes. These are:

P1 FUWD must be greater than a half. This is achieved by maximising the G/M overlap ratio. [real-time]

P2 Since it is apparently impossible to establish complete G/M overlap, the separation times must be minimised and they must be consistent. [virtual-real-time]

P3 If a special system architecture is required, it must be simple, and ideally it must be transparent to the host processor. Implicitly, the communication link between the the mutator and the collector must not be over elaborate. [not real-time]

These characteristics are shown in figure 3.3d.

Garbage collection is an inevitable nuisance to engineering applications. It heavily degrades a system's real-time response. The first task of this project was to identify an appropriate garbage collection algorithm. Candidates should possess the desirable properties outlined and be implementable in such a way that they operate with minimum real-time overheads. In the next chapter, the selection of the suitable algorithm is explained.

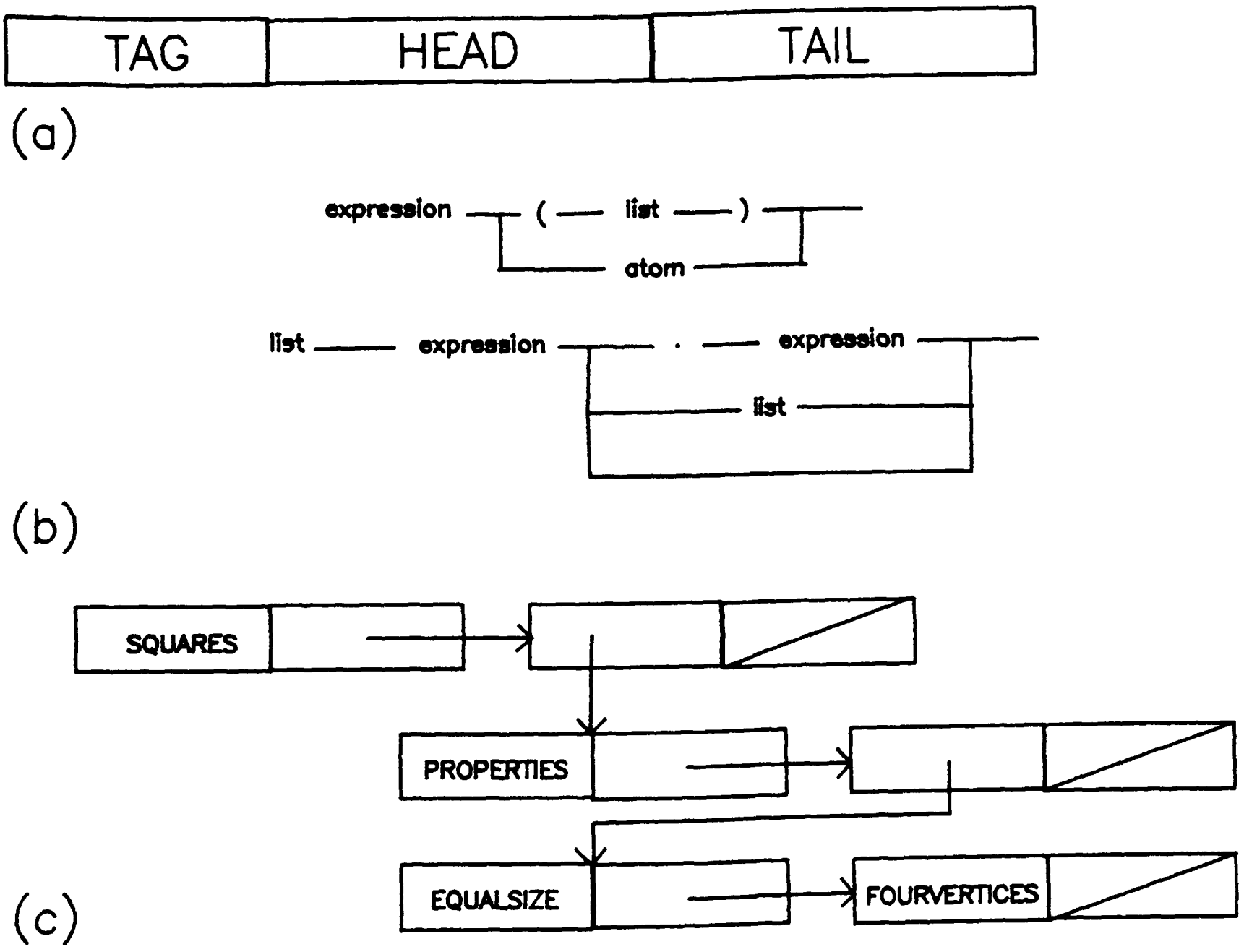


Figure 3.1:a) Definition of a List Cell.

b) Syntax Diagram of a List.

c) A Graphical Example of a List:

(SQUARE (PROPERTIES (EQUAL_SIZE FOUR_VERTICES))).

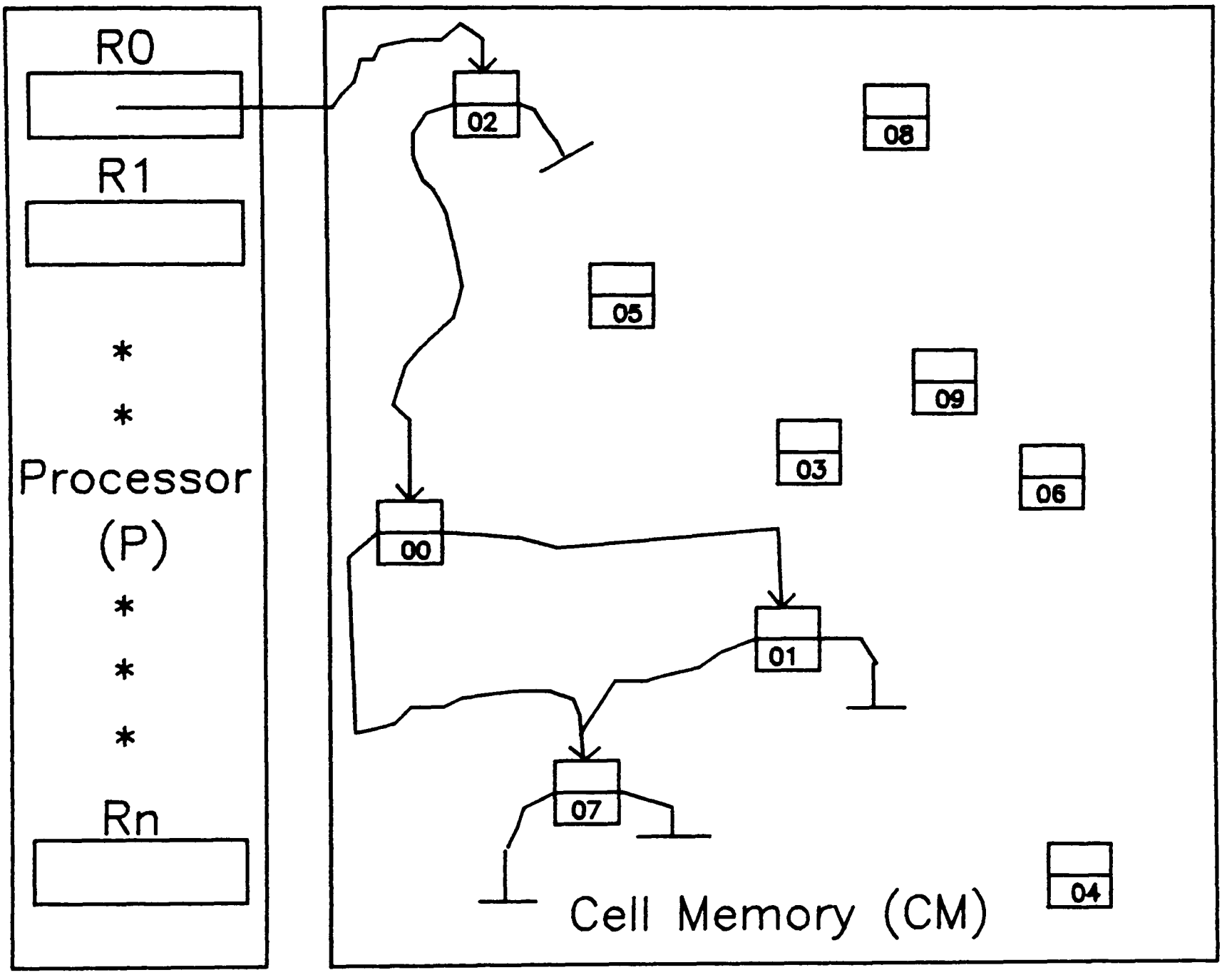


Figure 3.1.1: A Theoretical Machine.

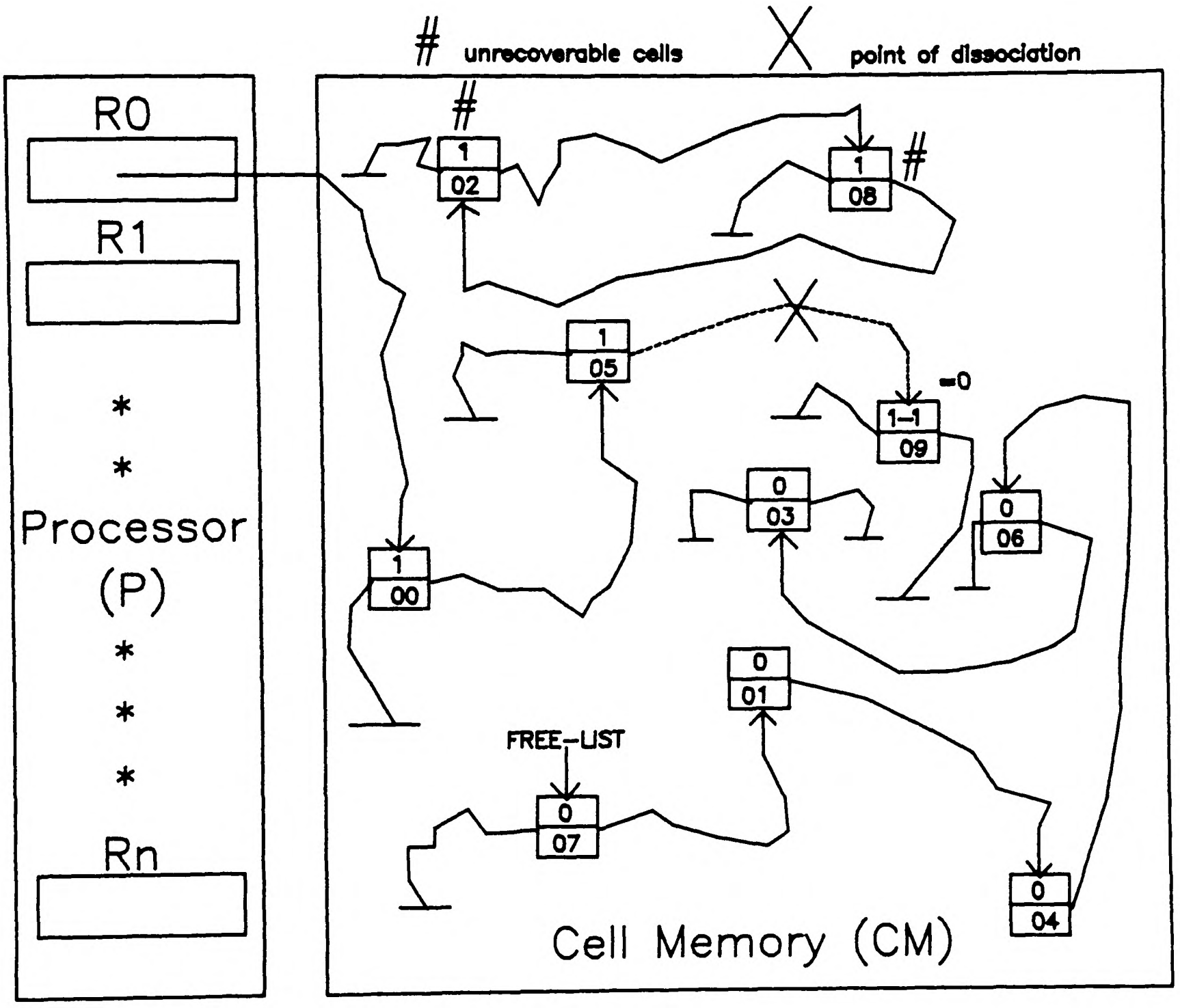


Figure 3.2.1: Example GC1 - Reference Count.

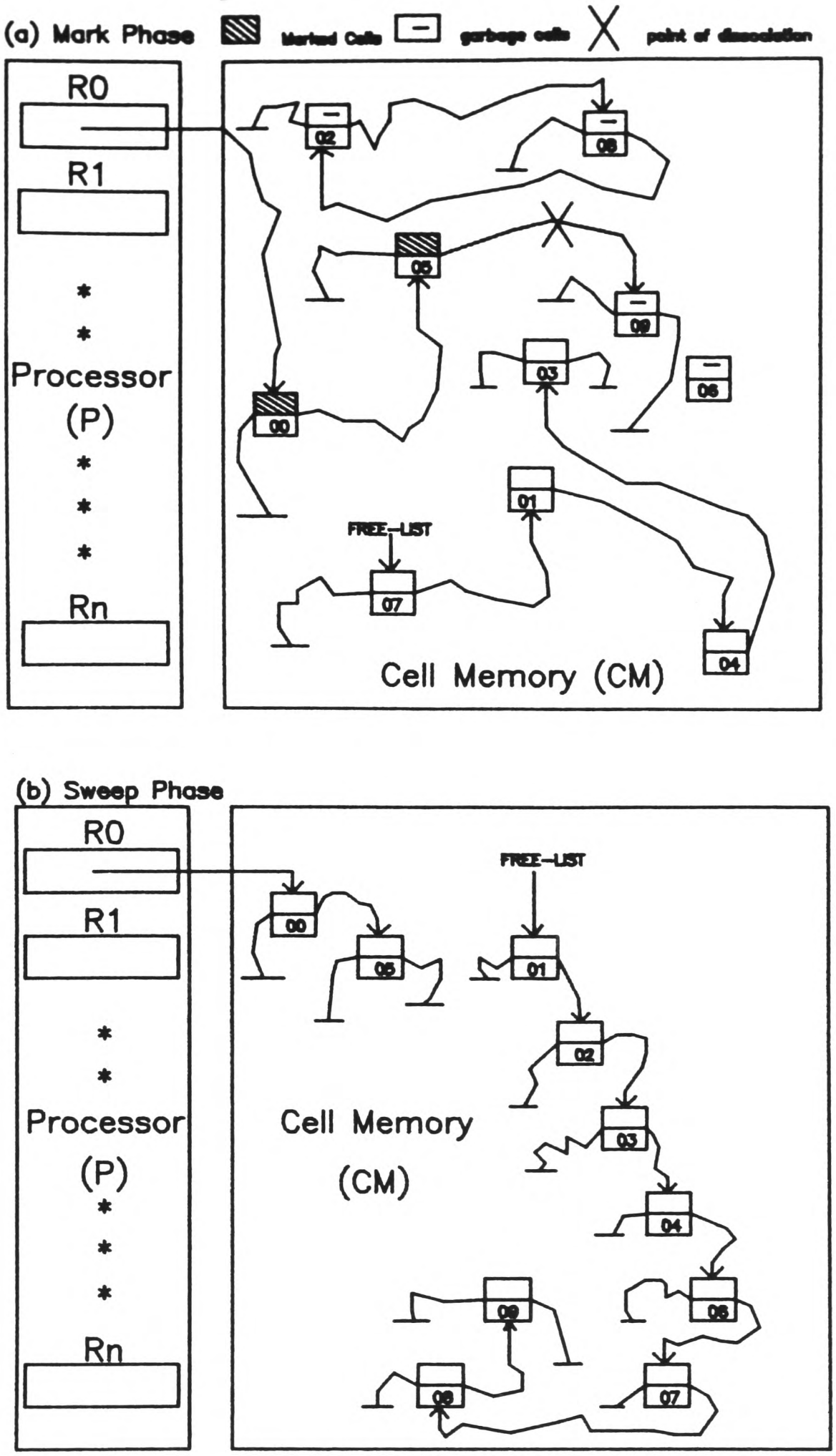


Figure 3.2.2: Example GC2 - Mark and Sweep : a) Mark phase, b) Sweep phase.

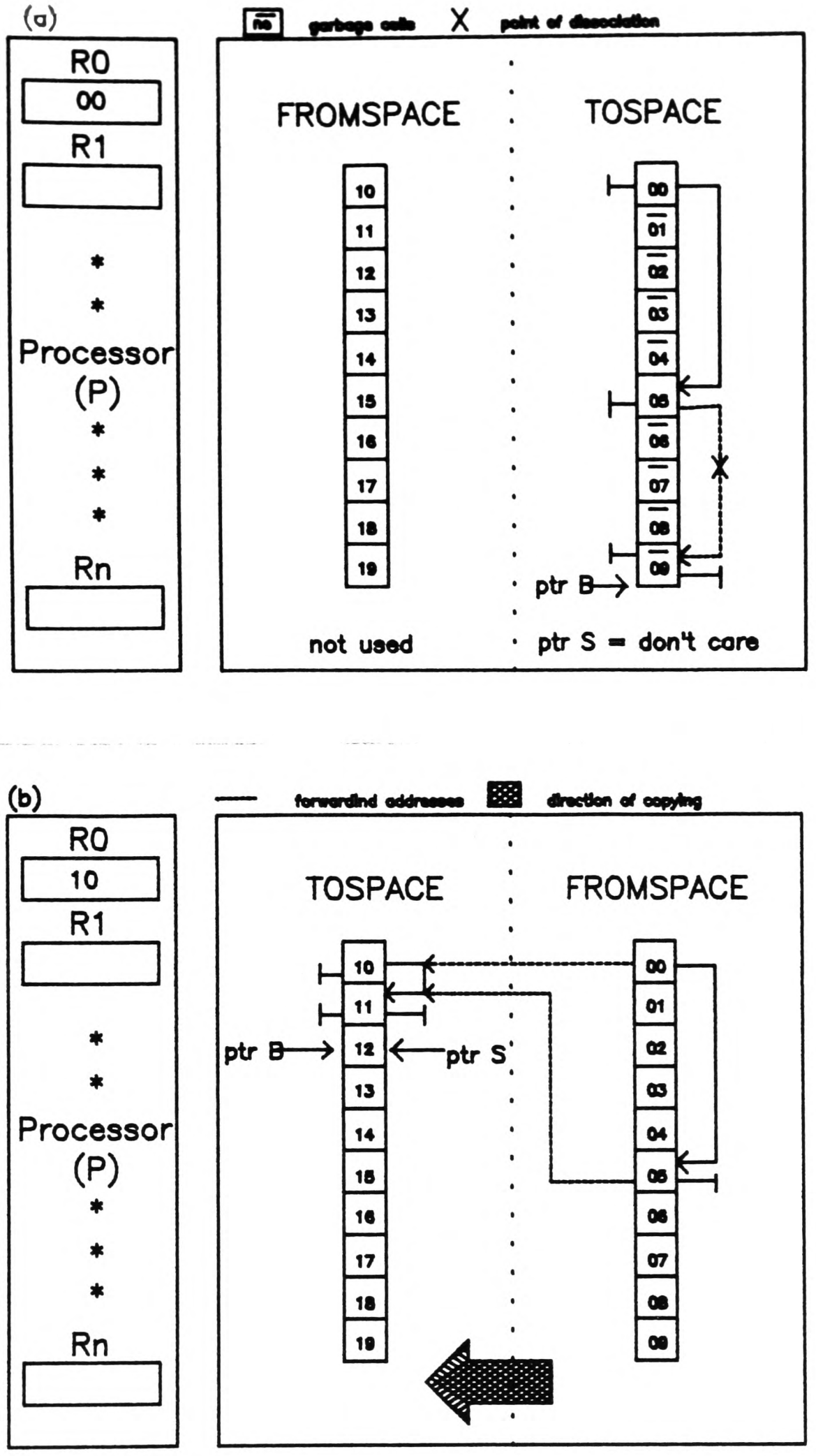


Figure 3.2.3:Example GC3 - Copying.

a) just after memory has exhausted;

b) just after GC.

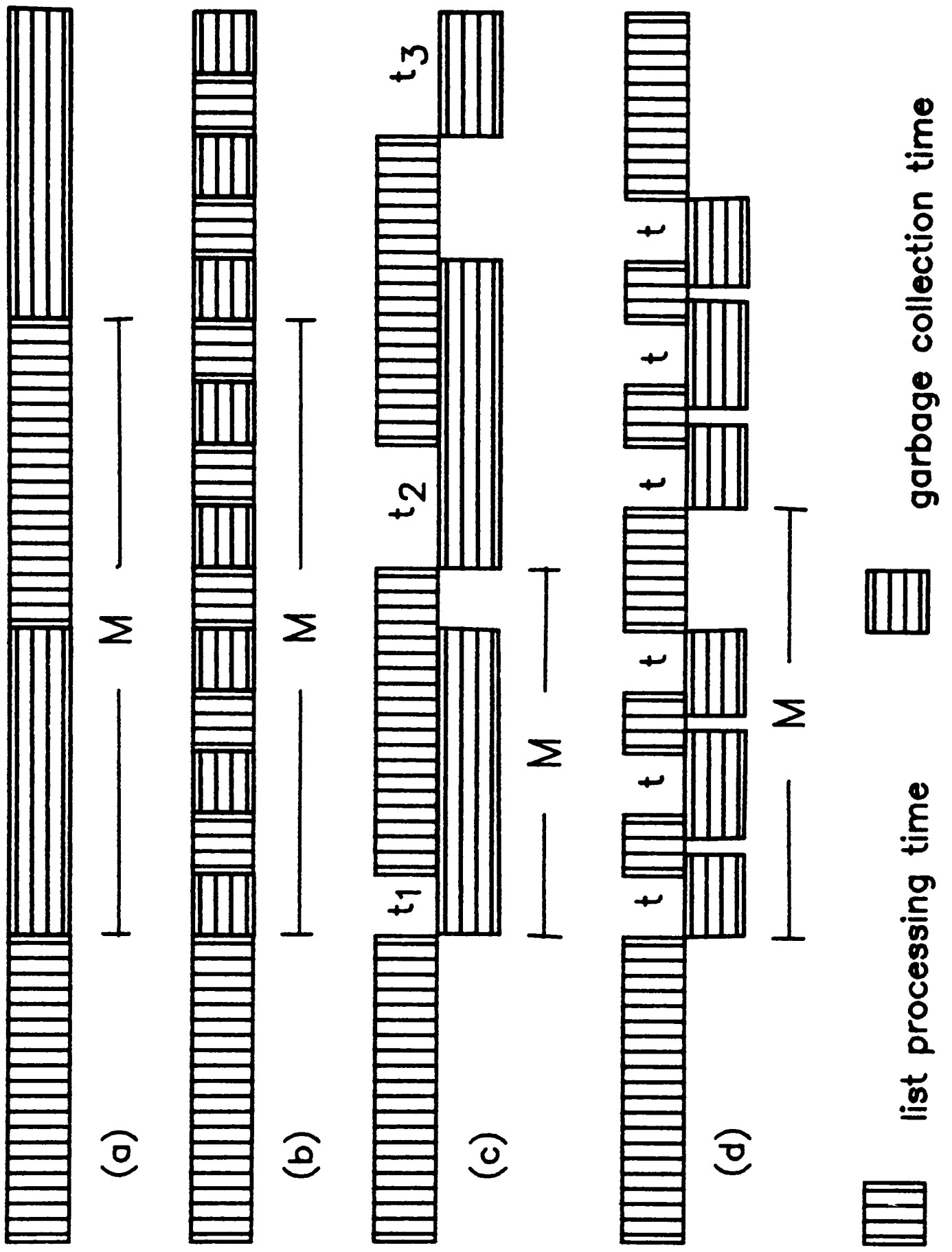


Figure 3.3:Time Distribution Graphs -

a) Non-Real-Time;

b) Virtual-Real-Time;

c) Real-Time; and

d) the ICMS (c.f. next Chapter).

CHAPTER IV
The Intelligent Cell Memory System

4.1 Introduction

Garbage Collection is a process to recycle deallocated memory cells. It is essential to all AI systems. Without it, systems would suffer from a shortage of memory and thus they would be impractical. Unfortunately, garbage collection is performed at the expense of degradation in the system's real-time performance. Host processing is frequently suspended waiting for garbage collection to complete. This feature of poor real-time response of garbage collection has rendered AI systems not suitable for engineering applications.

The aim of this project is to improve the real-time performance of AI systems so that they can be more applicable to engineering situations. In this chapter the design of a system, known as the Intelligent Cell Memory System (ICMS), which possesses the desirable properties outlined in chapter III is described.

4.2 The Design

Since garbage collection is practically unavoidable, the design phase of the project focused on deciding on a suitable algorithm and discovering a simple yet efficient way to implement it. The requirements for garbage collection in a real-time system have been identified in section 3.3.4.

The first requirement is *short intervals between processing* (P2 - Section 3.3.4), since they are inevitable. This narrows the scope of selection to *virtual-real-time* or *incremental* collectors only. Incremental algorithms are designed to divide garbage collection into small bursts and interleave them with host processing. Although the

reference count method would be the simplest to implement, it was rejected because

- it is useless in dealing with recursive list structures; and
- the time of collection is a variable which depends on how deep a node has penetrated within its "associated" lists.

The next choice was the colouring or shading algorithms proposed by Dijkstra et al [61] and Kung et al [62]. They are both mark and sweep collectors which could be organised to work incrementally and they offer a very fast response. However, they are not suitable for this project. Fast response time is achieved by a dual or multiple processors configuration. Reported configurations are complex - they are implemented as "one-off's" and are difficult to reproduce. Although average response time is fast, it is inconsistent. For instance, in Dijkstra's algorithm all processes are forced idle while the collector is in the sweeping phase.

Finally, Baker's copying algorithm[63] was chosen. Baker designed this algorithm with bounded (consistent) collection intervals in mind. The algorithm has been proved to work favourably in microcode on the MIT LISP Machines. The advantages of the algorithm are simplicity and potential for parallelism. The operational details of the Baker's algorithm are described in this chapter. The original Baker's algorithm is a classical virtual-real-time collector. It distributes garbage collection between host processing thus creating an illusion of transparency; but the efficiency of the host processor is still low as formulated by the Fraction of Useful Work Done (FUWD) being equal to ½ in equation 3.3.1.2.

Having decided on the specific algorithm, the next requirement is to *maximise its efficiency* (P1 - Section 3.3.4) i.e. how would it be best implemented ? This

requires :

- minimised time for garbage collection bursts thus speeded up system response;
- maximised overlap ratio between host processing and garbage collection thus increased system efficiency.

It seemed that Baker's implementation in ALGOL[63] had been already stretched to its limit using a serial computer. To achieve further speed up in garbage collection, a low level approach would be required. Such a level would enable the designer to access the processor hardware more readily. It was decided to employ microprogramming. Since it was impossible to use a commercially available processor, the design of special hardware was planned. The aim of the design was to realise the functionality of Baker's algorithm in hardware and to make it as fast as possible. A careful study of the advantages of this algorithm revealed that its simplicity would ease the task of microprogramming and by realising its potential parallelism in hardware considerable speed up could be achieved.

To increase system efficiency, a new mode of garbage collection was proposed. It is reported as **dual mode garbage collection** in Wong[71] (cf Appendix V). The dual mode comprises the advantages of both incremental and parallel garbage collection. It is realised in a dual processor architecture - one processor responsible for list processing (LP) and the other, the special hardware, for garbage collection (GC). Normally, the GC is performing garbage collection in parallel with host processing in the LP. When a new cell is required the GC is interrupted and a garbage collection burst is performed incrementally. The time distribution graph of this dual mode garbage collection approach is shown in figure 3.3d.

The third requirement is for *simple interfacing* (P3 - Section 3.3.4). If the

difficulties commonly associated with a coprocessor are to be avoided the interface between the LP and the GC processors should use a standard system bus. In view of the requirements for simplicity and flexibility a "plug-in" system was considered a prospect. Many AI systems are currently based on M68000 family processors and the associated VMEbus [72]; the latter, especially, is gaining increasing popularity and has been proposed as the IEEE1014 industrial bus standard. Because of its potential in AI/Engineering applications, a VMEbus based M68000 processor system was felt to be a very attractive option. It was, therefore, selected as the base of the special hardware.

This concept of using a standard bus architecture to implement an AI machine has been further extended in the design of THESIS - The Hardware Environment for Small Intelligent Systems [73,74] (cf Appendix V). THESIS is based on the independence of AI functions; thus, realising them as independent functional modules in hardware using a standard bus system. The proposed separation between program and data memory fits well to engineering applications. The provision of dedicated Cell Memory (CM) makes possible the design of a specialised hardware module the Intelligent Cell Memory System (ICMS) which incorporates garbage collection and cell storage. Other valuable features can also be identified in this design.

Various approaches to implement the special hardware were considered. A VLSI chip design was considered and rejected due to the cost and long design time which it would involve and it was decided to first build a prototype system based on bit-slice processors. This solution was adopted because of the flexibility offered by these devices and the high speed operation which would be possible. The possibility of implementing Baker's algorithm in a combination of hardware and microcode was another attraction offered by this approach.

Advance Micro Devices (AMD) 4-bit slices [75] were used because of their availability and popularity. The power consumption of these devices (Schotky-TTL) was the most obvious drawback followed by the large board area taken up by them. These advantages were regarded as acceptable in a prototype system.

4.3 ICMS Design

The ICMS project was completed in two stages. In the first stage an interactive LISP interpreter was written - simpleLISP (cf Appendix II). This was later used to produce the necessary programs for testing and verification of the hardware. In the second stage, the project concentrated on hardware design, its implementation and testing. The construction details of the ICMS are described in the following two chapters.

LISP was chosen to be the local programming language for the ICMS. Besides the fact that LISP is a popular language widely used for AI programming, there were two reasons that LISP was chosen :

(1) its extensive use of list data structures which requires only a small set of well defined functional primitives, namely : CAR, CDR, CONS, REPLACAR and REPLACDR;

(2) explicit dynamic memory allocation by the CONS functional primitive.

Theoretically, other AI languages, such as PROLOG, could have been the choice. Reason (2), however, makes LISP preferable. It is because CONS is the only primitive which 'creates' a cell. Therefore, by inspecting the free cell population for every call of CONS, garbage collection can be invoked in a controlled fashion. This is easy to implement in hardware by using either a polling or an interrupt mechanism. On the other hand, memory allocation of other AI languages is implicit and

this makes the occurrence of garbage collection completely random and uncontrollable. In fact, the proposed software environment for the original Baker's algorithm was LISP.

The ICMS is a hardware implementation of the Baker garbage collection algorithm. It is designed as a generic coprocessor unit without constraint to a specific host. The desirable features of a practical garbage collection system as outlined in section 3.3.4, are provided by the ICMS.

Normally, the Intelligent Cell Memory Controller (ICMC) is constantly performing garbage collection over the cell memory (CM) in parallel with the list processing by the M68000 host. Effectively, the host processor is totally unaware of the existence of garbage collection. Garbage collection is performed in bursts. When the host requests a cell, a hardware interrupt is generated in the controller. The latter finishes off the current collection burst before servicing the requested operation. Therefore garbage collection bursts are considered as unit operations. This approach provides some degree of overlap between garbage collection and list processing.

A more detailed consideration of Baker's algorithm at this point will highlight those features exploited in the ICMS design.

4.4 The Baker Copying Garbage Collection Algorithm

Baker's algorithm [63] was chosen for use in the ICMS. It is a copying scheme which is simple to construct in hardware, possesses potential for parallelism and offers a bounded list processing time. The algorithm was first reported to have been microcoded in the MIT LISP machine. The LISP machine approach was different from the ICMS : it set out to incorporate a complete LISP system on a customised

VLSI chip. It was considered successful. Its concept has been adopted by many commercial vendors, e.g. Symbolics Inc. However, all such commercial machines have been built for R&D purposes and exhibit complex interfaces which make them unsuitable for engineering applications. On the other hand, the ICMS is designed as an extension to existing technology. Therefore, improved real time response can be provided for engineering systems with minimum overheads. The philosophy of copying garbage collection has been generally described in Section 3.2.3. In this section, the operation of Baker's algorithm is explained in detail.

4.4.1 The Non-Real-Time Version

An extra pointer S is required, apart from B which points to the next available free cell (figure 3.2.3). Pointer S is termed the SCAN pointer which always addresses to the next target cell for GC. The objective of GC is to copy valuable lists from the FROMSPACE to the TOSPACE. Whenever a cell is copied, a forwarding address is left behind at the HEAD element of its original FROM reference. This prevents garbage collection of the same cell twice and redirects pointers which attempt to access the cell.

Subsequent to a cell allocation, B is incremented to point to the next free cell. The need for GC is evident when B points to the bottom of the semispace. With the non-real-time collector, list processing is suspended for as long as GC is in progress. GC starts with an operation called FLIP which interchanges the identities of the two semispaces. B and S are reset - pointing to the top of the new TOSPACE initially. Immediately after FLIPping, root registers are copied. Further copying proceeds from roots. Copying a cell, increments the B pointer but not the S. At any time, cells residing between S and B have already been copied but their contents have

not. Starting from the top of the semispace, SCANning is a process which inspects the nature of the contents of individual cells† pointed at by S - whether they are in the TO- or FROM- SPACE, and performs copying on them when necessary. Thus, GC is effectively a game of S chasing after B; it is completed when B is caught.

4.4.2 Incremental Collection

The original Baker's algorithm was modified to produce the incremental version. It was implemented on serial computers and was put into practice in one of the first LISP machines in M.I.T. Section 3.3.3 explains the basic philosophy behind this type of collector. Prior to adopting it for the ICMS, a software version of Baker's incremental algorithm was implemented and its correctness was verified [76]. In the following discussion, the special treatment of list primitives by the software version is presented††.

The incremental algorithm is based on two principal facts: (i) GC is redundant if no cells are requested; and (ii) at the instant of cell request, as long as there is one or more free cells, a system can survive. Functionally, a SCAN is performed per cell request (CONS). Again, when S has caught B, GC is finished.

CONS is a list primitive for construction which is invoked by CONS(X,Y). Its function is to assign its parameters to the HEAD and TAIL of a newly generated cell, respectively. Prior to assigning, X and Y are checked and copied to ensure they are valid addresses. Since the new cell and its contents have been ensured genuine, it would be a waste of effort to SCAN it. Being aware of this, an extra pointer T is introduced, which points to the next available free cell from the top. T is

† The inspection then copy procedure is termed MOVE(X).

†† In the original algorithm [3], the implementation is simplified - parameters passed to these primitives are assumed genuine thus less precaution for invalid data retrieval is required.

decremented after allocation, moving in the opposite direction from B. (The division of TOSPACE is illustrated in Figure 4.4.2). Gradually, T approaches B. When they meet, free memory is exhausted and GC is invoked. The longest time required to perform a CONS is when GC has just started. It is equal to the total time required for FLIP, parameter checking and copying, to COPY all roots and generate a cell. Generally, the upper bound of a system is the time required for the CONS operation which is in fact the longest of all list primitives. This upper bound affects system bandwidth which is a crucial engineering design factor particularly for time critical applications. Although CONS is the only function which creates a cell, special attention also has to be paid to cell accessing primitives; these include CAR/CDR and REPLACAR/REPLACDR [59].

CAR/CDR of X, returns the HEAD/TAIL element of the cell pointer X. Now, the list processor has to guarantee that the genuine results are returned; because if X is a FROMSPACE pointer, X may have been copied, so the forwarding address is used for data retrieval; and if X is a genuine FROMSPACE pointer, it is copied before its HEAD/TAIL is returned; finally, if X is in TOSPACE, no extra operations are required. Therefore, the worst case time performance for CAR/CDR equals the time required to perform the primitive plus the time for one COPY operation.

REPLACAR/REPLACDR is called with two parameters X and Y. It replaces the HEAD/TAIL content of X with Y. Both of these primitives affect connectivity, therefore, to be safe X and Y are inspected and copied (if necessary), before any replacement operations. In this case, the worst case time performance is the sum of the time for the operation itself and two COPY's.

In practice, Baker's algorithm is microcoded in the ICMS and it is performed in a dual operational mode: incremental and parallel. Moreover, with special hardware

support its operational speed is enhanced. In the next two chapters, the functional and construction details of the ICMS are described.

4.5 Parallelism in Hardware

Baker's algorithm was initially verified in software and preliminary studies of the generated code revealed some degree of hidden parallelism in the algorithm. It was realised that implementing these in hardware could provide a decrease in garbage collection time.

The definition of the cell storage structure is complicated but to access a cell many time consuming addressing instructions are required - over 50% of the generated code. A solution to the problem is to abandon the orthodox byte/word storage convention and treat a cell as an elemental storage entity. Addressing a cell then would be much more natural and require less time and effort. In addition, on average, up to two thirds of the cell address instructions are conditional. The cell to be addressed mostly depends on either the state of the collection process or the region to which the cell pointer belongs. Parallelism can be exploited by overlapping the testing and the cell addressing instructions. Since the testing instructions are simple, they can be realised using dedicated hardware, making them independent from cell addressing. Thus both instructions could be executed concurrently.

Pointer T can be discarded. Its significance in the original algorithm is to save the garbage collector from monitoring the newly created cells. In so doing the amount of work for garbage collection is reduced. On a serial computer the benefit is prominent. However, garbage collection with a separate unit can afford to do extra work. The work load of the host remained unchanged, because the process of garbage collection is transparent to the host. The advantage is that the hardware over-

head is reduced and less work is required for a collection burst. Thus a reduced separation time can be achieved.

The other two pointers, S and B, are similar in usage. Whenever they are accessed, they are incremented subsequently. Some degree of speed up can be achieved by implementing the B and S pointers in hardware, incorporating auto-increment capability.

4.6 Engineering Features

4.6.1 Simplicity

The ICMS is an independent functional module. From the host processor, it is seen as a passive storage device, accessible by writing or reading a bank of registers. Each register is responsible for a LISP primitive and is assigned with an address. Effectively, writing into a register is equivalent to passing a parameter and invoking the corresponding list operation at the same time. At the end of the function, return values are read from the same address.

A simplified block diagram of the host/ICMS interface is shown in figure 4.6.1.

4.6.2 Fast Response

Although the throughput of the host processor will be reduced by the frequent requirement for garbage collection bursts, the design of ICMS ensures that these take place at an acceptable speed by adopting several architectural features :

i) Cell Unit-addressing - Two bit slice ALUs are employed, one is responsible for the manipulation of the HEAD pointer and the other is dedicated to the TAIL. The operational speed for list functions on the cell is therefore enhanced. Moreover, this architecture does not exclude the possibility of either only one

ALU or one address/data being required. In such circumstances, the unnecessary device may be made redundant by disabling it using the appropriate micro control bits.

ii) Hardware Testing - Five tests are performed in hardware concurrently with the bit slice ALUs, namely

- *CELL(X)* is X a cell pointer;
- *OLD(X)* does cell pointer X exist in the OLD region;
- *NEW(X)* does cell pointer X exist in the NEW region;
- *GC* has garbage collection finished ? and
- *CMEND* has the cell memory been exhausted ?

iii) Content Prefetching - This idea derives from "instruction prefetch" in computer architecture design. Running in parallel with hardware testing, it ensures the correct address at the memory address port at the end of a conditional fetch instruction. Thus test and fetch instructions are executed in one machine cycle which otherwise would be two or more.

iv) Auto-incrementable Pointers - Special cell pointers B and S are constructed in hardware using flip-flop counters. They are incremented after being used for cell addressing.

4.5.3 Flexibility and Portability

System integration of the ICMS is made easy by using a standard bus interface. The ICMS is not restricted to particular processor types in contrast with many AI hardware designs which have complicated interface schemes to specific processors.

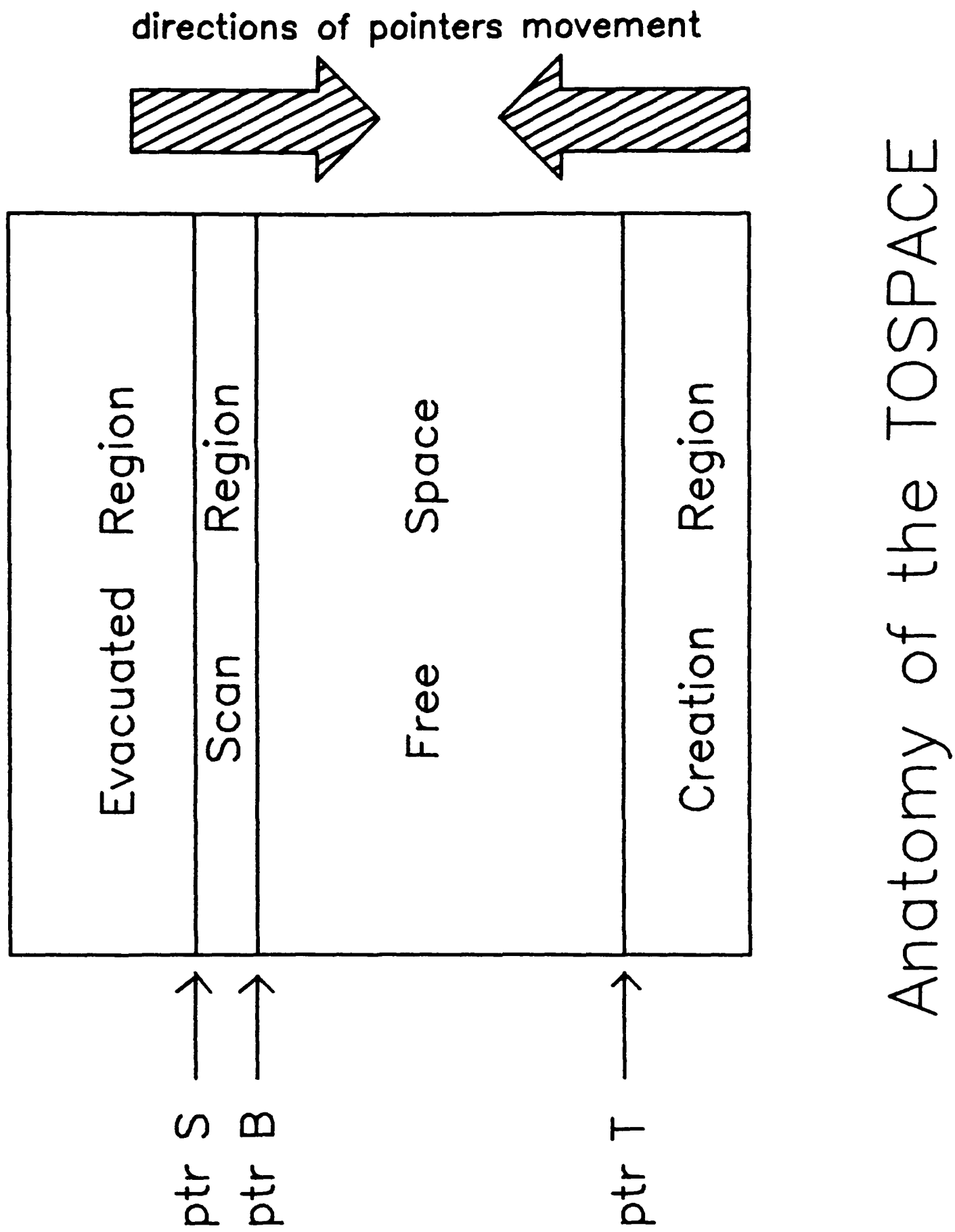


Figure 4.4.2: Anatomy of the TOSPACE of the Incremental Baker's GC.

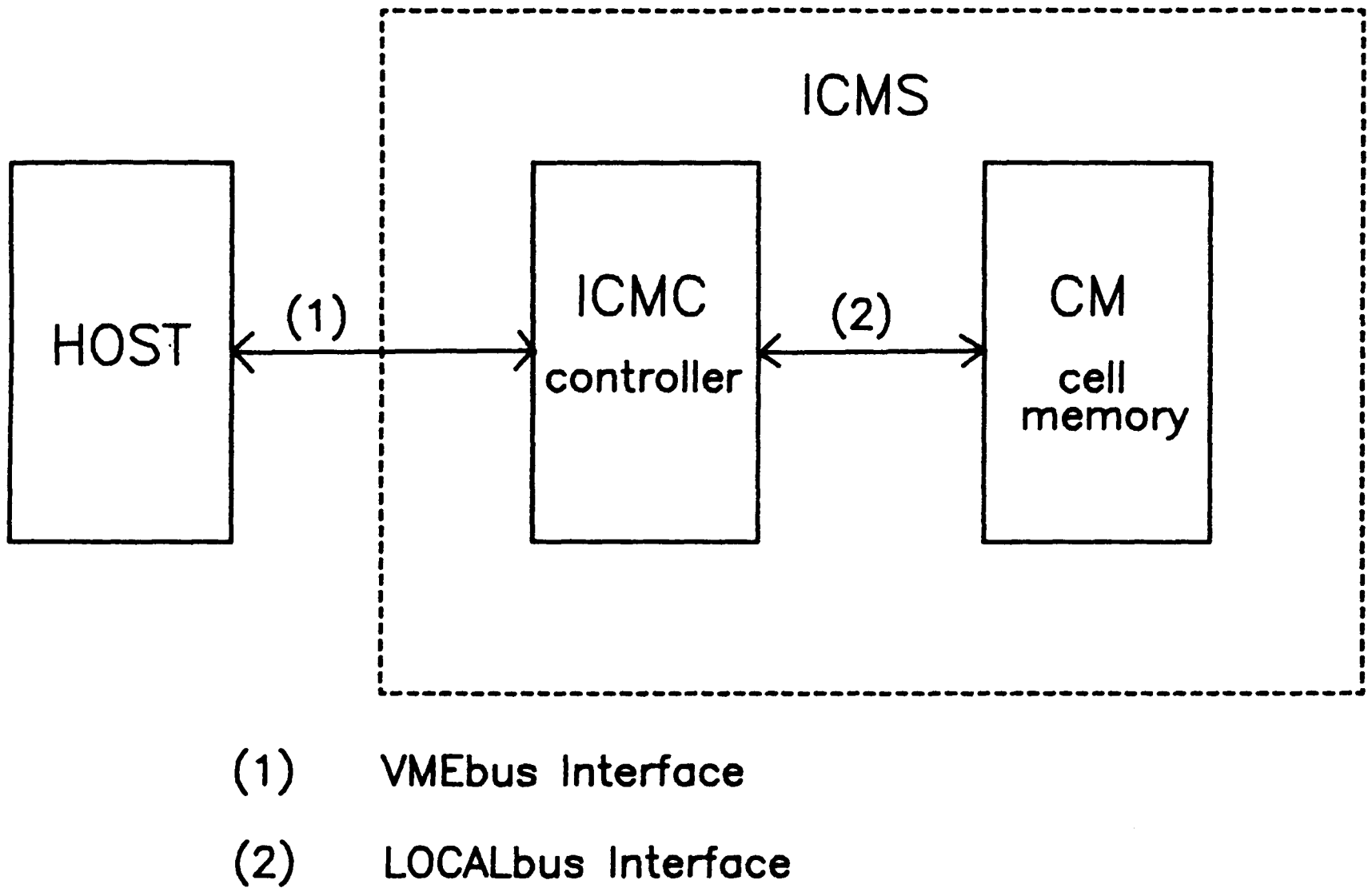


Figure 4.6.1: The Schematic Block Diagram of the Host-ICMS Interface.

CHAPTER V
Construction I: Hardware

The ICMS is a microprogram controlled bit slice coprocessor system. A photograph of the complete system is shown in plate 5. In this and the next chapter, construction details of the ICMS will be described. This chapter concentrates on the hardware aspect of the ICMS design and chapter six focuses on the development and design of the ICMS microprograms.

5.1 The LOCALbus and the Cell Memory

The architecture of the ICMS can be divided into two main parts: the cell memory (CM) and the controller (ICMC), as shown in figure 4.6.1. The interface between the two is achieved by a local bus system (LOCALbus). The structure of the ICMS is shown in more detail in figure 5.1 and plate 5.1 is a photograph of the ICMS circuit board.

The LOCALbus can be grouped into three channels: the control, data (BD) and address (YA) lines. There are 64 control signals which correspond to the 64 micro control bits and these are generated from a microprogram store within the controller. The BD and the YA channels originate from the processing units' input and output, respectively. Both of them consist of 48 lines, 24 for each HEAD and TAIL address/data. In addition, the LOCALbus is also used to integrate all functional units in the controller internally.

The cell memory (CM) is configured to be cell addressable. It is organised into two banks of identical Random Access Memory (RAM), one for each HEAD and TAIL element of a cell. Separate ALUs are assigned to each bank. The content of a

HEAD/TAIL element can either be an atom or a cell pointer. Both of them are addresses - the former points to numbers or strings and the latter references other cells. For use with a M68000 host, the size of each HEAD/TAIL element was chosen to be 24 bits. However, when the ICMS was implemented, this turned out to be an awkward choice. This is because, internally the data and the address registers of the 68000 are all 32 bits in size. When information is read from the cell memory, its upper byte (BIT_{24}-BIT_{31}) is in the high impedance state; thus FF (hex) is inserted in the most significant byte (MSB) of the registers. This would create problems for comparison instructions. To overcome this, all MSB's are ignored, or masked to zeros.

The cell memory can be accessed from the controller through the LOCALbus. The memory address is presented at the YA channel and data transfer is done on BD. Only three control lines are responsible for memory access. They are: Read/Write (@mem_read†), Chip Select (@mem_sel) and Output Enable (@mem_oe). At present, only 2k cells are implemented (split equally amongst TOSPACE and FROMSPACE). They are addressed by YA_0-YA_9, and YA_{10} is used to determine the semispace in which the cell is located. The remaining YA lines are used to differentiate a cell pointer from normal addresses. At least one bit has to be reserved for this purpose. Therefore, 8M (2^{23}) is the maximum number of possible cells with the current controller design.

Standard RAM devices - static CMOS RAM 6116 - are used for storage. There are two reasons for this choice. Firstly, it is to show that the enhanced system per-

† Throughout the thesis, all micro instruction fields are preceded with a "@". This is mainly to help differentiating them from other symbols or abbreviations. In the actual microprogram source, (shown in appendix IV) this character is dropped.

formance is brought about by the ICMS architecture rather than high-speed memory. Secondly, to achieve compatibility with popular memory devices to reduce system costs.

5.2 The Intelligent Cell Memory Controller (ICMC)

The controller is the heart of the ICMS. Its major role is to coordinate data transfer between the host and the cell memory. LSI Bit-Slice components from AMD (Advanced Micro Devices [75]) and 74LS series MSI devices were extensively used in its construction. The AMD 2900 family devices are widely used in bit slice microprocessor designs.

Internally, the ICMC is divided into four functional units. They are: the microprogram control unit (MPCU), the test circuit (TC), the Input/Output ports (IO) and the processing unit (PU). Synchronisation of these units is maintained by a system clock. It is generated from a clock driver which is driven by the 16 MHz $\overline{\text{SYSCLK}}$ VME line. The driver has a number of outputs which can be selected by micro instructions. Particular outputs of interest are ÷2 for normal operation, ÷4 for memory read and ÷5 for memory write. Debugging of the controller was made possible by providing the system clock with halt and single stepping facilities, using two SPDT switches.

5.2.1 The Micro Program Control Unit (MPCU)

The entire ICMS operates under microprogram control. To enable microprogram development a micro assembler, UASM, was written. UASM is a meta assembler; with which users can define the machine configurations in the definition phase. From then on, in the statement phase, lines of instructions together form microprograms. UASM accepts lines of micro instructions and translates them into the

required bit patterns for the pre-defined machines. The implementation and invocation details of the assembler are explained in the next chapter and the syntax specification of the micro assembly language is described in appendix I.

The generated microcode is down loaded into some fast bipolar Random Access Memory. This RAM forms a Writable Control Store (WCS). A photograph of the WCS is shown in plate 5.2.1. The Am93422, 256 by 4-bit bipolar RAM, is used. Sixteen Am93422 cascaded together form a 256 microinstruction store with each instruction 64 bits in length (c.f Section 6.4). The WCS can either be connected to the M68000 host or the ICMS, figure 5.2.1a. The access time from the M68000 host is 40 nano seconds, which is about one third of the M68000's clock period (125ns). Normally, it is connected to the M68000 through the VMEbus interface. The WCS is mapped into the VME bus address space so that the M68000 can read and write from/to it. Standard address decoding techniques are adopted. The lower byte of the M68000 address bus is used to reference any one of the 256 microinstructions. The remaining address bytes are compared with a pre-selected WCS base address, via two 8 bit comparators (74LS687). The result of which partially enables all Am93422 devices. The pre-selected address is changeable. It is set by 2 banks of SPDT switches. For simplicity, only the starting WCS address is checked. The WCS, therefore, has been restricted to start from the last address (i.e. FFFFFFFF(hex)) and expands upward.

Microprogram development is carried out when the WCS is "talking" to the host. Alternatively, there is a mechanical switch to permanently disable the decoders, thus the M68000 can no longer access the WCS. In this mode, a standard microprogrammed computer architecture is formed: with the output from WCS (microinstructions) controlling various functional parts of the ICMS.

The outputs from the WCS are buffered (74LS245) while it is connected to the M68000; and they are latched (Am2920A), when the WCS is connected to the ICMS. In so doing, a one stage pipeline is achieved. Instruction address N appears at the address port of the WCS while instruction N+1 had been pre-fetched from the previous cycle. Instruction pre-fetching is a popular technique in computer design. Instruction latching is edge-triggered under the system clock; therefore, transient states, commonly occurring during conditional branching on pipelined machines, are ignored.

The execution sequence of a microprogram is directed by the microprogram addresses. These addresses are 8 bits wide and are generated from the sequencer (SEQ'er - two Am2911A). There are several sources for the generated addresses. They can either be transferred from internal registers within the SEQ'er or redirected from the @BA (Branch Address). The choice is determined by a 4 bit micro instruction field, @Instr_sel. This allows 16 possible control sequences. Normally, execution is sequential. There is an internal microprogram counter which remembers the next consecutive micro instruction and is incremented after each cycle. Nevertheless, occasionally, branch instructions will lead execution to different points. When this happens the microprogram counter is automatically updated. The @BA can either be one of two types:

i) static - it is predefined as part of the microprogram; or

ii) dynamic - it comes from the output of the mapping ROMs (MAP) which depend on the host requests. The MAP is a pre-programmed PAL, Programmable Array Logic, N82S105 from Signetics which has an maximum access time of 65ns.

Also, a 16 entries deep stack is implemented in the SEQ'er. When a jump to subroutine instruction is called, the return address is automatically pushed onto the top of the stack and these return addresses are popped upon exit from the subroutine. This enables 16 level nesting in procedural/functional calls. At power up the SEQ'er is reset to zero. Location zero is the startup vector where the first micro instruction is placed.

12 out of 16 of @Instr_sel are conditional instructions which determine the outcome of the microprogram addresses. The conditions are based on the setting of a test bit which is selected from the 8 to 1 condition code multiplexer (MUX - Am2922) The inputs to the MUX are provided by the test circuitry whose selection is driven by the 3 bit @TEST_sel. A Next Address Generator (NAG - Am29811A) decides the source of the next address. It generates the required signals to either the sequencer or the MAP ROM. The block diagram of the complete MPCU is as shown in figure 5.2.1b.

The ICMS is interrupt driven. It is continuously performing garbage collection in bursts transparently to the host. During each burst, a micro control bit @INT is pulled high (refer to figure 5.2.1b). This forces the output of the OR gate high hence disabling the output from the MAP ROM. Effectively, interrupts are disabled. At the end of the burst, @INT is cleared which re-enables the interrupt capability. When the M68000 host requests a list operation, at this point, an interrupt is generated. The interrupt service vector is provided by the MAP ROM which is addressed by 8 of the VMEbus lines, namely address lines A_1-A_7 and the read/write control line.

5.2.2 The Test Circuit (TC)

The inputs to the test circuit are derived from the BD lines. Seven outputs are provided which are directly connected to the inputs of the MUX. The microprogram sequence is determined by selectively sampling them. In table 5.2.2a, the details of the test circuit outputs are summarised. The complete test circuit is illustrated in figure 5.2.2.

Pointers B and S are included in the test circuit. Both of them are implemented using three 4 bit asynchronously resettable counters. The first ten output lines are used allowing them to address 1k cell memory (one semispace only). The eleventh line is fed back to reset the counter to null. Reading and incrementing are controlled by 3 micro control bits, as shown in table 5.2.2b. The states of the counters affect the rest of the test circuits.

When B overflows, it means that the TOSPACE is exhausted. At this moment, a FLIP is performed. This simply toggles the flip flop (TFF1) whose outputs contribute to two events:

i) Q is read as the eleventh bit of B or S (B_{10} and S_{10}); and

ii) Q and $\overline{Q}$ are control inputs to the TO and FROM testers, respectively.

TFF1 remains unchanged until the next FLIP. The B overflow signal is also latched (DFF1) until the next B increment. This is interpreted as the FLIP test signal. The process of garbage collection is completed when the pointer S has "caught up" with B. Similarly, there is a case when S is caught by B - just after a FLIP, both B and S are reset, pointing at the top of the new TOSPACE. Therefore, the test signal GC is logically defined as:

$$GC \equiv (S \text{ equal to } B) \text{ AND } (\text{not FLIPped})$$

When this happens, test output GC is asserted by forcing the TFF2 to zero. This signal will last until the next B overflow, when TFF2 is toggled to a 1.

Whether an object is a cell or an atom is determined by the test signal CELL. The test is performed on the upper part of the BD lines, namely BD_{11}-BD_{23}, which reveals an address identity. Presently, the cell space is set from FFF800 to FFFFFF (hex). Once the CELL test on an object is true, the TO and FROM tests are enabled. BD_{10} is and'ed with the Q and $\overline{Q}$ of TFF1, respectively.

TRUE and FALSE are to provide "branch always/positive" and "branch never/negative" micro instructions. POLLED is a signal generated from the decoder's (DE1) output. At the beginning, a polling mechanism was adopted, and POLLED was set when the correct address from the VMEbus appeared at DE1. In its present form, the system is interrupt driven although POLLED still exists. This provides faster operation.

The size of the ICMS board as shown in plate 5.1 is too small to accommodate the testing circuitry. Therefore the latter was built on another Eurocard printed circuit board as shown in plate 5.2.2. A 64 line ribbon cable was used to interconnect the two boards.

5.2.3 The Input and Output Registers (IO)

Two pairs of memory mapped IO registers (twelve Am2920A) are incorporated. Separate registers are used for input and output, but the same address is shared. The selection between the two is governed by the read/write line. They can be accessed at several addresses from the VMEbus and each address is associated with one list operation. When an address is asserted, A_8-A_{23} are compared with a pre-defined base address. The base address is relocatable, and it is currently set to FF8000 (hex).

If these are matched, the data lines are latched into the IO, an interrupt is sent, the MAP ROM is enabled and the required list operation is executed. The role of these registers is for message passing between the host and the ICMS - messages include functional parameters and subsequent return values. Therefore, a list operation is simply invoked by writing the appropriate parameters into the desired IO registers. The programmer's model of the supported list functionality of the ICMS is shown in figure 5.2.3a.

The entire transaction is performed asynchronously. By definition, for the VMEbus protocol [77], an Address Strobe ($\overline{AS}$) is sent by the host to advise slaves of the availability of some information. In return, after acquiring the information, Data Transfer ACKnowledge ($\overline{DTACK}$) is generated by the slave to terminate the transfer cycle. Until $\overline{DTACK}$ is received, the host remains idle. For the ICMS controller, $\overline{DTACK}$ is generated by the circuit shown in figure 5.2.3b. Consecutive addresses are allocated for the two pairs of registers (2 by 4 bytes). A list operation is requested when the appropriate address of the second (or upper) IO register appears on the bus. The first (or lower) register is passive: the register is merely updated. For either register, data is transferred in two word (16 bit) cycles. Apart from the lower word cycle of the second register, $\overline{DTACK}$ is returned directly without waiting. During this lower word transfer, the interrupt signal is generated and the task of $\overline{DTACK}$ generation is passed to the microprogram, namely when @BUSY=1, at the completion of the list operation. Thus:

$\overline{DTACK} \equiv$ (not lower word access of the 2nd IO register)
OR (@BUSY is set)

5.2.4 The Processing Unit (PU)

The functions of the processing unit are to direct traffic on the data path and to

perform arithmetic and logical computation. Presently, however, only the first of these has been realised. The latter will be implemented in the future so that the ICMS could provide more list functions, such as ADD, AND, etc. There are two functional parts within the processing unit; these are two bidirectional buffers (six 74LS645) and two banks of ALUs (twelve Am2903A), figure 5.2.4a.

The two buffers are implemented to create communication links between the HEAD and TAIL data and address paths. One buffer is connected to the BD data channel and it is known as the @data_buf; the other, @add_buf, is interfaced to the YA channel. With these buffers one of five possible data/address routes can be established. These are: HEAD only, TAIL only, HEAD to TAIL, TAIL to HEAD or independent.

The ALUs are constructed using bit slice processors. There are two reasons for this choice. Firstly, it is because of the flexible nature of the bit slice design approach. This allows easy expansion of the data path in the future with minimal redesign being required. Secondly, these devices have potential speed advantages. The two banks of ALUs are responsible for the processing of the HEAD and TAIL information in parallel. Each consists of six 4 bit processors, together forming a 24 bit data path.

Am2903A 4-bit bipolar microprocessor slices are used. The hardware features of the Am2903A exploited in the ICMC are:

- An internal 16 word by 4 bit dual ported RAM which is infinitely expandable using external register files. That means that more working space (e.g. more ROOTS) can be built if required;
- Three ports: two bidirectional ('b' and 'y') and one input only ('a'). A total of

twelve Am2903A are used to provide the two 24-bit data paths for the HEAD and the TAIL.

The functions of the ALUs are controlled by a 9 bit micro control word @ALU_instr. The YA and BD channels of the LOCALbus are derived from the bi-directional ports of the ALUs, namely, 'y' and 'b' respectively. There is also an 'a' port which is responsible for internal data transfer. Each ALU contains sixteen dual ported registers. A particular register is addressed by two 4 bit fields @AA ('a' port address) and @BA ('b' port address). The 'b' port is read/write but the 'a' port is read only. The upper eight registers are assigned as roots. The flexibility of bit slice design is illustrated at this point, since dual ported RAMs could be used for expansion as explained in [75]. The integration of the processing unit with the rest of the ICMS can be represented, as shown in figure 5.2.4b. The test circuit is always connected and operates concurrently with the ALUs in every micro instruction. Apart from it, at any time only one functional unit is connected to the YA channel and another to the BD. This selection is microprogrammed.

5.3 Construction Details

The circuitry described here was constructed on extended Euro-card circuit boards using wire wrap techniques. Due to the high speed operation of the bit slice devices some problems were experienced with this technique. It proved necessary to redesign the initial prototype boards to improve the layout and avoid problems with electrical noise. Another minor problem is heat dissipation due to the bipolar technology used to fabricate AMD 2900 devices. This was solved by mounting a small extractor fan on top of the rack.

The large number of components was partly due to the choice of devices used. 4

bit slices were adopted because they were the only fully developed device set known at the time. Moreover, future expansion was envisaged; therefore the smaller the slice, the more flexible it would be. For future working systems, should the size of the datapaths be definite and no expansion be expected, the system could be designed with devices with "larger" bit slices e.g. Am29116, a 16-bit microprocessor slice.

The number and size of components meant that three circuits boards were required and the necessary wiring lengths and interconnections resulted in some timing problems. These could have been eliminated with a custom designed multilayer printed circuit board; but the high cost of this was not considered to be justified for the present prototype system.

Finally, the coordination and communication between individual units are under microprogram control and the details of this are given in the next chapter.

Tests	TC inputs	Connections to MUX	Comments
POLLED†	VME address lines	0	Has an operation been requested ?
CELL	BD_0-BD_9	1	Is the input a cell ?
GC	internal	2	Has garbage collection finished ?
TO	BD_{10}	3	Does input exist in TOSPACE ?
FROM	BD_{10}	4	Does input exist in FROMSPACE ?
FLIPPED	internal	5	Has the system just FLIPped ?
TRUE	none	6	Always true or high.
FALSE	none	7	Always false or low.
† Initially, a polling mechanism was used instead of interrupt.			

Table 5.2.2a: Details of the Test Circuit Outputs.

Functions	@sel_139	@A0_139	@A1_139
increment S	1	0	0
read S	1	1	0
increment B	1	0	1
read B	1	1	1

Table 5.2.2b: The Decoding of Read and Increment for the S and B pointers.

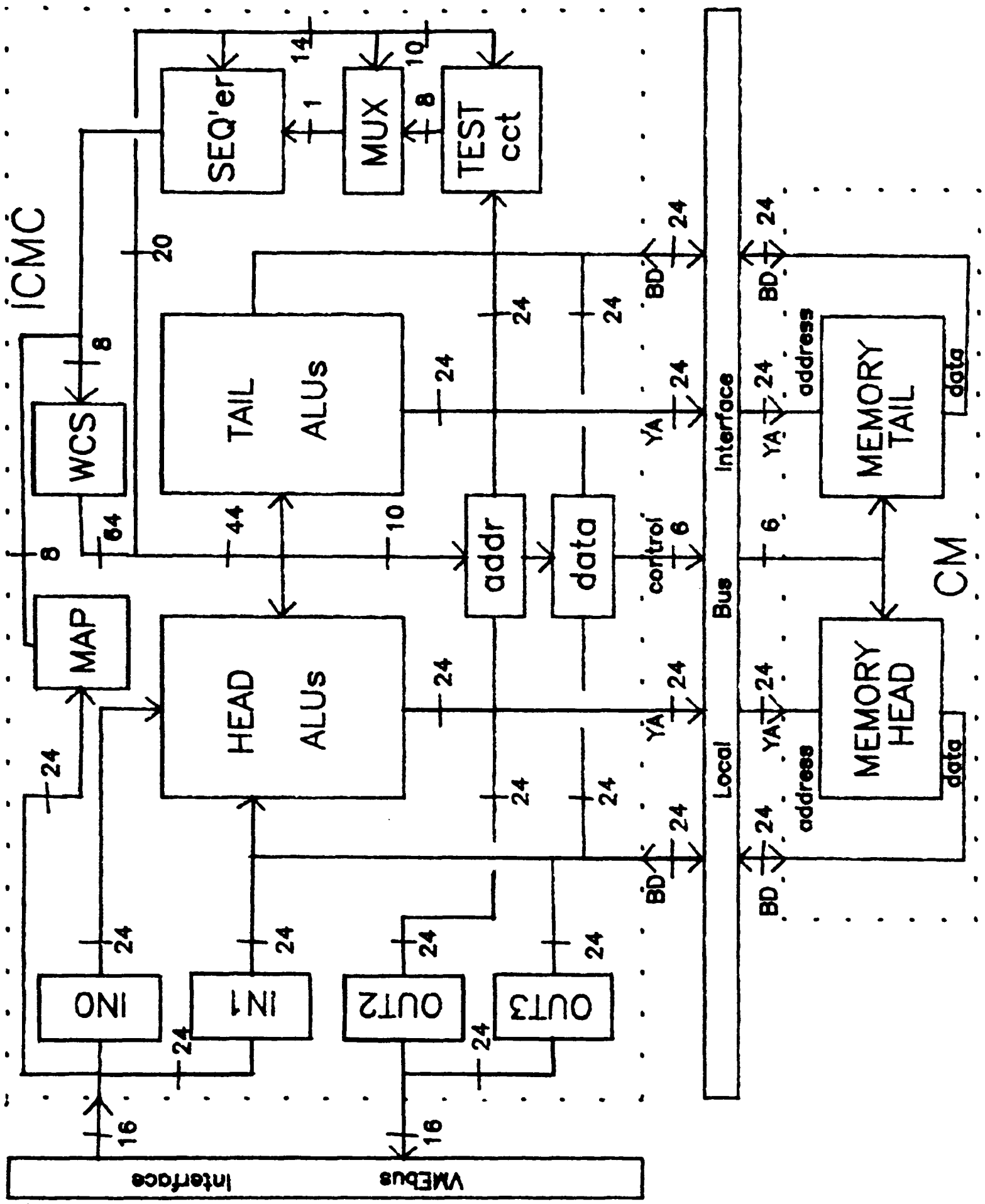
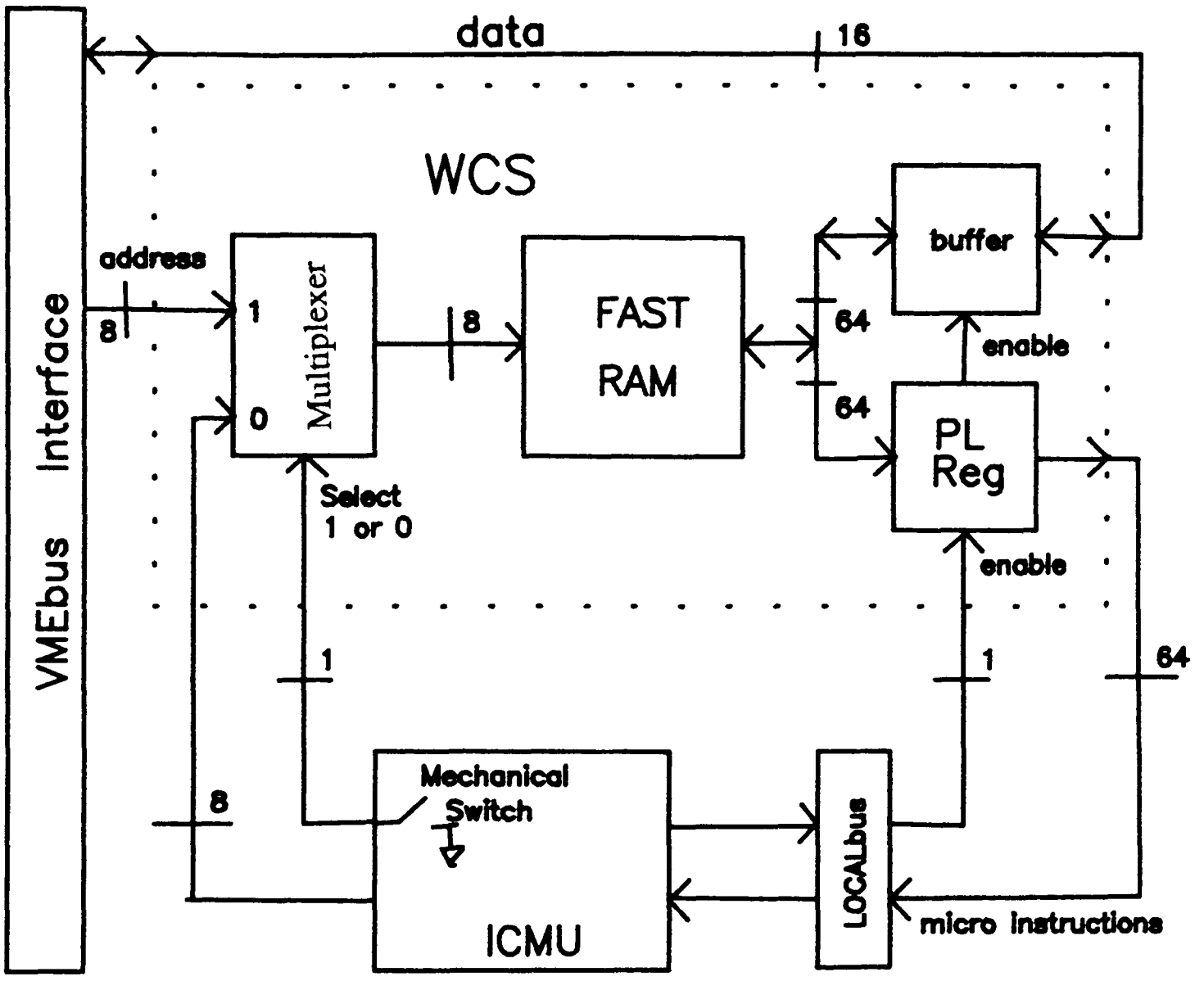


Figure 5.1 The Detailed Block Diagram of the ICMS Architecture.



Structure of the Writable Control Store

Figure 5.2.1a: The Writable Control Store (WCS).

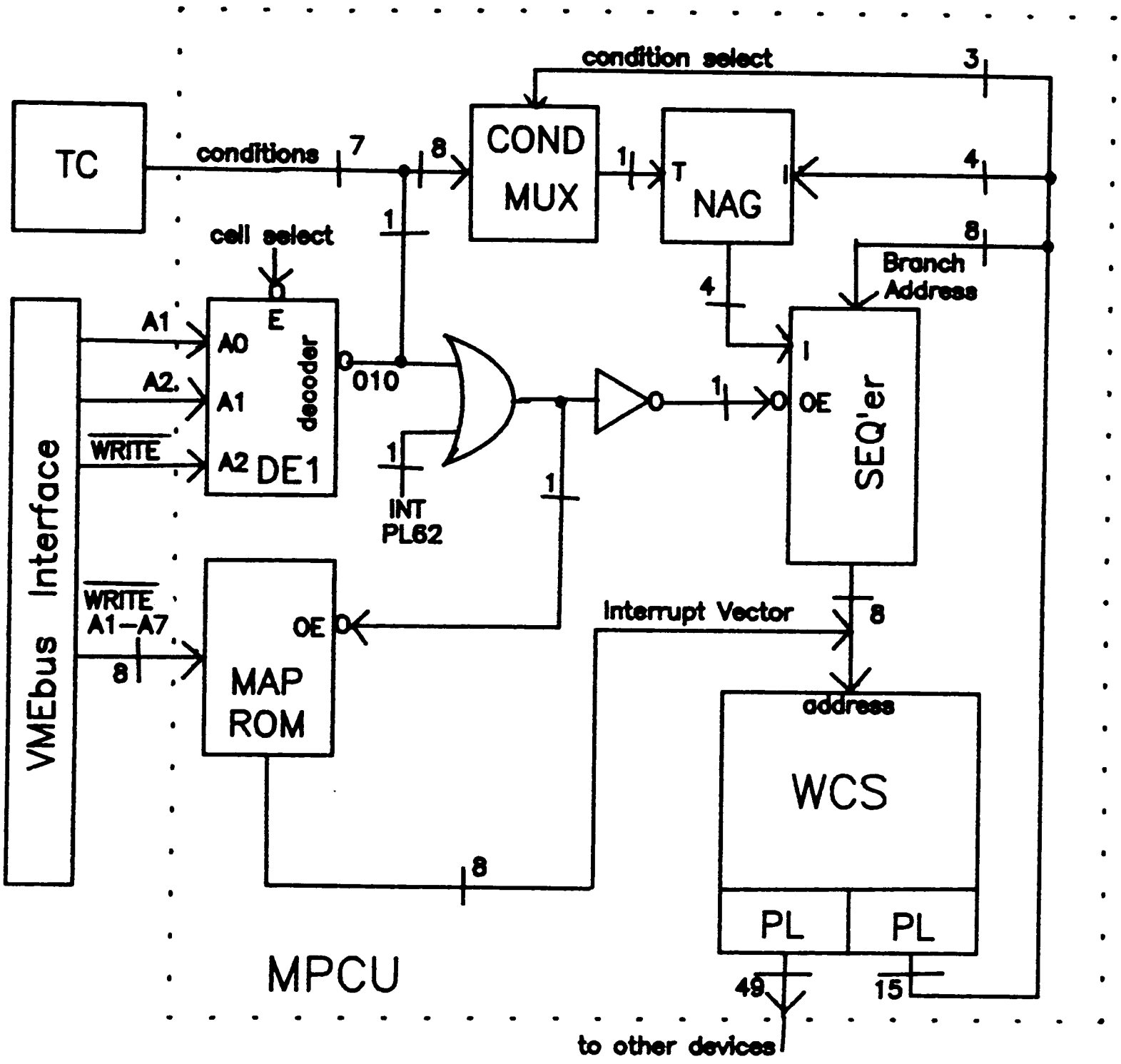


Figure 5.2.1b: The Block Diagram of the MPCU.

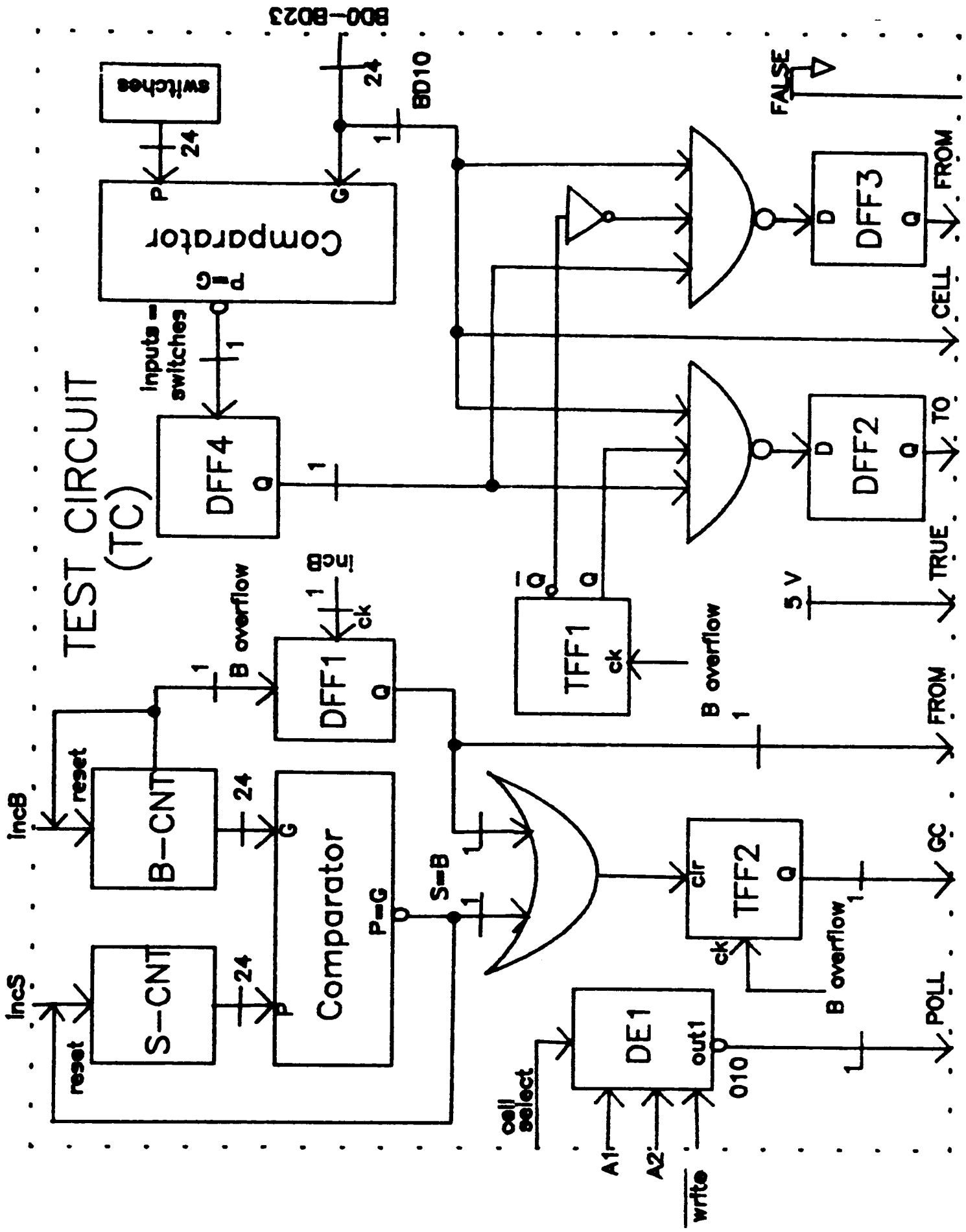


Figure 5.2.2: The Test Circuit (TC).

Functions	Parameters	Parameters	Addresses
setREG0	unused	ROOT0	FF8000
setREG1	unused	ROOT1	FF8008
⋮			⋮
setREG15	unused	ROOT15	FF8078
CAR	unused	X	FF8080
CDR	unused	X	FF8088
REPLACAR	X	Y	FF8090
REPLACDR	X	Y	FF8098
CONS	X	Y	FF80A0
	undefined	undefined	FF80A0
	Future Development		⋮
	undefined	undefined	FF80F8
	32 bits	32 bits	

Figure 5.2.3a:The Programmer's Model of the Supported List Functionality of the ICMS.

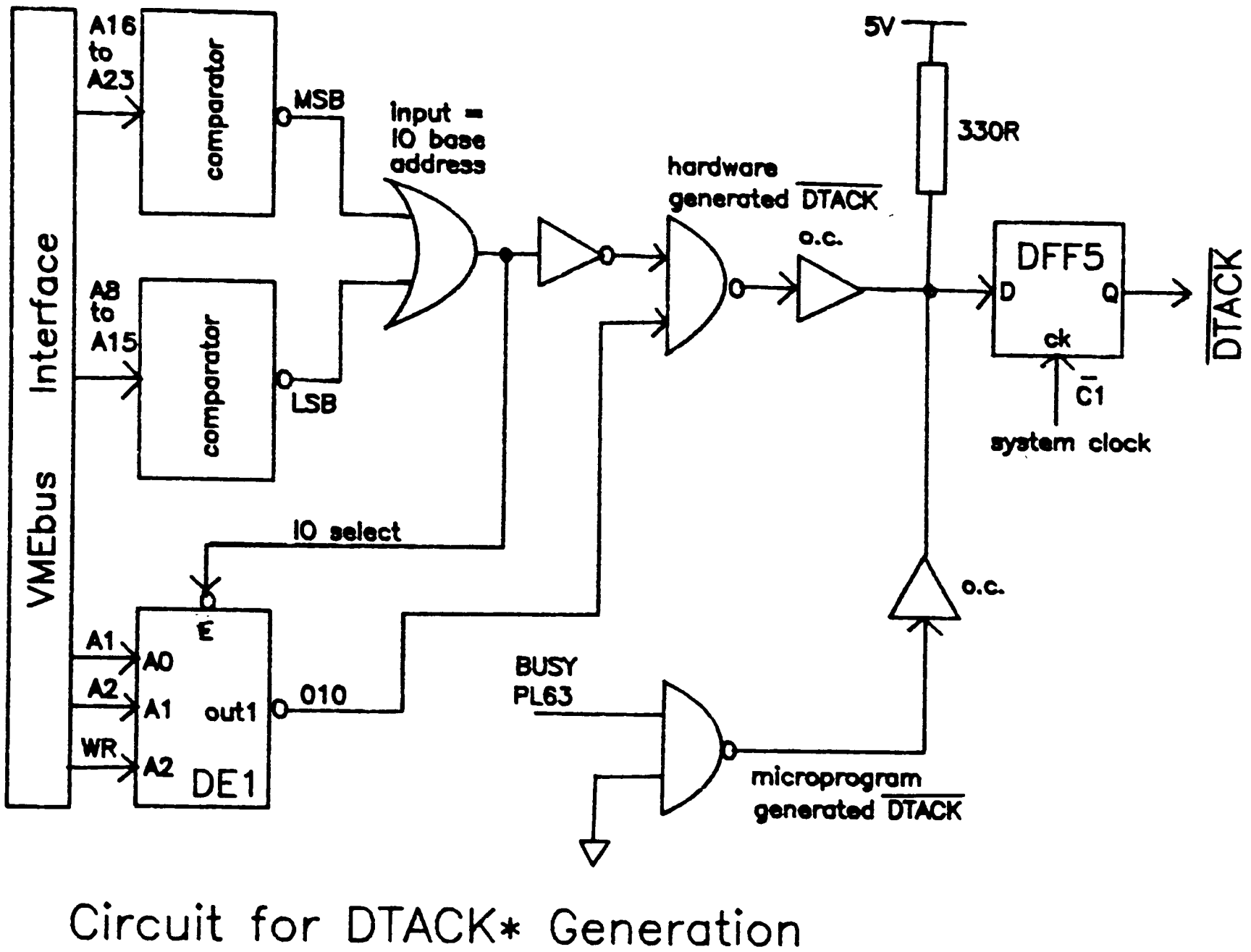


Figure 5.2.3b: The Circuit for $\overline{\text{DTACK}}$ *Generation.*

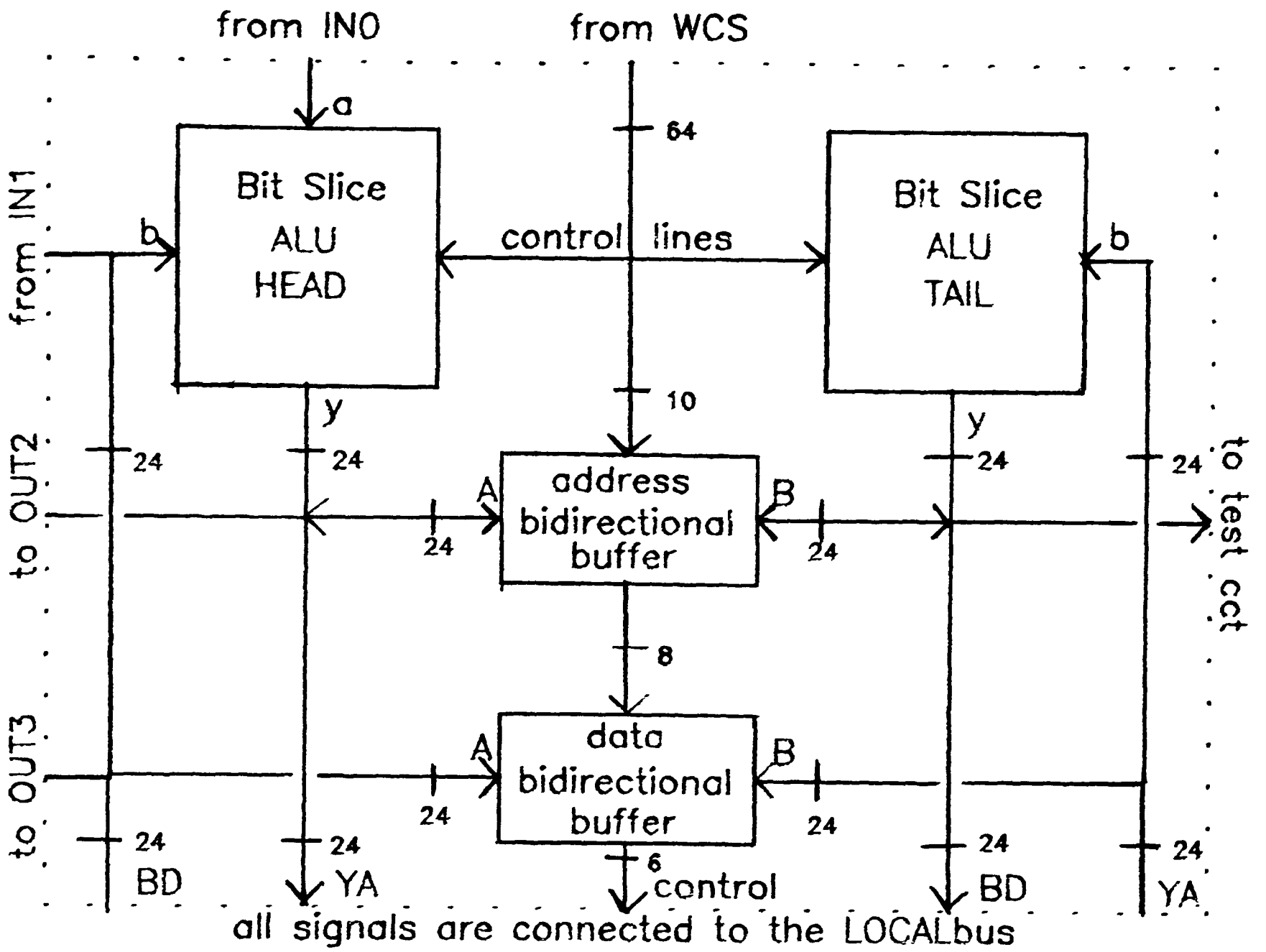


Figure 5.2.4a: The Processing Unit (PU).

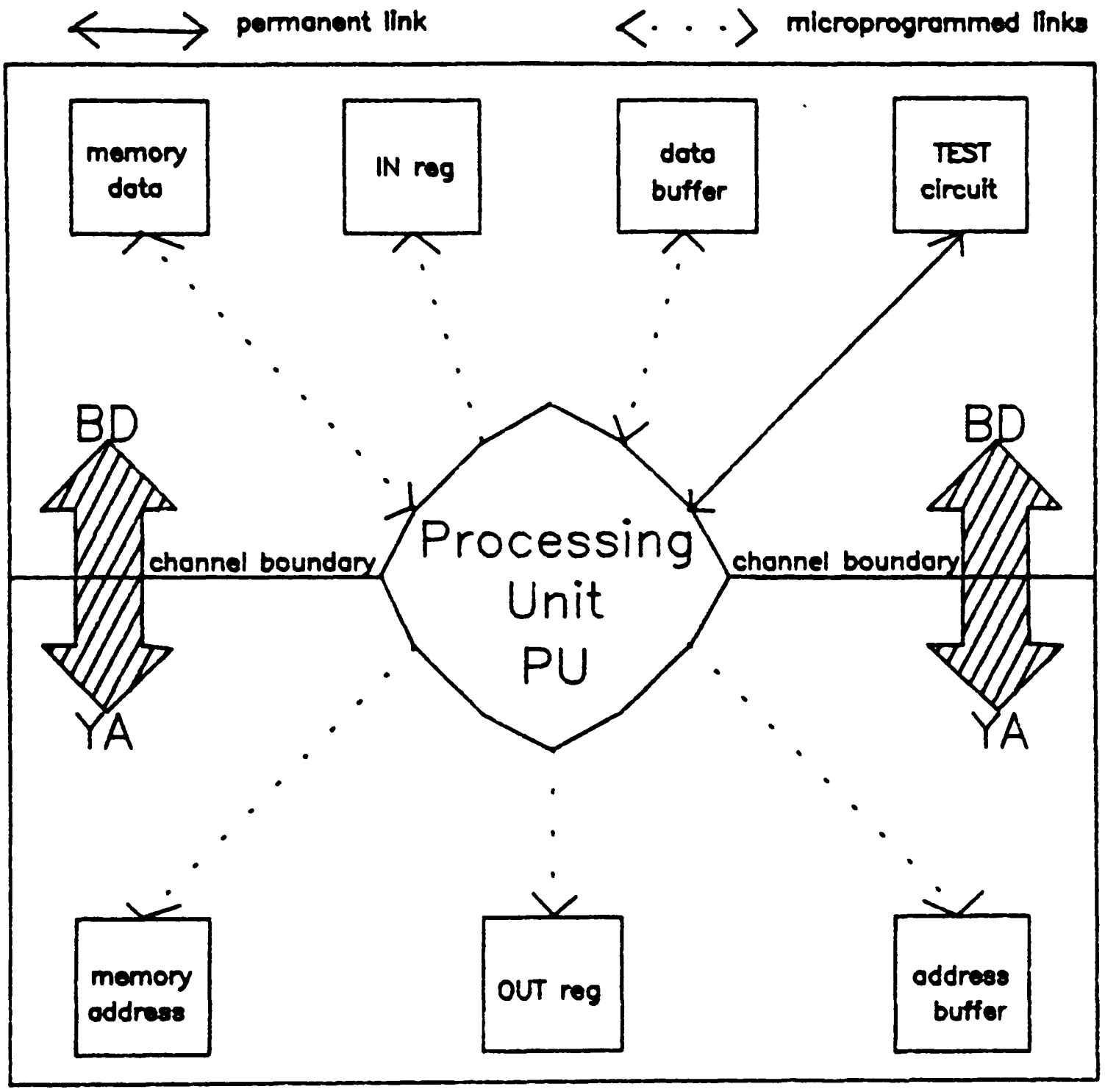


Figure 5.2.4b: The Integration of the PU with the ICMS.

Plate 5 : *The Complete ICMS prototype : (from left to right)*
the M68000 host occupies the first board and
the remaining three together form the ICMS.

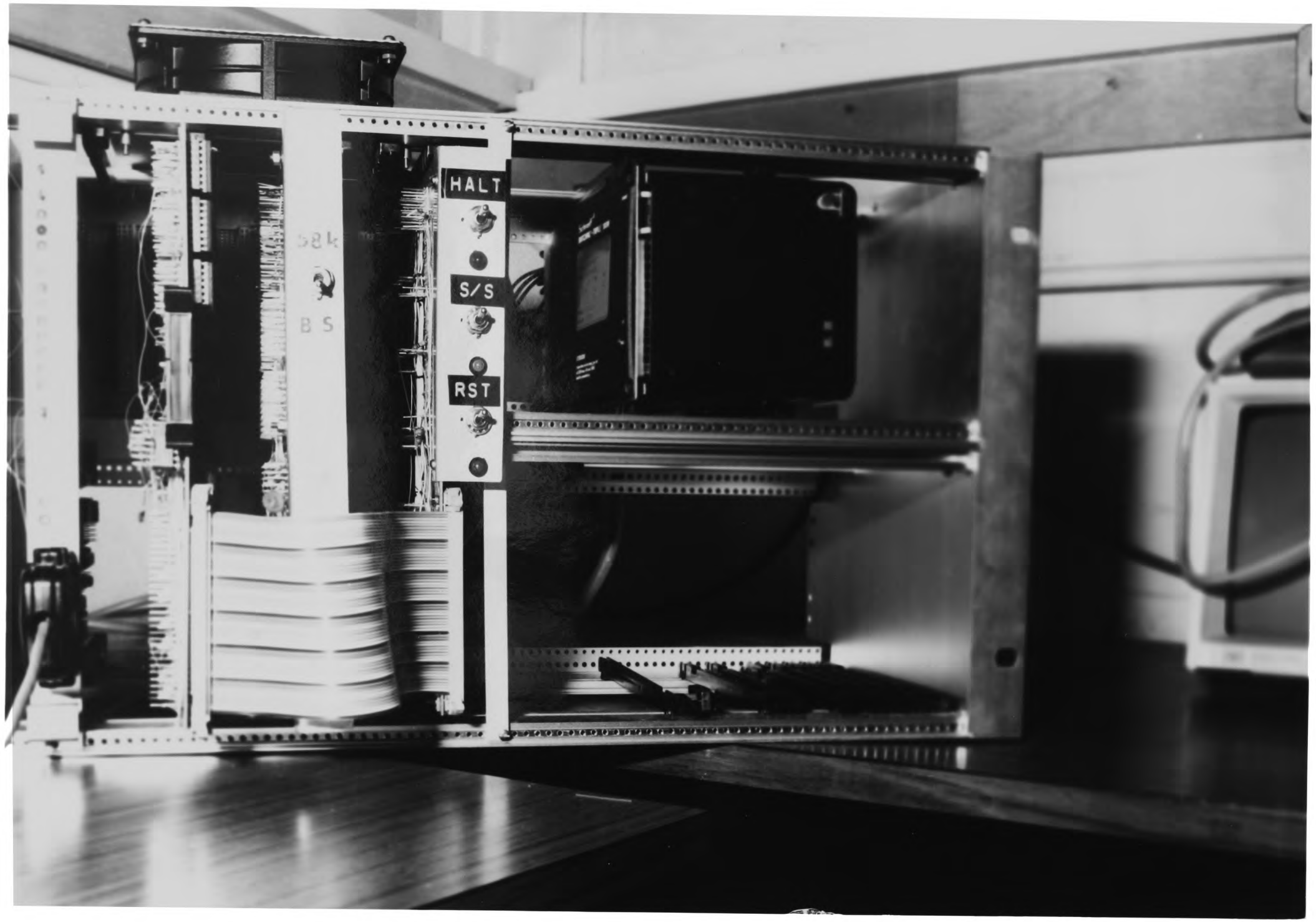

HALT
S/S
RST
58k
B S

Plate 5.1: The ICMS Circuit Board.

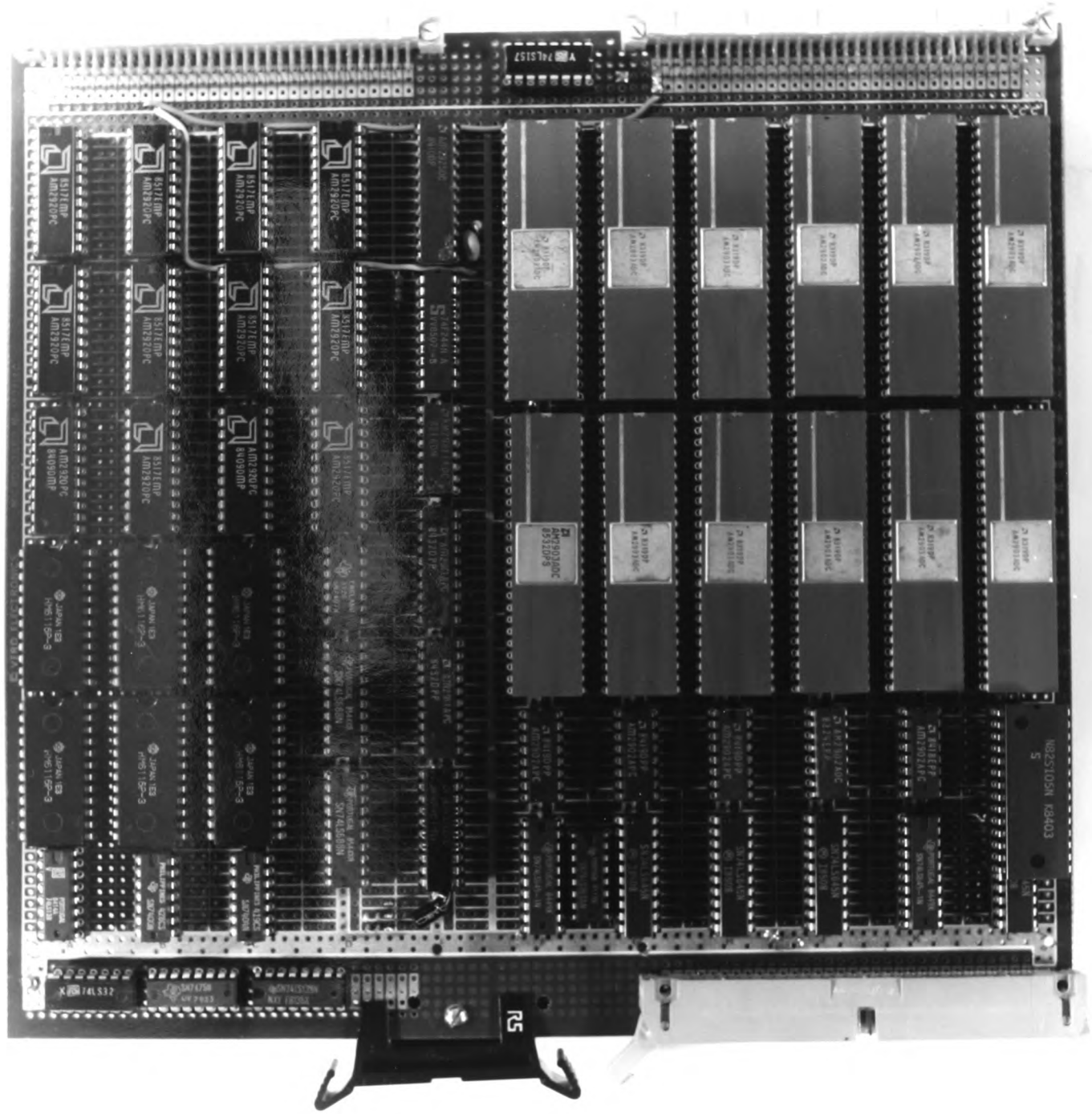


Plate 5.2.1: The Writable Control Store (WCS) Circuit Board.

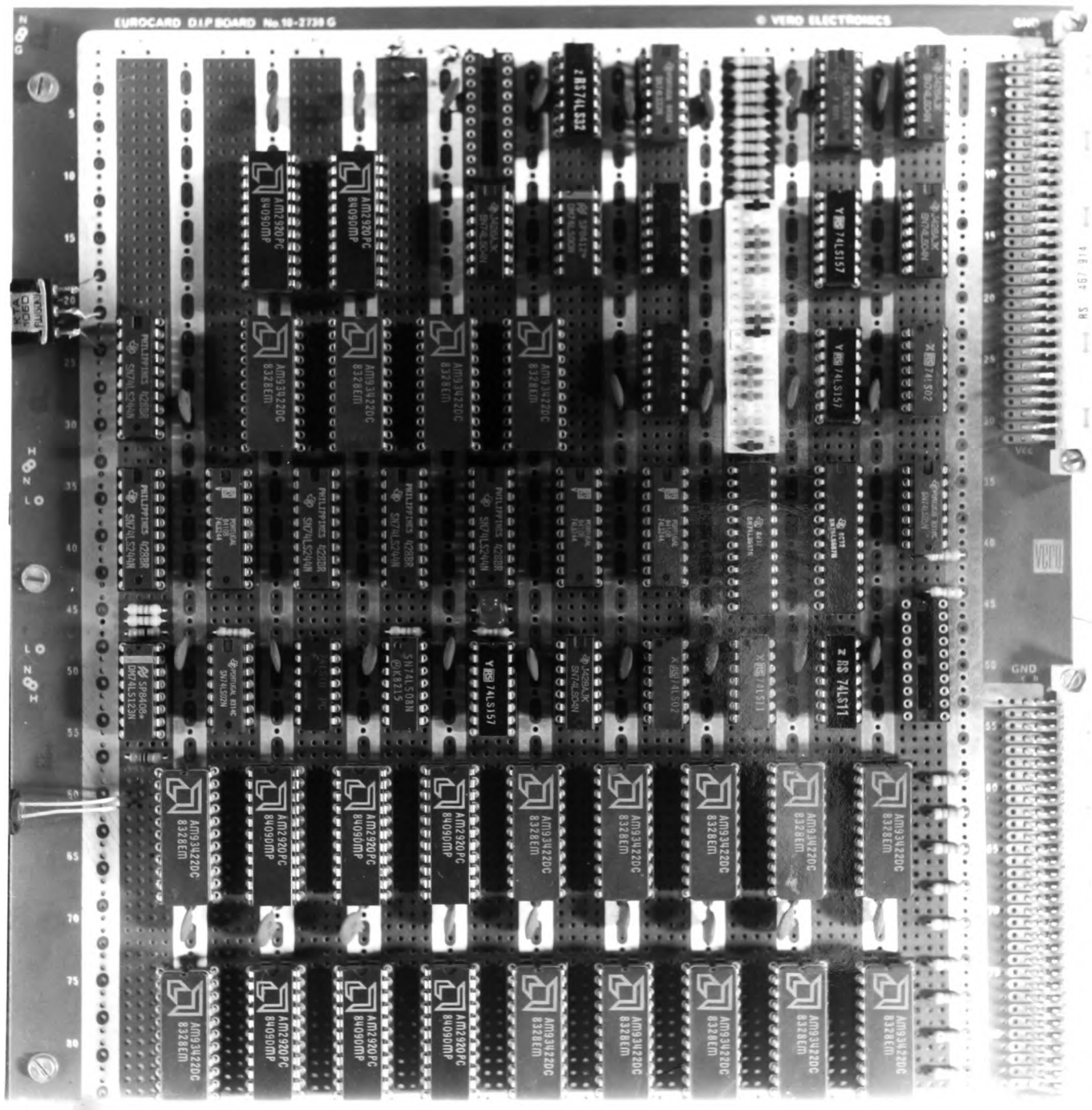
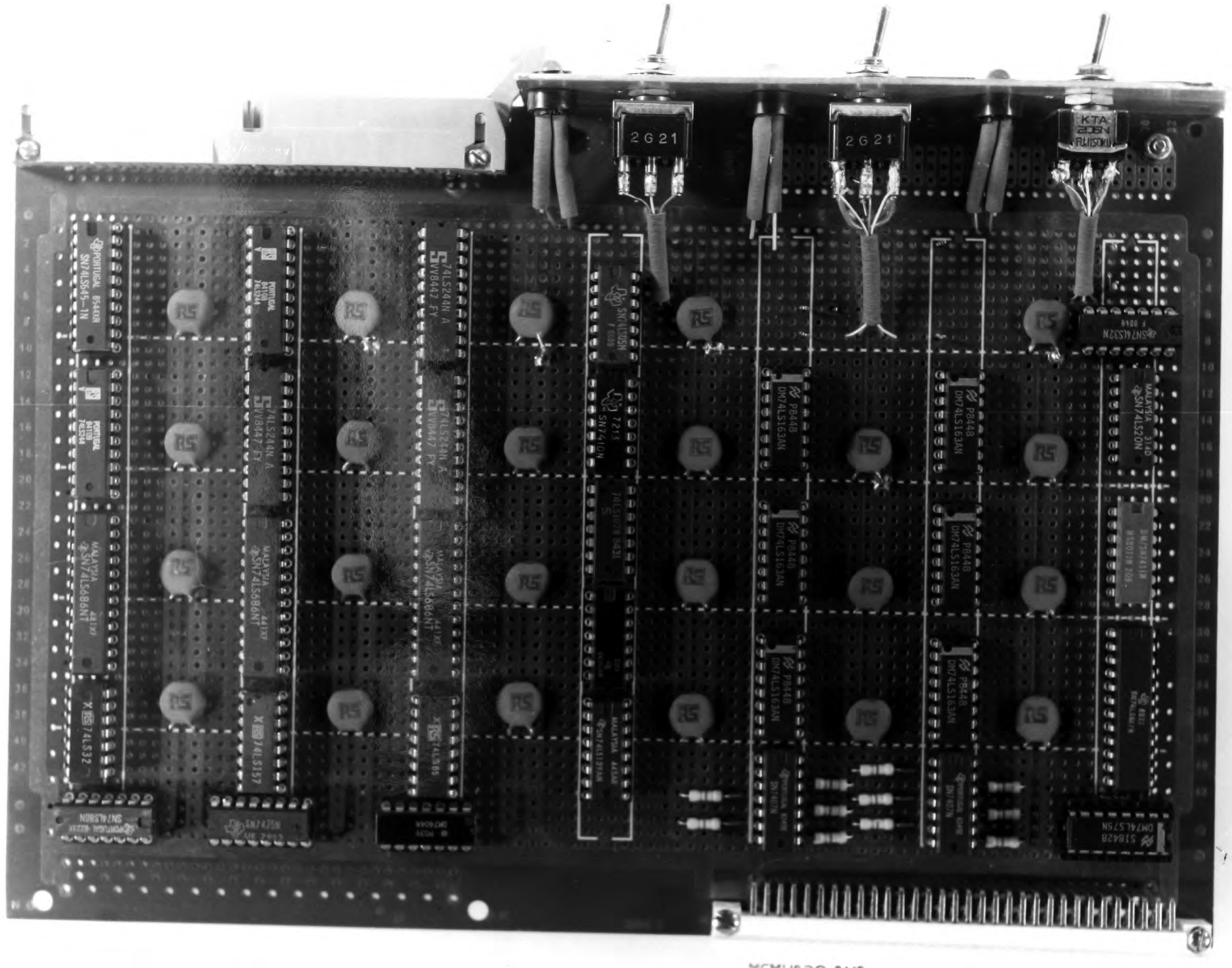

EUROCARD D.I.P BOARD No. 10-2738 G
© VERO ELECTRONICS
AM2920PC
8409DMP
AM93422DC
8328EM
74LS157
74LS02
74LS11
74LS32
GND
Vcc
RS 467 914

Plate 5.2.2: The Circuit Board for the Testing Circuitry (TC).



MCMURDO 8448

TP-100-3-96.PK.11-SG

CHAPTER VI
Construction II : Microprogramming

In order to drive the ICMS, a series of microprograms was produced. They were written using a custom micro-assembly language whose syntax is specified in Appendix I. A microprogram assembler, called *uasm*, has been developed to translate microinstructions into the appropriate control bit patterns.The construction of the microprogram assembler and the implementation of all ICMS microprograms is explained in this chapter.

6.1 Background

The control of bit slice devices [75] is governed by bit patterns which are stored in micro control memory. A sequence controller fetches bit patterns from the memory once every cycle with which it generates or redirects signals to control appropriate slices. The definitions of sequences and functions stored in the micro control memory comprise a *microprogram*. A good bit slice design requires the consideration of an enormous amount of detail, including selection, placement and connection of the chips. In addition, the choice of functions and bit patterns of the microprogram are equally critical. Just as in other fields of computer design, the designer needs good supporting tools. These supporting tools are needed to aid the designer to control the structuring of these bit patterns and to provide freedom for pattern alteration.

6.1.1 High Level Microprogramming

A natural choice of tool is a design language which enables the designer to specify the bit patterns in each microinstruction and hence the entire microprogram.

There has been much discussion in the field of computer engineering, about the level of programming for microprograms. A number of high level languages have been designed, e.g. STRUM, an algol like language, designed and implemented by Patterson [78] which is oriented towards the Burroughs D machine. These languages, however, are heavily machine dependent and hence lack portability. Dasgupta [79] has identified the necessity for high level microprogramming, and has discussed at some length the nature of the problems encountered by language designers. Also, he has suggested future directions in research.

6.1.2 Microprogram Assembler

In view of the unsolved problems in high level microprogramming, assembly level programming [80] is currently the most widely adopted approach. Microprogram assemblers are analogous to ordinary macro assemblers; they are code translators - from mnemonics to machine words.

A microprogram memory word or a microinstruction is unlimited in length and a typical microprogram consists of 1000 instructions or more. A microinstruction is segmented. These segments are called fields and convey control information, such as the microprogram execution sequence, functions of the ALUs, register addresses, timing and enabling conditions for fetches and switches, register pre-loading, instruction pre-fetching, masking, etc. The grouping of bits into fields is a crucial design choice which directly affects the performance of the ultimate system. Techniques of optimal encoding have been an active research topic. The two extremes are:

i) Minimal Encoding.

Each bit of the microinstruction is responsible for one function. In this way,

maximum parallelism is achieved. Thus, an increased throughput is possible. The disadvantage is the requirement for an enormous control store.

ii) Maximal Encoding.

The entire microinstruction is treated as a whole. It is encoded according to 2^b, where b is the microinstruction word size. This method requires a small control memory with the trade off of decreased throughput which is incurred by decoding.

In practice, micro instructions are designed as a compromise between the two extremes. The decision is based on the hardware configuration and the intended macro operation. Typically, a micro instruction consists of several fields. Each of them is designated with a different number of bits. Specifying the content of each field in an assembly language requires multiple assignments. This is equivalent to having multiple opcodes in one instruction. Different opcode patterns might call for different field groupings in successive instructions. A jump instruction, for example, might call for only two fields: the field specifying the jump function, and the other giving the jump address. On the other hand, an ALU operation such as C=A+B, requires several short fields containing codes giving the first and second operand source locations, the code for the ALU operation to be performed, the destination of the result and bits to control the handling of carries and conditional codes.

A special feature of bit slice design is its flexibility: it permits alteration to the architecture of the target machine. The assembler being a design tool for helping to construct the hardware/software combination would be expected to accommodate these changes. This is one of the design problems currently associated with high level microprogram languages. Their lack of "redefinability" has rendered present languages much less useful. This problem can be tackled by developing two phase

(or meta) assemblers. Firstly, in the definition phase lines of specification describing the target machine are read. These include sizes, positions and default values of all existing fields. In the second assembly phase, lines containing field assignments are scanned and translated into the required bit patterns.

6.1.3 Practical Considerations

In bit slice design environments, the assembler may be used by engineers who are not very familiar with the idea of microprogramming. An initial use might be for a very simple microprogram structure which should allow successful assembly without resorting to a large variety of special keywords, formats and constructs. Direct entry of binary contents of a microinstruction should be straight forward. At the other extreme, the same assembler should also allow use by designers who have already familiarised themselves with the reasons for and the means of using relatively advanced tools, such as macros. With such assemblers, the engineer can make more advanced use of the tool easily and naturally, as his applications and his skills become more sophisticated.

6.2 The Local Microassembler

There were no microprogram development tools available so a decision had to be taken whether to purchase these or to develop a customised set. As there did not seem to be any suitable tools available commercially which would run on local computer systems under UNIX, it was decided to develop a microassembler. The facilities provided under the UNIX operating system provided a valuable asset in this work and the resultant assembler is flexible and powerful and capable of being relatively easy modifiable for other bit slice targets.

6.2.1 The Command - Uasm

Uasm is a custom UNIX command developed using shell programming. The invocation of this command is described, in the form of a UNIX manual page, in Appendix I. It accepts a microprogram source and translates it into a loadable format for the M68000 host. Three processing stages are involved in it, namely: M4 → tst → srecord. In the first stage a named source file (suffix .um) is passed through M4, which is a built-in UNIX macro preprocessor [81]. The output of M4 is piped to the second stage, tst, directly. Tst is a microprogram assembler which performs the translation. It is written in yacc [82], a useful software design tool on UNIX. The syntax of the tst micro assembly language resembles the one described in [83] which is written for a bit slice graphics controller interface to a M68000 based workstation. Tst could stand on its own as a translator that allows users to 'touch' the microinstructions. By incorporating tst with the M4 front end, a powerful high level tool has been achieved. An intermediate file is created - um.out, containing the load addresses and the bit patterns in hexadecimal. Finally, um.out is fed into 'srecord,' another custom program written in C, which converts um.out into the loadable S-Record format.

6.3 The Implementation of Tst

Tst is a micro assembler which accepts a microprogram input file (in uasm format, c.f. Appendix I) and translates it into corresponding micro control word patterns and outputs them to a file (default 'um.out').

Tst is a two pass assembler. In the FIRST_PASS† the syntax is checked, literals (strings of alphanumerics, ≤25 characters) are collected in a symbol table and

† in this context, all words in capital letters are defined or key words in the assembler program.

branch addresses are calculated. All kinds of errors are reported in this pass. If there are none, tst proceeds to the SECOND_PASS, otherwise it stops and destroys um.out before exiting with a status 1, which is normally interpreted as incomplete task in UNIX.

6.3.1 First Pass

The FIRST_PASS is further divided into two phases. Phase one is the definition phase in which information defining the machine architecture is given. This includes the size of the control store (breadth and width) and the declarations of all the control fields and their default settings. The maximum width of a control word is 128 bits (4×sizeof(long), 32 bit), exceeding this would cause immediate system abortion. Each control field can only appear once in the definition phase. Any fields occurring twice will generate an error. If the size of the default value of a field is greater than the size of the field itself, the default value is only partially recognised - up to the size of the field.

Labels are entry points for branch instructions. They are identified in the second phase; branch addresses are calculated and assigned to the corresponding labels. These are kept in the symbol table and later retrieved during code generation. Labels can only be defined once; multiply defined labels would generate complaints. Labels are assigned to BA (Keyword: Branch Address) within the microprogram. Forward addressing† is allowed. To achieve this feature, undefined literals which appear in the microprogram are tagged as UNDEFs. Hopefully, UNDEF literals are re-defined as JUMP_ADDRESSes at a later stage. At the end of this second phase all proper entries in the symbol table should have been defined either as FIELDs or

† to use a branch label which is defined later

JUMP_ADDRESSes. UNDEFs are improper and will be picked up in the SECOND_PASS.

6.3.2 Second Pass

The SECOND_PASS is responsible for generating micro control words from the microprogram instructions. Four long words (OUTPUT_PAT) are dedicated to this task and initially, they are set to the default pattern, generated from all the default values arranged in the right order. The exact number of OUTPUT_PAT's used depends on the size of the control word as specified in the definition phase. It is not necessary to use all control FIELDs. The right hand side of a field assignment is read. This could be a straightforward number (binary, octal, decimal or hexadecimal) or an arithmetic/logic expression (+,-,*,/,<<,>>,| and &). The calculated value is shifted to the appropriate position corresponding to the control field. Unused FIELDS are padded out with their defaults.

During the translation process the address of the current micro word (DOT) and the physical memory address (PM) are recorded. Effectively, DOT is the micro-address viewed from the microprogram sequence controller; whilst the physical address is for down loading into the Writable Control Store (WCS) from the host (in this case M68000). Branch address calculations are performed using DOT. DOT may not be the same as the physical address of the memory depending on the setting of NFLAG. NFLAG is a variable which is set by the '-n' optional flag, included in the command invocation. If NFLAG is equal to 1 , DOT is incremented by 1 otherwise it is the same as PM, incremented by the number of bytes taken up by a micro word, after each microinstruction. The number of bytes occupied by the line together with the value of PM is written onto an output file 'um.out', followed by

the translated micro word.

Before exiting with a 0 status, a message is displayed on the screen informing the user of the number of micro words generated and all open files are closed.

6.3.3 Error Messages

Error messages are printed on the standard error output (STDERR). Errors occurring in the first phase are marked with "Definition error" at the beginning of the message followed by the description of the specific error followed by the line of occurrence. In the second phase, messages take a similar pattern with "Statement error" followed by the error message followed by the instruction number at which the error occurs. Because microinstructions extend over more than one line, to print errors by their line number is virtually meaningless.

6.4 Microprograms for the ICMS

In the microprograms developed for control of the ICMS, each microinstruction is 64 bits wide and is segmented into 35 different fields. The grouping of these fields and their practical meanings are summarised in Table 6.4.

The microprograms† for the ICMS were developed in a hierarchical fashion, see figure 6.4. The nuclei of all micro routines are based on the memory access instructions, namely

_mem_write(DATA_REG,ADD_REG,ADD_ROUTE); and

_mem_read(DATA_REG,ADD_REG,DATA_ROUTE,ADD_ROUTE).

Data and addresses of these operations are derived from the 16 internal registers, as specified in DATA_REG and ADD_REG. The last (16th) internal register is

† Throughout the thesis, micro instructions are preceded by "_", followed by the instruction names (in capital letters); parameters are embraced by "(" and ")" at the end of the names.

constantly regarded as the data transfer buffer. The data to be written/read is first transferred to/from them and the outputs are then gated. There are five possible routes of data and addresses to the cell memory through the control of the bi-directional buffers with the parameters DATA_ROUTE and ADD_ROUTE. Timing difficulties arise because of the use of ordinary "slow" RAM for the cell memory. This is overcome by stretching the clock, as shown in figure 6.4a. The mark/space ratio and the period are specially chosen to satisfy the worst case requirements. The worst case set up time for the ALU is 50 ns and the memory access time of the RAM (6116-2) is 120 ns. Therefore the worst case access for writing is 50+120=170ns. An extra cycle is needed for reading because the data fetched from the memory is going to be stored in the ALU; this gives 50+120+50=220ns.

The other routines include, _COPY(X) which copies X (a FROMSPACE pointer) into the TOSPACE location pointed at by the B pointer. A conditional _COPY is also provided, _MOVE(X); in this case, the origin of X is tested and it is copied if it is a FROMSPACE pointer otherwise it is left intact. The _SCAN micro routine which is based on _MOVE is the unit operation for garbage collection. The GC test signal is constantly monitored at the beginning of a garbage collection cycle. During this the ICMS is vulnerable to interruption. When GC is high and there is no request for a list operation, another _SCAN is performed. Normally, the function of _SCAN is to apply _MOVE to the HEAD and TAIL elements of cell addressed by the S pointer. In a special case when the system has just flipped, which happens rarely (only once every M allocations, where M is the total number of cells in the TOSPACE), internal registers 0 to 7 (the roots) are _MOVEd and their contents are updated. The final stage of microprogram development was the realisation of five list manipulation primitives: _CAR(X), _CDR(X), _REPLACAR(X,Y),

_REPLACDR(X,Y) and CONS(X,Y). _MOVE is applied to the parameters before they are loaded into the corresponding functional IO registers. There are also several supporting micro-subroutines:

- _INCB/_READB - increments and reads the B pointer.
- _INCS/_READS - increments and reads the S pointer.
- Eight conditional flag selectors:
  1) _IF_POLLED - has the host requested a LISP function ? (This was only used at the beginning. Polling has now been replaced by interrupt.)
  2) _IF_CELL - is it a cell pointer ?
  3) _IF_GC - has garbage collection completed ?
  4) _IF_TO - does a cell pointer exist in the TOSPACE region ?
  5) _IF_FROM - does a cell pointer exist in the FROMSPACE region ?
  6) _TRUE - always true or '1'.
  7) _FALSE - always false or '0'.
- Eleven conditional/nonconditional branch microsubroutines which control the execution sequence explicitly according to the result of a conditional flag selector :
  1) _JZ - branch† always to the the first location of the WCS. This, usually, is the first instruction of many bit slice microprocessors. The startup vector is placed in the first location of the store.
  2) _JSR - jump to subroutine unconditionally. Before jumping, the present

† Branch and jump are used differently here for better explanation. A branch is a "GOTO" operation. A jump is associated with subroutine calls in which return addresses are saved.

value of the program counter is pushed onto the NAG's (Am2911A) 16-level LIFO memory. The subroutine address is specified by the @BA field.

3) _CJSR - conditional jump to a subroutine if the result of the conditional flag selector is high.

4) _JMAP - branch always to the point specified by the output of the MAP ROM.

5) _JHI - branch to the point specified by @BA if the result of the conditional flag selector is high else continue.

6) _JPR - if the conditional flag selector returns low the present execution will be repeated; otherwise a branch is taken to @BA.

7) _JLO - branch to @BA if the conditional flag selector returns low.

8) _RTN - return from subroutine.

9) _CONTINUE - execution continues from the program counter.

10) _NOP - no branching, same as _CONTINUE.

11) _JBA - branch always to @BA.

- INPARA0(x)/INPARA1(x) - read in the content of the input register (IN0/IN1) and put into the xth internal register.
- OUTPARA3(x)/OUTPARA4(x) - write the content of the xth internal register and put it into the output register (OUT2/OUT3).
- _BERR - assert $\overline{\text{BERR}}$ onto VMEbus.

6.4.1 The LISP Primitives

Five LISP primitives are provided by the ICMS firmware, they are :

i _CAR(X)/_CDR(X):- This primitive extracts the HEAD/TAIL element of the cell pointer X. To invoke this function the host writes the content of X into the CAR/CDR register (FF8080/FF8088, cf Table 5.2.3a). Part of this address (FF8000, Bit_{8-23} & Bit_{0-2}) addresses the first of the two input registers. Immediately after the last write cycle to the register, an interrupt is generated to the ICMS. The remaining bits of the address (80/88, Bit_{3-7}) are used to address the MAP ROM so that the PU starts executing from the _CAR/_CDR micro-subroutine when the ICMS has finished the present garbage collection cycle. Upon execution, the PU reads in X from the input register, ensures X is a cell pointer, passes X into micro-subroutine _MOVE which copies X from one region to the other if required and updates all pointers, extracts X's HEAD/TAIL element, writes it to the output register and instructs the host that the answer is available by asserting $\overline{DTACK}$. $\overline{DTACK}$ is asserted by setting micro bit @BUSY to '1'.

ii _REPLACAR(X,Y)/_REPLACDR(X,Y) :- This primitive replaces the HEAD/TAIL element of cell Y to X. To invoke this function the host writes the contents of X and Y into the REPLACAR/REPLACDR registers (FF8094 & FF8090/FF809C & FF8098, respectively). The order of parameter setting is important, X has to be written before Y. Immediately after the last write cycle of Y, an interrupt is generated to the ICMS. $Bits_{3-7}$ (namely, 90/98) direct the PU to execute the _REPLACAR/_REPLACDR micro-subroutine after the present garbage collection cycle is finished. Upon execution the PU reads in X from the second input register and passes it to the _MOVE subroutine; then the

PU reads in Y from the first input register and passes it to the _MOVE subroutine. After X and Y have been _MOVEd, the PU places X into the HEAD/TAIL of the cell location referenced by Y and asserts $\overline{\text{DTACK}}$.

iii _CONS(X,Y) :- This primitive creates a new cell and places X and Y into its HEAD and TAIL elements, respectively. To invoke this function the host writes the contents of X and Y into the CONS registers (FF80A4 & FF80A0, respectively). Immediately after the write cycle of Y, an interrupt is generated to the ICMS. $\text{Bits}_{3\text{-}7}$ (A0) direct the PU to execute the _CONS micro-subroutine after the garbage collection cycle is completed. Prior to creating a new cell, X and Y are passed into _MOVE. To create a new cell, the B counter is incremented by the _INCB micro-subroutine and the value of the B is read (_READB) into the 16th internal register. Finally, X and Y are written into the cell location pointed at by B and $\overline{\text{DTACK}}$ is asserted.

The complete microprogram for controlling the ICMS is listed in Appendix IV.

6.4.2 The Microprogram Structure

The structure of the ICMS microprograms is very modular. Subroutine calling is extensively used. This is facilitated by the NAG (next address generator, Am2911A) which has a 16 internal LIFO register file for stacking return addresses and the 8 conditional/unconditional jump to subroutine instructions. Parameter passing is achieved via the last eight registers of the ALU. These registers are also used as the scratch pad. The 16th register is most heavily used because of its role as the data transfer buffer during memory access. This is restrictive because only up to eight parameters can be passed and their positions are fixed. During nested calls the value of the shared registers has to be saved and conversely, restored after returning.

Therefore, although the NAG architecture supports nested subroutines, it is only effective for non-parametric calls. Some external register files could be added to the ALU thus providing extra work space. Moreover, a parameter stacking mechanism could be incorporated, which would provide dynamic allocation of internal registers and save the ALU from saving/restoring during nested subroutine calls.

At present, the size of the Writable Control Store (WCS) is 2k bytes. There are 64 bits in one ICMS microinstruction word. This implies that the logical address space of the WCS is 256 words. The modular structure of the ICMS microprogram has resulted in 116 microcode words and they are comfortably fitted into the WCS.

6.4.3 Error Handling

There is a small amount of error handling capability built in with the microprograms. When an error is encountered a $\overline{\text{BERR}}$ is asserted on the VMEbus. This is, however, very crude and the source of error is not specified. This could be overcome by utilising the VME interrupt mechanism: i.e. assign interrupt levels (0-7) and vectors to individual sources. There are mainly two types of errors, namely:

Parameter errors - the Xs and Ys supplied to the list primitives are not the proper type; for example, it is illegal to apply _CAR(X) if X is not a cell pointer.

System error in cell allocation - supply is less than demand in cell allocation. In the ICMS, this happens when the GC test signal is high at the moment of FLIPping.

The second error is fatal. It is, in fact, a violation of Baker's second principle (section 3.4.2) which states that there should always be at least one free cell available at the instant of a cell request. In practice, it is possible to have some temporary cells

created while the garbage collection process is still ongoing. These are called floating garbage. The non-deterministic nature of dynamic memory allocation provides no means of foreseeing the amount of floating garbage that needs to be created. The condition for a machine to remain functional is derived as follows :

Let M be the total number of cells in TOSPACE, N be the number of inaccessible cells at the beginning of a FLIP and k be the number of garbage cells being collected at one burst. Accordingly, the number of cycles required for completion of garbage collection is $\frac{N}{k}$. After the $\frac{N}{k}$th cycle, $\frac{N}{k}$ new cells are created. To ensure that the memory is not over populated, which would lead to system failure, the following condition has to be satisfied:

$$\text{number of new cells} < \text{number of free cells}$$

$$\frac{N}{k} < M - N$$

$$\frac{1}{k} < \frac{M-N}{N} \tag{6.4.1}$$

M is fixed and N is fairly constant in the steady state, therefore the larger k is the more likely condition 6.4.1 is to be satisfied. Applying Baker's algorithm on a serial computer, when k=M, is the non-real-time realisation. Thus,

$$\frac{1}{M} < \frac{M-N}{N}$$

$$\frac{M+1}{M} < \frac{M}{N} \tag{6.4.1a}$$

This is always true because $M > N$, so the left hand side of 6.4.1a is ≈ 1 and the right hand side is ≥ 1. In fact for any k which is $\geq N$, a machine will be safe from system error (providing $M > N$). Nevertheless, more work will have to be done on each garbage collection burst for a large k; which would then result in a sluggish response time. On the other hand, when $k \rightarrow 0$, less and less effort is needed for each cycle;

thus ensuring prompt machine response. The trade off is to expose the machine to system error.

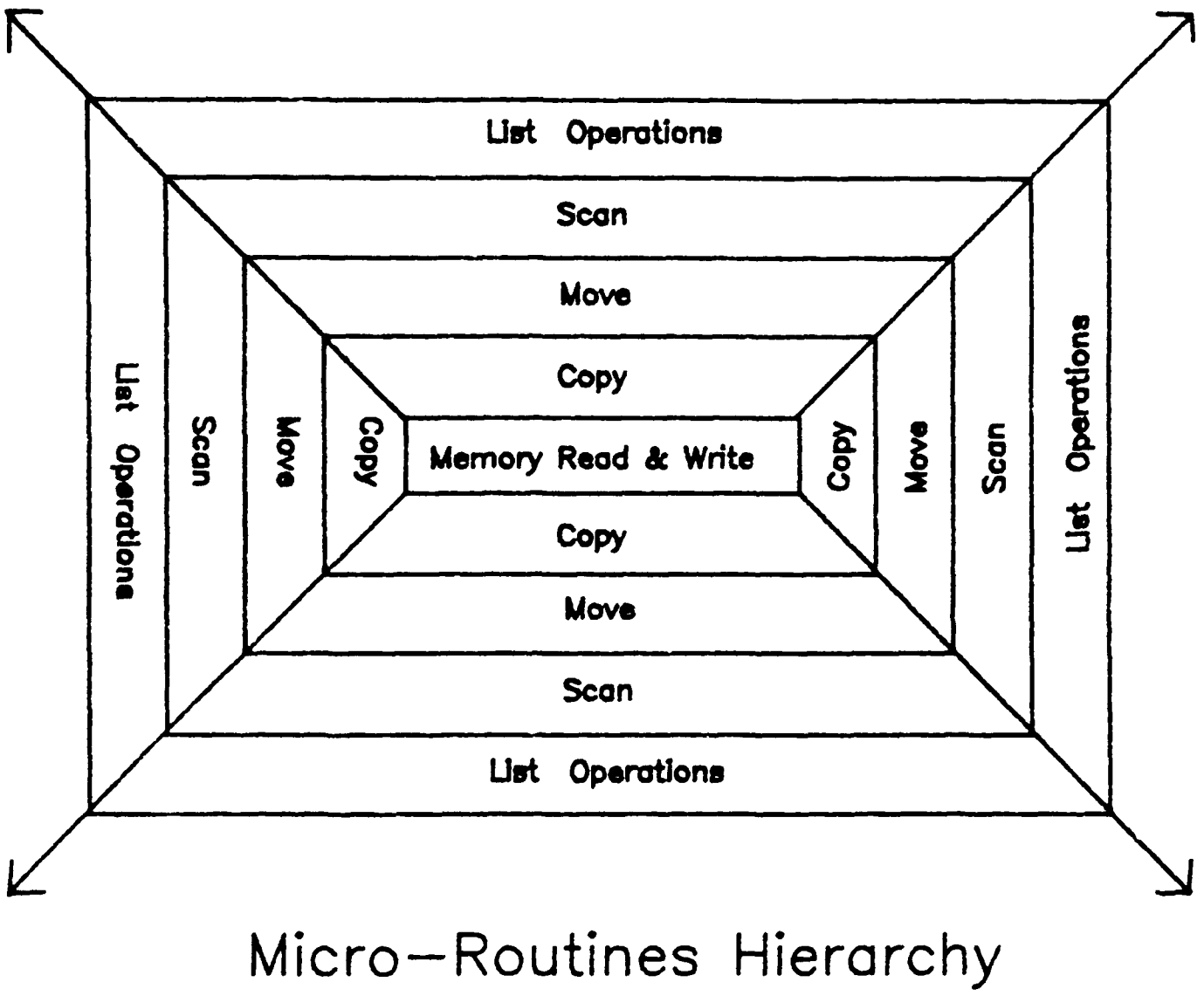
The advantage of the ICMS scheme is apparent at this point. A dynamic k is possible under a mixed real-time and incremental collection strategy. For as long as there is no list operation requested, garbage collection is performed in parallel therefore it takes less than $\frac{N}{k}$ cycles before the task is completed. Effectively, a large k is established. However, the garbage collection process is transformed into the incremental mode when list operations are requested. This ensures $k \rightarrow 0$ and machine response time remains bounded and tolerablely short. Therefore, by employing an ICMS based machine, the rate of system error is minimised and fast system response is guaranteed.

Fields	Bit no.	Default	Operations
BA	0-7	0	branch address
seq_instr	8-11	0	instruction control of 29811
RE2911	12	1	register enable of 2911
OE2911	13	0	output enable of 2911
IOsel	14,15	0	selection of IO registers : 00 read IN register 0 01 read IN register 1 10 load OUT register 2 11 load OUT register 3
OEy_h	16	0	output enable of HEAD ALUs
OEy_t	17	0	output enable of TAIL ALUs
addA	18-21	0	selection of the dual-ported internal ALUs' registers at port A
addB	22-25	0	selection of the dual-ported internal ALUs' registers at port B
data_buf_dir	26	0	control of the routes on the data path : 0 1
data_buf_en	27	1	enable of the data buffer; disable (1) implies separate routes of data
add_buf_dir	28	0	same as bit27 on address path
add_buf_en	29	1	same as bit28 on address path
ALU_instr	30-38	all 1's	function control of ALUs
A0_139	39	0	special decoder (SD) control (see later)
mem_read_h	40	1	HEAD memory read/write control : 0 Write 1 Read
mem_read_t	41	1	TAIL memory read/write control
mem_sel_h	42	1	HEAD memory select
mem_sel_t	43	1	TAIL memory select
oeb_h	44	1	enable output of HEAD ALUs registers at port b
oeb_t	45	1	enable output of TAIL ALUs registers at port b
ea_h	46	1	enable output of HEAD ALUs registers at port a

Table 6.4a: The Grouping of ICMS Microinstruction Word.

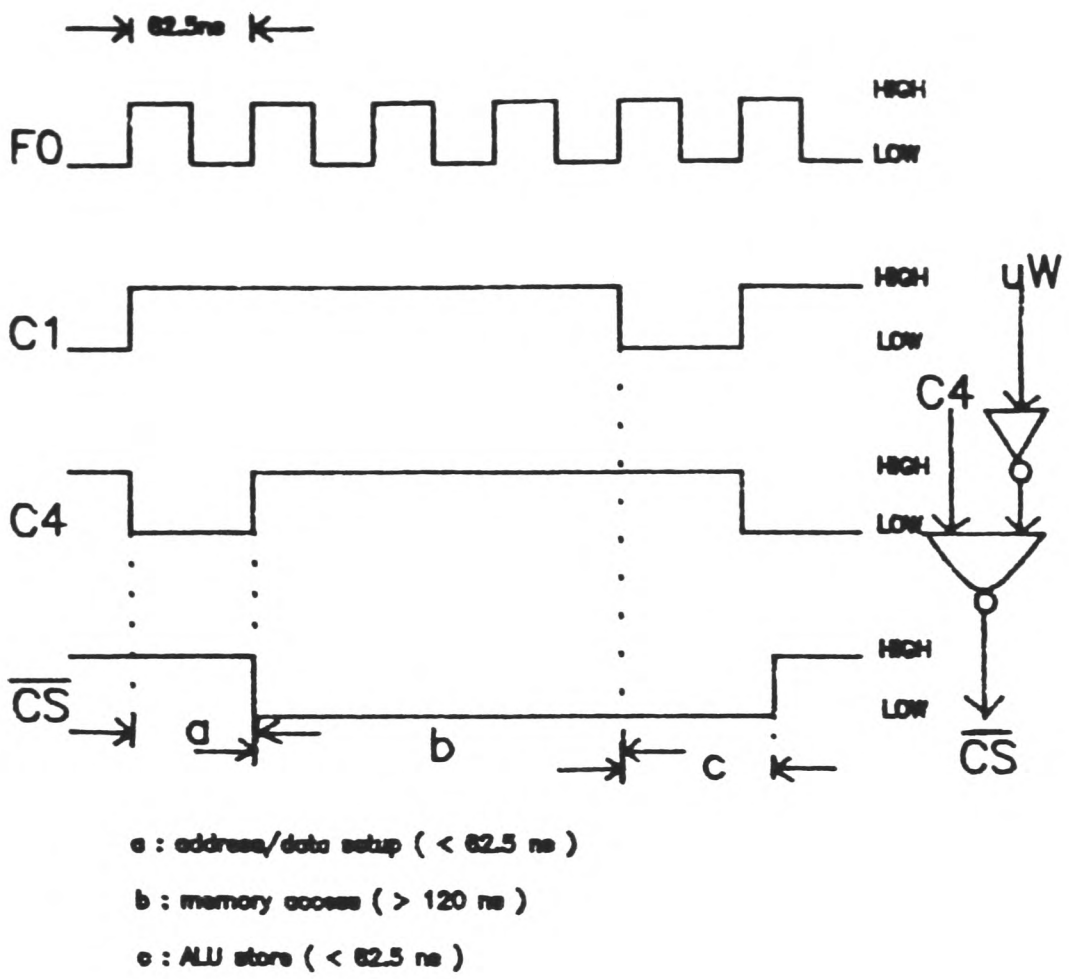
ea_h	47	1	enable output of TAIL ALUs registers at port a
IEN_h	48	1	instruction enable of HEAD ALU; when 0 internal register can be written at port b
IEN_h	49	1	instruction enable of HEAD ALU; when 0 internal register can be written at port b
A1_139	50	0	Special Decoder control (see later)
carry_h	51	0	carry of HEAD ALUs
carry_t	52	0	carry of TAIL ALUs
clock	53-55	0	clock rate select
sel_139	56	0	select of the Special Decoder (SD) : When sel_139 = 0 A0 A1 0 0 Default no action 0 1 Latch in test signals 1 0 Not used 1 1 Bus Error When sel_139 = 1 0 0 increment S ptr 0 1 read S ptr 1 0 increment B ptr 1 1 read B ptr
mem_oe_h	57	1	HEAD memory output enable
mem_oe_t	58	1	TAIL memory output enable
TESTsel	59-61	0	select output from the conditional multiplexer : 0 POLLED 1 CELL 2 GC 3 TO 4 FROM 5 FLIP 6 TRUE 7 FALSE
INT	62	1	68K-ICMS interrupt enable
BUSY	63	0	DTACK when 1

Table 6.4b: The grouping of the ICMS microinstruction word.(cont'd)



Micro–Routines Hierarchy

Figure 6.4: Stages of Microprogram Development



Operations	times (ns)	
@mem_write:	address/data setup	< 62.5
	memory access	> 120
		4 cycles = 250
@mem_read:	address/data setup	< 62.5
	memory access	> 120
	ALU store	< 62.5
		5 cycles = 312.5

Figure 6.4a: The Timing of Memory Access Cycles.

CHAPTER VII
Verification and Results

7.1 Verification

At the first stage of the project, both non-real-time and incremental versions of Baker's garbage collection algorithm were studied. The main objective was to prove their correctness. Apart from the necessary changes to account for the difference in cell definition, the realisation of the algorithms was simply a task of translating the original algorithms from ALGOL [63] into C. To assist this verification exercise, a simple LISP interpreter - simpleLISP - was written. It is an interactive variant of MACLISP [59] whose construction details are explained in appendix II. Although simpleLISP† is crude, it contains sufficient LISP primitives for the purpose of this project .

While simpleLISP is running, there are always two lists being maintained. A global list, or the program environment, contains all user-defined functions (created by defun) and objects with their associations (created by setq). Newly defined functions and objects are added to the top of the existing list; and, on the other hand, redundant ones are deleted by applying the primitives undef and unset, respectively. The other list is a temporary entity, the local environment, which is maintained during a user-defined function call. It consists of the run-time associations of all parameters and has a link to the global program environment at the time when a function is invoked. Effectively, it operates as a stack which facilitates dynamic late binding.

---

† The LISP primitives supported by simpleLISP are : add, append, assoc, atom, car, cdr, cons, cond, defun, difference, lambda, null, replacar, replacdr, setq undef, unset and zerop.

The information contained in both lists is essential and therefore their originating pointers are stored in the system's root registers. A practical criterion of the correctness of this implementation is that the integrity of both lists is retained without corruption for as long as simpleLISP is active and is subjected to, ideally, an infinite number of garbage collection cycles. In order to do this the familiar recursive function, fibonacci, was defined:

```
( defun fibonacci (n) ( cond ( (zerop n) nil) ( (equal n 1) nil ) ( t ( add ( fibonacci (
difference n 1 ) ) ( fibonacci ( difference n 2 ) ) ) ) ) )
```

This function was called many times. The characteristic of it is that the amount of cells required increases with the value of n. If n is large, at some point in the middle of an execution, the cell memory would be exhausted. Had information contained in both environments not been preserved, invalid answers would be the likely consequence.

Initially the exercise was scheduled on a VAX 11/750, then it was repeated on a M68000 with and without the ICMS. In all cases, both environments were never corrupted which suggested that Baker's garbage collection scheme has been successfully implemented. At a later stage, some timing measurements were taken (c.f. section 7.4). For ease of implementation, the simpleLISP cell space was installed with two thousands cells only (split equally between the TO- and FROM- SPACE). Unfortunately, this limited cell memory size caused the test to fail when $n \geq 9$. This was because fibonacci(9) consumes over 1000 cells in simpleLISP, more than the system could afford.

7.1.1 Graphics

In addition a graphical tree traversal program module was written which was

added to the existing software. The intention of writing this module was to provide extra information on the low-level operation of the collector. The details of the program are presented in appendix III. Practically, the module serves two functions :

i) to display an input list graphically in binary tree format;

ii) to reveal the route taken by the garbage collector during its execution.

Baker's algorithm is a copying type collector, and the routes taken by the garbage collection process are completely different from those in mark and sweep algorithms. In order to demonstrate this fact, a link-reverse mark and sweep collector was written, algorithm E in [56]. Applying the graphical module to Baker's and Knuth's, the routes of garbage collection were revealed, respectively, to be "breadth first" - cell inspection proceeds from left to right and roots to atoms, and "depth first" - cell inspection proceeds from a tail until an atom is reached and then resumes again from where it was diverted. This graphical module will be very useful for helping new comers to learn the concepts of list manipulation and garbage collection.

7.2 Results

Within the implemented primitive set, there are five functions which are directly concerned with Baker's algorithm (c.f. Chapter III). They are :

cons(x,y) create a cell and construct a list with x being the head and y being the tail;

car(x) and cdr(x) return the head and tail element of the cell x, respectively;

replacar(x,y) and replacdr(x,y) replace the head or tail element of the cell x with that of cell y, respectively.

The first experiment was to obtain the times required by each of these functions on a M68000, with and without the ICMS. The equipment used for this exercise was a

HP logic analyser (1630G), shown at the right of Plate 7.2 which is a photograph of the complete experimental setup. Before entering the functions, a flag was set and similarly, just after return from the calls, another flag was set. Using two analyser probes, one connected to each flag, provided the required information. Each function was called in turn and their execution times were recorded.

The results are tabulated in table 7.2. The worst case set up (row 1) happens when garbage collection is active and the parameters supplied to the functions reside in the FROMSPACE. When this occurs, extra effort is required to copy these parameters and to ensure that the return values are genuine (e.g. car could have returned the forwarding address of a moved cell). Therefore, as expected, the execution times for set up 1 are longer than those of 2; and for all functions, the amount of extra time required is approximately doubled. Set up 3 is configured with simpleLISP mounted on a M68000 interfaced to the ICMS which contains the functions in microcode. The difference between the worst and normal cases are unnoticeable because a garbage collection burst (a scan operation) is short as compared to the actual functions themselves. Rows 5 and 6 indicate the speed improvement between the M68000 with and without the ICMS in the worst and normal situations, respectively. Clearly, when interfaced with the ICMS, the M68000 performs much faster. For cons which involve several cell copying operations the improvement is most noticeable.

7.2.1 A High Level Evaluation

Since the five primitives tested were elementary, higher level functions can be built from them. Having shown that individual primitives are executed faster thus demonstrating the advantage of the ICMS, this fact may be justified further showing that the improved figures in table 7.2 are realistic. A similar experiment as

described in section 7.1 was performed. A user-defined fibonacci function was invoked with various parametric values. The results are given in table 7.2.1.

The numbers of cons, car, cdr, replacar and replacdr which have been called in fibonacci are directly proportional to n. Consequently, the improvement factors for all cases are fairly constant (~ 15). The variations are due to occasional bus delays. As mentioned in section 7.1, the evaluation of the fibonacci function requires an rapidly increasing number of temporary cells. These cells are mainly for late-binding of parameters and the numbers required increase monotonically with n as shown in column two of table 7.2.1. When this number is greater than or equal to the amount of cells available, (>=1000 in simpleLISP, i.e. fibonacci(9)), the system fails.

7.3 Baker's GC on the ICMS

A final experiment was carried out to show that the ICMS is superior to the version of Baker's algorithm running on a serial computer (M68000). This was done by creating global environments, whose size varied from 1% to 99% in 10% increments, and then 1000 cons functions were called. In so doing, at the 1000×(100-n)th cell requests, garbage collections were initiated, where n is the percentage of accessible cells with respect to the total number of available cells (i.e. 1000 for the ICMS). The measured times are tabulated in table 7.3.

Since the task of Baker's algorithm is to copy all accessible cells from the FROMSPACE to the TOSPACE, it is not surprising to see that as the number of accessible cells increases, the time required for garbage collection also increases. Notice that throughout the experiment, the ICMS times remain constant while the 68K times increase monotonically. Consequently the relative performance, which is

a ratio of the two, also increases.

At some point, the experiment failed for the M68000 alone set up. From 1% to 50% full, the system operated normally and the execution of the cons experiment involved only one garbage collection cycle. For 60% onwards, the system behaved differently. When the system was 60% full, two garbage collection cycles were required. At the beginning of the experiment, before the first cons operation, there were 400 free cells available. After 400 cons, the system entered into garbage collection for the first time. For the M68000 alone case, three garbage collection bursts (the scan quota) were performed for every invocation of cons. Theoretically, the worst case of 200 cons operations is required for a list of 600 cells. After the required 200 cons, the system was then 800 cells full (600+200). Finally, at the 200th of the remaining 400 cons operations, the system flipped again. Therefore, it can be seen that the execution time for the 60% case is slightly longer. Because of the same reason, the system crashed from 70% onwards. For the 70% full experiment, the system required (700÷3) ~266 cons in order to recover. Nevertheless, the system failed at the 250th cons. At which point, (4x250) 1000 cells existed, but garbage collection had not yet finished. Technically, system failures in these cases were caused by:

(S < B) and (the system has just flipped).

This predicament was not suffered by the ICMS because garbage collection was performed both in parallel and incrementally. Parallel collection operates in between cons - for instance, during parameter setting. In so doing a fast rate of cell recovery is guaranteed; thus avoiding the cell shortage problem when the M68000 operates alone. At the same time, short and constant garbage collection bursts are ensured; they are performed incrementally, as part of the cons operations. The dual operation

mode of the ICMS is an advantage for engineering applications. Effectively, this experiment is, in fact, an experimental justification of expression 6.4.1, with a dynamic k.

$$\frac{1}{k} < \frac{M-N}{N} \tag{6.4.1}$$

where $k \rightarrow M$ for parallel mode and $k \rightarrow 0$ for incremental mode (cf Section 6.4.3).

7.4 Conclusions

The results presented above clearly demonstrate the very considerable improvement in performance provided by the ICMS design. Such performance would obviously be very attractive in real-time engineering applications of AI.

#	Set Up	car/cdr	replace	cons
1	68K alone (worst case)	820.8	1392.0	2971.0
2	68K alone (normal case)	293.0	581.4	1361.0
3	68K with ICMS	57.1	65.2	53.8
4	Worst case Improvements (1+3)	12.6	25.9	55.3
5	Normal case Improvements (2+3)	5.1	8.9	25.3

N.B. All times measured in micro seconds (μs).

Table 7.2: Execution times of the basic LISP primitives.

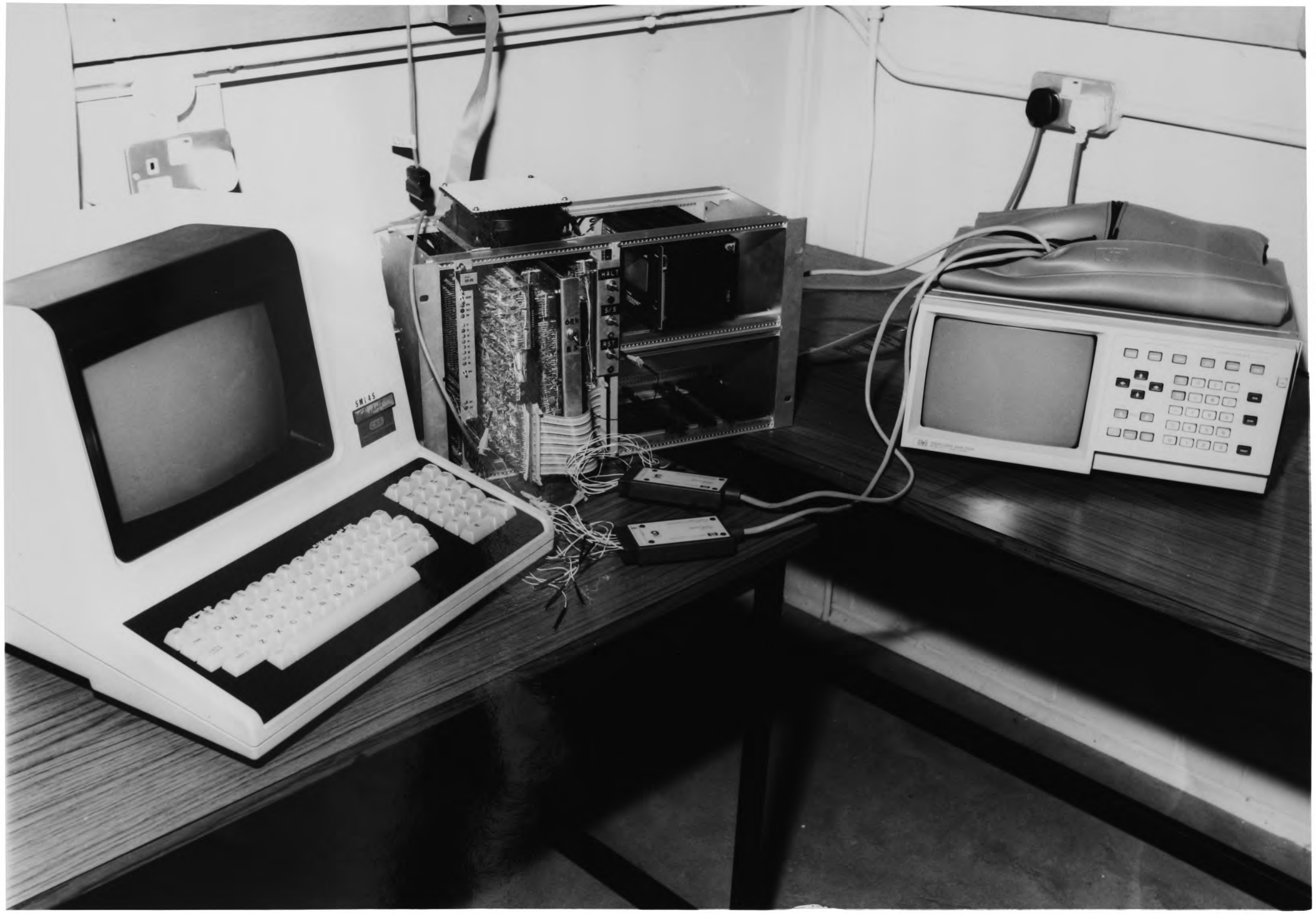
fibonacci(n)				
n	cells consumed	68K alone (ms)	68K+ICMS (ms)	Improvement ratio
0	20	23.17	1.553	14.9
1	20	34.44	2.31	14.9
2	36	105.9	7.12	14.9
3	56	190.8	12.53	15.2
4	92	351.2	23.77	14.8
5	148	602.5	40.28	15.0
6	240	1202.0	77.47	15.5
7	388	2536.0	172.51	14.7
8	628	5289.0	311.04	15.5
9	1016	invalid	invalid	invalid

Table 7.2.1: Execution times of fibonacci(n), where $0 \leq n \geq 9$.

Continuous Cons Experiment			
% of Accessible Cells w.r.t. total cells available (%)	68K alone (s)	68K+ICMS (ms)	relative performance
1	2.5	55.9	45.6
10	2.6	55.7	47.2
20	2.7	55.5	48.9
30	2.8	55.7	50.3
40	2.9	55.4	52.2
50	3.0	55.4	54.0
60	3.7	55.6	67.4
70	failure	55.4	invalid
80	failure	55.4	invalid
90	failure	55.7	invalid
99	failure	55.4	invalid

Table 7.3: Execution times of 1000 continuous cons operations, with varying number of accessible cells.

Plate 7.2: Experimental Setup (from left to right: an ESPIRIT terminal the ICMS and the HP1630G Logic Analyser).


HALT
S/S
RST

CHAPTER VIII
Conclusion

8.1 Summary

This thesis has described a special purpose coprocessor system known as the Intelligent Cell Memory System (ICMS). The fundamental role of the ICMS is to facilitate the employment of Artificial Intelligence techniques in real time engineering applications. One of the major difficulties in this area is poor response time caused by variable delays incurred by garbage collection. Effectively, the ICMS is a hardware realisation of Baker's garbage collection algorithm. The heart of the hardware is based on bit slice devices controlled by microprograms. The novelty of the ICMS is its dual operation mode in garbage collection - either parallel or incremental. Parallel mode provides a fast rate of cell recovery and incremental mode ensures short and bounded garbage collection intervals. Normally, the rate of garbage collection on a serial computer, increases with the amount of accessible cells. It was shown in section 7.3 that the ICMS is capable of keeping this rate constant.

On top of the garbage collector, five basic primitives for list processing : car, cdr, replacar, replacdr and cons, were also implemented in the ICMS. Therefore, functionally, the ICMS can be regarded as a linked list coprocessor. Interfacing the ICMS with a conventional host processor, the latter can be transformed into an A.I. machine with reasonable processing power. This would greatly reduce the overhead cost and time for installing new equipment. The ICMS is easy-to-use : a primitive is invoked simply by setting the appropriate memory mapped registers. Moreover, the ICMS is designed to be generic with its interface being host-independent. Providing

the VMEbus specifications are met, the ICMS can be interfaced to virtually any processor. Experiments have shown that with the ICMS connected, a processor can exhibit more than one order of magnitude speed improvement in list processing.

On top of the basic host-ICMS couple, the THESIS - The Hardware Environment for Small Intelligent System [73,74] - hardware design philosophy for engineering applications has been developed (chapter IV). It is intended to simplify hardware design by reducing system complexity; and to shrink system size - thus on-the-site intelligence is made possible.

8.2 The Future

At present, the storage size of the ICMS is inadequate for practical applications. There are two ways of expansion : either to provide the system with additional RAM, via the local bus, or incorporate disk memory with a virtual memory access technique. As usual, the choice between the two is a trade-off between speed and cost, which varies with the application. The second approach can be achieved in the same way as memory expansion for conventional computers. It is simply to place a memory management unit (MMU), a disk controller and some disk memory in between the cell memory controller (ICMC) and the RAMs.

The concept of garbage collection on objects' lifetimes, proposed by [64], is a desirable mechanism to adopt. The idea is that once a cell is discovered to have existed for a long time (several garbage collection cycles), it is regarded as permanent. All permanent cells are then transported out of the cell memory into some other storage area for archiving. Incorporating this mechanism in the ICMS would reduce the average time for garbage collection and would also provide a fairly constant workspace.

Although the speeds of execution of the 5 fundamental primitives are improved, this only provides a low level design aid. In practice, optimised high level algorithms are necessary for ultimate efficiency. Additionally, the use of well-written compilers, rather than interpreters, such as simpleLISP, to generate optimised machine codes would be most helpful if not essential. Both suggestions are outwith the scope of this project but are worth investigating in the future.

There are several unused addresses for future expansion. Frequently used high level functions can be microcoded, loaded into the ICMS and assigned with one of these addresses. Another desirable amendment to the existing hardware is the inclusion of error handling capability. Currently, an operational error would merely result in the assertion of the Bus ERRor ($\overline{\text{BERR}}$) signal on the VMEbus. This, however, is not very informative. There is no indication of the nature of the error. One solution is to employ interrupting with pre-defined interrupt vectors, which will specify the nature of the error.

The ICMS based THESIS design concept supports hardware modularity. Similar to the ICMS, other functional modules are independent units. Inter-module communication and coordination are scheduled by a host processor. The major advantage of the use of independent functional modules is flexibility. Normally, a processor can function adequately by itself. However, if provided with additional modules, the capability and the performance can be enhanced.

8.2.1 An Advanced Realisation - A VLSI/ICMS

The ICMS can be classified as a Single Instruction Multiple Data (SIMD) processor with redundancy. The target data structure of the current implementation is the list, but the philosophy is not restrictive. A list is an example of n-ary vector

structure (where n=2); the ICMS could be tailored to process other complex data structures (e.g. objects, IKBS tuples). Bit-slice design techniques would be most suitable for this application. There should be two types of slices : the controller and the memory; and together they form a column (or an element) of cell memory. Therefore, to implement a vector machine, one could simply concatenate the required number of slices.

Extending the concept of hardware modularity further and employing state-of-the art technology, the ICMS (and the other THESIS modules) could be converted into VLSI circuits. Each VLSI module would be stored independently in a design library. Designing a complete system would require establishing connections between them. In so doing, several benefits are predictable:

- cheap mass production cost;
- reduced design cost and time;
- minimised system size and possibly power consumption; and
- increased speed of execution because of closer coupling.

At the software level, it would be useful to have a special hardware design language. Thus, designing a processor would be reduced to specifying its required hardware functions with this language. The specification program would then be translated into VLSI details directly using a silicon compiler. Practically, this compiler would operate incrementally in order to enable modularisation of intermediate design or commonly used units into library cells.

Finally, it is conceivable that bus standards would not be confined to system level. Standard VLSI bus specifications would evolve. Designing the ICMS and other functional modules with it, the original generic nature of processor indepen-

dence would be retained. Another possible way of modular integration would be the use of programmable switch matrix bus architecture proposed by Chen [84]. The idea is that connections between two ports are programmable. Therefore, the switch matrix could be configured to adopt the VMEbus protocol.

APPENDIX I
UASM : the Language and the Assembler

A.1. Invocation of Uasm - a UNIX command

NAME
uasm - the local microprogram assembler.
SYNOPSIS
uasm file -**b**no -**s**[hex] -**o**name -**n** -**i**
DESCRIPTION
Uasm translates the *file* from microprogram format into S-record format. The options are:

-**b**no :
[optional] no of output words; should be hex and <= f. Only hex right next to "-b" is accepted; also defines the pc incremental quantum.

-**r**hex :
[optional] relocation address; hex should be a (< 4) bytes hex word.

-**o**name :
[optional] rename object load file from S.out (default) to name.

-**n** :
[optional] if n flag is on, the microprogram address is not the same as machine address. This is useful for target processors which can programme its own control store; in that case, all addresses will increment by one to access the next instruction, irrespective of its size. On the other hand, if it is not set (the default case) addresses will increment by the number of bytes equal to the instruction word size.

-**i** :
retain intermediate file (*um.out*).

FILES

~kfw/bin/uasm (executable source); um.out (intermediate output file) and S.out (output file in S-record format).

A.2 The Language Specifications

Uasm is a custom assembler and like any other language requires the input statements to conform to a certain syntax. The production of uasm was facilitated by the use of UNIX software development tools lex [85] and yacc [86]. The assembler also accepts macro definitions and calls in M4 [87] syntax which raises the level of microprogramming. The syntax specification of the uasm assembler language is shown in the following :

spec	:	defin	→	assem			
defin	:	**FORM**	→	sizasn	→	fieasn	→
		END	→	';'			
sizasn	:	**SIZE**	→	'<'	→	dig	→
		'>'	→	**EQU**	→	number	
number	:	hex					
	OR	oct					
	OR	bin					
	OR	dig					
hex	:	'x''	→	{ 0-9 }			
oct	:	'o''	→	{ 0-9 }			
bin	:	'b''	→	{ 0-9 }			
dig	:	{ 0-9 }					
fieasn	:	fieasn	→	';'			
	OR	fieasn	→	fieldequ			
fieldeq :	**FIELD**		→	'<'	→	dig	→
		'>'					
assem	:	origin	→	statment			
origin	:	**ORG**	→	number	→	';'	
statment	:	statment	→	bitset	→	';'	
bitset	:	label					
	OR	operation					
	OR	label	→	operation			
label	:	{ a-z }					
	OR	{ A-Z }					
operation	:	ops					
	OR	operation	→	','	→	ops	
ops	:	branchadd	→	'='	→	expr	
branchadd	:	{ a-z }					
	OR	{ A-Z }					
expr	:	number					
	OR	number	→	'+'	→	number	
	OR	number	→	'-'	→	number	
	OR	number	→	'/'	→	number	
	OR	number	→	'*'	→	number	
	OR	number	→	'<<'	→	number	
	OR	number	→	'>>'	→	number	
	OR	dot					
	OR	dot	→	'+'	→	number	
	OR	dot	→	'-'	→	number	

APPENDIX II
The LISP System : simpleLISP

A LISP look alike system was implemented to facilitate ICMS testing. It is known as simpleLISP and contains sufficient LISP functions for the purpose of the ICMS. The software was written in C. Other languages could have been used; but, since the development environment was a VAX minicomputer mounted with a UNIX operating system, C was the obvious favourite. Unfortunately, the syntax of C is not easily comprehensible by the uninitiated and hence in the following discussion a Pascal-like format is used to improve readability. There were three stages of development of the software. At first it was verified on the departmental VAX machine, latter it was modified to operate on a 68000 microprocessor and finally, the LISP primitives were transformed into an ICMS compatible format.

A.2.1 Basic Cell structure

The fundamental data item of LISP is called an atom. Groups of atoms form lists (or s-expression lists). Lists themselves can be grouped together to form more complex lists. Indeed, the ability to form hierarchical groups is of utmost importance. Atoms and lists are collectively known as symbolic expressions (or simply s-expressions). Working with these is what symbol manipulation using LISP is about.

In LISP systems, s-expressions are stored in binary-tree-like cell structures. Each node consists of a cell and each cell is composed of head and tail pointers (figure A.2.1.1a) The basic cell structure defined in the LISP system developed in this project is of the type *structure* occupying ten bytes. The first two 4-byte loca-

tions form the head and tail, respectively; the remaining 16-bits are used independently as the cell status flags (figure A.2.1.2). Knowing that both heads and tails can contain either pointers to other cells or atoms, they are defined as *unions*. A union is a pseudo structure which can hold different types, one at a time. The size of the 'union' is that of the longest type declared within it. But, the C language itself does not keep track of the type stored. This job is performed by the status flags. The functions of each bit are described in figure A.2.1.2.

A.2.2 The LISP System

In operation, the simpleLISP system developed resembles the 'OPUS version 36 Franz LISP system' used in the Artificial Intelligence Department in Edinburgh University. It reads in a LISP instruction, evaluates it, sets the programming environment and prints out the result accordingly.

SimpleLISP is very primitive in error handling as compare to OPUS. In OPUS, error handling is maintained by a technique called 'Spaghetti Stack' [26]. When an error is detected in the LISP instruction, OPUS will nest into the first level of the stack (prompt as <1>:); if the user then corrects the corresponding error, it will resume operation from the original level. If, however, an error occurs in the first level, OPUS will then nest into a second level (prompt as <2>:). If the amendment then made was correct, OPUS would unwind itself to the previous level; otherwise, such a nesting scheme would proceed into deeper levels. On the other hand, for simpleLISP, error handling is relatively crude: whenever an error is detected, it will abandon the current instruction, print an error message and prompt the user for the next instruction. A nesting scheme like that of OPUS, would of course be desirable if it could be installed in simpleLISP in the future; nevertheless, a lack of it does not

prevent investigation of engineering applications.

In the following sub-sections the implementation details of the simpleLISP system will be discussed.

A.2.2.1 Input/output

The purpose of the input routine is to read in a s-expression and from it to build up a data structure. The techniques adopted are conventional compiler/interpreter design practice, which proceeds in two phases: lexical analysis and syntax analysis.

a) Lexical analysis

S-expressions are entered one line at a time. Each line being stored in a line buffer before being scanned by a lexical analyser. The lexical analyser classifies the various symbols or groups of symbols into a series of tokens. There are four types of token:

TOKEN	TYPES
(< or [	delimiter - open parenthesis
) > or]	delimiter - close parenthesis
	separator - dot
e.g. TOM	alphanumeric
e.g. 4	numeric

Spaces and newline characters, act as separators between different symbol types; and they are ignored. The different forms of parenthesis are used to improve readability and have no other meaning.

b) Syntax analysis

The syntax diagram for an s-expression is shown in figure 2.2.1a. The s-expression is defined in such a way that it may be syntactically analysed by a method known as 'recursive descent' [88]. This is nothing more than writing a recursive procedure for each category of phrase: s-expression and s-expression list, whose job it is to scan the input and recognise the phrase with which it is associated. The ability to do this depends upon being able to predict the next token on the input.

The final stage of the input section constructs a cell structure in list space which represents the s-expressions. Up until this stage, the algorithms for both lexical and syntax analysis are derived from Henderson [88]. Henderson has also suggested an algorithm for the final stage, the main feature of which is depicted in figure A.2.2.2. His algorithm can be seen to be a direct translation from the syntax diagram (figure A.2.2.1). However, on careful examination, it was found that this algorithm is impractical. In the second procedure - getexplist() - the functions CAR and CDR are called, which are supposed to return the head and tail pointer, respectively. But, these cannot be evaluated without arranging the complete s-expression into the list space first! Thus the algorithm finally adopted was the one designed by Coghill[54].

Coghill based his getexp and getexplist procedures on a link-reversal technique, which is a classical algorithm most frequently applied to garbage collection [56]. The link-reversal algorithm will be explained in the next section.

For the output of s-expressions, the reverse of Henderson's algorithm in figure A.2.2.2 is employed. Since now, having the s-expressions already allocated in the list space, CAR and CDR can be invoked. In [54] a different output algorithm was developed which works on the reverse of his input method. Such an algorithm is less efficient since every node is visited thrice on completion - once for the forward trace and twice for the backward trace. In Henderson's algorithm, every node is only

visited twice. Whenever the program branches from a node the return address of that node is automatically saved on the hardware stack; thus, there is no need to keep track of the tracing route within the software as Coghill had done. For simplicity, there is no provision for 'pretty' printouts in the current implementation.

A.2.3 Garbage Collectors

SimpleLISP like any other list processing system comprises two related processes. The 'mutator' allocates cell from free storage and links them together to form data structures representing s-expressions. When the mutator drops a data structure it no longer needs, the abandoned cells remain in the system but are no longer accessible. In order to prevent the mutator from running out of cells to allocate and thus blocking computation, a 'collector' process gathers the inaccessible or garbage cells and returns them to free storage for reuse by the mutator.

Three versions of simpleLISP were implemented and each of them was based on a different garbage collection algorithm. The algorithms implemented were : the link reversal mark and sweep collector, the Baker copying collector and the Baker incremental collector.

A.2.3.1 Mark and Sweep Garbage Collector

The first garbage collector installed in simpleLISP was a classical mark and sweep type. As its name suggests, mark/sweep collectors proceed in two phases:

1 in the first phase (MARK), garbage cells are identified;

2 during the second phase, all marked cells are reclaimed and linked together in free storage - SWEEP.

Sweeping is a fairly straight forward process which requires a linear scan of the free

list space, unmarking and linking all marked cells as it goes. The result is a a chain of free cells ready to be reused by the mutator.

The simplest form of marking algorithm is the one shown in figure A.2.3.1, which is proposed in Winston[59]. This algorithm traces the complete cell structure by first marking the tail branches until it comes to a halt by an atom or a marked node; then it resumes tracing the heads from where it originally branches off. Note that the system stack is used for storing return addresses and each node is visited twice: once before marking the tail and once before marking the head. The major drawback of this type of collector is that it uses extra storage for stacking return addresses. In the extreme case, if a list space of N list cells have to be completely marked, the size of the stack required would be that of the list space itself (i.e. N). Provision of such a huge additional amount of memory could greatly reduce the effectiveness of garbage collection !

Deutsch, Schorr and Waite [60] had independently designed a similar marking algorithm without incorporating a stack. This algorithm was later refined by Knuth (algorithm E in [56]); and it is currently being employed in simpleLISP. Figure A.2.3.2 shows the algorithm.

The main idea of this method is that the nodes of a tree can be inspected by reversing successive links, using three pointers (p,q,t), until leaves (i.e. atoms) or already visited nodes are found. The link reversal can then be undone to restore the original structure of the tree. Referring to the figure A.2.3.2 : Initially, p is set to point at the root of the tree, q is set to point at the contents of the head of this root and t is set to NIL (a special LISP atom). The first cell is marked, t is placed in the head of the cell: this acts as the termination marker during the reverse trace; and p is set equal to q. This algorithm then moves to the next cell by setting q to the head of

p and t equal to p. When the bottom of the tree is reached the same procedure is carried out in reverse until the NIL termination marker at the root cell is encountered. At any node, the program has a choice of two routes: either via the head or the tail. To distinguish the routes that it has taken, the numeric atom bit is used. On branching from the head, the numeric atom bit is set. In this way, all the numeric atom bits will have been set on tracing the head links and these must be cleared during the reversal. Each time the head of any node has been traced, the tail of that node is immediately traced by the same method.

Although, the link-reversal algorithm has been widely accepted as a standard mark/sweep garbage collector, it is by no means ideal. There are two main identifiable drawbacks:

1 Each node is being visited three times and extra effort is required to set the numeric atom bit, thus rendering the technique less efficient. In recent years, various pseudo link-reversal and 'copying' collectors have been designed with improved efficiency [58].

2 The most annoying aspect common to all conventional garbage collectors is that program execution comes to a complete halt while the collection process is in action. This is annoying and is most apparent in multi-user systems where users may experience interruption lasting minutes. In extreme cases, successive collections may take place with little actual program execution between them, making continued computation impractical.

A.2.3.2 Copying Collector

After the suggestions made by Fenichel [65] and Cheney [66], various versions of the copying garbage collector have been designed. They all work under the same

principle, and differ only on the moving algorithm used.

The cell storage is divided into two semispaces whose roles are reversed with each garbage collection cycle. When collection begins, the semispace labeled TOSPACE is empty, and all accessible and garbage cells are in the semispace labeled FROMSPACE. The collector copies the accessible cells out of FROMSPACE into TOSPACE. When this process is completed, FROMSPACE contains only inaccessible cells, while TOSPACE contains all and only accessible cells. Now program execution can resume with the mutator allocating cells from the garbage free TOSPACE until it is again necessary to collect.

A.2.3.2.1 Storage Efficiency Improvement

The most promising algorithm which can solve the problems stated earlier is the one originally used by Fenichel and Yochelson in an early Multics LISP [65], elegantly refined in [66], and applied by Arnborg to SIMULA [67] (a LISP system). The principle is as previously described, i.e. it bisects the list space into two semispaces; and the moving algorithm works as follows :

During the execution of the user program, all list cells are located in the FROMSPACE. When garbage collection is invoked, all accessible cells are traced, and instead of simply being marked, they are moved to the other semispace. A forwarding address is left at the old location, and whenever an edge is traced which points to a cell containing a forwarding address, the edge is updated to reflect the move. The end of tracing occurs when all accessible cells have been moved into the TOSPACE and all edges have been updated. Since the TOSPACE now contains only accessible cells and the FROMSPACE contains only garbage, the collection is done and the program execution can resume with cell allocation proceeding in the

former FROMSPACE. This algorithm is depicted in figure A.2.3.2.1.

This method is simple and elegant because :

1 it requires only one pass instead of two as in the mark/sweep collector;

2 it requires no collector stack.

The stack is avoided through the use of two pointers, B and S (c.f. figure A.2.3.2.1). B points to the first free word (the bottom) of the free area, which is always in TOSPACE. B is incremented by the procedure COPY, which transfers old cells from FROMSPACE to the bottom of the free area, and by CONS which allocates new cells. S scans the cells in the TOSPACE which have been moved, and updates them by moving the cells they point to. S is initialised to point to the beginning of TOSPACE at every flip of the semispaces and is incremented when the cell it points to has been updated. At all times, then, the cells between S and B have been moved, but their heads and tails have not been updated. Thus when S=B all accessible cells have been moved into TOSPACE and their outgoing pointers have been updated. This method of pointer updating is equivalent to using a queue instead of a stack for marking, and therefore traces a spanning tree of the accessible cells in breadth first order.

A.2.3.2.2 Time Efficiency Improvement

To tackle the second inefficiency problem of classical collectors, an improved version of Arnborg's algorithm is suggested in [63] which makes real-time collection possible by sacrificing some time in each cell allocation call.

During each CONS operation, cells are requested from the list space. In classical collectors, a shortage of an upper bound (in the time domain) of CONS causes embarrassing program interruption - if CONS is called during exhaustion of either

semispace, garbage collection is invoked, whose functional time would, unpredictably, depend on the number of cells to be marked. Baker's real-time algorithm overcomes such non-deterministic behaviour by forcing an upper time limit on the CONS call; such that each time a cell is requested, a fixed number of cells, k, are moved from one semispace to the other. This implies that the two semispaces are simultaneously active. The moving of k cells during a CONS corresponds to the tracing of that many cells in the collection process. By distributing some of the collection tasks during list processing, this method provides a guarantee that the actual garbage collection cannot last more than a fixed (tolerable) amount of time : the time to flip the semispaces and to readjust a fixed number of points declared in the user's program. Thus, it is applicable to real-time situations.

A characteristic of Baker's real-time collector is that the size of the semispaces may have to be increased, depending on the value of k [63]. In other words, the choice of k expresses the trade-off between the time to execute a CONS and the total storage required. The following equation relates k and the maximum storage size required :

$$\text{MAX STORAGE REQUIRED} \leq N\,(2 + \frac{2}{k}) \qquad \text{A.2.3.2.2}$$

where N is the number of accessible nodes.

At first sight, equation A.2.3.2.2. reveals that increasing k decreases the storage requirement of a system. However, k should be bound, otherwise, program execution would be delayed for every CONS operation. Depending on the operational environment, therefore, one should choose a suitable k in order to obtain the maximum gain of the system. For example, if Baker's collector is applied for the management of a large database residing on secondary storage, k could be made a positive rational

number less than one. If k=3, for instance, one could have CONS perform an iteration of the collector only every third time it is called. The result of this is the requirement of a larger free storage, according to eqn. A.2.3.2.2; which physically means to provide a constant supply of accessible cells, and hence the program will avoid cell starvation during each CONS cycle.

A.2.3.3 Remarks

The correctness of the adopted collection algorithms was justified. This was achieved by first creating a useful list (e.g. an environment list) and then continuously forcing simpleLISP to garbage collect. The result was positive with all information retained uncorrupted. Another experiment was carried out to demonstrate the slow response of the non-real-time algorithms. A fibonacci(n) function was used. As the parameter n increased the number of cells allocated increased and at some stage garbage collection occurred. For small n there was a slight difference; as n increased over 10, however, the responses of the non-real-time collectors were very sluggish whilst the time taken for the real-time collector was reasonable. Unfortunately, no physical timing was taken; nevertheless, the relative response of the two collector types clearly favoured the real-time algorithm for engineering applications.

A.2.4 The Interpreter

A.2.4.1 Program Environment

One of the features of LISP software which makes it suitable for implementing intelligent systems is the interactive way in which instructions are interpreted. Thus, users can 'educate' the system as the system's environment varies.

LISP atoms are bound to objects. Unlike other languages, however, objects are s-expressions. An 'association list' is a list of bindings defining the pairing between various atoms and their bound s-expressions. During program execution, atoms are continuously being assigned and defined - using SETQ and DEFUN; such bindings are recorded in a global association list (i.e. accessible by the program at any time), called the 'program environment'. The program environment can be thought of as a dynamic memory which absorbs and remembers anything it has acknowledged.

The environment grows in two directions as computation progresses. When an s-expression is interpreted as an assignment statement, the association of that atom is appended to the tail of the existing environment. In this way it becomes a global variable which can be recalled later. The second way, alters the structure of the environment only temporarily by adding associations on to the head of it. This is called 'local binding' which occurs during function calls. During function execution, temporary variables may have been declared, the bindings of such variables are fairly loose: when they are assigned, their bindings are created; nevertheless, they are only valid within the context of the function and would be dropped when computation exits from this function.

A.2.4.2 The Interpretation Routines

The interpreter used in simpleLISP is based on [59] and works in three levels:

a) level 1:

The system starts from this level. The routine responsible for this top level is M_R_E_P, which reads expressions to be evaluated, evaluates them, and prints the result (figure A.2.4.2.1). M_R_E_P also arranges for function definitions to be stored

in the program environment to be supplied to the second level. The ability to define functions is another useful feature in having an interactive interpreter.

b) level 2:

Two functions are included in this level:

M_EVAL gets two arguments, an expression and the program environment (figure A.2.4.2.2). The job of which is to classify the s-expression and to decide how it should be evaluated. If the s-expression is an atom, it uses the environment to retrieve its corresponding association. Otherwise, it assumes the s-expression is a function with several arguments and evaluates the arguments in the way that is appropriate for the function.

M_APPLY gets three arguments, a function name or description, a list of arguments and the program environment (figure A.2.4.2.3). Its job is to classify functions and to arrange for their proper application. M_APPLY handles some simple functions directly. For others M_APPLY augments the environment and appeals to M_EVAL for help. For user defined functions, M_APPLY has to retrieve the function definitions from the program environment, and then, recursively, call itself with the original function replaced by the retrieved function definition.

c) level 3:

This lowest level comprises several individual procedures corresponding to each LISP operational primitive. At present, simpleLISP only supports a limited number of LISP primitives: namely, CONS, CAR, CDR, SETQ, QUOTE, APPEND, REPLACA, REPPLACD, APPEND, ATOM, NULL, DEFUN, LAMBDA, and EQ. It is intended that the system will be expanded in the future.

A.2.5 SimpleLISP On a M68000

SimpleLISP was later transported into a M68000 microprocessor. In order to use the original source interpreter, it had to be cross compiled. A C-to-M68000 cross compiler, CC68 was available on the original VAX machine. However, two problems with CC68 discovered during the debugging stage did cause some difficulties. These were the inability to define functions which returned union types and errors in the indirection of local pointer variables. Some rewriting of code was necessary to overcome these problems.

A.2.6 SimpleLISP for the ICMS

For the final stage, only a slight amount of modification to the M68000 version was required. The size of a cell is reduced to 48 bits, shared equally between the HEAD and TAIL elements. Differentiation between a cell pointer and an ordinary address was done by direct decoding. Therefore at least one bit is required for this purpose reducing the maximum amount of storage to 2^{23} cells. At present, only 1k cells are implemented.

The garbage collection module was removed since the ICMS provided this function. The algorithm adopted was the Baker real-time copying collector. It was microcoded and together with the special ICMS architecture the operational speed of the original algorithm has improved.

Five LISP primitives are built-in with the ICMS. They are _CAR(X), _CDR(X), _REPLACAR(X,Y), _REPLACDR(X,Y) and _CONS(X,Y). On the software level, to invoke these functions are simply to set the appropriate IO registers; such as, the 68000 code for CAR() is simply:

```
CAR:MOVE.L  X, $FF8000      ; parameter in X
    MOVE.L  $FF8000,D0      ; result return in D0
```

In addition there are 8 local functions which are used for setting and reading the root registers, which are situated internally within the ALUs. For these functions when a cell address is placed into the IO register, it implies a set operation; otherwise, for any other data placed, a read operation is performed. Finally, the memory map of simpleLISP with the ICMS, as shown in figure A.2.6, indicates the ranges at which simpleLISP is placed and where the IO registers are located.

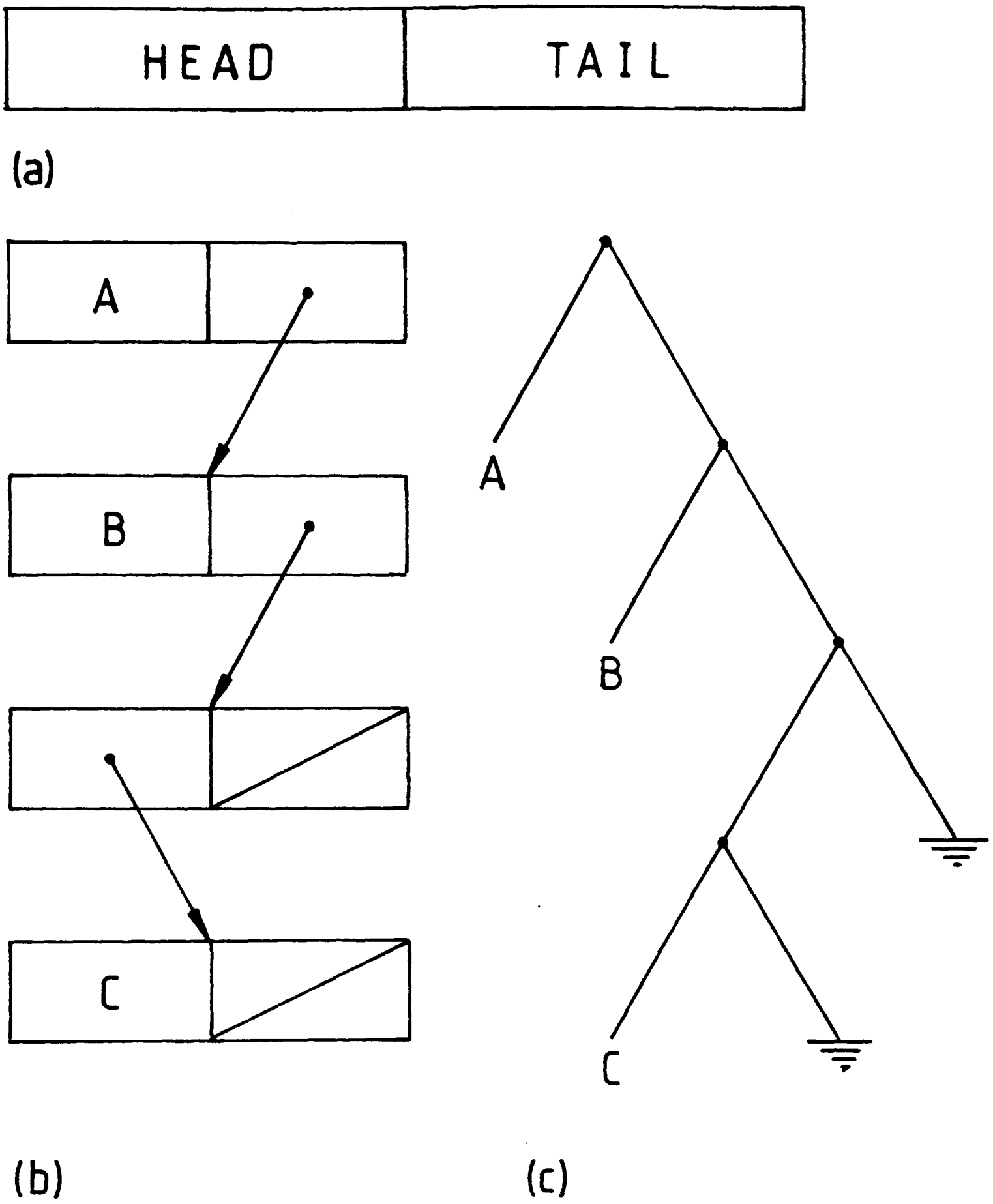
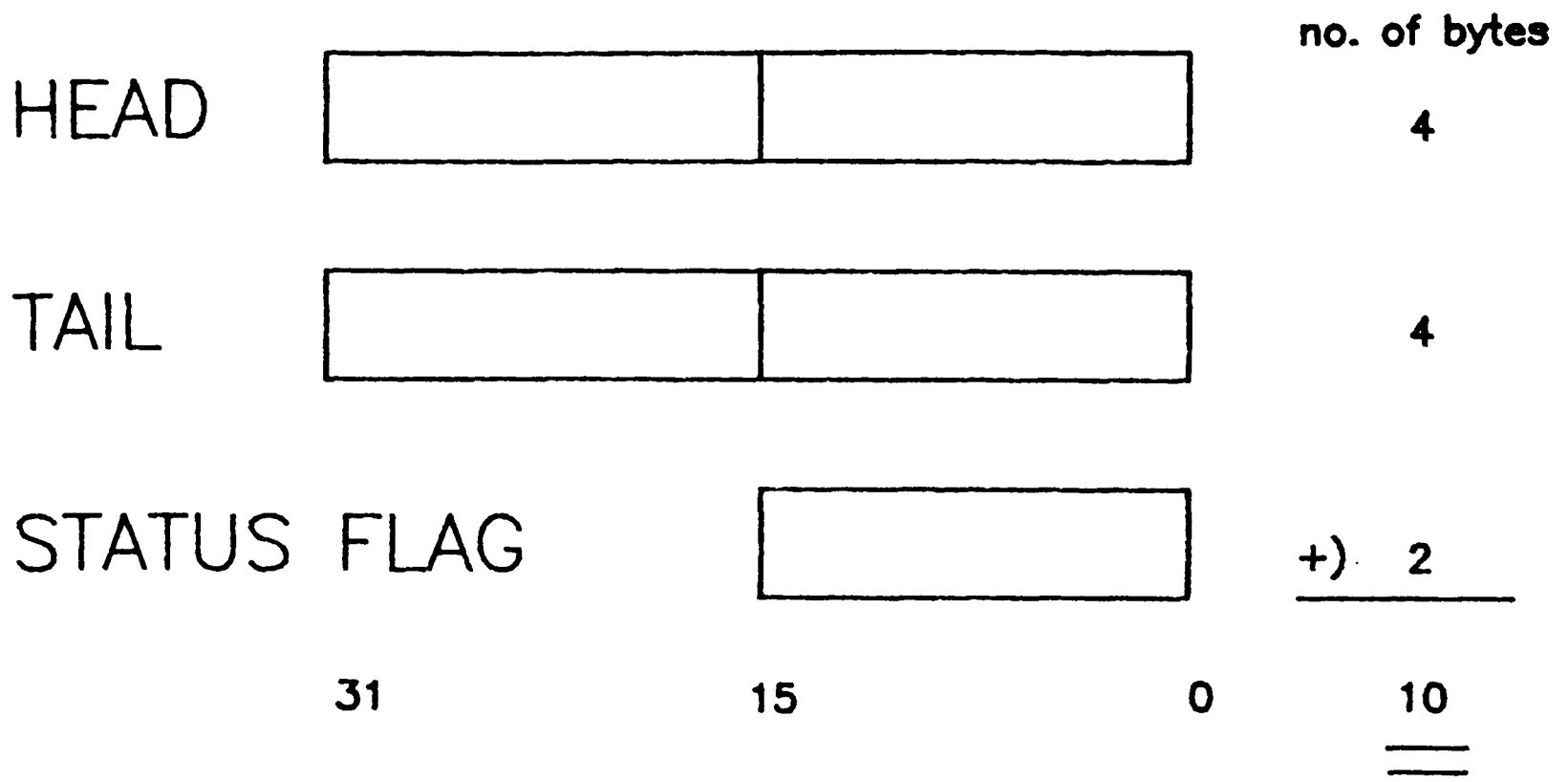
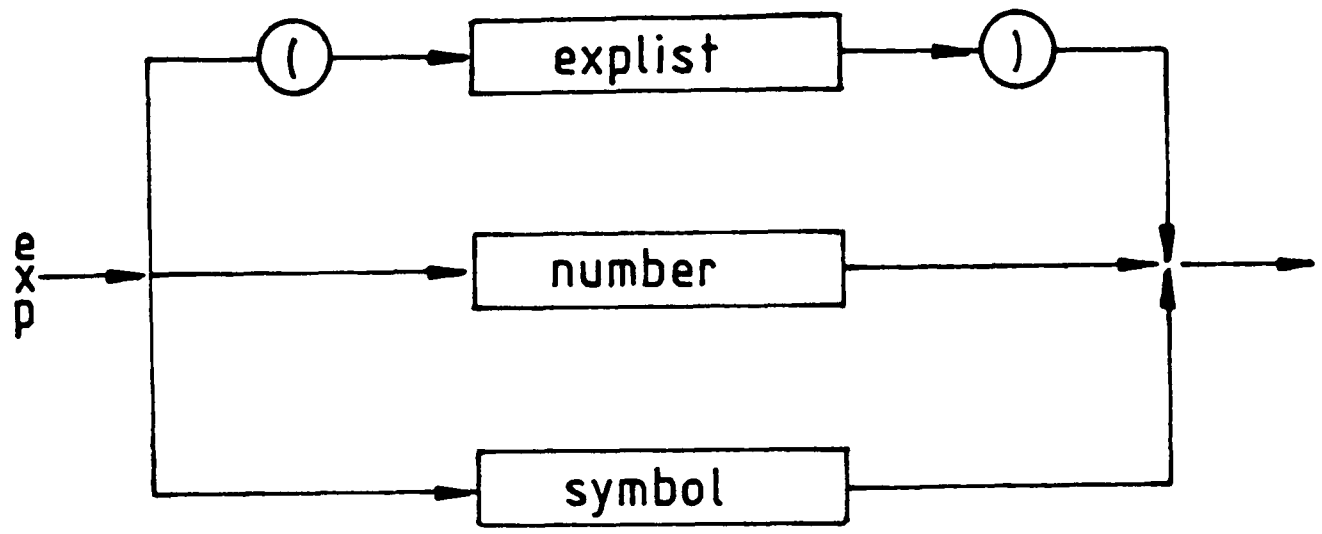


Figure A.2.1.1a: A Single Cell Unit;
Figure A.2.1.1b: An Example of a List - (A B (C)) in Cell Structure;
Figure A.2.1.1c: (A B (C)) in Tree Structure Form.

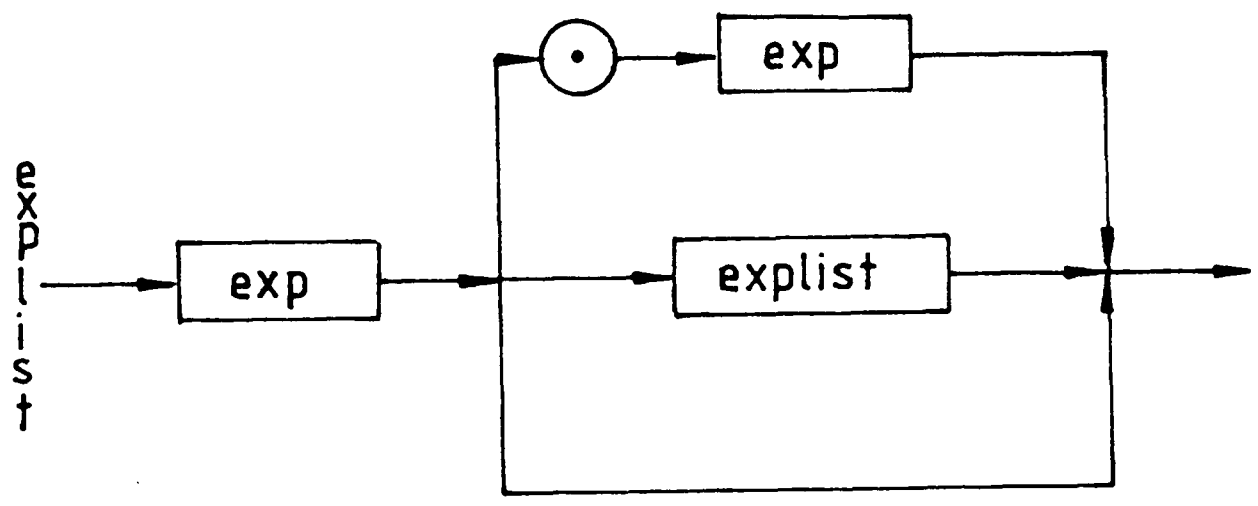


bit no.	bit name	description
0	mark bit	used in garbage collection
1	direction bit	used in input s-expression
2	numeric bit	set if cells contain an numeric atom
3	cons bit	set if cells are associated with lists
4,5	head type	indicate head type: 0: cons, 1: number, 2: literal, 3: keywords.
6,7	tail type	indicate tail type (as above)

Figure A.2.1.2: Description of the Cell Status Flag.



(a)



(b)

Figure A.2.2.1a: *Syntax Diagram of S-expression (abbreviated exp);*
Figure A.2.2.1b: Syntax diagram of S-expression list (abbreviated explist).

```
PROCEDURE getexp(e : pointer);
BEGIN
        IF (token = open_bracket) THEN
        BEGIN
            getexplist(e);
        END
        ELSE
        IF (token = number) THEN
            work out its value;
        ELSE
            % A string
            work out its symbol;
END % Getexp


PROCEDURE (getexplist(e : pointer);
BEGIN
        IF (e is a list) THEN
            getexp( CAR(e) );
        ELSE
        IF (token = dot) THEN
            getexp( CDR(e) );
        ELSE
        % End of a list
        IF (token = close_bracket) THEN
            set tail of e to be NIL;
        ELSE
            getexplist( CDR(e) );
END;    % Getexplist
```

Figure A.2.2.2: Henderson's program for input s-expression [88].

```
PROCEDURE putexp(e : pointer);
VAR p : pointer;
BEGIN
        IF (e is a symbol) THEN
        BEGIN
            lookup the dictionary;
            get the symbol;
            print it;
        END
        ELSE
        IF (e is a number) THEN
            print the number;
        ELSE
        BEGIN
            % Start of a new list
            WRITE('(');
            % Save e and use p for recursion
            p : = e;
            WHILE (p is a list) DO
            BEGIN
                putexp( CAR(p) );
                p := CDR(p);
            END;
            IF (p is a symbol AND its value is NIL)
            THEN GOTO finish;
            ELSE
            BEGIN
                WRITE('.');
                putexp(p);
            END;
finish:     WRITE(')');
        END;
        WRITELN;
END;    % Putexp
```

Figure A.2.2.3: Program for Output S-expression.

```
PROCEDURE markal(p : pointer);
BEGIN
        IF (p is not marked) THEN
        BEGIN
            mark node p;
            IF (p is not an atom) THEN
            BEGIN
                markal(head of p);
                markal(tail of p);
            END;
        END;
END;    % Mark
```

Figure A.2.3.1: The Simplest Marking Algorithm.

```
% Global cell pointers
VAR p,q,t : pointer;

PROCEDURE uphead();
BEGIN
        t := head of q;
        head of q := p;
        p := q;
END;    % Uphead

PROCEDURE uptail();
BEGIN
        t := tail of q;
        tail of q := p;
        p := q;
END;    % Uptail
```

Figure 2.3.2a: The Marking Algorithm (to be continued).

```
PROCEDURE downhead();
BEGIN
        q := head of p;
        WHILE (q is a cell AND it is not marked) DO
        BEGIN
            mark node q;
            IF (q is not an atom) THEN
            BEGIN
                % Indicate branching is from head
                set the direction bit;
                % Reverse pointers
                head of p := t;
                t := p; p := q;
                q := head of p;
            END;
            ELSE BREAK;
        END;
END;    % Downhead

PROCEDURE downtail();
BEGIN
        q := tail of p;
        WHILE (q is not a cell AND it is not marked) DO
        BEGIN
            mark node q;
            IF (q is not an atom) THEN
            BEGIN
                tail of p := t;
                t := p; p := q;
                % Traverse the head links of each tails
                downhead();
                q := tail of p;
            END;
            ELSE BREAK;
        END;
END;    % Downhead
```

Figure A.2.3.2b: The Marking Algorithm (continued from previous page).

```
PROCEDURE markal(p0 : pointer);
BEGIN
        IF (p0 is not a cell) THEN RETURN;
        IF (p0 has already been marked) THEN RETURN;
        % Initialise pointers
        p ;= p0; t := NIL;
        mark node p;
        IF (p is a number) THEN RETURN;
        downhead();
        WHILE (TRUE) DO
        BEGIN
            % A termination is reached
            IF (t is equal to NIL) THEN
            % Reverse links and start back trace
            q := t;
            IF (direction bit is not set) THEN
            BEGIN
                uptail();
                CONTINUE;
            END;
            clear direction bit;
            uphead();
            BREAK;
        END;
END;    % Markal
```

Figure A.2.3.2c: The Marking Algorithm (continue from previous page).

```
VAR
B : pointer; % B points to bottom of free area
T : pointer; % T points to top of free area
S : pointer; % S points to the next untraced cell

FUNCTION CONS(x : pointer, y : pointer) : pointer;
BEGIN
        % If there is no  more free space
        IF (B is equal to T) THEN
        BEGIN
            interchange semispaces;
            update all ROOT registers;
            x := move(x);
            y := move(y);
        END;
        % Scanning => trace untraced cells
        WHILE (S < B) DO
        BEGIN
            head of S := move( head of S );
            tail of S := move( tail of S );
            increment S pointer;
        END;
        IF (B >= T) THEN error;
        % Construct a new cell
        head of B := x;
        tail of B := y;
        increment B;
        RETURN( B-1 );
END;    % CONS

FUNCTION move(p : pointer) : pointer;
BEGIN
        IF (p is not in FROMSPACE) THEN RETURN( p );
        ELSE BEGIN
            IF (head of p is not in TOSPACE) THEN
            BEGIN
                % implies p is an atom
                head of p := copy(p);
                RETURN( head of p );
            END;
END;    % Move

FUNCTION copy(p : pointer) : pointer;
BEGIN
        IF (B >= T) THEN error;
        head of B := head of p;
        tail of B := tail of p;
END;    % Copy
```

Figure A.2.3.2.1: The Copying Garbage Collector.

```
VAR envir : pointer; % Program environment

PROCEDURE M_R_E_P();
VAR exp : pointer;
BEGIN
        % Environment is initially empty
        envir := NIL;
        WHILE (TRUE) DO
        BEGIN
        exp := input s_expression;
        IF (exp is an atom) THEN
            IF (exp is associated) THEN
                extract association from environment;
            ELSE
                WRITE("Unbound Variable....");
            ELSE
            IF (exp is a definition statement) THEN
            BEGIN
                set the environment accordingly;
                print the name of the defined function;
            END
            ELSE
                outexp( M_EVAL(exp,envir) );
        END;
END;    % M_R_E_P

FUNCTION M_EVAL(s,environment : pointer) : pointer;
BEGIN
        IF (s is an atom) THEN
            work out its association;
        ELSE
        IF (it is a QUOTE statement) THEN
            RETURN( CAR( CDR( s ) ) );
        ELSE
        IF (s is a COND statement) THEN
            pass to M_COND in level three;
        ELSE
        BEGIN
            % Here if s is a FUNCtion
            evaluates all the sub-expression;
            pass them as parameters into M_APPLY;
            and finally return the results;
        END;
END;    % M_EVAL
```

Figure A.2.4.2.2: Level 2(a) of the Interpreter : M_EVAL.

```
PROCEDURE M_APPLY(func,args,envir : pointer) : pointer;
BEGIN
        IF (func is an atom) THEN
            IF (it is CAR) THEN
                RETURN( head( head( args ) ) );
            ELSE
            IF (it is CDR) THEN
                RETURN( tail( head( args ) ) );
            ELSE
            IF (it is CONS) THEN
                RETURN( CONS( head(args), tail(args) );
            ELSE
            IF (it is ATOM) THEN
                RETURN( AOM( head(args) );
            ELSE
            IF (it is NULL) THEN
            BEGIN
                test if head of args is NIL;
                RETURN the atom T if true;
                otherwise RETURN atom NIL;
            END;
            % Additional LISP primitives can be added here, if M_APPLY
            % cannot cope with them itself,  they can be  passed  into
            % pre-defined modules in level three.
            ELSE
            % At this stage func must be a user defined function
            % whose context must have been associated in envir.
            BEGIN
                M_EVAL( func,envir );
                % Then re-apply M_APPLY on the result;
            END;
        END;
END;    % M_APPLY
```

Figure A.4.2.3: Level 2(b) of the interpreter : M_APPLY.

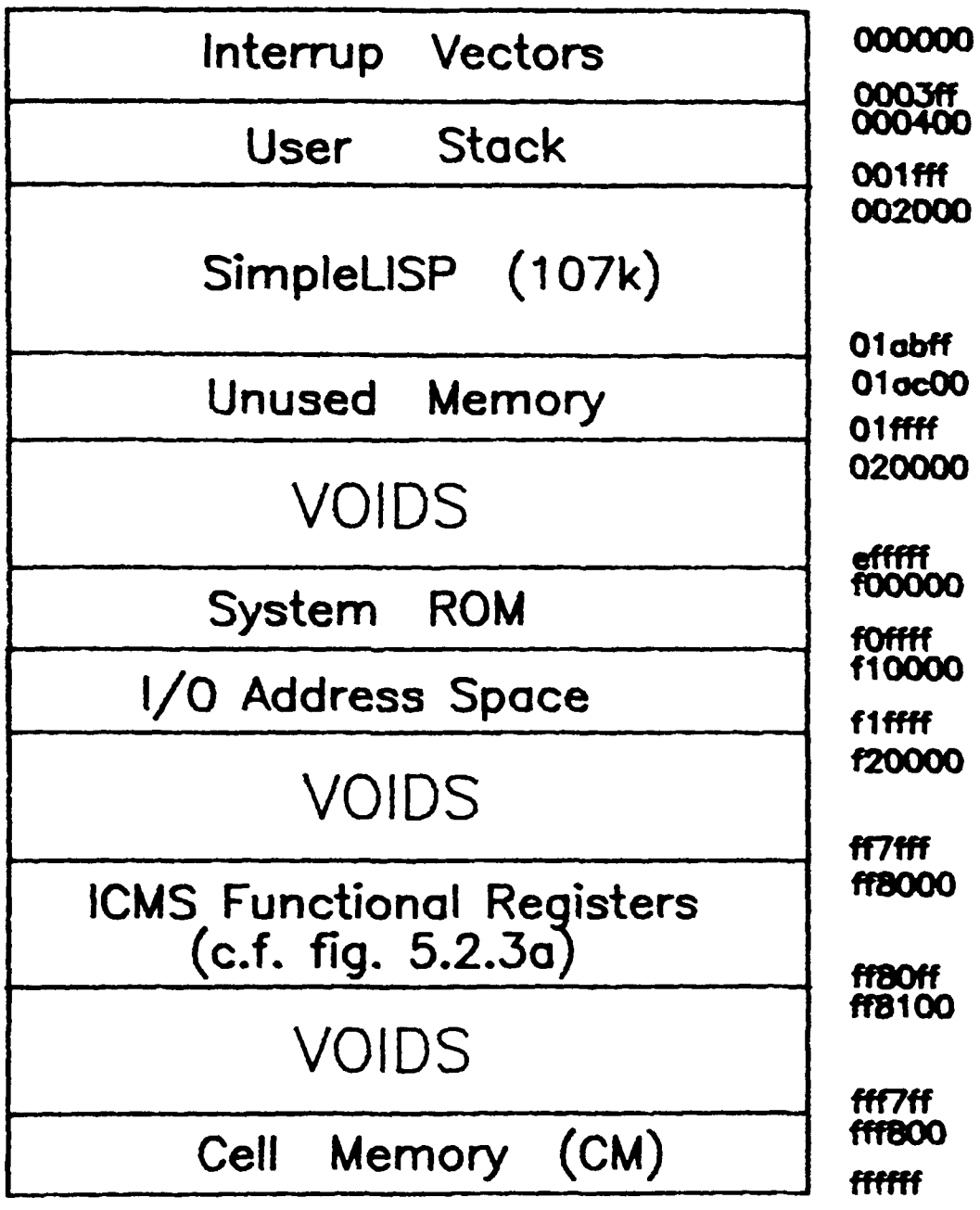


Figure A.2.6: The Memory Map of SimpleLISP with the ICMS.

APPENDIX III
The Graphical Program

The best way to study the garbage collection process is to obtain a visual picture of how the process works. This has been done by writing a graphical program to reveal the route which a collector takes during the inspection of a spanning tree. The program developed is in C on a HP9000, using the library graphical routines - Device-independent Graphics Library (DGL) [89].

A.3.1 Standard Tree Pattern

The first step is to identify the locations of all the nodes to be drawn in terms of a 'standard tree pattern'. Figure A.3.1 shows a standard tree; two pieces of information on the tree are most importance :

a the relationships between a node and its head and tail, as shown in the following equations:

$$\text{head}(n) = 2n + 1 \qquad \text{A.3.1.1}$$

$$\text{tail}(n) = 2n - 1 \qquad \text{A.3.1.2}$$

where 'n' is the node number.

b the number of nodes in each level :

$$\text{no.ofnodes} = 2^L \qquad \text{A.3.1.3}$$

where L is the level number.

A.3.1.1 Tracing

To identify the locations of the nodes, a link-reversal technique [54] is used. Before it can be applied, four extra items have to be declared. Arrays A and B: B

contains the current node number and A contains the number of the node from where B is branched; two integers: indexA and indexB to keep track of array A and B, respectively, during the process.

Initially, A[0], B[0], indexA and index B are all zero, since the first node being inspected is the root with node number equal to zero. From then on, during the forward trace, when a node branches from its head, the relationship between A and B is

$$B[indexB] = A[indexA] \times 2 + 1 \qquad \text{A.3.1.1.1}$$

and if it branches from its tail :

$$B[indexB] = A[indexA] \times 2 + 2 \qquad \text{A.3.1.1.2}$$

(where A contains the node number from which the branch is made). Every time A is set to the previous value of B and indexA and indexB are incremented after a new node number has been assigned to B. On the backward trip, indexA is decremented by two whilst other variables remain unchanged. This procedure continues until the root of the tree is encountered when the complete process terminates. On completion the node numbers contained in array B have already been arranged in the order in which the nodes are visited by the garbage collector. Thus to study the collection process, it is necessary only to translate the locations in B into graphical coordinates. The complete tracing algorithm is as shown in figure A.3.1.1.

A.3.2 Graphical Representation

One difficulty in drawing a graphical tree structure is to make it evenly displayed on the screen, this requires a prior knowledge of the size (depth) of the tree. This information is obtained by first copying array B into a similar array C, and later arranging the entries in C in ascending order; thus, from the anti-logarithm (base 2) of the last entry in array C, which must be one of the nodes on the deepest

level, the level number is determined. Knowing the depth of the tree, separation between levels is just equal to the quotient of the vertical screen size and the tree depth - in other words, the Y coordinates of each level are determined. The next step is to work out the horizontal separation between nodes on each level. This is done by dividing the breadth of the screen by the number of nodes on the particular level.

After obtaining the X and Y coordinates, the tree structure can be drawn. Drawing starts from left to right. After drawing one level the coordinates of all the nodes are saved. With the coordinates of these nodes, the program knows where to draw line segments when it is working on the level above. These lines correspond to head and tail links. A colour convention has been adopted : each node is represented by diamond markers and their colours indicate whether they are atomic (RED) or non-atomic (GREEN) nodes. Also, untraced lines are coloured white, and later over-coated by blue when they are marked.

The tree structure is first produced once with all white links then coordinates of all the nodes are re-arranged in such a way that they appear in the same order as in the original array B -- i.e the order of tracing. In this way, re-applying the drawing algorithm again, but using blue for each link, the complete garbage collection process can be graphically visualised.

A.3.3 Comment

The graphics program has been tested and proved to be capable of drawing tree structures as deep as seven levels. Higher level trees tend to have their nodes and links merged together if they are drawn; thus producing an unclear view. In practice many lists are much deeper than seven levels, but the present software does provide a useful visual indication of how garbage collection operates.

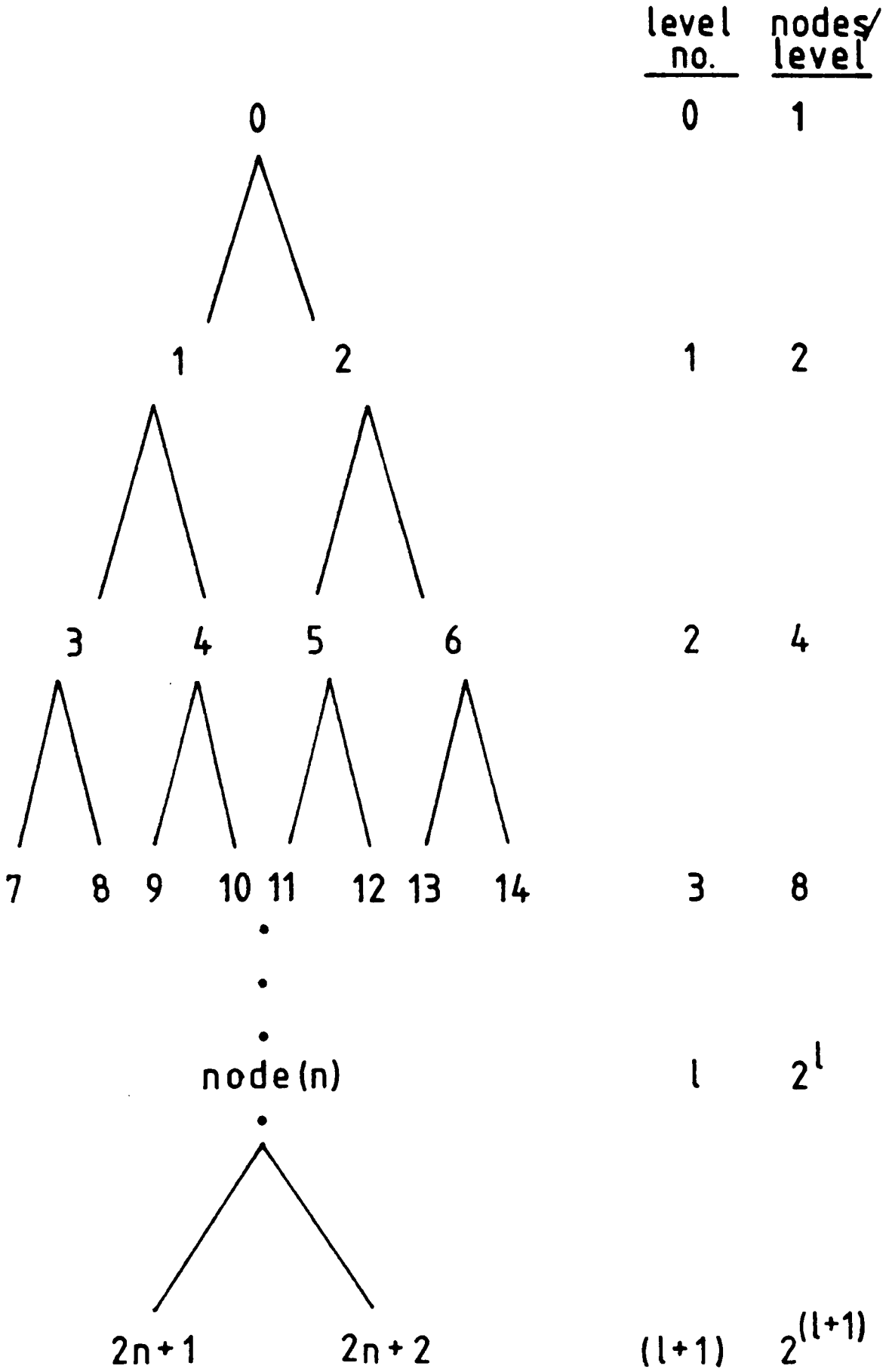


Figure A.3.1: Standard Tree Pattern.

```
VAR     A[200], B[200] : array of integer;%A[] & B[] contain node nos.
        indexA, indexB : integer:%Indices into A[] & B[];

PROCEDURE g_markal(p0:pointer);
BEGIN
        IF (p0 is not a cell) THEN RETURN;
        IF (p0 has not been marked) THEN RETURN;
        p := p0;
        t := NIL;
        A[0] := 0;  B[0] := 0;
        indexA := 0;  indexB := 0;
        mark node p;
        IF (p is a number) THEN RETURN
        g_downhead();
        WHILE
            g_downtail();
            WHILE (TRUE) DO
            BEGIN
                % Termination has been reached
                IF (t=NIL) THEN RETURN
                q := t;
                IF (direction bit is not set)  THEN
                BEGIN
                    uptail();
                    % Return to the node at a level above
                    update(1,indexB+1);
                    CONTINUE;
                END;
                clear direction bit;
                uphead();
                % return to the node at a level above
                update(0,indexB+1);
                BREAK;
            END;
        END;
END;    G_markal
```

Figure A.3.1.1a: The Graphical Tracing Algorithm.

```
PROCEDURE g_downhead()
BEGIN
        WHILE (head ofp points to a cell) DO
        BEGIN
            q := head(p);
            IF (q is not a cell) THEN
            % Head of p is a NIL
            BEGIN
                % equation A.3.1.1.1
                B[indexB] := A[indexA]*2 + 1;
                % return to branching node before return
                update(1,index);
            END
            ELSE
            BEGIN
                IF (q has not yet been marked) THEN
                BEGIN
                    set direction bit;
                    B[indexB] := A[indexA]*2 + 1;
                    % Trace head down
                    head(p) := t;
                    t := p; p := q;
                END
                % From here the inspected node is atomic, thus set
                % the content of B to be negative.
                B[indexB] := -(A[indexA]*2 + 1);
                update(0,index);
                increment both indexA and indexB;
            END;
        END;
END;    % G_downhead;
```

Figure A.3.1.1b: The Graphical Tracing Algorithm (continued from previous page).

```
PROCEDURE g_downtail()
BEGIN
        WHILE (head of p is pointing at a cell) DO
        BEGIN
            a := tail(p);
            IF (q is not a cell) THEN
            BEGIN
                % equation A.3.1.1.2
                B[indexB] := A[indexA]*2 + 2;
                % return to branching node
                update(1,index);
                RETURN;
            END
            ELSE
            BEGIN
                IF (q has not been marked) THEN
                BEGIN
                    mark node q;
                    B[indexB] := A[indexA]*2 + 2;
                    IF (this is the first round in this loop)
                    THEN indexB := indexA - 1;
                    ELSE indexB := indexB + 1;
                END;
                IF (q is not a number) THEN
                BEGIN
                    tail(p) := t;
                    t := p; p:= q;
                    g_downhead();
                END;
                ELSE BREAK;
            END
        END;
        % An atom !
        B[indexB] := -(A[indexA]*2 + 2);
        update(1,index);
        increment indexA and indexB;
END;    % G_downtail
```

Figure A.3.1.1c: The Graphical Tracing Algorithm (continued from previous page)

```
PROCEDURE uphead();
BEGIN
        t := head(q);
        head(q) := p;
        p := q;
END;    % Uphead


PROCEDURE uptail();
BEGIN
        t := tail(q);
        tail(q) := p;
        p := q;
END;    % Uptail


PROCEDUE update(n,x : integer);
VAR temp, temp1 : integer;
BEGIN
        temp := B[indexB];
        temp := modulus of temp;
        IF (n=1) THEN
            % Tail -> reverse of equation A.3.1.1.2
            temp1 := (temp-2)/2;
        ELSE
            % HEAD -> reverse of equation A.3.1.1.1
            temp1 := (temp-1)/2;
        temp := 0;
        WHILE (B[temp] not equal to temp1) DO
        % Get index, value for the branching node
            temp := temp+1;
        indexA := temp;
END;    % Update
```

Figure A.3.1.1d: The Graphical Tracing Algorithm (continued from previous page)

APPENDIX IV
ICMS Microprogram Listing

```
# ******************************************************************************
#
#     Micro program for the ICMS : by K F Wong : April,1986;
#
#     Electrical Engineering Department of Edinburgh University
#
# ******************************************************************************


# ****************************** MACRO DEFINITION ******************************


include(header)

changequote([,])

# ****************************** CONDITIONS ************************************

define(_IF_POLLED,TESTsel=0)
define(_IF_CELL,TESTsel=1)
define(_IF_GC,TESTsel=2)
define(_IF_TO,TESTsel=3)
define(_IF_FROM,TESTsel=4)
define(_IF_FLIPPED,TESTsel=5)
define(_TRUE,TESTsel=6)
define(_FALSE,TESTsel=7)

# ****************************** BRANCH MACROS *********************************

define(_JZ,seq_instr=0)                    # jump zero
define(_JSR,[seq_instr=1,BA=$1,TESTsel=6])     # jump to subroutine
define(_CJSR,[seq_instr=1,BA=$1])              # conditional jsr (branch if HI)
define(_JMAP,seq_instr=2)                  # jump map
define(_JHI,[seq_instr=3,BA=$1])               # jump to BA if test=HI else pc
define(_JRP,[seq_instr=7,BA=$1, RE2911=0])     # if LOW repeat else jump to BA
define(_JLO,[seq_instr=9,BA=$1])               # jump to BA if test=LOW else pc
define(_RTN,[seq_instr=x'a,TESTsel=6])         # return from subroutine
define(_CONTINUE,seq_instr=14)                 # continue from pc
define(_NOP,seq_instr=14)                  # no operation = continue
define(_JBA,[seq_instr=15,BA=$1])              # jump always to BA

# ****************************** OTHER MACROS **********************************

# BERR:       assert BUS ERROR .

define(_BERR,[
     sel_139=0, A0_139=1, A1_139=1
])
```

```
# INCS:       increment SCAN pointer.

define(_INCS,[
     sel_139=1,                # enable 74LS139
     A0_139=0,A1_139=0              # increment S
])

# READS(x):    read in value of SCAN pointer and store  it
#            in RAM[x].

define(_READS,[
     ALU_instr=b'111101000,         # Y=S+Cn
     carry_h=0, carry_t=0,          # Y = S pointer
     oeb_h=1, oeb_t=1,              # direct input from B port
     OEy_h=0, OEy_t=0,              # enable Y o/ps
     addB = $1,                # setup RAM address
     IEN_h=0, IEN_t=0,
     add_buf_en=0, add_buf_dir=1,   # HEAD data at TAIL input
     sel_139=1,                # enable 74LS139
     A0_139=1,A1_139=0              # read S
])

# INCB:       increment B pointer .

define(_INCB,[
     sel_139=1,                # enable 74LS139
     A0_139=0, A1_139=1             # increment B
])

# READB(x):    read in the value of B pointer and store it
#            in RAM[x].

define(_READB,[
     ALU_instr=b'111101000,         # Y = S+Cn
     carry_h=0, carry_t=0,          # Y = B pointer
     oeb_h=1, oeb_t=1,              # disable b o/ps => DB inputs
     OEy_h=0, OEy_t=0,              # enable Y o/ps
     addB = $1,                # setup RAM address
     IEN_h=0, IEN_t=0,
     add_buf_en=0, add_buf_dir=1,   # HEAD data at TAIL input
     sel_139=1,                # enable 74LS139
     A0_139=1, A1_139=1             # read B
])

# START:       keep on polling the test input of the micro-
#            program sequencer, if it is 0 jump to MAP
#            ROM address else START again.

define(_START,[
START:
     seq_instr=3,              # conditional jump
     BA=0                      # back to START
])
```

```
# INPARA0(x):  read in a parameter from INPUT register 0.
#            INPUT register 0 is connected to the DAs
#            input of 2903. Parameter is read into
#            RAM[[x]s of both HEAD and TAIL ALUs.
#            "x" must be a number between 0 to 15.
#                 ************************
#                 *****  IMPORTANT  *****
#                 ************************
#            For a 2 parameters instruction INPUT reg 0
#            should be set last because by addressing
#            this register the ALUs are initiated and
#            the host processor is suspended.

define(_INPARA0,[
     ALU_instr=b'111101100,        # Y=R+Cn
     ea_h=1, ea_t=1,               # DAs
     addB=$1,                      # latch INPUT into RAM[x]
     IEN_h=0, IEN_t=0,
     OEy_h=0, OEy_t=1,             # disabled TAIL output
     data_buf_en=0,
     data_buf_dir=1,               # HEAD only
     IOsel=0, IOenable=0           # select INPUT register 0
])

# INPARA1(x):  same as INPARA1(x). INPUT register 1
#            outputs are connected to Y bus of the
#            ALUs; therefore to read them OEy of both
#            head nd tail have to be disabled first.

define(_INPARA1,[
     ALU_instr=b'111110001,        # don't care
     addB=$1,
     IEN_h=0, IEN_t=0,             # latch input into RAM[x]
     OEy_h=1, OEy_t=1,             # disabled output
     data_buf_en=0,
     data_buf_dir=1,               # HEAD only
     IOsel=1,IOenable=0            # select INPUT register 1
])

# OUTPARA2:    output parameter from RAM[x]  onto OUTPUT
#            register 2. This register has the same
#            add. as INPUT register 0 on the host processor
#            side. On the BS side its input is connected
#            to the B bus of the ALUs. It is use for
#            *****ADDRESS OUTPUT******.
#            EXTRA :
#            normally, output is latched from the HEAD ALU;
#            a 2nd argument (=2) can be supplied which would
#            output the TAIL content instead. (optional)

define(_OUTPARA2,[
     ALU_instr=b'111110001,        # don't care
     addB=$1,
```

```
    ifelse($2,2,[
    oeb_h=1, oeb_t=0,            # TAIL only
    add_buf_en=0, add_buf_dir=0,   # TAIL output at HEAD
    ],[
    oeb_h=0, oeb_t=1,            # HEAD only ])
    IEN_h=1, IEN_h=1,
    OEy_h=1, OEy_t=1,
    IOsel=2, IOenable=0          # select OUTPUT register 2
])

# OUTPARA3:    ouput parameter from RAM[x] onto OUTPUT
#            register 3. This register has the same
#            address as INPUT register 0 on the host
#            processor side.   On the BS side, its
#            input is connected to the Y bus of the
#            ALUs. It is use for DATA OUTPUT.
#            EXTRA :
#            normally, output is latched from the HEAD ALU;
#            a 2nd argument (=2) can be supplied which would
#            output the TAIL content instead (optional).

define(_OUTPARA3,[
    ALU_instr=b'111101100,        # Y=R+Cn
    carry_h=0, carry_t=0,
    addA=$1,
    ea_h=0,ea_t=0,
    ifelse($2,2,[
    OEy_h=1, OEy_t=0,            # TAIL only
    data_buf_en=0, data_buf_dir=0,  # TAIL output at HEAD
    ],[
    OEy_h=0, OEy_t=1,            # HEAD only ])
    IEN_h=1, IEN_t=1,
    IOsel=3, IOenable=0          # select OUTPUT register 3
])

# MEMWRITE(data,add,dataflag,addflag):
#            write into memory with address specified in
#            RAM[add] and data in RAM[data].  Dataflag
#            determines the source of the data, and it
#            takes 3 possible values:
#            ALLDATA = 0 HEAD and TAIL have independent
#                    data;
#            HDDATA  = 1 HEAD data at TAIL also;
#            TLDATA  = 2 TAIL data at HEAD also;
#            HDONLY  = 3 HEAD data only;
#            TLONLY  = 4 TAIL data only.
#            Similarly, addflag determines the source of
#            the address, and takes 2 possible values:
#            HDADD   = 1 HEAD address only;
#            TLADD   = 2 TAIL address only.
#                 ***************************
#                 *****  IMPORTANT   *****
#                 ***************************
```

```
#           Address bus is connected to the B bus and data
#           bus is connected to the Y bus of the ALUs.

define(ALLDATA,0)

define(HDDATA,1)

define(TLDATA,2)

define(HDONLY,3)

define(TLONLY,4)

define(HDADD,1)

define(TLADD,2)

define(_MEMWRITE,[
    ALU_instr=b'111101100,        # Y=R+Cn
    addA = $1,                    # Data = RAM[data]
    ea_h=0, ea_t=0,
    carry_h=0, carry_t=0,
    addB = $2,                    # Address = RAM[add]
    ifelse($4,1,[
    oeb_h=0, oeb_t=1,             # HEAD address only
    add_buf_en=0, add_buf_dir=1,  # HEAD address at TAIL
    ],$4,2,[
    oeb_h=1, oeb_t=0,             # TAIL address only
    add_buf_en=0, add_buf_dir=0,  # TAIL address at HEAD ])
    ifelse($3,0,[
    OEy_h=0, OEy_t=0,             # independent data
    data_buf_en=1,                # disable buffer
    ],$3,1,[
    OEy_h=0, OEy_t=1,             # HEAD data only
    data_buf_en=0, data_buf_dir=1, # HEAD data at TAIL
    ],$3,2,[
    OEy_h=1, OEy_t=0,             # TAIL data only
    data_buf_en=0, data_buf_dir=0, # TAIL data at HEAD
    ],$3,3,[
    OEy_h=0, OEy_t=1,             # HEAD data only
    data_buf_en=1, data_buf_dir=0, # disable data buffer
    ],$3,4,[
    OEy_h=1, OEy_t=0,             # TAIL data only
    data_buf_en=1, data_buf_dir=0, # disable data buffer ])
    ifelse($3,3,[
    mem_sel_h=0, mem_sel_t=1,     # disable TAIL memory
    ],$3,4,[
    mem_sel_h=1, mem_sel_t=0,     # disable HEAD memory
    ],[
    mem_sel_h=0, mem_sel_t=0,     # otherwise, enable both ])
    mem_read_h=0, mem_read_t=0,   # assert WRITE*
    clock=1                       # slow down clock
])
```

```
# MEMREAD(add,dataflag,addFLAG):
#     read data from memory specified by RAM[add].
#     Dataflag and addflag have the same meanings as
#     MEMWRITE. This macro takes 2 micro-instructions.
#     In the 1st cycle, address is set up in RAM[15].
#     In the 2nd, memory signals are asserted and data
#     is read into RAM[15]. But be then, memory address
#     has already been latched.
#         *************************
#         *****  IMPORTANT  *****
#         *************************
#     RAM[15] is used as the memory address register
#     during address setup and data register during
#     the actual READ operation.
#     Therefore it is vital that no valuable inform-
#     action is stored in it prior a MEMREAD operation.

define(_MEMREAD,[
    ALU_instr=b'111101100,        # Y=R+Cn
    addA = $1,                    # address = RAM[add]
    ea_h=0, ea_t=0,
    carry_h=0, carry_t=0,
    addB=15,                      # RAM[15] acts as the DATA register
    IEN_h=0, IEN_t=0,             # write into RAM[15]
    OEy_h=0, OEy_t=0;             # RAM[15] = RAM[add]

    ALU_instr=b'111100001,        # don't care
    addB=x'f,                     # set up address
    IEN_h=0, IEN_t=0,
    OEy_h=1, OEy_t=1,             # disable ALU output
    ifelse($2,0,[
    mem_oe_h=0, mem_oe_t=0,       # independent data
    data_buf_en=1,                # disable buffer
    ],$2,1,[
    mem_oe_h=0, mem_oe_t=1,       # HEAD data only
    data_buf_en=0, data_buf_en=1, # HEAD data at TAIL input
    ],$2,2,[
    mem_oe_h=1, mem_oe_t=0,
    data_buf_en=0, data_buf_en=0, # TAIL data at HEAD input ])
    ifelse($3,1,[
    oeb_h=0, oeb_t=1,             # HEAD address only
    add_buf_en=0, add_buf_dir=1,  # HEAD address at TAIL
    ],$3,2,[
    oeb_h=1, oeb_t=0,             # TAIL address only
    add_buf_en=0, add_buf_dir=0,  # TAIL address at HEAD ])
    clock = b'101,                # slow down clock
    mem_sel_h=0, mem_sel_t=0,
    mem_read_h=1, mem_read_t=1    # assert READ pulse
])

# TRANSFER(x,y): transfer content of RAM[x] to RAM[y].

define(_TRANSFER,[
```

```
    ALU_instr=b'111101100,        # Y=R+Cn
    addA=$1,                      # RAM[x]
    ea_h=0, ea_t=0,
    carry_h=0, carry_t=0,
    addB=$2,                      # RAM[y]
    OEy_h=0, OEy_t=0,
    IEN_h=0, IEN_t=0
])

# COPY(x,dir): copy content of register x  from :
#          HEAD to TAIL, dir=0, default; or
#          TAIL to HEAD, dir=1.

define(_COPY,[
    ALU_instr=b'111101100,        # Y=R+Cn
    addA=$1,                      # RAM[a]
    addB=$1,
    ea_h=0, ea_t=0,
    ifelse($2,1,[
    OEy_h=1, OEy_t=0,             # disable HEAD output
    data_buf_en=0,data_buf_dir=0,  # TAIL output at HEAD input
    IEN_h=0, IEN_t=1              # write into HEAD only
    ],[                           # default
    OEy_h=0, OEy_t=1,             # disable TAIL output
    data_buf_en=0, data_buf_dir=1,  # HEAD output at TAIL input
    IEN_h=1, IEN_t=0              # write into TAIL only])
])

# DTACK : assert data acknowledge signal.

define(_DTACK,BUSY=1)

# LATCH_COND : latch in condition; applicable to conditions which test for
#          the nature of a cell pointer ie CELL, TO & FROM.

define(_LATCH_COND_CELL,[sel_139=0, A0_139=0, A1_139=1])
define(_LATCH_COND_TO,[sel_139=0, A0_139=0, A1_139=1])
define(_LATCH_COND_FROM,[sel_139=0, A0_139=0, A1_139=1])
define(_SAMPLE_GC_STATUS,[sel_139=0,A0_139=1,A1_139=0])


# ******************************************************************************
# ************************* MAIN STARTS HERE ************************************
# ******************************************************************************

ORG x'fff000;

ENTER:                        # start again
    _IF_GC, _CJSR(SCAN);          # garbage collect if required

                              # GC cond is latch once at the start
    _SAMPLE_GC_STATUS,            # ready to be tested in later cycle
    _IF_POLLED, _JHI(ENTER);      # Is it polled? repeat itself if not
```

```
    _JMAP;                  # jump map

    _NOP;                   # pad out

    _NOP;

    _NOP;

# ************************* OPCODE MAPPING ***********************************

MAP6:  _JBA(car);

MAP7:  _JBA(cdr);

MAP8:  _JBA(replacar);

MAP9:  _JBA(replacdr);

MAPa:  _JBA(cons);

MAPb:  _JBA(setreg3);

MAPc:  _JBA(setreg4);

MAPd:  _JBA(setreg0);

MAPe:  _JBA(setreg1);

MAPf:  _JBA(setreg2);

MAP11: _JBA(readreg0);

MAP12: _JBA(readreg1);

MAP13: _JBA(readreg2);

# ****************** MAPPING FUNCTION DECLARATION ****************************
# All routines will latch parameter into reg8  and reg9 is used as a working
# register;  return value is store in reg15.   Registers 0 to 7 are ROOTs.

# ********************************** CAR(X) ************************************
car:

    # car(X): return the HEAD (car) element of X.

    _INPARA0(8),            # read in parameter X
    _LATCH_COND_CELL;           # latch in CELL bit

    _IF_CELL, _JHI(ERR);        # if it's an atom => error

                    # gc X if necessary
                    # genuine cell pointer in REG15
    _JSR(MOVE);
```

```
    _MEMREAD(15,HDDATA,HDADD);     # fetch HEAD of X

    _JBA(EXIT),
    _OUTPARA2(15);              # TEST

# ********************************** CDR(X) ************************************

cdr:
    # cdr(X): return TAIL (cdr) element of X.

    _INPARA0(8),                # read in parameter
    _LATCH_COND_CELL;           # latch in CELL bit

                        # if it's an atom => error
    _IF_CELL, _JHI(ERR);

                        # gc X if necessary
                        # genuine cell pointer in REG15
    _JSR(MOVE);

    _MEMREAD(15,TLDATA,HDADD);     # fetch TAIL of X

    _JBA(EXIT),
    _OUTPARA2(15);              # set up return value

# ***************************** REPLACAR(X,Y) ********************************

replacar:

    # replacar: replace the HEAD (car) of X with Y.

    _INPARA1(8),                # read in Y then MOVE
    _JSR(MOVE);

    _INPARA0(8),                # read in X
    _LATCH_COND_CELL;           # latch in CELL bit

    _TRANSFER(15,9),            # store result from last MOVE
    _IF_CELL, _JHI(ERR);        # if X is an atom => error

                        # gc X if necessary
                        # genuine cell pointer in REG15
    _JSR(MOVE);

    # both X and Y are now genuine cell pointers
    _MEMWRITE(9,15,HDONLY,HDADD);  # REPLAce CAr of X

    _MEMREAD(15,ALLDATA,HDADD);    # TEST

    _JBA(EXIT);                 # no return value

# ******************************** REPLACDR(X,Y) ********************************
```

```
replacdr:

    # replacdr: replace the TAIL (cdr) of X with Y.

    _INPARA1(8),            # input Y then MOVE
    _JSR(MOVE);

    _INPARA0(8),            # read in X
    _LATCH_COND_CELL;           # latch in CELL bit

    _TRANSFER(15,9),            # store previous MOVE result
    _IF_CELL, _JHI(ERR);        # if X is an atom => error

                    # gc X if necessary
                    # genuine cell pointer in REG15
    _JSR(MOVE);

    _MEMWRITE(9,15,TLONLY,HDADD);  # REPLAce CDr of X

    _MEMREAD(15,ALLDATA,HDADD);    # read return values

    _JBA(EXIT);             # no return value

# ******************************** CONS(X,Y) ************************************

cons:

    # cons(X,Y) : construct a new cell with HEAD=X & TAIL=Y.

    _INPARA0(8),            # input X
    _JSR(MOVE);

    _TRANSFER(15,9);            # X is updated

    _INPARA1(8),            # input Y
    _JSR(MOVE);             # genuine Y is in reg15

    _READB(8);

    _MEMWRITE(9,8,HDONLY,HDADD);   # HEAD of new cell = X

    _MEMWRITE(15,8,TLONLY,HDADD);  # TAIL of new cell = Y

    _TRANSFER(8,15),            # set return value
    _INCB,                  # increment Bptr
    _IF_FLIPPED, _JHI(.+2);     # if system has just flipped
                    #       &&
    _IF_GC,_JHI(ERR);           # gc has not yet finished => ERROR

    _OUTPARA2(15),              # output return value
    _JBA(EXIT);             # start again

# ******************************** SETREG0,1,2,3,4(X) ***************************
```

```
setreg0:

    # setreg(X): register0 = ROOT[0] = environ = X

    _INPARA0(15);

    _TRANSFER(15,0);

    _JBA(EXIT),
    _OUTPARA2(0);

readreg0:
    _JBA(EXIT),
    _OUTPARA3(0);           # set return value

setreg1:

    # setreg1(X): register1 = ROOT[1] = X

    _INPARA0(15);

    _TRANSFER(15,1);

    _JBA(EXIT),
    _OUTPARA2(1);           # set return value

readreg1:
    _JBA(EXIT),
    _OUTPARA3(1);

setreg2:

    # setreg2(X): register2 = ROOT[2] = X

    _INPARA0(15);

    _TRANSFER(15,2);

    _JBA(EXIT),
    _OUTPARA2(2);           # set return value

readreg2:
    _JBA(EXIT),
    _OUTPARA3(2);

setreg3:

    # setreg3(X): register3 = ROOT[3] = X

    _INPARA0(15),
    _LATCH_COND_CELL;       # latch in CELL bit

    _IF_CELL, _JHI(.+2);
```

```
    _TRANSFER(15,3);

    _JBA(EXIT),
    _OUTPARA2(3);              # set return value

setreg4:

    # setreg4(X): register4 = ROOT[4] = X

    _READS(15);
    _OUTPARA2(15);
    _JBA(EXIT);

    _INPARA0(15),
    _LATCH_COND_CELL;          # latch in CELL bit

    _IF_CELL, _JHI(.+2);

    _TRANSFER(15,4);

    _JBA(EXIT),
    _OUTPARA2(4);              # set return value

# ********************** SUPPORTING ROUTINES ************************************


# ****************************** MOVE(X) ****************************************

define(_ENABLE_PORTB_OUTPUT,[oeb_h=0,oeb_t=0])

MOVE:

    # reg14 is contaminated !!

                        # transfer X from reg8 to 15
                        # => pre-load return value
    _TRANSFER(8,15),
    _ENABLE_PORTB_OUTPUT,      # expose reg[15] for testing
    _LATCH_COND_FROM;          # latch in FROM bit

    _IF_FROM, _JHI(NOTFROM);

YESFROM:

    _MEMREAD(8,ALLDATA,HDADD),     # fetch Xo, 2 cycles
    _LATCH_COND_TO;            # latch in TO bit

    _IF_TO, _JLO(NOTFROM);

NOTTO:
    _READB(14);

    _MEMWRITE(15,14,ALLDATA,HDADD); # content to be copied is in REG15
```

```
    _MEMWRITE(14,8,HDONLY,HDADD);  # leave forwarding address

    _TRANSFER(14,15),          # set return REG15 with Bptr
    _INCB,                     # increment Bptr
    _IF_FLIPPED, _JHI(.+2);    # if system has just flipped
                       #       &&
    _IF_GC, _JHI(ERR);         # gc has not yet finished => ERROR

NOTFROM:
    _COPY(15),                 # return value= HEAD15= TAIL15
    _RTN;                      # return from subroutine

# ***************************** SCAN ******************************************

SCAN:                          # Scavenge, 1 gc burst operation

    # reg8, 9, 14 and 15 are used


    _READS(9),
    _IF_FLIPPED, _JHI(S1);     # if just flipped, do the ROOTS first

    # Garbage Collection starts from ROOTS

    _TRANSFER(0,8),            # move ROOT0
    _JSR(MOVE);

    _TRANSFER(15,0);           # update

    _TRANSFER(1,8),            # move ROOT1
    _JSR(MOVE);

    _TRANSFER(15,1);           # update

    _TRANSFER(2,8),            # move ROOT2
    _JSR(MOVE);

    _TRANSFER(15,2);           # update

    _TRANSFER(3,8),            # move ROOT3
    _JSR(MOVE);

    _TRANSFER(15,3);           # update

    _TRANSFER(4,8),            # move ROOT4
    _JSR(MOVE);

    _TRANSFER(15,4);           # update

S1:

    _MEMREAD(9,TLDATA,HDADD);
```

```
                    # next, move TAIL of S
    _TRANSFER(15,8),          # HEAD of reg8 := TAIL of reg8
    _JSR(MOVE);

    _MEMWRITE(15,9,TLONLY,HDADD);  # update TAIL content of S

    _MEMREAD(9,HDDATA,HDADD);     # Scavenge starts here

                    # first, move HEAD of S
    _TRANSFER(15,8),          # set up parameter for MOVE
    _JSR(MOVE);

    _MEMWRITE(15,9,HDONLY,HDADD);  # update HEAD content of S


    _INCS,                # next cell to be scanned
    _RTN;

# ****************************** ERR ******************************************

ERR:  _BERR;                # generate bus error signal

                    # restart
    _JRP(ENTER);

# ****************************** WARM START ************************************

EXIT:
    _DTACK ;              # assert DTACK

                    # start again
    _JRP(ENTER);

# ***************************** END OF PROGRAM *********************************

# ************************** FOOT NOTES ****************************************
# N.B. All macro starts with an underscore;
#     main functional routines are in small letters;
#     supporting routines are in capital letters.
# ******************************************************************************
```

APPENDIX V
Related Publications

Paper1:

"A Specialised Microcomputer System for the Application of Artificial Intelligence Techniques to Engineering" by K.F. Wong, G.G. Coghill and J.M. Hannah in the Proceedings of the 10th Annual Workshop on Microcomputer Applications organised by Strathclyde University in 8-10 September 1986 at Glasgow, Scotland. [73]

Paper2:

"THESIS : The Hardware Environment for Small Intelligent Systems", by K.F. Wong, G.G. Coghill and J.M. Hannah in pages 173-176 of the Proceedings of the 86 Conference on Personal & Small Computers organised by ACM SIGSMALL/PC in 2-5 December 1986 at San Francisco of the U.S.A. [74]

Paper3:

"ICMS: Intelligent Cell Memory System", by K.F. Wong, G.G. Coghill and J.M. Hannah in pages 168-172 of the Proceedings of the 86 Conference on Personal & Small Computers organised by ACM SIGSMALL/PC in 2-5 December 1986 at San Francisco of the U.S.A. [71]

Note: In this appendix, all related publications are photo copies of the relevant proceedings. The original page numbers are retained. Page numbers with repect to this thesis are shown on the cover sheets of each of the papers.

Tenth Annual Microcomputer Applications Workshop

Department
of
Computer Science

UNIVERSITY OF STRATHCLYDE

8th to 10th September 1986

Strathclyde Business School

Department of Computer Science
University of Strathclyde
Glasgow, Scotland

- pages 180 to 192 -

Tel: 041-552 4400

A Specialised Microcomputer System for the Application of Artificial Intelligence Techniques to Engineering

K.F. Wong,BSc., J.M. Hannah,PhD. and G.G. Coghill,PhD.

Department of Electrical Engineering
Edinburgh University
The King's Buildings
Edinburgh EH9 3JL.

ABSTRACT

There is a growing interest in the application of Artificial Intelligence (AI) techniques in engineering. Existing AI systems are not suitable for many applications because of their unacceptable real-time response and complexity. In this paper an approach to system design which offers a potential solution to these problems is presented. The Hardware Environment for Small Intelligent Systems (THESIS) is based on a loosely-coupled bus architecture incorporating five basic functional modules: Input/Output Unit, Control Unit, Inference Engine, Knowledge Base and Intelligent Cell Memory System (ICMS). This design approach makes possible simple, flexible, expandable systems which can be easily implemented using standard microprocessor technology. The design of the ICMS provides virtually transparent garbage collection resulting in greatly enhanced real-time performance. A prototype implementation of an ICMS which performs parallel and incremental garbage collection ensuring a fast recovery rate and bounded separation times is described. The prototype system is based on the VME bus using mainly standard hardware to meet the design requirements that it should be easy-to-use, flexible, portable and suitable for engineering applications where fast response time is required. Results are presented which illustrate the dramatic improvement in real-time performance which can be achieved with this approach to system design.

1. Introduction

The adoption of AI techniques in engineering applications is becoming increasingly popular. AI in engineering combines the advantages of both automatic and manual systems; namely, the precision and robustness of the former; and the flexibility and simplicity, of the latter. At present, however, existing hardware has two main drawbacks which have been hindering more widespread use. They are :

1. bulk and complexity;

2. poor real time performance caused by the garbage collection process.

In may cases practical systems for engineering applications have to be small [1] in order to be easily transportable from one site to another. The ability to response instantaneously to external stimuli is often an important requirement for engineering systems.

A brief outline of the characteristics of intelligent systems, leading to an understanding of their shortcomings, is given in section two. In the third section, the proposed design approach is described A prototype system has been constructed and sections five and six depict some of the design details. Experimental results are given in section six and conclusions are drawn in section seven.

2. Intelligent Systems

AI is a science which attempts to provide machines (artificial entities) with human-like behaviour, such as the ability to store and acquire knowledge and make use of it to reason and to act on deductions as a human being would. Since the beginning of this decade, AI has started to break away from the laboratory environment and its potential is gradually being realised in many practical areas.

An intelligent system in this context is defined as a hardware/software system which exhibits AI capabilities. Basically, it consists of the following parts[2,3] (figure 1) :

1. a Knowledge Base (KB);

2. an Inference Engine (IE);

3. an Input and Output Interface (IO); and

4. a Control Unit (CU).

The KB is a massive data base consisting of facts which vary with the application i.e. domain-dependent. The IE encompasses rules and guidelines for heuristic deduction. These rules are problem-dependent - a set of solving techniques is applicable to the same class of problem, e.g. methods for disease diagnosis can be applied to fault diagnosis with minimal alteration. The IO interface provides the communication paths with the external world. These could be stimuli such as electric signals from transducers or input information keyed in from a terminal. The CU is responsible for directing the "traffic" within the system. It is essentially a program environment which consists of entities such as a run-time stack, static and dynamic variables and their associations, some free memory space and the program code.

2.1. Reasoning

Initially, the KB and the IE are pre-programmed with relevant information. When queries enter from the IO, the CU reacts promptly and monitors them. Accordingly, information frameworks are generated in the CU by applying IE's rules to the KB. The results are fed back into the CU which will send a response to the intended recipient - either to the IO interface asking for extra input or to the IE creating more tasks. Effectively, the CU's decision is made by applying heuristical inference techniques on "past experience".

2.2. Learning

Learning is the ability to acquire new knowledge. The normal way of learning is by *education*. New information is fed into the system manually through the IO interface. In addition, intelligent systems are capable of acquiring information automatically through learning by *analogy*. When a spurious signal, which the system has had no previous experience of, is read from the IO, the system endeavours to perform educated guesses using past experience. The final solution and the transformation steps leading to the solution are associated with the spurious input signal. Together they are remembered/stored. If, however, no solution can be derived, the system will attempt to summon human aid. Theoretically once an intelligent system, with a learning capability, is furnished with some basic knowledge, it can adapt itself to the surrounding environment.

2.3. The Shortcomings of Existing Systems

At present, intelligent systems are mostly implemented in two ways:

- software realisation on a conventional serial computer; OR novel computers with special architectures.

The first approach is the simplest. A well known example is an Expert System whose IO signals are interactive dialogue from a VDU. For engineering, existing intelligent systems of this type are only applicable to situations which are not very time critical; situations such as fault diagnosis, consultancy, engineering design aids, etc[4]. In these situations, a snap shot of the input is taken and upon which inferential procedures are applied. The solution may appear after some length of time, depending on the nature of the problem and the state of the processor.

Because of the way in which memory is dynamically allocated within AI systems the processor may temporarily run out of available memory. When this happens a time-consuming process known as **garbage collection**[5] is invoked for memory reclamation. Its responsibility is to recycle potentially reusable memory. When this happens, all other processing is stopped which results in occasional system interruptions. These processing interruptions have prevented the use of intelligent systems in real time engineering applications. Several methods [6] have been devised to shorten the interruption period. Fundamentally, they are variants of either the parallel collection scheme which exhibits a

very fast memory recovery rate with small but variable suspension intervals; or the incremental collection scheme which has small and bounded suspension intervals but a slow memory recovery rate. The preferred choice between the two is application dependent.

As hardware costs decrease, the implementation of intelligent systems using special architectures is an increasingly promising approach. Since the beginning of this decade, many special purpose AI machines have been launched into the market. These machines are mainly built for research and development applications and are powerful with support for a rich programming environment, e.g. a complete set of design tools. Nevertheless, the lack of hardware standards, particularly in interfacing, makes system expansion difficult. In addition, these machines are usually large - too large, in fact, for many on-site engineering applications. Their customised design often makes them unsuited to a wide range of operations.

3. THESIS[7]

The Hardware Environment for Small Intelligent Systems (THESIS) is a hardware design concept intended to overcome some of the technical difficulties in the application of AI to engineering. A major feature of THESIS is its hardware modularity based on a standardised interface. Individual blocks are implemented as separate hardware modules whose functions are independent. An additional module Intelligent Cell Memory System (ICMS) further enhances the system. Basically, this is a reservoir of free storage specifically set aside for dynamic memory allocation. Inter module communication is controlled globally by the Control Unit (CU) under a standard protocol. The CU is simply a conventional processor unit with some local heap memory. The structure of THESIS is shown in figure 2.

At first sight THESIS does not seem to be very different from classical intelligent systems. One difference is in the partitioning of storage between the CU and the ICMS. In classical intelligent systems, programs and data mingle together in the heap. In so doing, they exhibit *referential transparency*, one of the unique features of AI languages [8]. There is no difference between program and data - both are classed as symbols or objects. Each object is constructed from one or more cells† and has some properties associated with it. Properties may be either a collective list of other objects or a list comprising some functional specifications. The actual constituents are only reviewed at run-time. Thousands of objects form a program environment and new objects are created continuously whilst old ones are discarded during program execution.

Because AI programs are generally large and complex, with code and data stored in one common area, system performance can easily be degraded. For instance, if the memory is mostly filled with program code, it would not take long before memory exhaustion occurred. Moreover, the frequent need for garbage collection would cause intolerable

† The basic storage unit.

interruptions. In practice, programs are mostly permanent objects which make garbage collection on them not worthwhile.

The reason for separating the ICMS from the heap is to facilitate the division of program and data. Programs are stored in the heap within the CU. Since program codes are instructions to direct the CU, this ensures direct access to instructions from it. Cache memory can be used to further increase the rate of instruction fetching. For a large software environment, secondary memories, with page scheduling, can be attached to the heap. Data is restricted to the ICMS only. The dynamic memory allocation process will create and release cells from this region.

Unfortunately, ICMS/heap separation has partially sacrificed the characteristic of referential transparency. However, this may not be essential in an engineering environment. This contrasts with a development environment, where referential transparency is crucial because every object is dynamic and may be subject to change. In engineering applications, programs are well defined and mostly static. It is not worth including this code in the ICMS because this would do nothing but increase congestion. When objects become permanent during program execution, e.g. experiences which are learned, the ICMS can transfer them to other non-temporary memory modules (the IE and the KB).

Separate segments for programs and data allows more room for the CU to work on and the frequency of garbage collection is reduced. Also, the rate of the garbage collection process is speeded up due to the fact that the collector is dealing with cells only - cells are generally temporary entities[9] and so they required less effort to recycle. To rely only on the ICMS for cell objects, greatly reduces the CU's search paths and processor throughput is increased.

4. The Prototype System

A prototype system has been developed based on the THESIS design philosophy. This has largely been constructed from standard circuit boards and is based on the industry standard VMEbus. Figure 3 shows the structure of the prototype system. The CU is the host processor, a M68000, on an off-the-shelf VMEbus compatible CPU board. The on-board 128k dynamic RAM is used as the program space where the inference rules reside (IE). The KB is another standard VMEbus compatible memory board which has 512k bytes of storage. The I/O module is application dependent and could be selected from many VMEbus based I/O modules on the market, e.g. serial-in-parallel-out buffers, analogue to digital converters, etc. The ICMS is the only specialised board and the key to the improved system performance. The design and construction of the ICMS is described below.

5. The ICMS[10]

There are two submodules within the ICMS, see figure 4. They are the cell memory (CM) and the intelligent cell memory controller (ICMU, the controller for short). A cell comprises of two words, a head and a tail, and several tag bits. Abandoning the orthodox word addressing

mechanism, memory in the CM is addressable by cells. This is achieved by configuring storage into two banks and sacrificing one bit of each bank for tagging. The controller is a processor dedicated to extracting information from the CM and subsequently passing it on to the CU. It also monitors the state of the CM frequently; and performs garbage collection if the CM is exhausted. Physically, the controller is constructed with bit slice devices[11] under microprogram control. A writable control store (WCS), 256x64, has specially been constructed for this purpose. The flexibility of bit slice design is most suitable for this project because it has left open the cell size of the ICMS†.

Normally, the controller is constantly performing garbage collection. This is done in bursts and each garbage collection burst is indivisible - a unit operation. The CU is not aware of this happening and busily executing its own tasks. When the CU wants to request a cell operation, the controller is interrupted. It will then finish up with its present garbage collection burst and service the cell operation requested.

Baker's garbage collection algorithm[12] is used. A novel *dual* mode of operation of the algorithm is adopted in the ICMS. While the ICMS is garbage collecting invisibly from the CU, the *parallel* mode is active. Upon interruption, the ICMS garbage collects *incrementally* ensuring at least one free cell is available. A dual mode garbage collector offers a fast cell recovery rate with bounded collection intervals. Moreover, the speed of operation is enhanced by the following features.

5.1. Simplicity

From the M68000 the ICMS is seen as a passive storage device, accessible by writing or reading a bank of registers (IN0,1 and OUT2,3). Each register is responsible for a LISP primitive and is assigned with an address. Effectively, writing into a register is equivalent to passing a parameter and invoking the corresponding list operation at the same time. At the end of the function, return values are read from the same address.

The entire transaction is performed asynchronously. According to the VMEbus protocol, the address strobe signal (AS*) is sent by the M68000 host to inform the ICMS of the availability of some parameters. The ICMS then fetches them from the registers and performs the requested LISP primitive while AS* is still asserted. During this period, the host is suspended. Upon completion, the results are put back into the corresponding registers and the data transfer acknowledge signal (DTACK*) is set which activates the M68000.

---

† At present the ICMS is set up to be 48 bits wide, with 2 bits for tagging.

5.2. Fast Response

Although the throughput of the host processor will be reduced by the frequent requirement for garbage collection bursts, the design of the ICMS ensures that these take place at an acceptable speed by adopting several architectural features:

i) Cell Unit-addressing - Two bit slice ALUs are employed, one is responsible for the manipulation of the HEAD pointer and the other is dedicated to the TAIL. The operational speed for list functions on the cell is therefore enhanced. Moreover, this architecture does not exclude the possibility of either only one ALU or one address/data being required. In such circumstances, the unnecessary device may be made redundant by disabling it using the appropriate micro control bits.

ii) Hardware Testing - Five tests are performed in hardware concurrently with the bit slice ALUs, namely

- *CELL(X)* is X a cell pointer;
- *OLD(X)* does cell ponter X exist in the OLD region;
- *NEW(X)* does cell pointer X exist in the NEW region;
- *GC* has garbage collection finished ? and
- *CMEND* has the cell memory been exhausted ?

iii) Content Prefetching - This idea derives from "instruction prefetch" in computer architecture design. Running in parallel with hardware testing, it ensures the correct address at the memory address port at the end of a conditional fetch instruction. Thus test and fetch instructions are executed in one machine cycle which otherwise would be two or more.

5.3. Flexibility

The ICMS is not restricted to a particular type of processor. Theoretically, because the bus standard has been conformed to, it should be compatible with any host system.

6. System Performance

The ICMS prototype system is embedded in a custom LISP environment, known as simpleLISP. This is a simplified version of pure LISP, but it does include all the basic LISP primitives. The most time consuming primitive implemented is CONS(X,Y) which demands a new cell, checks the regions in which X and Y are located, performs collections on X and/or Y when necessary and finally places X and Y into the HEAD and TAIL of the new cell, respectively. Normally, there is no problem in cell allocation until no free cells remain. At this point, garbage collection is initiated. An analysis shows that the worst case of CONS is the most time consuming operation in the overall system. For the M68000

CPU with a 8 MHz clock, the measured time is 1.8 ms. Writing the algorithm in micro-code and running the operation on the ICMS gives a worst case CONS time of 8.3 μs.

Compared with the software implementation in the host, the ICMS is obviously much superior giving a greater than two orders of magnitude improvement in the worst case performance. Similar comparisons for other LISP primitives have been carried out and their results are shown in table 1. These results give a good indication of the improvement in overall system performance because application programs are mainly constructed from these primitives. In practice, the actual overall improvement factor will vary somewhat depending on the application.

A second experiment was arranged to verify that the ICMS does ensure a bounded response time. Initially, an useful list is created. The number of constituent cells was increased gradually each time, in the increment of 10% of the overall cell population. Every time 1000 CONS operations were performed, and the times of execution were recorded. As shown in table 2, the exection time is, in fact, fairly constant. On average, it is 55.5ms with a relative percentage error bound of ±0.2%.

7. Conclusion

THESIS is a design concept which overcomes many existing technical difficulties leading to the inclusion of AI functionality in engineering systems. The concept is based on hardware modularity and on a standard bus interface scheme. A prototype system has been implemented which is based on on the VMEbus specification. Standard circuit boards have been employed for the IE,KB and CU. The ICMS is specially designed using bit slice devices which is dedicated to list processing. Using it, the worst case speed of list manipulation on a M68000 system has been increased by more than two orders of magnitude.

In the future, realisation of the KB and the IE could take two forms. They could either be completely passive memory devices; in which case information retrieval would controlled by the CU. Alternatively, special purpose controllers, similar to the ICMS, could be implemented at the memory front-end to expedite the operational speed. Some suggestions are:

- to design the KB controller with multiple memory access modes such as random access, content addressing, hashing, etc.; and
- to adopt a logic driven architecture for the IE controller.

The implementation of the I/O unit will be application dependent. In some applications, for instance data logging, analogue-to-digital converters, filters, amplifiers, etc. would be required.

In many cases the widespread use of AI techniques in practical engineering systems still awaits the development of suitable applications software. The THESIS design approach described here offers a powerful environment to use this software.

References

1. H.W. Whittington and G.G. Coghill, "Hand-Held Digital Echo Sonic Pile Testing Systems," *Proceedings of International Conference on Structural Repair*, pp. 173-176 (1983). ISBN no: 0-947644-03-2

2. D.S. Nau, "Special Feature: Expert Computer Systems," *Computer*, pp. 63-85 IEEE, (Feb., 1983).

3. U.S. Department of Commerce, *An Overview of Expert Systems*, National Aeronautics and Space, Washington, D.C. (May, 1982).

4. Strathclyde University, *Proceedings of the Current Trends in the Application of Expert Systems*, IEE Scottish Centre Electronics Computer & Control Section, Glasgow, Scotland (2 April 1986).

5. D.E. Knuth, *The Art of Computer Programming Vol. I: Fundamental Algorithms*, Addison-Wesley, Reading, Mass. (1973).

6. J. Cohen, "Garbage Collection of Linked Data Structures," *Computing Surveys*, Vol 13, (3) pp. 341-367 The Association of Computing Machinery, (Sept. 1981).

7. K.F. Wong, G.G. Coghill, and J.M. Hannah, "THESIS: The Hardware Environment for Small Intelligent Systems, for Engineering Applications," *Proceedings of the 86 Conference on Personal & Small Computers*, ACM SIGSMALL/PC, (2-5 Dec, 1986). (to be published)

8. B. Barr and E.A. Feigenbaum, *The Handbook of Artificial Intelligence*, Pitman (1983).

9. H. Liberman and C. Hewitt, "Garbage Collection Based on the Lifetimes of Objects," *MIT AT Memo no.569* , M.I.T., (1981).

10. K.F. Wong, G.G. Coghill, and J.M. Hannah, "ICMS: An Intelligent Cell Memory System For Real-Time Engineering Applications," *Proceedings of the 86 Conference on Personal & Small Computers*, ACM SIGSMALL/PC, (2-5 Dec, 1986). (to be published)

11. J. Mick and J. Brick, *Bit Slice Microprocessor Design*, McGraw-Hill (1980).

12. J. Baker, "List Processing in Real Time on a Serial Computer," *Communications of the ACM*, Vol. 21, (4) pp. 280-294 (April, 1978).

LISP Primitives	The worst case no. of GC bursts	68K machine times (x μsec)	ICMS machine times (y μsec)	Improvement Factors (x/y)
CAR, CDR	1	165.8	6.2	31.9
REPLACA, REPLACD	2	453.8	7.5	60.5
CONS	11	1815.0	8.3	220

Table 1: Improvement in Performance for Various LISP Primitives.

% of Useful Cells	ICMS m/c times (ms)
1	55.6
10	55.6
20	55.4
30	55.6
40	55.4
50	55.4
60	55.5
70	55.4
80	55.4
90	55.6
99	55.4
average time : 55.5±0.1 ms	

Table 2: ICMS Execution Times for 1000 Consequtive CONS.

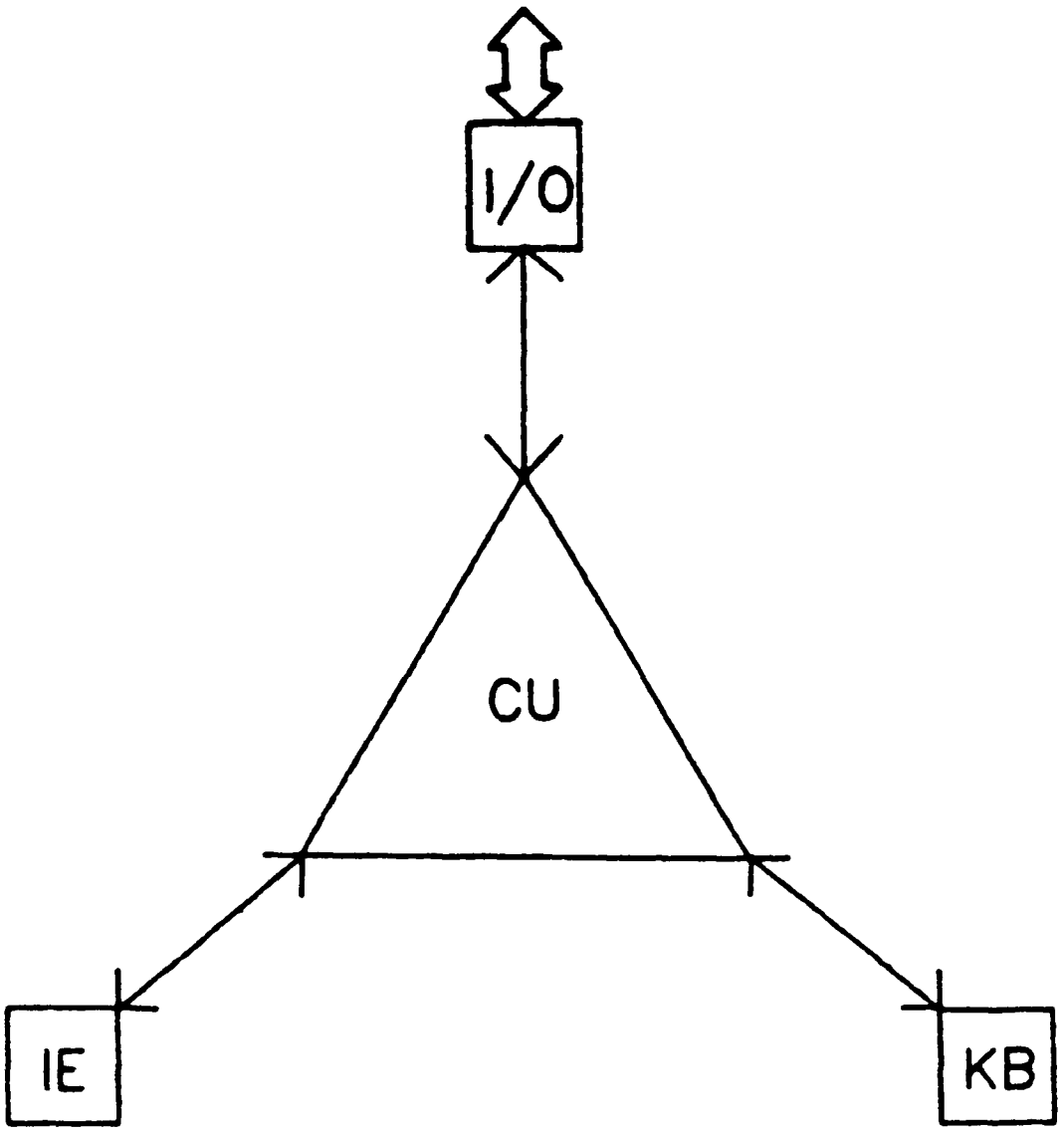


Figure1: Anatomy of an Intelligent System.

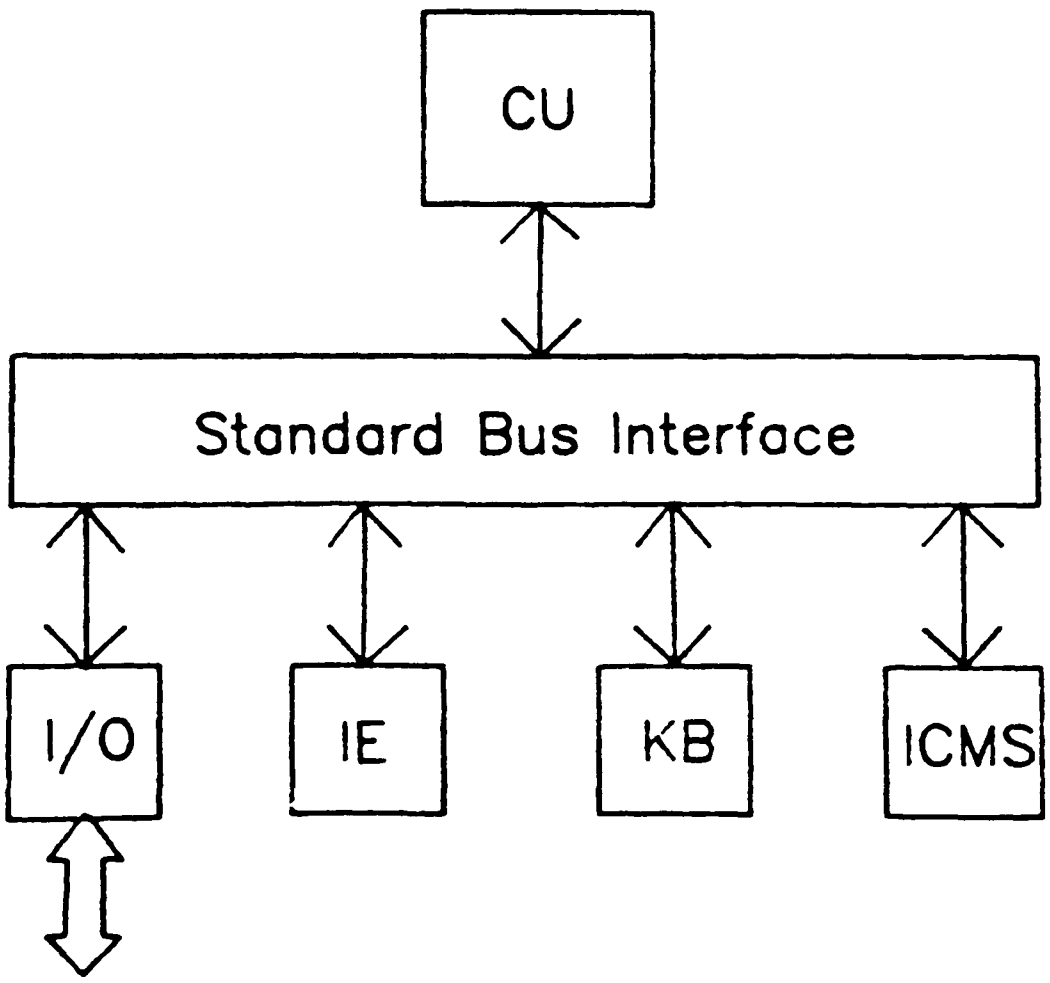


Figure2: THESIS – The Hardware Environment for Small Engineering Systems.

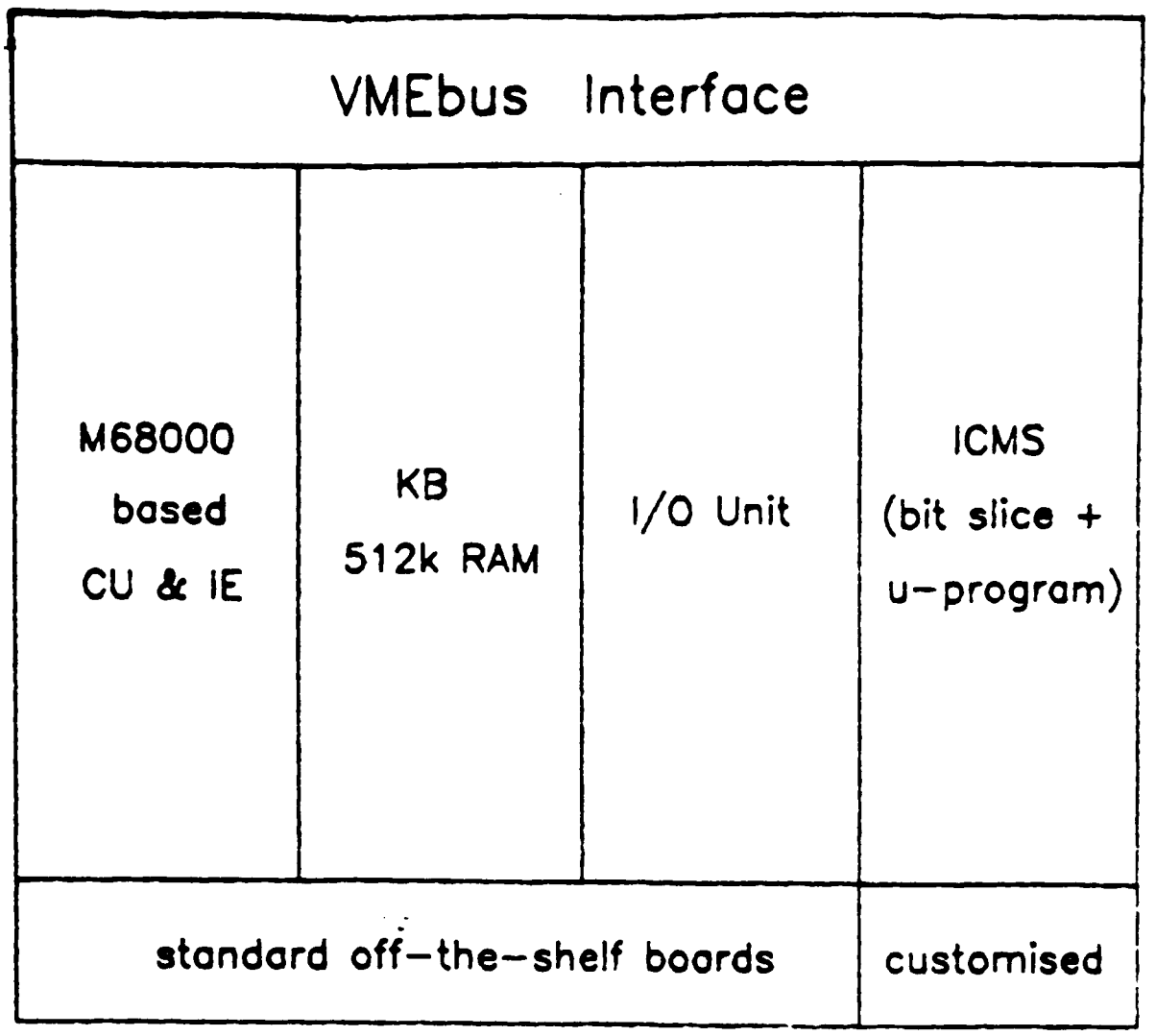


Figure3: The Prototype System.

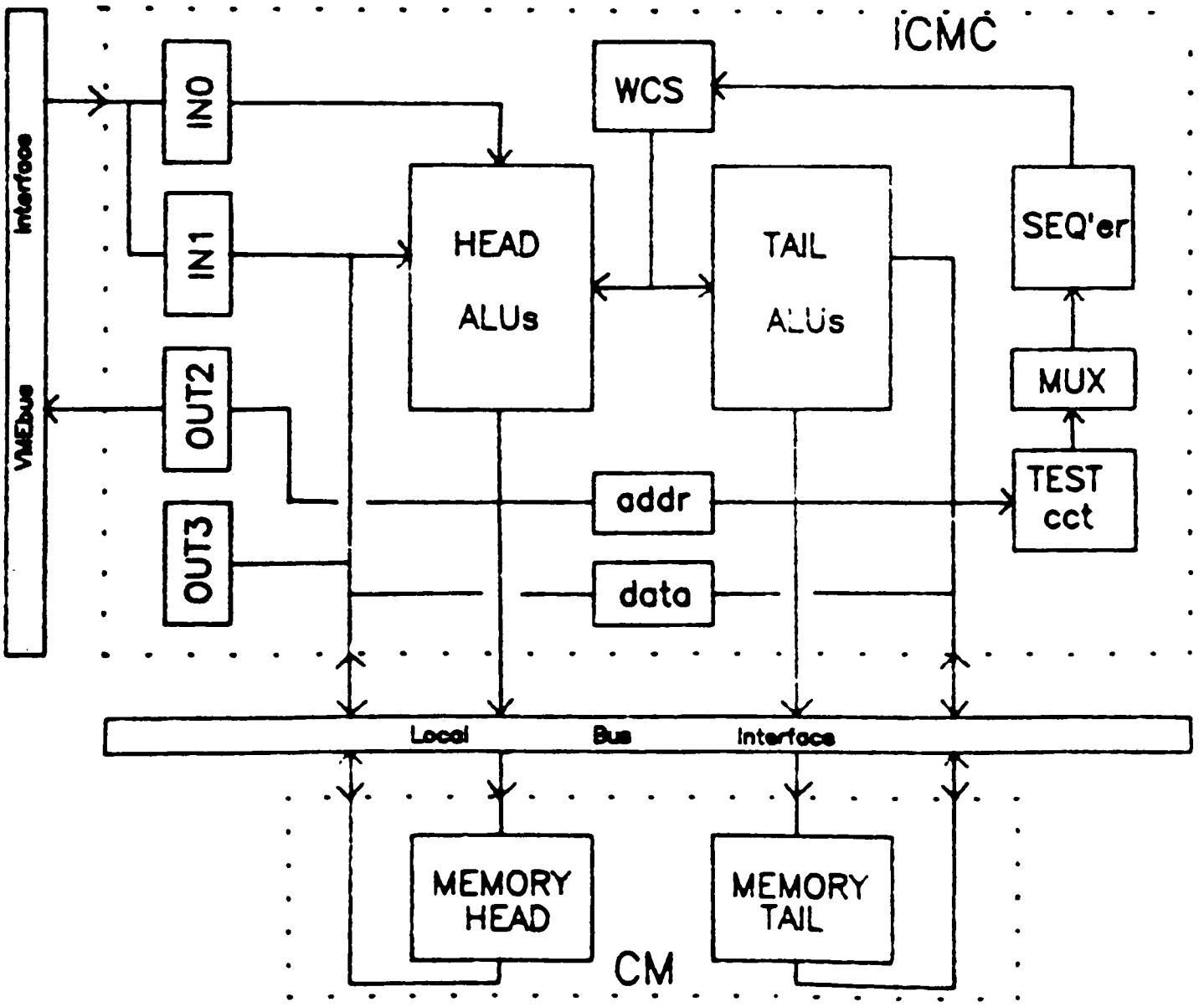


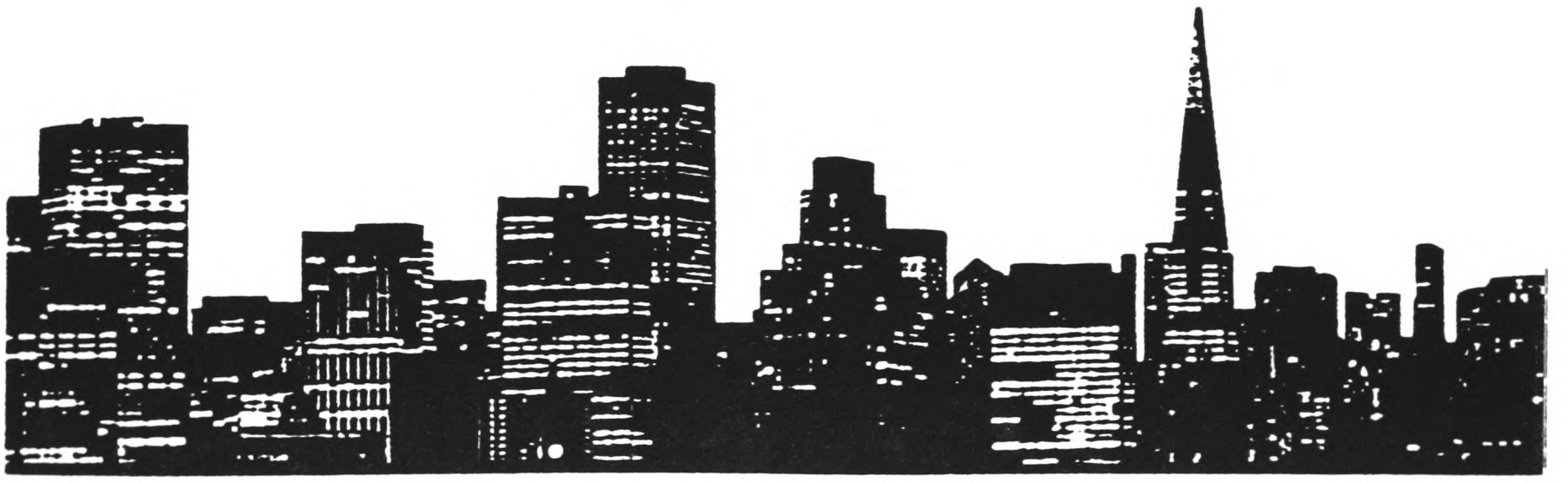
Figure4: The Schematic Block Diagram of the ICMS.

PROCEEDINGS

1986 ACM SIGSMALL/PC
Symposium on Small Systems

Sir Frances Drake Hotel • San Francisco, California

December 3-5, 1986



Sponsored by ACM SIGSMALL/PC

- pages 193 to 197 -

THESIS: The Hardware Environment for Small Intelligent Systems, for Engineering Applications

K.F. Wong, B.Sc., G.G. Coghill, B.Sc.,Ph.D. and J.M. Hannah, B.Sc., Ph.D.

Electrical Engineering Department, Edinburgh University,
The Kings' Buildings, Mayfield Road, Edinburgh, EH9 3JL,
Scotland.

ABSTRACT

System size, lack of standards and poor real time response have prevented the widespread acceptance of AI techniques in engineering applications. By tailoring the hardware configuration specifically for applications and by utilising an accepted bus standard, improved performance may be achieved. In this paper, the above approach is justified through the explanation of the principles of a hardware system, named The Hardware Environment for Small Intelligent Systems (THESIS) which is currently under development in Edinburgh University.

1. Introduction

Artificial Intelligence (AI) is a science which attempts to provide machines (artificial entities) with human-like behaviour such as the ability to store and acquire knowledge and make use of it to reason and to act on deductions as a human being would.

A study of how to best support AI applications programs in a real time engineering environment is being conducted in the Department of Electrical Engineering of Edinburgh University, Scotland [1]. Fields of interest include: signal processing, speech processing, engineering consultancy, monitoring, control and instrumentation.

It has been shown [2] that practical systems for engineering applications have to be small in order to be easily transportable from one site to another. Frequently, they are operated by non-technical personnel and therefore they should have user-friendly interfaces. The ability to respond instantaneously to external stimuli is often an important requirement for engineering systems. Commissioning considerations are also crucial as these can be more expensive than the system and its design cost taken together. For good design practice, factors such as ease of maintenance, serviceability and ease of future expansion, need to be taken into account as well.

At present, most small engineering systems are only semi-automated. Users still have to interact with them to ensure acceptable performance. Take for example the case of a spectrum analyser with data acquisition capability. Normally, engineers have to interact with the instrument, so that ambiguous data samples are rejected leaving only valid cases for consideration. With built-in intelligence, unacceptable signal patterns could be initially stored as background knowledge. Using inference techniques, the instrument can 'smartly' filter out unwanted signals and continue sampling with little, or even no, human intervention. Complete automation in such instruments has long been an engineering aspiration.

1.1. Section Summary

The factors prohibiting AI techniques from being considered as a practical proposition are identified in section two. A suggested remedy may be found through proper system configuration and the use of a standard system interface. This is described in the third section. Finally, in section four, conclusions are drawn with suggestions of other configuration possibilities.

2. Predicaments of Existing AI Technology

Presently, there are two major drawbacks of existing AI technology which have been prohibiting its wide acceptance in engineering applications. They are :

2.1. The bulk and lack of standards in AI systems hardware

AI software is generally too large and complicated for conventional programming languages e.g Pascal. Efforts have been made by researchers to produce special AI languages [3] with optimising compilers. In addition, compact data structuring techniques applied to advanced algorithms, have reduced program size and complexity.

Several small to medium scale systems are under research and development. Some of these have left the laboratory environment and have started to gain practical significance [3]. For example, Symbolic 3600, a commercial LISP machine which evolved from the CADR processor system first developed in Massachsetts Institute of Technology (MIT). These systems are meant to be design aids for AI software programmers, but they are unsuitable to be used for engineering applications.

2.2. Poor system response for real time applications

Computer intelligence implies *reasoning* with knowledge or applying rules to facts. During the process of reasoning, units of memory (cells) are dynamically being consumed and released. With a finite storage system, memory exhaustion is bound to occur unless released memory can be re-used. The process of memory reorganisation so as to reclaim re-usable cells is called *Garbage Collection*.

Garbage Collection is usually considered to be the responsibility of the host processor. Previously, to run such a process, the host would suspend all active jobs and dedicate itself solely to retrieving re-usable memory. This was acceptable in the research

and development system environments of the past. Nevertheless, the complexity of present AI software has rendered even this approach inadequate. If engineering applications had to stop at the middle of a continuous task the result could be costly or dangerous. Imagine an Intelligent System responsible for a manufacturing line halting during production to carry out a Garbage Collection cycle.

Currently, three practical solutions to improve real time AI performance are being investigated:

(1) software garbage collection algorithms with real time capability;

(2) special architecture AI machines with microcoded garbage collection schemes; and

(3) a dedicated hardware garbage collector in a multiprocessor environment.

The software approach is becoming obsolete; nevertheless, it is still viable if a quick and cheap solution demanding moderate response is required. Effectively, ideas two and three are hardware realisations of the software algorithms. The special purpose processor with its dedicated architecture is promising, but it would not be practical until some standardised architecture, or architecture with an internationally accepted interface standard has been achieved. Because of its cheap production cost and ease of expansion the multiprocessor approach is superior to the rest at present.

The multiprocessor solution consists of a coprocessing system working in parallel with the host, devoted mainly to Garbage Collection. In so doing, the problem of memory exhaustion is taken from the host. The result is a more dedicated host processing unit. As hardware costs are ever decreasing, this approach becomes increasingly attractive. Unfortunately, the existing systems are far from perfect. They are

- difficult to implement: usually crucial inter-processor communication protocols are required, e.g. dead-lock avoidance using semaphores. Consequently, debugging and maintenance problems prevail;
- lack of flexibility: existing systems have their their garbage collectors constructed around specific processors. Communication between them is via non-standard interface specifications. In so doing, system designers are restricted to one specific type of processor. Worst of all, random interfacing design methodology is error prone and laborious.

3. THESIS

The Hardware Environment for Small Intelligent System (THESIS) is a hardware system design tool. By down loading it with appropriate software, it can be customised to suit different engineering applications. Tolerable size, improved real time performance, ease of expansion and flexibility are its beneficial characteristics.

Figure one illustrates the anatomy of THESIS. Basically, it comprises five compulsory modules:

(1) the Host Processor (HP),

(2) Input/Output channels (I/0),

(3) the Inference Engine (IE),

(4) the Knowledge Base (KB), and

(5) the Cell Space (CS).

HP is the kernel of the system. It is the coordinator which controls flow of information between other parts and handles mutual communications. Internally, it comprises the processing unit and some primary memory (*heap*). For system expansion, secondary storage devices could be incorporated. I/O supplies communication paths with the external world. This could also be stimuli such as electrical signals from transducers. The IE encompasses rules and guidelines for heuristic deductions. The KB is a massive database containing facts. During reasoning, facts in the KB are scrutinised under the guidance of the IE. The role of the CS is to transport information to and from the KB according to the HP's instructions. Physically, CS is a writable memory store where cells are created.

3.1. Reasoning

Initially, the KB and the IE are pre-programmed with relevant information. When real world stimuli enters from the I/O, the HP reacts promptly and monitors these stimuli. Information frameworks are generated by applying the IE's rules to the KB. The received stimuli are compared against such frameworks. The results are fed back into the HP which will send a response to some intended recipient and create more tasks. Effectively, the decision of the HP is made by applying heuristical inference techniques on past 'experience'. In such a fashion, full system automation is established.

3.2. Adaption By Learning

In situations when a fatal error occurs, such as overvoltage in the case of a power system monitor, which could cause malfunctioning of the system, the HP would note such an error at the moment of break down. During bootstrapping, when the system resumes, the error pattern would be passed on to the KB; and from then on tagged as high priority. Moreover, the HP could then self-generate a set of precautionry procedures, according to existing 'experiences'. These procedures would be stored in the IE and would be invoked whenever the error recurs. Theoretically, the information stored in the KB and the IE would be ever increasing, thus providing the overall system with *adaptive* behaviour. Consequently, the concepts of *learning* and data acquisition, which are major AI functional issues are realised.

3.3. Separation of the CS from the Heap

At first sight THESIS does not seem to be very different from classical AI systems [3a]. The contrast lies in the separation of the CS from the primary heap memory. In classical AI systems, programs and data mingle together in the heap. In so doing, they exhibit *referential transparency*, one of the unique features of AI languages [3]. There is no difference between program and data. Both are classed as symbols or objects. Each object is constructed from one or more cells and has some properties associated with it. Properties can either be a collectiive list of other objects or a list comprising of some functional specifications. The actual constituents are only reviewed during run time. Thousands of objects form a program environment and new objects are created continuously whilst old ones are augmented during program execution.

Because AI programs are generally large and complex, with codes and data stored in one common area, system performance can easily be degraded. For instance, if the memory is mostly filled with program code, it would not take long before memory exhaustion occurred. Moreover, the frequent need for garbage collection would cause untolerable interruptions. In practice, programs are mostly permanent objects which makes garbage collection on them not worthwhile.

The reason for separating the CS from the heap is to facilitate the division of program and data. Programs are stored in the heap within the HP. Since program codes are instructions to direct the HP, this ensures direct access to instructions from the HP. Cache memory can be used to further increase the rate of instruction fetching. For a large software environment, secondary memories, with page scheduling, can be attached to the heap. Data is restricted to the CS only. The dynamic memory allocation process will create and release cells from this region.

Unfortunately, CS/heap separation has partially sacrificed the characteristic of referential transparency. However, this is not essential in an engineering environment. This contrasts with a development environment, where referential transparency is crucial because every object is dynamic and may be subject to changes. In engineering applications, programs are well defined

and mostly static. It is not worth including these codes in the CS because this would do nothing but increase congestion. When objects which become permanent during program execution, e.g. experiences which are learned, the CS can transfer them to other non-temporary memory modules (the IE and the KB).

In classical AI systems, the process of reasoning is implicit : it is regarded as part of the HP's responsibility. Users are virtually transparent to such processes until they are interrupted by the need for garbage collection to reclaim re-usable cells. Having separated segments for programs and data, provides more room for the HP to work on and the frequency of garbage collection is reduced. Also, the rate of garbage collection is speeded up due to the fact that the collector is dealing with cells only -- cells are generally temporary entities [4] and so they required less effort to recycle. To concentrate on the CS only, greatly reduces the HP's search paths and processor throughput is increased.

Currently, classical AI systems contain a limited amount of internal heap memory. Virtual memory techniques are often employed to accommodate large software systems. As a side effect of random dynamic cell allocation, non-contiguous cell distribution occurs. Inevitably, this would lead to frequent page swapping and memory fragmentation. It is an expensive practice to transfer information from secondary memory into the heap. To do it frequently could be disastrous, even with large mainframe systems as they could end up spending most of their time swapping pages.

This is a problem for the entire computer science community rather than simply for AI. To solve this problem, researchers have developed algorithms, such as compaction, cdr-coding, etc. The idea is to localise cell distribution and reduce thrashing. The penalties paid with these methods are the extra effort required for cell processing and prolonged garbage collections. However, since the overall performance has been improved, the drawbacks have to be accepted.

Program/Data partitioning in THESIS provides an alternative solution to the above problem, particularly suitable for small system designs. Since the CS comprises data cells only, there would be ample space for cell distribution. In most cases, the heap would be sufficient even without secondary memory support. This means that localisation techniques are not required any more. Without them process execution would resume after garbage collection with no extra effort required for compaction techniques etc.

In AI applications, cells are short-lived and a high percentage of them are re-usable [4]. The only trade-off is the frequency of garbage collection, but the rate of this process has already been increased since program codes which are permanent objects need not be collected. In addition, if a garbage collector which recycles useful cells and rejects 'garbage' [4] is used, fewer cells have to be collected.

4. The Prototype of THESIS

A prototype of THESIS is under construction using a VME multiprocessor bus system. VME is by far the most widely used industrial interface for system designs. By adopting this de facto standard, ease of maintenance, and design flexibility is accomplished. The overall system performance can be enhanced by interfacing it with dedicated coprocessor systems, e.g SUM : an AI coprocessor for unification [5].

A garbage collection coprocessor (GC) system is incorporated. The GC effectively sits between the HP and the CS which performs garbage collection in a incremental fashion. A modified *Hewitt's* garbage collection scheme [4] is adopted in the GC. Physically, the implementation is a direct microcoded translation of the algorithm. The microcodes control a *bit-slice* machine with a dedicated architecture, specially tailored to increase the rate of garbage collection. Architectural supports include dynamic type checking, cell structured storage, head/tail addressing and a flexible communication protocol with the host processor. The flexibility of bit-slice technology [8] has made it suitable for the design. Currently, a cell is 48 bits in size (24 for head and 24 for tail) with only two bits for type tagging.

The GC is responsible for the process of cell allocation. The HP will prompt it for a new cell and it will ensure that the CS is not exhausted before granting a cell. If the CS is running out of cells, garbage collection will be initiated†. As by products, other cell manipulation primitives are also included, e.g. HEAD, TAIL, etc. Therefore, this version of the GC can be considered as a cell processing accelerator.

Hewitt's algorithm is originally based on Baker's [6]. A Baker type garbage collector is preferable because it ensures a strict upper time limit on the process. In engineering, where system timing is a paramount design specification, unbounded process cycles would render timing calculations virtually impossible.

In Hewitt's algorithm, garbage collection is performed according to cells' lifetimes. Older cells are considered permanent. Virtually, they are archived and so seldom require to be collected. This maps exactly into the idea of CS/heap separation of THESIS. However, real systems which based on [4] (e.g. LM2 machine from LAMBDA) performs cell archiving during run time. In THESIS, program code are regarded as permanent cells thus archival is done at compile time. This is acceptable in an engineering environment where application programs are well defined.

The HP is an off-the-shelf CPU system with two serial I/O ports based on a Motorola 68000. In the preprogrammed section of the KB and the IE, Read Only Memory (ROM) devices are used, for simplification and to reduce the design cost. The remaining parts of the KB, the IE, and the CS are implemented with low power CMOS Random Access Memory (RAM) devices to reduce heat dissipation.

5. Conclusion

THESIS provides a flexible general purpose structure for designing engineering applications. It provides a solution for current AI shortcomings in engineering areas. This is achieved by using de facto industrial standards, e.g. VME bus, proper system configuration and the incorporation of a parallel garbage collector to take the problem of memory reclamation away from the host processor. This leads to increased processor throughput and improved real time response. Unfortunately, THESIS is still in its developement stage. Its idea has yet to be justified by applying it to practical applications.

THESIS is, initially, designed for small scale systems. By loading it with different applications software, THESIS could be programmed to cope with various real world situations. Nevertheless, the THESIS approach should not be limited to a specific system size. The same design philosophy could be applied to large multi-processor systems integrated within a local area network. Moving to the other extreme, miniature systems could be implemented using a standard chip interface scheme, such as programmable Switch Matrix technology [7], in a tightly-coupled fashion.

† Readers are suggested to refer to [4] and [6] for details.

References

1. Dept. of Electrical Engineering: project proposal, unpublished, internal document, Edinburgh University, Scotland.
2. Whittington,H.W., Coghill,G.G.: "Hand-Held Digital Sonic Pile Testing System", 1983, Ibid.
3. Barr,B., Feigenbaum,E.A.: "The Handbook of Artificial Intelligence", vol.2, chapter7, Pitman, 1983.

3a. Barr,B., Feigenbaum,E.A.: "The Handbook of Artificial Intelligence", vol.2, chapter6, Pitman, 1983.

4. Liberman,H., Hewitt,C.: "Garbage Collection Based on the Lifetimes of Objects", MIT AT Memo no.569, Cambridge, Massachetts, 1981. Communications of the ACM, 1981.

5. "SUM: an AI coprocessor", Byte, April 1985.

6. Baker,J: "List-processing in Real-time on a Serial Computer", Communications of ACM, vol.21, no.4, April 1978, pp280-294.

7. Chen,W. : "Programmable Switch Matrix", unpublished, 1st yr. Ph.D. report, Dept. of Electrical Engineering, Edinburgh University, Scotland.

8. Mick,J., Brick,J.: "Bit-Slice Microprocessor Design", McGraw-Hill, 1980.

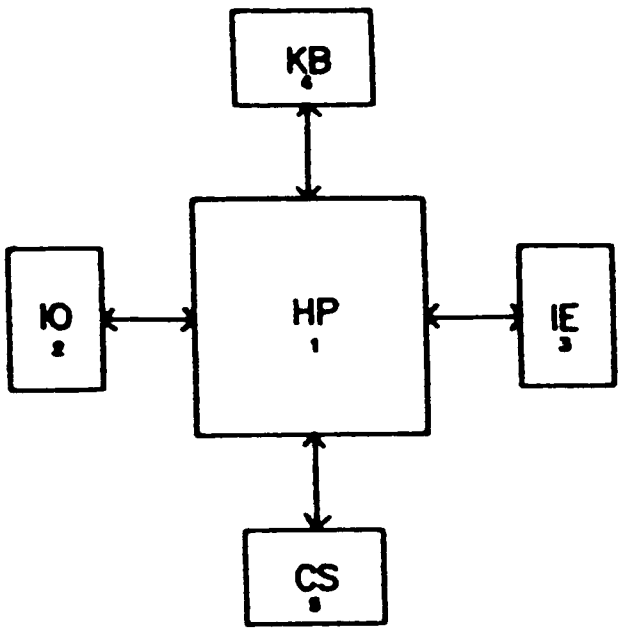


Figure 1 : The Anatomy of THESIS.

PROCEEDINGS

1986 ACM SIGSMALL/PC
Symposium on Small Systems

Sir Frances Drake Hotel • San Francisco, California

December 3-5, 1986

Sponsored by ACM SIGSMALL/PC

- pages 198 to 203 -

ICMS: An Intelligent Cell Memory System for Real-Time Engineering Applications

K.F.Wong, G.G.Coghill and J.M.Hannah

Department of Electrical Engineering,
Edinburgh University, The King's Buildings,
Mayfield Road, Edinburgh, EH9 3JL, Scotland.

Abstract

There is a growing interest in the application of Artificial Intelligence (AI) techniques in engineering. Existing AI systems are not suitable for many applications because of their unacceptable real-time response and complexity. An intelligent cell memory system (ICMS) is proposed which is capable of providing improved real-time performance. A prototype implementation of ICMS using dedicated bit-slice processors which perform parallel garbage collection with bounded separation time is described. The system uses an optimised implementation of Baker's algorithm to give high speed performance. It is easy-to-use, flexible, portable and suitable for engineering applications where fast response time is required.

1. Introduction

The potential of Artificial Intelligence (AI) is beginning to be realised in many practical fields. Potential application areas in engineering include fault diagnosis, process control, manufacturing, industrial automation, engineering consultancy and planning and design. Nevertheless, existing AI systems are not yet suitable for many engineering applications due to

- their unacceptable real time system performance; and
- the bulk and complexity of system hardware.

The distinct nature of AI techniques demands software with special features [1]. List processing capability is one of the fundamental requirements. The structure of lists naturally resembles the idea of objects and their associated properties. The elementary storage unit of lists is a cell which consists of a HEAD and a TAIL with additional bits for various purposes. HEADs and TAILs are either pointers to other cells or atoms, i.e. numbers or literal strings. From these basic cells an elaborate information frame-work can be constructed. All current AI languages are capable of list manipulation. The one most widely used is LISP [2]. On top of list processing capability, additional features are included in these languages to enhance their performance; for example, dynamic memory allocation. Normally, cells are located in a free cell space and they are detached from the space only when an extra piece of information requires storage. When a piece of information is no longer valid, its associated cell will be discarded. Continuous cell allocation would eventually exhaust the cell space, if the discarded cells were not recovered by *Garbage Collection*. In most systems, garbage collection is an expensive process. As the main list processing tasks are suspended while garbage collection is taking place, this results in long interruptions between continuous computation phases. Systems like this are unacceptable for many engineering applications where fast and predictable response is demanded.

The uniqueness of list structures leads to inefficient processing by conventional computers. Novel machines have been designed to improve processing efficiency. Special architectural features are incorporated in list processing machines to achieve maximum system throughput. Some machines are also capable of further expansion by adopting a standard interface, for example NuBus for LAMBDA2 (one of the most popular AI machines). However, the high cost of these machines (e.g. ~ US $50,000 for LAMBDA2) and their large size, make them only applicable to large scale engineering applications. There have also been attempts to convert list processing into silicon at chip level, such as the SCHEME project [3]. These chip systems offer their own instruction set but the inter-chip communication depends on complicated interface schemes which make them difficult to use.

1.1. Paper Outline

In this paper a real time garbage collection system suitable for inclusion in practical engineering designs is proposed. Following a brief overview of classical garbage collection schemes, the distribution of garbage collection processing time is considered. A practical system is then described which offers an attractive solution for applying intelligent processing to engineering situations. The system performance is shown to give a considerable improvement over conventional garbage collection systems. Finally, suggestions are made for possible enhancements of the system.

2. Garbage Collection Systems

Garbage collection systems are generally based on three classical methods [4]:

(I) reference counting

(II) marking and sweeping

(III, copying.

Systems based on (I) and (II) constantly maintain a special list of unused cells - the free-list. In (I) an extra word is added onto each cell to record the number of constituent cells which are pointing at the cell. When this count drops to zero, this implies that the cell is isolated and it is returned to the free-list. For (II) the extra storage required is less, only two or three additional bits per cell are necessary. When the free-list is

exhausted, garbage collection is invoked. The collection process is performed in two phases. In the first phase, useful lists are traversed and all accessible cells are tagged/marked. In the next phase, marked cells are identified and returned to the free-list. Prior to resuming normal list processing, all marked cells are unmarked, in preparation for the next collection cycle.

There is a slight different in the organisation of the cell space in (III). Initially the cell space is divided into two regions: OLD and NEW. Memory allocation always operates upon cells in the NEW region. Eventually, when there are no more cells available in NEW, collection is initiated. The two regions are now interchanged. Useful cells are then copied from the OLD into the NEW region until none are left in the former.

3. A Practical System for Time Critical Applications

In practice, using conventional computers, garbage collection can only be perform serially. Classical collectors are invoked when the cell memory is exhausted. This results in a timing pattern described in figure 1a. The actual time required for garbage collection depends on the size of the useful lists. With large systems, this activity could well take hours. The long interruption between list processing implies sluggish system response. This is unsuitable for real-time applications. Improvements have been made to minimise the idle time. One idea is to deceive the list processor that garbage collection is always finished. Effectively, both garbage collection and list processing are divided into small operational bursts (figure 1b). which are interleaved. This type of incremental collector is most suitable for interactive systems where collection can occur transparently in between commands. No matter how serial garbage collection is performed, processing efficiency can never be greater than 50%.

An alternative approach is to carry out garbage collection concurrently with list processing in a multiprocessor environment. Ideally, the system should be configured in such a way that garbage collection and list processing have no interaction - they should be mutually exclusive. In reality, this can never be completely achieved. Misalignments between them create idle intervals in continuous list processing (t_1, t_2 and t_3 in figure 1c). Parallel systems ensure that the list processor is doing useful work most of the time. Machine productivity is thus higher (>50%) than for serial collectors. However, existing implementations have two disadvantages which make them unattractive in engineering applications :

(i) Complexity - special architectures with complex interfacing are adopted

(ii) variable separation times - inconsistent separations cause unpredictable timing performance making engineering design difficult.

From the foregoing discussion, it can be seen that a practical garbage collection system should exhibit the best features of the incremental and the parallel schemes. These are :

- Maximum garbage collection and list processing overlap in order to increase the host productivity.
- Minimum separation times which should be bounded.
- simple system architecture - ideally, it should be transparent to the host processor. (In particular the communication link between the host and the collector should be straightforward.)

4. ICMS - The Intelligent Cell Memory System

A system which exhibits the desirable features described above is shown in block form in figure 2. It consists of an Intelligent Cell Memory Controller (ICMC) which acts as an interface between the cell memory (CM) and the host processor. The overall system (ICMS) has been designed to be suitable for engineering applications with time critical requirements.

Normally, the controller is constantly performing garbage collection over the cell memory in parallel with the list processing by the host. Effectively, the host processor is totally unaware of the existence of garbage collection. However, when the host requests a cell, a hardware interrupt is generated in the controller. The latter finishes off the current collection burst before servicing the requested operation. This approach provides overlap between garbage collection and list processing.

A suitable choice for garbage collection is Baker's algorithm [5] and this is adopted in the ICMS. Baker's collector is a copying type which is normally used in an incremental mode. The algorithm is simple to use and provides the desired bounded separation time. Special architectural features of the controller also reduce the garbage collection time thus a short separation is ensured.

The time distribution diagram of the ICMS is as shown in figure 1d. Separation times "t" are consistent and the overall machine time M (for the same task) is much shorter than the two serial cases. This means that the host is not responsible for garbage collection and more of its time contributes to useful list processing.

The ICMS host interface is based on a standard system bus and a memory mapped register structure which provide a simple link between the host and the controller. This interface scheme provides easy-to-use communication protocols which simplifies system design.

4.1. Design Features

Baker's algorithm was initially verified in software and preliminary studies of the generated code revealed some degree of hidden parallelism in the algorithm. It was realised that implementing these in hardware could provide a decrease in garbage collection time.

The definition of the cell storage structure is complicated but to access a cell many time consuming addressing instructions are required - over 50% of the generated code. A solution to the problem is to abandon the orthodox byte/word storage convention and treat a cell as an elemental storage entity. Addressing a cell then would be much more natural and require less time and effort. In addition, on average, up to two thirds of the cell address instructions are conditional. The cell to be addressed mostly depends on either the state of the collection process or the region to which the cell pointer belongs. Parallelism can be exploited by overlapping the testing and the cell addressing instructions. Since the testing instructions are simple, they can be realised using dedicated hardware, making them independent from cell addressing. Thus both instructions could be executed concurrently.

There are two pointers which are frequently used in Baker's algorithm, namely B and S. The B pointer always points to the next available cell. Functionally, whenever a cell is allocated, the B pointer is read and it is incremented afterwards. The job of the S pointer is to point at the cell which is next in turn to be collected. Similarly, its value is read during each collection burst and subsequently incremented. Some degree of speed up can be achieved by implementing the B and S pointers in hardware, incorporating an auto-increment capability.

In addition to Baker's algorithm six list processing primitives are catered for by the ICMS to give improved performance†. They are CONS, CAR, CDR, ATOM, REPLACA and REPLACD [2]. These primitives operate on one or two parameters and the function of the controller is to ensure that the parameters are correct and to garbage collect them if necessary.

† a through explanation of why special attention has to be paid to these primitives is given in [5].

4.2. Engineering Features

4.2.1. Simplicity

The ICMS is an independent functional module. From the host processor, it is seen as a passive storage device, accessible by writing or reading a bank of registers. Each register is responsible for a LISP primitive and is assigned with an address. Effectively, writing into a register is equivalent to passing a parameter and invoking the corresponding list operation at the same time. At the end of the function, return values are read from the same address.

4.2.2. Fast Response

Although the throughput of the host processor will be reduced by the frequent requirement for garbage collection bursts, the design of ICMS ensures that these take place at an acceptable speed by adopting several architectural features (see figure 3) :

i) Cell Unit-addressing - Two bit slice ALUs are employed, one is responsible for the manipulation of the HEAD pointer and the other is dedicated to the TAIL. The operational speed for list functions on the cell is therefore enhanced. Moreover, this architecture does not exclude the possibility of either only one ALU or one address/data being required. In such circumstances, the unnecessary device may be made redundant by disabling it using the appropriate micro control bits.

ii) Hardware Testing - Five tests are performed in hardware concurrently with the bit slice ALUs, namely

- *CELL(X)* is X a cell pointer;
- *OLD(X)* does cell ponter X exist in the OLD region;
- *NEW(X)* does cell pointer X exist in the NEW region;
- *GC* has garbage collection finished ? and
- *CMEND* has the cell memory been exhausted ?

iii) Content Prefetching - This idea derives from "instruction prefetch" in computer architecture design. Running in parallel with hardware testing, it ensures the correct address at the memory address port at the end of a conditional fetch instruction. Thus test and fetch instructions are executed in one machine cycle which otherwise would be two or more.

iv) Auto-incrementable Pointers - Special cell pointers B and S are constructed in hardware using flip-flop counters. They are incremented after being used for cell addressing.

4.2.3. Flexible and Portable

The ICMS interface uses a popular industrial bus standard, namely IEEE P1014 [6] or the VMEbus. Therefore, unlike systems such as [7,8] the ICMS is not restricted to a particular type of processor. Theoretically, because the bus standard has been conformed to, it should be compatible with any host system.

5. System Performance

The ICMS prototype system is embedded in a custom LISP environment, known as simpleLISP. This is a simplified version of pure LISP, but it does include all the basic LISP primitives. The most time consuming primitive implemented is CONS(X,Y) which demands a new cell, checks the regions in which X and Y are located, performs collections on X and/or Y when necessary and finally places X and Y into the HEAD and TAIL of the new cell, respectively. Normally, there is no problem for cell allocation until no free cells remain in the NEW region. At this point, garbage collection is initiated. This includes the collecting of ROOT registers and interchanging NEW/OLD regions which further extends the interruption to list processing. An analysis shows that the worst case of CONS is the most time consuming operation in the overall system. For the M68000 CPU with a 8 MHz clock, the calculated time is 1.8 ms. Writing the algorithm in micro-code and running the operation on the ICMS gives a worst case CONS time of 8.3 μs.

Compared with the software implementation in the host, the ICMS is obviously much superior giving a greater than two orders of magnitude improvement in the worst case performance. Similar comparisons for other LISP primitives have been carried out and their results are shown in table 1. These results give a good indication of the improvement in overall system performance because application programs are mainly constructed from these primitives. In practice, the actual overall improvement factor will vary somewhat depending on the application.

6. Conclusions

The ICMS is a hardware system which has been built for garbage collection for use in engineering applications. Its functional characteristics are a mixture of the incremental and parallel schemes. It is simple to use with a bounded response time, offers flexible expansion and portable. Although the primary objective of the ICMS is to improve garbage collection performance, it also caters for five list primitives : CAR, CDR, REPLACA, REPLACD and CONS. In practice, the ICMS may be regarded as a coprocessor system responsible for accelerating list manipulation.

The size of the cell memory is expandable with extension cell memory boards being added via the local bus. Also for designs which are limited by the available physical memory virtual memory techniques could be adopted by connecting the required management unit between the controller and the cell memory. However, the size of the cell memory required in many engineering applications is usually reasonably small, therefore, the virtual memory approach would rarely be employed.

Another possible enhancement to the current system is the concept of collection according to "objects' life-times" [9,10]. Once a cell had recognised to have persisted for a number of garbage collection cycles, it would be considered permanent. All permanent cells would then be transported out of the cell memory into some other memory area for archiving. This

LISP Primitives	The worst case no. of GC bursts	68K machine times (x μsec)	ICMS machine times (y μsec)	Improvement Factors (x/y)
CAR, CDR	1	165.8	6.2	31.9
REPLACA, REPLACD	2	453.8	7.5	60.5
CONS	11	1815.0	8.3	220

table 1: Improvement in Performance for Various LISP Primitives.

would greatly reduced the average time for garbage collection and ensure a fairly constant workspace.

The design of ICMS supports the philosophy of hardware modularity. In any VMEbus environment, a conventional processor, not designed for AI applications, could be transformed into a reasonablely powerful AI machine just simply by "plugging" in the ICMS coprocessor module. Moreover, the system is capable of integration on silicon to produce a low cost design thus expanding the range of engineering applications for AI techniques.

References

1. A. Barr and A.E. Feigenbaum, *The Handbook of Artificial Intelligence Vol. II*, William Kaufman, Menlo Park, California (1982).
2. J. McCarthy and et al., *LISP 1.5 Programmer's Manual*, MIT Press, Cambridge, Mass., USA (1962).
3. G.J. Sussman, J. Holloway, G.L. Steele, and A. Bell, Scheme79 - Lisp on a chip," *IEEE Computer* 14 pp. 10-21 (July 1981).
4. J. Cohen, "Garbage Collection of Linked Data Structures," *Computing Surveys* 13(3) pp. 341-367 The Association of Computing Machinery, (Sept. 1981).
5. H.G. Baker, "List Processing in Real Time on a Serial Computer," *Communication of the ACM* 21(4) pp. 280-293 (April, 1978).
6. W. Fischer, "IEEE P1014 - A Standard fot the High-Performance VME Bus," *Micro* 5(2) pp. 31-41 IEEE, (Feb., 1985).
7. G.L. Steele, "Mutiprocessing Compactifying Garbage Collection," *Communications of ACM* 18(9)(Sept., 1975).
8. E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten, and E.F.M. Steffens, "On-the-Fly Garbage Collectio: An exercise in Cooperation," *Communications of ACM* 21(11) pp. 966-974 (Nov., 1978).
9. H. Liberman and C. Hewitt, "A Real Time Garbage Collector Based on the Lifetimes of Objects," *Communication of the ACM* 26(6) pp. 419-429 (June, 1983).
10. D. Ungar, "Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm," *Proceedings of Software Engineering Symposiums on Practical Software Development Environments*, pp. 157-167 ACM, (1984).

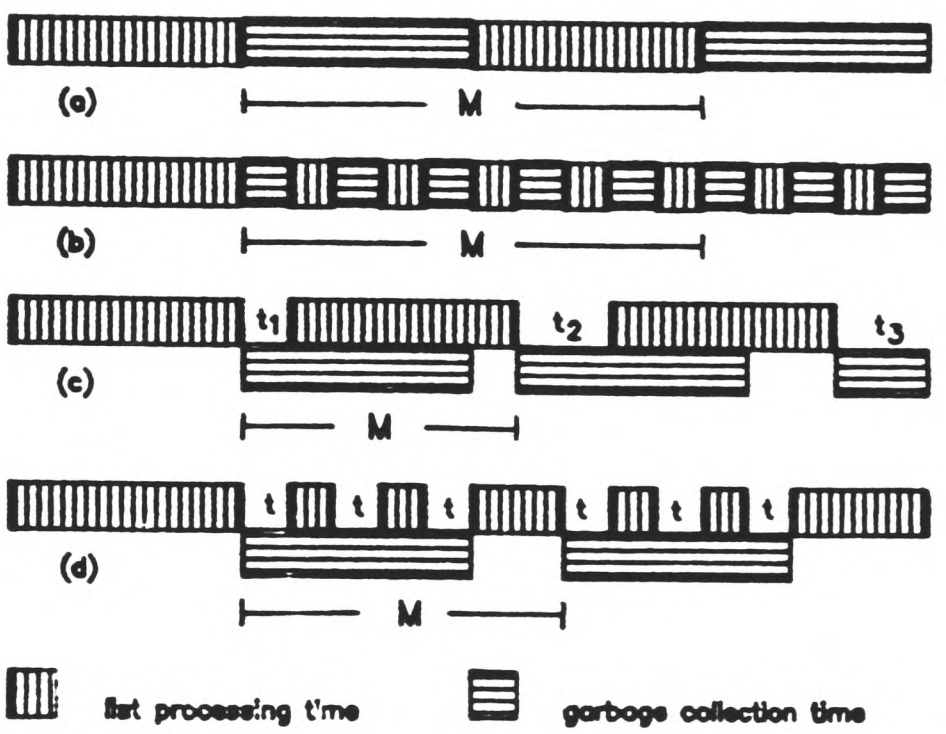


figure 1 : Time Distribution Diagrams of Garbage Collection Systems,
a) classical;
b) incremental;
c) parallel; and
d) ICMS scheme.

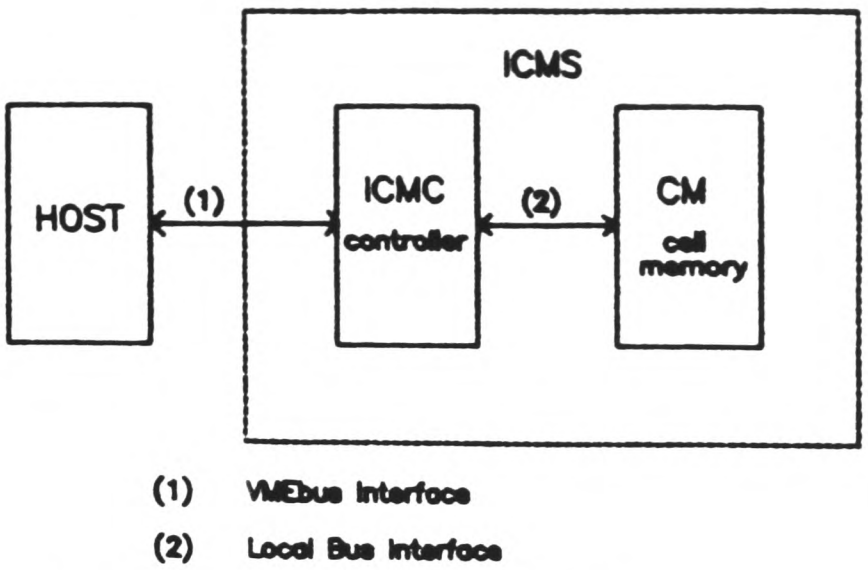


figure 2 : The Global Block Diagram of the ICMS;

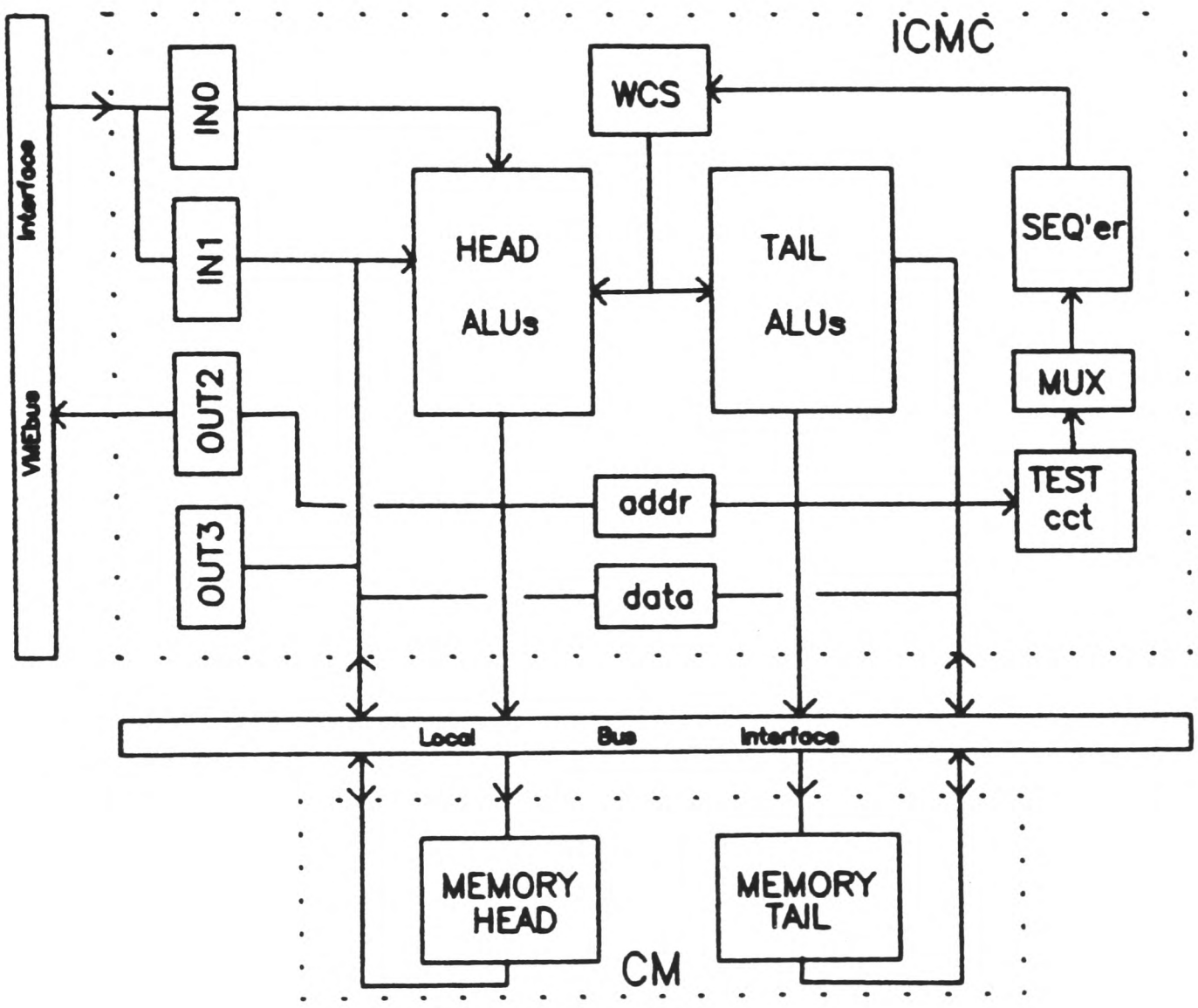


figure 3 : The Detailed Schematic Block Diagram of the ICMS Architecture.

Bibliography

1. R. Moore, "AI Must Cater to Nonexperts," *Computer Design*, p. 68, 15 Feb., 1986.

2. D.R. Brown, "R & D Plan for Army Applications of AI/Robotics," *Stanford Research Institute Project 3736*, SRI International, Stanford, USA, 1982.

3. J. Alty, "The Current States and Future Possibilities of Expert Systems," *IEE Symposium on "Current Trends in the Application of Expert Systems"*, Strathclyde University, Glasgow, Scotland, April 1986.

4. P. Winston, *Artificial Intelligence,* Addison-Wesley, Reading, Massachusetts, USA, 1977.

5. W.B. Gevarter, *Artificial Intelligence, Expert systems, Computer Vision and Natural Language Processing,* Noyes Publication, N.J., 1984.

6. N.J. Nilsson, *Principles of Artificial Intelligence,* Tioga Publishing Company, Palo Alto, California, 1982.

7. A. Barr and A.E. Feigenbaum, *The Handbook of Artificial Intelligence Vol. II,* William Kaufman, Menlo Park, Carlifornia, 1982.

8. Institution of Electrical Engineers, *IEE Symposium on "Current Trends in the Application of Expert Systems",* whole issue, Strathclyde University, Glasgow, Scotland, April, 1986.

9. T. Moto-oka, *Fifth Generation Computer Systems,* North-Holland, Amsterdam, 1982.

10. A. Burns, "Information Technology - for Better or Worse," in *New Information Technology*, ed. A. Burns, pp. 187-215, Ellis Horwood, England, 1984.

11. R.W. Milne and B. Chandrasekaran, "Fault Diagnosis and Expert Systems," *IEE Symposium on "Current Trends in the Application of Expert Systems"*, Strathclyde University, Glasgow, Scotland, April 1986.

12. M. Merry, "APEX3: An Expert System Shell for Fault Diagnosis," *GEC Research Journal*, vol. 1, no. 1, pp. 39-47, 1983.

13. N. Swindell and R.J. Swindell, "System for Engineering Materials Selection," *J. of Metals and Materials*, pp. 301-303, May 1985.

14. M.S. Fox and S.E. Smith, "ISIS - A Knowledge Based System for Factory Scheduling. Expert Systems," *International J. of Knowledge Engineering*, vol. 1, no. 1, 1984.

15. T. Crawford, *IEE Symposium on "Current Trends in the Application of Expert Systems"*, Strathclyde University, Glasgow, Scotland, April, 1986.

16. J.C. Francis and R.R. Leitch, "Knowledge-Based Process Control," *Proceedings of IEE International Conference on Control'85* , pp. 483-488, 1985.

17. G. Hetherington, "Expert Systems in VLSI Design," *IEE Symposium on "Current Trends in the Application of Expert Systems"*, Strathclyde University, Glasgow, Scotland, April, 1986.

18. P.W. Horstmann, "Expert Systems and Logic Programming for CAD," *VLSI Design*, pp. 37-46, Nov., 1983.

19. F. Hayes-Roth, "The Knowledge-Based Expert System : A Tutorial," *IEEE Computer*, pp. 11-28, Sept., 1984.

20. E. Rich, *Artificial Intelligence,* MacGraw-Hill , 1983.

21. D.G. Bobrow and B. Raphael, "New Programming Languages for Artificial Intelligence Research," *Computing Surveys*, vol. 6, no. 3, pp. 153-174, Sept., 1974.

22. L.D. Erman, F. Hayes-Roth, V.R. Lesser, and D.R. Reddy, "Hearsay-II Speech -Understanding System: Integrating Knowledge to Resolve Uncertainty," *Computing Surveys*, vol. 12, no. 2, pp. 213-253, Feb., 1980.

23. D.S. Nau, "Special Feature: Expert Computer Systems," *IEEE Computer*, pp. 63-85, February, 1983.

24. R.A. Corlett, "Features of Artificial Intelligence Languages and Their Environment," *IEE Software Engineering Journal*, pp. 159-164, July, 1986.

25. J. McCarthy and et al., *LISP 1.5 Programmer's Manual,* MIT Press, Cambridge, Mass., USA, 1962.

26. J.K. Foderaro and K.E. Sklower, "The Franz LISP Manual," in *4.2 UNIX on line manual*, Berkeley University, California, USA, 1980.

27. W.F. Clocksin and C.S. Mellish, *Programming in Prolog,* Springer-Verlag, 1984. (2nd ed.)

28. B. Ivan, *Programming for Artificial Intelligence,* International Computer Series, Wokingham Addison-Wesly, 1986.

29. D. Warren, D. Bowen, and L. Pereira, "C-PROLOG Manual Version 1.4.edai," in *Unix On-Line Manual at uk.ac.ed.edisg*, ed. F. Pereira, 1985.

30. F. Hayes-Roth, "Knowledge-Based Expert Systems," *IEEE Computer*, pp. 263-273, Oct., 1984.

31. W.E.Jr. Suydam, "AI Becomes the Soul of the New Machines," *Computer Design*, pp. 55-70, 15 Feb., 1986.

32. M.F. Deering, "Architectures for AI," *Byte*, pp. 193-206, April, 1985.

33. D.A. Patterson, "Reduced Instruction Set computers," *Communications of the ACM*, vol. 28, no. 1, pp. 8-21, Jan., 1985.

34. G.J. Sussman, J. Holloway, G.L. Steele, and A. Bell, "Scheme79 - Lisp on a

chip,'' *IEEE Computer*, vol. 14, pp. 10-21, July 1981.

35. J. Batali, E. Goodhue, C. Hanson, H. Shrobe, and G.J. Sussman, ''The SCHEME-81 Architecture - System and Chip,'' *Proceedings of Conference on Advanced Research in VLSI*, pp. 69-77, MIT Press, Cambridge, Massachusetts, USA, Jan., 1982.

36. R. Blau, P. Foley, D. Patterson, D. Simples, and D. Ungar, ''Architecture of SOAR: Smalltalk on a RISC,'' *Proceedings of the 11th symposiums on Computer Architecture*, pp. 188-197, ACM/IEEE, Ann Arbor, Michigan, June, 1984.

37. D. Ungar, ''Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm,'' *Proceedings of Software Engineering Symposiums on Practical Software Development Environments*, pp. 157-167, ACM, Pittsburgh, USA, 1984.

38. J.R. Lineback, ''Lisp Processor Chips Point to Desktop AI,'' *Electronics Week*, p. 17, 31, March, 1986.

39. Texas Instruments, *Explorer Technical Summary,* Texas Instruments Inc., Austin, Texas, USA, 1984.

40. ''Brighter Future For DEC's AI Vaxstation,'' *Expert Systems User*, vol. 2, no. 6, p. 7, Sept., 1986. (advertisement)

41. A. Bawden, R. Greenblatt, R. Hoolloway, J. Knight, D. Moon, and D. Weinreb, ''Lisp Machine Progress Report,'' *AI Lab. Memo 444*, MIT, Cambridge, Massachusetts, USA, Aug., 1977.

42. D.W. Clark and C.C. Green, ''An Empirical Study of List Structure in Lisp,'' *Communications of the ACM*, vol. 20, no. 2, pp. 78-87, Feb, 1977.

43. D. Bobrow and D. Clark, ''Compact Encoding of List Structures,'' in *Technical Report CSL-79-7*, Xerox Palo Alto Research Center, June, 1979.

44. Symbolics Inc., *Symbolics 3600 Technical Summary*, California, USA, 1983.

45. Symbolics Inc., "Low-Cost Work Station Aims to Move AI Out of the Lab," *Electronics Week*, pp. 55-56, 21, April, 1986.

46. M. Amundsen, B. Kastner, S. Krueger, G. Manuel, G. Mathews, R. Prentice, and M. Watson, "Compact Lisp Machine," *TI Engineering Journal*, vol. 3, no. 1, pp. 116-121, Texas, USA, Jan.-Feb., 1986.

47. Unknown, "Technology Newsletter : Bell Labs Develops Fuzzy-Logic Chip," *Electronics Week*, p. 11, 9, December, 1985.

48. C. Arthur, "Alvey Effort Gives Rise to Memory Chip," *Computer Weekly*, 1986.

49. S.R. Vegdahl, "A Survey of Proposed Architectures for Execution of Functional Languages," *Transactions on Computers*, vol. C33, no. 12, pp. 1050-1071, IEEE, Dec., 1984.

50. P.C. Treleaven, D.A. Brownbridge, and R.P. Hopkins, "Data-Driven and Demand-Driven Computer Architecture," *Computing Surveys*, vol. 14, no. 1, March, 1982.

51. P. Walker, "The Transputer," *Byte*, pp. 219-235, May, 1985.

52. J. Darlington and M. Reeve, "ALICE : A Multiple-Processor Reduction Machine For the Parallel Evaluation of Applicative Languages," *Proceedings of the 1981 conference on Functional Programming Languages and Computer Architecture*, pp. 65-75, Portsmouth, New Hampshire, 18-22, Oct., 1981.

53. K. Smith, "Probing The News : Britain makes Major Bid to Build commercial Fifth Generation Machine," *Electronics Week*, pp. 26-27, 8 July, 1985.

54. G.G. Coghill, "PLEIADES : A Multi-Microprocessor system for Knowledge Manipulation," *PhD thesis*, University of Kent of Canterbury, Kent, England,

1980.

55. R. Collett, "Dataflow Computers Encroach On von Neumann Territory," *Digital design*, pp. 25.90-25.94, Morgan-Grampian Publishing Co., Dec., 1984.

56. D.E. Knuth, *The Art of Computer Programming Vol. I: Fundamental Algorithms,* Addison-Wesley, Reading, Mass., 1973.

57. T.A. Standish, *Data Structure Techniques,* Addison-Wesley, Reading, Massachusetts, USA, 1980.

58. J. Cohen, "Garbage Collection of Linked Data Structures," *Computing Surveys*, vol. 13, no. 3, pp. 341-367, The Association of Computing Machinery, Sept. 1981.

59. P.H. Winston and B.K.P. Horn, *LISP,* Addison-Wesley, Reading, Massachusetts, USA, 1981.

60. H. Schorr and W. Waite, "An Efficient Machine Independent Procedure for Garbage Collection in Various List Structures," *Communication of the ACM*, vol. 10, no. 8, pp. 501-506, Aug., 1967.

61. E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten, and E.F.M. Steffens, "On-the-Fly Garbage Collection: An exercise in Cooperation," *Communications of ACM*, vol. 21, no. 11, pp. 966-974, Nov., 1978.

62. H.T. Kung and S.W. Song, *An Efficient Garbage Collection System and its Correctness Proof*, Department of Computer Science, CMU, Pittsburgh, 1977.

63. H.G. Baker, "List Processing in Real Time on a Serial Computer," *Communication of the ACM*, vol. 21, no. 4, pp. 280-293, April, 1978.

64. H. Lieberman and C. Hewitt, "A Real Time Garbage Collector Based on the Lifetimes of Objects," *Communication of the ACM*, vol. 26, no. 6, pp. 419-429, June, 1983.

65. R.R. Fenichel and J.C. Yochelson, "A LISP Garbage Collection For Virtual Memory Computer Systems," *Communication of the ACM*, vol. 12, no. 11, pp. 611-612, Nov., 1969.

66. C.J. Cheney, "A Non-Recursive List Compacting Algorithm ," *Communication of the ACM*, vol. 13, no. 11, pp. 677-678, Nov., 1970.

67. S. Arnborg, "Storage Administration in a Virtual Memory SIMULA system," *BIT*, no. 12, pp. 125-141, 1972.

68. T. Hichey and J. Cohen, "Performance Analysis of On-the-Fly Garbage collection," *Communication of the ACM*, vol. 27, no. 11, pp. 1143-1154, Nov., 1984.

69. L. Lamport, "Garbage Collection with Multiple Processes : An Exercise in Parallelism," *Proceedings of the IEEE Conference in Parallel Processing*, pp. 50-54, Aug., 1976.

70. G.L. Steele, "Multiprocessing Compactifying Garbage Collection," *Communications of ACM*, vol. 18, no. 9, Sept., 1975.

71. K.F. Wong, G.G. Coghill , and J.M. Hannah, "ICMS: Intelligent Cell Memory System," *Proceedings of the 86 Conference on Personal & Small Computers* , pp. 168-172, ACM SIGSMALL/PC, San Francisco, U.S.A., 2-5 Dec., 1986.

72. W. Fischer, "IEEE P1014 - A Standard for the High-Performance VME Bus," *Micro*, vol. 5, no. 2, pp. 31-41, IEEE, Feb., 1985.

73. K.F. Wong, G.G. Coghill , and J.M. Hannah, "A Specialised Microcomputer System for the Application of Artificial Intelligence Techniques to Engineering," *10th Annual Workshop on Microcomputer Applications*, Strathclyde University, Glasgow, Scotland, 8-10 Sept., 1986.

74. K.F. Wong, G.G. Coghill , and J.M. Hannah, "THESIS : The Hardware Environment for Small Intelligent Systems," *Proceedings of the 86 Conference on Personal & Small Computers* , pp. 173-176, ACM SIGSMALL/PC,

San Francisco, U.S.A., 2-5 Dec., 1986.

75. J. Mick and J. Brick, *Bit-Slice Microprocessor Design,* McGraw-Hill, 1980.

76. K.F. Wong, "The Feasibility of Expert Systems in Engineering Applications," in *Internal Document*, Department of Electrical Engineering, Edinburgh University, Scotland, March, 1984.

77. Microsystems Co., *VMEbus Specification Manual Revision C,* Motorola Semiconductor Product Inc., Arizona, U.S.A., Feb., 1985.

78. D.A. Patterson, "STRUM: Structured Programming System for Correct Firmware," *IEEE Transactions on Computer*, vol. C-25, no. 10, pp. 974-985, Oct.,1976.

79. S. Dasgupta, "Some Aspects Of High-Level Microprogramming," *ACM Computer Surveys*, vol. 12, no. 3, pp. 295-323, Sept., 1980.

80. M.V. Powers and J.H. Hernandez, "Microprogram Assemblers for Bit-Sliced Microprocessors," *IEEE Computer*, vol. 11, no. 7, pp. 108-120, July, 1978.

81. P.W. Kernighan and P.J. Plauger, *Software Tools in Pascal,* Addison-Wesley, Reading, Massachusetts , 1981.

82. "YACC - Yet Another Compiler Compiler," in *UNIX Programmer's Manual*, vol. 2B.

83. I. Hansen, "Computer Structures 1983-84 : Microprogram Exercise," in *Third Year Computer Science Lecture Notes*, Department of Computer Science, Edinburgh University, 1983.

84. W. Chen, "Programmable Switch Matrix," *Ph.D. project*, Dept. of Electrical Engineering, Edindurgh University, Jan., 1984. (to be finished in Dec., 1986)

85. M.E. Lesk and E. Scmidt, *Lex - A Lexical Analyzer Generator,* Bell Laboratory, Murray Hill, NJ 07974.

86. S.C. Johnson, *Yacc: Yet Another Compiler-Compiler,* Bell Laboratory, Murray Hill, NJ 07974.

87. W. Kernighan and D.M. Ritchie, *The M4 Macro Processor,* Bell Laboratory, Murray Hill, NJ 07974, 1979.

88. P. Henderson, *Functional Programming: Application and Implementation,* Prentice-Hall internation series in computer science, 1980.

89. Hewlett Parkard, ''Graphics/9000: Device Independent Graphics Library,'' in *HP9000 Reference Manual*, U.S.A., 1983.