MAPLE:

Multiprocessor APL machinE

by

Warren S. Snyder

B.Sc., Simon Fraser University, 1977

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in the department

of

Computing Science

Warren S. Snyder

SIMON FRASER UNIVERSITY

February 1982

Name: Warren S. Snyder

Degree: Master of Science

Title of Thesis: MAPLE- A Multiprocessor APL machinE

Examining Commitee:

Chairperson: W.S. Luk

E.M.Edwards
Senior Supervisor

Richard F. Hobson

Thomas W. Calvert

M.A. Jenkins
External Examiner
Department of Computing & Information science
Queen's University
Kingston, Ontario

Date Approved: 1982 Feb 26

An architecture was investigated which allows a high degree of concurrent processing during direct execution of APL statements. It consists of four modules called the Execution Unit (EXU), Data Manipulation Unit (DMU), Arithmetic-Logic Unit (ALU), and Input-Output Unit (IOU). Each module represents a subset of the processing needed to synthesize a complete APL environment. These modules communicate in a multiprocessing network.

Research was concentrated on the DMU, which implements an APL workspace and all array storage and access activities. A configuration was achieved that greatly minimizes the number of main memory accesses for all APL statements. For those operations which require array accesses, performances equal to main memory speeds can be achieved.

Table of Contents

Approval - - - - - - - ii

Abstract - - - - - - - iii

List of Tables - - - - - - v

List of Figures - - - - - - vi

Introduction - - - - - - - 1

Chapter 1
- 1.1 The APL Language - - - 4
- 1.2 APL Problems and Corrections - 7
- 1.3 APL Primitive Classes - - 19

Chapter 2
- 2.1 APL Systems - - - 23
- 2.2 Other Array Processing Systems - 29
- 2.3 Features Summary - - - 33
- 2.4 Array Processing Theory - 36

Chapter 3
- 3.1 Architectural Justification - 43
- 3.2 MAPLE's Architecture - - 48

Chapter 4 The Workspace - - - 51
- 4.1 Memory Management - - - 54
- 4.2 Data Types - - - 74
- 4.3 Workspace Structures - - 92

Chapter 5 DMU - - - 97
- 5.1 DMU Architecture - - - 99
- 5.2 DMU Instructions and Operation - 111

Chapter 6
- 6.1 Bus Arbitration - - - 120
- 6.2 Unit Specifications - - 125

Chapter 7 Architecture Evaluation - - 135
- 7.1 DMU Performance - - - 137
- 7.2 Performance of the EXU & ALU - 141
- 7.3 Summary - - - - 144

Appendix 1 APL Syntax - - - 146

Appendix 2 Array Access ALgorithms - - 150

Appendix 3 Execution Resource Usages - 159

Appendix 4 DMU Instructions - - - 163

Bibliography - - - - - - 167

Table		page
1.2.1	Examples of Lists	- 9
1.2.2	Examples of The Index Primitive	- 10
1.2.3	Extended Assignment	- 12
1.2.4	Examples of Extended Assignment	- 14
1.2.5	Extended Scalar Conformability	- 17
1.3.1	APL Primitive Subdivisions	- 20
1.3.2	Four Groups of APL Primitives	- 22
2.4.1	Principle of SS1 Descriptors	- 38
2.4.2	Examples of Drag Along	- 42
4.1.3	Properties of Hole Table Maintenance	- 63
5.2.1	DMU APL Instructions	- 112
6.2.1	ALU APL Instructions	- 129
6.2.2	IOU APL Operations	- 132
7.2.1	ALU Operational Times	- 143
A.1.1	APL Expression Syntax	- 147
A.1.2	Proposed APL Syntax	- 148
A.1.3	Examples of APL Syntax	- 149

Figure		page
2.2.1	PEPE	- 31
3.1.1	STARLET System	- 46
3.2.1	Bus Overview	- 50
4.1.1	Translational Memory Management	- 57
4.1.2	Hole Table	- 61
4.1.4	Virtual Address	- 72
4.1.5	Relocation Vector	- 73
4.2.1	Data Hierarchy	- 82
4.2.2	Type-Rank Header	- 85
4.2.3	Component Code	- 87
4.2.4	Size Code Examples	- 88
4.2.5	Descriptor Layout	- 90
4.2.6	Descriptor Examples	- 91
4.3.1	Array Reference Table	- 94
4.3.2	ST and SAV	- 96
5.1.1	Data Manipulation Unit	- 100
5.1.2	OMU's Structure	- 102
5.1.3	MMU's Local Store	- 106
5.1.4	Memory Addressing	- 108
6.1.1	Bus Signals	- 123

At present APL and LISP are the two main languages which provide powerful array processing primitive instructions. Both languages have a strong theoretical basis and a long history of implementation. APL was designed to be consistent with existing vector and tensor theory while LISP is based on Lambda Calculus [MIC73]. There have been tendencies to extend both APL and LISP to remove the inadequacies of each and even to combinine the two [JEN80].

The primary objective of this thesis is to reflect on the design of a memory architecture for a high performance multi-processor APL computer. This objective, however, requires some consideration of the structure of APL, some extensions to it, and the overall architecture of a machine to execute the language efficiently. These matters will be considered first as they influence the memory architecture.

The results of the investigations will be a hypothetical machine called "MAPLE" which should be manufacturable using current microelectronic engineering practices. The following is the set of objectives for MAPLE's design.

Objectives

-Define the language to be implemented and determine the nature of the tasks to be performed in its execution.

-Outline a machine architecture which can efficiently execute the language. An architecture which exploits parallel processing where possible, and is modular for ease of implementation.

-Define (in detail) the requirements for the memory architecture. This involves investigation of the workspace organization and array storage/access methods.

This thesis will address the above objectives with emphasis on the design of the memory architecture and workspace structure. Indepth investigation of the remaining aspects of MAPLE's design will be left for future research.

This thesis is divided into four areas of discussion. Chapter 1 covers the first area, discussing the properties, problems, and extensions of the APL language. In chapter 2 the State of the Art of APL and array processing systems will be discussed. Then in chapters 3 through 6 an implementation of MAPLE will be discussed. This involves descriptions of both its architecture and engineering. It is in this third area that MAPLE's main priorities will be covered. Chapter 7 examines the performance expected from the architecture along with some suggestions for future improvements with a summary of how well the objectives of this thesis were met.

The APL language provides a set of array processing operations suitable for a wide class of applications. K.E.Iverson [IVE62] is credited with the language's invention, proposing it as a mathematical notation for describing array theory as it applies to Tensor Algebra. The language's extensions and implementations are credited to numerous others [FAL64],[BRE68].

APL obtained worldwide recognition when the first machine implementations were successful during the late 1960's. However, its availablity was and is not now widespread partly due to the vast complexities in implementing the language on existing hardware and the lack of skilled personnel to do so.

Appendix 1 contains a partial description of the syntax for APL expressions (which is the basic procedural unit of the language). For a more complete description of APL's syntax and execution see [FAL79].

It is apparent from appendix 1 that the expression syntax is not complex. However, what is not obvious is the reason the language is almost always interpreted and not compiled. This can be explained by two facts: Firstly, interpretation is not a complex task, easing implementation and allowing a highly interactive environment for the user; Secondly, the dynamic nature of the data makes compilation very difficult (there have been a few partially successful incremental compilers for APL [JOHN79]).

The advantages of APL are far more numerous than its disadvantages. Its strengths arise from its powerful array manipulation primitives which allow concise statements about a problem in a consistent manner.

The problems of any APL system can be considered in two separate classes. The first class results from the implementation , i.e. how the logical and language concepts were transferred to an actual machine. The implementation has more effect than any other feature in reducing the usefulness of an APL system. The implementation affects the efficiency of the language in carrying out the instructions contained in it. However, all implementation problems can be cured by the suitable choice of machine architecture.

The second class of problems result from the language itself. The current definition of the language has many restrictions on operators, primitives, and user defined functions/arrays ([ABR75] describes some aspects of what is wrong with APL). Nearly all of these restrictions are the result of defining a useable system within the constraints of the machine technology at definition time.

An implementation effort may be divided into three phases, Workspace Management, Array Reference/ Operation Algorithms, and User/ Operation System Interfaces. Workspace Management includes dynamic changes in size and attributes of arrays during execution and maintaining associations between accessible arrays and their names. The specific problems of Workspace Management will be addressed in chapter 4.

The problem of referencing arrays in the various modes and manners which APL requires is a very complex task. These involve algorithms to select subsets of data from arrays in an almost arbitrary fashion. These algorithms often imply a large amount of data movement. If, however, it can be recognized when such movements are not required, then significant improvements can be realized. It will be the subject of chapter 5, on the DMU, to suggest an architecture for the efficient execution of these algorithms. This phase of an implementation covers what is generally called the execution (interpretation) of the language.

The remaining problems lie in interfacing the user to the system. This requires editing provisions for APL programs and data and a proper interactive environment. Some of these problems will be addressed in chapter 6 on the IOU.

The problems of the APL language and its restrictions will be discussed first and the implementation left for the later chapters. All APL arrays must be homogeneous. That is, all elements must be chosen from either character or numerical scalars. Certain applications, however, (such as those LISP is often used for) require that elements of an array be chosen from a set of arrays (an example is, lines of a paragraph which are varying length character vectors grouped together). This requires the concept of a "Generalized Array" where each element of an array can be any data object, allowing the expression of arrays of arrays.

A substantial effort in extending APL to include this concept has been addressed by E.Edwards [EDW73], R.Murray [MUR73], H.Haegi [HAE76], and T.More [MOR79] (also [VAS73], [GHA76], [GUL76], [JEN78], and [PIE79]). The extension proposed here follows that which E.Edwards proposed. The details of this extension are still controversial in the APL community and will be left somewhat open for future changes, but a need for this extension has been shown to be genuine.

This extension to generalized arrays can be summarized by two concepts. The first is that of a "List" scalar, which is the scalar element for all generalized arrays. The second concept is a set of mechanisms which allow the user to transform any array into a list scalar or any list scalar to an array. These are the dual processes of "Imbed" and "Expose" respectively. c.f. Table 1.2.1 for examples.

The rules for indexing are given in appendix 1 under the rules of expression syntax (subexpression Iexpression). Though the concept of indexing is well defined in APL it lacks the symmetrical form that all the other primitives have. This will be rectified by the introduction of the dyadic selection primitive "Index".

This primitive is described in the CDC*APL system [CDC*APL]. Basically, it is a dyadic function where the right argument is the array to index, while the left argument is a "List" of indices. This index list has as many components as the array has axes, and the arrays imbedded within this list are indices within these axes. Table 1.2.2 illustrates this primitive.

TABLE 1.2.1

EXAMPLES OF LISTS

```
X←'HELLO' ⋄⎕IO←0
Y←'THIS IS A LONGER SENTENCE'
Z←2 5⍴⍳10

L←(⊂X),(⊂Y),⊂Z
⍝CREATE THE LIST L

∧/, Y=⊃L[1]
3=⍴L
L←⊂L
1=⍴,L
∧/, Z=⊃(⊃L)[2]
N←3 3⍴⊃L
∧/, (⊃N[2;0])='HELLO'
```

TABLE 1.2.2
EXAMPLES OF THE INDEX PRIMITIVE

```
X[Y;Z]          ⍝CONVENTIONALLY
((⊂Y),⊂Z)⌷X     ⍝EQUIVALENTLY

I←(⊂Y),⊂Z
I⌷X             ⍝ALTERNATIVELY
X←2 3 3 5⍴Z     ⍝CONSIDER THE ARRAY X
X[;;1;Y]        ⍝ONE ACCESS OF X
X[Y;1;;]        ⍝ANOTHER SIMILAR ACCESS
E←⊂0⍴⊂0         ⍝AN EMPTY LIST
I←E,E,(⊂1),⊂Y
I⌷X             ⍝FIRST ACCESS
(⍬I)⌷X          ⍝SECOND ACCESS
```

It is not my intention to remove the syntax for the index subexpression, but rather to augment it with the index primitive. Since all former cases of indexing can be reduced to expressions involving the "Index" primitive, an actual machine implementation may translate the former syntax into equivalent expressions with "Index". However, this extension should be made available to the user as complex indexing expressions can therefore be assigned to variables for later use, something which is not possible with present APL.

In the introduction of the "Index" primitve it must be recognized that indexing is a true selection process and as such must be made to have the same syntax form as the other selection primitives. However, within the current APL, indexing is given a special property in that it can be involved in an assignment process. Table 1.2.3 shows the operation of assignment as currently supported along with the proposed extension to assignment.

Assignment's extension can be best described as allowing the assignment of one array into another array of similar shape and type, if the left argument represents a subset of a named object. The old definition restricted this to only a named object or a subset of a named object generated by indexing. This more general principle was recognized by a group at CDC when they implemented CDC*APL [CDC*APL], so they proposed and implemented part of the definition of table 1.2.3.

TABLE 1.2.3

EXTENDED ASSIGNMENT

```
PREVIOUS DEFINITION:  NOBJECT←EXPR
                      NOBJECT[IEXPR]←EXPR
EXPR:                 VALID APL EXPRESSION
IEXPR:                INDEXING EXPRESSION
NOBJECT:              A NAMED OBJECT

⍝AFTER ASSIGNMENT THE TERM TO LEFT OF ASSIGNMENT (LT)
⍝AND TERM TO RIGHT (RT) HAVE THE FOLLOWING PROPERTY
                      ∧/,LT=RT

PROPOSED DEFINITION:  SE (SELECTION EXPRESSION)
                      SE←EXPRESSION

SE:                   NOBJECT
                      ⌹SE
                      EXPR⌹SE
                      ⌽SE
                      EXPR⌽SE
                      EXPRρSE   ⍝NO CYCLES
                      EXPR/SE
                      EXPR∩SE
                      EXPR↑SE   ⍝NO OVERTAKE
                      EXPR↓SE
                      ,SE

⍝AND THE FOLLOWING SPECIAL CASE
                      EXPR\SE
⍝ALL THE ABOVE EXCEPT THE LAST FOLLOW (∧/,SE=EXPR)
⍝AFTER ASSIGNMENT .THE LAST DOES UNDER THE FOLLOWING RULE
⍝(EXPR\SE)←→(~EXPR)/EXPR\SE
```

The nature of the selection operations that were allowed in the CDC* proposal restricted the usable primitives to Transpose, Rotate, Take, Drop, Reshape, Index. That is, the pure selection primitives. The proposal to be made here is to also allow assignmemt into selection expression (either temporary or named objects) and to allow in addition the use of both compression and expansion as valid selection primitives. c.f table 1.2.4 . This extension was briefly introduced by E.Edwards in describing improvements to the APL language [EDW80].

A useful set of expansions to the APL language involves generalizations in the data types allowable, the first of which is the extension of the numerical scalars to the complex scalars. This allows the inclusion of the current reals and allows many primitives to produce complex results. Many of the primitives have natural extensions to the complex domain and have been treated by P.Penfield [PEN79].

While there is still debate concerning the extensions of some of the primitives to the complexes, one may choose at this time to trap their results to domain errors or some suitable value. It is important that this data type be allowable to extend the usefulness of APL to the scientific community.

TABLE 1.2.4

EXAMPLES OF EXTENDED ASSIGNMENT

```
A[(~T)/⍳1↑⍴A;]←B ⋄A←T⍀A
⍝INSERT NEW ROWS B, INTO MATRIX A, AS
⍝GIVEN BY T. WITH CONVENTIONAL ASSIGNMENT
⍝NOW WITH EXTENDED ASSIGNMENT
(T⍀A)←B

A[T/⍳1↑⍴A;]←B
⍝REPLACE ROW OF A BY ROWS B AS GIVEN BY T
⍝OR
(T⌿A)←B

G1[6+⍳6;3+⍳3]←G2
⍝AS OPOSED TO
(6 3↑6 3↓G1)←G2

(3 2↑B)←0⋄B←10 50⍴1
⍝INITIALIZE A MATRIX AND ITS SUBMATRIX
(1 1⍉I)←1⋄I←10 10⍴0
⍝CREATE IDENTITY MATRIX

G←(N,N)⍴' '
(1 1⍉G[X;Y])←'*'
⍝GRAPH PLOTTER
```

Another data type that needs expanding is character data. In almost all systems, except for advanced graphics, the set of characters is small and unalterable. That is, in Fortran, Pascal, and APL languages, the character sets are not user defineable. It will be the intention of this extension to APL to provide a high degree of flexibility in the definition of character data. These extensions will also allow the definition of other IO related data types such as speech or graphics.

An attempt to provide these types of extensions will be in allowing the user to access and modify the character set definition within the system. This involves the controlled assignment to the system object(s) which support the set of character scalars.

The most important extension to the APL language to be proposed here involves the manner in which arrays of dissimilar shapes are coerced into conformable shapes during the execution of dyadic "scalar" functions. Currently, two arrays are "conformable" if either, both have the same shape or one contains exactly one element. One element arrays are Reshaped into an array of the same shape as the other array (unless it has only one element in which case the array with the higher rank dictates shape).

This is an extremely useful concept. However,as Edwards (in a paper on simplifying APL concepts [EDW80]) and Breed a few years earlier [BRE71], point out, this principle should be extended to allow a more general coercion of arrays. This is the principle of "Extended Scalar Conformability". Its rules follow:

Any two arrays of the same rank are conformable IF (1) their shapes are identical OR (2) IF for the dimensions that differ, pairwise, one has dimension 1. The axes with dimension 1 are extended by replication along that direction to the dimension of the other array.

For arrays of rank differing by 1, the "smaller" array can have its rank extended by 1 at a location indexable by the index operator. Since the two arrays now have the same rank the first principle can be applied.

In the cases where the ranks differ by more than one the old principle of scalar conformability applies. c.f. Table 1.2.5 for examples.

TABLE 1.2.5

EXAMPLES OF EXTENDED SCALAR CONFORMABILITY

```
1+5 6ρι30          ⍝CURRENTLY ALLOWED
X←2 1 3ρ0
Y←2 4 3ρ1          ⍝TWO ARRAYS X Y

X∨Y                ⍝ONLY ALLOWABLE IF EXTENDED
                   ⍝SCALAR CONFORMABILITY
X←1 3ρ0
X+[0]Y             ⍝EXTENDS (ρX) TO 1 1 3
                   ⍝THEN REPLICATES X SO
                   ⍝THAT (ρX)=2 4 3

MARKS              ⍝ AN N,M MATRIX OF MARKS
WEIGHTS            ⍝VECTOR OF WEIGHTS (M=RHO)
MARKS÷[0]WEIGHTS   ⍝ASSOCIATED MARKS BY WEIGHTS
```

These principles will be taken a step further in the ability to coerce arrays. For the cases of ranks differing by two or more, the "smaller" array can be extended to the "larger" array's rank by the index operator, whose argument is a vector of intended axes. The default values will result in extending any array along its last coordinate axis, the same as the above proposal except more than one axis is involved.

This principle of Extended Scalar Conformability (E.S.C.) will dramatically influence the machine architecture of MAPLE. In Appendix 2 an algorithm for implementing E.S.C. is given.

From the primitives in their extended forms I shall investigate their classification into fundamental groups. The first group is obvious, being the "Numeric" primitives, c.f. Table 1.3.1. A primitive can be classified as numeric if the resultant array belongs to the complex numbers. This class contains the scalar numeric functions such as PLUS and TIMES. Also included are such primitives as MEMBERSHIP and INDEX-OF.

The numerical primitives all perform a transformation from one data type or value to another. There is a set of primitives which do not perform any such transformations. They are the "Selection" primitives, which simply modify the spatial arrangement of an array or else select a subset of the array, c.f. Table 1.3.1. These selection primitives are just those which were mentioned in section 1.2 on selection expressions.

The remaining few primitives can be clasified as either involving an IO function or some executive function of control or data association. These last few groups are called "IO", "Control", and "Other" primitives respectively. Under most circumstances the "operators" can be grouped as control operations or selection operations, but they will not be discussed here.

TABLE 1.3.1

APL PRIMITIVE SUBDIVISIONS

NUMERICAL PRIMITIVES

MONADIC

~	+	-	×	÷	⌈	⌊	*SCALAR*
|	○	*	⍟	?	!		*SCALAR*
⍳	⌹	⍋	⍒				*SPECIAL*

DYADIC

+	-	×	÷	⌈	⌊	|	*SCALAR*
○	*	⍟	=	≠	<	≤	*SCALAR*
>	≥	∧	∨	⍲	⍱	!	*SCALAR*
?	∈	⍳	⊤	⊥	⍕		*SPECIAL*

SELECTION PRIMITIVES

MONADIC

⍉	⊖	,	⊃	⊂

DYADIC

⍴	/	\	⍉	⊖	↑	↓

IO PRIMITIVES

MONADIC

⍞	⍎

⍝⎕=⍞⍎

DYADIC

⍞	⍎

NILADIC

⍝⎕=⍞⍞''

CONTROL PRIMITIVES

MONADIC

→	⍞

DYADIC

NONE

OTHER PRIMITIVES

MONADIC

⍴

DYADIC

,	←

It is not obvious to which classes if at all the "Other" primitives belong, but upon close observation they can be tied to the selection primitives. The primitive "Catenate" is simply a restructuring operation which joins two arrays together. "Assignment" performs no data transformations or re-arrangements but effects a naming of arrays. In some sense this is a selection operation.

The last primitive monadic "Rho", is technically a numerical primitive, but it is so closely tied to the array it is operating upon that I have also grouped it with the selection primitives. The result is shown in Table 1.3.2, where only four groups exist.

TABLE 1.3.2

THE FOUR GROUPS OF APL PRIMITIVES

MONADICS

NUMERIC

~ + - × ÷ ○ * ⍟ |

⌈ ⌊ ? ! ⍎ ⍳ ⍋ ⍒

SELECTION

⍴ ⌽ ⍉ , ⊂ ⊃

IO

⍞ ⍕

CONTROL

→ ⍎

DYADICS

NUMERIC

+ - × ÷ ⌈ ⌊ | ○ * ⍟

= ≠ < ≤ > ≥ ∧ ∨ ⍲ ⍱

! ? ∈ ⍳ ⍷ ⊥ ⊤

SELECTION

⍴ / \ ↑ ↓ ⌽ ⍉ ← ∩

IO

⍞ ⍕

CONTROL

NO EXPLICIT DYADIC CONTROL PRIMITIVES AS YET.

S/360 Implementation

This was the first successful implementation of APL, done in 1965 by IBM Inc. on an S/360 model 50 [BRE68]. It was highly successful and has set the standard for almost all other implementations.

The APL language was still under developement at this time. The result was most of the language as it stands today. Much of what was put into the implementation extended or modified Iverson's original proposal [IVE62]. It was, however, far from an optimal implementation as far as performance was concerned, nor has that company made significant attempts to correct the deficiencies in their software.

They represented data as either numeric, character, or function.The numerics were either 1 bit integers (booleans), 32 bit integers or 64 bit rationals. The M50 processor was equiped to deal with these data types at the scalar level. The booleans were packed efficiently into 32 bit words. Character data was encoded into 8 bits and packed 4 to a 32 bit word. The functions were condensed into an internal byte code string form, which was easier to interpret, allowing some of the text string searchs to be eliminated by doing the lexical analysis at edit time.

There were many restrictions on the users of this system, most of which were the direct result of restricting the workspace size to 2*15 bytes. However later implementations corrected this problem.

A number of inefficiencies existed in the execution of many of their primitives. All operations were explicitly carried out without the exploitation of "Drag Along" and "Beating" [ABR70]. This tended to produce very large temporary results compounding the problem of insufficient workspace size.

IO was primitive, providing support for hardcopy terminals only. There were no provisions to allow for future developements such as video terminals. Output of arrays was supported but input was character string oriented, allowing only a single line to be entered.

It is my opinion that the introduction of system function separate from the language was a mistake. This negates the possiblity of generating system level software in APL. Later IBM implementations corrected some, but not all, of these problems (some system primitives were added). It is interesting to notice that this attempt to provide operating system interfaces has become a standard for most implementations (the standard as proposed by A.Falkoff and D.Orth [FAL79]).

CDC* Implementation

This implemenation by Control Data Corporation on the CDC Star machine in 1973 was a radical advancement in array processing, both in the language and the hardware [CDC*STAR]. The Star architecture made heavy use of pipelining and vector machine instructions. Many of the scalar dyadic primitives could be implemented in a single machine instruction.

The software made use of the ideas of Abrams in his thesis on an APL machine [ABR70]. They employed two types of data storage: one form for objects and one for object descriptions. These storage types will be described in section 2.4. They allowed most selection operations to be delayed to the point were their explicit operation became unnecessary.

Data was represented similar to the IBM implementation except that all data was compacted into 64 bit words instead of 32 and 64 bit words. The integers were represented as a special case of the rationals so that they were all 48 bits in length within a 64 bit word. Booleans were compacted to fit within 48 bits of a 64 bit word. Character data was represented as 8 bit bytes, 8 bytes per word.

Their major extention to the APL language was the introduction of lists or generalized arrays. At present few other implementation have succeeded in this extention (a French system was described by M.Pierre having Generalized Arrays [PIE79]). Another improvement in the language was to allow assignment to an expression if that expression represented a subset of the data there (as proposed in section 1.2).

MCM Implementation

In 1973 MCM Canada succeeded in producing the first APL implementation to run on a microprocessor. They choose the Intel 8008 as it was effectively the only microprocessor available at the time. With some hardware enhancements to overcome the address space limitations of this processor, they managed to implement the full APL language.

The significant features of this machine were its size, user interface, and its IO facilities. The machine's physical size was small allowing it to be portable. In actual fact the first model MCM700 was a briefcase implementation with provisions for battery backup within the same enclosure.

The system supported object paged virtual memory from either tape or disk allowing the machine to use a very limited RAM storage yet allowing large workspaces. All operating system interfaces were provided via APL quad functions [MCM], allowing the user to develope higher level operating systems within the current system.

IO architecture was substantially improved in the MCM systems. They provided for interfacing almost any arbitrary IO device, supported through a set of user accessable system tables describing the current IO device and the protocols to use.

The MCM system took a radical approach to integer data types. Because the INT-8008 is an 8 bit processor, and the cost of RAM storage was prohibitively high for the application, they implemented an expandable integer word size. However, they failed to provide a separate boolean word size (they chose to represent booleans as 8 bit integers) The provided sizes are 1,2,3,4,5,6 and 7 bytes. The reals were similar to the format used by IBM's implementation, ie. utilizing 8 bytes.

The deficiencies of the MCM system were in the restrictions on the user, most of which resulted from MCM's efforts to minimize storage requirements. The most severe restricts arrays to dimensions of 0 to 255 (rank however was unrestricted, allowing up to rank 32).

In MCM's naming convention only the first three characters had significance. This has placed a significant burden on users in avoiding name conflicts. Along with the implementation deficiences was the machine's overall performance at array processing tasks. For many applications programs executed at speeds which are too slow for normal use and only suitable for nonsupervised stand-alone operation.

VANGUARD Z-80 Implementaion

This is a recent APL implementation(1979) for the Zilog Z-80 microprocessor [VAN]. It represents an almost complete implementation of IBM's APLSV, with a few restrictions on the domains of the primitives, and also with a few of the primitives missing.

It represents the state of the attempts to implement APL on modern low cost machines. It can best be described as a cumbersome and rather poorly thought out system. Very few options on data types were given, with booleans occupy one byte ,integers 2 bytes, and reals occupying 6 bytes as packed BCD numbers.

A serious fault of this system is it did not make use of the improvements from other implementations to date, such as virtual memory or generalized IO. It is certainly not a contender in the users market for APL or array processing.

The most noticeable array processor system other than an APL system is the STARAN by Goodyear Corporation. The architecture of this machine is radically different than the other machines described here. Effectively it consists of many parallel processors which can work independently of each other.

It can best be described as a multi-ALU system, one ALU per memory module, each memory module being a bit array of 256 by 256 bits. All data operations are on 256 bit wide words. These large superwords allow a high degree of parallelism in operations. Along with this large word width, is the ability to access the array along either of its two coordinates, with a third coordinate specifying modules. With the ability to have modules operate on data in parallel, ultra high vector operation rates could be achieved (approximately 40 million operations per second with only 4 modules).

The main disadvantage of the STARAN system is its 256 bit bus size. This is a very large physical bus size and leads to bus interfacing problems, making the system totally impractical for small or medium size machine design.

STARAN was originally indended as an associative memory subsystem which could be used in a parallel processing environment with a host processor. STARAN demonstrates the usefulness of having a separate data processor for efficient array processing and that if one has an easy access to array elements in an almost arbitrary manner, a significant improvement in performance can be achieved.

Another non APL system which was intended for array processing is the PEPE multiprocessor system [VIC78].It was once (1976), one of the most powerful systems in the world, being composed of up to 288 parallel processing elements, c.f. 2.2.1.

The basic units are independent processors, each of which derives its instruction from a common source. They all work essentially in parallel on separate data sets. Since the elements were weakly coupled, intercommunication was a bottleneck.

For arrays of modest size, many orders of magnitude improvement over conventional systems, such as S/360, can be achieved (that is, for operations on single sets of arrays such as simple scalar dyadics). Each of the units can handle a subset of the necessary scalar processes.

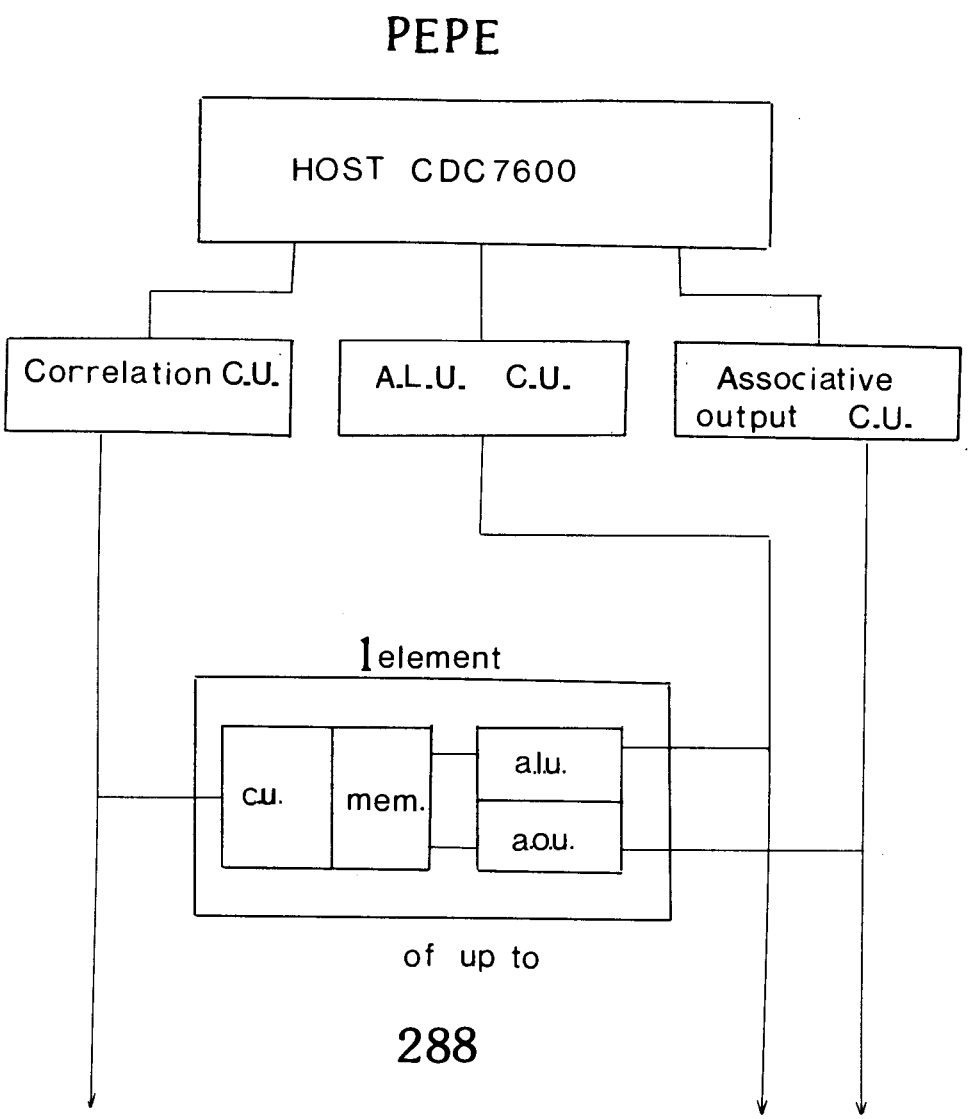

PEPE
HOST CDC7600
Correlation C.U.
A.L.U. C.U.
Associative output C.U.
1 element
cu.
mem.
a.lu.
a.o.u.
of up to
288

Figure 2.2.1

The problem with such a system is in data management.It becomes very difficult to move components between processing elements. With the lack of inter unit communications, complex operations such as array restructuring becomes a horrendous task. For these processes each processor in turn must be asked for data. Thus procesess will invariably be less efficient than a single memory stream instruction of the S/360 or the virtual restructuring of the CDC STAR.

The above is typical of using identical parallel processing elements. A tradeoff results between the efficiency of scalar dyadic operations and structural and interpretive operations.

From the various implementations and similar systems, it is possible to extract the best features and those aspects that should be incorporated in future APL implementations. These features fall within two categories, language and machine features. I will deal with language features first.

A significant language feature was the sucessful introduction of generalized arrays by CDC*STAR. Their aproach was similar to the proposed language extention of section 1.2. This same system also allowed limited specifications into selection expressions.

The above two language extentions are significant as they increase the ability of APL to express algorithms which were up to then extremely awkward to express. Assignment into a selection expression can reduce the number of steps in many APL algorithms. These two features will have a high implementation priority in the machine design to be undertaken here.

In looking at the MCM implementation, three facts about the language become apparent. First, object names should have no significant length restrictions.

The second was MCM-APL's inclusion of all system functions as primitives. The language then does not have two conflicting environments for the user to contend with.

The third important feature of the MCM systems was the ability for the user to reconfigure the IO system during execution. This is necessary for a single user machine which must adapt to changes in the IO devices connected to it.

All three of these features will have high priority in implementation.

The importance of boolean data strongly urges that booleans be compacted to a single bit per component. This results in order of magnitude increase in storage efficiency and a similar increase in streaming speed.

As far as reducing restrictions on array sizes, it is best to only restrict arrays by the amount of available memory and no other factors. All implementations which allow "Rank" of at least 16, appear to impose no apparent restriction on the users. MAPLE will allow "Rank" of at most 31.

If the workspace size is 2*32 words (unrealisticly large workspace for a single user) then dimensions should be allowed within [0,2*32]. The MCM systems in defining dimensions between [0,2*8] placed undue restrictions on the user.

The other array processing systems demonstrate the high efficiency of applying parallel processors to array processing, as increased performance for some problems results as more processors are introduced [MIT74]. However, the nature of these processors should not be as uniform as the PEPE system but allow specialization to achieve the increased efficiency.

In 1970 P.S.Abrams in his doctorial thesis laid the foundations for efficient processing of APL array structural operations and a method of improved interpretation [ABS70]. Since that time a number of implementations have succesfully adopted some of these techniques ([CDC*APL],[AMR73]).

A number of important principles for array accessing were introduced by Abrams. Many of these are the result of the IBM APL/360 implementation, such as the need for "Descriptors" to describe the properties of arrays.

All arrays have Rank and Dimensional parameters which are associated with the array itself. These must be encoded somewhere in memory to control the access to the arrays (any access algorithm must know the number of elements to access and in which order).

Abrams suggested a method by which, by the introduction of Rank plus 1 more components to a descriptor, one can describe a large class of array structural operations by simple manipulations of the overall descriptor. He called the use of principles such as this "Beating".

Associated with Beating is the "Reference Counter". It is a method by which unnecessary copies of arrays can be avoided but still preserve the illusion of individual copies. This is very useful in synthesizing "Call by Value". The basis of the reference counter is as follows: every array has associated with it a single word, which indicates a count of the descriptors which reference a subset of the array.

Some arrays can be described by just a descriptor and a reference count. Thus two types of descriptors are possible: those which directly describe an array without any transformation information and those which indirectly describe a new array based upon another array and some transformation information. The former will be called "Storage State Zero" (SS0) arrays and the latter called "Storage State One" (SS1) arrays [CDC*GID].

The nature of the transformation information will be discussed now. One of the components necessary is to indicate an "Offset" into the transformed array, while the remaining Rank components called the "Jump" vector indicate how to access array elements along each of the Rank axes. An example of this principle is illustrated in Table 2.4.1.

TABLE 2.4.1

PRINCIPLE OF SS1 DESCRIPTORS

OFFSET: *INDEX INTO SUBSPACE DEFINED BY ARRAY THAT IS INDIRECTLY REFERENCED.*
RANK: ⍴⍴*ARRAY*
JUMP: (⍴*J*)=*RANK*
RHO: (⍴*RHO*)=*RANK* ∘*RHO*=⍴*ARRAY*

IF 'I' ,A VECTOR (RANK=⍴I), REPRESENTS THE COORDINATES INTO AN ARRAY, THEN THE LOCATION OF THIS I'TH COMPONENT IS GIVEN BY

OFFSET+*JUMP*+.×*I*

THIS DEFINES THE JUMP VECTOR

⍝*EXAMPLE*

```
ARRAY1←⍳255 ∘⎕IO←0
ARRAY2←  3  4  5
         9 10 11
        15 16 17

OFFSET←3
JUMP←6 1
RHO ←3 3
⍝THEN

11=ARRAY2[1;2]
11=ARRAY1[OFFSET+JUMP+.×I←1 2]
```

Appendix 2 contains a practical set of algorithms which can work with SS1 descriptors to produce SS1 descriptors which reflect APL selection primitives. These algoritms were described by Abrams' thesis chapter 3, but in a slightly different notation.

These algorithms imply virtual operations on an array's entirety while having time complexities which are linear in the arrary's rank. This provides a substantial reduction in both the time and space complexities of APL structural primitives. Thus most selection operations can occur at speeds independant of the number of elements in an array.

In the realm of storage efficiency, it is noticed that the result of the monadic index generator primitive can be described as an initial value, a count of terms, and a step direction (bit). This was the proposal that P.Abrams introduced in his thesis.

The implementors of the CDC*APL extended this concept to an initial value, a count of terms and an increment value. This latter method describes a class of array called "Intervals". Intervals allow the expression of floating point arrays in which a constant term relates all elements. They require just three components to express any such vectors: a dimension term, start value, increment value. Such descriptors will be called "Storage State Two" (SS2) arrays.

An SS2 array requires no memory allocation except that which the descriptor itself requires. This leads to significant reduction in storage requirements for a large set of vectors starting from an initial "Index Generator" primitive.

It is my proposal to extend the concept of the Interval, from simply vectors to arbitrary arrays. Since all arrays have the RHO information present, this extention only requires an extra term per axis. This extention allows selection transformations on interval arrays without the conversion first to an SS0 array, and hense allow reduced storage requirement (a similar proposal is supposedly due to D.Samson [SAM79]).

The form of the descriptor is the same as that for an SS1 array except that the Offset and Jump terms have the different meanings of Start and Increment respectively. There exists one other difference between SS2 and SS1 array descriptors: an SS2 descriptor does not have a component indicating an indirect reference to another array. In this sense an SS2 array is similar to an SS0 array, in that other arrays may indirectly point to it.

Abrams also addressed, in his thesis, some problems of APL interpretation. He and others have noticed that many APL statements have the nonsymmetrical property that an equivalent statement can be synthesized with reduced complexity. A few examples are given in Table 2.4.2. He called this principle "Drag along".

His proposal was a stack architecture in which the instruction stream could be modified to effect equivalent semantic expressions. As this subject concerns more the tasks of APL interpretation and as this thesis will concentrate on the problem of data manipulations, it will be deferred as later research.

TABLE 2.4.2

```
⍝ EXAMPLES OF DRAG ALONG

3↑10+X    ←→    10+3↑X          ⋄X←1+⍳1000

⍝ THE LEFT REQUIRES 1000 '+' OPERATIONS
⍝ WHILE THE RIGHT ONLY 3 '+' OPERATIONS.
⍝ HOWEVER, LEFT DOES ALL 1000 DOMAIN CHECKS
⍝ WHILE RIGHT ONLY 3.

A←⍳1000
B←A*3.1
C←12
A+B×C      ←→ DO I←1 TO 1000 (A[I]+B[I]×C)

⍝ WHILE ACCESSING ELEMENTS OF ALL ARRAYS
⍝ SIMULTANEOUSLY ONE ELIMINATES TEMPORARY
⍝ ARRAYS.
```

As was seen in section 1.3, the division of the primitives of the APL language resulted in 4 classes of operations, leading to Table 1.3.2. These four groups are almost mutually exclusive and suggest differing hardware. It was this division which influenced MAPLE's design.

The developement of a processing machine for the numeric primitives is treated elsewhere [GIL74]. In particular, most of these primitives (scalar ones) lend themselves to efficient pipelining techniques. In general they can be classified as having one or two input data streams and one output stream, with no required storage of operands past the immediate operation.

IO processing involves single data stream paths, to and from IO devices, with possible data conversions along the way. The nature of the IO devices may be dynamic so that local control of them may be required. Such is the case of multiple devices requiring differing drive functions and the case of variable communication protocols. These functions can best be controlled at the interface level via an IO processor (this is the design philisophy of the IBM S/360 with its IO control units and channels).

The separation of the selection primitives, is again, a natural one. They perform no transformations upon the actual elements of an array, but only on its overall structure as it resides in memory. P. Abrams [ABR70] has shown that generalized selection operations on arrays can be performed by a single algorithm if a certain representation is taken.

The need for local control of memory management functions and some system operations was also shown to be true in an article by D. Samson [SAM79]. This lead to the concept of a smart memory machine,the Data Manipulation Unit (DMU), which is described in chapter 5.

Section 2.4 discussed the work by Abrams on APL machine design, which showed us that an APL machine must divide the work of program execution into exclusive tasks (chapter 3,4 [ABS70]). What these tasks are depends upon the approach taken. He describes a system of at least two processors: that of a D-machine and an E-machine, where the first produces code for the second. It was not his contention that such a system was optimal, but that an improvement in the execution could result if specialized processing was present.

The organization taken here is similar to that taken in the STARLET system [GIL74]. This hypothetical machine's architecture was also inspired by Abrams's thesis and is shown in fiqure 3.1.1. These people adopted a multiprocessor system with very specialized processors.

Where the STARLET was a tightly coupled system of dependent elements, MAPLE is a more loosely coupled system of independent processors. The major architectural similarities between MAPLE's design and the STARLET's, are that both are array processors and are multiprocessor based, and both may make use of pipelining techniques.

The design philosophy of MAPLE is to allow a maximum amount of concurrency to occur during execution and a very modular approach to the architecture. The result will be that changes in language or hardware will have minimal effect upon the organization of the machine. This will be realizable through four specialized processors, each of which has a minimal instruction set and no fixed internal organization.

Johannsen[JOH78] describes an architecture for a system of microprogrammable modules. Together they allow modular construction of complex processors. It was his contention that the separation of a single processor's functions into separate units (even at the microprogram level) allows for a more efficient processor design. Here, it is my intention to describe a more systematic and efficient approach to a high level machine design, through the use of parallel processing and multiprocessing.

STARLET SYSTEM

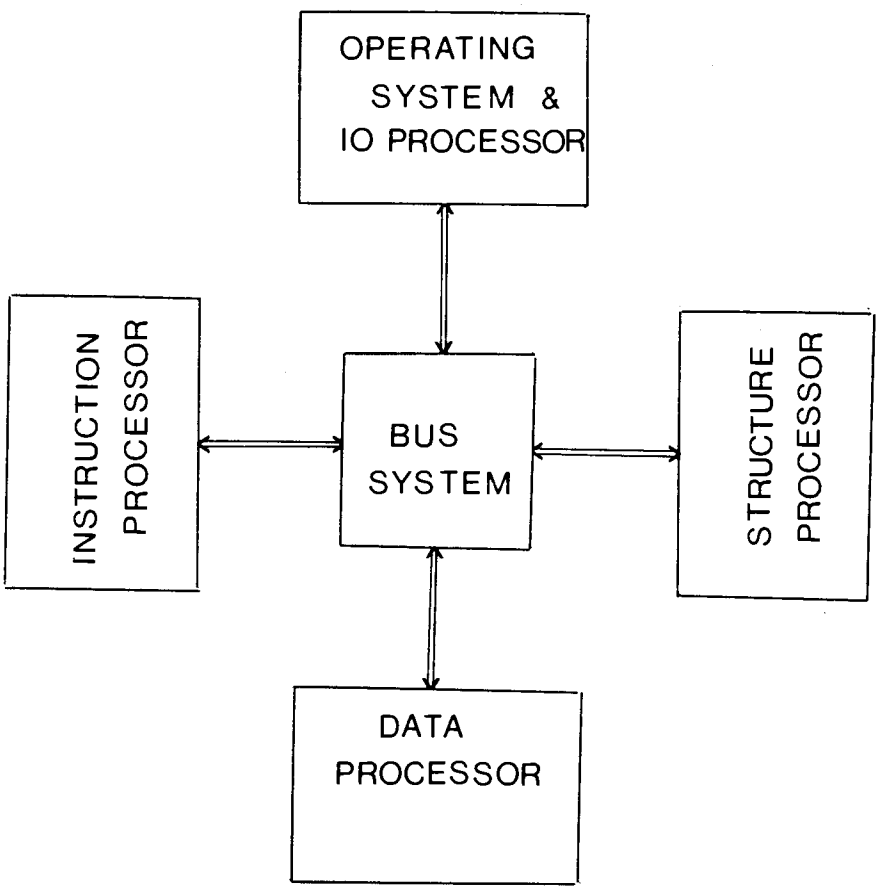


Figure 3.1.1

This attitude of modularity is shown strikingly in two papers by Hobson [HOB80-1][HOB80-2]. Described in the first is an approach towards specialized processing elements to achieve high level language implementations and in the second is a system which is suitable for high level langauge interpretation using multiprocessing (specifically for an array language like APL).

MAPLE is similar to a CDC STAR 100 computer system running APL [APL*STAR] in that both provide array operations at the machine level, and support an extended version of APL. They are, however, radically different as far as machine architectures are concerned [CDC*STAR].

MAPLE is a moderately coupled multiprocessor system, each processor having its own local store for buffering of data/instructions. There are four processors or units in this system each with specialized functions. They are the EXecution Unit (EXU), the Input Output Unit (IOU), the Arithmetic and Logic Unit (ALU), and the Data Manipulation Unit (DMU).

The EXU is responsible for interpretation and execution control of the language. It will cache the functions that it is interpreting and in so doing issue instructions to the other three units so that the statements can be executed.

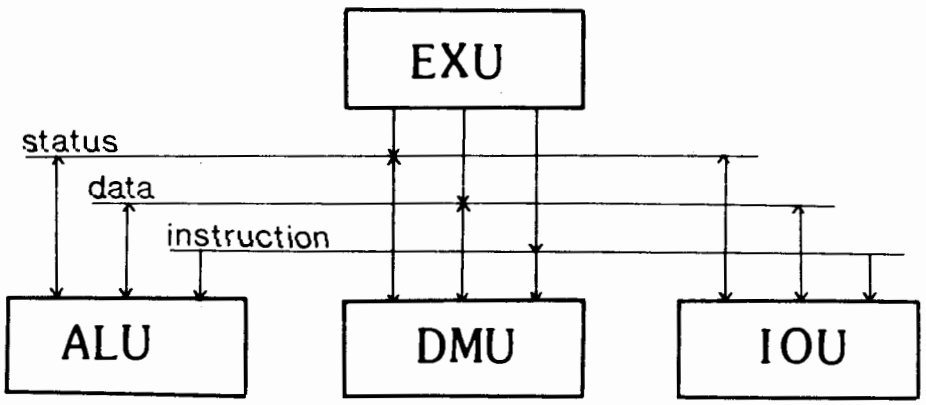
The IOU supplies the interface between the user and the system. This is accomplished through control of various IO devices and high level editing features. All data formatting and conversions for display are performed by this unit.

The ALU is a high perfomance arithemetic unit which performs all scalar operations. It produces numerical results only.

The DMU is the most important and complex of the four units. It performs all main memory storage and management functions along with the APL selection functions. It also performs all the associations between array names and their actual locations in memory. The actual accessing of arrays is through the DMU in the form of a vector data stream onto a common time multiplexed bus. Figure 3.2.1 shows a simplified view of the overall arrangement.

These four processors will be combinations of existing microprocessors and microprogrammed hardware. The total number of processing elements is not fixed but will be greater than four. Their functions and locations will be grouped into the four separate machine units.

The next chapter will investigate most of the problems in implementing the workspace in which all arrays reside (the workspace being a logical entity supported within the DMU).



BUS OVERVIEW

Figure 3.2.1

This chapter covers the data structures that exit within MAPLE and the implementation problems they generate. Here I am mainly concerned with implementing an APL workspace and the architecture necessary to support the workspace.

The workspace is constructed in R/W RAM memory by the organization of arrays into distinct subspaces of this memory. The memory architecture of MAPLE will consist of a single large bit addressable store in which all arrays and system objects are located.

The reason for only a single memory module was, the separation of memory into multiple units was not conducive to easy array maitenance and memory management. The only advantage of multiple memories seems to be in modularity but this is overshadowed by the simplicity of a single store. No speed improvements can be obtained unless multiple buses exist (a far too complex and untidy situation).

The majority of main memory accesses will be in the form of word fetches (a word is some number of bits). For efficiency's sake, main memory will accessed in word units but addressed at the bit level. The choise taken for MAPLE was 16 bits per word due to three factors: (1) increased word size must be offset by increased bus size and complexity. (2) memories are tending to be standardized in 8 bit wide I.C.s suggesting the desireability of multiples of this width. (3) 16 bits is becoming a popular width for microprocessors with resultant hardware availability.

As 16 bits is the fundemental access unit of memory it makes sense to base the addressing on this unit. However, 16 bits is totally inadequate to represent a sufficient address space (2*16 =65K) especially if bit addressing is desired. By using two words or 32 bits, 2*32 bits or 2*28 words can be addressed (16 bits per word). This amounts to 537 million bytes of addressable storage or one half gigabyte.

With the current trends in memory technology, between 10 to 100 megabytes of RAM storage, with 100 nanosecond access, will be available in the near future for a few thousand dollars and fit within a minicomputer frame. As this is being written, one megabyte of RAM storage costs a few thousand dollars and would occuppy four five by ten inch boards.

For many applications a few megabytes of storage is sufficient. However, if expansion is to take place without modification to the system software, the address space should be large enough. Thus the choice of a 32 bit address.

Note currently there are available 2*16 bit RAMs with the desired access times. With the introduction of 2*20 bit RAMs the memory size expectations will be realized.

All arrays require storage, with some requiring more or less depending on data class and structure. For the purpose of memory management, all arrays can be considered as memory segments characterized by a beginning address and length. Segments are contiguous nonoverlapping regions of address space.

Memory Management is now defined as the processes which handle both requests for new storage, called "Allocation" and the release of storage which is no longer in use, called "De-allocation". The problem that APL presents to memory management is that both the address and size of arrays are dynamically changing, in contrast to systems in which most arrays have static memory requirements. An example of APL's dynamic nature is exemplified in the primitive class of scalar dyadic functions. These primitives always generate a totally new array, which requires temporary storage based upon the size and nature of its arguments.

The most important requirement of an APL system is the ability to reassign to any object a totally new array with arbitrary characteristics. This has the effect that the old storage associated with an object is released, without being used towards the new storage for that same object. If the system tried to overwrite an array with its update then data integrity would be jeopardized. The new data is therefore generated and reassigned to the object, releasing the old data.

A second requirement on segments in an APL workspace is that the memory subsystem allow Random Access into arrays along with the usual sequential access for scalar dyadics. If the address subspace of a segment is contiguous then this is easily satisfied.

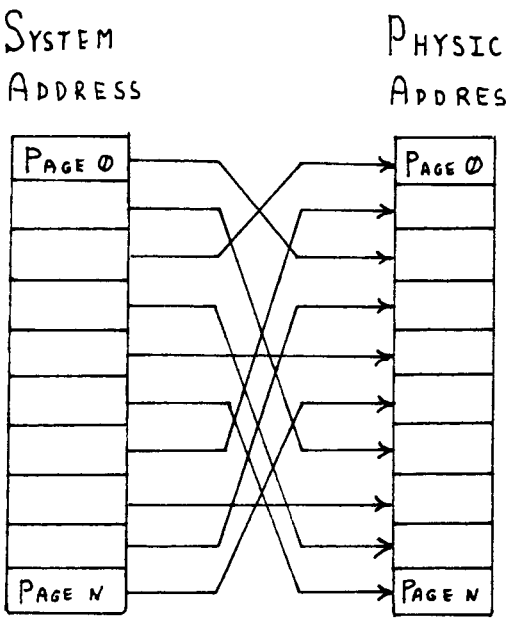
An APL workspace can be partitioned into two sets: the set of all segments associated with arrays and the set of all unused contiguous address subspaces (whose elements are called "holes"). These segments and holes can be scattered throughout the workspace or ordered depending on the memory management system used. All memory management systems (dynamic systems as defined in this thesis) can be characterized by the existance of holes and segments and the properties that, all "allocations" are taken from holes and all "de-allocations" turn segments into new holes.

There are basically two general methods for the management of memory, called: (1) Hole Table Maintenance and (2) Address Maintenance. The former maintains an inventory of segments and holes and rearranges segments, while the latter tries to remove the contiguous requirement placed on segments. Address Maintenance is examined first (it was not the method chosen for MAPLE's memory management).

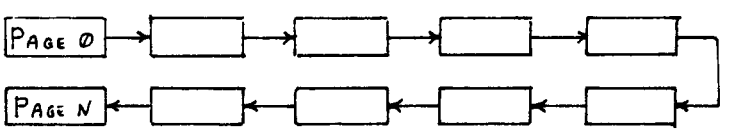
Address Maintenance attempts to allow pairs of segments or holes to be linked together to generate a new segment or hole, without the need to physically move any of these subspaces. There are two methods which accomplish this: first, regions (pages) of memory can be linked in a linked list where each region points to its successor; and secondly pages (regions) can be mapped to an isomorphic space via a translation table.

Both methods allow regions and hence segments to appear as contiquous sequential address subspaces, however, only the translation method allows random access within these subspaces (guaranteed by the definition of an isomorphism). cf. figure 4.1.1.

Translational Memory Management



Page Translation



Page Linking

Figure 4.1.1

Consider a pure Address Maintenance system, with no Hole Table maintenance. Within the subspace represented by the pages of a segment, there can be only one array. All arrays will have the same value of offset (usually zero) into a page for their first address, otherwise these offsets must be maintained (which is equivalent to Hole Maintenance). The above represents the condition that all segments are allocated in page size units, with only a single array per set of pages.

These pages have sizes from one word to some large fraction of the workspace. The ideal situation would be the ability to translate any word address so that every word was available to be linked together. However, for every page that can be relocated there must be an element of the page translation table, representing a possible significant overhead to support memory management.

Since array segments are allocated in page units, in a random distribution of segment sizes every segment will have an average waste of memory equal to one half page. Thus reducing page sizes reduces the amount of memory per segment which is not in use. However, reducing the page size inversely increases the page translation table size and its associated maintenance.

Page table systems have one very serious disadvantage, if all arrays are restricted to starting at a page boundry, then the maximum number of segments is directly related to the page size. In trying to reduce page table maintenance the page size must be increased, reducing the flexibility of the system in defining new objects.

Summarizing Page Table management:

Disadvantages;

-(1) there is a waste of memory per array depending upon the page size and number of segments.

-(2) there is an overhead in storage in the translation table which might not be negligible.

-(3) all addressing requires either an extra memory cycle to locate the next page or a comparison to determine if the address is within the current page.

-(4) the translation table must be modified to support memory moves resulting in non negligible processing.

-(5) *** the number of distinct arrays/scalars that can exist is strictly less than the number of pages in the system.

Advantages;

-(1) segments do not have to be physically moved to affect their motion in the workspace. Thus to rearrange the workspace the Translation table only need be rearranged.

Hole Table Maintenance (HTM) is the alternative to Page Translation. This performs the linking of holes by the actual physical relocation of arrays to produce a hole which will accommodate the allocation request,c.f. figure 4.1.2.

HTM has many forms, a common one is to allocate from only one hole till it is exhausted, at which time the entire workspace is reorganized to generate a single hole representing all free memory [SYK79]. This scheme is very simple but not very efficient. The inefficiency is due to the lack of use of the holes fragmenting the workspace from the continually releasing of array storage. It is highly probable that one of these other holes would be able to accommodate the request.

A solution is to maintain a table of all existing holes from which all requests are filled. This presents four main problems:

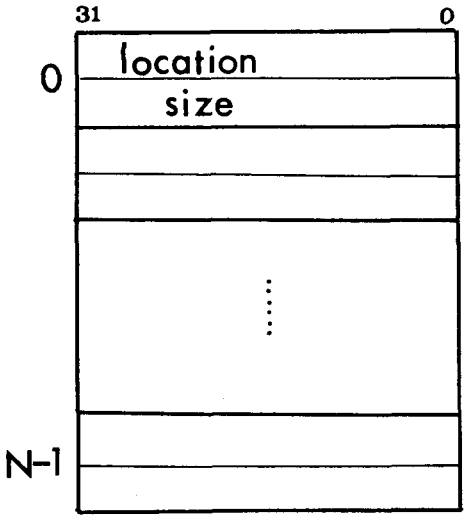
(1) which hole is to be chosen for a request?

(2) as the number of holes increases the time to search the hole table increases.

(3) the rate of releases of arrays may exceed the rate of requests for a time sufficient to overflow ANY hole table (the example is the return from a large recursive call).

(4) what is to be done when the hole table overflows?

Hole Table



N - current table dimension

Figure 4.1.2

The choice of the hole is usually either first fit or best fit depending on how the hole table is organized. First fit was chosen for MAPLE's memory manager as it has some useful properties: (1) The search time to locate an allocation request is better than for best fit; (2) With concurrent processing table updates do not play an important part in evaluating performance; (3) By ordering the hole table by increasing addresses successive memory requests will be more stable and tend to compact the workspace into arrays in the lower addresses and holes in the upper addresses.

Problem (2) is solved by limiting the hole table to a maximum size which does not present a serious search time. The exact size of this table will be the subject of future experimentation, however other researchers indicate that the optimal size is on the order of 64 entries [CDC*GID].

All Hole Table maintenance systems can be characterized by the few properties listed in table 4.1.3. The more efficient schemes reduce the probabilities of hole table growth, resulting in fewer conditions where objects must be relocated ("Garbage Collections").

TABLE 4.1.3
PROPERTIES OF HOLE TABLE MAINTENACE

```
H   <=>    CURRENT NUMBER OF HOLES
M          MAXIMUM NUMBER OF HOLES

∧/, H≤M    UPPER BOUND ON TABLE SIZE

S   <=>    SIZE OF ARRAY REQUESTED
           OR RELEASED.
L   <=>    LOCATION OF RELEASED ARRAY
HS  <=>    VECTOR OF HOLE SIZES
HL  <=>    VECTOR OF HOLE LOCATIONS

□WA=+/HS   SIZE OF FREE MEMORY

ALLOCATION:
H←H+0      IF HOLE CHOSEN IS >S
H←H-1      IF HOLE CHOSEN IS =S
H←H-X      X≥1 IF 'GARBAGE COLLECTION'(G.C.)

G.C. <=>   ∧/S>HS

DEALLOACTION:
AUGMENTATION     H←H+X
X=0         <=> (L∈HL+HS)≠(L+S)∈HL
X=¯1        <=> (L∈HL+HS)∧(L+S)∈HL
OTHERWISE          H←H+X
X=0              GARBAGE COLLECTION
X=1              (~G.C.)∧~AUGMENTATION

G.C.        <=> (H=M)∧(~L∈HL+HS)∧~(L+S)∈HL
```

Garbage collection need occur only if either: (1) no hole can accommodate the request and there are more than one holes or (2) a hole table overflow occurs. As workspace size increases the probability of not finding a hole large enough decreases and as MAPLE was intended to have a very large workspace the emphasis was placed on reducing hole table overflow.

It is therefore important to investigate the mechanisms which affect hole table growth. Table 4.1.3 indicated that two holes can be augmented, this occurs when a newly generated hole (released array) and an existing hole share a comman boundry address. Since memory can be allocated in words, there is a non zero probability that a new hole can be augmented with some current hole, resulting in the size of the hole table not changing or even reducing.

By ordering the hole table by increasing address and searching for first fit there is a trend to cluster segments in the lower addresses along with the associated clustering of holes with lower addresses. As the probability of an augmentation increases as the density of holes within a new holes address space increases, this clustering tends to increase augmentation.

Statistics taken from the CDC*APL system, which used first fit in a 64 entry hole table, showed that the execution time spent in doing garbage collections due to table overflows was negligible [CDC*GID]. However, attempts should be made to increase the efficiency of the garbage collection as in some processes (such as real time applications) excessive delays in moving memory can not be tolerated.

Garbage collection triggered by an allocation request results in locating a set of two or more holes, which upon relocation result in a single hole of size sufficient to accommodate the request. The most efficient garbage collection algorithm results in the movement of the least amount of memory and the least processing overhead.

Since the hole table is ordered by address it is relatively easy to find a minimal set of holes which will result in the least amount of array movement. Once this set is found all arrays contained within the range of addresses defined by these holes will be moved down into the lowest hole. This has the effect of bubbling holes up into the highest hole till only one high hole exists (within this subset of holes).

Moving an array involves two processes: (1) the sequential relocation of its elements and (2) the total update of all references to itself. The first process requires exactly two memory cycles per word of the array, while the second processes is heavily dependant upon the organization of the workspace. In systems which implement Lists there is the enormous problem of updating all forward pointers, with one solution being to use backpointers. In the next section the workspace organization is discussed and the solution to updating is shown to be near trivial.

The average of the minimum amount of memory that exists between the set of holes that are to be collected dictates the average overhead of garbage collection (assuming a near constant time to locate the set of holes to collect). This can vary from just a few words to many thousands of words, however, a measure of an upper bound on the amount of memory to be moved can be found.

The worst case condition occurs when all holes are separated by equal size regions of arrays. As the highest address in the address space is within the last hole the amount of storage represented by these regions is WorkSpace Used

---

Holes-1

It is expected that the dominant cause of garbage collections will be hole table overflows, which result in exactly two holes being collected. By choosing two holes with the least amount of storage separating them, the least time for a garbage collection will result. These two holes will be precomputed, before any garbage collection is required, concurrently with memory utilization to reduce the overhead of a garbage collection.

Stacks

There are some operations on arrays that are stack like (FIFO and LIFO), in which to have to regenerate the whole array for each operation cycle would be highly inefficient. A "Push" operation is logically the catenation of a scalar to a vector and a "Pop" is a last element take and drop. While takes and drops do not require memory movement a catenation always does, thus a mechanism must exist to do a Push without memory movement.

An obvious solution is to make all LIFO stacks fixed static objects with some maximum address space, such that no overflow can result. In APL the execution stack must be able to grow to fill the whole workspace available, or typically more than half of the entire workspace (to allow flexible recursion). Assigning the majority of the workspace's usable memory to any object is highly restrictive considering that most stacks have average sizes of only a few percent of their maximum size.

What is needed is a method of dynamic allocation which does not move memory to allocate increases in stack sizes and only utilizes as much memory as the stacks current size. This may be achieved using Address Translation memory management, but this method was not chosen for memory management of APL arrays. Stacks, however, are not a data type defined in APL (at least not at present) so it is not in conflict with the array allocation scheme if stacks utilize this scheme of memory management.

Dynamic allocation of system objects (such as stacks) use Page Table Maintenence for the following reasons:
(1) system objects are usually large so page size can be large, reducing table sizes and processing.
(2) system objects do not change their base address as the result of any operations on them.
(3) size modifications of system objects is simply a matter of linkng and unlinking pages.

Consider a workspace divided into "n" unit subspaces (pages). These pages can be characterized as one of three types:
(1) containing only an array or part of an array (no free space).
(2) containing a hole and an array (some free space).
(3) containing only free space.

The first type represents zero waste, the second represents waste which memory management handles, the last represents a page which is not in use at all and as such is free to be used in any possible way. It is the existance of the last type of pages which allows efficient dynamic allocation of system objects.

MAPLE's dynamic system-object maintenance operates as follows:

(1) each such object is allocated a fixed static subspace of the workspace (base address and size are fixed).

(2) a system of virtual memory is used to swap into these subspaces real pages to satisfy the required current sizes of these objects.

The mechanism for this virtual memory can be described as follows:

De-allocation;

(1) whenever a page which is entirely a hole is detected by memory management the real memory associated with this page is removed creating a "Black Hole" in the workspace, and the real page is placed on a free list of pages.

(2) an object's address space will be composed of both real and virtual pages (black holes), where the actual storage associated with the object is just the sum of the real pages.

Allocation;

(1) whenever an object must be expanded by "n" pages, if there are at least "n" black holes within the object and "n" free real pages, then these black holes will be filled with real pages.

(2) the replacement of one black hole with a real page produces another black hole elsewhere (the number of black hole pages is conserved).

(3) the movement of pages is supported via a translation table of pages, called the "Relocation Vector" (R.V.).

Intially in a clear workspace there exists a set of system objects and one Free Hole. All real pages, except one per system object, exist in the Free Hole. Therefore the amount of real memory in the system is dictated by the size of workspace the user sees. The size of the workspace the system sees is this user size plus the sum of system objects' maximum sizes, that is the virtual space is larger than the real space supported by RAM.

The choise made in MAPLE was to utilize two 16 bit words for all addresses for a 2*32 bit addressable memory space (usually the virtual space is defined as much smaller). The lower 16 bits of an address form a bit address into a word, and a word address into a page, given by the high order 16 bit word, c.f. figure 4.1.4. This mapping allows 2*16 pages of virtual space with pages being 2*12 words (8k bytes) in size.

Since there are few system objects the wasted memory generated by a Page Table maintenance system due to these large pages is negligible. A large page size also decreases the overhead in page translation, both in the size of RV and the overhead in locating the next page from RV. Figure 4.1.5 demonstrates the function of the RV.

In the next chapter the architecture and engineering of the memory management subsystem will be described.

Virtual Address

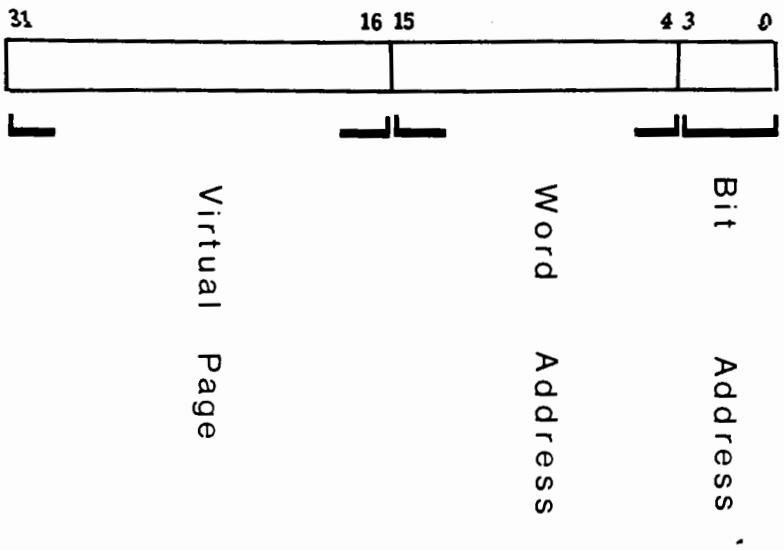
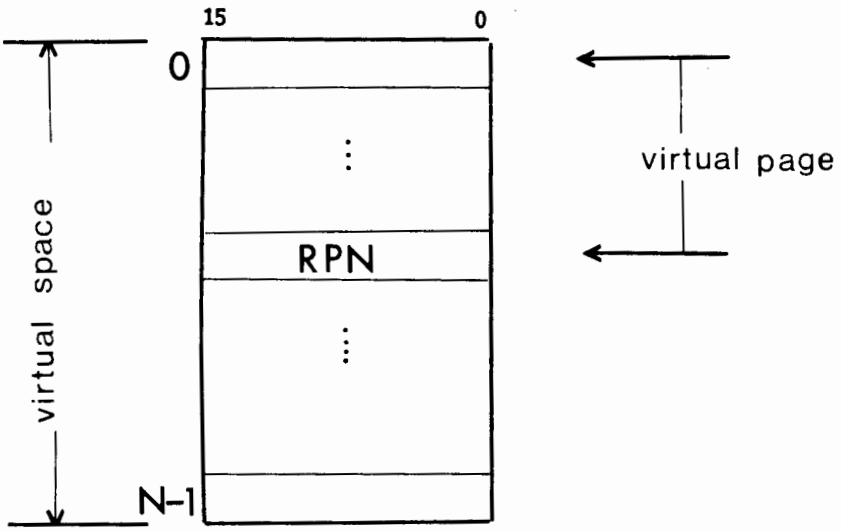


Figure 4.1.4

Relocation Vector



RPN = RV [virtual page]

Figure 4.1.5

All APL data is represented as one of the following: function, numeric, graphic, or list (generalized intermixed forms). In this system definition, each of the four classes of data may have many subclasses (which in turn may also be divided into subclasses) reflecting considerations such as integer/real distinctions of numerics. The implementation considerations will now be discussed for the different classes of data.

Numerics

There are two subclasses of numeric data, the real and complex numbers. To the extent practical, all possible values of these subclasses must be represented. This implies the need for a floating point representation for reals and (X+iY) of complexes.The complex numbers, X+iY, will be represented as an ordered pair X,Y of reals where X and Y will both be of the same numeric subclass. e.g. both boolean or floating point. For the reals one must approximate the irrationals to their nearest rational value as they can not be directly represented as numeric values. Similarly the rationals must be approximated to the nearest fixed length floating point number.

It is wasteful to represent the numerics in the range (0,1) (ie. Booleans) by the bit patterns for reals, since reals require a larger number of bits than Booleans. As booleans are numeric, they will require a separate bit representation. The obvious representation is for a single bit to be used, thus a boolean vector is simply a sequence of bits, the same number as its dimension.

With the notion that each component has a size based on the subclass it is in, an efficient representation can be chosen for each class or subclass. It is important to note that specifically within the class of numerics, the components of an array will invariably not be of the same subclass. An example is the intermixing of boolean values and rational values. However, it is not possible to achieve an efficient streaming of components in a generalized selection format if the components of the arrays have varing bit widths [LAW75].

This implies that all components of an array be in the same subclass even if storage efficiency is not optimal. Thus there will be conversions between subclasses. These conversions should be kept to a minimum, as they require processing that is not implied by the instructions that may initiate them.

Since integers occur frequently in data storage and information processing, a distinct set or division is in order. It is obvious that the booleans, which are integers, require a separate subclass therefore the integers will be divided into its own subclasses. In section 2.1 it was mentioned that the MCM system had 7 distinct integer sizes, it remains then to determine an optimal set of integer sizes for MAPLE.

The next class (the characters) will show that a useful component size is 8 bits allowing the representation of -128 to +127 integer values. Though this is a useful range of integer values, it is insufficient for most applications. Integers within the range +-10*9 should be allowed, or values within the range of the dimension of the ravel of the largest array allowed in the system. Since booleans serve special functions in information storage it would be improper to restrict the dimension of a boolean vector to less than half the available bit storage in main memory, which could be on the order of 10*9 bits.

As in most of the previous implementations,the integers were simply divided into two divisions: booleans (1 bit integers) and all others as 32 bit integers [BRE68]. This author advocates the above and also the further division into multiples of byte widths as in the MCM implementation [MCM].

MAPLE supports 1,2,4 and 6 byte multiples for integers. Byte multiples of 3 and 5 are dropped as their width oddness is unmanageable. The 1 ,2 and 4 are common sizes and most computer systems have them. The 6 byte integers are not useful for indexing as they are too large, but they are a convenient intermediary between the rationals and the other integers,as the rational coeficient is a 6 byte integer.

The floating point format has a length of 64 bits with a 16 bit binary exponent and a 48 bit integer coefficient. The coefficient will be right normalized to ease conversions between 48 bit integers and rationals. This format is a standard one as used in many of CDC's large mainframes [CDC*GID]. The format amounts to 4 words of main memory per component. This format for floating point numerics is also easy to microprogram.

It is important to note at this time that component sizes, though having to be uniform within any array, do not have to be of any specific bit width. It is only for the ease of construction of the component from word accesses that they are standardized. Since it is the objective of this machine's design to provide efficient bit addressing, the above restrictions on component sizes are an implementation consideration aimed at providing overall high speed memory access to all components.

Characters, Graphics and Others

This definition will include all data representations that I/O should handle. It may include forms of speech in manners which are unconventional, or special network interfaces. The main point to make is that classical APL and other programming languages are somewhat restrictive in their data representations. This implementaion will correct some of these deficiencies.

Within this class of data there will be the subclass of character data ,(present in other APL machines), as well as other possible subclasses which have not yet been defined.

For the classical case of a computer alphabet, one has between 64 and 256 characters to deal with [FAL79]. These are easily encoded into 8 bit codes. However there existed no mechanism by which one could perform operations on these elements or to form new elements to be added the set. Such an operation might be the logical OR between the boolean matrices representing the fonts of characters (ie. overstrike operations at the user level).

In this implementation, characters will be represented as integer indices of the definition objects for each character. Thus there will exist a system object called the Atomic Vector which is a list of definitions for each character. The IOU will interpret this list to perform the desired IO function. This list may, as above, define characters as boolean matrices for display fonts or as translate values to some other IO device.

Instructions

The number of primitives in APL is less than 255, so that 8 bits is sufficient to encode all of them. Since the Interpretation or Execution processor will be designed to interpret APL directly, it will only need to look at one of three types of data. The first is an APL primitive, second a named object, and third is a literal or character vector. The recognization of APL primitives is trivial while named objects require associative searches.

In APL the definition of names is relatively arbitrary, which presents the problem of encoding them for look-up during interpretation. The choices are to do an array search to identify the name (ie. no encoding), to hash the names, or to perform no searchs at all for name object associations. The latter is performed by doing all name-object associations at edit time for the text strings involved. The names are replaced by an index of a symbols table where the literal form of the name (or pointer to it) is stored. This eliminates a class of searches but not all searches. The others are necessary to support certain function parameter calling modes, which will be discussed later.

As described previously, character data will be mapped to integers, so literal strings will be integer vectors of usually 8 or 16 bit lengths (more than 2*15 character codes is excessive). Thus all APL text can be compressed into an integer vector as the storage form. The integer division used will either depend upon the maximum object/character code value encoded or it can be a unique division with varying component lengths.

This may seem in conflict to what was said earlier about uniformity in component sizes but it must be pointed out that selection operations on an APL line of code is at present not permitted and no proposals to allow such operations have been presented. Thus the primitives will be encoded into 8 bits, while the characters and names could be encoded into varying length bit patterns.

Lists

Lists are arrays of object references (ie. integers for indexing an object reference table). The name encoding above is an example of scalar lists. There are no restrictions on the classes of data which can be combined or imbedded within a list, nor are there any restrictions on the depth of imbed except through available memory.

Figure 4.2.1 shows the overall data hierarchy within this implementation. Shown are the four classes of data and their associated subclasses.

Descriptor Bit Representations

All objects have their own descriptors which give the object's shape ,location, and data class. Shape information was described in the section on selection descriptors in section 2.4. Location is given via a list scalar or object reference.

Data Hierarchy

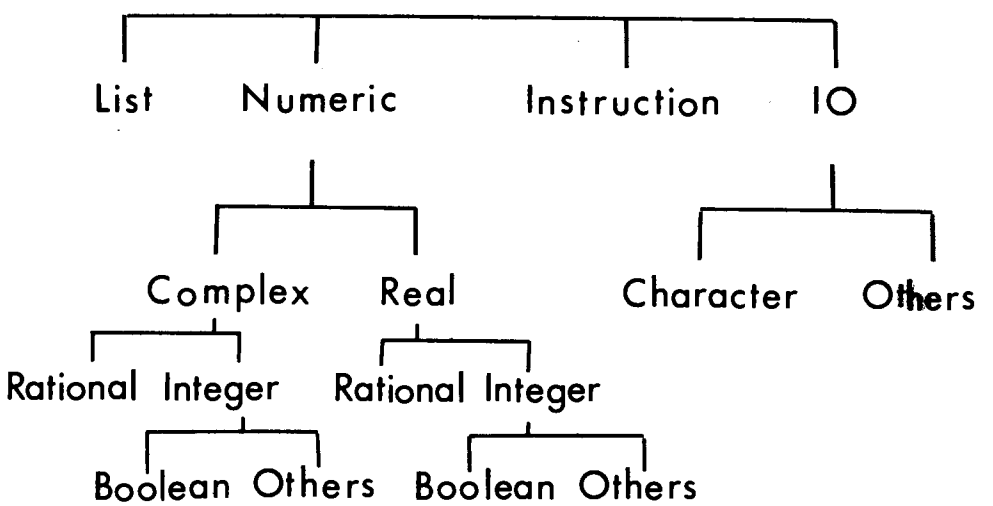


Figure 4.2.1

There are two basic types of descriptors, the storage state zero (SS0) and storage state one (SS1) descriptors. Since SS0 and SS1 arrays all contain rank,rho and type information their encoding will be discussed now.

Rank will be restricted to belong to [0, 31] for efficiency in the operations in the hardware selection unit. There are no algorithms in current publications on APL in which arrays of ranks greater than 31 are generated, so it is assumed this restriction will have negligible effect on the machine's intended purpose (H.Saal found that rank was usually less than 3 and rarely explicitly more than 4 [SAA75]). Thus rank can be encoded into 5 bits.

A valid argument against ranks greater than 31 is that any non-degenerate array of rank 'k' must have > or = 2*k elements which for the smallest elements (booleans) represents a full workspace at rank=31 (workspace size=2*31).

The maximum dimension of any axis will be restricted to 2*31 or by the amount of available memory, which ever is less Thus RHO information can be encoded into 32 bit integers. However, very few arrays will have any axes with dimensions greater than 2*15 so they will normally only require 16 bit integers. In many applications dimensions less than 128 are the case. For storage efficiency, two bits will be used to determine the component size of the descriptor itself. This allows 8, 16 and 32 bit integers for the RHO 'nformation contained within.

To differentiate between storage state(SS) descriptors, a single bit will be used. The distinction will be between SS0 and the higher storage states. Since primary memory accesses involve 16 bits, the rank, SS, and data type will be grouped together in a single word. This leaves 8 bits for data typing.

There are four data classes, so two bits are needed to determine which is represented, leaving 6 bits for subclass and division encoding. However only 5 will be used to simplify hardware selection of these bits (they have the same relative position as the rank information which is 5 bits in length). The complete Rank-Type word's layout is shown is figure 4.2.2.

Type – Rank Header

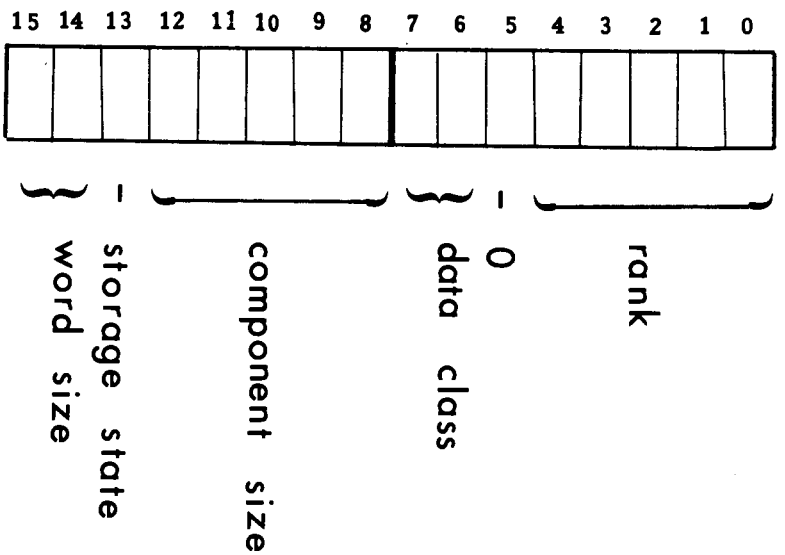


Figure 4.2.2

These 5 bits for component size will encode all the subclass and division information possible. They will solely determine the component bit sizes. Each of the 32 possible patterns available will represent a unique bit size regardless of the data class. This 5 bit number will be an index to a firmware table, inside all units, which maps to a 32 bit number representing component bit sizes. Fiqure 4.2.3 gives the bit sizes and the defined 5 bit codes that represent them. Note it is a simple matter of changing the entries within these firmware tables to effect changes in component sizes. Figure 4.2.4 gives examples of various bit patterns for a few assorted data types.

The bit encodings presented here are some of many possibilites. They represent an attempt to reduce the number of bits required to encode the necessary information without undue complexity in the extraction of this information. It will be shown that in the chapters on the architecture of MAPLE, changes to this encoding will have minimal effect in the structure of the overall machine (due to modularization).

Component Code

					Bit Size
0	0	0	0	1	1
0	0	0	1	0	8
0	0	0	1	1	16
0	0	1	0	0	32
0	0	1	0	1	48
0	0	1	1	0	64
0	1	0	0	1	2
0	1	0	1	0	16
0	1	0	1	1	32
0	1	1	0	0	64
0	1	1	0	1	96
0	1	1	1	0	128

Other codes are unassigned

Figure 4.2.3

Size Code Examples

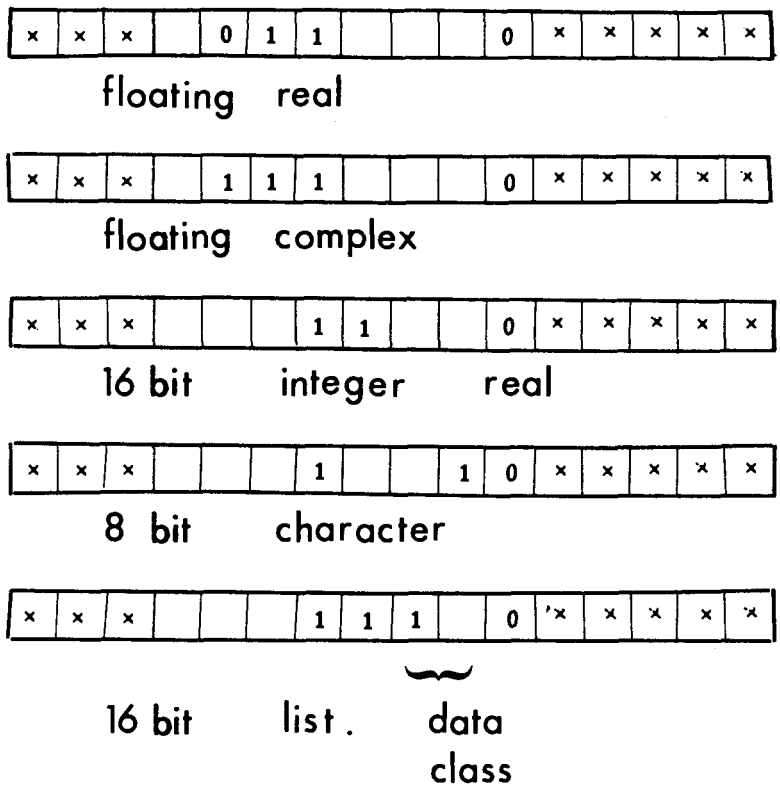


Figure 4.2.4

Within all descriptors, and hence objects, there will be a component which specifies the count of the number of times that object is referenced (Reference Counter) by any other object. This is necessay to support lists and generalized selection operations as described in chapter 2.4. This Reference Counter is simply be an integer value and is sized exactly as the rank-type word. Note that no object can exist which has a reference count of zero.

Also, within every descriptor there will be a unique component which is that object's own reference number. This is, in effect, a back index into any object reference table to facilitate the process of efficient garbage collection (see section 4.1 on memory managment). It also obeys the rule of size that the reference count obeys. The general layout of a descriptor or header is given in figure 4.2.5. It shows that part of a descriptor which is present in all arrays. Figure 4.2.6 shows how SS0 and SS1 arrays differ in their descriptors.

Descriptor Layout

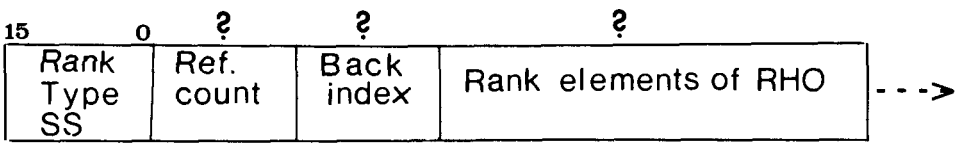


Figure 4.2.5

Descriptor Examples

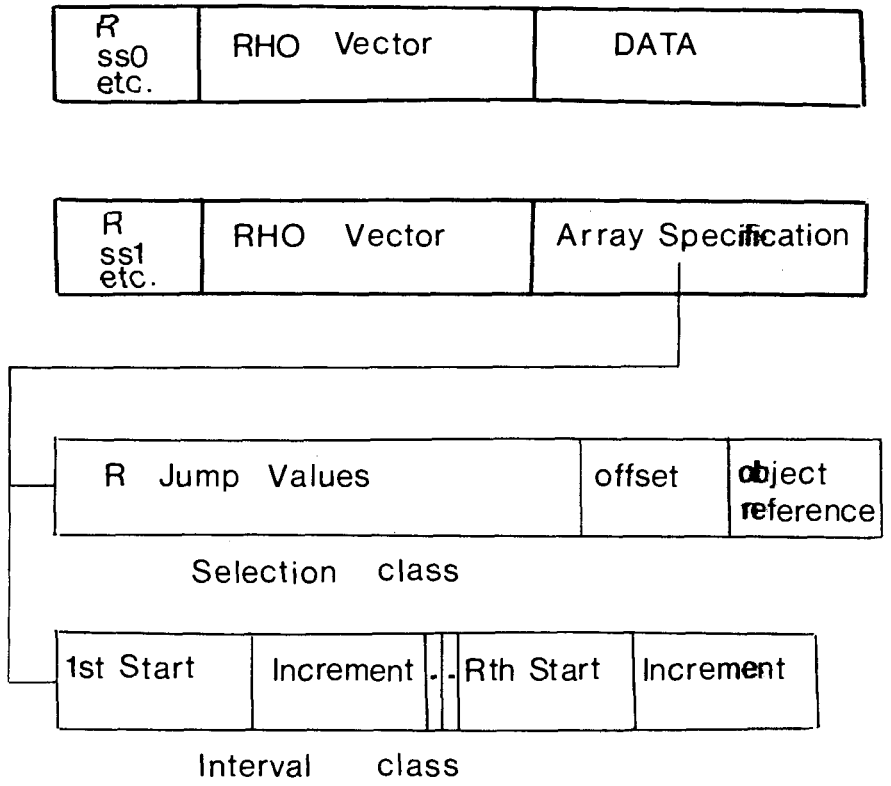


Figure 4.2.6

The workspace is the union of the set of all system objects and all objects generated, either directly or indirectly by the user. Such system objects are the symbol tables and stacks needed to allow the execution of APL statements. The user objects are all the arrays that a user can access or needs to know about. This section discusses more on the organization of the memory space and the structure of the objects themselves.

The workspace can be thought of as consisting of N distinct arrays, each with a descriptor and its associated storage state. All storage state zero (SS0) arrays reference the data which follows immediately after the descriptor. All storage state one (SS1) arrays reference data indirectly, based upon parameters given within the descriptor. For most SS1 arrays these parameters include an indirect reference to another array. To support this level of indirection one of two schemes can be used.

The first has the current address of the other array as a parameter. This presents significant problems as far as memory management is concerned, as the relocation of an object necessitates the modification of all indirect references to it. The problem of simply locating all such references is difficult since there may be more than one indirect reference. One must either dissallow multireferences or disallow addressing of an array via absolute address.

This leads to the second method, which involves storing a pseudo-name in the parameters, rather than the actual address. When it is necessary to locate the indirectly referenced array, the current address is obtained through some binding between this address and the pseudo-name.

This binding is supported through the Array Reference Table (ART). It is essentially a vector of all unique arrays in the system, the components of which are the actual absolute addresses for these arrays, and its indices are the pseudo-names. Figure 4.3.1 shows its structure and use. It is worthy of note that this system of a single reference table for all arrays greatly simplifies the task of pointer updating during garbage collections.

Thus, internally, all arrays are referenced by an integer value which is an index of ART. ART is a system object maintained by the memory manager. It is in the class of dynamically expandable system objects which the Memory Manager maintains. Note the addresses in ART are all 32 bits in length.

Given the above method of representing arrays in the system, the representation of Lists will now become clearer. Since a List is an array of arrays, its data structure is simply an integer array of said array pseudo-names (ie. object reference numbers).

Array Reference Table

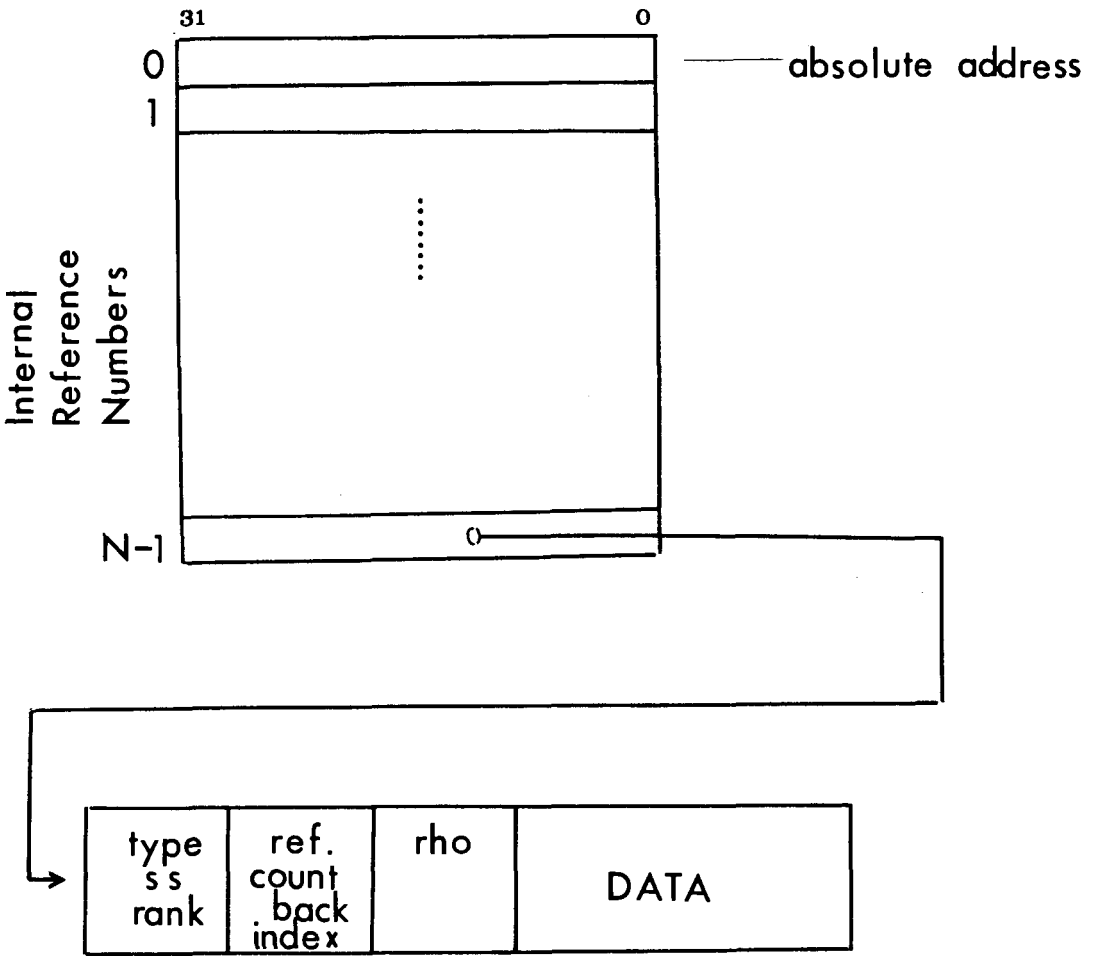


Figure 4.3.1

What is needed now is an association between the user supplied names and these internal pseudo-names. This is supported through the symbol table (ST) which is a list of literal representations, and the symbol association vector (SAV) which is a vector of associated psuedo-names. The chapter on the Execution Unit's operation will describe the use of these system objects in detail. For now, refer to figure 4.3.2 for a structural view of SAV and ST.

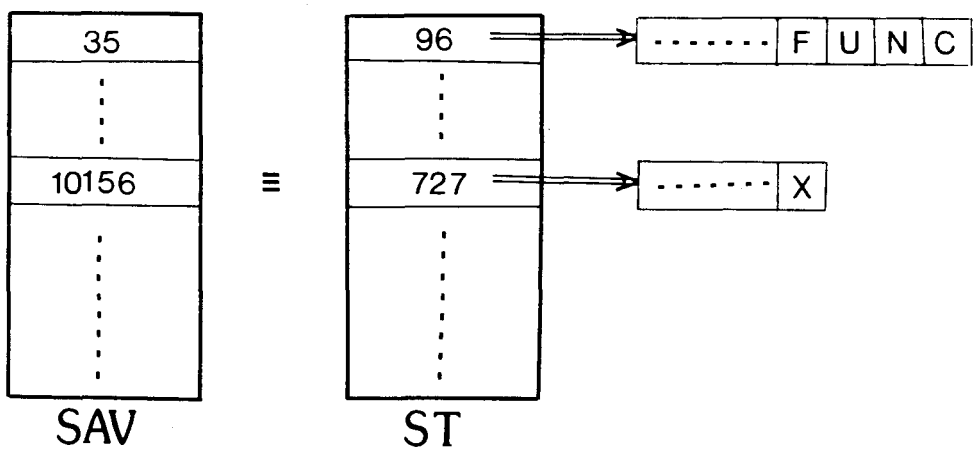


Figure 4.3.2

The heart of MAPLE is the DMU, as it maintains the workspace and allows arbitrary accesses to arrays within. Since the tasks of memory management and array accessing are somewhat independent and can often run concurrently, they shall be delegated to two separate subunits (each having their own processors). These two units are the Memory Management Unit and the Object Manipulation Unit, with the functions of the MMU described in the last chapter.

The allocation of memory for all arrays will be fully transparent to the user (where the "user" is another processor), with the user never needing to know the location or memory requirement of arrays. Therefore all references to arrays will be by names as described in section 4.3. The user will supply a numerical name which directly associates with an array within the DMU.

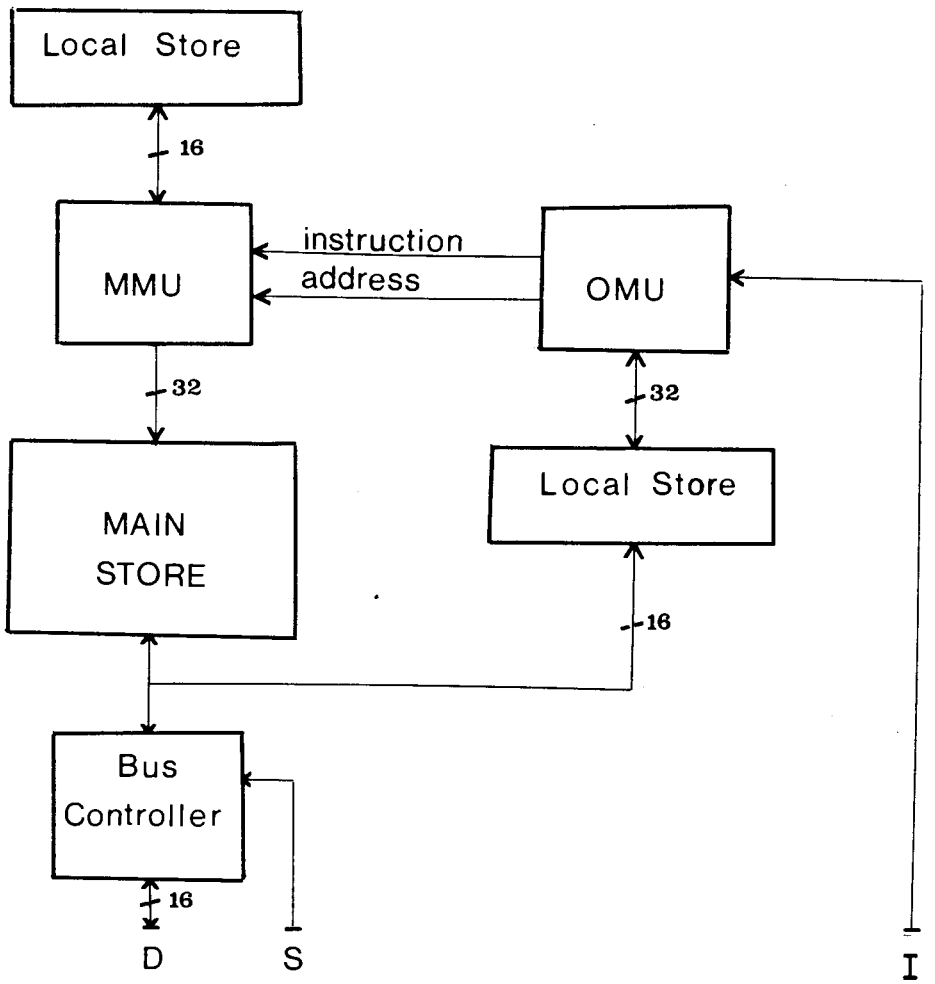
The DMU does not support any explicit associations between literal names at the APL system level. Instead it provides a powerful tool with which to implement many different levels of binding between names. This will be expanded upon in chapter 6 on the EXU's operation.

The objectives of the DMU are as follows: to provide arbitrary access to arrays for the user; support a large virtual memory space greater than 100 megabytes; provide associative lookup of objects; fetch array components at streaming rates equal to memory access speeds; allow multiple data streams to occurr concurrently; allow its instructions to be interpreted at the same time as executing previous ones. The architecture which will satisfy these objectives will now be described.

Each of the DMU objectives will now be discussed in turn, starting with access paths. Provision of general accessing of arrays requires manipulation of a large set of parameters [LAW75],[ABR70]. The DMU accomplishes this with the Object Manipulation Unit (OMU). The reponsibility of the OMU is the generation of absolute virtual addresses for every access desired.

The OMU must be able to generate addresses at least as fast as components are requested. The addresses generated go to the MMU which is responsible for accessing memory. The processes of memory fetch and address generation can be pipelined so that next address generation can occur concurrently with component fetch. It is not my intention to try to achieve optimal DMU performance at this time , but to demonstrate that a high degree of performance can be achieved with minimal hardware design.

Figure 5.1.1 shows the general architecture of the DMU. It shows the two subunits, the OMU and MMU and the other components of the DMU. All nontemporary information is contained within the main memory module except for some data in the MMU's local store. The Bus Controller is responsible for interfacing the DMU to the other units in the system.



Data Manipulation Unit

Figure 5.1.1

Object Manipulation Unit

The OMU is the heart of the DMU's functional power. It provides a number of address generation algorithms for component location. It has the conventional machine design analogy of the MAR (memory address register) and increment circuits. However, it is a few orders of complexity above such a simple function. It is capable of providing bit addresses to locate array components for every type of array access that the APL language requires.

To accomplish this task it requires a large set of internal registers to parameterize these accesses. These registers are loaded based upon the descriptor contained within every array (see sections 2.4 and 4.2), shown in figure 5.1.1 as part of the OMU's local store.

Combined with these access registers is the hardware to do the actual address generation. Figure 5.1.2 shows hardware that can implement the algorithm "AC2" from Appendix 2. AC2 produces sequential addresses into an apparent array from a SS1 descriptor. AC2 is based upon the desire to eliminate all multiplications during the actual address generation process. The performance of the address generation algorithms will be discussed chapter 7.

OMU's Structure

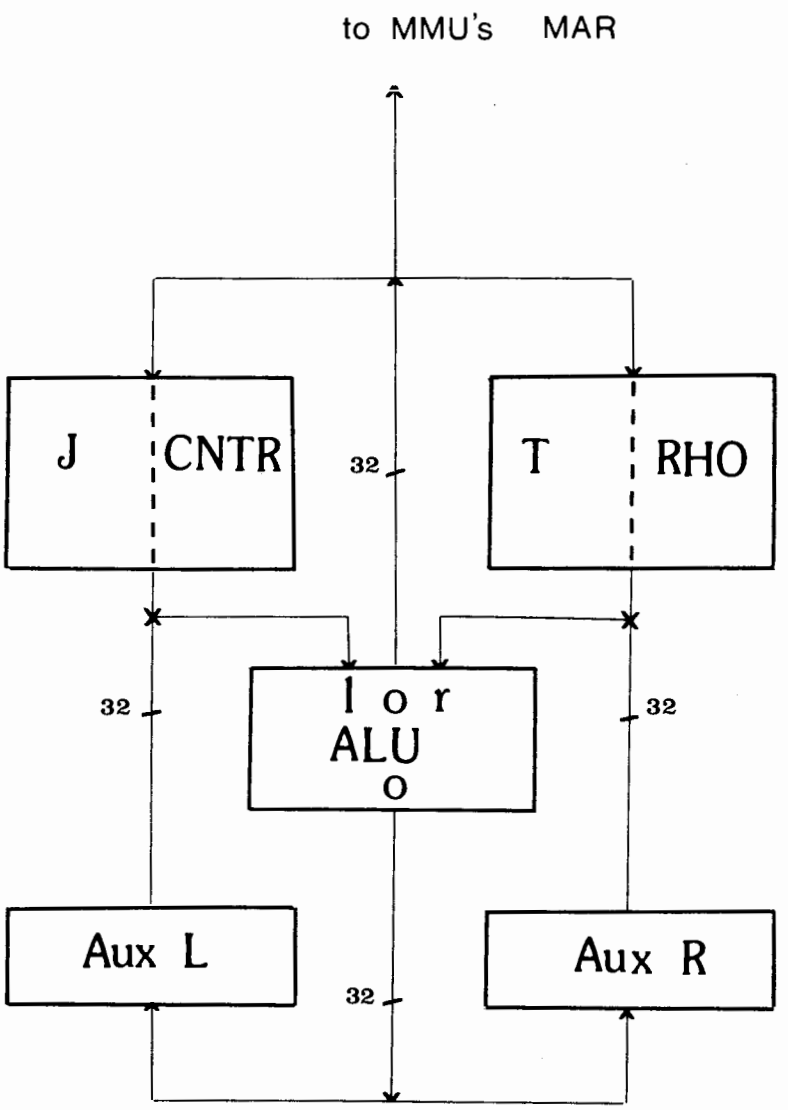


Figure 5.1.2

Not shown in figure 5.1.2 is the hardware to drive this address generation circuitry. A microprogrammed processor which accepts instructions from the instruction bus (c.f. figure 5.1.1) runs the address generation hardware. The exact nature of this processor is the subject of future research and not included in this thesis.

The hardware for address generation uses the register modules called J,CNTR,T,and RHO, contained within the local store of the OMU. They represent the necessary parameters for address generation that the indicated algorithm uses. Each of these modules is a set of registers of 32 words each (32 bit words). The number 32 was chosen because each scalar register of a set represents an axis of an array, that is, the maximum rank is 31 (rank belongs to [0,31]).

These four modules of 4 times 32 words each form an access set of parameters that, once setup, can adequately access an array in any monadic format. There exists one such set for every array that is currently being accessed. The number of arrays that will be currently accessable will be set to 16 for this implementation. This choice is somewhat arbitrary, based on purely physical constraints of local store memory. This memory amounts to 2048 32 bit words (a reasonably small amount).

Of course one can reduce this requirement by reducing the maximum allowable rank and the number of simultaneously accessable arrays. This will probably have no effect upon applications as they are presently devised but in the future the need for more complex inter-relationships between arrays may be in order. It does not, therefore, pay to restrict the design of this system.

The current state of a register set defines the address of the current component accessed. As there are 16 sets it is necessary to switch between sets to access different arrays. The process of this switch can be done via the register file addressing of the OMU's local store. Since only one array is active at any single access (memory is single ported) it is simply a matter of defining the high order address bits of local store to define a register set.

Modification of this address index can be accomplished in a single microinstruction, facilitating the ability to dynamically alter the register set that is active. This allows one to change the active array that is being accessed within a single microcycle. By doing so, the OMU can provide a time division multiplexed sequence of addresses for different arrays.

The above scheme allows multiple data streams to be synthesized over a common bus. It will become apparent that this allows for an efficient parallel processing of array elements. This will be discussed in section 7.1.

Memory Management Unit

Main memory was described in section 4.1 but can be summarized as follows: a large array organized as 1 to 256 million words of 16 bits each. In that same section it was mentioned that all addresses generated are effectively bit addresses within main memory.

The OMU generates bit level addresses by which the MMU accesses array components. cf. figure 4.1.4. The MMU regards the lowest four bits of any address as an index into a 16 bit memory word. The next 12 bits are word addresses into a virtual page. There can be 2*16 such pages. All addresses given to the MMU by the OMU are references to real objects within memory.

Along with the address of the component accessed, the MMU must know the component's size. Since all components are of the same length for any single array, this can be passed in a single transfer. The component sizes were discussed and defined in section 4.2. For sizes of 16 or more bits the problem of bit alignment is trivial. However, for subword sizes the generation of a component over a fixed size bus requires that they be uniformly aligned.

For memory management the MMU has the following objects within its local store: "Hole Table" (HT), "Relocation Vector" (RV), and "Free List" (FL), c.f. figure 5.1.3.

MMU Local Store

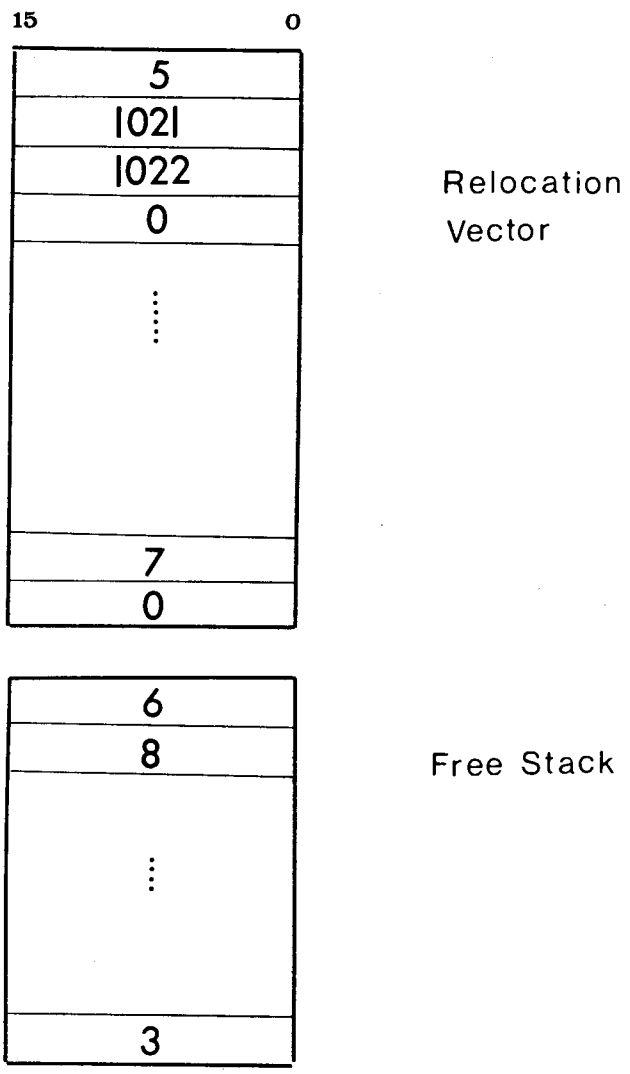
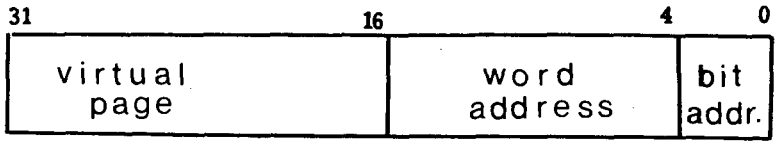


Figure 5.1.3

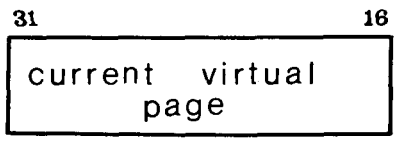
The size of the Hole Table will have a nominal maximum value of 64 with the option of either using smaller or larger tables. This is required as the optimal table size is somewhat application dependant. However, by allowing experimental changes in the HT it may be possible to determine optimal table sizes in differing environments.

The Relocation Vector is the mechanism which maps the virtual address space to the real memory of the system. RV along with FL provides dynamic allocation of system objects with a minimal amount of work, c.f. figure 4.1.5. To elliminate the overhead of having to index RV to obtain the associated real page a single cell associative memory is used, c.f. figure 5.1.4. If the next virtual address is within the current page (given via an equals comparison) then RV is not indexed, otherwise RV must be indexed to obtain the new page.

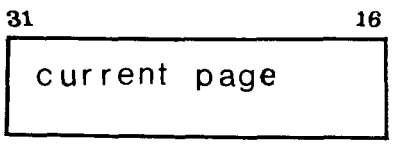
Use of only one cell of associative memory eliminates overhead in accesses within the current page, however, more cells would reduce page boundry crossing overheads. With page sizes of 2*12 words a single data stream is very unlikely to produce many page boundry crossings as most arrays have sizes less than 2*12 words. For multiple streams there would be a strong advantage in using an associative cell for each separate array.



As interpreted by MMU from OMU



CVP



CPR

Figure 5.1.4

Currently, associative memories are not available which are suitable for use in this machine. The prototype of MAPLE will use a single associative cell for RV indexing with the intention of Custom LSI design for such key MMU functions.

The contents of RV are real page numbers and also entries which indicate Black Holes. The value Zero will be used to indicate that no real memory is associated with the virtual page (implying that there does not exist a real page numbered zero).

The Free List (FL) is simply a stack within the MMU which contains the page numbers of all real pages which are totally free. These pages are used to replace Black Holes in the address space, defined by RV, therefore providing dynamic allocation.

The MMU's local store has the same width as main memory. It is possible to provide the MMU's local storage requirements within main memory but this would impose another level of indirection for memory accesses and prevent concurrent memory management. Since the cost of separate memory is almost insignificant compared to the overall design, and is dropping rapidly, it does not make sense to design a system which optimizes this memory useage.

Bus Controller

This is the only other control element of the DMU. It is responsible for the reception and transmission of components over the data bus, coordinating all transfers. Its basic structure will be that of a finite state machine used to synchronize bus signals.

A bus transfer sequence will interchange only a single component regardless of its length. Bit and byte components will be right jusiftied to a 16 bit word, leading bits zeroed. This is simply an implementation constraint to simplify the design. It will have little effect except in the operation of some boolean dyadic functions. Future research may investigate possible ways to improve on this.

The BC provides the handshaking signals necessary to allow any of the units attached to interchange a single component over the data bus. These signals involve strobes to indicate next word and address information concerning which unit is using the bus. A more complete description of the bus protocol is given in chapter 6.

The instructions that the DMU supports are described in this section. Most of the DMU instructions have direct APL equivalence while some are very Operating System like. The direct APL instructions are listed in table 5.2.1 and will be discussed first. They will be looked at in the order that they are presented in the table.

Quad Expunge is a system primitive which releases the object whose name is given as its argument. Since all arrays have a reference count associated with them, this operation first decrements that count. If that count equals zero, the object is non-referenced and the array's storage is released to memory management.

Copy has the effect of generating a new name and copy of the descriptor for X. That is, a totally new name is generated which references the same array as X. Assignment is a form of Copy, except that a new name is not generated. Instead the old array (Z) has its reference count decremented and the name is reassociated with X.

All the rest of the APL operations listed have two modes of operation. Either a descriptor (and hence array) is generated or the operation is parameterized for data streaming (as required for a scalar dyadic operation). These two modes are called the "generate" and "stream" modes respectively.

TABLE 5.2.1

DMU APL INSTRUCTIONS

⎕EX'X'	ERASE X
Z←X	COPY X OR ASSIGN X
Y⍉X	DYADIC TRANSPOSE
⍉X	MONADIC TRANSPOSE
Y⌽X	DYADIC ROTATE
⌽X	MONADIC ROTATE
Y,X	CATENATE
,X	RAVEL
Y↑X	TAKE (WITH OVERTAKE)
Y↓X	DROP
Yρ X	RESHAPE
ρX	RHO
Y/X	COMPRESSION
Y\X	EXPANSION
Y∩X	INDEXING
⊃X	EXPOSE (LIST PRIM)
⊂X	IMBED (LIST PRIM)

Transpose is a pure selection operation which allows the axes of any array to be re-arranged. In no case is any memory required for this operation except that which is needed for a descriptor, if requested. In the case of descriptor generation it will be a storage state one array (refer to section 2.4 for description of storage states).

Catenate always involves memory requests.It involves the generation of three data streams, totally internal to the DMU, which copy arrays X and Y into a new object Z. The result will always be a storage state zero array.

Ravel is a complex primitive whose actual internal function depends upon the storage state of the argument. For any storage state zero array the ravel is a simple storage state one descriptor. The ravel of a storage state one array must be a storage state zero array. The latter is to preserve the ability of selection operations to generate storage state arrays [CDC*GID].

Rotate is also a difficult primitive but for different reasons. The monadic primitive can always be performed by a storage state one transformation. However, the generalized dyadic form can not be so described. If an actual object need be generated, as for assignment, then an SS0 array will be generated. But if one only wishes to access the described array, then the DMU will do the streaming without memory request. The same is true of Ravel: if in the stream mode, only the array need be accessed.

Take is similar to transpose in that for most cases except during overtake it is possible to do an SS1 transformation to achieve the resultant array. In the generation mode it may, however, be necessary to do a memory request and SS0 generation. This is because in some APLs the Take function can be an over-take in which the extra elements are obtained from some fill identity. This DMU will support both Take functional forms (the former is a subcase of the latter).

Drop can always be performed by an SS1 transformation so the user has the choice of either the stream mode or generation mode.

Reshape is in the same class as Ravel, where an SS1 transformation can not always be readily performed on an SS1 array. Therefore the generate mode may create a new object in memory.

Rho always generates a new SS0 array (vector) in the generate mode. The reason for this is as follows: the rho information within every array's descriptor does not have a reference count, so an SS1 array pointing to this data cannot readily be generated. The obvious reason is that the rho vector is so short that it is usually much easier just to copy the data than to reference it (there are a maximum of 31 components to any rho vector).

Compression is a very special DMU instruction. In this implementation, the operation occurs explicitly but it is the contention of some [EDW80-2] that an SS4 array, known as a Sparce Array, synthesized via compression, is a valid extension to an APL system. In the stream mode a specified boolean left argument directs the selection of the elements from the right argument. In the generation mode, at this time, an SS0 array will be created.

Expansion is handled identically to Compression, with the fill element for read operations defined the same as conventional APL (zero for numerics and blanks for characters).

Index is supported as a dydaic primitive with the same form as the other selection primitives. It takes a list vector of imbedded axis indices. It supports both the stream and generate modes. In the latter an SS0 array is generated, as index can not be described via an SS1 transformation.

Expose is a very simple operation. It only operates upon list scalars to return the array imbedded by the list [EDW73].

Imbed is the complement to Expose. It takes any array and generates a list scalar from the result. Both Imbed and Expose have the two modes that the other selection operations have.

The above are the direct APL primitives that the DMU supports. The rest of the instructions are for controlling the memory mode of the operation and for memory management functions. Some of their description follows in these next few paragraphs.

It is necessary to remember that all arrays are accessed via a unique name (which is numerical).An instruction to the DMU must include this name or imply the last array accessed. That is, one can ask for the transpose of an array in the generate mode, then ask for a rotation, without the need to respecify the array.

As described in section 4.1 the DMU supports dynamic allocation for system objects at the hardware level with minimal overhead. Therefore the DMU can provide stack functions for external operations in a LIFO manner. The number of stacks that can be supported have few theoretical constraints. However, at least two should be provided; one for execution control and one for temporary storage.

There does not seem to be any need to provide the ability for array pushes or pops due to the multiple reference capability of reference counts, so stack operations within the DMU are limited to scalar values. The DMU itself only requires scalar stacking provisions.

The DMU will have the following stack operations: Create, which takes two arguments, the maximum depth the stack will ever be (never exceed depth), and stack's component size; Push places a single component on to the stack (which is an argument of the instruction); Pop removes a component from the stack.

There exists one implicit instruction of the DMU that must be mentioned. That is the automatic maintanance of reference counters for arrays. Whenever an array is accessed, via generate mode, and an SS1 descriptor is generated, the SS0 array that is the basic unit of the transformation will have its Ref-count incremented.

Along with the above implicit operation two explicit forms exist. The first was mentioned as Quad Expunge (the dereferencing of an array) and the other is its complement called "Ireference". This instruction has the effect of incrementing an array's reference counter. The usefullness of this instruction is in parameter passing for Call by Value. It allows one to logically specify that a copy of an array has been made, without the need for a physical copy.

There remains just one class of instructions. These are the Descriptor Creation instructions as opposed to descriptor selection instructions mentioned at the beginning of this section. These instructions allow the user to selectively create and modify array descriptors. Using standard rules for descriptor generation one can describe an arbitrary array.

The main purpose of these instructions is to specify complex streaming operations. The best example of this is the synthesis of a scalar dyadic reduction operation. For this APL operation an array is streamed to an ALU but the result is returned into an array of reduced rank but based upon the old array. This is the class of instructions which will take an existing descriptor for an array and produce the desired new descriptor to facilitate reduction.

These instructions fall into two subclasses. The first are just preset algorithms which allow the necessary transformations for the higher complexity APL operations. The best example of which is extended scalar conformability.

The second class allows direct manipulation of the components of descriptors such that the user must provide the algorithm for descriptor formation. The algorithms for the first type are given in appendix 2.

A Complete description of all DMU instructions is given in appendix 3. The format of these instructions will now be discussed.

All instructions operate upon register files which contain valid descriptors for arrays. There may be up to 16 such descriptor registers within the DMU. The need for this many is not obvious but becomes apparent if one is to allow each unit to have simultaneous array accesses.

The instructions from the EXU to the DMU are along a 16 bit bus and may be from 1 to 4 words long. Instruction format allows for a maximum of 64 instructions of which only approximately 32 have been defined.

This chapter deals with the other three units of MAPLE, but first the communications protocols between the units will be discussed.

As shown in figure 3.2.1 there are three buses between the four units. These are the Data, Instruction, and Status buses. All of these buses are bidirectional.

The Data Bus will have the same width as primary memory (16 bits). It is my intention to reduce bus widths so that the units can be condensed into VLSI chips in the future. The Data Bus transcieves components of sizes 1 to 128 bits ,with the words that make up a component exchanged between units, at main memory speed.

In combination with the Status bus, a single component is transfered between units from a source's output queue into the destination's input queue. This exchange untilizes the Data Bus for its entire duration (approximately 100 to 1000 nano seconds depending on the size of the component). The Status bus coordinates the transfer.

The next component transfered over the Data bus may be sourced from any of the four units, thus the Data Bus will be time multiplexed between four sources. This property allows the bus to be utilized at a high efficiency. Since most units cannot process components at bus rates, the bus could be free much of the time.

The effect is that four simultaneous data streams can be overlapped amongst themselves, allowing a high degree of parallelism and for pipelining of instruction execution. Instruction prefetch can then be done during IO or ALU operations.

The direction of all transfers (either in or out of the DMU) is controlled by the DMU via the status bus. Each of the other three units have an output status line (Ready line). A ready line indicates if the associated unit is ready to accept or send data. All three of these signals go into the DMU.

Transfers between output and input queues are controlled by the DMU via 6 output lines (relative to the DMU). The first is the Transfer Direction Line (TDL, a read write equivalent). TDL informs the units if the bus cycle is a DMU read or write operation.

Which unit the next transfer will occur with is given by three address lines called the Logical Unit Address lines (LUA). The transfer can occur beteen the DMU and one of the other three physical units (each of which can contain more than one logical unit).

To initiate a bus cycle, the DMU activates the Cycle Start Line (CSL). The CSL is a strobe signaling the start of a complete component transfer. The transfer is completely synchronous, with both units knowing the word length before the operation begins.

The last DMU status signal is the End Of Stream (EOS) line. When a data stream has finished, this line will be activated along with the unit address for which the stream was associated. This is usually used as an interrupt to the EXU for execution flow control. Figure 6.1.1 shows the expanded bus structure of this machine.

As shown in figure 6.1.1, the Instruction bus is 16 bits wide with five status lines for control purposes. The first two lines are for unit identification (UID) and are generated by the EXU to indicate which unit the instruction is for. They are bidirectional and are used to indicate the unit requesting an interrupt to the EXU. One of the remaining lines is the Instruction Strobe (IS) to indicate to the addressed unit that the Instruction Bus contains a valid instruction.

Bus Signals

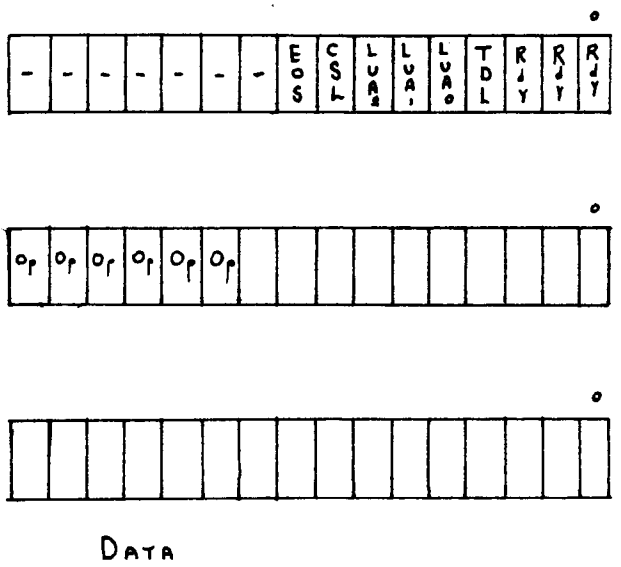


FIGURE 6.1.1

The remaining two lines are to coordinate interupts to the EXU. The first acts as a request for bus and the second is its acknowledgement signal. The three competing units are daisy-chained together. When the EXU acknowledges an interrupt, the appropriate unit will drive the UID lines, and drive the Instruction bus with the data it wishes to pass.

This separation of the buses allows for simultaneous instruction setup and execution. The net effect is a pipelinging of instruction fetch and execute for the four units.

It should be apparent by now that the DMU controls all data transfers between the units, and that these transfers must be between the DMU and some other unit. Thus the units/processors in the machine are coupled together so that no single unit can function with another removed.

The last statement is only partially true. The system cannot interact with the external world unless the IOU 's present and few operations can be performed with out the ALU. How these units interact is the topic of the next section.

An indepth look at the requirements for the other three units in the system will be discussed now. The features and interactions of all units will also be described.

EXU's Structure

It was mentioned in chapter 3 that the EXU was the control unit for program execution. By the appropriate instructions to the other three units a wide variety of array processing tasks can be accomplished.

Via the instruction bus, the EXU can initiate a large class of data transfers and transformations. The instruction sets of the three units allows the EXU to synthesize the complete APL language as described in chapter 1. However, by modification of the EXU, an endless class of array processing languages can be generated. This concept is exemplified by R.Hobson [HOB80-1] in his work on software sympathetic chip design.

Some of the architectural characteristics of the EXU's possible implementation follows. Firstly the hardware structure and then some of the principles of interpretation that the EXU uses.

Since the EXU is primarily intended to interpret APL, its internal structure will reflect this. The current executing line and its parameters are located within the EXU. This code expression is fetched from the DMU as a data stream destined to the EXU (which is usually done while another unit is executing an instruction).

The size of the EXU's local store determines the number of lines which can be cached within itself. It is possible to restrict the maximum length of an APL statement, and many implementations have done so (APL/360, MCM APL). Within MAPLE there will be no restriction on the length of an APL statement, however, the average line length can be used to determine the storage requirement for the EXU.

The average length is in the order of 50 characters [BIN75],[SAA75]. With such a value many lines can be easily buffered within the EXU. This results in significantly less data bus transfers due to the memory heirarchy between DMU and EXU's local store [TAN76].

The internal complexity of the EXU need not be great. Instruction generation can be made table driven and little actual processing of arrays is done by execution control. A conventional 16 bit microprocessor (such as the MC6809) would be adaquate for all its tasks.

The EXU is responsible for all syntax checks on the instruction stream. This is done by analysis of the cached APL statements and interrogation of the DMU over the machine's instruction bus. By pipelining this process with the process of instruction generation, the usual overhead of syntax checking can be reduced.

Most APL systems pre-encode the tokens at edit time so that lexical analysis is greatly reduced during execution [BAT73]. This system must obviously do the same. This is done by the EXU when a function's definition is closed or during the control primitive "Execute" (function token translation is done by Execute internally so that the APL supervisor can be written in APL).

In section 4.3 the two arrays for symbol table maintainance were introduced. The first (SAV) maps locally defined numerical names to global DMU names (the DMU does not support environment hierarchies). These local names are one to one associated with the literal tokens within functions. The second array (ST), the actual symbol table contains the literal values for the tokens.

There may be many versions of both SAV and ST present at one time each representing a different calling environment. To access an array, the EXU looks up its global name in the currently active SAV, then passes this name to the DMU. Since the size of SAV directly relates to the number of local variables, it can be expected to be rather small (in the order of 10 to 100 components). It is therefore reasonable to expect the EXU to buffer more than one version of SAV. Only when a function is entered or exited does SAV need to be accessed from the DMU. This reduces bus overhead and increases overall efficiency.

It should be mentioned that during program execution ST need only be accessed for the "Execute" primitive or to support a parameter/function calling hierarchy. At such times, a number of the accessable ST's must be searched to obtain a token's actual machine name. Currently the search will be done via the "Index" primitive which is an ultra fast linear search.

ALU's Structure

The ALU is the second processor unit that connects to the instruction bus. It is responsible for all scalar functions. Table 6.2.1 ia a complete list of the APL operations which it can perform. All of these operations take, as input, one or two vector data streams, of known parameters, and produce a single output vector data stream.

TABLE 6.2.1

ALU APL INSTRUCTIONS

⍺X AND Y DENOTE INDIVIDUAL SCALAR COMPONENTS OF ARRAYS.

Y+X	∘	+X	
Y-X	∘	-X	
Y×X	∘	×X	
Y÷X	∘	÷X	
Y⌈X	∘	⌈X	
Y⌊X	∘	⌊X	
Y|X	∘	|X	
YoX	∘	oX	
Y*X	∘	*X	
Y⍟X	∘	⍟X	
Y?X	∘	?X	
Y!X	∘	!X	
Y⌹X	∘	⌹X	⍺X Y DENOTE ARRAYS
Y⍳X	∘	⍳X	
Y=X			
Y≠X			
Y<X			
Y≤X			
Y>X			
Y≥X			
Y∧X			
Y∨X			
Y⍲X			
Y⍱X			
Y∈X			
Y⊥X			
Y⊤X			
		~X	

To accomplish many of the more complex instructions, the ALU keeps track of the indices for all data streams it associates with. This simply requires three 32 bit counters that increment on each bus transfer cycle. The ALU performs a reduction operation by knowing the correspondence between the lengths of the input stream to the output stream. This ratio is given via a special instruction to the ALU (from the EXU).

The ratio above relates the number of components that pass into the ALU to the number which leave it. Normally this value defaults to 1 for all scalar dyadic functions. This feature is necessary for "Index" ,"Reductions", and "Membership" primitives.

In chapter 2, the concept of the interval data class was introduced. The interval for numerics allows incredible compression of arrays with the properties so described. The ALU will allow numeric interval operations if the interval array is first transferred into the ALU. This requires a very special instruction, as the other dyadic argument is streamed/loaded separately to the first argument. There must be cooperation between the three units EXU, ALU, and DMU to accomplish this.

The amount of storage, an interval array takes is not significant, as it always has a length less than 64 components. It is therefore reasonable to cache all interval arrays within the ALU and allow highly asynchronous component outputing.

Given in Appendix 2 is an algorithm for the generation of SS2 interval arrays based upon their descriptors. It has not been decided whether or not the ALU or the DMU will generate the array elements. This is because intervals can have floating point formats, which the DMU does not support.

A most important ALU instruction is the data conversion primitive. It is monadic and allows both upwards and downwards length conversions of numerical arrays. At the discretion of the EXU any array can be converted into having either a smaller or larger number of bits per component. This instruction, along with the ALUs ability to monitor numerical overflows and oversizes, allows efficient data storage formats.

IOU's Structure

At present the IOU 's not very well developed. Table 6.2.2 shows all of the APL instructions that it supports. There are few IO functions in the APL language at present, which is reflected in the few instructions the IOU performs.

TABLE 6.2.2

IOU APL OPERATIONS

Y ⍞ *X*	
⍞ *X*	
Y ▼ *X*	
▼ *X*	
⎕*AV*	⍝*SYSTEM VARIABLE*
.....	⍝*YET TO BE DEFINED*
	⍝*IO SYSTEM FUNCTIONS*

However, the DMU was designed to allow a very large expansion in the subdivisions of graphical data. Even though there are few operations presently, there are very good indications that future requirements on computers will necessitate expansion. Since the IOU, and all units, have their own separate sets of instructions, there exists more than adequate room for future instruction encoding.

The temporary stack instructions within the DMU allows the IOU to obtain from the external world, arrays of highly variable lengths. By pushing datum as it is obtained onto this stack, an array whose overall size is not known can be handled. Remember that the DMU will allow dynamic stacks without tying up more memory than the current stack depth. stack requires.

Since arrays that are directed to output are usually of moderate size, the IOU should provide local store for most of the arrays it will need to buffer. For this purpose, between 8 and 64 thousand bytes of storage should be provided in the IOU. At current densities, this amounts to only 2 to 8 ICs, and can even be on chip if the IOU is ever implemented as a single chip unit.

It is my intention that the IOU contains all mapping arrays necessary for IO functions. Such an array is the Quad Atomic Vector, which defines the current character set. As was described in section 4.2, all character/ graphical data is internally represented as positive integers. These arrays are passed to the IOU which does the necessary translation to the various IO devices attached to the system.

A separate IO processor provides a high degree of insensitivity (in the overall system) to changes in IO configurations. This modular approach simplifies the overall design of the machine.

Parallel processing has the potential to greatly increase processing speeds; often increases are linear in the number of processors [MIT74]. Improved modularity of design will also result if tasks are properly divided amongst these processors [SWA77] . It has been demonstrated in this thesis that the execution of APL may be clearly partitioned amongst four major processors, each optimized for its assigned task.

An important consequence of MAPLE's multiprocessor architecture is its inherent modularity. One can employ a structured design philosophy to each of MAPLE's four main processors and debug each via simulators. The net effect is that MAPLE can be brought up "ON AIR" sooner than in more conventional designs.

MAPLE's memory architecture utilizes two co-processing units to implement all array accessing functions needed for APL,along with all workspace functions. These two units make up the DMU which this thesis has concentrated upon.

The DMU meets an objective of this thesis in that it combines all memory functions into a single processing module. A complete set of APL selection functions is implemented within the DMU producing a very "smart" memory machine.

The DMU, in providing all memory functions, essentially reduces the task of implementing an APL system to syntactic and scalar processing. This combined with MAPLE's simple multiprocessor network allows rapid developement of a complete APL machine.

The DMU is designed so that the operation "Beating" is highly developed with the possibility of "Drag Along" (section 2.4). These processes allow APL statements to imply a high degree of array restructuring without actually accessing the arrays in question. Usually one need only access an array directly when a scalar dyadic operation is to be done. This follows since, unless a storage state zero array must be built as a temporary during statement execution, most selection operations can be described via storage state one transformations (see section 2.4).

An important feature of the DMU is its ability to allow multiple simultaneously accessed arrays. This is accompanied by the DMU's ability to accept instructions while current accesses are taking place. This allows individual units connected to the DMU to access arrays at their own speed, without tying up the system bus or the DMU. An important consequence of this is the ability for the units to execute tasks concurrently.

To demonstrate the principle, consider the case of the APL primitive Exponential. It is unreasonable to expect this scalar operation to run anywhere near memory speed. Therefore while this instruction is proceeding, an instruction prefetch can occur along with a set up for the next array operation. The ability to multiplex the DMU's function leads to ease in pipelining.

With these abilities, the DMU can effectively provide virtual array access, at better than memory speed, to any single array or provide any array component, to any unit connected, as fast as requested. The net effect is that APL statements can proceed at the effective scalar unit speeds of the slowest unit or operation that needs to be done.

In Appendix 4, several APL statements are broken down into their steps to show how a conventional system would execute them as compared to their execution using the DMU (and the other units of this system). The steps in these examples show the differences in memory allocation processes during statement execution. The conclusion to be made is that there is a significant reduction in the number of requests to memory when the architecture of this thesis is used.

The performance of the MMU is very dependant upon application, tied to the type, size, and use of arrays by the user. In MAPLE's design for memory management there is very little overhead in processing time for array allocation or de-allocation. MAPLE's architecture allows changes in some of the mechanisms of memory management without affecting the rest of the system so if memory managment becomes troublesome corrections can still be introduced.

The performance of the OMU (array access and transformations) is related to the following properties:

(1) DMU instruction time

(2) Access setup time

(3) Component access rate.

The selection instructions will be based upon the algorithms given in Appendix 2. These have time complexities linear in the Rank of the array operated on. This results in the high performace of array structural operations. Special cases of selection operations do require that all of an array's components be accessed, so performance would be based on (3).

Most of the DMU instructions mentioned in Appendix 3 are either of the form above or are simple instructions only requiring constant time for execution. Such are "Setup", "Copy", "Read" etc. "Setup" triggers access of the specified array for data streaming between units, only requiring that a valid descriptor be loaded into the specified register file. Therefore the access setup time is constant in time.

The only setup instruction which requires a non constant time complexity is "Access", which loads a register file based upon a given array name. Access is linear in Rank as a descriptor must be moved from main memory to local store of the OMU.

The important performance factor of the DMU is (3), the time required to locate and fetch/store the next component. The access algorithm of the OMU will be based upon "AC3" of Appendix 2. Because of AC3's importance its operation will be described fully (for the analysis all time will be normalized to Main Memory Cycle time-MMC with any microinstruction timing approximated as 1/2 MMC).

AC3 is based upon two loops, which are executing concurrently, namely LP1 and LP2. LP2 is responsible for the generation of addresses along the current row (given to main memory for a pipelined memory access). This loop is composed of three microinstructions all of which can be executed in parallel, resulting in each address generation iteration of LP2 requiring only 1/2 MMC (less than the required 1 MMC to keep up with memory).

While LP2 is running LP1 is updating the offset to the next row of the array (if a scalar or vector LP1 never runs). LP1 has the capablity of updating this row offset at address generation rates IF the next row is within the current plane. Thus address generation into Matrices can proceed at memory speeds.

When there is a need to change planes of an array LP1 must perform 1+Rank addition operations to update the row offset. Now as most arrays have Dimensions greater than their Rank this overhead in changing planes will be small compared to time spent within the plane.

When the design of the EXU (section 6.2) was discussed it was mentioned that the EXU's local store would be used for caching executing APL functions and parameters. This reduces the number of memory accesses to a minimum for the interpretation of code strings, and allows the EXU to preprocess instructions. These two factors will guarantee efficient interpretation of the internal APL code.

The actual performance of the EXU can not be given here as its exact internal architecture has not been set. However, given MAPLE's architecture the flexibility to allow even an incremental compiler within the EXU is possible.

The ALU will ultimately be a microprogrammed microprocessor capable of scalar speeds on the order of main memory speeds, however, for prototyping there are many suitable NMOS processors which offer high performance (I432***, MC68000). Some operations (boolean OR etc) can run at better than main memory speeds implying that for these better than 3 million instructions per second rates can be achieved (based on 330 nanosecond memory).

Table 7.2.1 gives the component times for an IBM 4341-L01 running APL. These were obtained by performing the indicated scalar dyadics and measuring the average time to execute for 10000 elements. The average times for all the numeric scalar dyadics is on the order of 10 microseconds (per element). For integer PLUS,MAPLE requires only 6 memory cycles per element which at a modest 330 nano-second cycle time is 2 microseconds per element (a factor of 5 better than the 4341).

The above is only to demonstrate that all general purpose computers are inferior to an array streaming machine as to the efficiency of vector operations. The array processor has only one instruction to execute per array while the general purpose machine may have several instructions to execute per array component.

The IOU will not be discussed as its performance is not important to the study of MAPLE due to its low expected utilization. Since the IOU can cache data for IO it does not present any overhead to non IO APL execution.

TABLE 7.2.1
4341 ALU TIMES (PER COMPONENT)

```
X←⍳10000  ⋄  Y←1000+X

X+Y    =>  8.7E¯6  SECONDS
X-Y    =>  8.3E¯6   ''
X×Y    =>  13.8E¯6  ''
X÷Y    =>  17.2E¯6  ''

X←○X       ⋄  Y←1000×X

X+Y    =>  7.3E¯6 SECONDS
X-Y    =>  7.0E¯6   ''
X×Y    =>  10.9E¯6  ''
X÷Y    =>  16.3E¯6  ''

X←10000⍴10>?100⍴20 ⋄ Y←10000⍴10>?100⍴20

X∧Y   =>   2.5E¯6 SECONDS
X∨Y   =>   2.5E¯6  ''
```

The goal of this thesis was to investigate a possible architecture for a machine capable of efficient APL execution. This task was broken up into a rough description of the overall machine and a indepth study of the functions of its memory architecture.

The result was MAPLE and its four subunits: The EXU, a language executor; The ALU, a scalar arithmetic processor; The IOU, an input output processor; And the DMU which provides all memory functions.

MAPLE's modular architecture allowed the partial separation of the four units' interactions letting concentration fall on the functionality of the DMU. A complete APL workspace environmemt was described within the DMU, made up of memory management and accessing functions.

A complete set of data structures were developed for the DMU to implement the workspace. Included was a system of array descriptors which allows deferring most selection operations until actual data need be accessed. This along with concurrent memory management makes a very efficient memory architecture.

Work still remains in producing an actual functioning DMU. This involves obtaining suitable hardware to implement the algorithms and concepts described in this thesis. It is hoped that a complete DMU could be integrated into a single monolithic silicon chip. In this way a broad set of array processing systems could cost effectively utilize a DMU.

When a complete working DMU has been produced, the next step would be the construction of the scalar ALU. As a substantial number of I.C. manufacturers are currently working on monolithic arithmetic units,it is hoped that this hardware will soon be available.

The remaining tasks in MAPLE's construction are the design and building of the IOU and EXU. It is hoped that satisfactory performance can be obtained through the use of existing 16 bit microprocessors. If it ever becomes practical to produce single ICs for these functions then MAPLE could be realized as a modular four chip set.

APPENDIX 1

THIS APPENDIX GIVES A BRIEF DESCRIPTION OF THE FEATURES OF THE APL LANGUAGE. THE FIRST TABLE GIVES THE SYNTAX FOR EXPRESSION EVALUATION USED IN MOST APL INTERPRETERS, WHILE THE FOLLOWING TABLE GIVES A FEW EXAMPLES OF THIS SYNTAX.

TABLE A.1.1

APL EXPRESSION SYNTAX

EXPRESSION:	NOBJECT CONSTANT NILADIC FUSER MFUNCTION EXPRESSION EXPRESSION DFUNCTION EXPRESSION (EXPRESSION) EXPRESSION [IEXPRESSION]
NOBJECT:	A NAMED OBJECT
CONSTANT:	AN EXPLICIT CHARACTER OR NUMERIC SCALAR OR VECTOR
FUSER:	A USER DEFINED FUNCTION
MFUNCTION:	MONADIC FUSER MONADIC PRIMITIVE SDPRIM / SDPRIM \ SDPRIM /[IEXPRESSION] SDPRIM \[IEXPRESSION] ROTATE [IEXPRESSION]
DFUNCTION:	DYADIC FUSER DYADIC PRIMITIVE SDPRIM . SDPRIM ∘. SDPRIM IDPRIM [IEXPRESSION]
IEXPRESSION:	EXPRESSION IEXPRESSION ;IEXPRESSION NULL (EMPTY EXPRESSION)
SDPRIM:	A SCALAR DYADIC PRIMITIVE FUNCTION
IDPRIM:	AN INDEXABLE DYADIC PRIMITIVE FUNCTION ∩IDPRIM AND SDPRIM ARE DISJOINT SETS

TABLE A.1.2

PROPOSED APL SYNTAX

EXPRESSION:	NOBJECT CONSTANT FUSER MFUNC EXPRESSION EXPRESSION DFUNC EXPRESSION (EXPRESSION) EXPRESSION [IEXPRESSION]
NOBJECT:	A NAMED OBJECT
CONSTANT:	AN EXPLICIT GRAPHIC OR NUMERIC ARRAY
FUSER:	A USER DEFINED FUNCTION
FUNC:	FUSER FUSER [IEXPRESSION] PRIMITIVE PRIMITIVE [IEXPRESSION]
MFUNC:	FUNC FUNC MOP FUNC MOP [IEXPRESSION]
DFUNC:	FUNC FUNC DOPD FUNC ∘DOPD FUNC ⍝'∘' IMPLIES DEFAULT LEFT FUNC ARGUMENT
IEXPRESSION:	EXPRESSION IEXPRESSION; IEXPRESSION NULL (EMPTY EXPRESSION) ⍝IEXPRESSION CAN BE INDIRECTLY ⍝REPLACED BY THE INDEX PRIMITIVE
MOP:	MONADIC SYNTAX MONADIC OPERATORS '/\' ONLY SUCH OPERATORS DEFINED AT PRESENT
DOPD:	DYADIC SYNTAX DYADIC OPERATORS '.' ONLY SUCH DEFINED OPERATOR AT PRESENT

TABLE A.1.3

EXAMPLES OF APL SYNTAX

DYADIC PRIMITIVES

```
(X+Y)*3.1          ⍝ * +
3 1 2 ⍉X⍟Y⍴⍳Z      ⍝ ⍉ ⍟ ⍴
```

MONADIC PRINITIVES

```
*×Y                ⍝ * ×
⍉3 2⍴⍴⍳Z           ⍝ ⍉ ⍴⍳
```

DYADIC FUNCTIONS

(X PLUS Y) DEXP 3.1
MTRANSPOSE 3 2 DRHO MRHO IOTA Z

```
∇Z←L PLUS R
 Z←L+R
∇
∇Z←L DEXP R
 Z←L*R÷1.001
∇
∇Z←L DRHO GEORGE
 Z←L⍴GEORGE
∇
∇Z←MRHO T
 Z←⍴T
∇
```

OPERATORS

```
⍝OPERATORS ARE EITHER UNIVALENT OR DIVALENT
⍝(THE NUMBER OF FUNCS THEY TAKE AS ARGUMENTS)

      +.X          ⍝ . IS DIVALENT
      ∘.*X         ⍝ ∘. IS UNIVALENT
⍝BOTH THE ABOVE MUST BE USED IN A DYADIC
⍝SYNTAX MODE, AND ALL PRIMITIVES MUST BE
⍝SCALAR DYADICS

      +/X          ⍝ / IS UNIVALENT
      ≠\X          ⍝ \ IS UNIVALENT
⍝BOTH THE ABOVE MUST BE USED IN A MONADIC
⍝SYNTAX MODE, AND ALL PRIMITIVES MUST BE
⍝SCALAR DYADICS
```

```
      ⍝ THIS APPENDIX CONTAINS A GENERAL DESCRIPTION OF THE METHOD
      ⍝ BY WHICH ARRAYS WILL BE ACCESSED AND TRANSFORMED. EXAMPLES
      ⍝ OF ALGORITHMS AND THEIR RESULTS WILL BE SHOWN.

      ⍝ FOR THE PURPOSES OF THIS SECTION A SS1 DESCRIPTOR WILL
      ⍝ BE REPRESENTED AS A VECTOR OF (1+2×⍴⍴ARRAY) ELEMENTS.
      ⍝ THE FIRST ELEMENT BEING THE BASE ADDRESS INTO MEMORY
      ⍝ (ASSUMED TO BE WORD ADDRESSABLE,AND ALL COMPONENTS
      ⍝ HAVING WORD LENGTHS, FOR EASE IN DEMONSTATION).
      ⍝ THE NEXT ⍴⍴ARRAY ELEMENTS WILL BE THE ELEMENTS OF ⍴ARRAY,
      ⍝ AND THE LAST ⍴⍴ARRAY ELEMENYS WILL BE THE JUMP VALUES.
      ⍝ THROUGHOUT THIS SECTION ⎕IO=0 FOR SIMPLICITY.

      ⍝EXAMPLE

      XD←0 3 2 2 1
      ⍝XD IS THE SS1 DESCRIPTOR FOR X

X
0 1
2 3
4 5
```

```
      ⍝ 'AC1' IS BASIC ALGORITHM FOR ACCESSING ARRAYS TO PRODUCE A
      ⍝ SEQUENTIAL STREAM OF COMPONENTS FROM MEMORY. IT USES THE
      ⍝ PRINCIPLES OF THE JUMP VECTOR AND '+.×' TO GENERATE VALID
      ⍝ ADDRESSES. HOWEVER, BECAUSE OF THE NEED TO PERFORM ⍴⍴ARRAY
      ⍝ MULTIPLICATIONS TO GENERATE A SINGLE ADDRESS IT IS A POOR
      ⍝ CHOISE AS AN ALGORITHM TO IMPLEMENT IN  MICROCODE.
      ⍝ THIS ACCESS ALGORITHM  TAKES AS ITS ARGUMENT A SS1
      ⍝ DESCRIPTOR AND RETURNS AS ITS RESULT AN ARRAY OUT OF MEMORY
      ⍝ (A GLOBAL) AS DESCRIBED BY THE SS1 DESCRIPTOR.

      ⍝ THERE ARE FOUR GLOBAL VECTORS WITHIN THE ACCESS ALGORITHMS:
      ⍝ 'CNTR,RHO,T,J'  EACH OF WHICH HAVE LENGTHS EQUAL TO THE
      ⍝ RANK OF THE ARRAY TO BE ACCESSED. THESE VECTORS FORM PART
      ⍝ OF THE OMU'S LOCAL STORE. THE ROUTINE 'INIT' INITIALIZES
      ⍝ THEM.

∇Z←AC1 D    ;R;BASE;J;T;CNTR;RHO
 INIT
 ⍝
 ⍝ DC1 MODIFIES T TO INDICATE THE NEXT
 ⍝ COMPONENT AND DECREMENTS CNTR TO CONTROL
 ⍝ THE ACCESS OF THE ARRAY. DC1 RETURNS 1
 ⍝ IFF THE ENTIRE ARRAY HAS BEEN ACCESSED.
 ⍝
LP:→LP⌈⍳~ DC1 R⋄ Z←Z,MEMORY[BASE+J+.×T]
 ⍝
 ⍝ Z IS THE RAVEL OF THE ARRAY ACCESSED.
 ⍝ IT REPRESENTS THE DATA STREAM FOR THIS
 ⍝ ARRAY.
 ⍝
 Z←RHO⍴Z
∇

∇

∇INI
 BASE←D[0] ⋄R←(¯1+⍴D)÷2 ⋄'R IS THE RANK OF THE ARRAY'
 RHO←R↑1↓D ⋄J←(-R)↑D     ⋄'SEPARATE OUT RHO AND JUMP'
 CNTR←RHO  ⋄T←R⍴0        ⋄'INITIALIZE CNTRS AND T'
 R←R-1     ⋄Z←⍳0         ⋄'R INDICATES LAST AXIS'
 ⍝
 ⍝ T IS INITIALIZED TO REPRESENT THE FIRST ELEMENT
 ⍝ OF THE ARRAY'S RAVEL.
∇
```

```
      ⍝ 'AC2' HAS THE PROPERTY THAT WHILE GENERATING ADDRESSES
      ⍝ INTO MEMORY THE ONLY SCALAR FUNCTIONS REQUIRED ARE +,- .
      ⍝ THIS HAS SIGNIFICANT ADVANTAGES AS FAR AS THE MICROLEVEL
      ⍝ HARDWARE NEEDED TO IMPLEMENT THIS ALGORITHM.

∇Z←AC2 D   ;R;RHO;J;T;CNTR;BASE;OFFSET
 INIT
 OFFSET←BASE ⋄'ADDRESS OF FIRST COMPONENT'
 ⍝
 ⍝ T[R] REPRESETS AN OFFSET INTO THE CURRENT
 ⍝ ROW POINTED TO BY 'OFFSET', 'OFFSET+T[R]'
 ⍝ IS THEN THE ADDRESS OF THE NEXT SEQUENTIAL
 ⍝ ELEMENT OF THE DATA STREAM. NOTE THAT 'R'
 ⍝ IS A CONSTANT SCALAR TERM FOR THE NUMBER OF
 ⍝ AXES THE ARRAY HAS.
 ⍝
 ⍝ THE FUNCTION DC2 RETURNS 1 IFF THE ARRAY'S
 ⍝ ACCESSING IS COMPLETE. IT ALSO MODIFIES
 ⍝ BOTH T AND OFFSET TO COMPUTE THE NEXT
 ⍝ ADDRESS.
 ⍝
LP:→LP⌈⍳~ DC2 R⋄ Z←Z,MEMORY[OFFSET+T[R]]
 Z←RHO⍴Z
∇

∇F←DC2 A
 →0×⍳F←A<0                    ⋄'RETURN IF ALL AXES UPDATED'
 T[A]←T[A]+J[A]               ⋄'NEXT OFFSET INTO ROW'
 →0×⍳~F←0=CNTR[A]←CNTR[A]-1⋄'RETURN IF ROW ~FINISHED'
 T[A]←0⋄CNTR[A]←RHO[A]        ⋄'RESET ROW PARAMENTERS'
 F← DC2 A-1                   ⋄'UPDATE NEXT ROW'
 OFFSET←BASE++/T              ⋄'UPDATE OFFSET TO ROW'
∇
```

```
      ⍝ AC3 IS A REFINEMENT OF AC2. IT SEPARATES ADDRESS GENERATION
      ⍝ INTO A VECTOR GENERATION AND AN UPDATE FOR ARRAYS. THIS
      ⍝ ALLOWS EXTREMEMLY FAST ADDRESS GENERATION FOR VECTORS WITH
      ⍝ PARALLEL COMPUTATION OF THE PARAMEMTERS FOR HIGHER RANKS.

∇Z←AC3 D   ;RHO;T;CNTR;J;BASE;ADDRESS;R
 INIT
 ⍝
 ⍝ AC3 HAS TWO NESTED LOOPS, LP2 IN LP1.
 ⍝ LP2 CALCULATES ADDRESSES FOR SEQUENTIAL
 ⍝ ACCESS FOR THE CURRENT ROW OF AN ARRAY.
 ⍝ LP1 DOES THE UPDATES TO ALLOW ACCESSING
 ⍝ OF THE NEXT ROW. THIS IS DONE AT THE
 ⍝ START OF THE LOOP AND AT THE END WHERE
 ⍝ UPDATE IS CALLED. UPDATE RETURNS 0 IFF
 ⍝ THE ARRAY'S ACCESS IS COMPLETE .
 ⍝
LP1:ADDRESS←BASE++/T ∘CNTR[R]←RHO[R]
LP2:Z←Z,MEMORY[ADDRESS]
 ADDRESS←ADDRESS+J[R]
 →LP2⌈⍳0≠CNTR[R]←CNTR[R]-1
 →LP1⌈⍳ UPDATE R-1
 Z←RHO⍴Z
∇

∇F←UPDATE A
 →0×⍳~F←A≥0              ∘'RETURN IF ALL AXES UPDATED'
 T[A]←T[A]+J[A]          ∘'JUMP ALONG CURRENT AXIS'
 →0×⍳0≠CNTR[A]←CNTR[A]-1∘'RETURN IF ~FINISHED CURRENT AXIS'
 T[A]←0∘ CNTR[A]←RHO[A] ∘'RESET PARAMETERS FOR CURRENT AXIS'
 F←UPDATE A-1            ∘'UPDATE NEXT AXIS'
∇
```

```
      ⍝ SOME OF THE ALGORITHMS WHICH FOLLOW WERE FIRST DESCRIBED BY
      ⍝ P.ABRAMS IN HIS THESIS (CHAPTER 3). THEIR BASIC FORM IS THE
      ⍝ SAME BUT THEY HAVE BEEN MODIFIED TO OPERATE ON SS1 DESCRIPTORS.
      ⍝ THESE ALGORITHMS MODIFY SS1 DESCRIPTORS SO THAT AN ACCESS
      ⍝ ALGORITHM CAN PRODUCE THE DESIRED RESULT.
      ⍝ ALL ARGUMENTS ARE ASUMED TO BE WITHIN THEIR PROPER DOMAINS.

⍝DYADIC TRANSPOSE

∇RD←X TRANSPOSE D;RHO;RANK;J;I
 ⍝X IS VALID LEFT ARGUMENT
 ⍝D IS SS1 DESCRIPTOR
 RD←D⋄ RANK←(¯1+⍴D)÷2
 RHO←RANK↑1↓D⋄ J←(1+RANK)↓D
 I←0 ⋄RANK←1+⌈/X              ⋄'THE LRGEST VALUE IN X GIVES RANK'
LP:RD[I+1]←⌊/(I=X)/RHO        ⋄'DETERMINE ITH RHO VALUE'
 RD[I+1+RANK]←+/(I=X)/J       ⋄'DETERMINE ITH JUMP VALUE'
 →LP⌈⍳RANK>I←I+1
 RD←(1+2×RANK)↑RD             ⋄'DROP OFF EXCESS FROM OLD D'
∇

      ⍝EXAMPLES

      AC3 XD ← 0 3 4 4 1
0 1  2  3
4 5  6  7
8 9 10 11
      AC3 1 0 TRANSPOSE XD
0 4  8
1 5  9
2 6 10
3 7 11

⍝MONADIC TRANSPOSE

∇RD← MTRANSPOSE D;RANK
 ⍝D IS SS1 DESCRIPTOR
 RD←D ⋄RANK←(¯1+⍴D)÷2
 RD[1+⍳RANK]←⌽RANK↑1↓D
 RD[1+RANK+⍳RANK]←⌽(1+RANK)↓D
∇

      AC3 MTRANSPOSE XD
0 4  8
1 5  9
2 6 10
3 7 11

⍝MONADIC ROTATE

∇RD←I MROTATE D;RHO;RANK;J
 ⍝I IS AXIS OF ROTATION
 RD←D⋄ RANK←(¯1+⍴D)÷2        ⋄'RD IS THE SAME SIZE AS D'
 RHO←RANK↑1↓D⋄ J←(1+RANK)↓D⋄'EXTRACT RHO, J FROM D'
 RD[0]←D[0]+J[I]×RHO[I]-1    ⋄'MODFIY OFFSET'
 RD[1+I+RANK]←-J[I]          ⋄'MODIFY ITH JUMP VALUE'
```

```
∇
        AC3 0 MROTATE XD
8  9 10 11
4  5  6  7
0  1  2  3

⍝NO OVERTAKES ALLOWED

∇RD←X TAKE D;RHO;RANK;J
 ⍝X IS A VALID LEFT ARGUMENT
 RD←D⍴ RANK←(¯1+⍴D)÷2
 RHO←RANK↑1↓D⍴ J←(1+RANK)↓D
 RD[0]←D[0]+J+.×(X<0)×RHO-|X ⋄'MODIFY OFFSET/BASE'
 RD[1+⍳RANK]←|X               ⋄'REPLACE RHO VALUES'
∇

        AC3 2 ¯3 TAKE XD
1  2  3
5  6  7

∇RD←X DROP D;RHO;RANK;J
 ⍝X IS VAILD LEFT ARGUMENT
 RD←D⍴ RANK←(¯1+⍴D)÷2
 RHO←RANK↑1↓D⍴ J←(1+RANK)↓D
 RD[0]←D[0]+J+.×(X>0)×|X    ⋄'MODIFY BASE/OFFSET'
 RD[1+⍳RANK]←RHO-|X         ⋄'REPLACE RHO VALUES'
∇

        AC3 0 ¯2 DROP XD
0  1
4  5
8  9
```

```
      ⍝ GENERATE IS IDENTICAL TO AC2 IN ITS FORM WITH THE ONLY
      ⍝ DIFFERENCE BEING THAT INSTEAD OF GENERATING ADDRESSES
      ⍝ IT PRODUCES ACTUAL COMPONENTS WITHOUT ACCESSING MEMORY.
      ⍝ A LARGE CLASS OF ARRAYS CAN BE DESCRIBED SO THAT NO REAL
      ⍝ MEMORY IS REQUIRED FOR THEIR STORAGE REGARDLESS OF THEIR
      ⍝ APPARENT SIZES. WHAT IS REQUIRED IS EQUIVALENT TO A SS1
      ⍝ DESCRIPTOR IN FORM.
      ⍝
      ⍝ GENERATE CAN BE USED TO PRODUCE ONLY NUMERIC ARRAYS.

∇Z←GENERATE D;RHO;R;J;BASE;T;OFFSET
 INIT
 OFFSET←BASE
LP:→LP⌈⍳~DC2 R⋄Z←Z,OFFSET+T[R]
 Z←RHO⍴Z
 ⍝ GEN USES THE SAME 'DC2' ROUTINE TO UPDATE
 ⍝ OFFSET AND T AS DOES 'AC2'.
∇

      GENERATE XD←0 4 5 20 2.01
 0  2.01  4.02  6.03  8.04
20 22.01 24.02 26.03 28.04
40 42.01 44.02 46.03 48.04
60 62.01 64.02 66.03 68.04

      GENERATE MTRANSPOSE XD
0.00 20.00 40.00 60.00
2.01 22.01 42.01 62.01
4.02 24.02 44.02 64.02
6.03 26.03 46.03 66.03
8.04 28.04 48.04 68.04
```

```
      ⍝ 'ESC' PERFORMS THE TASK OF EXTENDED SCALAR COMFORMIBILITY
      ⍝ TEST AND GENERATION ON TWO SS1 DESCRIPTORS, GIVEN AS
      ⍝ ARGUMENTS. THE RESULT IS A SINGLE ARRAY BEING THE LAMINATION
      ⍝ OF THE TWO EXTENDED DESCRIPTORS. THE VECTOR I INDICATES THE
      ⍝ AXES OVER WHICH THE EXTENSION IS TO OCCUR.

∇RDS←YD ESC XD;XR;YR;XRHO;YRHO;XJ;YJ;T
 ⍝EXTENDED SCALAR CONFORMIBILITY
 ⍝BETWEEN SS1 DESCRIPTORS YD,XD
 XR←(¯1+⍴XD)⍳2⋄ YR←(¯1+⍴YD)⍳2 ⋄'XR,YR ARE X,Y RANKS RESPECTIVELY'
 XRHO←XR↑1↓XD ⋄ YRHO←YR↑1↓YD  ⋄'EXTRACT RHO INFO'
 XJ←(1+XR)↓XD ⋄ YJ←(1+YR)↓YD  ⋄'EXTRACT JUMP VECTORS'
 T[I]←0       ⋄ T←(XR⌈YR)⍴1   ⋄'I GIVES AXES TO CONFORM'
 ⋄⍎(XR>YR)/'YR←XR⋄ YRHO[I]←1⋄ YRHO←T\YRHO⋄ YJ←T\YJ'
 ⋄⍎(YR>XR)/'XR←YR⋄ XRHO[I]←1⋄ XRHO←T\XRHO⋄ XJ←T\XJ'
 ⍝
 ⍝ THE ABOVE GUARENTEES THAT THE RANKS ARE NOW EQUAL
 ⍝ AND IN THE PROCESS, MODIFIES RHO SO THAT RHO'[I] ARE 1
 ⍝ AND J'[I] ARE 0.
 ⍝
 XJ[((1=XRHO)∧1≠YRHO)/⍳XR]←0  ⋄'RESET JX[KS] WHERE NEEDED'
 YJ[((1=YRHO)∧1≠XRHO)/⍳YR]←0  ⋄'RESET JY[KS] WHERE NEEDED'
 ⍝
 ⍝ NOW THAT THE ARRAYS HAVE THE SAME RANK CHECK FOR
 ⍝ THE RHO VALUES BEING EQUAL OR SOME TO BE '1'.
 ⍝
 →ERR⌈⍳~∧/∨≠1=(≠≠T)/T←((1,XR)⍴XRHO),(1,YR)⍴YRHO
 XRHO←YRHO←XRHO⌈YRHO          ⋄'RHOS ARE MADE THE SAME'
 →0⋄RDS←(2,1+2×XR)⍴XD[0],XRHO,XJ,YD[0],YRHO,YJ
ERR:'ERROR- NOT SCALAR CONFORMABLE'
∇

      XD←0 3 1 1 1
      YD←0 3 4 4 1
      I←0
      YD ESC XD
0 3 4 1 0
0 3 4 4 1
```

```
      ⍝ THESE ARE MISCILLANEOUS FUNCTIONS USEFUL IT CONVERTING FROM
      ⍝ SS0 TO SS1 AND PERFORMING SOME SIMPLE OPERATIONS SUCH AS
      ⍝ RAVEL.

∇RD← RAVEL D;RHO;RANK
 RD←3⍴¯1↑D⋄ RANK←(¯1+⍴D)÷2
 RD[0]←D[0]
 RD[1]←×/RANK↑1↓D
 ⍝ GIVEN A SS1 DESCRIPTOR THAT IS
 ⍝ ESSENTIALLY A VECTOR RETURN A
 ⍝ A NEW DESCRIPTOR OF RANK 1
 ⍝ NOTE THAT NOT ALL SS1 ARRAYS
 ⍝ CAN BE RAVELLED BY THIS METHOD.
∇

∇RD← CONVERT D
 RD←0,D,1↓×\D,1
 ⍝ CONVERTS SS0 RHO INFORMATION
 ⍝ INTO A SS1 DESCRIPTOR.
∇

∇RD←X RESHAPE D
 RD←D[0],X,1↓×\X,1
 ⍝ FOR SS1 ARRAYS WHICH ARE ESSENTIALLY
 ⍝ VECTORS THIS ALGORITHM WILL PERFORM
 ⍝ RESHAPE FUNCTION. THIS ROUTINE WILL
 ⍝ NOT PERFORM CYCLIC RESHAPING.
∇
```

THIS APPENDIX CONTAINS TABLES OF APL STATEMENTS BROKEN DOWN INTO STEPS ILLUSTRATING HOW THE STATEMENTS ARE EXECUTED. EACH STEP IS A SINGLE FUNCTION (EITHER MONADIC OR DYADIC) THAT GENERATES A TEMPORARY ARRAY. THESE ARRAYS CAN BE REAL OR VIRTUAL OBJECTS OR REFERENCES ON SOME STACK DEPENDING UPON THE SYSTEM EXECUTING THE STATEMENTS. THESE DISTINCT TEMPORARIES ARE GIVEN NAMES T1 T2 T3 ETC. AND ARE SHOWN IN COLUMN 1 OF EACH TABLE.

A TABLE'S SECOND COLUMN INDICATES WHETHER OR NOT THE TEMPORARY RESULT REQUIRES MEMORY ALLOCATION (IN SYSTEMS WHICH DO NOT IMPLEMENT SS1 AND REFERENCING OPERATIONS). THE THIRD COLUMN INDICATES THE MEMORY REQUIREMENTS FOR MAPLE'S IMPLEMENTATION. THE TERM 'A' INDICATES THAT A NEW ARRAY RESULTED WHILE 'NA' INDICATES NO NEW ARRAY.

THE TERM 'BOTTLENECK' IS USED WITHIN THE TABLES TO INDICATE THAT THE OPERATION REQUIRES A SIGNIFICANT AMOUNT OF TIME TO EXECUTE.

EXAMPLE

```
(∧⌿(¯1+⍳⍴X)⌽X∘.=Y)/⍳⍴Y
```

IMPLIED	CONVENTIONAL		PROPOSED	
T1←Y	NA		NA	
T1←⍴T1	A		NA	
T1←⍳T1	A	BOTTLENECK	NA	
T2←Y	NA		NA	
T3←X	NA		NA	
T2←T3∘.=T2	A	BOTTLENECK	A	BOTTLENECK
T3←X	NA		NA	
T3←⍴T3	A		NA	
T3←⍳T3	A	BOTTLENECK	NA	
T4←¯1	A		A	
T3←T4+T3	A		A	
T2←T3⌽T2	A	BOTTLENECK	A	BOTTLENECK
T2←∧⌿T2	A	BOTTLENECK	A	BOTTLENECK
T1←T2/T1	A	BOTTLENECK	NA	

```
      ∇EXAMPLE
I←(N,N)ρ1,(N←1↑N)ρ0
```

IMPLIED	*CONVENTIONAL*		*PROPOSED*	
*T*1←0	*A*		*A*	
*T*2←*N*	*NA*		*NA*	
*T*3←1	*A*		*A*	
*T*2←*T*3↑*T*2	*A*		*NA*	
N←*T*2	*NA*	(*RELEASE N*)	*NA*	(*POSSIBLE RELEASE N*)
*T*1←*T*2ρ*T*1	*A*	*BOTTLENECK*	*NA*	
*T*2←1	*A*		*A*	
*T*1←*T*2,*T*1	*A*	*BOTTLENECK*	*A*	*BOTTLENECK*
*T*2←*N*	*NA*		*NA*	
*T*3←*N*	*NA*		*NA*	
*T*2←*T*3,*T*2	*A*	*BOTTLENECK*	*A*	*BOTTLENECK*
*T*1←*T*2ρ*T*1	*A*		*NA*	

```
      ∇EXAMPLE
(1<ρρN)∨1 2∧.≠×/ρN
```

IMPLIED	CONVENTIONAL		PROPOSED	
T1←N	NA		NA	
T1←ρT1	A		NA	
T1←×/T1	A	BOTTLENECK	A	BOTTLENECK
T2←1 2	A		A	
T1←T2∧.≠T1	A		A	
T2←N	NA		NA	
T2←ρT2	A		NA	
T2←ρT2	A		NA	
T3←1	A		A	
T2←T3<T2	A		A	
T1←T2∨T1	A		A	

```
      ∇EXAMPLE
      Z←Y/1 3↓X+⍳X
```

IMPLIED	CONVENTIONAL		PROPOSED	
T1←X	NA		NA	
T1←⍳T1	A	BOTTLENECK	NA	
T2←X	NA		NA	
T1←T2+T1	A	BOTTLENECK	A	BOTTLENECK
T2←1 3	A		A	
T1←T2↓T1	A	BOTTLENECK	NA	
T2←Y	NA		NA	
T1←T2/T1	A	BOTTLENECK	NA	
T2←Z	NA		NA	
T1←T2-T1	A	BOTTLENECK	A	BOTTLENECK

All DMU instructions have one of the following forms:

```
CODE Rd,Rs,m,m; D
CODE Rs,m,m; D
```

where CODE is a 6 bit encoding of the instruction (within the 16 bits of the instruction). Rd,Rs specify one of 16 register descriptor files (holding a complete SS1 descriptor). Rd is both a source and destination for the operation, while Rs is usually only a source. The two flags 'm' specify optional modes of operation that an instruction may have. The parameter 'D' specifies any optional words of information that may be necessary and which can't be specified within the instruction word.

[1] COPY Rd,Rs

The descriptor in register Rs is copied into Register file Rd.

[2] SETUP Rs,ss,u

The array specified by register Rs is activated in to the stream mode. ss-selects wether the array is a sink or source of data. u -selects which unit the data exchange will take place with.

[3] ACCESS Rd;N

Rd is the register to which the descriptor for the array is loaded into. N- is the name of the array that access is requested for. Returns the Rank-Type word for the array.

[4] SCALAR CONFORM Rx,Ry; I

Registers Rx,Ry are modified (if possible) to reflect extended scalar conformability of their respective arrays. I- specifies any additional axis parameters required if Rank X does not equal Rank Y.

[5] NAME Rs

Returns the name for the array indicated by register Rs. The DMU always maintains a naming for all arrays.

[6] ALLOCATE Rs

Register Rs implies an array so the DMU allocates storage for an array similar to Rs. It does not matter if Rs is a valid descriptor for an assigned array. A new array will always be allocated.

[7] READ Rs;I

Returns the Ith component of the descriptor Rs.

[8] WRITE Rs;I,D

Modifies the Ith component of Rs with the data given following index.

[9] REDUCTION Rd,Rs;I

Register Rd is made to represent the modified descriptor from Rs for the reduction operator. I- specifies the axis of the reduction.

[10] MTRANSPOSE Rs

Register Rs is modified to reflect a monadic array transpose operation.

[11] MROTATE Rs;I

Register Rs is modifies to reflect a monadic array rotation operation along the axis given by I.

[12] RAVEL Rs,sm

Register Rs is modified to reflect a ravelling of the array Rs specifies. sm- indicates wether or not the operation need produce an new array or not, ie. if a streaming is to take place or if a valid descriptor will be required.

[13] RHO Rs

Register Rs is modified to specify a new array given by Rho Rs. This new array will always fit within a register files confines so unless requested no new storage allocation will be performed.

[14] EXPOSE Rs

Rs is replaced with a new descriptor for the array indicated by the List scalar associated with Rs.

[15] IMBED Rs

Rs is replaced with a new descriptor representing a list scalar for the imbed of the old array associated with Rs.

[16] PUSH Rs

The NAME for the array associated with Rs is pushed onto the execution stack (which is internal to the DMU).

[17] POP Rd

The register file Rd is loaded with the descriptor for the array whoses NAME is popped off the execution stack.

[18] OUTER PRODUCT Rx,Ry

The registers Rx and Ry are modified to reflect the necessary structural transformations for the outer product operator.

[19] IREF Rs

The array specified by Rs will have its internal reference counter incremented. This operation is necessary for completeness.

[20] DREF Rs

The array specified by Rs will have its internal reference counter decremented. A count of zero will release that array's storage.

[21] DTRANSPOSE Rd,Rs

The array specified by Rd is transposed according to the vector specified Rs. The resulting descriptor is placed into Rd.

[22] DROTATE Rd,Rs,sm;I

The array specified by Rd is rotated along axis I according to the array given by Rs. The resulting array is associated with Rd. sm- selects wether a new array need be generated or if the data is to be immeadiately streamed.

[23] TAKE Rd,Rs

The array specified by Rd is selected from (Take operation) according to the vector associated with Rs.

[24] DROP Rd,Rs

The array specified by Rd is selected from (Drop operation) according to the vector assocaited with Rs.

[25] CATENATE Rd,Rs;I

The arrays implied by Rd and Rs are catenated together (lamenated) to form a new array. I- specifies the axis over which the operation takes place.

[26] COMPRESS Rd,Rs,s

The array given by Rd is compressed by the array given by Rs. Two modes exist. The first generates a new array which is associated with Rd. The second simply allows one to defer generation. This allows the selection operation to occur logically without storage allocation. The latter is required to allow assignment into a compression expression.

[27] EXPAND Rd,Rs,s

This operation is identical to Compression in all aspects except that the operation is a logical expansion.

[28] INDEX Rd,Rs,s

This instruction has two modes given by "s". The first performs an index of Rd's array via the index list specified by Rs. The second generates an Index Set for Rd's array based upon Rs's. An Index Set is a set of valid indices into an array. Here a new array is generated being isomorphic to the desired array. An Index Set is an internal DMU type which the user never sees. This latter mode allows assignment into generalized index expressions.

[29] RESHAPE Rd,Rs,s

Rd's array is reshaped according to Rs's vector. The flag "s" inicates if an index set should be generated , if a new real array should be generated, or if the array should only be setup for streaming. The former allows assignment into a reshape expression.

[30] STACK ALLOCATE cs;S

The DMU will support at least one temporary stack for other units to use. This instruction allocates a stack of maximum depth S with component size cs.

[31] TPUSH ;D

The data in the instruction is pushed onto the temporary stack.

[32] TPOP

Returns the last component pushed onto the temporary stack.

Abrams,P.S. 'An APL Machine'
PhD Thesis, Stanford University
SLAC report no. 114, 1970

Abrams,P.S. 'Whats Wrong with APL'
APL75 Congress Proceedings, page 1
ACM-STAPL 1975

Allen,E.F. 'A Formal Definition of APL Syntax'
APL75 Congress Proceedings, page 15
ACM-STAPL 1975

Amram,Y. deCosnac,B. Granger,J.L. Smoucovit,A.
'An APL Interpreter for Mini-computers
a microprogrammed APL machine'
APL73 Congress Proceedings, page 33

Batcher,K.E. 'STARAN Parallel Processor
System Hardware'
National Computer Conference, 1974

Battarel,G. Delbreil,M Tusera,D.
'Optimized Interpretation of
APL Statements'
APL73 Congress Proceedings, page 49

Bingham,H 'Content Analysis of APL
Defined Functions'
APL75 Congress proceedings, page 60
ACM-STAPL 1975

Breed,L.M. Lathwell,R.H.
'The Implementation of APL/360'
ACM Symposium on Experimental Systems
for Applied Mathematics
Academic Press Inc, N.Y., 1968

Breed,L.M. 'Generalizing APL Scalar Extension'
APL Quote Quad , March 1971
ACM-STAL

Brown,J.A. 'Evaluating Extensions to APL'
APL79 Conference Proceedings, page 148
ACM-STAPL 1979

CDC*STAR100 'Features of the STAR-100'
Control Data Corporation
Advanced Studies Division
4201 North Lexington
Saint Paul, Minnesota

CDC*APL — 'CDC*APL Reference Manual'
Control Data Corporation

CDC*GID — 'General Implementation Details'
Control Data Corporation

Edwards,E.M. — 'Generalized Arrays(lists) in APL'
APL73 Congress Proceedings, page 99

Edwards,E.M. — 'APL in the Classroom'
1980 APL Users Meeting
Toronto, Canada
I.P. SHARP Associates

Falkoff,A.D. — K.E.Iverson, E.H.Sussenguth
'A Formal Description of System 360'
IBM System Journal
Vol 3 #3. 1964, page 193

Falkoff,A.D. — Orth,D.L.
'Development of an APL Standard'
APL79 Conference Proceedings, part 2
ACM-STAPL 1979

Ghandour,Z. — 'A Simple Approach to the Empty
Generalized APL Arrays'
APL76 Conference proceedings, page 178
ACM-STAPL

Giloi,W.K. — Berg,H.
'STARLET- An Unorthodox Concept
of a String/Array Computer'
Information Processing 1974
North Holland Publishing Company

Gull,W. — 'Recursive Data Structures and
Related Control Mechanisms in APL'
APL76 Conference Proceedings, page 201
ACM-STAPL

Haegi,H. — 'The Extension of APL to Tree-like
Data Structures'
APL Quote Quad
Vol 7#2 Summer 1976, page 8

Hobson,R.F. — 'Software Sympathetic Chip Set Design'
National Computer Conference, 1981
Vol 50 May 1981, page 3

Hobson,R.F. 'Structured Machine Design: An Ongoing Experiment' Proceedings of 8th Annual Symposium On Computer Architecture May 1981, page 37

Iverson,K.E. 'A Programming Language' John Wiley & Sons Inc. New York, 1962

Iverson,K.E. 'The Role of Operators in APL' APL79 Conference Proceedings, page 128 ACM-STAPL 1979

Jenkins,M.A. Michel,J. 'Operators in an APL Containing Nested Arrays' APL Quote Quad Vol 9#2 December 1978, page 8

Jenkins,M.A. 'On Combining the Data Structure Concepts of Lisp and APL', 1980 Queen's University Department of Computing & Information Science Technical Report No. 80-109

Johannsen,D. 'Our Machine, A Microcoded LSI Processor' MICRO-11 Workshop Proceedings, 1978 ACM-SIGARCH ACM Special Interest Group on Microprogramming

Johnston,R.L. 'The Dynamic Incremental Compiler of APL/3000' APL79 Conference Proceedings, page 82 ACM-STAPL 1979

Lawrie,D.H. 'Access and Alignment of Data in an Array Processor' IEEE Transactions on Computers Vol C-24 #12, December 1975

McDonnell,E.E. 'Complex Floor' APL73 Congress Proceedings, page 299

MCM
'MCM/900 USER'S MANUAL'
Manual No. 018-0053
December, 1978
MCM Computers Ltd.
6700 Finch Avenue West
Suite 600
Rexdale, Ontario
M9W 5P5

Mebus,G.
'Laminar Extension: An Overlooked Capability and the Search for its Proper Home'
APL79 Conference Proceedings, page 36
ACM-STAPL 1979

Mitchell,J.
Knadler,C. Lunsford,G. Yang,S.
'Multiprocessor Performance Analysis'
National Computer Conference, 1974

More,T.
'The Nested Rectangular Array as a Model of Data'
APL79 Conference Proceedings, page 55
ACM-STAPL 1979

Murray,R.C.
'On Tree Structure Extensions to the APL Language'
APL73 Congress Proceedings, page 333

Pierre,M.
Pierre,P.
'GESOP: A Relational Data Base Using Generalized Arrays and Data Base Primitives'
APL79 Conference Proceedings, page 102

Penfield,P. Jr.
'Proposal for a Complex APL'
APL79 Conference Proceedings, page 47
ACM-STAPL 1979

Ruggiu,G.
Aigrain,Ph.
'Description of APL Operators'
APL73 Congress Proceedings, page 401

Saal,H.J.
Weiss,Z.
'Some Properties of APL Programs'
APL75 Congress Proceedings, page 292
ACM-STAPL 1975

Samson,D.
Reynaud,Y.
'Storage Management in APL Machines'
APL Quote Quad
Vol 10#2 December 1979, page 19

Samson,D. These Docteur
'Modele Une Machine A.P.L.'
Universite Paul-Sabatier de Touluse
(sciences) Oct 1979

Stritter,S. Tredennick,N.
'Microprogrammed Implementation
of a Single Chip Microprocessor'
MICRO-11 Workshop Proceedings, 1978
ACM-SIGARCH

Swan,R.J. Fuller,S.H. Siewiorek,D.P.
'CM*- A Modular, Multi-microprocessor'
National Computer Conference, 1977

Sykes,R.A. Jr. 'Efficient Storage Management in APL'
APL79 Conference Proceedings
ACM-STAPL 1979, page 226

Tang,C.K. 'Cache SYstem Design in the Tightly
Coupled Multiprocessor System'
National Computer Conference, 1976

Tokoro,M. Watanabe,T. Kawakami,K. Sugano,J. Noda,K.
'PM/II- Multiprocessor Oriented
Byte-sliced LSI Processor Modules'
National Computer Conference, 1977

VANGARD 'APL/Z80'
Quick Reference Quide
VANGARD System Corporation
6812 San Pedro
San Antonio, Texas
78216

Vick,C.R. 'PEPE Architecture- Present and Future'
National Computer Conference, 1978

Vasseur,J.P. 'Extension of APL Operators to
Tree-like Data Structures'
APL73 Congress Proceedings, page 457

APL73 Congress Proceedings
Copenhagen, Denmark, 1973
North Holland Publishing Company

APL75 Congress Proceedings
Pisa, Italy, 1975
ACM-SIGPLAN Technical Committee on APL
ACM, New York

APL76 Conference Proceedings
Ottawa, Canada, 1976
ACM-STAPL, New York

APL79 Conference Proceedings
Rochester, New York, 1979
APL Quote Quad Vol 9#4
ACM, New York

APL Quote Quad
ACM
1133 Avenue of the Americas