A novel microprocessor architecture using only dynamic shift registers as data storage elements

Robert A. Atkinson
University of Wollongong, uow@atkinson.edu.au

UNIVERSITY OF WOLLONGONG
COPYRIGHT WARNING

You may print or download ONE copy of this document for the purpose of your own research or study. The University does not authorise you to copy, communicate or otherwise make available electronically to any other person any copyright material contained on this site. You are reminded of the following:

This work is copyright. Apart from any use permitted under the Copyright Act 1968, no part of this work may be reproduced by any process, nor may any other exclusive right be exercised, without the permission of the author.

Copyright owners are entitled to take legal action against persons who infringe their copyright. A reproduction of material that is protected by copyright may be a copyright infringement. A court may impose penalties and award damages in relation to offences and infringements relating to copyright material. Higher penalties may apply, and higher damages may be awarded, for offences and infringements involving the conversion of material into digital or electronic form.

Unless otherwise indicated, the views expressed in this thesis are those of the author and do not necessarily represent the views of the University of Wollongong.

Recommended Citation
A novel microprocessor architecture using only
dynamic shift registers as data storage elements

A thesis submitted in fulfilment of the
requirements for the award of the degree of

Honours Master of Science
(Computing Science)

from

THE UNIVERSITY OF WOLLONGONG

by

Robert A. Atkinson, B.Math (Honours) W'gong

Department of Computing Science
1988
I hereby declare that I am the sole author of this thesis. I also declare that the material presented within is my own work, except where duly acknowledged, and that I am not aware of any similar work either prior to this thesis or currently being pursued.

R. Atkinson
ABSTRACT

This thesis describes the use of a ring of dynamic shift register stages to implement register storage in a general purpose microprocessor architectural paradigm. The feasibility of such an architecture is shown, and the performance and VLSI space requirements are discussed.

The cycling of machine registers through parallel shift registers can be used to distribute data between elements of the processor. This method replaces the traditional internal bus(es) and allows for concurrency of operations. A data identification scheme and control strategy is introduced to allow safe pipelining of instruction sequences that may be interdependent in respect to data produced.

The use of simulation to assess performance is discussed, and a brief performance evaluation presented. VLSI implementation costs are compared to those of a bus architecture, taking into account intrinsic overheads associated with controlling each architecture.
ACKNOWLEDGEMENT

This research grew out of one of many 'think-tank' sessions with my supervisor, Mr Gary J. Stafford, before this thesis was ever considered. His interest and academic stimulation, leading by example, are primarily responsible for my pursuit of novel approaches to traditional problems. For this, and his insightful comments on my tentative ideas, I am deeply grateful.

I must also thank the members of the Department of Computing Science for their unfailing interest and support, especially Mr John Fulcher, whose effort and care in vetting this document is most appreciated. Also I would like to thank the many people who have listened to my expositions on the potential of ring-based architectures and made helpful comments, not the least being Dr Graham Hellestrand of the Joint Microelectronics Research Group at the University of N.S.W., and also Mssrs Michael Shepanski and Peter Arnold.

And, finally, to Miss Traci Carse for reformatting the text after the word processor had reduced me to a seething frustrated computercidal wreck.
# TABLE OF CONTENTS

1. Introduction 1

2. A NOVEL ARCHITECTURE 3
   2.1 Data Storage Techniques and Architectural Paradigms 3
      2.1.1 Data storage in register files 3
      2.1.2 Pipelined Register Files 5
      2.1.3 An alternative: A Ring of Registers 5
   2.2 Implications of a Ring-Based Architecture 7
      2.2.1 Dynamic data storage in V.L.S.I. 7
      2.2.2 Controlling The Ring 7
      2.2.3 Advantages of Modularity 12
      2.2.4 Pipelining Instructions 13
      2.2.5 Microinstructions 15
      2.2.6 Performance Implications 15

3. AN IMPLEMENTATION OF A CONTROL STRATEGY 22
   3.1 Introduction 22
   3.2 One possible control structure to allow pipelining 23
   3.3 Performance considerations 27
   3.4 Development of a FMIFO design.
      3.4.1 Definition
         3.4.1.1 Logic definition 31
         3.4.1.2 Data definition 31
         3.4.1.3 Gross definition 32
         3.4.1.4 Definition of Individual Elements in the FMIFO 36
   3.5 Evolution of a nMOS Implementation 46
   3.6 Performance of the FMIFO Unit
      3.6.1 Scope of the Analysis 46
      3.6.2 A Note on Terminology 46
      3.6.3 The FMIFO Access/Modification Cycle 47
      3.6.4 Size vs. Speed 51

4. PERFORMANCE ANALYSIS 53
   4.1 Aims & Constraints 53
4.2 Comparison of Circuitry Performance Potential 55
4.3 Specification and Design Choices 57
  4.3.1 Circuitry speed 58
  4.3.2 Instruction Sets 59
  4.3.3 Implementation Choices 61
    4.3.3.1 FMIFO size 61
    4.3.3.2 Instruction Fetching Methods 61
4.4 Specification of an Experimental Processor 65
  4.4.1 The simulation regime. 67
4.5 Addition delays 68
4.6 Memory Speed 71
4.7 FMIFO Size 72
4.8 Number of Registers. 76
4.9 Instruction Order 78
4.10 Conclusions. 82

5 An Analysis of Data Storage Techniques in nMOS 84
  5.1 Introduction 84
  5.2 Data Storage Cells 84
    5.2.1 The pseudo-static memory cell 84
    5.2.2 The Dynamic Shift Register using Static Inverters. 88
    5.2.3 Clocked Dynamic Shift Registers 91
    5.2.4 Other shift register stages 96
  5.3 An alternative to synchronous master-slave storage cells. 96
  5.4 Topology Of The Ring 101
  5.5 A Coarse Estimate of Control Circuitry Overheads 101
    5.5.1 Fairness 101
    5.5.2 Select Circuitry for a Register File 104
    5.5.3 Register Identification Overhead For a Dynamic Ring 105
    5.5.4 Overhead for Asynchronous Ripple Clocking 105
    5.5.5 Comparative Area Requirements 105

6 FURTHER DEVELOPMENTS 109
  6.1 Context Switching 109
  6.2 Gallium Arsenide Implementation 111
  6.3 Multiprocessor Communication 112
  6.4 Further Research Direction 113

(vi)
1 Introduction

In recent years there has been a trend towards the use of large numbers of registers and a small set of simple instructions (RISC) in microprocessors. Such instruction sets tend to support register manipulations in favour of memory manipulations. Combined with the demand for longer word lengths it can be seen that the size of the register file is of great significance. This paper shall examine the conventional approach to register design, internal data storage and access. By abstracting from the accepted axioms for a bus-structured design this paper will identify an alternative and novel architecture that allows a reduction in the amount of VLSI area needed for data storage. It should be noted that the architecture developed is as applicable for complex instruction set computers (CISC) as for RISCs.

The new architecture uses a continuous cycle of data around a ring, each 'register' physically passing each active element (ALU, etc.) rather than a static data area with buses to distribute data. The problem of selecting a register and distributing data is converted to identifying the operations to be performed on the data currently accessible. A number of options for identifying registers (and in the more general case, data) are examined. One such solution is the use of tags specifying how each register is to be treated by the processing elements in the architecture. It can be seen how this method allows a high degree of instruction pipelining, and even parallel execution of non-interacting instructions.

The paper shall illustrate the feasibility of the concept by developing a viable, if not optimally efficient, control structure and show how it can be used to implement a sufficient set of typical instructions. This structure is a form of associative memory, organised as a queue, such that the first item that matches a template is selected. Using this structure, instructions can be broken into a set of tag specifications which are entered into the queue. Further development guarantees the correctness of parallel execution of instructions. The logical operation of the queue itself is formally specified, and used to derive a possible nMOS V.L.S.I. implementation.

Using the design developed above it will be possible to make a crude estimate of the performance of an architecture of the style being presented. Simulation results will be used to illustrate the advantages of further refinement of the control strategy. In particular it will be possible to identify simple methods of dividing up the work of the control unit to allow more parallelism at the expense of more complicated circuitry. The effects upon performance of different factors, such as the number of registers, the
configuration of the control unit and the order of instructions will be investigated through simulation of program execution.

A comparison of VLSI space demands for the bus and ring based alternatives and a rough analysis of timing considerations will be presented. The space required by a register file alone may be less than the space required by the data storage elements and shift control circuitry in the register ring, however the register ring incorporates the data transfer mechanism which is implemented by one or more bus structures. These data busses consume significant portions of VLSI real estate, and require considerable precharging times which may determine the ultimate microcycle time of the whole system. Along with the savings in chip space possible it will be demonstrated that for nMOS technology, at least, power consumption in the data storage elements can be reduced. The highly modular nature of the architecture presented will be suited to the use of automatic chip design techniques. Furthermore this paper will show how this approach potentially avoids some of the problems of testing dynamic logic at the VLSI level.

Although the majority of detail presented will use 5 μm nMOS technology as a referent, the use of other technologies will be discussed. In particular the potential savings in circuit complexity inherent in the presented architecture over traditional designs may enable fabrication in faster, but less dense technologies being developed. (e.g. GaAs)
2 A NOVEL ARCHITECTURE

2.1 Data Storage Techniques and Architectural Paradigms

2.1.1 Data storage in register files

A microprocessor is essentially a finite state machine that draws its flexibility from the fact that it can easily change its internal state by storing and retrieving data upon command. The amount of data might be quite large, consisting of several 'words' or collections of bits, and the number of possible commands (or 'instructions') may also be quite large. Storage of a word is done in a single operation, and the data is stored or retrieved from the storage area, or 'register'. A typical method of storing a 'bit' is the 'pseudo-static' memory cell, as shown in Figure 2.1. This consists of two inverters with feedback through a pass transistor. This transistor is turned on periodically by a clock to allow the charge on the input gate capacitance to be 'refreshed'.

![Pseudo-static memory cell](image)

**Figure 2.1**: Pseudo-static memory cell

Reading and writing from the circuit is performed when the refresh signal is not active. This is achieved by using a signal derived from a two-phase clock. The cell is analogous to a single bit shift register with its output connected to its input. Registers are constructed by having a whole row of such cells all accessed by the same control signals. Often a number of such registers are organised in a block, and accessed by some selection circuitry. This block is commonly referred to as a 'register file'. The machine
instructions indicate which register they would like to use as sources and destinations of data. Because most internal calculations require two operands it is conventional for two buses (at least) to be provided to enable both sets of data to move simultaneously. Typically the register cell will have read facilities onto A and B buses and write facility from just one. The extra space these bus wires and read/write circuitry takes up is considerable, as will be seen in the nMOS implementation described later. It is possible to use only a single bus line, which reduces space somewhat. The problem is that many instructions take two operands and combine them in some way to produce a result. The two operands would need to be read in separate operations. In most cases a two (or more) bus architecture is used, such as the one presented in Figure 2.2.

In order to select each register as an operand two sets of selection circuitry will be necessary. For a machine with N registers this usually means distributing 2N lines of microcode or 2\log_2N lines of binary coded register number. This then takes a significant amount of chip space to decode, as we will see later when we analyse space requirements for different data storage schemes.

Figure 2.2 : A typical two-bus microprocessor architecture.
2.1.2 Pipelined Register Files

One storage structure that has been presented as an alternative to the random access two-bus register file is the 'pipelined register file' [Sussman 1986]. This organises storage in the form of a queue, with writes occurring only at the tail, but with read-access being allowed for all elements. The individual storage cells do not have refresh capabilities, thus data will only last for a short period of time. The concept of a named register to store a value is no longer appropriate, and instead the code uses the pipelined register file for the temporary storage of operands and results as they are loaded from memory and produced by operations. Data is refreshed by a write operation, since a write operation shifts all the elements through the file, overflowing the last element. As a result of this the life expectancy of data also depends on the time since the last write, and the total number of writes since the data was first written.

The advantage is that each 'bit' stored in the pipelined register file needs only read circuitry. The refresh capability (a transistor switch in a feedback loop) is replaced by a shift capability (involving a pass transistor between each inverter. Like a conventional register file, a pipelined register file can be implemented with only one bus to save space, but once again no operations can take place in such an architecture until all operands have been accessed.

Any architecture using pipelined register files places important constraints upon program code. The code must allow for the limited life expectancy of data, and try to maximise life by minimising writes into the queue. The complexity of optimising such code shunts the responsibility onto a compiler as the only feasible way of keeping track of data life and usage. [Sussman,86]. The use of compilers to optimise code when architectural constraints apply will be introduced at a later stage, after the peculiar features of another alternative architecture have been presented.

2.1.3 An alternative: A Ring of Registers

Abstracting a level from the 1-bit shift register concept of the register cell, it can be seen that we can store the data for N (1-bit) registers in a N-bit shift register. A register file of N M-bit registers can be replaced with M N-bit shift registers all clocked in parallel (Figure 2.3).
It is possible to access any piece of data by monitoring a single (physical) row of cells. Thus the need for buses to distribute data is avoided. The cost that must be paid is of course in the latency time before data can be accessed. In addition, the problem of register selection is replaced by one of register identification. Instead of a traditional design in which the data are shuffled between a register file and the subcircuits the registers are continually passing the circuits waiting to be used. The architectural components that use the data (ALU, shifter, External bus interface etc., henceforth 'Active Elements': AEs) do not actually need to recognise which register is passing, as long as they are told what to do to it.

Figure 2.3: An architecture using parallel shift registers
2.2 Implications of a Ring-Based Architecture

2.2.1 Dynamic data storage in V.L.S.I.

Because the data is constantly moving around the ring the VLSI designer is free to use dynamic elements that only have a limited data retention time. These are invariably smaller than the corresponding static elements, and usually also dissipate less power. A detailed discussion of the design of shift register elements that may be used will be presented later, in Chapter 5. It is sufficient to predict that they will be smaller and more power efficient than the static memory cells that are often used in register files. Unlike the pipelined register file the data does not have a limited life expectancy, so the constraints implicit in that case will not apply here.

2.2.2 Controlling The Ring:

The 'control unit' (henceforth abbreviated to 'CU') has the task of telling the AEs what operations to perform on which register. A multiplicity of possibilities arises, depending upon whether the control unit communicates to the AEs via signals and global wiring, or by placing information in additional bits in each register. For any register operation there are two parameters which must be specified, register number and the operation. Registers can be identified by either 'name' (i.e. multi-bit data) or by a timing signal ("The register is the one which you have right now"). We can characterise a number of alternative schemes by imagining the communication between the AE and both the register (via the extra data) and the control unit:

1) Register : "I am Register #3"
   CU : "R #3 is operand A"

2) R : "I am operand A"
   CU :
3) R :
   CU: "The register there now is operand A"

4) R: "I am for you"
   AE: "I expect an operand A"

5) R:
   AE: "This is (by my calculation) R3, and the C/U told me in the
   past that R3 is operand A"

6) R: "I am for you"
   CU: "This next thing for you is operand A"

While all of these schemes could be made to work they demand differing amounts of intelligence
in the AE and the Control Unit (CU), and also in the amount of global wiring and extra data to be
'tagged' to the register.

Schemes that use register numbering to identify data (1 & 5) require either the AE to keep track
of which register is 'visible' (currently available), or the registers to carry this information in a tag field
(Fig 2.4). For N registers this means \( \log_2 N \) bits of extra data per register. Both methods also require
somewhat involved initialisation and communication between the CU and the AE to specify which
register is to be used and also how it is to be used. It would not appear to be helpful to use register
numbering as identification.
The next choice to be made is between the register identifying itself as belonging to the AE, or the AE being told by the CU. This is a tradeoff between a more complicated CU with extra wiring (Fig 2.5), and extra data with circuitry to decode it in the AE (Fig 2.6). If there are $N$ active elements, then there will be a need for at least $\log_2 N$ bits of 'tag'. (Typically $N$ would be only three or four - hence two bits).
Fig 2.5: Having the Control Unit tell the A.E.s what to do

Fig 2.6: Using a tag field to identify data to A.E.
Using the CU to synchronise data access, on the other hand, involves rather more complexity in the CU, as well as introducing inter-module wiring, thus compromising the modularity of the system, both physically and with respect to timing (discussed in section 2.2.3).

As well as finding a register the AE needs to know what to do with it. This leads to a choice between schemes 2 and 4. In scheme 4 the information about what to do with each register is implicit in the order they arrive. The advantage is that no more information need be passed in the tag field, however there are quite serious drawbacks. To manipulate the order in which registers arrive, the CU must be able to let registers pass on the first cycle without addressing them for the AE. This introduces further latency into the data access. Consider the following case:

An adder expects A then B then C, and performs A+B -> C.

A,B,C are registers 1,0 and 2 respectively and R0 is currently passing the CU.

The CU must wait for R1 and then a whole cycle for R0 to pass before the adder can start adding. This sort of latency would be expected to slow around half such instructions significantly. In addition, the case where two operands are the same register will cause a cycle to be lost. A further drawback is the lack of flexibility in specifying which of a set of possible operations an AE is to perform.

In the experimental simulated implementation is has been found sufficient to use 8 extra bits of data as a tag field for an ALU operand:

2 for identifying which AE ( CU,ALU,EBI )

2 for identifying which operand

00 : result , 01 : A , 10 : B , 11 : A & B

1 for specifying whether the register is to be complemented.

1 for the past carry value ( passed around to aid modularity )

2 for identifying the operation ( Add, OR , AND )
The amount of information available can be increased if two operands carry different data in their
tag fields.

N.B. This is for an arbitrary instruction set, and no particular importance should be given to
the exact size of the tag field. Many instruction sets or sets of active elements could manage with fewer
bits (for example the 'complement' bit might be unnecessary, and if separate units do Addition, AND,
OR etc then the last field will be smaller, but the first potentially larger).

The scheme adopted for controlling data access and manipulation can be summarised as :

"The control unit decodes instructions and places tags on registers to tell
active elements how to treat them. The active elements reset the tag upon
reception of the data so that the control unit can use the register for a
subsequent operation."

2.2.3 Advantages of Modularity

Signal interaction between active elements places constraints on the timing of the cycling
mechanism. Either the cycle must be slowed down so that all signals are guaranteed to be available, or
the designer must venture into asynchronous design on a chip-wide scale. By maintaining complete
modularity we can let any or all of the AE's function asynchronously or synchronously (internally). The
same freedom applies to the register cycling mechanism we choose to use. Each AE need be synchronised
only with the register-tag pair it is monitoring. Yet another advantage of modularity will be the ease of
'pipelining' instructions to minimise the cost of latency. If the CU needs to monitor register positions
and handle timing for two (or more) instructions simultaneously it will greatly increase complexity. On
the other hand, if a control strategy can be developed that allows the C.U. to dispatch 'tasks' to the A.E.s
as they become available (i.e. instructions are decoded) then we can can improve performance without
increasing timing and wiring complexity.
Modularity also greatly enhances ease of design and testing in VLSI implementations. It makes it relatively easy to formally specify what task a module performs, and how it interacts with other modules. This might lead to automatic layout generation from a manageable architectural specification, one of the current pushes in VLSI research (e.g. Ayres)

2.2.4 Pipelining Instructions

Having decided to control the use of data by associating 'tags' to registers it can be readily seen that operations on registers (load or store from AE to register) may take place simultaneously in different parts of the register ring. As long as any critical ordering of operations is maintained there can be any number of operations pending (i.e. the tag field already set and cycling round the ring). The operations need not be associated with the same instruction.

For example, consider the following set of instructions :

Add R1 to R2
Shift R3 right
store R4

These instructions have no effect upon each other, and can therefore be performed in any order, and can be decoded and the registers involved tagged without waiting for the previous instruction to complete. In general memory accesses will take the longest, and should be started first. In a processor with units that perform complex operations (e.g. floating point manipulations, trigonometrical operations, FFT's, etc) the savings that can be made by pipelining are considerable. It can be seen that the efficacy of pipelining is related to the relative speeds of internal operation and instruction fetching. This is a strong argument for the use of an internal instruction cache to maximise the apparent availability of the external bus for execution-related operations.
In a set of instructions such as:

Add R1 to R2
store R2 somewhere
Add R3 to R2

It is essential that the instructions get executed in the right order. There can still be some pipelining (instruction-fetch and decode, the second addition can take place while the memory is still being accessed, etc).

The salient points to be considered are:

1) efficient instruction fetching is necessary for a sufficient number of operations-to-be-done to be identified in time to allow significant overlap in instruction execution.

2) The order of instructions can be of great importance; in any given group of instructions that have no interaction with each other's operands or results the slowest instructions should be started first. This will lead to interesting compilation strategies for such an architecture.

3) The slowest operations in general-purpose microprocessors are likely to involve memory access. If instructions can be fetched out of an internal cache (sufficiently large to hold all the code for most tight loops - for example, moving blocks of memory around) then a real speed up is possible. An alternative approach is to use a separate instruction and data bus, though this leads to an increase in circuit complexity and the number of external connections that must be allowed for.

A control structure is required that will guarantee that the result of a previous instruction is obtained before it is tagged to be used again. With this condition satisfied there is no restriction on the number of instructions that may be pipelined.
2.2.5 Microinstructions

The use of a tagging scheme for identifying which operation to perform with the data in a register can be considered as a form of microcoding. Consider a simple instruction:

Add R1 to R2

this can be broken down into

Identify R1 as operand A for the ALU add function
Identify R2 as operand B
Identify R2 as the ALU result register.
(and, implicitly, : Fetch the next instruction).

Performing a macroinstruction is thus a matter of performing a set of trivial microinstructions (that do no more than set tag fields) in the correct order. Instruction pipelining can be achieved by queueing up a whole set of microinstructions from different macroinstructions. When a register is visible in the CU's window, the first microinstruction in the queue that operates on that register must be found.

Before progressing further, it should be noted that it is possible to control the whole operation of the architecture by these 'microinstructions' alone. Since the microinstructions are of short word length (for an N register machine: \( \log_2 N \) bits for register identification plus 8 or so for an operation specifier) this approach is quite cheap in terms of external connections (assuming that the ring is to be controlled from off-chip). The microinstructions must be generated by some external circuitry, and the overheads in off-chip communication times would seem to make it unattractive for a general-purpose microprocessor.

The simplicity of the microinstructions make it relatively easy to express higher-level instructions formally. Decoding of instructions becomes a matter of generating a sequence of microinstructions. These may be easily stored in a ROM and simply accessed by the instruction identification circuitry. This is the approach to be used in the experimental machine.

2.2.6 Performance Implications

This chapter has shown how a microprocessor architecture might be constructed around a ring of
registers rather than a traditional register file - bus structure. A number of potential advantages: modularity, size and inherent parallelism have been identified. The ultimate question is whether these advantages can be exploited sufficiently to enable such a processor to compete favourably with traditional designs.

The potential performance advantages of a ring-based architecture are based on the two largely unrelated factors:

1) Local control signals and absence of a global bus allows internal machine cycles to be significantly shorter than those for a bus-based design.

2) Programs exhibit potential for instruction-level pipelining.

These two advantages will be cumulative, unless the design proves to be so fast due to the first factor that it is impossible to feed instructions to it fast enough to gain any advantage from instruction level parallelism.

The example processor developed in the following chapters is intended to show that a processor can indeed be implemented without compromising the potential advantages of pipelining and localised control signals, and hence that the first factor is realisable. The design presented, being a simplified example, shows promising performance characteristics, in spite of the inexperience of the author who had to rely on a "cookbook" approach to the design. It is to be assumed that an experience chip design team could create a significantly more efficient implementation.

The degree to which programs exhibit the potential for parallel execution of instructions is not readily discernable without at least a series of stochastic investigation of the following influences:

- Prevalence of potential parallelism in HLL (high level language) constructs.
- Potential for pipelining in compiled code for a range of instruction sets.
- The ability of compilers to improve the pipelineability of compiled code
- Design or determination of an instruction set that facilitates pipelining, (RISC and/or CISC).
- Determination of the type and number of "Active Elements" (adders etc) that may be efficiently used.

Significant work has been done on instruction sets, instruction mixes and branch control strategies to maximise pre-fetch pipelining efficiency (for example Fairclough 82, Lunde 74, Gibson 70,
Yannacopoulos 77). Unfortunately however, this author has been unable to locate any work that addresses the specific issues that affect the pipelining potential of the ring-based architecture presented here. A detailed study of these issues is beyond the scope of this paper, but the issue of compilation strategies, in particular order or instructions, has been dealt with in sections 4.7, 4.8 and 4.9 below.

Pipelining is possible in at least three ways in a ring-based architecture:

1. Instruction fetch/decode and execution phase can be pipelined.
2. Instructions which use independent operands and operations can proceed simultaneously.
3. Instructions that use the same operation and independent operands can be allocated to multiple Active Elements.

Pipelining the fetch/decode and execution cycles may be implemented by conventional architectures, however the single bus structure makes it very difficult to implement the other two types of pipelining. The second type falls quite naturally out of a ring-based architecture, and the example machine shows that not only is it possible, but simple programs do in fact exhibit some pipelining, in spite of an instruction set that is not designed with pipelining instructions in mind. As Reduced Instruction Sets seem to be the most efficient method of using a machine that must finish a task before proceeding to the next, complex instructions sets, or at least complex operators in instructions, may be the best way of using the parallelism available ring-based processors. For an example floating point operations and mathematical functions are of paramount importance in many computationally intensive tasks, such as robotics and graphics transformations. Units to handle these functions could be readily included into a ring-based machine to create specially optimised processors.

The use of multiple Active Elements of the same type is more likely to fulfill the needs of specialised applications than general-purpose microprocessors. A stochastic analysis of code compiled for a chosen instruction set may show that it is worthwhile to have multiple ALU’s, or to separate the ALU functions into separate AND, OR, XOR, comparison, add, shift, etc. capabilities.

It remains to show, qualitatively only, that some well known HLL constructs give rise to code which allow pipelining to be used. The HLL used is "C"-like and is translated into a parse tree to show which operations must take place, and finally into a description of the order in which operations may take place if pipelining is possible.
Operations on vector elements:

\[ a[i] = b[j] + c[k]; \]

will produce a set of operations:

\[
\text{store} \\
+ \\
/ \\
+ \\
/ \ \\
+ \\
/ \\
\text{a i fetch fetch} \\
/ \\
\text{b j c k} \\
\]

which might be executed in the following manner:

result register:

- R1: \((b+j) \text{ (fetch)} \)
- R2: \((c+k) \text{ (fetch)} (R2 + R1) \text{ (STORE at R3)}\)
- R3: \((a+i)\)

which results in 5 internal cycles to perform 7 instructions, a saving of at least two execution cycles over a purely sequential execution.
As the expressions become more complicated the scope for simultaneous execution becomes greater.

Take for example a 1 x 3 vector dot product: familiar enough in spatial coordinate calculations:
\[ \text{res} = a[0] \times b[0] + a[1] \times b[1] + a[2] \times b[2]; \]
which might result in the expression tree (after optimising out zero indices):

```
+  
|   |  
|   |  
|   |  
|   |  
|   |  

\[ + \]
\[
\times \]
\[
\times \]
\[
\times \]
\[
\times \]
\[
\times \]

\[ 
\times \]
\[
\times \]
\[
\times \]
\[
\times \]
\[
\times \]
\[
\times \]
\[
\times \]
\[
\times \]
```

which might be executed in the following fashion (albeit somewhat profligate of registers):

<table>
<thead>
<tr>
<th>reg:</th>
<th>initial value:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>a</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R2</td>
<td>b</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R3</td>
<td>(a+1)</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R4</td>
<td>(b+1)</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R5</td>
<td>(a+2)</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R6</td>
<td>(b+2)</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This example shows 15 operations that may be evaluated in 8 internal calculation cycles without allowing any two operations of the same type to occur in the same cycle.
The code above uses absolute values for the vector indices, whereas it is obviously possible to increment the array pointers instead of indexing. In the examples below the ++ operator indicates the increment operator and the @ sign indicates an indirect load using a register as an address.

\[
\begin{array}{cccccccc}
\text{reg:} & \text{initial value:} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
R1 & a & - & ++ & ++ & @R1 & *R2 & +R3 \\
R2 & b & - & - & ++ & ++ & @R2 \\
R3 & a & @R1 & - & *R4 & - & - & +R4 \\
R4 & b & @R2 & - & @R2 & *R5 \\
R5 & a & @R1 & - \\
\end{array}
\]

Once again 8 cycles are required. The same number of cycles are required if the register addressing function allows automatic incrementing of the pointer, although this capability reduces the number of sequential cycles required from 15 to 11 (saving the four increment instructions).

For 3 dimensional vector cross products we find three expressions of the form:
\[
\]

Analysing this in the same fashion we find 4 unique index operations (4 addition + 4 access), three multiplication operations and two addition operations (13 operations in total). Since three equations using the same base addresses exist it is reasonable to allow the addresses of a[i] and b[i] to be precalculated (by an increment operator). The pipelined execution might be:

\[
\begin{array}{cccccccc}
\text{reg:} & \text{initial value:} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
R1 & b+i & - & @R1 & - & - & - & + \\
R2 & c & - & ++ & - & ++ & - \\
R3 & @R2 & - & *R1 & - & +R4 & +R4 & store @R5 \\
R4 & @R2 & *R1 & @R2 & *R1 \\
R5 & a+i & - \\
\end{array}
\]
It would certainly appear from these simple samples that it is possible that advantage can be taken of instruction-level parallelism, even with short, quick instructions. The savings will be most marked when one instruction takes significantly longer than normal (mathematical functions, floating point operations, memory accesses, etc) and subsequent instructions do not require the result of the operation immediately. Coordinate transformations are one important application for which such a processor will have a potentially large advantage.
3. AN IMPLEMENTATION OF A CONTROL STRATEGY

3.1 Introduction

This chapter pursues the idea of executing instructions by attaching 'tags' to registers identifying which operation is to be performed upon them. The tagged registers then circulate, eventually being processed by the so-called Active Elements (or A.E.s) of the micro-architecture, an element such as the Arithmetic and Logic Unit (ALU) being an example.

Using this technique as a model, a strategy is developed for controlling the execution of instructions so that as much parallelism as possible can occur. The constraints imposed by instruction interaction are investigated, and a method of dealing with these problems, while still allowing a high degree of pipelining, is proposed. This method is based on the use of a piece of associative memory with "first-in,first-out" (FIFO) capabilities to implement a queue of items such that the first item that matches a template (in this case a register number) is retrieved. For convenience sake this structure has been described as a FMIFO (First Match In - First Out).

The implementation of this hardware solution is described by developing a logical definition of the FMIFO's interaction with the external control circuitry. The practical realities of insertion, deletion and overflow conditions are introduced as their need becomes apparent. From this overview it is possible to derive the logical operation of the individual elements in the FMIFO, so that the whole unit is made of identical sub-units. This is of course an important consideration with V.L.S.I. design.

Using the derived specification it is relatively straightforward to design a circuit with confidence in the correctness of its operation. The circuits presented use nMOS technology, although the translation to mask layout is not supplied, this detail not being essential in the investigation of a control strategy.
3.2 One possible control structure to allow pipelining

A sequence of register-tag pairs is generated by the instruction decoding. These may belong to more than one instruction. The CU has to recognise the register visible, and then find a tag in the input sequence. There are two restrictions on the selection of a tag: for a given register the tags must be set in the order they arrive in the input sequence, and also the type of operation to be performed on the register must be the first of its type. A simple way of satisfying these criteria is to allow only the first element of the input sequence to be accessed (the input sequence is a queue). This approach will severely curtail the efficiency of pipelining however, unless it is possible to order the input queue somehow. This seems to be a difficult task to perform at the speed required. It is therefore desirable to be able to access the data in the queue by context. The contexts to be considered are:

1) register
2) operation

and 3) temporal ordering. (FIFO).

The item to be accessed is the first which is associated with the current register, unless the operation type appears before it in connection with another register. To illustrate the workings of this "First Match in First Out" (FMIFO) structure consider it to have just four entries in it:

<table>
<thead>
<tr>
<th>Registers</th>
<th>Operation type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
</tr>
</tbody>
</table>
If the current register was R1 then the first entry would be activated and all subsequent entries suppressed (matching takes place simultaneously, but there is arbitration logic to select the first entry). The operation tag 'A' would be written to the register tag field because there is no preceding operation 'A' in the queue. The first entry could then be removed (the queue is constructed as a set of shift registers with logic to control which bit is accessible). If, on the other hand, R3 was the current register, then the structure would identify operation 'A' but upon searching find that the first entry has priority, and thus no action is taken.

<table>
<thead>
<tr>
<th>Registers</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
</tr>
</tbody>
</table>
| Match     | 2         | B         | Match 'A'
| 3 **selected** | A |
| 4         | B         |

Such a control structure allows safe parallelism to a much higher degree than a simple FIFO.

For example consider the following instructions and generated operations:

Add R1 to R2
1) R1 - ALU operand A
2) R2 - ALU operand B
3) R2 - ALU result

store R1 at address in R3
4) R0 - EBI address
5) R1 - EBI data to write

Add R0 to R2
6) R0 - ALU operand A
7) R2 - ALU operand B
8) R2 - ALU result
Assume also that the current state of the machine is:

<table>
<thead>
<tr>
<th>Registers (first is visible to CU)</th>
<th>Operations (as numbered above)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[R0] R1 R2</td>
<td>1 2 3 4 5 6 7 8</td>
</tr>
</tbody>
</table>

The state of the machine at each step would be as follows:

<table>
<thead>
<tr>
<th>FMIFO operation</th>
<th>FIFO operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 R0 R1 R2</td>
<td>R0 R1 R2</td>
</tr>
<tr>
<td>1 2 3 4 5 6 7 8</td>
<td>1 2 3 4 5 6 7 8</td>
</tr>
<tr>
<td>2 R1 R2 R0</td>
<td>R1 R2 R0</td>
</tr>
<tr>
<td>1 2 3 5 6 7 8</td>
<td>1 2 3 4 5 6 7 8</td>
</tr>
<tr>
<td>3 R2 R0 R1</td>
<td>R2 R0 R1</td>
</tr>
<tr>
<td>2 3 5 6 7 8</td>
<td>2 3 4 5 6 7 8</td>
</tr>
<tr>
<td>4 R0 R1 R2</td>
<td>R0 R1 R2</td>
</tr>
<tr>
<td>3 5 6 7 8</td>
<td>3 4 5 6 7 8</td>
</tr>
<tr>
<td>5 R1 R2 R0</td>
<td>R1 R2 R0</td>
</tr>
<tr>
<td>3 5 7 8</td>
<td>3 4 5 6 7 8</td>
</tr>
<tr>
<td>6 R2 R0 R1</td>
<td>R2 R0 R1</td>
</tr>
<tr>
<td>3 7 8</td>
<td>3 4 5 6 7 8</td>
</tr>
</tbody>
</table>

(Bold, underlined print indicates a 'match')
It can be seen that not being able to extract items from within the queue is a major disadvantage (this example is favourable to FIFO as it stands - if the cycle started with R2 the FMIFO would be even further ahead).

In the above example we assumed that once the data had been 'dispatched' there was no more responsibility on the part of the CU to maintain synchronisation of different parts of the instructions, but this is not so in practice. If an AE takes a long time to compute its result (a slow memory fetch for example) then the register designated as the result register will remain in circulation, and may pass the CU again before it has been used. The CU must not redesignate to be used for some other purpose. This is easily satisfied by treating the CU like the other elements: it only recognises data that is tagged for it. When an AE is finished with a register (either loaded or stored a value) it resets the tag so that it now 'belongs' to the CU.

The other synchronisation problem that may occur is in the following situation:

<table>
<thead>
<tr>
<th>FMIFO#</th>
<th>Register</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>ALU Operand A</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>ALU Operand B</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>ALU Result</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>ALU Operand A</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>ALU Operand B</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>ALU Result</td>
</tr>
</tbody>
</table>

If the ALU takes a long time to evaluate its result, then register 3 may still be in circulation as the ALU result when R4 passes the CU. If the CU has removed the queue entry for #3 then there is nothing to stop R4 being flagged as an ALU result. The ring now has two of these, and the ALU will put the result of its first addition into whichever comes first, after it completes. If it is possible to guarantee the AE's will never be so slow then there is no problem, otherwise a modification of the FMIFO operation is called for, namely the provision of a single bit for each register-tag pair to signify whether it has been
dispatched. When the register is visible again (i.e. the AE has reset the tag to acknowledge receipt and the
register passes the CU again), the entry is removed from the queue. Until then its operation tag stops
other operations of the same type being dispatched. The presence of a dispatched item (currently being
removed) should not stop the next operation on the register from being recognised.

3.3 Performance considerations

Since the FMIFO structure needs to be accessed each time a register passes through the control
unit it can be seen that the speed of operation of the unit may easily be the limiting factor in the rate of
data transfer around the ring. It is thus imperative to make this unit as fast as possible. One method
available is to split up the two searches so that the next register number match is being performed while
the last result is being checked against the operation list. This is facilitated if the CU keeps a counter that
cycles through register numbers as the register ring circulates. The 'read point' can be a few steps ahead
of the 'write point' in order to allow the matching process to take place before the register is available for
modification (see Fig 3.1). The pipelining of FMIFO operations may then be used to make the total
FMIFO delay that of the slowest sub-operation.
One more point to be made is that since there are so few interconnections, synchronisation problems are virtually non-existent, and it is very easy to use an asynchronous approach to cycling the ring. When the FMIFO is nearly empty it will take less time to perform matching operations (due to a reduction of circuitry to propagate through) than when it is full. The larger the FMIFO the more marked will be this difference. In a synchronous circuit the worst case must be allowed for, so the larger the FMIFO (desirable to maximise pipelining of macroinstructions) the slower it will be. This will not be a problem if the FMIFO causes the ring to cycle when it has finished its operation. In the case when the register does not 'belong' to the CU the next cycle can occur as soon as possible.

Fig 3.1 : Pipelining FMIFO operation
The time taken for the active elements to perform their operations on a register will depend upon whether any of the operations are to be performed immediately upon the operand, or if a result is generated and held until a result register arrives. If all AE's only read or write data to the ring on each cycle, then a minimum cycle time will be the time taken for the following sequence of steps:

- Read tag field
- Compare with 'site I.D.'
- Identify operation
- Read/Write data
- Cycle ring

This time can be calculated and allowed for by the cycling mechanism. Thus a cycle will occur when the minimum time period has elapsed and the FMIFO has signalled that it is ready. As will be seen, once the design of the FMIFO has been specified and analysed, the time taken for the FMIFO operation will be critical to the speed of the processor. The operations that must be performed each cycle should be made to occur simultaneously for each entry in the queue. This will be the main consideration in the evolution of the design which follows.
3.4 Development of a FMIFO design.

In this section the logical design of the above control structure shall be described. The first step will be to define the action of the FMIFO as a unit, and then derive a definition of the individual elements. It is to be kept in mind that good V.L.S.I. design involves the repetition of standard cells, therefore each entry in the FMIFO should be stored in an identical element.

3.4.1 Definition

The FMIFO interacts with external circuitry on the basis of the data stored within it. There are thus two important classes of actions to be defined: data access and data modification. Data modification depends upon the past history of data access, in that modification includes deletion of items once they have been used. Data access is also the most common operation, as it will occur almost every machine cycle. As a result the efficiency of data access will take priority in the design, and this will be reflected in the assumptions made in the logical description. In particular there are a number of different possible mappings of logical data onto physical circuitry in a dynamic queue. The chosen method is for the head of the queue to always reside at the first physical location, and subsequent entries in successive physical elements. To achieve this it is necessary to physically move data around, but it has the advantage that it is easy to find the data in the queue at any given time. In this way priority is given to speed of retrieval.
3.4.1.1 Logic definition:

Actions will be defined using a predicate logic. The elements of this logic are:

- Syntax: \&, +, \sim, True, False, : (Ai : predicate)
- Explanation:
  - \&: Logical 'AND' operator
  - +: Logical 'OR' operator
  - \sim: Logical 'NOT' operator
  - True, False: Logical constants
  - : 'such that'
  - (Ai : predicate): For all integers 'i' such that 'predicate' is true
  - =: 'is defined as'
  - <-: is set to the value of
  - <, >, =: Relational operators

3.4.1.2 Data definition:

The operands in the predicates correspond to the data and structure of the FMIFO unit. They are defined below:
Name | Explanation
--- | ---
R | the current register (to be checked)
MAX | the number of items in the FMIFO
MAXSIZE | the maximum possible value of MAX
reg[k] | the kth register number in the queue
tag[k] | the tag associated with reg[k]
critical(tag[k]) | the bits of a tag field which define the unique destination of a register. (e.g. ALU operand A)
reg[1] | the first element in the queue
full | Whether the FMIFO is full.
tag_out | The tag field to be (possibly) attached to the current register
tag_valid | An indicator of whether the tag is to used or not

### 3.4.1.3 Gross definition:

The first definition describes the action of finding the tag associated with the first matching register number. The tag is written to a bus so that it appears on 'tag_out' and all the other elements can access it in order to check for clashes.

Defn. 1) \( \text{tag\_out} = \text{tag}[k] : ( k < \text{MAX} ) \& ( \text{reg}[k] = R ) \)

\& \{ A \ i : ( 0 < i < k ) \& ~( \text{reg}[i] = R ) \} 

The validity of the tag depends upon whether there is a tag before it in the queue which has an exclusive demand on the same resources that the selected tag requires. For example, two registers claiming to be ALU operand-A cannot be assigned to the ring simultaneously. The function 'critical(tag)' identifies the part of a tag which is to be compared. The function will be implemented in the hardware, and for simplicity sake should be a constant length bit field. In the experimental implementation 4 of the 8 bits of the tag field were compared.
It can be seen that the terms which define 'k' in Defn. 1 and Defn. 2 are common. They could be rewritten as:

Defn. 3) \( \text{Select[k]} \equiv (k < \text{MAX}) \land (\text{reg[k]} = R) \)
\[\land \{ A \ i : \ (0 < i < k) \land \neg(\text{reg[i]} = R) \} \]
\[\land \{ A \ i : \ (0 < i < k) \land \neg(\text{critical( tag[k] ) = critical( tag[i] )}) \} \]

Defn. 4) \( \text{tag_out} \equiv \text{tag[k]} : \text{Select[k]} \)

Defn. 5) \( \text{tag_valid} \equiv \text{Select[k]} \)
\[\land \{ A \ i : \ (0 < i < k) \land \neg(\text{critical( tag[k] ) = critical( tag[i] )}) \} \]

The definitions pertaining to the modification of the contents of the queue rely on the position of data within the structure. In order to facilitate knowledge of this it is possible to define an extra data field that has values defined by the predicate:

Predicate 1)
\[ (A \ i : \ ((0 < i <= \text{MAX}) \land (\text{valid[i]} = \text{True})) + (\text{MAX} < i <= \text{MAXSIZE}) \]
\[\land (\text{valid[i]} = \text{False})) \]}

In other words, all entries in the FMIFO has the associated value of 'valid' set to a logical 'True', and all the valid entries are stored contiguously in the first MAX elements of the queue. In the physical implementation, MAX will be defined by the data stored in the 'valid' entries:
Predicate 2) \( \text{valid}[i] = ( \ i < \text{MAX} ) \)

Using the 'valid' data set, we can define 'full' very simply by

\[ \text{Defn. 6) } \text{full} \equiv \text{valid}[\text{MAXSIZE}] \]

The action of writing a tag, \( T \) associated with a register number \( N \) into the FMIFO is defined as:

\[ \text{Defn. 7) } \text{write}(T,N) \equiv \neg \text{full} \]

\[ & \tag{\text{succ}(\text{MAX})} \leftarrow T \]

\[ & \reg{\text{succ}(\text{MAX})} \leftarrow N \]

\[ & \text{valid}[\text{succ}(\text{MAX})] \leftarrow \text{True} \quad (\text{therefore } \text{MAX} \leftarrow \text{succ}(\text{MAX})) \]

(Note the use of the function \( \text{succ}() \), the successor function from logic, because it more accurately reflects hardware reality than an addition operation)

Using predicate 2 we can rewrite this as:

\[ \text{Defn 7a.) } \text{write}(T,N) \equiv \neg \text{full} \& \neg \text{valid}[k] \& \text{valid}[i] \& (k = \text{succ}(i)) \]

\[ & (\reg[k] \leftarrow N) \& (\tag[k] \leftarrow T) \& (\text{valid}[k] \leftarrow \text{True}) \]

The most complicated of all the actions is that of deletion of an item. This is dependent upon the past history of the data, so once again an extra data storage element is used. This storage element will record the fact that at some (arbitrary) time in the past the associated item was selected and written to a register as a tag field. It was shown above that it is unwise to remove an item from the FMIFO until the tag has been used by the Active Element. The implication is that when the register is recognised again
the item may be deleted. For the sake of efficiency the existence of an item to be deleted should not stop another item from being selected. Thus it is important to modify Definitions 1 & 2.

The modified definitions associated with deletion are:

Defn. 3a) Select[k] == valid[k] & (reg[k] = R) & ~dispatched[k]

& \{ A i : (0 < i < k) & (dispatched[i] + ~(reg[i] = R)) \}

(Note that we have used predicate 2)

Defn. 8) dispatched[k] <- Select[k] & tag_valid

(Note that definitions 3a and 8 are potentially mutually recursive. This is analogous to a race condition in the hardware implementation. This is easily avoided by recognising that the operation '<- ' or 'is set to the value of' is implicitly 'will be set to the value of before the next operation of the FMIFO'. This implies, in its turn, that the operation '<- ' occurs in a different, mutually exclusive cycle of the FMIFO operation. The implementation will thus be driven by a two-phase (at least) clock. On one phase the matching may take place, on the other any data modification operations.)

Defn. 5a) tag_valid == Select[k] &

\{ A i : (0 < i < k) & (dispatched[i] + ~(critical(tag[k]) = critical(tag[i])))\}

Defn. 9) Delete[k] == valid[k] & (reg[k] = R) & dispatched[k]

(this identifies an element to be deleted)
Defn. 10)

\[
\text{Deletion}_o f(k) = \text{Delete}[k] \& \left( \forall i : (k < i < \text{MAXSIZE}) \&
\text{data}[i] \leftarrow \text{data}[\text{succ}(i)] \right)
\]

where

\[
\text{data}[i] = \text{tag}[i], \text{reg}[i], \text{valid}[i] \text{ and } \text{dispatched}[i]
\]

(This describes a shift operation on all the items behind item 'k')

3.4.1.4 Definition of Individual Elements in the FMIFO

From the definitions above, which describe the operation of the FMIFO as a whole, it is necessary to abstract a definition of an individual element. The task has been made easier by using a minimum of quantifiers in the preceding definitions, and by defining common terms (i.e. Definition 3a).

In order to complete the definition of individual elements it is necessary to remove all quantifiers, since any quantified term depends on data in other parts of the structure. Such data is not accessible to a hardware implementation or an individual element without involving extremely complex wiring. It is quite easy to access the terms of two adjacent elements. Quantified terms must be replaced, therefore, by terms involving adjacent elements, and predicates describing how adjacent elements interact. If \( k \) is an element then \( \text{pred}(k) \) is the element immediately in front of \( k \) in the queue, and \( \text{succ}(k) \) is the element immediately behind \( k \) in the queue. Element 0, or \( \text{pred}(1) \) is thus referring to a constant term to be supplied externally to the FMIFO.

The most important term to be treated is Select, in Defn. 3a.

The term \( \{ \forall i : (0 < i < k) \& \sim(\text{reg}[i] = R) \} \) can be replaced by

\[\sim \text{grant}[\text{pred}(k)]\]
and the definition

**Defn. 11)** \[ \text{grant}[0] == \text{False} \]

This gives us:

**Defn. 3b)** \[ \text{Select}[k] == \text{valid}[k] \& ( \text{reg}[k] = \text{R} ) \& \sim \text{dispatched}[k] \& \text{grant}[\text{pred}(k)] \]

The next definition that introduces quantifiers is **Defn. 5a**. The range specifier in the quantified predicate indicates that only the items in front of the selected item are of any importance. To represent this information it will be necessary to introduce a new term, 'ignore'.

**Defn 12)** \[ \text{Ignore}[k] == \text{Ignore}[\text{succ}(k)] \& \sim \text{Select}[k] \]

**Defn 13)** \[ \text{Ignore}[\text{succ}(\text{MAXSIZE})] == \text{True} \]

**Defn. 5b)** \[ \text{tag\_valid}[k] == \text{Select}[k] + \]
\[ \{ \sim \text{Ignore}[\text{succ}(k)] \& \text{tag\_valid}[\text{succ}(k)] \& \]
\[ ( \text{dispatched}[k] \& \sim (\text{critical}(\text{tag}[k]) = \text{critical}(\text{tag\_out}))) \} \]

The last definition which contains a quantifier is **Defn 10**, the description of the deletion action. This involves the moving of items towards the front of the queue. The quantifier merely indicates that all the items in a selected range (behind the item to be deleted) are to be moved. Implicitly, the use of the ‘<-’ operator guarantees that the data being shifted is read before any destructive writes take place. This demands the use of a two-phase master-slave clocking system, as is commonly used in shift registers. The action of deleting an item is locally the action of shifting in new data or not, hence:
Defn 14) \[ \text{Shift}[k] = \text{Delete}[k] + \text{Shift}[\text{pred}(k)] \]

Defn 15) \[ \text{Shift}[0] = \text{False} \]

Defn 10a) \[ \text{Deletion}_o(f)[k] = \text{Shift}[k] \land \text{data}[k] \leftarrow \text{data}[\text{succ}(k)] \]
3.5 Evolution of a nMOS Implementation:

Using the definitions developed above it is possible to design nMOS cells that satisfy the individual equations, and then combine them to form a single element in the FMIFO.

There are two basic types of cells required, those that store and compare data, and those used to arbitrate decisions based upon the data in the rest of the FMIFO. The requirements of the data storage cells are based upon the usage pattern of the FMIFO entries. One important consideration is how long instructions will take to execute, hence how quickly an item will be removed, and by corollary, how long the data needs to be retained. In general it is desirable to support relatively slow external devices, even asynchronous ones. If this occurs then it must be assumed that the retention time of the storage element needs to be arbitrarily long. To achieve this end a storage cell based upon the pseudo-static memory cell has been chosen. The other capabilities needed are based upon the definitions given above:

1) Data retention over an unknown period of time.

2) Be written from a single bus (Defn 7a)

3) Compare stored data with data on a single bus (Defns 3b,9,5b)

4) Combine with other cells to compare multi-bit data

5) Act as a shift register element when required. (Defn 10a)

In order to satisfy conditions 3 and 4, circuitry is needed to compare two multi bit values. Such a circuit is given in Figure 3.2:
In this circuit the two inputs A and B are each capable of pulling the "match" signal to ground if the other is asserted. The action will also effect the data signal, so the data signals must be buffered, which guarantees that they will be able to pull down the "match" line. The cells can be wired up in parallel to perform multi-bit comparisons, but only one pull-up load is required between all the cells.

The resulting data storage cell, without the write circuitry, is illustrated in Figure 3.3. This cell includes the refresh capability needed to retain data over an indefinite period, as well as being a shift register element when required.

The circuitry corresponding to Definition 3b can be typified by showing the connections between a series of cells (Fig 3.4). The names of the signals correspond to their logical function, each of which is a term in one of the definitions:

\[
\text{match} \quad (\text{reg}[k] = R) \quad (3b)
\]
\[
\text{ignore
tag} \quad \text{dispatched} \quad (5b,3b)
\]
The data terms 'dispatched', 'valid' and 'reg' occur in the boxes labelled 'select arbitration', 'write arbitration' and 'register identifier' respectively.

Figure 3.3: Associative storage cell with shift capability

Figure 3.4: Overview of Select Arbitration Circuitry
The arbitration cells all resolve such expression pairs as:

\[
\text{Select} = \text{Match} \& \text{grant}
\]

\[
\text{grant}_{\text{out}} = \text{grant}_{\text{in}} + \text{match}.
\]

The circuitry to achieve this is very simple, and is shown in Fig 3.5. This particular form is presented in Pucknell & Eshraghian. The problem with this design is that time must be allowed for the appropriate signals to propagate through a chain of such cells.

![Figure 3.5: NMOS Implementation of Arbitration Circuitry](image)

The select arbitration circuitry also works in conjunction with the Delete and Shift operations, as can be seen from the more detailed view given in Fig 3.6. In this circuit the term/label correspondences are:

- \text{shift}_{\text{request}} \rightarrow \text{Shift}
- \text{mark} \rightarrow \text{Select}
- \text{clash} \rightarrow \text{tag}_{\text{valid}}
Figure 3.6: Selection and Deletion Arbitration

The writing of the 'dispatched' flag occurs after the FMIFO access cycle. Careful attention must be paid here to the possibility of data modification and shifting occurring simultaneously. The shifting circuitry must allow the new datum to be shifted out as it is written, or to be written into the cell. This can be achieved by placing the writing switch as indicated in Fig 3.7, and making the 'write' signal coincident with $\phi_1$. If this happens, then on $\phi_1$ the new datum will propagate through to the output and also overwrite the datum stored by the refresh circuitry. The datum to be written needs to be in complemented form, so the read capability inverts the signal when it pulls down the precharged "data" wire. The first inverter is used to read the new datum in, while the second stage is providing the output, either the stored value or the new datum, to the next cell. After this has occurred then the datum is clocked into the second stage, where the value is accessible to the comparison circuitry.
Figure 3.7: Placement of 'Read' and 'Write' Capabilities

The cells used to store the tag field are identical to those used to store the register identifier, since the bus is unidirectional by nature. The tag field and clash arbitration cells are shown in Fig 3.8.

Fig 3.8: Overview of Tag Field Checking Circuitry
The significant difference in this subsection is that the clash arbitration allows any match to assert the 'clash' signal, if the 'ignore' signal is not asserted. Fig 3.9 shows the circuit.

![Figure 3.9: Tag field check and 'ignore' arbitration circuitry](image)

The FMIFO is completed by joining all the cells discussed together, and distributing the power supply rails and clock phases as necessary. The final configuration, including external connections is shown in Fig 3.10.

![Figure 3.10: FMIFO External Connections](image)
3.6 Performance of the FMIFO Unit

3.6.1 Scope of the Analysis

The speed of the FMIFO unit has been identified as the crucial factor governing how fast the ring of registers can cycle, hence the relative cost of latency (waiting for data) compared to bus-styled data access. Using the circuitry described above it is possible to gain an estimate of the unit’s performance. There are too many factors which determine the speed of a fabricated unit for any meaningful attempt to precisely measure performance. Instead a rough analysis, in the style of V.L.S.I. design texts, will be developed to give some idea of the delays inherent in this simple design. This circuitry is, after all, a straightforward translation from the logical specification with little or no attempt to produce the fastest possible circuit. Such a task is beyond the scope of the preliminary investigation of a new architecture. As inefficiencies are identified, however, possible alternatives will be suggested and the expected improvement in performance discussed.

3.6.2 A Note on Terminology

In analysing the performance of the nMOS implementation of the FMIFO, the Mead & Conway lambda-based methodology will be assumed to have been used to generate devices. This allows the use of the concept of a 'fundamental delay unit', \( \tau \). This represents the time taken to charge the gate capacitance of a \( 2\lambda \times 2\lambda \) transistor through a pass transistor of similar dimensions. Using \( \lambda = 2.5 \, \mu m \) this delay time is approximately 0.2 - 0.3 nanoseconds. To complicate matters static inverters exhibit asymmetry, because pull-up and pull-down devices are generally of different dimensions. Furthermore, the load being driven by an inverter affects its response time. The geometry of the mask layout determines the resistances of the pull-up and pull-down devices, and hence the overall response time. In general the faster the response time, the more power is dissipated. When a regular structure (such as a multiplexer or an array of memory cells) is being considered, space and power demands are likely to be more important than if a single inverter is being considered to drive a signal. In this case speed is likely to be the overruling factor.
For the most part standard results will be introduced without justification, and if unfamiliar with the details, the reader is referred to one of the V.L.S.I. texts that use Mead & Conway design rules as a basis (Mead & Conway, Pucknell & Eshraighan etc).

### 3.6.3 The FMIFO Access/Modification Cycle

To estimate the delay associated with the FMIFO's operation it is necessary to identify and calculate the delay associated with each individual part of the cycle. From this information the critical set of delays can be found, and the total access time estimated.

The tasks that need to be performed during each access cycle are, in rough sequence:

1) put the current register number onto the appropriate bus lines.
2) match stored register numbers against this data
3) arbitrate to select the first matched register
4) write the appropriate tag field onto the tag_out bus lines
5) match the critical parts of the tag fields against this data
6) arbitrate the tag data's validity
7) Use the tag data (write it into a register's tag if appropriate)
8) cycle ring
9) increment current register counter
10) Modify FMIFO data (delete, insert items)

Although this list is in sequential order, some of the tasks can take place simultaneously, for example the action of cycling the registers (8) and incrementing the current register counter (9). At this stage it is profitable to examine each task in more detail thereby arriving at a measure of performance. By examining the interrelation with other tasks the critical delays may also be identified.
Step 1: Writing the register number onto the bus lines:

This can occur as soon as the bus lines are not carrying the data to be inserted into the queue. If it is assumed that this data will be used on $\phi_2$ then the register number can be written on the falling edge of $\phi_2$, since the delay in charging the bus lines should be sufficient to avoid glitches being used as data during the $\phi_2$ insertion. This bus-charging delay is now of importance, since nothing can occur until the bus is charged. Unlike a bus in a register file, the bus in question needs only to be written by two sets of drivers, the microcode sequencing unit and the current register counter. The buffers writing the data can afford to be large and powerful for the sake of speed. The other important factor is that the bus itself is quite short, being local to the FMIFO, and as such has a very small intrinsic capacitance compared to a global data bus.

At the expense of running two sets of wires, the register number can be available as soon as the counter has incremented, and the data to be written can be placed upon the other set at any stage. Thus there is no need for bus arbitration, and the delay of writing onto the bus through a pass transistor is avoided. Using this modification it is probable that this particular delay will be quite small, the actual value being determined by layout, fabrication variances and the number of elements in the FMIFO unit. If the incrementer is able to supply the new value before the end of the $\phi_2$ clock period then it is likely that the delay will be 'swallowed up' during other delays.

Steps 2 & 3: Selecting the first element with a matching register number:

With the current register number simultaneously available to all the cells in the FMIFO, all matching of register numbers takes place at once. The circuitry presented allows all bits to be matched at once, through a wired-OR mechanism. Using minimum sized devices, the circuitry of Figure 3.2 will have a switching speed, driving a single gate capacitance, of $2\tau$, since there are two standard resistances in series in the pull-down path. Using transistors with length / width dimensions of $2\lambda \times 4\lambda$ the channel resistance is halved, but the input gate capacitances are doubled, hence the delay is still $2\tau$. The pull-up time is irrelevant, since this can happen while the ring is cycling (a technique known as precharging).
The other side of selecting an element is the arbitration to select the first match in the queue. Using the analysis given by Pucknell & Eshraighan for the circuit of 3.5 the delay for every 4 cells is $16\tau$ for the pass transistors and a further $5\tau$ for the inverter pair to restore logic levels, making a total of $21\tau$ per 4 cells. Thus the total delay of the arbitration will depend upon the number of cells in the FMIFO unit. Circuitry that performs the arbitration without using this pass-transistor chain would be advantageous if it were not too expensive in terms of complexity.

Thus the total delay to select an entry for an example FMIFO unit of 8 entries is $(21 + 21 + 2)\tau = 44\tau$. If there is no element selected then it would be possible to skip the rest of the FMIFO cycle at this stage, however this involves a sophisticated asynchronous control implementation.

Step 4: Writing the selected tag field onto the tag_out lines:

Contrary to the case of the register field, the tag field data is sourced from the FMIFO elements, therefore less scope for introducing powerful buffers is possible, since every element would require them. The standard delay is the time required to charge one standard capacitance (the gate of a minimum feature size transistor) through a standard resistance (the channel of such a device), so the delay for writing the tag data to the bus lines is determined largely by the capacitance to be driven. The tag_out data lines are made available to all the cells in the FMIFO, where they drive the gates comparison circuits similar to fig 3.2. Thus for N FMIFO elements there are at least $2N$ gates to be driven. The bus lines themselves (and hence all gate capacitances) can be precharged prior to this part of the access cycle, so the pull-down time is critical. A transistor with a low channel resistance (1:2 or greater) can be used with good effect, since a 1:k transistor will take $k\tau$ to switch, but will discharge $2N$ capacitances in time $2N/k\tau$. From this, it can be seen that the optimum value of $k$ occurs when

$$\frac{d(k + 2N/k)}{dk} = 0$$

i.e. when

$$1 - 2N/k^2 = 0$$

$$k^2 = 2N$$
so that for \( N = 8, k = 4 \) and a 1:4 pull-down device is called for to optimise speed, thus the total delay to use the select signal to write the tag data out for \( N = 8 \) elements can be estimated at \( 4 + 2 \times 8/4 = 8 \tau \).

Obviously, the more elements in the FMIFO the longer this will take, and the best value of \( k \) will be larger, however increasing the value of \( k \) increases the size of the pull-down device. Some compromise will become necessary if the device size becomes undesirable. For reasonably small values of \( k \) the speed factor is likely to be paramount, and device size relatively insignificant.

Steps 5 & 6: Checking tag validity:

The tag data matching circuitry acts in the same way as the register identifier matching circuits, hence a similar delay of \( 2\tau \) is expected for this operation.

The propagation of the information that an element before has been selected, in other words the negation of the 'ignore' condition, is a more significant delay. Using the circuitry given (figure 3.9), the ignore signal will have to propagate through one inverter and one NOR function and then have to drive at least 2 transistor gates, of minimum size. The delay for the NOR function to drive the two inputs will be \( 2\tau \) (pull-down) or \( 8\tau \) (pull-up). It can be seen that the ignore signal will normally be asserted throughout the arbitration chain, therefore only the pull-down time is important in this case. The inverter stage, on the other hand will be normally 'on', hence the pull-up time is important (typically \( 4\tau \)) hence a total delay of \( 6\tau \) per stage, hence \( 6N\tau \) for \( N \) element FMIFO. Using, once again, the example of \( N = 8 \), this gives us a total worst-case delay of \( 48\tau \) for the ignore propagation, thus \( 50\tau \) for the tag data checking operation.

This is the last operation that needs to be performed before the FMIFO output is ready.

Steps 7-10:

The remaining steps are concerned with the ring cycling mechanism, and the performance of the FMIFO will not affect these, as long as the FMIFO is capable of performing its modifications during the ring cycle. If the same two-phase clock is used, the FMIFO modification (delete & insert) can take place in one clock period (one pulse each phase).

Step 7, using the FMIFO's output data will occur during the first phase of the clock, hence it does not add to the delay of the FMIFO unit.
3.6.4 Size vs. Speed

The delays that have been calculated depend upon the number of entries supported by the FMIFO unit. At some point there will be a tradeoff between the number of entries (hence the amount of pipelining possible) and the speed of the unit. More detailed discussion of these considerations follow in Chapter 4, in which implementation choices and performance are analysed. The relationship between FMIFO size and matching speed is shown by Figure 3.11.

![Figure 3.11 FMIFO size vs. Matching Time (τ)](image)

Given a FMIFO with 8 elements, the total delay as calculated is:

\[
44 \quad \text{(first register match)} \\
8 \quad \text{(tag field output)} \\
50 \quad \text{(tag field check)} \\
= \quad 102\tau
\]
For analysis and comparison with bus-architectures both will be discussed in terms of their theoretical performances, there will, however, be some further delays (for example suppressing the 'register match' if the item is waiting for deletion) but these will tend to be insignificant (<10τ) In addition, some stray capacitances will slow the unit down somewhat. Finally, a margin for safety must be built in, so the clock speed will have to be slower than the estimated performance of the system indicates. To account for all these factors then an 8-element FMIFO could be expected to take less than 200τ for data access. This is comparable to the carry-chain delay in a 4-bit full adder, without the safety factor.

The cycle time of the machine is the total time for the FMIFO to generate the next tag plus the time taken to perform a shift operation. If the non-overlapping two-phase clock presented by [Mead & Conway, 80] is used to control the shift operation then the shift operation alone would take around 100τ. Thus when taking FMIFO size into consideration this overhead is of considerable importance. In practice it may be possible for the FMIFO operation to overlap with the shift operation - further reducing the relative effect of FMIFO speed on machine speed.

As an afterword, the FMIFO design presented is not claimed, or expected, to be an optimal solution, and with more time and experience a VLSI designer should be able to improve on this performance.
4 PERFORMANCE ANALYSIS

4.1 Aims & Constraints

The issue of performance evaluation of general-purpose microprocessors has always been a contentious one. Even between fully-developed products it is not always possible to identify a 'fastest', although each manufacturer seems to be able to justify their own product's claim to this position. One of the factors that leads to this confusion is the use of different instruction sets, and the emphasis, in terms of hardware support, on particular types of instructions. Added to this is the difference between CISC and RISC architectures and the associated dependency upon memory speed: RISC requires more instructions to be fetched, although each is processed more quickly. Different types of programs will be suited to some machines more than others, so the domain of benchmarking becomes highly complicated. Given these factors, it is readily seen that an absolute measure of performance is not readily achievable.

In this study an alternative architecture has been presented as a 'building block' with which different machines (instruction sets, register sets, etc.) can be implemented. Initially the speeds of operation within this building block are derived and compared to the analogues in a bus-oriented architecture. However, to analyse the performance of the 'building block' it is not sufficient to calculate the speed of the circuitry, because the efficiency of the circuitry in performing an instruction depends greatly upon the use that needs to be made of the circuitry: in short the number of 'microinstructions' needed to complete a 'macroinstruction'. If an existing machine (e.g. 68000) were to be implemented using the exact same silicon technology used in the original, then it would be possible to measure how good this particular architecture is at implementing one particular machine. If a range of machines were to be compared in this way, then a more accurate idea of relative performance might be gained, but then there will still be the possibility of a machine which utilises the potential of either this (ring-based) architecture or the traditional (bus-structured) architecture in an optimal fashion. In the history of microprocessors to date this problem of optimality has shown no signs of being addressed successfully.
Having said that it is not possible to gain an absolute measure of performance, it is nevertheless desirable to evaluate the performance of an example machine. The problem that ensues from this observation is one of specifying a suitable machine and designing the internal circuitry to implement these specifications. The resulting machine must be capable of showing the advantages of the new architecture without leaving one open to the claim of choosing an unrealistically contrived example to enhance the results.

Section 4.3 examines the relevant factors affecting the specification and design of an example processor. From this discussion a number of these parameters will be fixed, with justifications presented, to provide a generic machine with which to explore the effects of varying the remaining parameters. The results of extensive simulation studies will be used to examine the interdependence of performance and the design parameters, and to arrive at optimal values for these same parameters.

In conclusion, the performance of the derived machine will be predicted in absolute terms to provide a reference for comparison to existing machines.
4.2 Comparison of Circuitry Performance Potential

The aim of this comparison is to gain some idea of the tradeoff between having registers available to all destinations through a bus one at a time, and having registers available to a number of different locations at once, although a certain latency is expected before a particular register is available. Given that a single register access takes a certain time in a bused environment, there is a certain cycle speed at which a register access will take a comparable time in a ring-based system. The cycle speed achievable is based upon the FMIFO matching speed, and the two-phase clock cycle to shift the ring. Assuming a synchronous control approach, in which the ring is cycled at a constant rate which allows for the worst case FMIFO operation, the clock phase durations can be the minimum period which will allow the clock signals to propagate to the numerous gates of pass transistors. Fortunately the clock speed need not be synchronised with any off-chip activity and large (and/or multiple) driving buffers can be used. The net result is that the clock speed can realise the maximum rate arrived at by Mead & Conway of @100τ with the FMIFO operation period added, totalling around 200-300τ. Since this figure is in terms of τ there will be an increase in speed as λ is scaled down.

For a metal bus line the effect of scaling λ does not change the time it takes to charge up the stray capacitances, since the scaling of thickness increases resistance at the same rate that reduced length decreases it. Capacitance is not expected to change significantly [Mead & Conway,1980] so the RC product is constant. In the example presented by Pucknell & Eshraghian [Pucknell & Eshraghian,1985] a bus for a very simple 4-bit processor in 5 μm technology has an expected charge time in the order of 120 nsec. To be added to this value is the time it takes to select a register, a function of the demultiplexer circuitry which decodes the binary register number, and the two-phase clock used to synchronise signals. For 4 4-bit registers the same analysis predicts a delay of 43τ. Using the same circuitry and analysis then in general for M N-bit registers the delay will be in the order of \( \log^2 M + 8N \tau \), which would result in 8 32-bit registers needing about 250τ. Using 5 μm technology with \( \tau = 0.2 \) nsecs, the total delay will be 120 + 250*0.2 = 170 nsecs for a single bus access. In the same technology the ring cycle time would be in the order of 60 nsec. On this basis it would seem that a data transfer has to be completed every 3 ring
cycles to be competitive, but there are some important modifications that need to be made to the assumptions used.

Firstly, as the feature size decreases, the ring-based solution will become far more competitive. Although it is risky to make absolute statements about speed-up factors, if we assume the use of 1 μm technology then the value of $\tau$ will be around 0.04 nsecs, then the register access time will be closer to 130 nsecs while the ring-cycle will be about 12 nsecs.

In the bus access model no allowance was made for synchronisation of signals throughout the chip, usually requiring a system clock to drive it. From this it can be seen that a further clock period (say 100τ again) should be considered, although this will allow pre-charging of the bus lines to occur on $\phi_2$ which will offset the degradation in performance somewhat.

The final word on the comparison is that the ring-based architecture uses no global signals other than the clock, and can be expected to perform close to the theoretical prediction with confidence, whereas the bus architecture will require very careful and involved planning of control signal timing, probably involving the use of substantial safety margins to allow for 'clock skew' effects.
4.3 Specification and Design Choices

The performance of a processor can be coarsely described as the speed with which it can complete a wide range of tasks. From this it is obvious that there are external factors which greatly influence performance. The speed of external devices is significant, as is the quality of the program code to perform the tasks. Unfortunately it is impossible to divorce these factors from an analysis of the performance of a machine because they are often influenced by the details of the machine: code efficiency is tightly coupled to the instruction set, slow memory may be mitigated by minimising access - which itself depends upon the instruction set and number of registers available.

Recognising the interaction between internal and external factors, performance can be related to three different classes of factors: circuitry speed, instruction set choices and the complexity of the circuitry available, both internally and externally. For a ring-based architecture using the FMIFO described above these classes include the following sets of influencing factors:

1) Circuitry Speed
   - technology used
   - Cycle Speed
   - FMIFO matching speed
   - Asynchronous vs. Synchronous control
   - Relative speeds of cycling, ALU (and other active elements) and memory

2) Instruction Set
   - suitability to tasks
   - number of registers needed
   - pipelining potential
   - Decoding required
3) **Implementation Details**

- number of FMIFO elements
- instruction fetching method
- Caching: internal and external
- Number of different active elements

Sections 4.3.1 & 4.3.2 will investigate the interactions and significance of each of these classes, and attempt to draw a comparison between bus-architectures and the potential of ring-architectures.

### 4.3.1 Circuitry speed

The circuitry speed is heavily dependent upon the technology used, but this dependency is readily isolated by discussing performance in terms of the scalable RC time constant $\tau$ introduced earlier.

FMIFO and ring cycle times have already been discussed and shown to be dependent upon the FMIFO size, defining the effects of which is a major target of the simulation studies following.

An important choice to be made when designing any large VLSI system is between synchronous and asynchronous internal timing. Asynchronous timing allows the optimum speed of the circuit to be realised, at the expense of an extremely involved design. In practice to reduce this design task to manageable proportions approximations and safety factors are built in which severely reduce the advantages of asynchronous timing. On the other hand synchronous circuitry is a lot easier to design and scale. The performance costs are acceptable in this light.
4.3.2 Instruction Sets

The choice of an instruction set is of prime importance in determining the performance of a processor. The instruction set determines the number and size of general-purpose registers, any specialised registers, the Active Elements required and whether the instructions can be simply decoded or used to access a micro-code store. If pipelining is achievable by the architecture, the extent will be determined by the nature of the instructions. Finally, to complicate the issue beyond any hope of realistic performance measurement, the available instruction set will determine the frequency with which instructions are used.

In a ring-based architecture the number of registers to be supported is especially important, since the time it takes a register to be accessed is a linear function of how many are in the ring. The relationship is not as simple as this, though, because the more registers in the ring the greater potential there is for a multiplicity of instructions to be pipelined. There is here scope for much further research to explore this tradeoff. On a superficial examination it would appear that the fewer registers the quicker any one particular instruction will execute, but the throughput will not necessarily be higher.

The problem of maximising pipelining efficiency through instruction set choice is highly complex, if indeed it is possible to address analytically at all. It is possible to reach some interesting initial conclusions however:

Memory accesses may interfere with instruction fetching and hence hinder pipelining. Furthermore, it would appear dangerous to let memory accesses occur in a random order, since the accesses may be used to control the operation of some external device, or be used to synchronise with external processors. If a memory-mapped I/O is used to initiate the use of another piece of memory then it becomes critical that the data has been written if necessary. Similarly, if memory sharing was accomplished somehow by writing 'memory free' signals it is not possible to then finish writing to a block of memory that the processor no longer has any right to. The nett result is to suggest that register-intensive operations will best suit pipelining.
The use of Active Elements with relatively long processing delays (e.g. multipliers, dividers, etc.) can increase the amount of pipelining possible. A better way of looking at the situation is that a slow instruction will not necessarily hold up subsequent instructions as long as the result is not needed. The FMIFO-based control unit design described in chapter 3 will allow this to happen safely.

PC-relative addressing will be extremely different to implement, if the contents of the PC are allowed to change while the instruction is being executed. The problem of control flow and instruction pre-fetching is not a new one, but needn't cause serious difficulties here. It would be possible, for example, to generate and insert into the FMIFO microcode that modified the PC before the microcode that used it for the next instruction is inserted. In this fashion the instruction fetch is guaranteed to occur after the control-flow statement. This would tend to have the effect of flushing the FMIFO, since no new instructions would be in the process of being decoded. Another alternative would be to have control-flow instructions decoded and dealt with before the control unit sees it. A separate instruction fetcher would then handle control flow. Such an arrangement would tend to limit the complexity of control-flow instructions, and the problem of synchronising condition flags with conditional control-flow instruction execution merits some attention.

Instruction sets which tend to use a small working set of registers (i.e. a single accumulator) will not be particularly well suited to pipelining, since it will be most likely that instructions will depend upon the result of the immediately preceding instruction. Instructions for this type of machine would tend to come in pairs, one producing a result and the next storing away out of the work space. The use of a reasonably large set of registers, such that intermediate results can be assigned to different registers, would appear to be more suited to an architecture which supports concurrent execution of instructions. One interesting possibility is the use of a small ring which allows execution of instructions producing results in scratchpad registers, and a register file which reads data to and from the ring as required. The use of a large number of registers need not slow down the execution of individual instructions. If the result of one instruction is needed immediately then it need not be stored into the register file. This approach effectively introduces a further level in the memory hierarchy, possibly a little slower than primary registers but nevertheless faster than even cache memory, and supportable by code generation instead of cache-loading heuristics.
4.3.3 Implementation Choices

4.3.3.1 FMIFO size

FMIFO size, the number of pending operations that can be accommodated simultaneously, has two direct effects on the performance of a machine. The first, very easy to measure, is the relationship between the size and the time taken to generate the output signals (see Figure 3.11). The second effect, correspondingly very difficult to measure, is the amount of pipelining that may be achieved. Theoretically, as FMIFO size increases pipelining and hence instruction throughput increases asymptotically to a level determined by the interdependence of instructions upon previous results.

4.3.3.2 Instruction Fetching Methods

The choice of instruction fetching method is difficult in the new architecture because there are many possibilities. The most simple, in terms of circuitry required, is to have as part of the microcode of each instruction the microcode to fetch the next instruction, using one of the registers as a program counter (PC) and another as an instruction register (IR). For example the instruction:

R1 <- R1 + R2

is converted into the tag-codes:

<table>
<thead>
<tr>
<th>R1</th>
<th>ALU Operand A</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2</td>
<td>ALU Operand B</td>
</tr>
<tr>
<td>R1</td>
<td>ALU Result</td>
</tr>
<tr>
<td>PC</td>
<td>External Bus Interface: Address to Read From</td>
</tr>
<tr>
<td>IR</td>
<td>External Bus Interface: Result</td>
</tr>
</tbody>
</table>
The assumption here is that there must be some way of modifying the PC. In the simulated machine, the External Bus Interface (EBI) unit included an incrementer and any address could be incremented by 0, 1, 2 or 4: the number being specified in the tag field of the address. This could be written back into the address register if the incrementer was quick enough, or could suppress the ring cycling mechanism for a while. Alternatively a further entry could be made to the microcode sequence:

```
PC   EBI   Address   Result
```

Using the tag specification method of instruction fetching, PC modifications can be achieved by using the PC register as an operand and result for the ALU. For example consider an instruction:

```
Branch Relative +10
```

i.e. \( PC <- PC + 10 \)

One important feature of this instruction is the use of a literal value, '10'. Once this value has been extracted (either from the next instruction word or from the current instruction itself) the operation is in the normal form of an ALU operation:

```
PC   ALU operand A
IR   Write literal + ALU operand B
PC   ALU Result
PC   EBI Address
IR   EBI Result
```

The operation of 'write literal' was achieved in the simulation by having a scratch register in the control unit which held the extracted literal, and a bit in the FMIFO entry which specified whether or not to dump the literal register into the current register. This necessitates some additional circuitry, but is unavoidable if the efficiency of using embedded literals in short instructions is desired. If negative literals
are to be used then either the ALU needs a distinct subtract capability or the literal extraction circuitry must perform a sign-extension.

If it is assumed that an average instruction will take one or two complete cycles of the ring to complete, then obviously the instruction fetching method will need to be faster in order to allow a reasonable degree of pipelining, because memory accesses during instruction execution will tend to decrease pipelining by holding up instruction fetching cycles (if the instruction and data buses are shared). The cost of sending the PC as the address to the E.B.I. (an average of 1 complete cycle) can be avoided by keeping the PC as a separate register near the E.B.I. instead of in the ring itself. This register can be part of a separate unit, the Instruction Fetcher (IF) which contains the necessary circuitry to initiate instruction fetches. The E.B.I. needs to be dual-ported, so that it can accept data from either the IF or from the ring. This way the instruction fetch cycle can start as soon as the E.B.I. is idle. One problem that must be overcome is PC modification. The use of a separate IF unit allows PC modification to take place without using the ring's control unit. PC modification instructions could be handled by a separate adder unit within the IF, and the next instruction fetched without ever requiring the use of the FMIFO-based control unit itself.

The question of conditional PC modification is somewhat complicated. The condition codes could be relayed to the IF, which necessitates more complexity in the IF circuitry, or the PC could be dumped onto the ring and a microcode sequence initiated which generates the new PC depending upon the conditions tested. The latter approach, while slower, removes the need for a separate adder. Since most current architectures, for example the Motorola 68000 family (Stritter & Gunter), it was appropriate to allow the simulated machine to use an adder/incrementer for the PC. In general, care needs to be taken with internal states, such as the 'carry' flag, to ensure proper operation. In the experimental machine the carry flag was returned in the tag field of the ALU result, and another flag was included in the FMIFO entry stating whether the returning carry condition was to be remembered. This meant that the carry flags generated by PC modifications did not affect the conditions set by arithmetic instructions. A number of different approaches, at the hardware or instruction set level, should be possible to ensure that condition codes are handled properly.
Given the use of a separate IF unit for the sake of speed, the communication of instructions between the IF and the CU could take place either through wires or via the ring. The memory accesses themselves could be performed on a separate external bus, as used in many mainframe architectures and microprocessors such as the Fairchild 'Clipper'. The advantages of such an approach are already recognised, the cost being an increase in circuit complexity and the number of external connections needed on each chip. Regardless of whether a separate instruction bus is used, an IF unit could include a level of instruction caching. Tight loops (such as block memory moves, multiplication or division in RISCs, etc) could be accommodated quite easily within a very small (say 16 word) instruction cache. The cost of instruction fetching would decrease dramatically during these loops and the potential for pipelining could see an extremely high throughput of instructions. Another refinement that could be included in the IF is the use of multi-instruction fetches. If instructions were 8 or 16 bits long and the external bus was 32 bits wide (or more) the memory access overhead would be dramatically reduced. The circuitry needed would have to be able to distinguish which sections of the word belonged to which instruction. The use of variable length instructions will make this complicated, but not beyond management. If fixed-length instructions were specified then multi-instruction fetches would be very simple, however the instructions would tend to be longer and the scope for time saving smaller.
4.4 Specification of an Experimental Processor

The specification of an example processor to be built upon the architecture presented introduces a question of credibility. Perhaps the most contentious issue is the choice of instruction set. It is possible to build in extremely complex instructions that perform very specialised operations efficiently, and write programs that consist of very few instructions, reducing instruction-fetching overheads artificially. On the other hand it is possible to use an instruction set in which each instruction performs a single task at the lowest level (essentially just microcode). The instruction throughput of a processor based on this will be impressive, however the tasks may not be completed any quicker. Above all, for a machine intended as an example, the instruction set and associated hardware support should be simple. The final choice of an instruction set (for further information see Appendix B) was based upon a previously defined but unimplemented RISC processor intended for a multiprocessor message-passing environment [Stafford]. This instruction set had the advantage of simplicity, and since the processor had not been designed already, flexibility. In the final analysis the changes made were to the conditional branching instruction in the interests of conventionality instead of performance, to avoid the charge of choosing an unrepresentative instruction set. Similarly, the number of registers used, while chosen to be a variable parameter was restricted by the instruction set. The programmer’s model of the machine thus has a maximum of 8 registers, though RISC-style register windowing could easily allow greater numbers of physical registers to be used (see Chapter 6). Within this limitation the effect of varying the number of registers will be treated.

The instruction set calls for very little hardware support, in that none of the instructions involve more than a simple addition and/or single memory access. This is in fact a worst-case scenario for a ring-based machine, since the opportunity to pipeline slow instructions (for example multiplications or divisions) is reduced. The Active Elements demanded are just three - a Control Unit, an ALU and an EBI - once again the minimality is a worst-case scenario for pipelining efficiency. As discussed in Section 4.3.3.2 there are two simple instruction fetching methods to be examined. The first, with the instruction fetching operation integral to the control unit using a PC in the register ring, is attractive because of the absolute minimum of circuitry it requires. It is perhaps a suitable target for very fast, but not very dense,
technologies. The other method, using a separate Instruction Fetcher (IF) that looks after the PC has distinct advantages in terms of performance.

The instruction fetching unit extracts literal values from the instructions, or fetches the literals from the next half or full word. Unconditional PC modifications are performed immediately through the inclusion of a separate adder in the IF. It would be possible to make a quick jump mechanism based upon a replacement of the least significant bits of the PC. Relocateable code must then be aligned on suitable boundaries to use this mechanism safely. Conditional branches are achieved by using the same adder, and then ignoring the result or replacing the PC with the (precalculated) result when the Control Unit signals whether or not the jump was called for. The IF suspends operations when waiting for this signal.

A final refinement included was facility for a small, contiguous instruction buffer within the IF. The buffer is filled as instructions are fetched. When the buffer is full, the first element is shifted out as the new element is inserted. A simple calculation is all that is necessary to decide if a jump is outside the buffer range, if it is the buffer is flushed. Tight processing loops, so common in programs, can thus be executed without the instruction fetching overheads.

Thus two separate machines were designed, and simulation results will be presented for each of these machines.

For several reasons, not the least of which was to facilitate coding, a synchronous machine was assumed. No assumptions were made about the speed of the technology used, all timing is relative to the sum of FMIFO matching delay and the register shift delay (hereafter this compound time unit is called a cycle). External memory times are specified as a variable parameter in terms of this 'cycle' time. Also specified in the same terms is the ALU operation time, which although expected to be technology-dependent in the same proportionality as the rest of the timing, still merits further consideration (see Section 4.5).

These factors allow us to define different environments, in response to which the values of internal parameters will be chosen. These internal parameters are the number of registers, the size of the FMIFO and the number of instructions cached within the IF unit.

For details of the implementation of the simulators refer to Appendix C
4.4.1 The simulation regime.

The simulators based upon the above machine specifications were written to produce various statistics upon completion of a program. The basic results produced were counts of the number of instructions executed, memory accesses, additions using the ALU and the total number of machine cycles. As well as these, a record was made of the maximum number of FMIFO elements used and the number of times the FMIFO was full and the CU attempted to write to it. These counts indicate the overall performance of the machine. An option was also introduced to provide further statistics on the usage of the FMIFO. During each cycle, both the number of entries in the FMIFO and the number of different instructions represented were recorded.

Several conditions affected choice of test programs. The most important condition is credibility. The programs should be commonplace and simple, and preferably useful. They should not involve an artificially high reliance on computed results that can be stored conveniently in registers, as many computationally massive examples are. For initial testing, use was made of an insertion sort of integers and the "C" `strcpy()` function. Both of these make extensive use of conditional jumps and an even mix of memory and register accesses, and need little further introduction. Once trends have been identified a more involved spectrum of test programs can be used to demonstrate performance claims, and to identify which tasks make the best use of the pipelining potential available.

From the mass of results generated by varying the abovementioned parameters, we need to extract a clear picture of the performance. This can only be achieved by reducing the quantities of data to a manageable level without ignoring any important trends. It becomes important to identify those parameters which can be fixed at some discreet level in order to analyse the effects of other parameters. With two different implementations, each capable of several different register configurations each this task is by no means easy. The parameters will be treated in an order which allows the findings of each section to be used as an assumption for the next.
4.5 Addition delays

The time taken for the addition of two words is dependent upon the technology used to build the adder, therefore addition times will be discussed in terms of $\tau$. The time taken for an addition using a Manchester carry-chain adder [Mead & Conway, 1980] is the product of the time taken for the propagation of the carry signal through each single-bit adder and the number of bits. For an n-bit adder, this is $25n\tau$. Thus a 32-bit addition would take about 800 $\tau$ using this circuitry, or about 4 cycles with an 8-element FMIFO or 6 cycles with a 4-element FMIFO. The use of more advanced adder circuits, for example a n-bit parallel adder described by [Brent & Ewin, 1982] has an addition time described by the formula $(60\log_2 n - 12) \tau$. Thus a 16-bit adder would take 228 $\tau$ (1-3 FMIFO cycles) and a 32-bit adder would take 288 $\tau$ (2-3 FMIFO cycles). Another scheme, the predictive carry adder, which while essentially a serial adder, minimises the carry propagation delay by predicting the carry bits for each operation (a bitwise AND of the input words is used as the carry bit inputs), and comparing the prediction with the resulting carry bits generated. Usually only a few reiterations are needed, and the addition times compare favourably with serial adders. The net conclusion is that the ALU operation time will almost certainly be in the order of 5-20 FMIFO cycles. Given this prediction the addition time was varied over the range 1-100 cycles, and the resulting performance measured (Figs 4.1 and 4.2). The vertical axis is the averaged number of cycles per instruction normalised by dividing by the memory access time. The different plots correspond to different memory access times.
Fig 4.1 Relative Effects of ALU speed (No I.F.)

The machine simulated in Fig 4.2 made use of multiple instruction fetches in whole-word memory accesses. Most instructions were 16-bits in length (half-word), though some were 32 or 48 bit (including literals).
Fig 4.2 Relative Effects of ALU speed (With IF.)

The obvious conclusion that can be drawn from the graphs is that until memory performance becomes comparable with addition speed, the actual speed of the ALU (within the range that interests us) is not particularly critical. On this basis a representative value of 10 machine cycles can be assumed for ALU operation to analyse the effects of other parameters, unless otherwise specified.
4.6 Memory Speed

The range of memory device speeds spans several orders of magnitude with the use of several different technologies. The question of finding a representative memory speed to use with a simulation study becomes difficult. Perhaps the best solution is to calculate the relative speeds of a number of likely implementation technologies, and of memory devices in the same cost/performance range.

If we take as a starting point the cheapest prototyping medium available, 5 μm nMOS technology, then, as shown above, a machine cycle time of around 40ns is expected. Current memory devices in the 200-400ns access time bracket are freely available at reasonable cost. Thus a relative memory access time of 10 is a representative figure for this combination.

Faster technologies already available could allow a processor implementation to cycle at far higher speeds relative to such memory. For example 0.7 μm nMOS would result in 200ns memory being about 70 times slower than the machine cycle. If the processor was to be implemented in a more expensive, faster technology a designer would be more likely to use memory of the same technology, however.

Finally, in a multiprocessor environment, becoming more and more common these days to improve performance of systems, the relative memory speed could be expected to diminish due to contention between processors. The use of caches complicates the issue even more.

In the case of varying the FMIFO size, the relative memory speed will vary for the same absolute memory speed. In this case memory speeds will be related to the technology dependent τ (eg. for 5 μm nMOS τ = 0.2ns, therefore 200ns memory takes 1000 τ).

After examining the other parameters in the light of different memory speeds the effect of varying memory speed can be addressed in the conclusion to this chapter.
4.7 FMIFO Size

To examine the effects of FMIFO size upon performance three different machine implementations were chosen, and each was matched with two different memory speeds. The memory speeds were defined in terms of $\tau$, at 1000 $\tau$ and 10000 $\tau$, which corresponds, for example to 5 $\mu$m and 1.7 $\mu$m nMOS processors with 200ns memory.

The machines are labeled as "No IF" - the minimal hardware solution, "IF" - the solution with the separate instruction fetcher and "IF,c=16" which includes a 16-word instruction buffer.

The results of the simulation, initially in terms of machine cycles per instruction, were "normalised" by multiplying by the machine cycle time (FMIFO size related) to get a figure in $\tau$, and then dividing by the memory speed.

Three trends are evident in Fig 4.3. Firstly, as the relative speed of the processor to the memory increases, the less critical FMIFO size becomes, since the memory access time becomes the bottleneck. Secondly, as the number of elements in the FMIFO increases, the performance increases until the point is reached where the amount of parallelism is no longer justified by the degradation of machine cycle time. This tradeoff point appears to be around 4 elements, for the particular instruction set and program example chosen. As has already been discussed, the instruction set and programs have not been selected to assist parallelism, so for other implementations it is expected that having a slightly larger FMIFO will be advantageous. Thirdly, as memory speed becomes slower the more advantageous is the instruction buffer, even to the point of a slow memory with buffer being faster than the simplest version with fast memory.

Bearing this in mind, the utilisation patterns of an 8-element FMIFO were recorded for the machine with the instruction fetcher and cache (buffer) sizes of 1 (i.e. whole word instruction fetches) and 16. Figs 4.4 & 4.5 show the results when external memory had a relative speed of 10 & 50 cycles respectively.
One rather unsurprising result is that by using an instruction buffer the level of FMIFO utilisation increases dramatically. The patterns also show a trend for certain levels to be 'favoured'. This is connected with the number of FMIFO entries needed for certain instructions and the mix of these instructions. Each instruction set would probably have its own 'signature' in this fashion.
Fig 4.4 FMIFO Utilisation patterns (memory = 10 cycles)
As mentioned earlier, a record was kept of the maximum number of entries in the FMIFO. With the simulations performed with a FMIFO limit of 16 entries the only times this limit was reached was when memory was relatively slow (100 cycles) and the buffer size was large (16+entries). An upper limit of 8 entries was usual without the caching available.
4.8 Number of Registers.

The number of registers available to the programmer has a dual effect in a ring-based machine. Firstly the delay between each time a register is available to any given point is linearly related to the number of registers. Balancing this is the increase in the number of memory accesses necessary as the number of registers available to hold values decreases. Furthermore, the interdependence on registers between instructions increases as there are fewer registers to use. To maximise the pipelining potential the number of registers should be fairly large. For the sample machine the instruction set demanded at least 4 registers and imposed a limit of 8 user-referenceable registers. In addition an 'invisible' scratch register was used to hold intermediate values. Versions of an insertion sort were written using 4, 6 and 8 registers, and the resulting performances are shown in Figs 4.6 & 4.7.

The use of the minimum number of registers causes more memory accesses to be made, hence it is not surprising to find that this configuration is only competitive when the memory is relatively fast. The use of an instruction buffer (Fig 4.7) doesn't greatly affect the relationship. With so simple an instruction set the use of more registers is not very attractive, however if pipelining is better supported more registers will be advantageous.

Fig 4.6 Effect of No. of Registers (1 word cached)
Fig 4.7 Effect of No. of Registers (16 words cached)
4.9 Instruction Order

With the inherent parallelism of the ring-based architecture, it becomes possible for instruction execution to overlap. The amount of overlap depends on the interdependence of data between adjacent instructions and whether they compete for the same Active Elements. If there is no such relationship then the order in which the two instructions execute is unimportant. Suppose that instruction A takes 5 cycles to complete, and B takes 10, and fetching an instruction takes a further 2 cycles. Thus the time taken to execute A then B is $2 + 5 + 2 + 10 = 19$ cycles. If overlap is included then the timing becomes:

$t=0$ Fetch A (2 cycles)
$t=2$ Start A & Fetch B
$t=4$ Start B
$t=7$ Finish A
$t=14$ Finish B

If, on the other hand, B is initiated before A, then the overlap is more complete:

$t=0$ Fetch B
$t=2$ Start B & Fetch A
$t=4$ Start A
$t=9$ Finish A
$t=12$ Finish B

To investigate this effect instructions were swapped in both the sort and strcpy programs to see what effect it made. The scope for manipulation of instruction order was fairly limited, since both programs exhibited a high degree of sequential instruction dependency. The various versions of the sort were achieved by swapping the order of the two instructions marked in the listing given in Appendix D.3 showed no difference in execution speed, due it seems to the fact that the overlap with the following instructions eliminated the difference. The strcpy function, however did show a variation in execution speed under certain conditions. The two versions are provided below:
Definition:

strcpy is invoked with the source pointer in R1 and the destination pointer in R2. R0 is assumed to hold the literal constant '0' (zero). Characters (half word) are copied from source to destination until the last character was a '0' (ascii 0).

Version 1:

:loop LD.H R3,R0(R1) ! load character
LDA R1,1(R1) ! increment source pointer
ST.H R3,R0(R2) ! store character
LDA R2,1(R2) ! increment destination pointer
ADD R3,R0 ! add 0 to character
IF ~Z,loop ! if non-zero result loop
:end

In this version the pointer is incremented immediately.

For version two the incrementing instructions are done last:

Version 2:

:loop LD.H R3,R0(R1) ! load character
ST.H R3,R0(R2) ! store character
LDA R1,1(R1) ! increment source pointer
LDA R2,1(R2) ! increment destination pointer
ADD R3,R0 ! add 0 to character
IF ~Z,loop ! if non-zero result loop
:end
The `strcpy` program was run with a string 100 characters long and relative memory speeds of 10 and 50 cycles. The slower memory results showed no variation between versions, however for the faster memory there was a significant difference (~20%), as shown by Fig 4.9.

![Graph showing effect of instruction order](image)

**Fig 4.9   Effect of Instruction Order (Strcpy(), m=10)**

The reason why the order is unimportant when memory access times are relatively large is that the two memory operations take far more time than the two increment instructions (implemented as a Load Address into Rn using Rn as a base and the embedded literal value '1' as the offset). Hence, the two increment instructions are completed before the memory accesses, regardless of the order they arrive. This phenomenon is more marked, as expected, when the instructions (an excellent example of a tight processing loop) are held entirely within the instruction buffer. When memory is faster the increment instructions are not entirely executed in parallel. It is noticeable that the larger FMIFO size (8) does allow a greater degree of pipelining, making version #2 more efficient in relation to version #1. The FMIFO utilisation patterns were also recorded but showed little variation between the two versions, in spite of the total difference in execution time (Fig 4.10)
Fig 4.10 FMIFO Utilisation (Strcpy(), m=10, f=4)
4.10 Conclusions.

The investigations described above allow the optimum configuration for the example processor to be identified. For the simple instruction set used the scope for parallelism is limited, however it has been observed that concurrent execution of instructions does in fact take place, since the order of instructions becomes important only when memory speed is not the overwhelming factor. When memory was relatively slow ( > 50 machine cycles ) then it was noticeable that the latency due to the number of registers in the ring became less important than the extra memory manipulations needed. The conclusion that can be reached is that the control structure presented here in combination with the ring-of-registers concept does in fact achieve the pipelining of instructions that was predicted.

For the instruction set used it would appear that a FMIFO with 4 elements is the most efficient tradeoff between cycle speed and parallelism attained. The utilisation patterns showed that with only 4 elements 2 or 3 instructions were often represented. The element corresponding to the result of an instruction is always the longest lived, so if more complex or slower instructions (multiplications, divisions) were supported then a larger FMIFO would become more efficient.

Furthermore, the efficacy of instruction fetching determines to a large extent the number of instructions that can be pipelined. The use of an instruction buffer as described caused a significant increase in performance, especially with slow memory. The same effect could be achieved by following the example of processors such as the Fairchild 'Clipper' by having a separate instruction bus. This bus could be connected to a 'page' of very fast memory of modest size, and the main storage could be a large amount of much cheaper, slower memory.

In the case of the minimal hardware implementation the instruction fetching mechanism tends to use most of the parallelism available, and the performance is therefore noticeably poorer than the more sophisticated implementation.

To provide some relation to the real world, the theoretical performance of the simple machine with 6 addressable registers, a FMIFO size of 4 and a 32-bit serial adder implemented in 5 \( \mu m \) nMOS (\( \tau = 0.2 \text{ns} \)) coupled to 200ns memory was measured for the sorting problem. It was found that the average instruction took 37 machine cycles to complete, or 1.16\( \mu s \) per instruction. With \( \tau = 0.02 \text{ns} \) (corresponding
to 0.5 \( \mu m \) nMOS) and the same configuration the average instruction time was 0.37 \( \mu s \). A theoretical treatment of a data-path architecture in [Pucknell & Eshtaghian, 1980 pp 205-208] in 5 \( \mu m \) nMOS concluded that a 4-bit addition operation between two registers, without any instruction fetch or decoding would take of the order of 250ns. If the same amount of time were allowed for instruction fetching and decoding, and the memory access time also included, this translates to 700ns for a register-to-register operation. A 32-bit addition would take approximately 140 ns longer than the 4-bit operation, so a comparable minimum instruction time is 840ns. Any additional memory references would add considerably to this figure. Thus, in the simplest form, a ring based machine could compete fairly equally with a bus-based machine.

The addition of an instruction fetching unit to a ring-based machine with the same parameters resulted in an average instruction time of 520ns for 5 \( \mu m \) nMOS, 184ns for 0.5 \( \mu m \) nMOS. This improvement indicates that the modest amount of extra hardware necessary would appear to be justified. The addition of a small instruction buffer (16 words) further improved the performance figures when the memory was relatively slower (73ns per instruction for 0.5 \( \mu m \) nMOS).
5.1 Introduction

Perhaps the most attractive feature of the new architecture described here is the space efficiency of the data storage elements required. The use of dynamic shift register elements means that there is no necessity for refresh circuitry in the individual storage cells. Furthermore, the fact that the data is constantly circulating around a ring eliminates the need for both buses and the associated read/write circuitry. Presented here is a comparison of some of the data storage elements that may be used, showing the design, space requirements and power consumption, using single layer nMOS V.L.S.I. technology. In the analysis of a pseudo-static memory cell a number of nMOS circuit concepts will be discussed, these will be assumed for later analyses.

5.2 Data Storage Cells

5.2.1 The pseudo-static memory cell

Circuit :

This is the memory cell introduced earlier in the paper (figure 2.1). In this cell the data is stored at the input gate capacitance of the first inverter. On the second phase of the clock the output of the second inverter is fed back into the gate capacitance to restore the voltage level there. This means that reading and writing the data can occur only on the first phase of the clock.

Power consumption :

Because the two inverters are connected, then one will always be 'turned on'. The two inverters are of slightly different geometries however, and have different power consumptions. The difference is due to the fact that one inverter is being driven through a 'pass transistor' and sees a slightly degraded input voltage. [Mead & Conway, Pucknell & Eshraghian, etc]. Following Mead & Conway methodology, it
is usual for the inverter being driven by a pass transistor to have a pull-up/pull-down impedance ratio of 8:1. (The depletion-mode transistor used as a load, or pull-up, device has 8 times the impedance of the input, or pull-down transistor). The other inverter is a 4:1 inverter.

![Figure 5.1: 8:1 inverter geometries](image)

An 8:1 inverter may be implemented in many different geometries (See Fig 5.1). The power consumption depends on the length/width ratios of the transistors used. This determines the total channel resistance when both transistors are turned on. (This resistance is placed across the supply lines). A 4:1 inverter has an $8\lambda \times 2\lambda$ pull-up transistor and a $2\lambda \times 2\lambda$ pull down transistor. If we call the resistance of a $1\lambda \times 1\lambda$ square of turned-on transistor channel $R$ then a $2\lambda \times 2\lambda$ square also has a resistance of $R$ and an $8\lambda \times 2\lambda$ transistor has a resistance of $4R$. Thus a 4:1 inverter has a total channel resistance of $5R$. The two layouts of 8:1 inverter have total channel resistances of $9R$ and $4.5R$. As a rough approximation, it will be sufficient to assume that, since the layout presented here uses the second style of 8:1 inverter that the average resistance presented across the supply rails will be $5R$. For $5 \mu m$ nMOS technology this represents a power consumption of about $400 \mu W$ per bit. [Pucknell & Eshraghian, 1985, pp168].
Area requirements:

The layout presented in Fig 5.2 is that used in [Pucknell & Eshraghian, 85]. It gives an area of $49 \lambda \times 71 \lambda = 3479 \lambda^2$ per bit. In the Second Edition of the text an alternative layout is presented that has a $59 \lambda \times 45 \lambda = 2655 \lambda^2$ area. We will use this figure as a basis for comparison with other storage cells. It is possible to reduce this area by sharing power supply rails with adjacent cells, but this also holds for the other cells that will be analysed below. The layout may not be optimum either, but once again, the layouts for the other cells are drawn from the same source in most cases, and use the same (Mead & Conway) layout rules. It is thus fair to quantitatively compare the layouts presented in this paper.
Fig 5.2: 2-bus Pseudo-static memory cell
5.2.2 The Dynamic Shift Register using Static Inverters.

Circuit:

![Shift Register Cell using static inverters](image)

Fig 5.3: Shift Register Cell using static inverters

In this circuit there are two inverter stages separated by pass transistors. Data is shifted using the two storage elements (gate capacitances) in a master-slave fashion. A non-overlapping two-phase clock is used for this function. The associated timing diagram shows how this works. Both inverters are 8:1 stages because they are being driven through pass transistors.

![Timing for circuit of Fig 5.3](image)

Fig 5.4: Timing for circuit of Fig 5.3
Area Requirements:

There are two layouts presented here: using different geometries for the transistors. These layouts are also given in Pucknell & Eshraghian. The first (Figure 5.5) uses two stages each 21 \( \times \) 24 \( \lambda \) giving 1008 \( \lambda^2 \) per bit, whilst the second (Figure 5.6) uses 20 \( \times \) 32 \( \lambda \) stages giving 1280 \( \lambda^2 \) per bit.

Figure 5.5: Shift Register layout using Static inverters
Power Consumption:

When no clocking is taking place, the first inverter has the opposite value stored to the second inverter of the previous stage. Thus one of these inverters is turned on. During clocking it is possible for both inverters to have the same value, and may thus be both on or off for the duration of the clock pulses. It is reasonable to assume that on average one inverter is switched on. The first, smaller, layout has a resistance of about 3R while the second has a resistance of about 9R, and thus using only a third as much power. Obviously there is a tradeoff here between power and size.
5.2.3 Clocked Dynamic Shift Registers

Circuit:

Two circuits are presented here based on those given by various texts [for example Carr & Mize]. The important feature is that instead of a depletion-mode pull-up transistor the inverters have an enhancement mode transistor to precharge a capacitance. On one phase of the clock this transistor is turned on and a capacitance is charged. In the circuit of fig 5.7 the capacitance at point B is then discharged on the other phase if there is a 1 stored at the gate of the input transistor (point A). In order for this circuit to work properly the capacitance at point B must be significantly greater than the capacitance at point C, and this must be explicitly designed into the physical layout. This capacitance will slow the effective clock rate that may be safely used.

![Circuit Diagram](image)

*Figure 5.7: Dynamic Shift Register using Dynamic Inverters*
In the other circuit (Figure 5.9) the capacitance is not charged if a 1 is stored because the pull-down transistor will be turned on. During this period quite a high current will flow through both transistors, as can be seen from the accompanying timing diagram (Figure 5.10).
One point to note about these circuits is that the output is only valid during the first clock phase, and that the input need only be valid for the first clock phase. This is not a problem, in that any sampling signal needs only to be synchronised with the clock.

**Area requirements:**

One of the most attractive features of the clocked logic approach is that all transistors can be minimum size. This means that the layouts are compact. The layout presented here has an area of $756 \lambda^2$ for the 6 transistor cell (Figure 5.11). These cells may also share power supply rails with adjacent cells.
Figure 5.11: Layout For Circuit of Fig. 5.9
Power Consumption:

Defining the power consumption of these cells is not an easy task. In all clocked dynamic cells, current flows when charging capacitances. These capacitances depend on the technology being used, especially the feature size. The other crucial factor is the speed at which the device is being clocked, which depends on the rest of the circuitry. It is not appropriate to enter into a detailed discussion of VLSI circuit analysis here. Instead it will be sufficient to state that the power consumption will be a lot lower than for the equivalent static inverters. For the 6 transistor gate some current flows during a clock phase when a 1 is stored at one of the stages. Power consumption will depend on the duty cycle of the clock (how long a phase pulse is with respect to the clock frequency). The 8 transistor cell avoids this current at the expense of slightly more complex circuitry, and larger internal capacitances.

5.2.4 Other shift register stages:

Other than the shift register stages shown here there are the 'ratioless' circuits. These tend to draw their power from the clock itself. Fig 5.13 shows such a circuit. Such circuits demand more powerful drivers for the clock circuitry, which make it less reasonable to derive local synchronisation signals to use for clocking.

![Figure 5.13: Another Clocked Dynamic Shift Register Cell](image)

Figure 5.13: Another Clocked Dynamic Shift Register Cell
5.3 An alternative to synchronous master-slave storage cells.

All the shift register cells discussed above are master-slave combinations. They use a two-phase clock to ensure that the data is clocked out before it is overwritten by the preceding stage. The other advantage is that each cell outputs data in the same sense as it receives it. If this is not required, however, then it is possible to use a single inverter stage per bit as long as the data shifts properly.

Consider a single clock allowing data transfer between storage elements (Figure 5.14).

![Clocking a Shift Register with one clock phase](image)

**Figure 5.14**: Clocking a Shift Register with one clock phase

This arrangement will work if the clock pulse is shorter than the propagation delay of the individual inverter stages, so that by the time the new data appears at the output of an inverter the clock pulse has finished, stopping the data rippling through more than one stage. This is hard to guarantee, given fabrication variations, and such schemes are generally regarded as being unwise, although potentially very fast. Since the control functions must be performed every shift cycle it seems unlikely that the speed is going to be advantageous. If there are no global timing constraints then there is no particular reason why the register should shift all bits simultaneously. An alternative would be to delay the clock between each stage, so that the data has shifted out before the new data is shifted in. If the shift-register is set up as a continuous loop then the first piece of data must be stored temporarily until the last stage is ready to receive it (Fig 5.15).
The actual delay, $t_d$, required between stages is related to the time it takes to set-up an inverter (charge the input gate capacitance), $t_i$, and the clock pulse width, $t_c$:

$$t_d > t_c - t_i$$

The length of the delay, $t_f$, is related to the number of registers that are to be used. Given $N$ registers,

$$t_f > N * t_d$$

One suitable method of achieving this would be to store the first bit somewhere, and then clock it into the last shift stage using the clock derived from the final stage of the delay chain. This storage cell could probably be a simple wire with sufficient capacitance, but to be on the safe side an extra storage register could be incorporated. If there were an even number of registers in the ring then this extra register would have to be a non-inverting register (remembering that the single stage register inverts the data). If there were an odd number then the extra register need only consist of single inverter stages.

The sense of the data (true or complemented) in a register at a given point in the ring can be monitored by having an extra bit presented that always holds the same value, say a logical 1. If an AE reads the register and finds this bit is 0 then it knows to complement (if necessary) the data. An interesting corollary is that an AE can have access to both senses of the data if it monitors two adjacent physical registers. Whether this will be useful will depend upon the instruction set being implemented.

Some important considerations of the single stage shift register design are:
1) The clock can be a two-phase clock, provided that the delays are sufficiently homogeneous that the two clock phases stay in step with each other. In a long series of delays this may be difficult to achieve.

2) If the rest of the architecture is capable the clock can be derived from the last stage of the delay chain (forming a free-running ring oscillator). With a two phase clock some form of synchronisation between phases will become vital.

3) If the clock pulse is of short enough duration, then the delays can be provided by using a pair of inverters (Fig 5.16).

![Figure 5.16: Using inverters to delay the clock signal](image)

These inverters will readily fit into the width of a register pair, so the delay generators will not cause a waste of space between registers. The circuit uses two propagation chains so that the clock is available in true sense after every inverter space. The complemented signal must be delayed sufficiently so that the first clock pulse delivered by the associated chain falls between the first and second pulses generated by the other delay chain. The width of the clock pulse, $t_c$, must however obey the equation:

$$t_i < t_c < t_d$$
Now if $t_d$ is generated by a pair of inverters then (due to asymmetrical behaviour of nMOS static inverters [Pucknell & Eshraighain, 85]) $t_d = 5 \times t_i$ unless we use a geometry that has a greater delay. This would tend to relieve the rather tight restraints on clock pulse width.

4) If all the Active Elements in the design function wholly independently of any global wiring then the actual timing of the register cycling mechanism is not important, because it is possible to derive a locally valid timing signal for each register. Thus it is not necessary to wait for the whole set of registers to shift before the first register can be accessed.

5) It is possible to synchronously shift registers with only single active elements by using parasitic capacitances in a master-slave role (Fig 5.17). This circuit does require a 3-phase clock, which although something of a rarity is quite achievable. The way it works is:

$C_{in}$ is charged on $\phi_1$, the parasitic capacitance $C_{out}$ is precharged on $\phi_2$. On $\phi_3$ this capacitance is selectively discharged depending on the value stored in $C_{in}$. Upon $\phi_1$ the data is available for sampling by the next stage. The actual layout of such a cell would need great care to ensure that $C_{out}$ is large enough so that charge sharing between it and $C_{in}$ would not too seriously degrade the voltage stored.

![Figure 5.17: Inverting Shift Register Stage](image)
5.4 Topology Of The Ring:

All of the shift register cells presented above allow a metal wire to be passed over the pull-up device since there is never any data stored in the gate capacitance of this device. (The capacitance between the metal wire and the gate might destroy the stored charge otherwise. Thus a minimal ring can be lain out in a linear fashion (Figure 5.18).

![Linear layout of Register Ring](image)

Figure 5.18 : Linear layout of Register Ring

On the other hand, the insertion of AEs into the ring may make it more convenient to form the ring as a convex shape (Figure 5.19), to allow the AEs to reside at the vertices.

![Toroidal layout of Register Ring](image)

Figure 5.19 : Toroidal layout of Register Ring
5.5  A Coarse Estimate of Control Circuitry Overheads

5.5.1  Fairness

It is desirable to compare the two forms of register storage that have been presented: the traditional register file and the dynamic ring approach. Each architecture has different control circuitry associated with register selection, but is not really possible to analyse the rest of the control circuitry without implementing a range of different machines with each architecture. The circuitry used to select registers is vastly different in each case. In the traditional architecture it consists of a demultiplexer with associated buffer elements for each bus. In the ring, register selection is performed by setting tag fields. Thus the ring requires registers to have extra bits.

To further complicate the issue, the overhead incurred is neither constant nor directly proportional to either word size or the number of registers in each case. It is necessary to look at how the overheads are estimated in each case before a comparison can be presented.

5.5.2  Select Circuitry for a Register File:

The description here is based on the select circuitry presented in Pucknell & Eshraighian. Typically register select circuitry will consist of a demultiplexer circuit with as many channels as there are registers to identify (N). The actual operation (read/write) will not be used as data to be switched, but instead an 'enable' signal will be generated on one of N lines. This enable signal will enable whichever of the control signals is appropriate. The actual form of the demultiplexer is shown in Fig 5.20. It is assumed that the binary coded register number is available in complemented form, however this may have to be generated.
Figure 2.20: 2 Channel nMOS Multiplexer Implementation

It can be seen that for N registers there are \( \log N \) bits of register number, and this means a total of \( 2N\log N \) transistors in the multiplexer. Practical realities make this optimistic however. One of the rules of the Mead & Conway design methodology is that no more than 4 pass transistors may be connected together in a chain. Thus every 4 transistors 8:1 inverters need to be inserted for each signal. In any case the output signals must be buffered. The select circuitry given in Pucknell & Eshraghian is to select between 4 registers and has an area of 280 \( \lambda \) by 170 \( \lambda \). (Fig 5.21) If another 4 registers were to be included the width would double, which is a linear proportionality. The height would also change, due to the extra complexity of the multiplexer circuitry. The extra transistors in the multiplexer would take another 44 \( \lambda \) for each twofold increase in N. Extra buffering (due to 8:1 inverter stages followed by 4:1 inverter stages) would take approximately another 28 \( \lambda \) (if the inverters can be placed side by side). Thus the total increase in the dimension would be in the order of 72 \( \lambda \). Buffer stages would need to be inserted at \( N > 8 \).
Figure 5.21: Relative size of Select Circuitry
From these figures we can draw up a table giving the approximate size of select circuitry for different numbers of registers:

<table>
<thead>
<tr>
<th>N</th>
<th>Width (A)</th>
<th>Height (A)</th>
<th>Area (A^2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>280</td>
<td>170</td>
<td>47600</td>
</tr>
<tr>
<td>6</td>
<td>420</td>
<td>214</td>
<td>89880</td>
</tr>
<tr>
<td>8</td>
<td>560</td>
<td>214</td>
<td>119840</td>
</tr>
<tr>
<td>10</td>
<td>700</td>
<td>286</td>
<td>200200</td>
</tr>
<tr>
<td>12</td>
<td>840</td>
<td>286</td>
<td>240240</td>
</tr>
<tr>
<td>16</td>
<td>1120</td>
<td>330</td>
<td>396600</td>
</tr>
</tbody>
</table>

As can be seen, going from 4 to 16 registers involves nearly an 8-fold increase in area.

The word length does not affect the complexity of the select circuitry. It does tend to change the relative importance of the space overhead.

5.5.3 Register Identification Overhead For a Dynamic Ring

In the case of a ring based structure the registers can carry around with them the information about the use of the data, or they can carry register identification. In the case of carrying control information there is no relationship between the number of registers and the amount of data necessary per register. In the simulated example it was found that 8 bits in the tag field were sufficient. The longer the word length the less significant this becomes. It is perhaps relevant to note that there is no need for this control information to be distributed by wires on the chip; hence a hidden saving in chip real estate that is difficult to quantify.

In the case of registers carrying numbering information, for N registers it will be necessary to have at least \( \log N \) bits of numbering.

For comparison this scheme will be presented using the smallest of the dynamic two-phase inverters. (Figure 5.9).
5.5.4 Overhead for Asynchronous Ripple Clocking

In the case where the clock is rippled along the register array by a delay chain there is an overhead for the delay devices. If inverters are used, then this means approximately two 'bits' per register. The other overhead is for a 'save' register which is approximately 2 registers. This is because the save register needs to maintain the sense of the data to stop the data changing sense in each register every cycle. If N is odd then this is only one register. The actual cost is less than 2 complete registers because there is no need for one of the pass transistors between registers. For this clocking scheme we will assume the use of the smallest of the static inverters, to save the trouble of using a two-phase clock. This unit has a per-bit area of $504 \lambda^2$.

<table>
<thead>
<tr>
<th>Number of registers</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>12</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word length</td>
<td>4</td>
<td>16</td>
<td>20</td>
<td>24</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>24</td>
<td>28</td>
<td>32</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>40</td>
<td>44</td>
<td>48</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>72</td>
<td>76</td>
<td>80</td>
<td>88</td>
</tr>
</tbody>
</table>

5.5.5 Comparative Area Requirements

The area requirements calculated for the combined selection and storage circuitry is presented here. The three different tables are associated with register files, dynamic ring using non-inverting synchronous stages and dynamic ring using inverting stages and asynchronous ripple-clocking, respectively.
<table>
<thead>
<tr>
<th>Number of Registers</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word Length</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>90</td>
<td>154</td>
<td>204</td>
<td>306</td>
<td>368</td>
<td>567</td>
</tr>
<tr>
<td>8</td>
<td>133</td>
<td>217</td>
<td>290</td>
<td>415</td>
<td>495</td>
<td>736</td>
</tr>
<tr>
<td>16</td>
<td>218</td>
<td>345</td>
<td>460</td>
<td>627</td>
<td>750</td>
<td>1076</td>
</tr>
<tr>
<td>32</td>
<td>387</td>
<td>600</td>
<td>800</td>
<td>1051</td>
<td>1260</td>
<td>1756</td>
</tr>
</tbody>
</table>

**Area of Conventional Register File \((\lambda^2) \times 1000\)**

<table>
<thead>
<tr>
<th>Number of Registers</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word Length</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>36</td>
<td>54</td>
<td>73</td>
<td>91</td>
<td>109</td>
<td>145</td>
</tr>
<tr>
<td>8</td>
<td>48</td>
<td>73</td>
<td>97</td>
<td>121</td>
<td>145</td>
<td>194</td>
</tr>
<tr>
<td>16</td>
<td>73</td>
<td>109</td>
<td>145</td>
<td>181</td>
<td>218</td>
<td>290</td>
</tr>
<tr>
<td>32</td>
<td>121</td>
<td>181</td>
<td>242</td>
<td>302</td>
<td>363</td>
<td>484</td>
</tr>
</tbody>
</table>

**Area of Synchronous Dynamic Ring \((\lambda^2) \times 1000\)**

<table>
<thead>
<tr>
<th>Number of Registers</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>12</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word Length</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>46</td>
<td>60</td>
<td>88</td>
<td>117</td>
</tr>
<tr>
<td>8</td>
<td>44</td>
<td>62</td>
<td>81</td>
<td>117</td>
<td>153</td>
</tr>
<tr>
<td>16</td>
<td>69</td>
<td>95</td>
<td>89</td>
<td>125</td>
<td>226</td>
</tr>
<tr>
<td>32</td>
<td>117</td>
<td>159</td>
<td>202</td>
<td>282</td>
<td>371</td>
</tr>
</tbody>
</table>
Area of Asynchronous Dynamic Ring using Inverting stages $(\lambda^2)*1000$
From these figures it is quite obvious that either of the two dynamic ring storage techniques require far less area than the conventional register file. Furthermore, as the number and length of registers increases the savings become more marked, as the relative overhead of register identification to register selection diminishes. Perhaps the best way to visualise this difference is by graphing the two areas on the same axes (Figure 5.22). The areas shown are those calculated for 8 32-bit registers, with select circuitry.

Figure 5.22: Area Comparison of Static and Dynamic Storage of 8 32-bit Registers with select circuitry
6 FURTHER DEVELOPMENTS

Since this paper is a preliminary investigation of a new idea there is a great deal of scope for follow-up research. So far the feasibility of the idea has been explored and the critical areas that need attention identified, however there are some possibilities that present themselves even at this stage in the development. This chapter details a number of areas in which a ring-based architecture may offer advantages, and sketches out avenues for related research.

6.1 Context Switching

Today, and in the foreseeable future, most high-performance microprocessors are being asked to host operating systems with multi-tasking capabilities. In such an environment the operation of suspending one process and starting another is called 'context switching'. The handling of interrupts, in a single tasking environment, is also a case of context switching. Different processors have different means and capabilities to achieve this, however the basis is the same for all. In all cases the program counter is saved and a new one loaded for use. In this simplest case, all other information that needs saving must be explicitly saved by the program code. For example, interrupt handlers incorporate code to save all the registers and condition codes that may be affected at the beginning of the routine, and restore them again at the end. The introduction of this extra code impairs efficiency if context switching occurs at frequent intervals. The solution is to have hardware support for full register saves and loads (as is done by the i80386 processor) The extra circuitry cycles through the registers, generally storing to (or loading from) successive memory locations.

The cost of this extra circuitry may be quite considerable for a conventional architecture, however for a ring-based architecture this form of context switching can be very cheaply implemented. The reason for this is that all the registers pass through the External Bus Interface unit even when the machine is not executing any instructions. If, as described above, the EBI contains an incrementing element anyway, the dumping of all the registers can be achieved by including a counter to keep track of the registers, and the ability to step the ring at a speed at which memory can be accessed. The dynamic elements in an nMOS
implementation can be expected to hold data for at least one micro-second, which should be more than sufficient for most memory accesses.

An alternative approach to full register save/loads is the use of 'register windowing', or selecting a working set of registers from within a larger bank of registers. This technique may be even more appropriate for the ring structure because of the cheapness of data storage and access in terms of circuitry. For example, consider two separate rings of the N elements with a connection between them such that they can be joined together through a switching junction to make one large ring (Figure 6.1). Obviously if the large ring is split into two smaller ones after N cycles the data in the two rings will have swapped over. The technique could be used with more than two separate rings, or with rings of uneven size. The cost of the switching junction is small per bit of word-length, being just an extra pass transistor and control signal routing.

*Figure 6.1* : Joining Two Independent Rings To Form One
6.2 Gallium Arsenide Implementation

The use of Gallium Arsenide (GaAs) devices has been touted as the performance breakthrough of the near future. High-speed VLSI circuits have indeed been implemented in various GaAs logic families, but the common factor of all of these families is the comparatively low number of devices that can be fabricated on a single chip. This maximum transistor count has been estimated at one tenth of that of silicon MOS technology [Militunovic, Fura & Helbig, 1986]. A further restriction on the speed of GaAs processors is the relatively high cost of off-chip communication. This cost precludes the use of a high chip-count to offset the low complexity achievable.

The net result of these two restrictions is to limit the complexity of GaAs implementations. For many specialised applications, such as front-ends for digital signal processing complexity is not critical, but for general-purpose processors device count becomes a major stumbling block. Because of this consideration it is felt that the low complexity of the general purpose data-storage and inter-chip communication scheme presented in this paper may allow GaAs implementation. If necessary the control-unit can be curtailed severely, including no internal instruction-fetching mechanisms. The FMIFO entries can be loaded directly from outside, or, alternatively, no FMIFO is needed and instructions can be processed synchronously. Either way the amount of control circuitry required is small, and the advantages of small storage cells may be realised.

The use of advanced compiler techniques to make allowances for simple (and not necessarily robust) control strategies has been seen as the key to realising the performance potential of GaAs [Militonivie, Fura, Helbig & Linn 1987]. A ring-based architecture may allow greater freedom for code optimisation and more data storage in registers on-chip.
6.3 Multiprocessor Communication

Rings have been used to communicate between processors in local area networks for some time now, notable examples being the Cambridge and IBM Token Rings. The use of buses for the same task has also been explored in examples like Ethernet. It might be argued that the two approaches can be used as alternatives for each other in most communication applications where either one is suitable. The bus method uses less complex circuitry for communication and may interconnect any two nodes, but suffers from being able to support a single transaction at any time. Rings can support many transactions simultaneously, but at the expense of denying direct access between arbitrary nodes.

The ring described here is essentially a chip-level exploration of the basic ring communication concept. It has been shown how a single general purpose processing unit can be built with a ring of dynamic elements. The next stage is to extrapolate, as have many others before, to the possibility of implementing many processing elements on a single chip.

The ring of dynamic shifting elements with tag fields to identify data can be used to communicate between these processing elements. The actual configuration of the logical interconnection between processors is unimportant, they can all be achieved by the physical reality of a ring connecting them all. The cost is, of course, the potential indirectness of the communication path. Two possibilities immediately present themselves for data identification: the tag field states what is to be done with the associated data and any available processor that recognises the code can perform the operation, or the tag field identifies the processor that will use the data. The first scheme might be suitable for a data-flow arrangement where the processors are not necessarily identical, and the connections are made at run-time. The second is more suitable for a predefined configuration (albeit reconfigurable).
6.4 Further Research Direction

This initial research has concentrated on establishing the technical feasibility of a ring-based microarchitecture, and from there exploring as many of the potential avenues for development as possible. What is now required is a more in-depth investigation of factors affecting performance. This work is likely to be of considerable proportions, since it may have to involve physical realisation of the concepts discussed here. In order to make a comprehensive study it seems probable that many of the alternative options will also have to be implemented, though perhaps the implementations do not need to be taken past the stage where performance can be measured.

There is scope for implementations of specific machines (not necessarily already defined). The type of results gained should prove to be a fairly good indicator of the potential of the ring-based architecture in the future. VLSI design students looking for a project of some originality and medium complexity may find here a suitable task.

Apart from these tasks of immediate importance there are a number of more theoretical investigations that appear to be worthy of attention. One such topic is the development of an instruction set to make the most out of a ring-based architecture. The subject has been very briefly introduced in Chapter 4, and it is fairly clear that there is scope for a body of research on this matter.

Another possibility that has been suggested is the implementation of some form of 'tag calculus' to support the passing of the result generated by one Active Element directly to another AE without intervention by the control unit. Some form of coding this information in a small bit field in the tag would be required, so that each AE performs a fixed operation upon this field, with the end result determining the destination of the data. Simplifying this greatly, imagine the ALU exclusive-ORing the bit field with 1's. Let the ALU recognise the first bit being a 1 as its addressing identity and the control unit recognise a field of all 0's. If the CU wanted to dispatch data to the ALU and then get it back immediately then it could set the field to all 1's. After performing its operation the ALU would set the field to 0's (bitwise XOR of 1's) and the CU could then recognise the data as being available for further manipulation. By the same method if one of the original bits were a 0, then one of the XORed result would be a 1, hence the data would now be addressed to another Active Element on the ring. The definition of a set of operations and a set of tag field values that correspond to different sequences of operations is the 'tag calculus' mentioned earlier.

Finally, there is almost endless scope for the physical implementation of processors using the basic technique described here.
7 CONCLUSION

The parallel between communication over a shared bus and via a ring of connections can be drawn not only at a local area network level, but also at a microarchitectural level. At both levels the dichotomy is between direct communication with sequential arbitration and indirect, but potentially concurrent communication. The performance of any ring based communication method is a measure of at least three factors, the speed of the ring cycling mechanism, the number of nodes in the ring, and the ability to exploit the concurrency that is potentially available.

The first part of this discussion has concentrated on showing how concurrency can be exploited with safety, and a simple control strategy developed. While this strategy is designed to show only the feasibility of using a ring, not the performance that may be expected, this initial exploration shows promise of a performance at least comparable with traditional bus architectures. Furthermore, extrapolating to the technology available in the future, it can be seen that the performance comparison should improve favourably as fabrication resolution increases.

In Chapter 4 the relationships between physical configuration choices and performance were illustrated with simulation results. Based upon these, the absolute performance of a processor based upon a simple instruction set was calculated, and compared to a traditional architecture treated in the same manner. The resulting analysis indicated that the amount of parallelism inherent in a ring-based control and data communication method more than compensates for the latency experienced by any single data transfer.

Perhaps the most important advantage of using a ring of dynamic registers is the space savings that may be achieved through having the refresh function and register selection performed as a side-effect of the communication method. The space and power demands of a ring of registers are very significantly less than those of a register file built with pseudo-static memory cells with porting to one or more busses. It is this economy of circuitry which makes a ring-based machine a good contender for implementation in one of the very fast, but as yet not very dense technologies, such as Gallium Arsenide.
Since the use of a ring of parallel dynamic shift registers as both a storage and communication medium appears to be a break with convention some effort has been made to examine the implications for the future. The discussion is inevitably conjectural by nature, but a number of particularly promising avenues for development have been identified. It is hoped that this work will inspire other researchers to undertake to answer some of the challenges presented.
REFERENCES

Ayres, R.F.
   VLSI: Silicon Compilation and the Art of Automatic Microchip Design
Prentice-Hall, USA, 1983

Bernstein, K. and Furman, A.
   Multiport Register File Architecture
VLSI Tech. 1984

Brent, R.P. and Ewin, R.R.
   Design of an nMOS Parallel Adder

Brent, R.P. and Kung, H.T.
   A Regular Layout for Parallel Adders

Furht, B.
   A RISC Architecture with Two-Size, Overlapping Register Windows

Carr, W. N. and Mize, J.P.
   MOS/LSI Design and Application
Mcgraw Hill, USA, 1972

Fairclough, D. A.
   A Unique Microprocessor Instruction Set
IEEE Micro, Vol 2 Number 2, May 1982
Gibson, J.C.

The Gibson Mix
IBM Tech. Report TR-00.2043, June 18, 1970

Huguet, M. and Lang, T.

A Reduced Register File for RISC Architectures

Kress, C.E.

Design of an Advanced Architecture CMOS/SOS Microprocessor
VLSI Tech. 1981

Lunde, A.

Evaluation of Instruction Set Processor Architecture by Program Tracings.
Communications ACM Vol 20 Number 3, March 1977

Mead, C.A. and Conway, L.A.

Introduction to VLSI systems
Addison-Wesley, USA, 1980.

Milutinovic, V., Fura, D. and Helbig, W.

An Introduction to GaAs Microprocessor Architecture for VLSI

Milutinovic, V., Fura, D., Helbig, W. and Linn, J.

Architecture/Compiler Synergism in GaAs Computer Systems
IEEE Computer, 1987, pp 72-93
Pucknell, D.A. and Eshraghian, K.

Basic VLSI Design

Prentice-Hall, Australia,


Stafford, G.J.

Design of a RISC processor for a Multiprocessor Message-Passing Environment

Unpublished Thesis,

Dept of Computing Science, University of Wollongong

Stritter, E. and Gunter, T.

A Microprocessor Architecture for a Changing World: The MOTOROLA 68000

IEEE Computer, vol 12 pp 43-52 Feb 1979

Sussman, A.

Compilation for a Processor Containing Pipelined Register Files

Dept. of Computer Science, Carnegie-Mellon University

Preprint : CMU-CS-86-161, October 1986

Yannacopoulos, N. A. , Ibbet, R.N. and Holgate, R.W.

Performance Measurements of the M65 Primary Instruction Pipeline

Information Processing 77 , North-holland, 1977
APPENDIX A  KEYS TO FIGURES

Key 1 : NMOS Devices

[Diagram of enhancement mode FET]
Enhancement mode FET

[Diagram of depletion mode FET]
Depletion mode FET

[Diagram of inverter]
Inverter

Key 2 : NMOS Mask Layers

[Diagram of diffusion]
Diffusion

[Diagram of polysilicon]
Polysilicon

[Diagram of metal]
Metal

[Diagram of contact cut]
Contact cut
APPENDIX B : Instruction Set of Simulated Machine

The instruction set is based upon a 16-bit instruction, with a possible 16 or 32-bit literal following. The small number of instructions are identified by the most significant bit turned on in the upper 8 bits of the instruction.

The instructions and corresponding machine codes are:

**KEY:**
- RA is register A. It appears in the bit pattern as a three bit id : AAA
- Optional fields are enclosed in []
- Alternative fields are enclosed in () and separated by the 'T' character.
- A period '.' indicates an unused bit field.

<table>
<thead>
<tr>
<th>name :</th>
<th>mnemonic(s)</th>
<th>Bit pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load/Store</td>
<td>LD (HIL) RA,<a href="RC">RB</a></td>
<td>1SDU.AAA.BBB.CCC</td>
</tr>
<tr>
<td>ST (HIL) RA,<a href="RC">RB</a></td>
<td></td>
<td></td>
</tr>
<tr>
<td>comments:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>load/store RA from address RB+RC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S = size , 0 = half word (specified by 'H' in mnemonic), 1 = word ('L').</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D=0 : load, D=1 : store</td>
<td></td>
<td></td>
</tr>
<tr>
<td>U = 1 : use the offset register (B) field.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALU</td>
<td>ADD [C] RA, [-]RB</td>
<td>0100.AAAab.CCBBB</td>
</tr>
<tr>
<td>Operation</td>
<td>AND [-]RA,[-]RB</td>
<td>0101.....</td>
</tr>
<tr>
<td>OR &quot;</td>
<td>0110.....</td>
<td></td>
</tr>
<tr>
<td>XOR &quot;</td>
<td>0111.....</td>
<td></td>
</tr>
<tr>
<td>The fields 'a' and 'b' are directions to invert registers A and B.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The CC field is a carry preset :</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x = Carry set to x, 1x Carry used from status (Carry XOR x)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operation is RA &lt;- RA Op RB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load</td>
<td>LDA RA,Offset(RB)</td>
<td>001MAAAADDDBBB</td>
</tr>
<tr>
<td>Address</td>
<td>INC Rx</td>
<td></td>
</tr>
</tbody>
</table>
MM specifies offset

00 Offset = DDDDO (Delta *2)
10 Offset = DDDD1
01 Offset = Next Half Word
11 Offset = Next Word. (must be aligned on word boundary)

RA <- Offset + RB

INC Rx = LDA Rx,1(RB) = 00110xxx00000xxx

Hop HOP label 0001SDDDDDDDDDDDD
(Short Jump)

S = sign of offset.
PC <- PC +/- DDDDDDDDDDDDD

If IF [-]ZCN,label 00001.....cCzZnN + 16 bit offset

lower case bits indicate whether the flag is significant.
Jump occurs if all significant flags match the bits in the instruction.

Call CALL RA 0000001......AAA
PC <-> RA

Nop NOP 00000001...........

Halt HALT 00000000...........

Halt is a special case of a more general interrupt/context switch capability that would be needed in a real processor. It is used for the simulator to find a termination condition.
APPENDIX C : The Simulator

The simulator code followed very closely the structure of the hardware proposed. The mainline consisted of a loop, each cycle of which corresponded to a machine cycle. On each cycle the Active Elements were dealt with, via procedure calls to the modules. A module was passed the tag and data associated with the register in its operation window at the time. The registers were implemented in an array, which was shifted each cycle. The overflowing register was placed back in the front. All signals between modules were simulated by procedure calls.

The code samples presented here are extracts from, in order:

A) the main control loop
B) The Control Unit showing FMIFO operation
C) The Instruction Fetcher

A - the main control loop

/* definitions of data storage elements */

DATA regs[NUMREG+1];
OPTAG tags[NUMREG+1];

#define ENDS TATE halted

int halted = 0, PAUSE = 0;
int ADDTIME = 1, MEMTIME = 1, QUEUESIZE = 9, cache_size = 1;
long int icount=0, memcount=0, addcount=0, quemax=0;
float totalcount=0., quefulls = 0.;

int No_of_Regs_Used;
int PC, IR, SCRATCH;

/* main line causes loading of memory and interprets configuration information. Then it enters an interactive command interpreter. Commands may specify how many cycles to perform (value of 'cycle' variable). If cycle is negative then the machine cycles until completion or an interrupt. */

/* main control loop */

while( cycle && !ENDSTATE && !PAUSE )
{
    totalcount++;
    if( cycle > 0 ) cycle--;
    Do_shift(); /* what do active elements do? */
    Adder(&regs[ADDSLOT],&tags[ADDSLOT]);
    EBI(&regs[EBISPOT],&tags[EBISPOT]);
    Control(&regs[CONSPOT],&tags[CONSPOT]);
    Shifter();
    if( display )
        Display();
}
/* ring shifts round one */
Do_shift()
{
    unsigned i;
    for(i=No_of_Regs_Used;i;i--)
    {
        tags[i] = tags[i-1];
        regs[i] = regs[i-1];
    }
tags[0] = tags[No_of_Regs_Used];
regs[0] = regs[No_of_Regs_Used];

    /* each machine cycle a new element can be shifted into the FMIFO */
    FMIFOaddO;
}

- 123 -
B - The Control Unit

#include "orac.h"
#include "bits.h"

extern int No_of_Regs_Used,IR,PC,SCRATCH;

static BITS32 delta_if_data;
static BITS16 if_inst;

static NUMTAG current=0,targetA,targetB,targetR;

static FLAG Carry=0,Zero=0,Neg=0,Carry_pending=0,deltaused = 0,delta_queued = 0;
/* remember what the last result returned */

extern int icount,QUEUESIZE,quemax;
extern float quefulls,totalcount;
extern int halted;
/* associative queue of pending operations : data then primitives */

#define ENOUGH 100
#define NONE (-1) /* no match in queue */
#define DUMP 1
#define NO_DUMP 0

#define FLAGSBACK 2
#define SEEN 4

typedef struct que {
    NUMTAG which;
    OPTAG whofor;
    FLAG flags;
#ifdef stats
    int inst_number;
#endif
} Q,*QP;

#ifdef stats
/* queue utilisation statistics */
static unsigned int qlevels[16+1]; /* qlevels[x] = cumulative count of how many times x elements in FMIFO */
static unsigned int numinsts[16+1]; /* ditto for number of instructions represented */
static unsigned int irange[ENOUGH];
#endif

/***************************************************************************/
/* FMIFO operation code */
/***************************************************************************/
static Q pending[ENOUGH], MC[ENOUGH];
static int fullness = 0, mchead = 0, mcp = 0;

int match(reg)
NUMTAG reg;
{
    QP next = pending, last = pending + fullness;
    while( next != last )
    {
        if( next->which == reg ) break;
        next++;
    }
    return( next == last ? NONE : next-pending );
}

int check(from)
int from;
{
    OPTAG whatis;
    whatis = pending[from].whofor & ( IDFIELD | OPFIELD ); /* mask out details */
    while(from--)
    {
        if( (pending[from].whofor & ( IDFIELD | OPFIELD )) == whatis)
            break;
    }
    return(from); /* which clashed (-1 if none) */
}

/* set up FMIFO entry that will be cycled into FMIFO */

void add(reg, tag, dump)
NUMTAG reg;
OPTAG tag;
FLAG dump;
{
    MC[mcp].which = reg;
    MC[mcp].whofor = tag;
    MC[mcp].flags = dump;
    #ifdef stats
        MC[mcp].inst_number = icount;
    #endif
    mcp++;
    if( mcp == ENOUGH ) mcp = 0;
    if( mcp == mchead ) Error("Queue overlarge");
}

/* cycle an entry into FMIFO */
/* called once per cycle, also does stats */
void FMIFOadd0
{
if( mcp == mchead )
  ;/* nothing waiting */

else if( fullness == QUEUESIZE )
{
  quefulls++;
}
else  
{
  pending[fullness].which = MC[mchead].which;
pending[fullness].whofor = MC[mchead].whofor;
pending[fullness++].flags = MC[mchead].flags;
#ifdef stats
  pending[fullness].inst_number = MC[mchead].inst_number;
#endif
  mchead ++;
  if( mchead == ENOUGH ) mchead = 0;

  quemax = (fullness>quemax)?fullness:quemax;
}
#ifdef stats
  qlevels[fullness]++;
  stats_scan(irange,numinsts);
#endif
}

} /* delete */

void delete(this)
int this;
{
  fullness--;
  while( this < fullness )
  {
    pending[this].which = pending[this+1].which;
pending[this].whofor = pending[this+1].whofor;
pending[this].flags = pending[this+1].flags;
#ifdef stats
    pending[this].inst_number = pending[this+1].inst_number;
#endif
    this ++;
  }
}

**************************************************************************/

Control(reg,tag)
BITS32 *reg;
OPTAG *tag;
{
  int matched;
  /* look for new instruction then check work queue */
  if( (!delta_queued) && Inst_Fetch(&if_inst, &if_data ) ) /* newt instruction */
  {
```c
#ifdef DEBUG
    fprintf(stderr,"got instruction, data\n");
#endif

    Decode(ifInst);

    current = (-current + No_of_Regs_Used) % No_of_Regs_Used;

    if( (*tag & IDFIELD) != CONTROL )
        return;

    matched = match(current);
    /* is it an acknowledge ?? */

    if( pending[matched].flags & SEEN )
    {
        if( pending[matched].flags & FLAGSBACK )
        {
            Zero = (*tag & ZEROFLAG) != 0;
            Neg = (*tag & NEGFLAG) != 0;
            Carry = (*tag & CARRY) != 0;
            Carry_pending--;  
        }
        delete(matched);
        matched = match(current);
    }

    if( matched != NONE && check(matched) == NONE )
    {
        *tag = pending[matched].whofor;
        /* dump literal value into the register */
        if( pending[matched].flags & DUMP )
        {
            *reg = delta;
            deltaused = 0;
            if( delta_queued )
            {
                delta_queued = 0;
                delta = if_data;
                deltaused = 1;
            }
        }
        pending[matched].flags |= SEEN;
    }

    int Decode(inst)
    BITS16 inst;
    {
#ifdef DEBUG
    printf("nDecode got %x",inst);
#endif
        /* combinatorial logic that eats a instruction, and decides what to do */

        targetA = (inst & RegA) >> 8;
        targetB = inst & RegB;
        targetR = targetA;
    }
```
if (Bit(inst, LoadStore))
    SU_Loadstore(inst);
else if (Bit(inst, Alu))
    Setup_Alu(inst);
else if (Bit(inst, Loadaddress))
    SU_Loadaddress(inst);
else if (Bit(inst, Hop))
    ;
else if (Bit(inst, Call))
    SU_Call(inst);
else if (Bit(inst, If))
{
    if (Cany_pending)
        return(0); /* do not get next instruction, ignore this one for now */
    SU_If(inst);
}
/* else nop */
else if (Bit(inst, Nop))
    ; /* do nothing */
else /* switch */
    halted = 1;

return(1);

SU_Loadstore(inst)
BIT16 inst;
{
    NUMTAG offset_reg, use;
    OPTAG rw_mask, size;
    if (Bit(inst, 12)) /* use offset */
    {
        offset_reg = (inst & OFFSET_REG) >> 3;
        add(targetB, ADDID ILOADA, NO_DUMP);
        add(offset_reg, ADDID ILOADB, NO_DUMP);
        add(SCRATCH, ADDID IRESULT, NO_DUMP);
        use = SCRATCH;
    } else
        use = targetB;

    rw_mask = Bit(inst, 13) ? WRITEIT : READ;
    size = Bit(inst, 14) ? TWOWORD : 0;
    if (Bit(inst, 6)) size = INCREMENT;

    add(use, EBIID I rw_mask I size I LOADADDRESS, NO_DUMP);
    add(targetA, EBIID IDatahere I rw_mask, NO_DUMP);
}

SU_If(inst)
BIT16 inst;
{
    int jump;

    /* jump if flags match pattern in inst */
    /* have checked if carry valid at this stage */
    jump = 1;


if(Bit(inst,5)) jump = jump & (Carry == Bit(inst,4));
if (Bit(inst,3)) jump = jump & (Zero==Bit(inst,2));
if (Bit(inst,1)) jump = jump & (Neg==Bit(inst,0));

Jump(jump);  /* tell if about result */

Setup_Alu(inst)
BITS 16 inst;
{
    FLAG carry;
    OPTAG Atag,Btag,Rtag;
    /* define operation */
    Atag = Btag = Rtag = (inst & 0x3000)>>12;
    if (Bit(inst,4)) carry = Carry;
    else carry = 0;
    if (Bit(inst,3)) carry = !carry;
    if( carry ) carry = CARRY;  /* make it the right bit */
    Atag  |= ADDID | carry | LOADA;
    Btag  |= ADDID | carry | LOADB;
    if( Bit(inst,11) )
        Rtag  |= ADDID | carry | QUIET;
    else
        Rtag  |= ADDID | carry | RESULT;
    if( inst & ALU_A_COMPLEMENT ) Atag  |= COMPLEMENT;
    if( inst & ALU_B_COMPLEMENT ) Btag  |= COMPLEMENT;
    if( inst & ALU_R_COMPLEMENT ) Rtag  |= COMPLEMENT;
    add(targetA, Atag, NO_DUMP);
    add(targetB, Btag, NO_DUMP);
    add(targetR, Rtag, NO_DUMP|FLAGSBACK);
    Carry_pending++;
}

SU_Loadaddress(inst)
BITS 16 inst;
{
    NUMTAG wordsize;
    OPTAG neg_bits = 0, carryset = 0;
    Deltaset(if_data);
    if( inst & DELTA_SIGN )
    {
        carryset = CARRY;
        neg_bits = COMPLEMENT;
    }
    add( targetB , ADDID | LOADA | carryset, NO_DUMP);
    add(SCRATCH , ADDID | LOADB | carryset | neg_bits , DUMP );
    add( targetA, ADDID | RESULT | carryset, NO_DUMP | FLAGSBACK);
    Carry_pending++;
}

- 129 -
C - The Instruction Fetcher

/* This file contains the module for the instruction fetcher. */
/* the instruction fetcher is called every cycle by the control unit - it
returns 0 (not ready) or 1 and stores an instruction and data in the places
indicated by the args.
In its turn it calls the caching/fetching mechanism to get the data
from memory, which either makes it wait or finds the data.
Thus it has various internal states defined - depending on what it is
waiting for. */

/* various states the instruction fetcher can be in */
#define INST_WAIT 1
#define NEXT16_WAIT 2
#define NEXT32_WAIT 4
#define READY 0
#define JUMP_WAIT 8

extern int icount;
static char *state_desc[16] = {"READY","INST_WAIT","NEXT16_WAIT","3","NEXT32_WAIT",
"5","6","7","JUMP_WAIT","9","10","11","12","13","14","15"};

/* state variables - global so display can access them */
static BITS 16 instsave;
static BITS32 datasave;
static BITS32 PC=0,jumpreg;
static FLAG state = INST_WAIT;

void if_display()
{
    printf("PC = %d state = %s\n",PC,state_desc[state & 0xF]);
}

int Inst_Fetch(inst,data)
BITS16 *inst;
BITS32 *data;
{
    BITS16 temp16;
    if( state & INST_WAIT )
    {
        if( Next_16(&instsave) ) /* got it from memory already */
        {
            state = pre_decode(instsave,&datasave);
        }
    }
    if( state & NEXT16_WAIT )
    {
        if ( Next_16(&temp16))
        {
            datasave = temp16;
            if( state & JUMP_WAIT) /* data is offset */
            {
                if( Bit(temp16,15) )
                    datasave = PC + (datasave 10xFFFF0000); /* sign extend */
            }
        }
    }
}
else  
  datasave += PC;
}
  state = state & JUMP_WAIT ; /* JUMP or READY */
}
    else if ( state & NEXT32_WAIT )
      if ( Next_32(&datasave))
        state = state & JUMP_WAIT;

*inst = instsave;
*data = datasave;

if( state == READY )
{
  state = INST_WAIT;
  return(1);
} else if ( state == JUMP_WAIT )
{
  jumpreg = datasave;
  return(1);  /* to pass on instruction to cause jump */
}
/* have done nothing if waiting for a jump O.K. */
return(0);
}

Jump(dojump)
int dojump;
{
  if( dojump ) PC = jumpreg; /* precalculated */
  state = INST_WAIT;
}

Next_16(put)
BITS16 *put;
{
  if (in_cache(PC)) /* this gets it from EBI if necessary */
  {
    *put = halve( PC&1 , get_cache(PC));
    PC++; 
    return(1);
  } else
  {
    Kick_EBI( PC&~1);
    return(0);
  }
}

Next_32(put)
BITS32 *put;
{
  if (in_cache(PC)) /* this gets it from EBI if necessary */
  {
    *put = get_cache(PC);
    PC += 2;   /* whole word */
    return(1);
  } else
  {
    Kick_EBI( PC);
  }

- 131 -
return(0);
}

FLAG pre_decode(inst,put_data)
BITS16 inst;
BITS32 *put_data;
{
    FLAG st;
icount++;
#ifdef DEBUG
    printf("pre_decoding \%x\n",inst);
#endif
    switch( Sig_bit(inst)) {
        case Nop : case Alu : case LoadStore :
            st = READY;
            break;
        case Hop :
            /* handle sign extension of lower 12 bits of inst */
            PC = PC + ( ( inst & 0xFFF) | ( (inst & 0x800) ? SEX : 0 ) );
            st = INST_WAIT;
            break;
        case If:
            /* if procedure */
            st = JUMP_WAIT | NEXT16_WAIT;
            break;
        case Call:
            st = JUMP_WAIT | NEXT32_WAIT;
            break;
        case Loadaddress :
            /* Load address from things */
            if( Bit(inst,11) )
                {
                    st = Bit(inst,12) ? NEXT32_WAIT : NEXT16_WAIT;
                }
            else
                {
                    *put_data = ((inst & DELTA) >> 2) + Bit(inst,12);
                    st = READY;
                }
            break;
        case 7:
            st = READY;
            break; /* switch statement */
        default:
            fprintf(stderr,"something wrong in instruction fetcher \%x \%d\n",inst,Sig_bit(inst));
    }
    return(st);
}
APPENDIX D  Sort Program implementations

All these implementations assume R0 holds the literal 0, and leave one register unused (assumed to be an environment (stack) pointer.

1: 8 registers ( 6 used )

```
LDA R1,array(R0)

:LOOP1 LDA R2,zero(R0)  ! i = 0;
LDA R6,msize(R0)  ! negated array size

:LOOP2 LDA R2,two(R2)  ! i++
CMP R2,R6  ! i < size?
IF Z,HALT  ! if so goto HALT
LD L R4,R1(R2)  ! x = a[i]
LDA R3,zero(R2)  ! j = i;

:L2 LDA R3,-2(R3)  ! j--;
IF N,NEXT        ! FINISHED because j < 0!
LD L R5,R1(R3)
CMP R5,-R4  ! if a[j] < a[i]
IF N,NEXT
LDA R3,two(R3)  ! j = j+1;
ST L R5,R1(R3)  ! a[j'] = a[j]
LDA R3,-2(R3)  ! j--;
HOP L2

:NEXT LDA R3,two(R3)
ST L R4,R1(R3)  ! a[j] = x
HOP LOOP2

:HALT HALT
```
2: 6 registers ( 5 used )

LDA R1,array(R0)
:LOOP1 LDA R4,var_i(R0)
LDH R2,R0(R4)
:LOOP2 LDA R2,two(R2) ! i++
STH R2,R0(R4)
LDA R4,msize(R2)
IF Z,HALT
LDL R4,R1(R2)
LDA R3,zero(R2)
:L2 LDA R3,-2(R3) ! j--;
IF N,NEXT ! FINISHED because j < 0!
LDL R5,R1(R3)
CMP R5,-R4 ! if a[j] < a[i]
IF N,NEXT
LDA R3,two(R3) ! j = j+1;
STL R5,R1(R3) ! a[j'] = a[j]
LDA R3,-2(R3) ! j--; HOP L2
:NEXT LDA R3,two(R3)
STL R4,R1(R3)
HOP LOOP1
:HALT HALT
3: 4 registers (3 used)

:LOOP1  LDA  R2, var_i(R0)
        LDH  R1, R0(R2)

:LOOP2  LDA  R1, two(R1)  ! i++
        STH  R1, R0(R2)
        LDA  R3, msize(R1)  ! if i < size
        IF  Z, HALT
        LDA  R3, array(R1)  !
        LD L  R2, R0(R3)  ! x = a[i]
        LDA  R3, var_x(R0)
        ST L  R2, R0(R3)

:L2  LDA  R2, var_x(R0)

*  LD L  R2, R0(R2)  ! R2 = x
*  LDA  R1,-2(R1)  ! j--;
        IF  N, NEXT  ! FINISHED because j < 0!
        LDA  R3, array(R1)
        LD L  R3, R0(R3)  ! R3 = a[j]
        CMP  R3,-R2  ! if a[j] < x
        IF  N, NEXT
        LDA  R1, two(R1)  ! j = j+1;
        LDA  R2, array(R1)  ! R2 = &a[j+1]
*  ST L  R3, R0(R2)  ! a[j'] = a[j]
*  LDA  R1,-2(R1)  ! j--;
        HOP  L2

:NEXT  LDA  R1, two(R1)
        LDA  R3, array(R1)
        ST L  R2, R0(R3)
        HOP  LOOP1

:HALT  HALT

NB. Instructions marked * were swapped to investigate instruction order sensitivity.