A Hardware Implementation of a Knowledge Manipulation System for Real Time Engineering Applications

by

Stephen Hudson B.Sc. (Hons)

Doctor of Philosophy

University of Edinburgh

March, 1990

Table Of Contents

Acknowledgement

There are a great number of people to thank for their invaluable advice, assistance and continuous support. Dr. John Hannah, despite other responsibilities and a heavy work load, was always able to offer encouragement, constructive criticism and reminders of impending deadlines, when necessary! Dr. Keith Manning provided a great amount of guidance with the C language and UNIX environment, while I was still finding my feet.

I would also like to thank Paul Chung and Robert Rae, of the Artificial Intelligence Applications Institute, for helping me wade through the very plentiful AI literature. Finally, I must extend my gratitude to my colleagues for their constant support, including Henry Bruce for his time spent proof-reading, Shaun Lytollis for the graphics package and especially Robbie Hannah for dragging me out of the doldrums every now and again.

Declaration

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

The Author:

Stephen Hudson B.Sc. (Hons)

Abstract

Although recent years have seen the development of, and increased interest in, Intelligent Systems, particularly Expert Systems, their poor real time response makes them unsuitable for many engineering applications. The design and implementation of special purpose hardware support, based on a structured knowledge representation, and capable of enabling real time relational accesses of a Knowledge Base, is described in this thesis. The Structured Knowledge Manipulation System (SKMS) employs parallel and heuristic techniques to insert, delete, modify or retrieve specified, or partially specified, relations from a Knowledge Base. Relational algebraic operations and between-bounds matching are supported directly by the SKMS. Moreover, a concurrent memory allocation and reclamation algorithm is implemented by the hardware with little speed overhead and no memory overheads. The results of performance evaluation experiments and suggestions for future developments, based on the architecture developed herein, are presented.

Abbreviations

AI	Artificial Intelligence
APS	Associative Predicate Store
ART	Automated Reasoning Tool
CCS	Condition Code Selector
CISC	Complex Instruction Set Computers
CPU	Central Processing Unit
CSR	Control and Status Register
DRAM	Dynamic Random Access Memory
FOPC	First Order Predicate Calculus
FRP	Ferranti Relational Processor
GAM	Generic Associative Memory
GRIP	Graph Reduction In Parallel
IC	Integrated Circuit
IFS	Intelligent File Store
IKBS	Intelligent Knowledge Based System
IT	Information Technology
KB	Knowledge Base
KEE	Knowledge Engineering Environment
KME	Knowledge Manipulation Engine
LOOPS	LISP Object Oriented Programming System
LTC	Lexical Token Converter
MIMD	Multiple Instruction Multiple Data
MIPS	Million Instructions Per Second
MMI	Man Machine Interface
MMU	Memory Management Unit
MPS	Microprogram Store

PC	Program Counter
PCC	Parallel Comparator Circuit
PE	Processing Element
PSU	Power Supply Unit
QBRM	Qualified Binary Relationship Model
RAM	Random Access Memory
RAP	Relational Algebraic Processor
RF	Register File
RISC	Reduced Instruction Set Computer
RPU	Relational Processing Unit
SKMS	Structured Knowledge Manipulation System
SRAM	Static Random Access Memory
TMS	Truth Maintenance System
US	User Stack
VLSI	Very Large Scale Integration
WSI	Wafer Scale Integration
wff	well formed formula

List Of Figures

List Of Photographs

List Of Tables

CHAPTER 1

Introduction

1.1. Background

Despite the increase in use and popularity of expert systems and AI planning and simulation systems, they remain unsuitable for many engineering applications due to poor real time response. A major limiting factor is the rate at which information in a knowledge base can be manipulated. Much knowledge systems research has been concerned with faster manipulation methods, and a variety of techniques have been developed to do this.

This thesis describes the research, design, implementation and evaluation of special purpose hardware support for a Structured Knowledge Manipulation System (SKMS) for real-time engineering applications. The SKMS has been designed to manipulate information using a knowledge representation formalism, developed specifically for this purpose and described in the thesis. The system is intended as a low-cost, plug-in enhancement to a SUN workstation via a VMEbus, or to an IBM-type Personal Computer via a PCbus. Computer simulation (using the C programming language in a UNIX† operating environment) of a basic expert shell (c.f. "Knowledge Craft") was performed to investigate the suitability of the proposed knowledge formalism with regards to:

- flexibility
- ease of manipulation by *knowledge operators*

† UNIX is a trademark of Bell Laboratories.

- support for memory allocation and reclamation (garbage collection)
- knowledge retrieval speed

UNIX profiling operations were carried out to pinpoint major performance limitations. The hardware was then designed to alleviate the problems encountered in the simulation, and to exploit the strengths. Additionally, memory allocation and reclamation within the knowledge base was implemented by a free-list garbage collection algorithm, incorporated in the design.

1.2. Chapter Summary

A prerequisite to the design of a knowledge based system is the appreciation and understanding of what techniques are used within such systems, and to this end, the first section of Chapter 2 serves as an introduction to some basic AI techniques. Since the primary consideration of all intelligent system design is a suitable knowledge representation formalism, the next section of Chapter 2 describes the two major formalisms in general use. The remaining sections provide an overview of the current state of AI software and hardware research. The major limiting factor in hardware based systems, the inadequacy of Von Neumann computer architectures, is also discussed; which leads on to the attempts to develop suitable hardware based systems (both enhanced Von Neumann systems and those based on novel architectures). Finally, a summary of the current level of hardware research and proposals for future projects† is presented.

Chapter 3 describes the SKMS conceptual design; the development of the knowledge representation formalism and its relation to the manipulation operators and garbage collection, which leads to the definition of a functional specification for the system. The software simulation of a basic expert shell is also described, with

† Special Interest Group in Knowledge Manipulation Engines proposals to Alvey Directorate

particular attention to hardware design considerations and performance limitations.

The design and construction of the hardware is discussed in Chapter 4. The system is divided into its separate functional components, and their relation and interfaces to one another are described in detail. Appendices B and C provide an index of all the signals defined within the system and the microprogram control word. Several levels of software are required for the entire system, and are discussed briefly at the end of the chapter. A more detailed description of the software (the language and assemblers) can be found in Appendices D, E, F and G.

Chapter 5 describes the projected and actual performance evaluation of the SKMS, with comparisons between the software simulation running on a variety of systems, and with other hardware implemented knowledge based systems. Problem areas within the SKMS design are highlighted, and opportunities for improvement and their projected performance improvements are presented.

Chapter 6 summarises the salient points introduced and developed in this thesis. Conclusions relating to the performance evaluation and possible system improvements are also discussed. Future developments of the SKMS are investigated, and a basic design of a Parallel Relational Processor System, based on fabricated Relational Processing Units, is proposed.

CHAPTER 2

Intelligent Systems

2.1. Introduction

What is Artificial Intelligence (AI)? There are various opinions and definitions of the meanings of *Artificial* and of *Intelligence* and an excellent review of AI can be found in references [1,2,3,4]. However, the following two definitions sum up AI pretty well.

> "*Artificial Intelligence is the study of how to make computers do things at which, at the moment, people are better.*" from Rich [1].

> "*...many human mental activities...are said to demand 'intelligence'...several computer systems can perform tasks such as these...we might say that such systems possess some degree of 'artificial intelligence'...*" from Nilsson [3].

Alan Turing [5] proposed what is now known as the *Turing Test* to determine whether a machine could think. The test is conducted using 2 people and the machine under test. One person remains in one room while the second person and the machine are in a separate room. The first person then asks questions of the other person or machine. The role of the machine is to act like a person and if the interrogator is unable to determine who has replied (machine or person), then the machine has passed the test and is said to be able to think. Some people believe that no machine will ever pass the Turing Test.

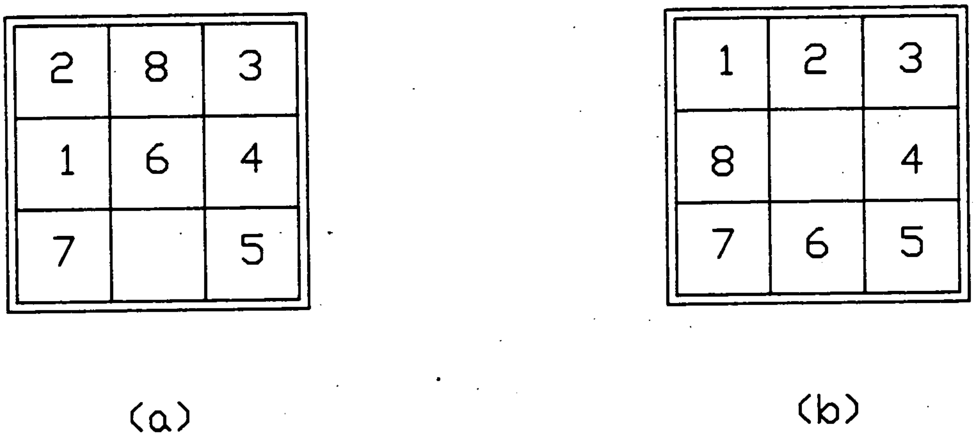
Initially, study into machine intelligence was generally confined to game playing and theorem proving software. The game of checkers (draughts) was used as an example of machine learning [6]; while playing, the program remembered moves which it could use later to improve its game. The Logic Theory Machine [7] was used to prove mathematical theorems, and the General Problem Solver [8] was developed to tackle simple tasks in reasoning such as scheduling meetings.

Systems of this type were, for a while, the dominant area of study in AI; the problems are structured, they do not require large amounts of data to work from (databases), and success or failure is easily measured – we either win the game or prove the theorem, or we don't. Unfortunately, many tasks, such as medical diagnosis [10], chemical analysis [9] and engineering design [11,13], are not so well structured. They require access to a great deal of knowledge and success often involves finding a satisfactory solution to a problem rather than the best. Since we generally associate such work with experts, then those programs which have been developed to tackle these type of problems are referred to as *expert systems*.

Expert systems are becoming the most widely used application of artificial intelligence. Several such systems have been applied in engineering, namely: monitoring and diagnosis [14], consultancy [11], modelling [12], and VLSI design [13]. Although applications might differ, the same AI principles generally apply to expert systems.

All AI applications involve problem solving; whether it is the solution required to win a game, the best design for a particular electronic circuit, or working out the meaning of some written text. Often, direct methods for determining solutions to problems cannot be employed. For example, consider the 8-puzzle of figure 2.1 where the task is to reach the goal state by changing the positions of the tiles. Clearly, any attempt to solve this must involve trial and error (ie search). For such problems, it is convenient for them to be described using *state space* representation and tackled using a state space search. First, a state space containing all the possible configurations of the tiles must be defined. States representing the start configuration and solution configuration must be specified. These are the *initial state* and *goal state* respectively. Note that complex problems may have more than one initial state or goal state. It is also necessary to specify a set of rules for moving from one state to another within the state space. For example, **move(8,5)** might be used to cause the tile in position 5 (the 6) to be moved to position 8 (the blank space) in figure 2.1(a).

It is necessary to consider how to represent information about the current state at each node. For the 8-puzzle, this is trivial. The node could be a series of numbers representing the values of the 8 tiles in the 9 positions. For example the node representing the initial state of the 8-puzzle in figure 2.1 might be (283164705); the zero being the empty space.



2	8	3
1	6	4
7		5

(a)

1	2	3
8		4
7	6	5

(b)

Figure 2.1: (a) initial state and (b) goal state of 8-puzzle.

The AI community have long recognised the growing importance of expert systems in several application domains and, as they increase in complexity, so must the machines they run on. This trend would eventually lead to expert systems becoming extremely expensive to develop. Since most expert systems are based on the same underlying principles, sophisticated tools were designed to facilitate their development. These tools are generally known as *expert shells*. Examples are LOOPS, KEE, ART and Knowledge Craft, which are discussed in Chapter 3.

The 8-puzzle problem provided an example of a simple numerical representation for each of the possible tile configurations in the state space. However, imagine that the problem is more complicated such as telling a robot to walk across a room. How do we represent the environment; such as the positions of chairs and windows. This problem of *knowledge representation* is very difficult and has not been completely solved. If we try to store too much information at each node we may eventually exhaust éven a very large memory. However, if we do not store enough, the problem could become extremely difficult or even impossible to solve. The most important task for a designer of an intelligent system, for example an expert system or expert shell, is how to represent the knowledge base. All other aspects of design, both software (such as search methods), and hardware (memory configuration etc), depend on the selection of a knowledge representation formalism.

Having provided a brief introduction to intelligent systems and knowledge representation, the rest of this chapter aims to provide a brief insight into the types of problems tackled by AI, and the techniques developed to solve them. Various software and hardware based systems are discussed, with particular attention being paid to the current trends – their features, applications and limitations. It is from conclusions based on these reviews that many of the ideas for this project have been developed.

2.2. AI Techniques

2.2.1. Production Systems.

Search is the basis of many intelligent processes. Some sort of structure, which simplified the search process, would be useful. Production systems [3,15] provide such a structure for problem solving and Nilsson [3] describes a basic production system algorithm. Such a structure is useful since new inputs into the database cause behavioural changes in the system, and new rules can be added easily without disrupting the whole system.

```
Procedure    PRODUCTION

DATA ← initial database
until DATA satisfies the termination condition, do:
    begin
        select some rule,R,in the set of rules that can be applied to DATA
        DATA ← result of applying R to DATA
    end
```

Consider the 8-puzzle again. A possible search strategy is **breadth-first** search. Here, it is necessary to construct a search tree with the initial state at the root node. Using the rules, generate all the possible subsequent states at the **daughter nodes.** Then, for each of the daughter nodes, create all the subsequent nodes; and so on until a goal state is reached. Figure 2.2 is the solution graph obtained by a breadth first search strategy. Another possibility is **depth-first** search. Here, a single branch is expanded until either a goal state, or a pre-determined depth, is reached (whereupon the next branch is searched). Such search strategies will find solutions to simple problems such as the 8-puzzle. However, not all problems are so simple.

Consider the following problem:

> **The Travelling Salesman Problem.**
>
> *"A salesman has a list of cities, each of which he must visit exactly once. There are direct roads between each pair of cities on the list. Find the route the salesman should follow so that he travels the shortest possible distance on a round trip, starting at any one of the cities and then returning there."* from Rich [1].

This problem could be solved by either breadth-first, or depth-first search. However, consider the problem if there are a large number of cities. For N cities, the number of different paths from initial to goal state is:

$$T_{sol} = (N - 1)!$$

Consequently, the time required to find the solution would soon become too great to be worthwhile. This limitation is known as **combinatorial explosion,** and should be avoided if possible. One improvement would be to perform a depth-first search and discard any branch whose length increased beyond the current shortest length. This method is known as **branch-and-bound** and is more efficient than the other two but it still takes a long time to solve the problem for a large number of cities.

2.2.2. Heuristic Search.

Since many systematic search techniques give rise to combinatorial explosion, it is often necessary to use a method which is not guaranteed to find the *best* solution, but will find a *good* solution. These techniques are known as **heuristics**. Their main advantage is that generally they give a greatly reduced solution time for hard problems. The **nearest neighbour algorithm** is an example of an heuristic search method and applied to the above problem would involve the salesman starting from any city he liked, then visiting the nearest city that he had not already visited and so on until he had finished. This particular heuristic allows the problem to be solved in polynomial time. In this case:

$$T_{sol} \;\alpha\; N^2$$

A great deal of work has gone into the study of heuristic techniques and this study is often termed **heuretics**.

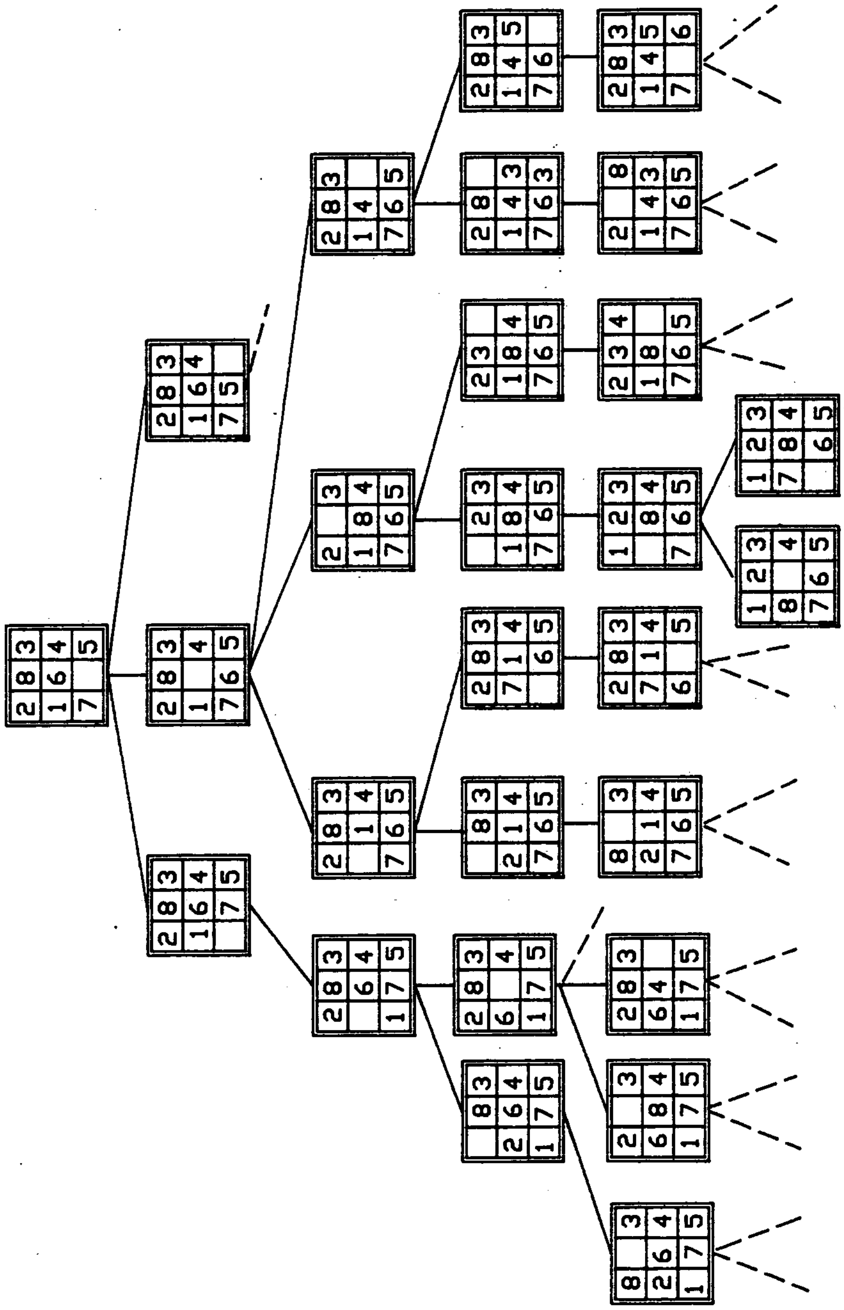


Figure 2.2: Solution graph created by breadth first search.

2.2.3. Rule Selection.

There are various different ways that we can select rules applicable to the search process. The primary requirement is to understand the structure of the rules themselves. In order to be able to match rules against situations, each rule must have a precondition. In other words, we must know what the current state must be for each rule to be applicable. Therefore, we can see that it would be possible to search through all the rules selecting all those with preconditions that match the current state. Unfortunately, this matching process is not always clear-cut and also, as problems become more complex, then the number of rules may increase and so a search before each rule execution would be inefficient. Heuristic techniques are often used to determine which rules should be used; as well as being included within the rules to aid the search process. For a good general overview on this subject, see Rich [1].

2.2.4. Weak Methods

This is the name given to a series of general-purpose search strategies. They are 'weak' because, in certain situations, they are susceptible to combinatorial explosion. The more common methods are described briefly.

Generate and test is a depth first procedure and is of the form: generate a possible solution; if it is a goal state, stop; otherwise generate the next possible solution; and so on. A highly successful program which finds the structure of organic compounds [9] uses a combination of planning and a generate and test search strategy know as plan-generate-test. **Hill climbing** is similar to generate and test except that it uses information fed back from the testing of the possible solution. Rich [1] describes a hill climbing procedure. This method is better than the last and can lead quite quickly to a solution. Unfortunately, hill climbing is susceptible to several problems and it is inefficient in large search spaces. **Breadth first** search was described earlier. Unfortunately, although this method is guaranteed to find a solution, it requires a great deal of memory and a great deal of work, since all nodes are expanded to find all their children. This is an illustration of the combinatorial explosion problem introduced previously. **Best first** search is a combination of depth-first and breadth-first search. The first act is to create all of the daughter nodes of the initial state. Some sort of heuristic function is used to determine how good each node is. The best node is then expanded and so on. **Means-end analysis** is a search technique which reasons both forwards and backwards. The strategy solves the major parts of the problem first and then goes back to solve the smaller problems which occur in fitting the major parts together. The process is based on discovering the differences between the initial and goal states. On finding a difference, an operator (rule) which can reduce the difference is selected. Unfortunately, it may not be possible to apply that operator because the pre-conditions are not correct. Hence, the new problem is to get from the initial state to the state which provides the right pre-conditions. Once the operator is applied, it may not produce exactly the required goal state. Therefore, the next problem is to get from the state produced to the goal state. Thus the problem has been subdivided into two sub-problems. The same process is then applied to the sub-problems. This is a basic example of **hierarchical** problem solving. The General Problem Solver [8] was the first AI program to use this method.

2.2.5. Planning.

Often, we require to solve very complicated problems with many different combinations of states. It is often convenient to decompose such problems into smaller, independent, manageable ones. Unfortunately, some problems are not decomposable since the sub-problems are not independent, but interact. These are often termed **nearly decomposable** problems and can be solved by a combination of decomposition methods and methods whereby interactions are detected, recorded and acted upon during the solution. These are known as **planning** techniques and are very important in instances such as card games where the outcome of a solution step is unpredictable and so the set of possible outcomes is accounted for during the planning stage. A great deal of work has been done in this field; often hand in hand with work on knowledge representation since plan representation, fast data retrieval and manipulation are very important for efficient planning systems. Rich [1] and Nilsson [3] both provide good overviews on this subject. Daniel and Tate [16] provide a more detailed example by means of a retrospective on a specific planning project.

2.2.6. Contexts

There are many problems encountered in AI where we may want to store more than one notional *state* of the database. This is best described by way of two examples.

Consider the problem of circuit design and imagine that we are at the stage of "setting the values" of some components. In circuit design, the alteration of the value of one component invariably alters the requirements for another. This is an example of a nearly decomposable problem which would be tackled using hypothetical reasoning or planning. It would be necessary to look at the effects, on the whole circuit, of making changes to single components combined with any consequent changes to the other components. In this case, it would be advantageous to support multiple values of components and the resultant state of the whole circuit, within the same database so that they can be compared and the best alternative chosen.

Now consider the problem of plotting the actions of a robot. Not only do we have to consider all the alternatives that it could take within its environment at any instance, but also the changes in the environment which may occur with the passing of time.

These different states of the database or knowledge base are known as **contexts.** Examples of AI database support systems which provide such a facility are CONNIVER [18], QA4/QLISP [19], PEARL [20], and HBASE [21]. Tate [17], provides a specification of the functions which he feels should be supported by a context database system for planning applications.

2.3. Knowledge Representation

2.3.1. Introduction.

It has become apparent from the above discussions that AI programs require a great deal of knowledge and associated manipulation methods. We require to store information about the problem domain as well as information about how to find a solution. Knowledge representations and their associated manipulation techniques have been the subject of intense study. Several formalisms have been developed, but generally, knowledge representation can be divided into two types:

- **logical expressions**
- **structures**

This chapter provides descriptions of each of these representations, their advantages and their limitations.

2.3.2. Knowledge Representation Using Logic.

2.3.2.1. Predicate Calculus.

First Order Predicate Calculus (FOPC) is a formalism which has been used for many applications in AI systems. Chapter 4 of Nilsson [3] provides a good introduction to predicate calculus in AI, its syntax and application. Any language is made up of symbols and expressions. In predicate calculus, these are known as well

formed formulae (wff). Well formed formulae are a means of representing facts in logic. For example, the fact "John was a man" could be represented as **man(John)**. Note that, unfortunately, we have not represented the idea of past tense.

We use the logical 'and' (&), 'or' (+) and 'implies' (→) connectives to make compound statements such as "John likes Mary and John likes Escorts" and "John drives either an Escort or a Porsche" and "If the car is John's car, then the car is an Escort".

likes(John, Mary) & likes(John, Escorts)

drives(John, Escort) + drives(John, Porsche)

owns(John, car1) → isa(car1, Escort)

A formula can be negated by preceding it with the symbol "~". Formulae can be quantified either universally (∀x) or existentially (∃x). For example, we would need to specify the information "For all x, where x is a man, then x is a person".

(∀x) man(x) → person(x)

Similarly, we could represent the information "There is a person who wrote Brave-New-World" as:

(∃x) write(x, Brave-New-World)

There is a powerful problem solving technique known as **resolution** which can be applied to knowledge represented in predicate calculus form. Unfortunately, everything has to be converted to clause form which, although not difficult, means that everything looks the same and is not easy for a human to interpret. This makes it difficult for us to interact with a resolution based system. Another difficulty with predicate calculus is that a great deal of formulae are required to describe even simple situations; and since search is an integral part of any knowledge system, it is clear that information retrieval from a large system representing data in predicate calculus form would be slow.

Another drawback with predicate calculus is its inability to represent **beliefs.** For example, how would we represent the following information?

> "I think that Liverpool are the best football team in the world, but Calum thinks that Forfar Athletic are."

There are two major logic techniques which have been developed to handle uncertain logic:

- **nonmonotonic logic**
- **probabilistic reasoning**

2.3.2.2. Nonmonotonic Logic.

Predicate calculus based systems are traditionally **monotonic.** This means that all statements in the database are true and any new fact added will not change the status of any previously inserted fact. This system has the advantages that no checks need be made on the database as new information is added and no record need be kept of the statements which support any new deduced statements. However, in the real world, we do not always have complete information and so we often have to make assumptions to support deductions. Also, the real world is not static and new situations provide new information and disprove some assumptions. **Default reasoning** is an example of nonmonotonic reasoning and is based on making assumptions about situations which are deleted if conflicting evidence is found. Since monotonic systems are a lot easier to deal with, it would be useful to be able to modify them to deal with nonmonotonic reasoning.

The Truth Maintenance System (TMS) developed by Doyle [23] allows monotonic systems to support nonmonotonic reasoning. Each statement in the TMS can be either IN or OUT. That is; either currently believed to be true or currently believed to be untrue. The validity of statements depends on a list of attached justifications. This system, therefore ensures that the database remains consistent. Untrue statements are retained since new information at a later date may cause us to believe them to be true after all.

2.3.2.3. Probabilistic Reasoning.

Often, in the real world, we do not have complete information describing the situation with which we are dealing. An example is medical diagnosis since nobody has a complete understanding of how our bodies function. Mathematics provides us with many theories for dealing with the random world and these theories can be incorporated into the heuristics of our reasoning system. MYCIN [10] is a medical diagnosis expert system which is based on probabilistic reasoning. Generally, however, probabilistic methods are avoided if possible. A number of such techniques are described by Hunt [24].

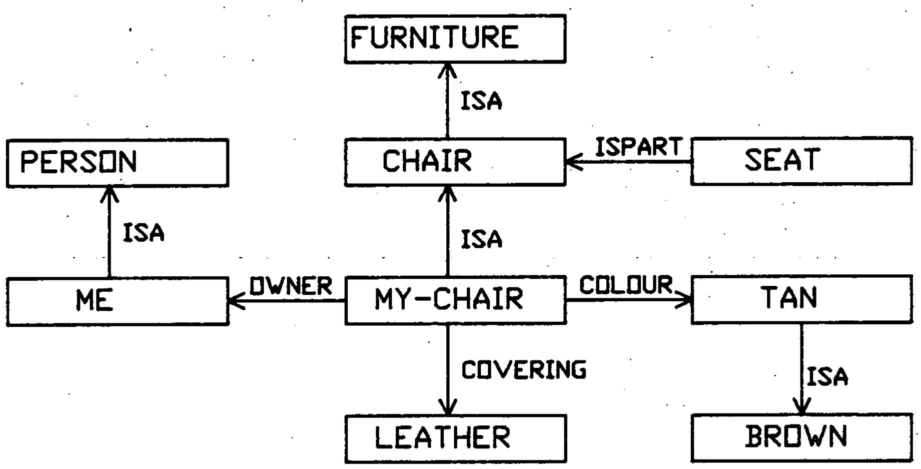
2.3.3. Structured Representations Of Knowledge.

2.3.3.1. Introduction.

As we have seen above, knowledge can be represented by several different logical forms which can be combined with reasoning techniques such as resolution to provide a variety of powerful intelligent systems. Unfortunately, it is very difficult to represent the complex structured and relational information that occurs a great deal in the real world. Structured knowledge representations have been developed to deal with this problem. Such structures must be able to represent *all* of the knowledge that we require; they must provide the ability to infer new information from the old; derive structures to deal with new knowledge; and retrieval of the information must be easy. Knowledge structures can represent patterns in information and as such are often termed **objects** or **schemas**. Generally, objects comprise a set of properties (or slots) which each have one or more values (or fillers), which themselves can be objects. Each object/property/value triple is known as a **relation**, or **fact**. For example: **Steve***(object)* **plays***(property)* **squash***(value)*.

Several schemas exist such as

- frames
- scripts
- stereotypes
- rule models
- semantic nets



(From Rich [1])

Figure 2.3: A Semantic Network

Frames are used to describe properties that objects possess. For example, a chair has a seat, a back and four legs. Scripts are used to describe the sequence of events that would take place in a typical situation such as registering with the Social Security. Stereotypes are used to describe typical characteristics of people, and rule models describe typical rule characteristics in rule sets.

Semantic Nets were developed to represent English words [24]. In a semantic net, relations are represented by sets of nodes connected by arcs. In this way, both objects and events can be described; figure 2.3 (from Rich [1]) is an illustration. Frames, scripts and semantic nets have been used widely in AI systems.

2.3.3.2. Relationships Between Objects.

It is clear that all the structures mentioned above involve relationships between objects. There are two major relationships that are required and these are:

ISA relations which describe the relations between classes and instances of objects. In this way, *set* information can be stored.

ISPART relations which describe the relations between objects and their constituents.

ISA and ISPART are known as **primitives** and are useful links between objects in many knowledge systems. Figure 2.4 (from Rich [1]) illustrates the ISA and ISPART relationships.

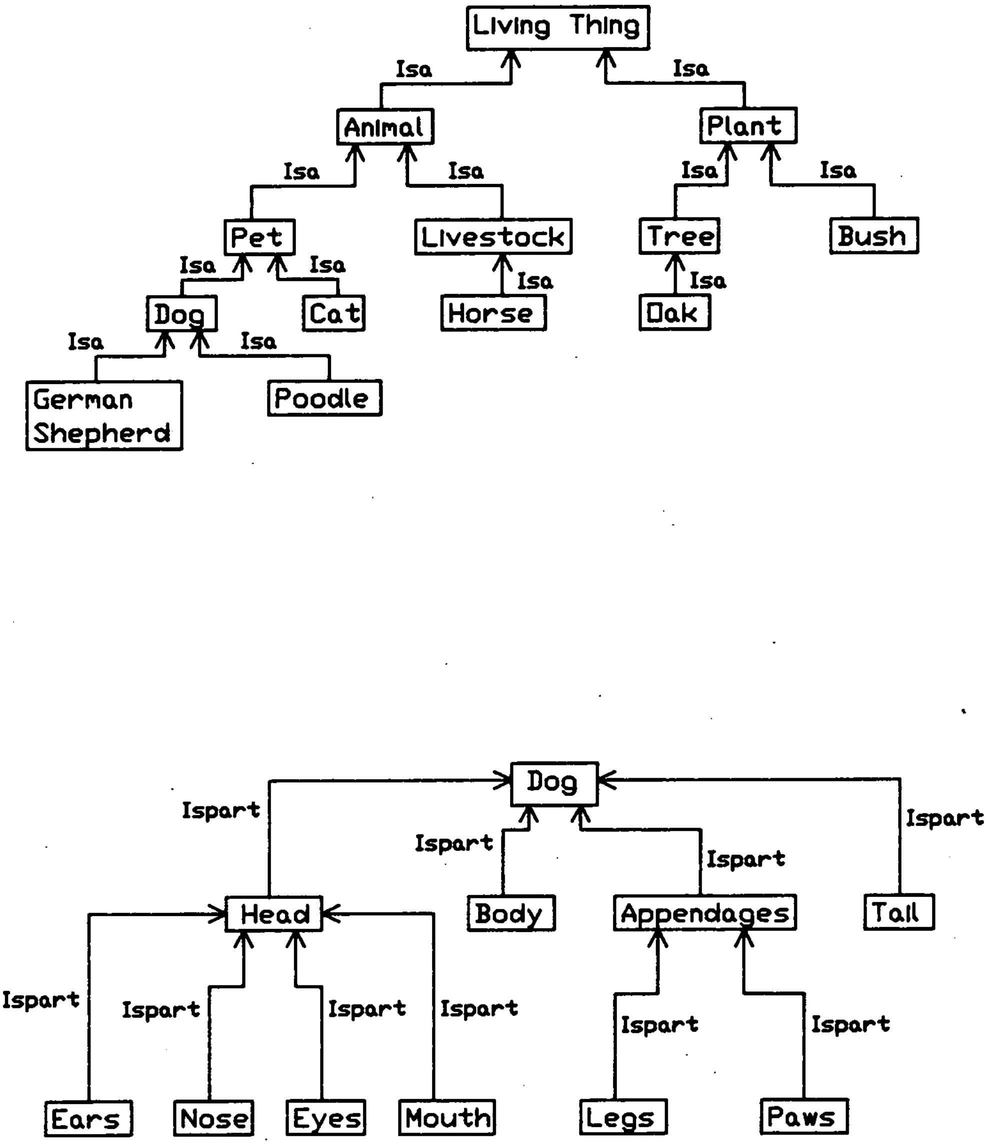


Figure 2.4: Examples of the ISA and ISPART relationships.

Inheritance is the term used to describe how information can be attributed to an object by passing it down from its super-class (ie hierarchically). For example if we know that a poodle is a dog and that a dog is a pet, then we also know that a poodle is a pet. We also know from the ISPART relations of the dog that a poodle has a head and a tail etc. Remember that in predicate calculus, we required separate statements for all of these facts. Note that it would be possible in such a system to represent complex relations; for example the script describing a typical meeting implied by the verb "met", in the relation "John met Mary" We may also want to define the set of operators, or rules, which can be used to alter information within a particular object. Software based on such structures is often termed object oriented programming. It is also possible to trigger procedures if certain properties associated with objects are accessed. This is known as access oriented programming. The LISP Object Oriented Programming System (LOOPS) [29] is an AI software tool which employs both of the above methods.

2.3.4. Summary.

We have seen that there are two major types of knowledge representation. Predicate Calculus seems to be a convenient formalism, which is easy to understand. It is also relatively easy to map such well defined representations onto hardware in the form of lists. Unfortunately, we have also seen that predicate calculus requires a great many well formed formulae to be defined to describe even simple situations. Structured formalisms, however, are able to describe very complex situations and relations quite easily. Unfortunately, because schemas must describe general and changing situations and must be able to cope as new information is added then they cannot be predefined standard structures. This means that mapping onto hardware is difficult.

As application requirements have become increasingly large and complex, much of the research into knowledge based systems has been concerned with decreasing the amount of time spent in searching. Software systems have concentrated on the advantages of inheritance lattices and set classification of structured techniques to increase speed, whereas research into intelligent hardware systems has tended to concentrate on calculus based techniques, since they are easier to define.

2.4. AI Software

2.4.1. AI Languages

The features of AI problems and their representations described above, have led to the development of AI languages to implement them. An important consideration in their design is the belief that intelligent behaviour can be represented by the manipulation of symbols. First order predicate calculus, described above, is an example of symbolic representation, and techniques such as resolution manipulate these symbolic expressions (well formed formulae). Most AI programs will need to create complex data structures to represent intermediate system states, such as partial solutions to a mathematical or game-playing problem, or parse trees for natural language understanding. Since these states cannot be predicted, then neither can the form nor number of the resulting intermediate data structures. To be able to solve complex problems, we need to have a great deal of background knowledge *(knowledge base)*, and our AI programs need to be able to manipulate this knowledge; ie: interrogate, insert, modify and delete it. Generally, search techniques are based on pattern matching of information in the knowledge base; to check the validity of conditions prior to the execution of a rule (IF *condition* THEN *action*), for example. Corlett [25] presents a list of AI language characteristics required for AI systems. These are summarised below:

(i) *Symbolic manipulation* should be easy

(ii) We need to create intermediate data structures, which are both arbitrary and complex. Linked lists are very useful for this purpose and so *list manipulation primitives* should be supported.

(iii) Since such intermediate structures are being created continually during problem solving, space must be allocated for them, and deallocated when they are no longer required. This operation should be transparent to the programmer and *automatic storage allocation and reclamation* (garbage collection) should be an integral part of any AI language.

(iv) We cannot predict the size of intermediate data structures or the type of symbols until they are created (at execution time). Consequently, *dynamic binding* of types and sizes should be supported. Generally, this must be performed sequentially, and is therefore slow, and architectures have been developed to implement dynamic binding (described below).

(v) *Pattern matching* facilities should be supported to identify symbols and control the execution of the problem solution. Logic programming (cf: production systems) is such a mechanism, and is of the form:

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

Both the condition and action could be a group of symbolic expressions. The condition would describe characteristics of the current state of the solution state-space, while the action would manipulate the state-space to cause changes, hopefully leading to a solution. A search strategy known as *forward chaining* simply starts at the initial state and fires rules until the solution is found. Another search strategy starts with the desired solution and the *goal* is to derive the initial state. Intermediate conditions are known as subgoals, and this form of reasoning is known as *backward chaining*. Intelligent search techniques and heuristics would be employed to control the search path, and a combination of forward and backward chaining would be used.

(vi) Since we need to create new procedures at run time to control the flow of the solution through the intermediate (partial solution) states, we must be able to pass procedures as data. Furthermore, it is desirable to pass procedures and data associated with a particular entity as a single object *(object oriented programming).*

Although there are several AI languages currently in use, no one language exhibits all of the above described features. The two most popular are LISP [26,27] and Prolog [28].

LISP was developed at MIT by John McCarthy in 1960 and has since been modified into several different *dialects,* although the trend recently has been to move towards a common standard known as *CommonLISP*. The name LISP is derived from its description: **LIS**T **P**rocessing, and is a language for manipulating *symbolic expressions*. Symbolic expressions are made up of two data types; namely, atoms and lists. Atoms are the basic symbolic entity and can be numbers, characters or character strings. Lists are composed of atoms, and expressions are built up from combinations of atoms and lists. Functional operators (primitives) manipulate expressions. For example: *(SETQ L '(A B C))* creates a symbol called L and assigns as its value the list (A B C). The expression *(CAR L)* returns the first element of the list; namely, A. The expression *(CDR L)* returns everything except the first element; namely, the list (B C). The LISP Object Oriented Programming System (LOOPS) is an example of a LISP based expert shell utilising a structured knowledge formalism, and is described below.

Prolog (**Pro**gramming in **log**ic) was originally developed by Alain Colmerauer at the University of Marseilles, in about 1970. It is based on first order predicate logic, and so its form is akin to the well formed formulae described above, although it is not as expressive as first order predicate calculus. Prolog has rapidly gained in popularity over the last five years, partly due to the decision by the Japanese to adopt a modified version of DEC-10 Prolog as the core language for their Fifth Generation Computer Project.

Other applications include:

- Automatic Theorem Proving
- Planning
- Compiler Writing
- Intelligent Knowledge Based Systems
- Natural Language Processing
- Expert Systems

Expressions are translated into predicates and their arguments in a similar form to the well formed formulae. For example:
"A man is happy if he is rich and famous" translates to:

```
happy(Person):-
        man(Person),
        rich(Person),
        famous(Person).
```

Prolog is also able to perform list processing in a manner similar to the LISP primitives. In prolog, the method is to *cut* a list, which is analogous to CAR and CDR in LISP.

[X|Y] = [s,t,e,v,e] will result in X = s and Y = [t,e,v,e]

The first element of the list is known as the *head* and the list formed by deleting the head is the *tail*. Rules are of the form:

GOAL is_true_if (SUB_GOAL_1 and SUB_GOAL_2 are true)

The goal is the solution to the particular problem, and the rule is evaluated by attempting to satisfy the sub-goals by pattern matching techniques. If a sub-goal is unknown, then backward searching is performed to find another rule which can be evaluated to satisfy that sub-goal, and so on.

2.4.2. Expert Shells

These tools have been designed for developing expert systems. Corlett [25] describes several desirable expert shell features which include the provision of a smooth man machine interface in the form of a window based interactive environment (which is controlled by a "mouse" and keyboard), an integrated editor, debugging aids, pretty printing, automatic filing, programming explanations and safeguards against program crashing.

LOOPS [29] was developed because AI systems are large and complicated and require different powerful techniques which may be applied to different parts of a problem. It supports several programming paradigms:

- procedure oriented programming.
- object oriented programming.
- access oriented programming.
- constraint oriented programming.

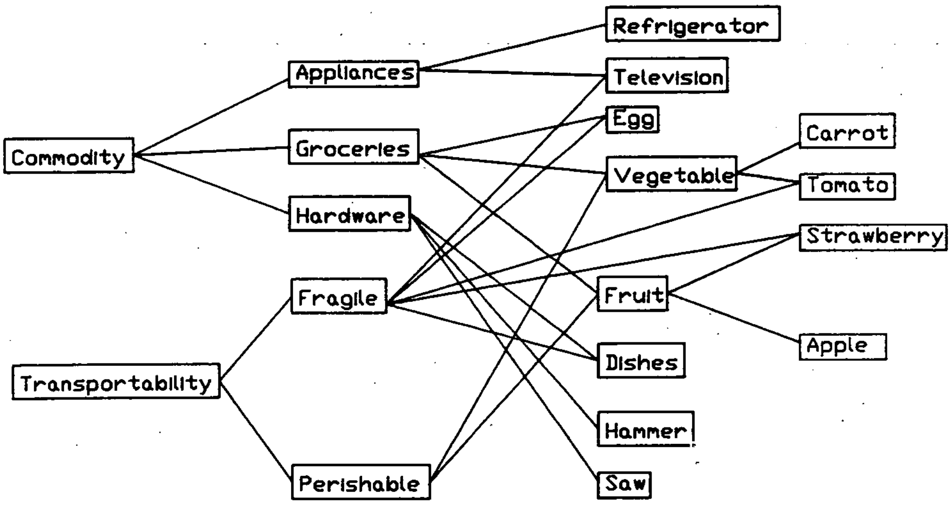


Figure 2.5: Example of an Inheritance Lattice (LOOPS)

Constraint oriented programming is really an application of access oriented programming which, along with object oriented programming, are the most widely used aspects of LOOPS. Knowledge is represented in the form of "objects" which are related to each other in class inheritance lattices. Figure 2.5 is an example of such a lattice. Objects have methods and variables associated with them. *Class variables* hold information shared by all instances of a class while *instance variables* contain information specific to a particular instance. *Methods* can be thought of as functions and can be sent *messages* to which they respond. Additionally, *active values* can be assigned to an object. If these are accessed, then some action is initiated (access oriented programming).

KEE † (Knowledge Engineering Environment) [30] is a similar system which is designed to facilitate fast prototyping of expert systems. Applications are chiefly diagnosis, simulation and planning. It provides object and access oriented programming paradigms and also a rule based programming paradigm for reasoning techniques. The basic knowledge structure is the frame; ie a slot and filler system. The slot is a property associated with a frame and can have a value (which may be a default value) or a method (procedure which is executed if the slot is accessed in a pre-determined manner). The environment is based on a class and subclass structure and slots are inherited from superclasses. New slots can also be created at any level of an inheritance lattice; these are then inherited by subclasses.

The *KEEworlds* (cf. contexts) facility is provided which is of particular use in planning applications. This facility allows multiple situations to be supported which can be thought of as hypothetical situations, states in problem solutions or the *time* dimension to a problem which is time dependent. The *root world* is the world of facts which are true in all situations. A truth maintenance facility is provided which checks for inconsistencies within worlds.

† KEE is a product of IntelliCorp.

Inference ART ‡ [31] (Automated Reasoning Tool) is very similar to KEE. It is a rule based system and the user is allowed to define an environment (knowledge base) and a set of rules which manipulate the environment to produce a behavioural pattern which describes the desired situation (either the scheduling of taxis in a city or the modelling of some electronic circuit, say). Rules are applied to the environment in order of priority and this priority or *salience* is defined by the user. ART also incorporates a truth maintenance system based on logical dependencies defined by the user. Objects, concepts and relations which make up the environment are described using a set of schemata of a slot and filler type. These schemata are used to create an inheritance lattice similar to KEE. ART also supports a context mechanism known as *viewpoints*.

Knowledge Craft † [32] incorporates several ways of both representing and reasoning with knowledge in an attempt to make it a tool which can be used to develop expert systems in different problem domains. These are (from [32]):

- A schema-based representation of knowledge, which permits inheritance of values between linked schemata.
- Object-oriented programming, using Common Lisp as the language for writing procedures which are called by objects.
- Rule-based programming using a forward chaining reasoning strategy. This is available via CRL-OPS, which is an extension of the rule based language OPS5.
- CRL-Prolog, which permits rule-based programming using a backward chaining reasoning strategy, and also provides most of the other facilities of DEC-10 Prolog.

‡ Inference is a trademark of Inference Corporation of Los Angeles.

† Knowledge Craft is marketed by the Carnegie Group, Pittsburgh, Pennsylvania.

- A context mechanism for hypothetical reasoning etc.

2.5. AI Hardware

2.5.1. Introduction

There have been many attempts to provide hardware support for AI systems; including advanced Von Neumann processors [36,37,38,39,40,41], VLSI processors [33,34,35] which support an AI language such as LISP, content addressable memories and processors [45,46,47] and specialised graph reduction engines utilising parallel networks of transputers [48,49,50]. This section provides an overview of the current state of AI Hardware technology, and summarises the advantages and disadvantages of various approaches with respect to this project.

2.5.2. Processors for AI

There have been two different trends in processor architecture development.

(i) architectures which support AI languages

Some manufacturers have produced processors which support high level AI languages – generally LISP. Since AI languages do not differentiate between functions and data, but refer to them as symbols, types have to be evaluated at run time. This would normally be a slow sequential operation. *Tagged* architectures allow the exploitation of parallelism, since dynamic type checking is performed at run time by using a few bits of each word for type identification. Associative memories, which are accessible by content rather than address, can be used to support the unification process in Prolog (ie: pattern matching during the rule evaluation process). These features have facilitated the development of lower cost AI workstations such as the Symbolics range of LISP machines [32,33].

(ii) enhanced Von Neumann architectures (RISC and CISC)

Most computers have been based on the Von Neumann architecture, and have employed complex instruction sets, coupled with simple compilers with basic memory-to-memory operational models. Since memory was slow and expensive, designers tried to replace groups of instructions with single, more complicated ones, thereby creating Complex Instruction Set Computers (CISC). Unfortunately, the decoding of such complex instructions into internal microcode not only added an extra layer of software to the system, but required several machine cycles to be executed. The introduction of pipelines and large numbers of internal registers have made possible the execution of many of these operations within a machine cycle.

Several major processor manufacturers such as Advanced Micro Devices and Intel have developed such processors; culminating in the release of the Am29000 Streamlined Instruction Set Processor [38] and the Intel 80386 [36,37]; which have been incorporated into advanced workstations. They are characterised by their large physical address spaces (approximately 4 gigabytes), and their very large logical address spaces (64 terabytes for the 80386). The main feature is to use VLSI techniques to pack as many of the normally external devices as possible onto a single IC. Examples are the Memory Management Unit on the Am29000, and on-chip storage for object code on the 80386. The Am29000 has 192 internal 32-bit general purpose registers, which can contain data or addresses, and can be accessed by any instruction. This drastically reduces the amount of time spent waiting for external data. A four stage pipeline is used to implement single cycle instructions, and an on-chip Branch Target Cache can be used to allow single cycle branches for program loops.

An alternative approach was to develop processors which executed a small, but well chosen instruction set, some of which could be mapped directly into hardware.

Consequently, programs are longer, but are executed very quickly. Such an architecture has become practical as a result of the development of large, high-speed memories. As a result, the philosophy of the Reduced Instruction Set Computer, *RISC*, was introduced [39]. The RISC architecture includes several features which aim to increase system performance. The reduced instruction set comprises simple, fixed-length instructions, which can be decoded quickly. Except for off-chip LOAD and STORE operations, all instructions are register-to-register. This shortens cycle time and simplifies virtual memory management. Single cycle execution is possible, since all operations, except off-chip communication, are internal. A delayed pipelined architecture allows the RISC to fetch the next instruction during the current one even for branch instructions. Examples of RISC based systems are the Acorn R140 (which is a UNIX based 4 MIPS RISC workstation) and the SUN SPARCstation (the SPARCstation 300 is a 16 MIPS UNIX machine) [40,41].

2.5.3. REKURSIV Processor

The disadvantage of RISCs is the need for compilers to produce large pieces of code to get round the limitations caused by the instruction set. The designers of the REKURSIV processor [43,44] have therefore moved in the opposite direction and have designed an enhanced instruction set, providing ultra high level data-driven primitives, such as tree-copying. Although this approach is akin to the enhanced CISC architectures, the REKURSIV is recursively microcodable and comprises a tightly coupled cluster of processing elements, each working on separate functions, such as type checking and range checking. The REKURSIV is controlled by a 160-bit control word.

> "*...a sensible microcode language not only removes the fetch overheads, it actually enables much of the 'algorithmic linkage' that exists at the start and end of classical instructions to be completely removed, then that which is left, the essence of the algorithm, can often be squeezed up in parallel in different parts of the processor...*" from Harland [43].

2.5.4. Associative Processors

This field of research has been active since the early 1960's and encompasses Content Addressable Memories, Content Addressable Processors, Associative Memories and Associative Processors. Effectively, a content addressable machine comprises memory cells, each of which has enough processing power to determine whether it contains the required data requested by some central controller. An appropriate analogy might be to say (Foster [45]):

"Will all cells containing the number 1234 please hold up their hands"

Unfortunately, since each memory cell requires extra processing circuitry, such devices are large and expensive. Consequently, they are still very much special purpose devices for small applications. Research is being stimulated, however, by the improvement in VLSI techniques and the prospect of WSI (Wafer Scale Integration), and several projects such as WASP (WSI Associative String Processor [46]), and GAM (Generic Associative Memory [47]) are in progress in the UK. Application areas include set processing (eg: class information in an information network), string processing and relational data processing (eg: expert and intelligent knowledge based systems).

2.5.5. The Transputer

This is a VLSI device which contains a 32-bit processor, local memory and communications links for direct connection to other transputers [48]. The internal memory reduces the requirement for slower off-chip memory accesses, therefore increasing run-time speed. The ability to interconnect to up to four other transputers makes feasible the development of a *connection machine*, or *Boltzmann machine* (*MIMD – Multiple Instruction Multiple Data machine*) [58], where each node in a semantic net is represented by a processing element. Another possible application is in the field of parallel graph reduction.

2.5.6. Graph Reduction Engines

The representation of problems in search trees, or *graphs*, was discussed earlier in this chapter. Graph Reduction is a technique which splits the problem graph into separate, decomposable parts and attempts to find solutions to these sub-parts. The sub-parts themselves are repeatedly sub-divided until a solution is found. The sub-parts would normally be distributed over a parallel processing system. There are two major projects involving graph reduction in progress in the UK.

- **GRIP** [50]
- **Flagship** [49]

GRIP (Graph Reduction In Parallel) is, as its name suggests, a parallel graph reduction machine and is under development at University College, London. The system comprises a group of loosely coupled Processing Elements (PEs) connected via a wide bandwidth bus (IEEE P896 Futurebus), under the control of a bus interface processor. In this case, the processing elements are Motorola 68020 microprocessors, with floating point co-processors, each with 128k bytes of local memory. One PE, the System Manager, communicates directly with a UNIX host and controls resource allocation. The graph (state space representation) is distributed over the PEs, which then perform the problem reduction and solution.

The Flagship project is being carried out by the University of Manchester, in conjunction with International Computers Ltd., and Imperial College, London. The Flagship architecture comprises several closely-coupled Processing Elements interconnected via a communication network. Sub-tasks are distributed over the network and are evaluated in parallel. In this respect there are many similarities between the GRIP and Flagship projects.

2.6. Other Systems

There are three Knowledge Based hardware systems which have moved beyond the concept stage into commercial products. For this reason, they are presented in a separate section, and are used as standards against which the SKMS prototype may be compared (see Chapter 5).

2.6.1. The Intelligent File Store.

The Intelligent File Store (IFS) provides hardware support for large knowledge bases [52,53,54,55]. It is aimed at applications including deductive databases, expert systems and cognitive modelling. The system uses first order predicate logic (FOPC – discussed above) to represent knowledge. The Qualified Binary Relationship Model (QBRM) [55] has been developed by the group as a means of decomposing semantic networks into FOPC form which are used by special purpose hardware for fast pattern directed searches. This hardware, at the lowest level, takes the form of a five field associative predicate store (APS [74]). Some sample entries might be, taken from [55]:

#3	John	Likes	Mary	*UNDEF.*
#4	David	Thinks	**#3**	*TRUE.*
#12	r1	IS.A	RULE	*UNDEF.*
#23	Fred	Lives in	Didsbury	*PROB.*
#24	**#23**	HAS.PROBABILITY	0.8	*TRUE.*
#11	**#12**	HAS.CONDITION	**#14**	*TRUE.*

A Lexical Token Converter (LTC) [54] converts each character string to a unique internal identifier. Note that each well formed formula is assigned a unique label so that multiple order information can be built up by referencing these labels in other wffs (eg. statement #4). Boolean values are also assigned to each wff to augment uncertain knowledge, such as beliefs (probabilities).

2.6.2. Generic Associative Memory.

Generic Associative Memory was developed at the University of Strathclyde and is concerned with parallel network architectures for large knowledge based systems [47,56]. The knowledge representation formalism is similar to that used in the IFS; however, no boolean value is included. In this case the formalism is constructed of 4-place relations known as **facts**. Each fact comprises a name (similar to the identifier in IFS), a *subject*, a *relation* and an *object*. Again, the fact name could be referenced within other facts to increase the order. Some typical facts might be:

Name	**Subject**	**Relation**	**Object**
"First"	"John Smith"	"works on"	"bridges"
"Fifth"	"bridges"	"carry"	"traffic"
"Ninth"	"First"	"Occurs"	"Tuesdays"

Note that set membership information can be represented in fact form also and so generic operations are possible in such a system. The Generic Associative Memory (GAM) is an associative processor which operates on classes, their members and sets. The Generic Associative Array Processor (GAAP [56]) is an array of 64 by 64 GAM devices each of which can perform fetch, insert, delete, join, union, intersection, difference, and division operations on the knowledge base (FACT store).

2.6.3. Ferranti Relational Processor

The Ferranti Relational Processor (FRP [72]) comprises a two-card VME/VSB bus compatible module, incorporating 8 Mbytes of RAM. It is targeted as a relational query system for real-time applications – for example, Ferranti quote highly flexible demand-driven applications for sensor-derived data systems such as Threat Evaluation and Weapon Assignment or interactive Captain's Combat Aid facilities. The FRP uses an automatic indexing technique to allow access of information by single or multiple *key* attributes, thereby simulating content addressable memory with conventional RAM. Information is stored in memory as tuples (well formed formulae). It implements all of the content-addressable memory operations, and supports all six comparison operations (<, <=, =, >=, >, !=). As a consequence, the system supports *between bounds* and *nearest to* searching. Hence, the following query type may be supported:

> *"Yield (specified) data on all targets within a (specified) bearing sector and a (specified) height band in descending order of threat" [72].*

2.7. Summary

As a consequence of the greatly increased interest in information technology (IT), the market share of data manipulation system applications is far in excess of that of numeric applications. Knowledge based manipulation systems and associated AI techniques are beginning to displace basic databases as the *core* of the IT system, and providing subsequent increases in power. This arises from the ability to store complex information and to perform inferential tasks, which lead to the evolution of new information. Structured knowledge representations reduce the amount of time spent searching the knowledge base, and in particular, facilitate the performance of set (or class-based) operations. These knowledge based systems have, for the most part, comprised highly complex software which has remained unnecessarily complicated since conventional, ie: Von Neumann, architectures have been unable cope with the functional demands that are required.

It has been estimated that, by 1992, knowledge based systems will account for about 13% of the annual UK computer market, which amounts to approximately £300 million of hardware and software sales [57]. Consequently, there has been an increase in research activity to develop new architectures able to support such systems.

All research into new architectures must take account of the hardware, software and theoretical issues; which include knowledge representation and manipulation mechanisms. This section has attempted to describe, albeit superficially, a cross-section of the developments which have arisen from current architectural research projects:

- record searching engines (content addressable memories)
- knowledge manipulation engines (IFS, GAM)
- graph reduction engines (Flagship, GRIP)
- cognitive modelling machines (connection machine)

The first two architectures are effectively back-end (ie: memory dominated) systems, whereas the latter two are front-end (ie: processor dominated) systems. Each type of architecture has its advantages and disadvantages, which tend not to overlap (see Table 2.1 adapted from Lavington [57]). It is perhaps understandable, then, why no one system has emerged, which provides the all round (memory and processor) support required by knowledge based systems.

Limitation	content addressable memory	knowledge manipulation engines	graph reduction engines	connection machine
unit of storage	x			
search capability	x			
functionality	(x)	(x)		
data capacity			x	x
cost performance			(x)	x
systems integration		x	x	x
portability		x	(x)	x

Table 2.1 limitations of existing architectures for knowledge based applications

Lavington [57] has suggested twelve topics of research in this area which he feels are likely to contribute to architectures for large knowledge based systems.

1. Memory structures for large concurrent systems involving objects.

2. Efficient strategies for integrity maintenance† in deductive databases.

3. Alternatives to depth-first, entity-at-a-time proof-mechanisation strategies. The emphasis would be on methods such as set-based resolution, graph paradigms, etc., which are capable of being supported by parallel knowledge manipulation engines (KMEs).

4. Knowledge representation formalisms which combine efficiency with expressive power.

5. The formal definition of a procedural interface from which compact user-level software can be constructed. This would aid inter-project collaboration by bridging the gap between advanced knowledge representations and practical

† cf: truth maintenance

Intelligent Knowledge Based System (IKBS) implementation languages.

6. The integration of logic programming and functional programming paradigms in an application driven approach to the design of large knowledge based systems.

7. Memory organisation for very large functional programs applied to non-numeric problems.

8. A parameterised synthetic knowledge base generator for use in an analytical approach to benchmarking.

9. Hardware assistance for concurrent-user control.

10. A knowledge base server for a graph reduction machine.

11. The design of a highly parallel KME.

12. Custom VLSI for relational algebraic processing.

This thesis addresses some of the issues raised by Lavington. A structured knowledge representation has been developed, involving objects which comprise relations constructed from properties and values, supporting multiple contexts. Status words are able to support relation *confidences* such as *true, false, probable* and *undefined*, which support a truth maintenance system. The structure can be used to build *ISA* links between objects, for use in set-based (relational algebraic) operations. Software simulation, described in Chapter 3, confirmed that the structured knowledge formalism is both expressive and can be supported by special purpose hardware. Moreover, set operations may be supported directly by hardware. A proposal for the development of a parallel KME (Parallel Relational Processor System) which performs breadth-first search (each processor performs depth-first search), involving several *VLSI Relational Processors* based on the architecture described herein, is discussed in Chapter 6.

CHAPTER 3

System Specification and Design

3.1. Introduction

The preceding chapters have concentrated on various aspects of artificial intelligence – knowledge representation and manipulation techniques. Several intelligent systems, both software and hardware oriented, have attempted to solve or ameliorate the problems associated with the heavy searching workload. Software solutions have concentrated on structured knowledge representations which allow the user to home into a particular piece of information by way of inheritance lattices, and to facilitate reasoning about the complex situations which arise in the real world. Hardware solutions have been varied, but are generally based on First Order Predicate Calculus (FOPC) methods. Such approaches are inherently slower than structured formalisms but easier to manipulate. Hence, hardware support for a structured knowledge-based system offers an attractive alternative solution, and the development of an expressive yet easily manipulated knowledge structure is an important feature of the Structured Knowledge Manipulation System (SKMS) described in this thesis.

This chapter describes the development of the knowledge representation formalism and a functional specification describing the desired manipulation facilities for a knowledge-based system. Software designed to investigate the structure is discussed, and opportunities for hardware support, consequent performance improvement, and various alternative architectural approaches, are presented.

The selection of an appropriate structured formalism is not a trivial task. A knowledge representation which has rigidly imposed structural components is easy to manipulate in hardware, but is restrictive for many "real world" applications. Whereas a flexible representation, able to grow and mutate with its knowledge environment, is generally difficult to support in hardware. Before defining a suitable structure, it was necessary to decide what information it should contain, and which operators should be allowed to manipulate it.

Another important consideration is the choice of memory technology. Content Addressable Memory (CAM) was investigated, but suffered from two major drawbacks. Firstly, it is very expensive. Secondly, it is difficult to integrate large quantities of memory onto a single chip with current technology. Consequently, this option was discarded, and it was decided to make use of standard, "*off the shelf*", components which would involve no special construction techniques and would keep down the cost. As a result, any structuring of information, by way of links between objects, has to be supported by linked-list type constructs. The pointers used to construct the lists are analogous to the identifier tags used in the IFS and FACT projects (see Chapter 2). In this case, however, the lists are used to build up knowledge *structures* instead of first order logical formulae (well formed formulae).

3.1.1. The Knowledge Structure

A data structure which provides maximum flexibility for evolving objects with associated general and class-based relationships, is the *general (n-ary) tree* [59,65]. The solution graph of a breadth first search (figure 2.2) and the ISA and ISPART relationships (figure 2.4) illustrated in Chapter 2 aıe botn examples of a general tree.

The easiest way of implementing a tree in standard Random Access Memory (RAM) is by way of linked lists. The root of the tree would be connected to its children by way of pointers. Each of these children (or siblings) might be connected to its own children by pointers. (If the children also point back up to their parents, then the list is *doubly-linked.*) So, each *node* or *cell* of a singly-linked tree comprises a name and a set of pointers to its children. This is illustrated in Figure 3.1.

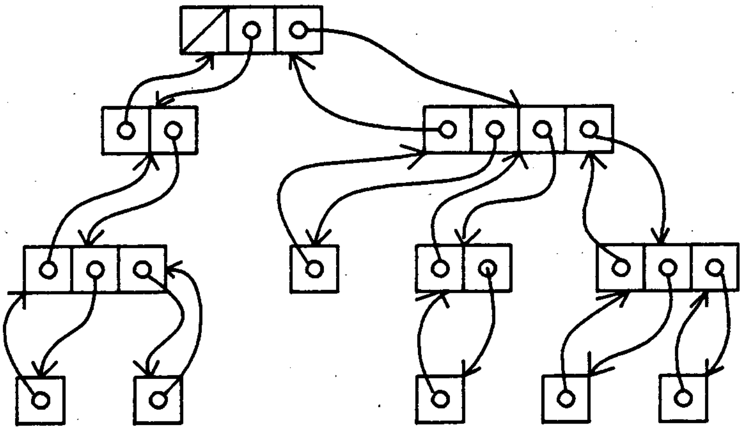


Figure 3.1: The linked list representation of a general tree

Unfortunately, if each node has a different number of children, then in the worst case, every node in the tree will have a different structure. One solution would be to allow variable size nodes, but assign each node a header field which keeps track of the number of links. This is fine if the only task is to search through existing information, but creates difficulties if we want to *modify* the information; ie: by adding or removing links. Another solution would be to allow a maximum number of children and reserve enough space to store all the required pointers. This option is wasteful of space if we decide to go for a very large number of children, but inflexible if we choose to allow a restricted number of children. A better solution is to represent our general tree in the form of a *binary tree*. A binary tree node comprises the node information and links to two other nodes. The first link is to its *first child* and the second link is to the *next sibling*. If there is no sibling, then the pointer will be NULL. Figure 3.2 is an example of a binary representation of a general tree. Such a tree is easy to build and easy to modify.

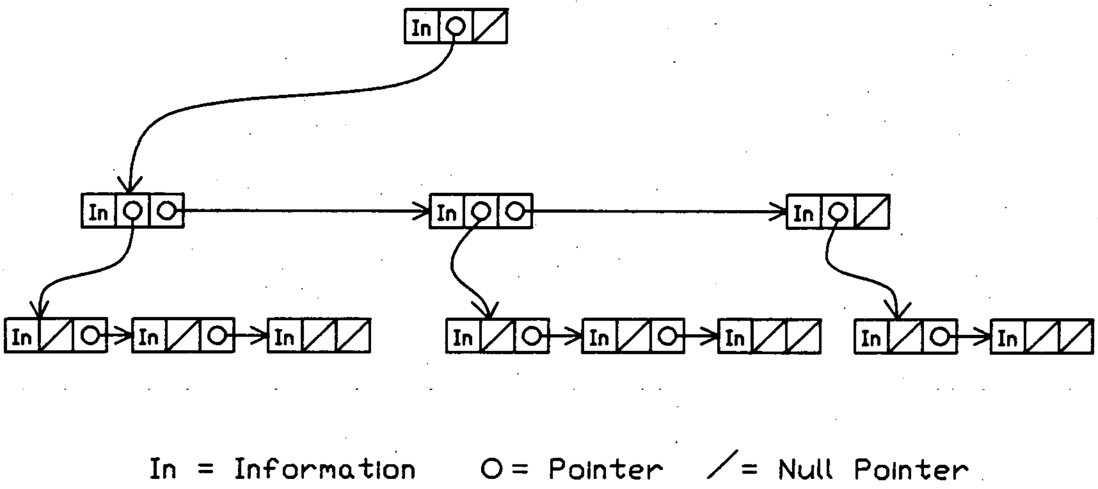
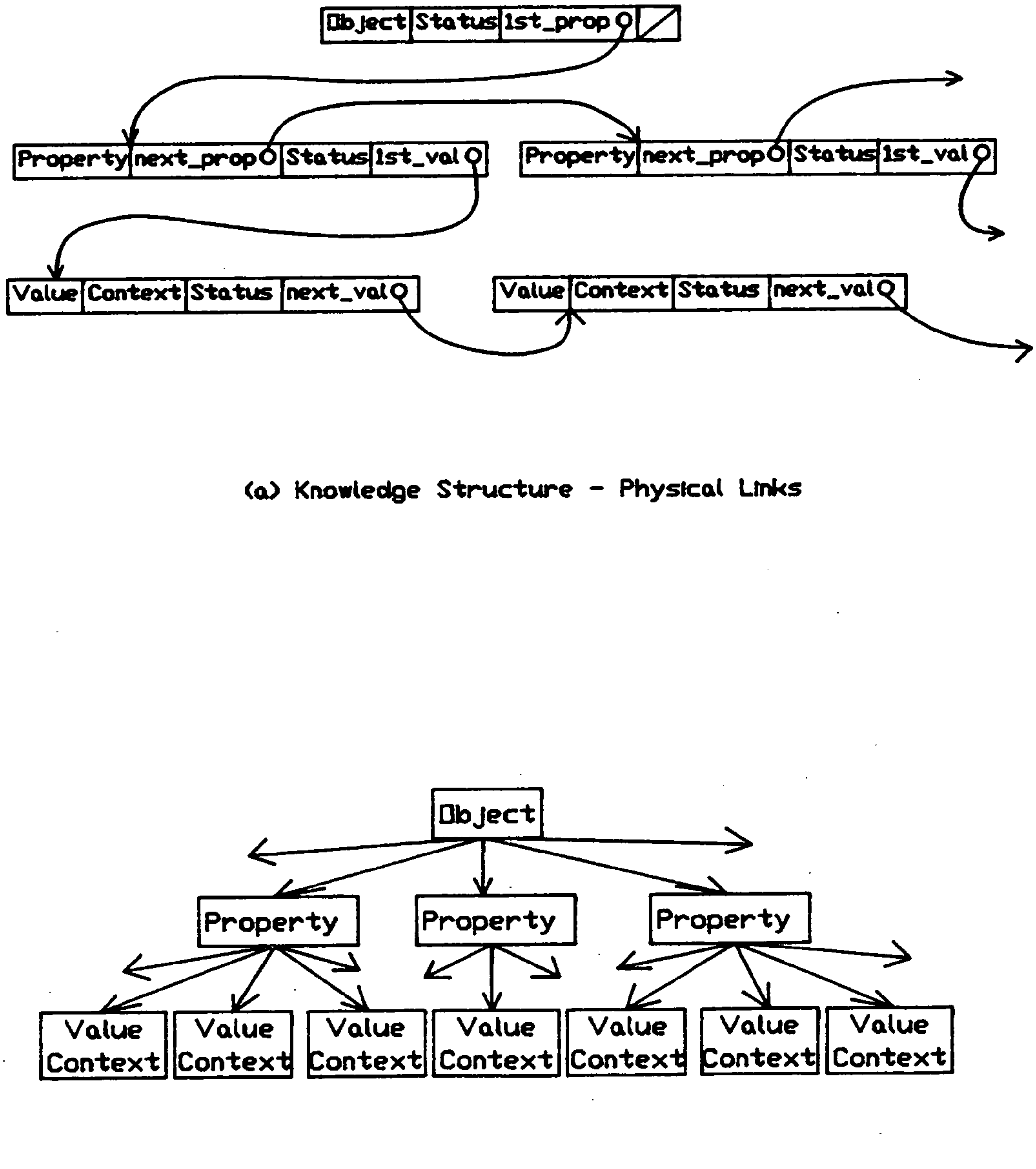


Figure 3.2: The binary representation of a general tree

This technique was adapted to create a general tree structure relating *blocks* of objects, properties and values, which provides the necessary flexibility, while maintaining a regular structural framework. Objects have properties (or slots) whose values (fillers) depend on the particular context being considered. These constructs are known as *relations*. The value can be either an atom, an object, or a link to another relation; either within the same object or within another. Each relation has a *confidence* associated with it, which can be TRUE, FALSE, UNDEFINED or PROBABLE (this information is stored in a status word). Status information is also included for tagging, marking and masking purposes. Objects can have any number of properties, which in turn, can have any number of value/context pairs. Thus, objects of any form can be developed. Each object, property and value block uses its *address* as a unique ID. Each relation associated with an object is analogous to the tuples developed for the Intelligent File Store, or the facts of the Generic Associative Memory project. The essential difference is that these relations are connected physically (by links) within a structure so that search algorithms can home in quickly to specific *subjects* of information.

The price of descriptive flexibility within strict structural constraints is the extra memory requirement for these pointers. However, the linked-list format can also be used to connect unused memory blocks, and a concurrent, *free-list* garbage collection algorithm with *no memory overheads* and *very little speed penalty* can be implemented. This approach, therefore, is very attractive. The knowledge structure, with its conceptual and physical links, is illustrated in figure 3.3.



(a) Knowledge Structure - Physical Links

(b) Knowledge Structure - Conceptual Links

Figure 3.3: The SKMS knowledge structure

3.1.2. Knowledge Manipulation

The term *knowledge manipulation* is usually used to describe fast pattern-directed search of a knowledge base. However, there are several other operations that must be supported. Clearly, before we can search a knowledge base, we must be able to create it. The basic production system algorithm described in Chapter 2 shows that to solve a problem, we must be able to infer new information from our present state. Therefore, we must also be able to append to the knowledge base, at any time. Additionally, we may find that information needs to be altered, either by modifying existing values, or by deleting them from the knowledge base. If a truth maintenance system is to be implemented, we need to be able to support confidences in our information (TRUE, FALSE, UNDEFINED, PROBABLE), and be able to modify them.

It may be that the knowledge base user does not have a full specification for the relation in which they are interested, but only part. Consequently, the system must be able to support wildcard values and so act upon either the first successful match, or all of them. In certain circumstances, it may be desirable to match against a *situation* rather than a single relation (*set* operations); for example:

DO

operation

IF

fact_1 is TRUE

AND

fact_2 is TRUE

etc...

...and...

DO

operation

IF

fact_1 is TRUE

OR

fact_2 is TRUE

etc...

Situations are constructed from logical connections between relations, and search based on such requests should also be supported. This ability is particularly important for performing set operations where the relation property is "ISA".

3.1.3. Garbage Collection

The ideas of creation, modification and deletion of linked-list based information, expressed in the previous section, require that we allocate memory space dynamically. Some applications of the knowledge system may only require moderate modification of the knowledge base. Others, such as hypothetical reasoning, may require the temporary creation of new contexts, such as the circuit design example of Section 2.2.6, which would mean that large sections of memory would be allocated and then discarded. In this case, even very large storage media would soon be exhausted. A solution is to employ regeneration algorithms which will identify discarded memory cells *(garbage)*, and re-allocate them in future operations. This technique is known as *garbage collection.*

Garbage collection solves our memory exhaustion problem, but unfortunately, introduces others. Firstly, such operations take time to execute and so reduce system performance. Systems, such as the Expert Shells described in Chapter 2, often employ two different garbage collectors. The first is usually a concurrent algorithm which can do a small amount of collection in idle times between operations. The problem occurs during long, complicated, memory-intensive operations, when the amount of free memory space becomes critically low, and a second garbage collector stops the system and recovers all of the unused space. These interruptions to processing are unpredictable and would be disastrous in terms of performance for real-time engineering applications.

Several garbage collection algorithms have been proposed, and Cohen [60] provides a good overview of the subject, although for our purposes, they can be divided into two main types:

- **free-lists**
- **copying**

Free-lists, as one would expect, are simply lists of unused memory cells connected together, and available for allocation. A basic example of a free-list method is the Reference Count method [60], which utilises *tag* bits in each cell which keep a count of how many other cells are linked to it. When the tag value is zero, then it is no longer required, and may be returned to the free-list. This algorithm is simple to understand and to implement, and so is very common in many systems. There are several variants of the *copying* type of garbage collection algorithm, but Baker's [61] is probably the most popular in use today. In this case, there is no free-list of unused cells. The memory space is divided into two halves. Normal dynamic memory allocation is employed in one half until all the space has been exhausted. At this point, the root cells, and all cells which are linked to them, are copied into the other half, and the unused cells left behind. Normal processing now utilises the second half, and so on. Wong [62] describes an Intelligent Cell Memory System for real time engineering applications, which employs a variant of Baker's method. The main drawback with such algorithms is the requirement for twice the amount of usable memory to be incorporated within the system, although compaction techniques can be used to alleviate this problem.

The algorithm used for the SKMS is based on a free-list. Two special pointers are maintained: *mem_ptr* and *free_ptr*. mem_ptr initially points to the start of memory and is incremented when the first cell is allocated. free_ptr is initially NULL and the address of the first cell to be deleted is copied into it. If any subsequent cell is deleted, then it is set to point to the current cell in free_ptr, and becomes the new value of free_ptr itself. If any subsequent cells are created, then free_ptr is tested; if it is not NULL, then the first cell in the free_list is allocated. Otherwise, the cell pointed to by mem_ptr is used, and mem_ptr incremented. If mem_ptr points to the end of memory, then we have no free space left and the operation will fail. Since the garbage collection algorithm is such that the free-list is updated concurrently, then system performance can be predicted. This is an important ability, since it ensures that no unpredicted bursts of garbage collection will occur, which would otherwise rule out operation as a real-time system.

3.2. Functional Specification

The SKMS functional specification (presented below) was specified in consultation with the Artificial Intelligence Applications Institute (AIAI – University of Edinburgh [17]). It defines the operations which manipulate the knowledge base, using the structure described above. These operations, therefore, are design requirements for the hardware implementation of the Knowledge Based system. Note that it incorporates the concept of contexts, discussed in Chapter 2. The knowledge base can notionally be accessed using a functional statement of the form:

function(argument 1, argument 2, ...)

for KB manipulations or:

?function(argument 1, argument 2, ...)

for information retrieval.

In the following specification, a functional syntax is used to illustrate the sort of commands which would be input to the knowledge system. Basically, these commands consist of a function name (such as *create*) followed by a series of arguments, which may be either a full or partial specification of a relation (*object, property, value, context, status information*), individual components of a relation, or new components to be substituted in place of existing ones. The mnemonics chosen to describe the arguments are easily interpreted. The *delete, modify* and *retrieve* functions may all be partially defined. In these cases, a wildcard (*) would be entered in the appropriate argument position. Arguments which have default values need not be entered (unless a value other than the default is desired). These are denoted by including them between square brackets. The context facility is incorporated to support hypothetical and temporal reasoning (see Chapter 2), and the *current context* is stored to keep a record of the context in which we are working.

3.2.1. Manipulation Of The KB

(i) Objects and object slots can be **created** at any time. If a slot is given no value, then it is either inherited from a parent (see Inheritance below) or it is given the value "undefined" (not to be confused with the boolean value). A *mask* flag may be set to prevent slots from being inherited by their sub-classes, if so desired. Unless stated otherwise, the *confidence* associated with a relation is *true* and the context is the current context (see below).

create(Obj_Name, Prop_Name, Value, [Confidence], [Context], [mask])

(ii) Objects and slots can be **deleted** from the KB at any time. If it is required to delete an entire object, despite the fact that other objects may be related to it, then the appropriate relations (links) are deleted from the other objects also. Again, unless stated otherwise, the context is assumed to be the current context. If the context field is a wildcard (*), then the slot is deleted for ALL contexts.

delete(Obj_Name, Prop_Name, Value, [Context])

(iii) It is possible to **modify** the values of slots either in the current context or a specified one.

modify(Obj_Name, Prop_Name, New_Value, [Context])

(iv) Similarly, boolean values of relations (confidences) can be modified.

modify_conf(Obj_Name, Prop_Name, Confidence, [Context])

(v) New contexts can be created at any time. These will be children of the current context.

create_ctxt(New_Context)

(vi) It is possible to change from one context to another.

set_curr_ctxt(New_Context)

(vii) Contexts can be deleted. If the particular context has any children, these are also deleted.

delete_ctxt(Context)

(viii) The Root context can be overwritten by another context. All the contexts in between are deleted. See figure 3.4.

root_ctxt(Context)

(ix) Two contexts can be merged to form a third context which is a child of each of them. To avoid conflict where a slot might have different values in each of the contexts, the value is taken to be the one supplied by the context which is placed first in the argument list.

merge_ctxt(Context_1, Context_2, New_Context)

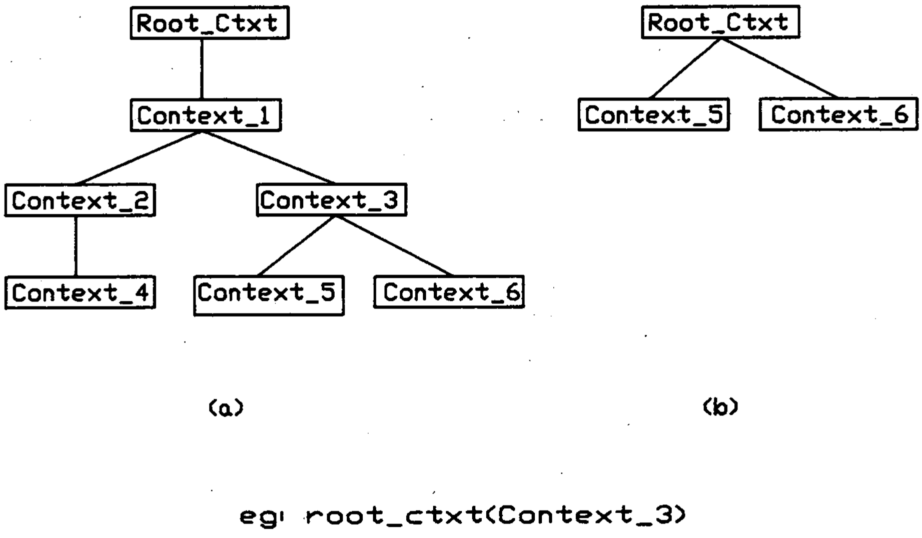


Figure 3.4: An example of a root_ctxt operation

3.2.2. Higher Order Relations

Consider the following information.

"John thinks that Steve is sometimes lazy"

We cannot represent this information using a simple, *first order* 3-place relation, since "John thinks something" and "Steve is lazy" require two separate first order relations to describe them. It is necessary to use a higher order representation instead (in this case second order). It is possible to build up higher order relations using the knowledge structure described above, since the value of a property may be a link to another relation; achieved using tag and status bits and a pointer mechanism inherent within the structure, but invisible to the user. How the layout would look to a user is illustrated in figure 3.5.

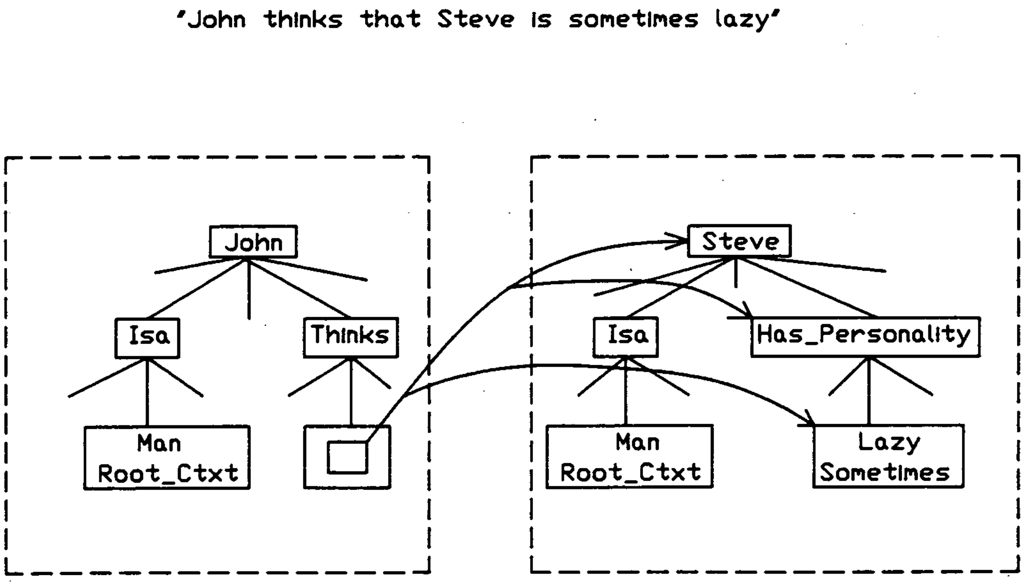


Figure 3.5: An example of a higher order relation

3.2.3. Inheritance

When a new object is created, it inherits all of the unmasked slots from its parent in the particular context. Values are inherited by an object from a parent only if they are not specified already by the user, or they have not been inherited from another parent. Consider the following functional statements made to the system:

create(My_Chair, Instance_Of, Stool, NIL,,)

create(My_Chair, Instance_Of, Chair, NIL,,)

If we suppose that both of the parents contain the property *Has_Legs*, then *My_Chair* will inherit the value supplied by the object *Stool* (namely, three). Clearly, the user can take advantage of this by defining the more specific parents first.

3.2.4. Retrieval

(i) It is possible to retrieve any relation(s) from the KB. A "*" in the place of one of the arguments is treated as a wildcard.

?(Obj_Name, Prop_Name, Value, Confidence, Context)

Note that *?(*,*,*,*,*)* will return all relations in the KB.

(ii) It is possible to return all the objects which meet the following specifications:

(a) *?not(Prop_Name, Value, Confidence, Context)*

(b) *?or((Prop_Name1, Value1, Confidence1, Context1),*
(Prop_Name2, Value2, Confidence2, Context2),...)

(c) *?and((Prop_Name1, Value1, Confidence1, Context1),*
(Prop_Name2, Value2, Confidence2, Context2),...)

Wildcards (*) may be substituted in place of any of the arguments.

3.3. Software Simulation

3.3.1. Introduction

Having defined a functional specification for the system, a software package was written, which would simulate the proposed ideas, and highlight any operations which could be executed directly by, or supported by, special purpose hardware. The simulation package was written in C [79] in a UNIX [80] environment, since C provides an excellent hardware interface.

The UNIX *profiling* facility was used to determine what percentage of CPU time was spent in performing specific operations, and how many times each operation was performed. This information was then used to pinpoint those operations which impair system performance, and hence require particular investigation.

3.3.2. Design Considerations

Since the primary objective was to produce a low-cost plug-in enhancement system for either a SUN workstation or Personal Computer, a decision had to be made quite early as to what development tools and equipment should be used. Due to the availability of several SUN workstations and mainframe systems (all running UNIX), it was decided to target the system as a co-processor to a SUN, interfaced via a VMEbus. In view of this, the software was constructed in two independent units. The first unit, known as the HOST program, simply interfaced with a user, parsing manipulation commands and retrieval requests (as described in the functional specification), and maintaining a *hash-table* of the input strings. The second unit, the MANIPULATOR program, performed the manipulation and retrieval operations on an area of memory known as the Knowledge Base. The interface between the two units was implemented via a communications mailbox, which simulated an area of dual-ported RAM (an area of RAM which can be accessed independently by two different processors via two sets of data and address buses). The HOST program wrote the command and the appropriate coded arguments into the mailbox, and then set a poll bit. The MANIPULATOR program was divided into several sub-units; each responsible for a different operation defined in the functional specification. A central control routine polled the HOST program, and then assigned the task to the appropriate sub-unit. Any returned data and status information was written into the mailbox by the MANIPULATOR controller and then the poll bit cleared to alert the HOST program. The programs were written in a modular fashion to facilitate modification, and interpretation of the UNIX profile information. This methodology had the added advantage of reducing compilation time, and hence speeding up the development time. Global variables were used to store parameters where it was felt that a dedicated hardware register would be of value. A block of memory (512 kbytes) was reserved for the knowledge base, and cells were allocated by the MANIPULATOR program as required. The free-list based garbage collection algorithm, described above, was also implemented.

3.3.3. Performance Limitations

All of the operations defined in the functional specification can be described in terms of four basic primitives.

(i) *create* (or insert) a specified relation

(ii) *modify* a specified relation

(iii) *delete* a specified relation

(iv) *retrieve* a specified relation

Therefore, to simplify matters, a less complex version of the simulation package was written, which performed just these operations. The first three primitives tend to be interactive with the user, and so speed of operation is not generally critical. The retrieval operations, however, are usually the crux of a knowledge based system, and so should execute as quickly as possible. With this in mind, detailed timing calculations were performed on the retrieval operations only. The UNIX profiling information was used to determine how many times each routine was called, so that an overall picture of the time spent at each task could be derived. Having located the problem areas, more specific timing calculations were performed to determine how long a 680X0 family processor would take to execute particular areas of code. This involved the (optimised) cross-compilation of the manipulation software from C into Motorola format 68010 assembly code to calculate the number of machine cycles required to perform the retrieval algorithms (calculated using the Motorola 68000 Reference Manual [63]).

As anticipated, a large percentage of this time was spent in traversing the linked-lists (approximately 1% increase in search time per link traversed). A second limitation was also identified – an average of 50% of total cpu time was spent performing hashing related instructions. However, the more complicated the search, the lower the *proportional* amount of time spent hashing. The projected simulation performances are presented and discussed in detail in Chapter 5, however, the limitations encountered by these calculations have great relevance with respect to the hardware design considerations.

In conclusion, therefore, there are two major performance bottlenecks associated with this system. Namely, linked list traversal and hashing. The next two sections examine these limitations and attempt to discover means to circumvent them. This leads to a discussion of the hardware design considerations arising from a review of the material so far amassed.

3.3.4. Hash Coding

Any user interactive system must include a string to look-up-code conversion mechanism to interface between the real world and the internal knowledge representation. Hash coding of strings is such a technique [64,65,66,67,68,69]. Unfortunately, hashing was found to be a severe limitation on the performance of the simulation package, and it was hoped that suitable hardware support could be designed to provide speed improvements. Consequently, a brief study was made of various hashing, or data conversion techniques.

The most common string to code conversion methods are based on mathematical functions which take as input a string *(key)* and output a unique code within a specific range. Unfortunately, it is very difficult to guarantee the uniqueness of a code since it is often necessary to encode large keys within fixed size conversion tables. Therefore, the function must spread the key codes as much as possible within the range available to avoid giving different keys the same code; a problem known as *collision*. This technique of randomisation is known as hashing and the key code is known as the hash-code. Unfortunately, it is impossible to avoid collisions for a large number of strings (the birthday paradox [64]) and so we must develop methods of safeguarding our data.

Hash-coding is a problem which is really associated with the user interface. Since this is a common limitation, and not specific to this project, then it is safe to ignore for the present, as long as the hashing algorithms are not integrated to such an extent that it is too difficult to modify them. To this end, the hash-coder has been designed as an independent module, which accepts a string and returns the appropriate code, and vice versa. A number of techniques have been developed to alleviate the problems, and could be employed successfully within the SKMS. Appendix A summarises the more common hashing techniques in existence, and the method used for the purposes of this thesis.

3.3.5. List Traversal

Unlike hashing; list traversal is a problem which is inherent in the functional primitives which comprise the system. It is towards this problem, that any hardware improvements should be directed. To attain an insight into the problem, it is first necessary to examine the tree search algorithm used in the MANIPULATOR program.

```
1.  Retrieve the first property block of the current object from the KB at address *1st_Prop. This address is stored in the HOST hash table and is supplied via the communications mailbox. The property block contains the property name (hash-code), status information, a pointer to the next property in this list, *next_prop, (which is the next branch, or sibling, in a general tree representation), and a pointer to the first value block, *1st_val, (which is the first daughter in a general tree representation) associated with this property.

2.  REPEAT
    (a) Compare the property code and status with the HOST specifications.
    (b) IF (codes match)

        [i]  Retrieve the first value block from the address *1st_Val (supplied by the property block). The value block contains the value name (hash-code), status information, a pointer to the next value in this list, *next_val, (which is the next branch in a general tree representation), and the context name (hash-code).

        [ii] REPEAT

             1)  Compare the value code, context code and status with the HOST specifications.
             2)  IF (codes match)
```

```
                a)  Set a success flag in the mailbox to inform the HOST of a
                    successful search.
                b)  EXIT
            3)  ELSE
                a)  Retrieve the next value block from *next_val (supplied by
                    the value block).
            4)  ENDIF
        [iii] UNTIL (*next_val is nil)
    (c) ENDIF
    (d) Retrieve the next property block from address *next_prop (supplied by the
        property block).
3.  UNTIL (*next_prop is nil)
4.  Set a failure flag in the mailbox to inform the HOST of an unsuccessful search.
5.  EXIT
```

The search algorithm, above, illustrates the basic principle that:

> *A code is compared with the specification; if the match fails, then get the next code along this list; if the match passes get the first code in the "daughter list" and match that against the appropriate specification, etc. The search fails when no match has been found by the time we reach the end of a list (ie: the link pointer is NULL).*

Clearly, there are two matches being attempted:

- does the code match the specification?
- is the link pointer NULL?

Here we have an obvious opportunity to reduce search time, by performing both of these matches simultaneously. Another possible improvement follows from a less clear observation. *In general, if we search a knowledge base for a particular relation, no matter how it is structured, probability theory dictates that we will find more failures than successes.* Consider, then, that we are attempting to match a particular property with the specification property. If we are processing the codes and pointers in parallel, then (referring to figure 3.3) we will have at our disposal either the property-name or status information, and either the pointer to the next property or the pointer to the first value in the daughter list; depending on which we decide to process. If we take the pessimistic view that the match, in most cases, will fail, then it would be advantageous to be holding the pointer to the next property, so that we could already be retrieving it while the match is taking place. If it turns out that the match was successful, then we can discard the next property and retrieve the first associated value. These ideas, possible architectures to support them, and the advantages and disadvantages which follow on from them, are discussed in the next section.

3.4. Hardware Design Considerations

The previous section pinpointed two major limitations in the system. The first, hashing, is a common problem associated with the HOST Man Machine Interface (MMI), and does not come under the scope of this project, although a possible improvement is described in Chapter 6. The second limitation is related to linked-list traversal of the tree-based knowledge structure, and some improvements were discussed, briefly, above. Since tree-traversal forms the main part of the four functional primitives (create, modify, delete and retrieve), then only those primitives need be supported by special purpose hardware. All higher level commands and interpretations can be executed by a HOST processor, as any speed gains would be comparatively small and not worth the development cost.

The special purpose manipulation hardware has to fulfill several demands if it is to be a viable alternative to standard Von Neumann processing by, say, an MC680X0 processor:

- low cost
- exploit parallelism → increased speed
- support large knowledge base
- easy to program

Several design approaches were examined. The first option would involve the use of an advanced Von Neumann type processor as a co-processing element to our host system. The Am29000 was considered (its features were discussed in Section 2.5.2.) The main drawbacks with such a design are the cost and the design complexity. It is also difficult to exploit the opportunities for parallelism, which are inherent within the knowledge structure. Another option worth considering is to retrieve all of the codes and pointers, associated with either a property or value block, simultaneously (eg: prop_name, *next_prop, prop_status, and *1st_val - [figure 3.3]). Four processing elements could then deal with all our information in parallel. The advantage of this method lies in its high functionality, since the scope to introduce more complex primitives into the support system is great. Again, the major drawback with this method is the expense, and design and construction difficulties involved with the complexity of that amount of parallel processing. In particular, the management problems of a parallel read/write of all four entities in a property or value block, or the timing control of interleaved memory access of two by two entities, was thought too complex and expensive to merit the modest projected improvement in performance. This follows from the premise, discussed earlier, that more failed matches will be retrieved and examined than successful ones, and so it is only really worth retrieving the pointer to the next sibling in an object tree, rather then any more information regarding the current entity or any of its children (if it is a property). Thus, we could design a system with only two processing elements – one for list codes, and another for list pointers. Such a design, involving bit-slice processors, would seem to be an attractive development option, since they are fast, expandable and easy to program. Figure 3.6 illustrates an appropriate architecture, comprising two processing elements (eg. Am2901), under the control of a sequencer (eg. Am2910A).

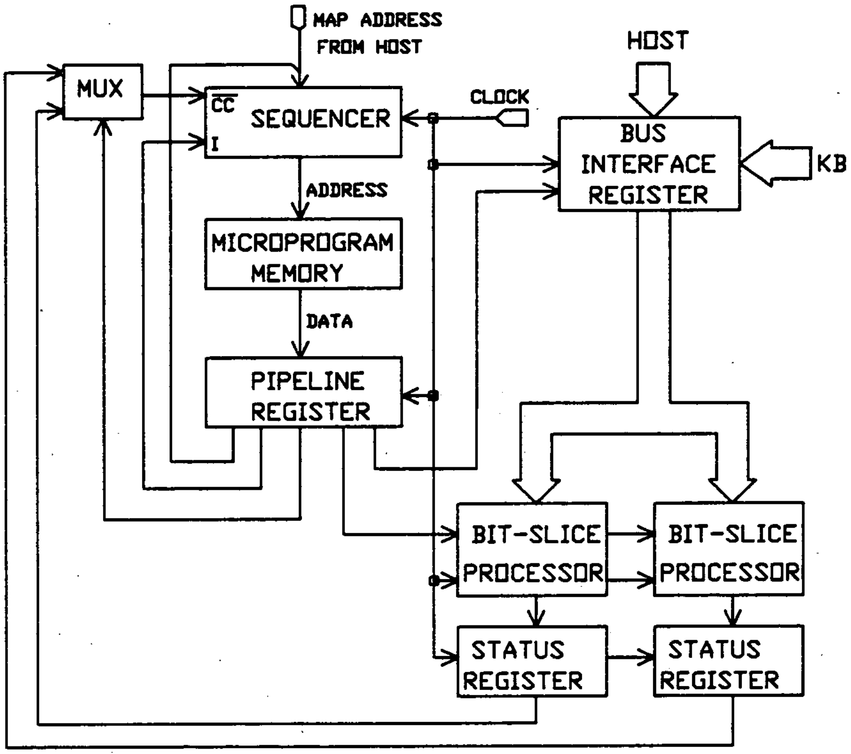


Figure 3.6: Typical bit-slice architecture.

The drawback becomes clear when we try to interface with a large knowledge base. To maintain a large number of different strings and be able to address a large enough storage space, we require large codes and pointers (eg. 32 bits). This would involve several processing elements and associated peripherals, which would entail a fair amount of expense and design complexity. Although this is not a major impediment, and certainly not as great a problem as that associated with an Am29000 based design, it should be borne in mind. The Am29300 32-bit processor family [70] is based around the Am2900 bit-slice family, and reduces the design complexity for large applications. However, the cost of these devices far outweighs the advantage of increased functionality. Furthermore, the system is restricted to the instruction set specified by the processor incorporated into the architecture.

Since the basic operation is the search-and-match and the retrieval algorithms are data-dependent, a system could be designed comprising ICs performing low-level functions, such as comparators and multiplexers, but no processing elements. This design would certainly pose complexity problems, but there are several advantages:

- almost total design freedom encourages exploitation of inherent opportunities for parallelism
- extremely low cost in comparison to previous proposals
- easy to fabricate thus further reducing the cost
- fabrication opens the door for further parallelism at the system level

For these reasons, this design option was selected and figure 3.7 illustrates the general design concept. Sequencing of the architecture is performed in the same way as a typical bit-slice processor application. The difference lies in the substitution of the processors by low-level functional blocks designed to manipulate the knowledge structure in the most efficient manner and exploit the opportunities for parallelism. Moreover, since the design is not restricted by the instruction set of the bit-slice (or other) processor, additional features may be supported, such as dedicated hardware to implement relational algebraic (set) operations on the knowledge base.

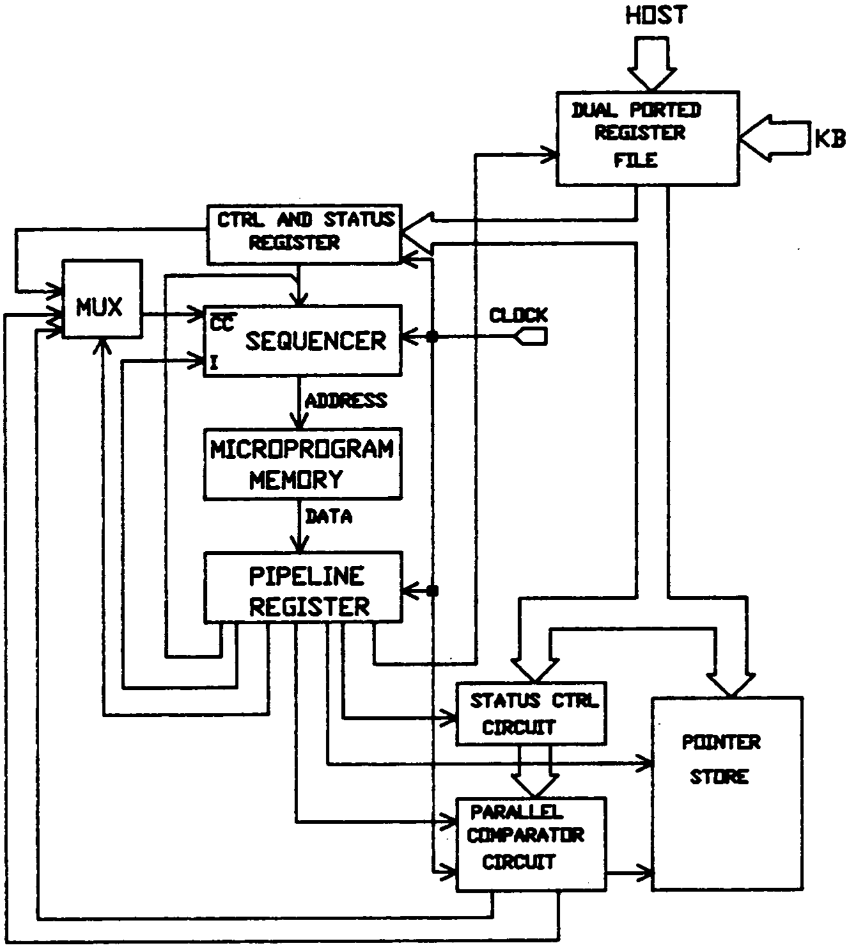


Figure 3.7: The Structured Knowledge Manipulation System Architecture.

The next phase in the design stage related to the construction of a prototype. There are two main considerations at this stage, which are in fact related.

1. What word and pointer sizes do we use (8, 16 or 32-bit)?

2. What type of memory do we use for the Knowledge Base (static or dynamic)?

There are certain criteria to be considered when deciding what form the memory design should take; some are general points, others are specific to the whole system in question.

□ General Points:

- cost
- power considerations
- physical dimensions
- memory-support device requirements
- ease of construction
- chip availability

□ Specific Points:

- data bus size
- access time
- number of "words" required

Designs involving dynamic RAMs have the main advantage of being cheaper and much smaller per byte than static RAMs; dynamic devices are available as 256k x 8bit modules for approximately £30 whereas static RAMs of 128k x 8bit modules cost about £90†. They also draw much less current than static RAMs while in the unselected state.

Unfortunately, since dynamic RAMs need to be constantly refreshed; this increases the number of support devices required and so adds complexity to the design. Moreover, no address decoding is provided on dynamic RAMs, and must be done off chip. This introduces further complexity into the timing, since the memory cells are organised in a row and column lattice which require to be accessed first by row and then by column. This explains why cycle times for static RAMs are generally much faster than for dynamic devices. Address decoding and memory refreshing can be

† Prices quoted by Hitachi suppliers in January 1989

simplified by the inclusion of DRAM controllers and timers. Such controllers have a maximum drive capability and so several may be required to support very wide data.

This leads us to the question of word size. Again, there are several considerations. The most important one is the dependence on the desired size of the Knowledge Base. Table 3.1 illustrates the relationship.

Word Size	**No. Of Strings Supported**	**Maximum Memory Supported**
8 bits	256	256 words
16 bits	65536	64 kilowords
32 bits	$> 4 \times 10^9$	4 Gigawords

Table 3.1: Relationship between word size, string support capability and memory support capability.

Referring to figure 3.3, each property and value block comprises 4 words. Therefore, a relation with one property and one value would require 8 words to define it†. On average, the number of words required per relation would be lower. We can see from the table that an 8 bit system would clearly be inappropriate, since our knowledge system would be able to support only 256 different strings in a memory space of only 256 words (32 relations in the worst case). Although a 16 bit system could support an adequate number of strings, the memory space may too small to be an effective Knowledge Base (8k relations). Clearly, a 32 bit system is very attractive. However, due to cost, complexity and time constraints, it was decided to build a 16 bit prototype with a static RAM Knowledge Base. Moreover, a 16 bit word size proved convenient, since the Am29334 4-port dual-access Register File (64 words by 18 bits wide) can be used to implement the communications mailbox simulated between the

† Note, however, that 8 words per relation is the worst case situation, which would occur only if no relations in the knowledge base were related to each other at all!

HOST and MANIPULATOR programs.

Although the memory size of a 16 bit system severely limits the applicability, it was felt that it was adequate for prototype demonstration purposes. A 32 bit enhancement, based on DRAM, would not create any major difficulties other than some increase in complexity and cost, although this upgrade would best be introduced during a fabrication design phase.

A more detailed discussion of specific hardware design considerations, construction and operation is presented in the next chapter.

CHAPTER 4

Hardware Design And Construction

4.1. Introduction

Chapter 3 discussed the software simulation of the Structured Knowledge Manipulation System (SKMS), and various proposals for hardware support. This chapter describes the design of an SKMS hardware prototype, and the development of the control software. Figure 4.1 is a block diagram of the entire system, illustrating the functional blocks and their interfaces. The prototype system consists of a terminal and host CPU (*HOST*), which performs the user interface and maintains the hash table, and is interfaced to each module of the SKMS via a VME bus. The hardware support comprises a Knowledge Base (*KB*), interfaced to a Relational Processing Unit (*RPU*) via a LOCAL bus, and programmed through a Microprogram Store (*MPS*).

The RPU functions as a 16 bit co-processing unit to the HOST, and manipulates information in the KB using the knowledge structure defined in Chapter 3. This structure is invisible to the HOST. An area of four-ported, dual access RAM in the RPU acts as a communications mailbox for parameter passing.

As far as possible, the mnemonics chosen for data, address and control lines in the following circuit descriptions are self-explanatory, and remain consistent. For example, VME address lines are of the form VA_i, and the Microprogram Store addresses are of the form μA_i. Signals which are active low, or are complemented, are "bar-ed", for example $\overline{DTACK}$ and $\overline{P=Q}$. Appendix B provides a list of *all* signals used in the design – their mnemonics and a brief description.

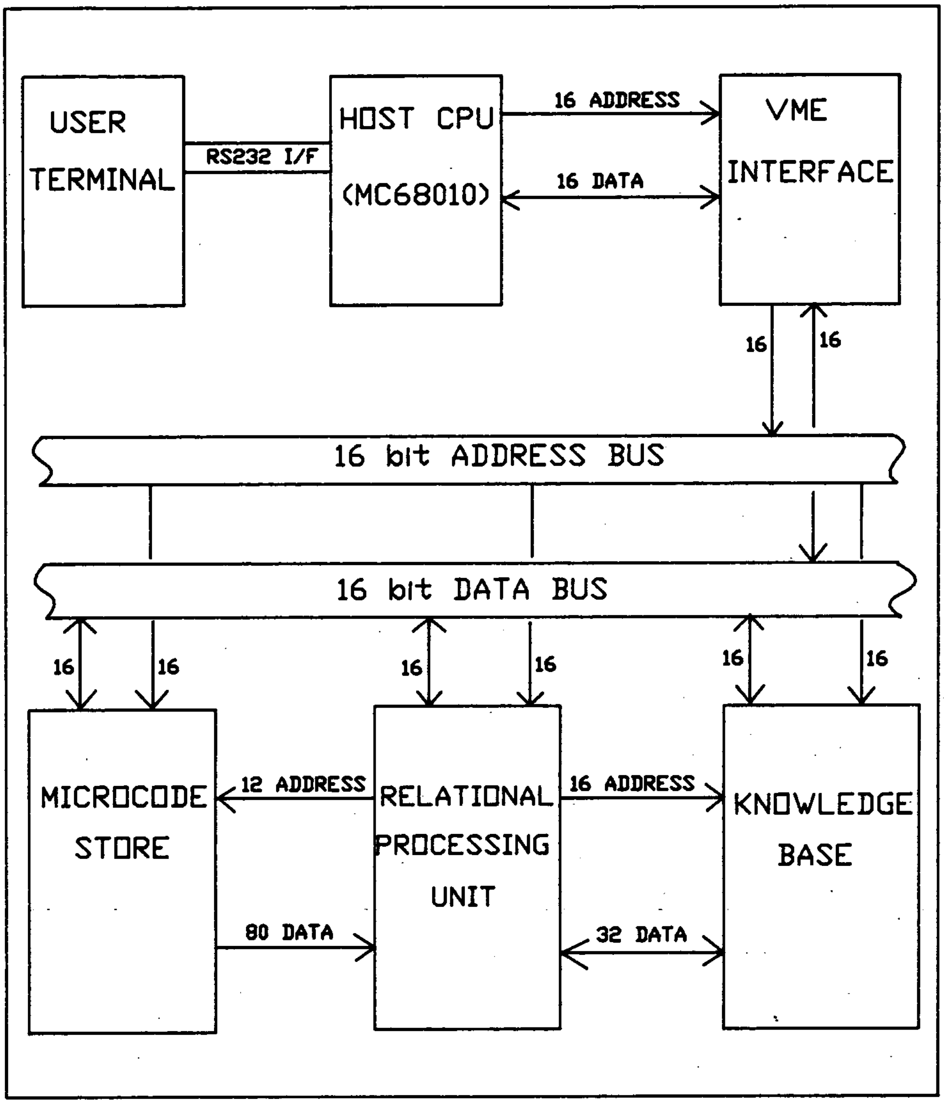


Figure 4.1: The Structured Knowledge Manipulation System – Block Diagram

4.2. The VME Interface

There are three communications paths from the HOST to the support system:

(i) The *Microprogram Store* may be written to and read by the HOST. Since we want the Relational Processing Unit to be fast, this store consists of 45ns access static RAM chips.

(ii) To allow the HOST to make functional calls to the Relational Processing Unit, a *communications mailbox* is incorporated into the manipulation hardware, which may be written to and read by the HOST. Again, fast manipulation hardware requires a fast access mailbox (~30ns).

(iii) The *Knowledge Base* must be accessible by the HOST; both for debugging during development, and to enable the saving and loading of data. Since it is intended to demonstrate that the system performance is based on architectural features, and not memory speed, and to facilitate performance comparisons with the simulation software running on contemporary systems (see Chapter 5), standard static RAM chips with access times of 150ns were utilised.

The VME Interface circuit must be able to support all such communications. In the following discussion, a working knowledge of the VME bus is assumed (see Fischer [71]).

The first thing to note is that each area of RAM has a different minimum access time. Secondly, each area has a different set of dimensions. Since a 16 bit prototype architecture has been adopted; the maximum space addressable by a pointer in the knowledge structure is 64k words, and a property or value block, comprising 4 words (see figure 3.3), is 64 bits wide. Consequently, the Knowledge Base is viewed by the RPU as being 64k words by 64 bits wide. Referring to Section 3.4, a design decision was made to retrieve only 32 bits at a time, since, for the majority of the search time, the system will be retrieving unsuccessful matches. Therefore, a microprogram control bit would be required to select between the least significant and most significant 32 bit segments (see Section 4.3). The Microprogram Store is viewed by the Relational Processing Unit as being 2k words long and 80 bits wide, and the communications mailbox is 64 words long and 16 bits wide. The latter consideration does not pose any major problems. Since the HOST is a 16 bit machine, and the three memory spaces are all multiples of 16 bits in width, it is possible to design an interface which is also 16 bits wide. So, five 16 bit VME transactions are required to write a word to (or read a word from) the Microprogram Store, four for the Knowledge Base, and one for the mailbox. Furthermore, if 8 bit transactions are disallowed, then the circuit design is simplified, since it is no longer necessary to include the $\overline{UDS}$ and $\overline{LDS}$ signals in the address decoding hardware. The former consideration, however, is not so straightforward, since the $\overline{DTACK}$ signal associated with different speed memory spaces would have to be activated at different times. This would require a separate $\overline{DTACK}$ circuit for each memory space. Therefore, it was decided to construct only one such circuit, which assumes that all memory addressed by the HOST has an access time of 150ns (ie: the slowest of the three). The only drawback with this decision is that the communication between HOST and RPU is slower than would otherwise be possible. Since this is only a prototype system, and the interface could be upgraded at a later date, the advantage of simplified construction was considered to outweigh this disadvantage. Moreover, the effect of such a change can be accounted for easily in any timing calculations.

Before designing the circuit, it was necessary to define when the HOST would be allowed to access each memory space, and when the RPU would. Table 4.1 summarises the desired allowable accesses.

MEMORY SPACE	Setup System		Normal Operation		Debugging	
	HOST	*RPU*	*HOST*	*RPU*	*HOST*	*RPU*
K.Base	√	×	√	√	√	√
Microstore	√	×	×	√	√	×
Mailbox	√	×	√	√	√	√

√ ... requires access

× ... don't care

Table 4.1: Allowable memory accesses within the SKMS

Figure 4.2 is a schematic diagram of the VME Interface circuit. Consider the KB first. Except during setup (ie: initialise memory spaces, load microcode etc), both the HOST and the RPU require access. However, the HOST would only require access for loading a saved state, saving the current state, or reading values during debugging. By far the greatest use is made of the KB by the RPU. Consequently, the RPU has control of the KB (via the LOCAL bus) until the HOST (via the VME bus) requires control, whereupon the RPU's operation is suspended. Three control outputs are supplied by the interface circuit: $\overline{HALT}$ is used to suspend RPU operation (see Section 4.5.1) while the HOST accesses the KB. $V/\overline{B}$ controls the HOST←→KB data bus transceivers and address bus buffers, and $\overline{BAE}$ controls the RPU←→KB data transceivers and address bus buffers. The HOST address buffering is provided for within the VME Interface circuit, whereas the RPU addresses are buffered via the tristate outputs of a dual input multiplexer (see figure 4.12). Note that the different ways the HOST and RPU view the KB means that HOST address $VA_{2\text{-}18}$ correspond to RPU addresses $BA_{0\ 16}$ (compare figures 4.2 and 4.12). Since 8 bit transactions are illegal, VME address line 1 (VA_1) is used to determine whether the odd or even address is accessed, and address lines VA_2 and VA_{18} are used in the KB chip select decoder (see Section 4.3). Address lines VA_3 to VA_{17} then map directly to the 15 address lines of the 32k RAM chips (MA_0 to MA_{14} – see Section 4.3), which make up the KB.

Since the HOST does not need to access the MPS during normal operation, both the MPS and Mailbox are mapped into the same logical space and a toggle switch is used to swap between the connections to the VME bus. VME address lines VA_{19} to VA_{23} are decoded by the interface circuit to define the logical memory spaces occupied by the KB, MPS and Mailbox. Since the prototype contains only one CPU, which is always the *master*, the address modifiers (AM_0 to AM_5) are ignored. The address assignments of the KB, MPS and Mailbox are illustrated in figure 4.3.

A shift register, clocked from the VME *sysclk*, is used to delay the memory select line ($\overline{board\_sel}$) which is derived from the HOST address strobe ($\overline{AS}$), and return it to the HOST as the $\overline{DTACK}$.

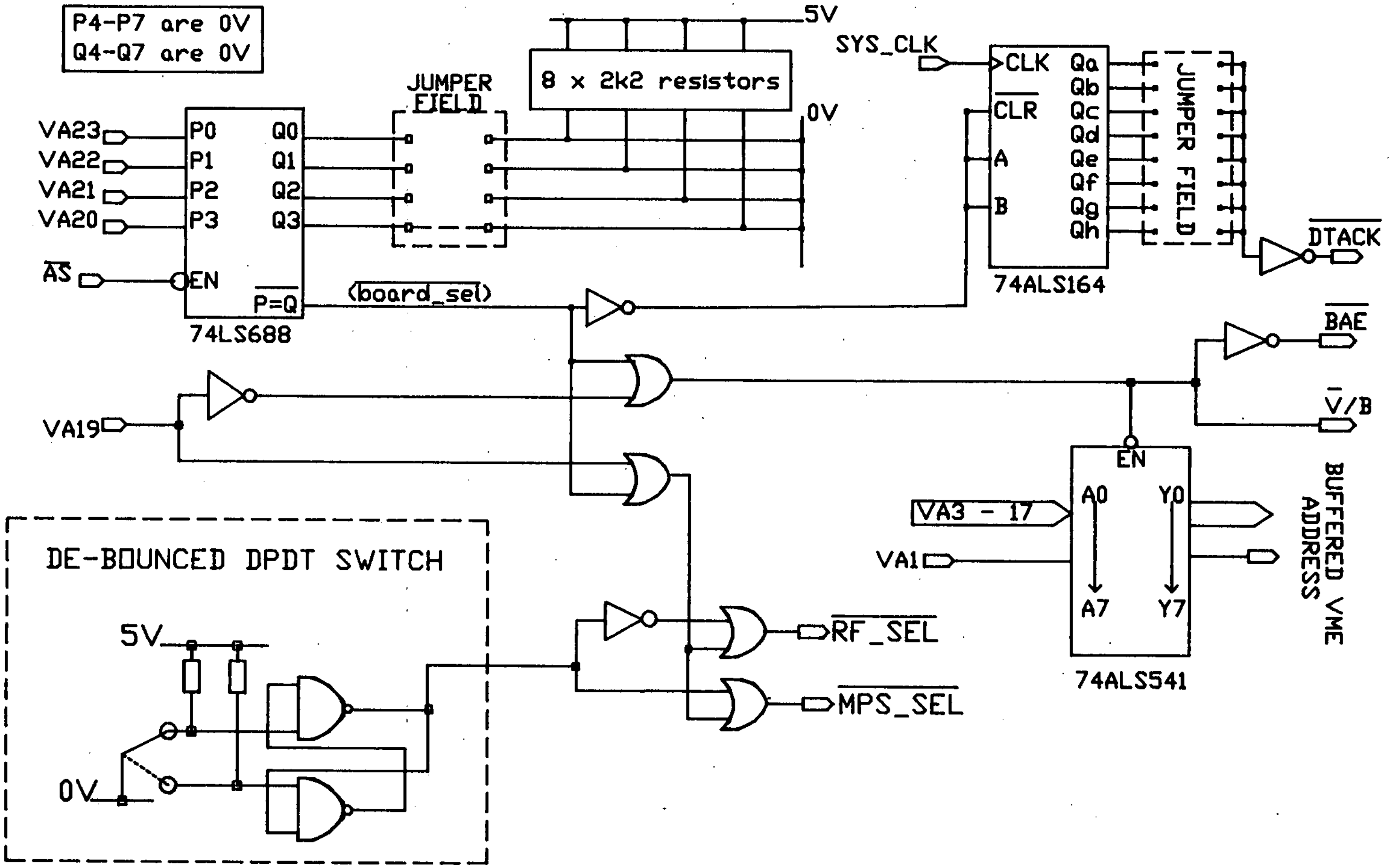


Figure 4.2: The VME Interface Circuit — Schematic Diagram.

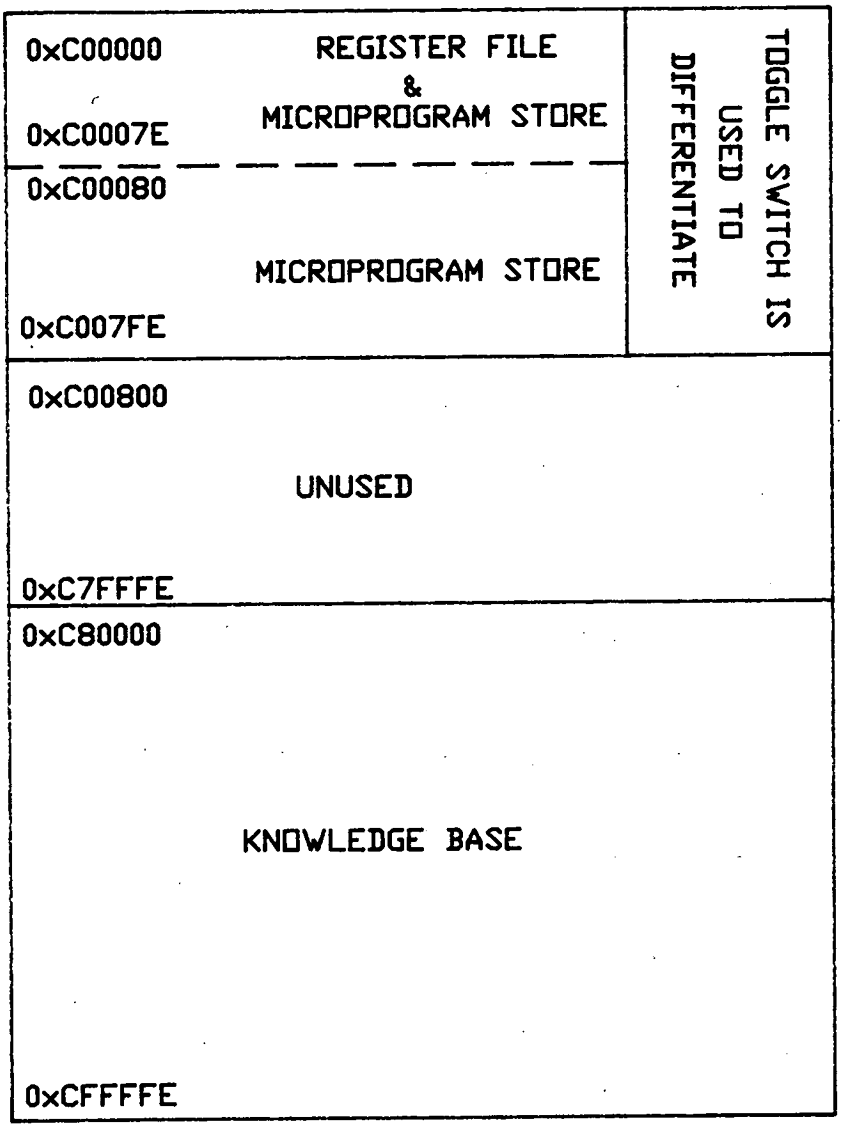


Figure 4.3: The KB, MCS and Mailbox Addresses (from the HOST).

4.3. The Knowledge Base

Figure 4.4 is a schematic diagram of the Knowledge Base (KB). The KB is constructed from standard 32k by 8 bit static RAM chips (D43256C-15L). The manipulation hardware (RPU) views the KB as 64k by 64 bits, and supplies 15 address lines (BA_1 to BA_{15}) which map onto the MA_0 to MA_{14} KB address lines. BA_0 is a microprogram control bit used to determine whether the odd or even 32 bit word is being accessed, and BA_{16} is used to differentiate between the 2 banks of 32k memory devices which make up the 64k words. The HOST views the KB as 256k by 16 bits, and supplies the appropriate address lines (VA_3 to VA_{17}) which map onto MA_0 to MA_{14}. VA_2 determines whether the odd or even 32 bit word is accessed, VA_1 determines whether the odd or even 16 bit word is accessed, and VA_{18} determines which bank of the 32k memory devices is being addressed.

The RPU$\longleftrightarrow$KB (LOCAL bus) data path is maintained by four 8 bit Input/Output Ports (Am2952A), which are controlled by signals from the RPU. Note that the $\overline{BAE}$ signal from the VME Interface circuit is used to disable this path when the HOST assumes control of the KB. The HOST$\longleftrightarrow$KB data path is maintained by four 8 bit bus transceivers (74LS645-1), which are controlled by the VA_1, $\overline{VA_1}$ and $\overline{V}/B$ lines. A quad 1-of-2 multiplexer (74ALS157), controlled by $\overline{BAE}$, is used to switch between the HOST and RPU memory control signals. Memory select decoding is performed by a dual 2-to-4 line decoder (74ALS139), under the control of VA_1 and $\overline{BAE}$.

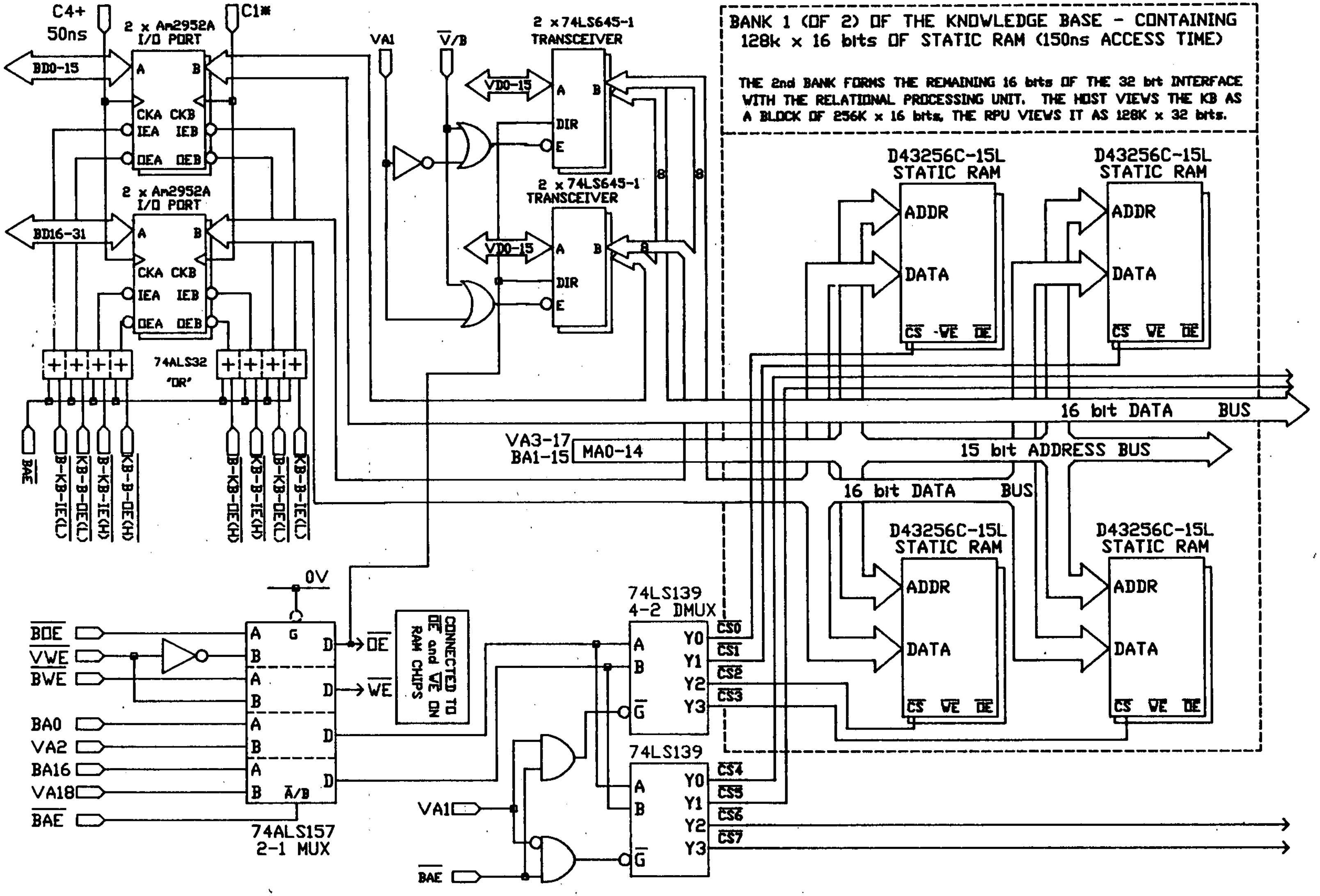


Figure 4.4: The Knowledge Base — Schematic Diagram.

4.4. The Microprogram Store

The Microprogram Store (MPS) is constructed from 2k by 8 bit static RAM chips (CYC128-45PC) and is shown in schematic form in figure 4.5. Since the RPU sequencer has an addressing facility of 4k words, only 11 of the 12 sequencer address lines are utilised. However, these ICs were chosen since preliminary hand-coding studies projected microprogram sizes of less than 512 words, and so the full 4k range was unnecessary. Access times of 45ns were chosen since the fundamental clock period of the Clock Generator (see Section 4.5.1) is 50ns. The memory is grouped into 5 banks of 2 ICs, and associated with each bank are two octal bus transceivers (74LS645-1), which form the 16 bit data interface with the HOST. A toggle switch disables the HOST←→MPS interface and enables the 80 bit data interface with the RPU. A 3-to-8 decoder (74ALS138) generates the appropriate chip select ($\overline{CS}$) signals derived from the HOST address lines VA_1 to VA_3, and 3 quad 2-to-1 multiplexers (74ALS157) are used to switch between the HOST and RPU addresses. Note, that since only five of the $\overline{CS}$ signals are used, corresponding to the five 16 bit memory blocks forming the 80 bit micro-control word, the last three 16 bit words in every eight are not accessible by the HOST.

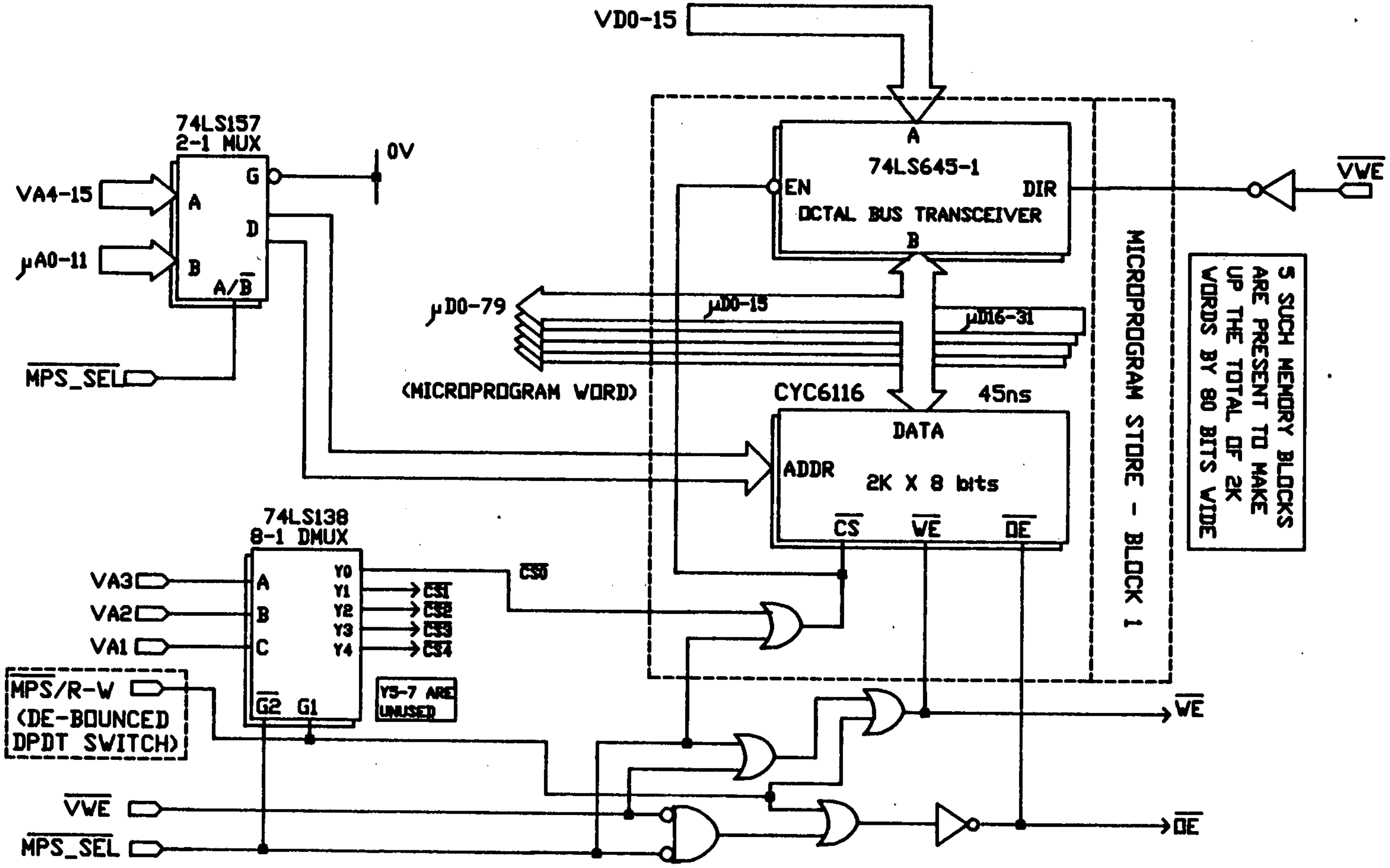


Figure 4.5: The Microprogram Store – Schematic Diagram.

4.5. The Relational Processing Unit

Figure 4.6 is a block diagram of the Relational Processing Unit (RPU). Control of operation is performed by the Sequencer – a toggle switch performs a $\overline{RESET}$ on the Sequencer, which resets its stack pointer and sets the program counter to zero. The Register File, in conjunction with a Control and Status Register (CSR), forms the HOST⟷RPU communications mailbox. Knowledge structures are created in and retrieved from the KB via the LOCAL bus Interface (described in Section 4.3, above). Linked-list pointers and the garbage collection free-list pointers are stored and manipulated by the Pointer Store Circuit, and the Parallel Comparator Circuit performs a dual matching operation – the list codes are compared with the specifications, supplied by the HOST via the Register File, and (simultaneously) the list pointers are compared with zero. The Status Control Circuit filters out non-relevant information from status words within the knowledge structure before comparison with a specification. It also sets or clears a mark bit in the status words to support set operations. The Condition Code Selector decides which test signal is to be used by the Sequencer for conditional operations. The timing of execution of the various sub-components of each operation associated with the different functional blocks which compose the RPU is critical, and a Clock Generator Circuit is used to generate the necessary clock signals.

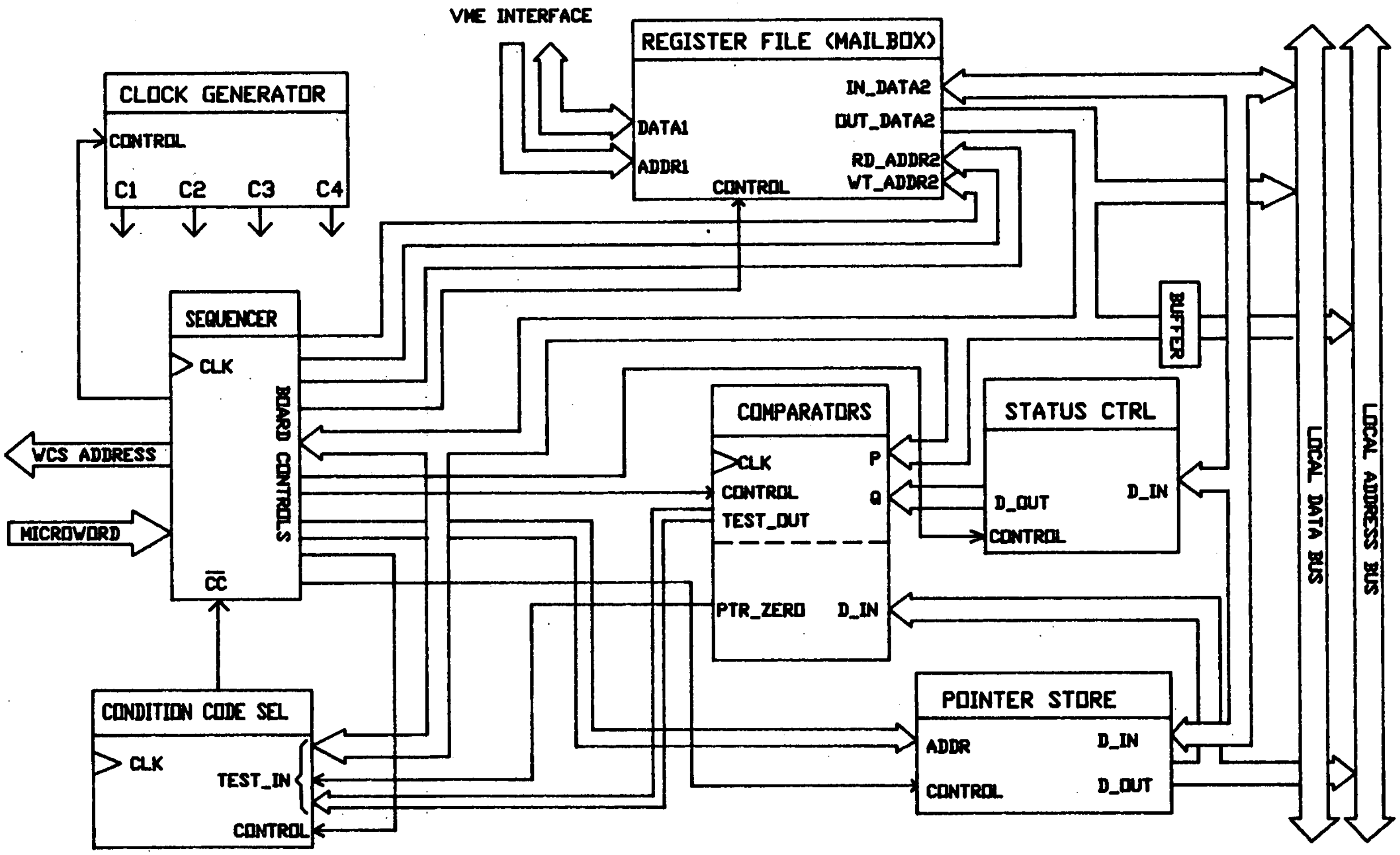


Figure 4.6: The Relational Processing Unit (RPU) – Block Diagram.

4.5.1. The Clock Generator Circuit

The Clock Generator Circuit, shown in schematic form in figure 4.7, is designed around the Am2925 Microprogrammable Clock Generator and Microcycle Length Controller. A 20 MHz crystal is used to create a fundamental clock (F_0) with period 50ns. This is converted into 4 different output waveforms (C_1, C_2, C_3 and C_4), which are used within the RPU. L_1, L_2 and L_3 are supplied by the microprogram to define the cycle length for the *next* microcycle. Figure 4.8 summarises the different clock waveforms available from this circuit. The Am2925 also incorporates two sets of switch debounced inputs to maintain manual RUN/HALT and single-step (SSNC/SSNO) toggle switches. The $\overline{HALT}$ input from the VME Interface Circuit is used, with the wait state control circuit within the Am2925, to suspend operation of the RPU (by "stretching" the clocks) while the HOST is accessing the KB. A shift register, clocked by F_0, is used to delay C4 by increments of 50ns, since C4+50ns and C4+100ns are both required within the RPU. For the same reason, C1 and C2 complemented signals (*C1** and *C2**) are also supplied.

The clocks are buffered by a 74LS77 dual 2-bit transparent latch, to provide extra drive (and hence fan out). The latch is not used to disable the clocks to suspend RPU operation, since it would not be possible to guarantee the *relative* state of the clocks at the latch output once it was re-enabled. This mistake was made at an early stage in the design and corrected by including the wait-state circuit shown in the figure.

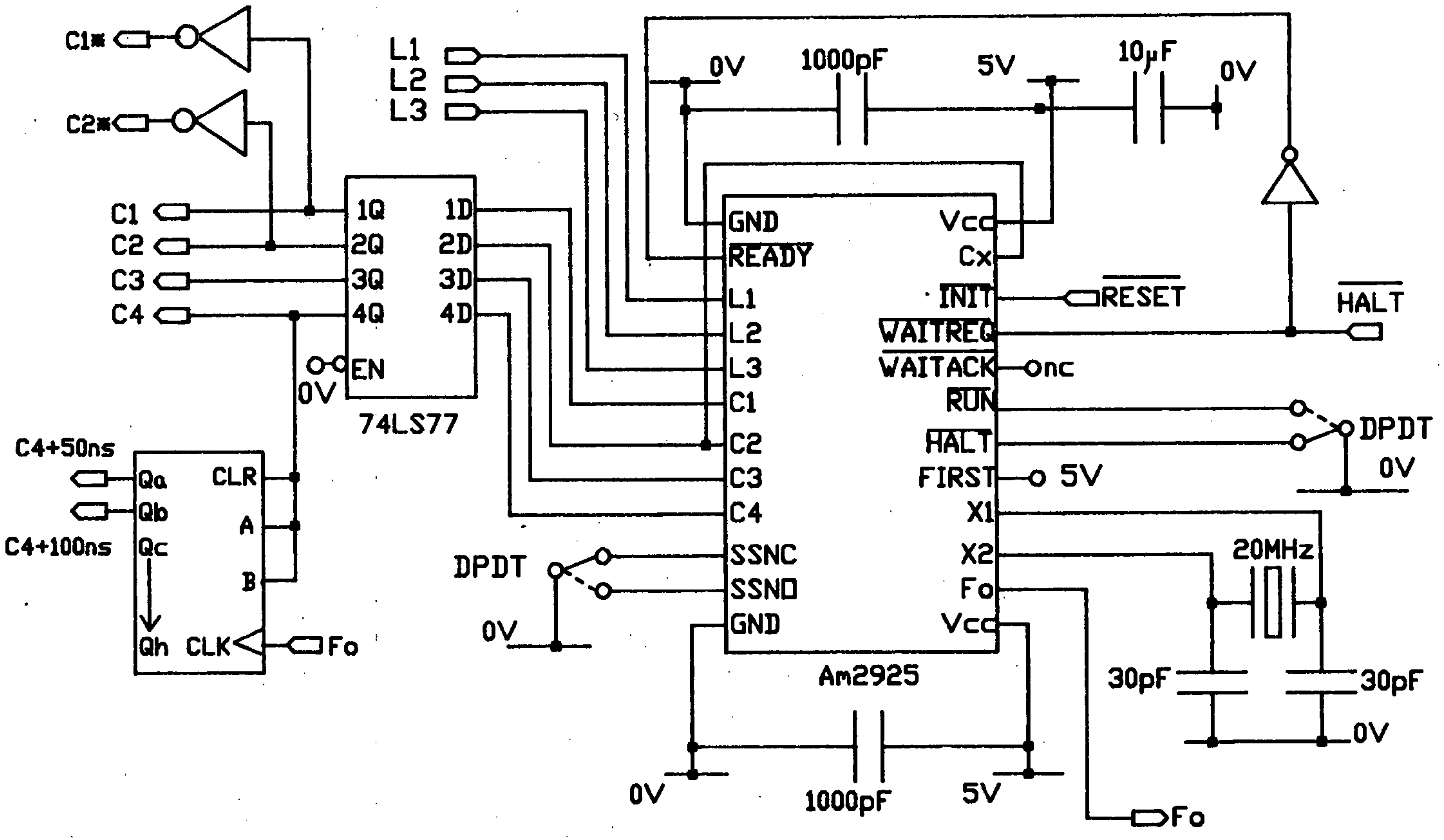


Figure 4.7: The Clock Generator Circuit – Schematic Diagram.

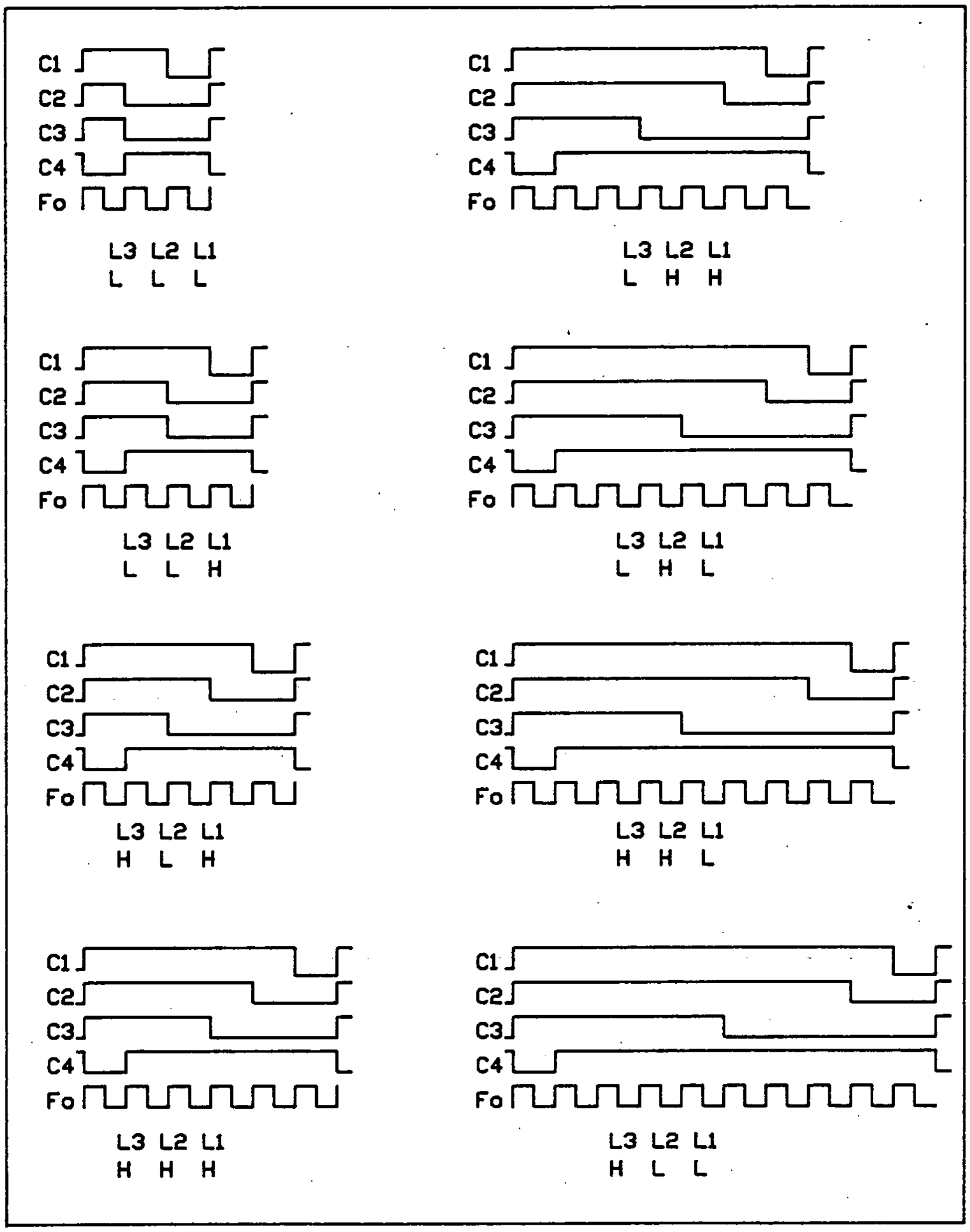


Figure 4.8: The Am2925 Clock Waveforms (adapted from [81])

4.5.2. The Sequencer

The Sequencer Circuit, illustrated in figure 4.9, is based around the Am2910A Microprogram Controller, which accepts a data input at D_0 to D_{11}, and outputs an address for a microprogram store at Y_0 to Y_{11}. It is reset by an external toggle switch ($\overline{RESET}$), and can normally execute 1 of 16 instructions, depending on the state of the instruction inputs I_0 to I_3. A control bit from the microprogram word, and feedback from the Comparator Circuit (see Section 4.5.4), is used to multiplex between two alternative jump addresses (*PLAddrA* and *PLAddrB*) fed through the Pipeline Register from the microprogram control word. Additionally, by permanently disabling the $\overline{VECT}$ input and connecting the $\overline{CCEN}$ input to I_3, only 6 of the defined Am2910A instructions, plus the addressing enhancement, are necessary for the purposes of the SKMS. The $\overline{PL}$ and $\overline{MAP}$ outputs are used to enable the Pipeline Register inputs and the Control and Status Register MAP inputs respectively. The Pipeline feeds jump addresses from the microprogram to the sequencer, and the MAP inputs provide microprogram start addresses (see Section 4.6) from the HOST, via the Mailbox. The microinstructions used to control the sequencer are described in detail in Section 4.6.

The Control and Status Register (*CSR*) comprises two 74ALS874 octal latches, which accept data from the Register File (Mailbox - see Section 4.5.3), and output to the Sequencer and Condition Code Selector circuits. The least significant 4 bits form the MAP address and are connected to D_0 to D_3 and are enabled by the $\overline{MAP}$ signal from the sequencer. The remaining bits are used within the Condition Code Selector (see Section 4.5.7), and are enabled by a microprogram control bit.

The Pipeline Register is built from ten Am25LS2520 octal latches and is used to latch each control word from the Microprogram Store. Appendix C provides a description of the microprogram control word. Since the operation of the sequencer is asynchronous, except for the Program Counter (*PC*) and Stack Pointer, the Y outputs generated from the microinstruction inputs must be latched before the sequencer is clocked. There were several problems associated with this part of the design before the correct relative timings were established – it is essential that the Y output from the Sequencer, generated as a consequence of the pipelined input at D, be latched *before* the Sequencer is clocked, otherwise an incorrect address will be presented to the MPS following a CONT (increment program counter) or RTN (return from subroutine) operation. Furthermore, due to the MPS access time, and propagation delays within the circuit, a minimum clock period of 200ns is required. This restraint, however, could be improved if faster (more expensive!) MPS memory was used, and if a fundamental clock frequency of greater than 20 MHz chosen. The Sequencer Circuit timing specifications are illustrated in figure 4.10 and the Control and Status Register in figure 4.11.

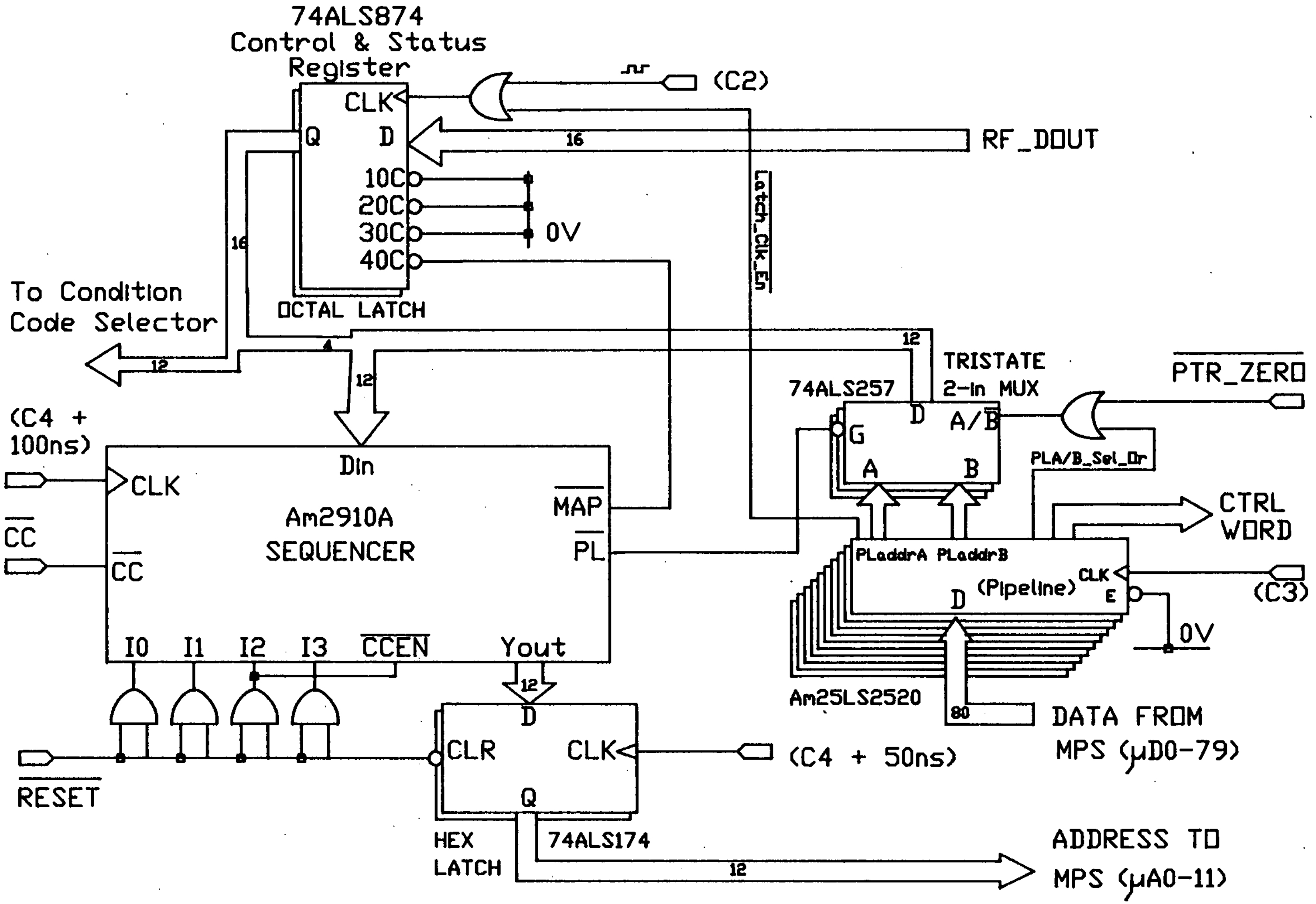
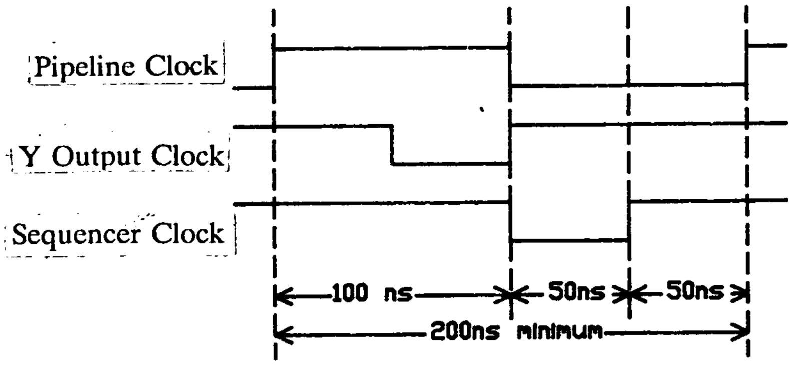


Figure 4.9: Sequencer Circuit – Schematic Diagram



(Assuming that Fo has period 50ns)

Figure 4.10: Sequencer Circuit Timing Specifications

Control And Status Register (CSR)		
Bit No.	**Name**	**Description**
0 → 3	JMAP	Provides the 4 bit JMAP address (op_code)
4	Poll	The poll bit input to the CC MUX
5	1st	The first-rel/all* input to the CCMUX
6 → 7	VCmp	The select i/p for the val-cmp MUX interface between the comparator outputs and the CC MUX val-cmp input. (6=0,7=0)→*LT* ; (6=0,7=1)→*GT* ; (6=1,7=0)→*EQ* ; (6=1,7=1)→$\overline{EQ}$
8	CCmp	The select input for the ctxt-cmp MUX interface; 0→$\overline{EQ}$, 1→*EQ* .
9	PCmp	The select input for the prop-cmp MUX interface; 0→$\overline{EQ}$, 1→*EQ* .
10	PWild	The prop-wild (wildcard) input to the $\overline{CC}$ FAIL MUX
11	VWild	The val-wild input to the $\overline{CC}$ FAIL MUX
12	CWild	The ctxt-wild input to the $\overline{CC}$ FAIL MUX
13 → 15	unused	These bits are unused, but could support further development

Figure 4.11: Control and Status Register

4.5.3. The Register File (Mailbox)

The Register File (RF) forms the RPU↔HOST interface, and is illustrated schematically in figure 4.12. The core component of the circuit is an Am29C334 Four-Port, Dual-Access Register File, which provides high speed storage for both the HOST and the RPU. The hardware design choices associated with this part of the circuit proved complex, since it was essential that the RPU↔HOST interface be as fast as possible, due to the functional requirement to return *all* of the relations which match a partly defined specification, or belong to a specified set. This ruled out the "cheap and easy" option of a shared single access RAM, since this would be slow, and therefore impair performance to such an extent that it is unlikely that the system would be viable in comparison to others available. The Am29C334 is part of the Advanced Micro Devices 29300 family of very high performance ALU and peripherals. With 64 words by 16 bits (plus 2 parity bits, which are unused in this application), relatively low cost, an access time of ~30ns, and an architecture which allows any combination of dual access (two reads, two writes or a read-write), it is ideal for use as a mailbox between the HOST and the RPU. The main drawback lies with the complexity of the pin-out (120 pins) and the consequent circuit construction problems.

The HOST gains access to the File via the VME Interface Circuit – VA_1 to VA_6 select the appropriate address, and $\overline{RF\_SEL}$ enables the two octal bus transceivers (74LS645-1). The microprogram control word supplies two addresses to the File; the *read* address (*BADDR1*) and the *write* address (*BADDR2*). The corresponding data input lines (RF_Din_0 to RF_Din_{15}) are multiplexed between the least significant 16 bits of the 32 bit data from the KB (BD_0 to BD_{15}) and the most significant 16 bits (BD_{16} to BD_{31}). The Register File data outputs (RF_Dout_0 to RF_Dout_{15}) are connected to five functional parts of the circuit (Status Control Circuit, Parallel Comparator Circuit, Pointer Store, LOCAL bus I/O Port and the Control and Status Register in the Sequencer) and also to the LOCAL address bus via a multiplexer. The use of the Register File during system programming is described in more detail in Section 4.6.

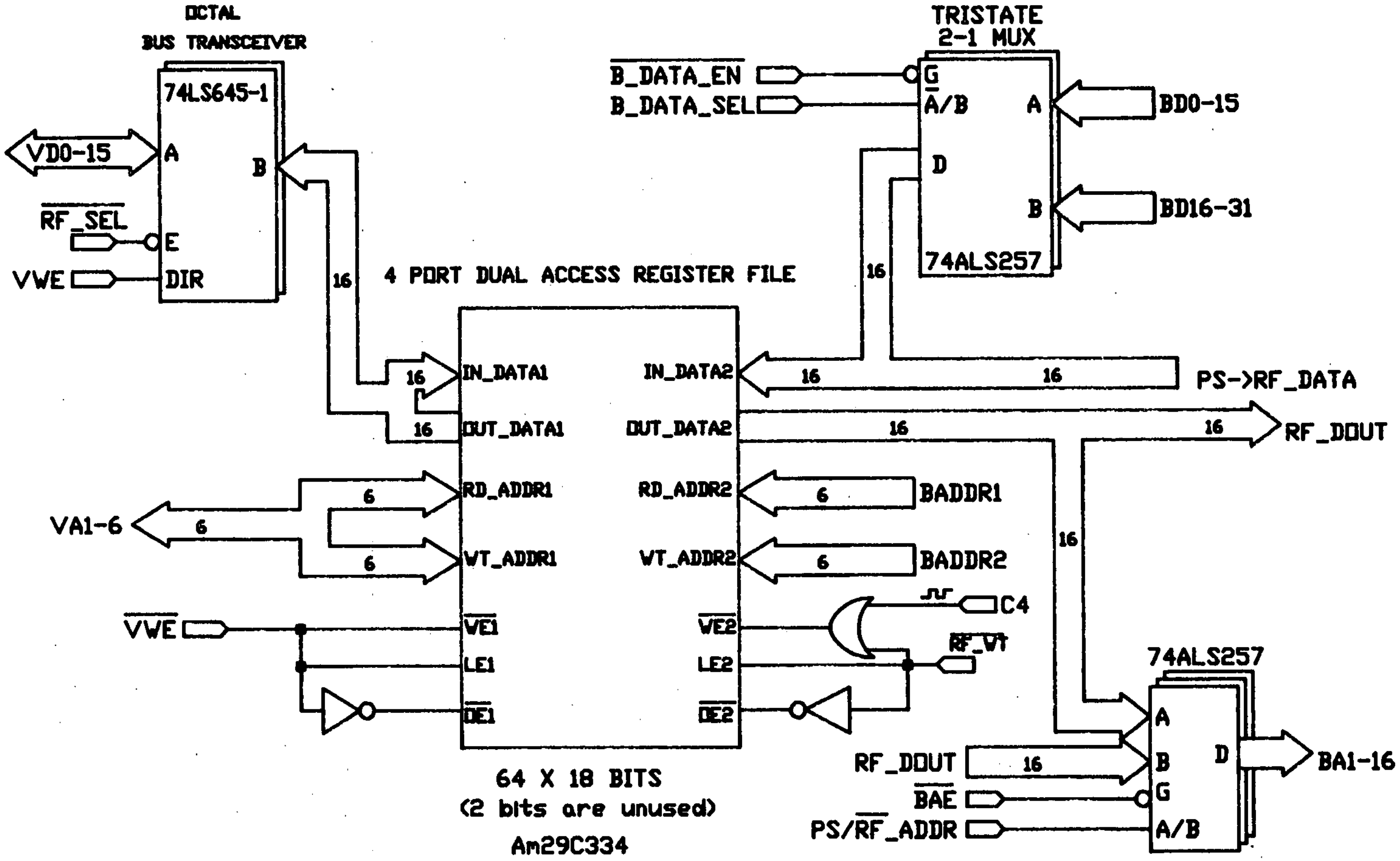


Figure 4.12: Register File (Mailbox) – Schematic Diagram

4.5.4. The Parallel Comparator Circuit

The Parallel Comparator Circuit (PCC) is shown in figure 4.13 and performs two comparisons in parallel. Referring to figure 3.3 in Chapter 3, for each 32 bit word in a property block, and for the least significant 32 bit word in a value block, BD_0 to BD_{15} will contain either a string hash-code or a status word, and BD_{16} to BD_{31} will contain a list pointer. The most significant 32 bits of a value block will contain no pointers, but will contain the hash-code of the context string. Consequently, two octal comparators (74AS866) are used to compare codes or status words with a specification. P is a specification code or status word supplied by the Register File, Q is multiplexed between the least significant I/O Port input word (via the Status Control Circuit) and the most significant input word. The comparator $\overline{P=Q}$, $P>Q$ and $P<Q$ outputs are fed back to the Sequencer via the Condition Code Selector circuit (CCS), via a latch. The complement of $\overline{P=Q}$ ($P=Q$) is also supplied.

The second comparator comprises a series of NOR (7425) and NAND (7420) gates which return a logical zero to the Sequencer via the CCS if the address supplied by the Pointer Store is NULL. This facility has a dual purpose. Firstly, it is used to test whether we are at the end of a linked-list (ie: the list pointer is zero), and secondly, it is used test whether the free-list pointer is NULL during garbage collection (see Section 4.6).

The test signals are fed back to the CCS via a latch, which ensures that the results of a particular test performed in one clock cycle are available (at the sequencer $\overline{CC}$ input) in time for interpretation in the next.

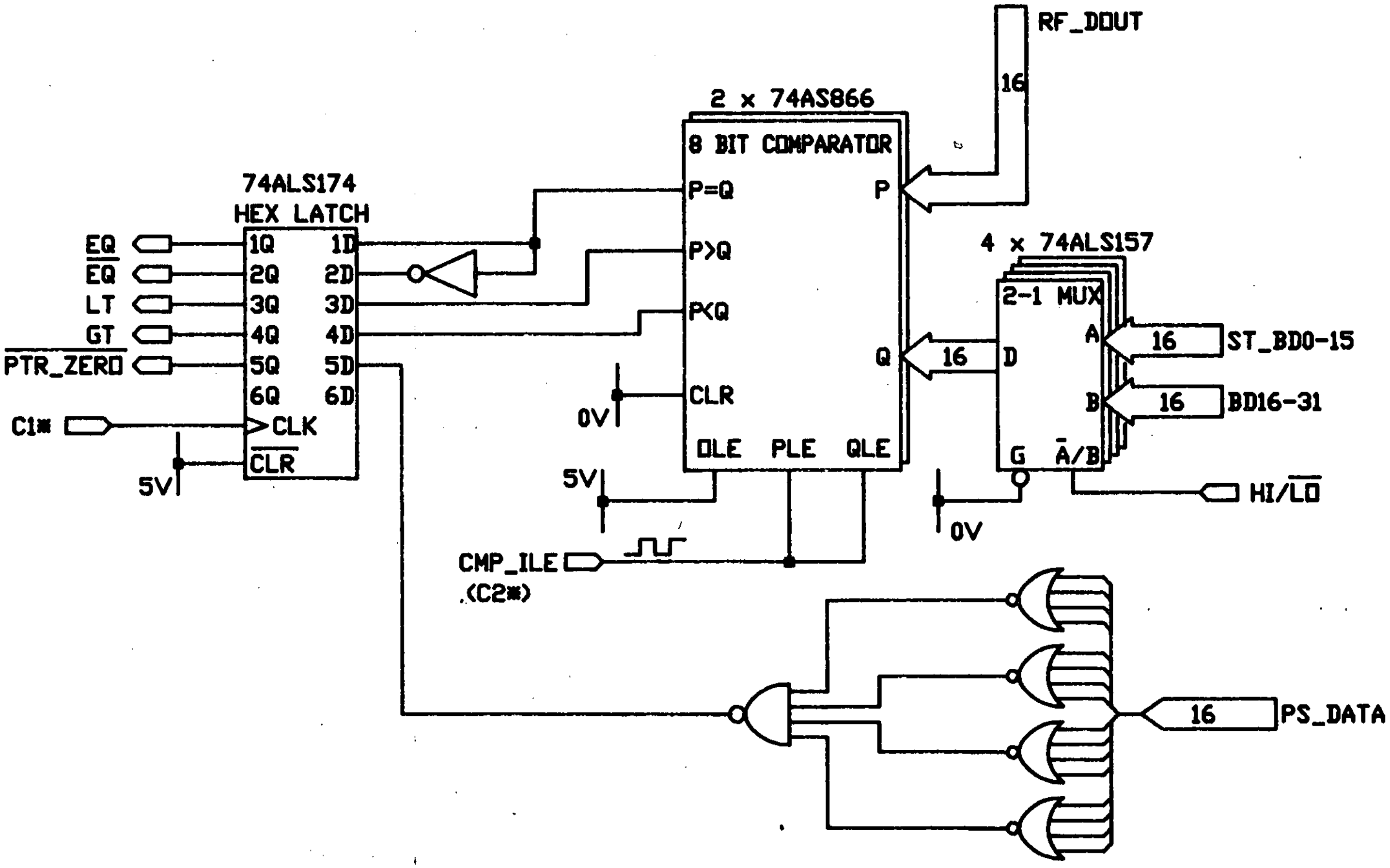


Figure 4.13: The Parallel Comparator Circuit – Schematic Diagram

4.5.5. The Pointer Store

The Pointer Store is illustrated in figure 4.14, and is used to store linked-list pointers, and maintain the garbage collection free-list. The store itself comprises two fast access 2k by 8 bit static RAMs (20ns access SSM6116) and is interfaced to a 16 bit counter (2 x 74AS867). Only 16 locations are used, requiring 4 address bits from the microprogram word. The counter is used to increment the "end of used memory pointer" (*mem_ptr*) as information is added to the KB (see Section 4.6), and to keep a count of the number of relations which match a particular specification (if requested). The input to the store can be sourced from three places; either the Register File or $BD_{16\text{-}31}$, via a 2-to-1 multiplexer, or from the store itself, via the counter. Note that the counter is connected such that it can either increment data as it passes through, or leave it unchanged. The output can be directed to the Comparator Circuit (for comparison with zero), the LOCAL address bus (to access the next property or value in a list), or $BD_{16\text{-}31}$ (to modify list pointers while inserting or deleting information in the KB). Octal bus buffers (74ALS541), controlled from the microprogram word, are used to perform "wired-OR" connections between the Pointer Store and its related functional blocks.

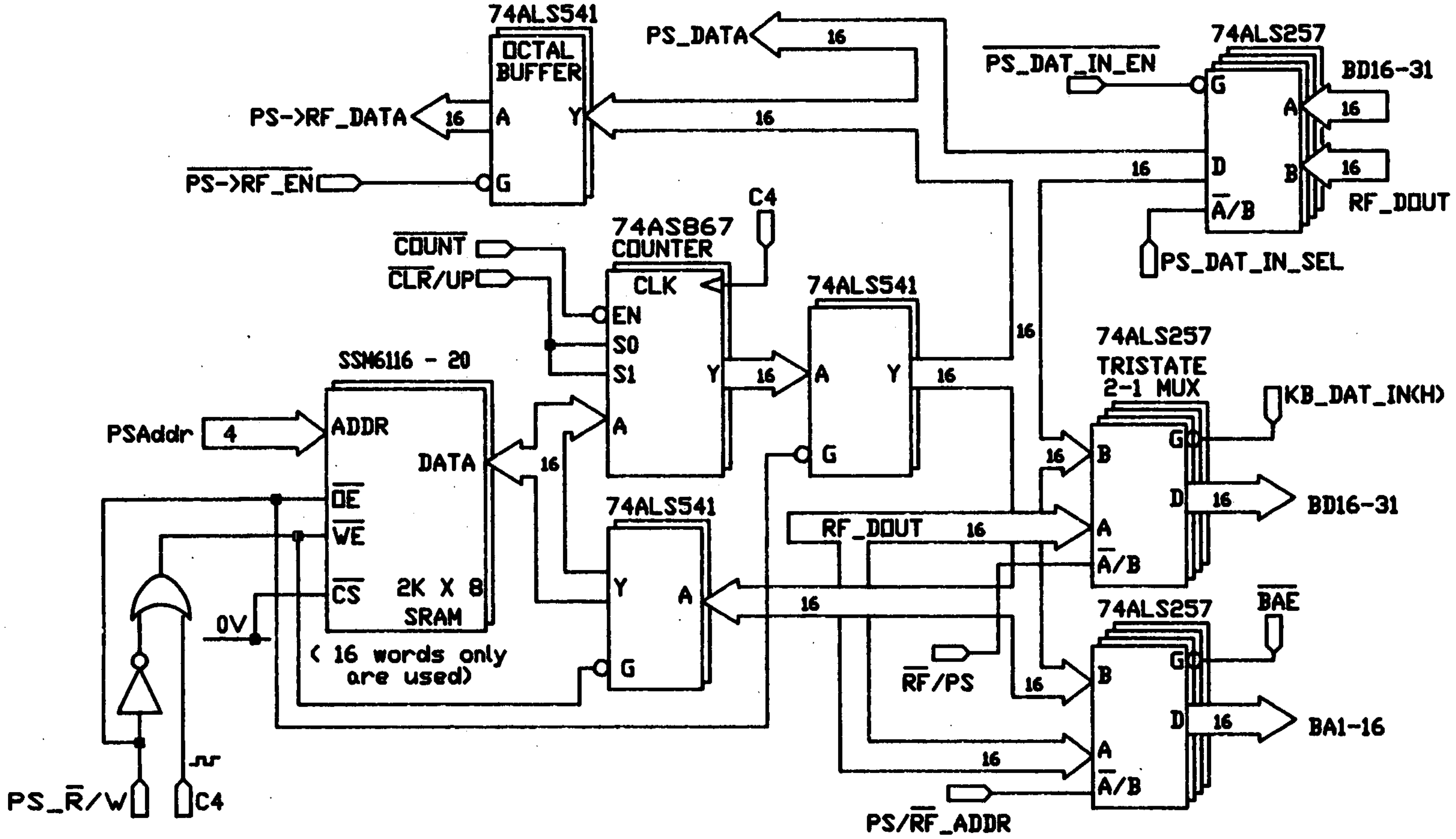


Figure 4.14: The Pointer Store – Schematic Diagram

4.5.6. The Status Control Circuit

The Status Control Circuit performs two operations. Each 16 bit status word is divided into two halves; 8 bits are available for future development, and pass through the Status Control Circuit unchanged; the other 8 bits are reserved for status information, including a mark bit for book-keeping during set operations. The first operation is to set or clear the mark bit, and control bits are provided by the microprogram to do this (SET_MARK and $\overline{CLR\_MARK}$). The mark bit can be cleared when read from the KB into the Register File (to perform a reset at the start of a set operation), and either set or cleared when a status word is being written from the RF to the KB (either to mark a relation as being part of the specified set, or to remove an ineligible relation from the set, respectively). The second operation (masking) involves the logical ANDing of the 7 remaining status bits with those in the specification word, so that only the relevant ones are compared. This function is enabled or disabled by a control bit from the microprogram ($\overline{TST\_STAT}$).

Two octal buffers (74ALS541) are used to disable the output path from the Register File to the least significant I/O Port when data is being input, otherwise contention would occur as the File attempted to output a specification to the P input of the code comparator, while information was being placed by the I/O Port at the Q input of the comparator. The Status Control Circuit is illustrated in figure 4.15.

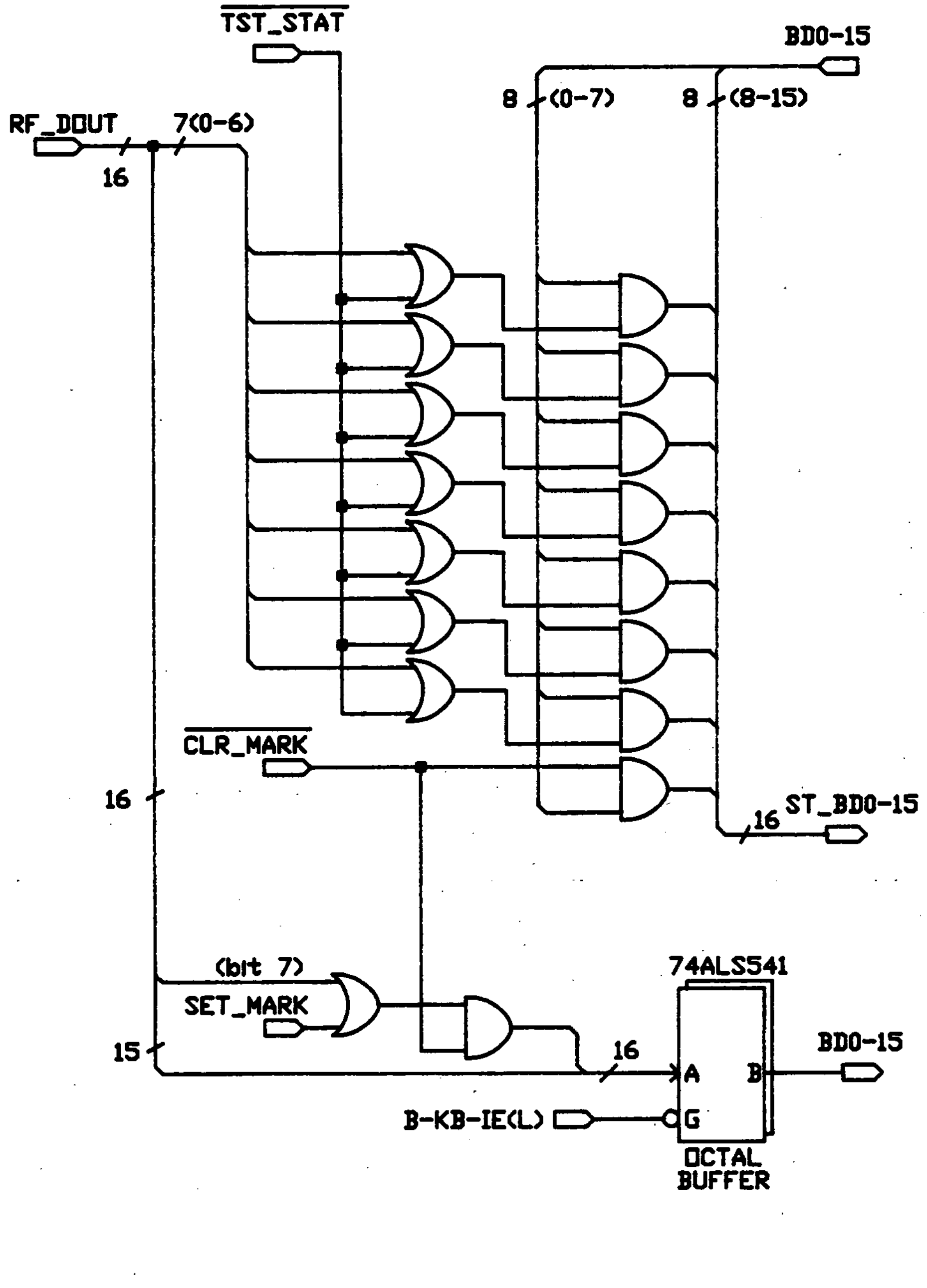


Figure 4.15: The Status Control Circuit – Schematic Diagram

4.5.7. The Condition Code Selector

The Condition Code Selector circuit comprises a series of 2-to-1 and 4-to-1 multiplexers and an Am2922 8 input Multiplexer with Control Register The circuit is used to direct either the $P=Q$, $\overline{P=Q}$, $P<Q$ or $P>Q$ test signals to the Sequencer condition code input, depending on which information is being compared (property, value, context or status words). Status bits from the CSR (including the Poll bit) and the output from the pointer comparator ($\overline{PTR\_ZERO}$) are also directed in this manner. A 0V input to the Sequencer $\overline{CC}$ can be used to force tests to pass. This facility is particularly useful for generating unconditional jumps or jumps to sub-routines.

A 4-to-1 multiplexer is used to force a fail input to the Sequencer Circuit depending on the value of the wildcard bits in the CSR. This is because a failed test indicates a successful match as far as the search algorithm is concerned (see Section 4.6). The circuit is illustrated in figure 4.16.

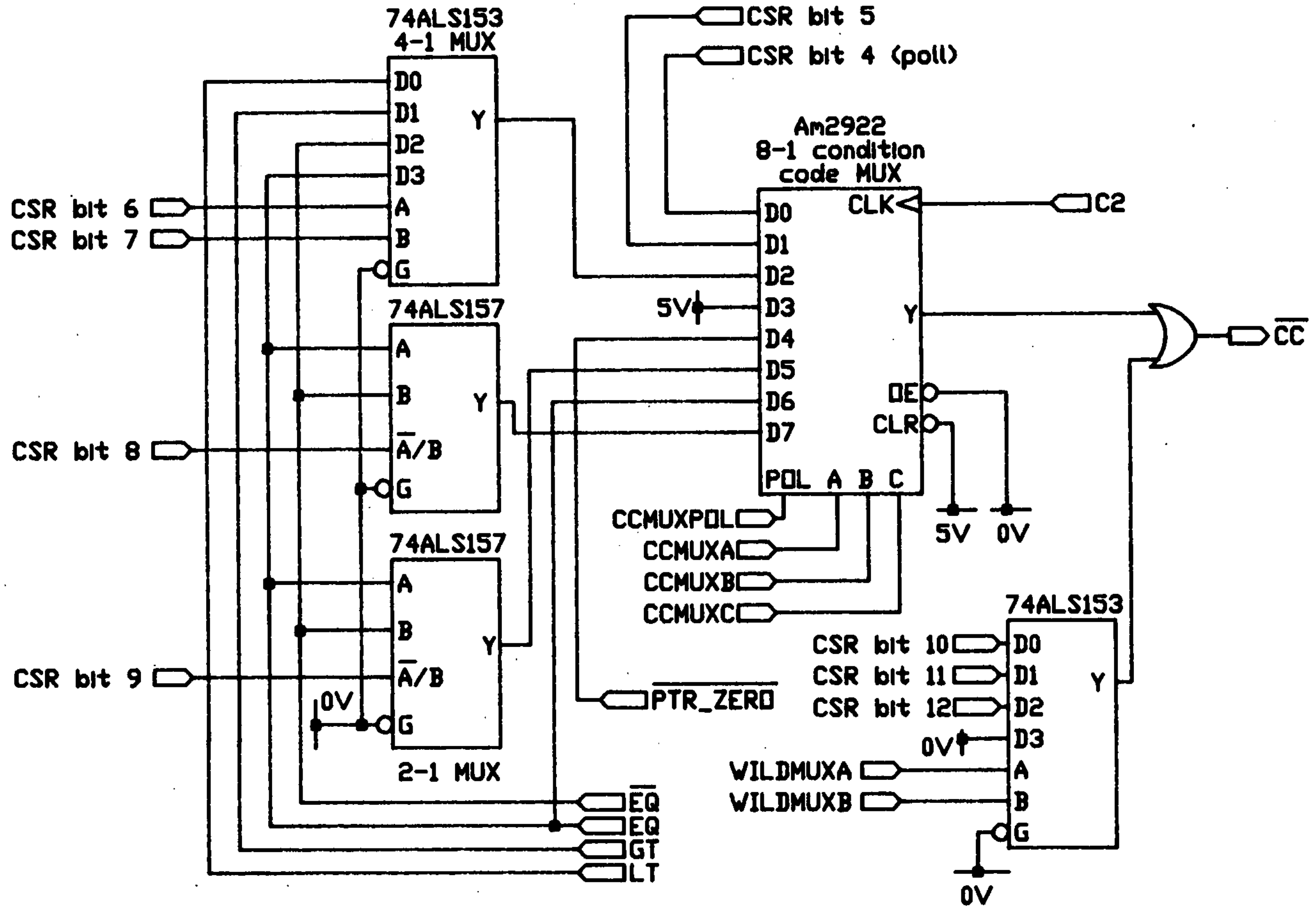


Figure 4.16: The Condition Code Selector — Schematic Diagram

4.6. SKMS Control and Operation

This section describes the control of the SKMS through the sequencing of microinstructions, and the interpretation of the outputs of functional components by the Condition Code Selector. This is followed by a discussion of the basic operation of the system, which is the hardware implementation of the software algorithms described in Chapter 3.

4.6.1. System Control

The SKMS operation is controlled directly via the microprogram in the MPS, and indirectly by the HOST via the Mailbox. The microprogram comprises a series of functional routines (create, delete, modify and retrieve relations) preceded by a startup routine of the form:

```
(i)   REPEAT
          no operation
(ii)  UNTIL (CSR Poll bit is set (see figure 4.11))
(iii) Jump to address given by MAP in CSR (JMAP)
```

Thus, the MAP address, which is used to set the sequencer program counter to the start of the desired routine, is effectively an SKMS *op-code*, set by the HOST.

The operation of the Am2910A sequencer has been modified by external connections (see Section 4.5.2) to execute a reduced instruction set. Table 4.2 summarises these instructions.

TABLE OF INSTRUCTIONS						
I_3-I_0	Mnemonic	Fail Test		Pass Test		Enable
		Y	*STACK*	*Y*	*STACK*	
0	JZ	0	clear	0	clear	PL
1	CJSR C2JSR†	PC	hold	D	push	PL
2	JMAP	D	hold	D	hold	MAP
3	CJPA C2JMP†	PC	hold	D	hold	PL
10	RTN	PC	hold	stack	pop	PL
14	CONT	PC	hold	PC	hold	PL

† These instructions are differentiated by a microprogram control bit (PLA/B_Sel_Or - see Section 4.5.2).

Note 1: Instructions 4→9, 11→13 and 15 are unused.

Note 2: Test passes if *CCEN** = H or *CC** = L and test fails if *CCEN** = L and *CC** = H

Table 4.2: SKMS Sequencer Instruction Set

The sequencer commands define the *next* value of the program counter (PC). The user stack (US) allows the nesting of sub-routine calls up to eight levels deep; [US] denotes the *contents* of the User Stack in the following description.

JZ PC = 0; US = 0.

This instruction specifies that the address output (ie: the address of the next microinstruction) is zero and resets the stack pointer. The RPU $\overline{RESET}$ switch forces this condition to ensure the correct startup state occurs. JZ is also used at the end of each microprogram function (as a "jump to start" command) to effect a soft RESET as a safety precaution.

CJSR If $CC*$ is low: [US] = PC + 1; PC = address; US = US + 1.

Else PC = PC + 1.

This instruction is a conditional jump to a subroutine, whose address is supplied by the Pipeline Register (PL). The $\overline{PL}$ output is activated to enable the PL input to the Sequencer. The jump occurs if the condition code input to the Sequencer ($\overline{CC}$) is low, otherwise, the program counter is simply incremented. The stack stores the old value of the program counter in preparation for a return at the end of the subroutine. If CJSR is specified, then the jump address is supplied by *PLAddrA*. If, however, C2JSR is specified, then the address is *PLAddrA* if $\overline{PTR\_ZERO}$ is low and *PLAddrB* if $\overline{PTR\_ZERO}$ is high, where $\overline{PTR\_ZERO}$ is the output from the Pointer Comparator (see figure 4.13).

JMAP PC = CSR(b3 - b0)

The program counter takes its value from the least significant 4 bits of the Control and Status Register. The $\overline{MAP}$ output is activated to enable the CSR input to the Sequencer. As described above, this facility enables the HOST to supply the RPU with an *op-code*.

CJMP If $CC*$ is low: PC = address; stack (US) unchanged.

Else PC = PC + 1.

This instruction is a conditional jump to the address supplied by the Pipeline Register. The program counter is not stored on the stack, so a *return* would result in an unknown state and is therefore illegal. In all other respects, CJMP and C2JMP act like CJSR and C2JSR.

RTN US = US - 1; PC = [US]

This instruction is used to branch back to the instruction following the last subroutine call, and takes the value of the program counter from the stack.

CONT PC = PC + 1

This instruction simply causes the program counter to be incremented, and the next sequential microprogram control word is executed.

To initiate a microprogram routine, the HOST writes the specification information into the Register File, and the appropriate op-code (MAP address) and status information into the CSR (see fig. 4.11). For each specification code in the Register File, there is a corresponding match code field – the address allocations are presented in figure 4.17. If wildcard matches are desired, then the HOST must set the appropriate fields in the CSR. Similarly, to define the type of comparison performed on the property, value and context codes (equal to, not equal to, greater than or less than), the appropriate bits must be set in the CSR.

As is shown in figure 4.9, there are 8 possible inputs to the sequencer $\overline{CC}$ input. One of these is simply connected to 0V so that it is possible to force a test to pass. This is particularly useful for generating unconditional jumps within the program. The microcode has been written such that a successful match equates to a *failed* conditional code test. Consequently, the wildcard fields in the CSR, if set, cause the $\overline{CC}$ tests to fail at the appropriate times.

CODE MNEMONICS	ADDRESS FROM HOST	ADDRESS FROM RPU
SPECIFIED NEW VALUE (S_NEWV)	0xC00000	0x0
SPECIFIED PROPERTY (S_PROP)	0xC00002	0x1
SPECIFIED PROP STATUS (S_PSTAT)	0xC00004	0x2
SPECIFIED VALUE (S_VAL)	0xC00006	0x3
SPECIFIED CONTEXT (S_CTXT)	0xC00008	0x4
SPECIFIED VAL STATUS (S_VSTAT)	0xC0000A	0x5
UNUSED	0xC0000C	0x6
MATCHING PROPERTY (B_PROP)	0xC0000E	0x7
MATCHING PROP STATUS (B_PSTAT)	0xC00010	0x8
MATCHING VALUE (B_VAL)	0xC00012	0x9
MATCHING CONTEXT (B_CTXT)	0xC00014	0x10
MATCHING VAL STATUS (B_VSTAT)	0xC00016	0x11
UNUSED	0xC00018 ↓ 0xC00070	0x12 ↓ 0x38
MEM_PTR	0xC00072	0x39
RF_ERROR	0xC00074	0x3A
CLEAR_PS	0xC00076	0x3B
FIRST_PROP	0xC00078	0x3C
PROP_PTR	0xC0007A	0x3D
KB_ADDRESS	0xC0007C	0x3E
CONTROL & STATUS REGISTER (CSR)	0xC0007E	0x3F

Figure 4.17: Register File Address Allocations.

4.6.2. Construction Details

The SKMS prototype was constructed in a VME development Euro-card frame (Plate 4.1). Two cooling fans were added to maintain a temperature within the commercial IC specification (0°C → 70°C). This was necessary, due to *hot spots* created by the close proximity of the circuit boards. Two switched mode power supplies provided the 5V, 12V and Ground lines. Each PSU was rated at 5V/10A and 12V/2A, and were doubled up to ensure that they were never overloaded. The power dissipated in the entire SKMS system is ~50W (this is divided into 5V/8.25A and 5V/2.09A over the two PSUs). At one time during the debugging stage, power supply problems were suspected as a cause of a temperature dependent marginal timing problem, which caused corrupted data to be written to the Knowledge Base. This suspiscion was supported by the fact that the original single PSU was marginally overloaded. Although the inclusion of a second PSU lessened the occurence of the fault, it did not eliminate it. The fault manifested itself very rarely, and is almost certainly due to propagation delays created by the large amounts of wire wrapped data buses in the system becoming significant in a high speed environment. One possible solution would have been to employ a custom designed multi-layer printed circuit board. However, the cost of this approach was not considered justified for a prototype.

The VME Interface and Knowledge Base were constructed on an extended double Euro-card circuit board (Plate 4.2). An extended double Euro-card was also used to construct the Microprogram Store (Plate 4.3). The Relational Processing Unit, however, was much more complex and required an extended double Euro-card with an extended single Euro-card daughter board (Plate 4.4). Since no processing elements were used in the RPU design, a great many discrete devices were required. The resulting component density caused many problems with regards to heat dissipation and device layout. Nevertheless, the advantage that lies with this approach is the possibility of implementation on an IC or IC set, which would solve the timing problems associated with the large amounts of wiring required in the prototype, and support a faster RPU cycle time.

The circuit descriptions, above, illustrate the complexity of control, and an 80 bit wide micro-control word was required to co-ordinate the system operation and communication – both between the functional modules comprising the RPU, and between the RPU itself and the MPS, KB and HOST. A microprogram language and micro-assembler were developed and are described in Section 4.7

4.6.3. System Operation

Prior to normal operation of the SKMS, the system must be set up correctly. The user must initialise all HOST accessible memory spaces† before downloading the microprogram. Care must be taken to ensure that the toggle switches are correctly set for the Mailbox and Microprogram Store.

The *search and match* algorithm described in Section 3.3.5 was translated into the following RPU implementation.

(a) The address of the 1st property block (*1st_prop) is written to the Register File from the HOST, and then into the Pointer Store. The address is simultaneously placed onto the LOCAL address bus and the least significant 32 bits of the 1st property block is loaded into the LOCAL bus I/O Port.

(b) The property ID code is copied into the Register File and placed at the Q input to the 74AS866 comparator, while the pointer to the next property (*pnext) is copied into the Pointer Store. The specified property code is placed at the P input of the comparator, in parallel to the above two operations.

† This includes the PME012D RAM board used for the hash and symbol tables.

(c) If the codes match, the most significant 32 bits of the property block are retrieved and compared in the same way. This time, however, the code is the property status word and the pointer is to the 1st value block associated with this property (*1st_val). If the status codes match...

[i] Retrieve the least significant 32 bits of the 1st value block into the LOCAL bus I/O Port from the address *1st_Val, which is supplied by the Pointer Store:

[ii] The value, status and context codes are stored in the Register File, and compared with the specified ones, in the same way as the property block. The pointer to the next value (*vnext) is also stored in the Pointer Store, as before.

[iii] If all of the codes match...

1) The Poll Bit in the CSR is cleared, which alerts the HOST that a successful match has been found.

2) The microprogram counter is reset to zero (JZ).

[iv] If any of the matches fail...

1) *vnext is compared with zero simultaneously to the code comparisons. If it is NULL (ie: at the end of a list), then *pnext is compared with zero. If this, too, is NULL, then the search has failed, and a failure flag is set in the Register File. The poll bit is then cleared to alert the HOST. If it is not NULL, then *pnext is used to retrieve the next property block and then the program counter jumps to (b).

2) If, however, *vnext was not NULL, then it is used to retrieve the next value block.

3) Go to (c)[ii]

(d) Else...

[i] *pnext is compared with zero simultaneously to the code comparisons. If it is NULL (ie: at the end of a list), then the search has failed, and a failure flag is set in the Register File. The poll bit is then cleared to alert the HOST.

[ii] Else retrieve the next property block from address *pnext in the Pointer Store.

[iii] Go to (b)

Clearly, this is a basic interpretation, and the contribution of the Status Control Circuit and wildcard facility have not been discussed. More detailed information can be elicited from the microprogram listing in Appendix F.

4.7. Programming

Since the RPU is composed of several inter-dependent modules, construction was *by degrees*. The first circuit to be built was the Clock Generator Circuit, since all others depend upon its outputs. Next was the Sequencer, then the Condition Code Selector, etc. Each time a new section of the circuit was added, new control bits were required to test it. Consequently, as the hardware developed in a structured fashion, so too did the micro-software.

The choice of microprogram language for an environment such as the SKMS is extremely important. Ideally, it should possess the following features:

- **flexibility** – to support different system designs or design alterations
- **minimal encoding** – each microinstruction bit is responsible for a different function, thereby maximising system parallelism and hence speed. Unfortunately, the result is often a very large control word, and some bits are generally encoded.

- **easy to use** – since microinstructions are often large, a microprogram language should support the definition of high-level macros and sub-routines, which can be assembled into the appropriate microcode.

Since the SKMS is composed of several functional modules operating independently (each requiring control by the microprogram), instead of a single processor, whose operations are easily defined and controlled through its instruction set, it was not possible to define single mnemonics for each operation of the system. Instead, several *sub-instructions* were required – one for each functional module. Consequently, there were no suitable microprogram languages or development tools available and a microprogram language and micro-assembler (*μa10A*) were developed specifically for the SKMS. As one might expect, due to the complicated nature of the circuit, μa10A is also complex, and each control word is of the form:

{

<IN> defines sequencer I_{0-3} inputs.

<CC> controls the Condition Code Selector.

<CL> defines the clock speed ($L1$, $L2$, and $L3$).

<CM> controls the Parallel Comparator Circuit.

<KB> controls the KB and I/O Port.

<PS> controls the Pointer Store Circuit.

<RF> controls the Register File.

<ST> controls the Status Control Circuit.

<WD> defines the inputs to the Wildcard MUX.

}

Writing programs in this format would be tedious and almost impossible to debug. Therefore, μa10A macros and sub-routines, defining all the operations required by a system programmer, were compiled and stored in libraries, and a macro pre-processor (*mp*) and high level assembler (*ksma*) were written to enable the user to develop easily understandable programs. *ksma*, *mp*, *μa10A* and their relationships are described more fully in Appendix E. UNIX shell scripts are used to perform the downloading of code into the Microprogram Store and the HOST – they are listed in Appendix D.

To test the performance of the SKMS, a basic C interface was written, which would allow the USER to manipulate the Knowledge Base via the HOST and RPU. Chapter 5 describes the use of a simple command line parser which calls these C→SKMS functions in order to time particular operations. This could be taken a step further by utilising a Prolog→C interface which would allow communication between the SKMS and a standard AI Language.

Plate 4.1: The Structured Knowledge Manipulation System (SKMS) Prototype

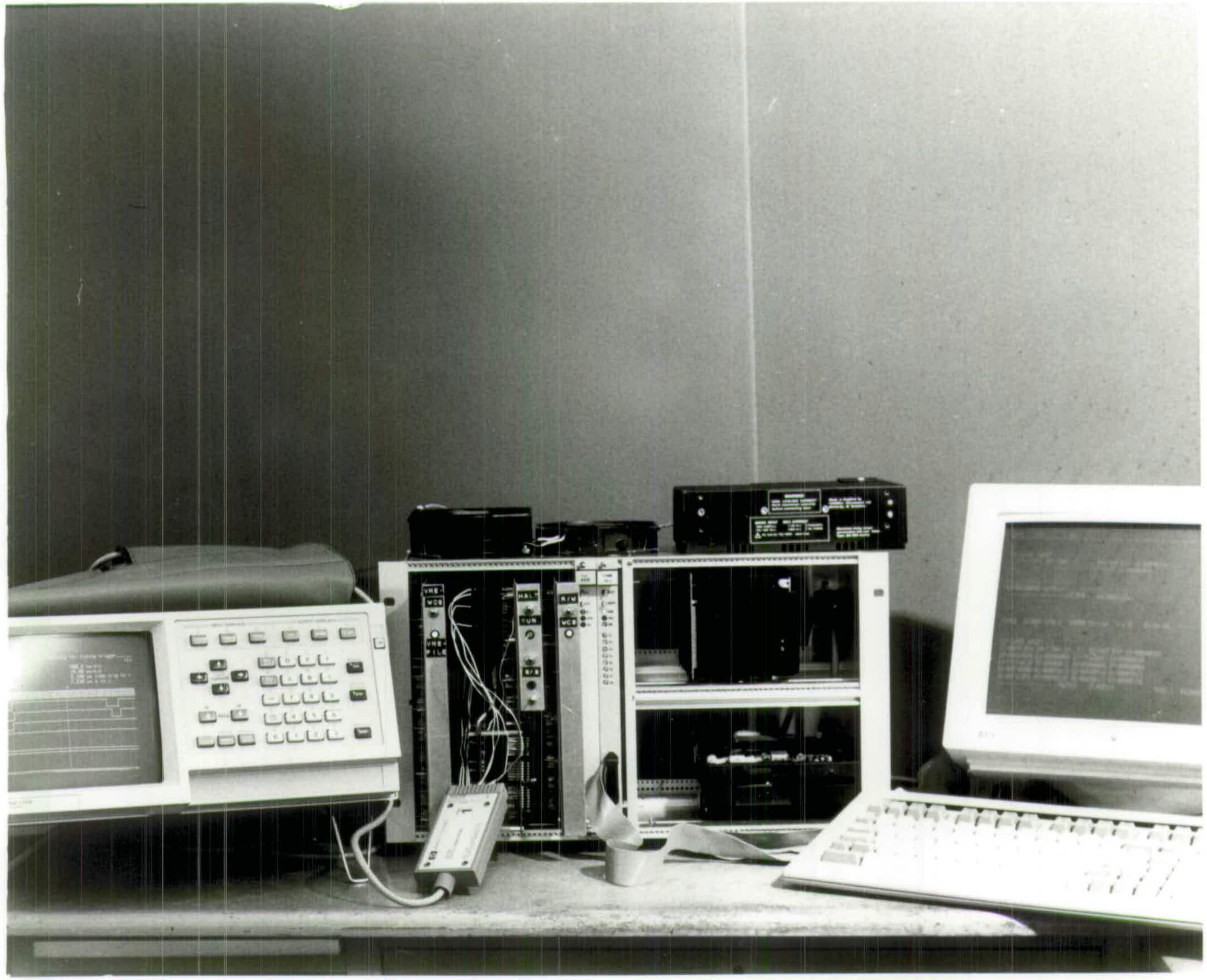

VME-
MCB
VME-
FILE
HALT
RUN
R/W
MCB

Plate 4.2: The VME Interface Circuit and Knowledge Base (KB).

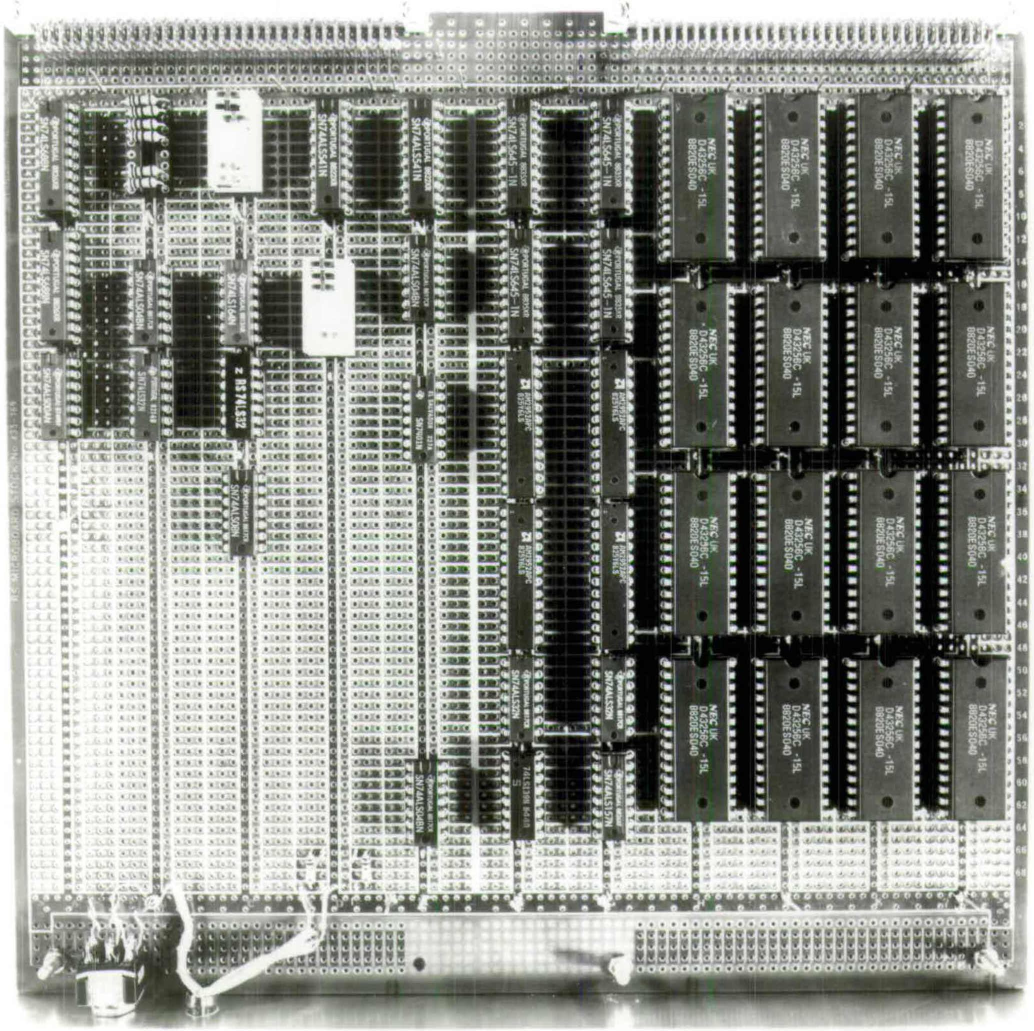

SN74ALS541N
SN74ALS645-1N
SN74ALS04BN
SN74ALS32N
SN74ALS157N
RS74LS32
NEC UK
D43256C-15L
8820ES040

Plate 4.3: The Microprogram Store (MPS).

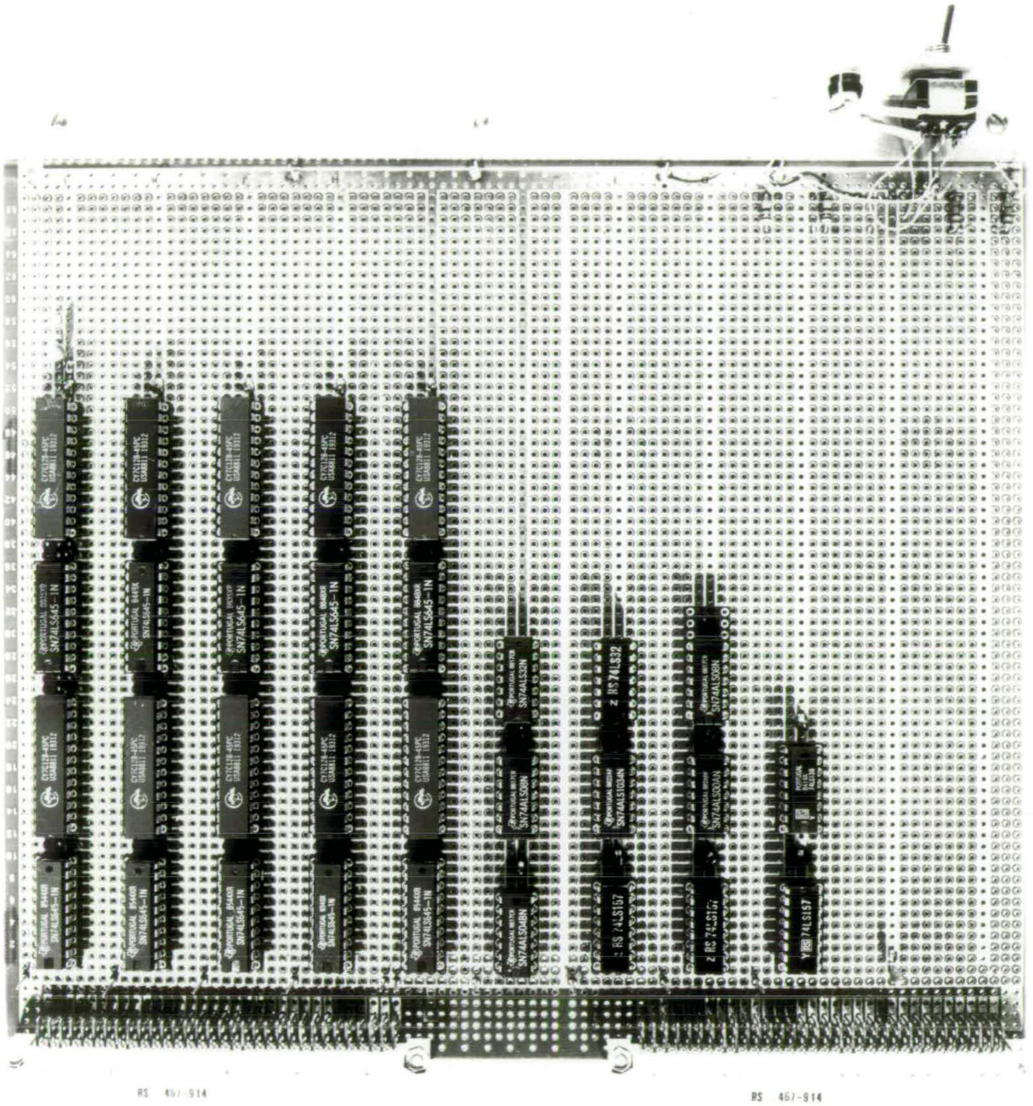

SN74ALS645-1N
CY7C128-45PC
SN74ALS32N
SN74ALS08N
SN74ALS04BN
RS 74LS32
SN74ALS1034N
RS 74LS157
SN74ALS00AN
RS 74LS157
RS 467-914
RS 467-914

Plate 4.4: The Relational Processor Unit (RPU) – Mother Board.

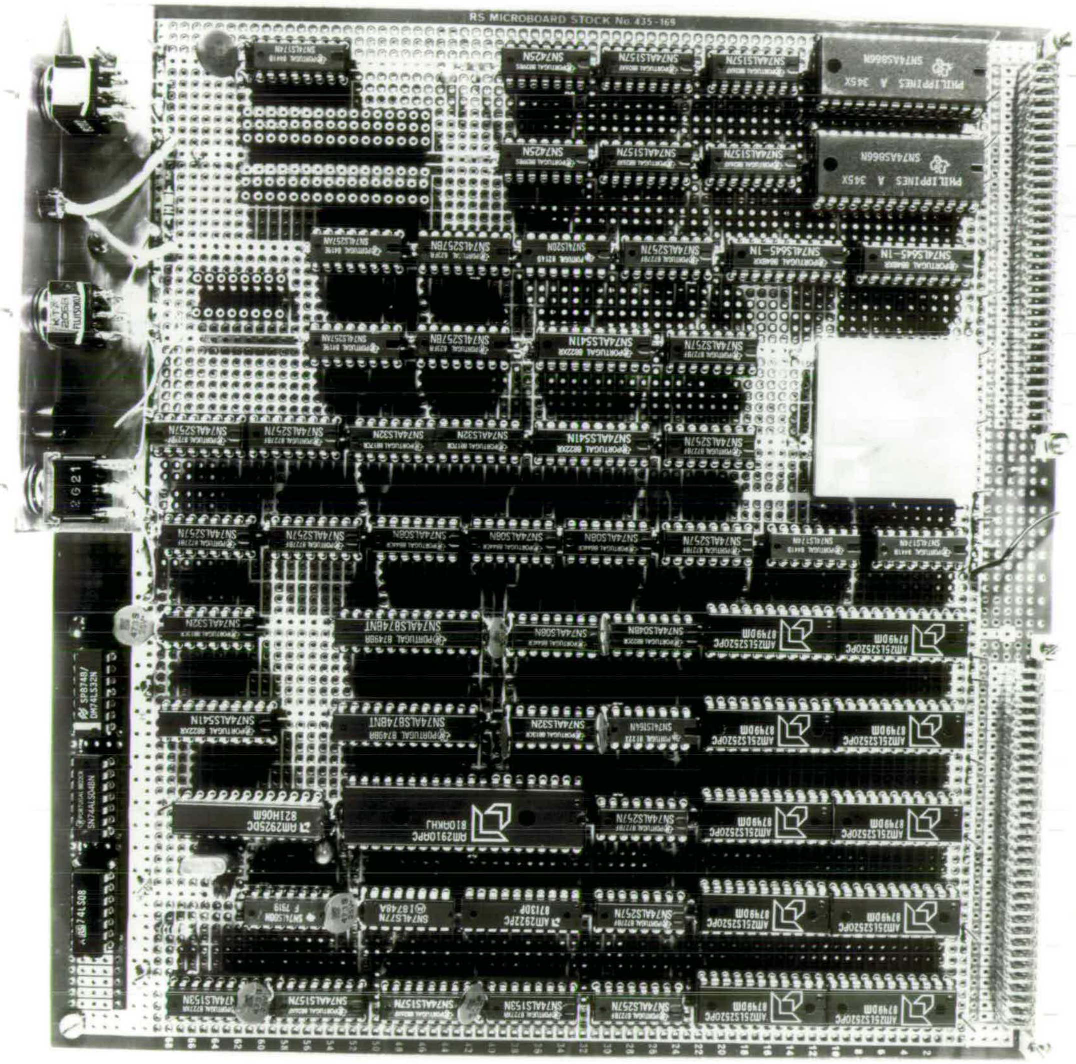
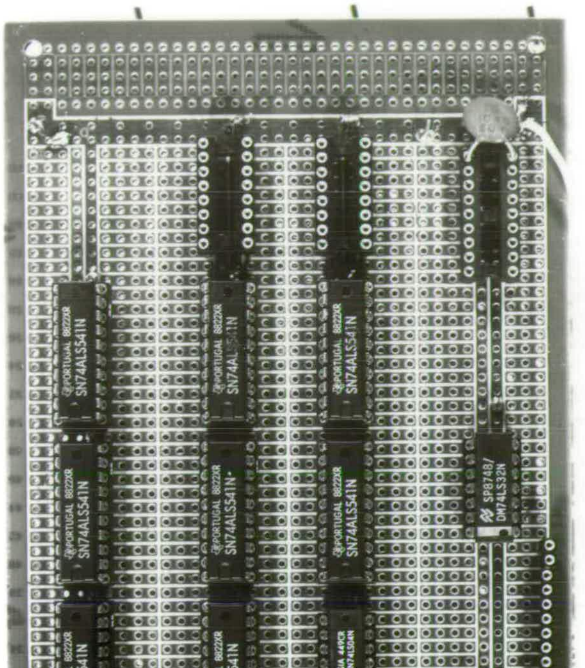

RS MICROBOARD STOCK No. 435-169
SN74LS157N
SN7425N
SN74LS157N
SN74ALS157N
PHILIPPINES A 345X
SN74AS866N
SN74LS257N
SN74LS257BN
SN74LS20N
SN74ALS257N
SN74ALS645-1N
SN74ALS541N
SN74ALS32N
SN74ALS08N
SN74ALS874BNT
AM25LS2520PC
8749DM
AM2910APC
AM2925DC
AM2922PC
SN74LS77N
SN74ALS153N
SP8748/DM74LS32N
74LS08
2 G 21

Plate 4.5: The Relational Processor Unit (RPU) – Daughter Board.


SN74ALS541N
SP8748/
DM74LS32N
SN74LS04N
SN74ALS257N
SN74AS867NT
SSM6116 20SC
8740 HS 34204
RS 467-914

CHAPTER 5

Verification and Results

5.1. Evaluation Systems and Methods

In this chapter, performance evaluation results are presented of the Structured Knowledge Manipulation System (SKMS) retrieval times. Two major comparisons are made, serving two different purposes. Firstly, to verify that the hardware support does indeed provide speed improvements over the software simulation (following from the features discussed in Chapter 3), the SKMS performance is compared with the software version running on a Motorola 68010 based Single Board Computer. Secondly, to evaluate the suitability of the knowledge structure developed and described in this thesis, and the effectiveness of the SKMS as a whole, the performance of the system is compared with other hardware based (or supported) Knowledge Manipulation Systems. The software was also compiled to run on two other systems; a Personal Computer based on an Intel 80286 processor and a Sequent Mainframe Computer based on the Intel 80386 processor. Their performances are also presented here. Projected performances are presented for the additional times required to execute the create (insert), modify and delete functions once the retrieval algorithm has located the target relation.

A simple command interface program was written which would parse a request from the user, calculate the string hash-codes and initiate the appropriate retrieval or manipulation routine. The same code was used to call either the software-based simulation routines, or the routines implemented in the SKMS hardware. In this way, it was possible to predict, and keep uniform, the user interface overheads. The 68010 based routines were cross-compiled using an optimised MIT Portable C Compiler and the total number of machine cycles used by each one was recorded. Given the clock rate of the system, and assuming no wait states, is was possible to derive the approximate times taken to perform each routine. Initially, considering a single knowledge structure (object) comprising 20 properties, each of which consists of 20 values (ie: 400 relations), approximate projected times were calculated, for both the software and hardware based systems, to retrieve the 1st, 100th, 200th and 400th relations for two different situations:

(i) The property, value and context are known (ie: a full specification).

?(Object_Name, Property_Name, Value_Name, Context)

(ii) Only the value is known; the property and context are wildcards. The object name is supplied since, in this case, there is only one object, and it is preferable to reduce the effects of hash table searching as much as possible, so as not to cloud the issue of *KB search* time measurement.

*?(Object_Name, *, Value_Name, *)*

These two search types are at either end of the retrieval time range; (i) being the fastest type of search and (ii) being the slowest. These results, presented in tabular form below, may be compared with the actual recorded values. The next step is to extrapolate the results to derive the projected performance of a hypothetical parallel relational processor, based on the SKMS architecture.

The evaluation system comprises five circuit boards connected via a VMEbus and two local buses (see Chapter 4). A FORCE Computers SYS68K CPU-3 Single Board Computer running at 10MHz is used both as the SKMS *HOST* and to run the software simulation system. 10MHz is the fastest clock which can most efficiently utilise the 150ns access SRAM Knowledge Base without wait states; since a write operation requires a minimum access time of 2½ clock cycles (250ns), and a read operation requires a minimum of 1½ cycles (150ns). A Plessey Microsystems PME012D 512 kbyte DRAM board is used to store the hash and symbol tables for the user interface routines. The MPS and RPU boards comprise the core of the SKMS hardware support (see Chapter 4), while the Knowledge Base (KB) board is used by both the hardware and software based systems to store relations.

Since both the software and hardware systems utilise the same SRAM board as their Knowledge Base, it is possible to calculate the projected times of each system running at maximum capability, ensuring that the performance comparison is dependent not on memory speed, but on architectural differences. Approximate times for the user interface overheads may also be calculated. Due to the dependence of such information on the size of the input character strings and the hash-table collision rate, it was assumed that all strings were 6 characters in length, and that an average of 3 collisions occurred per string (since a very simple hashing algorithm was selected). Although this overhead is important, and is discussed in more detail later, a more important consideration is the overhead relating to the mailbox polling-based communication between the 68010 host processor and the SKMS co-processing hardware. Again, the appropriate code was examined to determine the number of machine cycles involved, and hence the time taken to perform the operations.

The control software was modified so that it was possible to execute a particular search a definable number of times before printing out the result. Consequently, timing each operation was a simple matter of measuring the time taken to execute it many times. Since the number of repetitions necessary to produce times of easily measurable values (greater then 10 seconds) was of the order of 10000, it is quite reasonable to ignore the effect of a single execution of the input and output overheads. A single command (*test*) was used to create the test object comprising 400 relations, and the facility to execute only the parsing and hashing operations was included, so that this overhead could also be measured by timing many repetitions.

5.2. Performance of the Simulation System

Having reduced the simulation retrieval routine into independent sub-units, it is possible to derive an equation to calculate *approximate* numbers of machine cycles required for the retrieval of various relations:

$$\textit{search time} \simeq 250 + N_{pp}\{A + 230 + N_{vf}(B + 120) + N_{vp}(B + 650) + 230\}$$
$$+ N_{pf}(A + 100) \textit{ machine cycles}$$

where:

N_{vf} = number of failed value/context matches

N_{vp} = number of passed value/context matches

= 1, if match found; 0, if no match found

N_{pf} = number of failed property matches

N_{pp} = number of passed property matches

and:

A = 140 (wildcard property)	B = 280 (wildcard value)
A = 210 (specified property)	B = 350 (specified value)

This equation was derived by the examination of 68010 format assembly coded versions of the appropriate segments of the simulation software. By referring to the 68000 family User Reference Manual [63], machine cycle totals of the code segments could be calculated. Due to the difficulty in calculating machine cycles on a 68010, and accounting for *every* loop or jump in the software, these figures cannot be guaranteed to be exact, so should be used only to identify trends.

Assuming the 400 relation structure described in Section 5.1, table 5.1 summarises the expected times for a 68010 based system running the simulation software. Note that the times for the retrieval of the 100th, 200th and 400th relations given a full specification are constant, while the times when only the value is known increase more or less linearly. This is because the only way to find a relation given just the value is to perform a sequential search of the knowledge base; whereas, the 100th, 200th and 400th relations contain the 20th value of the 4th, 9th and 19th properties respectively, and so, if the properties are known, the search times are almost identical and, as expected, greatly reduced.

Approximate Projected Times Of Software Simulation Package

(based on a 68010 processor running at 10MHz)

Matching	*Only Value Known*		*Full Specification*	
Relation	*Machine Cycles*	*Time(μs)*	*Machine Cycles*	*Time(μs)*
1st	1800	180	1900	190
100th	43800	4380	10700	1070
200th	87200	8720	10700	1070
400th	173900	17390	10700	1070

Table 5.1: Projected Simulation Package Times.

It is also possible to derive the approximate projected time taken to retrieve each subsequent relation for this object type. At 10 MHz this figure is 44 μs/relation.

5.2.1. Overheads

For a 6 character word and an estimated collision rate of 3, the hashing and symbol table manipulation algorithm requires 4270 machine cycles. Since it is called 4 times (for object, property, value and context strings), we require approximately 17000 machine cycles. The command parse routine requires approximately 10000 cycles, which gives us a total of 27000 machine cycles, which is about 2700μs for a 10MHz system. Note that this time is of the same order as that required to retrieve a fully specified relation, and so the problem of hashing, although not addressed in detail in this thesis, requires further investigation. Table 5.2 summarises the results.

Approximate Projected SYS68K User Interface Overheads

(based on a 68010 processor running at 10MHz)

Operation	*Machine Cycles*	*Time(μs)*
Parsing	~10000	1000
Hashing†	17000	1700
Total User I/F	~27000	2700

† assuming strings of 6 characters, and an average hash hit rate of 3

Table 5.2: Projected User Interface Overheads

5.2.2. Motorola 68010 based Single Board Computer

Table 5.3 shows the actual times taken by the 68010 based software system. These measured times are broadly consistent with the times projected above. As expected, the user interface overheads are quite large in comparison to the search times, although their significance decreases as the search space increases. The measured time to retrieve each subsequent relation is 44.5 μs/relation, which agrees closely with the projected time of 44 μs. Extra searching in the hash table would explain the slight increase in search time for a fully specified relation as we go from the 100th to the 400th relation.

Measured Times Of The Software Simulation Package

(cross-compiled† onto a SYS68K CPU-3 [running at 10MHz])

Matching Relation	*Only Value Known*	*Full Specification*
	Time(μs)	*Time(μs)*
1st	220	240
100th	4500	1200
200th	8900	1200
400th	17800	1300
Overheads	2700	2700

† MIT Portable C Compiler

Table 5.3: Actual Simulation Package Performance

5.2.3. Intel 80286 and 80386 based Systems

Tables 5.4 and 5.5 summarise the results obtained for an Intel 80286 PC running at 20MHz, and an 80386 based mainframe. The search times for the 86286 are comparable to twice the projected 10 MHz 68010 based system, with a value of approximately 22 μs/relation. The time taken by the 80386 based machine to retrieve each subsequent relation is 9 μs/relation. Note that the time spent executing the overheads on the 80386 machine is not much less than the 80286, whereas the retrieval times are appreciably better. This is probably due to one or more of several reasons: the 80386's architecture being more efficient at dealing with the iterative nature of the search process, different memory access times, or greater memory caching in the Sequent machine.

Measured Times Of The Software Simulation Package (compiled† onto a VISION ATom286 [running at 16MHz])		
Matching	*Only Value Known*	*Full Specification*
Relation	*Time(µs)*	*Time(µs)*
1st	140	160
100th	2200	600
200th	4400	700
400th	8700	800
Overheads	900	900

† Turbo C Compiler

Table 5.4: Intel 80286 based PC Performance

Measured Times Of The Software Simulation Package (compiled† onto a Sequent 80386 Computer)		
Matching	*Only Value Known*	*Full Specification*
Relation	*Time(µs)*	*Time(µs)*
1st	50	50
100th	930	230
200th	1840	250
400th	3650	300
Overheads		
Parsing	280	
Hashing	580→620	
Total	860→900	

† C → 80386 Compiler

Table 5.5: Sequent 80386 Computer Performance

5.3. Performance of the SKMS

The projected timing information was based on the maximum clock speed for each instruction, assuming perfect hardware, an increased clock rate (created from a 48MHz input to the Am2925 clock generator) and no timing problems. By examining the appropriate microcode sections, an equation to calculate the time taken to find a relation at optimum speed may be derived:

$$\textit{search time} \simeq (11 + N_p(15 + 15N_v)) \times 0.1\ \mu s$$

where:

N_p = number of properties examined

N_v = number of value/context pairs examined

It should be noted that three different clock speeds are involved, but the variation between the different microcode words is such that the average clock period is ~100 ns (0.1μs).

Unfortunately, due to the large amount of wiring present in a wire-wrapped prototype, an unresolved marginal timing problem developed, (see Chapter 4) which caused occasional faulty memory accesses at the desired clock rate, and necessitated a reduction in the clock speeds, giving:

$$\textit{search time} \simeq (11 + N_p(15 + 15N_v)) \times 0.3\ \mu s$$

Projected retrieval times have also been derived for this situation (table 5.6). For the SKMS, the projected time taken to retrieve the subsequent relation in this search tree is approximately 1.5 μs/relation at optimum speed, and 4.5 μs/relation at the reduced speed. The maximum memory space addressable by a 16 bit SKMS is 512 kbytes, which corresponds to a maximum of 32767 relations (a single relation requires 16 bytes). If the memory space is fully used, then the maximum time required to find a relation is ~50 ms, at optimum performance.

Approximate Projected Times Of SKMS System						
Matching	*Only Value Known*			*Full Specification*		
Relation	*Machine*	*Time(μs)*		*Machine*	*Time(μs)*	
Number	*Cycles*	**A**	**B**	*Cycles*	**A**	**B**
1st	40	12	4	40	12	4
100th	1586	480	160	326	100	35
200th	3161	950	320	326	100	35
400th	6041	1800	610	326	100	35

A: SKMS running at reduced speed (3.3 MHz)

B: SKMS running at optimum

Table 5.6: Projected SKMS Performance

5.3.1. Overheads

The same user interface overheads, described for the software package above, apply in this case, since the same code was used to call the various functional routines. In this case, however, we also have to account for the time spent by the 68010 HOST and the SKMS co-processing system in communication , since this is an integral part of the architecture and hence, the search process. Since the Register File, used as the communications mailbox between the SKMS and the 68010, has an access time of 30ns, it may be utilised by the fastest generally available 680X0 family processor (20MHz) without wait-states. This would require a separate DTACK circuit for the VME→Register File interface from the all-purpose one used in the prototype (see Chapter 4), but this would pose no problems.

The 68010 and SKMS communication overheads can be split into three parts. The 68010 host must first set up the SKMS Register File (mailbox) with the values appropriate to the current search and then set the poll bit. Having done this, it repeatedly polls the mailbox to determine whether the search has been completed. Finally, the mailbox is read to verify the success of the search and collect the returned information. Therefore, these overheads define a minimum search time of 51 μs for a 20MHz host, and 102 μs at 10MHz. Note that the projected retrieval time for the 1st relation (12 μs for the prototype) is a great deal lower than the projected minimum overhead time (102 μs), so we should expect the actual measured time to be of the order of 102 μs. Also, we should add the communications overheads to the other projected SKMS times before comparing them with those recorded. Table 5.7 summarises the results.

Approximate Projected Communications Overheads			
Operation	*Machine*	*Time(μs)*	
	Cycles	**10MHz**	**20MHz**
SYS68K→SKMS	500	50	25
Poll SKMS‡	170	17	9
SKMS→SYS68K	350	35	17
Total Communications	1020	102	51

‡ this is the figure for *each* examination of the SKMS poll-bit

Table 5.7: Projected Communications Overheads

5.3.2. SKMS Practical Results

Table 5.8 shows the actual times taken by the SKMS. They closely match the times projected above; with the relation retrieval time being 4 μs/relation, as compared with the 4.5 μs/relation projection.

Measured Times Of The SKMS System ([running at 3.3MHz] interfaced to the SYS68K CPU-3)		
Matching	*Only Value Known*	*Full Specification*
Relation	*Time(μs)*	*Time(μs)*
1st	140	160
100th	500	240
200th	900	300
400th	1700	300
Overheads	2800	2800

Table 5.8: Actual SKMS Performance

As predicted, the time spent retrieving the 1st matching relation was the same order as the minimum time required due to the communications overheads, rather than the projected time. Note that, for both types of search, the time to retrieve the 1st relation should be constant; the small discrepancy in table 5.8 is almost certainly due to the difference in hashing time for the different requests becoming significant in comparison to the low search times. Extra searching of the hash-table would also explain the slight increase in search time for a fully specified relation as the relation number increases from 100 to 400. Again, the user interface overheads are quite large in comparison to the search times, with their significance decreasing as the search space increases.

The calculations summarised in table 5.6, and supported by experimental evidence, taking into account communication overheads, give a search time range for a fully utilised 16 bit SKMS as:

$$51\mu s \leq R_t \leq 50ms$$

Since knowledge base manipulations employ the same algorithm as the retrieve function, with extra code to perform the required operation once the appropriate relation has been found, then the times to delete, insert or modify a relation will be only a small percentage greater than the search times quoted above (see table 5.9). Thus, for a hypothetical parallel system (see Chapter 6) comprising independent SKMS modules, interfaced to a central controller (HOST), this time range is true for any number of relations (number of relations $\geq$ 32767 × number of modules). The major limiting factor now becomes the user interface hashing algorithm, and it is on this area that more effort needs to be spent.

A singularly clear advantage of the SKMS lies in the structured way in which it searches the knowledge base. Consequently, it is able to retrieve *all* of the relations which match a wildcard specification in one sweep of the search space, and so the time to retrieve 5 such matches, say, would be considerably less than the time required to retrieve 5 unrelated ones. Similarly, the deletion or modification of *all* of the relations which match a wildcard specification is performed in one sweep. However, a communications overhead of $\sim 50\mu s$ is associated with each relation passed from the KB to the HOST (table 5.7). This overhead could be drastically reduced if a dual ported relation buffer was employed, which was larger than the Register File incorporated in the prototype.

The status matching hardware (see Chapter 4) facilitates fast set operations such as the join or intersection of two different specifications. Consider the intersection (logical *and*) of several specified relations. During the first sweep of the KB, the mark bit in the status word of properties and values matching the first specification is set, and is performed within the time range (R_t) given above. During each subsequent search, only those branches whose mark bit is set are compared with the appropriate specification – if a marked relation is matched successfully, the mark bit is unchanged, otherwise it is cleared. Thus the increase in execution time per specified relation is better than linear, since each pass requires less time than the last. The join (logical *or*) operation is performed in a similar manner. The ability to search for numerical values which are *greater than, less than, equal to,* or *not equal to* the specification is also supported. Hence, along with the intersection facility, it is possible to match against *ranges* of values. These utilities, although supported in the hardware, have not yet been supported in the control microsoftware.

Additional Times For KB Manipulation **(SKMS running at optimum performance)**			
Operation	*Machine Cycles*	*Time (μs)*	*% Time Garbage Collection ‡*
Insert a value and context pair into a property	20	2	35
Insert a property, value and context into an object	35	3.5	40
Delete a relation†	10→50	1→5	20→50
Modify a value in a relation	2	0.2	0

† these times depend on which value and/or property in their respective lists are to be deleted. Garbage Collection is performed in 5 cycles for each value or property block deleted.

Table 5.9: Additional Manipulation Times

‡ describes time spent in memory allocation and reclamation

5.4. Other Knowledge Based Systems

5.4.1. Ferranti Relational Processor

The Ferranti Relational Processor (FRP) is described in Chapter 2 and claims to support real-time interpretation of radar data. In a Ferranti test involving 34 seconds of radar information, comprising 4000 messages concerning 90 radar tracks (objects) - each of which has 18 attributes (properties), three different basic operations were performed [72]:

1. insert a new track report into the database

2. modify information in an existing track

3. delete a track from the database

Additionally, since Ferranti are interested in targeting their equipment towards radar in particular, which include velocities and co-ordinates, the ability to perform value comparisons ("between bounds" ranging) within the searching process is an important feature and is supported by the FRP. The commercial database ORACLE required over an hour to perform this test, whereas the FRP ran in real time. Other quoted performance results for a database comprising 500 tracks, each comprising 9 attributes, are summarised in table 5.10. Note, that since the FRP does not support contexts, there is only one value per attribute.

Ferranti Relational Processor (FRP) Performance	
Operation	*Time(μs)*
retrieve value given object and attribute	145
modify value given object and attribute	225→322
intersection of two attribute and value pairs†	978
FRP→HOST communication overheads	100

† a *specific* example is described in [72], no fundamental values are presented

Table 5.10: FRP Performances

Intelligent File Store Performance†	
Operation	*Time(μs)*
search with no wildcards	**38**
search with one wildcard	**244**
search with two wildcards	**2000**
insert a relation	**38 → 76**
delete a relation	**4 → 38**
modify a relation	‡

† Note that there are only 64 kbytes per search engine, as opposed to 512 kbytes for the SKMS.

‡ There is no IFS facility to modify values in a relation. It is necessary to perform a composite operation; ie: a delete followed by an insert, which would require 42→76μs.

Table 5.11: IFS Performances

5.4.2. Intelligent File Store

The Intelligent File Store (see Chapter 2) utilises hashing and multiple search engines to achieve associative access, and a search rate of 250 Mbytes/second [74]. The following times are quoted for 3-place relations (2-place predicates) with 64 search engines operating on a 4 Mbyte search space [73,74,75] (Table 5.11):

Lavington also proposes a Relational Algebraic Processor (RAP [75]) as an add-on feature to the IFS Manipulation System. Like the basic IFS, the RAP comprises a parallel (SIMD) arrangement of search engines. The RAP performs set operations (eg: member/search, intersection, join) on the knowledge base, where a set comprises those relations returned from a partially specified match request. Note that the *member* operation is simply a straight-forward search of the knowledge base. Operations on multiple sets returned from the knowledge base, such as *intersection*, are performed by loading the first set into the RAP, and streaming the second set past it. Measured timing performances for set intersection range from less than a millisecond for set sizes of under about 200, to approximately 1 second for set sizes of order 10000 [73]. For sets of 1000 relations, the join operation is estimated (extrapolated from the intersection figures) as approximately 3ms, which is claimed to be about 2 orders of magnitude faster than the Ferranti Relational Processor (FRP) [73].

5.5. Performance Comparisons

5.5.1. SKMS with Simulation

If we compare tables 5.1 and 5.6, the SKMS, running at optimum speed, provides a performance improvement of almost 30 times. Even with the reduced clock rate of 3.3MHz, the SKMS's performance is almost an order of magnitude better than the simulation package. This justifies the effort associated with the project, particularly in view of the opportunities available for further parallelism of SKMS modules. The performance of both systems, however, is decreased by the user interface overheads, and, as was stated above, this area demands further attention.

5.5.2. SKMS with Other Systems

Ferranti Relational Processor

The performance figures presented for the FRP are based on specific examples and so are difficult to interpret and compare with others. However, considering the figure quoted for a straight-forward retrieval of a fully specified relation (table 5.10), it is approximately 4 times slower than the SKMS running at optimum (table 5.6). No FRP times are presented for wildcard searches.

Intelligent File Store

From table 5.11, we see that the IFS search times are in the range $38\mu s \leq R_t \leq 2ms$ depending on the number of wildcards included in the specified relation. If we consider an SKMS interfaced to 64 kbytes instead of its maximum of 512 kbytes (for the 16 bit prototype), then as is illustrated in table 5.11, we have a search time range of $51\mu s \leq R_t \leq 6ms$. Although marginally slower than the IFS at straight-forward searching, the SKMS has three advantages. Firstly, the knowledge structure, itself, supports the construction of frame based objects, with the ability to store different attribute values in different contexts. Secondly, the system incorporates a mark-bit book-keeping circuit, which facilitates set operations such as intersection and join, without the need for any add-on hardware. For a set size of the order of 10000, the IFS Relational Algebraic Processor takes approximately 1 second to perform a set intersection of two relations. The SKMS, running at optimum, would take a maximum of 6 ms per specified relation, *no matter what the set size*, plus 26 μs per matching relation in the intersection set for KB → HOST communication overheads (which could be greatly reduced by a Relation Buffer, as described above). This transforms to a maximum of about 270 ms for the example above. Thirdly, the ability to perform value ranging is extremely advantageous, particularly in applications such as radar interpretation (cf: FRP). The IFS employs a hardware hashing mechanism (Lexical Token Converter [54]) to help attain the fast retrieval times. A similar approach would prove beneficial in the SKMS.

Timing Comparison of Hardware Supported Systems		
search type	*IFS*	*SKMS*
search with no fields unknown	38μs	51μs
search with one field unknown	244μs	51μs → 6ms‡
search with two fields unknown	2ms	51μs → 6ms‡

‡ Considering a 64 kbyte search space per SKMS module (cf: IFS search engine).

Table 5.11: Timing Comparison Of Hardware Supported KB Systems

It is unclear how the IFS copes with garbage collection during information insertion or deletion, or if garbage collection has any effect on the search times quoted. The SKMS, however, performs garbage collection concurrently, so the time required for this function can be defined, and is included in the results presented.

Unfortunately, at the time of writing this thesis, no performance figures were available for the FACT system, comprising Generic Associative Memory (see Chapter 2). Since, this system was designed primarily with set-based operations in mind, it would be interesting to compare performances with the IFS and SKMS. Additionally, from the information available, it is also unclear how this system deals with the problem of garbage collection.

5.6. Summary

The SKMS prototype hardware provides a speed improvement of more than an order of magnitude over the simulation software running on a 10MHz 68010 SBC, and twice that of a Sequent 80386 based Computer. As discussed in Chapter 3, however, the SKMS is an ideal candidate for implementation on an IC, and hence further speed improvements above the optimum performance projected for the prototype, at low cost. Furthermore, faster RAM could be utilised in the Knowledge Base, to take advantage of the potential speed of such a *Relational Processor*.

Performance figures for the SKMS are comparable with those presented for the IFS – one of the major UK projects involved in research into knowledge manipulation engines. Although marginally slower at straight-forward retrieval, the SKMS has several advantages.

1. The knowledge structure facilitates fast traversal through objects and inheritance lattices.

2. Set based operations are supported directly in hardware by a mark-bit book-keeping circuit, and are therefore fast.

3. Searching for values within or outwith a specified range is supported directly by special purpose hardware.

4. Garbage collection is performed concurrently, by way of a pointer book-keeping mechanism, whose operation is invisible to the HOST system, and hence the user.

CHAPTER 6

Conclusions

6.1. Summary

AI systems, particularly Expert Systems, although increasing in popularity, remain unsuitable for many engineering applications due to poor real time response. Research has been concerned with faster knowledge manipulation techniques and enhanced Von Neumann or dedicated hardware systems, such as those described in Chapter 2. Software solutions have concentrated on structured knowledge representations which allow the user to home into a particular piece of information by way of inheritance lattices, and to facilitate reasoning about complex situations (eg: the real world!). Hardware solutions have been varied, but are generally based on First Order Predicate Calculus methods (IFS and FACT). Such approaches are inherently slower than structured formalisms but easier to manipulate. Hardware support for a structured knowledge manipulation system, as described in this thesis, offers an attractive alternative solution.

This thesis has described the research, design, implementation and evaluation of special purpose hardware support for a Knowledge Based System. The Structured Knowledge Manipulation System (SKMS) acts as a co-processor to a VMEbus based HOST, and performs four primitive operators on a Knowledge Base:

- **create** a specified relation
- **modify** a specified relation

- **delete** a specified relation

- **retrieve** a specified relation

Additionally, the Relational Processing Unit described herein provides hardware support for *algebraic relational* (set) operations, and the retrieval of relations with values *between specified bounds.*

The SKMS is based upon the manipulation of the binary representation of a general tree structure, which provides a flexible structured knowledge representation formalism, while maintaining a regular format, which can be exploited by dedicated hardware. Additionally, the binary linked-list format can be used to connect unused memory blocks, and a concurrent, *free-list* garbage collection algorithm with *no memory overheads* and *very little speed penalty* has been implemented within the system.

In conjunction with the Artificial Intelligence Applications Institute [17], a Functional Specification was developed around which a practical Knowledge Based System could be designed (see Chapter 3). Software simulation confirmed that a practical knowledge based system, supporting the knowledge operators and features described in the Functional Specification of Chapter 3, is a realistic proposition, and pinpointed performance limitations. Two limiting factors were isolated:

- **hashing**

- **linked list traversal**

Hashing is a user interface problem which limits the performance of most systems requiring input from a user. However, partly due to time constraints, and partly since the user interface mechanism can remain an independent functional block, whose internal operation is invisible to the rest of the system, the hashing unit can be ignored within the scope of this project (as long as the hashing unit – either software or hardware based – can be up-graded at a later date). The subject was researched briefly to determine whether there were any relatively simple approaches which could be adopted for the purposes of the prototype system, and a summary of this study is presented in Appendix A. In conclusion, the reliability/speed trade-offs encountered dictate that the only satisfactory answer would involve a high speed dedicated hashing engine. The IFS project incorporates a dedicated hashing engine which greatly improves performance. Such an approach would certainly be beneficial to the SKMS.

The other limiting factor, as expected, was found to be linked-list-traversal, and the hardware design strategy, described in Chapter 3, was optimised for handling linked-list codes and pointers in parallel. Garbage collection has presented many difficulties within existing AI software tools, and in several hardware systems designed specifically for Intelligent Knowledge Bases. Such problems were identified in the GRIP project [76]. The linked-list based architecture of the SKMS, however, supports a concurrent garbage collection algorithm abolishing the need for random system interruptions which would degenerate system performance to such an extent that real-time engineering applications would not be viable.

6.2. Performance Evaluation

Two conclusions may be drawn from the performance evaluation. Firstly, with a speed improvement of more than an order of magnitude over the software simulation, the limitations imposed by the knowledge structure have been overcome by the hardware architecture, and the design ideas developed and implemented in the system are proved successful. Secondly, since the retrieval times measured for the SKMS are faster than the Ferranti Relational Processor, and comparable with those quoted for the Intelligent File Store [73,74,75], the system makes an important contribution to research in the field of Knowledge Manipulation Engines. Moreover, the SKMS has several advantages:

1. Fast traversal through related objects (particularly inheritance lattices) is supported by the knowledge structure developed herein.

2. A hardware based mark-bit book-keeping methodology ensures that set based operations on the Knowledge Base are fast.

3. Direct hardware support is provided for searching for values within a specified numerical range.

4. Garbage collection is performed concurrently, does not cause the system to suspend operation to reclaim memory as in most Expert Shells, and is invisible to the user.

5. Other than the Am29C334 Register File, no *state of the art* components were used, and so the SKMS is a relatively low cost system.

6. Since no specific processing elements are used within the design of the Relational Processing Unit, the design is extremely suited for silicon fabrication, and hence reduced system design time, power consumption and further cost reductions.

In conclusion, the SKMS system performance, with integrated garbage collection, is sufficient to provide a low-cost PC/SUN enhancement as a knowledge manipulation engine (co-processor) for real-time engineering applications. For example, contemporary electronic control systems employ loop cycle times of the order of 50ms [72] to 250ms [73], so an up-graded SKMS (see below), if interrogated electronically, could certainly support such an application. Moreover, *second-to-second* applications such as that described in [67], would be particularly well suited to SKMS support.

6.3. The Future

As has been discussed earlier, the Relational Processing Unit is an ideal candidate for implementation in silicon. Such a development would undoubtedly remove the timing problems associated with the present design, caused by the large amounts of wire-wrapped buses necessary in a data dependent system such as the SKMS. Moreover, future development of this project would involve a loosely coupled processing system performing *breadth first search*; employing banks of *relational processors*, since the search and manipulation algorithms, in the present design, are essentially *depth first*, although heuristic techniques are used to improve performance. For example, the assumption (based on probability theory) that *more relations in the Knowledge Base will fail to match the specification than succeed*, is incorporated into the search algorithm. An unexpected, yet practical reason for employing parallel banks of such processors follows from the inadequacy of the prototype SKMS memory space. Since 16 bit pointers are used, the maximum address space is 64k by 64 bits; ie: 512 kbytes. If the system were simply upgraded to 32 bits, although the address space would increase to 4×10^9 bytes, the search time for partially specified relations at the end of this space would become unacceptably long. Consequently, parallel systems of search engines (such as the RPU), each with a smaller address space, are more practical. This approach is taken by the IFS group, who have an address space of only 64 kbytes per search engine.

Figure 6 illustrates the design concepts behind a Parallel Relational Processing System. A central processing unit (HOST) would interface with a user as in the current system, although a dedicated hardware hashing engine would almost certainly be employed in any future design. Several Relational Processors would perform a parallel search operation and would report back to the HOST independently. Memory management of dedicated magnetic storage would be provided locally, so that the specific details of the search operation would remain invisible to the HOST and hence the user.

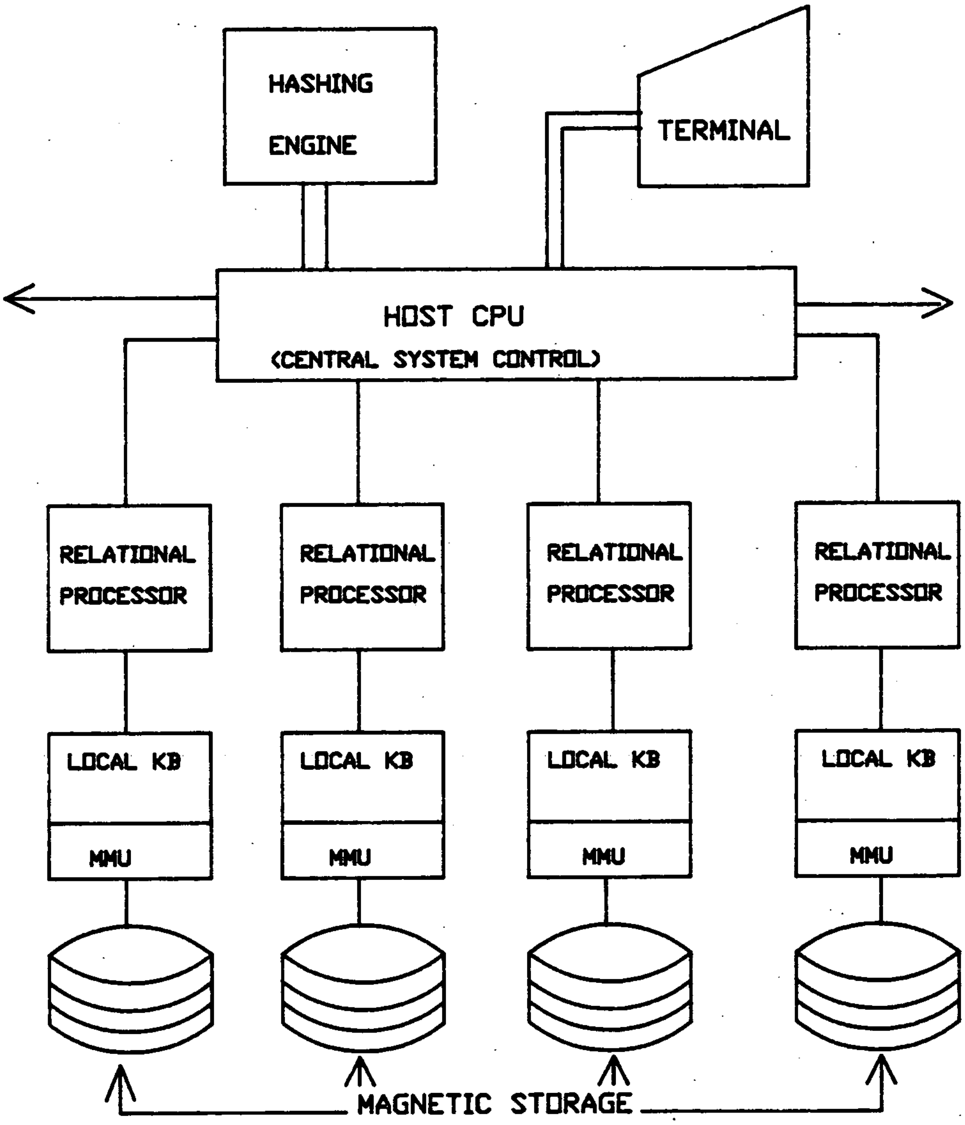


Figure 6: Proposed Parallel Relational Processing System.

APPENDIX A

Hashing Techniques - Overview

A.1. Introduction

Since we don't want to spend all our time in key to code conversion, we must ensure that the hash algorithm is computationally fast and that collisions are kept to a minimum. However, the less complex a hash function is, then the less effective it tends to be and so we get more collisions - a balance must, of course, be struck. Various methods have been proposed including folding, division, mid-square and algebraic coding, or a combination of these (see [65,66] for a good overview); however, the following function (based on division) has been found to be the most effective:

$$h(K) = K \bmod w \tag{A.1}$$

where w is a prime number, K is the key and h(K) is the hash code of K.

It has been noted that different choices of *w* give different performances. Another important conclusion which must be made from all the work that has gone into hashing techniques is that the effectiveness of the hash function depends on the data for which it is being used; some hash functions are very bad at differentiating between similar strings.

A.2. Fibonacci Hashing

Fibonacci Hashing is a technique whereby we multiply the key by a fraction (θ). The best values of θ have been found to lie in the ranges:

$$\frac{1}{4} < \theta < \frac{3}{10}, \ \frac{1}{3} < \theta < \frac{3}{7}, \ \frac{4}{7} < \theta < \frac{2}{3}, \ \frac{7}{10} < \theta < \frac{3}{4}$$

such that:

$$h(K) = \lfloor M(\theta K) \; mod \; 1 \rfloor \tag{A.2}$$

where M is a power of 2 (eg. 0x10000 for a 64k entry hash table).

An alternative is to calculate w, where $w = M\theta$ and proceed as per equation (A.1). We should choose w to be relatively prime (above about 20) and as close as possible to our desired hash-table size.

A.3. Algebraic Coding

This is an interesting method for selecting a suitable value of w which will guarantee hash-code uniqueness for keys which differ by fewer than a predetermined number of bit positions [65,68,69]. We regard the key as a representation of a polynomial K(x) where:

$$K(x) = k_{n-1}x^{n-1} + \cdots + k_1x^1 + k_0 \tag{A.3}$$

We then choose a polynomial P(x) such that:

$$P(x) = x^m + p_{m-1}x^{m-1} + \cdots + p_1x^1 + p_0 \tag{A.4}$$

and:

$$h(K) = K(x) \; mod \; P(x) \tag{A.5}$$

so P(x) represents (in radix 2) an appropriate value for w in equation (A.1).

Assuming we want to convert an *n-bit* key into an *m-bit* hash-code, such that we can guarantee that keys differing in t or fewer bits will have different hash codes (from

[65]), then: given n and t≤n and given an integer k such that

$$n = 2^k - 1 \tag{A.6}$$

P(x) can be found as follows.

- Let S be the smallest set of integers such that {1,2,...,t} ⊆ S, and (2j) mod n ∈ S for all j ∈ S (if n = 15, k = 4, t = 6, we have S = {1,2,3,4,5,6,8,10,12,9}), then:

  $$P(x) = \prod_{j \in S}(x - \alpha^j) \tag{A.7}$$

  where α is an element of order n in the Galois Field (GF) of 2 to the power of k:

  $$\alpha \in_{(order\ n)} GF(2^k) \tag{A.8}$$

A.4. Folding

Folding involves splitting a long key into several shorter parts of length ≤ to the hash code length. Each sub-key is then converted into a separate hash-code. These can then be *fused* into one hash-code, using logical operations such as exclusive-or, or by arithmetic operations. It is clear that a combination of folding and modulus by a prime would be less computationally expensive than dividing by a prime only. For example, if we have a 16 character string, less cpu time is required to split the string into 4 x 4 character keys (4 x 32 bit keys) and then to perform a modulus operation on each of them using a prime as near to 0xFFFF as possible, and finally exclusive-oring the results to return a 16 bit code, rather than performing a modulus operation on the 128 bit key to return a 16 bit code! This method is attractive since it opens the way for parallel computation on the sub-keys, hence providing hash-time speed-up.

A.5. Collisions

The problem of collisions (duplications) between codes occurs in all hashing algorithms to varying extents. There have been many methods proposed for dealing with collisions. However, only the two major methods will be discussed here:

(i) chaining

(ii) open addressing

The **chaining** technique is probably the simpler of the two to understand. The following algorithm (adapted from [65]) can be used to search an M-node table (symbol_table), looking for a given key K. If K is not in the table, and the table is not full, it is inserted.

```
start

(1) [Hash] Set $i \leftarrow h(K) + 1(1 \leq i \leq M)$

(2) [Is this entry used?] If symbol_table[i] is empty, goto (6). (Otherwise
    symbol_table[i] is occupied and must be checked)

(3) [Compare] If K = symbol_table[i].key, return(found), else;

(4) [Advance to next] if symbol_table[i].link $\neq$ 0, set i $\leftarrow$ symbol_table[i].link and goto
    (3)

(5) [Find empty node] The search was unsuccessful, so get R $(1 \leq R \leq M)$ and R is
    such that symbol_table[R] is empty. If $R > M$, then there are no empty nodes
    left and the algorithm terminates with overflow. Otherwise, set
    symbol_table[i].link $\leftarrow$ R and i $\leftarrow$ R

(6) [Insert new key] Mark symbol_table[i] as occupied with symbol_table[i].key $\leftarrow$ K
    and symbol_table[i].link $\leftarrow$ 0.

end
```

The **open addressing** method does not use links between keys with the same hash value and so is less expensive in its use of memory. Entries in the table are searched one by one until the correct key is found. A *probe sequence* is any rule which determines which table positions should be checked when a collision occurs. An example of such a sequence is *linear probing* which uses the cyclic sequence:

$$h(K),\ h(K)-1, \ldots,\ 0,\ M-1,\ M-2, \ldots,\ h(K)+1 \tag{A.9}$$

Linear Probing is described in the following algorithm (from [65]). The algorithm searches an M-node table (symbol_table) for a key, K. If K is not found, then it is inserted (if the table is not full).

```
start

(1) [Hash] Set $i \leftarrow h(K) (0 \leq i < M)$

(2) [Compare] If symbol_table[i].key = K, algorithm terminates successfully.
    Otherwise, if symbol_table[i] is empty, goto (4)

(3) [Advance to next] Set $i \leftarrow i - 1$; if now $i < 0$, set $i \leftarrow i + M$. Go back to (2)

(4) [Insert] (The search was unsuccessful.) If $N = M - 1$ (where N = no of occupied
    nodes), the algorithm terminates with overflow. Otherwise, set $N \leftarrow N + 1$, mark
    symbol_table[i] as occupied and set symbol_table[i].key $\leftarrow$ K.

end
```

Another similar technique uses the following probe sequence:

$$h_1(K),\ h_1(K) + h_2(K),\ h_1(K) + 2h_2(K), \ldots \tag{A.10}$$

The second hash (h_1) should be chosen to be relatively prime to the first (h_2).

A.6. Evaluation

Lum et al [66] provide several experimental results for a variety of hashing functions using both chained and open addressing collision resolving methods and conclude that open addressing is not a suitable technique for tables with fewer than 10 collisions per key, but can be applied to larger sizes using less storage space than chaining but with comparable performance.

A series of experiments was carried out to study the effectiveness of various hash functions. The primary objective was to discover functions which could perform their required task effectively, while using a minimum amount of CPU time. The secondary requirement was to look for opportunities for speed-ups by hardware support. Two different sets of data were used:

(i) the UNIX on-line dictionary

(ii) a file (FIRST100.kb) containing similar strings ("Person00, Person01...") and also with a high numerical content

It was decided to employ a chained collision resolving method since large numbers of collisions were neither desired nor expected and so open addressing would be less efficient in terms of computation time. All of the hash functions, except for the UNIX *spell* program hash function, are based on the following algorithm. Keys are allowed to be up to 16 characters, ie: 128 bits, long (longer ones being truncated).

start

(i) *[sub-divide problem]* Break the key up into four sub-keys, each four characters long (32 bits); denoted code1,code2,code3 and code4.

(ii) *[optional pre-coding]* The modulus by a relative prime of each code is taken to return four 16 bit codes.

(iii) *[combine sub-codes]* code1 ← code1 exor code2, and; code3 ← code3 exor code4, then either; code1 ← code1 exor code3, or; code1 ← |(code1 + code3)|.

(iv) *[optional post-coding]* The modulus by a relative prime is taken of the resultant code (either 16 or 32 bits, depending on stage (ii)).

end

Almost all the hash functions derived from this algorithm gave collision counts in the range 16% → 19% of the total table size when used to hash the on-line dictionary, and approximately 11% when used to hash FIRST100.kb. The exception was when 0xFFFE (65534) was chosen as the modulus. In this case, collision counts of about 26% where noted. It was also noticed that adding codes 1 and 3 in stage (iii), rather than exclusive-or-ing them, above made little difference to the collision count. This is fortunate, since addition is the more expensive operation. The collision count for the dictionary was at its lowest for this algorithm if stage (ii) was included with modulus 61259. However, not a great deal of difference was made to the count if stage (ii) was left out and 61259 was selected as the modulus in stage (iv).

The *spell* program on UNIX makes use of a very complicated hash function. Again, it is based on the "modulus by a prime" technique, but *eleven* different primes are used and the eleven codes are combined to return a highly unique code. When this function was used to hash the on-line dictionary, the collision count was found to be < 3%. The collision count for the data in FIRST100.kb was approximately 1%. The eleven primes have been specifically chosen for the application and when these were changed for different values, the collision count rose to almost 100%. The main drawback with this method is that the hash function takes a very long time to calculate in comparison with those described above.

It should be clear that hashing techniques are not cut and dried in their operation and the performance is usually dependent on the data being hashed. It should also be clear that we want to return as unique a code as possible to save on collision resolving, but that the time required to compute the code should be as small as possible: certainly a lot quicker than performing a sequential search!

An obvious opportunity for speeding up the hashing operation is the parallel execution of the "modulus by a prime" calculations on the separate sub-keys. The performance improvement gained from such a system is unlikely to justify the cost, although further investigation is warranted.

APPENDIX B

SKMS Signals - Summary

$\overline{AS}$ Address Strobe, supplied by the HOST via the **VME bus**, to signal that a valid address exists on the VME address bus.

BA_i The address bus supplied by the Relational Processing Unit to the Knowledge Base, via the **LOCAL bus**.

BA_0 RPU address line zero is supplied specifically by the microprogram (μD45), and selects either the least or the most significant 32 bits of a 64 bit knowledge structure property/value block.

BD_i These signals constitute the 32 bit **LOCAL bus** data interface between the Relational Processing Unit and the Knowledge Base.

$BADDR1$ This is the 6 bit address supplied by the **Microprogram** (μD50-55) to the Register File, specifying the location to be *read* by the Relational Processing Unit.

$BADDR2$ This is the 6 bit address supplied by the **Microprogram** (μD56-61) to the Register File, specifying the location to be *written to* by the Relational Processing Unit.

$\overline{BAE}$ Enables the **LOCAL bus** address multiplexer for communications between the Relational Processing Unit and the Knowledge Base. It is derived in the VME Interface Circuit.

$\overline{BOE}$

The Knowledge Base Output Enable signal, supplied by the Relational Processing Unit, derived from the **Microprogram** (μD47).

$\overline{BWE}$

The Knowledge Base Write Enable signal, supplied by the Relational Processing Unit, from the **Microprogram** (μD47). $\overline{RF\_SEL}$) are derived from it.

$\overline{board\_sel}$

This is an intermediate signal (VME Interface Circuit), which is active if the HOST wishes to access either the Microprogram Store, Register File, or Knowledge Base. The other memory select signals ($\overline{V}/B$, $\overline{MPS\_SEL}$ and

$\overline{B\_DATA\_EN}$

This signal enables the Register File inputs from the Knowledge Base via a multiplexer and the **LOCAL bus** – supplied by the **Microprogram** (μ49).

B_DATA_SEL

This signal selects between BD_{0-15} and BD_{16-31} from the **LOCAL bus** – supplied by the **Microprogram** (μ48).

$B\_KB\_IE(L)$

Relational Processing Unit → Knowledge Base (BD_{0-15}) I/O Port Input Enable – supplied by the **Microprogram** (μD62) .

$B\_KB\_IE(U)$

RPU→KB (BD_{16-31}) I/O Port Input Enable – supplied by the **Microprogram** (μD66).

$B\_KB\_OE(L)$

RPU→KB (BD_{0-15}) I/O Port Output Enable – supplied by the **Microprogram** (μD67).

$B\_KB\_OE(U)$ RPU-KB ($BD_{16\text{-}31}$) I/O Port Output Enable – supplied by the **Microprogram** (μD67).

$C1$ Clock output from the Clock Generator circuit.

$C2$ Clock output from the Clock Generator circuit.

$C3$ Clock output from the Clock Generator circuit.

$C4$ Clock output from the Clock Generator circuit.

$CCMUXA$ Condition Code MUX Select Line – supplied by the **Microprogram** (μD29).

$CCMUXB$ Condition Code MUX Select Line – supplied by the **Microprogram** (μD30).

$CCMUXC$ Condition Code MUX Select Line – supplied by the **Microprogram** (μD31).

$CCMUXPOL$ Condition Code MUX Output Polarity Control – supplied by the **Microprogram** (μD32).

$\overline{CC}$ The Condition Code Input to the Sequencer – supplied by the Condition Code Select Circuit.

$\overline{CLR}/UP$ If low, this signal clears the contents of the Pointer Store counter (74AS867). If high, the Pointer Store counter increments (if enabled). It is supplied by the **Microprogram** (μD69).

$\overline{CLR\_MARK}$ Clears BD_7, which is the mark bit in the status word of the property and value blocks.

CMP_ILE Enables the comparator (74AS866) data inputs – derived from C2.

$\overline{COUNT}$ Enables the Pointer Store counter (74AS867) – supplied by the **Microprogram** (μD68).

CSR_i Used to denote the Control and Status Register. The CSR is located in the Register File, and is loaded into a latch (74ALS874 - shown in the Sequencer Circuit) at a rising clock edge if $\overline{Latch\_Clk\_En}$ is active.

$\overline{DTACK}$ The *Data Transfer Acknowledged* signal supplied by the VME Interface Circuit to the HOST to report a successful VME access to the SKMS.

EQ The code comparator (74AS866) *equal to* output.

$\overline{EQ}$ The code comparator (74AS866) *not equal to* output.

Fo The fundamental clock frequency signal (20MHz).

GT The code comparator (74AS866) *greater than* output.

$\overline{HALT}$ Suspends SKMS operation if the HOST accesses the Knowledge Base via the **VME bus**.

$HI/\overline{LO}$ Selects BD_{0-15} or BD_{16} inputs to Code Comparator – supplied by the **Microprogram** (μD43).

$KB\_B\_IE(L)$ Knowledge Base → Relational Processing Unit (BD_{0-15}) I/O Port Input Enable – supplied by the **Microprogram** (μD64) .

$KB\_B\_IE(U)$ KB→RPU (BD_{16-31}) I/O Port Input Enable – supplied by the **Microprogram** (μD64) .

KB_B_OE(L) KB→RPU (BD_{0-15}) I/O Port Output Enable – supplied by the **Microprogram** (μD70) .

KB_B_OE(U) KB→RPU (BD_{16-31}) I/O Port Output Enable – supplied by the **Microprogram** (μD65) .

KB_DAT_IN(H) Disables the multiplexed Register File or Pointer Store output to the I/O Port (BD_{16-31}) – supplied by the **Microprogram** (μD62).

L1 Clock Generator Function Select Signal – supplied by the **Microprogram** (μD34).

L2 Clock Generator Function Select Signal – supplied by the **Microprogram** (μD35).

L3 Clock Generator Function Select Signal – supplied by the **Microprogram** (μD36).

LT The code comparator (74AS866) *less than* output.

$\overline{Latch\_Clk\_En}$ Enables the Control and Status Register Latch in the Sequencer Circuit – supplied by the **Microprogram** (μD33).

$\overline{MPS}/R\_W$ Defines whether the Microprogram Store acts as a block of read/write RAM, accessible by the HOST, or whether it is a block of read-only memory, accessible only by the Relational Processing Unit – supplied as a debounced input from a toggle switch.

$\overline{MPS\_SEL}$ HOST → Microprogram Store Access Select Line – derived in the VME Interface Circuit.

PLA/B_Sel_Or Selects either the A or B address from the Pipeline to the Sequencer – supplied by the **Microprogram** (μD28), and works in conjunction with $\overline{PTR\_ZERO}$.

$PLAddrA_i$ Sequencer jump address A – supplied by the **Microprogram** (μD4→15).

$PLAddrB_i$ Sequencer jump address B – supplied by the **Microprogram** (μD16→27).

$PS/\overline{RF}\_ADDR$ Selects between the Register File and Pointer Store address outputs to the Knowledge Base via the **LOCAL Bus** ($BA_{1\to16}$) – supplied by the **Microprogram** (μD46).

PSAddr The Pointer Store address – supplied by the **Microprogram** (μD72→75).

$\overline{PS\to RF\_EN}$ Enables the Pointer Store → Register File bus buffer – supplied by the **Microprogram** (μD78).

PS_DATA Denotes the data output bus from the Pointer Store.

$\overline{PS\_DAT\_IN\_EN}$ Enables the multiplexed $BD_{16\to31}$ or Register File inputs to the Pointer Store – supplied by the **Microprogram** (μD79).

PS_DAT_IN_SEL Selects either the multiplexed $BD_{16\to31}$ or Register File inputs to the Pointer Store – supplied by the **Microprogram** (μD80).

$PS\_\overline{R}/W$ The Pointer Store $\overline{read}$ or *write* signal – supplied by the **Microprogram** (μD71).

$\overline{PTR\_ZERO}$ This output signifies whether or not the comparator pointer input was NULL.

$\overline{RESET}$ Resets the Relational Processing Unit – the debounced output from a toggle switch.

$\overline{RF}/PS$ Selects between the Register File or Pointer Store output to the I/O Port (BD_{16}) – suplied by the **Microprogram** (μD44).

RF_DOUT The data output from the Register File to the Relational Processing Unit.

$\overline{RF\_SEL}$ Selects the Register File for access by the HOST via the **VME bus**.

$\overline{RF\_WT}$ Register File *read* or $\overline{write}$ line – supplied by the **Microprogram** (μD39).

SET_MARK Sets BD_7, which is the mark bit of the status word in a property or value block – supplied by the **Microprogram** (μD40).

ST_BD_i The masked data (status word) output from the Status Control Circuit.

SYS_CLK HOST system clock, used to clock the $\overline{DTACK}$ delay line (shift register) in the VME Interface Circuit.

TST_STAT Status Control Circuit masks the **LOCAL bus** data input if set – supplied by the **Microprogram** (μD42).

VA_i VME address bus.

VD_i VME data bus.

$\overline{V}/B$ Defines whether HOST (V) or Relational Processing Unit (B) has control of the Knowledge Base – derived from the VME Interface Circuit.

$\overline{VOE}$ Knowledge Base → HOST output enable signal – derived from $\overline{VWE}$ in the VME Interface Circuit.

$\overline{VWE}$ HOST → Knowledge Base write enable signal – supplied by the HOST via the **VME bus**.

WILDMUXA Select line for the wildcard MUX in the Condition Code Circuit – supplied by the **Microprogram** (μD37).

WILDMUXB Select line for the wildcard MUX in the Condition Code Circuit – supplied by the **Microprogram** (μD38).

μA_i The Microprogram Store address bus – supplied by the Sequencer in the RPU.

μD_i The Microprogram Control word.

APPENDIX C

The Microprogram Control Word

μD00	I0	μD24	PLAddrB8	μD48	B_DATA_SEL	μD72	PSAddr0
μD01	I1	μD25	PLAddrB9	μD49	$\overline{\text{B\_DATA\_EN}}$	μD73	PSAddr1
μD02	I2	μD26	PLAddrB10	μD50	BADDR1-0	μD74	PSAddr2
μD03	I3	μD27	PLAddrB11	μD51	BADDR1-1	μD75	PSAddr3
μD04	PLAddrA0	μD28	PLA/BSel_Or	μD52	BADDR1-2	μD76	$\overline{\text{PS->CTR}}$
μD05	PLAddrA1	μD29	CCMUXA	μD53	BADDR1-3	μD77	$\overline{\text{PS->RF\_EN}}$
μD06	PLAddrA2	μD30	CCMUXB	μD54	BADDR1-4	μD78	PS_DAT_IN_SEL
μD07	PLAddrA3	μD31	CCMUXC	μD55	BADDR1-5	μD79	$\overline{\text{PS\_DAT\_IN\_EN}}$
μD08	PLAddrA4	μD32	CCMUXPOL	μD56	BADDR2-0		
μD09	PLAddrA5	μD33	$\overline{\text{LatchClkEn}}$	μD57	BADDR2-1		
μD10	PLAddrA6	μD34	L1	μD58	BADDR2-2		
μD11	PLAddrA7	μD35	L2	μD59	BADDR2-3		
μD12	PLAddrA8	μD36	L3	μD60	BADDR2-4		
μD13	PLAddrA9	μD37	WILDMUXA	μD61	BADDR2-5		
μD14	PLAddrA10	μD38	WILDMUXB	μD62	$\overline{\text{B-KB-IE(H)}}$		
μD15	PLAddrA11	μD39	$\overline{\text{RF\_WT}}$	μD61	KB_DAT_IN		
μD16	PLAddrB0	μD40	SET_MARK	μD64	$\overline{\text{KB-B-IE}}$		
μD17	PLAddrB1	μD41	$\overline{\text{CLR\_MARK}}$	μD65	$\overline{\text{KB-B-OE(H)}}$		
μD18	PLAddrB2	μD42	$\overline{\text{TST\_STAT}}$	μD66	$\overline{\text{B-KB-IE(L)}}$		
μD19	PLAddrB3	μD43	HI/$\overline{\text{LO}}$	μD67	$\overline{\text{B-KB-OE}}$		
μD20	PLAddrB4	μD44	$\overline{\text{RF}}$/PS	μD68	$\overline{\text{COUNT}}$		
μD21	PLAddrB5	μD45	BA0	μD69	$\overline{\text{CLR}}$/UP		
μD22	PLAddrB6	μD46	PS/$\overline{\text{RF}}$_ADDR	μD70	$\overline{\text{KB-B-OE(L)}}$		
μD23	PLAddrB7	μD47	$\overline{\text{BWE}}$	μD71	PS_$\overline{\text{R}}$/W		

APPENDIX D
UNIX Shell Scripts

D.1. ksma – Knowledge System Micro-Assembler

```
set -e
#       This shell script has been written to parse the flags passed
#       to the uasm program and to remove any intermediate files
#       created during processing
#
echo        "KSMA.......KNOWLEDGE-SYSTEM MICRO-ASSEMBLER"
echo        "Author:    Steve Hudson"
echo        "Date:      19th June 1989"
echo        "Version:   2.1"

#
#       check for any bad flags before passing them onto the uasm program
#
for i in $*
do
        case $i in
        -T)     tflag=1;;
        -L)     lflag=1;;
        -E)     eflag=1;;
        -M)     mflag=1;;
        -*)
                echo $i "- Bad parameter!"
                echo "Usage:"
                echo "ksma [-T] [-L] [-E] [-M [<macrofile>.mac]] <file>.d"
                exit;;
        *.d)        infile=$i;;
        *.mac)          macfile=$i;;
        *)
                echo $i "- Bad filename!"
                echo "Usage:"
                echo "ksma [-T] [-L] [-E] [-M [<macrofile>.mac]] <file>.d"
                exit;;
        esac
done
#
```

```
#
#       run the macro preprocessor if necessary
#
if
        mflag=1
then
        macro=${macfile=$HOME/lib/lib.mac}
        echo ""
        echo "MP -- MACRO-PREPROCESSOR"
        echo ""
        if
                mp $macro $infile
        then
#                run the micro assembler
                uasm ${eflag+-E} ${lflag+-L} ${mflag+-M} $infile
        else
                exit
        fi
fi
#
#       remove the input file (with specific error information, unless
#       the -E option is specified
#
if
        tflag=1
then
        echo "## intermediate microcode written to mic.tmp ##"
else
        rm mic.tmp
fi
if
        eflag=1
then
        echo "##   parsed    microcode written to mic.out ##"
else
        rm mic.out
fi
```

D.2. sysld

sysld calls two other shell scripts which load the microprogram and application programs into the MPS and HOST respectively.

```
$HOME/TEST_CODE/LOAD
rm b.out SREC
$HOME/SKMS/LOAD
```

D.3. Load the Microcode.

```
pmeld -R 3000 $HOME/TEST_CODE/copy.b
stty -echo
sleep 4
echo BF 1000 2FFE 0
sleep 3
echo G 3000
sleep 1
stty echo
pmeld -R 1000 $HOME/TEST_CODE/microprog.b
stty -echo
echo "#### Microcode has been loaded at 0xC00000 ####"
sleep 5
echo G 3000
stty echo
```

D.4. Load the SKMS Application Program.

```
pmeld -R 4000 $HOME/SKMS/b.out
stty -echo
echo "#### HOST software has been loaded at 0x4000 ####"
stty echo
sleep 4
```

D.5. pmeld

```
#       The linked file is converted from raw object code to S-RECORD
#       format ready for downloading to the FORCE CPU board.
#       The Base address of the code may be specified by the environment
#       variable CC68BASE or default to value $1000

base=${CC68BASE=1000}                   # Default Base Address $1000
ld68 -R $base $* -lsup -lc          # Link files and C-library

ml68 -o SREC
stty -echo
echo
echo "Switch to UNIX-FORCE connection [position: 3]"
sleep 4
echo LO1
cat SREC
stty echo
echo    "#### Turn switch to position 2     ####"
stty echo
sleep 2
```

APPENDIX E

Microprogram Language and Assemblers

E.1. μa10A – The Language

The microprogram language ($\mu a\,10A$) was developed in a structured fashion to correspond to the gradual development and construction of the Structured Knowledge Manipulation System (SKMS). Each microinstruction takes the following form:

{

<IN> defines sequencer $I_{0\text{-}3}$ inputs.

<CC> controls the Condition Code Selector.

<CL> defines the clock speed ($L\,1$, $L\,2$, and $L\,3$).

<CM> controls the Parallel Comparator Circuit.

<KB> controls the KB and I/O Port.

<PS> controls the Pointer Store Circuit.

<RF> controls the Register File.

<ST> controls the Status Control Circuit.

<WD> defines the inputs to the Wildcard MUX.

}

E.1.1. Sequencer Commands <IN>

- **cont jmap jz cjmp c2jmp cjsr c2jsr rtn**

These are the commands recognised by the sequencer and decide what the *next* value of the Program Counter (PC) will be. The User Stack (US) allows the nesting of subroutine calls up to eight levels deep; [US] denotes the contents of the User Stack in the description below.

<IN> **cont** PC = PC + 1

<IN> **jmap** PC = CSR(b3 - b0)

the program counter takes its value from the least significant nibble of the Control and Status Register.

<IN> **jz** PC = 0

<IN> **cjmp addr**

if $\overline{CC}$ bit is low: PC = addr stack (US) unchanged

<IN> **cjsr addr**

if $\overline{CC}$ bit is low: [US] = PC; PC = addr; US = US + 1

<IN> **c2jmp addrl addr2**

if $\overline{CC}$ bit is low, and ptr_zero = 0, then PC = addrl

if $\overline{CC}$ bit is low, and ptr_zero = 1, then PC = addr2

if $\overline{CC}$ bit is high, then PC = PC + 1

<IN> **c2jsr addrl addr2**

if $\overline{CC}$ bit is low, and ptr_zero = 0, then [US] = PC; PC = addrl; US = US + 1

if $\overline{CC}$ bit is low, and ptr_zero = 1, then [US] = PC; PC = addr2; US = US + 1

if $\overline{CC}$ bit is high, then PC = PC + 1

<IN> **rtn** US = US – 1; PC = [US]

E.1 2. Condition Code Input Select <CC>

There are eight possible inputs to the $\overline{CC}$ input of the sequencer, and are selected via an 8→1 input MUX with polarity control (ccneg or ccpos). We can force a fail or a pass of the $\overline{CC}$ test using the commands

force pass and *force fail*

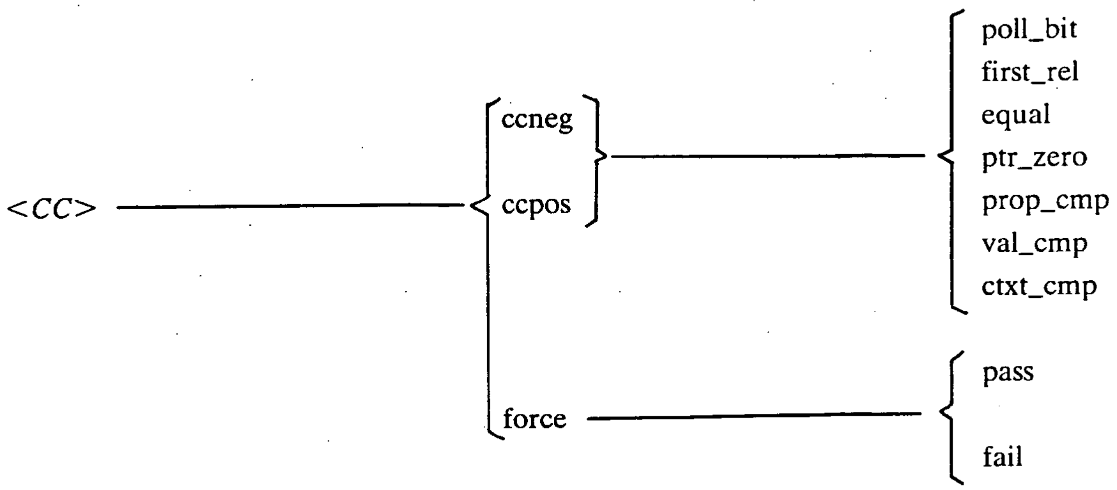
We can test the *equal* output of the Code Comparator (used for matching status info) and the *ptr_zero* output of the Pointer Comparator. The *poll_bit* from the CSR is also an input to the CC MUX and can be tested:

ccneg/ccpos equal/ptr_zero/poll_bit

If we are matching property, value or context codes, then we can decide whether we want the retrieved code to be greater than, less than or equal to the specification code or a wildcard (ie: always matches). This is performed via an interaction between the CSR and a series of MUXs. The outputs from these interactions are input to the CC MUX as: *prop_cmp, value_cmp* and *ctxt_cmp.*

- **Command Summary:**

Identifier **Command** **Argument(s)**



E.1.3. Clock Speed <CL>

The clock speed can be varied by altering some bits in the microcode. The commands:

<CL> *fast, medium* or *slow*

are used to select cycle times of 200ns, 250ns and 300ns respectively for the *next* clock cycle.

E.1.4. Code Comparator Input Select <CM>

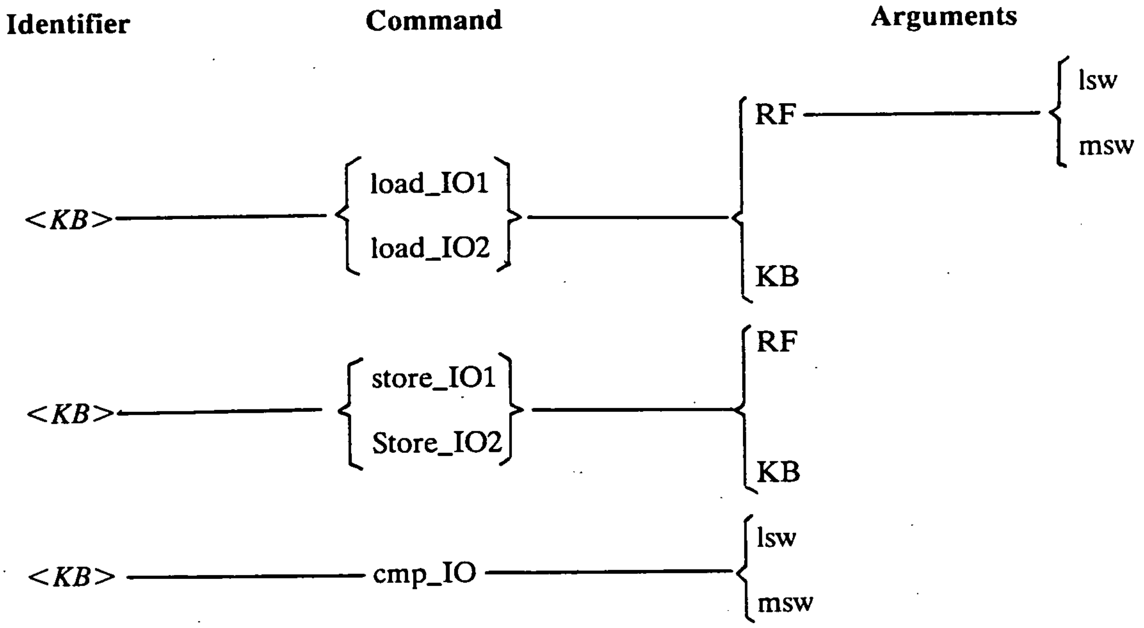
The Q input to the Code Comparator can come either from the least significant 16 bits of the data word (property, value or status words) or from the most significant 32 bits of the data word (context). This is controlled simply by:

<CM> *lsw* or *msw*

E.1.5. Knowledge Base and I/O Port Control <KB>

This area is controlled entirely by the state of the I/O Ports. We can load the ports from either direction and output their contents onto the data bus in either direction. Either the first or second 32 bits(*lsw* or *msw*) of a data block can be read from or written to the knowledge base by the I/O Port.

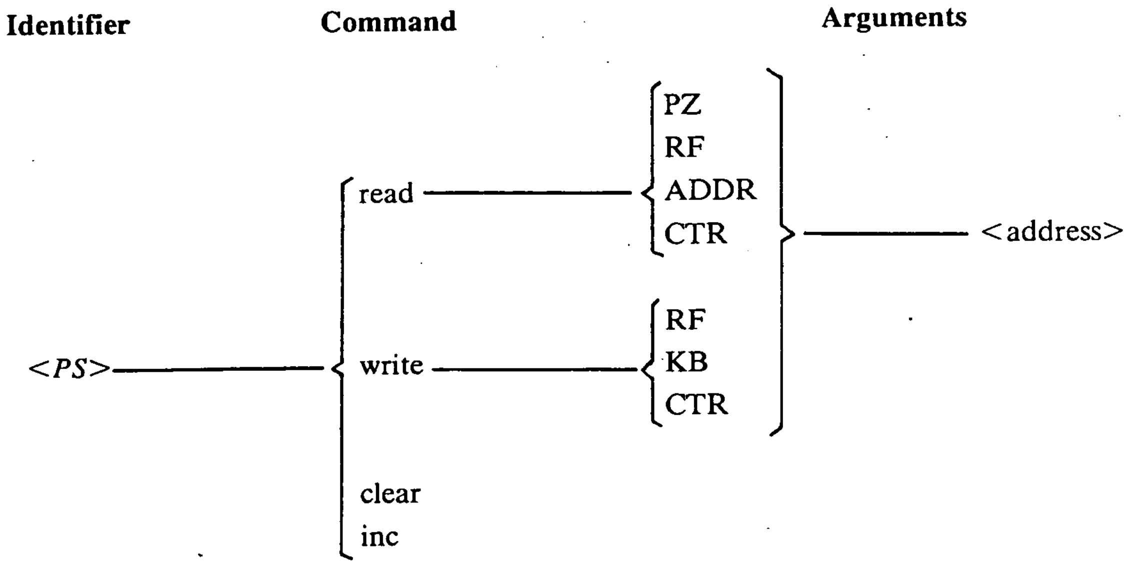
- **Command Summary**



E.1.6. Pointer Store <PS>

The Pointer Store can be written to or read by the I/O Port (msw) and the Register File. The output is also connected to the input of the Zero Pointer Comparator and to the A input of the KB address MUX. A buffered, counter is connected between the PS input and output buses, so that the mem_ptr can be incremented.

- **Command Summary:**



E.1.7. Register File Control <RF>

The Register File can be written to by either the *lsw* or *msw* I/O Ports (*write_KBL* and *write_KBH*) and by the Pointer Store (*write_PS*). It can be *read* by the Pointer Store, Knowledge Base, Knowledge Base Address MUX, and the Parallel Code Comparator, simultaneously.

- **Command Summary:**

Identifier	Command	Arguments
<RF>	read write_PS write_KBL write_KBH	<address>

E.1.8. Status Control Circuit <ST>

When the RPU is matching the status word in a property or value block, there are several operations which can be carried out. If $\overline{TST\_STAT}$ is active, then the status byte (bits 9 - 15) is not masked, and so are *ANDed* with the specification status bits prior to matching. This enables the system to examine only those bits which are considered relevant by the HOST. If $\overline{TST\_STAT}$ is not active, then bits 9 - 15 are left unchanged. The status mark bit (bit 8) can be cleared prior to matching, or either set or cleared prior to writing to the KB. Alternatively, it can remain unchanged.

- **Command Summary:**

Identifier	**Command**
<ST>	test set clear

E.1.9. Wildcard Codes <WD>

Bits in the Control and Status Register (CSR) can be used to ensure that a $\overline{CC}$ test fails ($\overline{CC}$ = 1) if a particular match is being performed – either property, value or context. Note that the code is written in such a way that a test fails (PC = PC + 1) if a match is successful.

- **Command Summary:**

Identifier	**Command**
<WD>	prop val ctxt

E.2. μa10A – The Micro-Assembler.

The micro-assembler translates μa10A microprograms into low-level microinstructions (see Appendix C). It accepts a microprogram source (suffix .d) and performs two passes creating two intermediate output files (suffix .mic and .s). A sub-routine library file (*lib.sub*) is appended to the microprogram before the assembler passes are initiated. lib.sub is listed in Appendix G. The first pass checks the program for the correct syntax, and outputs any error messages. The second pass calculates jump and sub-routine addresses. The first intermediate file (.mic) is a list of the microinstructions in hexadecimal format, the second intermediate file (.s) is passed to 68000 assembler (Motorola syntax), which creates an microprogram object file (suffix .b). This file is converted into S-record format by *pmeld*, and down-loaded to the Microprogram Store by *sysld* (UNIX shell scripts listed in Appendix D).

E.2.1. Invocation.

Name: *μal0A*

Synopsis: *μal0A [-L] [-M [<macrofile.mac>]] file.d*

Description:

μal0A translates *file.d* from the μal0A syntax into intermediate files (*file.mic, file.s*). *file.s* can be assembled into 68000 object code and downloaded via a HOST CPU to a Microprogram Store.

-L:

[optional] causes the symbol table, relating microprogram labels to addresses, to be output to *stdout*.

-M:

[optional] indicates that the input has been filtered through *mp*, a macro pre-processor, and therefore accepts the intermediate file (*mic.tmp*) as input instead of file.d.

E.3. The macro-preprocessor

The macro-preprocessor allows the programmer to define macros composed of μal0A instructions. mp accepts two arguments, the input file and a macro definition file. *lib.mac* is a library of macros created specifically for the SKMS system, and are listed in Appendix G. Macro definitions take the form:

```
#defmac      macro_name      [optional reference number]
{
|      Optional Comments
       <IN>    command    [.label1]    [.label2]
       <CC>
       <RF>
```

```
    <KB>

    <PS>

    <CM>

    <CL>

    <WD>

    <ST>

}
#endmac
```

where .label1 and .label2 accept jump addresses from the macro call (if supplied). The output file (*mic.tmp*) is normally passed to μa10A for micro-assembly.

E.4. ksma – The Knowledge System Micro-Assembler.

ksma is a UNIX shell script (listed in Appendix D) which controls the execution of mp and μa10A.

E.4.1. Invocation.

Name: *ksma*

Synopsis: *ksma [-T] [-E] [-L] [-M [<macrofile.mac>]] file.d*

[optional] passed to μa10A.

-M [<macrofile.mac>]:

[optional] causes the macro-preprocessor, mp, to be called. If a file is specified (suffix .mac) it is searched for the appropriate macros, otherwise, the default (lib.mac) is used.

-T:

[optional] causes the output file from mp (mic.tmp) , which is normally deleted, to be retained after execution.

-E:

[optional] causes *mic.out*, an intermediate file containing μa10A error messages, which is normally retained, to be deleted after execution.

APPENDIX F

Microprogram Listing

```
|   #####
|
|               KNOWLEDGE BASE SYSTEM MICROCODE
|
|               Author: S. Hudson
|
|               Download via the host CPU board to the WCS
|               Location:    0xC00000
|

|   Program startup sequence involves polling a control bit in the
|   Control and Status Register (CSR) of the Register File.
|   Once this is done, the Pointer Store is initialised in preparation
|   for the next command. A copy of the MEM_PTR is kept by the host CPU

|   #####

start:
    jsr     poll_cpu
    jsr     init

|   Jump to the start of the required procedure....
|   The address address is supplied by the least significant nibble of
|   the CSR.

    jmap

|          C_new_obj          location 0x0003
    jmp     C_new_obj

|          C_new_rel          location 0x0004
    jmp     C_new_rel

|          Retrieve_rel       location 0x0005
    jmp     Retrieve_rel

|          Modify_rel         location 0x0006
```

```
     jmp     Modify_rel

|            Delete_rel          location 0x0007
     jmp     Delete_rel

|            Retr_all            location 0x0008
     jmp     Retr_all

|            Init_PS             location 0x0009
     jmp     Init_PS

|    #####
|    START OF PROCEDURES....

|    This routine simply initialises the free_ptr in the Pointer Store

Init_PS:
     clear_PS     free_ptr
     jsr          clear_poll
     jmp          start

|    #####

|    This routine creates new objects with an associated
|    relation (OBJECT-->PROPERTY-->VALUE+CONTEXT)

C_new_obj:
     clear_PS     error_cond
     clear_PS     vlast
     clear_PS     vthis
     clear_PS     vnext

     clear_PS     plast
     clear_PS     pthis
     clear_PS     pnext

     jsr          create_P_space
     move_pnext_pthis
     get_1st_prop_ptr

     jsr          create_V_space
     move_vnext_vthis

     store_prop_KB_1
```

```
    store_prop_KB_2
    store_val_KB_1
    store_val_KB_2

    jsr       clear_poll
    jmp        start

|   #####

|   This routine appends new relations to existing objects

C_new_rel:
    clear_PS    error_cond
    clear_PS    vlast
    clear_PS    vthis
    clear_PS    vnext

    clear_PS    plast
    clear_PS    pthis
    clear_PS    pnext

    get_1st_prop_1

C_match_prop:
    match_prop_name         new_prop    C_get_next_p

|   if they match, then PC = PC + 1
|   if they don't match and pnext == 0, goto new_prop
|   if they don't match and pnext != 0, goto C_get_next_p

    get_next_prop_2

|   retrieve the second word of the current property

    match_prop_status       new_prop    C_get_next_p

|   if they match, then PC = PC + 1
|   if they don't match and pnext == 0, goto new_prop
|   if they don't match and pnext != 0, goto C_get_next_p

    get_1st_val_1
```

```
|    retrieve the first word of the first value in the list associated
|    with the current property

C_match_val:
     match_val_name          new_val     C_get_next_v

|    if they match, then PC = PC + 1
|    if they don't match and vnext == 0, goto new_val
|    if they don't match and vnext != 0, goto C_get_next_v

     get_next_val_2

|    get the second word of the current value

     match_val_status        new_val     C_get_next_v

|    if they match, then PC = PC + 1
|    if they don't match and vnext == 0, goto new_val
|    if they don't match and vnext != 0, goto C_get_next_v

     match_val_ctxt          new_val     C_get_next_v

|    if they match, then PC = PC + 1
|    if they don't match and vnext == 0, goto new_val
|    if they don't match and vnext != 0, goto C_get_next_v

|    clear the contents of the CSR to inform the CPU of the end of the
|    instruction, and to prepare for the next instruction

Cstop:
     jsr        clear_poll
     jmp         start

|    Return to the CPU Polling sequence at the start of the code

|    #####

|    This routine searches the Knowledge Base until either the 1st matching
|    relation is found, or the search terminates unsuccessfully
|    Wildcard values are allowed for the property, value and context
|    codes.
```

```
Retrieve_rel:
    clear_PS     error_cond

    get_1st_prop_1

R_match_prop:
    match_prop_name        Rstop        R_get_next_p
    get_next_prop_2
    match_prop_status      Rstop        R_get_next_p
    get_1st_val_1

R_match_val:
    match_val_name         Rfailed      R_get_next_v
    get_next_val_2
    match_val_status       Rfailed      R_get_next_v
    match_val_ctxt         Rfailed      R_get_next_v

Rpassed:
    test_ALL_flag          Cnt_Rel
    jsr         clear_poll
    jmp          start

Rfailed:
|   jumps to "Rstop" if prop is not a wildcard
    jmp_wild               Rstop        prop

|   jumps to R_get_next_p if pnext is != 0
    match_PS_zero          pnext        R_get_next_p

Rstop:
    jsr         ERROR
    jsr         clear_poll
    jmp          start

Cnt_Rel:
    jsr         INC_REL

Retr_all:
    match_PS_zero          vnext        R_get_next_v
    match_PS_zero          pnext        R_get_next_p

    jsr         ERROR
    jsr         clear_poll
    jmp          start
```

```
|   #####

|   This routine works in a similar manner to Retrieve_rel, except
|   that the new value is substituted when a match is found.

Modify_rel:
    clear_PS    error_cond
    get_1st_prop_1

M_match_prop:
    match_prop_name       Mstop       M_get_next_p
    get_next_prop_2
    match_prop_status     Mstop       M_get_next_p
    get_1st_val_1

M_match_val:
    match_val_name        Mfailed     M_get_next_v
    get_next_val_2
    match_val_status      Mfailed     M_get_next_v
    match_val_ctxt        Mfailed     M_get_next_v

|   At this point "vthis" holds the address of the matching value block,
|   so we can now perform the substitution.

Mpassed:
    subst_new_val

|   Now check if next relation matches the spec

    match_PS_zero         vnext       M_get_next_v
    match_PS_zero         pnext       M_get_next_p

    jsr       clear_poll
    jmp       start

Mfailed:
|   jumps to "Mstop" if prop is not a wildcard
    jmp_wild              Mstop       prop

|   jumps to M_get_next_p if pnext is != 0
    match_PS_zero         pnext       M_get_next_p

Mstop:
    jsr       clear_poll
    jmp        start

|   #####

|   This routine works in a similar manner to Retrieve_rel, except
|   that the matching relation is deleted

Delete_rel:
    clear_PS    error_cond
    clear_PS    vlast
    clear_PS    vthis
    clear_PS    vnext
    clear_PS    plast
    clear_PS    pthis
    clear_PS    pnext

    get_1st_prop_1

D_match_prop:
    match_prop_name        Dstop        D_get_next_p
    get_next_prop_2
    match_prop_status      Dstop        D_get_next_p
    get_1st_val_1

D_match_val:
    match_val_name         Dfailed      D_get_next_v
    get_next_val_2
    match_val_status       Dfailed      D_get_next_v
    match_val_ctxt         Dfailed      D_get_next_v

|   At this point "vthis" holds the address of the matching value block,
|   "vlast" holds the address of the previous value block, and
|   "vnext" holds the address of the next value block in the search list;
|   so we can now perform the deletion
|   Note that if vlast is NULL, then we are deleting the first value in
|   the list, and so 1st_val must be updated in the property block.
|   If vnext is also NULL, then we are deleting the ONLY value in the
|   property block, so the property block itself must be deleted.

Dpassed:
|   jumps to D_val if vlast is not NULL
    match_PS_zero          vlast        D_val

|   jumps to D_update_prop if vnext is not NULL
```

```
    match_PS_zero         vnext        D_update_prop

|   At this point we know that both vlast and vnext are NULL, so we must
|   delete the value block and look at the property list.

    jsr          delete_V_space
    move_vnext_vthis

|   At this point "pthis" holds the address of the matching property block,
|   "plast" holds the address of the previous property block, and
|   "pnext" holds the address of the next property block in the search list;
|   so we can now perform the deletion
|   Note that if plast is NULL, then we are deleting the first property in
|   the list, and so 1st_prop must be updated in the symbol table.
|   If pnext is also NULL, then we are deleting the ONLY property in the
|   list, so the symbol table must be updated accordingly.

|   jumps to D_prop if plast is not NULL

    match_PS_zero         plast        D_prop

D_update_sym:
|   We need to update 1st prop_ptr in the symbol table.

    delete_1st_prop
    jsr          delete_P_space
    match_PS_zero         pnext        D_get_prop

|   This informs the CPU that it must set the symbol table address to the
|   new value contained in p_ptr

    jsr          ERROR
    jsr          clear_poll
    jmp            start

|   Otherwise we simply delete the property block from the list

D_prop:
    delete_prop
    jsr          delete_P_space
    match_PS_zero         pnext        D_get_prop
    jsr          clear_poll
    jmp            start
```

```
D_update_prop:
|    At this point we know that we are at the start of the list, but
|    that there is at least one other value in the list, so we need to
|    update 1st_val in the property pointed to by pthis.

     delete_1st_value
     jsr          delete_V_space
     match_PS_zero            vnext          D_get_val
     jsr          clear_poll
     jmp           start

|    Otherwise we simply delete the value block from the list

D_val:
     delete_value
     jsr          delete_V_space
     match_PS_zero            vnext          D_get_val
     jsr          clear_poll
     jmp           start

Dfailed:
|    jumps to "Dstop" if prop is not a wildcard
     jmp_wild                 Dstop          prop

|    jumps to D_get_next_p if pnext is != 0
     match_PS_zero            pnext          D_get_next_p

Dstop:
     jsr          clear_poll
     jmp           start

|    #####
|    END OF MAIN CODE
|    #####

|    #####
|    START OF ANCILLIARY SUBROUTINE CALLS
|    #####

|    These routines retrieve the first word of the next property in the
|    current search list

C_get_next_p:
     get_next_prop_1
```

```
        jmp         C_match_prop

R_get_next_p:
        get_next_prop_1
        jmp         R_match_prop

M_get_next_p:
        get_next_prop_1
        jmp         M_match_prop

D_get_next_p:
        move_pthis_plast

D_get_prop:
        get_next_prop_1
        jmp         D_match_prop

|       These routines retrieve the first word of the next value in the
|       current search list

C_get_next_v:
        get_next_val_1
        jmp         C_match_val

R_get_next_v:
        get_next_val_1
        jmp         R_match_val

M_get_next_v:
        get_next_val_1
        jmp         M_match_val

D_get_next_v:
        move_vthis_vlast

D_get_val:
        get_next_val_1
        jmp         D_match_val
```

```
|   This section of code creates a new property/value pair in memory,
|   connects it to the previous property in the list,
|   and updates the Pointer Store contents accordingly

new_prop:
    jsr         create_P_space
    store_prop_ptr
    jsr         create_V_space
    move_vnext_vthis

    store_prop_KB_1
    store_prop_KB_2
    store_val_KB_1
    store_val_KB_2

    jsr         clear_poll
    jmp          start

|   This section of code creates a new value block in memory,
|   connects it to the previous value in the list,
|   and updates the Pointer Store contents accordingly

new_val:
    jsr         create_V_space
    store_val_ptr

    store_val_KB_1
    store_val_KB_2

    jsr         clear_poll
    jmp          start

|   END OF MICROCODE
|   #####

END

|   #####
```

APPENDIX G
Microprogram Libraries

G.1. lib.mac – Macro Library

```
#defmac         get_1st_prop_1          M2
{
        <IN>    cont
        <RF>    read            1st_prop
        <KB>    load_IO1        KB
}
{
        <IN>    cont
        <PS>    read            1st_prop        CTR
}
{
        <IN>    cont
        <PS>    write           pthis           CTR
}
{
        <IN>    cont
        <KB>    store_IO1       RF
        <RF>    write_KBL       b_prop
        <PS>    write           pnext           KB
}
#endmac

#defmac         match_prop_name         M4
{
        <IN>    cont
        <PS>    read            pnext           PZ
        <RF>    read            s_prop
        <KB>    cmp_IO          lsw
}
{
        <IN>    c2jmp           .label1         .label2
        <CC>    ccpos           prop_cmp
        <CL>    slow
}
#endmac
```

```
#defmac         match_prop_status     M5
{
        <IN>    cont
        <PS>    read        pnext      PZ
        <RF>    read        s_pstat
        <KB>    cmp_IO      lsw
}
{
        <IN>    c2jmp       .label1    .label2
        <ST>    test
        <CC>    ccpos       equal
        <CL>    slow
}
#endmac

#defmac         get_next_prop_1        M6
{
        <IN>    cont
        <KB>    load_IO1    KB
        <PS>    read        pnext      ADDR
}
{
        <IN>    cont
        <PS>    read        pnext      CTR
}
{
        <IN>    cont
        <PS>    write       pthis      CTR
}
{
        <IN>    cont
        <RF>    write_KBL   b_prop
        <KB>    store_IO1   RF
        <PS>    write       pnext      KB
        <CL>    slow
}
#endmac

#defmac         get_next_prop_2        M6
{
        <IN>    cont
        <KB>    load_IO2    KB
        <PS>    read        pthis      ADDR
}
```

```
{
        <IN>        cont
        <RF>        write_KBL      b_pstat
        <KB>        store_IO1      RF
        <PS>        write          1st_val        KB
        <CL>        slow
}
#endmac

#defmac             get_1st_val_1              M7
{
        <IN>        cont
        <PS>        read           1st_val        ADDR
        <KB>        load_IO1       KB
}
{
        <IN>        cont
        <PS>        read           1st_val        CTR
}
{
        <IN>        cont
        <PS>        write          vthis          CTR
}
{
        <IN>        cont
        <KB>        store_IO1      RF
        <RF>        write_KBL      b_val
        <PS>        write          vnext          KB
        <CL>        slow
}
#endmac

#defmac             match_val_name             M9
{
        <IN>        cont
        <RF>        read           s_val
        <PS>        read           vnext          PZ
        <KB>        cmp_IO         lsw
}
{
        <IN>        c2jmp          .label1        .label2
        <CC>        ccpos          value_cmp
        <CL>        slow
}
```

```
#endmac

#defmac           match_val_status      M10
{
      <IN>      cont
      <RF>      read          s_vstat
      <PS>      read          vnext          PZ
      <KB>      cmp_IO        lsw
}
{
      <IN>      c2jmp         .label1        .label2
      <ST>      test
      <CC>      ccpos         equal
      <CL>      slow
}
#endmac

#defmac           match_val_ctxt             M11
{
      <IN>      cont
      <RF>      read          s_ctxt
      <PS>      read          vnext          PZ
      <KB>      cmp_IO        msw
}
{
      <IN>      c2jmp         .label1        .label2
      <CC>      ccpos         ctxt_cmp
      <CL>      slow
}
#endmac

#defmac           get_next_val_1             M12
{
      <IN>      cont
      <KB>      load_IO1      KB
      <PS>      read          vnext          ADDR
}
{
      <IN>      cont
      <PS>      read          vnext          CTR
}
{
      <IN>      cont
      <PS>      write         vthis          CTR
}
{
        <IN>        cont
        <RF>        write_KBL       b_val
        <KB>        store_IO1       RF
        <PS>        write           vnext       KB
        <CL>        slow
}
#endmac

#defmac             get_next_val_2              M12
{
        <IN>        cont
        <KB>        load_IO2        KB
        <PS>        read            vthis       ADDR
}
{
        <IN>        cont
        <RF>        write_KBL       b_vstat
        <KB>        store_IO2       RF
}
{
        <IN>        cont
        <RF>        write_KBH       b_ctxt
        <KB>        store_IO2       RF
        <CL>        slow
}
#endmac

#defmac             store_prop_KB_1             M13
{
        <IN>        cont
        <RF>        read            s_prop
        <KB>        load_IO1        RF          lsw
        <PS>        read            pnext       KB
        <CL>        slow
}
{
        <IN>        cont
        <KB>        store_IO1       KB
        <PS>        read            pthis       ADDR
}
#endmac
```

```
#defmac              store_prop_KB_2              M14
{
        <IN>       cont
        <RF>       read              s_pstat
        <KB>       load_IO2           RF            lsw
        <PS>       read              1st_val       KB
        <CL>       slow
}
{
        <IN>       cont
        <KB>       store_IO2          KB
        <PS>       read              pthis         ADDR
}
#endmac

#defmac              store_val_KB_1               M15
{
        <IN>       cont
        <RF>       read              s_val
        <KB>       load_IO1           RF            lsw
        <PS>       read              vnext         KB
        <CL>       slow
}
{
        <IN>       cont
        <KB>       store_IO1          KB
        <PS>       read              vthis         ADDR
}
#endmac

#defmac              store_val_KB_2               M16
{
        <IN>       cont
        <RF>       read              s_vstat
        <KB>       load_IO2           RF            lsw
        <CL>       slow
}
{
        <IN>       cont
        <RF>       read              s_ctxt
        <KB>       load_IO2           RF            msw
        <CL>       slow
}
```

```
{
        <IN>        cont
        <KB>        store_IO2       KB
        <PS>        read            vthis       ADDR
}
#endmac

#defmac             match_PS_zero               M17
{
        <IN>        cont
        <PS>        read            .label1     PZ
}
{
        <IN>        cjmp            .label2
        <CC>        ccneg           ptr_zero
}
#endmac

#defmac             clear_PS                M18
{
        <IN>        cont
        <PS>        write           .label1     RF
        <RF>        read            clear_ps
}
#endmac

#defmac             jmap                        M19
{
        <IN>        jmap
        <CL>        slow
}
#endmac

#defmac             jsr                     M20
{
        <IN>        cjsr            .label1
        <CC>        force           pass
        <CL>        slow
}
#endmac

#defmac             jmp                         M21
{
        <IN>        cjmp            .label1
        <CC>      force          pass
        <CL>      slow
}
#endmac

#defmac              write_PS_from_RF            M23
{
        <IN>      cont
        <PS>      write          .label1          RF
        <RF>      read           .label2
}
#endmac

#defmac              write_RF_from_PS            M24
{
        <IN>      cont
        <RF>      write_PS       .label1
        <PS>      read           .label2          RF
}
#endmac

#defmac              get_1st_prop_ptr         M25
{
        <IN>      cont
        <RF>      write_PS       p_ptr
        <PS>      read           pthis          RF
}
#endmac
```

G.2. lib.sub – Sub-Routine Library

```
|S1
{
delete_P_space:
        <IN>      cjmp           S1_jump
        <PS>      read           free_ptr      PZ
        <CC>      ccpos          ptr_zero
}
{
        <IN>      cont
        <PS>      read           pthis        KB
        <KB>        load_IO1        KB
        <CL>        slow
}
{
        <IN>        cont
        <PS>        read            free_ptr      KB
        <KB>        load_IO1        PS
}
{
        <IN>        cont
        <PS>        read            pthis             KB
        <KB>        store_IO1       KB
}
{
S1_jump:
        <IN>        cont
        <PS>        read            pthis             CTR
}
{
        <IN>        cont
        <PS>        write           free_ptr      CTR
}
{
        <IN>        rtn
}

|S2
{
delete_V_space:
        <IN>        cjmp            S2_jump
        <PS>        read            free_ptr      PZ
        <CC>        ccpos           ptr_zero
}
{
        <IN>        cont
        <PS>        read            vthis         KB
        <KB>        load_IO1        KB
        <CL>        slow
}
{
        <IN>        cont
        <PS>        read            free_ptr      KB
        <KB>        load_IO1        PS
}
```

```
{
        <IN>      cont
        <PS>      read          vthis      KB
        <KB>       store_IO1      KB
}
{
S2_jump:
        <IN>      cont
        <PS>      read          vthis      CTR
}
{
        <IN>      cont
        <PS>      write         free_ptr    CTR
}
{
        <IN>      rtn
}


|S3
{
create_P_space:
        <IN>      cont
        <PS>      read          free_ptr    PZ
}
{
        <IN>      cjmp          S3_jump
        <CC>       ccneg         ptr_zero
}
{
        <IN>      cont
        <PS>      read          mem_ptr      PZ
}
{
        <IN>      cjmp          ERROR
        <CC>       ccpos         ptr_zero
}
{
        <IN>      cont
        <PS>      read          mem_ptr      CTR
}
{
        <IN>      cont
        <PS>      write         pthis       CTR
```

```
}
{
        <IN>        cjmp            S3_end
        <CC>         force           pass
        <PS>        inc             mem_ptr
}
{
S3_jump:
        <IN>        cont
        <PS>        read            free_ptr        CTR
}
{
        <IN>        cont
        <PS>        write           pthis           CTR
        <CL>         slow
}
{
        <IN>        cont
        <PS>        read            free_ptr        ADDR
        <KB>         load_IO1        KB
        <CL>         slow
}
{
        <IN>        cont
        <PS>        write           free_ptr        KB
        <KB>         store_IO1       RF
}
{
S3_end:         <IN>        rtn
}


|S4
{
create_V_space:
        <IN>        cont
        <PS>        read            free_ptr        PZ
}
{
        <IN>        cjmp            S4_jump
        <CC>         ccneg           ptr_zero
}
{
        <IN>        cont
        <PS>      read          mem_ptr      PZ
}
{
        <IN>      cjmp           ERROR
        <CC>       ccpos          ptr_zero
}
{
        <IN>      cont
        <PS>      read          mem_ptr      CTR
}
{
        <IN>      cont
        <PS>      write         vthis      CTR
}
{
        <IN>      cjmp          S4_end
        <CC>       force         pass
        <PS>      inc          mem_ptr
}
{
S4_jump:
        <IN>      cont
        <PS>      read          free_ptr     CTR
}
{
        <IN>      cont
        <PS>      write         vthis      CTR
        <CL>       slow
}
{
        <IN>      cont
        <PS>      read          free_ptr     ADDR
        <KB>       load_IO1        KB
        <CL>       slow
}
{
        <IN>      cont
        <PS>      write         free_ptr     KB
        <KB>       store_IO1       RF
}
{
S4_end:     <IN>        rtn
}
```

```
|S5
{
ERROR:     <IN>      cont
      <PS>      read            error_cond    CTR
}
{
      <IN>      cont
      <PS>      inc             error_cond
}
{
      <IN>      cont
      <PS>      read            error_cond      RF
      <RF>       write_PS         error_cond
}
{
      <IN>      rtn
}

| S6
{
init:
      <IN>      cont
      <RF>       read            clear_ps
      <PS>      write           clear         RF
}
{
      <IN>      cont
      <RF>       read            init_mem_ptr
      <PS>      write           mem_ptr         RF
}
{
      <IN>      cont
      <RF>       read            clear_ps
      <PS>      write           free_ptr        RF
}
{
      <IN>      cont
      <RF>       read            clear_ps
      <PS>      write           vlast         RF
}
{
      <IN>      cont
      <RF>       read            clear_ps
      <PS>      write           vthis         RF
```

```
}
{
        <IN>        cont
        <RF>         read                clear_ps
        <PS>        write               vnext          RF
}
{
        <IN>        cont
        <RF>         read                clear_ps
        <PS>        write               plast          RF
}
{
        <IN>        cont
        <RF>         read                clear_ps
        <PS>        write               pthis          RF
}
{
        <IN>        cont
        <RF>         read                clear_ps
        <PS>        write               pnext          RF
}
{
        <IN>        cont
        <RF>         read                clear_ps
        <PS>        write               1st_prop       RF
}
{
        <IN>        cont
        <RF>         read                clear_ps
        <PS>        write               1st_val        RF
}
{
        <IN>        cont
        <RF>         read                error_cond
        <PS>        write               error_cond     RF
}
{
        <IN>        rtn
}

| S7
{
poll_cpu:
        <IN>        cont
        <RF>        read            csr
}
{
        <IN>        cjmp            poll_cpu
        <CC>        ccpos           poll_bit
}
{
        <IN>        rtn
}

| S8
{
clear_poll:
        <IN>        cont
        <RF>        write_PS        csr
        <PS>        read            clear       RF
}
{
        <IN>        rtn
}
```

Related Publications

Hudson, S., Hannah, J.M., and Rae, R. *"Hardware Support For Knowledge Base Manipulation"* IEE Colloquium, *Current Trends in Knowledge Based Systems*, Heriot Watt University, April 1989.

A copy of the original text presented at the IEE Colloquium is included in the following pages. The original pagination is unchanged, but for reference within this thesis, appropriate page numbers are provided at the bottom of this page.

HARDWARE SUPPORT FOR KNOWLEDGE BASE MANIPULATION

Stephen Hudson†, John M. Hannah†, Robert Rae‡

University of Edinburgh - Dept. of Electrical Engineering
King's Buildings, Mayfield Road, Edinburgh

Abstract

Despite the increased use and popularity of expert and AI planning and simulation systems, they remain unsuitable for many engineering applications due to poor real time response. A major contributing factor is the inability to manipulate information in a knowledge base quickly enough. Consequently, in conjunction with the Artificial Intelligence Applications Institute (AIAI), we have studied hardware support for a knowledge retrieval system.

A knowledge structure has been developed which is suitable for manipulation by special purpose hardware. Having successfully implemented a basic expert shell in software using this structure, a prototype system is being constructed to demonstrate the concepts developed. A free-list garbage collection algorithm has been adopted which also makes use of the knowledge structure format which can be implemented within the hardware with little speed overhead and no memory overheads.

Background

A principal consideration is how the knowledge is represented. First Order Predicate Calculus (FOPC) appears to be a simple, convenient, formalism. It is also relatively easy to map such representations onto hardware. However, FOPC requires a great many "well formed formulae" to be defined to describe even simple situations. Structured formalisms, on the other hand, can describe very complex situations and relations quite easily. Unfortunately, because they must describe general and changing situations (ie: cope as information is changed, deleted or appended) they cannot be predefined standard structures. This means that mapping onto hardware is difficult.

† Dept. Of Electrical Engineering
‡ Artificial Intelligence Applications Institute

Much of the research into knowledge bases has been concerned with faster manipulation methods, and different techniques have been developed to do this. The software solution has been to develop structured knowledge representations which allow the user to home into a particular piece of information by way of inheritance lattices, and to facilitate reasoning about complex situations (eg: the real world!). Examples of these solutions are a series of advanced intelligent tools, such as LOOPS [1], KEE [2], ART [3] and Knowledge Craft [4], which were designed for fast prototyping of expert systems, reasoning systems etc.

The hardware solutions have been varied, but generally based on FOPC methods. Examples include the development of fast disc controllers which perform serial database search [8], associative processors [7], the intelligent file store [5,6] and REKURSIV [9]. Hardware approaches which involve predicate calculus formalisms are inherently slower than structured formalisms but easier to manipulate. The approach taken in this project is to merge both; that is to develop a structure which can be predefined as a standard format while still able to support changing information.

The Knowledge Structure

We required some method of representing our information which was both versatile and amenable to hardware manipulation. Hence, we selected a general (n-ary) tree structure relating objects, slots and fillers, and represented this with a binary tree using linked lists. The structure is illustrated conceptually in figure 1 and physically in figure 2.

Quite simply, objects have slots (or properties) whose fillers (values) depend on the particular context being considered. Each object/property/value triple is known as a relation and has a *confidence* associated with it, which can be TRUE, FALSE, UNDEFINED or PROBABLE (this information is stored in a status word). The *contents of* the value can be either an atom, an object or a link to another relation (either within the same object or within another object); allowing us to build up higher order relations. Status information is also included for tagging and masking purposes. As shown in figure 1, we might want to represent the information: *Steve probably plays squash on a Monday* where *Steve is a man*. [Note that the latter relation is always true, so the context *Monday* would be a child of *Always* .]

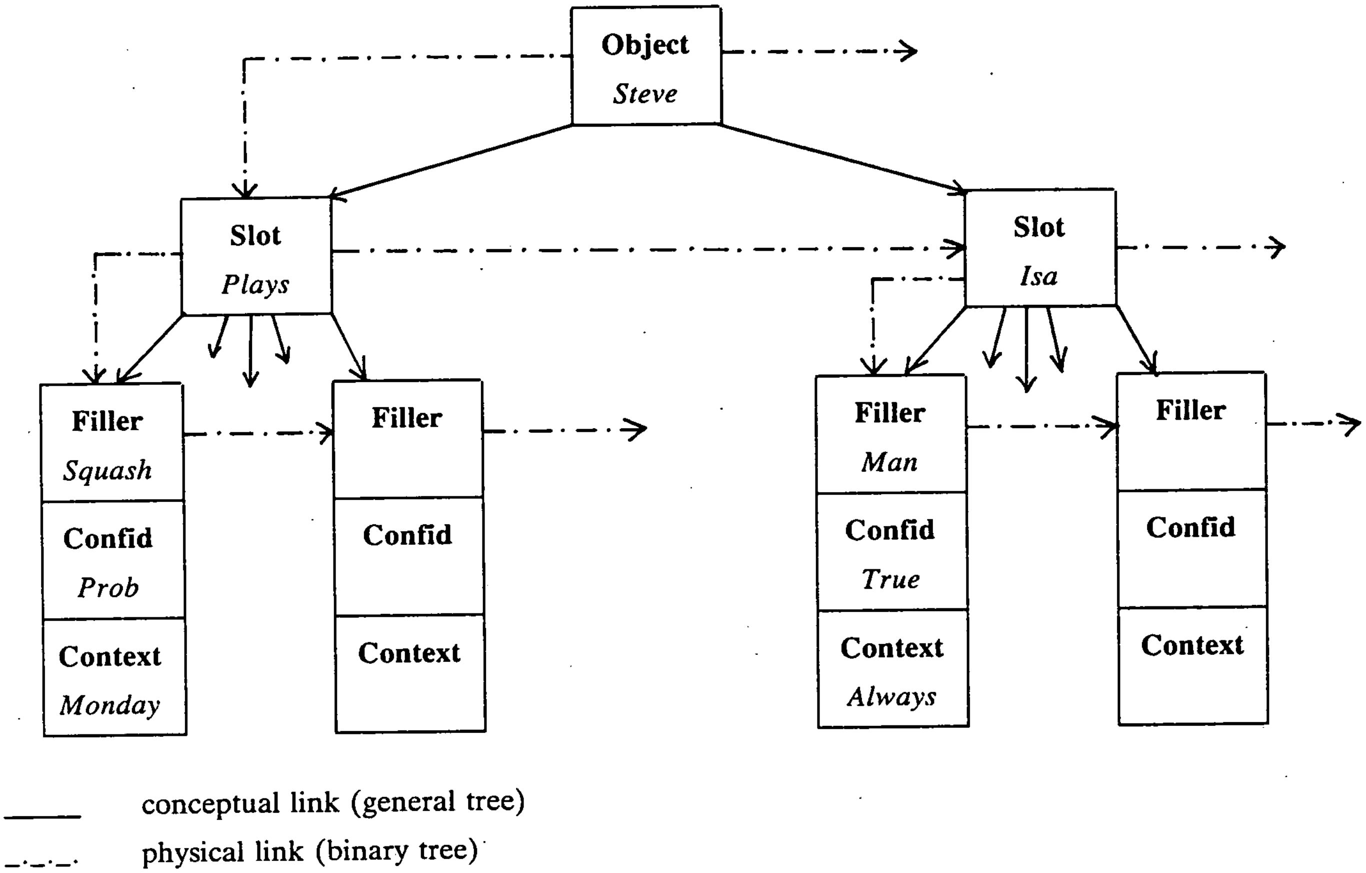


Figure 1: Conceptual Knowledge Structure

Object:	Obj_Code	*Next_Obj	Status_Info	*First_Prop
Property:	Prop_Code	*Next_Prop	Status_Info	*First_Val
Value:	Val_Code	*Next_Val	Status_Info	Context

(* represents a pointer)

Figure 2: Physical Knowledge Structure - Component Blocks

The Manipulation Facilities

The facilities supported have been chosen to comply with those described by Tate [11] as being practical for a context data base. New relations can be **created** which may or may not be related to existing information. If a property is given no value, then it is either inherited from a parent or it is given the value "undef" (not to be confused with the confidence). Slots can be masked so that they are not inherited by their sub-classes. Knowledge can also be **deleted** from memory. It is possible to delete either all or part of a structure. We can also **modify** any part of a structure: object name, property name, value name, context or confidence. The system keeps a record of the **current context** in which we are interested, which can be changed at any time, and all of the above operations can be performed within a specified, or the default (current), context. Contexts are defined in the same way as objects and so can be created, deleted or modified similarly. Any specified relation or partially specified relation(s) can be **retrieved** from the knowledge base. Mark bits are used to allow **logical connections** between relations within a specification. It is also possible to retrieve all of the relations which do not match a particular specification.

The Manipulation Hardware

The use of this knowledge structure and the manipulation techniques were initially investigated by computer simulation (in "C") of a basic expert shell (cf. "Knowledge Craft"), and profiling operations were carried out which pinpointed major bottlenecks. Since the primary limiting factor was found to be linked-list-traversal, a hardware design strategy was developed which dealt with linked-list codes and pointers in parallel; ie. code matching with specifications and linked-list book-keeping are performed simultaneously. The linked-list format is also used to connect unused memory blocks, and a *free-list* garbage collection algorithm is included within the microprogram with very little speed overhead. No special purpose hardware is used to perform the higher level manipulations of the knowledge base, since the speed gains would not be worth the cost or effort involved. Only the searching, matching, creation, deletion, modification and retrieval of a single specified relation (wildcard entries permitted) is performed by the support hardware. All higher level interpretations and control are performed by the "host" CPU. A block diagram of the complete system is shown in figure 3.

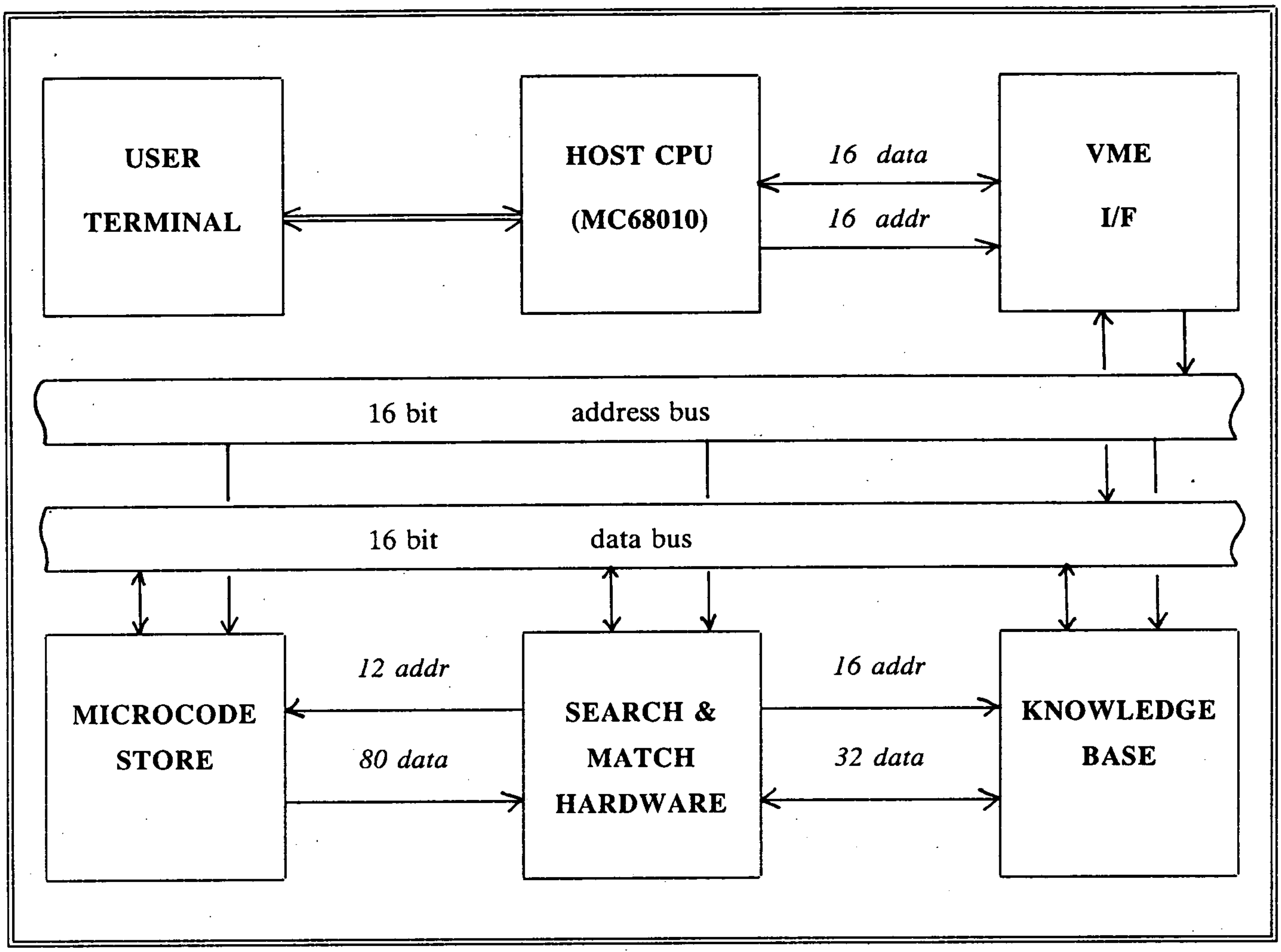


Figure 3: Knowledge Manipulation System - Block Diagram

A prototype system is currently being built to demonstrate these ideas. This incorporates the special purpose retrieval hardware, 512 kbytes of knowledge base memory and a MC68010 CPU host, interfaced via a VMEbus. The retrieval hardware acts as a slave system to the host and performs particular tasks on the knowledge base when instructed. No processing elements are involved in the special purpose hardware; a microprogram sequencer provides local control of storage and data multiplexing elements, but overall control and user interfacing is performed by the host CPU board. This means that the design is very suitable for silicon fabrication, and hence cost effective speed improvements.

The retrieval hardware is illustrated in Figure 4 and has been designed to implement a basic *search-match-operation* algorithm:

```
1.   Retrieve 1st object block from knowledge base address 1 (0 is unused)

2.   Compare object code and status with the specification supplied by the host

3.   If they match...

     (a)  Retrieve 1st property block from address *First_Prop
     (b)  Compare property code and status with the spec
     (c)  If they match...

          [i]   Retrieve the 1st value block from the address *First_Val
          [ii]  Compare the value code, context code and status with the spec
          [iii] If they match...

                1)   Inform the CPU
                2)   Stop after successful search

          [iv]  Else...

                1)   If *Next_Val is nil, go to (d)[i]
                2)   Else retrieve the next value block from address *Next_Val
                3)   Go to [ii]

     (d)  Else...

          [i]   If *Next_Prop is nil go to 4 (a)
          [ii]  Else retrieve the next property block from address *Next_Prop
          [iii] Go to (b)

4.   Else...

     (a)  If *Next_Obj is nil, inform the CPU and stop after unsuccessful search.
     (b)  Else retrieve the next object block from address *Next_Obj
     (c)  Go to 2.
```

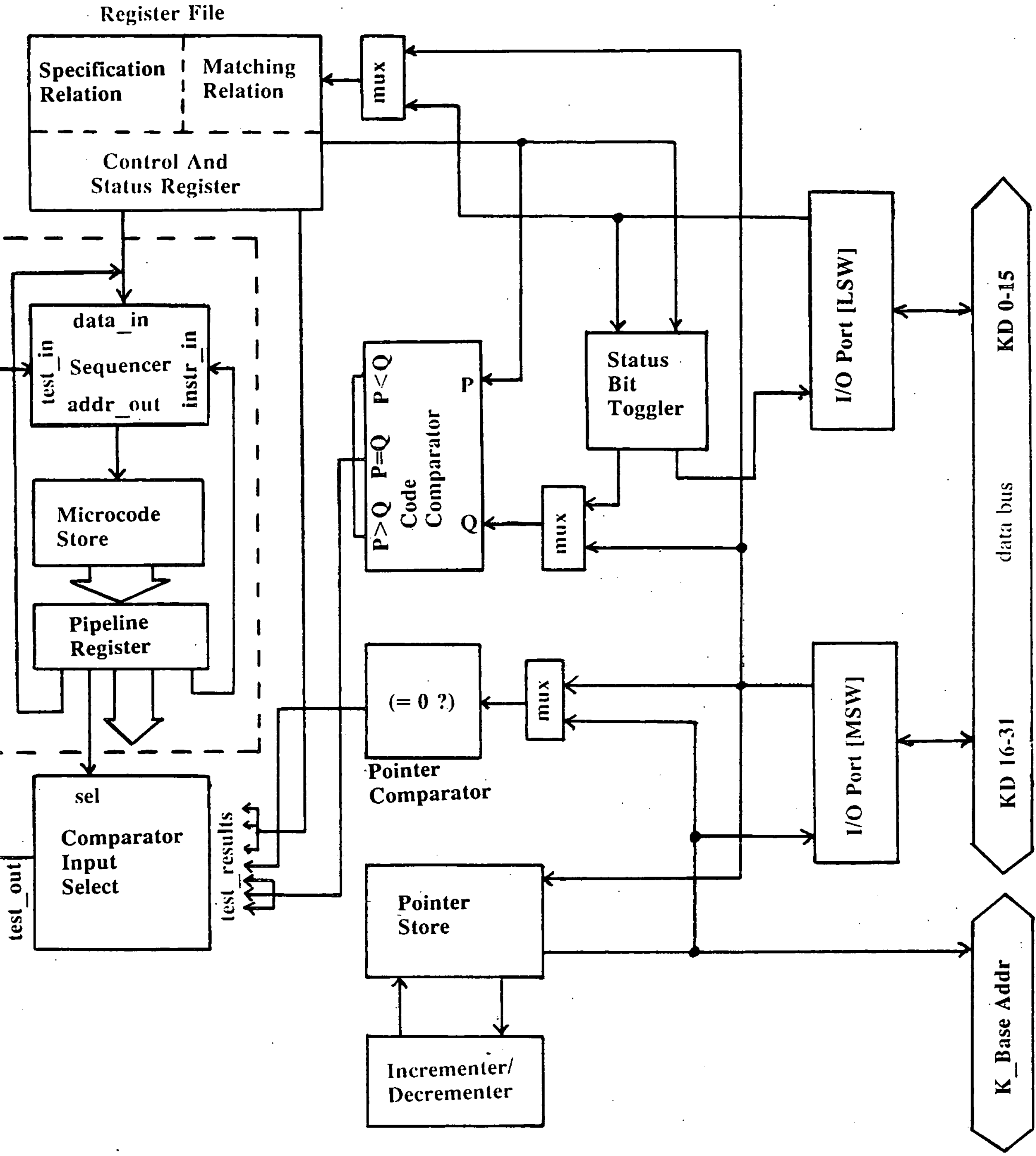


Figure 4: Retrieval Hardware - Block Diagram

Refering back to figure 2, each of the *blocks* which make up our knowledge structures comprises four words. During the search, the first two words are copied into the I/O Ports. Since, in general, we would expect there to be more non-matching than matching relations, the first pointer inspected is to the next entry on the same level of the tree rather than the first entry on the next level down the tree. The two comparator banks compare codes with the specifications *(greater than, less than,* and *equal to* outputs) and compare pointers with zero for *end of list* checking. If the codes match, then the next two words are copied into the I/O Ports. The Mark Bit Toggle circuit controls the status word mark bit when matching specification relations with logical connections. In the case of object and property blocks, the most significant word (MSW) now holds the pointer to the first entry in the next level down the tree, and the new block is read in. In the case of value blocks, we are at the bottom level of the tree, so there are no pointers. Instead, we must compare the context with the specification. Since the knowledge base input/output cycle times are longer than the sequencer's, the Comparator Input Select circuitry can sequentially inspect the results of the two comparison operations and present the appropriate pointer at the address bus in time for the next knowledge base access. The Comparison Input Select circuit comprises a group of multiplexers which select the appropriate test input to the sequencer from the comparators, under the control of the control and status register (written to by the host). As the codes and pointers are read from the knowledge base, they are stored in the Register File and Pointer Store respectively. The Register File is a 4 port dual access memory IC, which is also used to store the control and status register and the specification relation. The Pointer Store is also used for garbage-free-list book-keeping. The entire operation is controlled by a one level pipelined sequencer which communicates with the MC68010 host CPU via the Register File.

We are implementing a demonstration expert system for evaluation purposes. Three layers of software are required for the system. The first layer is the microcode for the structure traversal / matching hardware. The second layer [10] comprises the instruction set which implements most of the context data-base functions described by Tate [11]. These are supplied by the MC68010 as macro-instructions to the first layer (via the Register File). The third layer consists of the specific application - in this case, a demonstration expert shell.

Conclusion

With the belief that object oriented knowledge bases are more suitable for use within the real world than FOPC based systems, a general tree based knowledge structure has been developed which is flexible and amenable to hardware manipulation. Having tested the structure by software simulation of a basic expert shell, and having pinpointed the inherently slower operations, special purpose hardware has been designed which supports the desired facilities. Garbage collection, a recurrent problem in many intelligent systems, is performed concurrently with very little speed overhead and no memory overheads. The search and manipulation mechanism (Figure 4) contains no processing elements and so is well suited for integration. Although this design provides speed gains through a limited amount of parallelism, nevertheless it still performs a *depth first search.* It is envisaged, however, that maximum parallelism will be gained by utilising banks of such ICs under the supervision of a central controller, with each assigned to a *block* of memory.

References

[1] *LOOPS* Artificial Intelligence Applications Institute "AI for Engineers: Course Notes" University of Edinburgh 1987

[2] *KEE* Artificial Intelligence Applications Institute "AI for Engineers: Course Notes" Edinburgh University 1987

[3] *ART* Artificial Intelligence Applications Institute "AI for Engineers: Course Notes" Edinburgh University 1987

[4] *Knowledge Craft* Artificial Intelligence Applications Institute "AI for Engineers: Course Notes" Edinburgh University 1987

[5] Lavington, S.H. et al "*Memory Structures in the Intelligent File Store*" in "*Proc. first SIGKME workshop*" 1987

[6] Jiang, Y.J & Lavington, S.H "*The Qualified Binary Relationship Model of Information*" in "*Proc. fourth British National Conference on Databases*" Cambridge University Press 1985

[7] Oosthuizen, D.R. et al "*Parallel Network Architectures for Large Scale Knowledge Based Systems*" in "*Proc. first SIGKME workshop*" 1987

[8] Williams, M.H. "*The Design Of A Prolog Database Machine*" in "*Proc. first SIGKME workshop*" 1987

[9] Harland, D.M. "*A Recursively Microcodable Tagged Architecture*" in *ACM SIGARCH Vol 14 (No.3) pp 34-40,* June 1986

[10] Hudson, S. "*Novel Hardware For Knowledge Base Manipulation - Functional Specification*", Instrumentation and Digital Systems Group, *Internal Report,* University of Edinburgh, August 1987

[11] Tate, A. "*Functions In Context Data Bases*" Artificial Intelligence Applications Institute *Technical Report AIAI-TR-1* University of Edinburgh 1984

REFERENCES

[1] Rich, E. *Artificial Intelligence* McGraw-Hill series in artificial intelligence, 1983.

[2] Gevarter, W.B. *Artificial Intelligence, Expert Systems, Computer Vision and Natural Language Processing*, Noyes Publication, N.J., 1984.

[3] Nilsson, N.J. *Principles of Artificial Intelligence* Symbolic Computation, 1980.

[4] Winston, P. *Artificial Intelligence*, Addison Wesley, Reading, Massachusetts, USA, 1977.

[5] Turing, A. "*Computing Machinery and Intelligence*" in "*Computers and Thought*" ed. Feigenbaum, E.A. & Feldman, J. McGraw-Hill, New York, 1975.

[6] Samuel, A.L. "*Some Studies In Machine Learning Using The Game Of Checkers*" in "*Computers and Thought*" ed. Feigenbaum, E.A. & Feldman, J. McGraw-Hill, New York, 1963.

[7] Newell, A.J. et al "*Empirical Explorations with the Logic Theory Machine: A Case Study In Heuristics*" in "*Computers and Thought*" ed. Feigenbaum, E.A. & Feldman, J. McGraw-Hill, New York, 1963.

[8] Newell, A.J. et al "*GPS, A Program That Simulates Human Thought*" in "*Computers and Thought*" ed. Feigenbaum, E.A. & Feldman, J. McGraw-Hill, New York, 1963.

[9] Lindsay, R.K. et al "*Applications of Artificial Intelligence for Organic Chemistry: The Dendral Project*" McGraw-Hill, New York, 1980.

[10] Shortliffe, E.H. "*Computer-based Medical Consultations: MYCIN*" Elsevier, New York, 1976.

[11] Anderson, T.M. "*Expert Systems: An Appropriate Technology For Developing Countries*" Submitted to IEEE PES Summer Meeting, Minneapolis, July 1990.

[12] Walls, J. A. *Capacitance-Voltage Measurements: an Expert System Approach* Ph.D. Thesis, Edinburgh Microfabrication Facility, Dept. of Electrical Engineering, University of Edinburgh, 1990.

[13] Hetherington, G. "*Expert Systems in VLSI Design*" in IEE Symposium on "*Current Trends in the Application of Expert Systems*", Strathclyde University, Glasgow, Scotland, April 1986.

[14] Milne, R.W. and Chandresekaran, B. "*Fault Diagnosis and Expert Systems*" in IEE Symposium on "*Current Trends in the Application of Expert Systems*", Strathclyde University, Glasgow, Scotland, April 1986.

[15] Davis, R. and King, J. "*An Overview of Production Systems*" in *Machine Intelligence*, Vol. 8, pp. 30-332, ed. Elcock, E.W. and Michie, D., Wiley, 1977.

[16] Tate, A. & Daniel, L. "*A Retrospective on the 'Planning: a joint AI/OR Approach' Project*" Dept. of Artificial Intelligence Working Paper No.125, University of Edinburgh, 1982.

[17] Tate, A. *"Functions In Context Data Bases"* Artificial Intelligence Applications Institute, Technical Report AIAI-TR-1, University of Edinburgh, 1984.

[18] McDermott, D.V. and Sussman, G.J. *The CONNIVER Reference Manual,* MIT AI Lab. Memo No. 259, 1972.

[19] Sacerdoti, E.D. et al *"QLISP: a language for the interactive development of complex systems"* SRI Technical Note, SRI International, Stanford, 1976.

[20] Deering, M et al *"PEARL - a package for efficient access to representations in LISP"* in *International Joint Conference on Artificial Intelligence,* pp. 930-932, Vancouver, Canada 1981

[21] Barrow, H.G. *"HBASE: a fast clean efficient data base system"* POP-2 Program Library Documentation, University of Edinburgh, 1975.

[22] Quillian, R. *"Semantic Memory"* in *"Semantic Information Processing"* ed. Minsky, M., MIT Press, Cambridge, Massachusetts, USA, 1968.

[23] Doyle, J. *"A Truth Maintenance System"* in *"Artificial Intelligence"*, Vol.12, No.3, 1979.

[24] Hunt, E. *"Artificial Intelligence"* Academic Press, New York, 1975.

[25] Corlett, R.A. *"Features of Artificial Intelligence Languages and Their Environments"*, in *IEE Software Engineering Journal*, pp. 159-164, July, 1986.

[26] Winston, H.W. and Horn, K.P.H. *LISP*, Addison-Wesley, Reading, Massachusetts, USA, 1984 (2nd Edition).

[27] Foderaro, J.K. and Sklower, K.E. *"The Franz LISP Manual"* in *"4.2 UNIX on-line manual*, Berkeley University, California, USA, 1980.

[28] Clocksin, W.F. and Mellish, C.S. *Programming in Prolog*, Springer-Verlag, 1984 (2nd Edition).

[29] *LOOPS* Artificial Intelligence Applications Institute, "AI for Engineers: Course Notes", University of Edinburgh, 1987.

[30] *KEE* Artificial Intelligence Applications Institute, "AI for Engineers: Course Notes", University of Edinburgh, 1987.

[31] *ART* Artificial Intelligence Applications Institute, "AI for Engineers: Course Notes", University of Edinburgh, 1987.

[32] *Knowledge Craft* Artificial Intelligence Applications Institute, "AI for Engineers: Course Notes", University of Edinburgh, 1987.

[33] Symbolics Inc., *Symbolics 3600 Technical Summary* California, USA, 1983.

[34] Symbolics Inc., *"Low-Cost Workstation Aims To Move AI Out Of The Lab"* in *Electronics Week* pp. 55-56, April, 1986.

[35] Boothroyd, D. (Ed.) *"AI Hardware Is Poised For A Giant Leap"* in *intelligence* (TEXAS Instruments International Journal on Artificial Intelligence) No. 2, October, 1986.

[36] Intel Corporation, *386 Family Briefs*, USA, 1987.

[37] El-Ayat, K.A. and Agarwal, R.K. "*The Intel 80386 – Architecture and Implementation*" in *IEEE MICRO*, Vol.5, No. 6, pp. 4-22, December, 1985.

[38] Johnson, M. et al, *Am29000 Streamlined Instruction Processor – User's Manual* Advanced Micro Devices, USA, 1987.

[39] Patterson, D.A. "*Reduced Instruction Set Computers*" in *Communications of the ACM*, Vol.28, No.1, pp 8-21, January, 1985.

[40] Unknown, *SPARC Technology and Markets Update*, in *SunTech Journal* Vol. 2, No. 4, (Special Section), Autumn, 1989.

[41] SUN Microsystems Ltd., "*SUN Sets New Standard For Desktop Computing: 12.5 MIPS, RISC-Based System For Less Than £7,500*", Press Release (96), April, 1989.

[42] Lavington, S.H. et al "*Memory Structures in the Intelligent File Store*" in *Proceedings of the first Alvey-sponsored workshop of the Special Interest Group in Knowledge Manipulation Engines*, University of Reading, January, 1987.

[43] Harland, D.M. "*A Recursively Microcodable Tagged Architecture*" ACM SIGARCH Vol. 14, No. 3, pp. 34-40, June, 1986.

[44] Harland, D.M. et al, "*REKURSIV: An Architecture For Artificial Intelligence*" Presented at AI/EUROPA Conference, Wiesbaden, September, 1986.

[45] Foster, C.C. *Content Addressable Parallel Processors*" Van Nostrand Reinhold Company, USA, 1976.

[46] Lea, R.M. "*VLSI/WSI Associative String Processors Overview, Current Status and Future Prospects*" in *Proceedings of the first Alvey-sponsored workshop of the Special Interest Group in Knowledge Manipulation Engines*, University of Reading, January, 1987.

[47] Oosthuizen, G.D. et al, "*Parallel Network Architectures for Large Scale Knowledge Based Systems*" in *Proceedings of the first Alvey-sponsored workshop of the Special Interest Group in Knowledge Manipulation Engines*, University of Reading, January, 1987.

[48] INMOS Limited *Transputer Reference Manual*, Prentice Hall, 1988.

[49] Watson, I. et al, "*Flagship: A Parallel Architecture for Declarative Programming*" in "*IEEE Computer Architecture News*" ACM Press Vol. 16, No. 2, 15th Annual Symposium on Computer Architectures, Hawaii, 1988.

[40] Peyton Jones, S.L. et al, "*GRIP – a high Performance Architecture for Parallel Graph Reduction*" in *Proceedings of the 3rd International Conference on Functional Programming and Computer Architecture*, pp. 98-112, Portland, September, 1987.

[51] Miranker, D.P. "*A Survey of Specialised Parallel Architectures Designed to Support Knowledge Representation*", Artificial Intelligence Laboratory, The University of Texas at Austin, AI TR87-43, January, 1987.

[52] Lavington, S.H. "*IFS: Hardware Support for Large Knowledge-Based Systems*" University of Essex, Dept. of Computer Science, Internal Report IFS/3/86, April, 1986.

[53] Lavington, S.H. *"Technical Overview of the Intelligent File Store" "Knowledge Based Systems"*, Vol. 1, No. 3, pp. 166-172, June 1988.

[54] Lavington, S.H. and Wang, C. *"A Lexical Token Converter for the IFS"* University of Manchester, Dept. of Computer Science, Internal Report IFS/5/84, May, 1984.

[55] Jiang, Y.J. and Lavington, S.H. *"The Qualified Binary Relationship Model of Information"* in *Proceedings of 4th British Conference on Databases*, pp. 61-79, Cambridge University Press, July, 1985.

[56] McGregor, D.R. and Malone, J.R. *"An Integrated High Performance, Hardware Assisted, Intelligent Database System for Large-Scale Knowledge-Bases"* University of Strathclyde, Dept. of Computer Science, Internal Report, March, 1984.

[57] Lavington, S.H. *"Architectures for Large Knowledge-Based Systems: Future Research Directions Based on Graph Paradigms"* in *Proceedings of the Second Alvey-sponsored Workshop of the Special Interest Group in Knowledge Manipulation Engines"*, Brunel University, May 1987. (Available from the Alvey Directorate).

[58] Hinton, G. *"Learning in Parallel Networks"* in *Byte Magazine*, Vol. 10, No.4, 1985.

[59] Baron, R.J. and Shapiro, L.G. *Data Structures and Their Implementation* Van Nostrand Reinhold, USA, 1980.

[60] Cohen, J. *"Garbage Collection of Linked Data Structures"* in *"Computing Surveys"*, Vol. 13, No. 3, pp. 341-367, The Association of Computing Machinery, September, 1981.

[61] Baker, H.G. *"List Processing in Real Time on a Serial Computer"* in *Communications of the ACM*, Vol. 26, No. 6, pp. 419-429, June, 1983.

[62] Wong, K.F. *"An Intelligent Cell Memory System for Real Time Engineering Applications"* Ph.D. Thesis, University of Edinburgh, Dept. of Electrical Engineering, May, 1987.

[63] Motorola Limited, *MC68000 8-/16-/32-Bit Microprocessors User's Manual*, 1989 (6th Edition).

[64] Feller, W. *An Introduction to Probability Theory* Section 2.3, Wiley, New York, 1950.

[65] Knuth, D.E. *The Art of Computer Programming, Vol. 3: "Sorting And Searching"*, Addison-Wesley Series in Computer Science and Information Processing, Reading, Massachusetts, USA, 1973.

[66] Lum, Y.L. at al, *"Key-to-address Transform Techniques: A Fundamental Performance Study on Large Existing Files"* in *Communications of the ACM*, Vol. 14, pp. 228-239, 1971.

[67] Schay, G. and Raver, N.A. *"A Method For Key-to-Address Transformation"* in *IBM Journal of Research and Design*, Vol. 7, No. 2, pp. 121-129, April, 1963.

[68] Hanan, M. and Palermo, F.P. *"An Application of Coding Theory to a File Addressing Problem"* in *IBM Journal of Research and Design*, Vol. 1, No. 2, pp. 130-146, April, 1957.

[69] Peterson, W.W. *Error Correcting Codes* M.I.T. Press, Cambridge, Massachusetts, USA, 1961.

[70] Advanced Micro Devices *Am29300 Family: 32 bit Building Blocks* Product Information.

[71] Fischer, W. "*IEEE P1014 - A Standard for the High Performance VME Bus*" in "*IEEE Micro*", Vol. 5, No. 2, pp. 31-41, February, 1985.

[72] Dixon, K.B. "*The Ferranti/DY-4 DVME 785 Relational Processor*" Report No. 6902 Ferranti Computer Systems Limited, Bracknell Division, May 1987.

[73] Lavington. S.H. *IFS/1: Performance Figures* Private Communication (University of Essex), December 1989.

[74] Standring, M. et al, "*A 4 Mbyte Associative Predicate Store*" in *Proceedings of the First Alvey Sponsored Workshop on Architectures For Large Knowledge Based Systems*, University of Manchester, May, 1984. (Available from the Alvey Directorate).

[75] Lavington, S.H. and Waite, M. "*Handling AI Structures Declaratively*" in *Proceedings of the IED/SERC Joint Workshop of the Parallel Declarative Systems SIG and the Knowledge Manipulation Systems SIG on "Handling Large Knowledge Bases Declaratively*", September, 1989.

[76] Clack, C. *Notes of discussion sessions, and topics for future SIGKME activity: Statement 3* in *Proceedings of the Second Alvey-Sponsored Workshop of the Special Interest Group in Knowledge Manipulation Engines*, Brunel University, May, 1987. (Available from the Alvey Directorate).

[77] Jayasumana, A.P. and Jayasumana, G.G. "*On the Use of the IEEE 802.4 Token Bus in Distributed Real-Time Control Systems*" in *IEEE Transactions on Industrial Electronics*, Vol. 36, No. 3, August, 1989.

[78] Gater, C. *Fault Tolerant Distributed Measurement Systems* Ph.D. Thesis, University of Edinburgh, Dept. of Electrical Engineering, July 1987.

[79] Kernighan, B.W. and Ritchie, D.M. *The C Programming Language*, Prentice Hall Software Series, New Jersey, 1988 (2nd Edition).

[80] Kernighan, B.W. and Pike, R. *The UNIX Programming Environment*, Prentice Hall Software Series, New Jersey, 1978.

[81] Advanced Micro Devices *Am 2900 Family: Bipolar Microprocessor Logic and Interface Data Book*, 1985.