CONTENT ADDRESSABLE MEMORY:

Design and usage for general purpose computing

by

Gerard Miles Blair

Abstract

This thesis considers a Content Addressable Memory (CAM), from its MOS VLSI circuit design and successful fabrication, to its inclusion in a novel computer architecture. The design has a bit-serial, word-parallel configuration, implemented by semi-static shift registers, with multiple tags which also provide for freespace designation and retrieval. The contention through multiple responders is resolved using novel dynamic circuitry. The basic CAM design is enhanced with a natural fault tolerance and a scheme for hierarchical decomposition leading to a practical proposal for wafer scale expansion. The potential impact of Content Addressability, both in software engineering and in system design, is discussed in terms of its affinity to common abstract data types. After examining previous attempts to render innovative hardware accessible to the general programmer, the thesis develops a full programming environment for the CAM component within a novel variation to the von Neumann architecture which can be programmed through a tractable set of extensions to conventional high level language syntax.

CONTENT ADDRESSABLE MEMORY:

Design and usage for general purpose computing

by

Gerard Miles Blair

A thesis submitted to the Faculty of Science,

University of Edinburgh, for the degree of

Doctor of Philosophy.

Department of Electrical Engineering

1986

Declaration of Originality:

The material contained herein was researched and composed entirely by myself in the Department of Electrical Engineering, University of Edinburgh between October 1983 and July 1986.

Acknowledgements:

It is with pleasure, however, that I acknowledge the encouragement and guidance of my supervisors Dr Peter Denyer and Dr Gordon Brebner, and the assistance and conversation of the members of the Integrated Systems Group especially Dr David Renshaw and Dr William Blackley. I am also grateful to the staff of the Edinburgh Microfabrication Facility for accommodating my requests so readily.

Finally I wish to thank my Amy for understanding those evenings I spent with her only rival: the computer.

Ad Majorem Dei Gloriam

CONTENTS

Chapter 1

BACKGROUND AND MOTIVATION

This is a project on Content Addressable Memory (CAM). It considers a minimal component for the storage and retrieval of data according to the results of a masked comparison, and in this context examines Content Addressability across a wide spectrum of computing: from transistor level hardware, through computer architectures, to programming language development. It details a development of the question: what if we alter the computer's mode of addressing?

1.1 To begin at the beginning

In the "First Draft of a Report on the EDVAC"[1] in 1945, von Neumann delimited five parts of "the functioning of the contemplated device": central arithmetical, central control, memory, and input and output with the outside recording medium. Of the "memory part" he wrote: "While it appeared, that various parts of this memory have to perform functions which differ somewhat in their nature and considerably in their purpose, it is nevertheless tempting to treat the entire memory as one organ, and to have its parts even as interchangeable as possible". Subsequently, developers have indeed yielded to this temptation. The successors of EDVAC have all realized memory as fixed length binary storage locations which are addressed by their physical position; despite von Neumann's early flirtation with the neural studies of Pitts and MacCulloch. By 1946 the idea of a memory hierarchy was established[2], in 1949 the Manchester University group conceived of the index modification of addresses which lead to the virtual memory

concept, and in 1958 the stack was invented by Barton. These contributions are early examples of software abstractions over the memory realization: abstractions which have lead to the "conventional" programming languages of today. They have ensured, by their successful development on the EDVAC model, that the singular nature of the "memory part" has remained unchallenged within the von Neumann architecture.

The architecture itself, however, is much criticized. With the potential offered by the increasing complexity of Integrated Circuits, Computer Architects have used distributed processing to distance themselves from the von Neuman model – however, little consideration has been given to modest changes in architectural design. Specific problems have quite rightly received specific architectural solutions, but the development of general computing machines is still dominated by refinements to von Neumann's EDVAC. Alternatives such as dataflow and reduction machines are promising but as yet unproven, and these too retain the established addressing mode for the "memory part". Separate processors were developed in the 1950's to handle input and output so freeing the "central control part" from the most fundamental operations, co-processors, pipelined and finally hardwired special purpose arithmetic units have enhanced the "central arithmetical part" (culminating in "super computers" such as the Cray series) – but the model remains the same.

From EDVAC onwards, memory has been addressed by a "memory location number". An alternative is to address memory by content: to direct words according to what they have become, rather than where they were placed. This project investigates that alternative.

The motivation for this topic stems from the observation that Content Addressability would:

- assist software development by providing a new medium for the design of data structures and algorithms

and through its affinity with certain data types

- provide a general component to facilitate system design for innovative computer architectures and programming languages.

To corroborate this observation, this thesis sets out to describe the design of a suitable CAM component and to prescribe its inclusion in an example, full programming language environment. This approach was adopted because it is the author's belief that a component can only be correctly designed in the context of a parallel development of its projected environment. In this case, that philosophy dictates the development of a full programming environment which provides access to the CAM component in its role as a medium for software engineering.

The example considers the CAM in an enhancement to the von Neumann model: the "memory part" is altered. We could consider replacing the RAM with a CAM, and simulating the desirable RAM features using the CAM; however, we will consider instead an architecture which includes both modes of addressing implemented by their respective memory components. We will consider a general purpose computer architecture in which the central control interacts with both a RAM and a CAM, and we will call it the CAM-CPU-RAM *Triplet* model. With both types of memory available, the software engineer can choose the more efficient medium for the implementation of a data type. The questions we will consider are: the design of a CAM, the characterization of a CAM, the *Triplet* architectural organization, and the development of programming languages which render the CAM accessible to the general programmer.

This chapter continues with an historical survey of CAMs and an examination of the application of content addressability to basic computer programming. Chapters 2 and 3 consider the development of a VLSI design from a minimal specification, through the design, fabrication and testing of two prototype components, to a proposal for a natural extension into a wafer scale design. Chapter 4 considers the characteristics and possible role of the resulting component. In chapter 5, we look at the relationship between innovations in hardware and software in general, to discern how each has developed to accommodate the other, and so seek to establish guide lines to facilitate the development the full programming environment for the CAM component. We attempt, in chapter 6, to characterize the CAM in the context of the *Triplet* architecture and then as a computational abstraction – so that chapter 7 can demonstrate that these perspectives may be practically combined by a high level programming language.

1.2 Content Addressable Memory History

This section presents a brief review of content addressable memories and current MOS realizations; more comprehensive reviews can be found in Parhami[3] and Kohonen[4].

1.2.1 What is CAM?

Content Addressable Memory consists of storage locations divided into words which are addressed, and so directed, according to the results of a masked comparison. This operation is performed by a comparison between a specified *comparand* and each word in memory over those bits which correspond to the masked bits in a specified *mask* word. Thus an address is defined by a *comparand* and a *mask* word, and an address defines zero of more words in memory. In general the comparison may be made

according to a variety of relations: equality, inequality, greater-than, less-than, etc..

Elsewhere the term Content Addressable Memory is used to include a variety of SIMD processors and even "neural" networks: this thesis restricts itself to the design and use of a memory device. The stage at which a memory with distributed logic becomes a processor is, however, difficult to specify. This stems from the development of CAM in the context of logic distribution among the memory elements, in which comparison is seen as the simplest logic in a scheme which can include word specific ALU functions and inter-word communications – this has lead to a variety of Content Addressable Parallel Processors[5], including Batcher's Massive Parallel Processor[6] which was designed for image processing at NASA. We will consider only the hardware design and usage which is necessary to the *address* operation itself and exclude devices with additional local processing and communications; this rules out the various array processors with CAM capability, and also content driven special purpose hardware such as hardwired concurrent bubble sorts[7].

The only established use of CAM in commercial computing systems is in the implementation of virtual memory, first developed in the IBM system 360 mainframe series. The technique is to maintain a memory hierarchy whereby the sections (or *pages*) of memory which are currently being used by the programme are loaded into a faster (more expensive) forward memory device (or *cache*), and the words are accessed there rather than in the slower (cheaper) secondary storage. To provide a mapping to the memory locations in the cache, a content addressable store is maintained which maps the page address to a cache offset if that page is already present. This CAM realization was possible since the required component need perform only a limited set of operations, and need not be large since the cache address space is limited by the cost of the cache itself.

1.2.2 Categories and distinctive features

Even restricted to purely memory devices, CAMs offer a wide variety of possible architectures and operations. The fundamental categorization reflects the degree of parallelism inherent in the logic distribution within the memory store; other distinctions arise from the different solutions to common problems.

Parallelism

Content Addressable Memory requires that a comparison operation be performed on every stored word in memory, and the various CAM designs may be usefully categorized by the degree of parallelism used in this operation. The first examples can be found in multi-track disc access systems such as CASSM[8] and RAP[9]. These systems were seen as data base subsystems to the main computer and were generally supported by considerable local data base processing. The idea is that comparison and access logic is associated with long serial tracks of memory, and the design is geared to fast serial processing of specific operations. The parallelism stems from the simultaneous use of several disc tracks, and the cost of serial processing is offset by the 'tuned' design of hardware logic and choice of technology. Modern proposals include memories in CCD shift registers, bubble memories, magneto-optics, and holographs.

Integrated circuit CAMs differ in that the comparison and memory logic are both implemented in the same technology, and thus are designed with one of the four possible degrees of parallelism according to whether the words and/or bits are compared serially or in parallel. The complexity of the matching logic can vary from the implementation of a simple masked comparison to a range of boolean operations.

Full parallelism is the extreme view achieved by distributing processing logic to each memory bit cell. In these systems the comparison is performed simultaneously at each bit with the mask, comparand and control signals being distributed to each word.

The use of a bit-serial word-parallel configuration has the advantage that the cell size can be reduced to a storage element, with the comparison logic removed to one location for each word. A bit-parallel word-serial configuration similarly allows for a reduced bit cell size but, with a sequential processing of the memory words, it is severely limited in its access time by the size of memory. A fully serial configuration is even less attractive for the same reason.

Location addressing

Some CAM designs include the concept of location address producing a hybrid memory[10] where the portion of memory over which a comparison is to be performed is limited by the degree to which the physical location has been specified. This allows the memory to function in either location or content address mode. One advantage is with pointer driven programming systems, such as LISP, where the dual address mode allows the existence of pointers to a specific cell location to be deduced from a comparison over the pointer fields[11].

There are three consequences of the hybrid architecture:

- The memory space may be divided into different regions by the software. This allows different search spaces to be defined by the physical division of the memory rather than by a conceptual division through the dedication of a CAM field (thus saving memory space).

- The result of a comparison may be given as a physical address by which the word is subsequently directed, rather than using controlled access according to a tag store.

- The word-length of CAM may be increased if the comparison can be performed with reference to the result of the previous comparison on a physically adjacent word[12].

As an example of the latter, consider doubling the word length: firstly set all the result tags on each of the even addressed words in memory and compare with the first half of the new virtual word resetting the tag on a mismatch, then transfer that result into the next physically adjacent result store and perform the comparison on each odd addressed word with the second half of the virtual word – the tags which remain set indicate the address of the second half of each matching virtual word.

Multiple Responders

The main difference between addressing by content rather than by location is that in CAM the number of responders to any address is not fixed but rather is a function of the state of the computation. Thus there may be zero, one, or many words which are addressed as a result of a comparison. For some operations, such as *write* and *remove*, this can be used to advantage by performing the operation on all responders in parallel. With the *read* operation in word parallel architectures, however, this raises the problem of contention on the output data bus.

The problem is avoided in some CAM designs which are targeted at a specific system whose operation ensures that there is no more than one match for any valid address. For example, when using CAM as a fast lookup of cache memory addresses, an entry is only made if that page address was not found by the proceeding comparison. In such systems there is no multiple response, and therefore no contention. In general, however, some logic must be devoted to resolving this conflict. There are essentially two options: access one responder only, or access each responder in turn.

In the former case, some system must be devised whereby other responders can be accessed as the result of performing the search again. This can be implemented by setting a marker either in the word or in the hardware which prevents the already accessed word from responding again.

In the latter case, the scheme requires some form of local storage of the results of the comparison, and signal switching so that the 'first' responder is affected by subsequent memory operations. There must also be a mechanism whereby the 'first' responder can be passed to allow the next responder to be directed. A major problem is the delay in the propagation of the addressing signal through the memory. With a large number of words, this becomes prohibitive. There are two approached to this problem: the use of look ahead circuitry on the same principles as the Manchester Carry Chain Adder, or the use of a modular hierarchy with logic to prevent the addressing signal from descending along paths which lead to modules without a responding word.

Garbage collection

Free locations pose two problems: how is free space addressed, and how is assigned space returned to free space? The identification of empty words has received scant attention in previous designs. There are two approaches: unused words are identified by a bit field in the memory word by which they can be addressed (by its content), or alternatively some systems keep all the assigned words together. A recent scheme[13] was designed in which the (one if any) matching word is immediately read and removed. The memory system maintains a "depth of memory" pointer to which new words are written, and when a word is removed, the word at the bottom of memory is rewritten into the space vacated, and the pointer is reset accordingly. A similar design has been implemented[14] in which the memory is constructed as a bit parallel stack. Words are entered by pushing them onto the stack, and vacated space is filled by shifting upwards those words further down.

1.2.3 Previous MOS Designs

The majority of proposals for MOS CAMs have been fully parallel so as to take full advantage of the processing potential of the technology, despite the restriction on component capacity imposed by the large resulting bit size. Bit-serial word-parallel designs have been proposed mostly for the implementation of more complicated functions such as ordered retrieval[15].

The standard CAM cell design for a fully parallel system is based on a static RAM latch with additional circuitry for performing the masked comparison (for example, see Lea[16]). Commonly the comparison is performed by discharging a precharged word reference line through transistors associated with the mismatching bits. This design leads to a large cell size which was quickly recognized as a severe limitation. The use of dynamic storage nodes was seen as a partial solution to this problem. A four transistor bit cell was proposed by Mundy[17] which is based upon two storage nodes for each bit cell, and which allows for the storage of a "don't care" state in the actual memory cell as well as the normal use of an external mask. The design requires three distinct voltage levels, and the use of current flow is suggested as an analogue counter of the number of responding words. A half cell of the same design was proposed by Lea[18] as a memory which matches against only one logical value. A recent paper by Wade and Sodini[19] modified the Mundy cell by cross-coupling the bit lines to preserve better noise margins.

1.3 CAM as a Software tool

Let us now return to the observation that CAM will provide a new medium for software engineering. Conventional software techniques have evolved to counter the difficulties which have been found in computing with the von Neumann model. We will apply the topics from a standard text on computer algorithms and data structures, to a memory system which is addressable by its content; and so present CAM alternatives to techniques which "underpin much of today's computer programming"[20]. The CAM is used to implement common *abstract data types* in a manner which is usually simpler, more efficient, and often more powerful than the RAM equivalent.

We will leave the development of a precise terminology of CAM programming until chapter 6, and use instead the model of a number of fixed length words which are each conceptually subdivided into named *fields*. We will assume that the memory is 'sufficiently' large for the following abstract overview of CAM applications, and that the configuration is word-parallel so that the timing of a comparison operation is independent of the number of words in memory. The 'value' of a memory field may be considered as data or as address or as both. The use of field identifiers is similar to that in *record* structures of conventional RAM programming except that CAM words are addressed by the values of (some of) its fields rather than by the pointer identifiers which address *records*.

In the following, the CAM word will be referred to in terms of the field names associated with a particular problem. Thus

[field1 | field2 | field3]

depicts the CAM word as having three fields. However, this does not imply that only one convention may be extant in memory, since

[convention_field | field1 | field2 | field3]

could be a software convention for supporting several sub-conventions simultaneously. This flexibility is, of course, bought only by smaller virtual word sizes since some bits must be dedicated to the convention_field.

1.3.1 RAM simulation

We start with the trivial observation that a CAM may simulate a RAM by the convention:

[address | data]

RAM is normally accessed by passing the address through hardware decoders; CAM distributes the 'address', and 'decoding' is achieved by a comparison over the address field. The obvious disadvantage is the use of storage area to describe the address, but this inefficiency is in no way daunting since we should not expect CAM to surpass RAM direct through mimicry; the virtues of CAM will be found in exploiting its own characteristics.

A RAM may simulate a CAM, but the free specification of the mask word necessitates that the simulation is performed by a sequential search through all the memory words. Thus, although the CAM-CPU model is "computationally equivalent" to the RAM-CPU model, the RAM's simulation of CAM is of a higher time complexity – this disparity is due to the parallel processing inherent in the CAM model.

The value of the address field need not, however, be equivalent to a RAM memory location number. Instead it could be a representation of the variable's identifier from the high level programming language. To allocate storage in RAM, a compiler must translate a programme's variable names into numerical location addresses. A Fortran compiler makes all allocations before run time, while that of a recursive language must

specify storage in terms of an offset from a 'data frame base' address. If the memory were a CAM, the compiler could use the variable names from the high level language to generate symbolic addresses whose uniqueness, in any one procedure, is ensured by the programme semantics. By adding a procedure identifier to each symbolic address, a Fortran compiler could perform all allocation without counting storage cells. If a *recursion level* field were also included:

[unique_identifier | level | data]

the compiler of a recursive language could perform allocation without stack maintenance.

This is a possible approach for implementing the conventional languages on a *Triplet* machine, but the use of CAM alters some of the basic features of the programming environment which have directed their evolution. For instance, block structured scope rules reflect the attributes of the stack which provides structure to RAM usage; but with CAM the access to "non-local" variables can be no more expensive than to local ones. It thus seems likely that new 'CAM orientated' languages would evolve to reflect the new features which CAM provides.

So far we have seen that anything RAM can do, CAM can do also, and that CAM can do certain things better than RAM. We proceed by considering CAM-specific implementations of some basic abstract data types. In general, the point is not that CAM surpasses RAM, but rather that CAM provides a comparable implementation with distinctive features which a software programmer might wish to exploit; however, in some cases the data type's correspondence to the CAM itself affords a superior implementation in CAM than in RAM.

1.3.2 Vectors

Vectors do not fit into the unique naming scheme since the word to be addressed is normally determined during run time and so the compiler can not assign a unique name before hand. However, the same calculation of an "offset" value, as found with RAM, could be employed in a CAM implementation. The scheme:

[identifier | offset | data]

implements the vector data type, with slightly different features from the RAM implementation. For instance, this representation would accommodate the storage of large 'sparse' matrices, where only the (few) significant entries are stored. A RAM implementation of this technique requires the overheads of a pointer mesh, the CAM implementation is the same as for 'full' matrices.

A vector has two basic properties: an ordering of its elements (implicit in an order relation on the offset domain), and also an association under the common vector name. If order is required it can be provided explicitly: in the *offset* field. The number of elements so ordered is limited not by the vector size but only by the range of the assigned field – repeat entries are also possible. An ordering in CAM can be found in the necessarily sequential storage and reading of addressed words, but this is a feature of the hardware and not of the CAM model – any use made of this ordering must be described in terms of the specific design and thus is not valid testimony in support the general CAM.

However, vectors are sometimes used in programming for association alone, and the offset only assists in programmer maintained RAM allocation – in such cases the offset field is redundant in CAM: association follows directly under a common identifier.

1.3.3 Lists, Trees and Queues

Lists and trees are traditionally built out of pointer structures where the pointer is the 'address' of a topologically adjacent word, and like vectors they too provide both order and association. By replacing the address with a CAM word identifier, these structures may be realized with a CAM through mimicry of the RAM technique. An implementation of lists could use the fields

[identifier | next_identifier | data]

and this idea can easily be developed into tree building. A significant advantage of the CAM implementation is that 'backward chaining' is immediate without the need for extra storage space and maintenance of reverse pointers.

However, CAM has distinctive features which allow different concepts to be used in algorithmic design: the free choice of address 'names' allows the programmer to develop naming conventions so that the 'name' itself contains information. For instance, a binary tree can be specified by a binary code where the root is *1*, its two branches are *10* and *11*, the next level is *100*, *101*, *110*, and *111*, and so on. Similarly, queuing may be supported through a circular buffer by the data structure

[counter | data]

where the counter field is an integer maintained with addition modulo the field's range.

1.3.4 Hash tables

Deep in the folk-lore of RAM programming lurks the hash table. Even before FORTRAN, programmers needed a large table which could be quickly updated or searched for a specific entry, to assist the translation of symbolic names from assembler code; and so they devised the hash table. The implementation of the hash table in RAM

requires the permanent allocation of a large array (necessarily larger than the maximum possible number of entries), a judicious choice of 'key' calculating algorithms, and the choice of strategies to accommodate collisions involving heap storage maintenance or key recalculation. Despite these difficulties, hash tables are so widespread that some even advocate their inclusion in machine code instruction sets[21].

CAM is a hardware solution to the problem. Lookup is performed by addressing the word directly (according to *any* field), and update is lookup followed by writing. There is full utilization of memory with constant access times, no complicated hashing function, and no lists or incrementing of the index to handle collisions.

1.3.5 Graphs

A natural way to store graphs using fixed fields is according to the edges defined by their end nodes:

[tail_node | head_node | (cost)]

This is ideal for a (possibly costed) digraph, and non-directed graphs can be implemented by either recording an edge in both directions or searching both fields in the software. The most significant feature here is the closeness of this representation to its mathematical specification: there are no supporting data structures beyond the graphs actual definition.

1.3.6 Sets

We will consider the implementation of sets in some detail to provide a heuristic appraisal of CAM features necessary to support an efficient implementation of this data type. With RAM memory, sets and set functions are supported using hash tables, lists,

or complex tree structures, yet a set is simply an association of its members under a common identifier. A CAM could use the data structure:

[set_identifier | item]

to provide this association, allowing a set to be addressed directly by its identifier without any supporting data structures. In considering the implementation of set functions, however, we immediately encounter memory design issues.

Consider how to implement a *copy* function: create a set B with the same elements as set A. This requires that the set A be addressed by a comparison operation over the set_identifier field and each responder read (non-destructively) in turn and written to free space in memory with an updated set_identifier. If the comparison logic is involved in designating the free space, the write can not be performed without either destroying the results of the previous comparison, or using different comparison logic. One solution would be to place the result of the first comparison (the words addressed as set A) into an external buffer, and then to write the buffer to free space after a further use of the comparison logic – this is unattractive since it requires a large amount of mostly redundant circuitry. An alternative is to make the addressing of free space independent of the comparison logic used to address set A.

Consider now the *Union* of two sets: create set C which contains all the members of sets A and B, but without repetition. This can be performed by copying set A to C and then copying to C those members of B which are not in C already (or A). To do this, there must be some method for performing a comparison on each element of B against the elements in A *without* affecting the comparison which is addressing B. Again one solution would be to buffer the elements of B, and the alternative is to provide logic to interrogate the members of set A which is independent of the comparison addressing B.

Assuming the second solution in both cases, the standard set functions may be implemented as follows:

UNION(A,B,C):: copy A to C; while reading B, test each word for occurrence in A; if not in A, write that word to C. Timing is O(|A|+|B|).

INTERSECTION(A,B,C):: while reading A, test each word for occurrence in B; if present in B, write word to C. Timing is O(|A|).

DIFFERENCE(A,B,C):: while reading A, test each word for occurrence in B; if not present in B, write word to C. Timing is O(|A|).

MEMBER(x,A), **MAKENULL(A)**, **INSERT(x,A)**, **DELETE(x,A)**, and **FIND(x)** are all obvious and performed with one addressing of memory.

The boolean **EQUAL(A,B)** is achieved by performing DIFFERENCE(A,B,$) and DIFFERENCE(B,A,$) until an assignment is made to $, or both functions are finished. Timing is O(|A|+|B|).

The function **MIN(A)** returns the least member of a set where there is some ordering defined on its members. The simple CAM implementation is to address the set A and then cycle through its members performing a comparison with a 'currently minimum' value: timing is O(|A|). However, if the values are non-negative integers, the techniques mentioned in the next paragraph lead to an O(m) algorithm where m is the number of bits in the order field.

1.3.7 Sorting

As we have already seen, it is possible to impose an ordering on the elements of a CAM through the inclusion of an *order field* – however, this is effecting (through software) a property which is implicit in RAM and so is not evidence of CAM's claims over RAM for data structures which do not derive other benefits from CAM implementation. Sorting relies entirely on order and will, generally, be better implemented in RAM. However, there is a special case when the memory bits are interpreted as non-negative integers. The algorithms were described originally by Falkoff[22], and depend upon the partitioning of the binary tree inherent in the bit representation itself, through the manipulation of the comparand and the mask word. With these techniques, a sort of N values specified in a field of m bits requires time of O(Nm) – a fuller explanation appears in chapter 4.

1.3.8 Algorithms

The above is an example of a CAM specific algorithm, relying on subtle manipulation of the addressing mechanism. However, we might expect algorithmic design to be affected more generally by the use of CAM.

Consider the following algorithm for the traversal of a digraph, using the edge representation above and two sets ('visited' and 'to_do') with the associated set operations.

```
Procedure Traversal(start_node)
 {INSERT(start_node, visited)
  INSERT(start_node, to_do)
  while 'to_do' is not empty
   {select_node from 'to_do'
    DELETE(selected_node, to_do)
    foreach edge with tail_node = selected_node
     {if NOT MEMBER(head_node, visited)
       {INSERT(head_node, visited)
        INSERT(head_node, to_do)}}}}
```

Each node of the graph is visisted once by this procedure. The set 'to_do' is an unordered set of visited nodes adjacent to those which might not yet have been visited. The order in which the graph is traversed is unspecified. The algorithm is trivial – but, in CAM, so too is its implementation since the implicit data types map directly onto the memory architecture.

An order may be imposed on the traversal by including an index field with the node entries in the 'to_do' set to store the value of an incremented counter, thus numbering the members of that set to form a queue. The search is then breadth_first or depth_first according to whether the queue is FIFO or LIFO respectively. The same queues can be found in the least time programming in RAM where depth_first search is performed using the stack as a LIFO queue, and breadth_first search requires explicit FIFO queue construction – but in RAM this symmetry is obscured. The CAM version hides no complex structure and needs no implicit system (stack) maintenance; the mapping from the abstract algorithm to physical memory component is more direct with CAM than with RAM.

1.3.9 Assessment

CAM is a useful memory design. It is capable not only of mimicking RAM but also of surpassing it in some of RAM's most popular uses. Further since current techniques assume and accommodate a RAM architecture, a CAM based programming system can be expected to promote a new approach to programming which exploits the distinctive features.

Chapter 2

COMPONENT DESIGN

This chapter considers the design of a suitable Content Addressable Memory component. After stating the design criteria, we develop from an abstract definition of the CAM component into a description of its features; this construction being sufficient rather than necessary. A circuit level realization is then advanced from a general overview to the details of a specific chip's organization. This is followed by two modifications, providing fault tolerance and an increase in operating speed, and a projection of methods by which small CAMs may be configured into a large memory component.

2.1 Design Criteria

From the history of previous designs and the observed requirements of common abstract data types, we now consider the criteria with which a suitable CAM component may be designed.

2.1.1 Minimality

In the EDVAC report[1], von Neumann exhorted that: "The device should be as simple as possible, that is, contain as few elements as possible" and this emphasis on minimal hardware is perhaps the best explanation of the model's continuing success: a simple framework allowing diverse development. The effectiveness of minimality in VLSI systems is most forcefully advocated by the RISC school of designers who

originally set out "to obtain as much performance for as little complexity as possible"[23] and produced "a single-chip CPU that rivals board level designs"[24]. With this in mind, the CAM design was kept as simple as possible – whilst remaining useful.

The aim is to produce a general purpose memory component. The approach is to refrain from including complex processing and to use only those features which are necessary to that aim. The reasoning is that the simple design is likely to be both realizable and broadly applicable, and that by proving the kernel of possible components, we will implicitly approve them all. As will be seen however, once the basic design was fixed, the resulting component was both flexible and easily ramified.

2.1.2 Utility

The utility criterion will be interpreted as an efficient implementation of the data structures and algorithms described in chapter 1 since these cover the fundamental aspects of software engineering practice. In section 1.3.6, we found that an efficient implementation of *sets* requires:

- a write_new_word operation.
- a boolean test which determines whether or not there is a word in memory addressed by a specified content.
- a search_and_queue operation such that the queue of results is maintained whilst either of the other two operations are executed.

This is also sufficient for the other data types considered. These are the minimal *functional* requirements to implement *sets*, but 'design' minimality is approached by reducing the number of distinct functions to bc implemented. The boolean test

operation requires the same search to be made of memory as in the search_and_queue operation and so may be implemented by a second independent search_and_queue operation. These two operations must maintain their respective queues independently from each other and also from the execution of the write_new_word operation. The boolean test can be resolved according to the existence of words on a queue.

Having decided to include a second search_and_queue operation, we could proceed to include even more since the design effort involved in two may trivially be extended in more than two. However, two such operations (and an independent write_new) are sufficient to implement *sets*, and also the other data structures which we have considered, and so, invoking minimality, we will settle on two as a basis for the prototype design.

2.2 The Abstraction

With these criteria in mind, let us develop a specification of the component's organization; the resulting floor plan is shown in figure 2.1.

- Content Addressable Memory is storage addressed according to a subset of its contents.

- To realize this concept in physical storage space, there must be a comparison between the relevant bits of each memory word and the corresponding bits of an addressing comparand, according to which the addressed (matching) words may be accessed.

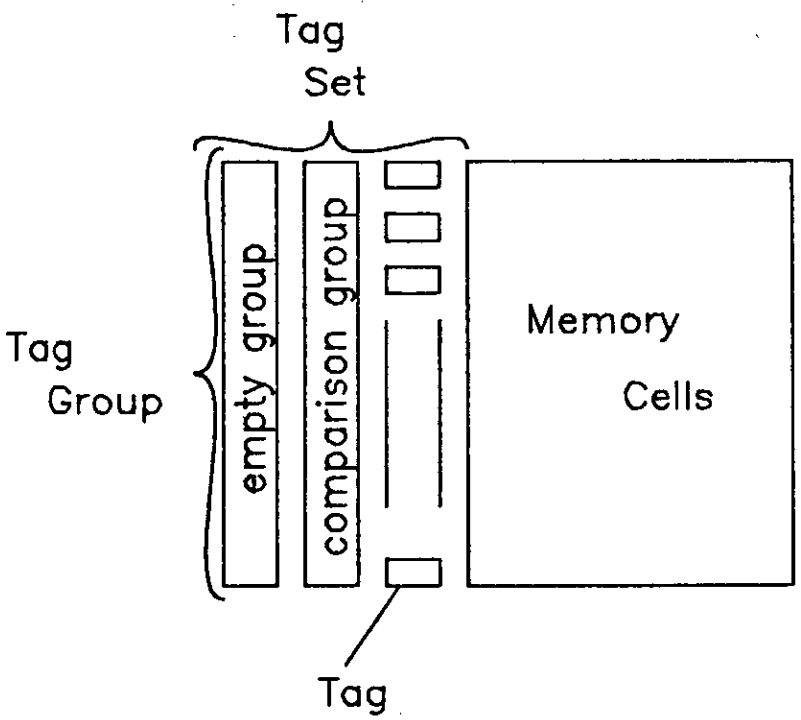

Figure 2.1: **the structural floor plan**

- To designate which bits of a word are relevant to the address, there is a *mask* word whose ONE bits denote relevance, and whose ZERO bits denote irrelevance, to the comparison of the corresponding bits in the comparand and memory words.

- The results of a comparison may be stored to avoid contention from multiple responders, and to allow controlled access. This storage must be associated with the corresponding word. The storage location with each word is known as a *tag*. A tag is said to be *active* if the word is addressed by the comparand, or *passive* otherwise.

- If the results of n (> 1) comparisons are required together there must be n independent *groups* of tags: n tags associated with each word. A word is said to have a *set* of tags associated with it.

- A word is manipulated by addressing a tag group and thereby addressing one of the words whose tag in that group is active. Such a word is said to be *currently addressed*. If there is no active tag in the group, a signal is output when the group is addressed.

- Further words with active tags in the group may only be addressed by making passive the active tag, in that group, of the currently addressed word.

- A currently addressed word may be written or read.

- The concept of a tag group is extended to address *empty* words: those whose contents are undefined or no longer desired to be addressed by a comparison. This tag group is not available to store the results of a comparison. It is known as the *empty tag* group, while the other groups are *comparison tag* groups.

- A word which is empty can not have active tags in its comparison tag set: in any set, an active empty tag implies that all the comparison tags are passive.

- The CAM is initialized, or emptied (re-initialized), by making all tags in the empty group active. The content of empty CAM words is undefined.

- When a word in memory is written, its contents are defined and the empty tag in its tag set becomes passive.

- A currently addressed word may be made undefined by making its empty tag active – this necessarily makes its comparand tags passive.

2.3 Fundamental Organization

This section presents the overall design decisions which determined the general organization of the hardware component.

2.3.1 Bit-serial Word-parallel Architecture

The operations described above occur only at word level: the addressing mechanism defines words, and so the addressing logic need only be distributed to the word level. With the comparison logic at word (rather than at bit) level, the bits of a word must be fed through the comparison logic sequentially and the operation may be performed in parallel on all words in memory.

By moving the logic from the bit to the word level, a smaller memory cell design is possible but the comparison can only be performed one bit at a time. This is a trade-off between the speed of operations and the size of the associated circuitry. Other advantages in making the chip bit-serial are that the data may be distributed to the words

as it arrives at the chip without local buffering, and that the (potentially large) number of pins is reduced.

2.3.2 Rotating Word Model

The comparison operation occurs at all words simultaneously as the whole of memory is rotated through a word cycle. I/O operations require the rotation of only one word, but by using the addressing mechanism as a communication switch between the data buses and the addressed memory word, the bulk memory rotation may still occur.

This leads to the design model of circulating memory words which may be implemented either as actually rotating storage locations: using shift registers, or by cyclically addressing a static configuration: using, for instance, bit slice addressing with standard RAM-like circuitry. The latter may eventually lead to higher bit densities with dynamic storage but this is not necessarily advantageous because the word pitch is also limited by the word dense addressing and comparison mechanism which requires static logic and so seems to set the lower limit.

For a practical realization, semi-static shift registers were used (see figure 2.2). With clock H off, clocks A and B form a dynamic shift register configuration and allow the memory word to rotate; with clocks A and H on and clock B off, the bits are statically held. This has a speed advantage over RAM in that there is no delay associated with charging a word line since the current bit is always in the cell adjacent to the switching logic. Also the shift registers lead to a very compact layout.

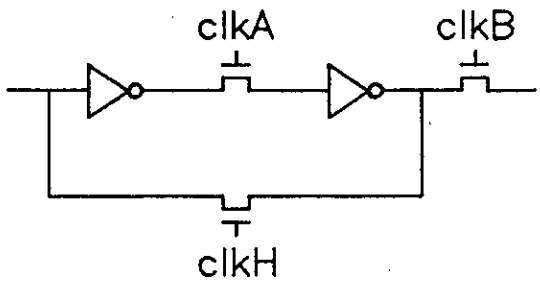

Figure 2.2: a semi-static shift register bit cell

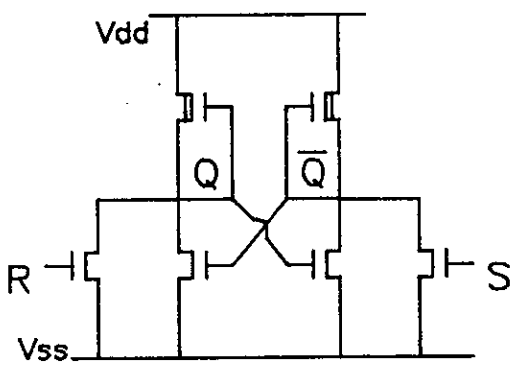

Figure 2.3: the latch realization of a tag

2.4 The Addressing Circuitry

We now consider specific circuitry for the realization of the abstract model. The major feature is the addressing mechanism which is to be based upon the value of tags associated with each word of memory; the problem is how to cope with multiple words matching on a comparison operation.

2.4.1 The Tag

The *tag* storage of a comparison's result is realized as a reset latch (figure 2.3). The value of the latch corresponds to Q: the latch is *active* if Q is high, and *passive* if $\overline{Q}$ is high. The value is switched by pulling the high arm to ground through the corresponding pass transistors: R or S. The value is changed either by commands acting on a single addressed word, or by the comparison and CAM initialization operations.

2.4.2 The Comparison

The primary function of the tag is to record the result of a comparison: the tag is active after a comparison if the corresponding word matches the comparand at all the masked bits. This may be translated into a practical algorithm as the tag becomes passive after a masked mismatch.

The bit-serial comparison operation begins by making all the tags in a group active. The masking of the comparison is performed by clocking the reset circuitry with a signal which occurs if and only if the corresponding mask bit is set. The reset signal at each memory word is generated from the comparand and word bits according to the logic in figure 2.4. The signal is high for a mismatch and low for a match, for instance: if m is high, the node x is connected to the arm with the value of c making x high if c matches

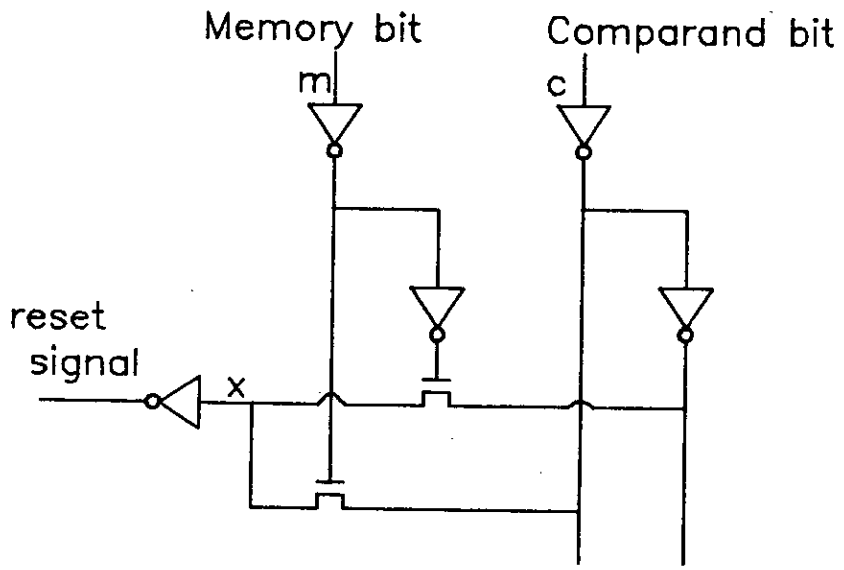

Figure 2.4: the comparison logic

m and low if not. To perform the comparison, the comparand bits and their compliments are broadcast sequentially as the memory words are rotated and the reset signal is clocked in the selected comparison tag group for those bits which are masked. The tag becomes passive on the first masked mismatch; that is, the tag is active at the end of the sequence if and only if there was no masked mismatch to reset it to passive.

2.4.3 Addressing Logic

A tag group consists of active and passive tags. When a tag group is addressed, one word with an active tag in that group is currently addressed. This is realized by using the latch values to pass a signal through the latches until the first active tag is reached. The circuitry at each word is shown in figure 2.5. The tag group is addressed by raising the group address signal line. If the tag is passive, $\overline{Q}$ is high and the pass transistor is *on* allowing the signal to propagate down the line. If the tag is active, this pass transistor is *off* and the isolated lower section of the group address line is pulled low since Q is high.

The word associated with the first active tag is then *currently addressed* as the word address line is pulled low through the two transistors gated by the latch value Q and the group address signal line. The word address line may be pulled low by any of the tag groups. Some operations on the word are performed by signals which affect only a word whose word address line is low: by design there is only one such word for each addressed tag group, and in normal usage only one tag group will be addressed at a time.

Addressing another word in that group is achieved by making the selected tag passive and then allowing the group address signal to propagate to the next active tag. This operation is initiated by a signal rather than automatically since there are various

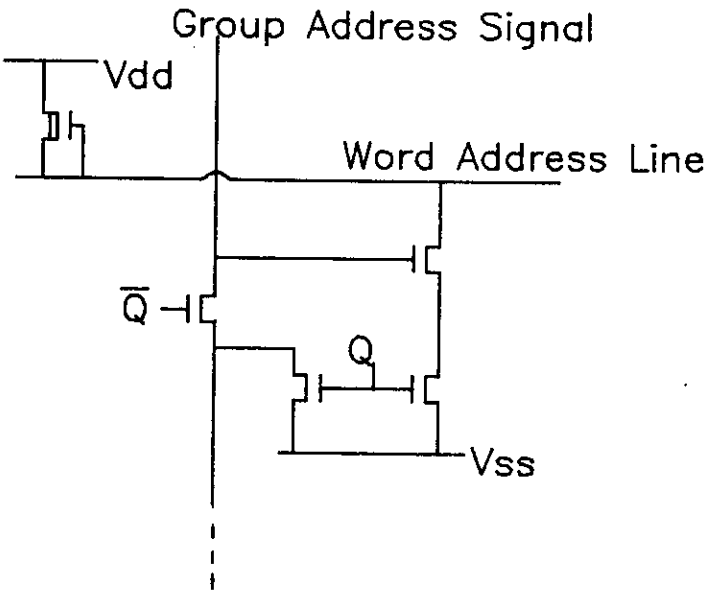

Figure 2.5: the addressing logic

(*Group address signal assisted by restoring buffers every four stages*)

operations which may be performed on the addressed word. This scheme is effectively daisy-chain addressing where each site is polled in succession until one responds, and the polling continues when the responding site has finished the required operations.

If there are no active tags in the group, the signal emerges at the end of the addressing line and is output from the chip.

2.4.4 Non-cascading signals

If the tag, by which a word is currently addressed, is made passive the group addressing signal propagates to the next active tag. There is here a potential race hazard since the signal to make the tag passive must be removed before the next word becomes currently addressed or its tag also will be made passive. The dynamic circuitry shown in figure 2.6 prevents this cascading. It was verified both by SPICE simulation and by fabrication as a small test structure before inclusion in the full design.

The control signal is generally high. If the word is not currently addressed the word address line is also high, making node x low, and connecting node y to Vss. A pulse on the control signal does not effect the node values. If the word address line then becomes low, y is isolated and discharged. If the control signal is subsequently lowered, node x rises and node y is connected to Vdd. The pass transistor beyond the buffer is *off*, since the control signal is low, and so the reset signal still remains low. When the control signal is raised, this pass transistor becomes *on* and the reset signal rises. Also the node x becomes low and node y is isolated but charged, dynamically maintaining the high input to the buffer. The circuit is self timing in that the reset signal remains high until the tag is made passive which in turn raises the word address line so discharging node y and lowering the reset signal. Thus, the system returns to the initial configuration.

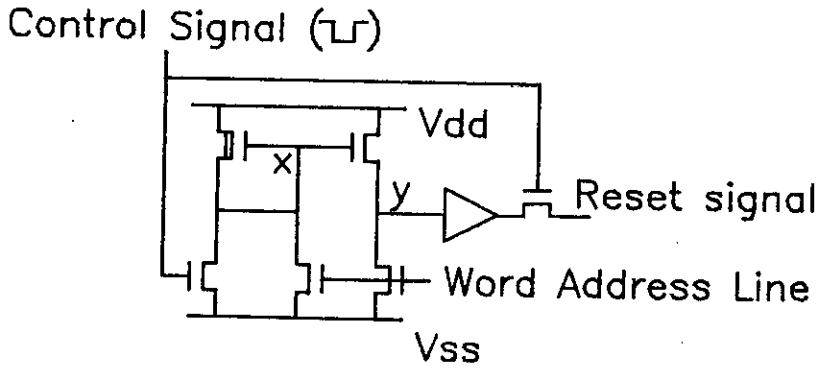

Figure 2.6: **the non-cascading unit**

2.4.5 The empty tag group

The idea of addressing the words in memory according to the setting of an associated tag is extended to overcome the problems of identifying free-space and garbage collection. A solution with similar features has been published independently by Ogura[25].

The empty tag operates without reference to a comparison operation. The whole group is made active by a signal to initialize the memory and an empty tag must be made passive when a word is written to its associated memory word. Thus the write operation is used to make passive the empty tag of the currently addressed word. An empty tag may be made active again to allow a word to be made unaddressable and so effectively erased, thus allowing explicit garbage collection to be performed through the control signals.

If a word's empty tag is active, the memory bits contain undefined data. To ensure that this does not lead to an undefined word being addressed through a (random) matching comparison, the comparison tags are constantly reset by the value of the empty tag. Thus the comparison tags of an empty word are never active.

2.5 Control Circuitry

To design an actual component requires the specification of the overall control logic. The following is one possible realization: the design is built using standard digital circuits and the two phase non-overlapping clock normally associated with NMOS shift registers. Other technologies or other realizations of the rotating word model might require different control techniques.

2.5.1 Internal Control Summary

Each comparison tag group has the control lines:

Reset: making all the tags active as the first part of the comparison operation.

Mask: clocks the reset of all the tags according to the value of the mask word during the comparison.

Next: resets the tag of the currently addressed word to passive using the non-cascading control unit.

Address: addresses the tag group.

The empty tag group has the control lines:

Empty: making all the empty tags active.

Defined: a non-cascading unit signal which is completed at the end of a write operation.

Remove: making the empty tag of the currently addressed word active and so undefining that word.

Address: addresses the tag group.

And the remaining general functions are:

Read: to output the contents of the currently addressed word.

Write: to input data into the currently addressed word.

2.5.2 Clock and input organization

The chip is driven by a two phase non-overlapping clock: clk1 and clk2. The memory word is held static until an operation on the memory bits is initiated so avoiding a word synchronization delay and (with some technologies) also saving power. The three clock signals for the memory cells (clkA, clkB and clkH) are controlled according to the value of an RS flip-flop (see figure 2.7). After *start operation*, clkA and clkB follow clk1 and clk2 respectively and clkH is low thus driving the shift register memory words; after *stop operation*, clkA and clkH are high and clkB is low to hold the data values in static latch configuration.

The timing of an operation is achieved by a special shift register driven by the same clocks as the memory cells. When the memory is static, this register is initialized to zero on all its cells; when the memory is rotated, a one is propagated through the register and becomes the *stop operation* signal when the word cycle is complete. The logic is self timing in that the signal causes the clock control RS flipflop to reset and the clocks to alter accordingly which thus initializes the special shift register and so removes the *stop operation* signal.

All input signals to the chip, including the data buses, are passed through D-flipflops clocked by clk1. Whatever the appropriate value of the internal signal, the input signal is designed to be significant on being raised to high. The *start operation* signals is designed to allow the longest possible time for the resetting of the clock control logic and the discharging of the clock lines which are held high in the static phase, before the remaining internal clock (clkB) rises. Thus the *stop operation* signal is generated on the falling edge of clk2 – figure 2.8 shows the logic design. As an input

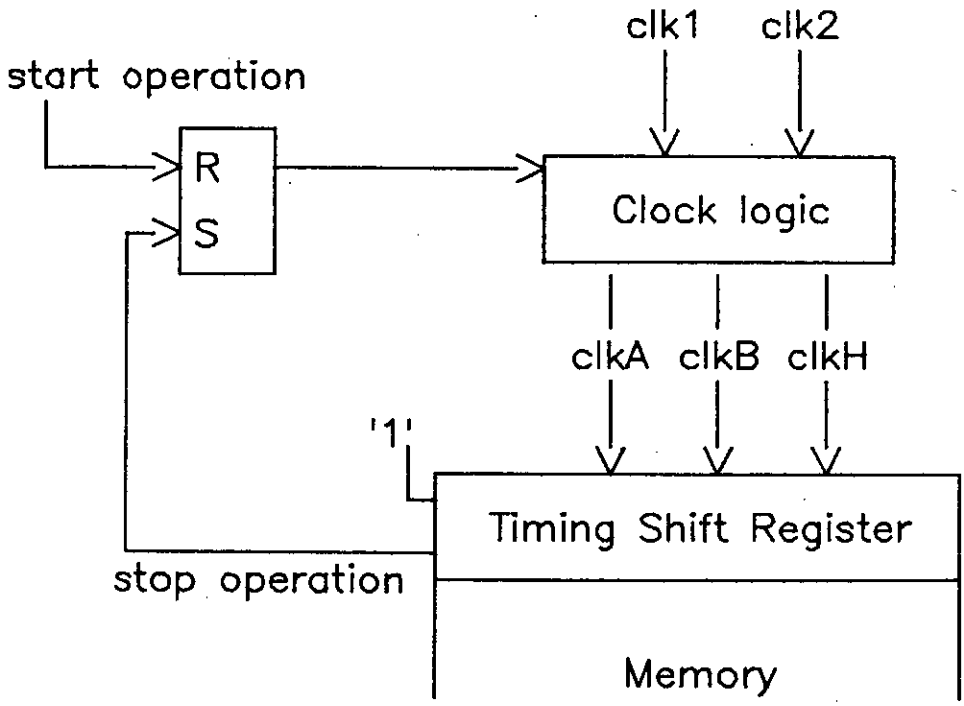

Figure 2.7: **the clock organization**

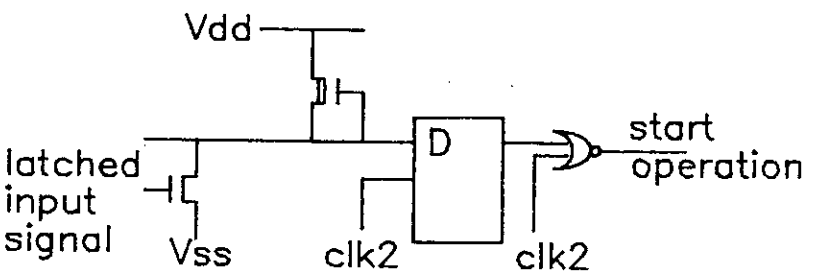

Figure 2.8: **the start operation signal**

signal is latched on clk1, the latch signal which is correspondingly raised is used to discharge a signal line whose value is subsequently clocked into a D-flipflop according to clk2. The value of this delay element can not be used directly as the *start operation* signal since this would cause clkB to begin to charge while clkA and clkH were still high; thus the signal is input, with clk2, into a NOR gate so that the resulting *start operation* signal is only raised when clk2 is lowered. This scheme implies that the memory rotation commences on the clock cycle following the initiating control signal.

The read and write operations are performed by switching the data buses into the addressed memory word through pass transistors activated by a control signal gated with the word address line. Figures 2.9 and 2.10 show the read and write interfaces respectively. The control signal is sent by lowering the control line which is combined with the word address line through a NOR gate, and so only the currently addressed word is affected. The signal must be held throughout the operation, and this is achieved by driving the internal read or write control signal from an RS-flipflop which is set at the beginning of the operation and reset by the *stop operation* signal. Recall that the transfer of data from one cell to the next occurs on the clk2/B signal. The input data signal thus has from the beginning of clk1 to (nearly) the ending of clkB to become fully established. The output signal is not clocked on the chip and therefore has a full clock period to establish itself on the output bus, starting at clkB or at the setting of the RS-flipflop after clkA on the first bit.

The comparison operation is initialized through the mask data input, by the setting of an internal RS-flipflop – subsequent inputs represent the mask which is gated with clkB to form the clock signal for the comparison latch reset logic. The RS-flipflop is again reset by the *stop operation* signal.

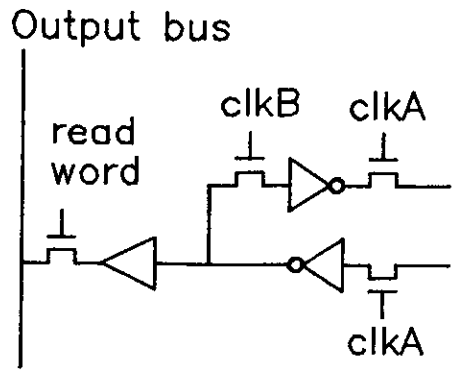

Figure 2.9: the read interface

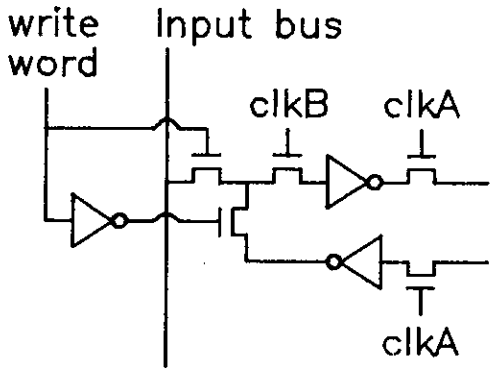

Figure 2.10: the write interface

The *defined* signal is taken from the *write* signal together with a non-cascading unit. Since the effect of the non-cascading unit occurs when the signal returns to high, the compliment of the internal write signal is used in making passive the empty tag of the currently addressed word. Thus the word remains addressed throughout the operation and becomes defined at the end, as is required.

There is also an *empty memory* signal which ensures that the RS–flipflops are initialized appropriately, by sending the *stop operation* signal, and that all the empty tags are active.

2.6 Resultant Structure

This completes the basic design of the minimal CAM component and the remainder of the chapter considers some practical enhancements which render it a viable computing component. The basic design leads to a simple but powerful configuration (see figure 2.11). On each word operation, all memory words rotate together. The details such as the number of tag groups and the memory word realization are independent from each other and so can be easily changed within the design.

At each word the current bit is boosted and so available to the output bus and the word specific comparison logic. The comparison operation is effectively a conditioned reset in a tag group and this occurs by sequentially providing the comparand on the comparison bus and the mask data on the selected tag group(s). The word address line spans the addressing logic of each word and may be lowered by any of the tag groups, to gate the word addressed operations. Input and Output are achieved by listening to or breaking into the rotating bits on the addressed word.

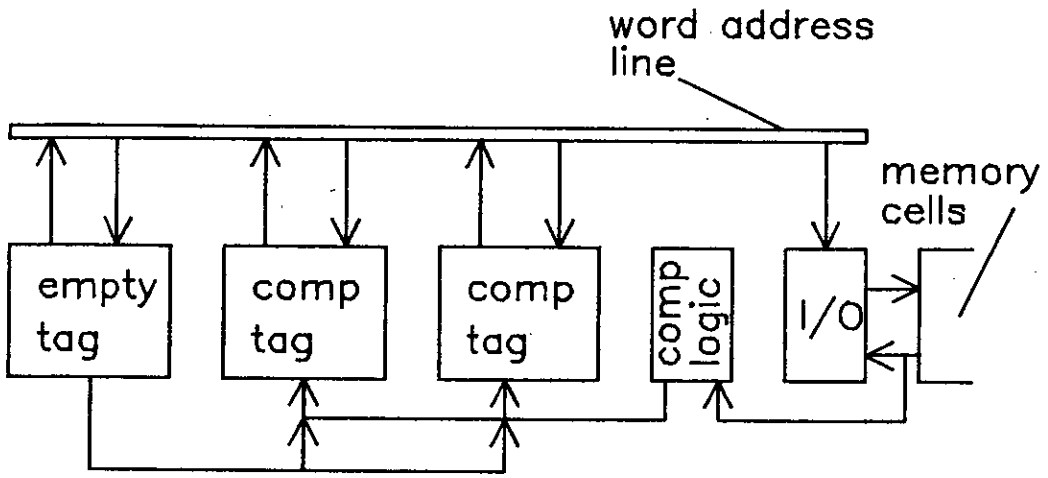

Figure 2.11: the basic word structure

2.7 Fault Tolerance

The size of an Intergrated Circuit is a compromise between the designers desire to fit as much logic as possible onto a contiguous area of silicon and the manufactures inability to prevent defects. The system designer seeks to minimize the delay in propagating signals, which is smaller within an IC than between them. On the other hand, the manufacturer is constrained by the yield of his product: a circuit is defective if it overlaps with a defective region on the wafer and the probability of this occurring increases exponentially with the size of the circuit.

2.7.1 Memory Defects

Defects in the wafer are of two major types: regional and point defects. The former results from processing effects where either the variation of the functional parameters has gone beyond the tolerance limits, or there are misalignments due to shrinkage or expansion of the wafer during fabrication. These global effects can only be minimized by either improved fabrication techniques or by more tolerant layout design rules.

Point defects are small localized faults which commonly are caused either by dust or other particles reaching the wafer or the lithography masks, or by isolated spikes or pinholes in the deposited films. These defects may be countered if the effected logic can be isolated from the operation of the IC. This is achieved in RAMs by incorporating redundancy[26]: extra word and bit lines are included into the design to replace lines which are found to be defective. The replacement of good for bad lines is achieved by blowing fuses in the addressing circuitry either by using lasers or by applying a high voltage across a high resistance. This post-manufacturing yield enhancement has made large RAM chips more feasible.

2.7.2 CAM Fault Tolerance

The possibility of including fault tolerance into CAM has often been mentioned in the literature, but little has been written concerning actual implementation. In RAM the addressing logic has to be changed so that the entire address space corresponds to functioning memory; in CAM the problem is more simply to eliminate defective words. This may be done by making the words unaddressable. In the same way that an active empty tag renders the comparison tags unaddressable by a hardwired reset, the signal from a further latch can be used to make the address tags passive.

The testing of a memory word is performed by checking that the word recalled is the same as the word stored. CAM already has the distributed logic for performing this test. If the same word is stored at every memory location and a comparison performed on the entire word, the comparison tag of each correctly recalled word will be active. Thus the value of the comparison latches may be used to set a latch which will eliminate defective words.

The fault recording logic at each word is shown in figure 2.12: the corresponding word is eliminated when the line y is high. The input to the D-flipflop is taken from the setting of the tag in one of the comparison groups: high if the tag is active, and low if the tag is passive implying that the comparison had failed. When x is high, the word is addressable as normal since the output y of the NOR gate is low. When x becomes low, the D-flipflop retains the value of its current input and y is subsequently held at that value's compliment.

The test pattern depends upon the realization of the memory. For shift registers it is sufficient to show that any sequence of bits with at least one 0 and one 1 can be stored and recalled. The test sequence is thus:

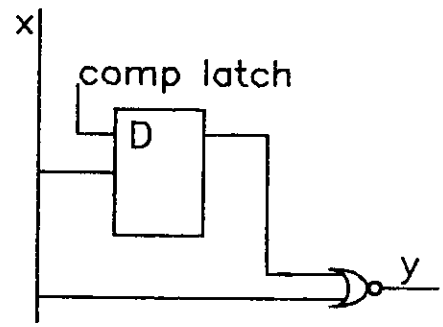

Figure 2.12: the fault tolerance logic

raise x line
empty memory
address empty tag group
while memory not full write pattern
perform comparison pattern
lower x line.

Only correctly functioning shift register words will be addressable so long as the x line is maintained low.

If the memory is realized using bit-sliced RAM cells, it is necessary to show that each cell can store and recall both 1 and 0. The test sequence is performed as above with an arbitrary pattern and then followed by:

empty memory
address empty tag group
while memory not full write compliment of pattern
perform comparison compliment of pattern
raise line x
lower line x.

Words which failed the second comparison will be removed as the D-flipflop is "clocked" by the x line. Words which failed the first part of the test will remain unaddressable since their comparison tags will necessarily be passive at the end of the second comparison.

This method of fault elimination has the advantage that it is effective against faults which develop after the component has been released by the manufacturer, it has the disadvantage that the test must be performed each time the component is powered up. However if the component is deemed to be fully functional, the test does not have to be performed when the component is used; an input may be tied to maintain the x line high and the component will operate as if there were no fault tolerance logic.

2.8 Look Ahead

The main limitation on operating speed is the time delay associated with the purely sequential nature of daisy-chain addressing. The signal is delayed both by the pass transistors and by the necessary booster gates. The system using the component is delayed for the time it takes for the signal to pass completely around the chain, which is proportional to the number of words. To allow for system growth, there must be a mechanism for avoiding some of this delay.

A simple circuit enhancement allows a lookahead signal which provides the output of the daisy-chain address signal without the inherent delay: it is high if and only if there are no active tags. The modification is to include a single metal line running parallel to the addressing line, which is pulled high by a depletion transistor unless pulled low by the output of one of the tags (see figure 2.13). Thus it can be determined if all the tags are passive without charging up the group addressing line.

This enhancement is very significant in the context of the functional operation of the CAM. Any addressed operation on memory must be performed by charging the address line and proceeding if, and only if, there is a word addressed. With lookahead, this decision may be taken without the addressing delay which, therefore, does not affect cases where there are no active tags (no words addressed by the comparison). In cases where there are active tags, the delay is reduced since the addressing may be initiated while the lookahead output is being interrogated by the controlling processor.

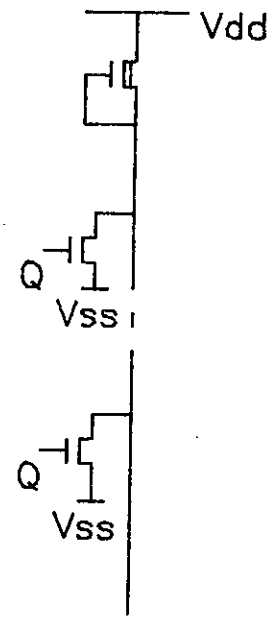

Figure 2.13: the look-ahead line

2.9 Expansion

The limitations on IC size necessitate that large CAMs are built from modules of small CAMs. This applies both to bread-boarding and to producing large ICs containing small units which may be isolated if defective.

2.9.1 Monolithic

The first approach is to combine several components in a series forming a single extended address bus by connecting the address outputs of each component to the corresponding inputs of the next. The address signal thus travels along the extended daisy-chain with the same organization as that of a single component, with more words than are practically possible due to the IC size constraints, but with additional off-chip delays.

The disadvantages of this scheme are that lookahead (and the associated delay savings) involves two extra ports and a ripple delay, and that the addressing delay is directly proportional to the number of words in the monolith which could seriously impede the performance of large memory systems.

2.9.2 Hierarchical

A better approach can be designed with a hierarchical address bus by using the lookahead output to bypass subtrees with no active tags. The CAM has three categories of signals:

- All but one type of input signals can be broadcast to the whole system. Those which affect all words can obviously be broadcast. Those which affect only the currently addressed word can also be broadcast since the signal is gated with the

word address line and the daisy-chain group addressing ensures that there is only one such word per group.

- The group address signals follow a sequential path through the memory words.
- The output from a memory word when it is read.

The approach is to broadcast all the signals for which it is possible, use lookahead to minimize the actual path of the daisy-chain address signal, and to use the address signal to switch the output of a memory word along a path of the same minimized length. The logic for the addressing and switching is shown in figure 2.14. When the addressing signal reaches x, it enters the module. If the look ahead signal is high (implying that there are no active tags), the signal continues through the pass transistor without delay; if the lookahead is low, the address signal is stopped and the address line beyond the pass transistor is pulled low. The output switch uses the fact that output can only occur if there is a currently addressed word within the module, which would imply that x is high and y is low; otherwise the module is isolated from the output line.

By including a lookahead line with this level of the addressing logic, it is possible to build modules of modules. Worst case delays are then a function of the sum of the path lengths for each level of the hierarchy, and not of the full size of memory.

2.9.3 Bread Boarding

The hierarchical approach may be applied to bread board assembly by designing a switching chip using the above logic. For each memory module attached to it, there are two pads for each group: address input and lookahead, a further pad for each comparison group: mask input, and one pad for the output signal. If any of these modules are unattached, the switch will continue to function correctly if the lookahead

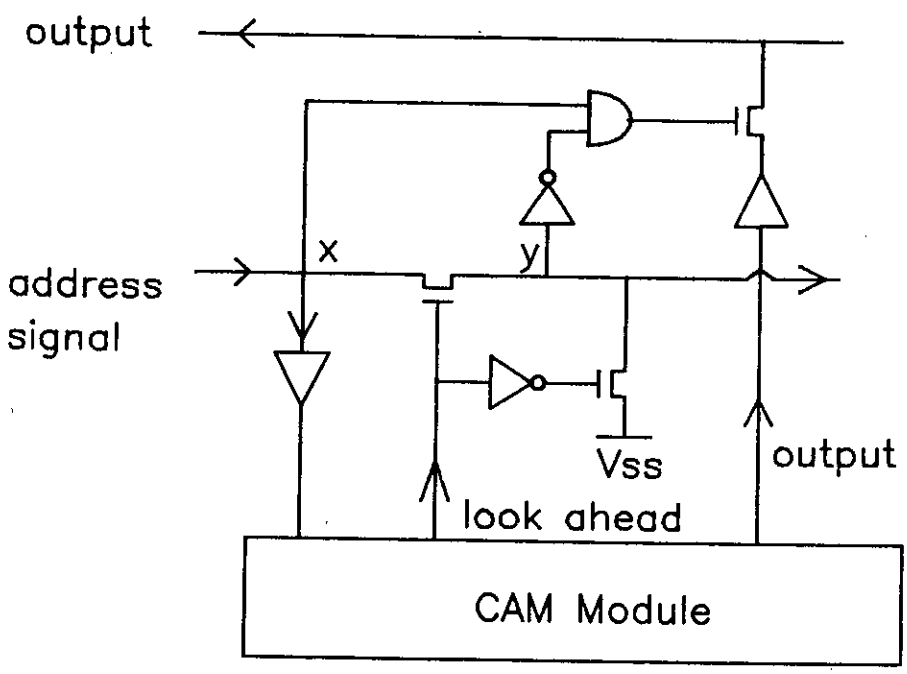

Figure 2.14: the switching logic

pad is tied high. The remaining pads on the switching chip form the address input, and data and lookahead output, of the resultant memory module which therefore has the same interface as its own submodules. Thus modules composed by a switching chip may themselves be connected into larger modules by the same switching chip design.

2.9.4 Wafer Scale Integration

Rather than bread boarding small packages, let us now consider the production of a large memory system on a contiguous wafer. The main advantages of wafer scale integration are the higher speeds, reliability and packing densities which are achieved by eliminating the need for packaging "chip size" components individually. The problem is that the defects arising in the manufacturing process limit the size of integrated circuits which could be produced with any reasonable yield. The solution is to design wafer scale components with small independent and testable sub-circuits which can be configured into a working system after testing. The configuration can be effected by either eliminating or establishing interconnections, by the use of lasers, fuses, further fabrication layers, or test and routing.

A wafer scale CAM has been proposed using test and routing[27]. This scheme routes a signal path incrementally through the good components by testing each potential addition to the path through the "known to be good" signal path that is already established. The disadvantage is that the signal path establishes a monolithic expansion of the CAM component and hence a long address delay – let us consider a different approach which will allow us to retain the hierarchical expansion.

Discretionary wiring dates from the late '60s. The technique is to test the components on a wafer and to use the results to define a further (unique) mask set for a final layer of metal to form appropriate interconnections between functioning units[28].

This scheme is now more attractive due to the development of E-beam direct write on wafers, and higher reliability in the fabrication of the final layers.

Content Addressable Memory in general is a suitable candidate for wafer scale integration because:

- There is no critical dimension in producing a working component. A general wafer scale system must have at least one correctly functioning circuit for each of its sub-units, a RAM must have sufficient words to fill the full address space; a CAM functions with whatever logic it can.

- The number of pins for the wafer is the same as for a single memory unit and does not follow the observed (so called "Rent's Law") relationship of increasing in proportion to the number of gates raised to the power of 0.6.

The CAM design of this thesis is particularly suitable because:

- Most signals are distributed on global buses.

- The expansion logic produces a modular hierarchy to distribute the address line and switch the output bus.

- The fault tolerance in the memory units allows the automatic isolation of defective words, and hence a larger practical size of sub-circuit.

- The modular design allows for isolation of a defective sub-circuit. This is achieved by tying the look ahead output of that module to high during the testing.

- The only critical area is the control centre.

The distribution network of signals (see figure 2.15) requires at least a process with two metal layers because of signal delays. Global signals are broadcast over the network, and the addressing and output signals are laid along the same paths. The actual decomposition of the modules depends upon the optimal numbers of modules to be included in each level of the hierarchy which in turn depends upon timing equations using the parameters of the chosen fabrication process. The control centre may be best placed at the actual centre of the wafer since this is the region least affected by regional processing defects.

The elimination of defective submodules requires the testing and possible modification of the fabricated wafer. The functioning of each independent unit must be verified, and the lookahead input to each switch connected by discretionary wiring either to the lookahead output of its subtree (if the subtree is correct) or to logical high (if defective). The salient features are that the proposed wafer consists of several units (the CAMs and the switches) which can operate independently, and that a defective unit may be isolated by tying the lookahead nodes of its communication bus to the power rail. If the units where designed with a probe pad on either side of a break in each connection to the control bus (see figure 2.16), an automatic tester could be used to verify each unit since they are each unconnected to the remainder of the circuits. If the unit functions correctly, the corresponding probe pads are connected by the further layer of metal; if the unit is defective, the lookahead inputs to the communications bus are connected to the power rail. The fabrication of this metal layer would require only a comparatively 'course' process, since it need only cross a purely substrate terrain between the two probe pads, and there is no significant advantage in using narrow tracks. A similar verification and isolation could be performed at each level of the switching network.

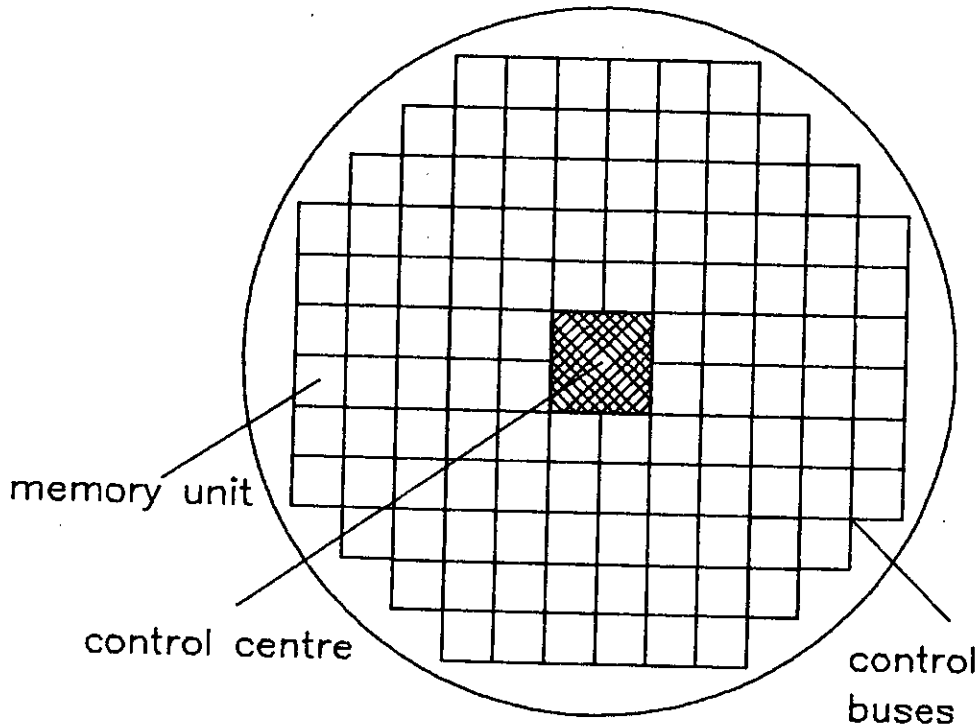

Figure 2.15: the signal distribution logic

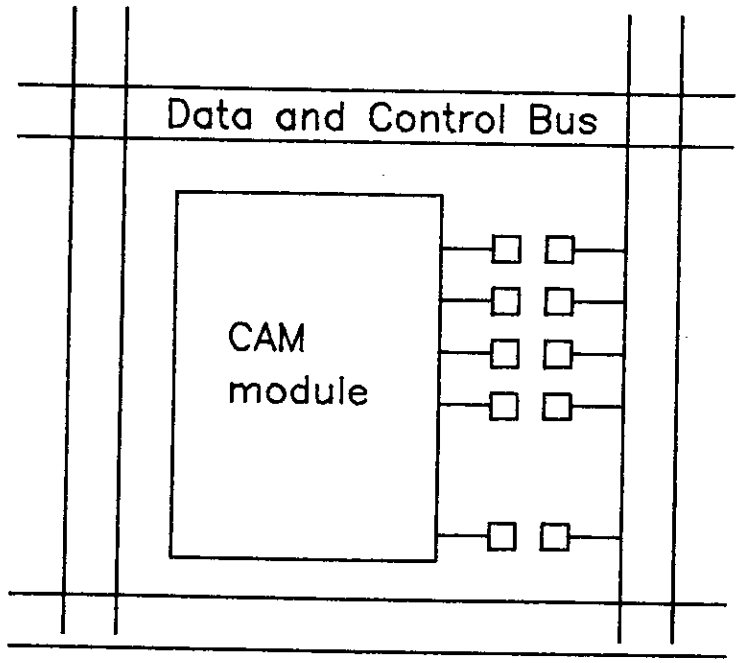

Figure 2.16: **a discretionary wiring scheme**

2.9.5 Silicon Wafer Packaging

While CAM is particularly suited, in comparison to other logic, to wafer scale integration there remain problems. The metal tracks of standard processes have a significant RC delay and power consumption when laid over such a network; the testing and commissioning stage requires the production of an extra, unique mask for each wafer or a unique programming for E-beam direct writing.

A middle, practical course is to produce the distribution network on an independent silicon wafer to which tested CAM units may be bonded. This allows the fabrication of the network using thick films for increased interconnection speed and low power consumption[29] without limiting the choice of technology for the memory units. The expansion logic is laid down with the network and the memory and control units are fabricated separately. Test structures may be included on the bonding wafer to check for regional defects at the bonding sites. The yield on the bonding wafer will be very high since there is very little active logic and the metal tracks have negligible sensitivity to point defects.

Chapter 3

PROTOTYPE REALIZATION

To validate the design, it is necessary to fabricate some components and to determine whether or not they perform the functions for which they were designed. A 1K bit component (16 x 64 bits: 4.53 x 3.56mm) of the basic design, and an 18 x 64 bit component (4.72 x 4.35mm) with lookahead and fault tolerance, were developed and fabricated on the Edinburgh 5μ poly-gate NMOS process. Plates 1 and 2 show the basic and enhanced components respectively, and plate 3 shows a portion of the basic component at the interface between three shift register words and their I/O and comparison tag logic. Twenty six basic components and six enhanced components were packaged and tested – this chapter details the methods used in the design and testing of these components, and the results obtained.

3.1 Computer Aided Design and Testing

This section details the design and testing strategies which have evolved during the course of the prototype CAM's development. It high-lights the features of the Computer Aided Design tools which assisted most in the chip development, and describes the software which was developed to demonstrate and test the fabricated components.

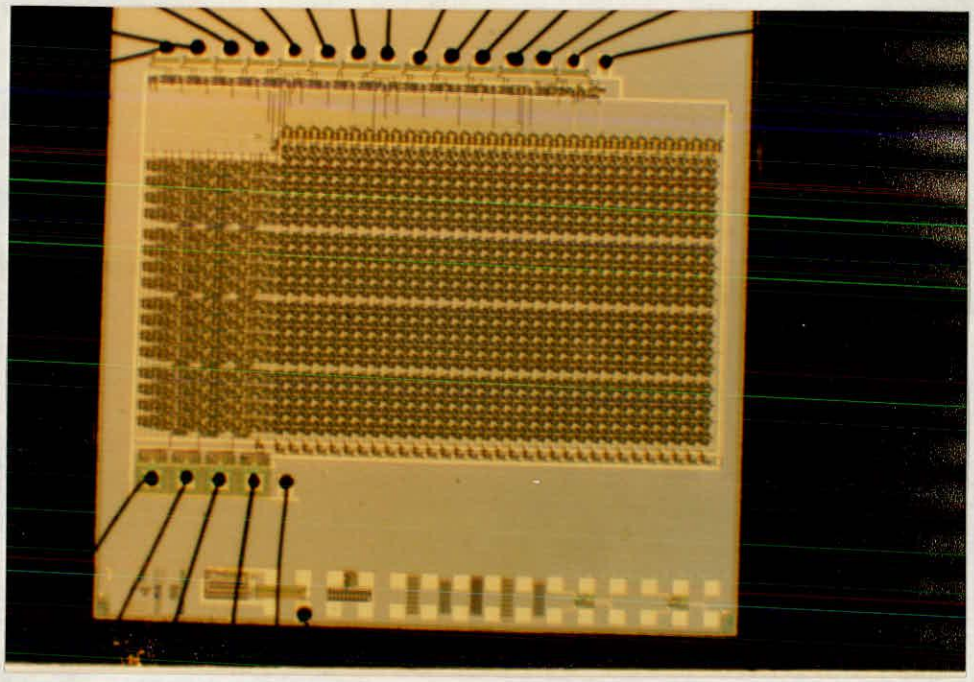

Plate 1: the basic component: 16 words of 64 bits, 4.53 x 3.65mm in a 5 x 5mm frame.

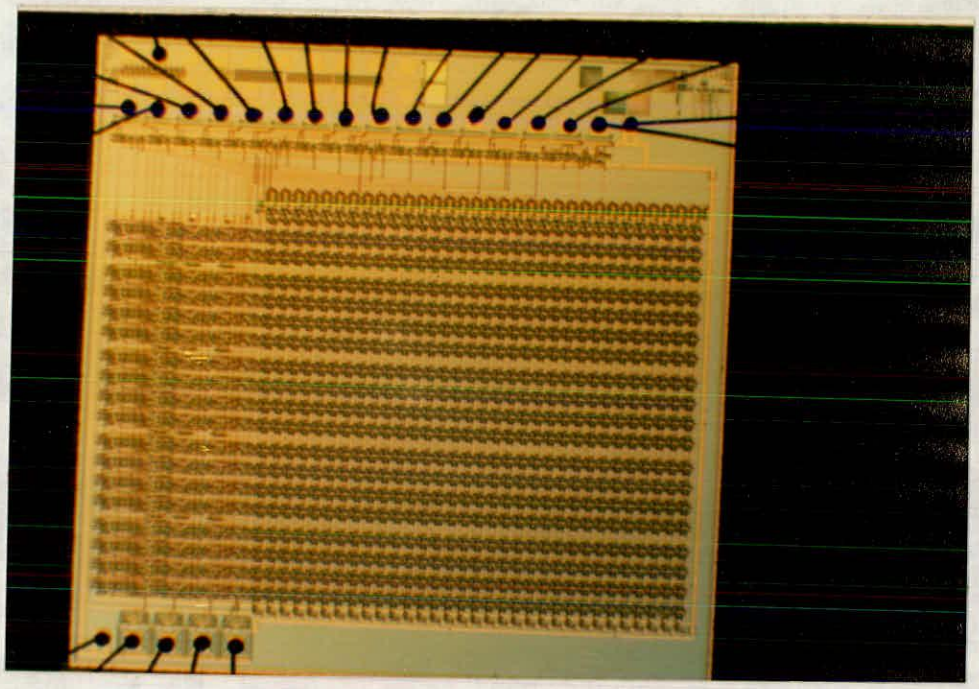

Plate 2: the enhanced component: 18 words of 64 bits, 4.72 x 4.35mm in a 5 x 5mm frame.

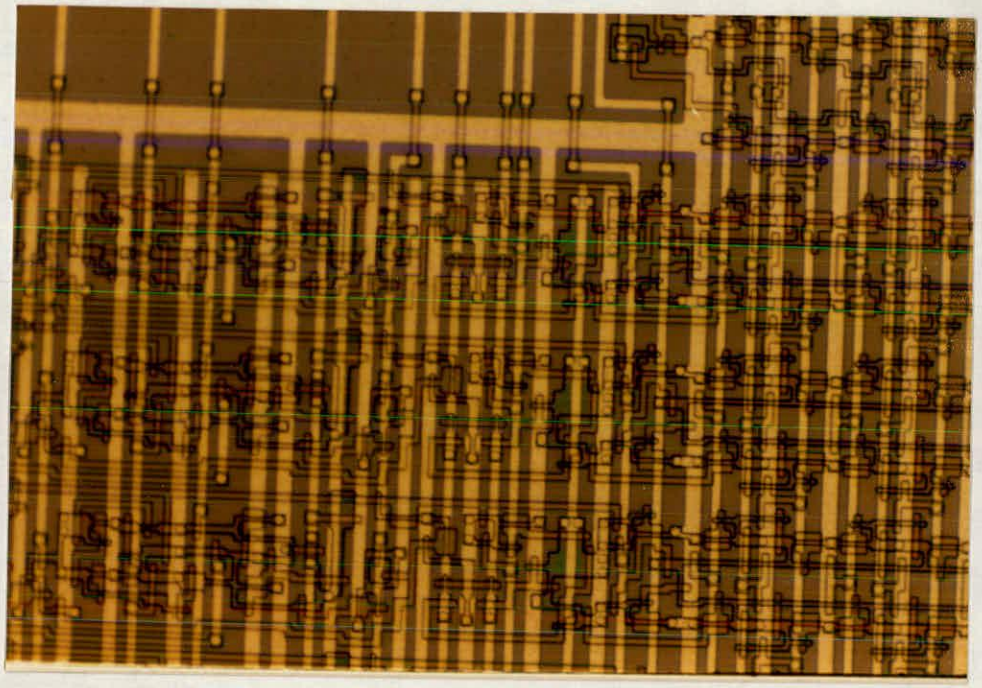

Plate 3: detail of basic component showing a portion of three horizantal words. In each, from left to right, can be seen a non-cascading unit, a latch, the addressing logic, the comparison logic, the I/O interface, and part of the shift register memory word.

3.1.1 Design Strategies

The Computer Aided Design tools used were due to the University of California, Berkeley[30] and enhanced by the consortium of the University of Washington and five Northwest companies (UW/NW VLSI Consortium)[31]. These include Caesar (a mask level VLSI layout system), Lyra (a layout rule checker), Mextra (circuit extractor), Esim (a switch level simulator), Pspice (a translator to spice format) and PLAP (a Pascal based layout programme). These were run on a SYSTIME 8750. The spice simulations were run on an HP 9000 series 540 computer, with colour graphics facilities, using the HPSPICE programme[32].

The design technique was to construct and test leaf cells which were combined into a full component using a PLAP programme. Each leaf cell was first designed in the Caesar editor, with the layout rules checked by Lyra and output in CIF. From this description, a circuit is extracted using Mextra and further converted to a spice file using Pspice. If the circuit does not contain floating (dynamic) nodes the switch level simulation allows for interactive verification, and for all circuits the spice simulation provides detailed circuit analysis including timing estimates. This environment lead to the easy assessment of different circuit and layout techniques, and also to cell design verification.

The use of PLAP to assemble a component allows for the easy modification of a large silicon layout: thus if a cell were altered, the full component could usually be regenerated with a single parameter change in the PLAP programme. Additionally, component parameters such as number of bits, and number of words, could similarly be altered. It was practical to run a spice simulation, in about 15 hours, on a full component with only one memory cell, and through the Pascal programme construction this served to validate the full sized memory. In the event, this procedure served to trap

wiring bugs in the Pascal programme, which the switch level simulator failed to do because of the dynamic nodes. The ease with which the component could be modified was particularly useful when incorporating the fault tolerance, which was achieved by modifying one cell at leaf level, changing a few programme parameters, and by correcting mis-alignments to adjacent cells directly by entering Caesar with the full, PLAP generated, component.

3.1.2 Test Strategies

Testing was performed with the aid of a Tektronix DAS 9100 and the 91DVV software package[33]. The DAS is a Digital Analysis System which allows for the generation of test patterns and the monitoring of the resultant output signals. It incorporates a console by which the test data may be entered by hand, but this is a lengthy and error prone occupation, especially for the testing of bit serial devices. Thus, the makers have released a software suite which translates a computer file into a binary instruction packet which can then be transmitted to the DAS. The format of this file (DasPat) is similarly cumbersome to generate by hand, but it can be generated by computer software.

The technique used was to test the chip functionally by generating patterns through C programme subroutines corresponding to the different operations of the component. Each subroutine produces the relevant test pattern in the DasPat format ready to be translated and transmitted to the DAS. At this stage, the simulation can be run and the output analysed on the DAS itself.

This approach was developed into a full "bread-board simulator" by making the subroutines accessible through an interactive interpretor and including a software analyser for the output patterns. By using a Lexical Analyser: Lex[34], and a "read-

command" loop, it is possible to input a command at the computer terminal, reply to questions concerning the values of the relevant parameters, and have any resulting output displayed. The Interpretor calls the relevant subroutine to generate the DasPat file, converts it to binary, transmits the binary file, sends the "start simulation" command, retrieves the resultant data file, extracts the relevant information and outputs it at the terminal. The information extracted is the condition of all the group output lines, and the value of the output data line on a *read*. The subroutines themselves use this information to trap "run time" errors such as trying to direct a word when none are addressed. The complete delay in response time (mostly due to I/O) is typically about 45 seconds.

3.2 Test criteria

Testing was performed by exercising the various functions of the component. The only output signals from the component are the three tag group address output lines and the data output line. Thus the testing of the component had to be performed from inferences gained from these.

3.2.1 Sub-criteria

For the memory to be deemed functionally correct, each word must be shown to perform its operations correctly. Each word consists of a shift register, I/O switches, comparison logic, two comparison tags and the empty tag. The sub-criteria are based upon the fact that if the sub-operations A and B are necessary in the performance of C, then the observation of C implies the occurrence of A and B. Thus to test the component, it is sufficient to test on each word the performance of a set of operations which depend upon (or *cover*) the performance of each of its functional parts.

If a (non-trivial) data word is written to a memory word and then recalled, then that shift register is verified since a break in the shift register would cause the output value to be stuck at one or zero. Also the I/O switches of that word must be functioning correctly.

If a tag can be used to address a word for any one of the addressed operations, then it is verified with respect to all of them since the switching of the control and data signals depends upon the tag's ability to lower the word address line which is independent of the specific operation.

For both of the comparison tags, both the overall comparison logic and the *next* function must be exercised. The comparison logic must be tested by ensuring that the tag remains active over unmasked bits and over a masked match, and is made passive following a masked mismatch or remains passive if the empty tag is active. For instance, if *ABC* is stored, the mask *010* and comparand *ABF* should succeed in addressing the word, and the comparand *ACC* with the same mask should fail. The effect of an active empty tag may be verified by addressing an "emptied" memory with a *null* mask: the comparison tag is then made passive only as a result of the corresponding empty tag being active.

The *next* function can be validated by testing whether the address signal has been advanced. This can be done either by counting the number of times the operation is performed after an unmasked comparison before the output address line is raised by the input address signal, or by writing distinct words to memory and following the location of the currently addressed word by reading its contents.

For the empty tag group, there are three functions (1) the initialization, (2) becoming passive after a write, (3) the addressed *remove* operation. By initializing the component, and being able to write 16 (and only 16) words to memory, both 1 and 2 are verified;

remove can be verified following a comparison in the same way as ***next***.

3.2.2 An example

The following is an example of a test sequence whose success would validate the basic component:

1 Empty Memory

2 Write the words (qwerty$x$, qweptyx: x = 1,8) – if the output signal from the empty tag address line rises after the 16th word has been written then the first two functions of the empty tag group have been validated.

3 Compare on one tag group using the comparand *qwerty0* and the mask *1011110* – the output address line should no longer follow the input address signal.

4 Address memory using that comparison group and *read* the addressed word; then perform *next*, *read* and repeat until the address output signal is raised: if the output data is (qwerty$x$: x = 1,8) then half the shift registers are validated, half that comparison group's tags have verified *next* and matching comparison logic, and the other half of the comparison tags have their mismatching comparison operation verified.

5 Repeat 3-4 with the comparand set to *qwepty0* and success implies the validation of the other halves of the comparison tags logic.

6 Repeat 2-5 with the other comparison tag group.

7 Perform a comparison using either tag group with a blank mask word so that all the words are addressed. Then use the *remove* command on each word in turn. If the output address line rises after the 16th operation, then *remove* has been verified for each memory word.

8 On both comparison groups, perform an unmasked comparison to verify that the (now active) empty tags prevent the comparison tags from becoming active.

3.2.3 Criteria for enhanced component

The enhanced component has two major differences from the basic design: (1) the output address lines are replaced by lookahead lines, (2) the fault tolerant latches have been included.

With fault tolerance switched off, the component can be tested in the same way as the basic component except that defective words do not constitute a fatal error. To test the fault tolerance it is merely necessary to show that a mismatch recorded on comparison group one will make the corresponding word unaddressable. This may be done by writing to the whole of memory and then ensuring that a mismatch occurs at each word; if the fault tolerance is then switched on, the whole of memory should be unaddressable even after an *empty memory* operation or an unmasked *comparison*. An individual component is verified if it can satisfy the criteria for a basic component after the test sequence (described in chapter 2) to eliminate defective words.

3.3 Actual testing

The test patterns were generated by the computer aided test environment described above. Some commonly used command sequences were made automatic. With the basic component, each input address line was raised at the end of each operation to determine whether any of the tags in their respective groups were active; this was unnecessary with the enhanced component because of the lookahead facility.

3.3.1 Initial elimination

In most cases the components were found to have fatal (probably regional) defects which prevented any word being recalled. To avoid the delay inherent in the Interpretor/DAS interface, a single test pattern was devised to determine whether the first word could store and recall a data word. The pattern relies upon the facts that:

- a read and a write operation may be performed simultaneously on the same currently addressed word,
- an *empty memory* operation does not affect the contents of the shift register memory words
- by making all the empty tags active, *empty memory ensures that the first word in memory will be currently addressed when using the empty tag group on the next addressed operation.*

The pattern performed an empty memory followed by a simultaneous *write* and *read* with a non-trivial data word. Once loaded into the DAS, the test pattern can be generated at the touch of a button. On the first occasion after power up, the data output line remains constant since the initial contents of the shift register cells all fall to the same value.

However, when the button is pressed a second time, the shift register is again addressed and should output the value of the previous write.

The basic component: With the basic component, failure of this test implies that at least the first word is defective (although the defect is probably more general). Since there is no fault tolerance, this alone is sufficient to reject the component. Out of the 26 basic components tested, 19 failed to output the test word. A further component was also rejected since the *out1* and *out2* signals failed to follow the corresponding input address signals after the *empty memory* operation.

The enhanced component: With the enhanced component, failure does not imply the rejection of the whole component since if the fault lies in the memory word itself, then the fault tolerance mechanism could remove it. However, of the 6 components, 5 failed this test, and were rejected after further testing showed that no words could be written and recalled on an unmasked comparison.

3.3.2 Testing the basic component

Of the remaining six basic components, only one was found to be good; for the remainder, the testing sought to establish the exact limitations of the device. The main technique was to write sixteen distinct words to memory, to address them using an unmasked comparison, and to read them out if possible.

Two basic errors could be found associated with any single word: either it output a constant value (implying either a fault in the I/O mechanism or, more likely, a break in the shift register), or it was unaddressable (implying a fault in either the comparison or in the tag logic). The actual results for four of the chips were: 12 unaddressable, 1 constant; 2 unaddressable, 1 constant; 2 constant; and 1 constant.

The errors in the remaining faulty chip lie in the address logic. There are two fatal errors: the output signal of the first comparison group rises to high, irrespective of its input, while the twelfth word is being written; and the address signal in the second comparison group can not progress beyond the thirteenth word after an unmasked comparison, presumably due to a defective non-cascading unit.

Although it would be possible to claim verification of the design from the observed behaviour of these defective components, it is more pleasing to be able to report the validity of the single remaining component. This was tested using the instruction sequence described above, and the results implied that the chip was indeed defect free and that it performs the desired functions.

3.3.3 Testing the enhanced component

The remaining enhanced component proved to be defective not only in the memory words (which is correctable), but also in the second address line which can not address any words beyond the first two. However, the main objective in the testing was to verify the features by which the component differed from the basic design. Hence it was sufficient to use the first address line to test whether the defective words could be eliminated.

In fact, the component had two defective words (the 3rd and 8th) which was ascertained by writing 18 distinct words to memory and recalling them with an unmasked comparison. A masked comparison was then performed which should have matched all the input words, and the fault tolerance invoked. The resulting configuration was tested as far as possible using the one address line and found to function correctly as a sixteen (18 – 2) word CAM – thus the fault tolerance logic had successfully eliminated the defective words. The lookahead circuitry was also seen to be correct

during this testing.

3.3.4 Timing and electrical results

The components are run with a 5V source and a −2.5V back bias; the current drawn was 0.06 amps implying a power dissipation of 0.3 watts.

The necessary clock period was found to be surprisingly large and considerable effort was spent in seeking the relevant factors. The signal distribution on chip is such that all paths consist of metal runs with short polysilicon branches, thus it had been hoped that the component would be reasonably fast.

To isolate one timing event for a comparison between the actual and the expected values, there was an investigation of the width of clk1 which was necessary to ensure that an input signal was latched by the corresponding D-flipflop. This was easily performed using the basic components by raising the address signals after an empty memory operation and observing whether or not a signal emerged on the output. Thus we are considering the clk1 input signal which is boosted by a super-buffer along a metal run to clock a D-flipflop. A Spice simulation using a full extraction of the flipflop and the super-buffer, with (double the) worst case estimate of the RC component for the intermediate path, revealed that an input clk1 pulse of 50ns was sufficient. Tests with the six basic components which passed the initial elimination produced the values (in nano-seconds) 80, 240, 280, 280, 400 and 560; that of the fully functional component being 280ns. Thus the typical component is about six times slower than pessimistic expectations. It was interesting to note that the address propagation delay was usually within 20ns of the minimum clk1 width; this implies that the delay in a 16 word address is no longer than the necessary input clock width.

Further experiments with the second chip producing the value 280ns were performed to establish the minimum clock widths necessary to initiate a full word rotation. The 'initial elimination' test pattern was used and the clock widths reduced independently until the output word deteriorated. The results were 640ns for clk1 and 680ns for clk2.

With these figures for the minimum widths of the pulses for the two phase clock, it was impossible to drive the component at the expected frequencies. Note, however, that if these values were reduced by a factor of three (half that suggested by the Spice above) then the component would be operating at a frequency of above 1MHz. As part of the manufacturing process, tests are performed by the Edinburgh Microfabrication Facility on a ring oscillator situated in a test strip on each component. Conversation with that department revealed that the results for the two wafers produced for the basic component were in fact considerably slower than usual.

3.4 DAS output

Figures 3.1–8 present a series of DAS displays which portray valid responses for the basic component; these are examples rather than a complete *proof* of the components correctness. The DAS display used here contains the twelve input lines and the four output lines. The labels correspond to operations on the CAM as explained elsewhere except that *zero* is another name for the *remove* operation. The plots are formatted as a single operation followed by raising the address lines to determine whether any tags are active in the corresponding groups.

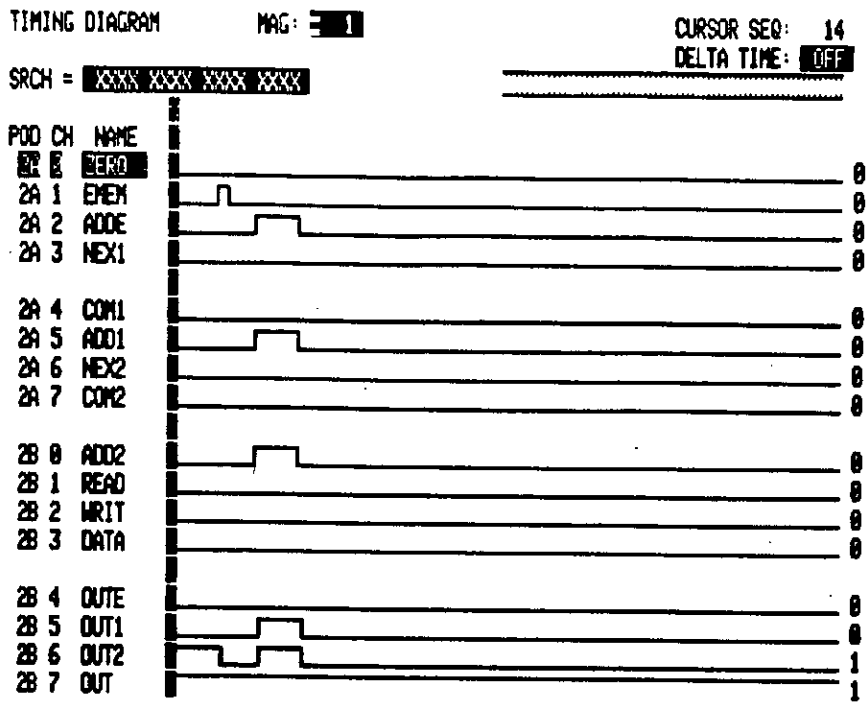

Figure 3.1: The *emem* signal initializes the chip by setting all of the tags in the empty tag group and by establishing the correct internal state. Note that the invalid *out2* setting is corrected.

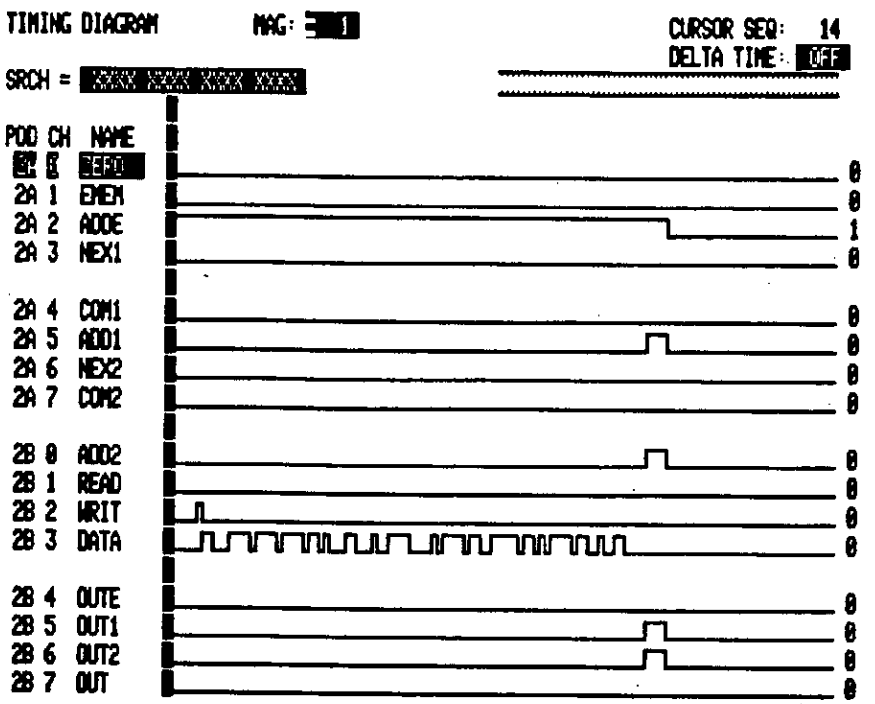

Figure 3.2: By addressing the empty tag group and initiating a *write* operation, a data word may be entered into memory.

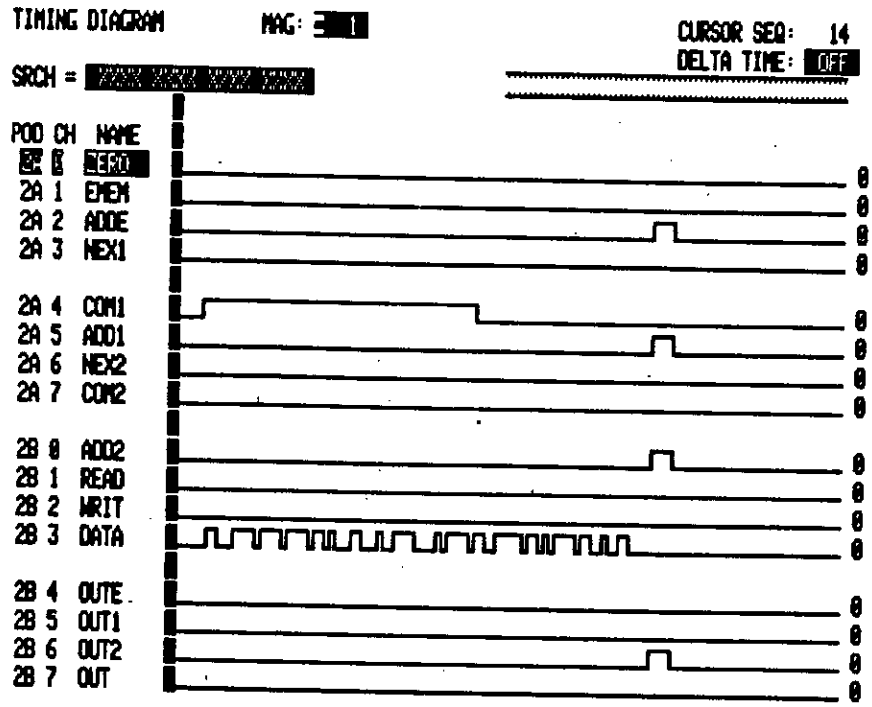

Figure 3.3: The same data work is used in a masked comparison, on tag group 1, and a successful matching is indicated by the fact that *out1* no longer follows *add1*.

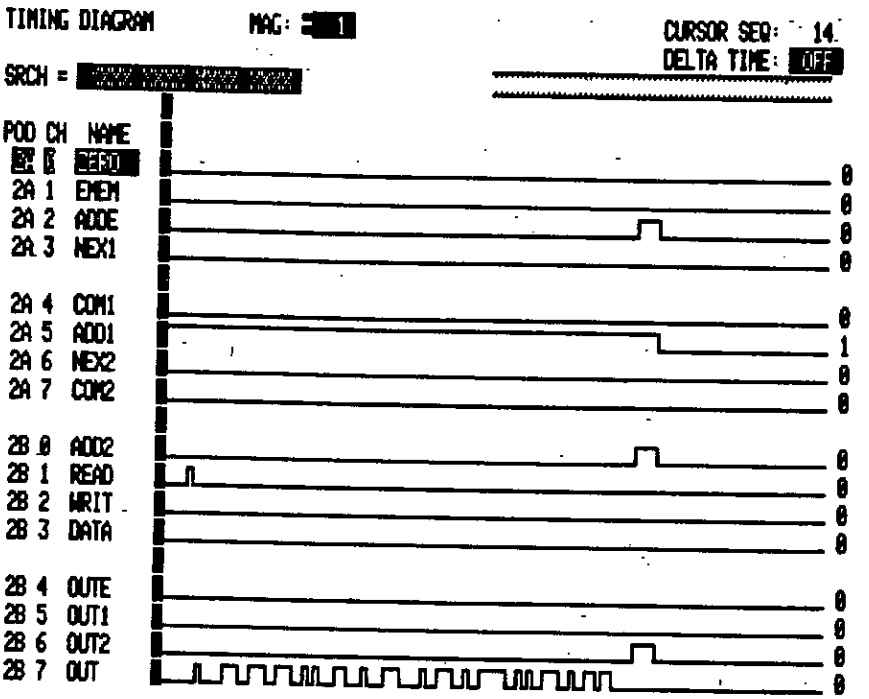

Figure 3.4: The matching word in memory is output by addressing comparison group 1 and initiating the *read* operation.

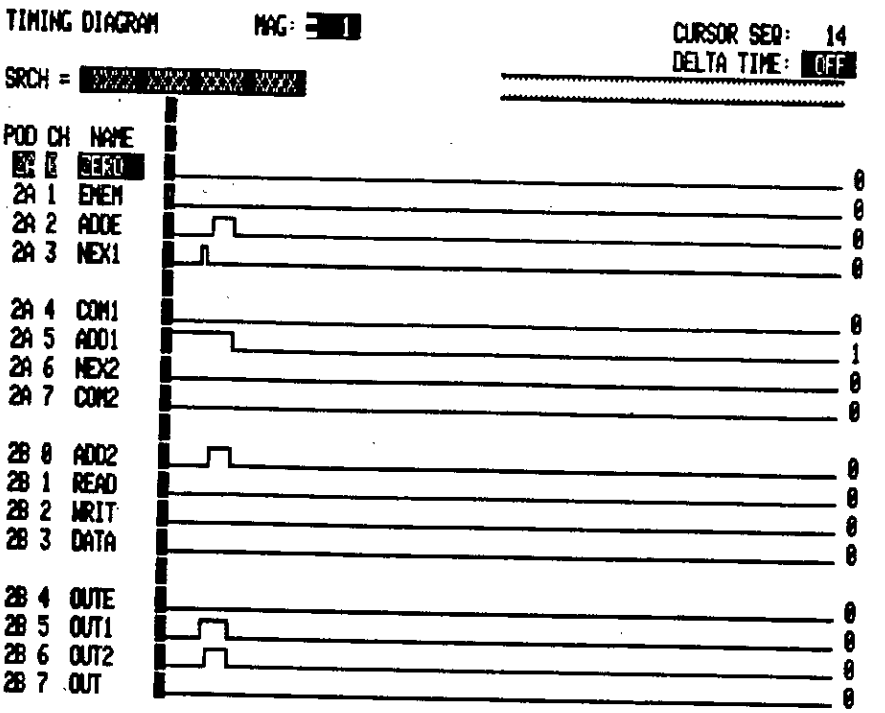

Figure 3.5: The comparison tag is reset by a *next* command ...

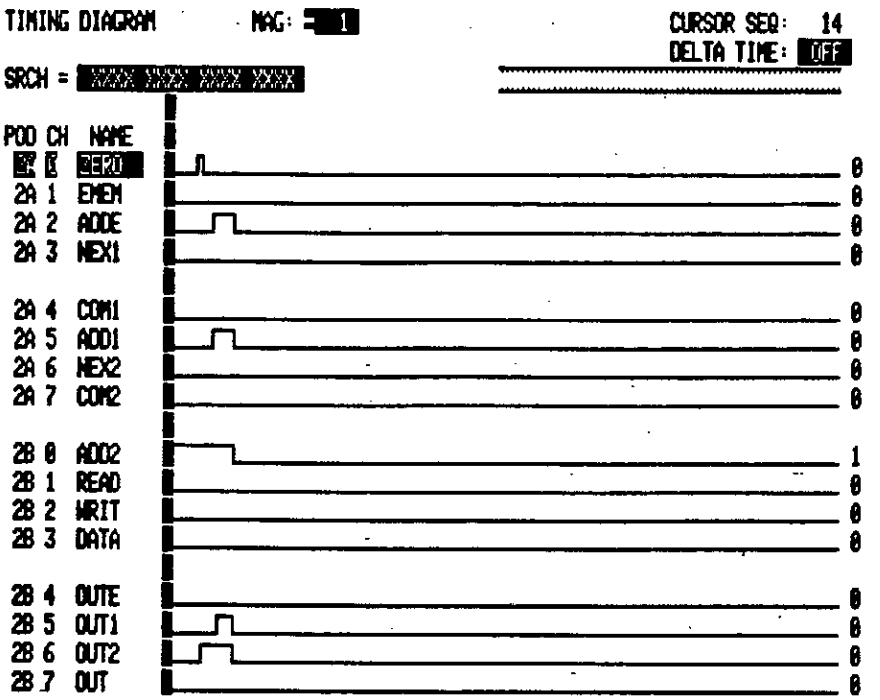

Figure 3.6: ... or by the *zero* command as shown here following a comparison with group 2.

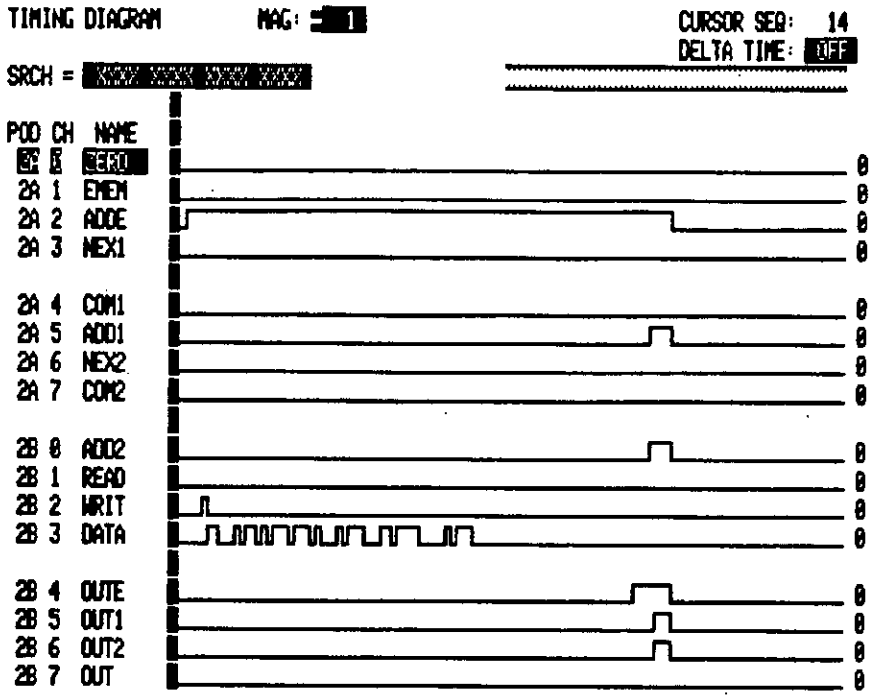

Figure 3.7: If the memory becomes full, the *oute* signal follows the *adde* input after the end of the *write* operation to the final free location.

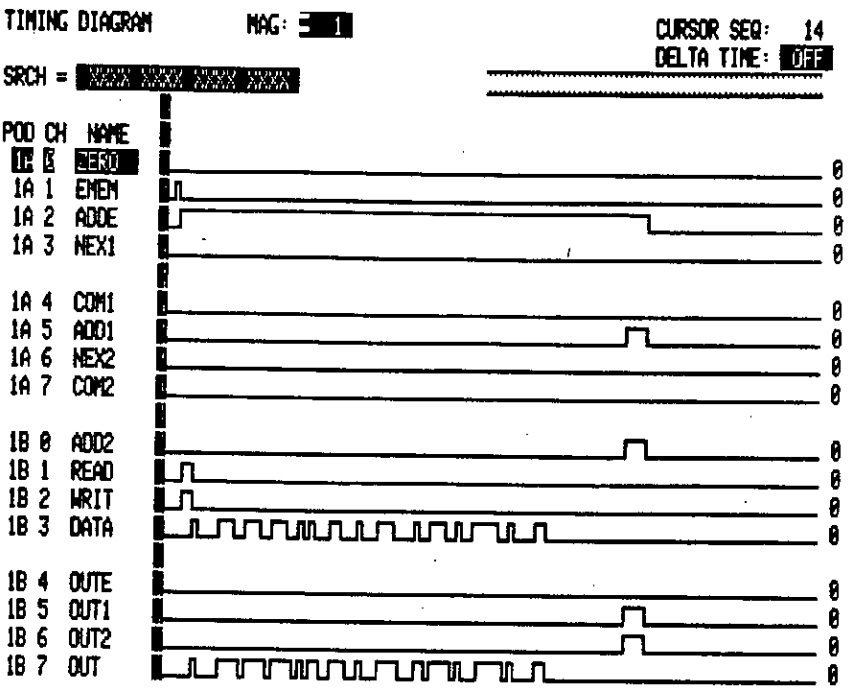

Figure 3.8: This display shows the signals associated with the initial elimination test pattern.

Chapter 4

COMPONENT CHARACTERISTICS

This chapter describes the characteristics of the CAM component and shows that they can be usefully developed both by software abstraction and by including the component in hardware systems.

4.1 CAM Hardware Operations

The characterization of the component is expressed in terms of the operations which it can perform. We consider the simple operations which correspond to the original abstraction specification, and then some useful effects which arose out of the actual design. Finally, we consider the component in terms of its limitations.

4.1.1 Basic Operations

Firstly, consider the component in terms of its pads. These consist of two data input pads: the memory word input, and the comparand; and eleven control pads: three address lines, read, write, two compare/mask lines, two next lines, remove, and empty memory. Their function is related to the storage and retrieval of words, and the direction of the addressing mechanism.

CAM operations can be described according to two dichotomies: word-length and immediate, addressed and non-addressed. Let n be the number of bits in a CAM word.

- Word-length operations involve the rotation of each memory word and so continue for n+1 clock cycles.

- Immediate operations are effective when a signal is applied for at least one clock cycle and then removed (this is the non-cascading unit described in chapter 2). The delay before a subsequent addressed operation depends upon the worst case delay of the addressing signal through the memory hierarchy.

- Addressed operations affect only the single word which is selected according to the tag setting in the addressed tag group. Normally only one tag group is addressed.

- Non-addressed operations affect every word in memory.

A word may be addressed either as a free memory location or as the result of a comparison operation. The set of free memory locations is maintained by the hardware; the whole of memory is placed into this set by an immediate operation: *empty_memory*.

Comparison is a non-addressed, word-length operation, initiated by raising one of the two comparison control lines (one for each comparison group) for one clock cycle. On each of the n-subsequent cycles the mask bit is entered on the same control line, and the corresponding comparand bit is entered on the comparand data line.

Words which match the masked comparison are addressed one at a time. The relevant group's address line is raised and the first matched word on the daisy-chain is addressed and will respond to any addressed operation. If either a *next* or a *remove* operation is performed, that word is no longer addressed and the next matched word on the daisy-chain becomes addressed. There is no possibility of returning to the first word without performing the comparison again – each matching word may be addressed, manipulated, and then passed by. *Next* is an addressed, immediate operation which

allows the next word in the daisy-chain to be addressed; *remove* has the additional effect of returning the currently addressed word to the group of free memory locations.

When all words in a group have been passed by, a signal is output. Thus the result of the operation "some-or-none matches to a comparison" is available directly after the comparison is performed. The number (x) of matches may be simply obtained by repeated *next* operations while incrementing a counter, in O(x) time. The corresponding signal from the empty group indicates when the CAM is full.

Both *read* and *write* are addressed, word-length operations. The whole word is read or written – this implies that a partial update of an addressed word must generally be implemented as: the word is completely read into a separate register, the relevant section altered, and the whole register is then written back into memory. This detail should certainly be hidden from the high level language programmer.

4.1.2 Developed Operations

This section examines some 'clever' effects which rely on particular features in the final design. Firstly, the data input, comparand input and data output lines are all independent which allows the combinations of some word-length operations. Secondly, once a word-length operation is started, it will continue for the n clock cycles even if other operations start during that time, thus a second operation may be interleaved with the latter portion of a word-length operation – but both operations terminate at the end of the first.

A *Unique Write* function is possible and potentially useful. If a write operation is performed to a free location and a comparison is performed with the same data, any matching words may be eliminated by repeated remove operations until the group's output signal specifies no further match. This is possible since the memory location to

which the new word is being written does not take part in the comparison. In general this function imposes an undetermined delay following each unique write operation since it depends upon the number of such words already in memory. However, if all write operations are performed uniquely this overhead will be at most one clock cycle. Uniqueness may be defined according to any subset of bits by selection of the mask values.

A read and a write operation on the same memory word may be combined. This function depends upon the shift register implementation of memory and the independence of the internal input and output buses. If a write is initiated during a read operation, the latter portion of the word is over-written. If the write is initiated at the beginning of the word cycle, the written word effectively replaces the read word in memory. If the computation involves alternating between destructive read and write operations, this technique would halve the time spent in performing these data transfers. One potential bug is that a word waiting to be written to memory would not take part in a comparison performed before the next read; this error could be avoided by either flushing the buffer into memory before a comparison, or by including a single word hardware comparator into the system design. This function could be readily used to advantage by a compiler optimizing code which iterates through the words addressed by a comparison.

Similarly a comparison may be combined with either a read or a write operation. With a read operation the word must be addressed by the tag group not involved in the comparison; and with the write operation, the written word does not take part in the comparison. Two comparisons may also occur during the same word cycle with the same comparand but possibly different masks.

In is also possible to remove from a comparison group those words which form part of the other comparison group by addressing through the second whilst applying *next* to the first. This would implement a type of subtraction operation. However, this function is verging on the idiosyncratic and so would be unsuitable for representation in a high level language.

4.1.3 Limitations

This thesis is concerned with the characteristics of the device and the development of a programming environment which affords the user direct access to the advantages of the underlying hardware. Thus we ignore software emulation of a virtual CAM with enhanced features so as to concentrate on the direct matching of the hardware to high level languages. This does not preclude the realization of other software abstractions through compiler action but we will consider only the actual CAM and its characteristics.

A system limitation is the number of words in its CAM. If the memory becomes full, there is no trivial software solution. This is reminiscent of the problems of RAM memory before the introduction of the virtual address space and this problem has recently re-emerged in some modern micros. It is not peculiar to CAM, but RAM users have become used to ignoring it. The swapping schemes of RAM virtual memory systems have no counterpart in CAM because they rely implicitly on *contextual adjacency* in RAM addressing. This is the observed phenomenon that in RAM successive address requests commonly correspond to physically close memory locations – this allows the block transfer of data with reasonable certainty that subsequent addresses will also be resolvable within that block (this assumption is actually becoming less valid in the "newer" programming languages); in CAM this assumption is groundless. On each address the whole of a virtual memory space must be searched; no

indication of the physical location of the next address can be found in the current address; the CAM is limited to its hardware size.

There are two parameters which are fixed at design time: the number of tag groups, and the length of the memory word. The number of tag groups is a design decision which must be based upon expected system usage. If the user can not solve the problem in the number of groups provided then a new component must be obtained. A limited solution is to provide a simple read of a group by placing the relevant fields on a software stack so that the comparison group may be used again when this information is processed.

The length of the memory word can not be suitably increased by software. There are many schemes for this, which essentially involve the increase in word size by extensions to multiple words associated by a common identifier and/or an extension counter. However, this involves complicated software support which does not emulate the advantages of the hardware word size. Let us consider an example: let the address field be in word *extension_1* and the required data be in word *extension_2*. Upon comparison, all the address words are *extension_1* words which have no physical relationship to the other extensions of the same virtual word. To access the required data, each addressed word must be read in turn and the identifier used to address each required word individually. This requires unique virtual word identifiers, indirection in addressing, and an address comparison per word. The difficulties increase if the address fields span two word extensions since the addressing requires the union of the responding identifiers of two separate comparisons. The virtual word is cumbersome.

The lack of order in the CAM model is not reflected in the thesis design because of the daisy-chain address path. This defines an order of addressing which may be exploited by low level programming. For instance, if data is stored in successive

operations, then it will be retrieved in the same order. If these entries are unmoved and only updated *in situ*, then the order is unchanged. However, the full operation set of the CAM limit the assumptions which can be made about this order. A new entry may not be added since a memory location before the other entries may have been freed (by the remove command) in which case the new entry will become the first rather than the last. In effect, the daisy-chain order may be safely used only in limited circumstances.

Finally, there is a logical difficulty in the CAM design. The tag groups are not independent in that it is possible to address the same word according to a different comparison and since it is possible to update (or indeed remove) an addressed location, the outcome of the computation may depend upon which tag group is used to address the common location first. This produces an element of non-determinism, and potential side-effects, which may undermind some attempts at programme proof theory and which may present pitfalls for the programmer. These are no more serious than the side-effect problems encountered in conventional programming, but it does make multi–tag CAMs less *elegant* than might have been hoped.

4.2 A Software Development

We now consider an example of a software abstraction realized by the selective manipulation of the comparand and mask values, using techniques first described at length by Falkoff[22].

The algorithms apply to cases where an order relation is defined on a memory field by interpreting its bits as a non-negative binary number. The idea is to use the comparand and mask values to progressively address smaller subsets of the numbers in memory by partitioning the binary tree inherent in their representation. An alternative perspective is that of tree walking on the binary tree with the branch chosen as the result

of a comparison operation to test for the existence of members in a sub-tree. The words can be accessed according to maximum, minimum, $\geq$, $\leq$, $<$, $>$, nearest value less/greater than, and as an ordered sort.

As an example, consider the following algorithm which produces all words in the CAM greater than a reference value M (say 110101).

1 Working from the most to the least significant bit of M, consider the first 0 to be the 'current' bit (that is 11C101).

2 The bits to the right of the current bit are masked off, the 'current' bit in the comparand set to 1 and the comparison is performed (on 111xxx)

3 Read out all responding words.

4 If there are no more zeros after the 'current' bit, terminate the algorithm; else, reset the 'current' bit to 0 in the comparand, consider the next zero to be the current bit and return to 2 (that is 1101C1 on second iteration).

The timing is thus one word cycle and one immediate operation to access each required word, and an overhead of α comparison operations where α is equal to the number of zeros in M (which is no greater than the number of bits needed to represent M).

To describe how an ordered sort is achieved on CAM, we will describe two sub-algorithms in terms of their tree walking paths.

[A] Mask out trailing zeros in comparand and change the least significant 1 to a 0.

Repeated application of this will address progressively larger subtrees, whose members are all less than the previous iteration, and therefore less than the initial comparand.

[B] Unmask the most significant masked position and set the corresponding comparand bit to 1. Perform comparison. If the addressed subtree is empty, set the comparand bit to 0. Repeat until the last bit in the word is finally set.

Given a non-empty subtree (defined by the value of the most significant bits), [B] finds the maximum by a decision on each of the lower branch nodes where, at each stage, the branch containing the larger values is tried and followed if it is non-empty. If it is empty, the other branch must be non-empty and so it is followed.

The sort algorithm determines values by repeated application of [B] to the first non-empty subtree found by [A] applied to the previous value, starting with the maximum possible. If there are N words of m bits in memory, then the sort can be performed in O(Nm) since each value may be found in no more steps than walking from a tree leaf, to the root, and back to another leaf, and there are N such walks.

The best known associative sort technique requires more complicated hardware[35] but is worth explaining. The enhancement requires immediate knowledge of the common value or disagreement of each bit column in the set of responding words. With this information, each search can partition the current set of words into two disjoint non-empty subsets by dividing it according to the most significant bit column where there is a disagreement. Thus exactly (2N – 1) searches are required to address each word separately and in order. The additional hardware prevents redundant searches, and removes the timing dependency on the number of bits.

4.3 System Design

A large CAM, available to system designers, would have the potential to affect and effect innovative architectural and linguistic design. In this section we consider a few examples of how the CAM component may be utilized in specific architectures. We deal here with applications for CAMs in general, and in some cases the nitty-gritty of applying the component of this thesis in particular.

4.3.1 Direct Execution Architecture

One proposal for the direct implementation of high level languages is the Direct Execution Architecture[36]. This is essentially a hardware interpretor which uses two CAMs as fast lookup tables: one to translate symbolic names into RAM location numbers, and the other to track jump entry points for programme flow: the CAMs provide the data structure support for what is essentially runtime compilation. The reported advantages are: interactive programming and debugging, architectural definition of programming language, and the measurement of language complexity.

4.3.2 Relational Data Bases

The use of a theoretical CAM has been considered at length as a basis for a relational data base machine[37]. This was a direct implementation of a Relational data base with the limitations of a fixed word-length, and the implementation of the function *join* only according to equality rather than the full set of relational operators. The conclusions were that "the architecture permits an O(log n) decrease in time complexity over the best known evaluation methods on a conventional computer system".

4.3.3 Multi-user Access

Can the CAM serve as a multi-user memory store? As a directly addressable component the answer is "no", since there must be some synchronization of the processors with respect to the word cycle, and multiple I/O, even with dedicated tag groups, would produce contention on the I/O buses. These problems are unlike those encountered with multi-user RAMs because such systems generally incorporate a memory hierarchy with local processor caches, and thus the difficulty is to maintain the integrity of the multiple copies from a single memory address.

However, if CAM access requests were processed by a dedicated memory handler, the concurrent operations of the CAM may be used to good effect. The efficiency of such a scheme would depend upon the number of processors and the frequency of memory requests: specifically if the rate of requests exceeds the rate of the handler's mean response time, the processors will become queue bound and the CAM will be the limiting factor in the system's performance. Thus the multi-user CAM could not serve as the main store for many processors. However it could be used as a system's global memory where the processors access this component at a rate lower than the critical value (although the propensity of the system for glitches in global memory access would have to be carefully assessed).

4.3.4 New Programming Languages

The disdain felt towards conventional programming languages is exemplified by the words of John Backus who has described them as "obese", and "fat and flabby"[38]. The major criticism is the lack of mathematics or logic in either their design, or their usage. Thus, much current research seeks to develop new programming languages upon secure mathematical systems, hoping to provide ease of language specification, clarity of

programming, and programme verification. The major problem with these languages is the difficulty in their implementation on conventional computers; but it is only through the provision of viable and efficient programming environments that new, well founded, programming languages will be accepted.

One such system is that of "axiomatic set theory" – whose elegance stems from the ease of its total development from a small number of formal primatives[39]. A restricted form of sets (sets of ordered pairs) leads to relational theory[40]. A *relation* is defined as a set of pairs interpreted as the mapping from the domain element to the range element. In CAM this could be implemented in the data structure:

[relation_id | domain_element | range_element]

and the computation is driven by matching the first two fields to derive the third. Such a language, for instance, is projected for the reduction machine Alice[41,42]: the Applicative Language Idealized Computing Engine, designed around the INMOS Transputer.

Let us consider the use of the thesis CAM in this system. The idea is to replace the transputer maintained relational language data base directly by a CAM. The computation is directed by matching function and argument fields, and reading the associated result. Each CAM is dedicated to a single processor which executes one packet at a time. In deterministic computation there should be only one match, which implies that input to memory should be conducted through the *unique write* form, possibly with an error signal emanating if a match is found. If the data is organized as:

[comparison_field | result_field]

the comparison and read of the records may be performed in one word cycle by initiating the read operation at the penultimate bit of the comparison_field during the comparison operation. There is some fine tuning of this scheme with regard to the

possible address propagation delay at the completion of the comparison – but it is a possible optimization. The significant feature is that the CAM represents the data in the same form as the mathematical specification without the need for any other supporting data structures or operations.

4.3.5 Token Matching in Data Flow

Data flow is a technique for decomposing a computation into its individual operations and the paths by which data flows between them. An operation can only be executed when its arguments have all been computed and assembled into an executable package. In one example, the Manchester Prototype Data Flow machine[43,44], the arguments destined for common operation nodes are matched through hash table search implemented concurrently on 16 breadboards with microprocessor control of auxiliary memory for overflow on collisions. A large CAM would perform the matching; the data flow system could be supported by a store for 10-100K argument packets[45] – which is a plausible outcome of the proposals in chapter 2.

With the Manchester machine, there is a dedicated queue handling processor which seeks a match according to a packet's destination field over the packets already in memory. If a match is found, the associated argument must be read; if not, the packet is itself written into memory. An extension to the scheme used with Alice is impossible because of this conditional write. However, the same saving may be obtained by interleaving the comparison with the read or write of the previous packet using a different tag group.

Let us follow this technique through in detail. Assume that the data is arranged in memory in the same format as in Alice above, and consider the progress of one packet through the handler when comparison line one is available.

1 The comparison, masked over the destination field, is performed on line one.

2 After the destination field has been input and compared (and while the word cycle continues), tag group one is addressed to ascertain whether there is a match.

3 If there is no match, the empty tag group is addressed and the packet write is initiated at the beginning of the next word cycle.

4 If there is a match, the read operation is initiated during the next word cycle with group one addressed, and the output becomes significant after the destination field of the next packet has been input. On the last bit of the word cycle the remove signal is sent – which acts at the beginning of the following clock cycle.

The actions 1 and 2 can be interleaved with either 3 or 4 on the other address line, and so allowing the concurrent processing of two, staggered, packets. As explained before, a packet waiting to be written must be compared externally with the next packet. If a match is detected then the write should be prevented or corrected.

4.4 Associons

Let us consider in detail one example of a well founded programming language to exemplify the influence a CAM computing system could have on the future of programming languages. Associons is a programming notation with tuples instead of variables developed by Martin Rem[46,47,48], with an emphasis on concurrent application. The memory was assumed to be content-addressable, all entities are considered to be *names*, and each word denotes a *relationship* between names:

"associons are ordered n-tuples of names".

An n-tuple is written in the form: (a, x, y), and the same format with square brackets denotes a presence condition for a corresponding associon. It is assumed that no more than one instance of any associon exists in memory. A programme progresses by constructing new associons according to the presence or absence of others. These are specified by presence conditions, punctuated with logical operators ('and', 'or', and 'not'), and a list of unknowns, thus:

x, y: [a, x, y] :=> (r, x, y)

means "for all x and y such that there exists an associon (a, x, y), create the (target) associon (r, x, y)". Each unknown must appear in at least one presence condition in each term to ensure that the number of target associons generated is finite. This format is referred to as a "closure" statement. The execution of a closure statement continues until the target associon is formed for all possible variables satisfying the presence conditions. A closure statement is said to be "cascading" if the target associon will match one of the positive presence conditions, thus:

x, y, z: [r, x, y] & [r, y, z] :=> (r, x, z)

It turns out that if the target associon matches a negative presence condition, then the computation is non-determinate – therefore such closure statements are disallowed.

The language contains the concept of local associons, where the format is described at the beginning of a programme block. As an example of the expressive power of this notation, consider the following programme:

```
local (r, ?, ?)
  x, y: [a, x, y] :=> (r, x, y)
  x, y, z: [r, x, y] & [r, y, z] :=> (r, x, y)
  x: [r, x, x] :=> (c)
end_local
```

which determines whether or not a graph is cyclic, with its edges specified by the set (a, tail_node, head_node).

The closure statement is then the basic statement which changes the sets of associons, and so also the "state" of a computation. It is based upon a mathematical formulation of "closure" whose definition is the smallest (provably unique and finite) set of associons which satisfy certain conditions relating it to the original set of associons with respect to a condition-generator pair. The closure statement is well founded in mathematics, as are computations whose state is directed exclusively by such statements.

The formulation of associons and the closure statement has two important properties:

- any closure statement can be translated (mechanically) into a set of simple closure statements whose conditions are each the conjunction of two presence conditions,

- any programme may be rewritten (mechanically) into an equivalent programme with fixed (> 2) length associons.

Thus a CAM which implements this reduced form will be able to implement associon programmes.

Pre-empting the penultimate chapter slightly, let us consider how the thesis CAM design might implement Associons. It turns out that for an efficient implementation of the closure statement, a third tag group is required. This does not invalidate the design

decision to include only two groups in the prototype, but rather demonstrates the need to match the exact hardware specification to the system requirements. In this case the closure statement:

x, y, z: [r, x, y] & [r, y, z] :=> (s, x, z)

can be performed by the CAM operations:

```
declare name variables x, y, z;
foreach word which matches ('r', =x, =y) using group 1
  foreach word which matches ('r', y, =z) using group 2
    perform a unique write of ('s', x, z) using group 3;
```

where the "=" sign implies assignment of that field to a local variable. For a cascading closure statement, this code block is repeated until it is executed without making an entry into memory.

Thus the closure statement may be implemented with a CAM. Associons demonstrate that the CAM may be the basis for language design which differs radically from the conventional von Neumann style.

4.5 Assessment

The examples in this chapter have demonstrated that the thesis CAM is a useful and relevant component in computer architecture and language developments, and that CAM in general allows for considerable simplification both of hardware systems and of software data structures. The breadth of examples is more significant than any one taken alone: hardware interpretor, support for multiprocessing, support for software applications, implementation of new programming methodologies – a CAM system would effect the progress of many disparate fields of research.

Chapter 5

PROGRAMMING LANGUAGES AND COMPUTER HARDWARE

Let us now consider the relationship between programming languages and computer hardware, and particularly the manner in which each has accommodated, or hindered, the development of the other. This chapter examines how innovative computer hardware has been incorporated into a computing environment by changes in programme language design, and conversely how the requirements of programming languages have influenced recent computer hardware projects.

5.1 Introduction

Development in programming language theory and in computer hardware design is each restrained by the other. Progress in language theory has always been hampered by the task of implementation in computer hardware, while the current surge of computing potential which has arisen from the development of VLSI is checked by programming difficulties. In this section we examine briefly the role of programming languages and the effects of VLSI on computer architectural design. We then establish a terminology for the subsequent discussion.

5.1.1 Role of Programming Languages

A computer programme is an intermediate representation of a computation, between the abstract problem and the control signals of the computer hardware. The programming language must be such that its translation from the abstract may be performed by the human programmer, and its translation to the hardware control signals may be effected by another computer programme. Some languages have been designed merely to reflect the hardware configuration and have therefore proved difficult to use; some languages were designed without reference to implementation and have not been implemented. A realistic language must consider both sides of its intermediate role by facilitating the writing of programmes whilst allowing access to the full features of the computer hardware. However, the two objectives are hard to reconcile, and conventional languages represent a compromise between simplicity of usage and the complexity of their execution.

5.1.2 The effects of VLSI technology

Although VLSI has brought a great freedom to architectural design, the resultant systems have been influenced by the characteristics of the medium:

- The finite switching speed of the MOS transistor has lead to the extensive use of computational parallelism.
- The production of processor and memory in the same material has blurred the distinction between them.
- Processing is cheaper than communications, which renders multiprocessing an attractive possibility.

- Chip boundaries both limit bandwidth and increase signal delays which implies that as many functions as possible should be included on a contiguous integrated circuit.

- The fabrication process imposes a limit on this size which is leading to the design of functionally simpler architectures.

The designs tend to be highly concurrent, of limited complexity, functionally autonomous, and often combined into multiprocessing architectures. Ironically, progress has been hampered by the flexibility afforded by VLSI which has destroyed conventional design structures and left the designer "spoilt for choice" with design options.

VLSI presents two immediate challenges with respect to programming languages: can languages express the new architectures which VLSI has made possible, and how can VLSI be best used to implement languages? This chapter considers recent developments relevant to these.

5.1.3 Terminology

To discus the relationship between Language and Hardware, we define the idea of a computing *form*. The term is introduced to encompass both what the hardware and software perform and what the designer is aiming to implement. It is borrowed from neo-Platonic philosophy where it represents the *essence* of a concept (such as "a chair") without the *accidents* (such as the "colour" or the "number of legs") associated with it.

- *A form is an abstraction, or a potential in the computing hardware or software.*

Integer addition is a form which is realized in the ALU; lock-stepped parallelism is a form realized in the architecture of some array processors; the *RECORD* data type is a form which is realized in a PASCAL compiler. Within the hierarchy of hardware component architecture, (microcoding), assembler, compiler, programming language; a form may both appear and become lost – lost in the sense that it is no longer expressible. For example, forms realized in the microcode become lost if the compiler does not utilize the relevant assembler code.

A Computer Language provides a set of forms in which the programmer must express the computation – but these forms must be expressible in those forms which are realizable by the computer hardware. The distance between these forms is bridged by the compiler which acts by expressing the forms of the programming language in terms of the forms of the assembler code. Viewed in the opposite direction, the forms of the computer hardware are developed by abstraction towards the forms of a general computation. However, these abstractions do not always correspond to the hardware forms. This disparity between the high level forms and the hardware forms is often referred to as the "semantic gap" in that the 'meaning' of the high level statements can not be directly expressed in the hardware. One of the rationals behind high level languages is that they provide a mode of expression which is independent from the underlying computer architecture: this necessarily precludes a high level statement of non-standard architectural forms which therefore have to be translated, or deduced, from the high level language.

As an historical example, consider a form which has always been present in high level languages, but which has been realized at different levels of the hierarchy. Floating point operations were extant (with implicit declaration) in the first FORTRAN specification in 1954. The operations were realized through the generation of assembler routines by the formula translating system. The FORTRAN programmer was unaware

of these routines and was affected by their nature only through the restrictions on mixed number expressions. The original specification did not envisage these restrictions, but they appeared in the first release and were not removed until the inception of ALGOL. The assembler routines were later moved to the microcode affecting only the compiler, and the arrival of dedicated floating point arithmetic logic was then transparent even to the assembler programmer. *Floating point numbers* is thus a form whose realization has moved from compiler, through assembler and microcoding, to a hardware component.

5.2 Programming Languages for Computer Hardware

As a new form evolves in either hardware or programming language theory, the other must develop in order to accommodate this form and to allow its inclusion in a complete programming environment. This section surveys the nature, and success, of such Programming Language development.

5.2.1 Vector Processors

Vector Processors provide the form which is the potential to execute a sequence of operations concurrently on a sequence of data such that the hardware for several instructions is active at any one instant. At the simplest level: if a vector pipeline of N units is running, the associated computation runs N times faster than it could without the concurrency. The major problem is how to arrange the computation into streams of data destined for the same operational sequence.

The best known example of a vector processor is the Cray-1[49], which was programmed using the CFT compiler. This was designed to compile Ansii 66 Fortran IV "to best advantage" by "vectorizing" inner DO loops which manipulate and store the

results in arrays and which contain no conditional or jump statements. However, the compiler detection of parallelism is cumbersome. In theory the high level language programmer needs to know nothing about the underlying mechanism and should code the problem in a conventional sequential manner; in practice programmers try to understand the nature of the code which is best optimized and then manipulate the programme accordingly. In effect, they are writing a parallel algorithm in sequential code in such a way that the compiler can best detect the inherent parallelism. These contortions occur because the hardware form is not present in the high level programming language, and they result in inefficiency. For example, in using the Cray-1 to execute circuit simulation programmes "even the most advanced compiler technology has shown poor speedup [achieving only] 12-15 percent hardware utilization"[50].

A variety of languages have been developed for vector pipelined machines which allow the explicit statement of vector structures and operations. Commonly these allow the programmer to specify a computation on all, or a subset of, the vector elements without an explicit DO loop. They do not, however, deal with the techniques for increasing the efficiency of the pipeline's usage which can be as low as 44% with conventional code due to hazards between adjacent instructions[51].

5.2.2 Array Processors

The form of Array Processors is the potential for concurrent execution of a single instruction at many hardware sites, commonly interconnected on a nearest neighbour basis. The assembler instructions provide for data transfers in the various directions as well as local computations at the individual sites.

Array processors have been available since the Illiac IV was produced in 1973 and it is instructive to consider the languages on this machine in detail since sufficient time has elapsed for the first ideas to have been evaluated and corrected. The Illiac IV consists of a central control unit directing 64 arithmetic units each with 2K words of local memory. These units may be selectively disabled according to a 64 bit control register word, and data may be routed between them according to a routing register. The working idea is that an array is stored by mapping one of its dimensions across the distinct local memories; operations on that dimension may then be performed in parallel at each of the arithmetic units.

There are two ways in which a language may be transformed for parallel execution: by providing syntax for the programmer to explicitly code the parallelism, or by analysing the sequential code – that is to say: the form may be stated or deduced. The latter is attractive since it allows existing software to be used on the new machine by providing a new 'clever' compiler. The Illiac IV language of this type is Ivtran: a Fortran analyser. The new syntax approach was used in Glypnir (Algol-based) and CFD (Fortran based). The assembly language ASK was also distributed.

A survey was conducted in the late '70s upon programmer reaction to these languages, and the results used to formulate a new syntax for both array and vector processors: Actus[52]. The results of the survey showed that programmers preferred that the parallelism was explicit but objected to the extent to which Glypnir and CFD reflected the idiosyncrasies of the hardware. For instance in CFD, data structures have to be declared with the parallel component as the first dimension of the array and manipulated in units of 64, and the disabling word was explicitly set by the programmer. Some programmers stated that they had found it necessary to resort to ASK coding to achieve the desired results. It was concluded from the survey that the syntax should "provide the programmer with data and programme structures which reflect the type of

parallelism under consideration ... [and] enable the expression of parallelism in a manner which is suited to the problem and which can be easily exploited by a parallel architecture". Actus was designed to allow the parallelism to be controlled both explicitly and by the data, while removing the effects of the hardware's dimensions as much as possible. Specifically the parallel dimension may be freely chosen both in size and its position within the array, the disabling register is set by condition loops and case statements, index variables can be declared, and data shift and rotate operations are written explicitly.

Similar conclusions were drawn from experience with PascalPL0 which was designed for the CLIP3 image processor: it was "a significant step above CLIP assembly language" but was "too close in spirit" to assembler and one particular approach to pattern recognition "to be successful as a general high-level language"[53]. IPC (Image Processing C) is now the main language for programming the CLIP computers, providing real support for the array handling operations. More generally, there has been a proliferation of image processing machines and of the languages to support them. The trend[54] has been to provide access to the image arrays through FORTRAN subroutine libraries, subroutines which are machine specific. This is also true for general purpose array processors such as the ICL DAP.

5.2.3 Concurrent Processing

The form of Concurrent Processing results from the existence of independent processors whose operations are combined by the high level language primitives of the operating system. Programming for a multiprocessor machine requires the identification of sections of the code which may be run together. This type of divergence has existed in operating system theory for some time through the FORK and JOIN primitives which spawn and consume "processes". There is a problem in providing safe access to data

shared by more than one process. A solution to this is found in Hoare's Monitors[55] which were included into the language: Modula[56]. This implements the form of modules of code which may be entered concurrently by calls from other active modules. The syntax is provided for protecting critical sections of code, and the general technique is that manipulation of shared data is performed only through code which is thus protected. This approach may be seen as a scheduling operating system where modules defined by the programmer form the units for processing. The programmer is given full knowledge of the possible parallelism, and must be aware of the pitfalls and their solutions. Most significantly for the future, these ideas are incorporated into the ADA language which was commissioned and funded by the American Department of Defence as a "standard" language.

5.2.4 Multiprocessing

The concurrent processing in the previous section relies on the programmer to specify and design those portions of code which may be executed in parallel; this was achieved through syntactic extensions to conventional languages. However, the conventional or "imperative" languages have developed very closely to the form of the von Neumann model which seldom match the forms of parallelism which can be found in hardware, and which gives rise to features which hinder multiprocessing. Thus to accommodate the form of cooperating independent processors, non-von Neumann languages must be considered.

The central concept of imperative languages is the "current state" of the computation: a point in multidimensional vector space defined by the values of the programme variables. The current state is altered when a variable is "updated" and this may result from the execution of distinct portions of code. It is this ability of one section of code to affect the environment of another which precludes their parallel

execution. This feature of destructive updating has hindered not only multiprocessing but also the development of programme "proof" or "verification" techniques, and there has therefore been substantial theoretical work into alternative programming styles.

Dataflow is a technique for decomposing a computation into its individual operations and the paths by which data flow between them. The computation may be driven by more than one processor in parallel, since the data dependency is inherent in the graphical decomposition. The organization of dataflow computers may be viewed as a scheduling operating system as was concurrency processing, but with individual operations as the units for processing. The most common architecture is a high speed communication ring serving a pool of processors and a packet store. The instruction code may be also accessed through the ring or, alternatively, duplicated into each processors local memory. The packet consists of an instruction address and (possibly) the result of previous packet executions. A free processor takes an executable packet, executes the instruction, and places the resultant packet(s) back onto the ring.

Languages to support this architecture are formed from several approaches. There is currently research into the technique of entering the data flow graph directly through a graphical language or even schematic capture – in this manner, the programmer would present the algorithm directly in the architectural form. As with the pipelined and array processors, there are also techniques for detection of potential parallelism in sequential code of existing languages (especially for Fortran because of the existing investment in Fortran written software). However the limited utilization of the potential concurrency has prompted designers to draw upon the theoretical studies into "single assignment" and "functional" languages.

An imperative language programme is a sequence of memory updates. Dataflow computing differs fundamentally in the lack of reassignment to a variable: a value is created (possibly in several packets) and then processed. Single assignment languages reflect this exactly. In writing with them, the programmer is constrained by the rule that "no variable is assigned values by more than one statement". The resultant languages are similar to conventional languages with the exception of the lack of pointers, and of lists and other structures normally associated with pointers. Iteration with assignment statements is included with the understanding that each "variable", to which assignment is made, is distinct from any variable of the same name in the previous iteration. With this syntactic structure, the programmer is constrained to producing code which reflects the dataflow execution: the programme is in fact "a linear form of the programme graph". For instance, the favoured language for the Manchester Prototype Dataflow machine (described in the previous chapter) is a single assignment language: SISAL.

Functional languages are based upon the application of functions[57]. There is no current state, no storage of values, and no programme counter. The language consists of objects, functions on objects, and operators to combine functions – the programme is itself a function formed as a combination of other functions. The significant advantage is the natural detection of parallelism in the programme code which allows multiprocessing. There are two multiprocessor representations of the programme structure: Dataflow, where the computation is driven by the flow of data through the programme graph; and Reduction, where the source-and-input is transformed into the output by successive reductions. The mapping of the language form into the multiprocessing architectural form is performed by the operating system and the system architecture.

A radically different approach to multiprocessor programming can be found in the Poker Programming Environment[58]. This was developed to programme a machine of parallel communicating processors by providing an environment which supports a "specificational form close to that used in the theoretical literature to describe display algorithms". The difficulty which Poker addresses is that abstract algorithms have been deliberately developed to match the form of small communicating processors without any corresponding programming language to express the algorithms in a form which is translatable to machine code.

To resolve this, Poker provides interactive graphics whereby the programmer can define the communications graph between labelled processor nodes. The processor labels correspond to a block of sequential code in a conventional style of language with the addition of the data type: *ports*. These *port* variables correspond to the communications ports into the processor and the identifiers are assigned to the communications graph by labelling its arcs at each processor. The data-driven semantics are transformed into synchronous processing by compiler optimization.

5.2.5 Assessment

A new hardware feature provides a new form which must be expressed in other forms if it is to be available to the programmer; we have seen that this outlook has generally been ignored, thus limiting the impact of innovative architectures in a full programming environment. There are three levels at which new hardware forms may be integrated with a high level programming environment:

- A compiler which implements a standard language by utilizing the new feature.

- Extensions to the syntax of conventional high level languages to allow for explicit direction by the programmer

- Design of a new type of programming language which better matches the underlying machine.

A new hardware form should be expressed at as high a level as is necessary to allow the programmer full access.

Compiler detection is a very common technique because of the commercial investment in existing software; however, it is often inefficient because of the information gap between the conventional programming languages and the novel machine features. If the new forms can not be successfully detected they must be expressed explicitly in the high level language. Syntactic extensions to conventional languages are attractive since they retain the established (and familiar) constructs whilst expressing the new features directly. However, if the machine architecture is fundamentally altered (for instance, to non-sequential control flow), then the old language constructs will not apply.

Let us summarize the issues in designing high level language constructs to express a new hardware form:

- New language constructs must really be high level and not merely a syntactic reflection of an assembler language.

- The idiosyncrasies of the hardware should not be found in high level programming languages, but rather hidden in the compiler.

- The constructs should be designed to discourage the production of inefficient code.

- The design must be relevant to the abstract computations which the constructs will be used to express.

The core of the high level language should be the union of the hardware's potential and the computational abstractions.

5.3 VLSI Hardware for Programming Languages

The previous section was concerned with how computer languages have developed to express the features of innovative hardware; this section considers the converse – how hardware, with the potential offered by VLSI, is currently developing in the expression of programming languages.

The advent of VLSI technology has promoted the design and implementation of hardware components where the emphasis is no longer upon language design to accommodate the existing architecture, but rather upon the architectural design to accommodate not only languages but also the needs of the abstract problem. For instance, image processors are designed as pixel specific communicating units; some chips implement single algorithms, such as an RSA-coding chip[59]; and some architectures are designed to execute a single specific programme, such as the MOSSIM hardware simulation accelerator[60]. In the design of the general purpose computer, VLSI affords the freedom to match hardware directly to abstract forms. Considerable attention has been paid to the forms of conventional high level languages[61], particularly that of recursion. Another approach has been to isolate the most frequently used software forms and to optimize these in the hardware (at the expense of the

infrequent forms) to produce faster, more streamlined, designs.

5.3.1 Revised instruction sets

The size constraints of VLSI imply that a single chip processor must be of lower complexity than in conventional architectures. This has resulted in the introduction of the reduced instruction set architectures, first implemented in the Berkely RISC I and II processors[23,24]. The rational is to avoid complex control structures and to aim at fixed length, regular instructions with single clock cycle execution. The disadvantage is that more complex instructions have to be coded in the implemented instruction set by the compiler; this increases compiler complexity (considered a desirable trade-off) and also the memory bandwidth. The forms of the architecture are made simpler while allowing the rejected forms to be realized by the compiler using macros of the simpler assembler forms.

The merits of moving the realization of forms from the hardware into the software can be compared in the two DEC VLSI implementations of the VAX processor. The VLSI VAX[62] is a nine chip implementation of the full VAX instruction set emulating a VAX-11/780 class processor; the MicroVAX-32[63] is a one chip implementation of a subset of the VAX instruction set. The MicroVAX-32 partitions the 304 VAX instructions into: 175 instructions implemented directly, 70 floating point instructions implemented by an additional floating point unit or by macrocode, and the remaining 59 instructions implemented by macrocode alone. The significant result is that the 57.6% instruction subset chosen requires only 20% of the full microcode whilst accounting for 98.1% of the observed execution frequency. If a floating point unit is included, the observed performance degradation due to macrocode emulation, rather than microcode implementation, is only about 4%. The simplifications between the the two VLSI designs (including some alterations to memory mangement) allowed the MicroVAX-32 to be

realized on only one chip with a reduction from 1250K to 101K transistors and only a 20% impact on performance. This supports the suggestion that some forms may be profitably realized in the compiler rather than in the hardware.

5.3.2 Closeness of hardware and assembler forms

The idea of shipping the realization of forms to the software was continued in the design of the Stanford MIPS processor[64]. The design philosophy was to expose the forms of the hardware directly in the instruction set making the software able to perform (and so responsible for) the optimization of the hardware performance. The MIPS is considered to have two low levels of languages: a machine level language, and the more usual assembler language. Assembler is *reorganized* to machine level code by software which performs four major functions: the optimization of implementation dependent forms, the expanding of instruction macros, packing multiple assembler instructions into single machine instructions, and the detailed avoidance of pipeline dependencies and branch delays. This allowed the architecture to dispense with microcoding and pipeline interlocks.

5.3.3 Support for procedure calls

The design of the RISC II processor also contained features to directly assist the execution of a high level language. By supporting only 31 instructions, the designers were able to devote a significant area to a large register file which is used to store frequently used operands so as to minimize the flow of data across the chip boundary. The organization of this on-chip memory is specifically designed for efficient realization of procedure call and return. In general, when a procedure is called, a new "current data frame" is instantiated. At the register level, this means that the previous values are stored and that new registers are allocated and (for passed parameters) initialized. With

a small register set this involves considerable data traffic. In RISC II the large register set is arranged in multiple register banks corresponding to different procedure levels. Each procedure has access to ten local registers, to six *high* and six *low* registers which overlap with the next and the previous procedure windows for parameter and result passing, and also to ten global registers. This arrangement is supported by a circular buffer of 8 register windows maintained by a current window pointer. On overflow, one (or two) windows are written to main memory and are restored only after underflow.

This is an example of hardware design which is aimed directly at high level language forms. The memory arrangement is designed for procedure call intensive C programmes. Programmes without procedure calls derive no benefit from the large local register set and are restricted to the one window; programmes with few calls benefit only slightly. Other programming languages present difficulties through the nested scope rules and the increasingly common use of a large number of global variables. It is possible that better use might be made of the large register set if it could be controlled directly by an optimizing compiler.

5.3.4 Assessment

It is possible to provide features in VLSI designs which assist in the implementation of specific software forms, conversely fixing high level language forms in hardware limits the generality of that component. In designing a VLSI processor the task is to select the forms which are to be implemented, and then to decide at which level they may be best realized. The experience of the previous projects has confirmed the virtues of a reduced instruction set approach which shifts the realization of some forms out of the hardware and into the compiler – but the verdict is still "not proven". The 31 instructions of the RISC chips appears extreme in contrast to the 175 instructions of the MicroVAX. The success of the RISC chips is more due to the large number of

registers[65], than to the streamlined instruction set. The 20% speed reduction of the MicroVAX–32 over the VLSI VAX questions the supposed speed advantage of a single chip implementation, and the nine chip VLSI VAX denies the necessity of compiler assistance in VLSI realization of system complexity.

The above examples do show, however, that VLSI allows for a complete reassessment of the conventional roles of hardware and software in computing systems. Further, new software techniques are evolving (at the compiler level) to encompass the features which are emerging from this hardware medium – on the other hand, new hardware designs are evolving to realize software forms. In this, the use and applicability of VLSI must be seen in the context of compiler techniques and software requirements; but if a high level form can not be realized efficiently through software (as with Content Addressability), a designer must recourse to specialized hardware.

Chapter 6

THE CONTEXT FOR LANGUAGE DESIGN

The original motivation for this thesis was that a CAM component could provide a new medium for the design of data structures and algorithms by software engineers; to establish this claim, it is necessary to develop an example programming environment which is clearly practical and accessible to the high level language programmer.

In the previous chapter we found that a high level language is the necessary link between the abstract computation and the executing hardware, and that the language must therefore combine both the desired computational, and the practical hardware, forms. This chapter is concerned with the development of a computer architecture which incorporates a CAM, and an abstract characterization of content addressability itself. In this way we establish a programming environment which the language development of the next chapter will complete.

6.1 Triplet Machine Architecture

The aim is to explore a practical architecture for the development of a high level language – to this end, we consider a possible organization for the CPU in the *Triplet* model and develop a possible set of assembler instructions upon which higher levels of programming languages may be based. This set ignores (possibly idiosyncratic) pin level and microcode optimization of more complex operations; it is designed to encompass the pertinent features of the CAM and so to validate subsequent linguistic constructs.

6.1.1 The *Triplet* Model

We saw in chapter 1 not only that CAM has advantages over RAM in the implementation of some data structures, but also that RAM has advantages over CAM in the implementation of others: thus to enhance the software design environment, it is desirable to include both modes of addressing. The *Triplet* architecture consists of a CPU connected to both a CAM and a RAM – it is the conventional von Neumann architecture enhanced with a second mode of addressing.

Conventional computer operations are controlled by a single CPU. Programme instructions are read sequentially; operands are addressed and passed along buses to areas where the logical operations may be performed. The *Triplet* model makes no explicit attempt to change this mode of operation: the CAM is viewed as sitting on the end of a CPU data bus, and the CPU's instruction set includes the commands necessary to drive the CAM. It is viewed as an equal partner with the RAM in providing memory which the programmer may use as is appropriate to the particular programme. The CAM introduces a mode of addressing which was prohibitive with the Von Neumann bottleneck, but the sequential central control remains.

6.1.2 Design Issues

Since the CAM and RAM are being viewed as equal partners in the storage of computer data, they will interact with each other during the CPU driven computation, and we need to examine the implications of this interaction on the machine organization. The CPU registers of a *Triplet* machine must provide:

- a scribble pad for CAM-RAM communications

- manipulation and arithmetic with RAM words
- manipulation and arithmetic with CAM fields

There are two points of difficulty:

- For effective usage, the CAM word length will be larger that of the RAM.
- In RAM, the conventions for the representation of data in bit codes are dependent upon fixed word sizes and the flexibility inherent in CAM field lengths poses the problems of compaction and alignment.

6.1.3 Control

For the two types of memory to communicate, the relevant RAM storage address must be specified – which implies that data transfer between RAM and CAM must be supervised by some controller. Current large computer systems incorporate I/O handlers which are essentially limited processors to supervise the transfer of data from one storage medium to another. These would not be applicable in this case where the interaction between the CAM and its controller is more programme dependent: frequently there is no bulk transfer of data.

The CAM is more 'active' than conventional storage media. It is not simply storage with novel addressing but has characteristics which tie it closely to the operation of the CPU: a variety of different operations can be performed on each of the words addressed by a single comparison, perhaps conditionally upon the word's value. For such an algorithm, the CAM would best be controlled by the same unit which drives the computation: the CPU itself.

6.1.4 Register Alternatives

Let us consider the options for the CPU registers. Historically, the CPU interacts with RAM using five broad categories of high speed registers incorporated in the CPU itself:

a. With no local registers, the CPU communicates directly with main memory.

b. A single, often extendible, register which holds a single operand for both unary and binary operators, and then the result; giving a uniform operating sequence for the ALU.

c. Multiple arithmetic registers which can provide the second operand for a binary operator, and can be paired to provide an extension.

d. A hierarchy in that the registers can be viewed as a local cache with high access speed, and the interface as essentially being the same type as (a).

e. A stack to assist in the sequencing of ALU operations.

In deciding the register arrangement in a *Triplet* architecture, let us consider how a CAM might function in a computer with these types of CPU register design:

a. It would be inefficient to store every CAM word accessed in main RAM before performing simple field operations. For instance, if there were no local registers when updating CAM entries, each CAM word would be passed twice along the RAM bus for only temporary storage. Thus there should be registers for CAM-only operations located in the CPU.

b. The single arithmetic register is specifically designed to the needs of the ALU, and adaptation to accommodate a CAM might defeat this purpose. However, a specially designed single register for the assembly and decomposition of the CAM words is a possible approach.

c. A common use of multiple arithmetic registers is to provide extensions of the standard word size to form larger words: in DOUBLE PRECISION. If the CAM word length were limited to multiples of the RAM word length, CAM word assemblage could occur in the ordinary registers. The CAM instruction set could then form an extension to the existing machine code. This poses a restriction on the alignment of CAM fields in that conventional addressing of registers is only to the byte level and non-byte aligned fields would require complex manipulation without bit addressing; thus it may be useful to limit the fields to multiples of bytes. This is the usual convention for type representation.

d. A high speed cache might alleviate the problems associated with CAM words of larger length than RAM word, and this is essentially a development of (c).

e. A stack alone would be of little use to CPU-CAM interface since the stack would provide the same sequential access without allowing the other CAM functions. However, a hardware stack or stacks designated for CAM interfacing would provide a software simulation of read-only group. This would not be as powerful as additional hardware comparison groups, but might be a useful alternative.

The above analysis suggests that a "CAM register" in the CPU may be either a specialized register with bit or byte addressing or an aggregation of RAM-sized registers with byte addressing. The former is a specialized interface for assembly and dispersal, whilst the latter allows the registers to be general purpose.

6.1.5 Bit *vs* Byte

As a first step to designing the CPU architecture, we must allow the trivial operation of reading a value from CAM and storing it in RAM in the appropriate representation and word size. For a system to support CAM fields of arbitrary bit size, it must provide bit manipulation functions. For instance, consider translating an integer and floating point number into a larger field size: the hardware must right align and expand the most significant bit for the integer, and left align and pad to the right with zero bits for the floating point. These operations would require bit orientated functions which are not easily provided without specialized registers.

The problem is not so acute if the fields are of arbitrary byte size. Then the byte addressing in many conventional CPU register sets allows for simple algorithms provided a "test most significant bit" operation exists. For the *Triplet* machine, CAM fields of multiple bytes seem to be preferable; primarily for the simplification of the CAM-RAM interface, but also for the efficient addressability of the field boundaries.

6.1.6 ALU organization

The design of the CPU must accommodate the performance of arithmetic logic functions on locally stored data. Arithmetic and logical functions may be performed on RAM words of fixed length and CAM fields of variable byte length. This is possible if the ALU is designed byte serially, and the instruction set contains a field to specify the number of bytes through which the instruction cycles. The machine instruction is thus the instruction-identifier, the number of bytes, and the initial register address. This approach removes all problems with alignment to word boundaries whilst maintaining the conventional capabilities of the RAM machine. The full assembler language could

also contain arithmetic instructions which assume standard length operands.

6.1.7 Local memory size

The response by CPU designers to VLSI has been either to incorporate a large numbers of functional units, or to increase the capabilities of a few. In the latter category is the INMOS transputer which has a reduced instruction set and has dispensed with floating point hardware, but instead incorporates a novel communications interface and 4k bytes of local high speed memory. The number of high speed registers is no longer so limited by the speed of the technology.

To perform the CAM control efficiently, the registers should hold at least five CAM words (3 data + 2 mask) independently. This ignores possible compression of the mask storage. If a CAM word is 16 bytes (for instance), this would imply the need for over 64 registers and so an address of at least 7 bits. With recent design capabilities in mind, this suggests a CPU which uses a full byte address extension in its instruction set giving 256 single byte registers.

6.1.8 Group Register

To interface with the daisy-chain addressing system, the CPU must specify an address group and provide a mechanism to interpret the addressed group's output signal. The hardware design of the CAM is such that non-addressed operations are uneffected by a word being addressed, so there is no need to 'de-address' a group when there is no addressed operation. The physical effect of addressing and de-addressing a tag group is to charge and discharge a signal line, but programme flow involving repeated use of the same tag group is common. By maintaining the charge on the line until it 'must' be discharged, the system both saves power and possibly operating time. Consider the

following algorithm:

```
while (there are set tags in group 1)
{READ <LINE 1>
  ...
 NEXT <LINE 1>
}
```

If the address line is discharged after each instruction, this would occur three times in each loop wasting power in the CAM and time in the necessary delay to allow the signal to propagate through the system.

The alternative is to introduce the concept of a "current group" control register which maintains the signal on the line corresponding to the last value which was loaded. If this register is simply loaded at the execution of each addressing machine instruction, the same delays result since the value loaded might have changed. The delay could be avoided by using a microcoded test-and-set arrangement to determine if the value has changed. This entails a more sophisticated register and a machine instruction delay each addressed operation.

6.1.9 Condition Flag

The output signal of the addressed group must be available to the CPU. This can be achieved in the same manner that overflows, logical comparisons, etc., are handled with conventional ALUs. A condition flag is set to be ON when the current tag group is empty, and OFF if the group addresses a word. The flag can then be tested by the machine code using conditional jumps, and in more sophisticated systems used to trap, with an error message, any addressed instruction on an untagged group. The above code could now be written as:

```
JUMP_IF_FLAG_ON 1 on
#top
READ 1
 ...
NEXT 1
JUMP_IF_FLAG_OFF 1 top
#on
```

This flow control allows simple programming of CAM iterations, and fits into the established operating principles of CPU architectures.

The output of the empty tag group signals the fact that memory is full. If this occurs in conjunction with a write-new-word operation, then there is an error condition which may be dealt with by invoking a standard error-interrupt mechanism.

6.1.10 Alternative assembler style

An alternative would be to leave the setting of the group register to the assembler programmer. This has the advantage of forcing the hardware form into an assembler form so promoting a closer match as further software forms are devised. This conforms to a current trend for moving responsibility for 'house-keeping' functions from the hardware to the software. Additionally the assembler code is simplified and the efficient use of the CAM (through iterations on a group) is reflected in 'clean' assembler code generation. The micro-coded test-and-set operation becomes less important as the update of the register is less frequent; it may be rejected altogether as a trade-off between CPU complexity and operating speed. With the explicit setting of the group register, the above code becomes:

```
LOAD_GROUP_REGISTER 1
...
JUMP_IF_FLAG_ON on
#top
READ
...
NEXT
JUMP_IF_FLAG_OFF top
#on
```

6.1.11 Assembler Instructions

By way of a summary, the following is a possible set of assembler instructions which provide the interface between the CAM and the CPU. The data movement instructions all act on the number of bytes in a full CAM word starting at the designated register in the CPU. The variables l, m, and n are each a value or a register address.

Memory Control:

EM - Empty Memory: initially, and at any stage of the computation, the CAM may be emptied by resetting all the empty tags.

CMP n m l - Compare: perform the masked comparison on group l using the CAM sized word beginning at byte n as comparand, and that at byte m as the mask (this operation is independent from the current value of the group register).

LDG n - Load the group register with n, to specify the currently addressed group.

RE - Remove the word currently addressed (on the group designated by the group register) is removed by setting its empty tag.

NX - Next: the word currently addressed is de-addressed by making its address tag passive on the current group.

MP 1 - Make Passive the tag in group 1 in the tag set of the currently addressed CAM word.

Data Movement:

CLD n - Load the currently addressed CAM word into the registers beginning at register n.

CST n - Store the word beginning at register n into the currently addressed CAM word.

Programme Flow Control:

JP label - Jump to #label if the condition flag is ON, implying that there all the tags of the currently addressed group are Passive.

JNP label - Jump to #label if the condition flag is OFF.

The following is a brief example of assembler code using this instruction set:

```
LDG 1
 ...
JP on
#top
CLD n
 ...
(LDG 1)
NX
JNP top
#on
```

which gives the now familiar read-while loop. The second "LDG 1" command is only necessary if the group register was changed during the proceeding code; this would be checked by an optimizing compiler.

6.2 An Abstract Computational Form

Before developing a high level language syntax to run on the *Triplet* machine, we must consider the computational form which this is likely to express. Chapter 5 indicated that a high level programming language should unite the hardware and computational forms, and so we will now develop content addressability as an abstract computational form. This evolves into a characterization of 'CAM orientated' problems. The relationship of the abstraction to the CAM hardware form is emphasised by examples of assembler code, and we discus the necessary features of a programming language to complete the environment.

6.2.1 A new terminology

Previously we have considered CAM in terms of its affinity with common data type abstractions, now we seek a new terminology to express the CAM abstraction itself.

The words in a pure CAM model have no defined order. When addressed, the component allows access to the active words one at a time, but only in very controlled conditions may that order be taken as significant. To emphasis this quality, we affirm that the address defines a (possibly empty) *bag* of words. A *bag* is a mathematical entity similar to a *set* except that the *bag* may contain repeated instances of the same element.

A CAM word consists of one or more sub-words or *fields*. When addressing CAM, a programmer is concerned with two classes of fields: the address fields, and the data fields; the known, and the unknown.

- The classification is not fixed, but re-specified in each comparison.
- Fields of both classes may be updated while the word is addressed.
- Either class of field may be empty; for instance, the whole of defined memory may be read, or an address may be used purely to ascertain the existence of a word.

Unlike the order of words in a bag, the order of fields within a word is significant, since this defines its interpretation: the pair of fields (field_x, field_y) will render different values when accessing a word if the order is reversed (field_y, field_x). Thus a word is viewed as an ordered collection of fields. We will describe the general word of 'n' fields as an *ordered n-tuple*.

To summarize:

- an address is defined by the values of zero or more specified fields,
- an address defines a bag of ordered n-tuples.

6.2.2 Variformity

The variable classification of fields affords a *variformity* to CAM words. The *dual* value of a field is both data and address – which can allow the direct algorithmic construction of addresses (well beyond the simple base/offset manipulation) such as the computed value of a function's argument being used to address its value. As a combined

effect of this duality over all the fields, the same collection of words may be placed into different collections of bags according to which field or fields are defining the address. For instance, a directed graph is reversed by using the head_node instead of the tail_node as the address field: a simple device to implement back tracking. Thus an algorithm may progress by directing the data according to one collection of bags towards a solution specified by another.

As an example, consider the outline of an algorithm to determine the components of an undirected graph: that is, given a graph G = (G, E), determine the maximal connected subgraphs {G1, G2, ...}. The algorithm itself is trivial, but it demonstrates the technique of switching perspective on the CAM word. The Traversal algorithm of chapter 1 for undirected graphs is used with an extra field 'graph_number' in the created set 'visited'.

```
Set a variable counter = 1;
WHILE there exists an unvisited node
 {Select any one and Traverse from it making the
     graph_number field of each visited node equal
     to counter;
  Increment counter;}
```

Following this search, the component graphs are retrieved by addressing the visited set according to the numerical value of the graph_number field: the workspace has become the address.

6.2.3 Bag operations

Let us consider the type of operations which might be performed on a bag of ordered n-tuples. The question is this: we have performed an addressing comparison on memory which has defined a (possibly empty) bag of n-tuples, what can we do with

the bag?

Boolean test: This operation determines whether or not the bag is empty. A conditional jump can be programmed by the "J(N)P label": jump if (not) passive, assembler instructions. This differs from normal boolean operations in that the jump is arbitrated according to a condition flag set by the addressing mechanism rather than by the ALU.

Pick out one n-tuple: In some cases an algorithm may require a single n-tuple from the bag. There are two problems in deciding the exact coding of this operation

- How do we cater for an empty bag? Some iterations will be of the form: WHILE there exists – select one, in which case the programme flow is already determined; but if there is no such guard, it is possible for control to pass to statements addressing an empty tag group and this must be a runtime error. Unguarded picking could be rendered impossible through the language design.

- When does the tag group become available for further comparisons? The answer must be that the tag group forms the same bag until a new one is defined, but it might be useful to provide language constructs which limit the textual distance of a bag's manipulation from its definition.

The assembler code for this operation is a comparison, the loading of the relevant number into the group register and (possibly) a software error trap:

```
CMP n m l
LDG l
(JP error)
```

Iterate with each addressed n-tuple: Each n-tuple is to be taken and directed in turn. This seemingly simple iteration has two complications:

- The bag must remain defined throughout and so the operation monopolizes a tag group for the whole block of code. We can not re-address each time at the head of the block since it may have new n-tuples or a different picking order. This dedication imposes limitations on the code within the iterative block, and especially on the nesting of bag operations.

- The fate of each n-tuple determines the instruction by which the next one becomes available. If the n-tuple remains (possibly altered) in the same location in memory, then the iterative loop must use the instruction "NX"; if the n-tuple is removed ("RE") from memory, then the next n-tuple is automatically addressed through the hardware. A general language construct could either impose one such action and prevent the other, or require their explicit specification for each iteration.

As an example, the following is the assembler code for an iteration which removes each addressed n-tuple from memory;

```
CMP n m l
LDG l
JP onwards
#top
RE
JNP top
#onwards
```

6.2.4 Bags as a resource

Let us consider the limitations imposed by the hardware on the design of an efficient programming language. The component offers two comparison tag groups, and one empty tag group: only two bags can be defined at any one stage of the computation, although it is always possible to write to a new location in memory.

If an algorithm is constructed as: perform this code block on each n-tuple in the bag *as it is now defined*, then one of the comparison groups must be dedicated to preserving that bag throughout the algorithm: the code can only direct one bag at a time. Of course, if this code block contains the same type of algorithm, then the inner code block will be unable to define a bag at all. These restrictions apply also with respect to the coding of functions or subroutines called in the operations.

6.2.5 Assessment

In a computer architecture which incorporates a CAM and a RAM, data may be stored under either mode of addressing. The choice depends upon the data structure, its manipulation, and the ease with which these can be mapped onto the memory components. CAM should be selected for:

- the storage of data elements whose structure is: inherent in their names; can be represented through ordered n-tuples of their names; or can be transformed into such by introducing new names to impose an association hierarchy,

- manipulation which exploits, or is better expressed by using, the duality of fields,

- algorithms which apply the variformity of CAM words.

Consider a bag defined by the values of certain fields. If another field were then introduced into the class of address fields, a *sub-bag* is defined. Indeed we could say that any bag is the union of all bags defined by all collections of fields which have that bag's address fields as a sub-collection with their values fixed – "but what does it profit a man?".

The bag model emphasizes the association between words afforded by CAM, and sub-bags show that the information in such an association is related to the degree of precision with which the bag is defined. Ultimately, the objective of an address specification could be to define a bag of one and only one n-tuple: to address a single word in memory. Yet in this too, associations implicitly exist through the bags of less precise definition. The significant feature of CAM is its affinity with such an association, unlike RAM in which association requires data structures and software support – an affinity which allows simple and efficient manipulation of the elements of an association, both individually and collectively.

Chapter 7

HIGH LEVEL LANGUAGE DESIGN

This chapter considers a possible language syntax to direct the CAM component in a *Triplet* machine. The intention is to show that this architecture is plausible as a general purpose computer by prescribing a suitable programming language based on the assessment of the similar projects surveyed in chapter 5.

It is not the intention to define a somehow optimal syntax but rather to demonstrate a practical and efficient interface between the programmer and the CAM component. We develop through low level language constructs which are closely associated with the hardware, to higher levels where the abstractions of the previous chapter dominate the appearance of the syntax. This follows the strategy of deriving a language from the union of the hardware forms and the desired computational abstraction.

7.1 Strategy

Firstly, we must decide what level of changes are necessary to provide the programmer with access to the enhanced hardware. We saw in chapter 4 that the CAM form can accommodate a variety of new programming language styles (and also new architectures) but it is not *necessary* to alter conventional language structure to direct the *Triplet* architecture. In this example, the conventional language constructs will run on the conventional computer configuration (a CPU−RAM pair), and it is only the CPU−CAM interaction which requires linguistic development.

On the other hand, alterations in the compiler alone are not *sufficient*. This would limit the benefits of the CAM to a novel compiler storage allocation scheme, and perhaps efficient array handling for an array processor: the hardware CAM form is not expressed in conventional languages, and can not be deduced from them.

Thus we proceed with the middle option described in chapter 5, that of syntactic extension. The aim is to extend a conventional programming language, to one which can direct the actions of a *Triplet* machine. We will consider various options and examine their suitability for providing the necessary syntactic extensions to accommodate the *Triplet* architecture. These options were developed and investigated through the design of prototype translators and a software functional simulation of the CAM. The development of these "compilers" afforded the ability to write and run *Triplet* language programmes, and isolated the pertinent issues in the language development.

7.2 Methodology

Let us briefly consider the organization of this software. The base language was chosen to be C[66]. The CAM simulation is implemented by arrays corresponding to the memory and tags, with serial searches to perform the comparison. Access to this data base is through subroutines corresponding to the CAM functions which are essentially the assembler commands with parameters in place of register numbers. The CAM word is modelled as a byte array. The object of the translator is to convert the extended syntax into standard C code including the simulator subroutines. In the event, several translators were developed implementing progressively higher levels of programming language.

The first translator converted CAM assembler code inserts into the appropriate subroutine calls and goto statements. This required the manipulation, by standard C code, of dedicated 'CAM' array variables which were then passed as parameters to the simulation subroutines. The translation was essentially textual substitution and was effected by the use of the lexical analyser: Lex[34].

The next translator converted a low level programming language. This allowed the declarations of the new variable types which were maintained by the compiler rather than programmer conventions. Simple CAM programming statements were included, and so a syntax analyser (Yacc[67]) was required. The technique was to filter out and modify the extended syntax only. This required knowledge of each variable's 'type' (to trap the extended syntax), so the translator maintained a hash table data base derived from the declaration statements. Further 'traps' were provided for the analysers by baroque tokens in CAM statements (such as "[(", or ",,"); these would not be needed if the programme were fully parsed – they are a feature of the translator and not the syntactic extensions. As an example of this language, the following is the coding, and the translated C code, of the set theoretic operation *Intersect* using the data structure and algorithm described in chapter 1:

```
Intersect(A,B,C) char A,B,C;
{cword cee, masc;
 masc <- [( MASK set_id ,, MASK elem )];
 foreach [( A set_id )] 1 -> cee
  {cee = [( B set_id )];
   set_tag(cee, masc, 2);
   if (next(2))
    {cee = [( C set_id )];
     enter(cee);};};}
```

```
Intersect(A,B,C) char A,B,C;
{int cee[2], masc[2];
*masc= *(masc+1)=0;
_temp= MASK ;
full_write(masc,*set_id,*(set_id+1),&_temp,1,32,0);
_temp= MASK ;
full_write(masc,*elem,*(elem+1),&_temp,1,32,0);;
*_mask = *(_mask + 1) = 0;
_temp = A ;
_for(*set_id,*(set_id+1));
_i = 1 ;
set_tag(_comp,_mask,_i);
while (next(_i) ? load(cee,_i) : 0)
 {_temp= B ;
  full_write(cee,*set_id,*(set_id+1),&_temp,1,32,0);;
  set_tag(cee, masc, 2);
  if (next(2))
   {_temp= C ;
    full_write(cee,*set_id,*(set_id+1),&_temp,1,32,0);;
    enter(cee);};};}
```

This language is still very closely related to the hardware operation. For example, there is the explicit creation of a *mask* word, and most CAM iterations had to be explicitly developed from standard C statements.

The final translator removed some of these problems, and developed more sophisticated iterative and assignment statements. The shortcomings of this compiler lead to further ideas on appropriate syntax. The above example would be coded as:

```
Intersect(A,B,C) char A,B,C;
{cword cee;
foreach (@ set_id A @) -> cee
  forfirst (@ set_id B elem elem<@cee [# 2 #] @);
  else enter(@ set_id C elem elem<@cee @);}
```

7.3 Bag names

The most severe limitation discussed in chapter 6 was the number of comparison tag groups and hence the number of bags which can be defined at any one stage. This number is a feature of the hardware and was selected in the prototype deign to be two according to the criteria of chapter 2; in future system design the number may be increased without prejudice to this discussion.

While the high level language should protect the user from the idiosyncrasies of the hardware, it should not do so at the expense of efficiency. The high level programmer must know that there are only two bags, since if he used more bags there would have to be continual re-addressing of each bag by the compiler generated code, and also restrictions on the bag operations. The programmer must also be able to specify to which of the two bags he is referring, which implies the need for some syntax to distinguish between them.

The translators all referred to the comparison groups by number and assigned the addressed word (if any) to a declared RAM variable. There are two objections to this approach: the use of numbers to designate hardware control lines does not promote the aspect of a high level language, and the necessary assignment to a RAM variable defeats the advantages of the CPU register organization which was designed to allow the processing of CAM words without interaction with the RAM. The following is a possible convention which overcomes these problems.

At any one stage of the computation there are two (possibly empty) bags which are *in scope*: up to two words in CAM which may be directed. The scope of a bag, according to hardware, extends until the bag is redefined – until a comparison is performed using that tag group. In performing any bag operation, including redefinition, it is necessary to specify which of the two bags in scope is to be affected.

This implies that they must have names. Since there is no distinction between them, we shall christen them as Tweedle*dee* and Tweedle*dum*. The convention will be to augment a command word which directs a bag with the name of the bag: thus next becomes next_dum or next_dee. This allows the manipulation of the dee and dum variables as compiler maintained CPU register locations: no RAM locations are involved.

7.4 Basic Operations

The operations to manipulate bags were introduced in chapter 6. This section considers the design of a syntax which expresses these operations clearly and without ambiguity. We start with basic functions and then develop a syntax with features to assist the high level programmer.

The programmer needs direct access to the fundamental addressing functions. These are the comparison, next, remove, and the boolean output of the address line (see chapter 4). These are sufficient for the programmer to fully direct CAM addressing; their necessity will be considered after discussing a syntax.

7.4.1 Simple boolean

Let the syntax for the boolean test be the reserved word *empty* which has the value true if and only if the associated bag is empty. The compiler will be able to trap this word and implement the form using the specific assembler jump statements. The high level programmer may perceive it as a system maintained boolean. Thus we have conditional branches such as:

```
while (!empty_dum) /* while the bag dum is not empty */
if (empty_dee)     /* if the bag dee is empty */
```

7.4.2 Address Drivers

Similarly let the two commands which drive the daisy-chain address signal be the reserved words *remove* and *next*. It may be useful to consider these as returning a boolean value corresponding to whether the bag is empty following the operation; this leads to a more succinct programming style but also gives rise to a source of programming bugs. Assume that the returned value is true if the bag is not empty (this is the opposite of *empty*, but enhances programme readability), and consider the three loops:

```
while (next_dum) <statement>
do <statement> while (next_dum)
while (!empty_dum) {<statement> next_dum}
```

The first loop has the effect of ignoring the first member selected from the bag which would probably not be intended. The second loop avoids this pitfall but, by moving the test to the end of the statement, allows the execution of the statement even if the bag is initially empty; again, this may not have been intended. It depends upon the implementation whether or not a CAM access on an empty bag is trapped as a run time error, but this is a pernicious bug in that it is computation dependent. The third loop lacks the simplicity of the first two and ignores the returned value of next_dum, but it does provide a *safe* algorithm.

7.4.3 Bag definition

Let the new definition of a bag be achieved by the reserved word *define*. This is a function which takes a <pattern> as an argument to describe which fields are relevant to the addressing comparison and to specify their values. This raises two issues: should *define* return a boolean value, and should there be two statements provided in the syntax to allow separate coding for whether of not <pattern> defines an empty bag? These features are illustrated by the following three (possibly) equivalent pieces of code:

```
define_dee(<pattern>)
if (!empty_dee) <statement1>
else <statement2>

if (define_dee(<pattern>)) <statement1>
else <statement2>

define_dee(<pattern>) <statement1>
else <statement2>
```

The first example has only the operation of defining a new bag followed by a branch dependent on whether or not the bag is empty; the second uses the returned value of *define* to decide the branch; and the third allows *define* to replace the *if* statement. These are essentially three styles which are compatible within one language. The second is desirable for reasons of consistency. The third has the advantage of encouraging the programmer to consider which portions of the code are dependent on the bag being non-empty and to gather these into <statement1> – clearly this programming technique is possible without the aid of special syntax, but a syntax should be conducive to *safe* programming. The first style could be used if either, or both, of the other styles were included in the syntax.

7.4.4 Example of necessity

It has not yet been demonstrated that a syntax without the low level memory commands (*next* and *remove*) could not perform the same bag manipulations in a more abstract syntax. Consider the following code:

```
define_dee(<pattern>)
define_dum(<pattern>)
while (!empty_dee && !empty_dum)
 {if (<conditional>) remove_dum
  else next_dee}
```

This is an example of the type of bag interaction to which a programmer should have access but which a more abstract syntax could not embrace – the commands are necessary.

7.5 Scope and iterations

A high level syntax can provide iterative constructs to replace the explicit coding of common loops; a CAM orientated syntax can provide iterations which support the bag abstraction. A common bag operation is the execution of a portion of code on each member of the bag:

```
define_dee(<pattern>)
while (!empty_dee) {<statement> next_dee}
```

This can be replaced by a syntactic construct such as:

```
foreach_dee (<pattern>) <statement>
```

An added advantage is that this allows for compiler checking of the scope rules, namely that the <statement> contains no redefinition of the *dee* bag and (possibly) also no next_dee or remove_dee commands.

A similar loop could be effected by using the remove_dee command to finish the code block rather than next_dee. This results in the removal of each word addressed by <pattern>, thus:

fromeach_dee (<pattern>) <statement>

Consider the contrast between the *foreach* loop and the following:

while (define_dee(<pattern>)) <statement>

This specifies that the comparison operation is performed on each iteration. This was found to be a common construct where the <pattern> is used as a dynamic queue of items to be processed with <statement>. An intermediate form was also commonly encountered: namely, where the code block is repeated on each member of the bag and then the comparison is performed again and the form repeated if the bag is non-empty. This is cumbersome to express in the syntax developed so far, but the assembler coding is trivial:

```
#top_top
CMP n m l
LDG l
JNP bottom
#top
. . . . <code block>
LDG l
RE
JP top
JUMP top_top
#bottom
```

The algorithmic effect of this code is to implement a breadth_first search since the members of each new bag were created by the processing of the previous bag. This is an important form which can be efficiently implemented in the assembler code, hence the high level language must allow its easy expression, thus:

repeat_with_dee(<pattern>) <statement>

7.6 Removal of CAM words

The CAM component has the facility to make all locations 'empty' by setting all the empty tags to active. This is a necessary prelude to the beginning of a programme's execution and should be generated automatically by the compiler. In addition, a high level programmer may wish to use this operation explicitly during the programme; the appropriate form already exists in the syntax as:

fromeach_dum();

since the *null* pattern assures that all of memory is addressed, and the *null* iteration block implies that only the implicit *remove* operation would be performed on each addressed word. The compiler should trap this expression and use the appropriate single assembler instruction, since the naive coding would take time preportional to the number of words in memory.

Consider a temporary data structure built in CAM, perhaps used in a subroutine. There are obviously cases where such a structure must be unbuilt and removed after the immediate algorithm is complete. The syntax developed so far would enable the programmer to achieve this explicitly, but in a block orientated language it is desirable to introduce syntax which presents this garbage collection in the usual manner. For this, we define the *local* declaration which is written with the normal type declarations at the beginning of a block:

local (<pattern>), (<pattern>);

The semantics are that, at the end of the current block, all bags are removed which are defined by the declared addresses. This declaration refers only to CAM words; it does

not initiate; an allocation of space nor a writing to memory; no action occurs until the end of the block.

7.7 Field specification

The syntax for assignments and address specification depends upon the manner in which the fields of a CAM word are specified and accessed. A field has two properties which the syntax must allow to be expressed: the position of the bytes within a CAM word to which the field refers, and the interpretation (or "type") associated with these bytes. We will start by imposing constraints on the fields and develop different styles of syntax as these are relaxed.

7.7.1 Fixed size with explicit ordering

Initially we consider all CAM words to have fields of the same fixed sizes. In this case the fields may be syntactically specified by their relative textual position within an n-tuple:

(field1, field2, field3, field4 ,etc)

Not all the fields will be relevant to each reference, and so to maintain a readable format we include a *wild card*: *, whose exact meaning depends upon context. Irrelevant fields at the right hand side of the word are not written down, thus the following are equivalent forms:

('V', 23, *, TRUE, *, *)
('V', 23, *, TRUE)

Addressing

An address is defined by the value of zero or more specified fields; in this case the fields are specified by their relative position and the wild card indicates that a field does not form part of the address definition. Thus the previous example denotes the address of a word whose first field is 'V', whose second field is 23 and whose fourth field is TRUE.

Assignment

Reading the value of fields from memory only applies to those fields which were not specified in the address and so we may replace the wild card of such fields with an assignment operator, thus:

foreach_dum('V', 23, *, = fred) <statement>

where the address is ('V', 23) and at each iteration the fourth field is assigned to the variable *fred*. An important point is that the interpretation of the fields' bit pattern is dependent only on the "type" of the variable to which it is assigned.

Writing a value to memory requires the specification of the values of all the relevant fields which may be achieved in the obvious manner. There are two types of writing: one to a new memory location, and one to the location within the bag definition. The former does not have a *dum/dee* extension, so we will define a further extension: *new*, thus:

enter_new('V', 23, *, 'a')
enter_dum('V', 23, *, 'a')

We will assume the convention that when updating a bag member, the value of the wild card fields is unchanged: it is the same as that already in memory. The value of wild card fields with the enter_new command is (arbitrarily) zero.

There is one special case which should have a syntactic expression, and that is the copying of a CAM word from one location within memory to another. For this the words *dum* and *dee* are considered as reserved words, and used to designate their respective 'current' n-tuples. This gives rise to expressions of the forms:

```
enter_dum(dee)
enter_new(dum)
```

As an example of the syntax developed so far, the following is a possible coding of the Traversal procedure outlined in chapter 1:

```
Procedure Traversal(start_node) char start_node;
 {char selected_node, head_node;
  enter_new('visited', start_node);
  enter_new('to_do', start_node);
  while (define_dum('to_do', = selected_node))
   {remove_dum();
    foreach_dee('edge', selected_node, = head_node)
     {if (!define_dum('visited', head_node)
       {enter_new('visited', head_node);
        enter_new('to_do', head_node);}}}}
```

7.7.2 Variable size with explicit ordering

If the words in CAM do not all have the same field boundaries, then the syntax must allow the declaration of a CAM word type: that is, an identifier associated with a sequence of specified field sizes. This can then be used to indicate the field boundaries of each CAM word as it is directed. The declarations could be of the form:

```
CAM_word shape(2,4,1,8)
```

which associates the identifier "shape" with a CAM word whose first field is two bytes, second field is four bytes, etc.. The use of explicit numbers may not appear to be a high

level language convention, but some specification of field size is necessary if it can vary. An alternative is to use keywords to correspond to different numbers, similar to the implied numbers in conventional language type declarations.

There need be no alteration to the syntax of the previous section except that the identifier must be specified with each <pattern>, thus:

```
define_dum(shape(16, *, 'x'))
```

7.7.3 Ambiguity due to variable boundaries

There is a possible source of errors with variable field sizes (and types). Consider the example:

```
CAM_word alpha(1,1), beta(2);
enter_new(beta(21));
define_dum(alpha(0,*));
```

The second *enter* command will create a CAM word whose first byte is 0, since the two byte hexadecimal representation of 21 is 00 15. Thus the comparison will address a non-alpha CAM word, which may not be intended.

This may be resolved by applying the convention that a common portion of each CAM word contains an identifier to ensure that only the relevant part of memory can be addressed; thus the CAM is subdivided into sections of common field boundary identifiers. This could be achieved in the compiler by assigning an identifier associated with the declared CAM_word identifier. Access would thence be only through patterns with this byte implicitly maintained by the compiler. Alternatively, the programmer could be left to programme with these conventions himself.

7.7.4 Variable size with implicit ordering

If the fields are not distinguished by their relative textual position, then it is the fields themselves which must be declared and specified. A field has both size and offset within the CAM word, which is equivalent to offset of first and last byte. In the latter style, we might have declarations of the form:

```
field set_id(1), cost(2,5)
```

which declares two fields: *set_id* which is the first byte in a CAM word, and *cost* which is four bytes in length starting at byte two. Again, the use of numbers lacks the aspect of a high level language; but the programmer should be allowed a free choice in allocating the CAM word.

Addressing

The <pattern> for the CAM is now expressed by a list of specified fields and their values, thus:

```
define_dee(set_id = 'V', cost = 3)
```

The order in which they are specified is no longer relevant, nor is the wild card. The CAM word type is specified by the fields of which it is composed, so the CAM_word declaration is redundant.

Assignment

The bag naming convention can be extended to allow the manipulation of the *dum/dee* variables as compiler maintained CPU registers. Conceptually, *dee* and *dum* are the currently addressed members of their respective bags, and access and updates performed upon dee and dum are effective upon the addressed memory location. The compiler will optimize this access by reading the CAM location into CPU memory on the first access and, if an update has occurred, rewriting only before another member is selected or a

new bag is defined.

However, this declaration syntax is not sufficient since it contains no statement of the field's type. This was not relevant to the discussion with fixed fields since they each inherited the typing of the variable to which it was assigned, or of the expression assigned of it. In this case, the field may be accessed without a clear typing inference. Thus the field declaration must contain a specification of the associated type:

```
field set_id(1)char, cost(2,5)int
```

Reading the value of fields from memory is achieved by specifying the bag and the field identifier. Thus an expression of the form:

```
dum.cost
```

refers to the cost field in the current n-tuple of the dum bag.

Writing to memory may also be achieved in a similar syntax, specifying the bag and the field on the left hand side of an assignment statement:

```
dum.cost = dee.cost + 1;
```

using the obvious convention that only the specified field is altered by such an assignment.

The *enter* command, is retained for writing CAM_words from one bag to another. The writing of a new word to CAM can not be accomplished by extending the bag variable concept to the empty tag group. The difference is that the assignment to a *dum/dee* bag is made by the compiler before the "current" word changes, but with the empty tag group the word remains "current" until it is written. Specifically the compiler must know when a new word has been defined so that the word may be written to memory. If new words were created by successive *new.field* assignments, there would

have to be an explicit statement in the high level language to designate when the word was assembled. The alternative is to retain the *enter* command to apply necessarily to the *new* bag and optionally to the *dum/dee* bags, thus:

```
enter_new(cost = dee.cost, set_id = 'B')
```

With this field specification, the Traversal example becomes:

```
Procedure Traversal(start_node) char start_node;
 {char selected_node;
  enter_new(set_id = 'visited', item = start_node);
  enter_new(set_id = 'to_do', item = start_node);
  while (define_dum(set_id = 'to_do'))
   {selected_node = dum.item;
    remove_dum();
    foreach_dee(set_id = 'edge', tail = selected_node)
     {if (!define_dum(set_id = 'visited', item = dee.head)
       {enter_new(set_id = 'visited', item = dee.head);
        enter_new(set_id = 'to_do', item = dee.head);}}}}
```

7.8 The type 'cword'

The final field specification syntax is reminiscent of the manipulation of fields within records as found in some conventional high level languages, and it seems natural to provide the programmer with the syntax to construct such RAM records and to interface these with the CAM equivalents. For instance, this would allow the software emulation of a read-only bag by reading a bag into a record stack. To assist this, we introduce a new type of RAM variable: the *cword*, which is of the same length as a CAM word, and which is accessed by the same field references and syntax as the pseudo *dum/dee* words.

A cword is location addressed RAM. In the same manner as other RAM variables, it has a declared identifier, thus:

```
cword cw1, cw2;
```

declares two cwords variables: cw1 and cw2, in RAM. Cwords are manipulated according to fields:

```
cw1.set_id = 'A';
cw2.cost = cw1.cost + 99;
```

in the same manner as CAM words, and *dum/dee* may be viewed as a special example of a cword. Cwords may be assigned to each other in their entirity through a statement of the form:

```
<cword> = <cword>;
```

No arithmetic operators are defined upon a cword.

There are natural extensions to the syntax which allow for the direct transfer of data between CAM and RAM. A cword may be written to CAM through the *enter* command family, or by assignment to the pseudo variables *dum*, *dee* or *new*; and reading from CAM is the assignment of dum or dee, thus:

```
enter_dum(cw1);
new = cw2;
cw1 = dum;
```

7.9 An Example

Finally, let us consider an example programme written in this syntax. Perhaps the best known graph theoretic algorithm is that of Dijkstra for the solution to the single source, shortest paths problem: given a costed graph and a single node, find the length (cost) of the shortest path to each of the other nodes.

7.9.1 The algorithm

Let the initial node be i, and the graph be $G = (V, E)$ where V is the set of nodes and E is the set of (costed) edges. The algorithm is based on induction. Suppose that there are two subsets of V:

A: the nodes to which the length of the shortest path is known and is no longer than any path to nodes not in A

B: nodes which are adjacent to, but not members of, A

and assume that the length of the shortest path from i to each node in B, *using only nodes in* A, is also known.

Let j be a node in B which has the shortest such path. Then j can be transferred to A, and the knowledge of the members of B can be updated to consider the new paths which include j.

This task is simplified by the following reasoning. The shortest path to any other node (k) in A does not contain j, since then there would be a path to j which was shorter than the shortest path to k, and this contradicts the inductive hypothesis regarding A. The shortest path to the nodes not in A (using only nodes now in A) either does not contain j, and so is unchanged; or contains j

as the penultimate node – thus only nodes which are adjacent to j (and not in A) need to be considered when updating.

The basis for the induction is that i is a member of A, and that B consists of the nodes which are adjacent to A whose 'path' length is known from the cost of that edge.

7.9.2 The encoding

The encoding (see page 157) of the algorithm is based on the use of two n-tuples, the costed edge representation of the graph found in chapter 1:

[Edge_tuple | from_node | to_node | cost]

and workspace of the form:

[Work_tuple | node | cheapest_known_cost | set_flag]

where the value of the flag is used to denote whether the corresponding node in set B (if FALSE) or in set A (if TRUE).

The initial node is placed in set B, and is made the selected node for transfer to set A. The main block has three sections:

- transfer selected node from B to A
- update set B by considering paths through the selected node to its adjacent nodes
- select a new node from the members of B, or quit if B is empty.

```
/* Dijkstra's algorithm computes the cost of the shortest paths from a given
   node. This realization is based upon the costed edge representation:
            (E, from_node, to_node, cost)
   and a work space n-tuple of the form:
            (W, node, cheapest_known, set_flag) */

/* field declarations */
field tuple_type(1)char, node(2)char, flag(3)char, cheap(4,8)int;
field frnode(2)char, tonode(3)char, cost(4,8)int;

/* sample graph's specification */
char edgelist[][3] = {
        {'A','B',8}, {'A','G',7}, {'A','H',6), {'A','J',4}
        {'A','C',3}, {'C','D',3}, {'C','E',2), {'D','F',1}
        {'E','F',1}, {'E','J',12}, {'H','I',9} };
int no_arcs = 11;

main()
{int i, minnode, mincost, sum;

/* create the costed edge list */
for (i=0; i<no_arcs; i++)
    {enter_new(tuple_type = 'E', frnode = edgelist[i][0],
                    tonode = edgelist[i][1], cost = edgelist[i][2]);
     /* and in reverse for bi-directional edges */
     enter_new(tuple_type = 'E', frnode = edgelist[i][1],
                    tonode = edgelist[i][0], cost = edgelist[i][2]);}}

/* initiate the data structure with the starting node */
minnode = 'A';
mincost = 0;
enter_new(tuple_type = 'W', node = minnode, cheap = 0, flag = FALSE);
```

```
/* the main block: repeat until a break */
while (1)
  {/* flag current node, so transferring it to set A */
   define_dum(tuple_type = 'W', node = minnode)
     flag.dum = TRUE;

   /* for each node adjacent to the current node –
     update its shortest known path */
   foreach_dum(tuple_type = 'E', frnode = minnode)
     {/* compute the cost of a path using the current node */
      sum = mincost + dum.cost;
      /* and compare this with the best known route */
      define_dee(tuple_type = 'W', node = dum.tonode)
        {/* if the entry exists and ...
           ... if the new route is cheaper */
         if (sum < dee.cheap)
           /* update the value */
           dee.cheap = sum;}
      /* if there is no entry, create one */
      else enter_new(tuple_type = 'W', node = dum.tonode,
                     cheap = sum, flag = FALSE);}

/* of the nodes in set B, select the one with
  the shortest known path */
define_dum(tuple_type = 'W', flag = FALSE)
  {minnode = dum.node; mincost = dum.cost;
   while (next_dum)
     {if (mincost > dum.cheap)
        {minnode = dum.node; mincost = dum.cheap;}}}
else /* if no more nodes in set B */ break;}

/* print out results and remove the workspace */
fromeach_dum(tuple_type = 'W')
  printf("node %c at cost %d\n", dum.node, dum.cheap);}
```

7.9.3 Assessment

The use made of the subset B, as a dynamically constructed set of nodes adjacent to A, is unusual and replaces the use of $(V - A)$ with which the algorithm is usually described; it is more efficient but it is not dependent upon CAM although the perspective from which it was reached was engendered by considering the CAM implementation. The main points, however, are that the whole data structure has a direct correspondence to CAM without the need for abstractions over the hardware form, and that this correspondence may be expressed clearly and simply in the syntactic constructs which have been developed in this chapter.

Chapter 8

CONCLUDING REMARKS

The implicit strategy of this thesis has been the continual interaction between the projected software usage and the hardware design, and the success of this approach can be seen in the ease with which the novel hardware was rendered accessible to the general programmer. The initial consideration of the software's needs lead to the inclusion of multiple tag groups, this gave rise to the hardware organization for hardwired garbage collection, which in turn shaped the functional characteristics of the component and so the assembler forms. Unlike the examples of chapter 5 which showed the short-comings of an imbalanced design effort, the novel hardware of this thesis has been fully integrated into a complete programming environment.

The *Triplet* architecture forms a general purpose computer in the same sense as the von Neumann model and exemplifies CAM's immediate potential as a general programming tool. The main significance of the syntactic extensions is that they render the innovative architecture and hardware accessible without fundamentally altering the structure of conventional programming languages, and this will facilitate the general acceptance of the novel architecture. The effect is the same as presenting the high level programmer with an abstract data type and advocating its usage on the grounds of efficient implementation, and this data type allows the programmer to construct data structures and algorithms in a manner which was previously impractical on a RAM only machine.

Content Addressability allows certain abstract data types to be implemented with greater clarity and efficiency; and, in the context of the hardware implementation of data types, CAM may be used in system design to engender new computer architectures ranging from simple interpretors, to multiprocessor implementations of novel programming languages: it is evident that CAM will assist the progress of many disparate fields of computer science. Furthermore, Content Addressability will have a profound effect on both computing theory and practice since it transfers a specific type of processing to beyond the "von Neumann bottle neck". This alone will change the perspectives for general software techniques and computational complexity analysis. For instance, the "current state" of a computation could no longer be characterized as a point in a multi-dimensional vector space, but rather must be viewed as the collection of elements in the memory *bag*.

The success of the prototype components validated the CAM's design, and the programming language extensions, coupled with the discussion of possible architectural developments, demonstrated the design's potential. The need is for a large scale component, and the proposal for wafer scale integration shows that not only is this design suitable for a commercial memory project but also that it would form an excellent basis for work on wafer scale integration *per se*. Content Addressable Memory has a future – which the design proposed in this thesis could effect.

The main omission of this thesis is in the expansion of the basic CAM ICs into the large scale memories which will be necessary for the successful introduction of CAM orientated systems. Much work is needed on the wafer scale integration proposals and to determine the optimum decomposition of the memory hierarchy[68]. The multi-write capability of CAM, which was mentioned in chapter 1, was not included in the prototype component design despite the ease with which this could have been done. The intention was to implement Content Addressability in its 'purest' form without including enhancements which appeared attractive due to the characteristics of the actual implementation. Similarly other design strategies such as the use of distributed units of word-serial bit-parallel comparison logic (attractive due to the possible use of conventions RAMs for data storage) were not fully investigated as the intention was to emphasize the direct link between the abstract, software, and hardware forms.

The main point of the work reported in this thesis is that CAM is a viable and useful medium for data structure and algorithmic design, much work remains to done in quantifying the relative advantages of CAM and RAM implemetations for each application. Further work, too, is needed in developing the concept of *variformity* introduced in chapter 6 which embodies the form of Content Addressability without pointing clearly to application areas beyond the most simple use of field duality. The use of CAM in system design should be investigated by isolating those system 'blocks' which commonly appear and which bear a close affinity to CAM. Finally, the most exciting prospect is an investigation of programming languages which are based upon the charactoristics of CAM alone – and not hampered by the developments of the past forty years. The question should be asked: how would programming techniques have evolved if CAM rather than RAM had been the memory medium implemented in the EDVAC and its descendents?

REFERENCES

[1] J von Neumann, "First Draft of a Report on the EDVAC," in *The Origins of Digital Computers*, ed. B Randell, pp. 383-397, Springer-Verlag, 1982.

[2] A W Burks, H Goldstine, and J von Neumann, "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument," in *The Origins of Digital Computers*, ed. B Randell, pp. 399-413, Springer-Verlag, 1982.

[3] Benrooz Parhami, "Associative Memories and Processors: An Overview and Selected Bibliography," *Proc IEEE*, vol. 61, no. 6, pp. 722-730, June 1973.

[4] T Kohonen, *Content-Addressable Memories*, Springer-Verlag, 1980.

[5] Caxton C Foster, *Content Addressable Parallel Processors*, Van Nostrand Reinhold Company, 1976.

[6] K E Batcher, "Massive Parallel Processor," *IEEE Trans Comp*, p. 936, 1980.

[7] M Edelberg and L Robert Schissler, "Intelligent Memory," *AFIPS Conf*, p. 393, 1976.

[8] G L Livovski, "Architectural features of Cassm," *ACM SIGARCH*, p. 31, April 1979.

[9] S A Schuster, "RAP.2 - An Associative Processor for Data Bases," *ACM SIGARCH*, p. 52, April 1979.

[10] A Weinberger, "The Hybrid Associative Memory Concept," *Computer Design*, vol. 10, pp. 77-85, Jan 1971.

[11] J G Bonar and S P Levitan, "Real-time Lisp using Content Addressable Memory," *Proc Int Conf on Parallel Processing*, Aug 1981.

[12] Stuart J Adams, Mary Jane Irwin, and Robert M Owens, "A Parallel, General Purpose CAM Architecture," *Proceedings of the Fourth MIT Conference in Advanced Research in VLSI*, pp. 51-72, MIT Press, Boston, 1986.

[13] Hiroshi Kodata, Jiro Miyake, Yoshito Nishimichi, Hitoshi Kudo, and Keiichi Kagawa, "An 8Kb Content-Addressable and Reentrant Memory," *IEEE ISSCC*, pp. 42–3, Feb 1985. Matsushita Electric Industrial Co., Ltd.

[14] David Rees, Computer Science Dept., University of Edinburgh, Scotland, 1982. Personal Communication

[15] C V Ramamoorthy, James L Turner, and Benjamin W Wah, "A Design of a Fast Cellular Associative Memory for Ordered Retrieval," *IEEE Trans Comp*, vol. C-27, no. 9, pp. 800–815, Sept 1978.

[16] R M Lea, "Design of a High Speed MOS associative Memory," *Electronics Letters*, vol. 8, no. 15, p. 391, July 1972.

[17] J L Mundy, J F Burgess, R E Joynson, and C Neugebauer, *IEEE JSSC*, vol. SC-7, no. 5, pp. 364-369, Oct 1972.

[18] R M Lea, "Low-cost High-speed Associative Memory," *IEEE JSSC*, vol. SC-10, no. 3, pp. 179-181, June 1975.

[19] J P Wade and C G Sodini, "Dynamic Cross-Coupled Bitline Content Addressable Memory Cell for High Density Arrays," Dept Elec Eng and Comp Sci, MIT, Cambridge MA, 1985.

[20] Alfred V Aho, John E Hopcroft, and Jeffrey D Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1983.

[21] W D Manner, "Hash Table Methods," *Computing Surveys*, vol. 7, no. 1, pp. 5-20, ACM, 1975.

[22] A D Falkoff, "Algorithms for parallel search memories," *Journal ACM*, vol. 9, pp. 488-511, Oct 1962.

[23] C H Sequin and D A Patterson, "Design and Implementation of RISC 1," in *VLSI architecture*, ed. B Randell, pp. 276-297, Prentice-Hall, 1983.

[24] R W Sherburne, M G H Katevenis, D A Patterson, and C H Sequin, "A 32–bit Microprocessor with a Large Register File," *IEEE JSSC*, vol. SC–19, no. 5, pp. 682-689, Oct 1984.

[25] Takeshi Ogura, Shin-Ichiro Yamada, and Tadanobu Nikaido, "A 4–kbit Associative Memory LSI," *IEEE J Solid–State Circuits*, vol. SC–20, no. 6, pp. 1277–1281, Dec 1985.

[26] S E Schuster, "Multiple Word/Bit Line Redundancy for Semiconductor Memories," *IEEE JSSC*, vol. SC–13, pp. 698-703, Oct 1978.

[27] Ivor Catt, "1985 - the year of the first pre-production working wafers," *Silicon Design*, vol. 3, no. 1, pp. 8-9, Jan 1986.

[28] J F McDonald, E H Rogers, K Rose, and A J Steckl, "The trials of wafer–scale integration," *Spectrum*, vol. 21, no. 10, pp. 32-39, IEEE, Oct 1984.

[29] G F Taylor, B J Donlan, J F McDonald, A S Bergendahl, and R H Steinvorth, "The Wafer Transmission Module - Wafer Scale Integration Packaging," *IEEE Custom Integrated Circuits Conf.*, pp. 55-57, 1985.

[30] R N Mayo, J K Ousterhout, and W S Scott, *1983 VLSI Tools: Selected Works by the Original Artists*, Comp Sci Div, EECS Dept, University of California, Berkeley, 1983.

[31] UN/NW VLSI Consortium, *VLSI Design Tools Reference Manual*, Dept Comp Sci, University of Washington, Seattle, Washington, Oct 1984. Release 2.1

[32] Hewlett-Packard Company, *HPSPICE Users Guide*, Cupertino, California, Aug 1983.

[33] Tektronix Inc, "91DVV - The DAS VLSI Verification System," *91DVVI Operation Manual*, Tektronix Inc. Software Release 1

[34] M E Lesk and E Schmidt, *Lex - A Lexical Analyzer Generator*, Bell Laboratories, Murray Hill, New Jersey 07974.

[35] A Wolinsky, "A simple proof of Lewin's Ordered-Retrieval Theorem for Associative Memory," *Comm ACM*, vol. 11, no. 7, July 1968.

[36] Y Chu and M Abrams, "Programming Languages and Direct-Execution Computer Architectures," *Computer*, vol. 14, no. 7, pp. 22-33, July 1981.

[37] David Shaw, A Hierarchical Associative Architecture for the Parallel Evaluation of Relational Algebraic Database Primatives, Artificial Intelligence Laboratory Stanford University, Oct 1979.

[38] J Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," *Comm ACM*, vol. 21, no. 8, pp. 613-641, Aug 1978.

[39] J T Schwartz, *On Programming: An Interim Report on the SETL Project*, 2, New York University, Oct 1973.

[40] John G Sanderson, *A Relational Theory of Computing*, Lecture Notes in Computer Science, Springer-Verlag, 1980.

[41] J Darlington and M Reeve, "Alice," *Proc ACM Conference on Functional Programming Languages and Computer Architecture*, pp. 65-74, 1981.

[42] M Reeve, "The Alice Prototype," Internal Report, Imperial College, London, Nov 1983.

[43] J R Gurd, C C Kirkham, and I Watson, "The Manchester Prototype Dataflow Computer," *Comm ACM*, vol. 28, no. 1, pp. 35-52, Jan 1985.

[44] C C Kirkham, "The Manchester Prototype Dataflow System," Basic Programming Manual, 4th edition, Nov 1983.

[45] J R Gurd, *Personal Communication*, April 1985.

[46] Martin Rem, *Associons and the Closure Statement*, Mathematisch Centrum, Amsterdam, 1976.

[47] Martin Rem, "Associons: A Program Notation with Tuples instead of Variables," *ACM Trans Prog Lang and Systems*, vol. 3, no. 3, pp. 251-262, July 1981.

[48] Martin Rem, "The Closure Statement: A Programming Language Construct Allowing Ultraconcurrent Execution," *J. ACM*, vol. 28, no. 2, pp. 251-262, April 1981.

[49] R M Russel, "The Cray–1 Computer System," *Communications of the ACM*, vol. 21, pp. 64-72, 1978.

[50] A R Newton and A L Sangiovani-Vintcentelli, "Computer-Aided Design for VLSI Circuits," *Computer*, vol. 19, no. 4, pp. 19-37, IEEE, April 1986.

[51] P M Kogge, *The Architecture of Pipelined Computers*, McGraw-Hill, 1981.

[52] R H Perrott, "A Language for Array and Vector Processors," *ACM TOPLAS*, vol. 1, no. 2, pp. 177-195, Oct 1979.

[53] L Uhr, *Algorithm-Structured Computer Arrays and Networks*, Academic Press, 1984.

[54] K Preston, "Progress in image processing languages," in *Computing Structures for Image Processing*, ed. M J B Duff, pp. 195-211, Academic Press, 1983.

[55] C A R Hoare, "Monitor: an operating system structuring concept," *Comm ACM*, Oct 1974.

[56] N Wirth, "Modula: a language for modular multiprogramming," *Software Practice and Experience*, vol. 7, no. 1, pp. 3-35, 1977.

[57] Steven R Vegdahl, "A Survey of Proposed Architectures for the Execution of Functional Languages," *IEEE Trans Comp*, vol. C-33, no. 12, pp. 1050-1071, Dec 1984.

[58] L Snyder, "Parallel Programming and the Poker Programming Environment," *Computer*, vol. 17, no. 7, pp. 27-37, IEEE, July 1984.

[59] E R Berklekamp, "Bit-Serial Reed-Solomon Encoders," *IEEE Trans Inf Tech*, vol. IT28, pp. 869-874, Nov 1982.

[60] W J Daly and R E Bryant, "A Hardware Architecture for Switch-level Simulation," *IEEE Trans Comp-Aid Design*, vol. CAD-4, no. 3, pp. 239-250, July 1985.

[61] John L Hennessy, "VLSI Processor Architecture," *IEEE Trans Comp*, vol. C–33, no. 12, pp. 1221-1246, Dec 1984.

[62] W N Johnson, W V Herrick, and W J Grundmann, "A VLSI VAX Chip Set," *IEEE JSSC*, vol. SC–19, no. 5, pp. 663–674, Oct 1984.

[63] R M Supnik, "MicroVAX 32, A 32 Bit Microprocessor," *IEEE JSSC*, vol. SC–19, no. 5, pp. 675-681, Oct 1984.

[64] J Hennessy, N Jouppi, S Przybylski, C Rowen, and T Gross, "Design of a high performance VLSI processor," *Proc 3rd Caltech Conf VLSI*, pp. 33-54, Mar 1983.

[65] R P Colwell, C Y Hitchcock III, E D Jensen, H M Brinkley Sprunt, and C P Kollar, "Computers, Complexity, and Controversy," *Computer*, vol. 19, no. 9, pp. 8-19, Sept 1985.

[66] B W Kernighan and D M Ritchie, *The C Programming Language*, Prentice-Hall, 1978.

[67] S C Johnson, *Yacc: Yet Another Compiler-Compiler*, Bell Laboratories, Murray Hill, New Jersey 07974.

[68] C A Mead and M Rem, "Cost and Performance of VLSI Computing Structures" *IEEE JSSC*, vol. SC–14, no. 2, pp. 455-462, April 1979.