On the Feasibility of Implementing a Rule-Based System Using a Data Flow Architecture

by

Peter Craig John Graham

A thesis
presented to the University of Manitoba
in partial fulfillment of the
requirements for the degree of
MASTER OF SCIENCE
in
Computer Science

Winnipeg, Manitoba

ON THE FEASIBILITY OF IMPLEMENTING A RULE-BASED SYSTEM USING A DATA FLOW ARCHITECTURE

BY

PETER C.J. GRAHAM

A thesis submitted to the Faculty of Graduate Studies of
the University of Manitoba in partial fulfillment of the requirements
of the degree of

MASTER OF SCIENCE

ABSTRACT

This thesis discusses the problems associated with both Dataflow and Rule-Based systems and then introduces the idea that an amalgamation of the two concepts may be exploited to solve some of these problems. Furthermore, a prototype design is discussed as well as the question of extensibility to a general parallel system.

ACKNOWLEDGMENTS

The author would like to recognize and thank the persons who have contributed to this thesis. First and foremost of these must be his thesis advisor Dr. M.S. Doyle of the University of Manitoba. Dr. Doyle has provided many valid criticisms and suggestions that have proven to be of immense help. The author must also gratefully acknowledge Dr. Forbes J. Burkowski of the University of Waterloo for instilling in him an initial interest in parallelism and dataflow. Finally, a word of thanks must be given to his parents for their help and encouragement during the time this thesis was being prepared, to Ms. Patricia G. Bessler for her proof-reading and moral support and to Michael J. Rogers for his help with APL.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter I
INTRODUCTION

1.1 DATAFLOW COMPUTERS

This section describes the concepts behind dataflow processors and reviews some of the current work being done on dataflow machines. The information represents a summary of research relevant to this thesis.

1.1.1 Concepts

Dataflow, as the name implies, is a system in which program execution is determined by the flow of data. Unlike common Von Neumann architectures, dataflow machines are free to execute an instruction as soon as its operands have been "properly" defined (i.e. have had values assigned to them). This means that many instructions from a single program may be executing simultaneously. In fact, these instructions may be from distinct sections of the program, and thus the existence of a location counter in a dataflow machine is obviated.

In a dataflow system there is no concept of a current instruction. Instead, every instruction is either "enabled" or "disabled". An instruction is said to be enabled (and is therefore eligible for execution) once its input operands have been defined. This mechanism may be represented using a flow graph such as the one shown in Figure 1. In this example, once 'A' and 'B' have been assigned values the operation '+' may be performed on them to yield the result

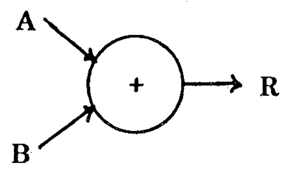


Figure 1: A Flow Diagram Node

'R'.

A dataflow program is composed of many such nodes with each one forwarding its results to others (see Figure 2). Thus, when one instruction completes and produces a result, it may implicitly enable additional instructions. It is this implicit enabling of instructions that defines the order of execution of a program's instructions and thereby effectively eliminates the need for conventional control structures.

There are two critical observations to be made at this point. The first is to reaffirm that instruction execution is affected only by procedural constraints. An

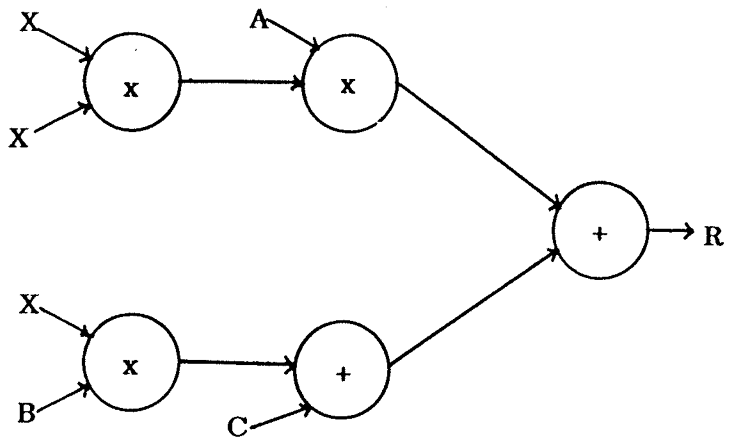


Figure 2: A Flow Graph to Calculate: ax^2+bx+c

instruction's physical location within a dataflow program does not affect its order of execution relative to the other instructions. The second is that the sequence of statement execution may be a non-deterministic process. Undoubtedly, the implementation will place certain restrictions on the number of concurrent operations. Therefore, at some point in time, there may exist a queue of enabled but non-executing instructions. The order of execution within this queue (i.e. of enabled instructions) is not guaranteed.

From a hardware viewpoint, a dataflow system may be seen as having the structure given in figure 3. In this diagram, the processing elements (P.E.s) may be thought of as forming an A.L.U. (Arithmetic/Logic Unit), the activity store as forming a memory, and the fetch and update units as forming the control unit.

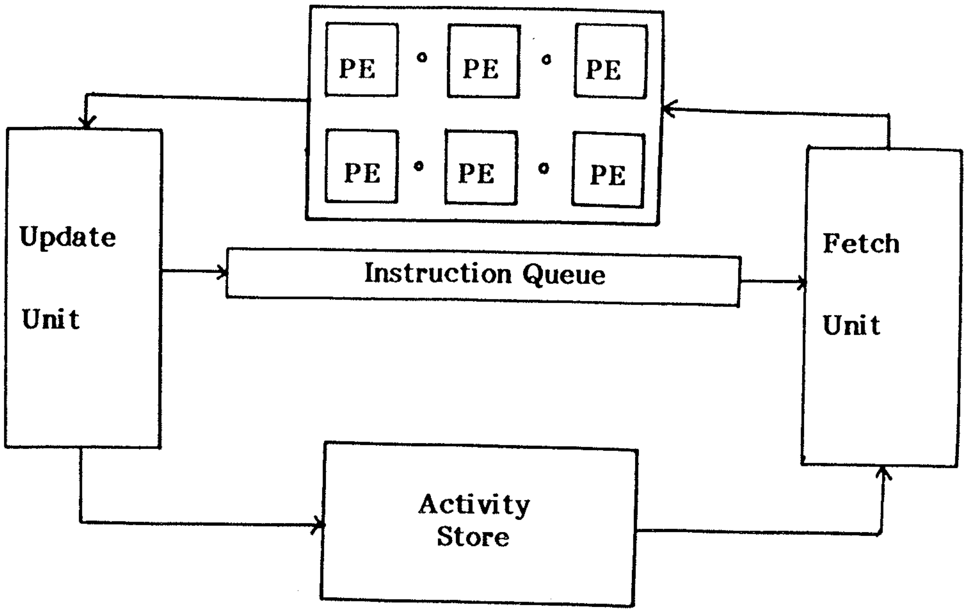


Figure 3: A Dataflow System

The activity store contains current instructions that will be executed when new values are available for their operands. Each instruction may be stored in the form of a template which contains an opcode, storage for any required operands, and a forwarding address for the result. Figure 4 gives the structure of a template for an instruction with

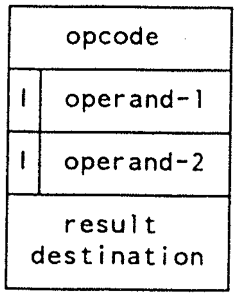


Figure 4: An Instruction Template

two operands and a single result.

Note the indicator bit(I) associated with each operand field. As an operand is copied into this field, the appropriate indicator bit is set. These bits may then be used to determine eligibility for execution. Constants involved in operations are pre-loaded into templates with their indicator bits set. This guarantees that a template does not wait for a constant to become defined.

Enabled instructions, as determined by the update unit, will have their identifiers/addresses placed in the instruction queue (I.Q.). The fetch unit will then select templates identified within the queue and retrieve them from the activity store, to be executed by the system's functional units (D-units).

The only special-case processing to be considered occurs during the initiation of a dataflow program. When a program begins execution there have been no previous instructions to enable the mainline. This means that each dataflow program must have an initial state associated with it. The desired effect may be achieved simply by specifying a single instruction whose identifier is loaded immediately into the instruction queue. This instruction is then used to enable other instructions in the program.

Clearly, by applying the dataflow approach, any available parallelism in an algorithm may be exploited in order to maximize its execution speed.

1.1.2 Current Work

Research currently being done in the field of dataflow falls into two basic categories:

- Designing Dataflow Machines
- Programming Dataflow Systems

There has been considerable progress made in the first of these two areas. J.B. Dennis and his group at M.I.T. (among others) have been successful in the design and implementation of prototype dataflow processors.

The second area, however, has seen far less progress. There are many difficulties when programming in a fully parallel environment. This suggests that programming should continue in a serial fashion, and efforts should be concentrated on translating serial algorithms to parallel ones. Unfortunately, this may also be an impractical solution. What is required is a compromise between the two approaches. By avoiding the need for the programmer to specify the low-level parallelism, it is possible to make programming easier. This leaves only the higher-level parallelism up to the programmer, and simplifies the translation process. Thus, what will eventually be proposed is a "control structure free" functional language for the description of parallel algorithms. Before doing this, however, it is worthwhile to examine some of the work which has already been done.

Most dataflow processors have been designed to execute programs which are expressed in terms of program flow graphs. Not only is this an exceedingly difficult and tedious way of expressing a program, but it also tends to produce machines which are restricted to the template implementation. As will be seen later, this may not be the most efficient implementation possible.

This is not the most serious problem with the simplistic dataflow architecture as discussed. The classic dataflow machine shown in Figure 3 is known as a circular pipeline. That is, each unit is concurrently active and is thus analogous to a stage in a pipeline. (It is circular for obvious reasons.) Ultimately, the level of concurrency in such a system is limited by the bandwidth of the interconnections between the various components in the pipe. Specifically, the data paths to and from the activity store are bound to be overloaded even if a high speed memory is used.

The basic mechanism may be drastically improved upon by linking many such units into a "dataflow multiprocessor" as described by Dennis [13]. This yields a group of dataflow processing elements on what is, effectively, a packet-oriented network. The communications network provides the ability for any processing element to access the activity store of any other processing element. Thus, the activity stores together form a common, global address space with access arbitrated by the communications network. In this scheme, the dataflow program is distributed between the various processing elements in such a way as to minimize the more expensive non-local memory references. The number of functional units per processing element is decreased and the number of activity stores is increased, thus eliminating some of the bandwidth problems. Obviously, the limiting component is now the communications network. While it is generally inappropriate to implement something as elaborate as a crossbar switch, a good network which supports concurrent traffic is required.

A second architecture, also proposed at MIT, takes a somewhat different approach. In the dataflow multiprocessor architecture (above) not all packets are equally accessible to all processing elements. This does not necessarily degrade performance but it does complicate the hardware by producing a system where access times are non-uniform. In the "cell block" architecture, however, all templates are made equally accessible to all processing elements. Thus, access times to all templates by all processors are consistent. The cell block architecture would seem to imply an activity store containing many templates, each of which may then be selected by the distribution network to be forwarded to some operation unit. This approach is obviously impractical (once again due to bandwidth problems). If the simple solution is taken, breaking up the activity store into separately addressable cells, then the bandwidth problem is eliminated but a tremendous connection/packaging problem arises. The solution is, of course, a compromise between a single activity store containing N templates and N activity stores each containing a single template. Consider the diagram given in figure 5.

In this scheme, the cell blocks form the activity store and each cell block replaces not only a number of instruction cells but also the corresponding portion of the distribution network. Results are delivered to the cell

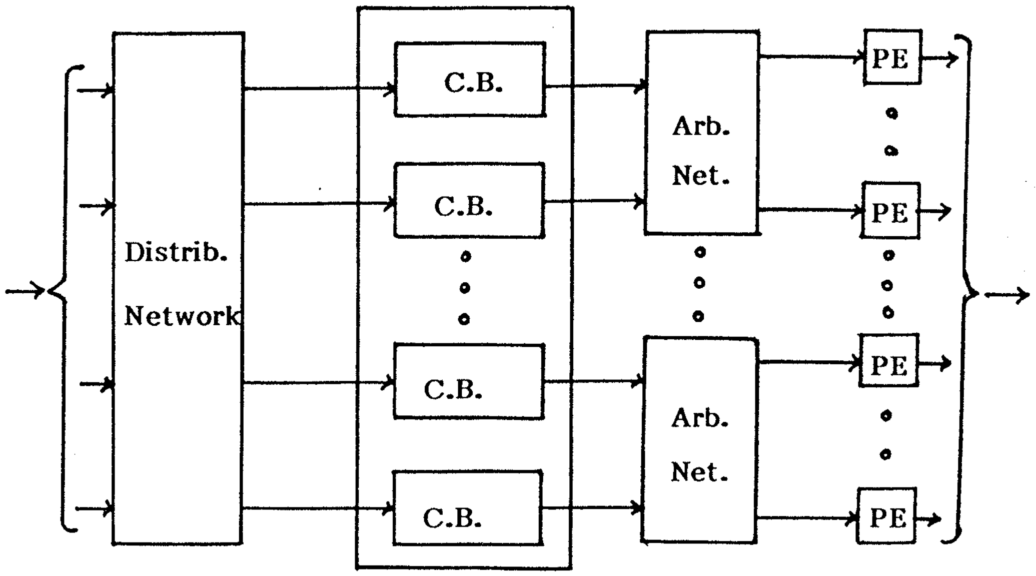


Figure 5: The Cell Block Architecture

blocks by the distribution network. The operation packets produced in response to these results are sent to an arbitration network which routes them to an appropriate functional unit for execution. Note that each arbitration network has a group of shared execution units so as to reduce overloading and bottlenecks at the processors. Each cell block is roughly equivalent to the traditional dataflow processor less the functional units which have been factored out. Thus, the original system has been subdivided into a number of smaller subsystems with the hope that communications between the subsystems will be limited.

Two distinct types of dataflow machines may be distinguished. These are the "static" and "dynamic" approaches to dataflow. (Both the dataflow multiprocessor and cell block architectures are static.) Static machines are those such as have already been described. Such machines are characterized by unlabelled data tokens. This means that only a single token may be present on each arc at any given time. In a dynamic dataflow machine, the data tokens are labelled. Thus, many tokens may exist on a given arc and the label uniquely identifies the context of each token. This provides the ability to support loops in the flow graph and thereby ensures maximum parallelism.

In software, the approach has been to allow specification of algorithms in a serial fashion and then to use system software which is capable of detecting and exploiting any possible parallelism. As it turns out, many of the techniques used by optimizing compilers may be applied to the problem of detecting parallelism. Languages such as

VAL[2] and ID[5] are examples of this approach. They are what are known as applicative or functional languages and are appropriately structured for easy translation to flow graphs. Such languages have one advantage over conventional programming languages: they are free from side-effects.

One of the major problems with translating from serial to parallel algorithms is detecting side-effects such as that shown in figure 6.

The primary source of such side-effects is the design of the language itself. Variables (or objects) are themselves addresses and as such may be inadvertently referred to. In functional computing, with languages such as VAL, there are

```
PROCEDURE switch(VAR a,b:INTEGER);

BEGIN
a:=xor(a,b);
b:=xor(b,a);
a:=xor(a,b)
END;
```

Note: if the same variable is passed to both 'a' and 'b' then it is set to zero rather than remaining unchanged. (Due to the call by address).

Figure 6: A Side-Effect

no variables per se. Instead, operations produce values which are directly consumed by other operations. Thus, there is no place where one operation can mistakenly refer to or unintentionally modify data. (VAL is then value oriented rather than address oriented.) By enforcing the use of a functional language, the task of identifying possible concurrency is greatly simplified. This restriction may be implemented by simply adhering to the single assignment rule. This rule states that only one assignment may be made to a variable within its scope. This precludes the reassignment of variables, but should not be considered restrictive as extensive reassignment is bad programming practice.

In VAL, detection of parallelism occurs in two situations. Compound expressions are broken down into simple expressions which may then be executed in parallel subject to procedural constraints. Also, vector and array operations are designed to allow concurrency identification. Due to this limited identification of parallelism, VAL (unlike CLU[17] on which it is based) is restricted to the domain of numerical mathematics.

What has yet to be attempted is the development of a programming language/system in which the programmer is responsible for some of the control of parallelism. This concept is addressed later.

1.2 RULE-BASED SYSTEMS

This section, like the first, is used to provide necessary background information for the thesis.

1.2.1 Concepts

Rule-based programming systems represent a relatively new approach[1] to coding algorithms. They provide only a single format for statements which is shown in Figure 7. However, this simple form allows the programmer to explicitly direct

```
( conditions ) -> [ actions ]
```

Figure 7: Production Format

the machine in either a serial or parallel fashion.

The "conditions" section of each rule-based statement is composed of an arbitrarily complex logical expression which specifies the condition(s) under which the associated "actions" section is to be executed. Each of these statements is said to be a rule or production, and a rule-based program consists of a series of logically related rules.

[1] The approach is new although the idea itself is quite old.

The logical structure of such a system is given in Figure 8. The inference engine serves as the control mechanism for the system. It selects enabled rules from the rule base and

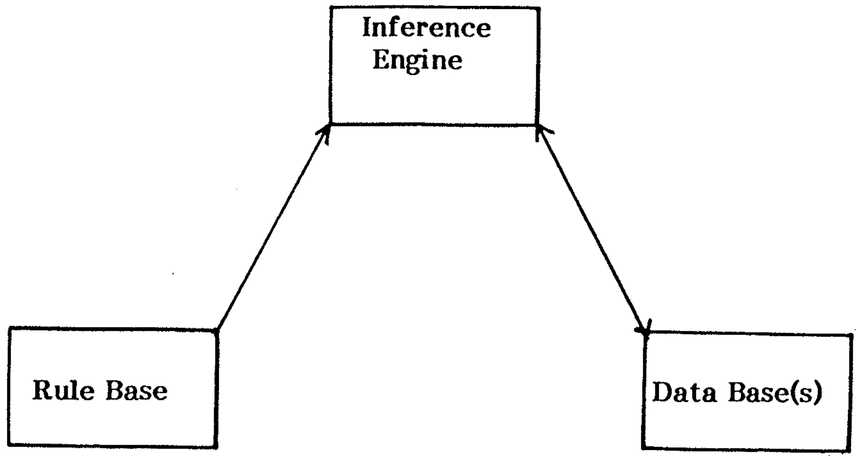


Figure 8: Rule-Based System Structure

executes them, thereby modifying values in the database(s). The modification of data within the database(s) may then set a condition in another or the same rule to 'TRUE' and thereby enable its actions. Consider the program given in Figure 9. This classic algorithm provides a good example of serial rule-based programming. A line by line description of the program's activity is given in Table 1.

Notice that, like dataflow, there is no need for explicit control structures. The flow of control is determined implicitly by the order in which conditions become true. It should again be noted that the order of execution of enabled statements is not guaranteed; only the conditions under which a statement will be enabled is guaranteed.

```
        (* PRIME - Boolean Vector               *)
        (* NPRIME - Next Prime                  *)
        (* MULTIPLE - Composite to be removed *)
        (* N - Size of Vector                   *)


1)   **FIRST** -> [ prime(*) <- TRUE;
                    nprime <- 1;
                    multiple <- n+1; ]

2)   (nprime > SQRT(n)) -> [ PRINT,prime(*);
                             STOP; ]

3)   (multiple>n & prime(nprime+1)=TRUE &
      nprime<=SQRT(n))
                           -> [ nprime <- nprime + 1;
                                multiple <- nprime * 2; ]

4)   (multiple>n & prime(nprime+1)=FALSE &
      nprime <= SQRT(n))
                          -> [ nprime <- nprime +1; ]

5)   (multiple<=n & nprime <= SQRT(n))
                          -> [ prime(multiple) <- FALSE;
                            multiple <- multiple +nprime; ]
```

Figure 9: Rule-Based Program for the Sieve of Eratosthenes

The strength of a rule-based language for dataflow programming is dependent on the base instruction set of allowed actions. If a sufficiently extensive set is used, much of the difficulty inherent in parallel programming may be shifted onto the machine.

Consider matrix multiplication. At the lowest level this is a fairly complex parallel process. If however, vector

line #	Description
1	initialization (executed once)
2	the terminating condition
3	next prime found
4	looking for next prime
5	knocking out composites

TABLE 1

Description of Figure 9 .

operations are allowed as actions, then the problem becomes trivial. Thus, by carefully choosing the dividing line between hardware-implicit and user-controlled parallelism, the coding of such problems may be made considerably easier.

1.2.2 Current Work

Almost all work in the area of rule-based or production systems has been done in implementing so called "expert systems". This area of Artificial Intelligence (AI) deals with knowledge engineering. Its goal is to provide a computer system which contains the accumulated knowledge of many experts and which may then be used as an assistant in its particular area of expertise.

Systems have been produced which are being used in such areas as medicine (MYCIN[9]), geological prospecting

(PROSPECTOR[8]) and physics (DENDRAL[7]). The rule base for an expert system consists of a series of assertion-assumption pairs. That is, a collection of assertions which, when shown to be true, permit certain assumptions to be made.

Consider a system trying to ascertain whether or not a given patient has a particular disease. Through prompting for input (i.e. medical test results) the system may make initial decisions. These lead to other decisions being made which eventually lead to a resolution of the problem at hand.

This is very much a non-deterministic process in which a given rule may lead to many alternatives. Each of these alternatives must in turn be examined in order to select the "best" one. The question at hand is: "Which is the best one?" The answer may be found using a certainty factor which is associated with each rule or production. The certainty factor indicates how strongly the expert whose data is stored in the system feels this rule applies.

For example, a given set of symptoms may suggest two or more possible ailments. Presumably the likelihoods of the patient having each ailment are different, given this set of symptoms. Thus, the most likely one could be assigned a high certainty factor while a less likely one could be assigned a lower certainty factor.

Many of the existing systems simply follow the one or two most likely paths and hope that they prove correct. This is done mostly for reasons of efficiency. A search of all possible paths may simply be too expensive. Unfortunately, this approach also tends to lessen the effectiveness of expert systems. If all possible paths can be checked then deductions which initially look unlikely but which are subsequently supported by new evidence are not overlooked.

Consequently, the ability to check all paths will decrease the importance placed on the order in which the questions are asked. In most systems it is important to ask questions in such an order as to quickly and precisely subdivide the problem into a number of primarily independent sub-cases. When an "all-paths" search is done, the problem of overlooking certain paths because they do not appear promising is avoided. If sufficient computing power is available to support such a search, the construction of the expert system may be simplified.

The primary area of application of expert systems should be in remote areas where a human expert is unavailable. This means that a general practitioner could then act as the attendant and successfully perform as if he were an expert in some specific medical field. Ideally, an expert system should also be marketed with a good selection of "knowledge" and "rule" bases.

Unfortunately, such an expert system with applicability in a variety of question domains is not available. This is due to a number of factors including:

- Lack of compact computing power
- Lack of compact storage media for maintaining the rule and data bases
- Poor design practices (often related to the first two points)

With the proper hardware (including a special purpose architecture) and better software, a viable general purpose expert system may be achievable.

Chapter II

PARALLEL PROGRAMMING AND DATAFLOW

A fundamental problem still to be solved before parallel computation is widely accepted is the question of programming in a non-Von Neumann environment. Programmers are taught from the very beginning to think in a serial fashion and this leads to inefficient algorithms when coding for a parallel machine. Some rethinking of the fundamentals of programming must be done in order to yield a useful parallel language.

Common serial languages fail to recognize and exploit the available parallelism in an algorithm. Furthermore, they are incapable of expressing truly parallel algorithms (that is, algorithms where the programmer specifies which operations are to take place concurrently). At best, some languages provide facilities to allow concurrency at the procedure level. Unfortunately, this is far too coarse a level to be of use in any massively parallel system such as a dataflow machine.

Two general approaches may be taken when programming parallel machines:

- user-controlled parallelism
- parallelism detected by system software

Let us first divide the available parallel machines into two classes according to Flynn[15]. Those machines which are typified by single instructions being performed on multiple pieces of data at once are called SIMD (Single Instruction, Multiple Data) machines. Falling into this category are the vector and array processors (i.e. those processors which operate on multiple data items synchronously or in lockstep). The second class of machines known as MIMD (Multiple Instruction, Multiple Data) are typified by many instructions working on many different pieces of data concurrently. Dataflow machines fit into this category.

Consider first the SIMD machines. Regardless of whether or not the user does the identification of parallelism, somewhere there must be intimate knowledge (either on the part of the applications programmer or the systems programmer) of the machine's internal architecture in order to program effectively. For example, a vector processor consists of a finite number of processing elements and therefore the number of elements that exist must be known in order to partition larger problems. Thus, any software written for such a machine is not general.

The task of identifying possible parallelism via system software is not difficult for this type of machine. Vector and array processors are designed for a specific class of problems. Thus, if a high level language which allows array operations is used as a programming environment, identification of simple parallelism is not only trivial but also immediate. Consider the following example:

```
DCL
    A,B(1000,1000);

A=A+B;  /* obviously a parallel operation */
```

Even if such a language is not used, it is relatively easy to identify matrix operations and optimize at least the inner loop of such an operation. Techniques similar to those applied in optimizing compilers may be applied to perform the required analysis.

Programming for SIMD machines has become fairly well established. Several high level languages have been developed for specific machines and are working out well. Additionally, a variety of lower level parallel algorithms have been developed. Matrix multiplications in which the innermost loop has been replaced by a parallel operation are now commonplace on machines such as the CRAY-1 and CYBER-205 (not to mention true array processors). Even many inherently serial problems have been reworked for parallel solution. A good example of this is a vector summation. In log2n steps, all the sums of from 1 to 'n' of the elements of a vector may be calculated.

Unfortunately, there are problems. Perhaps the most dramatic of these is the need to partition large problems for solution. Should the number of P.E.'s in a vector processor be smaller than the number of vector elements, the vector must be subdivided and this subdivision causes many algorithms to degenerate. (The vector summation algorithm for example.) Furthermore, it is undesirable that programs be architecturally dependent. Portability is a very appealing concept especially for such special purpose programs. For these reasons, a SIMD type architecture is not an ideal base for a parallel system.

There is a better chance of designing a general parallel language for the MIMD class of machines. It is irrelevant whether the architecture at hand is a dataflow machine or a network architecture. Both are essentially the same, the only differences being the structure of the network, the distance between nodes in the network, and the type of programming currently being done on each type of machine. If these differences are disregarded and both machines are simply treated as a collection of processors and memories on some sort of network, they may be considered equal. This means that results attained using the dataflow model may be extended to this more general class of machine.

In a MIMD system such as that just described, the goal is not to produce calculations of a special form but rather to produce as many single heterogenous calculations as possible at any given time. For this reason, such a system is architecturally independent and therefore much better suited to producing a general parallel programming system.

Consider programming in such an environment. If the programmer is left to determine parallelism, he will invariably code in a serial fashion and then analyze the program to detect parallelism. A part of the analysis which must be done is the reduction of complex expressions to simple ones of the form:

```
result <- opnd1 oper opnd2
```

These simple expressions are then examined to determine inter-statement parallelism. In addition to facilitating the detection of parallelism, this permits the repeated evaluation of common subexpressions within loops to be easily factored out to a single operation. However, this process is exceedingly tedious and as such should be performed by the machine. Thus, a general parallel language should allow complex expressions which may be broken down and analyzed at translation time[2].

Such a language should also be side-effect free so as to simplify this analysis. To be side-effect free, it would seem that assignment should be eliminated altogether (as in LISP). This, however, is not necessary if the single assignment rule is followed as is done in ID[5] (the Irvine Dataflow language). This rule permits the use of assignment in a restricted capacity. Specifically, a single assignment is allowed to any given variable within its current scope. This guarantees that since only a single assignment is made,

[2] These very operations are in fact commonplace in "peephole" optimizers.

inadvertent modification of a value is impossible. The assignment is considered to be a "naming" of some result. It is a non-executable operation and should not be associated with any sort of write-to-memory function. This naming is easily performed at compile time.

The presence of assignment affords the programmer a more familiar environment and may therefore simplify the translation process. When assignment is permitted the user may, for the sake of simplicity, factor out some sub-expressions on his own. This relieves the translator of having to perform the removal. Consider the following code segment:

```
part1 <- SQRT(b**2-4*a*c)
part2 <- 2*a

root1 <- (-b+part1)/part2
root2 <- (-b-part1)/part2
```

Furthermore, the use of temporary values will not result in additional templates (since assignment is not an executable operation).

The high level constructs of programming languages should clearly also be allowed. Functions and procedures are merely logical abstractions of complex operations. They present no difficulty as far as architectural independence is concerned. If necessary, each subprogram may be treated as "inline" code (this is not necessary as will be shown shortly). In fact, in systems where independent activity stores are provided, the presence of subprograms provides a logical division point for assignment to different activity stores. The use of these division points may improve performance.

Presumably all operations in a subprogram are related and, as such, should be grouped together. Values used by the subprogram will also be local to the processor since global variables cannot be allowed without possible side-effects. The only exceptions to this rule are the parameters to a subprogram. These, however, may be copied to the local activity store and then back again in order to minimize network traffic. A "smart" loader will attempt to place functions from a program into an activity store with other related functions from the same program. This is also done to minimize the more expensive non-local references.

The one question that does arise is how to translate middle-level serial constructs to their parallel equivalents. This is, of course, assuming that the existence of any possible parallelism may be detected.

Consider a high level language which does not support array operations. In this case, a loop or set of nested loops must be used to accomplish the desired array operation. In most cases, this sort of occurence is fairly easily detected and the translation required is obvious. In other cases however, the translation to a parallel form is not so obvious. These cases, when exploited, may yield significant performance increases. It is for this reason that some of the parallelism will eventually have to be specified by the user.

The conversion to a parallel language as just described for a dataflow system is fairly straightforward. The most difficult area is in the handling of complex loops and other middle level constructs. (High level parallelism may be left to the programmer and low level parallelism is easily translated.) What must be done with an array operation such as the one shown below is not clear.

```
FOR i:=1 TO 2000 DO
   FOR j:=1 TO 5000 DO
      a[i,j]:=b[i,j]**2;
```

This statement yields ten million independent exponentiation operations. Each of these operations could conceptually be executed in parallel but this is obviously going to be impossible due to physical limitations.

Aside from the exceptionally large number of processing elements required to support such parallelism, ten million templates for each of 'a' and 'b' is obviously unacceptable due to memory contraints. Either a very large memory or a demand paged memory system would be required. Instead, 'n' templates of a general nature acceptable for executing this instruction may be used to calculate the squares of all elements of 'b'. Thus, the problem is effectively subdivided or "partitioned" into smaller problems. The number of templates actually used will correspond directly to the number of available processing elements. Each set then creates and enables the next. Furthermore, this approach permits the situation where the range of elements to be squared in 'b' is not known until run time. Simple static (compile time) generation of the templates does not support this feature.

A related problem is the question of program segments such as those in a subprogram which are frequently entered. If these subprograms are invoked many times concurrently, then multiple copies of some templates may result. Such a piece of code will contain operations whose templates will be repeatedly scheduled into an instruction queue. This means that there may be many active versions of a template. These templates must be physically distinct from one another. That is, an identifiable copy of a template is needed for each concurrent use of that template.

A typical example of such a function is a system service routine which is being shared by many users. Since data is effectively stored with the program (in the templates), there is no possibility of reentrancy. In a static computer system this would force one copy of the program for each user. In a dataflow system, though, the duplication of the entire function may be avoided. Only those templates which must be duplicated will be. This is accomplished using dynamic creation of template versions. Since they are created, and presumably destroyed, dynamically, it is possible to detect when a particular template has multiple versions concurrently active (perhaps via a usage count). This also provides a natural basis for the solution of the many-identical-operations problem (as typified by the ten million element example).

The idea of template versions is not unique. The unravelling interpreter[5] used in the Irvine dataflow machine does exactly this. The difference is that the analysis is performed almost exclusively at compile time and thus cannot handle the dynamic cases effectively. By allowing dynamic template allocation and freeing, additional complexity is introduced into the hardware but significant performance gains may be recognized. Additionally, this scheme permits large problems to be solved within the constraint of a relatively small activity store.

What is clear from this discussion is that although much may be done in translating serial to parallel algorithms, existing methods are imperfect and must remain so for reasons of complexity. A better approach is to provide a language in which the programmer specifies the "trickier" parallelism but leaves the rest to be detected by the system.

Chapter III

A RULE-BASED DATAFLOW PROCESSOR

3.1 SYSTEM SIMILARITIES

In order for any two ideas to be successfully merged, they must have some things in common. It is the similarities between systems that are exploited to permit their combination. This is particularily true of hardware-software combinations. By providing good architectural support for a programming system, much effort may be saved. This is, of course, in addition to the performance increases which may be realized.

If the criteria for merging a rule-based system with a dataflow machine is strictly the system similarities then it should definitely be attempted. In gross structure the two are quite similar. Both have a control unit which fetches "appropriate" operations, schedules them for execution and then returns results which enable other operations. The only differences between the two are purely cosmetic. (For example, a rule-based system tends to implement two physically distinct memories while a dataflow system typically implements only one.)

In terms of functional principles, the two are identical. Both systems have some actions executed conditionally. In rule-based systems there is an arbitrarily complex boolean expression determining whether or not the actions should be taken, while in the dataflow system there are actions which are taken when their operands are defined. These systems are simple isomorphisms of one another as can be shown by the following argument:

Assuming that all values initially have the undefined value (undef_val), a flow graph for:

A + B

may be easily expressed as the rule-based statement:

(A<>undef_val & B<>undef_val) -> [A+B]

Perhaps worthy of note is the fact that the enabling conditions for executing a dataflow operation as a rule always involve the operands of that operation. In this respect at least, it might be argued that the isomorphism is in fact imperfect.

Equivalence in the other direction can easily be seen if the structure of templates and the way in which they are enabled is modified slightly. Normally, a template (and the instruction it contains), is enabled when its operands are defined. What is additionally required is an enable signal for the template as a whole. That is, a template will not be eligible for execution until this new enable signal is asserted and its operands are correctly defined. In most cases, this enable signal will be initially set (enabled). However, in some cases it will not be set until the successful completion of some other dataflow instruction. Thus, a new instruction type may effectively pass an enable token to those templates which are logically dependent on it. This enable token could be easily replaced by the passing of a boolean value which was subsequently discarded. By doing this, a new template type could have been avoided since the conditional enabling would have fit into the existing structure. However, since the enabling values would simply be immediately discarded, they may as well be considered to be enables so that the user need not worry about them.

The format for such an instruction's template is given in Figure 10. Notice that by providing enable outputs for both success and failure, the basis for an IF-THEN-ELSE structure is immediately provided. From this it is clear that such a system fits more naturally the architecture outlined for a static dataflow machine. This is not to say that a dynamic architecture is inappropriate but simply that a rule-based system is conceptually closer to the static system.

The conditional expression from a rule-based instruction may be taken and translated into a series of these conditional templates. Each sub-condition may enable the next sub-condition and the final one may enable the actions of the rule. This is obviously an inefficient serial

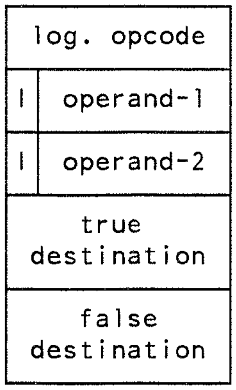


Figure 10: A Conditional Template

approach and a better mechanism should be developed. Regardless of the efficiency of the implementation though, it is clear that the dataflow program and the rule-based program are logically equivalent.

3.2 DESIGN CONSIDERATIONS

Having seen that rule-based and dataflow systems are similar, it must now be decided whether or not the merger of the two is viable. This is determined by considering the efficiency of the implementation as well as its feasability.

In terms of efficiency the combination is excellent. The primary problem with rule-based systems has been their inefficient execution on serial processors. They produce many operations which may be done in parallel and as such they are better suited to a parallel architecture.

Furthermore, rule-based systems produce many heterogeneous operations which make them suited to the MIMD rather than SIMD class of parallel machines.

Of the available MIMD type processors, static dataflow processors most closely parallel the rule-based approach. All of the operations in both the conditions and actions sections may be performed in parallel on such a machine just as they are intended to be. Also, the dataflow machines' "update" unit is perfectly suited to the rule-based environment.

Most of the questions which arise are not due to the merger of the two systems but are due to the practicality of dataflow machines themselves. While it is clearly an efficient system conceptually, there are some implementation difficulties to be overcome. Depending on implementation strategies, there may be cost-feasability problems or performance inefficiencies in a dataflow system. A typical example of this is using destination addresses in templates as opposed to the use of CAM (Content Addressable Memory) tags on instances of variables. In the first implementation, there is a problem in distributing the result to multiple destinations. To overcome this, multiple levels of destination-extension templates (as shown in Figure 11) may be used. This approach introduces an inherent inefficiency into the system in that overhead is incurred in fetching and decoding these templates and then

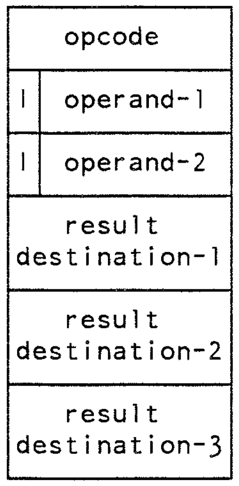


Figure 11: A Destination-Extension Template

re-routing the results as specified. The inefficiency results from the fundamentally serial nature of the process. Logic, if not economics, dictates that better performance may be achieved by the use of CAM tags on specific instances of variables. Nevertheless, most implementations have favoured templates due to the natural mapping from flow graphs. In contrast, the use of CAM tags is exceedingly efficient and permits parallel broadcast of results to all appropriate templates. In this system, each operand field in a template has associated with it a CAM tag field. When the broadcast tag matches the tag field, the corresponding operand field accepts the value being presented to it. While this scheme is efficient it is also exceedingly expensive and a single memory implies serialization of broadcasts. The cost of CAM is high and the quantity required is large. This makes the use of an activity store where all templates have CAM tags impractical. What is required is, as usual, a compromise. Some special templates may have the desired tags and be used for operations where this facility is required. This assumes the dynamic template allocation/freeing system discussed earlier.

Another question of practicality arises from whether or not a system can afford to support the number of processing elements required to make a dataflow machine practical. Something on the order of several hundred or more likely thousand P.E.'s are required for a typical multi-user system. If a general purpose P.E. is used, then the cost of each unit will be quite high. The use of special purpose units will decrease this cost but this introduces the need for a more complicated arbitration network to manage the scheduling of operations to appropriate processing elements. Obviously, custom SSI or MSI units will quickly price such a machine out of normal markets and will most certainly leave it non-portable. The use of VLSI designs avoids these problems. Not only are they cheaper, but circuit densities are far superior. They do generally lack in performance, since MOS technologies are slower than ECL or TTL (in terms of switching speed), but this is not as severe a problem as it may appear. Since the key to a dataflow machine's performance is not faster components but rather greater parallelism, slower units may be tolerated. (The net performance increase through parallelism greatly overshadows the performance increase attainable through the use of fast components.)

The use of MOS VLSI designs is also suitable for the production of activity store components. The fabrication of static RAMs is a well documented process and the addition of simple routing hardware onto such chips seems a simple task. Thus, building blocks for activity stores such as Dennis' cell blocks might be easily realized using VLSI.

Further gains may be achieved in this area through the use of microprogrammed control units. Much of the control function has already been relocated from the P.E.'s to the other components in the dataflow machine and that part which remains does not warrant a significant investment in hard-wired logic. Instead, a small, fast microstore should be used and the cost of the microprogramming may be amortized over the number of units produced. This is an ideal approach since many such units are produced even for a single dataflow machine.

The use of microprogramming frees up chip area in VLSI designs which may then be used for other purposes. The sort of P.E. used in a dataflow machine is effectively a RISC[19] (Reduced Instruction Set Computer) processor. If the RISC philosophy is followed for these P.E.'s, then the saved chip area can be used to incorporate performance enhancement features (extra registers, pipelines, etc.). For dataflow P.E.'s, extra registers may not be suitable but a simple pipeline should be implemented.

A pipelined processing element is a particularly attractive feature for a dataflow machine. Just as for vector and array processors such as the CYBER-205, very many consecutive identical operations may be generated in a dataflow system (as in the array example). Furthermore, in a dataflow machine this is true for all operation types, and is especially true if a set of vector operations is provided as a programming base. Thus, the pipelining of functional units may yield a significant performance increase in a machine such as that being proposed.

The other advantage provided by the use of microprogramming is the easy adaptation of a general purpose unit to any required P.E. type. P.E.s performing different functions may be implemented from the same basic P.E. simply by rewriting the microcode. Admittedly, this is not the most efficient approach but the reduction in both cost and space requirements outweigh any lost performance. Furthermore, this approach may be taken on a larger scale if writable control stores are used. When this is done, a specific program may have special-purpose microcode loaded for it so as to enhance performance. Thus, depending on the language being used to program an application, different microcode may be loaded to ensure optimal execution time. Such an approach was taken on both the Burroughs B1700 and the Xerox Alto computer. Thus, its feasability is assured.

Allowing different types of P.E.'s, whether through microprogramming or not, is beneficial if not overdone. That is, a limited number of P.E. types should be supplied so as to minimize the complexity of the arbitration network. A good balance between the number of P.E. types and the number of functions per P.E. must be made. If too few P.E. types are chosen, in an effort to minimize arbitration complexity, the amount of control required in each P.E. is increased. Since P.E.'s are more numerous than the arbitration networks, this will result in higher costs. However, if too many P.E. types are chosen, arbitration becomes complex and design costs increase.

The two fundamental control functions of a dataflow machine also present some implementation problems. Both the fetching of enabled instructions and the update of templates based on results are complicated, highly parallel functions. The update function may be performed indirectly by the distribution network if we assume a destination address system. If a CAM system is used, then the update function is implemented in the control store itself. The only update hardware required is then a broadcast facility.

The problem that remains is how to detect enabled templates and how to efficiently schedule them for execution on the available functional units. Whenever a result is forwarded to some template, the system must check to see if this enables the given template. The checking process will incur some overhead, particularily if the result is being forwarded to many templates. Furthermore, memory contention problems may arise. A series of results which are forwarded to many templates will overrun the capabilities of a single unit attempting to check for enabled templates. This means that multiple, concurrent units will be required which introduces the problem of multiple, concurrent accesses to the activity store (a good reason for advocating many local activity stores). Thus, the detection of enabled templates must be accounted for in any design attempted.

One solution to the above problem is the use of separate memories (like cell blocks) and possibly even division of the memories into individually addressable banks or blocks. This would most certainly alleviate the problem of memory contention.

Another approach is to use special hardware built into the memory to test for all operands being defined whenever an operand field is accessed. When all operands are defined, the address of the template is returned to the unit which schedules templates for execution. This distributes the detection of enabled instructions throughout the memory thereby alleviating the contention/bandwidth problem. The disadvantage to this approach is that the activity store may no longer be implemented using commercially available memories. Instead, specially designed memories will have to be used and this again results in increased costs even when the implementation technology is MOS VLSI.

The scheduling of instructions to available units requires some sort of hardware queue. In particular, the system must be able to tolerate sudden streams of many identical operations. This "clustering" phenomenon means that, from time to time, more enabled operations will exist than appropriate P.E.'s to service them. For this reason, there must be some facility provided to handle the overflow.

A 'queue' implies FIFO service. However, this is not an ideal structure for maintaining enabled instructions. The goal in dataflow is to maximize the amount of concurrent activity and a simple FIFO queue may cause the following undesirable result. If there is no unit available to service the operation at the front of the queue, issuing of operations suspends even if there are units available for later operations in the queue. A more flexible structure is therefore required.

A simple queue is acceptable only if a change is made to the way it is used. The scheduler must be able to remove activities, determine that they cannot currently be executed (due to resource availability) and return them to the far end of the queue. This should not affect the efficiency or correctness of execution due to the very nature of a dataflow machine ("PC-free"). Re-ordering of instructions is done freely.

Another alternative is to maintain separate queues for each type of processing element. This clearly solves the problem but forces the update unit to perform at least a partial decode on the operations to determine the queue into which they should be placed. This is not difficult to do and is, in fact, probably simpler to implement than the previous scheme. It does, however, introduce an undesirable dependency between the update unit and the fetch unit and the P.E.'s. If a new type of P.E. is added, the only component that should have to be modified is the routing portion of the fetch unit. If multiple queues are used, the update unit will also have to be modified. Thus, the ease of extendability is decreased when using multiple queues.

There are many factors to be considered before deciding on a final design. The features chosen and thereby the inefficiencies accepted will depend highly on several factors:

- Programming Environment
- Cost constraints
- Availability of VLSI fabrication facilities
- End use of the system

Finally, the design of a prototype rather than a production system affects many design choices. In a protoype system, such as that proposed in the following chapter, decisions will be made so as to maximize modifiability and extendability. These two features are absolute necessities in a prototype since it is desirable to gain as much information per design dollar as possible. Without existing hardware, the correctness of a prototype design is not assured. This means that every possible step should be taken to detect errors and to correct them. Also, the cost of the prototype should be low enough to allow for one or more complete design failures.

Chapter IV
A PROTOTYPE DESIGN

4.1 SYSTEM DESCRIPTION

This section outlines the fundamental design decisions made with respect to the prototype rule-based dataflow processor. The description is primarily that of a dataflow processor but the structure of the processor is directly affected by the fact that it is to execute rule-based programs.

Before being able to design any portion of the processor, it is necessary to consider the types of operations to be performed upon it. In the case of a rule-based dataflow processor, this involves examining the base set of allowed actions within each rule. Since it is handled inherently within the structure of the processor, the enabling of each rule based on the conditions section need not be considered. The update unit is, in a sense, an integral part of the activity store. The actions section is executed by one or more P.E.'s as determined by some sort of arbitration network. Operations specified in the actions section affect not only the structure of the P.E.'s but also the organization of storage within the system.

The general architecture of a rule-based dataflow processor should be static as was decided in the preceeding chapter. Thus, the prototype design will be static in nature. The term "static" applies only in the traditional sense of dataflow. Operations are passed on various system buses in a complete form along with control tokens and therefore no token matching section is required. Many functions of the prototype system are definitely dynamic. In particular, the expanding (generic) templates may be considered to be a dynamic function since templates are being created dynamically.

The base language chosen for implementation in the prototype is a restricted version of APL. The APL language offers a number of interesting operations that provide ample opportunity for parallelism and the appearance of APL should tolerate the necessary syntactic extensions to incorporate rules. If a programmer can face normal APL, the syntax for specifying rules should present no problems. Also, the use of APL provides at least a somewhat familiar environment for the programmer. This is preferable to using flow graphs or other unconventional approaches which are foreign to most programmers. Finally, APL is certainly suitable for implementing expert systems in that it supports all the required operations and more.

In the prototype, only integer values are recognized and therefore only the integer operations are supported. (In APL this means nearly all operators are supported.) Two basic types will be defined:

- scalar integers
- non-scalar integers

Beyond this, no data typing is performed. This suits APL and is perfectly legitimate as was shown in ID[5] (which is totally typeless). For the prototype, 16-bit integer values are used to provide not a useful processor, but one which is cheap and still representative of the machine which is eventually desired/required.

Function definition is supported in much the same way as in normal APL with two exceptions. The exceptions are made to ensure no side-effects and are the following:

- No global variables (locality of effect)
- A single assignment rule is in effect.

Clearly the presence of global variables (and VAR parameters as well) will jeopardize the correct functioning of a program due to possible side-effects. Unfortunately, in many languages, these features also make functions somewhat restrictive. This restriction is due to a lack of the ability to return complex results. Fortunately, such is not the case in APL. Since there is no restriction on the "type" of the value being returned, the result may be as complex as the user desires. The sécond exception to normal APL syntax is also required to guarantee that the system will be side-effect free as discussed in earlier chapters.

Normally, a single assignment rule makes programming in a conventional style difficult. A simple example of where this is true is when using a counter variable. The statement:

```
INDEX<-INDEX+1
```

is invalid within a single assignment rule. Due to the obvious inconvenience caused, this restriction has accounted for statements such as "FORALL" in VAL.

It is clear that some form of iteration is required in any parallel system. (Both for specific applications and to provide familiarity within existing systems.) In APL, such standard iteration methods are particularily necessary since the syntax will not happily tolerate the introduction of high level looping constructs.

In order to permit the implementation of loops in APL within a single assignment rule, a new operator is introduced. This operator, the "rename" operator, allows <u>explicit</u> re-use of names. Thus, the statement:

```
X<+X+1
```

is valid, whereas the statement:

```
X<-X+1
```

is not. The renaming does not result in a second assignment to the variable X but rather results in the creation of a new variable with the old name. In other words, a new instantiation of X is done.

By using the renaming operator, a programmer is acknowledging the reuse of the variable name. In physical terms, this will result in distinct templates before and after the renaming and hence precludes the possibility of loops in the resultant flow graph. Finally, this provides a convenient and familiar environment for the programmer.

Note that multiple assignments to the quad (I/O) operator are always acceptable. This is necessary to facilitate serial output operations, and the renaming operator should not be used with quad.

An addition to APL is necessary in order that it may be used in a rule-based dataflow processor. Normally, a variable's type is an unknown quantity until run time. This is not possible in the prototype system. Vector and scalar instructions have different formats. (Either an explicit or generic template will be generated.) This means that it must be known at compile time whether a given variable is a vector or a scalar. In order to do this, a simple syntax change is made. A definition of all variables must be made and a syntax must be provided to specify whether each one is scalar or N-dimensional in nature.

A complete description of the language used in the prototype is given in Appendix A. This includes the rule-based syntax for APL and a list of those APL operations supported. The instructions/template operators used to implement these operations are shown in Appendix B. Programming examples are given in Appendix D. (Note: the programming examples apply to a rule-based APL such as that in the prototype which supports INTEGERs). The choice of INTEGER-only support for the prototype was a cost decision not a functional one. A block diagram of the overall system structure is given at the start of Appendix C. This diagram should be referred to throughout this section whenever clarification is needed.

Storage in the prototype system is divided into two fundamental parts. The first, the activity store, contains operation templates as might be expected in a dataflow machine. The second is used for data structures which are not easily placed within an instruction template (eg. vectors, arrays, etc,). This second memory is known as the vector store. The idea of having more than one memory in a dataflow machine has been explored before [5,22]. Implementation of the vector store is a similar idea but is used in a unique way.

The inclusion of an arbitrary length vector within a template is an impractical approach. Similarily, compile-time generation of individual templates for each of the vector components is inappropriate or even impossible in many instances. The prototype system includes templates which will spawn other templates to perform such complex operations. These spawned templates will not contain the data they operate upon but rather pointers to the data within the vector store. Thus, vector operations in APL will generate these generic templates which dynamically produce templates to perform the specific element operations. This seems to produce a problem with respect to detecting side-effects. However, if handled correctly, it does not. If an operation of rank greater than one is such that elements are not independent of one another, then it becomes difficult to detect possible side-effects. Spawned templates must not enable other templates aside from the ones which spawned them. A generic template spawns all its children and then awaits their completion (via enabling rules). Each child, on completion, partially re-enables its parent. When all children have executed, the parent (generic) template is re-enabled and it may then enable the next <u>logical</u> operation(s) in the dataflow program.

This differs from the approach taken at M.I.T. which uses "streams". When using a stream as a data structure, a suffix function G may begin operating on elements of a stream X before a prefix function F has finished with them in a manner analogous to a pipeline (see below).

$$\begin{tikzcd} \boxed{F} \arrow[r, "X"'] & \boxed{G} \end{tikzcd}$$

This is possible only since a stream is a uni-directional data structure. The stream concept permits a high degree of parallelism but only at the expense of generality. Element by element enabling of templates for multi-dimensional operations is often not applicable. For instance, in a matrix multiplication, elements are non-independent. Thus, each vector operation is executed indivisibly as if it were one operation and side-effects are eliminated. Naturally, if the programmer specifies incorrect rules in his/her program, side-effects may still occur. If vectors, etc. are treated as single logical items, however, this problem will not arise.

The activity store is subdivided as in the cell block architecture. Each block contains not only the instruction cells and routing hardware but also circuitry to test for enabling of instruction cells. This hardware will decrement an operand count on every reference to the template which updates the corresponding operand. Thus, each template will be created with this field initialized to the number of operands it is awaiting, and every time an operand is routed to it, the count will be decreased by one. When the count reaches zero, the instruction is enabled and the template's identifier will be forwarded to the appropriate unit for execution scheduling. Once an instruction has been enabled in this manner, it will be queued for execution at an appropriate P.E.. The routing of the enabled template to a P.E. is performed by an arbitration network which examines the first few bits of the operation code within the template. These bits determine whether the operation is complex (i.e. this is a generic template which must spawn element templates) and therefore destined for a "decomposition" P.E., or if not, which simple P.E. type is required to execute the operation. It is the additional responsibility of the arbitration network to distribute the workload over all available P.E.'s. This prevents bottlenecks and therefore provides for maximum possible parallelism. A queue is maintained in the prototype for each P.E. type so as to limit unnecessary delays and to simplify the queue management hardware. In the case of generic operations, the template will be passed to a special unit which handles the production of specific templates. Results are forwarded to awaiting templates via the distribution network in the simple case and results are returned to the vector store and their enable tokens forwarded when dealing with generic operations.

The distribution network connects the outputs of each P.E. with all cell blocks. It must be possible for the output of any P.E. to be routed to any cell block since there is no guarantee of which P.E. will execute a given instruction. This means the network must be fully connected, but it does not imply that it must be an immediate connection. Such a network implies a tremendous packaging problem, increases cost dramatically, and is unnecessary since a dataflow machine may tolerate some blockages during the transmission of result packets.

4.2 PROCESSOR STRUCTURE

The prototype system contains eight fundamental P.E. types. These types reflect the various different sorts of operations that may be performed. Despite the fact that there are eight different P.E. types, there are only two different P.E. organizations. These two organizations correspond to those instructions which deal inherently with complex data items (such as a dyadic transpose) and to those which are primarily scalar or which are easily derived from scalar operations. The two organizations differ in the way in which they are designed with respect to accessing the vector store. Those dealing with complex data directly implement small pipelines to speed multiple consecutive accesses to the vector store.

Distinguishing between P.E. types within each organization is a matter of examining the microcode used to implement their functions. A more efficient implementation of all P.E.'s could be done using custom logic but the inefficiencies of a microprogrammed P.E. are more than acceptable for a prototype. Not only is it a cheaper approach but it also provides some margin for error in initial designs. Should a function be incorrectly implemented, it may be changed with relative ease in microcode, whereas it cannot be easily changed in hardware.

Each P.E. is implemented using Am2901 bit slice components. The choice of this family was made for three reasons:

- Microprogrammability
- Extendability (Bit Slice)
- Availability of Information.

A block diagram for a typical processing element is given in Appendix C. Microcode for the P.E.'s is (fortunately) beyond the scope of this thesis. A description of each of the following eight P.E. types is included later in this section.

1) Addition & Subtraction
2) Multiplication & Division
3) Vector Decomposition
4) Matrix Decomposition
5) Pipelined Addition & Subtraction
6) Pipelined Multiplication & Division
7) Logical and Comparison
8) Special Functions

For the prototype, there are two groups of P.E.'s, each servicing one half of the cell blocks. Each group consists of an entire set of P.E.'s (the quantities of which were chosen based on expected usage) as detailed below:

- 3 - add/subtract P.E.'s
- 4 - multiply/divide P.E.'s
- 2 - vector decomposition P.E.'s
- 1 - matrix decomposition P.E.
- 1 - pipelined add/subtract P.E.
- 1 - pipelined multiply/divide P.E.
- 2 - logical and comparison P.E.'s
- 1 - special function P.E.

To avoid problems with locality, the division into halves is in an interleaved fashion. This should guarantee a performance advantage even for programs which occupy only a small(local) part of the activity store. This division will hopefully limit the contention for P.E.'s when only a single P.E. is implemented for that operation type.

Each P.E. has associated with it a hardware queue for incoming templates. The size of this queue depends on the type of the P.E.. The more often a particular P.E. type is expected to be subjected to sudden bursts of activity, the more elements it will have in its queue. Data from the vector store may be prefetched for inclusion in the templates from within this queue. An efficient mechanism for routing such incoming data to the appropriate templates must be provided. A small associative store may be appropriate.

The add/subtract P.E. is the simplest of all. It provides 16-bit two's complement addition and subtraction almost directly in hardware. The Am2901 processor slice will directly perform these operations and thus the microcode should be exceedingly simple thereby providing a P.E. which executes very quickly. This is desirable since simple addition will undoubtedly be the most frequently used operation.

The multiply/divide P.E. is nearly identical in structure to the add/subtract P.E. but makes use of extensive microcoding to perform its functions. This P.E. type will also be used quite heavily and, as such, its turnaround time should be as fast as possible. Due to a microcoded implementation, however, each unit's performance will be somewhat limited. For this reason, four such P.E.'s are implemented within each group. By having multiple units, the throughput rate on series of these operations is increased. Furthermore, the queue size on this unit is larger than that on the add/subtract P.E.. Thus, it is able to buffer a sudden burst of multiply/divide operations.

The pipelined versions of these units (add/subtract and multiply/divide) are nearly identical except for the implementation of a small pipeline to enhance performance for operations such as reductions (+/, -/, x/, etc.). The pipeline divides the execution of each operation into five phases:

1. Operation Identification (Phase O)
2. Result Initialization (Phase I)
3. Operand Fetch (Phase F)
4. Execution (Phase E)
5. Result Handling (Phase R)

The possible overlap in this system is shown in the Gantt chart of Figure 12.

The additional pipeline hardware which enhances access to the vector store should provide a very efficient implementation for reduction and similar operations. Since these are inherently serial operations, they are better performed by a pipeline processor than by a more advanced

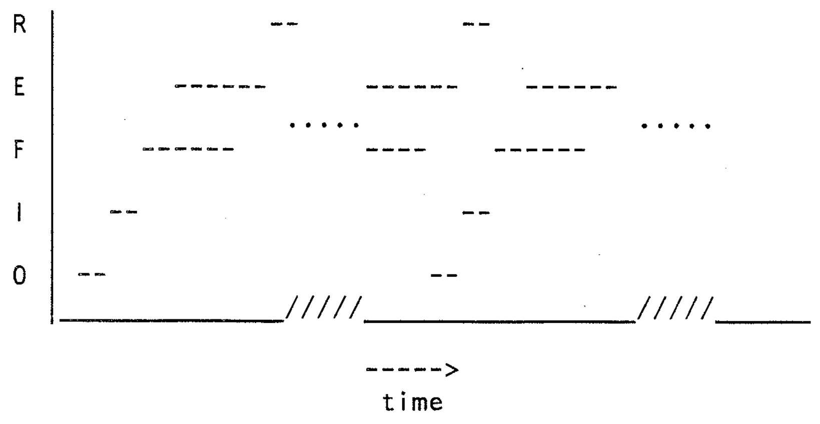


Figure 12: Gantt Chart for Pipeline Overlap

SIMD or MIMD machine.

The logical and comparison P.E. is again a very simple unit with an efficient and small microcode implementation. This P.E. is hardware-equivalent to the add/subtract P.E.. Its microcode is, naturally, entirely different. Unexpectedly, it is also one of the most heavily used P.E. types. This is due to the fact that the conditions section of each rule must generate at least one comparison or logical operation template except in the degenerate case. Due to the number of logical operations that will be generated, multiple units are again implemented.

The matrix and vector decomposition P.E.'s perform simplification (unravelling) operations on generic templates. The matrix decomposer accepts templates referring to data items of rank greater than one and produces templates which describe an equivalent set of operations on data items of exactly rank one. The vector decomposer accepts data items of rank one (generated by either the matrix decomposer or explicitly by the programmer) and generates operation-explicit templates to perform the required operation on an element by element (i.e. scalar) basis. This is a concept similar to the "I-Structure Producer" of Arvind[4].

Both the matrix and vector decomposers operate in conjunction with the activity store manager. This storage manager accepts templates from both decomposers, allocates space for them in the activity store, and copies them into the store. It maintains a free-space list for the activity store and controls the movement of all data both into and out of the activity store. Thus, the downloads and uploads that occur between the prototype and its host are coordinated and controlled by the activity store manager. I/O templates (as will be discussed shortly) are stored in low memory and the storage manager ignores them. It begins allocating template space above these fixed locations.

Unfortunately, once something has been dynamically allocated, it must also be dynamically freed. This represents a significant problem for the storage manager. Dynamic freeing is necessary since template sizes differ between generic and non-generic instructions and thus a simple list of free and allocated template "slots" is inadequate. There must be some means of tracking dynamically allocated templates so that once they have been executed they may be reclaimed. This may be accomplished fairly easily by including a hardware check on entry to each P.E. (excluding the matrix decomposer). At this point, if the P.E. can determine that the template in question was dynamically allocated, it already has the address of the template available and it may directly inform the storage manager. The information required to determine this is easily provided by the use, of a single status bit within each template, which indicates whether it has been statically or dynamically allocated.

The storage manager must also worry about free space amalgamation in an activity store which has the potential to become severly fragmented. This is a problem which does not lend itself to a simple hardware solution and therefore, the storage manager is microprocessor based. The relatively complex functions required are then easily implemented. Clearly, however, the capabilities of a microprocessor will be quickly exceeded in such an environment. To alleviate this problem, special support hardware must also be provided. The microprocessor has access to an associative store for the free list. By making clever use of this store, very fast free space amalgamation may be performed.

Addresses of entries in the queue may be compared to all possible boundary addresses in parallel to determine adjacency. This saves the normal cost of free list searching which is incurred using software. This also provides a more general, albeit more expensive, approach than the "buddy system" for storage management since no restrictions are placed on the size of allocations by the implementation. The benefits of such an associative store are also evident during allocation. If comparison is made for greater than or equal rather than for equality and if a field is provided containing the size of the free areas, then the first responder yields a sort of first-fit selection. See Figure 13.

Finally, the special function P.E. uses a large microstore to facilitate the execution of those APL operations which are not easily or efficiently implemented

size_entry	address_entry
⋮	⋮

Figure 13: Associative Store for Allocation & Freeing

using the P.E. types already discussed. These operations include:

- Grade-up and Grade-down
- Transpose and Dyadic Transpose
- All shape related operators
    - Shape and Reshape
    - Ravel, Laminate, and Catenate
- Take and Drop

The characteristic that distinguishes these operations from all others is a high percentage of execution time spent manipulating bytes in memory. Such storage-intensive operations do not suit parallel execution any better than they do serial. In a single memory such as the vector store, nothing is to be gained from coding the data movement in a parallel fashion. Many parallel requests to memory will simply be serialized by the hardware. To improve this situation, only a single such unit is provided and, as shall be seen shortly, the vector store is interleaved.

All P.E. types have the ability to access data from the vector store. This is necessary since a generic template may generate element templates of any operation type. Unfortunately, a potential for heavy contention is created. At any given time, all twelve P.E.'s as well as the storage manager may be attempting to access the vector store. The use of a high speed memory and a clever cache organization may alleviate many of the problems but is still insufficient to guarantee freedom from contention. Thus, an effort must be made by the rule-based APL compiler to detect compile-time unravellable operations. Wherever possible and practical, the unravelling must be done at compile time.

In an effort to avoid delays at execution time, each P.E. will, if necessary, prefetch data from the vector store while a template is awaiting execution in the queue. Hopefully, this will ensure that the data from the vector store will always be available to the P.E.'s when needed. This is important if a program contains many decomposable vector/array operations since these will generate very many element templates which refer to the vector store.

4.3 INSTRUCTION SETS AND MICROPROGRAMMABILITY

This section describes the instruction set (if it may be called that) implemented on the prototype system. For a complete description of the "template set" see Appendix B.

The instruction set is comprised of a number of generic and non-generic templates which correspond to the primitive operations used to implement all of the rule-based APL operations. Unexpectedly, this amounts to a relatively small number of operation templates. Many of the more advanced APL operations are easily decomposed to expressions involving only these primitive scalar functions.

There are templates required to perform basic addition, subtraction, multiplication and division. Additionally, the comparison operations and the residue function must have primitive templates. Aside from these, however, few others are needed. In fact, others are provided simply for efficiency reasons since they are derivable from these basic operations. For example, signum may be implemented as a series of the following three parallel operations:

```
(x=0) -> [signum<- 0]
(x>0) -> [signum<- 1]
(x<0) -> [signum<- -1]
```

This means that even many of the APL primitive scalar functions may be implemented in terms of still more primitive functions. These most fundamental functions may be referred to as the "basic" functions.

A distinction is made between vector/array and scalar operations. Vector and array operations generate generic templates which have a different format from their non-generic counterparts. A generic template must specify not only the non-generic (element) form of the template but also the extent to which it applies. This amounts to the specification of index ranges for which the element

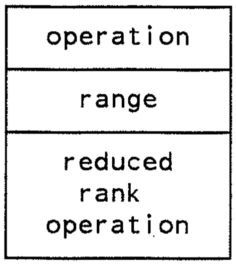


Figure 14: Format of a Generic Template

operation is to be generated. See Figure 14. Thus, template sizes will be directly proportional to the rank of the data item the operation is referring to. For each rank greater than one, both an index range specification and a reduced-rank form must be specified. Matrix addition, for instance, will generate a generic matrix template which specifies a generic vector template as its element template and which generates the range of all possible rows as its extent. Each of these templates in turn will generate a non-generic element template and as its extent the column subscript within the row already established for the vector addition. Similarily, many other non-scalar operations may be decomposed into scalar operations. In general, an n-dimensional generic template generates sufficient n-1 dimensional generic or non-generic templates to cover the nth dimension.

This scheme does yield a large number of templates (one for each scalar element operation plus those for the generic operations) but since the vast majority of them are temporary, the cost may be ignored. An activity store which may accomodate all templates at once is not required. Since many templates will exist only for a short period of time, the activity store may be much smaller. In fact, it may be quite small indeed if an intelligent storage manager is used. When this is the case, the storage manager may simply suspend production of new reduced-rank templates until older ones have executed and been freed.

All of these basic operations may be trivially implemented either in hardware or via microcode. It is the more complex operations which are questionable. What is clear, however, is that should there be any difficulty, it will undoubtedly be easier to implement the operations in microcode than hardware. Thus, a microprogrammed implementation is a good choice, but as with any microcoded application, performance is exchanged for simplicity. High speed performance of these operations is not going to be achieved without a significant investment in custom hardware.

4.4 THE COMPILATION PROCESS

The statement that APL is the base language for the rule-based dataflow processor is not an entire truth. The language used for programming is rule-based APL. It should be clear from the preceeding section that the base language is in fact a compiled version of APL. The hardware does not recognize APL directly but rather accepts a template oriented version which is easily derived from rule-based APL.

The end performance of the prototype will depend quite heavily on the quality of the translation done by the compiler. Since some of the parallelism is being detected by the compiler, it is necessary that the compiler be able to detect and exploit as much parallelism as possible. But, more importantly, it is necessary that the compiler be able to deal with detected parallelism "intelligently". There are instances in which taking the simple route to parallelism will produce an inefficiently compiled form (for example: when compile time unravelling is postponed until run time).

The compilation of APL (as opposed to interpretation) is a feasible but non-trivial problem. The structure of the language is such that it is difficult to compile using simple techniques. This relates to both the syntax and operation of the language.

As far as syntax goes, APL is geared towards interactive execution. This means that there is no concept of a mainline. The mainline is simply whatever is typed in by the user. This is not a distinct problem, since many interpretive languages (such as LISP) have compiled versions which run well in a batch environment. By omitting statements which directly violate compilation (such as the EVAL statement in LISP), an interpretive language may be rewritten as a compiled one. In APL this means that the "execute" operation must be eliminated. Having omitted execute, as is done in rule-based APL, all other language constructs are close to compilable. (The remaining problems will be discussed shortly.)

By defining a rule-based APL program to be a series of function specifications followed by a series of grouped APL statements which use the functions, an unpleasant but acceptable syntax is provided. The series of trailing statements forms the mainline of the program. A facility is also included for grouping mainline statements. By enclosing such statements within '[' and ']', the programmer may specify that the enclosed operations may occur in parallel. This provides the ability to specify procedure-level parallelism.

The argument may be made that the user should still be permitted to interact with the functions provided. This argument is incorrect since the construction of a supercomputer, such as the rule-based dataflow processor, is done to provide the ability to solve large and complex problems. Having user interaction with the program directly conflicts with the theory behind such a machine. Either the user will slow the machine down due to heavy interaction or, if the problem is suitably complex, the user will be left waiting during the solution of intermediate problems. Thus, it makes more sense to include the function invocations as a part of the program (i.e. as a mainline).

Two operational features of APL also make its compilation difficult. Firstly, the type of a variable is undefined until run time. Fortunately, this is not a problem in rule-based APL. Due to the support of only integers, the partial typing (as to rank) of variables and the single assignment rule, the compiler need not worry about a variable's type changing during the execution of a program. Declaration of variables also solves a problem of parseability. (The statement 'A+B' may be ambiguous. Its two possible interpretations are dyadic '+', and monadic '+' where A is a function with a single argument.) The second feature of APL which causes difficulty is that the shape of a data item of rank greater than one is not fixed. The size of a data item is not static and may change as execution proceeds. This means that the compiler must generate special run-time information which is to be stored with each data item. The net result is increased code size due to the addition of code for both run-time checks and code to calculate information at run-time. Additionally, a relatively substantial package of runtime routines must be supported. In particular, both a good garbage collector and/or space allocator must be provided. There is some question, however, as to whether these should be implemented in software or in hardware.

In general, the translation of a rule-based APL statement will be as follows: Each statement consists of two portions, a conditions section and an actions section. Statements from the conditions section will be translated to a series of conditional templates as appropriate and necessary. The output of the last template in this series will serve to enable the templates corresponding to the actions section of the rule. Each action within the actions section will generate at least one generic or non-generic template and more likely will generate a sequence of templates. The completion of these actions will then indirectly enable other rules and their associated templates.

4.5 MEMORY STRUCTURES

The system memory is divided into two parts:

- Activity Store
- Vector Store

These two parts serve different functions (as already described) and their organizations differ greatly due to function.

The activity store is a simple memory, being no different in organization from a single bank of memory in any microprocessor. There is only one unit which fetches enabled templates from the activity store and the fetches are in no way related. This means that the use of multiple memory banks and/or a cache has no affect on performance. Due to the nature of a dataflow machine, locality of reference is low.

The vector store on the other hand is exactly the opposite. It will contain vectors and arrays organized in conveniently adjacent locations. These vectors are accessed more or less as a whole (albeit by many different templates) and thus locality of reference is high. A prefetch buffer may be loaded with data from the vectors with assurance that the data will soon be required. Despite the fact that different templates are accessing the data, references should be fairly uniform. All such element templates will be enabled at the same time and therefore will likely begin to execute at approximately the same time. This means that much may be gained by using conventional memory enhancement approaches. In the prototype, this implies fetching four consecutive words from the vector store at a time and the use of a large high speed buffer. These additions should provide a significant increase in performance while the machine is executing vector operations. Their use, however, is not as straightforward as it may seem.

Prefetching multiple bytes per access is no problem. Multiple banks are used and a fetch from some location results in the fetch of corresponding locations in other banks. A cache, however, presents a somewhat different problem. Although the basic strategy is the same, a dataflow machine forces the development of a new cache replacement algorithm. In a dataflow machine, it is quite conceivable that a required block may go unaccessed in the cache for some time. A vector operation may be started which creates templates to perform the desired operation on each vector element. These element templates are then queued for execution by the P.E.'s just as any other templates are. Thus, a vector may be fetched into the cache as the first few element templates are executed. After this, all available P.E.'s may be occupied executing different templates and it may be some time before another access to the vector occurs. This does not present a problem if no other vectors within the cache are being accessed. However, if they are (i.e. if the other templates being executed are for generic vector instructions), then a normal cache replacement algorithm might replace blocks which are soon to be needed.

In both memories, there is the possibility of overflow occurring. In a practical system, this possibility would have to be dealt with by some sort of automatic data migration. This migration would take the form of paging hardware and software in a conventional computer system. In a dataflow system, this approach is not feasible due to the lack of execution locality. A swapping system may be more appropriate within certain loading restrictions. Fortunately, this problem is not addressed in the prototype system. Should an overflow situation arise, it will be detected and the system will shut down gracefully.

4.6 ROUTING NETWORKS

The type of routing network employed in this design is known as a Benes network. Although it is a blocking network (i.e. some delays in routing may occur due to channel activity), it does offer a general connection between n inputs and n outputs. This network ideally meets the needs of a dataflow machine. It is cheap yet permits direct connection from input to output and, due to the nature of a dataflow machine, its inherent inefficiencies may be tolerated.

The construction of such a network is trivial when given a simple 2x2 routing element as a building block. Figure 15 gives a description of an 8x8 Benes network and Figure 16 describes the 2x2 router on which it is based. A simplified circuit diagram for the router may be found in Appendix C. Note that the REQ and ACK handshaking signals shown in Appendix C are required due to the asynchronous nature of

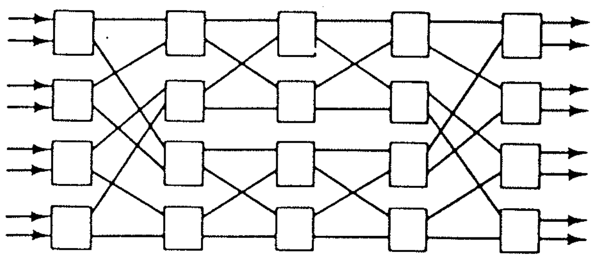


Figure 15: An 8x8 Benes Routing Network

the network. The choice of an 8x8 router is primarily because such a router is VLSI implementable. It is likely that a more advanced 64x64 version might also be fabricated and, should this be the case, its design may be easily extrapolated from the 8x8 network shown.

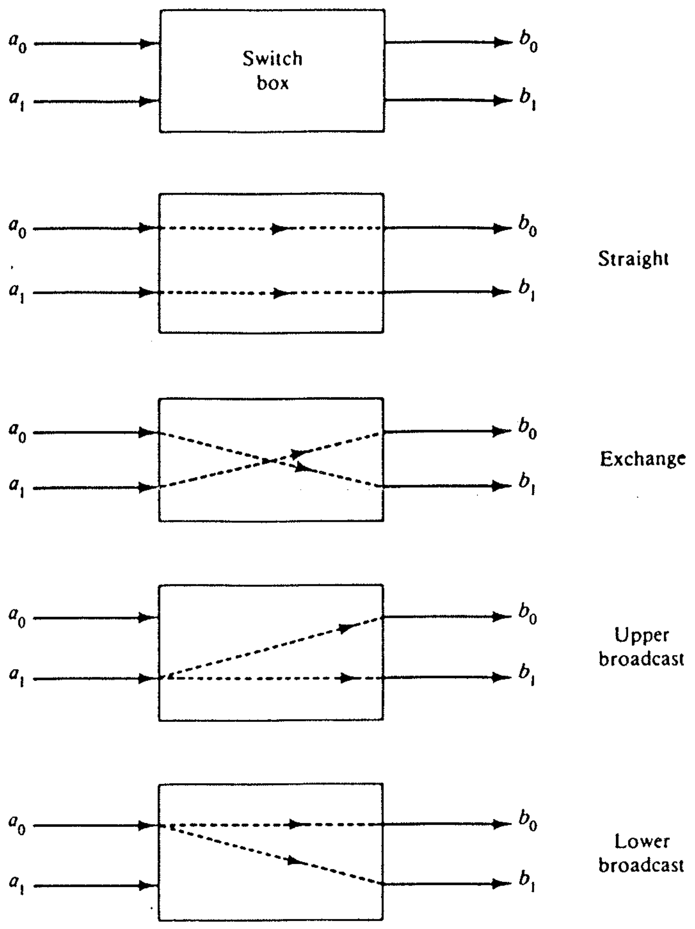


Figure 16: Actions of a 2x2 Routing Element

The Benes network was chosen as the cheapest form of network that met all the requirements. A baseline network is cheaper and does meet the general N:N connection criteria but suffers a higher probability of blockage. If a still simpler network were to be used, recirculation would be necessary and this would add to the hardware complexity. On the other hand, a Clos or similar network would provide the required connectivity but its design makes it cost-prohibitive. In a Clos network cost is incurred for the unnecessary feature of non-blockage.

Each 2x2 router accepts a packet which has the form:

dest.	data

The router examines the leftmost bit to determine on which of its output lines it should place the packet. Before forwarding the packet, it does a rotate one bit position to the left. After passing through the entire routine newtork, the result is a packet which has the form:

data	dest.

This approach also guarantees that, at every stage, the bit to be used in routing is in the leftmost position (this is convenient for implementation purposes). Given an 8x8 Benes network constructed of these 2x2 routing elements, an efficient network for dataflow machines may be constructed.

There are five levels in each 8x8 network but only eight possible outputs. At first, this seems to present a problem. Five levels of 2x2 routers requires five bits of destination address for routing but only three are required to select an output. Consider Table 2. From this table it is clear that the destination (output line) may be selected using only the last three bits of the five. Fortunately, due to the structure of the Benes network, the first two destination bits are "don't cares". This means that they may be chosen so as to maximize efficiency. By dynamically choosing from

Output	Destination Address
0	xx000
1	xx001
2	xx010
3	xx011
4	xx100
5	xx101
6	xx110
7	xx111

TABLE 2

Choice of Destination Bits

the four possible paths, the likelihood of collisions is reduced and so is network congestion. The code set of output/destination addresses for an 8x8 network at some

Output	Destination Address
0	00000
1	11001
2	01010
3	10011
4	10100
5	01101
6	11110
7	00111

TABLE 3

Output Address Code Set

arbitrary time might be that shown in Table 3. Thus, by using this network, the overhead of extra address bits is incurred (causing buses to be wider) but simplicity and relative efficiency are maintained.

4.7 THE I/O SUBSYSTEM

A fundamental question in any dataflow system and one which has not yet been addressed in this thesis is that of how I/O is accomplished. If the dataflow machine is treated simply as a high speed attached processor for computationally complex problems, then I/O is non-critical. In this case, any data may be downloaded with the dataflow program to be executed. This effectively eliminates the need for an I/O system since data can be downloaded, and results may be uploaded at program completion. If a general purpose dataflow machine is to be implemented, however, a truly sophisticated I/O subsystem must be incorporated. Users must certainly have direct access to the machine via a terminal or other peripheral device and there will doubtlessly have to be support for a file system and likely for a demand-paged operating system as well. The I/O subsystem is normally seen as fitting into a dataflow system in the manner shown in Figure 17.

It is not immediately clear how an I/O subsystem should physically fit into a dataflow architecture. Consider, particularily, such advanced devices as DMA disks and bit-

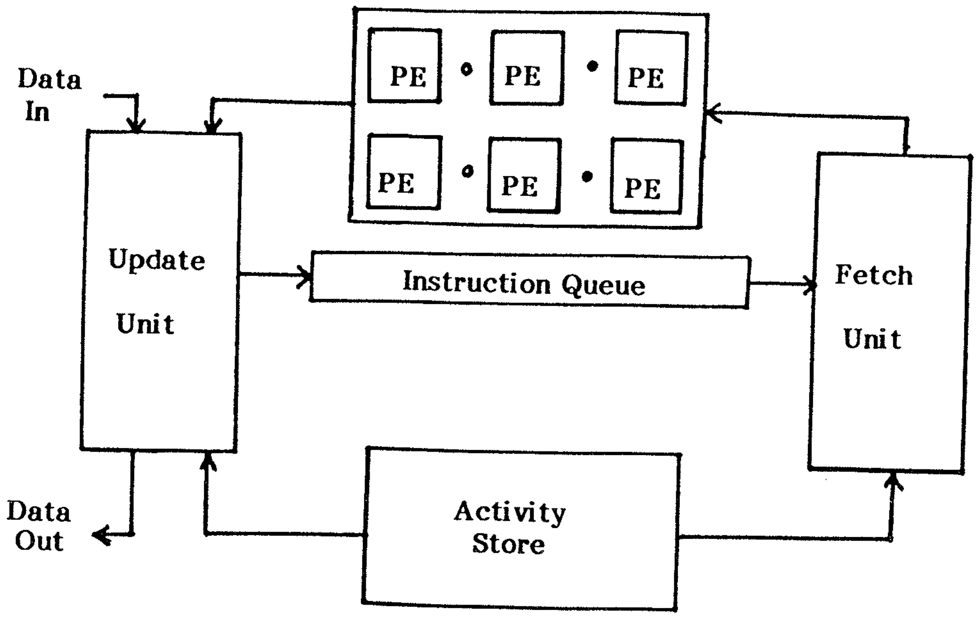


Figure 17: The Dataflow I/O Subsystem

mapped displays. It would seem that these devices are partially incompatible with the dataflow model and that special considerations must be made in order to provide support for them. If all I/O devices are divided into two classes based on whether or not they perform DMA then the classes may be dealt with separately and more simply.

Consider first the non-DMA devices such as terminals and printers. These peripherals are controlled in one of two ways. They are either interrupt-driven or polled. This distinction may disappear in a dataflow machine depending on the choice of implementation. Each device may be thought of as producing a byte of information asynchronously which is then to be used in one or more templates (in the sense of input at least). Before data has actually been input, the templates making use of that data will be disabled. Once it is input, those templates should be enabled. This enabling is accomplished at the time of template update just as it would be if the data had been produced by the execution of an operation from another template. The connection between an I/O-produced result and a template-produced result may be made if a new template type is considered.

Assuming the memory mapped I/O scheme used in most micro and mini-processors, we may define special input templates at the locations in the activity store corresponding to the I/O ports of various devices. When a byte is input, it is placed into the data portion of the input template and the enable bit for that template is set. The setting of this bit is analogous to the setting of a "register-full" bit in the status register of a peripheral device (or to the generation of an I/O completion interrupt). When this template is enabled, it functions as a destination-extension template. In other words, the data value will be distributed to one or more waiting templates. These templates will be those that were awaiting the input of the data value.

So far, this description has dealt with the handling of input operations only. Output operations may be handled in a similar manner. In the case of output, however, the dataflow program must wait only until the desired output device is available. Thus, the template which copies data into the output-template should be enabled when that device is ready to accept another datum for output. This is only slightly different from the input scheme and is no different from standard computer systems. For example, the M6850 serial asynchronous interface for the MOTOROLA 6800 family of microcomputers provides a TDRE (Transmit Data Register Empty) flag in its status register. This flag indicates when it is prepared to accept more output.

The format of input and output templates is shown in Figure 18. The input template is very straightforward having a data field and three (an arbitrary number) destination address fields. The output template contains a data field and a single destination address. However, this destination address is used in a different way for output templates. Instead of routing data produced to the given destination, it accepts data from it when enabled. This may also be thought of as passing an enable token to the template

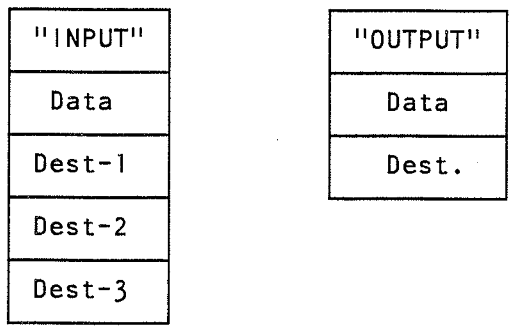


Figure 18: Input and Output Template Formats

producing the data to be output.

The class of devices which support DMA must be handled in a very different manner. The technique of using status information returned by the device/peripheral to enable appropriate templates is still valid except that data receipt/transmission must be handled differently. Normally, data is seen as being an integral part of the templates in a dataflow machine. This is easily maintained in the non-DMA devices through the use of destination-extension templates. It is, however, not feasible for DMA transfers. This implies that DMA must be performed to various scattered addresses within the activity store which correspond to template locations. The situation may be improved upon slightly if the DMA transfer is made to a fixed set of addresses which then broadcast the data to appropriate templates (much as was done in the non-DMA case). From a physical viewpoint, however, this is exceedingly restrictive and difficult to implement. Furthermore, it presents a difficult programming environment. A better solution is to permit data and templates to be isolated from one another such that the templates simply contain pointers to data areas within the separate vector store (just as was considered earlier for vector operations). The instruction which initiates the operation may then partially enable an instruction which will await the I/O completion interrupt in order to provide the balance of its enabling requirements. (The partial enabling is exactly analagous to passing only one value to a template which requires two for enabling.) This template may then enable other templates to continue with the data that has been input. Again there is a similar approach which may be taken for output.

I/O, as usually envisioned on a serial machine, is inappropriate on a dataflow computer. Most algorithms tend to run I/O loops (with data processing within the loop) on a single group of input data. This is clearly inefficient in a case in which sufficient processing elements exist to allow more than one set of data to be processed concurrently. It would be far better if multiple data items could be read and processed at the same time. Exactly how this should be accomplished is not clear. Should data be stored on multiple volumes so that more than one datum may be loaded for processing? Should data simply be prefetched in sufficient quantities from a single device or should some other foreign approach be taken? The multi-volume approach is unlikely. Firstly, the complexity of maintaining a dynamic database on multiple volumes makes this approach much less than desirable. Secondly, it is impossible to use on systems where only a few storage volumes are available.

The "buffering" technique is a far more likely candidate. Not only is it a concept with which programmers are familiar, but also one which is easily implemented. It is, of course, inappropriate for any interactive applications where storage is not involved (i.e. where values are directly consumed/produced by a running program) but so are all other schemes. Although a dataflow machine is not seen as being an ideal interactive architecture, if a demand paged activity store and vector store are provided, interactive applications may be performed no less efficiently than on current architectures.

The prototype system currently being described does not support I/O operations of a general nature. Instead, it simply provides the more common upload and download facilities and support for non-DMA devices. These capabilities are sufficient to permit testing of the data driven mechanism. If a second prototype were to be designed following the success of the first, a major area of concentration should be the I/O subsystem.

Chapter V
CONCLUSIONS

Both the rule-based and dataflow systems are quite similar. This thesis has shown that they are, in fact, isomorphisms of one another. The dataflow model provides an efficient system on which to implement a rule base, while the rule-based model provides an improved programming environment for dataflow machines. These two systems do, in fact, suit one another very well and when merged, provide a foundation for a viable parallel system.

5.1 FEASABILITY

The question of feasability is intimately tied to the implementation of the data driven mechanism. Since the software model seems sound (particularily in certain application areas), it should not present any overly complex problems. The limititations on implementation will undoubtedly be imposed by the hardware. In particular, bandwidth problems will likely limit the size of feasible implementations.

Nevertheless, successful prototype dataflow systems have been built and this lends support to the belief that the system is feasible. This success, coupled with the advances being made in VLSI, should make the rule-based dataflow processor not only feasible but also economically viable within a few years. As the cost of VLSI designs comes down, it will be possible to efficiently fabricate the various system components of the rule-based dataflow processor.

All this speculation about future trends and system efficiency is encouraging. However, very little accurate information may be gleaned without the use of extensive simulations and a running prototype. Realizing this, further conjecture on feasibility is inappropriate.

5.2 APPLICATIONS

Clearly, there is a wide range of applications for any parallel system and the rule-based dataflow machine is no exception. With a suitable base language, the system provides a very usable parallel environment. If, for instance, the base language were chosen to be APL (as in the prototype but with added support for both real numbers and characters), it would be useful in many areas such as graphics and simulations.

The rule-based dataflow processor does not represent a general applications programming system (after all, FORTRAN is not supported). Rather, it is oriented towards the construction of expert systems. Nevertheless, the applications domain which is addressable may be changed quite drastically (within the rule-based format) by a change in the base language (this change might even be accomplished via a microcode load). Possible alternate language candidates are PROLOG[25], LISP and other functional languages. Also, some research has advocated the use of a dataflow machine in database applications[10]. Consequently, some sort of relational calculus or algebra may also form a base language.

If the fifth generation push takes hold, and the new directions in programming (PROLOG, etc.) are accepted, then the rule-based dataflow concept may yet prove to be a general applications system. If not, by varying the base language and providing some hardware support, it will still be a useful system for many problems. Also, if a programmer is willing to sacrifice the benefit of middle-level parallelism, he may code in a more familiar language (APL,LISP,etc.) and still enjoy a significant performance benefit.

Appendix A

RULE-BASED APL

This appendix serves to provide a detailed description of the syntax of the language used to program the prototype rule-based dataflow processor. The following documents the APL operators supported for the prototype:

<	comparison for less than
>	comparison for greater than
=	comparison for equals
≥	comparison for greater or equal
≤	comparison for less or equal
≠	comparison for not equal
∨	logical OR
∧	logical AND
⍱	logical NOR
⍲	logical NAND
-	negation
-	subtraction
+	identity
+	addition
÷	division
x	signum
x	multiplication
?	roll
?	deal
∊	membership
⍴	shape
⍴	reshape
¬	logical NOT
↑	take
↓	drop
⍳	index generator
⍳	index of (ranking)
⌽,⊖	reversal
⌽,⊖	rotate
⍉	transpose
⍉	dyadic transpose
*	power
⌈	maximum
⌊	minimum
⍋	grade up
⍒	grade down
!	factorial

!	combinations
[]	indexing
⊥	decode
⊤	encode
|	absolute value
|	residue
,	ravel
,	catenate
,	laminate
f/,f⌿	reduction
f\,f⍀	scan
/,⌿	compression
\,⍀	expansion
o.f	outer product
f.g	inner product
⎕	input/output
←	assignment
<⊦	renaming
->	implication (rule-based)

These operations appear within a general syntax for rule based programs. The syntax of a rule-based APL statement is:

(APL conditional expr.) -> [APL statements]

Where: 'APL statements' is a series of APL statements separated by diamond characters ('◇').

This means that a typical rule-based APL program will consist of a number of these statements assembled as rule-based APL functions. The structure of such a function is given below:

```
∇R← OP1 NAME OP2 ; VAR-DECLS
[ <INITIAL-CLAUSE> ]
            .
            .
BODY OF FUNCTION
            .
            .
∇
```

The format for "VAR-DECLS" is as would be expected for any APL local variable declaration. The only exception is in the case of multi-dimensional (rank greater than one) variables. They are suffixed by a subscript specification which gives their rank. For instance, a declaration of "FLANGE[2]" would give the variable FLANGE a rank of two. Being consistent with APL, the exact size of each variable may change within this rank. Variable declarations are separated by semicolons.

The <initial_clause> is optional and contains a series of initialization statements. The syntax is identical to that of a <mainline> as will be discussed shortly.

The point of interest in this syntax is the specification of variables as being either scalar or N-dimensional array. This must be done for ALL variables in a rule-based APL program. Other important differences between APL and rule-based APL are the single assignment rule, lack of global variables and the illegality of modifying input parameters to a function[3]. All of these "restrictions" are made to ensure that no side affects will exist in the compiled program.

---

[3] In actuality, modification of input parameters is not syntactically incorrect. Rather, the assumption that the values will be propogated back is. Note that this is consistent with the use of the renaming operator.

The structure of a rule-based APL program is the following:

```
<program> ::= ∇∇ <progname> IS { <function_defs>
                                  <mainline> }  ∇∇

<function_defs> ::= <function_def> |
                    <function_def> <function_defs>

<function_def> ::= ** a function definition as
                      previously described. **

<mainline> ::= <mainline_stmts>

<mainline_stmts> ::= <mainline_stmt> |
                <mainline_stmt> <mainline_stmts>

<mainline_stmt> ::= [ <apl_stmts> ]

<apl_stmts> ::= <apl_stmt> |
                <apl_stmt> <apl_stmts>

<apl_stmt> ::= ** any legitimate APL statement
                  within the constraints of the
                  rule based implementation **
```

There is a strict declaration-before-use rule which accounts for the declaration of functions at the start of the program. The grouping of statements in the mainline using "[" and "]", permits the specification of parallelism within the mainline. This simplified syntax for parallelism will allow inexperienced parallel programmers to make effective use of the system via the invocation of pre-written parallel functions.

Appendix B
INSTRUCTION SET

This appendix describes the instruction formats used in the prototype system. They correspond directly to the APL operations supported and the special functions (i.e. generic templates) provided.

The various rule-based APL statements are divided into groups based on their implementation. This subdivision is based on their fundamental functions and how they relate to the P.E. types discussed earlier.

The logical and comparison P.E. has by far the largest instruction set which consists of twelve primitive operations. These operations support sixteen fundamental APL operations. The twelve operations are the following:

LT	Opcode=00	Less Than
LE	Opcode=01	Less Than or Equal
GT	Opcode=02	Greater Than
GE	Opcode=03	Greater Than or Equal
EQ	Opcode=04	EQual
NE	Opcode=05	Not Equal
OR	Opcode=06	logical OR
AND	Opcode=07	logical AND
NOT	Opcode=08	logical NOT
NOR	Opcode=09	logical NOR
NAND	Opcode=0A	logical NAND

The comparison operations generate dyadic comparison templates which produce general true and false enable signals. All logical operations are dyadic except of course for NOT. These primitives also directly support the MINIMUM, MAXIMUM, SIGNUM, MEMBER and INDEX operations.

The addition and subtraction P.E.'s support five basic operations. They are the following:

NEG	Opcode=10	arithmetic NEGation
ABS	Opcode=11	ABSolute value
ADD	Opcode=12	2's complement ADDition
SUB	Opcode=13	2's complement SUBtraction
SHP	Opcode=14	APL SHaPe operation

The first four operations are very straightforward and as one might expect. SHP (shape) is included with the addition and subtraction P.E. primarily as a convenience. The execution of a shape operation amounts to a simple memory access to retrieve the shape information prefixed to each data structure. Both ADD and SUB are also implemented in the vector addition and subtraction P.E..

The P.E.'s which support multiplication and division have three operations to perform:

MUL	Opcode=20	2's complement MULtiplication
DIV	Opcode=21	2's complement DIVision
RES	Opcode=22	2's complement RESidue

All of these operations are also implemented in the pipelined version of this P.E.. The non-pipelined multiplication/division P.E. is also solely responsible for the execution of APL's FACTORIAL, COMBINATION and POWER operations.

All storage intensive operations are handled by the so-called special P.E.. The operations supported by this P.E. support the execution of the following APL operations:

reshape
take
drop
compress
expand
ravel
catenate
laminate
transpose
dyadic transpose
rotate
reversal
index generator

In order to perform these operations, the following storage management primitives are provided.

TFR	Opcode=30	TransFeR bytes in memory
ALL	Opcode=31	ALLocate memory
FREE	Opcode=32	FREE memory

These operations provide the basis for the storage manipulation required to perform many basic APL functions. ALL and FREE make their requests directly to the storage manager.

Certain APL operations must be implemented in software. There is no means of implementing their functions in hardware without going to great expense. For the amount of use many of the more esoteric functions (such as ROLL and DEAL) get, it is not worthwhile to attempt an efficient hardware implementation. Instead, these operations are implemented as a series (or hopefully a collection) of templates which together accomplish the required function. For example, the ROLL operation might be implemented using a simple congruential random number generator. The only thing that these operations have in common is that they all make use of the basic machine operations to accomplish their functions. The functions which are implemented in this way are:

roll
deal
subscripting
grade up
grade down
encode
decode

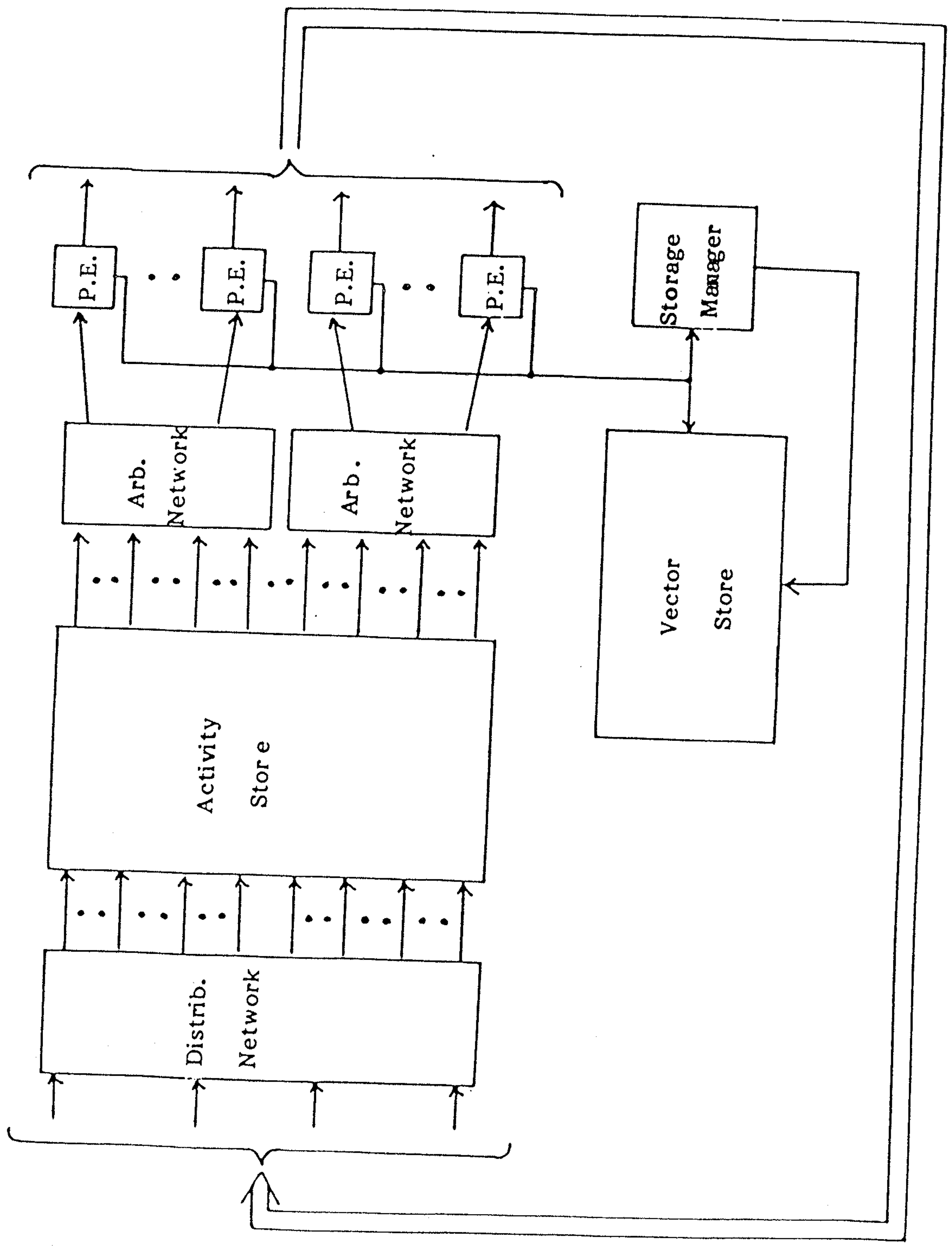
Finally, the decomposition P.E.'s offer an alternative approach to use of the pipelined units. They are made use of in those circumstances where the compiler cannot determine that a pipelined unit can be safely used or when it determines that a non-pipelined approach is more efficient. These units are envisioned as handling a major part of the workload for operations on complex data structures. The instructions they support are exactly those which are supported by the other element P.E.'s. The template formats consist of an opcode and references to the base addresses of the data structures involved. The high order three bits of each opcode delimit up to eight levels of complexity (although only two are really needed). A pattern of '111' might represent a seven dimensional data structure. It may then be reduced to a collection of six dimensional operations dynamically by the matrix decomposer and so on until only element operations remain. The reduction consists of reproducing the template with the top three opcode bits decremented by one and with new base addresses calculated.

Some general comments may also be made about this instruction set. Firstly, I/O (the QUAD operator) is implemented using I/O templates at fixed locations as described earlier. Therefore, they were not included in this discussion. Secondly, the upper nybble of the byte-wide opcode determines the P.E. type required to execute the instruction. This simplifies decoding greatly. Finally, the use of the reshape operator is limited. This is due to the restriction in rule-based APL which disallows changing the rank of a data item dynamically. Thus, reshape may alter the extents of a data item but not its rank.

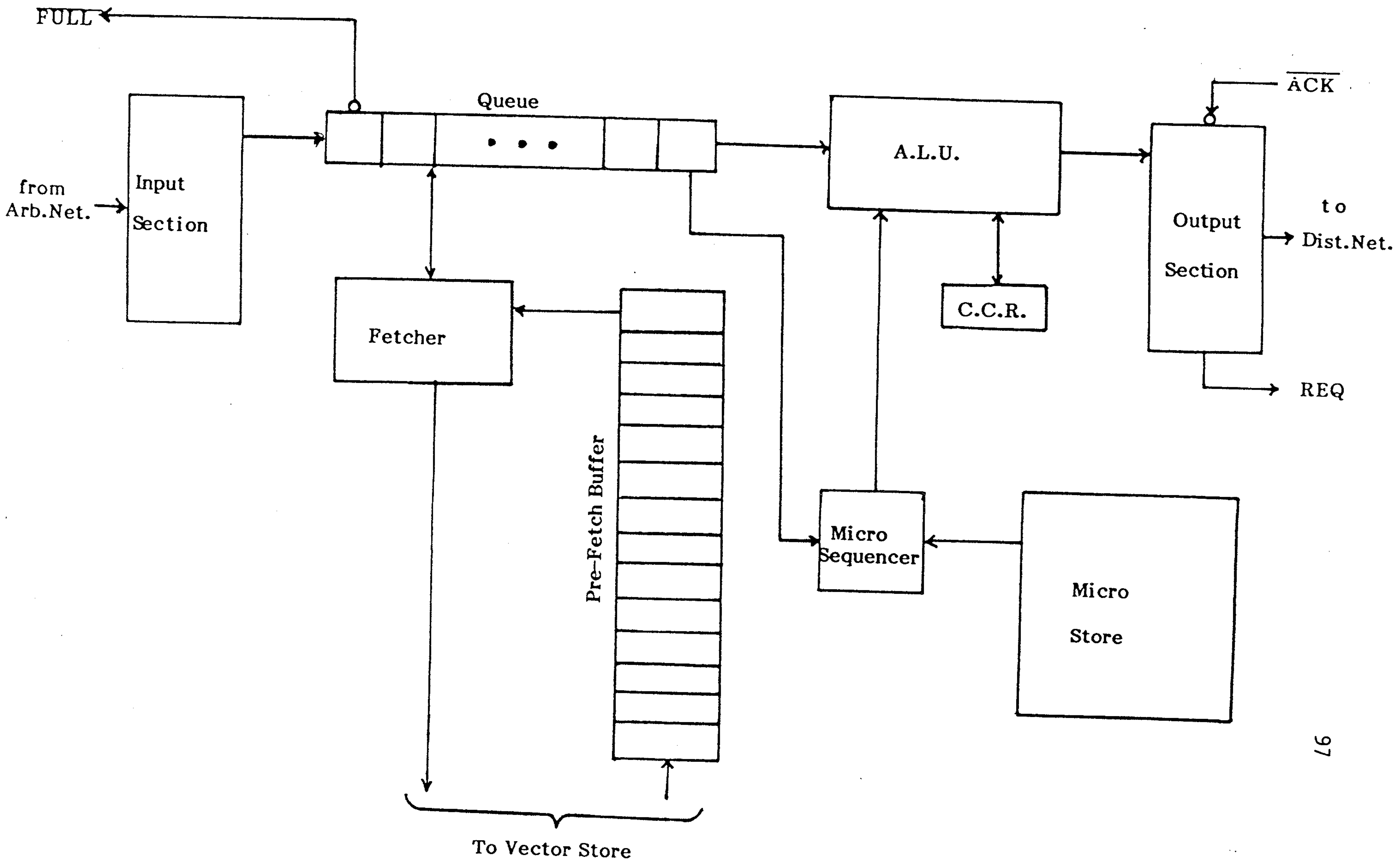
Appendix C
CIRCUIT DESCRIPTIONS

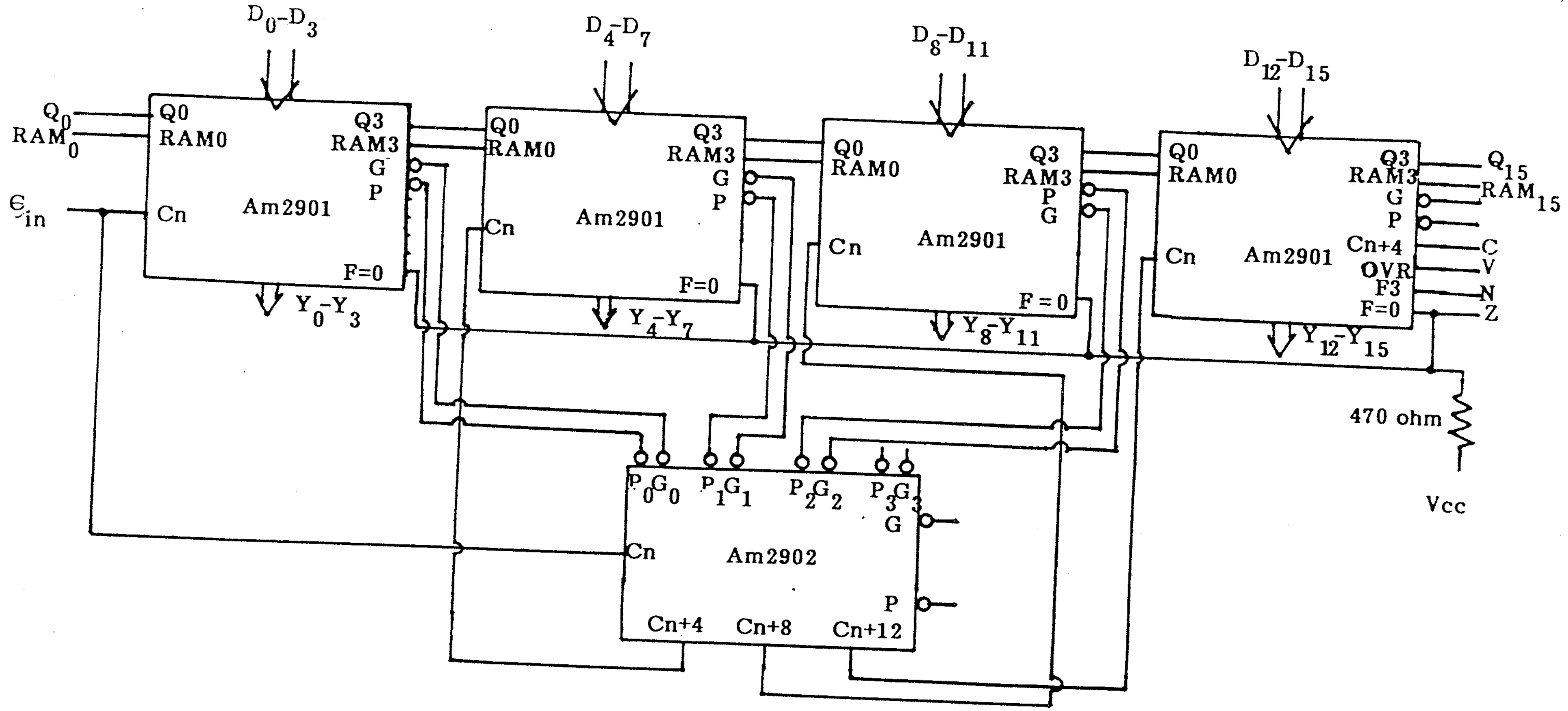
This appendix contains a collection of block-level diagrams describing the various components of the rule-based dataflow processor. The thesis does not discuss the gate-level implementation of the prototype processor.

Overall System Structure

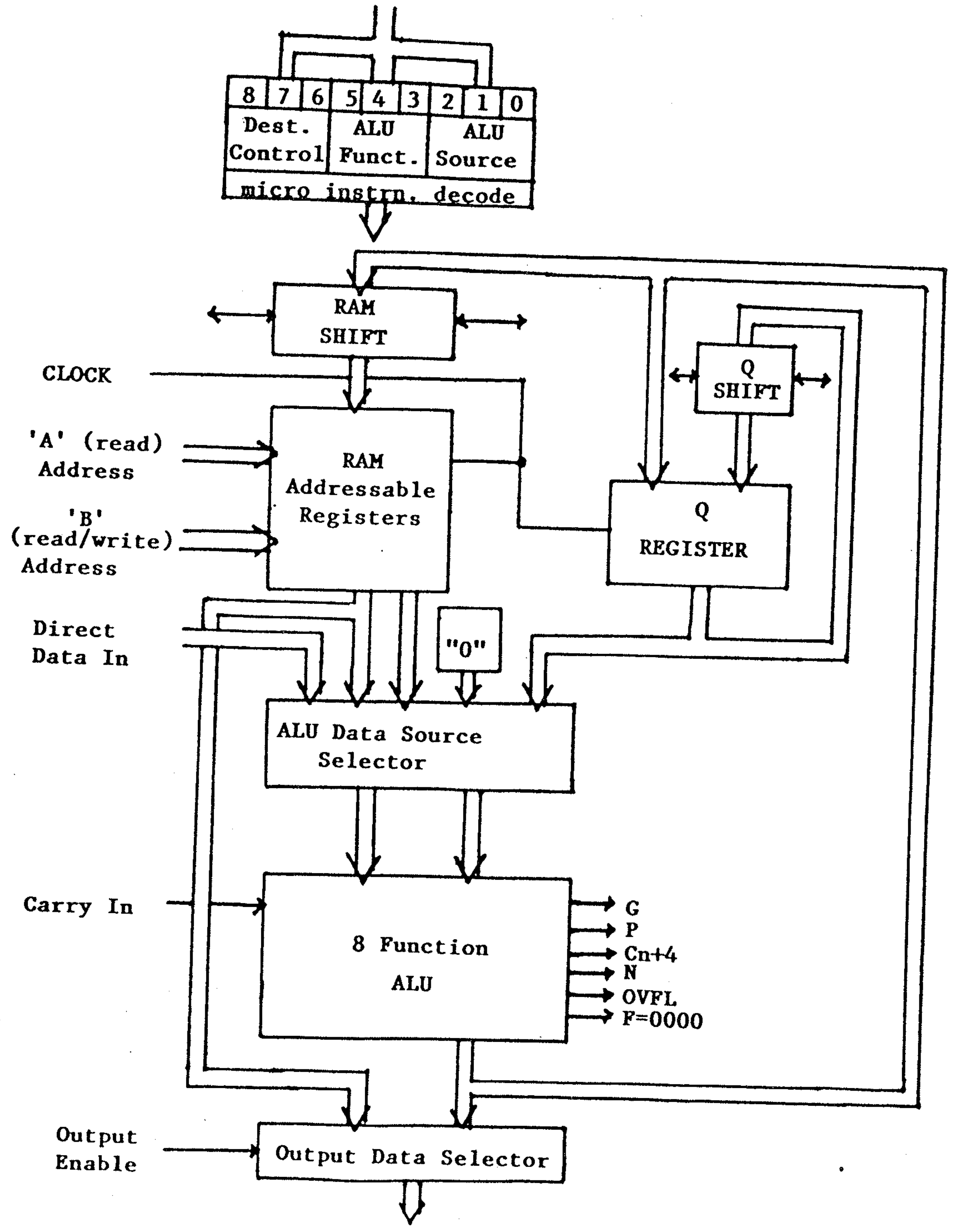


Processing Element Structure

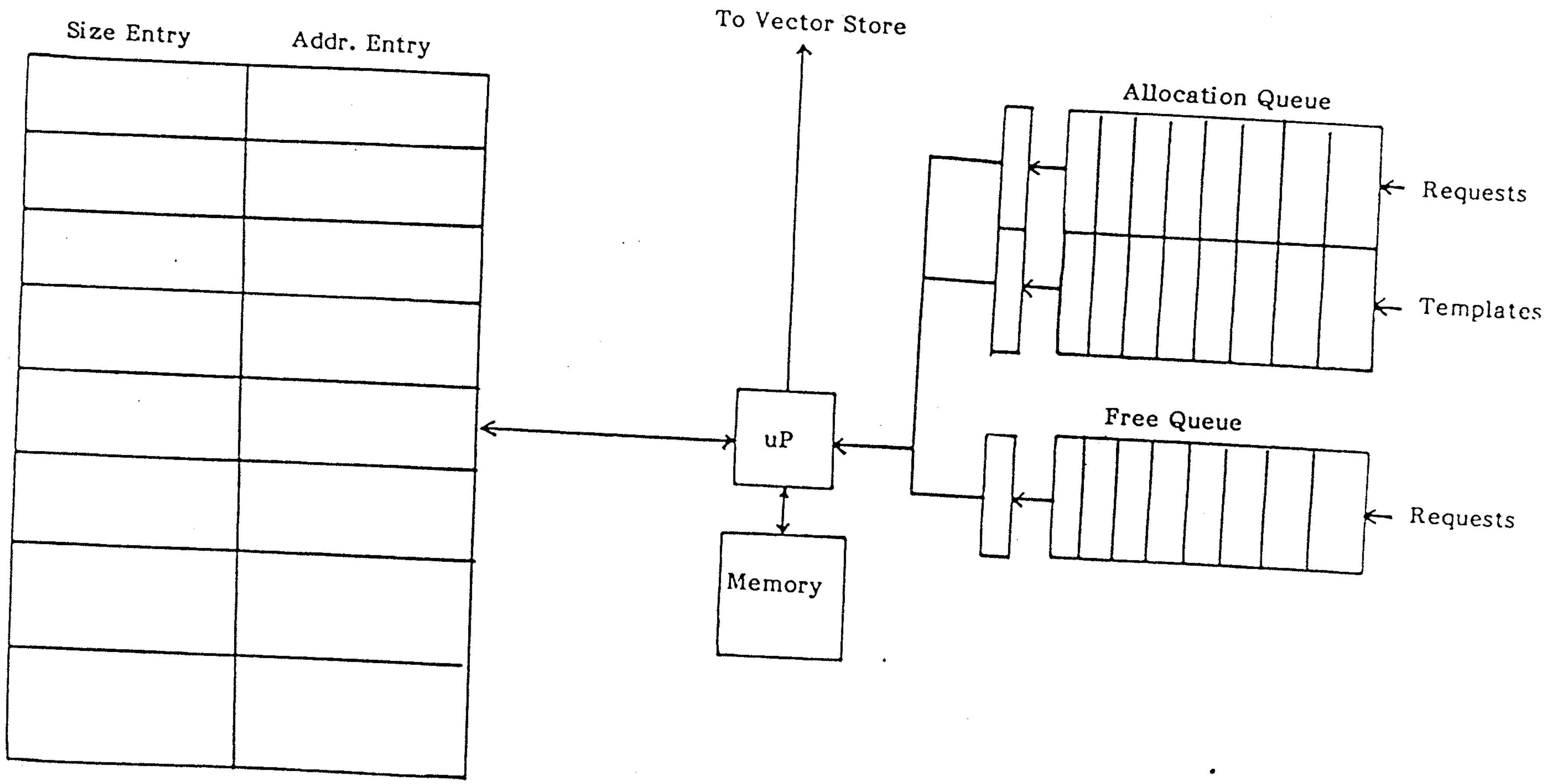


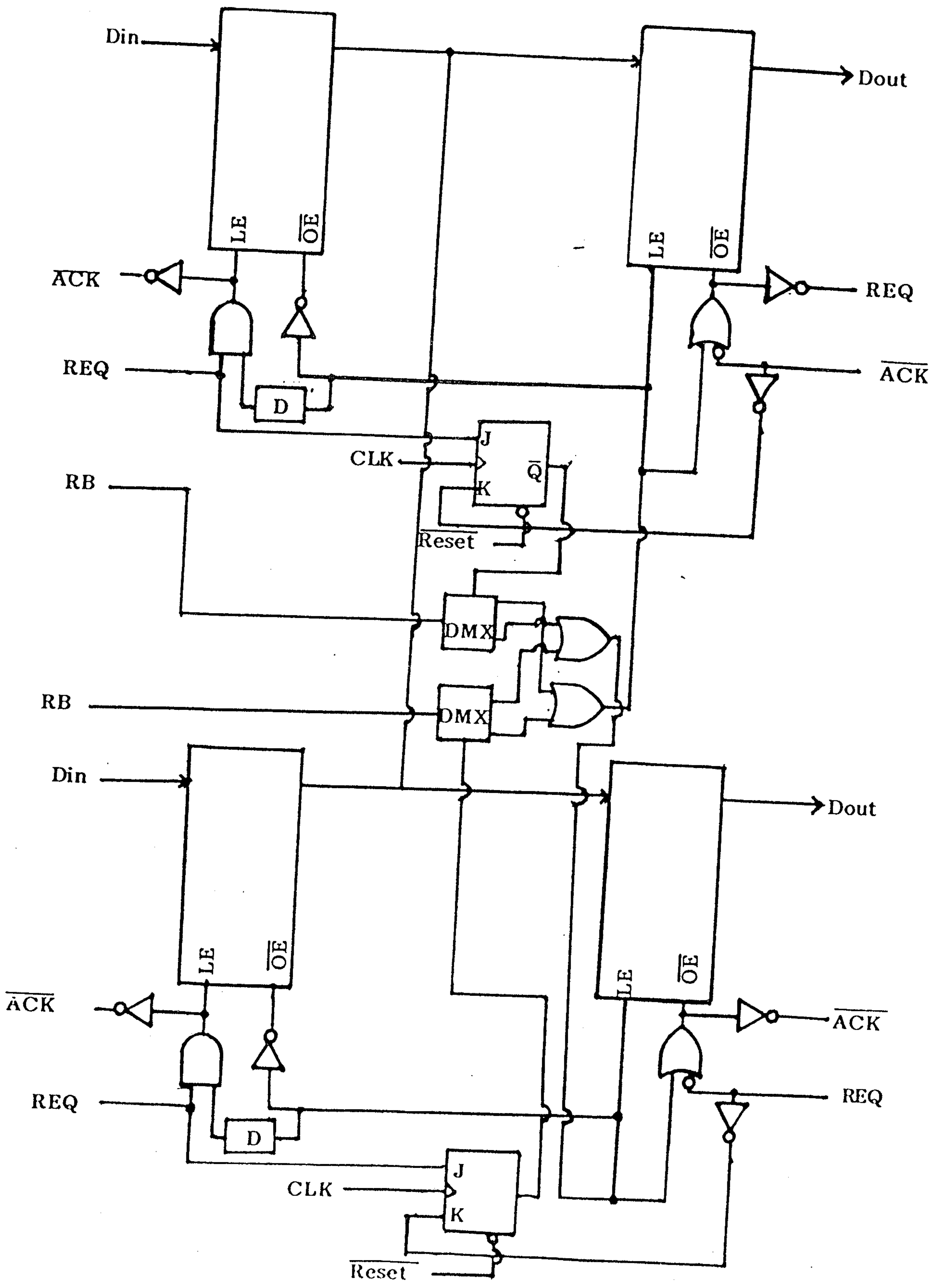

D0-D3
D4-D7
D8-D11
D12-D15
Q0
RAM0
Cin
Q0
RAM0
Q3
RAM3
G
P
Cn
Am2901
F=0
Y0-Y3
Y4-Y7
Y8-Y11
Y12-Y15
Cn+4
OVR
F3
Q15
RAM15
C
V
N
Z
470 ohm
Vcc
P0 G0 P1 G1 P2 G2 P3 G3
Am2902
Cn+8
Cn+12

Internal Bit Slice Organization



Storage Manager




Din
LE
$\overline{OE}$
Dout
$\overline{ACK}$
REQ
D
CLK
J
K
$\overline{Q}$
RB
$\overline{Reset}$
DMX
DMX
RB
Din
LE
$\overline{OE}$
Dout
$\overline{ACK}$
REQ
D
REQ
CLK
J
K
$\overline{ACK}$
$\overline{Reset}$

Appendix D

PROGRAMMING EXAMPLES

This appendix gives three examples of rule based APL programs. They are:

1. A parallel sort
2. A solution to the convolution problem
3. A line enhancement process

```
⍝
⍝ NOTE THAT THIS EXAMPLE IS RATHER UNLIKELY AS SUCH SORTING
⍝ WOULD NORMALLY BE DONE USING THE GRADE-UP OPERATOR (⍋).
⍝
∇∇SORTERPROG IS
∇R←SORT VEC[1];EV1[1];EV2[1];OD1[1];OD2[1];BITSTR[1];PLACES[1];
                TMP[1];GO1;GO2;DONE1;DONE2
⍝
⍝ THIS FUNCTION SORTS AN INTEGER VECTOR INTO ASCENDING ORDER.
⍝ THE ALOGRITHM USED IS GUARANTEED O(N) IF NO PARTITIONING OF
⍝ THE PROBLEM IS DONE AND IS AS FOLLOWS:
⍝
⍝   REPEAT UNTIL SORTED...
⍝         1) COMPARE ALL PAIRS OF ADJACENT NUMBERS BEGINNING
⍝            AT EVEN INDEXES AND SWAP ALL THOSE WHICH ARE OUT
⍝            OF ORDER.
⍝         2) COMPARE ALL PAIRS OF ADJACENT NUMBERS BEGINNING
⍝            AT ODD INDEXES AND SWAP ALL THOSE WHICH ARE OUT
⍝            OF ORDER.
⍝
[EV1←2×⍳⌊.5×⍴X]
[OD1←EV1-1⋄EV2←¯1↓OD1]
[OD2←1↓OD1⋄DONE1←DONE2←0⋄GO2←0⋄GO1←1]

(GO1∧0=+/VEC[OD1]>VEC[EV1])->[DONE1<†1]
(GO2∧0=+/VEC[OD2]>VEC[EV2])÷>[DONE2<†1]
(GO1∧0≠+/VEC[OD1]>VEC[EV1])->[GO1<†0⋄BITSTR<†VEC[OD1]>VEC[EV1]⋄
                              PLACES<†VEC[OD1]×BITSTR⋄
                              PLACES<†(PLACES≠0)/PLACES⋄
                              TMP<†VEC[PLACES]⋄
                              VEC[PLACES]<†VEC[PLACES+1]⋄
                              VEC[PLACES+1]<†TMP⋄GO2<†1]
(GO2∧0≠+/VEC[OD2]>VEC[EV2])->[GO2<†0⋄BITSTR<†VEC[OD2]>VEC[EV2]⋄
                              PLACES<†VEC[OD2]×BITSTR⋄
                              PLACES<†(PLACES≠0)/PLACES⋄
                              TMP<†VEC[PLACES]⋄
                              VEC[PLACES]<†VEC[PLACES+1]⋄
                              VEC[PLACES+1]<†TMP⋄GO1<†1]
(DONE1∧DONE2)->[R←VEC]
∇
⍝
⍝   MAINLINE.
⍝
[X←⎕]
[⎕←SORT X]
∇∇
```

```
⍝
⍝  THIS EXAMPLE DEMOSTRATES HOW EASILY SOME OTHERWISE COMPLEX
⍝  PROBLEMS MAY BE SOLVED USING RULE BASED APL.
⍝
∇∇CONVPROG IS
∇R←X CONVOLUTION W;N
⍝
⍝ THIS ROUTINE SOLVES THE CONVOLUTION PROBLEM FOR THE GIVEN
⍝ INPUT DATA.
⍝
⍝    GIVEN: (W1,W2,...,WK) WEIGHTS AND
⍝           (X1,X2,...,XN) INPUTS,
⍝
⍝    CALCULATE: (Y1,Y2,...,Y(N+1-K)) OUTPUTS
⍝
```

⍝   WHERE: $Y_i = W_1 X_i + W_2 X_{i+1} + \ldots + W_k X_{i+k-1}$

```
⍝
⍝

R←W+.×⍉(0,-N-1)↓(¯1+⍳N)⌽((N←(⍴X)-(⍴W)-1),⍴X)⍴X
∇

⍝
⍝  MAINLINE
⍝
[ X←⎕ ]
[ W←⎕ ]
[ ⎕← X CONVOLUTION  W ]
∇∇
```

```
⍝
⍝ THIS EXAMPLE SIMPLY SHOW HOW A PROGRAM MAY CONTAIN TWO OR MORE
⍝ FUNCTIONS WHICH INTERACT WITH ONE ANOTHER. THE OBJECT OF THE
⍝ PROGRAM IS ABSURD.
⍝
∇∇ENHANCERPROG IS
∇R←COORDS ENHANCEPOINT MTX;VERMAJ;HORMAJ;X;Y;DONEV;DONEH
⍝
⍝ THIS FUNCTION ACCEPTS A MATRIX AND ENHANCES VERTICAL AND
⍝ HORIZONTAL LINES WITHIN THE THREE BY THREE BLOCK GIVEN BY
⍝ 'COORDS'. NO ATTEMPT IS MADE AT ERROR CHECKING. A MATRIX OF
⍝ THE CORRECT FORM IS ASSUMED.
⍝
[X←1↑COORDS◇Y←1↓COORDS◇DONEV←0◇DONEH←0]
[VERMAJ←+/MTX[X;¯1+ι3+Y-1]◇HORMAJ←+/MTX[¯1+ι3+X-1;Y]]

(VERMAJ>1)->[MTX[X;¯1+ι3+Y-1]<†1◇DONEV<†1]
(HORMAJ>1)+>[MTX[¨1+ι3+X-1;Y]<†1◇DONEH<†1]
(DONEV∧DONEH)->[R←MTX]
∇
∇R←ENHANCEMTX MTX;X;Y;XBOUND;YBOUND
⍝
⍝ THIS FUNCTION REPEATEDLEY INVOKES 'ENHANCEPOINT' TO ENHANCE
⍝ VERTICAL AND HORIZONTAL LINES WITHIN A TWO DIMENSIONAL MATRIX
⍝ OF APPROPRIATE DIMENSION. (THIS MATRIX SUPPOSEDLEY CONTAINS
⍝ A BITMAP IMAGE.)
⍝
[X←Y←1◇XBOUND←(ρMTX)[1]◇YBOUND←(ρMTX)[2]]

(X<XBOUND∧Y<YBOUND)->[(X,Y) ENHANCEPOINT MTX◇Y<†Y+1]
(X<XBOUND∧Y=YBOUND)->[X<†X+1◇Y<†1]
(X=XBOUND)->[R←MTX]
∇

⍝
⍝   MAINLINE
⍝
[MTX←⎕]
[⎕←ENHANCEMTX MTX]
∇∇
```

REFERENCES

[1] Ackerman, William, B. Data Flow Languages. "IEEE Computer" : vol. 15, no. 2.

[2] Ackerman, William B. & Dennis, Jack B. VAL -- A Value Oriented Algorithmic Language: Preliminary Reference Manual. MIT/LCS/TR-218.

[3] Aoki, Donald J. A Machine Language Instruction Set for a Data Flow Processor. MIT/LCS/TM-146.

[4] Arvind, V. Kathail & Thomas, Robert E. I-STRUCTURES: An Efficient Data Type for Functional Languages. MIT/LCS/TR-218.

[5] Arvind, V. Kathail A Dataflow Architecture with Tagged Tokens MIT/LCS 17 June 1980.

[6] Arvind, V. Kathail A Multiple Processor Dataflow Machine that supports Generalized Procedures. Proceedings of the Eighth Annual Symposium on Computer Architecture.

[7] Barr, Avron & Feigenbaum, Edward A. "The DENDRAL Programs", Handbook of Artificial Intelligence. Vol 2. William Kaufmann Inc. 1982.

[8] Barr, Avron & Feigenbaum, Edward A. "PROSPECTOR", Handbook of Artificial Intelligence. Vol 2. William Kaufmann Inc. 1982.

[9] Barr, Avron & Feigenbaum, Edward A. "MYCIN", Handbook of Artificial Intelligence. Vol 2. William Kaufmann Inc. 1982.

[10] Boral, Haran & DeWitt, David J. Applying Dataflow Techniques to Database Machines. "IEEE Computer" : vol. 15, no. 2.

[11] Buchanan, Bruce G. & Duda, Richard O. Principles of Rule-Based Expert Systems. Stanford University report no. STAN-CS-82-926.

[12] Burkowski, Forbes, J. Instruction Set Design Issues Relating to a Static Dataflow Computer. Proceedings of the Ninth Annual Symposium on Computer Architecture.

[13] Dennis, Jack, B. Data Flow Supercomputers. "IEEE Computer" : vol. 13, no.11.

[14] Dennis, Jack, B., etal. Building Blocks for Data Flow Prototypes. Proceedings of the Seventh Annual Symposium on Computer Architecture.

[15] Flynn, M.J. "Very High Speed Computing Systems", Proceedings of the IEEE, No. 54, December 1966. pp.1901-1909

[16] Leth, James W. An Intermediate Form for Data Flow Programs. MIT/LCS/TM-143.

[17] Liskov, Barbara, etal. CLU Reference Manual, Lecture Notes in Computer Science #114, Springer Verlag, 1981.

[18] McDermott, J. & Newell, A. & Moore, J. The Efficiency of Certain Production System Implementations. Pattern Directed Inference Systems, Academic Press, 1978.

[19] Patterson, D.A. & Sequin, C.H. "RISC I:A Reduced Instruction Set VLSI Computer" Proceedings of the Eighth Annual Symposium on Computer Architecture.

[20] Rosenschein, Stanley J. The Production System: Architecture and Abstraction, Pattern Directed Inference Systems, Academic Press, 1978.

[21] Rychener, Michael D. & Newell Allen An Instructable Production System: Basic Design Issues. Pattern Directed Inference Systems, Academic Press, 1978.

[22] Sowa, Masahiro & Murata, Tadao A Dataflow Computer Architecture with Program and Token Memories. "IEEE Transactions on Computers" : vol. C-31, no. 9.

[23] Srini, V.P. An Architecture for Extended Abstract Data Flow. Proceedings of the Eighth Annual Symposium on Computer Architecture.

[24] Watson, Ian, etal. A Practical Data Flow Computer. "IEEE Computer" : Vol. 15, no. 2.

[25] Wise, M.J. EPILOG=PROLOG+Dataflow : Arguments for Combining PROLOG with a Data Driven Mechanism. ACM Sigplan Notices, Vol. 17, Num. 12 December 1982.

[26] Zucker, Steven W. Production Systems with Feedback. Pattern Directed Inference Systems, Academic Press, 1978.